Using Background Colors to Support Program Comprehension in Software Product Lines

Janet Feigenspan*, Michael Schulze*, Maria Papendieck*, Christian Kästner†, Raimund Dachselt*,
Veit Köppen*, Mathias Frisch*
*University of Magdeburg, Germany
{feigensp, mschulze}@ovgu.de, maria.papendieck@st.ovgu.de, {dachselt, koeppen, mfrisch}@ovgu.de
†Philipps University Marburg, Germany
kaestner@informatik.uni-marburg.de

***Abstract*—*Background*: Software product line engineering provides an effective mechanism to implement variable software. However, the usage of preprocessors, which is typical in industry, is heavily criticized, because it often leads to obfuscated code. Using background colors to support comprehensibility has shown effective, however, scalability to large software product lines (SPLs) is questionable. *Aim*: Our goal is to implement and evaluate scalable usage of background colors for industrial-sized SPLs. *Method*: We designed and implemented scalable concepts in a tool called FeatureCommander. To evaluate its effectiveness, we conducted a controlled experiment with a large real-world SPL with over 160,000 lines of code and 340 features. We used a within-subjects design with treatments colors and no colors. We compared correctness and response time of tasks for both treatments. *Results*: For certain kinds of tasks, background colors improve program comprehension. Furthermore, subjects generally favor background colors. *Conclusion*: We show that background colors can improve program comprehension in large SPLs. Based on these encouraging results, we will continue our work improving program comprehension in large SPLs.**

I. Introduction

Today, *software product lines (SPLs)* provide an efficient mechanism to implement variable software. They allow deriving several distinguished program variants – *variants* for short – by selecting or deselecting features. A *feature* is a user-visible characteristic of a software system (Clements and Northrop, 2001). Variable code implementing a feature is called *feature code* and is only contained in a variant if the according feature is selected. In contrast to feature code, *base code* implements commonalities of an SPL and, thus, is part of every generated variant. As example, consider a customer buying a specific car model: She cannot choose the car body (base), but the type of engine and color (features).

In industry, SPLs are usually implemented with preprocessors. In Fig. 1, we show a source code excerpt of Berkely DB, in which the C preprocessor is used: #if(n)def and #endif statements (e.g., Line 13 and 15) are used to mark the beginning and end of variable code fragments.

Both benefits and drawbacks of preprocessors are discussed controversially. Benefits are that preprocessors (a) are simple to use (Favre, 1997; Muthig and Patzke, 2003), (b) are flexible and expressive, (c) can be used uniformly for different languages, and (d) are already integrated as part of many languages or environments (e.g., C, C++, Erlang, Fortran, Java Micro Edition). In contrast, in literature, preprocessors are heavily criticized and considered "harmful" (Spencer and Collyer, 1992) or even as "#ifdef hell" (Lohmann et al., 2006). Numerous studies argue that preprocessor usage leads to complex and obfuscated code that is difficult to comprehend, thus leading to high maintenance costs (Favre, 1997; Krone and Snelting, 1994; Lohmann et al., 2006; Pohl et al., 2005; Spencer and Collyer, 1992).

Despite the controversial discussion of preprocessors, they are still common in practice, although there are several approaches for implementing SPLs in a modular way, such as components (Heineman and Councill, 2001), aspects (Kiczales et al., 1997), or mixin layers (Smaragdakis and Batory, 1998). The problem is that introducing novel concepts in industry is a time-consuming and difficult process, especially when large amounts of legacy code are involved.

Hence, rather than arguing that new approaches should be preferred instead of preprocessors, we target the question how we can improve the usage of preprocessors. Specifically, we aim at supporting *program comprehension*: On average, a maintenance programmer spends 50 – 60 % of her time with understanding source code (Standish, 1984). Furthermore, maintaining software is the main cost factor in software development (Boehm, 1981). Thus, by improving comprehensibility of source code using preprocessor statements, we can decrease the time and cost of software maintenance without enforcing to change the typical industrial way to implement SPLs.

To this end, we introduced background colors in some code editors to highlight feature code (Feigenspan et al., 2010; Kästner et al., 2009). The benefit of colors compared to text-based annotations as with preprocessors is twofold: First, the annotations clearly differ from source code, which helps a developer to distinguish feature code from base code. Second, humans process color considerably faster than text (Goldstein, 2002). This allows a programmer to identify feature code at first sight. Consequently, a programmer can get an overview of a software system considerably faster. However, scalable usage of background colors is questionable in large SPLs with several hundred of features. In this paper, we introduce a tool called *FeatureCommander*, in which we implement concepts to scale the use of background colors to industrial-sized SPLs with several hundreds of features. In a controlled experiment with C programmers, we show the scalability of our approach.

```
1  static int __rep_queue_filedone(dbenv, rep, rfp)
2    DB_ENV *dbenv;
3    REP *rep;
4    __rep_fileinfo_args *rfp; {
5  #ifndef HAVE_QUEUE
6    COMPQUIET(rep, NULL);
7    COMPQUIET(rfp, NULL);
8    return (__db_no_queue_am(dbenv));
9  #else
10 db_pgno_t first, last;
11 u_int32_t flags;
12   int empty, ret, t_ret;
13 #ifdef DIAGNOSTIC
14   DB_MSGBUF mb;
15 #endif
16 // over 100 lines of additional code
17 #endif }
```

Fig. 1. Code excerpt of Berkeley DB, illustrating fine granularity, nesting, and long annotations with preprocessors.

II. Problem Statement

To understand the problems that accompany preprocessor usage, consider the source code excerpt in Fig. 1. One problem is that very long code fragments can be annotated with an #ifdef statement. For example, the comment in Line 16 states that there are more than 100 lines of code. Hence, corresponding #ifdef and #endif statement usually do not appear on the same screen. Thus, without further support, it is difficult to keep track of the according feature belonging to the annotated code fragment.

Furthermore, #ifdef statements can be *nested* (i.e., within a code fragment that is annotated with one #ifdef, a different #ifdef statement occurs). For example, in Fig. 1, the #ifdef in Line 13 is stated within another #ifdef, which begins in Line 5. It might be ok to deal with two nested #ifdefs (a nesting level of two), however, typical industrial SPLs can have a nesting level of up to 24, which are hard to keep track of (Liebig et al., 2010).

Additionally, #ifdef statements are textual and, thus, do not differ that much from source code itself. They can be overlooked easily, which makes them harder to track. These problems illustrate the problems for comprehensibility of preprocessor-based SPLs and the potential increase to software development costs.

Preprocessors and Background Colors: Previous Results

To improve the understandability of preprocessor-based software, we introduced and evaluated the use of background colors to highlight code fragments that are annotated with #ifdef statements (Kästner et al., 2009; Feigenspan, 2009; Feigenspan et al., 2009). This way, we profit from the facts that background colors clearly distinguish from source code and that humans perceive background colors preattentively[1].

In this approach, we used one-to-one mapping of background colors to features, such that each feature has one background color. In a previous experiment, we evaluated how background colors influence program comprehension (Feigenspan, 2009; Feigenspan et al., 2009). We used a medium-sized SPL with about 5,000 lines of code, four features (Figueiredo et al., 2008a), and a nesting level of two in three occurrences.[2] To determine the background color for the nested features, we blended the colors of both participating features. The SPL was implemented in Java Micro Edition (ME) with Antenna, a preprocessor for Java ME.[3] In our setting, we compared two version of this SPL that only differed in one facet: one had background colors, the other had no background colors. As subjects, we recruited under-graduate students from a programming course, in which advanced programming paradigms, such as preprocessor-based SPLs, were taught.

The results are encouraging: We found that for certain kinds of tasks, the version with background colors speeded up the comprehension process of subjects. Furthermore, subjects rated the idea of background colors very positive.

However, one limitation of our study is caused by the source code we used, a medium-sized SPL with only four features, in which a one-to-one mapping of colors to features was feasible to support program comprehension. In such a small setting, it may not sound surprising that colors speed up program comprehension. However, the scalability of the results to realistic, industrial-sized SPLs is questionable for two reasons: First, human working memory capacity is limited, and, second, human ability to distinguish colors is limited, as well. First, in human working memory typically 7 ± 2 *items* can be stored, where an item is a unit of information (Miller, 1956). Examples of items are digits, such as in telephone numbers, or the features a developer is working with. If working memory capacity is exceeded, information units are forgotten if not stored in another way, for example by writing it down. Second, human capability to distinguish colors is limited. In direct comparison (i.e., when colors are displayed next to each other), humans can tell about two million colors apart (Goldstein, 2002). Without direct comparison, humans can only distinguish few colors (Rice, 1991). Hence, the scalability of background color usage to SPLs with several hundred of features is questionable. Clearly, a one-to-one mapping of colors to features is not feasible in large SPLs.

Instead, we suggest an as-needed mapping of colors to features, such that a developer can assign colors to features as she thinks is appropriate for her current activity. This concept is based on a number of observations of preprocessor-based software. First, for most part of the source code, only three features are present at one code fragment that fits on a screen (Kästner, 2010). Second, bugs can often be narrowed down to features or feature combinations (Kästner, 2010). Hence, most of the time, a developer needs to deal with only few features at a time, so a customizable mapping of colors to features should support program comprehension. Based on these observations and the results of our previous experiment, we present our solution to scale the use of background colors to large SPLs.

III. FeatureCommander

We developed a tool called *FeatureCommander*, in which we implemented several concepts to make background colors

[1] Preattentive perception describes the fast recognition of certain visual properties (Goldstein, 2002)

[2] Material and results of this study are available at http://fosd.de/exp_cppcide.

[3] http://antenna.sourceforge.net/.

feasible for realistic industrial SPLs. In Fig. 2, we show a screenshot of FeatureCommander displaying Xenomai, a real-time extension to Linux with several hundred of features. We refer to the numbers in the Fig. 2 when explaining the according concepts in the next paragraphs.

FeatureCommander is a prototype for preprocessor-based SPL development. It offers multiple visualizations that support program comprehension in large SPLs. The basic characteristic of FeatureCommander is the consistent usage of colors throughout all visualizations. Users can assign colors to features by dragging a color from the color palette (1) and dropping it on a feature in any of the visualizations. For efficiency, users can also automatically assign a palette of colors to multiple features (2). Furthermore, color assignments can be saved and loaded (3), so that a developer can easily resume her work. When no color is assigned to a feature, it is represented by a shade of gray in all visualizations.

Using the color concept, we address both aforementioned problems (i.e., the restricted human working memory capacity and the restricted human ability to distinguish colors without direct comparison): First, with the customizable color assignment to features and the default setting (shades of gray), we support the limited working memory capacity. A developer can select the features that are relevant for her task at hand, which is typically in the range of 7 ± 2 (cf. Section II). Hence, she can immediately recognize that she is looking at a relevant feature, because it is colored, while the non-relevant features do not stand out, because they are gray. Second, the developer only has to tell the same number of colors apart, which is well within the human range to distinguish colors without direct comparison. Furthermore, we support a developer in switching between tasks with different color assignments to features, because color assignments can be easily saved and loaded.

Similar to other IDEs, we provide different views: *source code view*, *explorer view*, and *feature model view*. In the *source code view* (4), the background color of source code fragments indicates to which features fragments are related. To compromise between code readability and feature recognition, users can adjust the transparency of background colors (5). With the adjustable transparency, we address the problem that too intensive colors can be distracting (Feigenspan et al., 2009).

If nested features occur, we display the color of the innermost feature (6). This way, we do not have to blend colors anymore, which would lead to confusion in large SPLs with several hundreds of features, because it is hard to decide for a user whether a color is blended from several colors or whether it is just a color of one feature. Instead, to visualize nested features, we use sidebars on both sides of the source code view (7)(8). For each feature, one vertical bar in either gray or an assigned color is shown. Both sidebars provide tool tips that show the according feature when hovering over a vertical bar. The sidebar on the right (7) displays the occurrence of each feature and according nesting hierarchies scaled to the complete file. To support navigation, users can click on the vertical bars to get to a desired position in the source code file (e.g., where four features appear at the same time in one fragment). The sidebar left of the source code view (8) shows the nesting hierarchy of features in the currently visible source code fragment.

With our concept to deal with nested #ifdefs, we address both problems: First, when a developer is working with a set of features, she does not have to memorize additional colors, which would occur if we blended the features. Second, the limited capability to distinguish colors without direct comparison is not exceeded, since the number of colors is not larger than the number of currently relevant features. In addition, the sidebars allow a developer to navigate to feature code very efficiently.

In the *explorer view* (9), users can navigate the file structure and open files. Files and folders are represented by their name and horizontal boxes, in which we visualize whether a file/folder contains feature code or not: If a file/folder does not contain any feature code, we leave the horizontal box empty (10). If a file/folder does contain feature code, we display vertical bars of different colors: When a feature has no color assigned, we use a shade of gray to indicate the occurrence of feature code (11). To allow a developer to distinguish subsequent features without an assigned color, we use alternating shades of gray. When a feature has a color assigned, we show the according color in the explorer view (12). Furthermore, the amount of feature code in a file/folder is indicated by the length of each vertical bar. For example, if half a file contains feature code, then the horizontal box is filled half with vertical bars.

By using a visual representation to highlight files/folders, we allow a developer to efficiently get an overview of a software system. She immediately recognizes whether a file/folder contains feature code and whether the feature code is relevant for her current task. By using alternating shades of gray in the default setting, we allow a developer to recognize the presence of different features in a file/folder, including the amount of feature code, without opening it.

To further support the developer in navigating in a large SPL, we provide two tree representations of the project: One ordered according to the file structure, as displayed in Fig. 2 (middle). The other representation is ordered by features (left in Fig. 2). For each feature, the files and folder hierarchies are displayed, including the horizontal boxes and vertical bars indicating percentage of feature code in a file/folder. This way, if a developer wants to get an overview of all files of a feature, she just activates the feature representation of the explorer view and can see the according files at one glance. In both representations, tool tips show the features of a file/folder.

With the representation ordered by features, we support a developer in getting an overview of an SPL. This way, she immediately recognizes the files/folders in which a feature is defined without having to open it.

Finally, in the *feature model view* (13), the feature model is shown in a simple tree layout. Features that are currently not of interest to a developer can be collapsed. Colors can be dragged and dropped on features, as well as deleted. This helps a developer to quickly locate a feature relevant for her task and assign a color. After color assignment, since the other views of FeatureCommander use the assigned colors, the developer can effi-

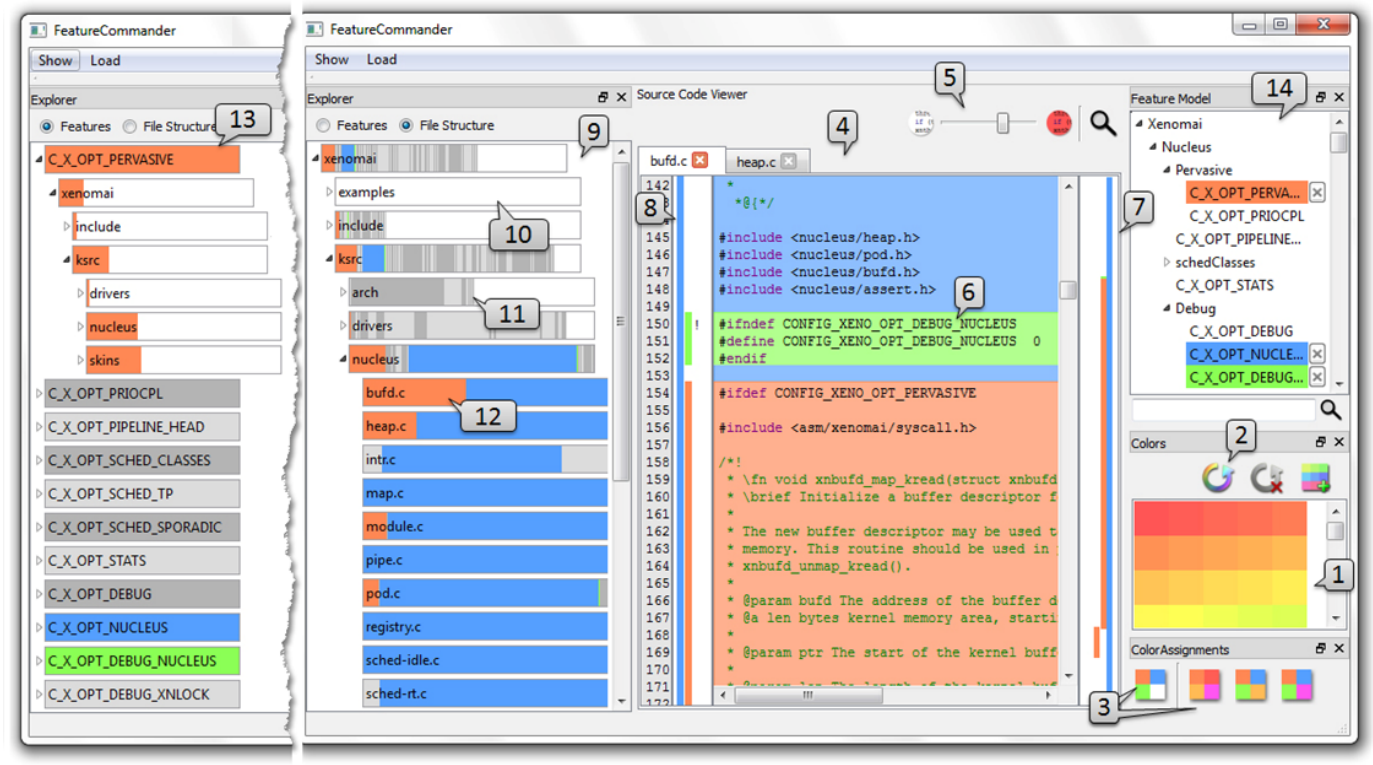

Fig. 2. Screenshot of FeatureCommander. The numbers designate concepts we explain in detail.

ciently locate feature code in files and folders of all other views.

To sum up, with FeatureCommander, we address both aforementioned problems, i.e., the restricted human working memory capacity and the restricted human ability to distinguish colors without direct comparison. Both human limitations pose problems to a scalable use of background colors to large SPLs. A developer can assign colors to features as needed. Since she typically works with few features at the same time, we do not exceed her working memory capacity or ability to distinguish colors. Furthermore, tool tips in the explorer view and source code view as well as assigned colors in the feature model view support a developer: When a developer has forgotten or cannot tell to which feature a color belongs, she can easily look it up.

In addition to the human-related problem, we address the problems of preprocessor statements: Long annotated code fragments, nested statements, and similarity to non-preprocessor code. First, since we frame code fragments with background colors, #ifdef and according #endif statements can be easily spotted. Furthermore, we display vertical bars left and right of the source code editor, which visualize the features of the currently displayed code fragment (left) or the features scaled to the complete file. Hence, the beginning and ending of each feature can be spotted easily. Second, we visualize nested statements. We always show the color of the innermost feature and the nesting hierarchy with the vertical bars, which allow a user to easily identify the location of nested features in a file. Third, since background colors clearly distinguish from source code and colors are processed preattentively, FeatureCommander helps to locate feature code at first sight.

In the remainder of this paper, we describe a controlled experiment in which we evaluated whether the implemented concepts indeed scale to large industrial SPLs. Since there is no empirical work on any of the concepts we implemented and we have strict resource constraints regarding time and subjects, we restrict our comparison to one concept, i.e., scalable usage of background colors. We decided for background colors, because it is the basic concept of FeatureCommander and we can profit from prior empirical work.

IV. Experiment Planning

In this section, we present our experimental setting. We describe the setting in a great level of detail to enable the reader to draw her own conclusion from our data and other researchers to replicate our experiment. The material we present here (e.g., questionnaires, tasks, tools, results of statistical tests) are available online at the project's website: http://fosd.de/fc.

A. Objective

Our goal is to evaluate whether the use of background colors improves comprehensibility in large SPLs. Since usually only few features are visible at the same time on a screen (cf. Section II), we argue that their use does scale. Hence, our first research hypothesis is:

RH1: *Background colors improve program comprehension in large SPLs.*

Large means, that the source code consist of at least 40,000 lines of code (von Mayrhauser and Vans, 1995) and considerably more than 7 ± 2 features, such that humans cannot distinguish colors without direct comparison, if we used a one-to-one mapping of colors to features.

In addition to the performance of our subjects, we analyze the opinion of subjects toward background colors. Since we found in previous experiments that subjects like the usage of background colors, we assume that this is the case for large software projects, too. Hence, our second research hypothesis is:

RH2: *Subjects rate background colors more positive compared to no background colors in large SPLs.*

B. Experimental Material

For our experiment we, used a large SPL – Xenomai a real-time extension for Linux.[4] Xenomai follows a dual-kernel approach, which means that it acts as primary kernel, having real-time capabilities; the Linux kernel is executed within Xenomai's idle task whenever nothing else has to be done in real time. Xenomai runs on different platforms, supports a variety of features including real-time communication and scheduling. The whole source code consists of about 165,082 lines of code including 2 lines of feature code and 343 different features, which are responsible for selecting platform specific code, special hardware drivers, or different timing policies.[5]

To present the source code to our subjects, we generated two tailored versions of FeatureCommander. This was necessary to evaluate our research question, whether the use of background colors scales to large SPLs. In a *color version*, we deleted the explorer view ordered by features and the sidebars in the source code view. Otherwise, subjects could use the functionality of the sidebars to locate feature code and we would measure its usage instead of the scalability of using background colors. Furthermore, we defined different sets of colors for each task, which subjects were instructed to load, depending on the features that were relevant for each task. We selected colors, such that they were consistent between tasks (e.g., if a feature occurred in two tasks, it had the same or similar color in both tasks) and such that humans can clearly distinguish all colors without direct comparison (Rice, 1991). We decided to specify the colors, so that subjects would not spend their time with assigning colors to features, but work on tasks. In a *colorless version*, we removed everything associated to colors.

For both versions, we implemented three search functionalities: First, search in the complete project, second, search in an opened file, and, third, search in the feature model view. This is necessary because of the size of the project, so that we do not measure how fast subjects can find a certain file in about thousand files or a certain feature in about 350 features. Instead, the search functions allow subjects to locate a certain code fragment within a file or feature within a list of features in both versions of our tool. Furthermore, this is a more realistic setting, since typical IDEs provide similar search functionalities.

[4]http://www.xenomai.org.

[5]Analyzed with cppstats, available at http://fosd.de/cppstats.

For both versions, we implemented a window, in which we present the questions and text fields to record the answer of subjects. To prevent subjects from getting stuck on a task, every 15 minutes a pop up notified subjects about the time passed.

Furthermore, we gave subjects paper-based questionnaires, on which they evaluated the difficulty of each task, the motivation to solve the task, and their estimated performance, if they had worked on the according task with the other version of the tool. At the end of the experiment, we asked subjects whether they preferred background colors over working without colors, and whether they think background colors are more suitable when working with preprocessor compared to no colors. Additionally, we encouraged subjects to leave remarks, for example, about the experimental setting or the tool.

C. Subjects

As subjects, we recruited 9 master and 5 PhD students from the University of Magdeburg, Germany. Master students were enrolled in the course *Embedded Networks*, in which extended knowledge of operating systems and distributed networks was taught. To complete the course, students were required to hand in several assignments, in which they implemented code regarding operating systems and networks, such as clock synchronization of different computers. The PhD students' expertise was also in the operating and embedded systems' domain.

Master students could participate in the experiment instead of completing one assignment. The performance in the experiment was not part of the master students' grade for this course. To recruit the PhD students, we sent a mail to those who worked in the domain of operating and embedded systems as well as real-time properties. Subjects were aware that they took part in an experiment and could leave any time they wanted.

To measure programming experience, we administered a questionnaire before the experiment, in which a low value (min: 5) indicates no experience, a high value (over 60 – the scale is open-ended) high programming experience. All subjects were familiar with C (median: 4, on a five-point Likert scale (Likert, 1932), 1 meaning very unexperienced, 5 very experienced). All subjects were male; none was color blind[6]. We created two groups with comparable programming experience according to the value of the programming experience questionnaire.

D. Tasks

To measure program comprehension, we designed tasks that can only be solved if subjects understand according source code. We used two kinds of tasks: maintenance and static tasks (Dunsmore and Roper, 2000). In maintenance tasks, subjects are instructed to locate/fix a bug, while in static tasks, subjects should examine the structure of the source code. In a previous experiment (Feigenspan, 2009), we found that colors speed up program comprehension only for static tasks, but not in maintenance tasks. Hence, we focused on static tasks, but included few maintenance tasks to control whether our results still hold.

[6]The chosen colors were not tested regarding their suitability for color blindness. For future work, this is an important issue to evaluate. However, for now our goal is to evaluate whether colors can help at all.

All tasks are typical for a maintenance programmer, when she is looking for bugs in certain features and/or files. Altogether, we had 10 tasks: 2 warming up tasks, 6 static tasks, and 2 maintenance tasks. The warming up tasks were designed to let subjects familiarize with the experimental setting and were not analyzed. Regarding static tasks, we had three different types:

1: Identifying all files in which source code of a certain feature was implemented.
2: Locating nested #ifdef statements.
3: Identifying all features that occur in a certain file.

For each type, we prepared two tasks. As example, we present the first static task (S1):

S1: In which files does feature `CONFIG_XENO_OPT_STATS` occur?

For maintenance tasks, we carefully introduced bugs into the source code, which subjects were instructed to locate (name file and method, why it occurs, and how it could be solved). Those bugs and according bug descriptions we presented subjects were typical in C programs implementing software in the domain of operating systems, which we made sure by consulting an expert in C and Xenomai. We present the bug description of the first maintenance task to illustrate them:

M1: If the PEAK parallel port dongle driver (`XENO_DRIVERS_CAN_SJA1000_PEAK_DNG`) should be unloaded, a segmentation fault is thrown.
The problem occurs, when features `CONFIG_XENO_DRIVERS_CAN` and `CONFIG_XENO_DRIVERS_CAN_SJA1000` and `CONFIG_XENO_DRIVERS_CAN_SJA1000_PEAK_DNG` are selected.

E. Design

We conducted the experiment in two phases. In the first phase, group A worked with the color version and group B with the colorless version. In the second phase, we switched the groups: Group A now worked without colors, and group B with colors. In each phase, we applied the same tasks to both groups. Hence, for both groups, the sequence of tasks was: W1, S1, S2, S3, M1, and, in the second phase, W2, S4, S5, S6, M2. The task designator indicate the kind of task (W: warming up, S: static, M: maintenance). The static tasks S1 and S4 were of the same type, as were S2 and S5, as well as S3 and S6. We designed the tasks of both phases to be comparable regarding difficulty and effort (e.g., the same number of features had to be entered as solution), such that we can compare the results within phases (i.e., between groups) and between phases (i.e., within groups).

F. Conduction

The experiment took place in June 2010 instead of a regular exercise session. We booked a room sufficiently equipped with equivalent working stations. All computers had 17" TFT displays. Before the experiment started, we gave an introduction to our subjects, in which we explained the proceeding of the experiment, including how to use the tool. After all questions were answered, subjects started to work on the tasks on their own. When a subject had finished a task, he immediately switched to the next task, except when he finished the last task of a phase. At the end of each phase, we gave subjects a questionnaire to assess their opinion (difficulty, motivation, performance with other version). After the second phase, we additionally assessed which version subjects like better and which they think is more suitable to work with preprocessor-based implementations. Three experimenters checked that subjects worked as planned. No deviations occurred.

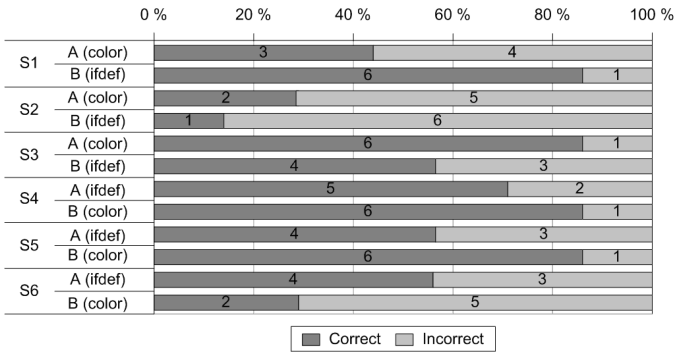

Fig. 3. Correctness of answers.

Having provided a detailed description of the experimental setting and conduction, we can present the analysis of our data.

V. Analysis

In this section, we present the analysis of our data. We strictly separate analyzing our data from interpreting the results, so that we enable the reader to put her own interpretation to our data.

A. Descriptive Statistics

From our tasks, we can use two measures to assess how subjects understood a program: correctness and response time (i.e., how long subjects needed to solve a task). In Fig. 3, we show how correct answers differ between both groups. We omitted maintenance tasks in Fig. 3, because we could not rate any of the solutions as correct, although subjects could often narrowed down the problem to the correct file and function. We discuss this issue in Section VII. For static tasks, we can see that the difference in correctness of answers is the largest for the first static task (S1): Only three subjects solved it correctly in group A (with colors), but six in group B (without colors).

In Fig. 4, we show the response time of our subjects.[7] We can see that for the first two static tasks (S1 and S2), group A (color version) is faster than group B.

The estimation of subjects regarding difficulty, motivation, and performance with the other version (color/colorless) are shown in Fig. 5. We can see that for difficulty, in four static tasks (S1: locating files of a feature; S2, S5: locating nested #ifdefs; S3: locating all features in a file) and one maintenance task, the median is the same. For the other tasks, the median differs by 1. For motivation, the deviation within a group is larger than for difficulty, meaning that subjects

[7] Fig. 4 uses a *box plot* to describe data (Anderson and Finn, 1996). 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.

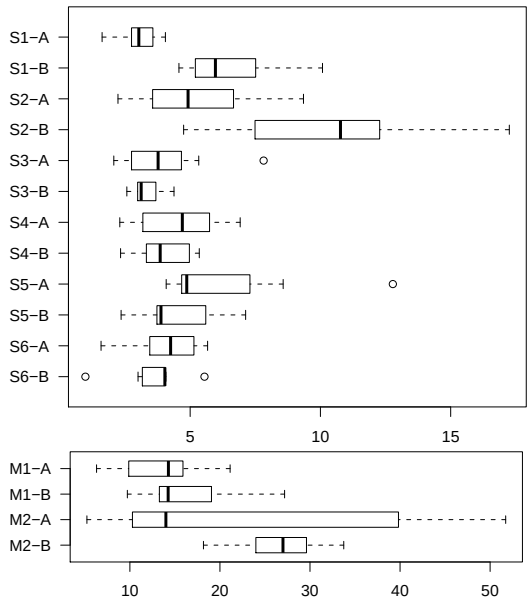

Fig. 4. Response time of subjects in minutes.

rated their motivation more heterogeneously. Both groups were motivated for all tasks at least to a mediocre level. For the first maintenance task (M1), the motivation for group A (with colors) was very high, in contrast to group B with a mediocre motivation. For estimation of performance with the other version, we see that in both phases, subjects that worked with the color version thought they perform worse with the colorless version, and vice versa.

When asked what version they prefer, 12 subjects said they like the color version better and 13 said the color version is more suitable when working with preprocessor-based SPLs. One subject did not answer any of both questions.

B. Hypotheses Testing

In this section, we evaluate whether our research hypotheses hold. To this end, we conduct several statistical tests to check whether the differences we observed are significant. We start with correctness of answers. Since we compare frequencies, we need to conduct a χ^2 test (Anderson and Finn, 1996). To meet its requirements, we summarize the correctness of answers for the static tasks of each phase, such that we add the number of correct and incorrect answers for each phase.[8] Hence, we compare the number of correct and incorrect answers of tasks $S1 + S2 + S3$ and $S4 + S5 + S6$. The χ^2 test indicates no significant differences in the number of correct answers for static tasks. Since for maintenance tasks, none of the subjects provided a correct solution, we do not need to test for significant differences in correctness of maintenance tasks.

For response time, we make several comparisons for our data: Group A vs. group B, group A (first phase) vs. group A (second phase), and group B (first phase) vs. group B (second phase). Since we make multiple comparisons, we need to adjust the significance level. To this end, we use a Bonferoni correction, which leads in our case to a significance level of 0.017 to observe a significant difference (Anderson and Finn, 1996).

[8]Expected frequencies are too small due to the small number of observations.

Opinion	Task	S1	S2	S3	M1	S4	S5	S6	M2
Difficulty	U value	20.5	24.5	17.5	18	10.5	0	15.5	24
	significant	no	no	no	no	yes	yes	no	no
Motivation	U value	24	18.5	22	9.5	15.5	15	19.5	23
	significant	no	no	no	yes	no	no	no	no
Other version	U value	6	2.5	0	13.5	2	3	2	4.5
	significant	yes	yes	yes	no	yes	yes	yes	yes

TABLE I
MANN-WHITNEY-U TEST FOR SUBJECTS' OPINION.

First, we compare the results of both groups. We applied t-tests for independent samples (Anderson and Finn, 1996), since the response times are normally distributed (tested with a Kolmogorov-Smirnov test, (Frank Massey, 1951)). We only observed significant differences for tasks S1 (p value: 0.001) and S2 (p value: 0.017). Hence, only for the first two tasks, subjects that worked with the color version (group A) were faster. However, when we switched the versions, such that group B worked with the color version, we could not observe any differences.

Second, we make pairwise comparisons within our groups of tasks of both phases, i.e., S1 vs. S4, S2 vs. S5, S3 vs. S6, and M1 vs. M2. We only observed significant differences in group B, such that the response times for S4 and S5 were significantly faster than for S1 and S2, respectively. Hence, when subjects switched from the colorless to the color version, their performance for two tasks increased. Group A, on the other hand, was not slower in the second phase, which we expected since they worked without colors now. The results regarding response time speak both in favor of and against our research hypothesis. Hence, we can neither confirm nor reject our research hypothesis.

Finally, we compare the opinion of subjects. Since they are ordinally scaled, we use a Mann-Whitney-U test (Anderson and Finn, 1996). In Table I, we summarize the results of this test.[9]

We can see that for difficulty, subjects of group B rated S4 and S5 significantly easier than subjects of group A. This is also reflected in the performance, such that subjects of group B are faster in these tasks (S4 vs. S1, S5 vs. S2). For motivation, we observe a significant difference for the first maintenance tasks, such that subjects of group A were more motivated to solve this task compared to group B. For estimation of the performance with the other version, we obtain significant differences for all tasks (except M1), such that subjects that worked with the color version expect that they had performed worse with the other version. The results regarding the estimation of performance and how subjects liked the color version and evaluated its suitability speak in favor of our second research hypothesis

[9]To meet the requirements of the Mann-Whitney-U test, we use the probability function of the U distribution for critical values (Giventer, 2008) to state whether the observed differences are significant. Some U values are 0, yet the difference is significant, because of the extreme distribution of answers.

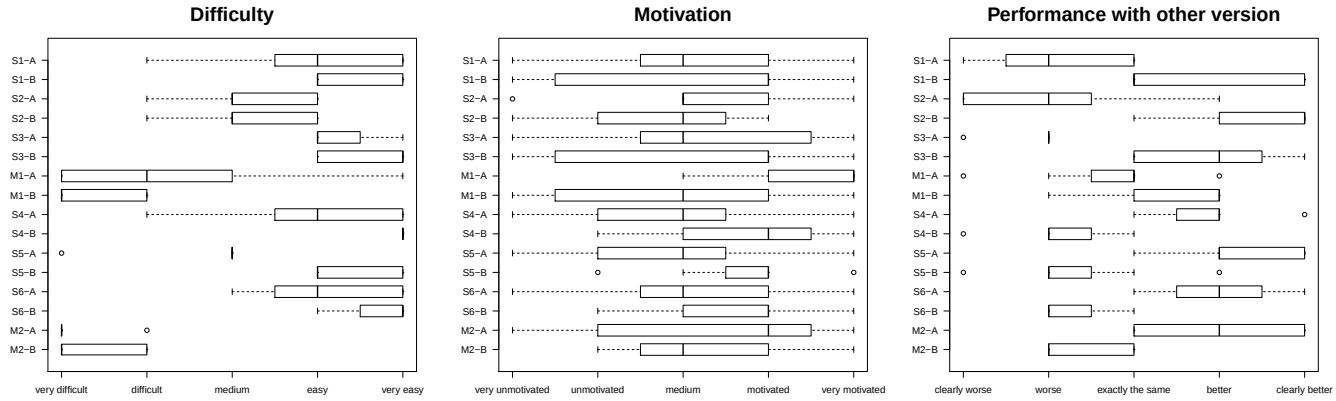

Fig. 5. Box plots of subjects' opinion.

(i.e., that subjects rate background colors more positive).

VI. Interpretation

We discuss the implication of our results for each hypothesis.

RH1: *Background colors improve program comprehension in large SPLs.*

The data we observed do not allow us to clearly confirm or reject this hypothesis. To evaluate this hypothesis, we measured the correctness of answers of subjects and the response time. There was no difference regarding correctness of static tasks. For response time, we found that in the first phase, for two static tasks, subjects that worked with background colors were significantly faster. In the second phase, in which we switched the versions, we did not observe any significant differences in response time between both groups. However, we found that within group B (i.e., subjects that started to work with the colorless version and switched to the color version) were faster in tasks S4 and S5, compared to S1 and S2, respectively, for which we observed a significant difference in the first phase.

Hence, for two static tasks, background colors improve program comprehension. One reason that for the third kind of static tasks (i.e., locating all features in a file), background colors showed no improvement, could be that we had 12 features and, thus, 12 different colors in this task. Although we carefully chose colors, such that subjects could distinguish them easily (Rice, 1991), 12 colors might be too much for subjects. Furthermore, with 12 different colors, the working memory capacity is clearly exceeded. In the other tasks, 9 colors at most have to be kept in mind, which is in the top end of 7 ± 2. However, we cannot be sure whether this result occurred because of too many different colors or because of the kind of task, since we only combined 12 features with this kind of task.

To sum up, when subjects start to work with the color version and then switch to the colorless version, it has no effect on their response time. When subjects work without colors and then switch to background colors, their performance increases. Thus, to familiarize with a large SPL, background colors *can* help, especially to get an overview of the files in which code of a certain feature occurs and to locate nested #ifdefs. This result aligns with the results of our previous experiment, in which we found that for the same types of static tasks in small SPLs, background colors speed up program comprehension, however have no effect on correctness.

RH2:*Subjects rate background colors more positive compared to no background colors in large SPLs.*

Regarding difficulty, we found that subjects that worked with the color version in the second phase rated static tasks easier than subjects that worked without colors. Hence, when we add background colors, tasks seem to become easier for the according subjects. Regarding estimation of performance with the other version, we found a strong effect in favor of background colors (i.e., subjects that worked with the color version estimate they would perform worse without colors). Furthermore, all subjects who answered this question thought that colors were more suitable to work with preprocessor-based SPLs and all besides one subjects liked the color version better. Hence, we can confirm our second research hypothesis. Furthermore, our data regarding opinion of subjects also align with the results of our first experiment.

VII. Threats to Validity

Like in each empirical study, threats to validity occur. Validity can be divided into *internal* (degree to which we have controlled confounding variables for program comprehension, e.g., programming experience) and *external* validity (generalizability of results to other experimental settings).

A. Internal Validity

Since there is no standardized assessment of programming experience, we applied our own questionnaire to our subjects and created homogeneous groups. However, we cannot be sure how well we measured programming experience. To reduce this threat, we developed the questionnaire based on literature research and programming experts, who rated the questionnaire regarding how well it measures programming experience (Feigenspan, 2009).

Another problem is that none of the subjects solved any of the maintenance tasks correctly. Most likely, since we

designed the maintenance tasks to be realistic, the tasks were too difficult, given the subjects' expertise and time constraint of the experiment. Experienced developers with more time should find this bug eventually. Furthermore, maintenance tasks were not our primary focus, since we found in an early study that colors did not affect the comprehension process in these tasks (Feigenspan, 2009; Feigenspan et al., 2009).

Our sample is rather small. However, we used several mechanisms to deal with this issue: We used a within-subjects design to apply both versions of our tool to all subjects. Furthermore, we applied variants of standard significance tests that were developed to deal with small sample sizes. However, we could not deal with biased response time due to wrong answers.

B. External Validity

One threat is caused by our sample, which consisted mostly of master students with relatively little programming experience. We could reduce this threat by including PhD students in our sample with several years of experience in the domain of embedded and operating systems, so our results can be carefully applied to experienced programmers.

Furthermore, we only tested one SPL in one programming language of one domain, to which we can apply our results. However, we used a typical industrial system (large C SPL in embedded systems), instead of an artificial research software. Hence, our results can be applied to real-world industrial settings and, thus, are of interest for industry.

VIII. Related Work

There is a lot of work dealing with visualization of preprocessors, e.g., with control flow graphs (Hu et al., 2000; Krone and Snelting, 1994; Pearse and Oman, 1997). For example, Hu et al. propose the analysis of control flow graphs based on preprocessor directives to get an overview of the inclusion structure of a file. In contrast, our focus lies on supporting preprocessor statements on the source code level and, thus, keeping context information of preprocessor statements.

Another way to handle the complexity of preprocessors is to provide views on source code (Atkins et al., 2002; Chu-Carroll et al., 2003; Kästner, 2010; Singh et al., 2007). A view on a variant or feature shows only relevant code for a feature and its combination and hides all remaining code. In some tools, annotations are hidden entirely and the developer works on a single variant without even being aware of other variants or features. In an empirical study on views in the Version Editor (Atkins et al., 2002), Atkins et al. measured a 40 % increase in developer productivity. Nevertheless, hiding feature code is not always desirable; for example, when fixing a bug, a developer may need the context of the entire SPL to fix the bug not only in a single, but in all variants. Views on the source code and colors are complementary and have strength for different tasks.

Regarding colors, there is a huge body of work on using colors for various tasks, such as highlighting source code according to semantic of statements or control structure (Rambally, 1986), error reporting (Oberg and Notkin, 1992), or merging (Yang, 1994). We focus only on work that addresses program comprehension in SPLs or for scattered concerns. In prior work, we used background colors to represent annotations in our SPL tool CIDE, with which we explore improvements of preprocessors (Feigenspan et al., 2010; Kästner, 2010; Kästner et al., 2008); and we showed that for small SPLs, background colors can improve program comprehension (Feigenspan, 2009; Feigenspan et al., 2009). Closest to our representation with background colors are the model editors *fmp2rsm* (Czarnecki and Antkiewicz, 2005) and *FeatureMapper* (Heidenreich et al., 2008), in which model elements can be annotated and removed to generate different model variants. Both tools provide views and additionally can represent some or all annotations with colors. Furthermore, *Spotlight* (Coppit et al., 2007) addresses scattered concerns (outside of the context of SPLs). Spotlight uses vertical bars in the left margin of the editor to visualize annotations. Again, different colors represent different concerns. Bars of different colors are placed next to each other. Compared to background colors, lines are more subtle and can represent nesting easily. In all cases, the impact of visualizing annotations was not measured empirically so far.

There is a lot of empirical work regarding the evaluation of SPL implementing techniques. Especially aspect-oriented programming is at focus of several researcher groups (e.g., (Figueiredo et al., 2008a,b; Greenwood et al., 2007; Hanenberg et al., 2009)). For example, Figueiredo et al. and Greenwood et al. evaluated several facets of aspect-oriented programming, such as maintainability or design stability. However, the assessment is conducted without human subjects, but with software measures. In the work of Hanenberg et al., the understandability of aspect-oriented programming is compared to object-oriented programming. In an experiment, subjects had to implement crosscutting code into a small target application, one implemented in AspectJ, the other in Java. Depending on the kind of code changes, AspectJ had positive or negative influence on the development time of subjects.

IX. Conclusion and Future Work

Software product lines are typically implemented with preprocessors in industry. To overcome its obfuscation issues, we use background colors to highlight annotated code fragments. In this paper, we present a tool called FeatureCommander, in which we implemented concepts to support program comprehension in large SPLs. We showed in a controlled experiment that the core characteristic of FeatureCommander, the usage of background colors, scales to a large SPL with over 160,000 lines of code and 340 features: Subjects that work with colors are faster for certain tasks and rate background colors as pleasant and suitable to work with preprocessor-based implementations. Hence, the use of background colors to support program comprehension in large SPLs is very promising.

In future work, we plan to evaluate open issues we discovered in our experiment to confirm the positive effect of background colors on program comprehension and broaden our understanding of how background colors can be used to support program

comprehension. Additionally, we can evaluate how other implemented concepts in FeatureCommander, such as the explorer view ordered by features, improve program comprehension.

ACKNOWLEDGMENT

Feigenspan's, Köppen's, and Frisch's work is supported by BMBF project 01IM08003C (ViERforES). Kästner's work is supported in part by ERC (#203099). Dachselt's work is funded by the "Stifterverband für die Deutsche Wissenschaft" from funds of the Claussen-Simon-Endowment. Thanks to Jana Schumann for support during the conduction of the experiment and Jörg Liebig for computing statistics of Xenomai.

REFERENCES

Paul Clements and Linda Northrop. *Software Product Lines: Practice and Patterns*. Addison Wesley, 2001.

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

Dirk Muthig and Thomas Patzke. Generic Implementation of Product Line Components. In *Int'l Conf. NetObjectDays*, pages 313–329. Springer, 2003.

Henry Spencer and Geoff Collyer. #ifdef Considered Harmful or Portability Experience With C News. In *Proc. USENIX Conf.*, pages 185–198. USENIX Association, 1992.

Daniel Lohmann, Fabian Scheler, Reinhard Tartler, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 191–204. ACM Press, 2006.

Maren Krone and Gregor Snelting. On the Inference of Configuration Structures from Source Code. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 49–57. IEEE CS, 1994.

Klaus Pohl, Günter Böckle, and Frank van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

George Heineman and William Councill. *Component-Based Software Engineering: Putting the Pieces Together*. Addison Wesley, 2001.

Gregor Kiczales et al. Aspect-Oriented Programming. In *ECOOP '97: Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

Yannis Smaragdakis and Don Batory. Implementing Layered Designs with Mixin Layers. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 550–570. Springer, 1998.

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

Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

Janet Feigenspan, Christian Kästner, Mathias Frisch, Raimund Dachselt, and Sven Apel. Visual Support for Understanding Product Lines. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 34–35. IEEE CS, 2010.

Christian Kästner, Sven Apel, Salvador Trujillo, Martin Kuhlemann, and Don Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *TOOLS EUROPE '09: Proceedings of the 47th International Conference Objects, Models, Components, Patterns*, pages 174–194. Springer, 2009.

Bruce Goldstein. *Sensation and Perception*. Cengage Learning Services, fifth edition, 2002.

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. ACM Press, 2010.

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

Janet Feigenspan, Christian Kästner, Sven Apel, and Thomas Leich. How to Compare Program Comprehension in FOSD Empirically - An Experience Report. In *Proc. Int'l Workshop on Feature-Oriented Software Development*, pages 55–62. ACM Press, 2009.

Eduardo Figueiredo, Nelio Cacho, Mario Monteiro, Uira Kulesza, Ro Garcia, Sergio Soares, Fabiano Ferrari, Safoora Khan, Fernando Filho, and Francisco 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 Press, 2008a.

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

J Rice. Display Color Coding: 10 Rules of Thumb. *IEEE Software*, 8(1): 86–88, 1991.

Christian Kästner. *Virtual Separation of Concerns: Preprocessors 2.0*. PhD thesis, University of Magdeburg, 2010.

Anneliese von Mayrhauser and Marie Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.

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

Alastair Dunsmore and Marc Roper. A Comparative Evaluation of Program Comprehension Measures. Technical Report EFoCS 35-2000, Department of Computer Science, University of Strathclyde, 2000.

Theodore Anderson and Jeremy Finn. *The New Statistical Analysis of Data*. Springer, 1996.

Jr. Frank Massey. The Kolmogorov-Smirnov Test for Goodness of Fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

Lawrence Giventer. *Statistical Analysis for Public Administration*. Jones and Bartlett Publishing, second edition, 2008.

Ying Hu et al. C/C++ Conditional Compilation Analysis using Symbolic Execution. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 196–206. IEEE CS, 2000.

Troy Pearse and Paul Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 270–277. IEEE CS, 1997.

David Atkins, Thomas Ball, Todd Graves, and Audris Mockus. Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor. *IEEE Trans. Softw. Eng.*, 28(7):625–637, 2002.

Mark Chu-Carroll, James Wright, and Annie Ying. Visual Separation of Concerns through Multidimensional Program Storage. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 188–197. ACM Press, 2003.

Nieraj Singh, Celina Gibbs, and Yvonne Coady. C-CLR: A Tool for Navigating Highly Configurable System Software. In *Proc. Workshop Aspects, Components, and Patterns for Infrastr. Software*. ACM Press, 2007.

Gerard Rambally. The Influence of Color on Program Readability and Comprehensibility. In *Proc. Technical Symposium on Computer Science Education (SIGCSE)*, pages 173–181. ACM Press, 1986.

Bruce Oberg and David Notkin. Error Reporting with Graduated Color. *IEEE Software*, 9(6):33–38, 1992.

Wuu Yang. How to Merge Program Texts. *Journal of Systems and Software*, 27(2):129–135, 1994.

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008.

Krzysztof Czarnecki and Michal Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Proc. Int'l Conf. Generative Programming and Component Engineering (GPCE)*, pages 422–437. Springer, 2005.

Florian Heidenreich, Jan Kopcsek, and Christian Wende. FeatureMapper: Mapping Features to Models. In *Comp. Int'l Conf. Software Engineering (ICSE)*, pages 943–944. ACM Press, 2008.

David Coppit, Robert Painter, and Meghan Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 754–757. IEEE CS, 2007.

Eduardo Figueiredo, Claudio Sant'Anna, Alessandro Garcia, Thiago Bartolomei, Walter Cazzola, and Alessandro Marchetto. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 183–192. IEEE CS, 2008b.

Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Claudio Sant'Anna, Sergio Soares, Paulo Borba, Uira Kulesza, and Awais Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 176–200. Springer, 2007.

Stefan Hanenberg, Sebastian Kleinschmager, and Manuel Josupeit-Walter. Does Aspect-Oriented Programming Increase the Development Speed for Crosscutting Code? An Empirical Study. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 156–167. IEEE CS, 2009.