Empir Software Eng
DOI 10.1007/s10664-012-9208-x

Do background colors improve program comprehension in the #ifdef hell?

Janet Feigenspan · Christian Kästner · Sven Apel · Jörg Liebig · Michael Schulze · Raimund Dachselt · Maria Papendieck · Thomas Leich · Gunter Saake

Editor: Ahmed E. Hassan

Abstract Software-product-line engineering aims at the development of variable and reusable software systems. In practice, software product lines are often implemented with preprocessors. Preprocessor directives are easy to use, and many mature tools

J. Feigenspan (✉) · R. Dachselt · M. Papendieck · G. Saake
School of Computer Science, University of Magdeburg,
Magdeburg, Germany
e-mail: feigensp@ovgu.de

R. Dachselt
e-mail: dachselt@ovgu.de

M. Papendieck
e-mail: maria.papendieck@st.ovgu.de

G. Saake
e-mail: saake@ovgu.de

C. Kästner
Philipps University Marburg, Marburg, Germany
e-mail: christian.kaestner@uni-marburg.de

S. Apel · J. Liebig
University of Passau, Passau, Germany

S. Apel
e-mail: apel@uni-passau.de

J. Liebig
e-mail: joliebig@fim.uni-passau.de

M. Schulze
pure-systems, Magdeburg, Germany
e-mail: michael.schulze@pure-systems.com

T. Leich
Metop Research Institute, Magdeburg, Germany
e-mail: thomas.leich@metop.de

are available for practitioners. However, preprocessor directives have been heavily criticized in academia and even referred to as "#ifdef hell", because they introduce threats to program comprehension and correctness. There are many voices that suggest to use other implementation techniques instead, but these voices ignore the fact that a transition from preprocessors to other languages and tools is tedious, erroneous, and expensive in practice. Instead, we and others propose to increase the readability of preprocessor directives by using background colors to highlight source code annotated with *ifdef directives*. In three controlled experiments with over 70 subjects in total, we evaluate whether and how background colors improve program comprehension in preprocessor-based implementations. Our results demonstrate that background colors have the potential to improve program comprehension, independently of size and programming language of the underlying product. Additionally, we found that subjects generally favor background colors. We integrate these and other findings in a tool called FeatureCommander, which facilitates program comprehension in practice and which can serve as a basis for further research.

Keywords Empirical software engineering · Software visualization · Program comprehension · Software product lines · Preprocessors · FeatureCommander

1 Introduction

Software-product-line engineering provides an efficient means to develop variable and reusable software (Clements and Northrop 2001; Pohl et al. 2005). Different program variants—*variants* for short—of a *software product line (SPL)* can be generated from a common code base by including or excluding features. A *feature* is a user-visible characteristic of a software system (Clements and Northrop 2001). Variable source code that implements a feature is called *feature code*, in contrast to *base code*, which implements the common base shared by all variants of the product line.

There are many technologies for the implementation of SPLs, from conditional compilation (Pohl et al. 2005), to components and frameworks (Clements and Northrop 2001), to programming-language mechanisms such as subjects (Harrison and Ossher 1993), aspects (Kiczales et al. 1997), mixin layers (Smaragdakis and Batory 1998), and to combinations thereof (Apel et al. 2008). Although, in academia, most researchers focus on programming-language mechanisms, in practice, companies implement SPLs mostly with conditional compilation using preprocessor directives. There are many examples of industrial SPLs developed with preprocessors such as HP's product line *Owen* for printer firmware (Pearse and Oman 1997) (honored as best practice in the Software Engineering Institute's *Software Product Line Hall of Fame*). Preprocessors are used to *annotate* feature code with *#ifdef* and *#endif* (or similar) directives, which are removed before compilation (including the annotated code, when certain compiler flags are not set).

Preprocessors are popular in industry, because they are simple to use, are flexible and expressive, can be used uniformly for different languages, and are already integrated as part of many languages or environments (e.g., C, C++, Fortran, and Java Micro Edition) (Favre 1997; Muthig and Patzke 2003). However, in academia, many researchers consider preprocessors "harmful" or even as "#ifdef hell" (Lohmann et al. 2006; Spencer and Collyer 1992), because the flexibility and expressiveness can lead to complex and obfuscated code that is inherently difficult to understand and can lead to high maintenance costs (Favre 1997; Krone and Snelting 1994; Pohl et al. 2005).[1]

Hence, preprocessor usage potentially threatens program comprehension. It is imperative to consider comprehensibility of source code, because understanding is a crucial part in maintenance: Maintenance programmers spend most of their time with understanding code (Standish 1984; Tiarks 2011; von Mayrhauser et al. 1997). Furthermore, most of the costs for developing a software product are caused by its maintenance (Boehm 1981). Hence, by ensuring easy-to-understand source code, we can reduce software development costs.

To increase program comprehension in practice, one could encourage practitioners to use different implementation approaches that modularize feature code, but introducing novel languages or concepts in industry is a difficult process, especially when large amounts of legacy code are involved. Therefore, we target a different question: *Is there a way to improve readability of existing preprocessors to improve program comprehension?*

We propose to use background colors to highlight feature code: In a source-code editor, feature code is displayed with a background color that distinguishes feature code from code of other features and base code.

So far, little is known about the influence of background colors on program comprehension used in source-code editors. To evaluate whether and how highlighting feature code with background colors improves program comprehension in preprocessor-based software, we conducted three controlled experiments with a total number of 77 subjects. In the first experiment, we evaluated whether background colors can improve program comprehension in a preprocessor-based SPL with about 5,000 lines of code and 4 features. We found that colors can speed up the comprehension process in terms of locating feature code up to 43%. In a second experiment, we evaluated whether and how subjects use background colors when given a choice between background colors and *ifdef directives*. We found that subjects preferred background colors. Based on the encouraging results of both experiments, we evaluated in a third experiment whether background-color usage scales to a large SPL with over 99,000 lines of code and 340 features. Here, we also found a speed up of comprehension time in terms of locating feature code of up to 55% when using background colors.

The results of our experiments are promising and provide first insights into the requirements of source-code editors that explicitly support the development of variable software with preprocessors. Based on the results of our experiments, we developed a tool called FeatureCommander (Section 9), which provides scalable, customizable usage of background colors. With FeatureCommander, we provide a good basis for other research groups to analyze how the readability of *ifdef directives* can be improved. Furthermore, we give practitioners a tool that improves program comprehension in preprocessor-based software, which can save time and costs of software maintenance.

The results of the first experiment have been briefly mentioned in a workshop paper motivating empirical research to the SPL community (Feigenspan et al.

[1]We discuss problems arising from preprocessor usage in Section 2.

2009). The focus of this paper was not on the experiment, but on the necessity of empirical research. Furthermore, the results of the third experiment have been published before with focus on tooling (Feigenspan et al. 2011b). Additionally, we have published a tool demo of FeatureCommander (Feigenspan et al. 2011a), focusing on its functionality, but not on empirical evaluation (see Section 10.1 for more details). In this article, we put the focus on details of the experiments and put the results in a broader perspective: Our team, consisting of tool developers, software-engineering researchers, and psychologists, collected empirical evidence on the influence of background colors on program comprehension in the context preprocessor-based SPLs for over two years.

The remainder of the paper is structured as follows: In Section 2, we give an overview of problems caused by the use of *ifdef directives* and present possible solutions. We give an overview of program comprehension and the logic of experiments in Section 3. In Section 4, we describe the common framework of our experiments. In Sections 5–7, we give a detailed description of the three experiments we conducted. We put the results of all three experiments in a broader perspective in Section 8. In Section 9, we present a summary of the results and the prototype implementation of FeatureCommander. We present prior and related work in Section 10 and conclude in Section 11.

2 Welcome to the #ifdef Hell

To implement variable source code, practitioners often use *ifdef directives*, as illustrated in Fig. 1 with an excerpt of Berkeley DB.[2] Identifying code fragments annotated with *ifdef directives* can be problematic, especially when

(1) *ifdef directives* are fine grained,
(2) *ifdef directives* are scattered,
(3) *ifdef directives* are nested, and/or
(4) long code fragments are annotated,

which often occurs in preprocessor-based software (Liebig et al. 2010, 2011).

First, *ifdef directives* can be 'hidden' somewhere within a single statement at a fine grain. For example, a programmer may annotate a variable or a bracket. Such annotations are difficult to locate, because they can hardly be distinguished from 'normal' source code. Another problem is that fine-grained annotations can lead to syntactic errors after preprocessing, because a closing bracket may be annotated, but not the corresponding opening one. Tracking these errors at source-code level is difficult, because both brackets are visible in the source code.

Second, *ifdef directives* are typically scattered across the code base. In Fig. 2, we illustrate this problem with a source-code excerpt from the Apache Tomcat web server, showing session management. Implementing an optional session-expiration mechanism involves the addition of code and *ifdef directives* in many locations. The red background color illustrates the scattering of feature *Session expiration* over the complete implementation of session management, which makes implementing

[2] http://www.oracle.com/technetwork/database/berkeleydb

```
 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
18 }
```

Fig. 1 Code excerpt of Berkeley DB

and tracing this feature a tedious and error-prone task. A developer must take into account all affected modules when keeping track of the *Session-expiration* feature.

Third, *ifdef directives* can be nested. For example, in Fig. 1, Lines 13 to 15 are defined within another *ifdef directive*, starting in Line 5. It might not be difficult to keep track of a nesting level of two (as in this case), which is typical for most projects. However, in practice, nesting levels of up to 24 may occur (Liebig et al. 2010).

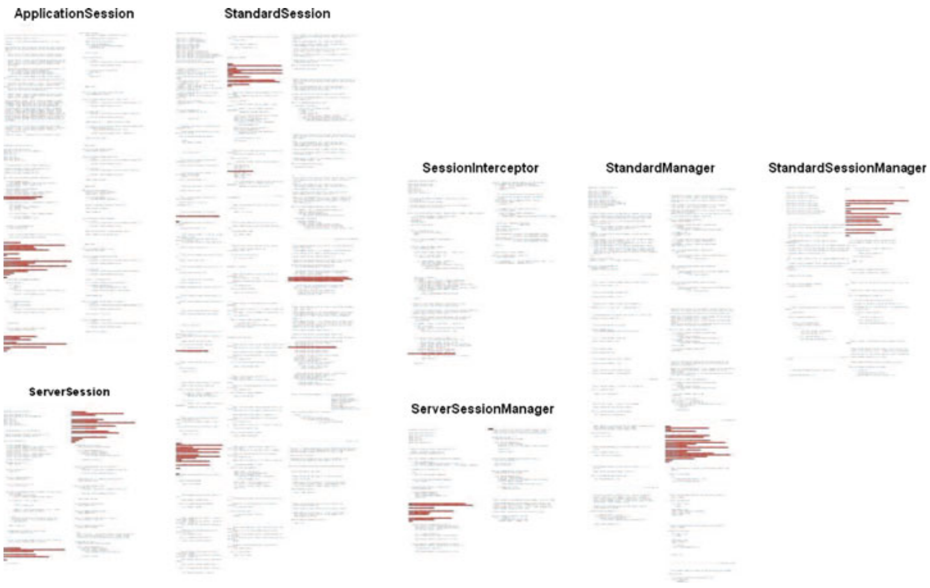


Fig. 2 Apache Tomcat source code illustrating scattering of session-expiration source code. This figure is from a tutorial on AspectJ: http://kerstens.org/mik/publications/aspectj-tutorial-oopsla2004.ppt

Fourth, long code fragments can be annotated, as indicated in Fig. 1: Line 16 states that over 100 additional lines of code occur, after which the according *#endif* of the *#ifndef* in Line 5 occurs. To keep track of this fragment of feature code, a developer typically has to scroll and, thus, keep in mind which code fragments belong to the according feature and which do not. A surrounding annotation might not be visible from the source-code excerpt shown in an editor.

How can we overcome these problems?

2.1 Stairway to Heaven?

To escape the "#ifdef hell", several approaches have been developed that aim at improving the readability of preprocessors, for example, by hiding selected feature code such as in the Version Editor (Atkins et al. 2002), CViMe (Singh et al. 2006), or C-CLR (Singh et al. 2007) or by annotating features with colors such as in Spotlight (Coppit et al. 2007) (with vertical bars next to the code editor), NetBeans (one background color for all features), or CIDE (a previous tool of our's, see Section 10.1) (Kästner et al 2008).

In Fig. 3, we illustrate how background colors can be used to annotate source code. All source-code lines that are annotated are displayed with a background color. Code of feature *HAVE_QUEUE* (Lines 5 to 16) is annotated with yellow background color. The according else directive (Line 8) has the same color, because the according annotated code is also relevant for this feature. Code of feature *DIAGNOSTIC* (Lines 12 to 14) is annotated with orange. In this example, we see how we deal with nested code: We display the background color of the inner feature *DIAGNOSTIC*, which is orange. In an early prototype, we blended the colors of all features that are nested. However, this would introduce more colors than necessary and make distinguishing code of different features more difficult. Additionally, with a deeper nesting level it becomes difficult to recognize all involved features, because the blended colors would result in a shade of gray.

```
 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     return (__db_no_queue_am(dbenv));
 8   #else
 9     db_pgno_t first, last;
10     u_int32_t flags;
11     int empty, ret, t_ret;
12      #ifdef DIAGNOSTIC
13        DB_MSGBUF mb;
14     #endif
15     // over 100 lines of additional code
16  #endif
17 }
```

Fig. 3 Excerpt of Berkeley DB with background colors to highlight feature code. Lines 5 to 16 are *yellow*, Lines 12 to 14 *orange*

With background colors, we use a *highlighting* technique that supports users in finding relevant information (Fisher and Tan 1989; Tamborello and Byrne 2007). Highlighting emphasizes objects that users might look for, such as menu entries or certain code fragments. It can be realized with different mechanisms, such as blinking or moving an object. In past work, colors have been shown to be effective for classifying objects into separate categories and can increase the accuracy in comprehension tasks (Chevalier et al. 2010; Fisher and Tan 1989; Ware 2000).

The benefit of colors compared to text-based annotations is twofold: First, the background colors clearly differ from source code, which helps distinguish feature code from base code. Second, humans process colors preattentively[3] and, thus, considerably faster than text (Goldstein 2002). This allows a programmer to identify feature code at first sight and distinguish code of different features. As a consequence, a programmer should be able to get an overview of a software system considerably faster.

Based on the comparison of the code fragments in Figs. 1 and 3, one could intuitively argue that one approach is better than the other or that both should be combined. For example, one could argue that colors are distracting (Fisher and Tan 1989) or do not scale for large SPLs, or colors do improve program comprehension due to preattentive perception (Goldstein 2002). So, we can discuss both benefits and drawbacks of colors, and the effect of background colors is not as obvious as it may appear at first sight. However, since program comprehension is an internal cognitive process, we can only assess it empirically (Koenemann and Robertson 1991)—plausibility arguments are not sufficient. Hence, to answer whether background colors improve the readability of preprocessor directives, we need to conduct controlled experiments. In this paper, we evaluate in three controlled experiments, whether

– background colors improve program comprehension at all (*Experiment 1*),
– subjects use background colors when given the choice (*Experiment 2*), and
– the use of background colors scales to large product lines (*Experiment 3*).

3 Measuring Program Comprehension

To evaluate how background colors influence program comprehension, we have to take care of two things: First, we have to measure program comprehension and, second, we have to control confounding variables for program comprehension. In this section, we explain how we can take care of both. Readers familiar with empirical work may skip this section. It is aimed to support researchers and practitioners of the SPL community who might not be familiar with empirical research.

3.1 Program Comprehension Measures

Program comprehension is an internal cognitive process, which means that it cannot be observed directly (Koenemann and Robertson 1991). To understand the complexity of program comprehension, we give a short introduction. Typically,

[3]Preattentive perception is the fast recognition of a limited set of visual properties (Goldstein 2002).

models of program comprehension describe top-down, bottom-up, and integrated comprehension. Top-down comprehension is used when a programmer is familiar with a program's domain (e.g., operating systems). *Beacons* (i.e., familiar code fragments or identifiers) help to form an understanding of source code (Brooks 1978). Using top-down comprehension, a developer forms a general hypothesis of a program's purpose and refines this hypothesis by analyzing source code in more and more detail. Examples of top-down models are described by Brooks (1978), Shaft and Vessey (1995), and Soloway and Ehrlich (1984). If a developer has no domain knowledge, she uses a bottom-up approach, which means she analyzes the source code statement by statement. She groups source-code fragments into *semantic chunks* and—by combining these chunks—generates hypotheses about a program's purpose. Examples of bottom-up models can be found in Pennington (1987) and Shneiderman and Mayer (1979). Typically, a developer uses top-down comprehension where possible and switches to bottom-up comprehension where necessary. This behavior is described by integrated models, for example, by von Mayrhauser et al. (1997) and von Mayrhauser and Vans (1995).

Program comprehension is a rather complex process for which we have to find a reliable measure to assess it. Several methods to measure program comprehension have been proposed in the literature, for example, think-aloud protocols (Someren et al. 1994) or tasks that can be solved only if a programmer understands a program. Typical kinds of such tasks include static tasks (e.g., examine the structure of source code), dynamic tasks (e.g., examine the control flow), and maintenance tasks (e.g., fix a bug), as summarized by Dunsmore and Roper (2000). Furthermore, we need to choose a concrete measure for a task, such as response time or correctness of a solution (Dunsmore and Roper 2000).

In our experiments, we use static and maintenance tasks and analyze response times and correctness of solutions. We use static tasks, because locating feature code is one major part of comprehending source code annotated with *ifdef directives*. For example, in Fig. 2, we can see that source code of feature *Session expiration* is scattered over the complete software system. Hence, locating all occurrences of this feature is one important step in comprehending this feature (e.g., when we are searching for a bug that we know is related to feature *Session expiration*). We decided to use maintenance tasks, because, if subjects could offer a solution for a bug, then program comprehension must have taken place. Additionally, a lot of experiments described in the literature use tasks, as well, so we can relate our results to other experiments (e.g., Boysen 1977; Hanenberg 2010; Prechelt et al. 2002).

3.2 Rationale of Experiments

When conducting experiments, confounding variables need to be controlled. *Confounding variables* influence program comprehension in addition to the intended variables (in our case, the kind of annotation, either background colors or *ifdef directives*). Examples are the programming experience of subjects or the underlying programming language. Both may bias the results and can lead to a false outcome.

Confounding variables threaten the *validity* of results if not handled correctly. Two kinds of validity are typically considered: Internal (the degree to which we have controlled confounding variables) and external validity (the generalizability of results). In our experimental settings, we maximize internal validity, so that we can draw sound conclusions from our results. For example, we keep the influence of confounding parameters on program comprehension constant (e.g., programming experience, domain knowledge). As a consequence, we can attribute the measured differences regarding program comprehension to the different kinds of annotation. However, at the same time, this focus on internal validity limits external validity. For example, in the first experiment, we measure the influence of annotations for *specific tasks* in a *specific program* with only *four features* in a *specific domain* with *students*. To be able to generalize the results to other tasks, domains, programs at different scales in different programming languages, or professional programmers, further investigations are necessary. Our experiments and tool FeatureCommander can be the basis for such follow-up experiments.

Another reason for focusing on internal validity and not conducting more experiments with high external validity is the feasibility (Hanenberg 2010; Tichy 1998). Preparing and designing experiments requires considerable effort: We have to identify and control confounding variables, design the experimental material and tasks, for which we needed several months and a master's thesis (Feigenspan 2009), only for the first experiment. We had to find appropriate subjects (i.e., who are familiar with SPL and preprocessor directives). In our case, we were rather lucky, because one co-author offers an advanced programming-paradigm lecture at his university, from which we could recruit our subjects.

Hence, replicating experiments with slightly modified settings requires often too much effort for one research group. Instead, it is reasonable and necessary to publish results even with a narrow scope, because it makes other research groups aware of interesting topics. It is necessary to motivate other research groups to conduct experiments on the same topic, because they may have the resources or suitable subjects or ideas to extend the results obtained in one experiment.

To enable researchers to replicate experiments and to check how well threats to validity have been controlled, the experimental design, conduct, analysis, and interpretation have to be presented in sufficient detail. Some redundancy is necessary, especially when describing three experiments. In the next sections, we give an overview of all three experiments and present our them in a proper detail. Material of all three experiments is available online.[4]

4 Family of Experiments

In this paper, we present three controlled experiments that analyze whether and how background colors can improve the readability of preprocessor directives. Each experiment focuses on a different aspect of background-color usage. By putting the results of all three experiments together, we aim at providing a deeper understanding of the effect of background colors on program comprehension in preprocessor-based SPLs. For a better overview, we describe each experiment using the goal-question-metric approach in Table 1 (Basili 1992).

The focus of the first and third experiment lies on program comprehension, whereas the focus of the second experiment lies on the behavior of subjects, i.e., how

[4]http://fosd.net/experiments

Table 1 Description of all three experiments using the goal-question-metric approach. We emphasized differences of experiments

GQM	Experiment 1	Experiment 2	Experiment 3
Analyze	Background colors	Background colors	Background colors
Purpose	Evaluation	Evaluation	Evaluation
With respect to	Program comprehension	*Use of opportunity to switch*	Program comprehension
Point of view	Developer	Developer	Developer
Context	Medium preprocessor-based SPLs	Medium preprocessor-based SPLs	*Large preprocessor-based SPLs*

subjects use the opportunity to switch between background colors and preprocessor directives. The context of the first and second experiment is on medium-sized SPLs, whereas the last experiment uses a large SPL. In all other criteria of the goal-question-metric approach, the experiments are the same. Due to this small delta between the experiments, we can thoroughly investigate the effect of background colors on the readability of preprocessor-based software.

Combining the results of all three experiments lets us draw conclusions about the scalability of background-color usage. Since in the first experiment we showed improvements of program comprehension using a medium-sized SPL, and in the third experiment we also showed an improvement, but using a large SPL, we showed a scalable use of background colors. Although we have no results for a small SPL, we argue that we would observe an improvement in program comprehension, too, because the limits to human perception are stressed even less.

To avoid threats to validity of our results by introducing learning or maturation effects, we recruited different subjects for the first two experiments. In the third experiment, one subject participated who also took part in the second experiment. However, since we had different research hypotheses and different material, we argue that no learning or maturation effects could have occurred.

In the next sections, we present each experiment in detail. The detail is the greatest for the first experiment, because we need to introduce the material, setting, and tasks. In the subsequent experiments (Sections 6 and 7), we focus more on the differences of the experiments to the first experiment. Nevertheless, the description may seem redundant. However, we aim at providing as much detail as possible to enable other researchers to replicate any of the three experiments. To put the results of all three experiments in a broader perspective, we explain our conclusions based on all three experiments in Section 8.

5 Experiment 1: Can Colors Improve Program Comprehension?

In this section, we present the design of our first experiment. In a nutshell, we evaluated whether background colors improve program comprehension in preprocessor-based SPLs compared to *ifdef directives*, by means of a medium-sized[5] Java-based SPL with four optional features. We found that, for locating feature code, back-

[5]Size is between 900 and 40,000 lines of code (von Mayrhauser and Vans 1993).

ground colors significantly speed up the comprehension process, but also that unsuitable background colors can slow down program comprehension. In the next sections, we describe our experiment, including design, conduct, analysis, and interpretation in detail. For all three experiments, we use the guidelines presented by Kitchenham et al. to present empirical studies (Kitchenham et al. 2008).

5.1 Experiment Planning

5.1.1 Objective

The objective of this experiment is to evaluate the effect of background colors on program comprehension in preprocessor-based SPLs. We expect that colors indeed improve program comprehension because of two reasons: First, background colors clearly differ from source code, which allows a human to easily locate feature code (which is annotated with a background color) and tell it apart from base code (which has no background color). Second, humans process colors preattentively, which means that they do not have to turn their attention to the perceptions process (Goldstein 2002). Hence, the perception process is very fast, so that humans can spot a color at first sight. However, if the number of colors gets too large, humans have to turn their attention to identify them. So, at least for an SPL with a small number of features, we expect that background colors allow subjects to locate feature code faster, compared to conventional *ifdef directives*. Hence, we restrict our evaluation to a medium-sized SPL with only a few features.

We distinguish static tasks, in which subjects should locate feature code, and maintenance tasks, in which subjects should identify a bug. Since in maintenance tasks, subjects should spend most of their time with closely examining code fragments, we do not expect a strong improvement by colors.

Additionally, both annotations provide the same amount of information, that is, information about feature code and to which feature it belongs. Hence, we do not expect a difference in correctness of answers, but only in response time. Thus, we state the following research hypotheses for medium-sized SPLs:

RH1: In static tasks, colors speed up program comprehension compared to *ifdef directives*.
RH2: In maintenance tasks, there are no differences in response time between colors and *ifdef directives*.
RH3: There are no differences in the number of correctly solved tasks between colors and *ifdef directives*.

Another hypothesis is based on an observed mismatch between actual and perceived performance (Daly et al. 1995) and empirical evidence that subjects like the idea of combining colors and source code (Rambally 1986). In a study, Daly et al. (1995) found that subjects estimated their performance worse than it actually was, when they worked with a source-code version they did not like. We expect that subjects like the color idea and that this influences their estimation of performance. Hence, our fourth research hypothesis is:

RH4: Subjects estimate better performance with background colors than with *ifdef directives*.

Next, we present the material we used to evaluate our research hypotheses.

5.1.2 Experimental Material

For the first experiment, we decided to use source code that is implemented in Java, because we had the opportunity to work with a large group of subjects experienced with Java. Furthermore, variability is also required in Java and sometimes conditional compilation is used for product-line development, especially in the domain of embedded and mobile devices, using the Java Micro Edition—a Java version developed for embedded devices (Riggs et al. 2003).

As material, we used the medium-sized SPL *MobileMedia* for manipulating multimedia data on mobile devices, which was developed by Figueiredo et al. (2008). It is implemented in Java with the Java ME preprocessor *Antenna*, which provides *ifdef directives* like the C preprocessor, but requires that *ifdef directives* are stated in comments, so that they do not interfere with the Java syntax in existing editors. MobileMedia is well designed, code reviewed, and provides a suitable complexity for our study with about 5,000 lines of code in 28 classes and four optional features (*SMSFeature*, *CopyPhoto*, *Favourites*, *CountViews*).[6] On three occurrences, two features (i.e., *SMSFeature* and *CopyPhoto*) share code, which is included for compilation if at least one of both features is selected. MobileMedia is neither too small, so subjects could understand it after the first task, nor too large, so subjects spend their time sifting through source code that is irrelevant for a task. Additionally, this size (i.e., four features) ensures preattentive color perception, which is necessary to test our hypotheses.

From the original source code annotated with *ifdef directives* (referred to as *ifdef version*), we created a version that uses background colors (referred to as *color version*) instead of *ifdef directives*. The decision not to combine background colors and *ifdef directives* may seem puzzling at first. However, to the best of our knowledge, there is no prior empirical work regarding the effect of colors on program comprehension in the context of preprocessor-based SPLs on which we can base our experiment. Thus, to not confound the effect of text and background colors, we explicitly compare the two extremes of pure textual annotations versus pure graphical annotations with background colors.[7] In our third experiment, we combine both kinds of annotation.

For code fragments that were shared by the features *SMSFeature* and *CopyPhoto* (see Fig. 3 for an example of shared/nested code), we selected a separate color. We selected the following bright and clearly distinguishable colors as background colors:

- *SMSFeature*: red (rgb: 255-127-127)
- *CopyPhoto*: blue (rgb: 127-127-255)
- *Favourites*: yellow (rgb: 255-255-127)
- *CountViews*: orange (rgb: 255-191-127)
- *SMSFeature & CopyPhoto*: violet (rgb: 170-85-170)

[6]MobileMedia was developed in eight releases, from which we took the fifth, because it offered the best balance between size and complexity for our experiment. We omitted 9 exception classes and 2 small features for different screen resolutions, because they are irrelevant for understanding the source code and fixing the bugs.

[7]In the source code, there is no #else combination of *ifdef directives*, so it was always clear from the background colors that feature code concerned selected features.

The color selection is not optimized for avoiding visual fatigue or for color blindness. Instead, we selected the colors such that they are clearly distinguishable. At the time we designed this experiment, we did not consider guidelines for choosing color palettes (e.g., Levkowitz and Herman 1992; Rice 1991; Wijffelaars et al. 2008). However, for the third experiment, we took existing guidelines into account (cf. Section 7.1.2). Nevertheless, since we are exploring whether background colors can improve program comprehension in preprocessor-based SPLs at all, and the chosen colors are clearly distinguishable, the color selection is suitable to test our hypotheses.

To exclude the influence of tool support (such as navigation support, outline views, code folding, etc., with which some subjects may be more familiar than others), we created an HTML page for each source-code file with the default Eclipse syntax highlighting and presented it in a browser (Mozilla Firefox). Furthermore, searching functionality could be provided for both textual annotations and colors with proper tool support, but we decided to forbid search to exclude this influence of tool support as well. Again, we ensure a high degree of internal validity this way. To present the tasks to subjects and collect their answers, we used a web-based survey system.

To evaluate our last hypothesis, whether subjects prefer the *color version* over the *ifdef version* (RH4), we gave subjects a paper-based questionnaire at the end of the experiment, in which they should evaluate their motivation to solve the task and whether their performance would have increased with the other version of the source code, both on a five-point Likert scale (Likert 1932). Additionally, we encouraged subjects to leave remarks (e.g., about the experimental setting), in this and the other experiments, as well.

5.1.3 Subjects

We recruited 52 students from the University of Passau in Germany who were enrolled in the 2009 graduate course *Modern Programming Paradigms (German: Moderne Programmierparadigmen)*. We chose this course, because students were introduced to SPLs and according implementation methods (including an assignment on preprocessor-based implementations). This way, we did not have to train the subjects specifically for this experiment, but they learned the necessary information in the lecture. Students were required to participate in our experiment to finish the course, which could have influenced their motivation. However, for all tasks, we found a medium to high motivation (determined by the questionnaire). Subjects could enter a raffle for a gift card (30 Euros). In this and the other two experiments, as well, all subjects were aware that they participated in an experiment, that their performance does not affect their grade for the course, and that they could leave any time.

Since programming experience is a major confounding parameter for program comprehension, we measured and controlled it. To this end, we administered a programming-experience questionnaire six weeks before the experiment, in which a low value (minimum: 5) indicates no experience, a high value (over 60—the scale is open-ended) high programming experience (see Feigenspan (2009) for details on the questionnaire). We used the value of the questionnaire to create homogeneous groups regarding programming experience (for the remaining experiment as well). To ensure genuine answers, we anonymized our subjects, such that the answers in the questionnaire (or the experiment) cannot be traced back to the identity of subjects. Additionally, we asked with which domains subjects were familiar and whether subjects were color blind. One color blind subject worked with the *ifdef version* of the source code. For our analysis, we had to exclude nine subjects who did not complete the programming-experience questionnaire or did not complete it genuinely (which was obvious from the answers). Hence, our sample consisted of 43 subjects.

5.1.4 Tasks

For assessing program comprehension, we designed two static tasks (S1, S2) and four maintenance tasks (M1–M4).

In static tasks, subjects should locate feature code. In the first static task (S1), subjects should, for each feature, locate all files containing feature code and mark the results on a sheet of paper (referred to as *grid template*). It showed the relationship of code to features in a matrix, such that the columns contained the file names, and the rows the feature names. For the *color version*, the feature names of the grid template had the same background color as in the source code, whereas for the *ifdef version*, the grid template had no background colors. In the second static task (S2), subjects should locate shared code (i.e. code that concerned more than one feature, e.g., *SMSFeature & CopyPhoto*). Locating feature code is a typical task for a developer, when she is familiarizing herself with an SPL. Furthermore, a developer is often looking for feature code when solving a bug, because bugs can often be narrowed down to certain features or feature combinations. Especially, combinations of features are of interest in the implementation of SPLs, since they can represent feature interactions that are especially difficult to get right (Kästner 2010).

For all maintenance tasks, we carefully introduced different bugs into the source code, which were all located in annotated code fragments. In a pre-test with 7 students, we selected bugs that were neither too easy nor too difficult to find. Four bugs matched our criteria, which we ordered by increasing difficulty according to the results of our pre-test. For each bug, subjects received a bug description similar to the ones users would enter in a bug-tracking system. The description also named the feature in which the bug occurs. This assured that we evaluate the influence of background colors, because subjects focus on feature code and, thus, background colors, instead of spending their time in non-annotated code fragments. For each task, subjects should locate the bug (name class and method), explain why it occurs, and suggest a solution. Using this information, we judged whether the cause of a bug was located correctly.

As an example, we present the bug description of the first maintenance task:

M1: If pictures in an album should be sorted by views, they are displayed unsorted anyway. Feature, in which the bug occurs: *CountViews*.

The bug was located in the class `PhotoListController` and caused by an empty method body of `bubbleSort`.

In addition to the six tasks, we designed a warming-up task to let subjects familiarize with the experimental setting (subjects should count the number of features of MobileMedia). The result of this task was not analyzed.

5.1.5 Design

To evaluate our research hypotheses, we used a between-subjects design, which means we split our sample in two groups and compared the performance between both groups, the *ifdef group* (21 subjects) and the *color group* (22 subjects). The *ifdef group* worked with the *ifdef version*, the *color group* worked with the *color version* of the source code. To assure that both groups are comparable, we matched both groups according to the value of the programming experience questionnaire, age, and gender. One subject was color blind and assigned to the *ifdef group*.

5.1.6 Conduct

The experiment took place in June 2009 in Passau during a regular lecture session in a room with about 50 computer working stations. All computers had Linux as operating system and 19" TFT screens. We started with an introduction, in which we recapitulated relevant terms regarding preprocessors and background colors as annotation. After all questions were answered, each subject was seated at a computer and started to work on the tasks on her own. Each task had the same structure: First, the task was introduced and it was explained what we expected from the subject. Second, when subjects were clear on the instructions, they displayed the next page with the concrete task. Only the latter part was measured as response time.

The experiment (and the remaining two, as well) lasted about two hours, including the introduction. Subjects worked by themselves during that time, including the decision to move on the next task. If subjects completed all tasks, they could leave quietly without disturbing the others. After the two hours were over, subjects were not allowed to finish the tasks. Three experimenters regularly checked that subjects worked as planned without using additional tools such as the search function of the browser. A few weeks after the experiment, subjects were told the correct answers of the tasks in a lecture, as well as some information about the purpose and results of the experiment.

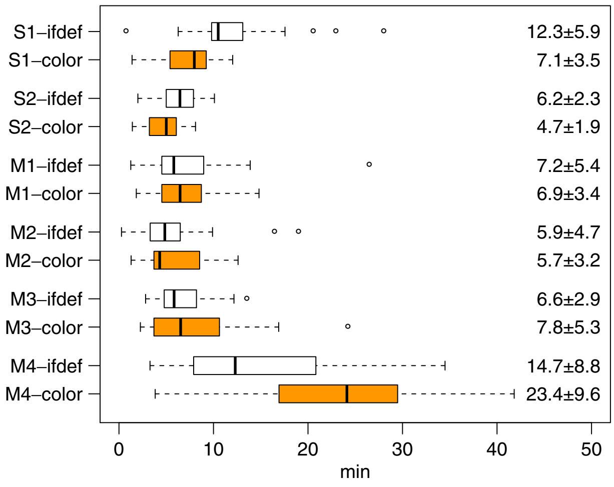
5.1.7 Deviations

Despite all careful planning, deviations occurred, which is common for *every* experiment. Hence, it is important to describe deviations, so that the reader can take them into account when interpreting our results. Additionally, other researchers who plan to replicate the experiment are prepared and can avoid these deviations.

For one subject of the *color group* we had no grid template, so she worked with a grid template of the *ifdef group* instead (in which the features had no background colors). Furthermore, some subjects arrived late and were seated in another room to not disturb the others. In order not to jeopardize their anonymity, we decided not to track them. Our sample is large enough to compensate for these deviations.

In addition, for estimating performance with the other version at the end of the experiment, we forgot to include the last task, because we had one task less in the pre-test. As soon as we noticed that, we asked subjects to evaluate the seventh task on the sheet of paper. Unfortunately, some of the subjects had already left the room at that time, so we only have the opinion for that task of 13 subjects of the *ifdef group*, and 16 subjects of the *color group*. We discuss the influence of all deviations on our results in Section 5.4.

Fig. 4 Experiment1: response times for static (S1–S2) and maintenance tasks (M1–M4). *Colored/gray boxes* refer to the *color group*. Numbers on the right denote mean ± standard deviation



5.2 Analysis

In this section, we present the analysis of our data. It is necessary to strictly separate data analysis from interpretation (which follows in Section 5.3), so that a reader can draw her own conclusions of our data and other researchers replicating our experiments can compare their data with ours.

5.2.1 Descriptive Statistics

The descriptive statistics of response times and correct solutions can be found in Figs. 4[8] and 5. The differences in response time are the largest for the first task (ifdef: 12 min, color: 7 min) and last task (ifdef: 15 min, color: 23 min). Furthermore, the last task took the most time to complete.

Regarding correct solutions, we can see in Fig. 5 that most errors occurred for static tasks. Moreover, the difficulty of the maintenance tasks seems to increase for the last tasks.

For the estimation of performance with the other version (cf. Section 5.1.6), subjects who worked with the *ifdef version* thought that they would have performed equivalently or better with the *color version* (medians for each task vary from 3 to 5), and subjects who worked with the *color version* thought they would have performed worse with the *ifdef version* (medians are 2 for each task).

5.2.2 Hypotheses Testing

To evaluate our research hypotheses, we applied a number of statistical tests. They indicate whether an observed difference is significant or more likely to be caused

[8]Figure 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.

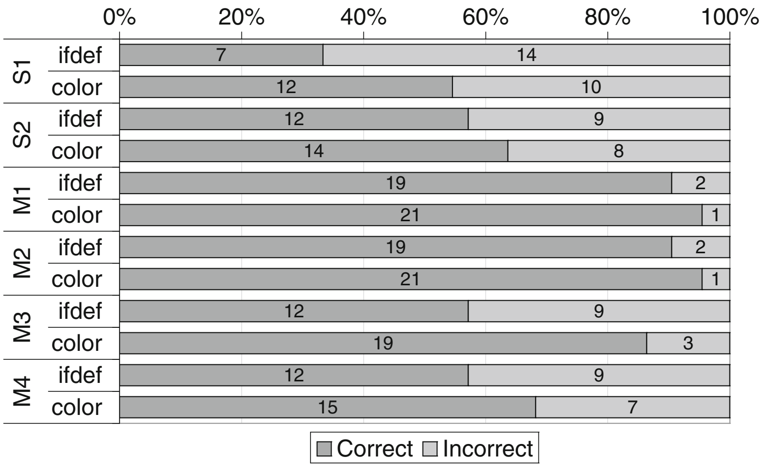


Fig. 5 Experiment1: frequencies of correct solutions

randomly (Anderson and Finn 1996). Based on a probability value or *significance level* (p value), hypotheses are rejected (> 0.05, i.e., observed difference occurred randomly) or accepted (≤ 0.05, i.e., observed difference is statistically significant).

To test **RH1** and **RH2** (response times for static/maintenance tasks), we conducted a Mann–Whitney–U test (Anderson and Finn 1996), because the response times are not normally distributed (as revealed a Shapiro–Wilk test (Shapiro and Wilk 1965)). Since the correctness of a solution can have an influence on response time (e.g., a subject may deliberately enter a wrong solution just to be faster, Yellott 1971), we omitted response times for wrong answers. Our sample is large enough to compensate the missing cases. The observed differences for both static tasks regarding response time are significant, such that subjects who worked with the *color version* were faster (S1 & S2: $p < 0.001$). Hence, we can accept our first research hypothesis. To have a better impression of the size of the effect, we also computed the effect sizes for both tasks. Since we used a non-parametric test, we computed Cliff's delta (Cliff 1993). For S1, Cliff's Delta is −0.61, indicating a large effect. For S2, the value is −0.39, which indicates a medium effect.

For three of the four maintenance tasks, we found no significant differences in response time. For the last maintenance task (M4), subjects with the *color version* were significantly slower than subjects with the *ifdef version* (M4: $p < 0.04$). Thus, we reject our second research hypothesis. Cliff's Delta for the last maintenance task is 0.49, indicating a large effect.

For the number of correctly solved tasks (**RH3**), we conducted a χ^2 test (Anderson and Finn 1996), which checks whether the observed frequencies significantly differ from expected frequencies under the assumption that the null hypothesis is valid (i.e., that no differences between number of correct answers exist). We found no significant differences in the correctness for any task. Hence, we can accept our third research hypothesis.

For the estimation of performance with the other version (**RH4**), we conducted a Mann–Whitney–U test (because the data are ordinally scaled) and found significant differences for all tasks in favor of the *color version* ($p < .013$ for M4, $p < 0.001$ for all other tasks). Hence, we can accept our last research hypothesis.

5.3 Interpretation

RH1 *Response Time for Static Tasks* Regarding static tasks, we can accept that colors speed up program comprehension in preprocessor-based SPLs, compared to *ifdef directives*, because the observed differences in response time for both static tasks are significant. In S1, the speed up is 43%, in S2 it is 25%. The effect sizes indicate a large (S1) and medium (S2) effect, showing that not only the size of our sample lead to a significant difference. We can explain this difference with the preattentive color perception, compared to attentive text perception (Goldstein 2002). Subjects of the color group have to look only for a color, not read text to solve these tasks. However, the benefit in S2 is smaller than in S1. We suspect two reasons responsible for the difference between S1 and S2: First, when subjects searched for shared code in S2, they had already familiarized themselves with the source code in the warming-up task and in S1. Second, in S1, subjects that worked with the *color version* could simply check whether a background color was present in a class at all and then mark it in the grid template accordingly. However, in S2, they additionally had to discriminate different background colors, not only recognize the presence of a background color. Both reasons could lead to the decrease in the performance benefit for S2. In summary, when a developer needs to get an overview of an SPL, background colors can speed up the familiarization.

RH2 *Response Time for Maintenance Tasks* For the first three maintenance tasks, there is no significant difference in response times. However, for the last maintenance task, subjects of the *color group* were significantly slower (35%) than subjects of the *ifdef group*. Cliff's Delta shows a large effect, indicating the importance of this difference. Hence, we cannot accept our second research hypothesis.

To understand what could have caused the slow-down, we take a closer look at how the last maintenance task differs from the other three maintenance tasks. Therefore, we examine the location of the bug of M4: class `SmsSenderController`. Since the entire class belongs to the feature *SMSFeature*, it is entirely annotated with a red background in the *color version*. This is in contrast to the other bugs, where only small parts of a class were annotated, none of them with red. When looking through the comments subjects were encouraged to leave, we found that some subjects criticized the annotation with red in this task.

We conclude that colors can also negatively affect program comprehension if not chosen carefully (i.e., if they are too bright and saturated). Consequently, we have to carefully consider which colors to use, because an unsuitable color (e.g., saturated red) can make the source code difficult to read or cause visual fatigue, which can negatively affect program comprehension.

RH3 *Correctness of Solutions* Although subjects of the *color group* performed slightly better in most tasks and solved more tasks correctly (cf. Fig. 4), this difference is not significant. Since both kinds of annotation provide information about feature code and the feature to which it belongs, subjects are enabled to correctly solve our tasks, independently of the kind of annotation. The kind of annotation only influences the response time.

RH4 *Estimation of Performance* Almost all subjects who worked with the *ifdef version* estimated that they would have performed better with the *color version*, whereas subjects who worked with the *color version* thought they would have performed worse with the *ifdef version*. This counts even in the last task, in which subjects of the *color group* were significantly slower than subjects of the *ifdef group*. Hence, we found a strong effect regarding subjects' estimation that is in contrast to subjects' actual performance. When looking through the comments of subjects, we found that some subjects of the *color group* were happy to get to work with it, whereas some subjects of the *ifdef group* wished they had worked with the *color version*. This could explain the difference in estimating the performance, because some subjects liked the *color version* better, which they reflected to their performance.

5.4 Threats to Validity

5.4.1 Internal Validity

Some threats to internal validity are caused by the deviations that occurred (cf. Section 5.1.6). However, to assure anonymity of our subjects, we did not retrace the deviations to the subjects. Our sample is large enough to compensate the deviations. They may have intensified or weakened the differences we observed, but they were too small compared to our large sample to significantly bias our results.

A further threat to internal validity is caused by our programming-experience questionnaire. Since no commonly accepted questionnaire to measure programming experience exists, we designed our own. Hence, we cannot be sure how well we have measured programming experience. However, we constructed the questionnaire with the help of programming experts and a literature review (cf. Feigenspan (2009) for more details), so we can assume that we measured programming experience well enough for our purpose.

Another threat might be the different reading times of the subjects. To diminish this threat, we split the task description in two parts, such that we first explained the general settings of the task and what we expect from them, and when subjects were clear on these instructions, they could display the actual task. Only the time of the actual task is measured as response time. Additionally, the description of the actual tasks were kept as short as possible, such that subjects knew what to do, but had not to read too much text. Hence, we argue that the reading time of subjects did not significantly influence the outcome.

5.4.2 External Validity

In this experiment, we maximized internal validity to feasibly and soundly measure the effect of different annotations on program comprehension in pre-processor-based SPLs. Thus, we deliberately accepted reduced external validity as tradeoff for increased internal validity (cf. Section 3.2). In the experiments to follow, we generalize our experimental settings based on sound results to increase external validity.

One important issue is the selection of colors. We selected the colors, because they are clearly distinguishable for subjects. If we chose other colors (e.g., less saturated), we could have received different results (e.g., no significant differences for the last maintenance task). However, we wanted to make sure that colors are easily perceived and distinguished by subjects. In our third experiment (Section 7), we use different color settings to generalize our results regarding the use of colors and find optimal colors for highlighting feature code.

Another important aspect of our experiment, which influences external validity, is whether colors scale for a large number of features. Since we had an SPL with only four features, we cannot generalize our results to larger SPLs. To address this threat, we conducted the third experiment, which we explain in Section 7. Next, we evaluate whether subjects prefer colors over *ifdef directives* when given the choice.

6 Experiment 2: Do Subjects Use Colors?

The results of our first experiment indicate that subjects like the color idea, but that carelessly chosen colors are disturbing (as some subjects noted) and can slow them down. This indicates that different kinds of annotations might be suitable for different tasks, and we should offer developers the opportunity to switch between them as needed for the task at hand. Hence, instead of evaluating whether background colors affect program comprehension, we evaluated whether developers would use the option to switch between background colors and *ifdef directives*. Our results indicate that subjects prefer background colors, even if they slow them down. We had the chance to perform this experiment twice, first in 2010, then we replicated it with different subjects with similar background in 2011. Hence, we have two instances of our second experiment. Since both instances differ only in few details, we describe them together, and present information about the replication in angle brackets, ⟨like this⟩.

6.1 Experiment Planning

The setting of both instances of our second experiment is very similar to our first experiment. Hence, we concentrate on the differences.

6.1.1 Objective and Material

The goals of the follow-up experiment are different than of the first experiment: Rather than examining the effect of background colors on program comprehension in preprocessor-based SPLs, we evaluate whether and how subjects use the chance to switch between *ifdef directives* and colors as annotations. Based on the insights from the first experiment, we state the following hypothesis:

RH5: For locating feature code, subjects use colors, while for closely examining feature code, subjects use *ifdef directives*.

We used the same source code and background colors as for our first experiment. To present the source code, we implemented a tool similar to the browser setting. In addition, we provided two buttons to enable subjects to switch easily between *color* *version* and *ifdef version*. Our tool logged each button click with a time stamp, such that we can analyze the behavior of subjects.

6.1.2 Subjects and Tasks

We asked students who were enrolled in the 2009 ⟨2010⟩ course about product-line implementation at the University of Magdeburg, Germany to participate, which was one of multiple alternative prerequisites to pass the course. The course was very similar to that of our first experiment (cf. Section 5.1.3), so the background of students was comparable. Additionally, two graduate students who attended that course in the fall term 2008 volunteered to participate as well. Altogether, our sample consisted of 10 ⟨10⟩ subjects. One week before the experiment, we administered the same programming experience questionnaire as in the first experiment. None of the subjects was color blind, and 1 ⟨0⟩ was female.

We used the same tasks as for our first experiment, including the warming-up task (W0). However, we changed the order of the tasks to M1, M3, S1, M4, M2, S2. We alternated static and maintenance tasks, such that we could observe whether subjects actually switch between both representations in line with our hypothesis.

6.1.3 Conduct

We booked a room with 16 seats. All computers had Windows XP as operating system and 17" TFT screens. The experiment took place in January 2010 ⟨January 2011⟩ in Magdeburg instead of a regular lecture session. We gave the same introduction as for the first experiment, with the addition that we showed how subjects could switch between *ifdef directives* and background colors. We did not provide any information on which annotation style is most suitable for which task, so that we could observe the behavior of subjects unbiased. Since we had a smaller sample, two experimenters ⟨one experimenter⟩ sufficed to conduct the experiment.

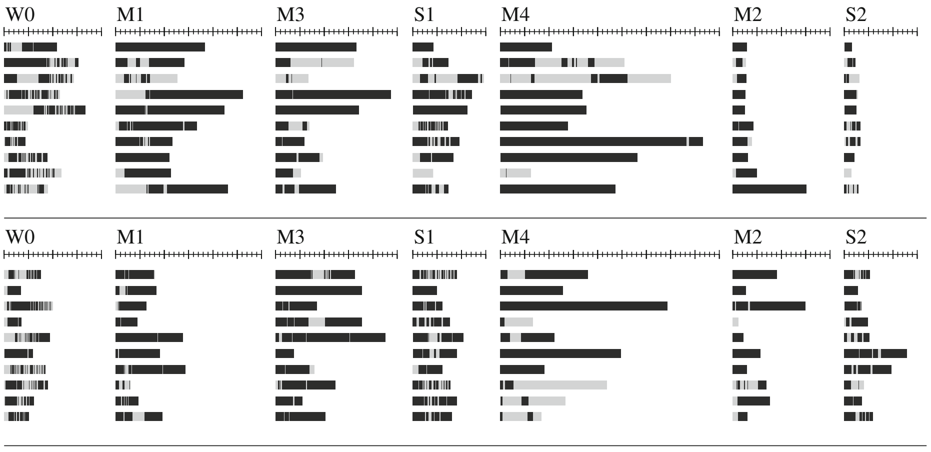
Having learned from our first experiment, we made sure that the same deviations did not occur. There are no other deviations to report.

6.2 Analysis

We show only the information necessary to evaluate our hypothesis. Figure 6 shows how subjects switched between the annotation styles in each task (light gray: ifdefs; dark gray: colors). Each row denotes the performance of a subject. For example, if we look at the first row, we can see that for W0 (warming-up task), the subject switched between annotation styles (light and dark gray alternate). For all remaining tasks, the subject used background colors only.

The lengths of the bars indicate the time subjects spend with a task. For example, the first subject needed considerable more time to solve M1 than to solve M2.

An interesting result can be seen in M4, the task, in which the target code was annotated with a red background color and subjects of the *color group* performed significantly worse in our first experiment. Although subjects of our first experiment complained about the background color, most subjects of our follow-up experiment used mainly the *color version*; only 3 of 10 ⟨4 of 10⟩ subjects spent more time with the *ifdef version*.



Legend: ▭ : ifdef; ■ : color; ⊢⊣ : 5 minutes

Fig. 6 Experiment 2: timeline how subjects switched between textual and colored annotations. *Top*: first instance 2010; *bottom*: second instance 2011

In this figure, we included the warming-up task W0 (counting the number of features), because it allows an interesting observation: We can see that all subjects switched between the annotation styles in this task. As the experiment went on, subjects tend to stick with the *color version*. Hence, we have to reject our research hypothesis.

6.3 Interpretation and Threats to Validity

The results contradict our hypothesis. Based on the result of the first experiment and on the comments of some subjects that the background color in M4 was disturbing, we assumed that subjects would switch to *ifdef directives* when working on maintenance tasks, especially M4, in which the entire class was annotated with red background color. However, most subjects used the *color version*.

We believe that most subjects did not even notice the disturbing background color. When we observed our subjects during the experiment, we found that some of them, currently working with the *color version*, moved close to the screen and stared at source code with red background color. Hence, we could observe that subjects behaved like the background color was disturbing, but did not notice this consciously; they did not think of switching to ifdefs. We could have made our subjects aware of the unpleasant background color. However, this would have biased our results, because our objective was to evaluate whether and how subjects used the opportunity to switch between *ifdef directives* and colors.

This leads us to the conclusion that subjects did not necessarily recognize the disturbing effect of the background color. As a consequence, they were slowed down, such that they were as fast as the subjects of our first experiment who also had the *color version* (Mann–Whitney–U test revealed no significant differences between subjects of this experiment and the color group of the first experiment). This result illustrates the importance of choosing suitable background colors, because developers may not always be aware that their screen arrangement is unsuitable. Furthermore, since we did not tell our subjects when to use *ifdef directive* and when to use background colors (we only showed them *how* they could switch), our result indicates that developers need to be trained in using a tool that uses background colors to highlight source code. We come back to the discussion of how to design proper tool support in Section 9.

The same threats to validity as for the first experiment occur here (except for the ones caused by the deviations of the first experiment).

7 Experiment 3: Do Colors Scale?

A question that immediately arose, even before the first experiment, is whether background-color usage scales to large software systems. Obvious objections are that in real-world SPLs with several hundred of features, there would be considerably more colors than a developer can distinguish and that the nesting depth of *ifdef directives* would be too high to be visualized by blending colors. Hence, in a third experiment, we concentrate on the scalability issue. In a nutshell, we could confirm the results of our first experiment for a large SPL with over 99,000 lines of code and 346 features implemented in C, in that we could show an improvement of program comprehension for locating feature code when using background colors. In this section, we present the details of this experiment.

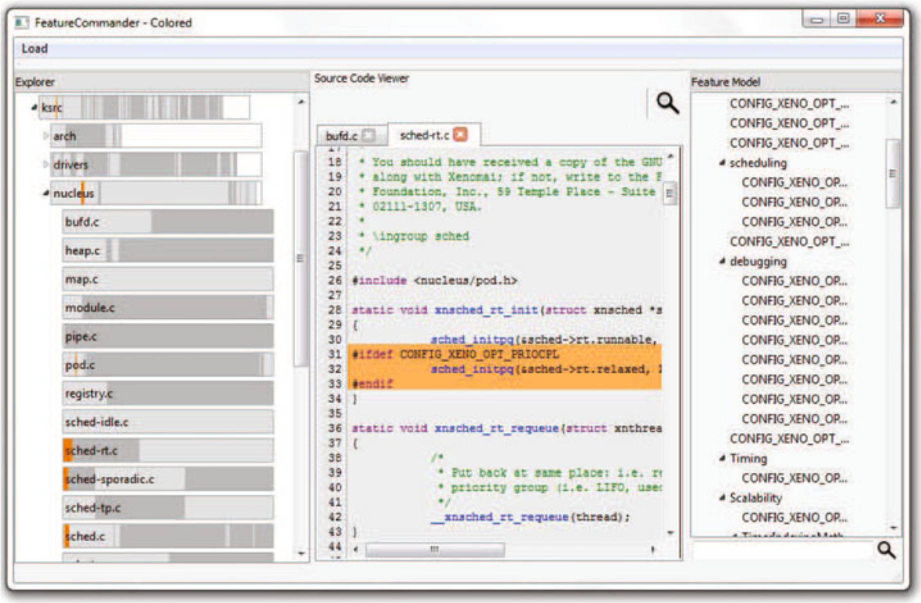
7.1 Experiment Planning

7.1.1 Objective

In this experiment, we evaluate whether background colors improve comprehensibility in large SPLs. To evaluate this issue, we have to understand human limitations on perception. First, preattentive perception is limited to only few items (e.g., few different colors, Goldstein 2002). When there are too many distinctive items, the perception process is slowed down considerably, because more cognitive resources are required (e.g., to count the number of items). Second, human working memory capacity is limited to about 7 ± 2 items (Miller 1956). When there are more items to be kept in mind, they have to be memorized otherwise (e.g., by writing them down). Third, human ability to distinguish colors without direct comparison (i.e., when they are not shown directly next to each other) is limited to only few colors (Rice 1991).

These limitations make a one-to-one mapping of colors to features not feasible in large SPLs with several hundred of features. Therefore, we suggest an as-needed mapping, such that only a limited subset of colors is used at any time, which facilitates human perception. Our as-needed mapping is based on previous investigations of occurrences *ifdef directives* in source code. First, for most parts of the source code, only two to three features appear on one screen (Kästner 2010). Second, most bugs can be narrowed down to certain features or feature combinations (Kästner 2010). Hence, a developer can focus on few features most of the time, such that she avoids limitations to her perception.

Thus, we propose a customizable as-needed mapping, which we show in Fig. 7 (we present an extension of this tool in Fig. 11). We provide a default setting, in which two shades of gray are assigned to features. Code of features located nearby in the



```
 * You should have received a copy of the GNU
 * along with Xenomai; if not, write to the F
 * Foundation, Inc., 59 Temple Place - Suite
 * 02111-1307, USA.
 *
 * \ingroup sched
 */

#include <nucleus/pod.h>

static void xnsched_rt_init(struct xnsched *s
{
        sched_initpq(&sched->rt.runnable,
#ifdef CONFIG_XENO_OPT_PRIOCPL
        sched_initpq(&sched->rt.relaxed, 
#endif
}

static void xnsched_rt_requeue(struct xnthrea
{
        /*
         * Put back at same place: i.e. r
         * priority group (i.e. LIFO, use
         */
        __xnsched_rt_requeue(thread);
}
```

Fig. 7 Experiment 3: Screenshot of tool infrastructure of the color version

source-code file has a different shade of gray, such that a developer can distinguish them, but not recognize the features. Additionally, a developer can assign colors to features she is currently working with. Since she is working only with a few features at a time, her perception limits are not exceeded. Hence, our research hypotheses is:

RH6: Background colors improve program comprehension in large SPLs.

Large means, that the source code consists 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.

Regarding the opinion of our subjects, we assume that they like background-color usage in large software projects, because they were positively perceived in our first and second experiment. Hence, our last research hypothesis is

RH7: Subjects prefer background colors over *ifdef directives* in large SPLs.

7.1.2 Experimental Material

To evaluate our hypotheses, we replace our medium-sized SPL MobileMedia (5,000 lines of code, 4 features) by Xenomai,[9] a large real-time extension for Linux implemented in C. It consists of 99,010 lines of code including 24,709 lines of feature code and 346 different features. Xenomai can be configured for different platforms and provides numerous features, such as real-time communication and scheduling.

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

Table 2 Overview of complexity of different systems

System	LOC	NOFC	LOF (%)	AND	SD	TD	ND
Apache	212 159	1 113	44 426 (20.9)	1.17	5.57	1.74	5
FreeBSD	5 902 461	16 135	841 360 (14.3)	1.13	10.48	2.51	24
Linux	5 985 066	9 093	642 304 (10.7)	1.09	4.66	1.68	6
Solaris	8 238 178	10 290	1 630 809 (19.8)	1.12	16.17	2.72	8
SQLite	94 463	273	48 845 (51.7)	1.29	7.59	1.67	5
Sylpheed	99 786	150	13 607 (13.6)	1.06	6.31	1.38	6
Xenomai	99 010	346	24 709 (25.0)	1.21	6.07	1.44	5

LOC: Lines of code; NOFC; Number of features; LOF: Lines of feature code; AND: Average nesting depth; ND: maximum nesting depth; SD: Occurrences of features in different ifdef expressions; TD: tangling degree of expressions in ifdef directive

There are a number of projects using Xenomai for real-time behavior, for example *RT-FireWire*,[10] *USB for Real-Time*,[11] and *SCALE-RT Real-time Simulation Software*.[12]

To ensure the comparability of Xenomai with other real-world systems, we compared it with Apache, FreeBSD, Linux, Solaris, SQLite, and Sylpheed. To this end, we used cppstats,[13] which computes several metrics to analyze the complexity of *ifdef directives*. In Table 2, we give an overview of the metrics (Liebig et al. 2010). We can see that the systems have different sizes (LOC) and different number of features (NOFC), some in the same range (e.g., SQLite), some larger (e.g., Linux) than Xenomai. Regarding the usage of *ifdef directives*, Xenomai has the second highest percentage of annotated code (LOF): A fourth of the code is annotated. It has a comparable average nesting depth (AND). The scattering degree (SD) indicates how often a feature occurs in different ifdef expressions, whereas the tangling degree (TD) indicates the number of different features in an ifdef expression. In both metrics, Xenomai shows similar values as Apache, Linux, SQLite, and Sylpheed. The same counts for the maximum nesting depth (ND).

We did not base this experiment on Java as the other experiments, because it was rather difficult to find a large-scale SPL implemented in Java. The largest we are aware of is ArgoUML, which consists of more than 100,000 lines of code, but has only 8 features (Couto et al. 2011). We could have developed our own SPL in Java, but this would have been very time consuming and could have easily lead to a biased program (in that we design the SPL such that it confirms our hypotheses). Since there are numerous SPLs implemented in C (Liebig et al. 2010), we decided to use an existing large-scale SPL, even though it was in a different language.

To present the source code to our subjects, we implemented our own tool infrastructure including a source-code viewer using standard syntax highlighting and background colors. In Fig. 7, we show a screenshot to give a better impression. We provided a file-browsing component, a list of all features as tree structure derived from Xenomai's build system, and a menu to load predefined color assignments. The

[10]http://rtfirewire.dynamized.com

[11]http://developer.berlios.de/projects/usb4rt

[12]http://www.linux-real-time.com

[13]http://fosd.de/cppstats

file-browsing component had horizontal bars for each folder and file, which indicates whether and how much feature code a folder or file contains.

In this SPL, *else* and *elif directives* occurred.[14] We decided to assign the same color to each *else* and *elif directive* as to the according *ifdef directive* for two reasons. First, the code is still relevant for the same feature, because the selection of a feature has an effect on all accordingly annotated code fragments. This way, we can visualize that the same feature influences the annotated code fragments. Second, we did not want to introduce more colors than necessary because of the limits of human perception. Annotating each *else* and *elif directive* in a different color would exceed the limit of human perception faster. In Section 9, we present additional concepts to visualize nested *ifdef directives* as well as *else* and *elif directives*, which we did not evaluate in this experiment.

To ensure an optimal color selection for each task and to prevent subjects from having to search their own preferred color assignment, we defined a set of colors for each task. We ensured an optimal color selection by having consistent color assignments across tasks (i.e., a feature that occurred in more tasks has the same or similar color in all tasks) and by having colors that subjects can distinguish within a task without direct comparison (Rice 1991). We chose more transparent colors than in the first two experiments and additionally allowed subjects to adjust the intensity of background colors with a slider. In this experiment, we displayed the *ifdef directives* in the *color version* (instead of removing them as in the first experiment), because in the previous experiments, we showed a benefit of pure background colors. Furthermore, to scale background-color usage to large systems, we do not have a one-to-one mapping of colors to features, so we need the textual information to tell to which feature a colored code fragment belongs. Additionally, we do not blend colors of nested *ifdef directives*, because we did not want to introduce more colors than necessary. Instead, we always display the color of the innermost feature and use vertical bars next to the source-code editor to visualize the nesting of *ifdef directives*.

In addition to the *color version*, we designed another version, in which we removed everything associated to colors (*ifdef version*). Since the source code was large, we provided search functionalities for both versions.

In a second window, we presented the tasks to subjects and provided text fields for their answers. Furthermore, to support subjects in keeping track of time and preventing them from getting stuck on a task, a pop up appeared every 15 min to notify subjects about the time that had passed.

As in the previous experiments, we gave subjects paper-based questionnaires to collect their opinion (i.e., estimation of difficulty, motivation, and performance with the other version, cf. Section 5.1.2).

7.1.3 Subjects

Our sample consisted of 9 master's and 5 PhD students from the University of Magdeburg, Germany. The master's students were participants of the 2010 course *Embedded Networks*, in which they completed several assignments regarding operating systems and networks, such as the implementation of clock synchronization of different computers. They were offered to omit one implementation assignment as

[14] Code of an *else directive* is selected when code of an according *ifdef directive* is not selected.

reward for participating in the experiment. The PhD students were experienced in the operating and embedded-systems domain and invited via e-mail. They participated without reward.

We measured programming experience with the questionnaire described in Section 5.1.3. All subjects were male; none was color blind. As in the first experiment, we created two comparable groups regarding programming experience according to the value of the questionnaire. Additionally, we matched both groups according to the familiarity with Xenomai, because some subjects had some experience with the source code of Xenomai.

7.1.4 Tasks

To measure program comprehension, we designed a number of tasks. We focused on static tasks, because we found in our first experiment a benefit of background colors for static tasks, but not for maintenance tasks. However, we included a few maintenance tasks to control whether our findings still hold.

Altogether, we had 10 tasks: 2 warming-up tasks (W1, W2; not included in the analysis), 6 static tasks (S1–S6), and 2 maintenance tasks (M1, M2). We had three different types of static tasks, two tasks per type:

1: Identifying all files in which a certain feature was implemented (S1, S4).
2: Locating nested *ifdef directives*, which is important for reasoning about feature interactions, cf. Section 5.1.4 (S2, S5).
3: Identifying all features that occur in a certain file (S3, S6).

As example, we present the first static task (S1):

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

For maintenance tasks, we proceeded as for the first experiment. That is, we introduced bugs into the source code and gave subjects a typical bug description that included the feature selections in which the bug occurred. We consulted an expert in C and Xenomai to make sure that the bugs were typical for C programs. As example, we present the first maintenance task:

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.

In the code, we omitted the check whether a variable was null. Instead of `if (ckfn && (err = ckfn(block)) != 0)`, the code said `if ((err = ckfn(block)) != 0)`. If that variable would be accessed when it is null, a segmentation fault would be thrown.

7.1.5 Design

Since our sample was rather small, we used a within-subjects design with two phases (i.e., we let each subject complete tasks with both tool versions). Group A worked with the *color version* in the first phase and switched to the *ifdef version* in the second phase, whereas group B worked with the *ifdef version* in the first phase and switched to the *color version* in the second phase. In each phase, both groups worked with the same tasks in the following order: W1, S1, S2, S3, M1 in the first phase, and W2, S4, S5, S6, M2 in the second phase.[15] Hence, group A solved tasks W1, S1, S2, S3, and M1 with the *color version* and W2, S4, S5, S6, and M2 with the *ifdef version* (vice versa for group B). Corresponding tasks of both phases (i.e., W1/W2, S1/S4, S2/S5, S3/S6, M1/M2) were designed to be comparable (e.g., the same number of features had to be entered as solution). This allows us to compare the results within phases (between groups), and between phases (within groups).

7.1.6 Conduct

The experiment took place in June 2010 instead of a regular lecture session in a room with sufficient working stations (Windows XP) with 17" TFT displays. We gave an introduction, in which we explained the procedure of the experiment and how to use the tool. After the introduction, subjects started to work on their own. When a subject finished the last task of a phase, we gave him the usual paper-based questionnaire to assess his opinion. Three experimenters checked that subjects worked as planned. No deviations occurred.

7.2 Analysis

7.2.1 Descriptive Statistics

Like we did in the first experiment, we examined response times and correctness of tasks. In Fig. 8, we show the response times of our subjects. We can see that for the first two static tasks (S1 and S2), group A (*color version*) is faster than group B: In S1, group A needed only 3 min, compared to 6.6 min of group B (speed up by 55%). In S2, group A needed 5.3 min, and group B 10.3 min (speed up by 49%). Furthermore, maintenance tasks needed considerable more time (note the different scale in the lower part of Fig. 8).

In Fig. 9, we show the correctness of answers. We omitted maintenance tasks in Fig. 9, because we could not regard any of the answers as correct, although most subjects narrowed the problem down to the correct file and method. We discuss this issue in Section 7.4. In S1, the difference is the largest, such that subjects of group B (without colors) performed better than subjects of group A.

In Fig. 10, we present the opinion of subjects, which we asked after each phase. In the first phase, subjects of group A thought they would have performed worse with the *ifdef version* (medians for each task range from 2 to 3), whereas subjects of group B thought they would have performed better with the *color version* (medians for each task vary from 3 to 5). In the second phase, this estimation was reversed for each group in consistence with our expectation, such that subjects of group A thought they would have performed better with the *color version* (medians of 4 in each task), and vice versa for group B (medians of 2 in each task). For difficulty, we see that 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 remaining tasks, the median differs by 1. Regarding motivation,

[15]The tasks are available at the project's website.

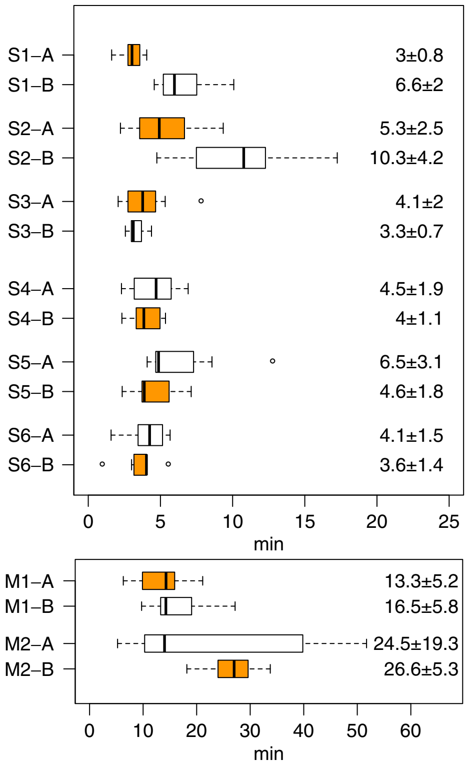


Fig. 8 Experiment 3: response time of subjects in minutes. *Highlighted boxes* indicate groups that worked with the *color version*

we can see that subjects rated their motivation more heterogeneously. The median shows at least a mediocre level of motivation. For the first maintenance task (M1), the motivation for group A (with colors) was very high, compared to group B with a mediocre motivation.

In addition, we asked what version subjects prefer: 12 subjects 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.

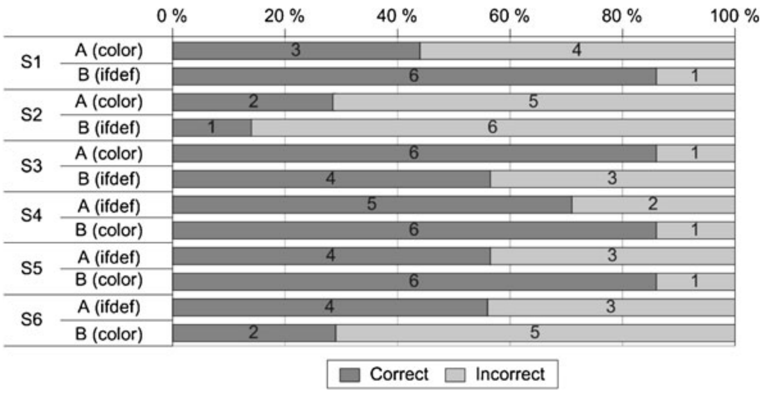


Fig. 9 Experiment 3: frequencies of correct solutions

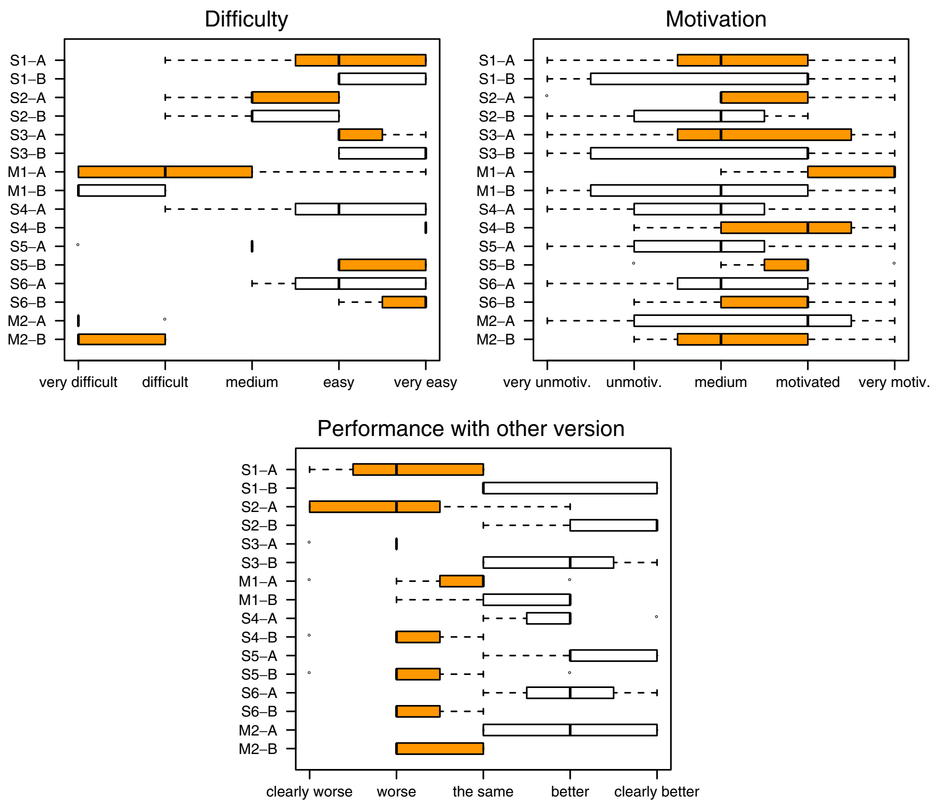


Fig. 10 Experiment 3: box plots of subjects' opinion

7.2.2 Hypotheses Testing

To evaluate our research hypotheses, we proceed as for the first experiment. We start with comparing the response times of subjects in static maintenance tasks (**RH6**), for which we make several comparisons: between groups, which means group A vs. group B, as well as within groups, which means group A (first phase) vs. group A (second phase) and group B (first phase) vs. group B (second phase). Since we make 3 comparisons on the same data, we need to adjust the significance level, for example, with a Bonferoni correction (Anderson and Finn 1996). In our case, we have to divide the significance level by three (because of 3 comparisons), which leads to a significance level of 0.017 to observe a significant difference (instead of 0.05).

We start with group A vs. group B. We applied t tests for independent samples, since the response times are normally distributed (Anderson and Finn 1996). In this experiment, we included incorrect answers, because our sample was too small to delete them. We discuss this in Section 7.4.1. 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. In the second phase, we did not observe a benefit of background colors for program comprehension. As effect size, we computed Cohen's d, because the t test is a parametric test (Cohen 1969). For S1, the value is -2.29 and for S2, the value is -1.46, both indicating a large effect.

Next, we compare the response times of corresponding tasks between both phases (within groups), that is, S1 vs. S4, S2 vs. S5, S3 vs. S6, and M1 vs. M2. For group A, we did not find any significant differences. However, for group B, we observed a significant speed up for S4 (compared to S1; p value: 0.007) and S5 (compared to S2; p value: 0.011). Hence, when adding background colors, the performance of according subjects increased for two tasks. The effect size for both tasks indicates a large effect (S1/S4: 1.56, S2/S5: 1.79). On the other hand, removing background colors does not seem to affect performance, because subjects of group A were not significantly slower in the second phase. Hence, the results regarding response time speak both in favor of and against our research hypothesis.

Regarding correctness of answers, we conduct a χ^2 test. To meet its requirements[16] despite our small sample size, we summarize the number of correct and incorrect answers for the static tasks of each phase. Hence, we compare the frequencies 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 (p values: 1.000 and 0.747, respectively). Since for maintenance tasks none of the subjects provided a correct solution, we do not need to test for significant differences here.

Finally, we compare the opinion of subjects (**RH7**). A Mann–Whitney–U test reveals that the difference regarding estimation of performance with the other version is significant for all tasks but M1, the first maintenance task.[17] 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 only for the first maintenance tasks, such that subjects of group A were more motivated to solve this task compared to group B.

7.3 Interpretation

RH6 *Background Colors Improve Program Comprehension in Large SPLs* Our data can be interpreted both in favor of and against this hypothesis. When comparing the response times between groups, we observed significant differences only in the first phase for two static tasks, such that subjects working with the *color version* were up to 55% faster. In the second phase, we did not observe any significant differences between groups. However, we observed that when we add colors in the second phase, the comprehension process of according subjects (group B) got faster by up to 55%, which indicates a large effect according to Cohen's d. For maintenance tasks, we did not observe a significant difference in response time. Hence, we found that background colors improve program comprehension in preprocessor-based SPLs in two static tasks.

For the third kind of static tasks (i.e., locating all features in a file), we did not observe significant differences. A possible reason is that in these tasks, the number

[16]Expected frequencies for single tasks are too small due to the small number of observations.

[17]We cannot provide p values in this case, because due to our small sample, we had to look up whether observed differences are significant in a table of the U distribution (Giventer 2008).

of relevant features was 12, which means that subjects had to work with 12 different colors. Although we selected colors to be clearly distinguishable without direct comparison, 12 might be too much and exceed the limits of human perception (cf. Section 7.1.1). Additionally, the working memory capacity of 7 ± 2 is exceeded with 12 features. For the other tasks, only 1 (S1, S4) or 2 (S2, S5) features had to be kept in mind. However, since we only combined 12 features with the third kind of static tasks, we can only theorize why this result occurred.

Furthermore, none of our subjects solved a maintenance task correctly. The most likely explanation is that these tasks were too difficult given the short time of the experiment. We discuss this problem in more detail in Section 7.4.

To sum up, background colors can help to familiarize with a large SPL, especially to get an overview of the files of a feature or of nested *ifdef directives*. When we add background colors in the second phase, the performance of according subjects increases. When we remove colors, it has no effect on the performance of according subjects. Our observations align with the results of the first experiment that background colors can improve program comprehension in static tasks.

RH7 *Subjects Prefer Background Colors Over* ifdef directives *in Large SPLs* We can accept this research hypothesis, because we found a preference for background colors. Subjects who worked with the *color version* estimate they would perform worse without colors, even when we observed no difference in performance. We found the same effect in our first experiment. Additionally, all subjects rate colors as more suitable when working with preprocessor-based SPLs, and all but one subject preferred colors over no colors (except one subject who answered neither of both questions). This is also in line with the first experiment, in which background colors were rated positively.

Hence, in large SPLs, background colors have a potentially positive impact on program comprehension in preprocessor-based SPLs in terms of locating feature code. This means that we can circumvent human limitations regarding (preattentive) color perception and working memory capacity. Instead of a one-to-one mapping, we used an as-needed mapping based on observations about the occurrences of *ifdef directives* in source code, which scales to large SPLs with over 300 features.

7.4 Threats to Validity

7.4.1 Internal Validity

One problem is that we could not rate any solution for maintenance task as correct. However, subjects often named the correct file and method, which indicates that if subjects had more time, they might have succeeded eventually. We believe that the realistic nature of the maintenance task (ensured by an expert on C and Xenomai) was too difficult for the time constraint and subjects' expertise, despite pre-tests. Furthermore, our primary focus were static tasks.

Another threat is caused by our small sample. To deal with it, we used a within-subjects design and applied variants of standard significant tests that were developed for small sample sizes.

Additionally, we did not correct the response times for wrong answers. The size of our sample does not allow us to omit wrong answers. Another possibility is to compute an efficiency measure as combination of correctness of answers and response time (e.g., Otero and Dolado 2004). However, it is not clear whether the use of such a measure may lead to falsely accept or reject a hypothesis, because there are several ways to define a measure. To the best of our knowledge, there is no agreed and evaluated efficiency measure. Since we found no indication in our data that subjects entered a wrong answer deliberately (i.e., wrong answers often missed only one or two features and the response times did not deviate very much toward zero from the mean), we argue that results regarding response time and correctness are still valid.

7.4.2 External Validity

Our sample consisted mostly of master's students who were rather unexperienced with large SPLs. However, we also included some PhD students who had several years of programming experience in the domain of operating and embedded systems. Hence, our results can carefully be interpreted for experienced programmers, as well.

8 Summary of the Experiments

All three experiments analyzed how background colors influence the readability of preprocessor-based SPLs. The focus of the first experiment was on program comprehension in small preprocessor-based SPLs, the focus of the second experiment on the behavior of subjects using medium-sized preprocessor-based SPLs, and the focus of the third experiment was on program comprehension in large preprocessor-based SPLs. In Table 3, we summarize the results of our three experiments to give a better overview.

Interpreting the results of all three experiments together yields the following conclusions:

1. Carefully chosen colors improve program comprehension in preprocessor-based SPLs in terms of locating feature code, independently of size and programming language of the underlying SPL.
2. Colors with a high saturation can slow down the comprehension process in terms of bug fixing.
3. Subjects like and prefer the color idea.

First, we could show that carefully chosen background colors lead to a performance increase of subjects for static tasks. This generalizes to medium-sized and large SPLs. Additionally, we observed a performance speed up with both Java and C.

Table 3 Summary of main findings for all three experiments

Experiment	Source code	LOC	Features	Result
1	MobileMedia	5 000	4	Colors speed up static tasks; no effect or slow down for maintenance tasks
2	MobileMedia	5 000	4	Subjects are unaware of the potentially negative effect of colors
3	Xenomai	99 010	346	The positive effects found in experiment 1 scale for large SPLs

Although we showed the positive effect only for two different sizes and two different programming languages, we expect similar positive effects also with medium-sized SPLs and other programming languages.

Second, we found that highly saturated background colors can slow down the comprehension process when subjects are fixing bugs. We believe that visual fatigue causes this slow down. However, when given the choice, subjects do not seem to be aware that a background color is disturbing and slowing them down. Nevertheless, for locating feature code, we did only find positive (or no) effects of background colors. Hence, depending on the task, the saturation of colors may play an important role. Thus, we suggest that source-code editors using background colors provide the option to adjust the saturation of background colors.

Third, the majority of our subjects favored background colors. This is encouraging, because a new concept that is not accepted by the ones who are supposed to use it will hardly be used in practice. Hence, the acceptance of background colors is an important positive result.

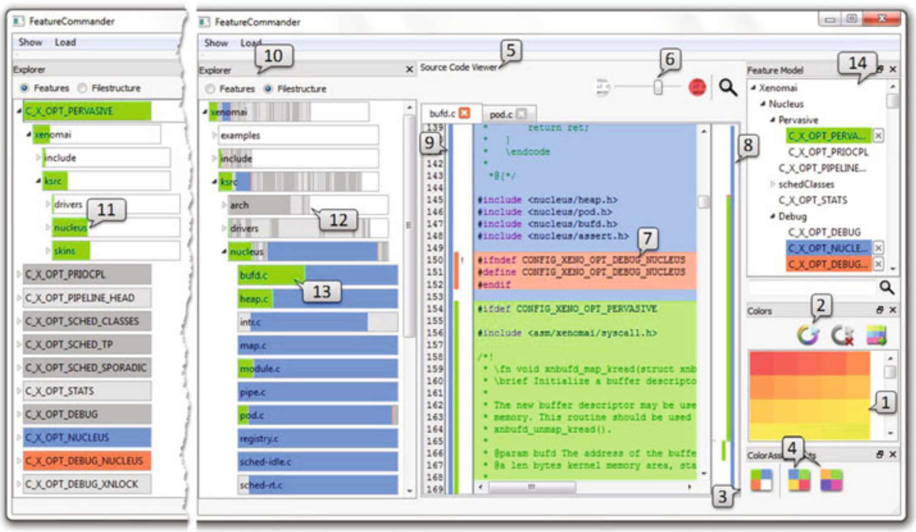
However, we have to be careful with our conclusions. We cannot state that background colors are always helpful in every situation in which preprocessors are used to implement variability. Instead, we have to keep in mind the context of our experiments. We used mostly students for our experiments with considerably less experience than experts having spent years and years on developing and maintaining preprocessor-based SPLs. Furthermore, we only used two different SPLs. Our results only apply to similar SPLs, although we have evidence that many open-source systems, such as FreeBSD, Linux, Solaris, SQLite, and Sylpheed have similar characteristics. If the nature of an SPL is different, we can only theorize how background colors affect the comprehension of preprocessor directives. Hence, future experiments with different experimental contexts are necessary to build a more complete understanding of the effect of background colors on program comprehension in preprocessor-based SPLs.

To sum up, all results encourage us to use background colors more often in source-code editors. Hence, we developed a prototype FeatureCommander, which we present next.

9 Toward Better Tool Support

Our experiments were based on a relatively simple concept of background colors, much like our browser setting in the first experiment (cf. Section 5.1.2). Specifically, we based our work on CIDE (Kästner et al 2008), a tool that uses background colors to visualize feature code. With our experiments, we gained useful insights into tool requirements for preprocessor-based product-line development. Based on these insights, including comments and suggestions from subjects about functionalities, and by consulting other similar tools (e.g., Spotlight) and literature on software visualization (e.g., Diehl 2007), we implemented a tool called *FeatureCommander*.[18]

[18]http://fosd.de/fc. On the website, there is also a video demonstrating the use of FeatureCommander. This video shows *all* functionality of FeatureCommander, not the reduced set we used in the third experiment. FeatureCommander with the reduced set is also available at the website.



```
 *         return ret;
 *    }
 *    \endcode
 *
  *@{*/

#include <nucleus/heap.h>
#include <nucleus/pod.h>
#include <nucleus/bufd.h>
#include <nucleus/assert.h>

#ifndef CONFIG_XENO_OPT_DEBUG_NUCLEUS
#define CONFIG_XENO_OPT_DEBUG_NUCLEUS
#endif

#ifdef CONFIG_XENO_OPT_PERVASIVE

#include <asm/xenomai/syscall.h>

/*!
 * \fn void xnbufd_map_kread(struct xnb
 * \brief Initialize a buffer descripto
 *
 * The new buffer descriptor may be use
 * memory. This routine should be used
 * xnbufd_unmap_kread().
 *
 * @param bufd The address of the buffe
 * @a len bytes kernel memory area, sta
```

Fig. 11 Screenshot of FeatureCommander. The numbers designate concepts we explain in detail in the text

FeatureCommander is a prototype for preprocessor-based SPL development. It offers multiple visualizations that support program comprehension in large preprocessor-based SPLs. The basic characteristic of FeatureCommander is the consistent usage of colors throughout all visualizations. In Fig. 11, we show a screenshot of FeatureCommander displaying source code from our third experiment. We refer to the numbers in Fig. 11 when explaining the according concepts in the next paragraphs.

To assign colors to feature, we provide two different options: First, users can assign colors to features by dragging a color from a color palette (1) and dropping it on a feature in any of the visualizations.[19] For efficiency, users can also automatically assign a palette of colors to multiple features (2). The automatic color assignment chooses colors such that they are as different as possible in the hue value of the HSV color model. Furthermore, color assignments can be saved (3) and loaded (4), so that a developer can easily resume her work. This way, we support an as-needed mapping of colors to features. When no color is assigned to a feature, it is represented by a shade of gray in all visualizations.

Similar to other IDEs (Kästner et al. 2009b; Stengel et al. 2011), we provide different views: *source-code view*, *explorer view*, and *feature-model view*. In the *source-code view* (5), the background color of source-code fragments indicates to which features fragments are related; according *ifdef directives* are also shown. To compromise between code readability and feature recognition, users can adjust the opacity of the background color (6). This way, we address that too highly saturated colors negatively affect program comprehension. If a code fragment is assigned to multiple features, we show only the background color of the innermost feature (7).

[19]To recognize feature code, FeatureCommander uses a file that describes where an *ifdef directive* starts and where it ends.

The other features are visualized in the sidebars on either side of the code view, which visualize features as bars, ordered by the nesting hierarchy (8, 9). The right sidebar gives an overview of the whole document (8), the left sidebar shows the hierarchy of features of the currently displayed source code (9). Both sidebars are interactive, such that clicking them immediately shows the according code fragment. We implemented both sidebars, because it further supports a user in locating a code fragment (although we did not evaluate the impact on program comprehension).

In the *explorer view* (10), users can navigate the file structure and open files. We provide two tree representations of the project: One ordered according to the file structure, the other ordered by features (11). In the background of each tree node, we display the percentage of each feature that occurs in the represented file or folder, either with default alternating shades of gray (12) or with the assigned color (13). Thus, users get an overview of the distribution of features without having to open files.

In the *feature-model view* (14), the feature model is shown as a simple tree. Features that are currently not of interest can be collapsed.

With FeatureCommander, we give researchers a tool that implements several concepts that improve program comprehension in large-scale preprocessor-based SPLs. Hence, we created a base, from which further research on concepts and their effect on program comprehension can emerge. Additionally, we provide practitioners with a tool that has shown that it can improve program comprehension in preprocessor-based SPLs in terms of locating feature code, and, consequently, can reduce the cost of software maintenance.

10 Previous and Related Work

10.1 Previous Work

In prior work, we described the first experiment (Feigenspan 2009; Feigenspan et al. 2009). It was conducted as part of the first author's master's thesis (Feigenspan 2009). We subsequently briefly summarized the results in a workshop paper, motivating the necessity of empirical research and explaining the path we took toward this paper (Feigenspan et al. 2009). The primary intent was not to present the experiment nor its results, but to analyze the feasibility of program-comprehension experiments in the context of feature-oriented software development (Apel and Kästner 2009). The present work is the first time we present the experiment as well as its results and their implications in detail.

We published our third experiment with focus on our tool FeatureCommander and the implementation of concepts to support a developer in her comprehension process (Feigenspan et al. 2011b). We also published a tool demo of FeatureCommander, in which we only focus on the tool, not on the evaluation (Feigenspan et al. 2011a).

Our first experiment was based on the background-color concept implemented in *CIDE* (Feigenspan et al. 2010; Kästner et al 2008). CIDE was developed to support a programmer in decomposing legacy applications into features. Besides background colors, it provides code folding of feature code (i.e., it hides source code of selected features) and different views on source code. Furthermore, it enforces disciplined annotations, leading always to syntactically correct code when feature code is removed to create a variant. For example, an opening bracket can only be annotated with a corresponding closing one. Furthermore, we provide *FeatureIDE* (Kästner et al. 2009b), a tool that also supports the development of SPLs. In contrast to FeatureCommander, FeatureIDE supports more SPL-implementation techniques in addition to preprocessor directives and aims at the complete development process of SPLs (i.e., from the design phase to the maintenance phase).

Another tool of our's is *View Infinity*, which also aims at improving program comprehension in preprocessor-based SPLs (Stengel et al. 2011). In addition to background colors, ViewInfinity provides semantic zooming from the feature-model level over the file structure to the source-code level. An empirical evaluation focused on how experienced SPL developers use and like the semantic-zooming concept, not on background-color usage on source-code level.

In a parallel line of work independent of colors, visualization, and tooling, we explored the discipline and granularity of *ifdef directives* in 40 medium- to large-scale software systems (Liebig et al. 2010, 2011). We found that developers use *ifdef directives* to a large extent, and that most directives are disciplined and occur in the source code at a fine grain. This supports our argument that *ifdef directives* can cause problems and that their readability can be improved with background colors.

10.2 Related Work

In literature, the C preprocessor is often heavily criticized. Numerous studies discuss the negative effect of preprocessor usage on code quality and maintainability (e.g., Adams et al. 2008; Ernst et al. 2002; Favre 1995, 1997; Krone and Snelting 1994; Spencer and Collyer 1992). However, researchers have also explored different strategies to deal with these problems.

One group of approaches attempts to extract structures from the source code (e.g., nesting, dependencies, and include hierarchies) and visualize them in a separate view (Krone and Snelting 1994; Pearse and Oman 1997; Spencer and Collyer 1992). We follow this line of work and use similarly simple structures, but we focus on supporting developers directly in working with the annotated source code and integrate a visual representation of annotations with the underlying source code.

The model editors *fmp2rsm* (Czarnecki and Antkiewicz 2005) and *FeatureMapper* (Heidenreich et al. 2008) allow a user to annotate model elements to generate different model variants. Both tools can represent annotations with colors. The tool *Spotlight* (Coppit et al. 2007) uses vertical bars in the source-code editor to represent annotations, which are more subtle than background colors. Spotlight aims at improving the traceability of scattered concerns, which are represented by different colors. *SeeSoft* (Eick et al. 1992) represents files as rectangles and source-code lines as colored pixel lines. The color is an indicator of the age of the according source-code line. In contrast to our work, the influence of visualizations of these tools on program comprehension has not been assessed empirically.

In addition to visualizations, also views on configurations have been explored, which show only part of the feature code and hence reduce complexity (Atkins et al. 2002; Chu-Carroll et al. 2003; Hofer et al. 2010; Kästner et al 2008; Singh et al. 2007). A view on a variant or a view on a feature displays only feature code of selected features and hides all remaining code. Some tools even hide annotations completely, such that a developer only works on one variant and may not even be aware of other variants or features (Atkins et al. 2002). In an analysis of the change history of a large telephone switching software system, Atkins et al. showed a productivity increase of 40%, when developers work with views provided by the Version Editor. However, hiding feature code may not always be feasible: For example, when code of a hidden feature shares code with a feature in which a developer fixes a bug, she might introduce bugs into the hidden feature code without knowing it (Ribeiro et al. 2010). In this case, a developer needs the context of the complete SPL to fix a feature-specific bug. Hence, views on source code and background colors *complement* each other for different tasks.

Furthermore, a severe problem for many approaches is precise fact extraction from unpreprocessed C code, especially if one wants to reason not only about the preprocessor directives, but also about their combination of C code. Many researchers have attempted analysis and rewrite systems for unpreprocessed C code (Aversano et al. 2002; Baxter and Mehlich 2001; Garrido and Johnson 2005; Hu et al. 2000; Livadas and Small 1994; Overbey and Johnson 2009; Tartler et al. 2011; Vidacs et al. 2004). For example, Ernst et al. (2002) identify problematic patterns and quantify them in a large code base, Tartler et al. (2011) search for code blocks that are dead in all feature configurations, and Hu et al. (2000) use control flow graphs to analyze the inclusion structure of files. However, all these approaches aim not directly at improving program comprehension by developers, but form underlying mechanisms and can be used to build tools.

Whereas we focused solely on conditional compilation, also lexical macros can pose significant stress on program comprehension. Several researchers have investigated analysis and visualizations strategies that can explain macro expansion and add debugging tasks. For example, Spinellis (2003) provide an automated approach to rewrite macro expansions. Kullbach and Riediger (2001) present *folding* to hide or show preprocessor-annotated code as needed. These mechanisms are complementary to our approach.

To overcome preprocessor understanding problems in general, many researchers recommend to abandon preprocessor usage in favor of more disciplined implementation approaches, such as feature-oriented programming (Prehofer 1997) and aspect-oriented programming (Kiczales et al. 1997), or syntactic preprocessors such as ASTEC (McCloskey and Brewer 2005). Several researchers have even investigated automated refactorings (Adams et al. 2009; Kästner et al. 2009a). But preprocessors are still prevalent in practice and the vast amount of legacy code will not disappear soon. Hence there is still significant need for tools like ours that support developers when forced to deal with legacy code.

Finally, the idea of using colors to support a developer is not new. Early empirical work was published in 1986 (Rambally 1986). In this experiment, Rambally found that annotating source-code fragments with colors according to their functionality improves program comprehension, compared to control-structure color-coding scheme (e.g., loops, and if-then-else statements), and no colors at all. Furthermore, color usage for various tasks is evaluated by several research groups, for example, highlighting source code for error reporting (Oberg and Notkin 1992) or merging (Yang 1994). In 1988, the ANSI/HFS 100-1988 standard[20] was published, which

[20]http://www.hfes.org/web/Standards/standards.html.

included recommendations about the contrast of background colors and foreground colors. Today, syntax highlighting (i.e., coloring of syntactic elements) is an integral part of most IDEs (e.g., Eclipse). However, to the best of our knowledge, the use of background colors in preprocessor-based software has not been evaluated empirically.

11 Conclusion

Preprocessors are frequently used in practice to implement software product lines. However, they introduce threats to program comprehension and are even referred to as “#ifdef hell”. To improve readability of preprocessor-based software, we proposed to use background colors, such that source code annotated with *ifdef directives* is highlighted and can be easily perceived at first sight.

In three controlled experiments, we revealed both benefits and drawbacks of background-color usage with regard to supporting program comprehension in preprocessor-based SPLs. The results clearly showed that background colors have the potential to improve program comprehension. We could show that background colors positively influence program comprehension in terms of locating feature code, independently of size and language of the underlying projects. Additionally, we found in all experiments that subjects favor background colors. This is an important result, because the attitude of developers toward the tool they are working with can significantly affect their performance (Mook 1996), for example, because they may stick longer with a task (and not get frustrated by the tool). This effect is exploited in many tools, which typically have numerous customizing options, so that users can adjust the tool according to their preferences.

However, we also found that colors have to be chosen with great care. Otherwise, they could slow developers down. Our results indicate that bright, saturated colors, such as we used in our first setting, are distracting and cause visual fatigue. Consequently, developers need more time when working with colors, for example, because of a need to rest the eyes. Hence, developers should be able to customize color settings according to their needs (e.g., like we provided in the second and third experiment). For example, when a developer has located a code fragment that she suspects to cause a problem, she can turn colors off or adjust the saturation to a low degree.

Based on the results of our experiments, we implemented the prototype *FeatureCommander*, in which we realized scalable background-color usage. Developers can efficiently adjust color settings to their needs, for example by adjusting opacity. Thus, customizable background-color concepts as implemented in FeatureCommander can increase the efficiency of maintenance developers and reduce the cost of software development.

Acknowledgements We are grateful to Mathias Frisch for helpful discussions and to Jana Schumann, Veit Köppen, and Thomas Thüm for their support with the experiments. Also thanks to all the reviewers of this article. Feigenspan’s and Saake’s work is supported by BMBF project ViERforES (01IM10002B), Kästner’s work partly by ERC (#203099), and Apel’s work is supported by the German Research Foundation (DFG—AP 206/2, AP 206/4, and LE 912/13). Schulze’s work is part of the project SAFE. SAFE is funded by BMBF (project ID: 01IS11019) and DGCIS and is in the framework of the ITEA2, EUREKA cluster program Σ! 3674. Dachselt’s work is funded by the

"Stifterverband für die Deutsche Wissenschaft" from funds of the Claussen-Simon-Endowment. The first experiment was conducted as part of Feigenspan's master's thesis while she visited the Metop Research Center.

References

Adams B et al (2008) Aspect mining in the presence of the C preprocessor. In: Proc. AOSD workshop on linking aspect technology and evolution. ACM Press, pp 1–6

Adams B et al (2009) Can we refactor conditional compilation into aspects? In: Proc. Int'l conf. aspect-oriented software development (AOSD). ACM Press, pp 243–254

Anderson T, Finn J (1996) The new statistical analysis of data. Springer

Apel S, Kästner C (2009) An overview of feature-oriented software development. J Obj Techn 8(4):1–36

Apel S et al (2008) Aspectual feature modules. IEEE Trans Softw Eng 34(2):162–180

Atkins D et al (2002) 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

Aversano L et al (2002) Handling preprocessor-conditioned declarations. In: Proc. IEEE int'l workshop on source code analysis and manipulation. IEEE CS, pp 83–92

Basili VR (1992) Software modeling and measurement: the goal/question/metric paradigm. Tech. Rep. CS-TR-2956 (UMIACS-TR-92-96)

Baxter ID, Mehlich M (2001) Preprocessor conditional removal by simple partial evaluation. In: Proc. working conf. reverse engineering (WCRE). IEEE CS, pp 281–290

Boehm B (1981) Software engineering economics. Prentice Hall

Boysen J (1977) Factors affecting computer program comprehension. PhD thesis, Iowa State University

Brooks R (1978) Using a behavioral theory of program comprehension in software engineering. In: Proc. int'l conf. software engineering (ICSE). IEEE CS, pp 196–201

Chevalier F et al (2010) Using text animated transitions to support navigation in document histories. In: Proc. conf. human factors in computing systems (CHI). ACM Press, pp 683–692

Chu-Carroll M et al (2003) Visual separation of concerns through multidimensional program storage. In: Proc. int'l conf. aspect-oriented software development (AOSD). ACM Press, pp 188–197

Clements P, Northrop L (2001) Software product lines: practice and patterns. Addison-Wesley, Reading, MA

Cliff N (1993) Dominance statistics: ordinal analyses to answer ordinal questions. Psychol Bull 114(3):494–509

Cohen J (1969) Statistical power analysis for the behavioral sciences. Academic Press

Coppit D et al (2007) Spotlight: a prototype tool for software plans. In: Proc. int'l conf. software engineering (ICSE). IEEE CS, pp 754–757

Couto MV et al (2011) Extracting software product lines: a case study using conditional compilation. In: Proc. Europ. conf. software maintenance and reengineering (CSMR), pp 191–200

Czarnecki K, Antkiewicz M (2005) Mapping features to models: a template approach based on superimposed variants. In: Proc. int'l conf. generative programming and component engineering (GPCE). Springer, pp 422–437

Daly J et al (1995) The effect of inheritance on the maintainability of object-oriented software: an empirical study. In: Proc. int'l conf. software maintenance (ICSM). IEEE CS, pp 20–29

Diehl S (2007) Software visualization: visualizing the structure, behaviour, and evolution of software. Springer

Dunsmore A, Roper M (2000) A comparative evaluation of program comprehension measures. Tech. Rep. EFoCS 35-2000, Department of Computer Science, University of Strathclyde

Eick S et al (1992) SeeSoft—a tool for visualizing line oriented software statistics. IEEE Trans Softw Eng 18(11):957–968

Ernst M et al (2002) An empirical analysis of C preprocessor use. IEEE Trans Softw Eng 28(12):1146–1170

Favre J-M (1995) The CPP paradox. In: Proc. European workshop on software maintenance

Favre J-M (1997) Understanding-in-the-large. In: Proc. int'l workshop on program comprehension (IWPC). IEEE CS, p 29

Feigenspan J (2009) Empirical comparison of FOSD approaches regarding program comprehension—a feasibility study. Master's thesis, University of Magdeburg

Feigenspan J et al (2009) How to compare program comprehension in FOSD empirically - an experience report. In: Proc. int'l workshop on feature-oriented software development (FOSD). ACM Press, pp 55–62

Feigenspan J et al (2010) Visual support for understanding product lines. In: Proc. int'l conf. program comprehension (ICPC). IEEE CS, Demo Paper, pp 34–35

Feigenspan J et al (2011a) FeatureCommander: colorful #ifdef world. In: Software product line conference (SPLC), paper 3. ACM Press, pp 1–2

Feigenspan J et al (2011b) Using background colors to support program comprehension in software product lines. In: Proc. int'l conf. evaluation and assessment in software engineering (EASE). Institution of Engineering and Technology, pp 66–75

Figueiredo E et al (2008) Evolving software product lines with aspects: an empirical study on design stability. In: Proc. int'l conf. software engineering (ICSE). ACM Press, pp 261–270

Fisher D, Tan K (1989) Visual displays: the highlighting paradox. Human Factors 31(1):17–30

Garrido A, Johnson RE (2005) Analyzing multiple configurations of a C program. In: Proc. int'l conf. software maintenance (ICSM). IEEE CS, pp 379–388

Giventer L (2008) Statistical analysis for public administration, 2nd edn. Jones and Bartlett Publishing

Goldstein B (2002) Sensation and perception, 5th edn. Cengage Learning Services

Hanenberg S (2010) An experiment about static and dynamic type sytems. In: Proc. int'l conf. object-oriented programming, systems, languages and applications (OOPSLA). ACM Press, pp 22–35

Harrison W, Ossher H (1993) Subject-oriented programming: a critique of pure objects. In: Proc. int'l conf. object-oriented programming, systems, languages and applications (OOPSLA). IEEE CS, pp 411–428

Heidenreich F et al (2008) FeatureMapper: mapping features to models. In: Comp. int'l conf. software engineering (ICSE). ACM Press, Demo Paper, pp 943–944

Hofer W et al (2010) Toolchain-independent variant management with the Leviathan filesystem. In: Proc. int'l workshop on feature-oriented software development (FOSD). ACM Press, pp 18–24

Hu Y et al (2000) C/C++ conditional compilation analysis using symbolic execution. In: Proc. int'l conf. software maintenance (ICSM). IEEE CS, pp 196–206

Kästner C (2010) Virtual separation of concerns: preprocessors 2.0. PhD thesis, University of Magdeburg

Kästner C et al (2008) Granularity in software product lines. In: Proc. int'l conf. software engineering (ICSE). ACM Press, pp 311–320

Kästner C et al (2009a) A model of refactoring physically and virtually separated features. In: Proc. int'l conf. generative programming and component engineering (GPCE). ACM Press, pp 157–166

Kästner C et al (2009b) FeatureIDE: tool framework for feature-oriented software development. In: Proc. int'l conf. software engineering (ICSE). IEEE CS, Demo Paper, pp 611–614

Kiczales G et al (1997) Aspect-oriented programming. In: Proc. Europ. conf. object-oriented programming (ECOOP). Springer, pp 220–242

Kitchenham B et al (2008) Evaluating guidelines for reporting empirical software engineering studies. Empir Software Eng 13(1):97–121

Koenemann J, Robertson S (1991) Expert problem solving strategies for program comprehension. In: Proc. conf. human factors in computing systems (CHI). ACM Press, pp 125–130

Krone M, Snelting G (1994) On the inference of configuration structures from source code. In: Proc. int'l conf. software engineering (ICSE). IEEE CS, pp 49–57

Kullbach B, Riediger V (2001) Folding: an approach to enable program understanding of preprocessed languages. In: Proc. working conf. reverse engineering (WCRE). IEEE CS, pp 3–12

Levkowitz H, Herman GT (1992) Color scales for image data. IEEE Comput Graph Appl 12(1): 72–80

Liebig J et al (2010) An analysis of the variability in forty preprocessor-based software product lines. In: Proc. int'l conf. software engineering (ICSE). ACM Press, pp 105–114

Liebig J et al (2011) Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In: Proc. int'l conf. aspect-oriented software development (AOSD). ACM Press, pp 191–202

Likert R (1932) A technique for the measurement of attitudes. Arch Psychol 22(140):1–55

Livadas P, Small D (1994) Understanding code containing preprocessor constructs. In: Proc. int'l workshop program comprehension (IWPC). IEEE CS, pp 89–97

Lohmann D et al (2006) A quantitative analysis of aspects in the eCos kernel. In: Proc. Europ. conf. computer systems (EuroSys). ACM Press, pp 191–204

McCloskey B, Brewer E (2005) ASTEC: a new approach to refactoring C. In: Proc. Europ. software engineering conf./foundations of software engineering (ESEC/FSE). ACM Press, pp 21–30

Miller G (1956) The magical number seven, plus or minus two: some limits on our capacity for processing information. Psychol Rev 63(2):81–97

Mook D (1996) Motivation: the organization of action, 2nd edn. W.W. Norton & Co

Muthig D, Patzke T (2003) Generic implementation of product line components. In: Int'l conf. NetObjectDays. Springer, pp 313–329

Oberg B, Notkin D (1992) Error reporting with graduated color. IEEE Softw 9(6):33–38

Otero M, Dolado J (2004) Evaluation of the comprehension of the dynamic modeling in UML. J Inf Softw Technol 46(1):35–53

Overbey JL, Johnson RE (2009) Software language engineering. In: Generating rewritable abstract syntax trees, pp 114–133.

Pearse T, Oman P (1997) Experiences developing and maintaining software in a multi-platform environment. In: Proc. int'l conf. software maintenance (ICSM). IEEE CS, pp 270–277

Pennington N (1987) Stimulus structures and mental representations in expert comprehension of computer programs. Cogn Psychol 19(3):295–341

Pohl K et al (2005) Software product line engineering: foundations, principles, and techniques. Springer

Prechelt L et al (2002) Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. IEEE Trans Softw Eng 28(6):595–606

Prehofer C (1997) Feature-oriented programming: a fresh look at objects. In: Europ. conf. on object-oriented programming (ECOOP). Springer, pp 419–443

Rambally G (1986) The influence of color on program readability and comprehensibility. In: Proc. technical symposium on computer science education (SIGCSE). ACM Press, pp 173–181

Ribeiro M et al (2010) Emergent feature modularization. In: Proceedings of the ACM international conference companion on object oriented programming systems languages and applications companion (SPLASH). ACM Press, pp 11–18

Rice J (1991) Display color coding: 10 rules of thumb. IEEE Softw 8(1):86–88

Riggs R et al (2003) Programming wireless devices with the java 2 platform, micro edition. Sun Microsystems, Inc

Shaft T, Vessey I (1995) The relevance of application domain knowledge: the case of computer program comprehension. Inf Syst Res 6(3):286–299

Shapiro S, Wilk M (1965) An analysis of variance test for normality (complete samples). Biometrika 52(3/4):591–611

Shneiderman B, Mayer R (1979) Syntactic/semantic interactions in programmer behavior: a model and experimental results. Int J Parallel Prog 8(3):219–238

Singh N et al (2006) CViMe: viewing conditionally compiled C/C++ sources through java. In: Companion to the 21st ACM SIGPLAN symposium on object-oriented programming systems, languages, and applications. ACM Press, pp 730–731

Singh N et al (2007) C-CLR: a tool for navigating highly configurable system software. In: Proc. workshop aspects, components, and patterns for infrastr. software. ACM Press

Smaragdakis Y, Batory D (1998) Implementing layered designs with mixin layers. In: Proc. Europ. conf. object-oriented programming (ECOOP). Springer, pp 550–570

Soloway E, Ehrlich K (1984) Empirical studies of programming knowledge. IEEE Trans Softw Eng 10(5):595–609

Someren M et al (1994) The think aloud method: a practical guide to modelling cognitive processes. Academic Press

Spencer H, Collyer G (1992) #ifdef considered harmful or portability experience with C news. In: Proc. USENIX conf. USENIX Association, pp 185–198

Spinellis D (2003) Global analysis and transformations in preprocessed languages. IEEE Trans Softw Eng 29(11):1019–1030

Standish T (1984) An essay on software reuse. IEEE Trans Softw Eng SE–10(5):494–497

Stengel M et al (2011) View infinity: a zoomable interface for feature-oriented software development. In: Proc. int'l conf. software engineering (ICSE). ACM Press, pp 1031–1033

Tamborello F, Byrne M (2007) Adaptive but non-optimal visual search with highlighted displays. Cogn Syst Res 8(3):182–191

Tartler R et al (2011) Feature consistency in compile-time configurable system software. In: Proc. Europ. conf. computer systems conference (EuroSys). ACM Press, pp 47–60
Tiarks R (2011) What programmers really do: an observational study'. In: Proc. workshop software reengineering (WSR), pp 36–37
Tichy WF (1998) Should computer scientists experiment more? Computer 31(5):32–40
Vidacs L et al (2004) Columbus schema for C/C++ preprocessing. In: Proc. Europ. conf. software maintenance and reengineering (CSMR). IEEE CS, pp 75–84
von Mayrhauser A, Vans A (1993) From program comprehension to tool requirements for an industrial environment. In: Proc. int'l workshop program comprehension (IWPC). IEEE CS, pp 78–86
von Mayrhauser A et al (1997) Program understanding behaviour during enhancement of large-scale software. J Softw Maint: Res Pract 9(5):299–327
von Mayrhauser A, Vans M (1995) Program comprehension during software maintenance and evolution. Computer 28(8):44–55
Ware C (2000) Information visualization: perception for design. Morgan Kaufmann
Wijffelaars M et al (2008) Generating color palettes using intuitive parameters. Comput Graph Forum 27(3):743–750
Yang W (1994) How to merge program texts. J Syst Softw 27(2):129–135
Yellott J (1971) Correction for fast guessing and the speed accuracy trade-off in choice reaction time. J Math Psych 8(2):159–199

Janet Feigenspan is a PhD student at the University of Magdeburg, Germany. She received her master's degree in Computer Science in 2009, and her master's degree in Psychology in 2005, both from the University of Magdeburg. For her computer-science master's thesis, she received the award from the Metop Research Institute for best thesis and the award from the "Industrie und Handelskammer Magdeburg". Her research focuses on program comprehension in the context of feature-oriented software development.

Christian Kästner is a PostDoc at the Programming Languages group of Klaus Ostermann at the Philipps University Marburg, Germany. He received his Ph.D. in Computer Science in 2010 from the University of Magdeburg, Germany. For his dissertation on virtual separation of concerns, he received the prestigious GI-Dissertation Award for the best computer-science dissertation 2010. His research focuses on correctness and understanding of systems with variability, including work on implementation mechanisms, tools, variability-aware analysis, type systems, feature interactions, empirical evaluations, and refactoring.

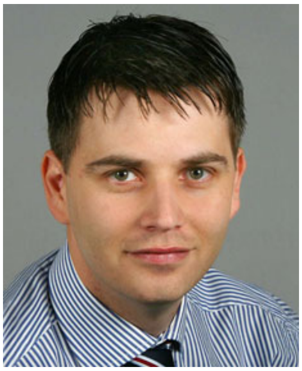
Sven Apel is the leader of the Software Product-Line Group funded by the esteemed Emmy Noether Programme of the German Research Foundation (DFG). The group resides at the University of Passau, Germany. Dr. Apel received his Ph. D. in Computer Science in 2007 from the University of Magdeburg, Germany. His research interests include novel programming paradigms, software engineering and product lines, and formal and empirical methods. He is the author or coauthor of over a hundred peer-reviewed scientific publications. Sven Apel is a member of the IFIP Working Group 2.11 (Program Generation), and serves regularly in program committees of highly ranked international conferences. His work received awards by the Ernst Denert Foundation and the Karin Witte Foundation.

Jörg Liebig received his Master's degree in Computer Systems in Engineering from the University Magdeburg, Germany, in 2008. After that, he joined the programming group of the department of informatics and mathematics at the University of Passau. His research interests include software product lines, variability management tools, analysis and transformation of software systems.

Michael Schulze is technology project leader of the pure-systems GmbH. He received his diploma degree in computer science from the University of Magdeburg, Germany in 2002. After some years in the industry he went back to the University of Magdeburg to work on his PhD that he received in 2011. His research was focused on embedded and operating systems, adaptable event-based communication middleware, and on mechanisms and concepts for resource constraint devices. At pure-systems he leads national and international technology projects and works also as consultant mainly in the area of product-line development.

Raimund Dachselt is a professor for User Interface & Software Engineering at the University of Magdeburg, Germany. In 2004, he finished his PhD thesis on the component-based development of interactive 3D applications and Information Rich Virtual Environments at TU Dresden, Germany. Currently, his research focuses on the field of novel visualization and interaction techniques for future user interfaces combining multiple input modalities and interactive displays.

Maria Papendieck is a master's student with the major Computational Visualistics at the University of Magdeburg. She received her bachelor's degree in Computational Visualistics from the University of Magdeburg in 2011. During her work as student assistent for the department of User Interface and Software Engineering, she was responsible for the implementation of FeatureCommander.



Thomas Leich is currently working toward the Ph.D. degree in Computer Science at the University of Magdeburg, Germany. He is head of the Department of Applied Informatics and member of the management at the METOP Research Institute, Magdeburg, Germany. His research interests are tailor-made and embedded data management and software product lines.

Gunter Saake received the diploma and a PhD in Computer Science from the Technical University of Braunschweig, F.R.G. in 1985 and 1988, respectively. From 1988 to 1989, he was a visiting scientist at the IBM Heidelberg Scientific Center, where he joined the Advanced Information Management project and worked on language features and algorithms for sorting and duplicate elimination in nested relational database structures. In January 1993, he received the Habilitation degree (venia legendi) for Computer Science from the Technical University of Braunschweig. Since May 1994, Gunter Saake is a fulltime professor for the area “Databases and Information Systems” at the Otto-von-Guericke University, Magdeburg. His research interests include database integration, tailor-made data management, object-oriented information systems and information fusion.