Supporting Program Comprehension in Large Preprocessor-Based Software Product Lines

Janet Feigenspan*, Michael Schulze*, Maria Papendieck*, Christian Kästner†, Raimund Dachselt*, Veit Köppen*, Mathias Frisch*, Gunter Saake*

*University of Magdeburg, Germany {feigensp, mschulze, dachselt, koeppen, mfrisch, saake}@ovgu.de, maria.papendieck@st.ovgu.de

†Philipps University Marburg, Germany kaestner@informatik.uni-marburg.de

Abstract

Background: Software product line engineering provides an effective mechanism to implement variable software. However, using preprocessors to realise variability, which is typical in industry, is heavily criticised, because it often leads to obfuscated code. Using background colours to highlight code annotated with preprocessor statements to support comprehensibility has shown effective, however, scalability to large software product lines (SPLs) is questionable. *Aim*: Our goal is to implement and evaluate scalable usage of background colours for industrial-sized SPLs. *Method*: We designed and implemented scalable concepts in a tool called FeatureCommander. To evaluate its effectiveness, we conducted a controlled experiment with a large real-world SPL with over 99,000 lines of code and 340 features. We used a within-subjects design with treatments colours and no colours. We compared correctness and response time of tasks for both treatments. *Results*: For certain kinds of tasks, background colours improve program comprehension. Furthermore, subjects generally favour background colours compared to no background colours. Additionally, subjects who worked with background colours had to use the search functions less frequently. *Conclusion*: We show that background colours can improve program comprehension in large SPLs. Based on these encouraging results, we continue our work on improving program comprehension in large SPLs.

I. Introduction

Today, *software product lines (SPLs)* provide an efficient mechanism to implement variable software. They allow deriving several distinguished program variants – *variants* for short – by selecting or deselecting features. A *feature* is a user-visible characteristic of a software system [6]. Variable code implementing a feature is called *feature code* and is only contained in a variant if the according feature is selected. In contrast to feature code, *base code* implements commonalities of an SPL and thus is part of every generated variant. As example, consider a customer buying a specific car model: She cannot choose the car body (base), but the type of engine and colour (features).

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

Both benefits and drawbacks of preprocessors are discussed controversially. Benefits are that preprocessors (a) are simple to use [11], [38], (b) are flexible and expressive, (c) can be used uniformly for different languages, and (d) are already integrated as part of many languages or environments (e.g., C, C++, Erlang, Fortran, Java Micro Edition). In contrast, in literature, preprocessors are heavily criticised and considered “harmful” [48] or even as “#ifdef hell” [34]. Numerous studies argue that preprocessor usage leads to complex and obfuscated code that is difficult to comprehend, thus leading to high maintenance costs [11], [30], [34], [42], [48].

Despite the controversial discussion of preprocessors, they are still common in practice, although there are several approaches for implementing SPLs in a modular way, such as components [23], aspects [29], mixin layers [47], or combinations [2]. The problem is that introducing novel concepts in industry is a time-consuming and difficult process, especially when large amounts of legacy code are involved.

Hence, rather than arguing that new approaches should be preferred instead of preprocessors, we target the question how we can improve the usage of preprocessors. Specifically, we aim at supporting *program comprehension*: On average, a maintenance programmer spends $50 - 60\,\%$ of her time with understanding source code [50], [52]. Furthermore, maintaining software is the main cost factor in software development [4]. Thus, by improving comprehensibility of source code using preprocessor statements, we can decrease the time and cost of software maintenance without enforcing to change the typical industrial way to implement SPLs.

To this end, we introduced background colours in some code editors to highlight feature code [14], [28]. The benefit of colours compared to text-based annotations as with preprocessors is twofold: First, the annotations clearly differ from source code, which helps a developer to distinguish feature code from base code. Second, humans process colour considerably faster than text [19]. This allows a programmer to identify feature code at first sight. Consequently, a programmer can get an overview of a software system considerably faster. However, scalable usage of background colours is questionable in large SPLs with several hundred of features.

This article is an extended version of the conference paper ‘Using Background Colours to Support Program Comprehension in Software Product Lines’ published at the conference ‘Evaluation and Assessment in Software Engineering’ [15]. In this version, we provide a more detailed description of our experimental setting to provide deeper insights. Furthermore, we conduct additional analysis of data (Section VII), in that we analyse the behaviour of subjects during the experiment. This way, we get a deeper understanding of how background colours can help to improve program comprehension.

The remainder of this paper is structured as follows: In the next section, we give a detailed problem statement of preprocessor usage and show how background colours can improve their readability. In Section III, we present our prototype *FeatureCommander*, in which we implement concepts to scale the use of background colours to industrial-sized SPLs with several hundred of features. In Sections IV to VIII, we present a controlled experiment, in which we evaluate whether our background-colour concept scales to large SPLs. We conclude our paper with related work and an outlook for future work.

II. Problem Statement

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

Furthermore, #ifdef statements can be *nested* (i.e., within a code fragment that is annotated with one #ifdef, a different #ifdef statement occurs). For example, in Fig. 1, the #ifdef in Line 13 is stated within another #ifdef, which begins in Line 5. It might be ok to deal with two nested #ifdefs (a nesting level of two). However, typical industrial SPLs can have a nesting level of up to 24, which are hard to keep track of [31].

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

Preprocessors and Background Colours: Previous Results

To improve the understandability of preprocessor-based software, we introduced and evaluated the use of background colours to highlight code fragments that are annotated with #ifdef statements [28], [12], [13]. This way, we profit from the facts that background colours clearly distinguish from source code and that humans perceive background colours preattentively[1].

In this approach, we used one-to-one mapping of background colours to features, such that each feature has one background colour. In a previous experiment, we evaluated how background colours influence program comprehension [12], [13]. We used a medium-sized SPL with about 5,000 lines of code, four features, and a nesting level of two in three occurrences [16].[2] To determine the background colour for the nested features, we blended the colours of both participating features. The SPL was implemented in Java Micro Edition (ME) with Antenna, a preprocessor for Java ME.[3] In our setting, we compared two versions of this SPL that only differed in one facet: one had background colours, the other had no background colours. As subjects, we recruited under-graduate students from a programming course, in which advanced programming paradigms, such as preprocessor-based SPLs, were taught.

The results are encouraging: We found that for certain kinds of tasks, i.e., locating annotated code fragments, the version with background colours speeded up the comprehension process of subjects. Furthermore, subjects rated the idea of background colours very positive.

However, one limitation of our study is caused by the source code we used, a medium-sized SPL with only four features, in which a one-to-one mapping of colours to features was feasible to support program comprehension.

[1]Preattentive perception describes the fast recognition of certain visual properties [19].

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

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

In such a small setting, it may not sound surprising that colours speed up program comprehension. However, the scalability of the results to realistic, industrial-sized SPLs is questionable for two reasons: First, human working memory capacity is limited, and, second, human ability to distinguish colours is limited, as well. First, in human working memory typically 7 ± 2 *items*, that is units of information, can be stored [36]. Examples of items are digits, such as in telephone numbers, or the features a developer is working with. If working memory capacity is exceeded, information units are forgotten if not stored in another way, for example, by writing it down. Second, human capability to distinguish colours is limited. In direct comparison (i.e., when colours are displayed next to each other), humans can tell up to two million colours apart [19]. Without direct comparison, humans can only distinguish few colours [44]. Hence, the scalability of background-colour usage to SPLs with several hundred of features is questionable. Clearly, a one-to-one mapping of colours to features is not feasible in large SPLs.

Instead, we suggest an as-needed mapping of colours to features, such that a developer can assign colours to features as she thinks is appropriate for her current activity. This concept is based on a number of observations of preprocessor-based software. First, for most part of the source code, only three features are present at one code fragment that fits on a screen [26]. Second, bugs can often be narrowed down to features or feature combinations [26]. Hence, often a developer needs to deal with only few features at a time, so a customisable mapping of colours to features should support program comprehension. Based on these observations and the results of our previous experiments, we developed a concept of scalable background-colour usage for preprocessor-based software, which we present next.

III. FeatureCommander

We developed a tool called *FeatureCommander*, in which we implemented several concepts to make background colours feasible for realistic industrial SPLs. In Fig. 2, we show a screenshot of FeatureCommander displaying Xenomai[4], a real-time extension to Linux with several hundred of features. We refer to the numbers in Fig. 2 when explaining the according concepts in the next paragraphs.

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

Using the colour concept, we address both aforementioned problems (i.e., the restricted human working memory capacity and the restricted human ability to distinguish colours without direct comparison): First, with the customisable colour assignment to features and the default setting (shades of grey), we support the limited working memory capacity. A developer can select the features that are relevant for her task at hand, which is typically in the range of

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

7 ± 2 (cf. Section II). Hence, she can immediately recognise that she is looking at a relevant feature, because it is coloured, while the non-relevant features do not stand out, because they are grey. Second, the developer only has to tell the same number of colours apart, which is well within the human range to distinguish colours without direct comparison. Furthermore, we support a developer in switching between tasks with different colour assignments to features, because colour assignments can be easily saved and loaded.

Similar to other IDEs, we provide different views: *source-code view*, *explorer view*, and *feature-model view*. In the *source-code view* (4), the background colour of source-code fragments indicates to which features fragments are related. To compromise between code readability and feature recognition, users can adjust the transparency of background colours (5). With the adjustable transparency, we address the problem that too intensive colours can be distracting [13].

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

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

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

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

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

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

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

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

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

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

FeatureCommander and we base our experimental design on prior empirical work [13].

IV. Experiment Planning

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

A. Objective

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

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

Large means that the source code consists of at least 40,000 lines of code [54] and considerably more than 7 ± 2 features, such that humans cannot distinguish colours without direct comparison, if we used a one-to-one mapping of colours to features.

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

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

B. Experimental Material

For our experiment, we used a large SPL – Xenomai a real-time extension for Linux. Xenomai follows a dual-kernel approach, which means that it acts as primary kernel, having real-time capabilities; the Linux kernel is executed within Xenomai's idle task whenever nothing else has to be done in real time. Xenomai runs on different platforms, supports a variety of features including real-time communication and scheduling. It is used in several systems to assure real-time behaviour, for example *RT-FireWire*[5], *USB for Real-Time*[6], and *SCALE-RT Real-time Simulation Software*[7]. The whole source code (not including Linux) consists of about 99,010 lines of code including 24,709 lines of feature code and 346 different features, which are responsible for selecting platform-specific code, special hardware drivers, or different timing policies.[8] It is comparable with other real-world systems, such as SQLite (94 463 lines of code, 48 845 lines of feature code, 273 features) and Sylpheed (99 786 lines of code, 13 607 lines of feature code, 150 features).

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

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

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

[8]Analysed with cppstats, available at http://fosd.de/cppstats.

To present the source code to our subjects, we generated two tailored versions of FeatureCommander. This was necessary to evaluate our research hypothesis **RH1**, whether the use of background colours scales to large SPLs. In a *colour version*, we deleted the explorer view ordered by features (cf. left in Fig. 2) and the sidebars in the source-code view. Otherwise, subjects could use the functionality of the sidebars to locate feature code and we would measure its usage instead of the scalability of using background colours. Furthermore, we defined different sets of colours for each task, which subjects were instructed to load, depending on the features that were relevant for each task. We selected colours, such that they were consistent between tasks (e.g., if a feature occurred in two tasks, it had the same or similar colour in both tasks) and such that humans can clearly distinguish all colours without direct comparison [44]. However, we did not optimize the colour selection for colour-blind people. In future work, guidelines for choosing colours should be considered.[9] We decided to specify the colours, so that subjects would not spend their time with assigning colours to features, but work on tasks. In a *colourless version*, we removed everything associated to colours.

For both versions, we implemented three search functionalities: First, search in the complete project, second, search in an opened file, and, third, search in the feature-model view. This is necessary because of the size of the project, so that we do not measure how fast subjects can find a certain file in about thousand files or a certain feature in about 340 features. Instead, the search functions allow subjects to locate a certain code fragment within a file or feature within a list of features in both versions of our tool. Furthermore, this is a more realistic setting, since typical IDEs provide similar search functionalities.

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

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

C. Subjects

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

[9] See, for example, http://www.lighthouse.org/accessibility/design/

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

To measure programming experience, we administered a questionnaire before the experiment, in which a low value (min: 5) indicates no experience, a high value (over 60 – the scale is open-ended) high programming experience. All subjects were familiar with C (median: 4, on a five-point Likert scale [33], 1 meaning very unexperienced, 5 very experienced). All subjects were male; none was colour blind.[10] We created two groups of seven subjects each with comparable programming experience according to the value of the programming-experience questionnaire.

D. Tasks

To measure program comprehension, we designed tasks that can only be solved if subjects understand according source code. We used two kinds of tasks: maintenance and static tasks [10]. In maintenance tasks, subjects are instructed to locate/fix a bug, while in static tasks, subjects should examine the structure of the source code. In a previous experiment [12], we found that colours speed up program comprehension only for static tasks, but not in maintenance tasks. Hence, we focused on static tasks, but included few maintenance tasks confirm our results.

All tasks are typical for a maintenance programmer, when she is looking for bugs in certain features and/or files. We had 10 tasks: 2 warming up tasks (W1, W2), 6 static tasks (S1 – S6), and 2 maintenance tasks (M1, M2). The warming up tasks were designed to let subjects familiarise with the experimental setting and were not analysed. Regarding static tasks, we had three different types:

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

For each type, we prepared two tasks. As example, we present one task for each type:

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

Altogether, the feature occurred in 9 files that were distributed in two folders: One include folder that contained the header files, and one other folder that contained the c-files. In this task, the feature was annotated with a yellow background colour. Hence, subjects with the colour version only had to look for a yellow background colour in the software project.

S2: Do features `CONFIG_XENO_OPT_PRIOCPL` and `CONFIG_XENO_OPT_SCHED_SPORADIC` occur together (i.e., nested) somewhere? If yes, in which files? At which lines does the inner feature start and end?

[10]The chosen colours were not tested regarding their suitability for colour blindness. For future work, this is an important issue to evaluate. However, in this paper, our goal is to evaluate whether colours can help at all.

Both features were nested in the file *xenomai/ksrc/nucleus/sched-sporadic.c*, once in the middle, once near the end of the file. Nested #ifdef statements are especially error prone and difficult to get right, which is why we included this type of task. We assigned yellow and blue to both features, because those colours are clearly distinguishable. To solve this task, subjects with the colour version had to look for a joint occurrence of yellow and blue, and make sure that the according #ifdef statements are nested, not just used subsequently. In the second task of this type, the nesting occurred only at one position, instead of two.

S3: Which features occur in file *xenomai/ksrc/nucleus/sched.h*?

In this task, subjects had to identify twelve different features. Hence, subjects with the colour version had to look for twelve different colours (see project's website for colour assignment). This is an important task to get an overview of a file.

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

M1: If the PEAK parallel port dongle driver (`XENO_ DRIVERS_CAN_SJA1000_PEAK_DNG`) should be unloaded, a segmentation fault is thrown.

The problem occurs, when features `CONFIG_XENO_DRIVERS_CAN` and `CONFIG_XENO_DRIVERS_CAN_SJA1000` and `CONFIG_XENO_DRIVERS_CAN_SJA1000_PEAK_DNG` are selected.

The bug was located in class *xenomai/ksrc/nucleus/heap.c* in function `xnheap_test_and_free`, Line 669. The correct code would say `if (ckfn && (err = ckfn(block)) != 0)`, but we deleted the check whether variable `ckfn` is null: `if ((err = ckfn(block)) != 0)`. Hence, when this variable is accessed and it is null, a segmentation fault would be thrown. This may seem like a difficult task, however, it is realistic. Hence, we can interpret our results in relation to a realistic setting.

E. Design

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

Group	1. Phase	2. Phase
A	W1, S1, S2, S3, M1	W2, S4, S5, S6, M2
B	W1, S1, S2, S3, M1	W2, S4, S5, S6, M2

TABLE I

OVERVIEW OF EXPERIMENTAL DESIGN. GREY CELLS INDICATE THAT THE ACCORDING GROUP WORKED WITH THE COLOUR VERSION.

F. Conduction

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

V. ANALYSIS

In this section, we present the analysis of our data. First, we present some descriptive statistics, then we analyse whether our hypotheses hold. We strictly separate analyzing our data from interpreting the results, so that we enable the reader to put her own interpretation to our data.

A. Descriptive Statistics

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

In Fig. 4, we show the response time of our subjects with a *box plot* [1]. It plots the median as thick line and the quartiles as thin line, such that 50 % of all measurements are inside the box. Values that strongly deviate from the median are outliers and drawn as separate dots. We can see that for the first two static tasks (S1 and S2), group A (colour version) is faster than group B. For all other tasks, the difference between both groups is rather small. We can also see that subjects needed considerably more time to solve the maintenance than static tasks (note the different scales between both figures).

Task	Group	Correct	Incorrect	χ^2	degr. freedom	p value
S1+S2+S3	A	11	10	0	1	1.000
	B	11	10			
S4+S5+S6	A	13	8	0.104	1	0.747
	B	14	7			

TABLE II
χ^2 TEST FOR CORRECTNESS OF EACH SOLUTION.

The estimation of subjects regarding difficulty, motivation, and performance with the other version (colour/colourless) are shown in Fig. 5. For difficulty, in four static tasks (S1: locating files of a feature; S2, S5: locating nested #ifdef statements; S3: locating all features in a file) and one maintenance task, the median is the same. For the other tasks, the median differs by 1. For motivation, the deviation within a group is larger than for difficulty, meaning that subjects rated their motivation more heterogeneously. Both groups were motivated for all tasks at least to a mediocre level. For the first maintenance task (M1), the motivation for group A (with colours) was very high, in contrast to group B with a mediocre motivation. For estimation of performance with the other version, we see that in both phases, subjects that worked with the colour version thought they would have performed worse with the colourless version, and vice versa.

When asked what version they prefer, 12 of our 14 subjects said they like the colour version better and 13 said the colour version is more suitable when working with preprocessor-based SPLs. One subject did not answer any of both questions.

B. Hypotheses Testing

In this section, we evaluate whether our research hypotheses hold. To this end, we conduct several statistical tests to check whether the differences we observed are significant. We start with correctness of answers. Since we compare frequencies, we need to conduct a χ^2 test [1]. To meet its requirements, we summarise the correctness of answers for the static tasks of each phase, such that we add the number of correct and incorrect answers for each phase.[11] Hence, we compare the number of correct and incorrect answers of tasks $S1 + S2 + S3$ and $S4 + S5 + S6$. The χ^2 test indicates no significant differences in the number of correct answers for static tasks, as shown in Table II. Since for maintenance tasks, none of the subjects provided a correct solution, we do not need to test for significant differences in correctness of maintenance tasks.

For response time, we make several comparisons for our data: Group A vs. group B, group A (first phase) vs. group A (second phase), and group B (first phase) vs. group B (second phase). Since we make multiple comparisons, we need to adjust the significance level. To this end, we use a Bonferoni correction, which divides the significance

[11]Expected frequencies are too small due to the small number of subjects.

Task	S1	S2	S3	M1	S4	S5	S6	M2
Mean A	3.04	5.28	4.07	13.32	4.54	6.52	4.10	24.45
Mean B	6.58	10.33	3.34	16.55	4.01	4.57	3.57	26.59
t value	-4.276	-2.759	0.919	-1.096	0.639	1.425	0.690	-0.282
p value	0.001	0.017	0.376	0.295	0.535	0.180	0.503	0.783

TABLE III

SIGNIFICANCE TESTS FOR RESPONSE TIMES BETWEEN GROUPS.

Task	S1/S4	S2/S5	S3/S6	M1/M2
Group A				
t value	-2.117	-0.932	-0.042	-1.649
p value	0.079	0.387	0.968	0.150
Group B				
t value	3.973	3.591	-0.451	-2.770
p value	0.007	0.011	0.668	0.032

TABLE IV

SIGNIFICANCE TESTS FOR RESPONSE TIMES WITHIN GROUPS.

level (in our case, 0.05) by the number of comparisons [1]. This leads to a significance level of 0.017 ($= 0.05/3$) to observe a significant difference.

First, we compare the results of both groups. We applied t-tests for independent samples [1], since the response times are normally distributed (tested with a Kolmogorov-Smirnov test [35]). In Table III, we present the results of our tests. We can see that only for tasks S1 and S2, subjects of group A (colour version), were significantly faster than subjects of group B (p value is smaller than 0.017). Furthermore, we computed Cohen's d [7], which indicates a large effect for both tasks (S1: -2.29; S2: -1.46). When we switched the versions, such that group B worked with the colour version, we could not observe any differences.

Second, we make pairwise comparisons within our groups of tasks of both phases, i.e., S1 vs. S4, S2 vs. S5, S3 vs. S6, and M1 vs. M2. In Table IV, we show the results of the t-test for independent samples. We only observed significant differences in group B, such that the response times for S4 and S5 were significantly faster than for S1 and S2, respectively (again, Cohen's d indicates a large effect: S1/S4: 1.56, S2/S5: 1.79). Hence, when subjects switched from the colourless to the colour version, their performance for two tasks increased. Group A, on the other hand, was not slower in the second phase. This observation is unexpected, because they worked without colours now. The results regarding response time speak both in favour of and against our first research hypothesis **RH1**. Since we have ambiguous results, we have to reject our first research hypothesis.

Finally, we compare the opinion of subjects. Since they are ordinally scaled, we use a Mann-Whitney-U test [1].

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

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

In Table V, we summarise the results of this test. To meet the requirements of the Mann-Whitney-U test, we use the probability function of the U distribution for critical values [18] to state whether the observed differences are significant. Some U values are 0, yet the difference is significant, because of the extreme distribution of answers.

We can see that for difficulty, subjects of group B rated S4 and S5 significantly easier than subjects of group A. This is also reflected in the performance, such that subjects of group B are faster in these tasks (S4 vs. S1, S5 vs. S2). For motivation, we observe a significant difference for the first maintenance tasks, such that subjects of group A were more motivated to solve this task compared to group B. For estimation of the performance with the other version, we obtain significant differences for all tasks (except M1), such that subjects that worked with the colour version expect that they had performed worse with the colourless version. The results regarding the estimation of performance and how subjects liked the colour version and evaluated its suitability speak in favour of our second research hypothesis (i.e., that subjects rate background colours more positive). Hence, we can accept our second research hypothesis **RH2**.

VI. INTERPRETATION

In this section, we discuss the implication of our results for each hypothesis.

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

The data we observed lead us to rejecting this hypothesis, because our results speak both in favour of and against our hypothesis. Since this experiment is not exploratory, but rather to confirm previous results, it is custom to accept the null hypothesis. To evaluate this hypothesis, we measured the correctness of answers of subjects and the response time. There was no difference regarding correctness of static tasks. For response time, we found that in the first phase, for two static tasks, subjects that worked with background colours were significantly faster. In the second phase, in which we switched the versions, we did not observe any significant differences in response time between both groups. However, we found that within group B (subjects that started to work with the colourless version and switched to the colour version) were faster in tasks S4 and S5, compared to S1 and S2, respectively, for which we observed a significant difference in the first phase.

Hence, for two static tasks, background colours improve program comprehension. One reason that for the third kind of static tasks (locating all features in a file), background colours showed no improvement, could be that we use 12 features and thus 12 different colours in this task. Although we carefully chose colours, such that subjects could distinguish them easily [44], 12 colours might be too much for subjects. Further research on this topic would be interesting, for example to find out a more certain number about the limit of directly discriminable features. One problem could be that several colours are perceived rather in the periphery of the eye, in which colour perception and the ability to discriminate colours deteriorates [37]. Furthermore, with 12 different colours, the working memory capacity is exceeded. In the other tasks, 9 colours at most have to be kept in mind, which is in the top end of 7 ± 2. However, we cannot be sure whether this result occurred because of too many different colours or because of the kind of task, since we only combined 12 features with this kind of task. Nevertheless, is an interesting issue to analyse more closely in future work.

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

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

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

VII. Exploratory Analysis

To get a deeper understanding of how background colours influence the behaviour of our subjects, we analyse how subjects used the provided search functions. In this section, we explore the log data without stating a specific research question. Based on our analysis, we can derive research questions for future experiments. Note that we strictly separate reporting the data from their interpretation.

To support the subjects in getting and keeping an overview, we implemented three different search functions. As typical for IDEs, subjects were able to search for a term in the complete project (project-wide search) or in a

Task	A	B	χ^2	significant
S1	28	33	0.41	no
S2	28	316	241.116	yes
S3	0	85	85	yes
M1	186	248	8.333	yes
S4	94	10	67.846	yes
S5	525	17	476.132	yes
S6	188	72	51.754	yes
M2	671	496	26.243	yes

TABLE VI

FREQUENCY OF OVERALL SEARCH USAGE AND ACCORDING χ^2 VALUES REGARDING SIGNIFICANCE OF DIFFERENCE. GREY CELLS INDICATE THAT A GROUP WAS WORKING WITH THE COLOUR VERSION.

certain file (file-wide search). Furthermore, subjects could search for a certain feature in the feature-model view (feature-wide search), for example, to help them remember the according colours. In this section, we analyse and compare the behaviour of subjects of both groups. First, we start with comparing the overall search behaviour for each task. Then, we analyse each search functionality individually.

A. *Overall Search Usage*

We logged every use of each search functionality for each subject. In Table VI, we show how frequently groups used any of the search functions. To obtain these values, we added each use of any search function of all subjects per group. We found that subjects that used the colour version (highlighted) used the search less frequently. For one task (S3), subjects with the colour version did not use the search at all. In this task, subjects should identify all features that occurred in a specified file. In the first static tasks, in which subjects should locate all files of a feature, the difference in frequency of search usage is rather small, compared to all other tasks.

We also conducted significance tests to analyse whether the observed differences in frequencies are significant. The right columns of Table VI contain the χ^2 values and whether it indicates a significant difference. Since the degrees of freedom is 1, and we set a significance level of 0.05, χ^2 values larger than 3.84 indicate a significant difference. Only for the first task S1, the difference is not significant. For all other tasks, subjects with the colourless version used the search functions significantly more often.

Next, we analyse each search function individually, starting with the project-wide search.

B. *Project-wide Search*

When getting familiar with a system or tracing the call of a function, a developer often searches information scattered through the complete system. Hence, we provided this kind of search for our subjects, as well. This was also necessary because of the size of Xenomai with about 1000 files. Furthermore, since colours were also visible

Task	A (%)	B (%)	χ^2	significant
S1	3 (10.7)	10 (30.3)	3.769	no
S2	8 (28.6)	58 (18.4)	37.879	yes
S3	0 (n/a)	1 (1.2)	1	no
M1	11 (5.9)	49 (19.9)	24.067	yes
S4	27 (28.7)	6 (60.0)	67.846	yes
S5	76 (14.5)	10 (58.8)	13.364	yes
S6	0 (0.0)	0 (0.0)	n/a	no
M2	51 (7.6)	84 (16.9)	8.067	yes

TABLE VII

FREQUENCY OF USAGE OF PROJECT-WIDE SEARCH AND ACCORDING χ^2 VALUES REGARDING SIGNIFICANCE OF DIFFERENCE. GREY CELLS INDICATE THAT A GROUP WAS WORKING WITH THE COLOUR VERSION.

in the explorer view, subjects that worked with the colour version would have had a clear advantage. To be able to make a fair comparison between both groups, we also needed the project-wide search. In Table VII, we summarise how often subjects used the project-wide search. To have a better impression of the frequency, we additionally report the percentage of the project-wide search in comparison to the overall usage of the search functions. For example, subjects of group A used in 10.7% of the cases they used a search the project-wide search.

We can see that subjects used the project-wide search most often for the first two kind of static tasks, i.e., finding the occurrence of a feature across files and locating nested #ifdef directives, respectively. Additionally, group B used this search for the maintenance tasks more often than group A, independent of the version the groups worked with. For this search, we also conducted significance tests. For the first static task (S1) as well as for the third type of static task (i.e., identifying all features of a file), we found no significant difference in the frequencies. All other differences are significant, such that subjects with the colourless version used the search more frequently (except for the last maintenance task).

C. File-wide Search

To get an overview of a file, the file-related search can be used. In our tool, we provided this search to our subjects because of the size of the files (sometimes more than 1000 lines). In Table VIII, we summarise how often subjects used this search. We can see that subjects in general used this search quite often. Subjects of the colourless version always used this search more often than subjects of the colour version (except for S1, i.e., finding all files in which a feature occurs). The observed differences are also significant, except for S1 and the first maintenance task.

D. Feature-wide Search

Since the list of features was quite large with over 300 features, we also provided a search for a certain feature in a list of all features. In Table IX, we show how frequently this search was used. We see that subjects used this

Task	A (%)	B (%)	χ^2	significant
S1	15 (53.6)	14 (42.2)	0.034	no
S2	20 (71.4)	253 (80.1)	198.861	yes
S3	0 (n/a)	84 (98.8)	84	yes
M1	174 (93.6)	188 (76.4)	0.541	no
S4	63 (67.0)	4 (40.0)	51.955	yes
S5	449 (85.5)	7 (41.2)	428.43	yes
S6	188 (100.0)	68 (94.4)	56.25	yes
M2	614 (91.5)	381 (76.8)	54.562	yes

TABLE VIII

FREQUENCY OF USAGE OF FILE-WIDE SEARCH AND ACCORDING χ^2 VALUES REGARDING SIGNIFICANCE OF DIFFERENCE. GREY CELLS INDICATE THAT A GROUP WAS WORKING WITH THE COLOUR VERSION.

Task	A (%)	B (%)	χ^2	significant
S1	10 (35.7)	9 (27.3)	0.053	no
S2	0 (0.0)	5 (1.6)	5	yes
S3	0 (n/a)	0 (0.0)	n/a	no
M1	1 (0.5)	11 (4.5)	8.333	yes
S4	4 (4.3)	0 (0.0)	4	yes
S5	0 (0.0)	0 (0.0)	n/a	no
S6	0 (0.0)	4 (5.6)	4	yes
M2	6 (0.9)	11 (2.2)	1.471	no

TABLE IX

FREQUENCY OF USAGE OF FEATURE-WIDE SEARCH AND ACCORDING χ^2 VALUES REGARDING SIGNIFICANCE OF DIFFERENCE. GREY CELLS INDICATE THAT A GROUP WAS WORKING WITH THE COLOUR VERSION.

search not very often. Only in the first task, in which subjects should locate all files in which a feature occurs, the search is used to about a third (group A) or a fourth (group B) of the time of the complete search usage. For the feature-wide search, we have significant difference in S2, S4, and M1, in which group A used it more often, and in S6, in which group B used it more often.

E. Interpretation

For identifying all files of a feature (S1, S4) and identifying feature interactions (S2, S5), subjects use the project-wide search 10 % to 60 % of the time. Since in these tasks, subjects had to identify several files, the project-wide search is a good support. However, when subjects had background colours, they used this search function less frequently than subjects who worked with the colour version. An interesting result occurred for the maintenance tasks, in which group B always used the project-wide search more frequently, independent of the version they worked with. When we look at the motivation of group B, we can see that they were less motivated to solve these tasks than group A. This could be the reason they used the project-wide search more often, because they were not motivated to look into details. Instead, they may have been looking more superficial at the source code. Hence, they looked at more different code fragments, which they located using the project-wide search. This could explain why subjects of group B always used the project-wide search more often for the maintenance tasks.

The file-wide search was used most of the time when a search function was used. This could be caused by the fact that after identifying a file, the according feature still needed to be found in a file. When looking at the terms subjects looked for, subjects entered most of the time a feature-related expression, such as `CONFIG_XENO` or `#ifdef`. This indicates that they were looking for the occurrence of a feature in a file. However, for the colour version, according subjects did not use the file-wide search as often as subjects that worked with the colourless version. This indicates that subjects with the colour version rather scrolled through the file to locate features than using the file-wide search.

For the feature-wide search, subjects with the colourless version also seemed to use this search more often. However, the amount of usage of this search is very small compared to the total amount. Only for S6 (finding all features in a file), subjects that worked with the colour version used the feature-wide search more often than subjects that worked with the colourless version. We assume that this result is caused by the large number of features for this task (12) and the fact that the experiment was near its end, which could mean a loss of concentration or fatigue of subjects. Hence, subjects might had to look up the colour of features in the feature-model view more often. However, we cannot be sure why this result occurred.

Putting it all together, the analysis of the behaviour of subjects speaks in favour of background colours. In general, subjects with the colourless version used the search more frequently. Only for S1, this difference is not significant, although the tendency is the same. Using the search function often interrupts the workflow of a programmer, because she has to bring a search window or pop-up to the front and enter the according search string. Then, she has to look through the search results and decide which one is relevant for her current task and which not. Since interruptions of the work flow can be very expensive [49], they should be avoided where possible. Our results indicate that background colours can limit the amount of interruptions caused by search usage. This result encourages us to use background colours more often in preprocessor-based software.

VIII. Threats to Validity

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

A. Internal Validity

Since there is no standardised assessment of programming experience, we applied our own questionnaire to our subjects and created homogeneous groups. However, we cannot be sure how well we measured programming experience. To reduce this threat, we developed the questionnaire based on literature research and programming experts, who rated the questionnaire regarding how well it measures programming experience [12].

Another problem is that none of the subjects solved any of the maintenance tasks correctly. Since we designed the maintenance tasks to be realistic, we supsect that they were too difficult, given the subjects' expertise and time constraints of the experiment. Experienced developers with more time should find this bug eventually. Furthermore, maintenance tasks are not our primary focus, since we found in a previous study that colours do not affect the comprehension process in these tasks [12], [13].

Our sample is rather small. However, we used several mechanisms to deal with this issue: We used a within-subjects design to apply both versions of our tool to all subjects. Furthermore, we applied variants of standard significance tests that were developed to deal with small sample sizes. Additionally, Cohen's d indicates large effects. Hence, we controlled threats caused by the small sample sizes.

Additionally, we did not correct the response times for wrong answers. This should be done to deal with the problem that wrong answers may lead to different response times than right answers would have (e.g., a subject might enter an answer just to finish a task). There are two common ways: First, omitting the response time of wrong answers from the analysis. However, the size of our sample does not allow us to do so. Second, computing an efficiency measure as combination of correctness of answers and response time (e.g., [40]). However, it is not clear whether the use of such a measure may lead to falsely accepting or rejecting 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. Hence, we decided to analyse correct and wrong answers in the same way. Since our results do not indicate that subjects deliberately entered a wrong answer just to be finished with a task, that is, only one or two features or files were missing and none of the response time for any task deviated considerably towards zero, we argue that this threat is negligible in our study.

B. *External Validity*

One threat is caused by our sample, which consisted mostly of master students with relatively little programming experience. The benefit of recruiting students is that we can offer them credits for the course, instead of money for their participation. Of course, the suitability of students is discussed controversially [24], [51], [53]. We reduced this threat by including PhD students in our sample with several years of experience in the domain of embedded and operating systems, so our results can be carefully applied to experienced programmers.

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

IX. Related Work

There is a lot of work dealing with visualisation of preprocessors, for example, with control-flow graphs [25], [30], [41]. For example, Hu et al. [25] propose the analysis of control-flow graphs based on preprocessor directives to get an overview of the inclusion structure of a file. In contrast, our focus lies on supporting preprocessor statements on the source code level and, thus, keeping context information of preprocessor statements.

Another way to handle the complexity of preprocessors is to provide views on source code [3], [5], [26], [46]. A view on a variant or feature shows only relevant code for a feature and its combination and hides all remaining code. In some tools, annotations are hidden entirely and the developer works on a single variant without even being aware of other variants or features. In an empirical study on views in the Version Editor, [3] measured a 40 % increase in developer productivity. Nevertheless, hiding feature code is not always desirable; for example, when fixing a bug, a developer may need the context of the entire SPL to fix the bug not only in a single variant, but in all variants. Views on the source code and colours are complementary and have strength for different tasks.

Regarding colours, there is a huge body of work on using colours for various tasks, such as highlighting source code according to semantic of statements or control structure [43], error reporting [39], or merging [55]. We focus only on work that addresses program comprehension in SPLs or for scattered concerns. In prior work, we used background colours to represent annotations in our SPL tool CIDE, with which we explore improvements of preprocessors [14], [26], [27]. We showed that for small SPLs, background colours can improve program comprehension [12], [13]. However, the scalability of CIDE is questionable, because we used a one-to-one mapping of colours to features and we blend colours of nested #ifdef directives. With FeatureCommander, we provide a scalable background-colour concept in large SPLs. We provide an as-needed mapping of colours to features and use sidebars to visualise nested #ifdef directives.

Other tools also use colour for highlighting. Closest to our representation with background colours are the model editors *fmp2rsm* [9] and *FeatureMapper* [22], in which model elements can be annotated and removed to generate different model variants. Both tools provide views and additionally can represent some or all annotations with colours. Furthermore, *Spotlight* [8] addresses scattered concerns (outside of the context of SPLs). Spotlight uses vertical bars in the left margin of the editor to visualise annotations, which we also use in FeatureCommander (cf. Fig. 2, (8)). Again, different colours represent different concerns. Bars of different colours are placed next to each other. Compared to background colours, lines are more subtle and can represent nesting easily. In all cases, the impact of visualizing annotations was not measured empirically so far.

There is a lot of empirical work regarding the evaluation of SPL implementing techniques. Especially aspect-oriented programming is at focus of several researcher groups, for example, [16], [17], [20], [21]. For example, [16], [17] and [20] evaluated several facets of aspect-oriented programming, such as maintainability or design stability. However, the assessment is *not* conducted with human subjects, but with software measures. In the work of [21], the understandability of aspect-oriented programming is compared to object-oriented programming. In an experiment, subjects had to implement crosscutting code into a small target application, one implemented in AspectJ, the other in Java. Depending on the kind of code changes, AspectJ had positive or negative influence on the development time of subjects.

Independent of visualisation, other researchers analyze the discipline and granularity of #ifdef directives [31], [32], [45]. The studies found that #ifdef directives are often used, often disciplined, and have a fine granularity. This supports our argument that the preprocessor will be used at least in the medium-term future and that we need to improve its readability. With our background-color concept, we provide one approach.

X. Conclusion and Future Work

Software product lines are typically implemented with preprocessors in industry. However, preprocessors are often considered harmful, because they can lead to obfuscated and difficult-to-comprehend source code. Hence, we introduced scalable concepts to highlight annotated code fragments with background colours. In our prototype FeatureCommander, we implemented these concepts to support program comprehension in large SPLs. To scale background-colour usage, we used a default setting, in which annotated code is highlighted with alternating shades of grey. A developer can assign colours to features on an as-needed basis. Furthermore, we provide easy storing and loading of colour assignments, so that switching between different tasks is supported.

To evaluate whether our background-colour concept scales to a large SPL with over 99,000 lines of code, 340 features, and 25,000 lines of annotated code, we conducted a controlled experiment. In this experiment, we designed several comprehension tasks to analyse the effect of background colours. We found that for locating feature code, subjects tend to be faster. Furthermore, subjects like the usage of background colours and rate them as suitable when working with preprocessor-based software. Additionally, the search behaviour of our subjects underlined the positive effect on background colours: Subjects that worked with background colours did not have to use the search that often, compared to subjects that worked without background colours. Hence, the results of our experiment are very promising and suggest using background colours more often in preprocessor-based software.

In future work, we plan to evaluate open issues we discovered in our experiment to confirm the positive effect of background colours on program comprehension and broaden our understanding of how background colours can be used to support program comprehension. For example, how can background colors help colour-blind programmers? Additionally, we can evaluate how other implemented concepts in FeatureCommander, such as the explorer view ordered by features, improve program comprehension. Furthermore, it would be interesting to evaluate how other concepts in conjunction with highlighting preprocessor statements, for example, hiding annotated code, influence program comprehension.

Acknowledgment

Feigenspan's, Köppen's, Frisch's, and Saake's work is supported by BMBF project 01IM10002B (ViERforES-II). Kästner's work is supported in part by ERC (#203099). Dachselt's work is funded by the "Stifterverband für die Deutsche Wissenschaft" from funds of the Claussen-Simon-Endowment. Thanks to Jana Schumann for support during the conduction and analysis of the experiment. Thanks to Jörg Liebig for computing statistics of Xenomai.

REFERENCES

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

[2] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Trans. Softw. Eng.*, 34(2):162–180, 2008.

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

[4] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

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

[6] P. Clements and L. Northrop. *Software Product Lines: Practice and Patterns*. Addison Wesley, 2001.

[7] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. Academic Press, first edition, 1969.

[8] D. Coppit, R. Painter, and M. Revelle. Spotlight: A Prototype Tool for Software Plans. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 754–757. IEEE CS, 2007.

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

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

[11] J. Favre. Understanding-In-The-Large. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, page 29. IEEE CS, 1997.

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

[13] J. Feigenspan, C. Kästner, S. Apel, and T. Leich. How to Compare Program Comprehension in FOSD Empirically - An Experience Report. In *Proc. Int'l Workshop on Feature-Oriented Software Development (FOSD)*, pages 55–62. ACM Press, 2009.

[14] J. Feigenspan, C. Kästner, M. Frisch, R. Dachselt, and S. Apel. Visual Support for Understanding Product Lines. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 34–35. IEEE CS, 2010.

[15] J. Feigenspan, M. Schulze, M. Papendieck, C. Kästner, R. Dachselt, V. Köppen, and M. Frisch. Using Background Colors to Support Program Comprehension in Software Product Lines. In *Proc. Int'l Conf. Evaluation and Assessment in Software Engineering (EASE)*, pages 66–75. Institution of Engineering and Technology, 2011.

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

[17] E. Figueiredo, C. Sant'Anna, A. Garcia, T. Bartolomei, W. Cazzola, and A. Marchetto. On the Maintainability of Aspect-Oriented Software: A Concern-Oriented Measurement Framework. In *Proc. Europ. Conf. Software Maintenance and Reengineering (CSMR)*, pages 183–192. IEEE CS, 2008.

[18] L. Giventer. *Statistical Analysis for Public Administration*. Jones and Bartlett Publishing, second edition, 2008.

[19] B. Goldstein. *Sensation and Perception*. Cengage Learning Services, fifth edition, 2002.

[20] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza, and A. Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 176–200. Springer, 2007.

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

[22] F. Heidenreich, J. Kopcsek, and C. Wende. FeatureMapper: Mapping Features to Models. In *Comp. Int'l Conf. Software Engineering (ICSE)*, pages 943–944. ACM Press, 2008.

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

[24] M. Höst, B. Regnell, and C. Wohlin. Using Students as Subjects – A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empirical Software Engineering Journal (ESE)*, 5(3):201–214, 2000.

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

[26] C. Kästner. *Virtual Separation of Concerns: Preprocessors 2.0*. PhD thesis, University of Magdeburg, 2010.

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

[28] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for all Product Line Variants: A Language-Independent Approach. In *Proc. Int'l Conf. Objects, Models, Components, Patterns (TOOLS EUROPE)*, pages 174–194. Springer, 2009.

[29] G. Kiczales et al. Aspect-Oriented Programming. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 220–242. Springer, 1997.

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

[31] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 105–114. ACM Press, 2010.

[32] J. Liebig, C. Kästner, and S. Apel. 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, 2011. To Appear.

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

[34] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 191–204. ACM Press, 2006.

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

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

[37] J. Moreland. Peripheral Color Vision. *Handbook of Sensory Physiology*, 7(4):517–536, 1972.

[38] D. Muthig and T. Patzke. Generic Implementation of Product Line Components. In *Int'l Conf. NetObjectDays*, pages 313–329. Springer, 2003.

[39] B. Oberg and D. Notkin. Error Reporting with Graduated Color. *IEEE Software*, 9(6):33–38, 1992.

[40] M. Otero and J. Dolado. Evaluation of the Comprehension of the Dynamic Modeling in UML. *Journal of Information and Software Technology*, 46(1):35–53, 2004.

[41] T. Pearse and P. Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 270–277. IEEE CS, 1997.

[42] K. Pohl, G. Böckle, and F. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

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

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

[45] S. Schulze, E. Jürgens, and J. Feigenspan. Analyzing the Effect of Preprocessor Annotations on Code Clones. In *Proc. IEEE Int'l Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 115–124. IEEE CS, 2011.

[46] N. Singh, C. Gibbs, and Y. Coady. C-CLR: A Tool for Navigating Highly Configurable System Software. In *Proc. Workshop Aspects, Components, and Patterns for Infrastr. Software*. ACM Press, 2007.

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

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

[49] J. B. Spira and J. B. Feintuch. *The Cost of Not Paying Attention: How Interruptions Impact Knowledge Worker Productivity*. Basex, 2005.

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

[51] M. Svahnberg, A. Aurum, and C. Wohlin. Using Students as Subjects – An Empirical Evaluation. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, pages 288–290. ACM Press, 2008.

[52] R. Tiarks. What Programmers Really Do: An Observational Study. In *Workshop Software Reengineering (WSR)*, pages 36–37, 2011.

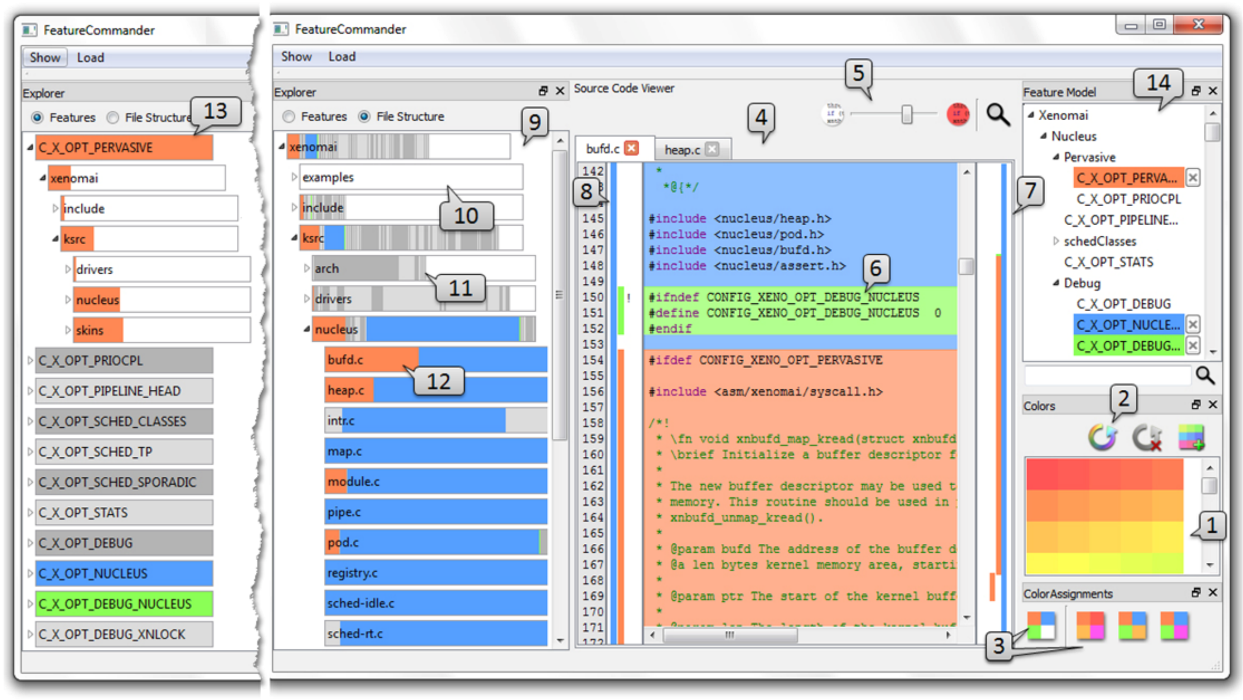
[53] W. Tichy. Hints for Reviewing Empirical Work in Software Engineering. *Empirical Software Engineering Journal (ESE)*, 5(4):309–312, 2000.

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

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

```
static int __rep_queue_filedone(dbenv, rep, rfp)
   DB_ENV *dbenv;
   REP *rep;
   __rep_fileinfo_args *rfp; {
 #ifndef HAVE_QUEUE
   COMPQUIET(rep, NULL);
   COMPQUIET(rfp, NULL);
   return (__db_no_queue_am(dbenv));
 #else
   db_pgno_t first, last;
   u_int32_t flags;
   int empty, ret, t_ret;
 #ifdef DIAGNOSTIC
   DB_MSGBUF mb;
 #endif
   // over 100 lines of additional code
 #endif
}
```

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



```
 *
  *@{*/

#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  0
#endif

#ifdef CONFIG_XENO_OPT_PERVASIVE

#include <asm/xenomai/syscall.h>

/*!
 * \fn void xnbufd_map_kread(struct xnbufd
 * \brief Initialize a buffer descriptor f
 *
 * The new buffer descriptor may be used t
 * memory. This routine should be used in
 * xnbufd_unmap_kread().
 *
 * @param bufd The address of the buffer d
 * @a len bytes kernel memory area, starti
 *
 * @param ptr The start of the kernel buff
 *
```

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

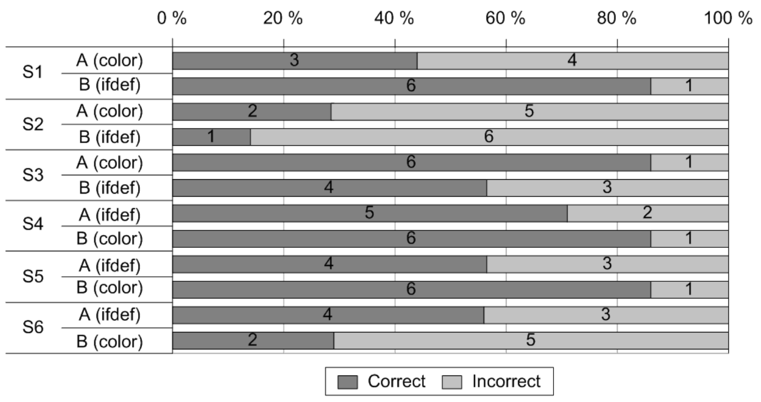


Fig. 3. Correctness of answers.

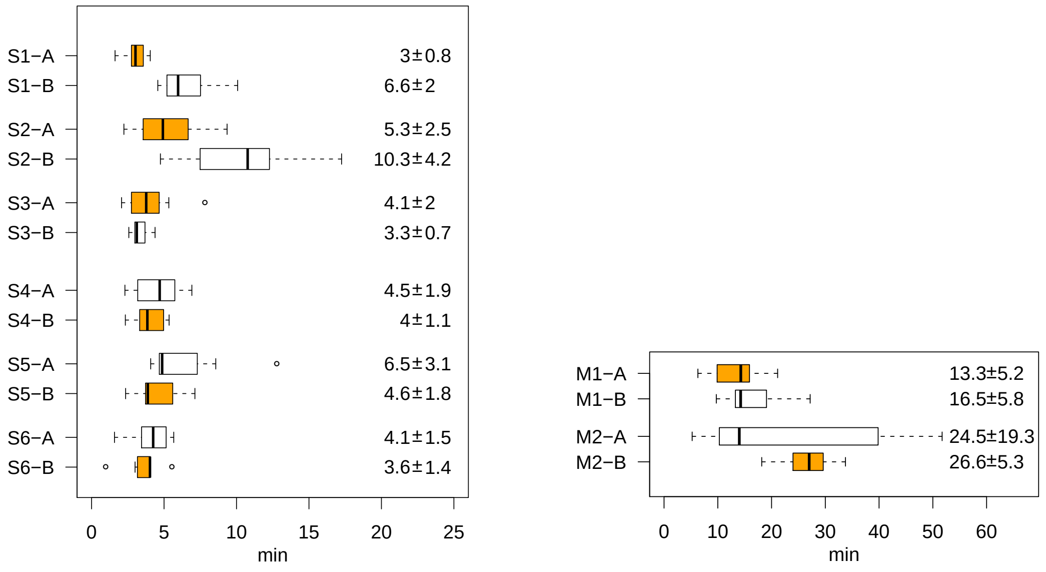


Fig. 4. Response time of subjects in minutes. Coloured boxes indicate that the according group worked with the colour version. The numbers indicate mean and standard deviation.

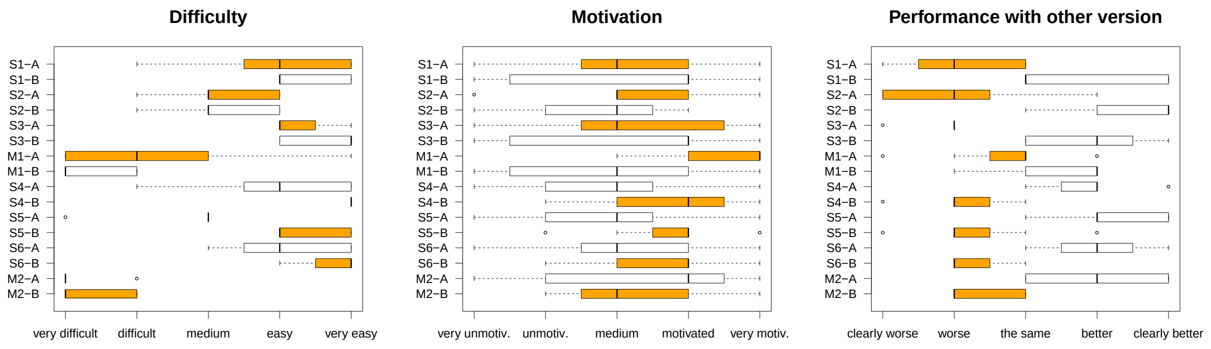


Fig. 5. Box plots of subjects' opinion. Coloured boxes indicate that according subjects worked with the colour version.