Program Comprehension of Feature-Oriented Software Development

Janet Feigenspan
University of Magdeburg, Germany
feigensp@ovgu.de

***Abstract*—Feature-oriented software development is a promising paradigm to implement variable software. One advantage is that crosscutting concerns can be modularized, which in turn has a positive effect on program comprehension. However, benefits for program comprehension are mostly based on plausibility arguments and theoretical discussions. This is not sufficient, since program comprehension is an internal cognitive process that requires empirical evaluation. Up to today, there are only few empirical studies that evaluate the effect of feature-oriented software development on program comprehension. With our work, we aim at filling this gap and providing sound empirical evidence about the effect of feature-oriented software development on program comprehension.**

***Keywords*-program comprehension, feature-oriented software development, variable software**

I. Introduction

Since the development of the first programmable computers, several software development approaches have been introduced, including assembler languages [45], procedural programming languages such as Ada [8] and Fortran [3], and *Object-Oriented Programming (OOP)* [36]. Today, OOP is the state of the art programming paradigm [36]. Several experiments proved the positive effect of OOP on program comprehension (e.g., [11], [27]).

However, new requirements of contemporary software products exceed the limits of OOP: Today, variability of software products is crucial for successful software development [41]. One mechanism to provide the required variability are Software Product Lines (SPLs), which are inspired by product lines in industry, like used in the production of a car or a meal at some fast food restaurant [41]. In order to implement SPLs, several approaches can be applied, for example *Aspect-Oriented Programming (AOP)*, *Feature-Oriented Programming (FOP)*, *C/C++ Preprocessor (CPP)*, and *Colored Integrated Development Environment (CIDE)*, which some researchers summarize as *Feature-Oriented Software Development (FOSD)* approaches [1]. In this context, a feature is a user-visible characteristic of a software system [1]. Source code that implements a feature is called *feature code*, whereas source code that implements commonalities of an SPL is called *base code*. Base code is present in each product or *variant* that is derived from an SPL; different variants contain different selections of features, such that not each feature is contained in each variant. This way, different products can be created by just composing them from existing source code.

Realizing variability in software reduces time and cost of the software-development process. However, at the same time it also introduces new challenges to program comprehension, because now, programs are not developed for a fixed set of requirements, but for several requirements, such that different products composed from the same SPL can fulfill different requirements [41].

Although issues like scalability and design stability of FOSD approaches have been evaluated [4], [19], understandability was mostly neglected. Despite theoretical discussions about effects of FOSD approaches on understandability and anecdotal experience [2], [29], [34], [37], sound empirical evaluations beyond anecdotal results from case studies are missing. It is important to gather empirical evidence regarding program comprehension to identify whether FOSD approaches, compared to contemporary programming techniques, provide benefits on program comprehension and under which circumstances. This would allow us to focus on those approaches that actually benefit program comprehension and to enhance them further. This way, we can reduce software-development costs.

Our primary objective we address with our thesis is to evaluate how FOSD influences different human factors, such as program comprehension. However, this is not as straightforward as one might think [13]. To understand how difficult it is to measure program comprehension, we give a short overview of program comprehension in the next section. Then, we describe an experiment in which we analyzed whether software measure can be used as comprehensibility predictors (Section III). In Section IV, we present ongoing work on how we can reduce the effort of designing and conducting program-comprehension experiments. Then, we describe three experiments, in which we analyzed how background colors can improve program comprehension of the C/C++ preprocessor (Section V). In Section VI, we discuss how we plan to evaluate whether modularization of (crosscutting) concerns improves program comprehension. Sections III to VI constitute the intended core content of our thesis. Finally, we present related work in Section VII and conclude our paper in Section VIII.

II. Program Comprehension

To have a better understanding of program comprehension and how it can be measured, we give an introduction to program-comprehension models. We can classify them into

three categories: top-down models, bottom-up models, and integrated models.

Top-down comprehension is the process of deriving and refining hypotheses of the purpose of a program. First, a developer derives a general hypothesis about a program's purpose while neglecting details. During this step, *beacons* (i.e., parts of source code that indicate occurrences of certain structures or operations [6]) help to determine the purpose of a program. Once a hypothesis is formulated, the developer evaluates it by looking at details and refining her hypothesis stepwise by developing and refining subsidiary hypotheses. To be able to determine the purpose of a program top down, the programmer has to be familiar with the domain of a program. Several examples of top-down models can be found in the literature [6], [46], [49].

Bottom-up comprehension describes how a program is understood when a programmer has no knowledge of a program's domain. In this case, a programmer examines statements of a program and groups them into semantic chunks. These chunks can be combined further until the developer has an understanding of the general purpose of a program. Several examples of bottom-up models can be found in the literature [40], [48].

Integrated models combine top-down and bottom-up program comprehension. For example, if a programmer has some knowledge about a domain, she starts with top-down comprehension. When she encounters code fragments she cannot explain using her domain knowledge, she switches to bottom-up comprehension. Typically, a developer uses top-down comprehension when possible, because it is more efficient. An example of an integrated model is described by Mayrhauser and Vans [52].

Program comprehension is an internal cognitive process that we cannot observe directly. Additionally, a programmer usually uses different comprehension models, depending on her familiarity with source code and domain. So, how can we measure program comprehension? Dunsmore and Roper summarized several indicators that can be used to measure program comprehension [12]. For example, the correctness of a bug fix or the time it took a developer to fix a bug: If a developer fixes a bug, she has to understand the faulty source code. Otherwise, she cannot succeed. Hence, we have to carefully define how we can measure program comprehension. Next, we evaluate whether software measures can be used.

III. Using Software Measures to Assess Program Comprehension

Software measures, such as lines of code or complexity, are often used to assess several facets of a software system. For example, there is a huge line of work that compares AOP with OOP based on software measures [7], [19], [22], [24], [31], [38]. Software measures are based on properties of source code, which is both a benefit and a shortcoming. The benefit is that software measures are a cheap and convenient way to assess comprehensibility. The shortcoming is that the relationship of software measures and program comprehension is not easy to describe, because program comprehension is an internal cognitive process, which cannot be simply measured by properties of source code. Since the focus of our thesis lies on program comprehension, we analyzed whether we can use software measures to assess program comprehension. This way, we could avoid conducting time-consuming experiments with human subjects.

To evaluate whether and how software measures correlate with program comprehension, we conducted a controlled experiment. We used MobileMedia, a software product line for the manipulation of multi-media data on mobile devices [19]. MobileMedia was implemented in two versions in eight scenarios, while with every scenario, the program was extended. One version was implemented in AspectJ, the other in Java ME with a preprocessor, Antenna. Both version were carefully designed and evolved with the goal to be comparable (see [19] for details). Based on software measures, both versions were compared and a superiority of the AspectJ version in terms of software measures was shown. For our experiments, we concentrated on the four measures lines of code, complexity [35], concern operations, and concern attributes [21].

As subjects, we recruited 21 students, which we split in two groups; one group worked with the AspectJ version, the other with the Java version. To measure program comprehension, we carefully introduced six bugs to MobileMedia and asked our subjects to locate them. Now, both versions of MobileMedia considerably differ in software measures, such that the AspectJ version suggests better comprehensibility. If software measures and program comprehension indeed have a relationship, then we should observe a difference in program comprehension between both groups in favor of the AspectJ version.

Our results, however, showed no significant difference in program comprehension, measured in terms of correctness and response time for the tasks. The results indicate that software measures and program comprehension have no relationship (if they had, AspectJ subjects should have made less errors and/or be faster). Details of this experiment can be found in [14].

IV. Methodology

Since we found that software measures cannot be used to assess program comprehension, there is no way around controlled experiments with human subjects. During planning our first experiments, we reached the limits of common empirical research techniques to measure program comprehension. To succeed with our experiments, we developed our own tools and instruments, so that we can reliably measure program comprehension. This way, we can extend

contemporary methodologies for empirical research in software engineering and provide a common framework for researchers to conduct program-comprehension experiments. We are currently refining the tools and instruments. Specifically, we are working on, a list of confounding parameters for program comprehension, a questionnaire to measure programming experience, and an extensible, customizable tool infrastructure for conducting experiments.

First, by providing an overview of confounding parameters, we aim at facilitating the design phase of experiments: The most important part in experiments is to define the variables that should be measured. The dependent variable in our context is program comprehension, and the independent variables are FOSD approaches or facets of them. Now, the problem is that the dependent variable is not only influenced by the independent variable, but also by confounding variables. However, we do not want that program comprehension is influenced by other facets than those we are interested in. Hence, we have to control the influence of confounding parameters. To control them, we have to identify them. With our work, we plan to provide an exhaustive list of confounding parameters. When a researcher is designing an experiment to measure program comprehension, she can simply consult our list and select suitable mechanisms to control the influence of confounding parameters.

To create the list, we are currently analyzing the last ten years of Empirical Software Engineering (ESE), International Conference on Program Comprehension (ICPC), and International Conference on Software Engineering (ICSE) for program-comprehension experiments. To extract the papers, we use a systematic approach as suggested by Kitchenham [30]: We first read the abstract to find out whether an experiment was conducted. If it was, the paper was added to our list. If it is not conclusive from the abstract, we use keywords associated with program-comprehension experiments (program comprehension, programming experience, expert, subject, participant). Based on this analysis, we either add a paper to the selection or dismiss it. We carefully read each selected paper for confounding parameters, how they were identified, and how they were controlled. At this point, we like to have advice on the selection of the journal (ESE) and conferences (ICPC, ICSE) regarding their suitability and whether further conferences and/or journals are required to have a representative selection of papers.

Second, we are developing a questionnaire to measure programming experience, which we found to be the most important confounding parameter based on an expert survey (cf. [13]). So far, we conducted an experiment with over 100 second year students of computer science at the Universities of Passau, Marburg, and Magdeburg. We gave our subjects tasks and compared the correctness and response time with their answers in a questionnaire. The questionnaire is based on self estimation of subjects and contained questions as *How experienced are you with Java/C/Haskell?* (on a five-point Likert scale [32]) or *For how many years have you been programming?* Currently, we are analyzing different ways of summarizing the answers to questions to a single value that indicates programming experience. Preliminary results show that the programming-experience value correlates with the number of correct answers in the experiment. Furthermore, for certain kind of task, there is also a relationship with response time. Although the results are still preliminary, we are sure to have a reliable programming-experience questionnaire in the near future.

Third, we have developed a tool infrastructure to conduct program-comprehension experiments called PROPHET.[1] We define an experiment's description, including tasks and source-code files to be opened. Furthermore, we can simply define whether we want syntax highlighting, whether source code can be modified by subjects, whether we display line numbers in source-code files, and several other options. Additionally, PROPHET provides a mail feature that zips all saved data of subjects and sends them to a specified mail address. An experiment is stored as XML-file, and can simply be reused, for example for replicating the experiment. In addition, if a certain feature for an experiment is not yet present in PROPHET, it can be easily implemented and plugged in without having to change the current source code of PROPHET. This way, we can support numerous program-comprehension experiments, as we could show based on a systematic literature review of the last five years of papers in ESE, ICPC, and ICSE. We plan to submit a short paper about PROPHET to ESEM this year.

V. Tool Support in FOSD

FOSD approaches usually use two mechanisms to implement software: Either tool support is used or language mechanisms. In a first series of experiments, we have concentrated on whether and how tool support can overcome the shortcomings of contemporary OOP techniques.

Specifically, we focused on how background colors can improve the understandability preprocessor directives. The problem with preprocessor directives is that they are tangled with the source code, which makes them difficult to detect (see Figure 1, left). Some researchers consider them "harmful" [51] or even as "hell" [33]. To make preprocessor directives more visible to a developer, we evaluated how highlighting them with background colors can help. In Figure 1, we give an example. On the left side, we see how preprocessor directives are used. On the right side, we can see how they are highlighted with colors. We decided to use colors, because humans process them preattentively and, thus, considerably faster than text [23]. Additionally, background colors clearly differ from source code. Hence, a developer can spot at first sight whether a code fragment is annotated

[1] PROPHET was implemented and evaluated as part of an internship and bachelor's thesis; http://fosd.net/prophet

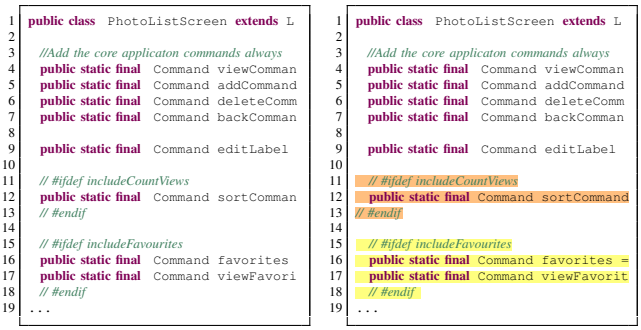

Figure 1. Comparison of *#ifdef* directives (left) and colors (right). In the colored version, Lines 11 to 13 are annotated with orange background color, Lines 15 to 18 with yellow.

with a preprocessor directive. This way, we can support a developer in quickly getting an overview of a software system.

We conducted three experiments, starting with a general evaluation whether background colors can help at all (Section V-A). In the subsequent experiments, we analyzed whether subjects use background colors when given the choice (Section V-B), and whether the use of background colors scales to large software systems (Section V-C).

A. Background Colors and Program Comprehension

In a first study, we evaluated whether background colors can improve program comprehension. For our experiment, we used MobileMedia [19], specifically the resulting product of the fifth development step of the Java version. From this version, we created a second version, in which we deleted the preprocessor statements and annotated the according code fragments with background colors.

We designed six tasks of two kinds, two static and four maintenance tasks. In static tasks, subjects should locate feature code, which was highlighted with a background color (one color per feature). In maintenance tasks, subjects should locate and identify the cause of bugs that were carefully introduced into feature code. Our sample consisted of 50 subjects, which we split in two comparable groups.

To measure program comprehension, we assessed whether a task was solved correctly and the response time of subjects. We found that background colors had no influence on the correctness of any task. For the response time, we found that in static tasks, the comprehension process was speeded up by up to 43 %. In maintenance tasks, we could not find a difference except for one task, such that subjects who worked with the color version were significantly slower. We suspect the reason for this slow down in the unsuitable background color of this feature, a saturated red, because some subjects complained about this color. Detailed information about the experiment can be found in [13], [15].

B. Background Colors and Program Comprehension – Subjects' Choice

The results of our first experiment, especially the comments of our subjects, indicated that developers should be able to choose between background colors and preprocessor statements as needed. Hence, we conducted a follow-up study in which subjects could choose between both kinds of annotations. Our sample consisted of 10 subjects, and we replicated the experiment with 10 different subjects a year later. To evaluate whether subjects switch between annotations, we implemented our own tool infrastructure, in which we logged everything a subject did. We used the same source code and tasks as for our first experiment (with a different ordering).

Based on the results of our first experiment, we expected that subjects use background colors to locate feature code, and preprocessor statements to locate bugs. However, we observed that subjects tended to switch less and less between both kinds of annotation with ongoing time of the experiment. Especially for the task, in which the unpleasant background color was displayed, surprisingly only three of ten subjects (four in the replication) used the preprocessor statements. We even observed that subjects moved closer to the screen to look at the source code. We explicitly did not remind subjects to switch to preprocessors, so that we could observe how subjects behaved. Otherwise, we could have instructed subjects to behave like we expected, which would have biased our results.

C. Background Colors and Program Comprehension – Scalability

An interesting point when using background colors is their scalability: The version of MobileMedia consisted of only four features, to which a one-to-one mapping of colors to features is feasible. But what about typical industrial-sized software systems with several hundred features? To address scalability, we conducted an experiment with a large software system, Xenomai. It is a real-time extension for the operating system Linux, implemented in C. It consists of more than 145,000 lines of code, of which over 30,000 lines are feature code, and about 350 features.

To deal with the large number of features, a one-to-one mapping of features and colors is not feasible. Instead, we define a default setting, in which we assigned two alternating shades of gray to feature code. As in the first experiment, we designed tasks to assess program comprehension. Since we showed that colors have no influence on program comprehension in maintenance task (if chosen carefully), we focused on static tasks. For each task, a certain set of features was relevant. Now, we assigned colors to those features, such that the colors were clearly distinguishable [44]. We used a tool infrastructure to realize the assignment of background colors, called FeatureCommander (available at http://www.fosd.de/fc). In Figure 2, we show a screenshot to

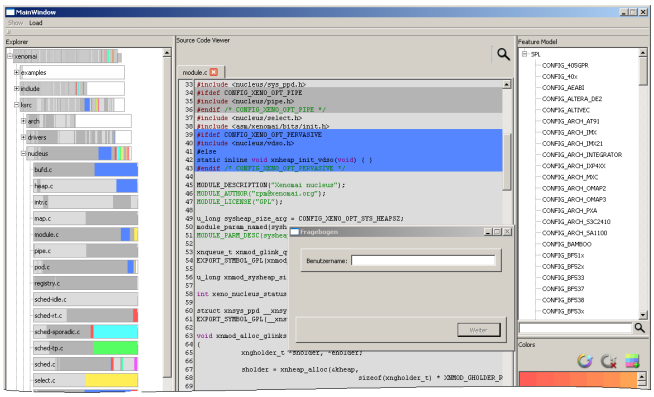

Figure 2. Screen shot of the color version of FeatureCommander. The small pop-up window displayed tasks and collected answers of subjects.

give a better impression of the tool. To evaluate the benefit of colors, we created another version of this tool without background colors.

We recruited subjects who where experienced with C and operating systems. We split our sample in two groups (A and B) of seven subjects each. The experiment was conducted in two phases, which differed in the tool versions and tasks. In the first phase, group A worked with the color version and group B without colors, while both groups worked on the same tasks. In the second phase, we switched the versions and gave another set of tasks to both groups.

We found that in the first phase, group A was faster than group B for some tasks. In the second phase, we did not find a significant difference in response time. However, we found that subjects of group B were faster in the second phase. This indicates that background colors can help to familiarize with a large system with several hundred features. More details about this experiment can be found in [17].

With those experiments, we feel that we have thoroughly analyzed how background colors influence program comprehension. However, at this point we like to discuss and have advice on whether we should focus further on the effect of background colors on program comprehension to have even more exhaustive data, or whether it is better for the big picture to now start working on different facets, such as modularization, as we introduce next.

VI. Language Mechanisms

To evaluate how language mechanisms are used to implement software and how they influence program comprehension, we are currently planning an experiment. Specifically, we aim at analyzing how modularization of (crosscutting) concerns influences program comprehension. We selected modularization, because this is an often claimed benefit for program comprehension of FOSD approaches in comparison to OOP. With our experiment, we plan to provide first empirical insights on whether modularization indeed improves program comprehension.

VII. Related Work

There is an interesting body of work that is related to ours. For better overview, we divide them into categories: We start with work regarding measuring program comprehension. Next, we describe work on background colors and program comprehension. We conclude with some work regarding the influence of source-code properties on program comprehension.

A. Measuring Program Comprehension

Program comprehension and its measurement is an issue in research at least since the late 1970s. Boysen [5] conducted a series of experiments to measure program comprehension and its relation to reading comprehension. For example, he assessed the effect of logical connectives of sentences, such as *and* and *or*. The results indicated that sentences connected with *and* are better understood than sentences connected with *or*. This can be applied to statements in source code, in the sense that statements connected with *and* should be preferred to statements connected with *or*, because programmers can understand them better. He also showed that software measures are not sufficient as comprehension measure, because they focus only on one facet of a program. Hence, they should only be used as program-comprehension measures in conjunction with empirical assessment of program comprehension.

Today, several methods exist to measure program comprehension, for example think-aloud protocols [50] or tasks that can only be solved if a programmer understands a program (see [18] for an overview of program-comprehension-measures). Using think-aloud protocols, subjects are instructed to speak out their thoughts during an experiment. Ensuring that subjects behave appropriately and analyzing the protocol data requires a lot of effort. Hence, tasks are often used to measure program comprehension. Typical kinds of tasks include static tasks (e.g., examining structure of source code), dynamic tasks (e.g., examining control flow), and maintenance tasks (e.g., fixing a bug) [12].

A closely related work illustrates how tasks can be used to measure program comprehension. Hanenberg et al. [25] evaluate empirically the benefit of AOP compared to OOP. 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.

B. Colors and Program Comprehension

Regarding colors, there is some work on using colors for various tasks, such as highlighting source code according to semantic of statements or control structure [43], error

reporting [39], or merging [53]. In prior work, we developed CIDE to highlight feature code [28], [16]. Closest to our representation with background colors are the model editors *fmp2rsm* [10] and *FeatureMapper* [26], in which model elements can be annotated and removed to generate different model variants. Both tools provide views and additionally can represent some or all annotations with colors. Furthermore, *Spotlight* [9] addresses scattered concerns (outside of the context of SPLs). Spotlight uses vertical bars in the left margin of the editor to visualize annotations. Bars of different colors are placed next to each other. Compared to background colors, lines are more subtle and can represent nesting easily. In all cases, the impact of visualizing annotations was not measured empirically so far.

C. Source-Code Properties and Program Comprehension

There are several studies, in which the influence of properties of source code on program comprehension is evaluated. For example, Prechelt et al. [42] evaluated how comments influence program comprehension of source code. In their setting, they had two versions of source code that only differed in the comments. The source code implemented several design patterns. One version of the comments named the design pattern the source code was implementing, whereas the other did not. The result is that naming the design pattern saved time and reduced errors.

In a different study, Sharif and Maletic [47] evaluated the effect of camelCase and under_score identifier style on program comprehension. In this setting, subjects should locate a correct identifier among several others, which were all presented on one screen. To analyze the eye movements of subjects, an eye tracking system was used. The authors found that under_score identifiers could be located faster than camelCase identifiers.

There is a lot of work on how source code properties can be captured in terms of software measures, such as lines of code and complexity. Those software measures are then used to assess the comprehensibility of a program without consulting subjects. One line of research develops and tests measures especially for aspect-oriented software development [19], [20], [24], [38]. In this work, several software projects are evaluated with the developed software measures. For example, Figueiredo et al. [19] assessed the design stability of aspect-oriented systems based on software measures. As example systems, they compared the two versions of MobileMedia (cf. Section III). Based on software measures, both versions were compared and a superiority of the AspectJ version was shown. However, the authors did not work with real subjects.

VIII. Conclusion

Feature-oriented software development is a promising paradigm to solve the limits of contemporary object-oriented programming techniques. However, claimed benefits of FOSD on human factors are based on plausibility arguments or anecdotal evidence. With our work, we aim at closing this gap. Our general research hypothesis is to evaluate the effects of FOSD on program comprehension. Since this is a complex topic, we defined three specific research questions for our thesis:

- How can we reduce the effort of planning and conducting program comprehension experiments?
- How does tool support improve program comprehension?
- How do language mechanisms help to improve program comprehension?

By systematically addressing these research questions, we contribute to our major goal to evaluate FOSD approaches. Furthermore, we provide other research groups with a methodology to design and conduct experiments. We also provide examples of how this methodology can be used.

Acknowledgment

We like to thank Christian Kästner from the University of Marburg, Sven Apel and Jörg Liebig from the University of Passau, as well as Michael Schulze and Raimund Dachselt from the University of Magdeburg for their support in planning and conducting the experiments.

References

[1] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(4):1–36, 2009.

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

[3] J. Backus, R. Beeber, S. Best, R. Goldberg, L. Haibt, H. Herrick, R. Nelson, D. Sayre, P. Sheridan, Stern, I. Ziller, R. Hughes, and R. Nutt. The FORTRAN Automatic Coding System. In *IRE-AIEE-ACM '57: Western Joint Computer Conference: Techniques for Reliability*, pages 188–198. ACM Press, 1957.

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

[5] J. Boysen. *Factors Affecting Computer Program Comprehension*. PhD thesis, Iowa State University, 1977.

[6] R. Brooks. Using a Behavioral Theory of Program Comprehension in Software Engineering. In *Proc. Int'l Conf. Software Engineering (ICSE)*, pages 196–201. IEEE CS, 1978.

[7] A. Bryant, C. Sant'Anna, E. Figueiredo, A. Garcia, T. Batista, and C. Lucena. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 109–121. ACM Press, 2006.

[8] W. Carlson, L. Druffel, D. Fisher, and W. Whitaker. Introducing Ada. In *ACM '80: Proceedings of the ACM 1980 Annual Conference*, pages 263–271. ACM Press, 1980.

[9] 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.

[10] 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.

[11] J. Daly, A. Brooks, J. Miller, M. Roper, and M. I. Wood. The Effect of Inheritance on the Maintainability of Object-Oriented Software: An Empirical Study. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 20–29. IEEE CS, 1995.

[12] 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.

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

[14] J. Feigenspan, S. Apel, J. Liebig, and C. Kästner. Exploring Software Measures to Assess Program Comprehension. In *Proc. Int'l Symposium Empirical Software Engineering and Measurement (ESEM)*, page To Appear, 2011.

[15] 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*, pages 55–62. ACM Press, 2009.

[16] 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. Demo Paper.

[17] 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)*. Institution of Engineering and Technology, 2011. To Appear.

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

[19] 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.

[20] 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.

[21] E. Figueiredo, J. Whittle, and A. Garcia. ConcernMorph: Metrics-based Detection of Crosscutting Patterns. In *Proc. Europ. Software Engineering Conf./Foundations of Software Engineering (ESEC/FSE)*, pages 299–300. ACM Press, 2009.

[22] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *Proc. Int'l Conf. Aspect-Oriented Software Development (AOSD)*, pages 3–14. ACM Press, 2005.

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

[24] 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.

[25] 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.

[26] 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. Demo Paper.

[27] S. Henry, M. Humphrey, and J. Lewis. Evaluation of the Maintainability of Object-Oriented Software. In *IEEE Region 10 Conf. Computer and Comm. Systems*, pages 404–409. IEEE CS, 1990.

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

[29] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proc. Int'l Software Product Line Conference (SPLC)*, pages 223–232. IEEE CS, 2007.

[30] B. Kitchenham. Procedures for performing systematic reviews. Technical Report TR/SE-0401, 0400011T.1, Keele University, National ICT Australia, 2004.

[31] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena. Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study. In *Proc. Int'l Conf. Software Maintenance (ICSM)*, pages 223–233. IEEE CS, 2006.

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

[33] 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.

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

[35] S. McConnell. *Code Complete*. Microsoft Press, second edition, 2004.

[36] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

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

[38] A. Molesini, A. Garcia, C. Chavez, and T. Batista. On the Quantitative Analysis of Architecture Stability in Aspectual Decompositions. In *Proc. Working IEEE/IFIP Conf. on Software Architecture (WICSA)*, pages 29–38. IEEE CS, 2008.

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

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

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

[42] L. Prechelt, B. Unger, M. Philippsen, and W. Tichy. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Trans. Softw. Eng.*, 28(6):595–606, 2002.

[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] D. Salomon. *Assemblers and Loaders*. Ellis Horwood, 1992.

[46] T. Shaft and I. Vessey. The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension. *Information Systems Research*, 6(3):286–299, 1995.

[47] B. Sharif and J. Maletic. An Eye Tracking Study on camelCase and under_score Identifier Styles. In *Proc. Int'l Conf. Program Comprehension (ICPC)*, pages 196–205. IEEE CS, 2010.

[48] B. Shneiderman and R. Mayer. Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. *International Journal of Parallel Programming*, 8(3):219–238, 1979.

[49] E. Soloway and K. Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Trans. Softw. Eng.*, 10(5):595–609, 1984.

[50] M. Someren, Y. Barnard, and J. Sandberg. *The Think Aloud Method: A Practical Guide to Modelling Cognitive Processes*. Academic Press, 1994.

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

[52] A. von Mayrhauser and A. Vans. From Program Comprehension to Tool Requirements for an Industrial Environment. In *Proc. Int'l Workshop Program Comprehension (IWPC)*, pages 78–86. IEEE CS, 1993.

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