How Explicit Feature Traces Did Not Impact Developers' Memory

Jacob Krüger*, Gül Çalıklı†, Thorsten Berger‡, and Thomas Leich§
*Otto-von-Guericke University Magdeburg, Germany †University of Zurich, Switzerland
‡Ruhr-University Bochum, Germany & Chalmers | University of Gothenburg, Sweden
§Harz University of Applied Sciences Wernigerode & METOP GmbH Magdeburg, Germany

***Abstract*—Software features are intuitive entities used to abstract and manage the functionalities of a software system, for instance, in product-line engineering and agile software development. Nonetheless, developers rarely make features explicit in code, which is why they have to perform costly program comprehension and particularly feature location to (re-)gain knowledge about the code. In a previous paper, we conducted an experiment on how explicit feature traces impact developers' program comprehension by facilitating feature location. We found that annotating features in code improved program comprehension, while decomposing them into classes had a negative impact. Additionally, but not reported in that paper, we were concerned with understanding whether the different traces would impact developers' memory regarding the code and its features. To this end, we repeatedly asked our participants questions about the code on different levels of detail within time periods of two weeks. Since developers' memory decays over time, we expected that our participants would provide fewer correct answers over time, with differences depending on the feature traces in their code. Unfortunately, the actual results were inconclusive and up for interpretation, particularly due to challenges in designing an experiment on developers' memory. In this paper, we discuss our experimental design, the null results, and challenges for improving the methodology of future studies in this direction.**
***Index Terms*—Feature orientation, traceability, human memory, program comprehension, psychology**

I. Introduction

The notion of software *features* is an intuitive abstraction to specify, document, and communicate the functionalities of a software system, and thus is established in practice despite missing a clear definition [4], [6], [8], [20]. Features are the primary concern of interest in engineering software product lines—large and complex software platforms [3], [29] that are usually highly configurable to derive individual variants. Features in this context are typically associated with configuring, taking the perspective that features are optional and represent configuration flags (e.g., as C preprocessor macros or feature toggles) [26], [34]. So, features are usually made explicit—they are identified, modeled [15], [27], mapped to implementation assets, and tested in various configurations [3], [19], [29]. In contrast, agile methods, such as SCRUM, XP, or FDD, use features to organize development and to represent primarily mandatory functionality, for example, in the product backlog that is essentially a list of features [24]. As such, features are usually not limited to the perspective of optional and configurable functionality, they can represent any functionality of a system. Still, while features are recorded, they are usually not made explicit in code, as opposed to the product-line perspective.

Adopting the broader perspective of features from agile methods, while making them explicit as in the product-line perspective, promises to facilitate program comprehension and extend automated analyses [2], [5], [9], [13], [16], [35]. To support this claim, researchers conducted empirical studies on different techniques for explicit feature traceability. Unfortunately, many of such studies investigate a single property of a specific traceability technique (e.g., configurability of preprocessor macros), but do not compare different techniques. For instance, studies show that developers have problems understanding the configurability of the C preprocessor [11], [22], [23], leading to more bugs and problems using configurator tools [1], [7]. Similarly, a few controlled experiments have been conducted to compare C preprocessor macros with background colors [10] or feature-oriented programming [30], [31]. However, such experiments focus again on techniques that tangle traceability and configurability, without considering missing or non-configurable traces.

To tackle these problems, we [17] conducted an experiment in which we compared three different variants of the same code example with each other: code (i) without traces, (ii) with feature annotations, and (iii) with decomposed features (see Section II for details). Our results showed that feature annotations had a positive impact on developers' comprehension of features, while a decomposition had negative impact on their ability to localize bugs. Additionally, our experiment included a second part on the impact of explicit feature traces on developers' memory that led to null results, and which we did not report, yet. In this paper, we focus on this second part by describing its setup and results (Section III) as well as by discussing challenges for improving future studies in this direction (Section IV). While our experiment led to null results, we argue that more studies on developers' knowledge and comprehension of features are needed to investigate the hypotheses we derive—namely, that developers can recall the features of a system and that feature traces do not impair developers' memory. This would support the claim that features are an intuitive notion for software engineering and that their locations should be traced.

II. Explicit Feature Traces

Design. In the first part of our experiment [17], we investigated whether explicit feature traces would facilitate developers' program comprehension. For this purpose, we used a class

of the Mobile Media system [36], which is regularly used as a subject system for such studies because it provides feature locations that have been annotated by the original developer. We used three different variants of that class:

i pure object-oriented code without any feature traces;
ii code with feature annotations in comments that replaced the originally used preprocessor macros; and
iii decomposed code, for which we modularized the code of a feature into a separate class, removing all annotations.

Consequently, none of the three variants comprised the configurability of the originally used preprocessor macros, untangling traceability and configurability. Moreover, we could compare feature traces (ii, iii) to missing traces (i).

We invited 144 personal contacts from various organizations and countries, and asked them to share the invitation with interested colleagues. For each participant, we first quantified their experience level to then assign them randomly to one of the three code variants, ensuring the same ratio of novices to experts for each variant. The actual experiment was unsupervised and online, for which we implemented, tested, and published our own infrastructure. In the first iteration of our experiment, our participants inspected the code to understand its behavior. To assess the impact of explicit feature traces on their comprehension, we asked three questions relating to the behavior of features, and defined three tasks in which the participants had to localize bugs. After this iteration, we collected each participant's subjective perception on feature traceability and the usefulness of the traces they worked on.

Results. We received 49 responses for this iteration of our experiment (e.g., 20 from Turkey, 13 from Germany, 7 from the USA) and found that the distribution of participants according to their experience was comparable. Moreover, we checked and removed responses that took too long or for which the participants stated interruptions. This way, we mitigated potential threats to validity of our study. In summary, we found:

- Annotations had a significant positive impact on our participants' comprehension of features and their interactions, while not impacting their ability to localize bugs.
- Decomposing features did not impact our participants' ability to comprehend features, but they performed significantly worse on localizing an interaction bug.
- None of the traces led to significant changes in our participants' efficiency (i.e., time spent on a task).
- Our participants had a positive perception of using explicit features traces, as long as they are trustworthy.

These results indicate that particularly annotations may be a good way to adopt feature traceability in practice. Besides these results, we were also concerned with understanding whether feature traces would impact developers' memory, for which we extended our experiment as described in the following.

III. The Memory Experiment

For the extended experiment, our idea was to identify (1) indicators regarding whether developers can remember a feature's functionality (even though its implementation and behavior may change), and (2) whether feature traceability could facilitate not only program comprehension, but also remembering. To this end, we contacted our participants again to ask questions about the code they worked on, building on previous works on developers' memory [14], [21]. In the following, we discuss the details of this extension.[1]

Design. After the first iteration (cf. Section II), we sent mails in three more iterations (each two weeks after a participant's last response) to ask our participants to fill in a survey with comprehension questions about the code (without showing the code). To investigate whether our participants could correctly remember the code, we designed three multiple-choice questions on code parts they worked on. These questions were concerned with (number of available / correct answers in brackets):

- Details of one specific feature and its interactions (4 / 3);
- Types of bugs that existed in the code (5 / 2); and
- Causes for these bugs, such as feature interactions (4 / 1).

In total, a participant could achieve 13 points in each iteration, and we reduced these points by one whenever (i) a wrong answer was selected or (ii) a correct answer was not selected. We did not check whether our participants correctly understood the answers to these questions in the first iteration (when they could analyze the code)—since we were more interested in the evolution of their responses (i.e., changes in the third and fourth iteration), indicating changing memory.

Unfortunately, but not surprisingly, of the 49 participants who participated in the first iteration of our experiment, few responded to the following iterations. In fact, only 19 participants participated in the second iteration (seven for annotations, five for modules, and seven for no traces). This dropped even more in the remaining iterations, and only eight participants responded to all iterations.

Results. In Figure 1, we display the correctness of our participants during the last three iterations of our experiment. For each data point, we show how many days after the previous iteration the questions were answered. Data points that represent the same participant are connected with lines. Due to the small number of participants, we cannot conduct statistical tests, since these would be unreliable. Moreover, we cannot observe a systematic difference between the code variants, essentially meaning that we obtained null results regarding our actual goal. Nonetheless, we derive two hypotheses that we argue are important to *properly confirm or reject in future studies* to better understand the usability of features and feature traces in software engineering, as well as their impact on developers' memory.

Hypothesis 1: Developers are good at remembering features. Related studies [14], [18], [21] indicate that developers can remember specific types of knowledge better. For example, developers forget methodological knowledge much quicker than domain concepts (which are usually expressed as features of a system). Since our questions were concerned with features and bugs, both representing code abstractions, we argue that developers should be able to remember such knowledge quite

[1] Replication package: https://doi.org/10.5281/zenodo.4417629

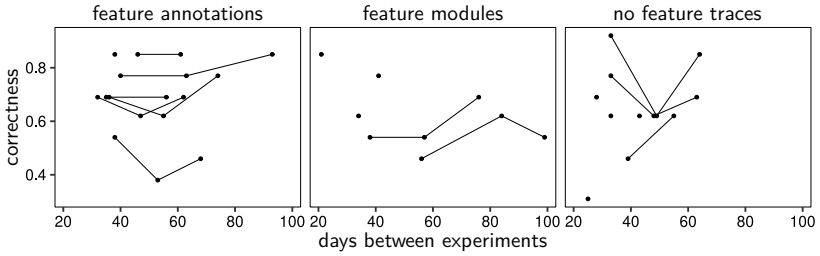

Fig. 1. Our participants' ability to answer questions based on their memory over time, separated by the feature traces used in the code.

well. In Figure 1, we can see only a few strong changes in the correctness of our participants' answers, with most data points being in the frame of 60 % to 80 %. Interestingly, increasing correctness was often connected to participants selecting fewer wrong answers. So, this hypothesis focuses on the indicators of previous studies that features are a suitable abstraction to guide software evolution, since developers may recall them better than code details.

Hypothesis 2: Feature traces do not impair developers' memory. We do not observe any substantial changes in the knowledge about features and bugs over time. While the patterns of the different groups differ, most participants performed similarly in the second and in the fourth iteration. Interestingly, many participants had a drop in correctness in the third iteration, which we cannot explain without further studies. In close relation with the first hypothesis, we hypothesize that feature traces do not impair developers' ability to remember features and bugs in code. So, explicit feature traces may not be helpful for remembering the behavior of a system, but can guide program comprehension and automated analyses.

IV. Lessons Learned

To support future studies on software evolution and its impact on developers, we report four closely related challenges we faced while designing our experiment.

Confounding Factors: Studying program comprehension and developers' memory is challenged by confounding factors and cognitive biases [28], [32]. Moreover, investigating developers' memory during software evolution becomes even more challenging, since the psychological aspects of memory are also involved. For instance, memory strength varies among different people, and some recall different types of knowledge better than others. Particularly, memories are not high fidelity recordings that we can store: Some code-related aspects a developer focused on during program comprehension will be central (i.e., will be remembered better), while others will remain as peripheral details. Central and peripheral aspects will not be the same for all developers. Moreover, all related code elements, including sensory elements serving as a primer for code-related information, are processed in different parts of the brain. All these different elements are pulled together by the medial temporal lobe. So, what a developer remembers is different from what another one remembers. *Therefore, a challenge for conducting studies on developers' memory is the identification and mitigation of confounding factors and cognitive biases.*

Measuring Memory: Measuring memory is challenging on its own [18], [21], for example, how to quantify what a developers still knows about a system? On what granularity should we ask questions? What factors or system properties impact developers' memory? Arguably, many studies are needed to properly understand developers' memory and the factors influencing it. In psychology, studies on forgetting often involve a single subject [25]. *Therefore, for software engineering and program comprehension—involving teams editing the same artifacts and automated code transformations, a challenge is how to properly design studies on developers' memory and its decay.*

Motivating Participants: As we can see in Figure 1, participants dropped out during our study. Most likely, this issue was connected to general time and motivational issues. Still, we cannot rule out that the dropout rate was influenced by the technique we used to trace features, seeing that we faced this problem particularly for feature modules. For the first iteration, we prepared program comprehension tasks that require deeper thought processes, which take place in developers' working memory. Moreover, the subject system is closer to industrial software rather than a small piece of code. As a result, the difficulty of the tasks may make participants reluctant to continue with the study. However, understanding the effects of feature traceability on program comprehension and memory requires a subject system that is more than a single piece of code and sophisticated questions. *Therefore, a particular challenge is how to design subject systems and tasks that allow to answer research questions, while keeping developers motivated to participate throughout a series of studies.*

Improving Validity: A regular question in empirical software engineering is whether and how to focus on internal or external validity [33]. Particularly considering the previous challenges, validity becomes an important and challenging concern on its own. Consider, for instance, the evolution of an artifact, such as a piece of code. To improve internal validity, we would need to ask the same questions about the code without anyone or any tool changing it. However, this immediately causes threats of developers memorizing particularly that artifact, impairing internal validity. Moreover, artifacts change all the time in practice, wherefore the external validity would be low. *Therefore, it is an open challenge to define study designs that would allow us to focus on, or balance between, internal and external validity.*

We remark that we report these four challenges, because we deemed them most important, arguing on how to improve experimental designs. Still, studies with human subjects and particularly on memory face numerous additional threats and challenges that must be considered.

V. Conclusion and Prospects

In this paper, we reported on the extension of a previous experiment in which we investigated the impact of explicit feature traces on developers' memory. Unfortunately, we obtained null results regarding our actual goal. However, we identified several challenges and ways to improve studies on developers' memory and the notion of software features. In this direction, we suggest the following prospects for future research:

- *Improving the Designs of Studies on Developers' Memory* to tackle the challenges we described in Section IV, and thus enable us to obtain actual, reliable data.
- *Analyzing Developers' Memory* to understand how they forget what knowledge, enabling us to improve automated techniques, for instance, for identifying experts [12].
- *Studying the Hypotheses* we discussed in Section III to investigate whether program comprehension and remembering are facilitated by using features and feature traces.
- *Comparing Notions of Features* to understand what different stakeholders (e.g., testers) consider important knowledge for a feature, defining what information to collect, manage, and provide.
- *Advancing Feature Traceability*, which is a decades-old problem that can be addressed with lightweight feature annotations and automated, knowledge-dependent analysis techniques on those.

Investigating these directions enables us to understand how developers comprehend and recall information, to define opportunities for managing software features, and to extend corresponding automation.

Acknowledgments. This research has been supported by the German Research Foundation (SA 465/49-3, LE 3382/2-3), the Swedish Research Council Vetenskapsrådet (257822902), and the Wallenberg Foundation.

References

[1] I. Abal, J. Melo, Ş. Stănciulescu, C. Brabrand, M. de Medeiros Ribeiro, and A. Wąsowski, "Variability Bugs in Highly Configurable Systems: A Qualitative Analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 3, 2018.

[2] B. Andam, A. Burger, T. Berger, and M. R. V. Chaudron, "FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces," in *VaMoS*. ACM, 2017.

[3] S. Apel, D. Batory, C. Kästner, and G. Saake, *Feature-Oriented Software Product Lines*. Springer, 2013.

[4] S. Apel and C. Kästner, "An Overview of Feature-Oriented Software Development." *Journal of Object Technology*, vol. 8, no. 5, 2009.

[5] T. Berger, "Feature-Oriented Traceability," in *Grand Challenges of Traceability: The Next Ten Years*. CoRR, 2017.

[6] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines," in *SPLC*. ACM, 2015.

[7] T. Berger, J.-P. Steghöfer, T. Ziadi, J. Robin, and J. Martinez, "The State of Adoption and the Challenges of Systematic Variability Management in Industry," *Empirical Software Engineering*, vol. 25, 2020.

[8] A. Classen, P. Heymans, and P.-Y. Schobbens, "What's in a Feature: A Requirements Engineering Perspective," in *FASE*. Springer, 2008.

[9] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software Traceability: Trends and Future Directions," in *FOSE*. ACM, 2014.

[10] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake, "Do Background Colors Improve Program Comprehension in the #ifdef Hell?" *Empirical Software Engineering*, vol. 18, no. 4, 2013.

[11] W. Fenske, J. Krüger, M. Kanyshkova, and S. Schulze, "#ifdef Directives and Program Comprehension: The Dilemma between Correctness and Preference," in *ICSME*. IEEE, 2020.

[12] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill, "A Degree-of-Knowledge Model to Capture Source Code Familiarity," in *ICSE*. ACM, 2010.

[13] W. Ji, T. Berger, M. Antkiewicz, and K. Czarnecki, "Maintaining Feature Traceability with Embedded Annotations," in *SPLC*. ACM, 2015.

[14] K. Kang and J. Hahn, "Learning and Forgetting Curves in Software Development: Does Type of Knowledge Matter?" in *ICIS*. AIS, 2009.

[15] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-Oriented Domain Analysis (FODA) Feasibility Study," Carnegie Mellon University, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[16] S. Krieter, J. Krüger, and T. Leich, "Don't Worry About it: Managing Variability On-The-Fly," in *VaMoS*. ACM, 2018.

[17] J. Krüger, G. Çalıklı, T. Berger, T. Leich, and G. Saake, "Effects of Explicit Feature Traceability on Program Comprehension," in *ESEC/FSE*. ACM, 2019.

[18] J. Krüger and R. Hebig, "What Developers (Care to) Recall: An Interview Survey on Smaller Systems," in *ICSME*. IEEE, 2020.

[19] J. Krüger, W. Mahmood, and T. Berger, "Promote-pl: A Round-Trip Engineering Process Model for Adopting and Evolving Product Lines," in *SPLC*. ACM, 2020.

[20] J. Krüger, M. Mukelabai, W. Gu, H. Shen, R. Hebig, and T. Berger, "Where is My Feature and What is it About? A Case Study on Recovering Feature Facets," *Journal of Systems and Software*, vol. 152, 2019.

[21] J. Krüger, J. Wiemann, W. Fenske, G. Saake, and T. Leich, "Do You Remember This Source Code?" in *ICSE*. ACM, 2018.

[22] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi, "An Empirical Study on Configuration-Related Issues: Investigating Undeclared and Unused Identifiers," in *GPCE*. ACM, 2015.

[23] J. Melo, C. Brabrand, and A. Wąsowski, "How Does the Degree of Variability Affect Bug Finding?" in *ICSE*. ACM, 2016.

[24] B. Meyer, *Agile!* Springer, 2014.

[25] J. M. J. Murre and J. Dros, "Replication and Analysis of Ebbinghaus' Forgetting Curve," *PLoS ONE*, vol. 10, no. 7, 2015.

[26] S. Nadi, T. Berger, C. Kastner, and K. Czarnecki, "Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study," *IEEE Transactions on Software Engineering*, vol. 41, no. 8, 2015.

[27] D. Nešić, J. Krüger, Ş. Stănciulescu, and T. Berger, "Principles of Feature Modeling," in *ESEC/FSE*. ACM, 2019.

[28] C. Parnin and S. Rugaber, "Programmer Information Needs after Memory Failure," in *ICPC*. IEEE, 2012.

[29] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering*. Springer, 2005.

[30] A. Rodrigues Santos, I. do Carmo Machado, E. Santana de Almeida, J. Siegmund, and S. Apel, "Comparing the Influence of Using Feature-Oriented Programming and Conditional Compilation on Comprehending Feature-Oriented Software," *Empirical Software Engineering*, vol. 24, no. 3, 2018.

[31] J. Siegmund, C. Kästner, J. Liebig, and S. Apel, "Comparing Program Comprehension of Physically and Virtually Separated Concerns," in *FOSD*. ACM, 2012.

[32] J. Siegmund and J. Schumann, "Confounding Parameters on Program Comprehension: A Literature Survey," *Empirical Software Engineering*, vol. 20, no. 4, 2015.

[33] J. Siegmund, N. Siegmund, and S. Apel, "Views on Internal and External Validity in Empirical Software Engineering," in *ICSE*. IEEE, 2015.

[34] R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat, "Feature Consistency in Compile-Time-Configurable System Software: Facing the Linux 10,000 Feature Problem," in *EuroSys*. ACM, 2011.

[35] T. Vale, E. S. de Almeida, V. R. Alves, U. Kulesza, N. Niu, and R. de Lima, "Software Product Lines Traceability: A Systematic Mapping Study," *Information and Software Technology*, vol. 84, 2017.

[36] T. J. Young, "Using AspectJ to Build a Software Product Line for Mobile Devices," Master's thesis, University of British Columbia, 2005.