# Visualizing Feature-Oriented Software Evolution

Kevin Hermann
Ruhr University Bochum
Germany

Johan Martinson
Ruhr University Bochum
Centiro Solutions
Germany, Sweden

Thorsten Berger
Ruhr University Bochum and
Chalmers | University of Gothenburg
Germany, Sweden

## Abstract

Software systems are becoming increasingly complex, and so does their evolution. Development processes often use the notion of features to plan and organize the development of software systems. While features facilitate understanding and communicating about software systems, the connection to code is often lost, challenging their evolution. Developers often need to recover the location of features in code, which is laborious and error-prone, especially for scattered features. The vision of feature-oriented software evolution advocates maintaining traceability between features and the codebase during development, which requires techniques to make features more comprehensible to stakeholders.

In this work, we show how traceability links can be utilized to lift the abstraction level of software assets to the feature level. We present a visualization for facilitating comprehension of software evolution by visualizing their evolution through a timeline, addressing a significant gap in realizing the vision of feature-oriented software evolution. Our visualization fosters system evolution comprehension, and improve feature documentation by providing developers with short term benefits of recording feature locations.

## CCS Concepts

• **Software and its engineering → Software product lines**; **Software maintenance tools**.

## Keywords

feature location, visualization, feature-oriented software evolution

**ACM Reference Format:**
Kevin Hermann, Johan Martinson, and Thorsten Berger. 2025. Visualizing Feature-Oriented Software Evolution. In *Proceedings of 29th International Systems and Software Product Line Conference (SPLC'25).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

## 1 Introduction

Whether to satisfy customer demand or to fix defects, such as security vulnerabilities, developers frequently add, maintain, or evolve features [9, 11, 17, 18] during software development. Features offer a way to facilitate the comprehension of software functionalities, which are often scattered over the codebase, or otherwise require extensive domain knowledge to understand them. As such, they aid stakeholders in understanding software without deep knowledge about the codebase. Feature-driven development processes use the notion of features to plan and organize the development process [4].

To facilitate system comprehension, it is therefore crucial to raise the level of abstraction to a level that all stakeholders involved in a software project understand. Features can be used as a common notion to facilitate system comprehension, since they can be leveraged to express software components at different levels of abstraction, ranging from low level technical details to high-level concepts.

Feature-oriented software evolution advocates establishing traceability links between features and a software's assets by making features explicit in code to enable analysis of its architecture, and the impact of changes [27]. Manually recovering traceability links after features have been implemented is laborious and error-prone, as they are often scattered over multiple assets [28]. Automatic techniques have shown to produce too many false positives and do not scale well in large systems to be usable in practice [2, 6, 7, 15, 29]. Traceability links can be established by relating code assets to their corresponding features during development through, for instance, embedded feature annotations, which developers integrate into their code when the location of the features is still fresh in their minds [31]. Opposed to manually recovering features from the source code, annotating assets during development has shown to be effective in saving feature location costs [16].

However, developers rarely document the location of features they are working on, and knowledge about them deteriorates over time. As knowledge about them fades, the effort to recover feature locations increases over time, especially when the developer responsible for a feature has left the organization. Even though the saved effort outweighs the effort in creating traceability information [16], the lack of short term benefits could be an impediment for establishing traceability links during development, since developers may not notice their advantages as they write code. Our long-term goal is to raise the level of abstraction at which software systems are managed, establishing features as an interface to software systems.

Our research objective is to *establish a feature evolution timeline that enables developers to reason about software evolution at the feature level, and provide them with short term benefits for recording feature locations during development.* We present a feature evolution timeline that visualizes software and its evolution at the feature level using feature traceability information, which developers create while they write code. It provides an overview to enable stakeholders to reason about features without knowledge about the codebase, addressing a significant gap in the realization of the vision of feature-oriented software evolution [27]. Finally, we evaluated the usability and effectiveness of our timeline by implementing it in a prototype and conducted a controlled experiment with users.

## 2 Background

We now present the prerequisites for feature visualizations.

### 2.1 Feature Traceability

Feature visualization requires feature traceability data to be in place. In this work, we employ embedded feature annotations [10, 16, 31], but our visualization works independently of the used feature traceability technique. Embedded feature annotations advocate recording feature locations while writing code, when the knowledge about the feature's location is still fresh in the developers mind. Code assets are mapped to their corresponding features by directly embedding them into the assets through a lightweight embedded feature annotations system. Documenting features first requires a feature model, which organizes features in a separate file based on their hierarchy. We expect the following information to be available to operate, however, not all of them must be present in the project:

**Folder mappings:** A folder is mapped to a feature.
**File mappings:** A file is mapped to a feature.
**Fragment mappings:** A code fragment is mapped to a feature.

Files and folders are mapped to features through textual files, while fragment mappings are integrated as comments within code. We utilize this traceability data to locate changes made to features.

### 2.2 Feature Visualizations

Previous work investigated numerous techniques for tracing and visualizing features [26, 30]. FeatureIDE is an Eclipse plugin that uses preprocesser directives to load configurations to enable or disable features in a system [24]. Furthermore, the Eclipse plugin Colligens maps C preprocesser directives to a feature model and shows the number of files and lines of code which implement a feature [23]. Moreover, FLOrIDA is a tool, which extracts traceability information from assets to visualize feature metrics such as feature size, scattering or tangling in different views [3]. Featuredashboard uses the same metrics to visualize relationships between features and assets as well as common features between projects [13]. FeatureVista provides tool support, that visualizes to what extent classes in the codebase contribute to a feature and how features interact with each other [8]. CIDE provides further support in highlighting feature-specific code and hiding code fragments for improved visibility on code [19]. Similarly, FeatureMapper provides a view that shows what assets realize a feature through a UML model [14].

Feature survival charts [32] provide an overview on when a feature was introduced or removed, but do not provide support for understanding the evolution history between these points of time [20, 27]. To this end, we developed a novel timeline to aid developers in understanding how features evolve over time.

## 3 Feature Evolution Timeline

As software systems expand, managing and evolving them grows increasingly complex. Therefore, research has focused on structuring software evolution around features [17, 27]. While feature traceability techniques facilitate feature location, developers still encounter challenges in understanding feature evolution. Traditional version control systems, such as Git, track changes at the file and code level, but not at the feature level. In the absence of tools

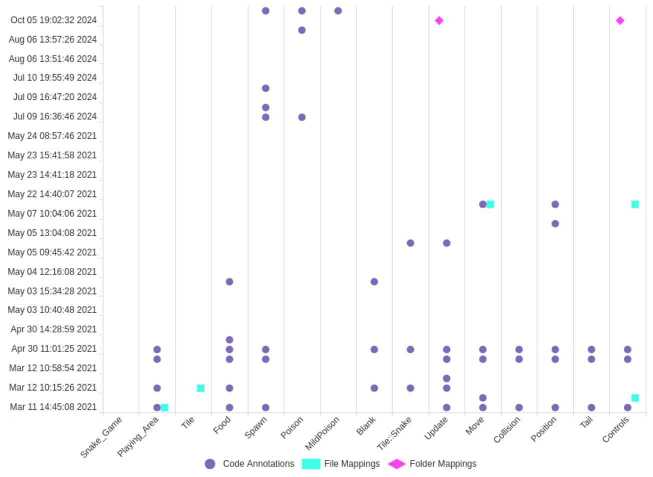


**Figure 1: Feature Evolution Timeline**

that aid developers with this task, they must manually track feature histories by navigating Git's command-line interface (CLI). This process, as we will show, is both time-consuming and error-prone, and highlights the need for visualizations that enable developers to understand the history of individual features.

### 3.1 Extracting Feature-Related Commits

To track the evolution of features, and subsequently visualize their evolution, we need to explicitly know a feature's location, and when it was added, modified or deleted. Therefore, we used Git to extract the traceability data from the commit history of a project to associate features with their corresponding files, folders, and commits. We extracted the feature location within commit histories, which provide insights into the addition, modification, and deletion of features. We utilized the Git repository data to extract metadata, such as commit timestamps, authors, and messages, as well as to categorize changes by type (e.g., code annotations or file mappings). This tracking ensures an understanding of feature evolution and the relationships between features and their implementation.

Deleted features are specifically interesting and presents challenges as it may not be obvious when a feature has been deleted. To track when a feature was deleted, we checked whether a feature was removed from the feature model and recorded their final commit details. All this information about the adding, changing, and deleting of features ensures that changes are not overlooked and that the evolution of features can be fully understood.

### 3.2 Visualizing Feature Evolution

We propose a feature evolution timeline which visualizes feature evolution across Git commits over the whole project history. The visualization (Fig. 1) enables developers to analyze the evolution of features using a timeline chart. The vertical axis represents the chronological order of commits, while the horizontal axis lists all extracted features. Each feature-commit pair is represented by a data point, which is color-coded and annotated to indicate the type of change, such as changes at the code, file, or folder level. The view is interactive, allowing developers to filter data by time ranges, features, or types of annotations. When hovering over a data point, commit metadata such as the hash, author, timestamp, and message is displayed. The view provides a clear and structured representation of feature changes over time, enabling developers to track the introduction, modification, and removal of features.

Additionally, the *Deleted Features View* shows features that have been removed from the project, displaying the commit in which they were last present. This functionality supports tracing the history of removed features and understanding their removal, making it particularly useful for new developers by providing a clear and concise way to navigate historical changes in the repository.

## 3.3 Evaluation

We conducted a controlled experiment with users to investigate how the comprehension of feature evolution can be assisted by using a timeline compared to using traditional Git CLI. We implemented our timeline in HAnS-viz, an IntelliJ plugin for feature visualization [21]. All data related to our experiment is publicly available [1].

**Research Question.** To investigate the usability of using our timeline for understanding feature evolution related tasks, our controlled experiment investigates the following research question:

**RQ:** *How usable is the feature evolution timeline for understanding feature evolution compared to traditional Git CLI?*

**Hypotheses.** We investigate the following hypothesis.

**$H_\alpha$:** A feature evolution timeline is more usable compared to traditional Git CLI for understanding feature evolution.

We formulate our corresponding null hypothesis as follows.

**$H_0$:** A feature evolution timeline is equally usable compared to traditional Git CLI for understanding feature evolution.

**Variables.** The goal of our controlled experiment is to measure the task completion times, and usability of the feature evolution timeline and Git CLI for understanding feature evolution. Therefore, there is one independent variable, which is the modality used to complete tasks related to understanding feature evolution. In our setting, it has two possible values: the feature evolution timeline and Git CLI. The dependent variables are the task correctness, completion time, and the usability of the use modality.

**Participants.** We recruited 14 students from our institution's faculty of computer science for our experiment. Following a within-subjects design, we randomly assigned them to two groups, with group A consisting of 7 participants (4 graduate, 2 undergraduate, 1 PhD), and group B as well (3 graduate, 3 undergraduate, 1 PhD). All participants rated their experience with programming and Git on a 5-point Likert-scale prior to the experiment. They expressed an average programming experience level of 3.5 (mean) ± 0.8 (standard deviation), and an average experience level with Git of 3.4 ± 1.0.

**Experiment Design.** Group A first solved the tasks using the Git CLI, before solving the tasks using the feature evolution timeline. For Group B, we changed the order by asking participants to first solve the tasks using the feature evolution timeline, and then using the Git CLI. Employing two groups minimizes period effects (e.g., fatigue) and carryover effects (e.g., changing opinions about a tool) to affect our results. We provided participants with a repository containing the installation files and a guide for the feature evolution timeline along with introductory documents explaining their concepts and functionalities. Then, we asked participants to download the repository of our subject system, a small Snake game, which contains around 300 lines of fully annotated java code in 8 files and 3 folders with 17 features, and a commit history of 71 commits. While it is small in size, it provides a suitable commit history for a controlled experiment. A large commit history could quickly overwhelm participants that use the Git CLI, which could impact the results. After training, we asked participants to perform warm-up tasks using both modalites to familiarize themselves with them, to ensure that they are properly trained.

Participation in the experiment was voluntary, and did not involve any advantage or disadvantage to participants. The circumstances of the experiment were explained to participants prior to their agreement to participate. If participants were unable to solve a task, or a task took longer than 10 minutes to solve, we asked them to skip it. All participants were assured that their data would only be published in aggregated or anonymized form.

**Tasks.** We asked participants to complete 6 tasks:

*1) Understanding evolution of a feature.* We asked participants to identify the commit hash, in which a feature was first added. We asked participants to solve this task once for fragment annotations (Task 1), file annotations (Task 2), and folder annotations (Task 3). For the fragment annotation task, we additionally asked them to enter the number of commits that performed changes to it.

*2) Understanding feature change impact.* We asked participants to identify the commit hash, in which a feature was added, and what other features were changed during the same commit (Task 4).

*3) Understanding deletion of features.* We asked participants to identify the commit hash, in which a feature was deleted (Task 5), and to identify two other features deleted from the project (Task 6).

**Analysis.** Participants gave answers by filling out a digital questionnaire after completing each task. For each task, we asked participants to track the time they required to complete it. Afterward, we asked them to fill out the System Usability Scale to assess the usability of the modality, and asked them to answer open-ended questions about their experience to receive further qualitative data.

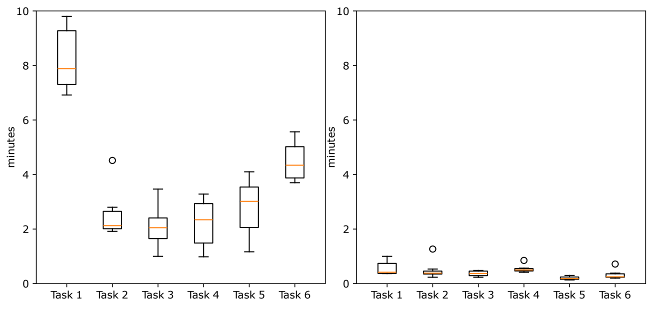
## 3.4 Results.

**Task Correctness.** We notice a large discrepancy in task completeness and correctness when comparing the two modalities.

Overall, only 5 participants were able to solve all tasks correctly when using the Git CLI, with 5 participants being unable to solve tasks within the given timeframe. Most notably, 5 participants were unable to correctly identify the number of commits that changed a feature. When asked to list all features that were changed during a commit that introduced a feature, 4 participants were unable to correctly list them. Finally, 3 participants were unable to identify commits that deleted a feature. Using the feature evolution timeline, 11 participants solved all tasks correctly, with a total of 3 individual errors, resulting in a significantly higher task correctness rate.

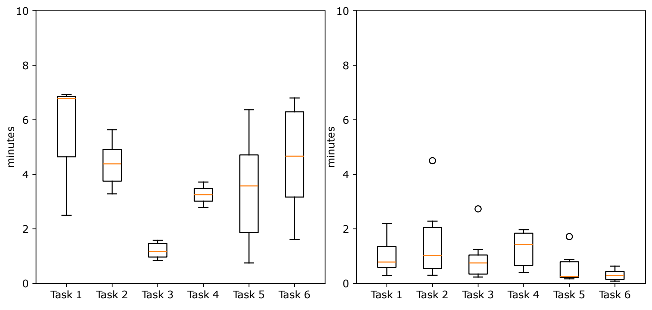
Our experiment shows that using a feature evolution timeline is a simple and effective way for understanding feature evolution compared to traditional git CLI. Especially when investigating what changes to a feature impacted other features, the feature evolution timeline proved to be less error-prone than the Git CLI, since it offers visual guidance for, and does not require using any commands.

**Task Completion Times.** The task completion times for our experiment are presented in Fig. 2 and Fig. 3. We only consider the



(a) Git CLI (b) Feature Evolution Timeline

Figure 2: Task completion times for group A



(a) Git CLI (b) Feature Evolution Timeline

Figure 3: Task completion times for group B

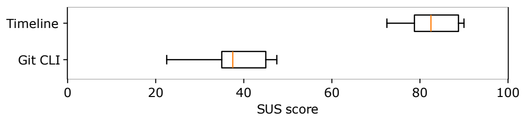
times for correctly solved the task, since incorrectly solving them may contain actions that do not represent correct usage.

For Git CLI, we observe an average task completion time of 22 minutes and 43 seconds for group A, and 22 minutes and 41 seconds for group B. Not surprisingly, participants were able to solve all tasks much faster when using the feature evolution timeline, with group A requiring an average of 2 minutes and 35 seconds (879% faster), and group B 5 minutes and 43 seconds (397% faster).

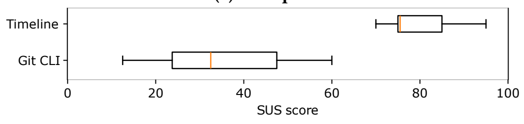
Task 1 required, on average, the most time to complete for both groups when using Git CLI with group A requiring 8 minutes and 13 seconds, and group B 5 minutes and 24. Since it required examining a series of commits one by one, it involved a lot of effort to investigate how many commits modified a feature. In contrast, the timeline visualizes which and how many commits modified a given feature, and does not require multiple commands and reading lines of code. The same reasoning applies to the other tasks as well.

While the comparison of the task completion times does not reveal surprising results, it shows that feature evolution can be understood efficiently through a feature evolution timeline. Traditional tools such as the Git CLI severely lack the functionalities to effectively understand feature evolution in reasonable time.

**SUS Scores.** The SUS scores are presented in Fig. 4. We observe that all participants rated the feature evolution timeline significantly better than the Git CLI in terms of usability. The mean SUS score of the Git CLI from group A is 38.2 ± 8.6, while they rate the feature evolution timeline with a SUS score of 82.9 ± 6.9 on average. For group B, we observe a mean SUS score for the Git CLI of 35.4 ± 18.5 and for the feature evolution timeline a mean score of 79.6 ± 9.1. The scores show that Git CLI completely fails in providing



(a) Group A



(b) Group B

Figure 4: SUS scores

acceptable usability for understanding feature evolution, while the feature evolution timeline achieves near excellent usability [5].

**Qualitative Data.** Participants expressed satisfaction with the feature evolution timeline. Five participants stated the feature evolution timeline was easy and straightforward to use. Independently, five participants claimed the timeline provides a clear and well-structured overview over the evolution of features. In contrast, participants were strongly dissatisfied with the Git CLI. Seven participants mentioned difficulties in using the Git CLI, with 2 of them claiming it was barely suitable for understanding feature evolution. One participant summarized that, *"it was a horrible experience"*. The qualitative data supports our quantitative data, showing that our timeline provides a fast and easy to use method to understand feature evolution, while the Git CLI lacks the functionality to do so.

## 3.5 Hypothesis Evaluation

Table 1 shows the result of our hypothesis tests. We computed pairwise differences in task times and SUS scores between the two modalities, and assessed normality using the Shapiro-Wilk test ($p \geq 0.05$). When normality was satisfied ($p_2 \geq 0.05$), we used a paired t-test, else we applied the non-parametric Wilcoxon signed-rank test instead. We used a significance level of $\alpha = 0.05$ for all

| G | Task | W | $p_1$ | Test | t | $p_2$ | n |
|---|---|---|---|---|---|---|---|
| A | Task 1 | 0.959 | 0.811 | t-test | 15.043 | 0.000 | 6 |
| A | Task 2 | 0.765 | 0.028 | Wilcoxon | 0.000 | 0.031 | 6 |
| A | Task 3 | 0.938 | 0.619 | t-test | 5.528 | 0.001 | 7 |
| A | Task 4 | 0.947 | 0.716 | t-test | 5.561 | 0.005 | 5 |
| A | Task 5 | 0.880 | 0.228 | t-test | 6.279 | 0.001 | 7 |
| A | Task 6 | 0.881 | 0.274 | t-test | 12.582 | 0.000 | 6 |
| A | SUS | 0.957 | 0.791 | t-test | -12.417 | 0.000 | 7 |
| B | Task 1 | 0.803 | 0.122 | t-test | -3.635 | 0.068 | 3 |
| B | Task 2 | 0.904 | 0.431 | t-test | -2.429 | 0.072 | 5 |
| B | Task 3 | 0.813 | 0.055 | t-test | -0.683 | 0.520 | 7 |
| B | Task 4 | – | – | – | – | – | 2 |
| B | Task 5 | 0.936 | 0.630 | t-test | -2.780 | 0.039 | 6 |
| B | Task 6 | 0.905 | 0.407 | t-test | -4.868 | 0.005 | 6 |
| B | SUS | 0.956 | 0.783 | t-test | 5.299 | 0.002 | 7 |

**Table 1: G = group, W = Shapiro-Wilk test statistic, $p_1$ = p-value for assessing normality, Test = applied test based on normality, Test Stat = test statistic, $p_2$ = p-value for asessing significance. '–' indicates insufficient data.**

statistical tests. Recall, that we only consider the task completion times if the task was correctly solved. Since only two participants in Group B completed task 4, we could not test for normality.

Our results show that the timeline significantly improves task performance and usability for Group A. For Group B, no significant differences were found in the first three tasks, due to incorrectly solved tasks using Git, reducing sample size and statistical power.

In summary, using a feature evolution timeline leads to statistically significant improvements in usability and task performance compared to the traditional Git CLI, supporting the hypothesis that our timeline enhances users' ability to understand feature evolution.

## 4 Discussion

From our experiment, we can conclude, that using traditional version control systems such as Git are not suitable for correctly understanding feature evolution, even for systems as small as around 300 lines of code with feature annotations. Not surprisingly, the feature evolution timeline is less error-prone, more efficient, and more usable than using the Git CLI for understanding feature evolution.

### 4.1 System Comprehension

We explored how developers can comprehend the history of feature evolution using a feature evolution timeline. Such a timeline was also proposed by Passos et al. [27]. While previous work focused on the extraction of changes made to features over time [12], such a timeline was not visualized or empirically validated before. Participants were able to identify commits that changed features, and efficient than when using the traditional Git CLI. Although they were not familiar with the system, they were able to complete tasks such as identifying the commits that added, changed, or deleted a feature when using the feature evolution timeline. Our experiment, therefore, shows that a feature evolution timeline can effectively aid developers in comprehending a system's evolution, which is otherwise not effectively possible by e.g., using the Git CLI.

*A feature evolution timeline enables the comprehension of software evolution at the feature level without knowledge about its code.*
Finding 1

### 4.2 Incentives for Recording Feature Locations

Establishing traceability links by e.g., using embedded feature annotations requires effort when writing them, which saves feature location costs when developers need to evolve or maintain features [16]. However, such benefits become only visible to developers in the future, and not when they create them. They, require lightweight tooling for editing to reduce overhead [22], and short term benefits to give them incentives in using them. To give developers an incentive, our timeline must be useful and usable.

Bangor et al. provide a scale for interpreting the SUS score, with a score of at least 70 being acceptable, and least 85 being excellent [5]. The average SUS score of our prototype implementation for our timeline is close to an excellent rating. Therefore, our timeline is useful and usable for understanding feature evolution.

*Our timeline is usable and useful for understanding feature evolution, which may give an incentive to developers to record feature locations by establishing traceability links during development.*
Finding 2

### 4.3 Future Work

We identified future work directions in the field of feature evolution.
**Feature Traceability.** Our results show how feature traceability information can be used to improve the comprehension of feature evolution. However, feature traceability techniques have not found adoption in practice yet. Future work should investigate how developers can be further encouraged in applying such techniques.
**Feature deletion.** Our feature evolution timeline overapproximates feature deletions by marking all features absent from the model as deleted. Future work should better distinguish features that were deleted, merged, split, or refactored.

## 5 Threats to Validity

**Internal Validity.** Participants' prior experience with Git could impact the result of our experiment. Participants who quickly understood the Git CLI's or evolution timeline's functionality during warm-up may perform better, while others may need more time.Still, our experiment shows that a feature evolution timeline enables developers to understand feature evolution with less training required.
**External Validity.** The sample size of our experiment may threaten the generalizability of our work. In particular, we observed a high standard deviation in task completion times and SUS scores when participants used the Git CLI. However, the results of our feature evolution timeline show a significantly lower standard deviation.

## 6 Conclusion

We presented a feature evolution timeline that assists developers in comprehending the evolution of software at the feature level. Our visualization utilizes traceability information from a software project and its commit history to visualize the evolution of features over time. We implemented our prototype as an IntelliJ plugin and evaluated it in a controlled experiemnt with users. The timeline was found to be both useful and usable for feature evolution comprehension, which is crucial to give developers incentives to adopt feature traceability techniques. Lifting the level of abstraction of software components to the feature level is an imperative step in realizing the vision of feature-oriented software evolution, to which we contribute a mean to make features tangible to developers.

While our prototype assumes embedded feature annotations, other traceability methods could also be applied. However, proactively recording feature locations during development has shown to be more effective than retroactive recovery [29]. To support wider adoption, future work should reduce the overhead of such techniques—e.g., by using machine learning to assist developers during development and provide immediate benefits [25].

## Acknowledgments

We thank Riman Houbbi for her contribution on the prototype. This work was supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

## References

[1] 2025. Replication Package. https://doi.org/10.5281/zenodo.16020563.

[2] Hadil Abukwaik, Andreas Burger, Berima Kweku Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *ICSME*.

[3] Berima Andam, Andreas Burger, Thorsten Berger, and Michel R. V. Chaudron. 2017. FLOrIDA: Feature LOcatIon DAshboard for extracting and visualizing feature traces. In *VAMOS*.

[4] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines: Concepts and Implementation.*

[5] Aaron Bangor, Philip Kortum, and James Miller. 2009. Determining what individual SUS scores mean: adding an adjective rating scale. *JUS* (2009).

[6] Lotfi ben Othmane, Golriz Chehrazi, Eric Bodden, Petar Tsalovski, and Achim D. Brucker. 2017. Time for Addressing Software Security Issues: Prediction Models and Impacting Factors. *Data Science and Engineering* (2017).

[7] Lotfi ben Othmane, Golriz Chehrazi, Eric Bodden, Petar Tsalovski, Achim D. Brucker, and Philip Miseldine. 2015. Factors Impacting the Effort Required to Fix Security Vulnerabilities. In *ISC*.

[8] Alexandre Bergel, Razan Ghzouli, Thorsten Berger, and Michel R. V. Chaudron. 2021. FeatureVista: Interactive Feature Visualization. In *SPLC*.

[9] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *SPLC*.

[10] Thorsten Berger, Wardah Mahmood, Ramzi Abu Zahra, Igor Vassilevski, Andreas Burger, Wenbin Ji, Michał Antkiewicz, and Krzysztof Czarnecki. 2025. Cost and Benefit of Tracing Features with Embedded Annotations. *TOSEM* (2025).

[11] Jan Bosch. 2000. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach.*

[12] Nicolas Dintzner, Arie van Deursen, and Martin Pinzger. 2018. FEVER: An approach to analyze feature-oriented changes and artefact co-evolution in highly configurable systems. *EMSE* (2018).

[13] Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. 2019. Visualization of feature locations with the tool featuredashboard. In *SPLC*.

[14] Florian Heidenreich, Ilie Savga, and Christian Wende. 2008. On Controlled Visualisations in Software Product Line Engineering. In *SPLC*.

[15] Rattikorn Hewett and Phongphun Kijsanayothin. 2009. On modeling software defect repair time. *EMSE* (2009).

[16] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining feature traceability with embedded annotations. In *SPLC*.

[17] Kyo Kang, Sholom Cohen, James Hess, William Novak, and A. Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report.

[18] Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *VAMOS*.

[19] Christian Kästner, Salvador Trujillo, and Sven Apel. 2008. Visualizing Software Product Line Variabilities in Source Code. In *SPLC*.

[20] Roberto E Lopez-Herrejon, Sheny Illescas, and Alexander Egyed. 2016. Visualization for software product lines: A systematic mapping study. In *VISSOFT*.

[21] Johan Martinson, Kevin Hermann, Riman Houbbi, David Stechow, and Thorsten Berger. 2025. Visualizing Feature-Oriented Software Evolution. In *SPLC*.

[22] Johan Martinson, Herman Jansson, Mukelabai Mukelabai, Thorsten Berger, Alexandre Bergel, and Truong Ho-Quang. 2021. HAnS: IDE-Based Editing Support for Embedded Feature Annotations. In *SPLC*.

[23] Flávio Medeiros, Thiago Lima, Francisco Dalton, Márcio Ribeiro, Rohit Gheyi, and B Andfonseca. 2013. Colligens: A Tool to Support the Development of Preprocessor-based Software Product Lines in C. In *CBSoft*.

[24] Jens Meinicke, Thomas Thüm, Reimar Schröter, Sebastian Krieter, Fabian Benduhn, Gunter Saake, and Thomas Leich. 2016. FeatureIDE: Taming the Preprocessor Wilderness. In *ICSE*.

[25] Mukelabai Mukelabai, Kevin Hermann, Thorsten Berger, and Jan-Philipp Steghöfer. 2023. FeatRacer: Locating Features Through Assisted Traceability. *TSE* 49, 12.

[26] Renato Lima Novais, André Torres, Thiago Souto Mendes, Manoel Mendonça, and Nico Zazworka. 2013. Software evolution visualization: A systematic mapping study. *IST* (2013).

[27] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. 2013. Feature-oriented software evolution. In *VaMoS*.

[28] Leonardo Passos, Jesus Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. 2015. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *14th International Conference on Modularity (MODULARITY)*.

[29] Julia Rubin and Marsha Chechik. 2013. *A Survey of Feature Location Techniques.*

[30] Hani Bani Salameh, Ayat Ahmad, and Ashraf Aljammal. 2016. Software evolution visualization techniques and methods - a systematic review. In *2016 7th International Conference on Computer Science and Information Technology (CSIT)*.

[31] Tobias Schwarz, Wardah Mahmood, and Thorsten Berger. 2020. A Common Notation and Tool Support for Embedded Feature Annotations. In *SPLC*.

[32] Krzysztof Wnuk, Björn Regnell, and Lena Karlsson. 2009. What Happened to Our Features? Visualization and Understanding of Scope Change Dynamics in a Large-Scale Industrial Setting. In *RE*.