Lightweight Visualization of Software Features with HAnS-viz

Johan Martinson
Ruhr University Bochum
Germany

Kevin Hermann
Ruhr University Bochum
Germany

Riman Houbbi
Ruhr University Bochum
Germany

David Stechow
Ruhr University Bochum
Germany

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

ABSTRACT

Features offer a way to plan software development, but their locations in software assets are often not known. Existing techniques, such as feature-oriented software development, enable traceability of features by implementing features modularly, but are hard to adopt, since they require heavyweight tooling. We believe that feature traceability should be added during development, using lightweight tooling close to the developers' activities. However, adding traceability requires encouragement—ideally in terms of techniques that provide immediate benefits to developers.

We present HAnS-viz, an IntelliJ IDE plugin that provides feature-oriented visualizations that support developers understand and reason about software systems at the feature level. The visualizations present different kinds of feature characteristics and their location in code. Building on our previous work, HAnS, it uses embedded feature annotations that developers create as they write features to lift code-level assets to feature-level representations. A demo video is available at https://youtube.com/watch?v=e4j40dvJQiQ.

CCS CONCEPTS

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

KEYWORDS

feature location, feature traceability, visualization

ACM Reference Format:
Johan Martinson, Kevin Hermann, Riman Houbbi, David Stechow, and Thorsten Berger. 2025. Lightweight Visualization of Software Features with HAnS-viz. In *Proceedings of 29th International Systems and Software Product Line Conference (SPLC'25).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Feature-driven development processes use the notion of features [8, 11, 20, 21] to plan and organize releases of software systems [3]. Besides creating additional features for these systems, developers often require evolving or maintaining existing features to add further functionality to the system or to fix defects such as security vulnerabilities, which requires quick response to minimize damages and costs. Although developers know which feature they have to work on, they still require its location in the assets of the system. In fact, feature location is one of the most common activities of developers, as the location within the assets of the system remains unknown, since they are rarely recorded [10, 13, 22, 29, 35]. Due to the lack of documentation, locating features in the codebase becomes a significantly difficult task, especially if it was implemented a long time ago or the responsible developer left the organization. Manually recovering feature locations after they have been implemented is laborious and error-prone, as features are often scattered over multiple assets [27], and the knowledge about them fades over time. Automatic techniques produce too many false positives and do not scale well in large systems to be usable in practice [1, 5, 6, 17, 29].

Opposed to manually recovering features from the source code, annotating assets using embedded feature annotations has shown to be effective in saving feature location costs [9, 18]. In this process, developers annotate the assets along with working on the feature, while their location is still fresh in their minds. These annotations have a standardized notation and record the location of a feature as well as the relation to specific assets like files and directories. Instead of showing different configuration options that variability annotations offer, feature annotations explicitly specify what feature an asset realizes. Since the annotations are embedded within the source code, they co-evolve with the assets when they are copied, reused or evolved. This emphasizes the benefit in recording feature locations during development as annotating them in source code outweighs the cost of retroactively recovering them.

To effectively create embedded feature annotations, developers require tool support that is close to their development activities. HAnS [24] is an IntelliJ IDE plugin that offers editing support for embedded feature annotations. However, although it equips developers with tools to establish traceability between features and their assets, it still lacks functionalities that assist them in understanding software at the feature-level. Such functionalities can help developers to make features more comprehensibile to them.

Our long-term goal is to raise the level of abstraction at which software systems are managed, establishing features as a better interface to software systems. We present HAnS-viz, an IntelliJ IDE plugin offering visualizations for features of software projects that can be used by developers to identify entry points for changes. Our

SPLC'25, September 01–September 05, 2025, A Coruña, Spain

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

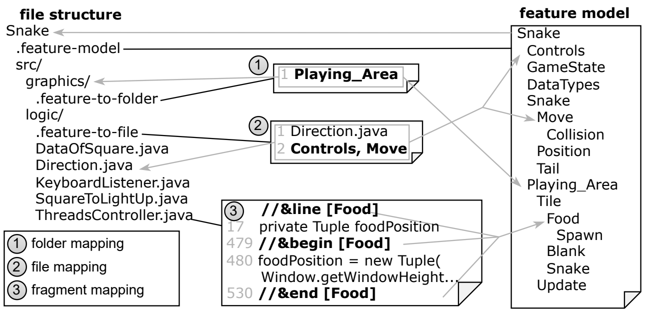


Figure 1: Embedded feature annotation system [9, 33]

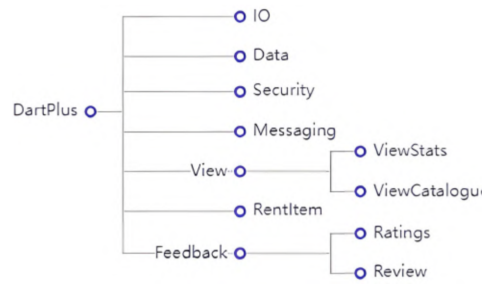


Figure 2: Tree view showing features in their hierarchical order

visualizations are designed to describe software systems at the feature level using feature traceability information, which developers create while they write code. They offer developers an overview of a project's features, their scattering across the codebase, and their tangling, without requiring knowledge of the codebase. Being integrated directly into IntelliJ, it is close to developer's coding activities, requiring no external or heavyweight tools to work with features. HAnS-viz is publicly available on Github[1].

2 BACKGROUND

We present how features can be traced, and what characteristics are commonly visualized.

2.1 Feature Traceability

Feature visualizations require feature traceability data to be in place. As an extension to HAnS, we employ embedded feature annotations, but our visualizations work independently of the feature traceability technique. Embedded feature annotations advocate recording of features while writing code, when the knowledge about its location is fresh in the developers mind [33]. Code assets are mapped to their corresponding features by directly embedding them into the assets through a lightweight embedded feature annotations system illustrated in Fig. 1. Based on embedded feature annotations, HAnS-viz assumes 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. Our visualizations utilize this feature traceability data for several operations, such as the recovery of all feature locations within the codebase, or the refactoring of feature names.

2.2 Feature Characteristics

Scattering and tangling are widely considered undesirable characteristics of features [3, 4, 15, 28, 32]. A feature is considered to be scattered, when it is not implemented in a modularized way and distributed over the codebase [27]. Therefore, their maintenance requires analyzing and changing multiple locations in the codebase [28]. Features are tangled when a single software asset realizes multiple features [34]. Likewise, feature tangling reduces comprehensibility, ease of evolution, and reusability of assets [32]. To this end, HAnS-viz visualizes various feature characteristics:

Lines of code: The size of a feature in the codebase
Scattering degree: The degree to which a feature is distributed over the codebase
Tangling degree: The degree to which a feature intersects with other features in the codebase

3 THE HANS-VIZ PLUGIN

To raise the abstraction level of software assets from code to the feature level HAnS-viz uses traceability data to offer a series of feature visualizations to developers directly in the IntelliJ IDE.

3.1 Tree View and Tree Map View

When working with features, considering their hierarchical structure is essential. The hierarchical structure is typically depicted as a tree, which shows the relationship between parent and child features. This tree could be conveyed as an indented text-file, however, visualizing it in a tree view makes features more tangible to users and gives them an overview of features of a system. HAnS-viz uses a tree view, which shows the hierarchical relations of all features in the project, based on the underlying feature-model (Fig. 2).

Our tree view is interactive and provides editing support for the feature model by providing functionalities for adding, refactoring, moving and deleting features. When deleting features, the view offers a choice in deleting features either only from the feature model along with their traceability information, or along with the annotated code. To avoid breaking functionality of tangled features, the visualization checks if the feature is tangled with other features, and guides the user through each location and asks to resolve them, similar to how developers solve merge conflicts.

Alternatively, to obtain information beyond the hierarchy representation, such as line count, HAnS-viz displays features in a tree map, as they are ideal for displaying large hierarchical structures in a confined space [19]. The size of each tree map field is based on the number of lines of code of each feature (Fig. 3).

3.2 Scattering View

Visualizing the location of scattered features enhances comprehension of them. HAnS-viz visualizes the feature locations in a graph that contains a central node connected to other nodes that map the locations of the feature in different files and packages (Fig. 4). The

[1] github.com/isselab/HAnS-viz

Figure 3: Tree map view showing features based on their hierarchical order and size

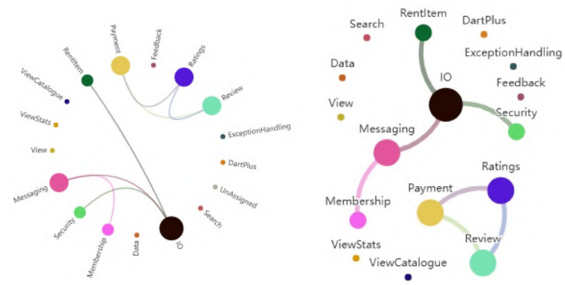
width of each edge connecting the feature to its files corresponds to the feature's coverage, indicating each file's contribution. This visualization allows developers to quickly understand the scattering and distribution of feature implementations across the codebase.

3.3 Tangling Views

Whenever developers make changes to assets, they always need to consider how they impact other features. Feature traceability techniques relate assets to features, so that they can immediately perceive what features are affected by changes to their assets. This allows them to estimate what other features are impacted by changes.

Determining and visualizing the tangling of features aids in obtaining a simplified and clear overview of features affected by a change. HAnS-viz provides tangling views (Fig. 5) developers can use to quickly gain an overview of tangled features and their size in two ways: via a view that shows the relation between all features, and via a view that shows the tangling of feature clusters. The first tangling view is organized as a circular graph to provide an overview of all intertwined features of the project in a clearly arranged way (Fig. 5a). Each node represents a feature, and each edge between the nodes indicates tangling. However, for large projects or features with a high degree of tangling, displaying them in clusters through a non-circular graph may offer a clearer overview (Fig. 5b).

Figure 4: Scattering view showing to what extent each file implements a feature



(a) Circular tangling view **(b) Non-circular tangling view**

Figure 5: Tangling views showing what feature each feature is tangled with

4 PRELIMINARY EVALUATION

HAnS-viz aids developers in understanding features in a software system. Therefore, we evaluated its usability in a user study.

Participants. We recruited 15 students (8 graduate and 7 undergraduate) as suitable stand-ins for professionals [30, 31].

Study Design. We asked participants to solve a series of tasks using the views, and rate their experience with the tool via a questionnaire. As a subject system, we employed the repository of HAnS, which offers 28 features implemented in around 6,000 lines of annotated Java source code spread over 129 files and 24 folders.

Tasks. We asked participants to solve tasks with each of our views.
1) Tree View. Participants identified 3 features, then named 1 child and 1 sibling feature of 2 given features.
2) Tree Map View. Participants located 2 features, identified the 3 largest ones by size, and determined another's largest child feature.
3) Circular Tangling View. Participants identified a feature's tangling degree and its tangled features.
4) Non-Circular Tangling View. Participants identified a feature's tangling degree and the most tangled associated feature.
5) Scattering View. Participants examined a feature and identified the feature coverage of one of the files it appears in.

Analysis. All tasks were provided via a digital form with 5-point Likert scale questions on the user experience when using each view. The questionnaire included open-ended questions for qualitative insights. As a post-task questionnaire, we employed the System Usability Scale (SUS) [12] to assess the usability of HAnS-viz.

Results. All participants were able to solve the tasks for each view. The visualizations were well received by the participants, resulting in an average SUS score of 78. Table 1 shows the mean results of the 5-point Likert-scale questions. The tree view was particularly well received by the participants, who expressed that it was easy to use (4.3), and felt confident in using the view (3.9), as well as would like to use this view frequently (4.3). One participant stated that they *could easily see the structure of the project, and find the necessary features* using the tree view. Still, the other views, particularly the tree map view and circular tangling view received means similar to the tree view. Here, one participant states *"It is easy to locate the biggest features in the project with the tree map view"* and it *"also gives important information about the project structure."* They also expressed that the circular tangling view *"gives a very nice overview about all features and connections between them."*

Question	T	TM	CT	NCT	S
I would like to use this view frequently	**4,3**	3,7	3,6	3,3	3,3
I found the view unnecessarily complex	1,6	1,9	2,4	**1,3**	**1,3**
I thought the view was easy to use	**4,3**	4,1	3,9	3,5	3,5
I found the view very awkward to use	1,5	2,1	2,1	**1,3**	**1,3**
I felt very confident using the view	**3,9**	**3,9**	3,7	3,7	3,7
I found the view too cluttered	2,1	2,5	2,5	**1,2**	**1,2**
It was not pleasing to interact with the view	1,5	2,1	2,2	**1,3**	**1,3**

Table 1: Mean results for each question for each view. T = Tree View, TM = Tree Map View, CT = Circular Tangling View, NCT = Non-Circular Tangling View, S = Scattering View

5 RELATED WORK

Previous work investigated techniques for tracing and visualizing features. FAXE extracts embedded feature annotations and proposes feature-based partial commits to trace features to commits [33]. The Eclipse plugin Colligens maps C preprocessor directives to a feature model and shows the number of files and lines of code which implement a feature [25]. Moreover, FLOrIDA extracts feature annotations from artifacts to visualize feature metrics such as feature size, scattering or tangling in different views [2]. Similarly, featuredashboard uses the same metrics to visualize features-to-asset relationships and common features between projects [14]. FeatureVista interactively visualizes to what extent classes in the codebase contribute to a feature and how features interact with each other [7]. FeatureIDE is another Eclipse plugin that uses a preprocessor to load configurations to enable or disable feature in a system [26]. CIDE provides further support in highlighting feature-specific code and hiding code fragments for improved visibility on code [23]. As an extension, we implemented a feature evolution timeline in HAnS-viz that visualizes the commit history for features [16].

6 CONCLUSION

We presented HAnS-viz, an IntelliJ IDE plugin that offers visualizations to assist developers in comprehending software at the feature level. As an IDE plugin, it is lightweight and close to the activities of developers. Our visualizations utilize traceability information from a software project to visualize features and their characteristics. Users expressed satisfaction with HAnS-viz when using it to understand features. While HAnS-viz assumes the presence of embedded feature annotations, other traceability techniques could be used as well. However, proactively recording feature locations has shown to be more effective than retroactively recovering them [9, 29].

ACKNOWLEDGMENTS

We thank Philipp Kuzmiers, and Mariana Hohashvili for their contributions on the visualizations. Partially funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

REFERENCES

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

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

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

[4] Sven Apel, Thomas Leich, and Gunter Saake. 2008. Aspectual Feature Modules. *TSE* (2008).

[5] 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).

[6] 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*.

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

[8] 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*.

[9] 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).

[10] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. 1994. Program understanding and the concept assignment problem. *Commun. ACM* (1994).

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

[12] John Brooke. 1996. *SUS – a quick and dirty usability scale.*

[13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature location in source code: a taxonomy and survey. *Journal of Software: Evolution and Process* (2013).

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

[15] Favre. 1996. Preprocessors from an abstract point of view. In *ICSM*.

[16] Kevin Hermann, Johan Martinson, and Thorsten Berger. 2025. Lightweight Visualization of Software Features with HAnS-viz. In *SPLC*.

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

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

[19] Brian Johnson and Ben Shneiderman. 1998. *Tree-maps: A space filling approach to the visualization of hierarchical information structures.* Technical Report.

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

[21] 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*.

[22] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. *Features and How to Find Them: A Survey on Manual Feature Location.*

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

[24] 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*.

[25] 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*.

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

[27] 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)*.

[28] Rodrigo Queiroz, Leonardo Teixeira Passos, Marco Túlio Valente, Sven Apel, and K. Czarnecki. 2014. Does feature scattering follow power-law distributions?: an investigation of five pre-processor-based systems. In *FOSD*.

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

[30] Per Runeson. 2003. Using students as experiment subjects - An analysis on graduate and freshmen student data. *EASE* (2003).

[31] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. 2015. Are Students Representatives of Professionals in Software Engineering Experiments?. In *ICSE*.

[32] Claudio Sant'Anna, Alessandro Garcia, Christina Chavez, Garcia Chavez, Carlos Lucena, and Arndt Staa. 2003. On the reuse and maintenance of aspect-oriented software: An assessment framework. *SBES* (2003).

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

[34] Periklis Sochos, Matthias Riebisch, and Ilka Philippow. 2006. The feature-architecture mapping (FArM) method for feature-oriented development of software product lines. In *ECBS*.

[35] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How developers perform feature location tasks: a human-centric and process-oriented exploratory study. *Journal of Software: Evolution and Process* (2013).