HAnS: IDE-Based Editing Support for Embedded Feature Annotations

Johan Martinson*
Herman Jansson*
Chalmers | University of Gothenburg
Sweden

Mukelabai Mukelabai
Chalmers | University of Gothenburg
Sweden

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

Alexandre Bergel
Dept. of Computer Science,
University of Chile, Chile

Truong Ho-Quang
Chalmers | University of Gothenburg
Sweden

ABSTRACT

When developers maintain or evolve software, they often need to know the locations of features. This proves challenging when the feature locations are not documented, when the code was written by different developers who may have left the organization, or when the developer's memory of the implementation has faded. Automated feature location techniques are hard to adopt in practice, especially since they boast too many false positives. To address these challenges, embedded feature annotations have been proposed to allow developers to *trace* features in code during development with minimal effort. However, tool support is needed for developers to effectively record and use these annotations. We propose HAnS as a tool to meet this need; it is implemented as an IntelliJ IDE plugin to support developers seamlessly record feature locations while they write their code. HAnS supports developers when mapping features to software assets, such as files and code fragments, with code completion and syntax highlighting. It also provides functionality to browse feature definitions and locations, as well as refactor features. A demo video is available at https://youtu.be/cx_-ZshHLgA.

CCS CONCEPTS

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

KEYWORDS

feature location, embedded feature annotations, tool support, IDE

ACM Reference Format:
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 *25th ACM International Systems and Software Product Line Conference - Volume B (SPLC '21), September 6–11, 2021, Leicester, United Kingdom.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3461002.3473072

*Both authors contributed equally.

SPLC '21, September 6–11, 2021, Leicester, United Kingdom

ACM ISBN 978-1-4503-8470-4/21/09…$15.00
https://doi.org/10.1145/3461002.3473072

1 INTRODUCTION

Features [**? ? ?**] drive the development of software-intensive systems. Almost all agile processes, such as SCRUM and XP, rely on features to organize teams and software releases [**?**]. While developers may document the list of features available in a system at the time of release, it is often the case that they do not record where those features are located in code, but instead retain that knowledge in their minds. Consequently, when they need to evolve or maintain features, for instance, to fix bugs, developers spend considerable effort to first locate the features in code before proceeding with their maintenance tasks—in fact, feature location is one of the most common activities of developers [**? ? ?**]. However, it is a daunting task, especially, when the features being located were implemented long after implementation, possibly by different developers who may have left the organization; or when the features cross-cut several folders, files, or code fragments within files.

While the feature location problem may not be prevalent in software product lines [**? ?**], which already have techniques to manage features, it is a significant problem in many long-lived systems, especially when these realize variants using clone & own. Latest, when cloned variants are re-engineering into configurable product-line platforms [**? ? ?**], features and their locations need to be known, which have then ideally already been tracked during clone & own. Since manual [**?**] and automated [**?**] retroactive feature location techniques have been shown to be ineffective or inefficient[**? ?**], developers must record feature locations early during development. To that end, embedded feature annotations [**? ?**] have been proposed to record features during development. These annotations have been standardized with a common notation [**?**] that developers can use, and empirical evidence [**?**] suggests that the benefits of the annotations outweigh the costs of recording them. Since the annotations are embedded within the assets, they naturally co-evolve with the assets when the code is copied or moved around, and developers do not need to maintain external documentation of the feature locations, e.g., in traceability databases [**?**].

However, tool support is needed to effectively use feature annotations and to encourage developers to record them with minimal intrusion during development activities. We propose HAnS[1] (Helping Annotate Software) as an IDE plugin to help developers annotate software assets, such as folders, files and code fragments, with the features they implement. HAnS provides editing support for developers to (i) map features to assets, with code completion for feature

[1] https://bitbucket.org/easelab/hans-text/

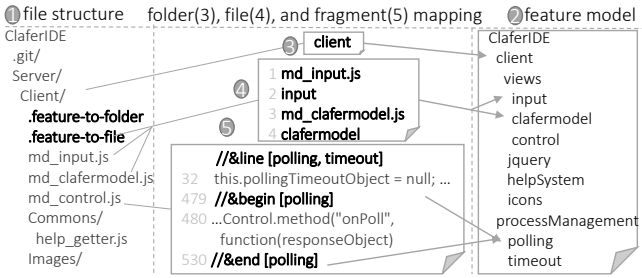

Figure 1: Embedded feature annotations [? ?]

names and syntax highlighting, (ii) browse feature definitions and locations, and (iii) refactor features, e.g., rename or remove features.

We demonstrate HAnS, targeting developers who write feature-oriented code and need to trace the locations of the features in code. We discuss the challenges it addresses, the annotation concepts it uses, and finally present its capabilities and how effective they are in supporting 13 developers who participated in our initial experiments to record and maintain feature annotations.

2 BACKGROUND AND CHALLENGES

Recording feature locations proactively during development poses a few challenges that we seek to address with our tool HAnS.

Challenge 1: How to record feature locations with minimal intrusion to development activities. Since recording features long after implementation or externally neither minimizes developer effort nor improves accuracy when recovering feature locations, HAnS is engineered as an IDE plugin with editing support for developers to seamlessly annotate code with features as they write the code. HAnS' annotation mechanism allows developers to record feature locations at folder-, file-, code fragment-, and line-level [**?** ?], according to the project's notion of features. Note that these are traceability, not variability annotations.

Figure 1 shows the annotation mechanism HAnS uses [**?**] (specification available at [**?**]). Folders and files are mapped to their features using textual files placed within the project structure, while fragments and individual lines of code are mapped using in-line comments. For instance, in Fig. 1, the folder *Client* is mapped to the feature *client* by adding the file .feature-folder within the folder and writing the feature *client* within this file as shown in part 3 of the figure. The files *md_input.js* and *md_clafermodel.js* are mapped to the features *input* and *clafermodel* respectively, by adding the textual mapping file .feature-files in the same folder where the files reside then recording the mapping as shown in part 4 of Fig. 1. Fragments and lines are mapped as shown in part 5 of the figure; for lines, the annotation appears at the end of the line mapped.

HAnS' annotation mechanism is programming language independent since it uses plain text files for folder and file annotations while it relies on the commenting style of each programming language for fragment annotations. As an IDE plugin it works for all languages the IDE supports. Even though HAnS is currently an IntelliJ plugin, it provides the basis for similar plugins in other major IDEs, such as Microsoft's Visual Studio or VSCode.

Challenge 2: How to effectively record feature locations. To effectively record feature locations, developers must be able to (i) consistently use feature names—correctly spelled or qualified; (ii) correctly map features to assets—adhere to the annotation specification to allow visualization tools retrieve all locations; (iii) browse features and their code, e.g., traverse from the feature model to all locations of a selected feature; and (iv) refactor annotations, e.g., renaming a feature. HAnS addresses these challenges through consistency rules [**?**] and a wide range of features we discuss in detail in Sec. 3.

3 THE HANS PLUGIN

To illustrate HAnS' functionality, let us assume Gretel is a software developer building a Java Snake game. She wants to use HAnS to record feature locations during development.

3.1 Mapping Features to Code

To start with, Gretel must create a text file called *.feature-model*, in the project's root folder, in which she would add all features of her project. This file is needed for all other functionality, such as code completion for feature names and feature browsing, to work. Note that sub-features can have same names in the feature model, provided they have unique parent features. However, they must be *least partially qualified* when mapping them to assets. Gretel might want to map features to an entire folder, file, code fragment, or even an individual line of code. To map a folder to one or more features, she adds the text file *.feature-to-folder* within the folder and writes the features in the file—one line per feature (part 1 in Fig. 2). Similarly, to map files to their features, she adds the text file *.feature-to-file* within the folder where the files to be mapped are located, and then she writes each mapping using two lines: top line for the files, the bottom line for the features mapped to the files (part 2 in Fig. 2). HAnS provides menus to create and edit all the above feature mapping files. To map features to fragments or individual lines of code, she writes the annotations within Java's single-line comments, wrapping the features in *&begin[]* and *&end[]* or *&line[]* tags respectively (part 3 in Fig. 2). She can write fragment annotations faster in two ways: She can either type any of the three different tags and press TAB to complete it: (i) *&begin*—creates a *&begin[|]* tag with a matching *&end[|]* tag; (ii) *&line*—creates the *&line[|]* tag and; (iii) *&end*—creates the *&end[|]* tag; or, she can select any code fragment and press *CTRL-ALT-J*, which displays options to surround the code with different templates. If she chooses the *&begin* template, HAnS will wrap the code between *&begin[|]* and *&end[|]* tags as shown in Fig. 3.

3.2 Browsing Features

Gretel might want an easier way to browse through her hierarchical list of features or even find all locations of a selected feature. HAnS provides a graphical view of the textual feature model with *The Feature Model View* window (part 4 in Fig. 2). If Gretel wants to view all usages of the feature *Food*—a snake eats food to grow in length, she can *right-click* the feature and select *Find Usages* in the context menu that appears. The resulting view shows all the usages of the feature Food (left side of Fig. 4); clicking on any of the usages opens the specific location of the feature (right side of Fig. 4). If Gretel chooses, she can also directly open the *.feature-model* file and *ctrl-click* on the feature Food to browse its locations as described above. The syntax highlighting makes it easy for Gretel to quickly spot her feature. Additionally, syntax highlighting is context aware: if code

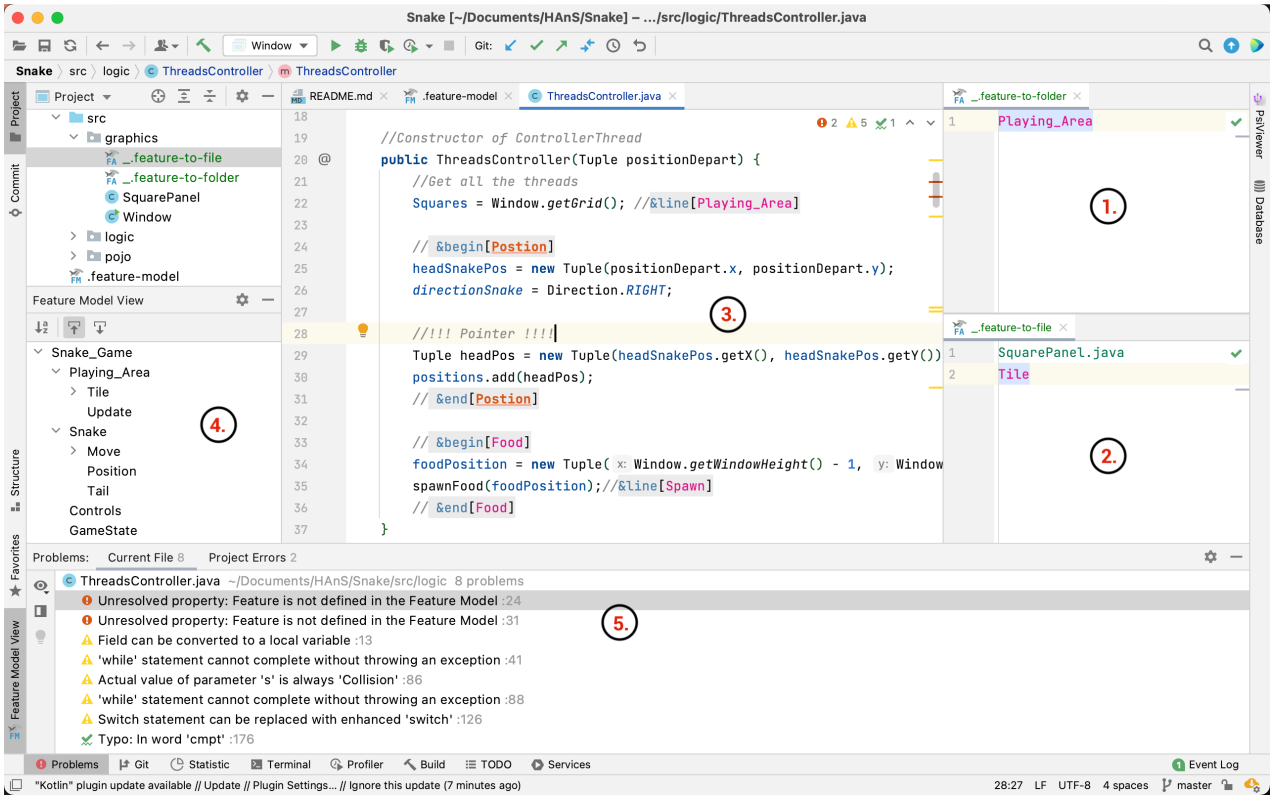

Figure 2: Example of the HAnS plugin in action

is annotated with a feature absent from the feature model, it is highlighted differently, and it notifies developers in the problems tab of IntelliJ. Part 5 of Fig. 2 shows this for a misspelled feature *Positions*.

3.3 Refactoring Annotations

If Gretel wants to rename a feature, she can use IntelliJ's refactoring short-cut—*SHIFT-F6*—from within an annotation and choose a new name. This will rename all references to the feature. Or, she can use the context menu in the *Feature Model View* to rename the feature. From the *Feature Model View* she can also add or delete features.

3.4 HAnS Visualization

We also extend HAnS with interactive visualizations and metrics that would allow Gretel to visually explore, for instance, how scattered or tangled features are, thereby reducing Gretel's reliance on external visualization tools. Since visualizations allow developers to comprehend more complex systems, they can further encourage developers to use annotations. Inspired by our previous tools [? ?], the current visualizations show (i) the mapping between features and code as a graph, (ii) the tangling of features in a graph, where the width of edges corresponds to the degree of tangling between features (nodes), and (iii) metrics such as scattering degree or lines of feature code for a given feature. The plugin HAnS-vis is still under development, but can be used prototypically already.[2]

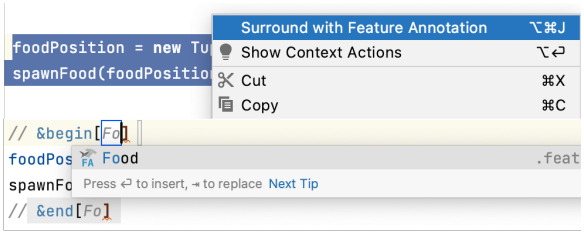

Figure 3: Example of using the surrounding live templates

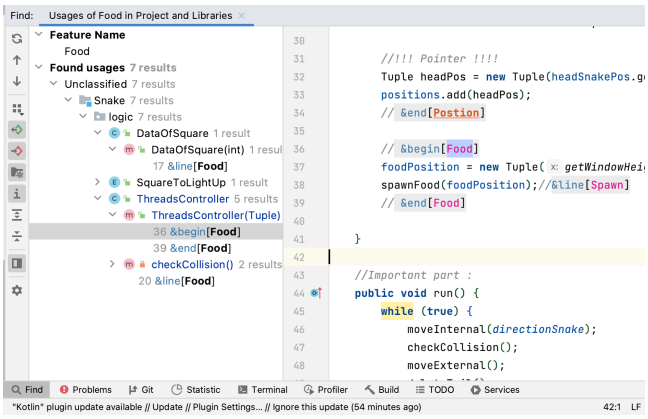

Figure 4: Example of using find usages

[2] https://bitbucket.org/easelab/hans-vis/

4 BENEFITS AND EXPERIENCES

To gather qualitative feedback on HAnS' effectiveness and usability, we conducted a user study with 13 participants with an average programming experience of 2.5 years; 7 were professional developers, and 6 were Master's students. The participants were tasked to extend a Snake[3] game by adding new features and refactoring some existing ones, with and without the plugin. Afterwards, they were asked to complete a questionnaire about their experience.

While basic IDE functionalities such as code completion and syntax highlighting did not influence the participants' opinions for or against HAnS, since they were expected, feature browsing and refactoring did. For tasks that required participants to map features to code, there were marginal differences in participants' perceived effort required, especially since the subject project was small and the assigned features were few. However, participants found HAnS' feature referencing and refactoring functionality very useful since it was much harder for them to rename all references of a feature, or to view all locations of the feature without HAnS than with HAnS.

We also found that, overall, participants made fewer annotation mistakes when working with HAnS than without because of HAnS' support for code completion and syntax highlighting. The mistakes included *forgetting to qualify* features whose names need to be least partially qualified, using the *wrong annotation syntax*, *forgetting to define* a feature in the feature model, and *misspelling* a feature's name. However these mistakes were also influenced by how familiar the participants were with the annotation mechanism: we observed that those participants who first performed some tasks with HAnS enabled then switched to working without HAnS still made mistakes even when they worked with HAnS.

Overall, while the participants indicated that they were motivated to use embedded annotations, and found HAnS useful and effective for recording feature locations, some still found difficulties to reason about the different granularities of features and when to annotate specific code with a feature.

5 RELATED WORK

While most feature visualization tools target features in software product lines or highly configurable systems [**?** **?**], a few target embedded feature annotations. Among these are: FLORiDA [**?**]—a standalone Java application that provides several feature views and metrics e.g., tangling and scattering degree; FeatureVista [**?**]—a standalone application that provides interactive feature views for object oriented systems; and FeatureDashboard [**?**]—an eclipse plugin that provides views and metrics similar to FLORiDA. However, these tools offer no editing support.

FAXE [**?**] is a feature annotation extraction engine that formalizes the common notation which HAnS uses to represent embedded annotations proposed by Ji et al. [**?**]. It is available as a Java library that retrieves annotations, calculates metrics, finds inconsistencies, and renames features; but offers no editing support.

Seiler et al. [**?**] propose Java-specific annotations to trace code and requirements, and provide a recommender system to suggest annotations to developers. Similarly, Abukwaik et al. [**?**] propose a recommender to suggest Ji et al.'s [**?**] annotations when developers forget to annotate their source code during commits. While they offer no editing support, such recommenders complement HAnS by ensuring that developers *continuously* annotate code.

6 CONCLUSION

We presented HAnS, an IntelliJ IDE plugin to help developers seamlessly record feature locations while they write their code. HAnS effectively supports developers to record features and map assets to them, offering code completion of feature names and syntax highlighting to obtain consistent annotations. It allows browsing feature definitions and locations, and it supports refactoring of features. Future work includes disseminating the annotation tool and collecting longitudinal data about its effectiveness.

[3] https://github.com/johmara/Snake