A Common Notation and Tool Support for Embedded Feature Annotations

Tobias Schwarz
Chalmers | University of Gothenburg

Wardah Mahmood
Chalmers | University of Gothenburg

Thorsten Berger
Chalmers | University of Gothenburg

ABSTRACT

Features are typically used to describe the functionalities of software systems. They help understanding systems as well as planning their evolution and managing systems. Especially agile methods foster their use. However, to use features, their locations need to be known. When not documented, they are easily forgotten and then need to be recovered, which is costly. While automated feature-location techniques exist, they are not usable in practice given their inaccuracies. We take a different route and advocate to record locations early using a lightweight annotation system, where feature information is embedded in software assets. However, given the potential design space of annotations, a unified notation and tool support is needed. Extending our prior work, we present a unified, concise notation for embedded annotations, which we implemented in FAXE, a library for parsing and retrieving such annotations, useable in third-party tooling. We demonstrate its use, especially for an advanced use case of feature-oriented isolated development by automating partial commits.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Traceability**.

KEYWORDS

feature location, embedded annotations, partial commits

ACM Reference Format:
Tobias Schwarz, Wardah Mahmood, and Thorsten Berger. 2020. A Common Notation and Tool Support for Embedded Feature Annotations. In *24th ACM International Systems and Software Product LineConference - Volume B (SPLC '20), October 19–23, 2020, MONTREAL, QC,Canada.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3382026.3431253

1 INTRODUCTION

Features are commonly used to abstractly and intuitively describe the functionality of software systems [6]. They allow keeping an overview understanding of complex systems, providing a common language for different stakeholders, ranging from developers to domain and business experts. Agile development methods, such as Scrum, XP and FDD rely on features to plan the development. In variant-rich systems, such as software product lines, features help distinguishing the variants. Furthermore, many developers label commit messages with the feature they implement, as well as they align commits with features.

Using features to manage and evolve systems requires knowing their locations. If not documented, the developers' knowledge about features diminishes quickly and requires recovering the feature locations in software assets (e.g., code, models, requirements, documentation), which is in fact one of the most common activities of developers [11, 15, 18]. In large and complex systems, this activity can easily become laborious and error-prone [18], especially when features are scattered [13] across the software assets. Even though, automated feature-location tools have been proposed [15], they require project-specific setup effort and often yield too many false positives to be useful in practice.

To avoid these problems, features and their locations should be documented explicitly. Two strategies exist: documenting feature information externally to the assets, such as in feature databases, or documenting feature information internally by embedding it into the software assets [4, 7–10, 16, 17]. The former strategy requires external tools (e.g., feature databases [14]), a universal way to exactly refer to locations inside software assets, as well as a method of keeping them updated during software evolution. The latter strategy requires an annotation system with a standardized concise and intuitive syntax to embed the feature information into software assets. But, adding embedded annotations during development is cheap, and the annotations evolve naturally together with the assets, with little maintenance overhead [8].

We advocate that developers add embedded feature annotations into software assets during development. The annotations facilitate locating and browsing features and their locations quickly, for instance, when maintaining features (e.g., delete or split features), and propagating feature implementations across cloned variants in clone & own scenarios. In addition, embedded annotations enable feature-oriented software evolution [12], by exploiting feature locations to calculate feature metrics and provide feature visualizations, which allow intuitively monitoring software progress. Figure 1 shows some visualizations from FeatureDashboard [7]. On the left are mappings between features and (model, code and documentation) files; on the right are mappings between features and folders. The annotations also facilitate re-engineering cloned variants into an integrated platform, where they can be converted into variability annotations (e.g., #IFDEF preprocessor annotations or feature flags) to make features optional or to capture the differences between cloned and adapted features. In contrast to variability annotations, as we will show, our embedded annotations are (i) more flexible, as they do not need to align with syntactic structures of programs, (ii) more lightweight, as they do not require heavyweight tooling and project setups, and (iii) they also capture mandatory features.

However, no unified notation exists for embedded feature annotations. To exploit their potential, a concise and flexible syntax

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPLC '20, October 19–23,2020, MONTREAL, QC, Canada
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7570-2/20/10…$15.00
https://doi.org/10.1145/3382026.3431253

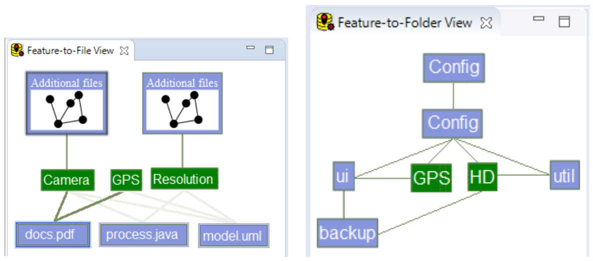


Figure 1: Visualizations of feature locations in files and folders using the tool FeatureDashboard [7]

is needed, that developers can rely on. Specifically, it should be intuitive, easy to learn and use, and concise. It should also provide the basis for further tools (e.g., for visualizing or browsing features using feature dashboards [4, 7, 12]). In fact, given the huge design space for such annotations, we experienced first hand that, without well-defined syntax and semantics for embedded annotations, (tool) developers interpret our annotation system differently, challenging the reuse of annotations and tool interoperability.

We present a unified syntax for specifying embedded feature annotations and tool support for managing annotations in software assets. Our contribution is threefold. First, building on our prior work [3, 4, 7, 8], we design and evaluate a consolidated notation for embedded feature annotations. The syntax is programming-language-independent, established with input and experience from industry. Second, we provide the tool FAXE (Feature Annotation eXtraction Engine) [2]—a stand-alone library usable in third-party applications, such as feature dashboards—to process feature annotations in large software projects. Third, we demonstrate a more advanced use case enabled by our annotations and FAXE: feature-based partial commits to facilitate isolated development on the granularity of features (instead of whole branches or forks). These allow to align commits with features, among others, easing release engineering (e.g., via cherry-picking) and collaboration of developers who are typically assigned to features.

2 EMBEDDED ANNOTATIONS

The annotation system allows mapping features to many kinds of software assets (code and non-code)—at the granularity of whole folders and files, and also textual assets at an arbitrary granularity (e.g., classes, methods, code blocks, lines). The exact specifications with grammars can be found online [1]. For illustration, our running example is Bitcoin-Wallet, an Android app we annotated before [9].

2.1 Design Methodology

We iteratively designed the system, discussing many design alternatives (discussed shortly), which we assessed regarding the design properties: *intuitive*, *useful*, *easy-to-learn*, *easily applicable*, *flexible-to-use*, *non-redundant*, *succinct*, *robust during software evolution and reuse*, and requiring *minimal developer effort*. We documented the rationale and discussed also the placement in assets, the respective keywords used in the syntax, and ways of referring to features in the annotations. We evaluated the syntax with respect to the design properties using a survey with fellow researchers and industrial collaborators.

2.2 Embedded Annotations Design Decisions

We utilize the specific escape syntax for comments in the host language to make the specification independent of the programming language. For specifying code annotations, we choose the 'begin' keyword instead of 'open' and 'start,' as we find that more intuitive. We allow developers to interleave feature annotations, only requiring the developers to add an ending annotation for every begin annotation. For mapping single lines of code to features (i.e., line annotations), we discussed two alternative ways: adding the annotation *before* the line or adding it *at the end of* the line, favoring the latter to facilitate parsing. We allow specifying multiple features in a single annotation, in addition to having separate ending annotations for each feature. To provide flexibility, we do not require feature names to be unique across the (containing) feature model. To uniquely reference features in the annotations, we employ the use of least partially qualified (LPQ) paths, which are shortest possible paths needed to uniquely identify a feature (explained in detail in Sec. 2.3).

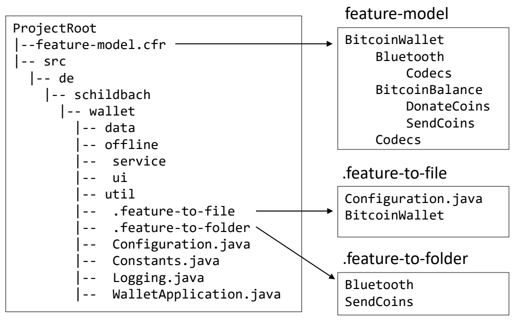
2.3 The Notation and Its Practical Use

Feature Model. To organize features in a hierarchy, we use a simple textual notation inspired by the Clafer modeling language [5]—allowing easier creation and modification without dedicated tools.

Developers add the feature-model file in the project's root folder. The first line is the *root feature*, which is named after the project. Each feature is listed as a line in the file, with tab-based indentation reflecting the feature hierarchy. While not needed in our use cases, developers could also specify relations among features, which might be useful later e.g., when making features optional. Figure 2 (top-right) illustrates the feature-model design.

Feature References. As mentioned above, we do not enforce feature names to be unique in the model to provide flexibility. To concisely refer to non-unique features, we use their LPQ path related to the feature hierarchy. A feature is prefixed with the shortest excerpt of its full path that makes it unique. The feature names in LPQ are conjoined with "::". See the features named *Codecs* in Fig. 2 (top right). They can be uniquely referred to as *Bluetooth::Codecs* and *BitcoinWallet::Codecs.*

Feature-to-Folder Mappings. To map a folder to a feature, the developer adds a .feature-to-folder file in the folder itself. Features mapped to the folder are listed in individual lines. Such feature-to-folder mappings are more stable than feature-to-file mappings during evolution, as they are usually retained during folder rename, move and clone & own operations. An example .feature-to-folder file is shown in Fig. 2 (bottom right).


```
ProjectRoot
|--feature-model.cfr
|-- src
  |-- de
    |-- schildbach
      |-- wallet
        |-- data
        |-- offline
        |--  service
        |--  ui
        |-- util
        |--  .feature-to-file
        |--  .feature-to-folder
        |--  Configuration.java
        |--  Constants.java
        |--  Logging.java
        |--  WalletApplication.java
```

feature-model
```
BitcoinWallet
    Bluetooth
        Codecs
    BitcoinBalance
        DonateCoins
        SendCoins
    Codecs
```

.feature-to-file
```
Configuration.java
BitcoinWallet
```

.feature-to-folder
```
Bluetooth
SendCoins
```

Figure 2: Feature model, feature-to-file, and feature-to-folder mappings illustrated for our running example

```
58   public class WalletBalanceFragment extends Fragment { ...
219      //&begin[DonateCoins]
220      private void handleDonate() {
221          //&begin[SendCoins]
222          SendCoinsActivity.startDonate(activity, null,
                 FeeCategory.ECONOMIC, 0);    //&line[Fee]
223          //&end[SendCoins]
224      }
225      //&end[DonateCoins]
```

Listing 1: Illustration of fragment and line annotations

Feature-to-File Mapping. Feature-to-file mappings are stored in a separate .feature-to-file file, which resides in the same folder as the mapped files. To map a file to a feature, the developer adds or extends this mapping file. Files names and feature references are written in individual lines, separated by commas, where the file(s) in one line map to the feature(s) specified in the next line. All file assets, regardless of their type (e.g., binary, model, and test) can be mapped to features using our syntax. An example file .feature-to-file is shown in Fig. 2.

Fragment Annotations. As explained above, developers specify embedded code annotations in comments. The above is followed by an ampersand and keyword ‘begin’ or ‘end’, followed by comma-separated feature references in LPQ. The *scope* of a fragment annotation is the set of lines between the respective begin and end.

Line Annotations. Line annotations are a special case of fragment annotation, where the scope of the annotation is limited to one line of text. Line annotations are also specified as comments, where the ampersand is followed by the ‘line’ keyword and the feature references in LPQ. Listing 1 demonstrates fragment and line annotations for three features (*DonateCoins*, *SendCoins*, *Fee*) in BitcoinWallet.

3 FAXE OVERVIEW

The tool FAXE automatically extracts and processes embedded annotations specified in the proposed syntax from a given asset. It is a lightweight tool which requires no installation by the developer. FAXE is implemented as a Java library under the APACHE 2.0 license [2]. To facilitate integration with IDEs and other tools (e.g, for visualization), we provide the implementation as a single jar file, with all dependencies contained inside. At the core of the engine is the annotation parser built with the ANTLR4 parser generator. It relies on syntax of our annotation system specified as an ANTLR4 grammar. Given an asset (a project, folder or specific file), FAXE extracts annotations from all sub-assets recursively, down to the line annotations. It is language-independent; extracting annotations from textual assets written in any language.

Presently, users can interact with FAXE in two ways; integrate the library in the client project and use its API directly, or use its command line interface. FAXE builds on an object model; for each API request, FAXE extracts the location of features and returns an object list. The returned data includes asset type, asset name, index of begin and end, and the feature(s) referred to in the annotation. Features from the annotations that do not exist in the feature model are added to it dynamically by FAXE. Listing 2 shows an excerpt of the command-line output from FAXE for BitcoinWallet.

3.1 FAXE Commands

FAXE offers a set of commands to interact with it. Two basic commands -h and -v display help content and version, respectively. For

```
// Type      Asset                                   Start  End  LPQ  Reference
{FRAGMENT  wallet\ui\WalletBalanceFragment.java  220  226  DonateCoins},
{FRAGMENT  wallet\ui\WalletBalanceFragment.java  222  224  SendCoins},
{LINE      wallet\ui\WalletBalanceFragment.java  223  223  Fee},
{FILE      wallet\Configuration.java              –    –    BitcoinWallet},
{FOLDER    wallet                                 –    –    Bluetooth},
{FOLDER    wallet                                 –    –    SendCoins},
```

Listing 2: FAXE example command-line output

extracting, refactoring, and checking annotations, there are four commands. The first is *getEmbeddedAnnotations*, which returns embedded annotations in a given asset path. The second is *calculateMetric*, which calculates a feature metric given an asset. FAXE can calculate all metrics supported by FeatureDashboard [7]. The third command is *checkConsistency*, which checks for syntactic issues in embedded annotation specification. Examples of such issues are different parenthesis for opening and closing an annotation, having no end annotation for a begin annotation, having an annotation without any mapped asset, and features missing in the feature model but referred to in annotations. Lastly, *rename* renames the feature referred to in the given lpq to a new given name. Table 1 shows brief descriptions of the commands.

3.2 Feature-Based Partial Commits

Developers often work on different parts of the same file and make commits at a fine-grained level. They want to track changes at the source-code level and quickly repair if the system fails. For commits with many fine-grained changes, it is difficult to track changes individually. Also, deciding which parts of code change together is non-trivial. Git partial commits allow developers to commit only parts of changes instead the whole change-set. At present, this process is entirely manual and requires a high degree of developer interaction, spanned over many steps. We propose feature-based partial commits—an intuitive way of isolating development at the granularity of features. Feature-based partial commits allow developers to align commits with features and work on more meaningful

Table 1: FAXE Commands

getEmbeddedAnnotations path *lpq export*
Extracts and returns embedded annotations from an asset’s path for the feature in lpq. Exports the output to a file if the flag export is set. If lpq is not specified, it extracts all annotations from path.
calculateMetric path *metric* lpq *export*
Calculates and returns required metric (enum) for the feature in lpq from the given path. It also exports the output to a file if the flag export is set. If *metric* is not specified, all metrics are calculated and exported.
checkConsistency path *export*
Identifies inconsistencies among annotations and feature model (e.g., feature referenced, but not declared in the model) in the given path scope. Exports the report into a file if the flag export is set.
rename path lpq newname
Renames the feature in lpq to newname in the feature model and all annotations within the path scope.

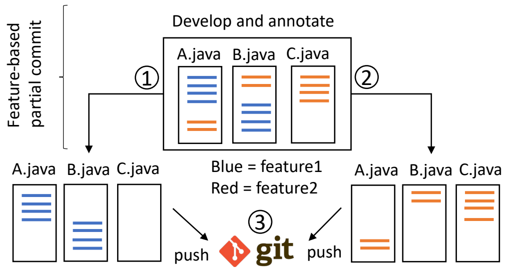


Figure 3: Workflow of feature-based partial commits

chunks of code. Since features are cross-cutting and fine-grained, knowing their locations in source code enables developers to contribute to features. The tool's implementation is a combination of a Bash script git-pfc and the FAXE engine. An overview and comparison of Git partial commits and feature-based partial commits is illustrated below using a simple scenario.

Git Partial Commits. For making a partial commit, the developer manually traverses all changes and uses "git add -p" to commit them. Git splits all changes into "hunks," which are blocks consisting of one or more lines of code. For each hunk, she has to choose from a variety of options: **y** (stage this change), **n** (don't stage this change), **q** (quit), **a** (stage this hunk and all later hunks in the file), **d** (do not stage this hunk or any of the later hunks in the file), **g** (select a hunk to go to), / (search for a hunk matching the given regex), **j** (leave this hunk undecided, see next undecided hunk), **J** (leave this hunk undecided, see next hunk), **k** (leave this hunk undecided, see previous undecided hunk), **K** (leave this hunk undecided, see previous hunk), **s** (split this change further), **e** (manually edit the current hunk), and **?** (print help) before proceeding. After choosing the appropriate option, she commits the changes with a fitting commit message. When done, she pushes.

Feature-Based Partial Commits. Figure 3 illustrates the workflow for feature-based partial commits. Let us assume a developer works on two features, 'feature1' and 'feature2,' scattered across the files A.java, B.java and C.java. After extending and modifying the code together *with* embedded annotations, she uses git–pfc to see all features contained in the changeset, from which she can select features and automatically commit changes belonging to those features. As such, the benefit lies in omitting feature identification and location, which can become laborious for commits with large changesets and many, potentially scattered features. When committing, she can extend the auto-generated commit message by FAXE. She finally pushes the changes to the remote repository. The steps with and without FAXE are illustrated below.

Traditionally using plain Git:

(1) Call "git add −−patch" to view changes.
(2) Git splits changes and displays hunks one by one.
(3) For every hunk in A.java, decide if it belongs to feature1 (feature identification and location). If yes, stage.
(4) Repeat step 4 for B.java and C.java
(5) Specify a commit message and commit changes to feature1.
(6) Repeat Step 1-5 for feature2.
(7) Push.

With FAXE:

(1) Call "git-pfc" to view features contained in the changeset.
(2) Select feature1, enter commit message (optional), commit.
(3) Call "git-pfc –f feature2" to commit changes to feature2.
(4) Push.

4 CONCLUSION

Features describe systems and distinguish them from one another. Knowing their locations in assets is required to effectively reuse and maintain systems. To alleviate feature location costs, we advocate embedded annotations to record feature locations pro-actively in software assets using a flexible and lightweight annotation system. Our tool FAXE extracts features annotations specified in the proposed syntax from software assets. It is language independent, and can be conveniently integrated with IDEs. We also present and automate an interesting use case of FAXE—feature-based partial commits, which allows organizing commits along features. As future work, we aim to use our implementation to drive the development of open source systems. We also intend to conduct usability studies of our tools with developers. Subsequent research can be directed into development of large distributed teams and how to plan feature-oriented software evolution using embedded annotations.

Acknowledgements. We thank Jan-Philipp Steghöfer for feedback, the Wallenberg Academy, and the Swedish Research Council.

REFERENCES

[1] 2020. Embedded Annotations Specification. https://bitbucket.org/easelab/faxe/src/master/specification/embedded_annotation_specification.pdf.
[2] 2020. FAXE Project Repository. https://bitbucket.org/easelab/faxe/.
[3] Hadil Abukwaik, Andreas Burger, Berima Andam, and Thorsten Berger. 2018. Semi-Automated Feature Traceability with Embedded Annotations. In *ICSME*.
[4] Berima Andam, Andreas Burger, Thorsten Berger, and Michel R. V. Chaudron. 2017. FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces. In *VAMOS*.
[5] Kacper Bąk, Zinovy Diskin, Michał Antkiewicz, Krzysztof Czarnecki, and Andrzej Wąsowski. 2016. Clafer: unifying class and feature modeling. *Software & Systems Modeling* 15, 3 (2016), 811–845.
[6] 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*.
[7] Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. 2019. Visualization of Feature Locations with the Tool FeatureDashboard. In *SPLC*.
[8] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *SPLC*.
[9] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my feature and what is it about? A case study on recovering feature facets. *Journal of Systems & Software* 152 (2019), 239–253.
[10] Jacob Krueger, 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 *Twelfth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*.
[11] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2018. *Features and how to find them: a survey of manual feature location.* LLC/CRC Press.
[12] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. 2013. Feature-Oriented Software Evolution. In *VaMoS*.
[13] Leonardo Passos, Rodrigo Queiroz, Mukelabai Mukelabai, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Jesus Padilla. 2018. A Study of Feature Scattering in the Linux Kernel. *IEEE Transactions on Software Engineering* (2018).
[14] Martin Robillard and Gail Murphy. 2007. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology* 16, 1 (2007), 3–es.
[15] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*. Springer, 29–58.
[16] Marcus Seiler and Barbara Paech. 2017. Using Tags to Support Feature Management Across Issue Tracking Systems and Version Control Systems. In *REFSQ*.
[17] Marcus Seiler and Barbara Paech. 2019. Documenting and Exploiting Software Feature Knowledge through Tags. In *SEKE*.
[18] 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* 25, 11 (2013), 1193–1224.