

Visualization of Feature Locations with the Tool FeatureDashboard

Sina Entekhabi*

Middle East Technical University
Ankara, Turkey

Jan-Philipp Steghöfer

Chalmers | University of Gothenburg
Gothenburg, Sweden

Anton Solback

Chalmers University of Technology
Gothenburg, Sweden

Thorsten Berger

Chalmers | University of Gothenburg
Gothenburg, Sweden

ABSTRACT

Modern development processes and issue trackers often use the notion of features to manage a software system. Features allow communicating system characteristics across stakeholders and keeping an overview understanding—especially important for systems that exist in many different variants. However, maintaining, evolving or reusing features (e.g., propagating across variants, or integrating into a platform) requires knowing their locations to prevent extensive feature-location recovery. We advocate the use of embedded annotations, added directly into software assets by the developers during development. To support this process and provide immediate benefits to developers when using such annotations, we present the open-source tool FeatureDashboard. It extracts and visualizes features and their locations using different views and metrics. As such, it encourages developers recording features and their locations early, to prevent feature identification and location efforts, as well as it supports system comprehension.

CCS CONCEPTS

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

ACM Reference Format:

Sina Entekhabi, Anton Solback, Jan-Philipp Steghöfer, and Thorsten Berger. 2019. Visualization of Feature Locations with the Tool FeatureDashboard. In *23rd International Systems and Software Product Line Conference - Volume B (SPLC '19), September 9–13, 2019, Paris, France*. ACM, New York, NY, USA, 4 pages. <https://doi.org/10.1145/3307630.3342392>

1 INTRODUCTION

Features are typically used to describe the functional and non-functional characteristics of a software system. Modern agile development as well as software product-line engineering (SPLE) processes use features to manage single or variant-rich systems. Features

*Sina Entekhabi contributed to the tool during an internship at Chalmers University of Technology, Sweden.

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 '19, September 9–13, 2019, Paris, France

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-6668-7/19/09...\$15.00

<https://doi.org/10.1145/3307630.3342392>

are units of communication, reuse, maintenance, and evolution, and they allow keeping an overview understanding of a system [5–7].

Effectively using features requires knowing their locations within the source assets of a software system. When not recorded, this information needs to be recovered, which is a daunting and error-prone task [15]—despite being one of the most common activities of developers. Notably, the locations of features are often scattered across the codebase [18, 19], further complicating this task. While automated techniques to recover feature locations have been proposed, none of them has found industrial adoption, mainly due to their low accuracy [9, 20], challenging their application in practice.

Using features is especially important in variant-rich systems. Organizations often realize variants through clone&own [8, 10] and propagate features across the cloned variants. When the number of variants challenges system evolution and maintenance, organizations often migrate the cloned variants into an integrated platform, which requires identifying features and their locations, and organizing the features in a feature model.

We take a different route and advocate the use of features early, and embedding feature locations directly into software assets. We rely on a lightweight annotation technique we developed before [2, 12], comprising in-code annotations, file and folder mappings, and a textual feature model in Clafer [3] syntax. Adding annotations during development is cheap, and they naturally co-evolve with the code (reducing annotation maintenance). They were also shown to be beneficial for variant maintenance and platform migration [12] as well as program comprehension [14]. However, standard code editors lack support for such annotations, while it is difficult for developers to extract feature information from large codebases manually. Consequently, using such annotations requires tool support that can extract and visualize the developer-defined features and annotations, supporting browsing and utilizing features.

We present FeatureDashboard,¹ a lightweight tool usable as an Eclipse plugin or standalone program that extracts features from the feature model as well as feature-to-code mappings and embedded annotations, and visualizes these using different views, as demonstrated below. We provide a demo video on the project's website,² showing a walk-through of FeatureDashboard's views.

2 TOOL OVERVIEW

We assume that a developer continuously documents features in a textual feature model (created in the project's root folder) and

¹<https://bitbucket.org/easelab/featuredashboard>

²<https://bitbucket.org/easelab/featuredashboard/wiki/Demo>

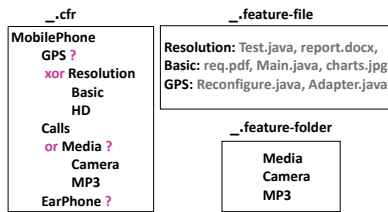


Figure 1: Examples of a textual (Clafer) feature model and mapping files, relating features to files and folders, used in FeatureDashboard

feature locations using in-code annotations as well as file and folder mapping files. The latter allow relating entire files or folders to a set of features (cf. Section 3). We encourage developers to perform this documentation continuously and immediately, when the feature information is still fresh in the developer’s mind. To utilize FeatureDashboard’s views, the developer selects the desired project in Eclipse’s *Project Explorer*, and then opens the FeatureDashboard view. This view is the entry point for the tool, and it allows scanning the selected project’s source tree for the feature model, the mapping files, and the in-file annotations. Once the scan is complete, the FeatureDashboard view represents all features in the project as extracted in the project’s feature model, the annotated files, and the mapping files. By selecting features in this view and opening other, dedicated views of FeatureDashboard, the features are visualized for the developer. The developer can synchronise FeatureDashboard’s view at any time when changes have been made in the project.

Feature location extraction and its visualization simplify common tasks such as change impact analysis where a developer needs to understand which parts of the code need to be changed when a feature is changed. It is also useful when introducing new features to understand the relationship to existing ones and to get a better understanding of the features, where they are implemented, and how they are tangled. FeatureDashboard can thus be used during the entire development lifecycle. FeatureDashboard’s views can be classified into graphs and metrics views, and the majority work on one specific project. FeatureDashboard also offers a view to compare features across projects, which helps to detect inconsistencies (e.g., in the naming of features across projects) or to compare projects based on features.

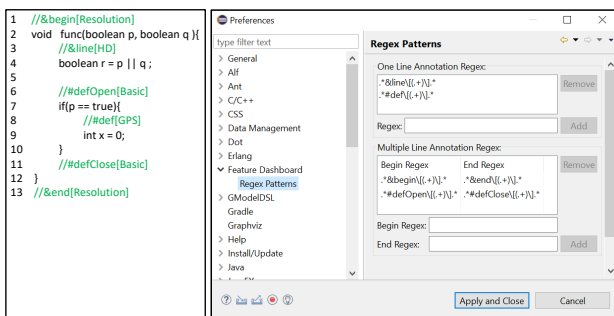


Figure 2: Java code with embedded annotations, whose syntax is customizable with regular expressions

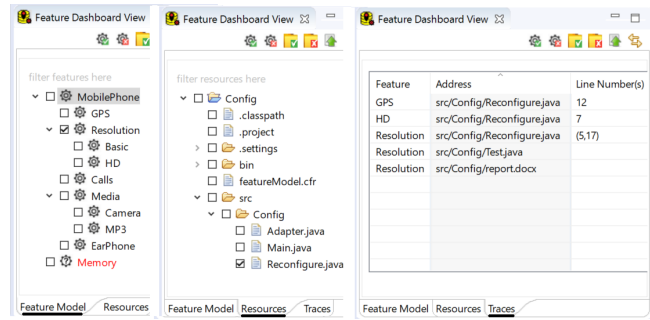


Figure 3: FeatureDashboard view

3 FEATURE DOCUMENTATION

Features and their locations are specified by developers as follows. **Feature Model.** We support feature models expressed in the Clafer syntax [3], having an arbitrary name with *cfr* extension, and stored in each project’s root folder. Currently, only the first file found will be parsed; future tool versions might include the possibility to merge feature models. Figure 1 shows an example. In Clafer, each feature is represented by its name in a single line, with the hierarchy being represented by tab indentation. Feature constraints (e.g., dependencies) can also be expressed, but currently only for documentation reasons (e.g., for a future product-line migration), without any visualization.

Embedded Annotations. Relying on our annotation approach [12], two kinds of annotations exist: for relating a feature to a single line of a file, and for relating a feature to multiple consecutive lines. The annotations are always escaped through the respective language’s *commenting* syntax, to avoid affecting any other tooling (e.g., editors or compilers). Single-line annotations contain the name of exactly one feature. To relate consecutive lines of file to a feature, a pair of begin and end annotations is used in FeatureDashboard, each of which containing the same feature name. In the current version of FeatureDashboard no multiple features supported in one annotation, but it can be easily extended to support that.

In the Preferences of FeatureDashboard, developers can customize the syntax of these in-file annotations with regular expressions. FeatureDashboard recognizes the feature name as the first capturing group. Consequently, each regular expression must contain a capturing group sub-expression like `(.+)`. Figure 2 shows an example with Java code.

Mapping Files. Whole files and folders are mapped to features via simple text files named with the extensions `.feature-folder` and `.feature-file` (to avoid Eclipse hiding them, we suggest using `_` as the file name). A `.feature-folder` file is put into the respective folder and just contains the names of the features (per line) that are intended to be mapped to that folder. The `.feature-file` files are also stored in folders, but contain feature names, separated with a colon from the list of file names (separated with commas) mapped to the feature. Each line of this file maps a feature to some files which exist in the same folder. Figure 1 shows examples.

4 VIEWS

Feature Dashboard View. This view is the main entry point for working with FeatureDashboard. For the selected project it shows

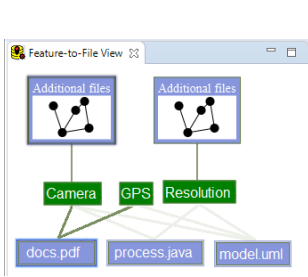


Figure 4: View for relationships of features to files

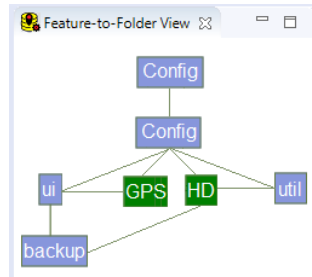


Figure 5: View for relationships of features to folders

the feature hierarchy (from the model) and feature locations. Users can modify the preferences to filter files (e.g., specific file extension or folders). Three tabs are provided: *Feature Model*, *Resources*, and *Traces*. The features and the resources shown within the first two tabs are selectable, and the selections are used in the *Traces* tab.

The *Feature Model* tab shows the selected project’s features along with their hierarchy according to the feature model defined in Clafer syntax in the project. The feature model representation of the model in Fig. 1 is shown in Fig. 3 within this tab. Here, features that are annotated in some source files of the project but not in the feature model are represented in red as root features. In this tab the features can be filtered and selected. Features selected in this tab are used by the tab *Traces* to show the selected features’ locations and by other visualisation views, as explained in the following.

The *Resources* tab represents all files and folders in the imported project with their corresponding hierarchy, and allows filtering and selecting the resources for the user as shown in Fig. 3. Resources selected are used in the *Traces* tab to show the features related to those resources, and by other views of FeatureDashboard as well. Together with the *Feature Model* tab, it is thus possible to focus on relevant features and resources in all views of FeatureDashboard.

The *Traces* tab shows the feature locations of the selected features in the *Feature Model* tab and the selected resources in the *Resources* tab. Fig. 3 shows an example of feature locations in the *Traces* tab for the features and resources selected in the *Feature model* and *Resources* tabs. Each feature location is listed with its feature, the path within the project, and where applicable, the line numbers. If the relation of a feature and a file or folder is stated in a mapping file, the *line numbers* column is empty. The table of traces can be sorted and its data can be exported. In addition, double-clicking on a table row containing an embedded annotation will open the corresponding file and highlight the annotated line or code block.

Feature-to-File View. The relations between features selected in the FeatureDashboard view and relevant files are visualized in this view. Figure 4 shows an example. A connection between a feature (represented by a green rectangle) and a file (blue rectangle) indicates that the relation is established in a mapping file or by feature annotations within the file itself. It is also indicated if multiple selected features are implemented in the same files.

Double-clicking the *Additional files* box will show additional files that a feature is implemented in, but the other features are not. The reasoning behind this is to divide the information into different modules to reduce the elements in a single view. In this view, clicking on a file (docs.pdf in Fig. 4) will highlight the connections, which

Feature	Config	DB	Network
MobilePhone	Green	Green	Green
GPS	Green	Green	Green
Resolution	Green	Green	Green
Basic	Green	Green	Green
HD	Green	Green	Green
Calls	Green	Green	Green
Media	Green	Green	Green
Camera	Green	Green	Green
MP3	Green	Green	Green
Memory	Green	Green	Green
Battery	Green	Green	Green

Figure 6: Common features between projects

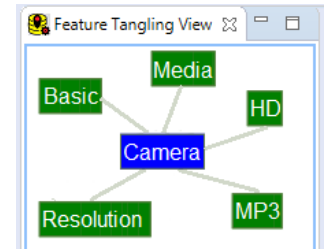


Figure 7: Features tangled with Camera

makes it easier to see which features are implemented in that file. Clicking on a file box will open that file in the editor and highlight the annotated code block.

Feature-to-Folder View. Similar to the *Feature-to-File* view, this view visualizes the relations between features and folders, as shown in Fig. 5. Green and blue rectangles represent features and folders, respectively. The links between a folder and a feature indicate that either the feature is annotated in a file within that folder or the relation is directly mentioned by a mapping file. Hierarchical folder representations are also shown by links between folders.

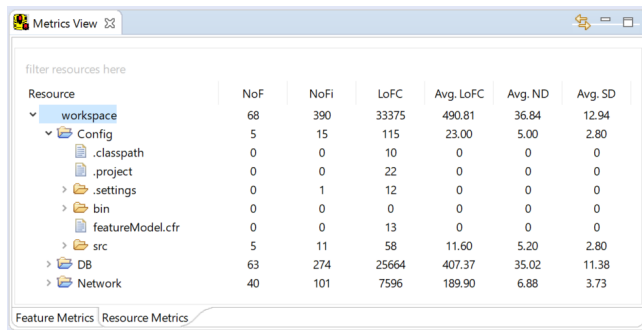
Common Features View. This view visualizes features different projects (or variants) have in common. A matrix that relates all open projects (rows) to all found features (columns) indicates that a feature is contained in a project with a green cell. An example is shown in Fig. 6.

Feature Tangling View. This view visualizes features that are tangled with each other. This is useful for identifying the parts of a system that are potentially affected if that particular feature is modified. It is also possible to see whether some selected features are tangled with each other in the *Feature-to-File* view. The *Feature Tangling* view, however, visualizes all tangled features for a selection. As with the other views, the considered feature must be selected in the *FeatureDashboard* view first. Figure 7 shows the tangled features with feature *Camera*. The blue rectangle in this view represents the selected feature, and the green ones are the features tangled with it. The link between the features can be double-clicked to see more detailed information regarding the two features in the *Feature-to-File* and *Feature-to-Folder* views.

Metrics View. All metrics proposed by Andam et al. [2] are calculated and shown in this view. The *Feature Metrics* tab shows feature metrics for the selected features in the selected projects, as shown in Fig. 8. The *Resource Metrics* tab shows feature-related metrics for different resources and averages for suitable metrics from the *Feature Metrics* tab. If the user has not selected any resources in

Feature	SD	NoFA	NoFoA	TD	LoFC	AvgND	MaxND	MinND	NoAu
Resolution	4	3	0	0	14	2.25	3	3	0
Config	4	3	0	0	14	2.25	3	3	0
etc	4	3	0	0	14	2.25	3	3	0
Config	4	3	0	0	14	2.25	3	3	0
Reconfigure.java	1	0	0	2	13	3.00	3	3	0
Test.java	1	0	0	1	0				0
_feature-file	2	0	0	2	1	3.00	3	3	0
report.docx	0	0	0	0	0				0
BitcoinBalance	48	0	0	37	1226	3.17	5	1	0
SetDefault	9	0	0	15	88	2.78	4	1	0
HD	1	0	0	2	1	4.00	4	4	0

Figure 8: Metrics for features



Resource	NoF	NoFi	LoFC	Avg. LoFC	Avg. ND	Avg. SD
workspace	68	390	33375	490.81	36.84	12.94
Config	5	15	115	23.00	5.00	2.80
.classpath	0	0	10	0	0	0
.project	0	0	22	0	0	0
.settings	0	1	12	0	0	0
.bin	0	0	0	0	0	0
.featureModel.cfr	0	0	13	0	0	0
.src	5	11	58	11.60	5.20	2.80
.DB	63	274	25664	407.37	35.02	11.38
.Network	40	101	7596	189.90	6.88	3.73

Figure 9: Metrics for resources

the *FeatureDashboard* view, metrics for the entire workspace are shown. Figure 9 demonstrates resource metrics in this tab.

History view. An experimental view was created to show how a selected metric for a feature evolved over time. Figure 10 shows how a feature’s lines of feature code (LOFC) have change in successive commits based on the git history. This experimental feature will be developed further in the future.

5 RELATED WORK

The most closely related work is our previous tool FLOrIDA [2], from which *FeatureDashboard* gathers inspiration. However, FLOrIDA could not be released as open source and was a standalone Java Swing application, limiting integration into modern tools as well as extensibility. Both *FeatureDashboard* and FLOrIDA encourage developers to use embedded annotations [12], which can in the future be supported by a recommender system [1]. *FeatureDashboard* provides a superset of FLOrIDA’s functionality, but does not have a built-in, automated feature-location tool (FLOrIDA has one based on the Lucence search engine).

Other works, such as CIDE [13], *FeatureCommander* [11], PEoPL [4, 17], and *ViewInfinity* [21], focus on visualizing variation points and optional features (omitting mandatory features, which are the focus of *FeatureDashboard*). They also focus on visualizing those variation points using different coloring techniques.

6 CONCLUSION

We described *FeatureDashboard*, an open-source tool to extract features and their locations in different artifacts, to calculate useful metrics for features, and to visualize the results. *FeatureDashboard* was designed to support developers recording feature information early and continuously, while obtaining immediate benefits

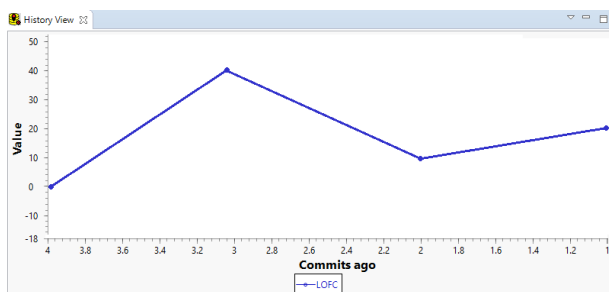


Figure 10: Experimental view showing a feature’s history

through the different views that help browsing feature locations and keeping an overview understanding of the project. *FeatureDashboard* is open for extensions. We consider integrating it with *Eclipse Capra*³ [16], a traceability management tool, which would allow managing and visualizing the traceability between features and many more types of assets, such as requirements, design models, and documentation with non-textual representations. In addition, we are currently evaluating the use of *FeatureDashboard* in industrial settings with our partners with a focus on the visualisations and the usefulness of the metrics.

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 RV Chaudron. 2017. Florida: Feature location dashboard for extracting and visualizing feature traces. In *VaMoS*.
- [3] Kacper Bak, Zinovy Diskin, Michal Antkiewicz, Krzysztof Czarnecki, and Andrzej Wasowski. 2016. Clafer: unifying class and feature modeling. *Software & Systems Modeling* 15, 3 (2016), 811–845.
- [4] B. Behringer, J. Palz, and T. Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *ICSE*.
- [5] 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*.
- [6] Thorsten Berger, Divya Nair, Ralf Rublack, Joanne M. Atlee, Krzysztof Czarnecki, and Andrzej Wasowski. 2014. Three Cases of Feature-Based Variability Modeling in Industry. In *MODELS*.
- [7] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*.
- [8] John Businge, Openja Moses, Sarah Nadi, Engineer Bainomugisha, and Thorsten Berger. 2018. Clone-Based Variability Management in the Android Ecosystem. In *ICSME*.
- [9] 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* 25, 1 (2013), 53–95.
- [10] Yael Dubinsky, Julia Rubin, Thorsten Berger, Sławomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *CSMR*.
- [11] Janet Feigenspan, Maria Papendieck, Christian Kästner, Mathias Frisch, and Raimund Dachselt. 2011. *FeatureCommander: colorful# ifdef world..* In *SPLC Workshops*. 48.
- [12] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining feature traceability with embedded annotations. In *SPLC*.
- [13] Christian Kästner, Salvador Trujillo, and Sven Apel. 2008. Visualizing Software Product Line Variabilities in Source Code.. In *SPLC (2)*. 303–312.
- [14] Jacob Krueger, Gul Calikli, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. In *FSE*.
- [15] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2018. *Features and How to Find Them: A Survey of Manual Feature Location*. Taylor & Francis Group, LLC/CRC Press.
- [16] Salome Maro and Jan-Philipp Steghöfer. 2016. *Capra: A Configurable and Extensible Traceability Management Tool*. In *RE*.
- [17] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. 2018. Multi-View Editing of Software Product Lines with PEoPL. In *40th International Conference on Software Engineering (ICSE), Demonstrations Track*.
- [18] 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 *MODULARITY*.
- [19] 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). Preprint.
- [20] Julia Rubin and Marsha Chechik. 2013. A survey of feature location techniques. In *Domain Engineering*.
- [21] Michael Stengel, Mathias Frisch, Sven Apel, Janet Feigenspan, Christian Kästner, and Raimund Dachselt. 2011. *View infinity: a zoomable interface for feature-oriented software development*. In *ICSE*.

³<https://eclipse.org/capra>