FeatureVista: Interactive Feature Visualization

Alexandre Bergel
Department of Computer Science (DCC), University of Chile, Chile

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

Razan Ghzouli
Chalmers | University of Gothenburg Sweden

Michel R. V. Chaudron
Technische Universiteit Eindhoven Netherlands

ABSTRACT

Comprehending and characterizing the spread and interaction of features in a software system is to know to be difficult and error-prone. This paper presents FeatureVista, a lightweight tool providing interactive, glyph-based, and iconic visualization concepts designed to visually characterize the feature locations in software assets (source code). FeatureVista supports navigating between software components and features in an equal fashion. Our pilot study indicates that FeatureVista is intuitive and supports comprehending features. It helps to precisely characterize relations among features in large software systems and to contrast explicit software component definitions (e.g., package, class, method) with annotated feature portions—which so far was a largely manual and error-prone activity, albeit essential to get an adequate understanding of a software system. We suggest research directions for true, feature-oriented interfaces that can be used to manage software assets.

CCS CONCEPTS

- Software and its engineering → Object oriented development; Software maintenance tools.

ACM Reference Format:

1 INTRODUCTION

Software is often built around the notion of features—abstract entities describing the functional and non-functional aspects of a software system [15]. Modern agile processes, including feature-driven development, SCRUM, and XP, often rely on features to plan and manage software. In product-line engineering [3, 16, 29], features represent the common and variable aspects of individual products, typically organized in a feature model [20, 32]. Other common software entities (e.g., files, classes or components) are usually confined to abstraction boundaries enforced by a technological space. However, in reality, rather few concerns adhere to abstraction boundaries when implemented—which features are not confined to [6, 34] and therefore provide a unique perspective on software systems.

Explicit feature representations are known to improve the comprehension and maintainability of software systems [19, 24, 26, 27, 33, 38, 43]. However, features are often scattered [35] across software assets, which hampers their comprehension—especially understanding the relationship between features and software assets (e.g., packages, classes, methods), as well as understanding the interaction between features [6]. Given this challenge, modularizing features has long been one of the holy grails in research, as witnessed by the proposal of different modularization technologies [4, 5, 7, 9–11, 23, 37, 39, 40], which are often specific to a programming paradigm (mostly object-oriented programming) or even a programming language. However, modularizing features and adopting such a modularization technology requires substantial overhead and changes developers’ workflows [25]. We take a different stance and advocate that developers declare features with lightweight techniques (e.g., embedded annotations [19, 30, 41, 42], explained shortly) and use novel ways of interacting with the software assets via features.

We envision an interactive way of interfacing with assets via features. In this paper, we work towards this vision by providing interactive and feature-oriented visualization concepts that are related to (and also visualize) object-oriented software structures. We adopt concepts from the visualization community and provide an integrated, feature-oriented view on object-oriented programs, together with interactive navigation facilities. We hope that these visualizations provide a basis to eventually create more modern ways of interacting with assets, establishing features as pivotal entities that bridge the gap between domain experts and developers.

We propose a novel, interactive visualization called FeatureVista. It supports comprehending a complex feature-annotated software system by contrasting feature definitions with an explicit, object-oriented software structure. Navigation is supported through a navigation technique that equally balances features and structural components. While this short paper aims at demonstrating the concepts, we report on an early a pilot study with a software engineer. The engineer was able to complete a set of tasks related to system comprehension about features and their—potentially scattered—locations in the system, whose object-oriented structures it also visualizes.

We contribute interactive feature-oriented visualization and navigation concepts, implemented in the tool FeatureVista. We provide a demonstration video [1]. The feature-annotated dataset of our running example (explained shortly) is also available online.1

1https://bitbucket.org/easelab/datasetbitcoinwallet
2 FEATUREVISTA
We now discuss the main visualization concepts of FeatureVista. Its current implementation realizes a visualization of a codebase where features are defined in a textual feature model and their locations directly represented as embedded annotations in the source code following the FAXE syntax [41]. As a running example we use a popular open-source Android app: Bitcoin Wallet, a popular implementation of a cryptocurrency wallet. It is as large as 50 KLOC spread over 269 Java classes and interfaces. The app comprises 99 features organized in a feature hierarchy of depth 5. The features were manually declared and annotated in the Bitcoin Wallet app in previous work [27, 28].

2.1 FeatureVista in a Nutshell
Figure 1 gives an overview of the visualization of FeatureVista on a version of the Bitcoin Wallet app annotated with feature annotations. Since FeatureVista is interactive, we recommend watching an online video that demonstrates the visualization [1]. FeatureVista is made of some visual and connected panes, detailed below.

Initial pane. Pane A is called the initial pane and represents the initial visualization of FeatureVista. The initial pane gives an overview of the feature model and the code base, and it shows where a program understanding activity begins from. The feature model, represented as a tree, is given on the left-hand side (C in Figure 1). Large gray boxes (D) indicate packages contained in the Bitcoin Wallet app. Inner boxes (E) represent classes and interfaces of the app.

Iconic visual representation. FeatureVista employs numerous techniques, called iconic visual representation [21], that map visual attributes to multidimensional data elements. The initial visualization contains various such iconic visual representations. For example, the feature model (C) maps the font size used to write the feature name with the number of classes the feature contributes to. A feature that contributes to many classes is written using a large text font while a feature that contributes to none or a few classes uses a small text font. Classes are represented as inner-boxes (E) and their size reflects the number of lines of code that define the class.

Interactivity. Visualizations offered by FeatureVista are highly interactive and provide numerous ways to highlight and focus on a particular set of features and/or source code units. Each feature shows in the feature model (C) acts as a switch for which a practitioner can activate by simply clicking on it. As illustrated in Figure 1, the feature Codecs is highlighted in yellow, the feature ExchangeRates is in red, and BlockchainSync in green.

Class inspection. A class (i.e., inner boxes, E) may be selected for further inspection by left-clicking on it. A selected class is surrounded by a thick blue border and a new pane appears on the right to inspect some properties of the class. In Figure 1, a class is selected, near the E mark, and its selection had the effect to open pane B.

https://github.com/bitcoin-wallet/bitcoin-wallet
The B pane provides various relevant information about the class, including source code, metrics, and the dependency visualization, which is shown in Figure 1 (due to space restriction, only the dependency visualization offered by the class inspector is detailed in this paper). For the selected class (TransactionsAdapter in the figure, marked with G) the dependency visualization shows classes that depend on it (i.e., incoming classes, marked with F), and classes that the selected class depends on (i.e., outgoing classes, G). The class selected in pane A is at the center of pane B represented using a class glyph (described below). Incoming classes are located on the left and outgoing classes are located on the right of the inspected class. The dependency visualization also lists the features (I) that contribute to the selected class.

**Feature inspection.** When a feature is selected in (i) the feature model (C) provided by the initial pane (A) or (ii) the small panel that lists features that contribute to a class (I), a feature inspector is open as a new pane.

By clicking on FeatureWallet, located in C, the B pane is replaced by the J pane, as shown in Figure 2. The feature inspector lists the classes that are contributed by the selected feature. These classes are ordered along their size (in terms of lines of code).

**Navigation.** FeatureVista allows one to navigate through the source code either by following dependencies between classes and between features. Each class or feature can offer an inspector by clicking on it. The navigation follows a cascading list technique (also called Miller columns, adopted by Finder on macOS). Selecting one element opens a new pane or replaces the one located on the right.

Figure 3 illustrates a chain of four panels, starting from the initial pane, J, located on the left-most side. In J a feature is selected, which opens the feature inspector in K. A class is selected in K, which opens a class inspector in L. A feature is selected in L, which opens a new feature inspector in M.

**Class glyph.** A class is represented as a vertical gray bar. The height of the bar indicates the number of lines of code. For example, in Figure 1, the class TransactionsAdapter (G) is higher than all the classes that it depends on (H) since its representing gray box is higher. One of the incoming classes is however larger.

---

**Figure 2: Feature inspector**

**Figure 4: Class glyph**

Annotations are represented by superposing smaller boxes. Horizontal offset indicates nesting of the annotations. Consider Figure 4. The figure illustrates how the class glyph is generated from a small code excerpt. The code portion belongs to the feature ExchangeRates, marked by the FAXE annotations [41]. Since the feature ExchangeRates is selected in Figure 1, C, this annotation is highlighted in red in the class glyph. The code portion contains other annotations, OwnName, QRCode, and Generation. Since none of these features are selected in C, they are painted in dark gray in Figure 4, the default color for non-selected features. Annotations can be nested, as such, Generation is within QRCode, itself contained in ExchangeRates. The depth of the nesting is visually indicated with a translation to the right. The class glyph is used to represent each class in the represented software system.

### 2.2 Visual Properties

Our visualizations cover the following comprehension use-cases.

**Features locations.** Features may be selected in the feature model (C). Selecting a feature underlines classes that are contributed by the selected feature. Several features may be selected to reveal their locations in terms of their class contribution.

The feature model in Figure 1, marked with C, have three selected features, Codecs, ExchangeRates, SendCoins, and BlockchainSync. The contribution of each of these features is highlighted in the classes (D) by underlining these classes. Each feature is associated to a color, and the underline uses the feature color. For example, we see a large class that is contributed by three features, located near by the mark D, Figure 1. Overall, we deduce many points of interaction between these features across the classes composing the system analyzed.

**Context scoping.** In the initial pane (A in Figure 1) features may be selected in the feature model (C). Selected features define a context that is kept along the navigation. Whenever one navigates through the system by selecting features or classes, only the feature that were initially selected are highlighted. Other features are painted in dark gray. For example, Figure 3 shows a chain of four panes.
Features that are highlighted in panes K, L, and M are the ones selected in pane J.

**Contribution to a particular class.** Moving the mouse cursor above a class in the initial pane (A in Figure 1) highlights the features that contribute to the class indicated by the cursor.

Figure 5 illustrates the highlight triggered by moving the mouse cursor above the class named WalletTransactionsFragment. In addition to displaying a little popup window that indicates the class name, number of fan-in and -out, all the contributing features are highlighted. Each feature has a color that is determined from a linear distribution across all the leave features of the feature model.

Since contributing features are highlighted by simply moving the mouse cursor, a very low-effort action, a practitioner can browse the system by simply moving the mouse around to immediately see the features contributing to a particular class.

### 3 PILOT EVALUATION

We evaluated whether FeatureVista supports the ability to form a comprehension view of the different software components (e.g., package, class) and features by designing a formal pilot experiment [22]. In the following, we describe our experiment design and report preliminary results.

**Pilot Design.** The pilot was guided by three research questions:

**RQ1.** Can FeatureVista support developers when identifying features with characteristics of their relationship with different software components and metrics?

**RQ2.** Can FeatureVista support developers when identifying software components with characteristics of their relationship with features and software metrics?

**RQ3.** Does FeatureVista help developers understanding the relationships (e.g., feature interaction) between the different features?

With RQ1 and RQ2 we aim to evaluate our tool ability to provide a comprehensive view of the different software components and features and inspect their characteristics. With RQ3 we want to investigate the extent in which our tool provides a support to understand the feature relationships. Using the running example in Section 2, we created nine task-based questions to answer our RQs, see table 1. The questions were shared with a software engineer, which we asked to also share how she used the tool to answer each question and her opinion about the tool. To prepare the engineer for using the tool a demo video that briefly describes the tool was shared with her. The data collected from the participant was qualitative one combining her task answers and a free-text to reflect on each task experience. We used Miles and Huberman [31] qualitative data analysis method with thematic analysis [44] to identify and analyze a theme patterns in the provided answers. We were able to group questions’ answers by improvement comments.
Related Question | Question
--- | ---
RQ1 | Q1. Which feature is the largest in the system, in terms of classes?
Q2. Which feature is the largest in the system, in terms of packages?
Q3. Which feature is the largest in the system, in terms of lines of code?

RQ2 | Q4. Which class is the most diverse in terms of features?
Q5. Which class is the largest, in terms of lines of code?
Q6. Which packages is the largest, in terms of number of classes?

Q7. Is there an interaction between feature A and feature B on the class level?
Follow-up if yes: Q7.1 How many classes contribute to both feature A and B?
Q8. How do features interact in class X?
Q9. How is the feature C spread along the dependency graph, of the class X?

Pilot Results. The results from the participant showed the ability to use FeatureVista to answer all the tasks correctly. An overall satisfaction for using the tool was reported and it was described that the tool was intuitive and easy to use for forming a comprehensive understanding of the Bitcoin software features and software components. The following comments for improvement were given. For Q3 and Q5, she reported that the intuitive visual representation that reflects using the size of an element (class or feature) what is usually a code level metric makes it easy to spot this information. However, one concern was raised in case two elements’ size looked pretty similar in the visual inspection. The same comments were given to Q1, where the size of feature font was the used indicator. In Q2, Q4 and Q6, by navigating pane A and B, refer to (A and B in Figure 1), the participant was able to easily spot the answers. But for Q2, an additional interaction was needed with the feature model in pane A by selecting each feature and counting the highlighted packages. The same applied to Q4 but using pane B to extract the information. It was highlighted for both questions that it could be cumbersome for users with bigger feature model to extract the information. For Q7, Q8 and Q9, the participant was able to identify the feature interactions and their spread along different dependent classes (Q9) using the tool interactivity and the different views it provides (e.g., dependencies tab, classes tab). The participant described that the tool gave her an overall understanding of the features interactions.

4 RELATED WORK

Various prior works proposed visualization technique to assist practitioners in understanding feature location and feature interactions. Greevy et al. [18] combine static information and dynamic behaviors by providing a 3D visualization to assess execution traces. Urli et al. [46] proposed variability blueprint to visualize large feature model. Their blueprint represents the visualization of constraints between features. Entekhabi et al. [17] and Andam et al. [2] propose dashboards to visualize features and their locations and to ease browsing them, for which they rely on embedded feature annotations as we do [19, 41]. For such annotations, Martinson et al. [30] provide an Intellij plugin helping to write such annotations during development and assuring their consistency. Pleuss et al. [36] propose a visual approach to assist practitioners to configure systems with the help of feature models. Trinidad et al. [45] develop a 3D visualization to draw large feature models. Such representation has the benefit over 2D representation to reduce the amount of necessary scrolling.

FeatureVista was designed by considering the experience of these previous works [8, 12, 18, 46], but has novel concepts to navigate and assess the mapping between features and a base source code.

5 CONCLUSION AND DIRECTIONS

This paper contributes to the state of the art in understanding feature interaction using an interactive visualization. FeatureVista produces an interactive visualization from an annotated source code to explore features' location with respect to other software components and their interaction. The results of our pilot experiment show that FeatureVista is easy to use and supports forming a comprehensive understanding of feature interactions and location. A natural next step is to empirically assess FeatureVista by designing and applying a mixed-method evaluation. In particular, we plan to collect and contrast results of a narrative data (e.g., think aloud protocol) with numerical data (e.g., metrics describing performance to complete well defined tasks).

In the longer-term, towards evolving these feature-oriented visualizations into a real interfaces that can be used to manage the underlying (object-oriented) assets, we envision the following directions for future work.

FeatureVista has been designed for Java, but it is largely applicable to other programming languages, even in presence of particular notion of component (e.g., traits as in Scala and Pharo [14], class extensions as in Swift [13]).

Feature-Oriented Workflows. While agile engineering methods are focused around the notion of features, managing assets via them is still an open problem. We need to understand and establish efficient workflows [25] that are realizable in visual, feature-oriented interfaces, ideally building upon those that FeatureVista provides.

Feature-Oriented Views. Feature-orientation helps managing complex software systems. However, to support working on individual features or subsets of features, we need to establish effective filtering mechanisms for the visualizations proposed by FeatureVista. This needs to address the problems of re-integrating (i.e., considering the view-update problem) edited views.

Visualizing Feature-Oriented Evolution. Developers often need to understand the past evolution of a software system to effectively evolve it. FeatureVista’s visualizations only show a snapshot of a system. Conceiving effective ways to visualize the prior evolution in terms of features is an open problem, calling for proposals from the community.

ACKNOWLEDGEMENTS

Bergel thanks the ANID Fondecyt Regular project number 1200067. Ghzouli and Berger thank the Knut and Alice Wallenberg Foundation as well as the Swedish Research Council.
REFERENCES

demo
[40] Ina Schafer and Ferruccio Damiani. 2010. Pure delta-oriented programming. In ICSD.