Where is my Feature and What is it About? A Case Study on Recovering Feature Facets

Jacob Krüger[a,b,*], Mukelabai Mukelabai[d], Wanzi Gu[c], Hui Shen[c], Regina Hebig[d], Thorsten Berger[d]

[a] *Otto-von-Guericke-University Magdeburg, Germany*
[b] *Harz University of Applied Sciences Wernigerode, Germany*
[c] *Chalmers University of Technology, Sweden*
[d] *Chalmers | University of Gothenburg, Sweden*

Abstract

Developers commonly use features to define, manage, and communicate functionalities of a system. Unfortunately, the locations of features in code and other characteristics (feature facets) relevant for evolution and maintenance, are often poorly documented. Since developers change and knowledge fades with time, such information often needs to be recovered. Modern projects boast a richness of information sources, such as pull requests, release logs, and otherwise specified domain knowledge. However, it is largely unknown from what sources features, their locations, and their facets can be recovered. We present a case study on identifying such information in two popular, variant-rich, and long-living systems: The 3D-printer firmware Marlin and the Android application Bitcoin-wallet. Besides the available information sources, we also investigated the projects' communities, communications, and development cultures. Our results show that a multitude of information sources (e.g., commit messages and pull requests) is helpful to recover features, locations, and facets to different extents. Pull requests were the most valuable source to recover facets, followed by commit messages, and the issue tracker. As many of the studied information sources are, so far, rarely exploited in techniques for recovering features and their facets, we hope to inspire researchers and tool builders with our results.

Keywords: Feature location; Marlin; Bitcoin-wallet; case study; feature facets; software product line

1. Introduction

Features are commonly used to specify, manage, and communicate the functional and non-functional properties of a software system. Features support developers in comprehending, reusing, and adapting these systems [Apel et al. 2013; Kang et al. 1990]. As such, features are useful entities to support software development, maintenance, and evolution [Berger et al. 2015; Passos et al. 2013]. Yet, features are often poorly documented, including their locations in the source code, but also many other *facets* [Berger et al. 2015] that are relevant for evolving and maintaining them, such as the responsible developer, binding time, rationale (i.e., why a feature is introduced) or architectural responsibility of the feature. When a system evolves over time, the knowledge about features, their facets, and their locations often fades and has to be recovered [Ji et al. 2015; Krüger et al. 2018c]—an activity known as *feature location*. In fact, feature location [Assunção and Vergilio 2014; Assunção et al. 2017; Dit et al. 2013; Lozano 2011; Rubin and Chechik 2013] is one of the most common and expensive activities in software engineering [Biggerstaff et al. 1993; Ji et al. 2015; Poshyvanyk et al. 2007; Wang et al. 2013].

Several automated techniques have been proposed to recover features and their locations [Dit et al. 2013; Olszak and Jorgensen 2011; Razzaq et al. 2018; Rubin and Chechik 2013]. Unfortunately, these techniques generally exhibit a low accuracy, need substantial effort (e.g., calibration and adaptation for specific projects), and often only exploit a single source of information source, such as execution traces or code comments. Other feature facets, such as the rationale or architectural responsibility, are even more difficult to extract, as corresponding information sources are largely unknown and developers may have varying understandings of these facets.

To improve techniques for feature location and for recovering feature facets, we need to improve our empirical understanding of features. This includes knowledge about information sources we can utilize for these purposes, about strategies to exploit these information sources, and about the facets of features. Particularly interesting are *modern open-source projects* that are developed on software-hosting platforms such as GitHub and BitBucket, which provide additional capabilities for maintaining and documenting a project. Such platforms boast a richness of different information sources (e.g., pull requests, change logs, release logs,

*Corresponding author
Email addresses: jkrueger@ovgu.de (Jacob Krüger), mukelabai.mukelabai@cse.gu.se (Mukelabai Mukelabai), wanzi@student.chalmers.se (Wanzi Gu), huish@student.chalmers.se (Hui Shen), regina.hebig@cse.gu.se (Regina Hebig), thorsten.berger@chalmers.se (Thorsten Berger)

commits, Wikis, issue tracker) from which such information can be recovered—and that can be present in similar form in industrial settings. Furthermore, realistic datasets of feature locations and feature facets are necessary to test, evaluate, and compare corresponding techniques.

We present an exploratory study of identifying and locating features and their facets in two open-source systems: The 3D-printer firmware Marlin and the Android app Bitcoin-wallet. Both systems exhibit characteristics of software product lines, relying on established variability mechanisms (C preprocessor and runtime parameters, respectively) [Apel et al. 2013; Gacek and Anastasopoules 2001] to allow customization. As Marlin and Bitcoin-wallet are hosted on GitHub, they exhibit a richness of different and varying information sources we can explore. Moreover, the specific culture, processes, and communication styles are important to understand and can also be exploited for recovering features and their facets. After analyzing these specifics, we performed manual feature identification and location based on similar patterns for each information source. We investigated various feature facets that help to comprehend features and that are relevant for maintaining and evolving features.

Overall, our contributions comprise:

- an analysis of the development process of the open-source systems Marlin and Bitcoin-wallet;
- a set of consolidated search patterns to identify and locate features;
- empirical data on the facets of the identified features in both systems; and
- an online appendix[1] containing our feature fact sheets, feature models, and the annotated code bases.

We provide insights into the development of open-source software that comprises several levels of variability, ranging from cloning over preprocessor directives to runtime parameters. Our results show that different information sources can be exploited to varying extents to locate features and to identify their facets. Specifically, we find that pull requests are the source that helped us most to recover different feature facets, followed by commit messages, and the issue tracker. Only few sources have a rather narrow usefulness to obtain information, but these often document specific facets well. For example, due to the development process applied for Marlin, we could use the contributor list and commit author information only to identify developers that are responsible for a specific feature, but this facet can easily be extracted from these sources. We did not find any information source that was not useful, but the extraction effort differed depending on its usage in a project, and whether it is updated. For instance, it is challenging to extract facets from a contributor's GitHub page. Furthermore, we usually needed to include different information sources into our analysis to recover all facets of a feature. For instance, we found that some mandatory and optional features that are bound during implementation and build-time, respectively, may comprise further runtime variability that is not explicitly documented on the same level as other features, but is discussed in commit messages. Consequently, solely analyzing preprocessor directives may neglect not only mandatory features, but also dynamic variability, potentially biasing the results. After investigating Bitcoin-wallet, we also identified the reverse pattern: Features seem to be dynamically bound and configurable at runtime, while the checked parameter is a constant set at compile time. Finally, our results illustrate the importance of considering different feature facets.

An earlier version of this article appeared as a conference paper [Krüger et al. 2018b]. There, we reported a case study on manually locating 43 features in Marlin. We (i) explored what information sources help identifying and locating features, and (ii) compared characteristics of optional and mandatory features. For instance, we found that optional and mandatory features exhibit different characteristics, which challenges the validity of studies that derive conclusions for mandatory features based on analyzing optional ones. In this article, we focus on information sources for feature location and investigate to what extent we can use these sources to recover different feature facets. We applied our methodology not only to Marlin, but also to a second system, namely Bitcoin-wallet. In contrast to Marlin, Bitcoin-wallet follows a less structured development process and comprises more dynamic variability—challenging the identification of feature locations and facets—but we can rely on similar information sources. Finally, we also provide more details on Marlin, specifically its feature development process, and report in detail what search strategies we applied while recovering feature locations.

2. Feature Facets

A software feature is a relatively abstract and vague concept. Consequently, it is not surprising that several notions of features exist [Apel et al. 2013; Classen et al. 2008]. Berger et al. [2015] provide a list of different feature facets, relevant for describing features in their full richness. They also describe rationales and example values for these facets that are derived from interviews with industrial practitioners.

In the following, we briefly describe those facets that comprise information connected to developing features. Foremost, we focus on the *nature* (i.e., optional or mandatory) and on the *binding time and mode*, which define what features are included in what way into a variant. The other facets are important to scope features and to manage development tasks.

2.1. Facet: Nature of a Feature

This facet describes whether a feature primarily represents a unit of variability (optional) or a unit of functionality (mandatory). In particular, distinguishing between these

[1] `https://bitbucket.org/rhebig/jss2018/`

notions is important in the context of software-product-line engineering: While optional features allow to customize a variant, the intended benefits of reuse are heavily driven by mandatory features that are part of every variant. As software-product-line engineering is often focused on the notion of variability, this seems to neglect important parts of such systems.

Features as Units of Variability. In software-product-line engineering, features are primarily seen as units of variability, due to the widespread use of annotation-based variability mechanisms—usually conditional compilation (e.g., #ifdef) [Apel et al. 2013; Medeiros et al. 2015]. We display a code snippet from Marlin in Listing 1, in which the preprocessor macro `NOOZLE_PARK_FEATURE` represents a variation point for an optional feature that is excluded if this macro (configuration parameter) is disabled. To represent code-level dependencies and to foster automated configuration, optional features are often declared in a variability model [Berger et al. 2013; Czarnecki et al. 2012; Nadi et al. 2015], which is usually the input for a configuration tool (in contrast, Marlin relies on configuration files). Given the availability of many open-source systems that comprise such optional features, several studies on these features' code-level characteristics have been conducted, resulting in extensive knowledge about some of their facets [Apel et al. 2013; Berger and Guo 2014; Liebig et al. 2010; Lillack et al. 2019; Passos et al. 2015].

In this notion of variability, the annotated feature locations only represent variable parts, while any mandatory code that also belongs to a feature is not annotated. Consequently, locating variable code is simple, but recovering and distinguishing the locations of mandatory parts is difficult and costly. Likewise, completely mandatory features may not be represented in the variability model. Overall, this notion is useful if features are only used as configuration parameters, but not if features shall also be used to plan the development, to communicate, to maintain the system, to fix bugs, or to re-engineer the system, among others.

For example, consider the re-engineering of cloned products into a software product line [Dubinsky et al. 2013; Krüger et al. 2017; Stănciulescu et al. 2015], specifically, consider a single feature that is cloned among two variants, and slightly modified in one variant. If the feature is integrated into a common platform, only the differences will be annotated (likely, a new feature representing this variability is introduced). The actual location of the whole feature is not annotated and needs to be recovered to allow correct configuring and to facilitate maintenance and evolution.

Features as Units of Functionality. A broader notion of features is to consider them as units of functionality. In this notion, a feature represents a functionality (or concern) in a system, regardless of whether it is an optional or mandatory functionality of a software product line. This notion of features is more common in industrial software engineering [Berger et al. 2015] and in research on concern

Listing 1: Preprocessor code in `Marlin_Main.cpp`.

```
#if ENABLED(NOZZLE_PARK_FEATURE)
/**
* G27: Park the nozzle
*/
 inline void gcode_G27() {
// Don't allow nozzle parking without homing first
  if (axis_unhomed_error()) return;
  Nozzle::park(parser.ushortval('P'));
 }
#endif // NOZZLE_PARK_FEATURE
```

location [Eaddy et al. 2008; Figueiredo et al. 2009; Robillard and Murphy 2007]. Mandatory features and their locations are rarely documented, for example, with feature-traceability databases [Robillard and Murphy 2003] or embedded feature annotations [Ji et al. 2015], which is why recovering their locations is costly and error-prone [Krüger et al. 2018a; Wang et al. 2013]. Even automated or semi-automated feature-location techniques require substantial manual effort (e.g., to calibrate them to a system or for providing so-called seeds from which they start exploring) and fall short in accuracy [Abukwaik et al. 2018; Rubin and Chechik 2013]. Furthermore, they often consider only a single information source (e.g., code comments), while it is unclear which other information sources can be utilized for systems rich in meta-data, such as projects developed in version control systems, with potentially relevant information in issue trackers, pull requests, or Wiki pages.

2.2. Facets: Binding Time and Mode

The binding time of a feature refers to the point in time a feature is included into the system [Berger et al. 2015], such as at implementation, compile, build, load, or run time [Kang et al. 1990; Lee and Muthig 2006; Rosenmüller 2011]. Binding mode refers to the ability to re-bind features at runtime, where we distinguish between static (a bound feature cannot be re-bound during program execution) and dynamic binding (a feature can be re-bound while the program is executed). Considering the natures of features, the question arises if a dominating variability mechanism (i.e., preprocessor directives) may comprise dynamic variability that is not obvious to researchers—thus, impacting the results for optional (e.g., not all variability is annotated) as well as mandatory (e.g., unawareness of dynamic variability) features. This is especially interesting as the software-product-line engineering community performs analyses based on static preprocessor annotations, which may not capture the whole scope of variability. For example, some features in Marlin comprise these types (not annotated and unaware) of dynamic variability based on runtime parameters to react to different input values that depend on the printer's hardware and context. In contrast, most features in the Bitcoin-wallet seem to be mandatory and can be rebound at runtime by the user. However, some of these features only appear to be dynamic because an if statement checks a parameter: Instead, these parameters are constants that developers define before compiling the application and, thus, the features cannot be changed and are, in fact, optional and static.

Static Binding. By using static binding, the features of a software product line are bound before a concrete variant is executed. Thus, the variability is resolved and the variant is customized before it is deployed. In practice, especially the C preprocessor (cf. Listing 1) is used to implement static binding. The C preprocessor relies on annotations to mark features and remove them during a preprocessing step and, thus, bind each feature even before compilation. Afterwards, an instantiated variant can only be changed by reconfiguring and recompiling it.

Dynamic Binding. In contrast to static binding, dynamic binding binds feature only when a program is started or during its execution. This allows the developers to react to changing demands while the program is running. Usually, this is implemented by using runtime parameters that can be set by the program's user and are checked in the control flow. However, there are also more advanced techniques for dynamic binding, which led to the introduction of dynamic software product lines that are focused on reacting to changes in the program context [Capilla et al. 2014].

Static and Dynamic Binding. Static as well as dynamic binding have pros and cons, which make them more suitable for different application scenarios. Consequently, several techniques aim to combine them at different points in time. Such techniques include [Rosenmüller 2011]:

- Early (static) and late (dynamic) binding in object-oriented programming;
- Combined usage of different variability mechanisms (e.g., preprocessors and runtime parameters); and
- Integrations of both binding times into one variability mechanism (e.g., for feature-oriented programming [Prehofer 1997]).

In particular interesting for our work is the second example. We investigated to what extent static and dynamic binding are present in parallel within Marlin and Bitcoin-wallet that appear to have predominant binding modes: Static and dynamic, respectively.

2.3. Other Facets

Besides the facets **nature**, **binding time**, and **binding mode**, we also investigate what information sources in Marlin and Bitcoin-wallet can help to identify the following facets that are relevant for developing features [Berger et al. 2015]: First, the **rationale** describes why a feature has been developed, for example, due to customer requests or platform adaptations. Consequently, this facet defines the purpose and requirements connected to a feature. Second, a feature's **architectural responsibility** describes how a feature is connected to the system's architecture, for example, to the application logic or user interface. Thus, this facet provides a clue about the architectural parts affected by a feature. Third, the **definition and approval** facet captures how the feature has been defined and approved to be included into the system, for example, during workshops or comparisons to existing products. This is particularly interesting to understand the development processes (e.g., quality assurance) of features. Fourth, **responsibility** is concerned with the developers that manage a specific feature. Based on such information, tasks may be assigned or experts identified. Fifth, the **evolution** of features is important to see their changes over time. In particular, this can help to identify features that are regularly changed or may require an update, due to a longer cycle without updates. Finally, in the **quality and performance** facet, non-functional characteristics of a feature are captured. Thus, it can be ensured that requirements—other than functional ones—are fulfilled and tested appropriately. We focus on these facets, as they define the development processes that are applied to a system, for example, which developer updates or tests a feature.

3. Study Design

To efficiently engineer features for long-living, variant-rich systems developed by a larger community or team, it is necessary to record many different information that align to these features. Such information allow to evolve and maintain a feature in a consistent and documented way. In addition, the information can clarify communications, identify responsible developers, and describe a feature's origin, reasoning, and evolution, allowing developers to coordinate their tasks. Despite their importance, these information are often not recorded and, thus, need to be recovered. Consequently, the question arises, *what information sources are available and suitable for this task?*

We conducted a case study on Marlin, a variability-rich 3D printer firmware, and Bitcoin-wallet, an Android application for Bitcoins, that both comprise several information sources and variability mechanisms. Precisely, we report our analysis of the corresponding communities, our feature location process, our search patterns, and the used information sources for feature facets. Analyzing the communities and their development processes as well as defining search patterns for information sources can help to improve techniques for feature location and facet identification. Thus, the results can help to automate the analysis of modern software systems, for example, for maintenance or re-engineering activities.

3.1. Research Questions

We define the following four research questions:

RQ$_1$ *How are features developed in Marlin and Bitcoin-wallet?*
We studied the feature-development processes as exercised by the Marlin and Bitcoin-wallet developers. Both systems are developed on a project-hosting platform that documents and tracks many information on the development of the software and captures discussions, documentations, or bugs. Marlin is particularly

interesting, due to its larger contributing community, which can provide insights into the usage and communication of features and their facets. Thus, systems like Marlin allow us to gain insights into the practices of developing features in a larger team, providing details on the design, development history, and quality assurance. In contrast, Bitcoin-wallet is driven by a single developer, who is sparsely supported by other developers. Understanding such processes helps to consolidate current or best practices, identify potential for improvements or automation, and puts our research into context.

RQ$_2$ *What information sources help to locate features to what extent?*
Automated feature-location techniques usually exploit a single information source, such as the source code or requirements documents. For Marlin and Bitcoin-wallet, several modern information sources, for instance, release logs, issue trackers, and pull requests, exist. We systematically analyze which of these sources can facilitate the task of recovering feature locations. To this end, we focus on the differences between optional and mandatory features, as especially mandatory features are challenging to locate (cf. Section 2.1) and identify variations in the dominating binding mode of each system (cf. Section 2.2).

RQ$_3$ *What search strategies help to recover features?*
This work is driven by manually analyzing the Marlin and Bitcoin-wallet communities and systems. As we adapted similar search processes for each information source, we consolidate these into common patterns. Such patterns help to scope further automation for recovering features and their locations. Furthermore, they can be used by researchers and practitioners to analyze other systems.

RQ$_4$ *What information sources help to identify feature facets and to what extent?*
Feature location techniques usually neglect the rich set of information sources that is available in modern project-hosting platforms. The same accounts for other recovery techniques that focus on specific feature facets, for example, the rationale or responsibilities. Systematically analyzing information sources and the extent to which they can be used to recover feature facets helps to improve corresponding techniques.

Answering these research questions provides insights into Marlin, Bitcoin-wallet, and similar systems.

3.2. Subject Systems

Marlin. Our first subject system is Marlin, which reflects three common representations of software-product-line variability: First, Marlin relies on the C preprocessor to implement variation points in its platform. Thus, optional features are defined as *preprocessor macros* in the code and can be selected in the two configuration files `Configuration.h` and `Configuration_adv.h`. Marlin's build system is based on plain Makefiles, which contain conditionals (e.g., `ifeq`) that define which files to select and build based on a configuration.

Second, Marlin exists in over 4,600 forks developed by different users that extend and adapt it to their own needs (the *clone-and-own* approach [Dubinsky et al. 2013; Ray and Kim 2012]). An existing analysis by Stănciulescu et al. [2015] and our investigations show that, while such forks often only comprise changed configuration files, they are used to implement new features that are later merged back. Our analysis is based on the mainline of Marlin, specifically *Release Candidate 8*, and ranges from November 2011 until December 2016.

Third, our analysis also shows that not all variable parts of Marlin are annotated in preprocessor annotations. Instead, the system also comprises *runtime parameters* to make dynamic changes, depending on the context. For this reason, we investigated the features we identified in more detail to analyze if they comprise such dynamic variability—arguing that this could hide other features.

Bitcoin-wallet. Our second subject system is Bitcoin-wallet, which is an Android application that relies on runtime parameters to implement variability and has been forked more than 1,200 times. As this system relies on runtime parameters, many of its features can be customized by the users. This comprises the selection of features as well as setting a parameter to change the application's behavior, for instance, to customize the displayed accuracy of Bitcoins (denomination). For this system, we finally annotated version 6.3, which was committed on October 1st 2018 and the history ranges back to March 2011.

Our analysis revealed that not all features rely on the same binding mode, despite using the same variability mechanism. Several features are only active if a constant is set already before compiling the application. Thus, such features actually represent static variability to which the Bitcoin-wallet developers refer to as *compile-time flags*. Again, we investigate the features in more detail to analyze differences between dynamic and static variability.

3.3. Methodology

For both of our subject systems, we applied the same methodology with slight adaptations. These adaptations depend on the kind of system, namely, embedded printer software that is connected to hardware in contrast to an Android application. We display an overview of our method and the analyzed features in Figure 1.

Domain Analysis. First, we performed a domain analysis of our subject systems to identify an initial set of features. To this end, for each system, two of the authors build and used it in different settings.

For Marlin, we constructed two 3D printers: A Delta printer—which moves arms up and down to position the printing-nozzle based on trigonometric functions—and a

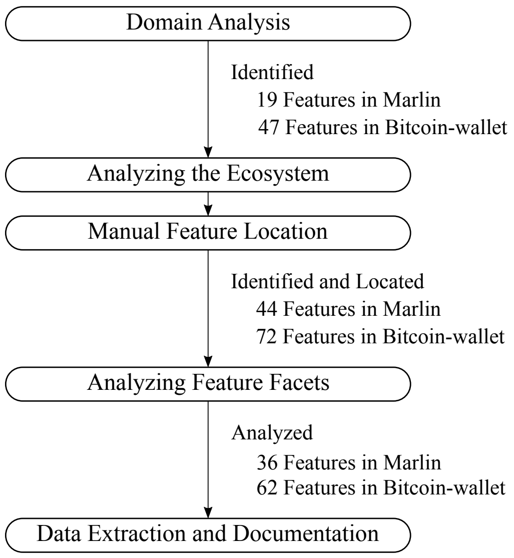

Figure 1: Overview of the applied methodology.

Cartesian printer—which uses a rail on each axis to move the printing-nozzle based on Cartesian coordinates. During this phase, we learned about the hardware components by following the instructions described in the manual. We then installed the Marlin firmware onto the printers' motherboards and tested different configurations. Thus, we got an understanding of the functionality of hardware components and how they are connected to the firmware. In particular, we learned which optional features that are defined as preprocessor macros are represented by which hardware in our two printers. As a result, we also identified hardware commonalities between both printers, which are the once fundamentally necessary for 3D-printing, such as temperature sensors. At the end of this construction phase, we created a first version of a feature model comprising 6 optional and 13 mandatory features that was based on our understanding of the hardware components.

For Bitcoin-wallet, we installed the application on different devices and emulators to test its functionalities. In particular, we explored the options we could set on the user interface and differences between devices. Based on this, we were able to understand the functionalities that are provided by Bitcoin-wallet. We discussed the different features we explored to provide a ground-truth we could agree on. Thus, we identified a total of 47 features, of which we found 17 by customizing the application, indicating optional and alternative features. Again, we derived a feature model, in which we used 9 abstract features to structure the commonalities and variabilities we identified.

Analyzing the Ecosystems. In the second phase, we familiarized with the development processes, community, and evolution of our subject systems. We aimed to find additional information sources that help us to locate and identify further features as well as their facets, for example, pull request reviews and the contributor list. To this end, we performed a pilot study on each system's ecosystem, for instance, identifying the main contributors that implement new features.

For Marlin, we identified 18 developers that are most actively extending and maintaining the firmware. To understand how the community works, communicates, and implements new features, we investigated their development processes. For this purpose, we analyzed the release log that is maintained on the GitHub website and tracked it to pull requests and commits. We investigated the life-cycle of the corresponding features to understand how they are implemented and integrated into the firmware.

For Bitcoin-wallet, we found that the development is heavily driven by a single developer. Other developers support the implementation with small contributions, opening issues, and discussions. However, it did not seem like there was a thorough process for development and communication, as in Marlin. While most issues and pull requests for Marlin are linked and tagged, we found this rarely for Bitcoin-wallet. Thus, the system seems to be less driven by a community and more dependent on a single developer.

Manual Feature Location. The previous steps improved our understanding on both of our subject systems and their ecosystems, allowing us to identify some mandatory and optional features as well as additional information sources. Next, we annotated the identified feature locations—if these were not yet in preprocessor directives (in the case of optional features in Marlin)—by using an embedded feature-annotation approach [Ji et al. 2015] for which we can utilize a tool to visualize these features and their annotations [Andam et al. 2017]. These annotations are lightweight: //&begin[<feature name>] and //&end[<feature name>] associate the lines between these comments to the feature specified with its name. In contrast, //&line[<feature name>] annotates a single line of source code that is separated from the rest of its feature. As these annotations are based on comments, they do not interfere with the code or preprocessor, but are solely for documentation purpose. We refined the feature models to include newly identified features and dependencies.

For Marlin, we especially identified domain knowledge and the release log, with corresponding pull-requests and commits, as our initial information sources. We then completely manually located features by performing a systematic code review, relying on the information sources we discuss in Section 4.3. To this end, we started with Marlin's main file, continued to read comments, G-Code documentation, and aimed to understand the code. Altogether, we identified 44 features. Out of these features, we decided to ignore one in our later analysis: A feature to *cancel the heat-up phase* was not implemented, only some empty methods existed. During our study, discussions between developers on how to implement this feature are ongoing. As we can only guess that this feature may be optional—the printers work without it and the same behavior can be achieved with workarounds, we excluded it from our following analysis.

For Bitcoin-wallet, we identified the change log and wikipages as additional information sources. However, these information are not linked to the source code and we did not find a release log or similar system that links feature to code, as we found for Marlin. Thus, these information sources did not provide entry points for feature location, but only to identify features and their facets. For this reason, we relied on another code review, starting from the configuration file to locate features. Overall, we identified 72 implemented features for Bitcoin-wallet.

Analyzing Feature Facets. After the feature location phase, we analyzed the features' facets. For each feature, we tracked down the artifacts belonging to it in each information source. We did this based on keywords that are consistently used by the community. By manually analyzing the identified artifacts, we consolidated the existing knowledge of feature facets in the version control system.

For Marlin, we investigated 36 of the identified features. We excluded 8 features that a) are repetitions (e.g., different unit transformations), b) are rather feature interactions and glue-code (e.g., movement specifics that require adaptations to the printer hardware), or c) are small parts encapsulated by other features (e.g., changing the units for movement from coordinates to radius). For the remaining 36 features, we used the release log and website as starting points for our analysis of feature facets. We identified further information sources that were not helpful for feature location, but solely to recover their facets, such as contributor lists, the contributors' websites, or pull request reviews and discussions. Overall, we identified 10 sources that are partly more fine-grained than those for feature location.

For Bitcoin-wallet, we applied the same methodology, but deviated from it based on the availability of information sources. In this case, we included 62 features into our facet analysis. The excluded features are interchangeable options, namely for denomination of the displayed amount of Bitcoins and codings that can be used to transfer data. Overall, we relied on the same information sources as for Marlin, but they are less connected to each other.

Data Extraction and Documentation. For each identified feature, we created a *feature fact sheet* to document the following extracted information, depending on the availability in each system:

- Name of the feature
- The feature's name in preprocessor directives and annotations
- Description of the feature's intent
- Used information sources to identify and locate the feature
- Applied search strategies for feature location
- Release version
- Feature characteristics (lines of code, scattering degree, tangling degree)
- Pull request comprising commit links, numbers, names, and code changes
- Identified facets
- Value of each facet
- Used information source for each facet

All feature fact sheets, the corresponding data, the constructed feature models, and the feature facets are publicly available in our repository.[1]

.

Example: Homing. In the following, we describe our analysis process on one concrete example feature. The Marlin feature `Homing` is responsible for positioning the extruder of a printer into a stop position when it is not printing. For feature location, we relied on our domain knowledge from observing this behavior, connecting it to G-Codes, comments in the code, and our systematic code review (i.e., using the keyword *home*). During this phase, we already found that this feature is mandatory. To identify the facets, we relied on different information sources. As rationale, we see `Homing` as a necessary feature derived from the *technical environment*, which we derived from our domain knowledge and Marlin's G-Code documentation. This G-Code documentation helped us further to identify that the architectural responsibility of the feature is in the *application logic*. Considering the definition and approval, we had to dig into the commit messages, in which the developers indicate that this feature is essentially necessary for any 3D-printer to be usable—connecting this facet to a *market analysis*. To identify the binding time and mode of `Homing`, we could use the source code we identified during feature location, but also looked into source code changes in commits. We found that despite being bound at *implementation time*, the feature comprises *dynamic* variability, reacting to the decision why homing is necessary (e.g., for cleaning) and allowing to home a specific axis. For the responsibility, we only identified who committed changes and found that these are *platform developers*. Finally, the release log indicates that evolution-wise, the feature was rolled out with release *1.1.2*.

4. Results

We first report the insights we gained during our pilot studies, to then answer our research questions.

4.1. Pilot Studies

During our pilot studies, we explored both subject systems with their communities and development cultures.

Marlin. We found that the primary means of communication are issue trackers and pull requests. Moreover, pull requests are linked to the release log, in which developers track development, quality improvements, and bug fixes of each release. Interestingly, pull requests are labeled and categorized by Marlin's developers, for example, as `PR:Bugfix`, `PR:Coding Standard`, and `PR:New Feature`. By analyzing the commits that are linked to a pull request, we found that feature names are derived from the preprocessor directives, for example, `PRINTCOUNTER` in Listing 1, and are used consistently through all discussions and documentations. Thus, we identified the release log with the corresponding pull requests and commits as information source to identify and locate features as well as their facets.

Marlin's developers rely on the notion of optional features and structure their communication around them, similar to the software-product-line engineering community. A unified terminology seems to be in use from the source code up to the tracking systems and release log to communicate about these features. In contrast, we found no explicit use of mandatory features in the release log or pull requests. We also learned that Marlin's main file `Marlin_Main.cpp` contains the core logic for 3D-printing, with a code analysis contributing as the most general information source. The file handles input commands and interprets them into electrical functions. As a result, the file is the largest in Marlin with over 10,000 lines of code. During our analysis, we also experienced that Marlin reacts dynamically to its environment (e.g., temperature) within few of its features. Thus, because of the missing notion of mandatory features, there seems to be runtime variability hidden within Marlin that the developers do not consider as separate features. We discuss this issue further in Section 4.5.

We also found an additional, domain-specific information source: G-Code instructions [EIA RS-274-D]. G-Code is a numerical control programming language that is used in computer-aided manufacturing to operate machine tools—specifying the system's behavior to the machine controller, for example, the direction and speed of a movement. These instructions are directed into corresponding implementations that command electrical units of the 3D printers, as we illustrate in Listing 1. Because G-Code instructions and their domain functions are well-documented, we were able to utilize them as information source for locating features and identifying their facets.

Bitcoin-wallet. In contrast to Marlin, the development process of Bitcoin-wallet seems to heavily rely on a single developer. Other developers communicate and make contributions through the issue tracker and with pull requests. However, both are less structured and not tagged, which makes it more challenging to link these information to each other, the code, and features. This also seems not necessary, as the contributions of other developers are sparse and small. Thus, most of the development, integration, and issue solving effort depends on the main developer, who interacts in issue discussions and merges pull requests.

Compared to Marlin, we were also able to utilize the application's description in the Google Play Store to identify features.[2] In contrast, we could not rely on preprocessor directives or G-codes. Moreover, we found that Bitcoin-wallet is implemented with runtime variability, but actually comprises static binding times. Thus, while the source code initially indicates that the system is highly configurable by the user, several of its options are defined by constants and have to be customized before deployment. Consequently, Bitcoin-wallet also comprises a configuration file, but the options are actually defined in the `Constants` file.

4.2. RQ$_1$ - Feature Development Process

To understand how Marlin is developed and maintained, we investigated the interactions of developers with the version control system and each other in detail. The whole development is strongly connected and structured around the issue tracker and forking capabilities of GitHub. Overall, we found that Marlin has 283 contributors, of which 18 are regularly active, a group of five to seven seems to be core developers, and especially one of these is arguably driving the development forwards. Moreover, there are two main branches: First, the release candidate branch (`RC`) in which the core system is stored and driven towards releases. Based on this branch, several pre-releases and the stable release are published. Second, the bug fixing release candidate branch (`RCBugFix`) is used to fix bugs and merge new features.

We display a typical development process for a new feature in Figure 2. At first, anyone can raise an issue within Marlin's issue tracker to propose ideas for new features, bug fixes, or quality improvements. If this issue is unclear, the community will discuss about the technical solutions, coding standards, or relevant pull requests. After the issue is clarified and scoped, developers usually assign an issue to themselves—taking the responsibility for it—and fork Marlin into a private fork (which is a requirement of the community) to implement the issue. Interestingly, as the issue assignment is decentralized, it can happen that multiple developers implement the same issue separately. This is resolved by comparing the final solutions and selecting the best one during the review phase. Before any solution is merged back into the main fork, the developer has to create a pull request. Then, one contributor of the core group reviews the code to point out bugs and assure the quality. Only if a solution is finally accepted, it is pulled and merged into the `RCBugFix` branch. In this branch, the code is again tested, partly automatized, but mainly by developers that review the updated version and install it on their printers. If the `RCBugFix` branch comprises enough new content of a certain quality, it is merged into the main `RC` branch, which is driven towards a release. Thus, new features undergo several quality assurance cycles before they are finally released. Moreover, during all these phases data, discussions, and documentations are created, which present information sources that are usually not considered for locating features and identifying their facets. For example, we already emphasized that developers who assign themselves to a feature take the responsibility for it, clearly indicating the corresponding feature facet.

In contrast to Marlin, we found no community-driven development process in Bitcoin-wallet. This system comprises 26 contributors, but only one of them is regularly working on the code and owns around 99% of the repository's content. As there is also less of a structure around

[2]`https://play.google.com/store/apps/details?id=de.schildbach.wallet`

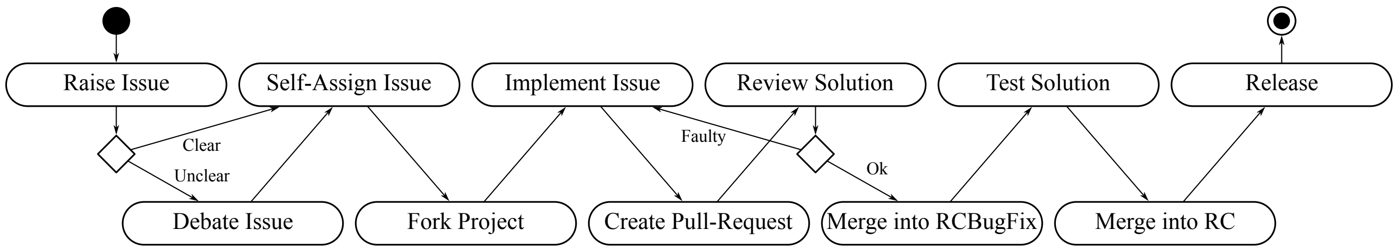

Figure 2: Typical development process of a new feature in Marlin

features or involvement of a release log, the main source of input is the issue tracker. In this tracker, developers and users can raise issues and, identical to Marlin, they can also implement solutions and open pull-requests. While the main developer is heavily involved in discussions and reviews others' solutions, we found no hints of a structured process. Thus, considering the process we depict in Figure 2, it seems to be rarely applied to that extent. Instead, raised issues are often directly addressed on the main branch.

Discussion. Marlin has a well-defined and structured development process for features and bug fixes. Several steps are concerned with quality assurance and, while everyone can contribute an issue or implement it, a subset of contributors is responsible for accepting them. Besides ensuring quality, this process also serves as a detailed documentation and allows tracking changes and decision-making processes. This illustrates the potential for improving automation for feature location and for recovering feature facets based on such modern information sources. Still, while we found a common notion of features for the Marlin community around which the communication is structured, this may not be the case for other systems. It seems interesting to test techniques based on natural language processing to connect artifacts, such as source code, commits, and discussions—aiming to identify and locate features as well as their facets. If a common terminology is established in projects, this may allow to considerably improve automated analyses of legacy systems.

The development process is also interesting, due to the way the variability mechanism of cloning is used: Usually, it is assumed that clones are forked out and then developed completely separated to customize them to customers' needs. However, in Marlin most forks only adapt configuration files, which is hardly a clone in that sense. Instead, the clones serve only as starting point for developing a new feature or for fixing a bug. As soon as possible, the updates are merged back into the base system, meaning that the `RC` branch will comprise all approved features. Still, up to this point, no bugs can be introduced into the stable system, which limits the risks of developing faulty code.

As a result, the question arises if the defined process may be a best practice for developers. Marlin has been developed for more than 7 years (excluding its predecessors) and comprises more than 4,600 forks. Thus, this development process seems to be established and ensures constant, qualitative implementation of new features, while allowing the integration of third-party developers.

Our analysis of Bitcoin-wallet indicates that the same process is not applied on all open-source projects. However, Bitcoin-wallet has far less contributors, forks, and issues, indicating less popularity compared to Marlin. Thus, the differences in the development processes may not be due to a strict hierarchy or a developer keeping all responsibility, but simply due to size issues.

4.3. RQ$_2$ - Entry Points for Feature Location

Overall, we identified and located 43 features in Marlin, of which 31 are optional and 12 are mandatory. We display the information sources we used to find entry points in Figure 3. This figure has a minor correction to our previous work, as we verified the location of a mandatory feature in preprocessor compilation and reconsidered it to be optional. Mainly, we have been able to utilize the release log (with its connected pull requests and commits), #ifdef annotations, G-Code instructions [EIA RS-274-D], domain knowledge, and analysis of all other code parts. Code analysis, #ifdef annotations, and domain knowledge (obtained by building and testing two printers) are well-known entry-points. A more unique entry point are G-Codes, which are a domain-specific information source and operate the hardware, for example, to park a printer's nozzle in Listing 1. Still, most interesting in the context of modern software development are the release logs, pull requests, commits, and information sources that are automatically created and managed in version control systems. Other mechanisms of software-hosting platforms connect communities even further by providing, for example, issue trackers, Wiki pages, or discussion forums that are used for communication and documentation. However, during our analysis of Marlin as well as Bitcoin-wallet, we found these information sources rarely useful, as we usually identified them by following the links from release logs and commits. Thus, they appear rather late and also comprise few information on locations except for such links. In contrast, such sources are helpful to identify feature facets.

Besides domain knowledge and the release log, our main entry points are different source-code elements. In Marlin, we considered #ifdef directives that, unsurprisingly, are present for all optional features we identified. Another helpful means were G-Code commands that are present in four mandatory and one optional feature. Due to the

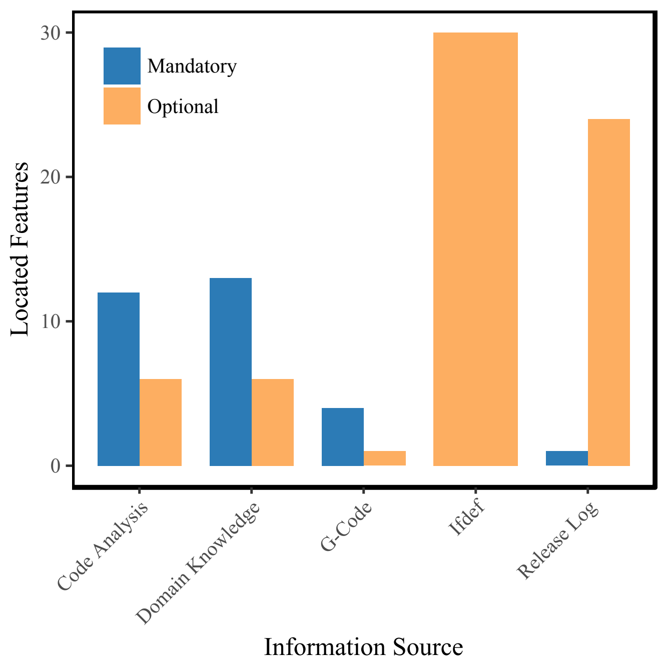

Figure 3: Entry points used to locate features in Marlin.

G-Code documentation and the G-Codes' strong connection to a printer's hardware they control, it is fairly easy to understand the behavior that is implemented in these features. Finally, we investigated the remaining source code based on comments and keywords, which helped us to identify and locate 12 mandatory and 6 optional features.

During our code analysis, we found that some features, for instance `Endstop`, are easy to locate with keyword searches, as all locations contain this term. However, in other cases we need additional domain knowledge to refine the used keywords. For example, we identified the term *feedrate* multiple times in the code and in comments. Only our domain knowledge from building the printers helped us to connect this term to the speed of the motor that feeds material to the extruder. This suggests that syntax-based feature location techniques highly rely on a good understanding of the domain.

Analyzing Bitcoin-wallet was more challenging, as the code provided fewer entry points and helpful code constructs. More precisely, we could only rely on our domain knowledge and a detailed code analysis, as we had neither code constructs to rely on nor a linked release log. Consequently, we started our code review by identifying keywords defined in the `Constants` file that indicate compile-time features. Again, we applied syntax-based feature location to locate features, searching for identified keywords as well as inspecting calls to classes, variables, and methods.

Discussion. Due to the existing notion of features being optional, Marlin's developers do not provide much information about mandatory features in the version control system or the release log. Thus, these information sources are not suitable for locating this type of features. Besides the actual source code and its elements, mainly domain knowledge helps to identify mandatory features of Marlin—in our case heavily based from constructing the actual hardware. As a result, we argue that feature-location techniques can be improved by considering different types of documentation while analyzing the source code. Especially comments seem interesting, as they are directly connected to the corresponding source code in most cases. However, several questions arise, for example, how to ensure that the used documentation is maintained simultaneously to the code [Fluri et al. 2007; Nielebock et al. 2018]. Other domain-specific information sources may be helpful, such as the G-Code commands in our study, but also require domain knowledge to identify them. Ultimately, we found five complementary information sources that were helpful to identify and locate features in software hosted on version control systems, which we show in Figure 3:

- Domain knowledge (e.g., building two printers)
- Release log (i.e., pull requests, commits)
- Code analysis (i.e., comments, dependencies)
- #ifdef annotations
- G-Code commands

Using these information sources and a combination with other artifacts, such as models or requirements, can facilitate identifying and locating both types of features. In particular, we experienced that domain knowledge is necessary to identify features and to find their locations.

Unfortunately, Bitcoin-wallet does not provide such a rich set of entry points for feature location. Especially the missing linkage between the release log and code, the limited variability representation, and missing notion of features made it challenging to analyze the code. Unsurprisingly we found it more challenging to track information for most features in Bitcoin-wallet compared to Marlin. In particular, as we did not implement the application, we were only trying to obtain domain knowledge. Thus, we cannot be sure about the actual intent of the developer, which hampers feature location.

4.4. RQ$_3$ - Search Strategies for Features

Our search strategies for feature locations can be abstracted into two categories: Either analyzing the release log or the source code and its elements. In the following, we will report both strategies in more detail, describing the applied processes and helpful structures in each of them. We remark that domain knowledge is an important information source to support and facilitate these strategies.

Search Through Release Log. Searching through Marlin's release log is considerably different compared to searching in source code. First, it has the advantage that optional features are directly listed and also linked to other artifacts. Thus, the main effort is to browse through all these artifacts and track down feature locations in the code. While, in the end, we also have to read code, the links facilitate the identification of seeds considerably. Still, new problems with this information source arises, for instance:

- Analyzing pull requests and commits involves reading natural language, which may pose problems due to language barriers and ambiguities.
- The release log only contains new features from the latest releases, while older features are not documented. Thus, we have to consider other sources as well to locate such older features.
- In the case of Marlin, mostly (only a single mandatory) optional features are listed in the release log, making this particular information source hardly usable for mandatory ones.

Despite such problems, the release log is a well-documented source for a number of features, which allows us to identify those pull requests and commits in which a new feature is introduced. As these pull requests are linked to commits, we gain excellent entry points for several features. Considering that each feature in the release log is connected to a maximum of six pull requests out of around 4,000, this tremendously reduced the analysis effort. In total, we analyzed 38 pull requests, their 100 connected commits, and the tracked code changes to locate 24 optional and 1 mandatory feature.

Search Through Source Code. To overcome the limitations of the release log in Marlin and its missing links in Bitcoin-wallet, we performed systematic code reviews. For Marlin, this process required around 25 hours in total, starting from the most important file `Marlin_Main.cpp`. It turns out that most mandatory features in Marlin are, partially or completely, located in that file. We systematically studied the whole file to locate features based on our previously obtained domain knowledge. From there, we continued to analyze all other files and located 18 (6 optional, 12 mandatory) additional features. Note that we did not just locate already know features, but also identified some features in this step we were unaware of before.

During our code review, we relied on different search patterns, mainly based on the two code-specific entry points we show in Figure 3:

- If present, #ifdef annotations in the code served as a fast to identify entry point to locate code belonging to optional features.
- G-Codes indicate feature locations that are associated with hardware components and are a fast entry point for such features.

Based on these entry points and when we could not use one of these anymore, we had to systematically go through the remaining source code. To this end, we heavily relied on keywords that we identified in the source code, in comments, or derived from our domain knowledge. In some cases, it was sufficient to use a feature's name to locate all code that belongs to it. For instance, `Endstop` controls the corresponding hardware component, which finds reference points for motor movements and searching for this term already located most of the feature's code. In contrast, for other features we needed to refine keywords, for example, the term *feedrate* appears several times in the code and comments. However, there is no corresponding feature and, initially, we could not connect it to any hardware component. Finally, we identified that it refers to the speed of the motor that feeds printing material to the extruder. Thus, source code that is connected with this term belongs to one of two features that are concerned with extruders. For each identified location, we also investigated method calls and other dependencies to identify further potential feature locations. Overall, our code analysis resembles an extended—due to the additional entry points of #ifdef annotations and G-Codes—combination of the information retrieval and exploration-based search patterns described by Wang et al. [2013]

For Bitcoin-wallet we relied on the same analysis process, but had fewer information sources, namely preprocessor directives and G-Codes were unavailable. Instead, we were able to link variables and keywords in `Constants.java` and `Configuration.java` to features. We extended these feature locations by following the variables and analyzing their surrounding context, which took us approximately 20 hours. During our analysis, we only investigated the Java code of the application and focused on the features we identified before. We did not locate code for 6 features:

- There are 3 options for the `BlockExplorer` feature, namely `blockchain`, `blocktrail`, and `blockcypher`. In the code we identified a single parameter that just takes one of these options as input from the user interface, which is defined in XML.
- Similarly, the `Localization` feature to change the language is defined with a separate configuration file that is automatically processed by Android. Thus, there are no locations in the actual code of Bitcoin-wallet, even though the feature is statically bound and can be dynamically changed.
- We did not find any specific locations for the 3 trading methods `Email`, `Cloud Storage`, and `Webpage file download`. However, these methods represent corner cases and the Bitcoin-wallet Wiki states that they are by-products of other methods. Consequently, there seems to be no implementation that specifically belongs to them.

Overall, we experienced that design decisions, the targeted platform, and the used variability mechanism can facilitate, but also heavily hamper feature location.

Discussion. Our analysis indicates that different information sources require adapted search approaches, but can then facilitate the analysis. Consequently, we also have to adapt automated techniques accordingly. Regarding the artifacts we considered, this is rather unsurprising: Source code is differently structure and provides additional sources compared to the release log and its connected artifacts—except for the code differences stored in each commit. Still, the release log proved to be an effective and cheap way to identify and locate optional features in Marlin.

Table 1: Number of features corresponding to a specific value in a feature facet for the Marlin case.

Feature Facet	Value	#Features
Rationale	Business reasons - Customer requests	18
	Aspects of the technical environment	12
	Social aspects - Usage context	7
Nature	Unit of variability	31
	Unit of functionality	12
Architectural Responsibility	Application logic	28
	User interface	14
Definition and Approval	Customer requests	25
	Market analysis	19
	Competitors	5
Binding Time	Compile time	35
	Implementation	7
	Link time	3
Binding Mode	Static	34
	Dynamic	3
Responsibility	Application developer	7
	Platform developer	30
Evolution	Rolled out	35
Quality and Performance	Code optimization	7
	Reliability	4
	Memory consumption	3
	Safety	3
	Clone avoidance	2
	Response time	2
	Usability	1
	Power consumption	1
	Resource consumption	1
	Recoverability	1

Table 2: Number of features corresponding to a specific value in a feature facet for the Bitcoin-wallet case.

Feature Facet	Value	#Features
Rationale	Business reasons - Customer requests	15
	Business reasons - Market demand	6
	Aspects of the technical environment	3
	Social aspects - Usage context	27
	Social aspects - User needs	9
Nature	Unit of variability	7
	Unit of functionality	34
	Configuration/calibration parameter	20
Architectural Responsibility	Application logic	39
	User interface	19
	Infrastructure level task	3
Definition and Approval	Customer requests	16
	Market analysis	9
Binding Time	Compile time	2
	Configuration time	2
	Design time	39
	Runtime	18
Binding Mode	Static	43
	Dynamic	18
Responsibility	Application developer	0
	Platform developer	60
Evolution	Rolled out	61
Quality and Performance	Accessibility/visibility	1
	Accuracy (or precision)	1
	Availability	1
	Cost	2
	Performance	1
	Precision	1
	Privacy	1
	Reliability	1
	Response time	4
	Security	9
	Size	1

4.5. RQ_4 - Information Sources for Feature Facets

In Table 1 and Table 2, we summarize the identified feature facets for Marlin and Bitcoin-wallet, including the number of features that correspond to these values. We see that in both systems most facets comprise multiple values. For example, in Marlin, the rationale for features originates from customer requirements, the technical environment (hardware) of the printers, as well as the users' context and needs. An exception is the evolution facet, for which we find only one value, rolled out, in both systems. This is reasonable, as we only analyzed features in the main branches of the two systems. In both systems, features that are not rolled out are only present in other forks. Bitcoin wallet includes 18 dynamic features. This number is lower for Marlin, which relies on preprocessor annotations. However, we nonetheless found three dynamic features in Marlin that are bound at link time.

In Figure 4 and Figure 5, we display the information sources that we used to identify the feature facets. Compared to the sources for feature location (cf. Figure 3), we can see that we used far more sources to identify facets. An explanation for this is the diversity of information associated to facets, which often includes information that cannot be found in the code itself. In contrast, feature location can in doubt be done using the code as single information source and additional sources are mostly used to facilitate the task.

Focusing on the used information sources for both systems, we find similarities and differences. In both cases, we used commit messages for multiple facets (i.e., architectural responsibility, definition and approval, rational, quality and performance) and source code changes from commits as main sources to identify binding time and binding mode. Some information sources are rather specifically aligned to one feature facet, for instance, a commit's author and the contributor list only help to identify responsibilities.

Similarly, we found information on the evolution facet only based on pull requests (Marlin) or release logs (Bitcoin-wallet). This depends on how the projects are using these sources: Marlin only provides a list of features, but links them to the code, while Bitcoin-wallet misses the links, but shows version numbers and corresponding features. Thus, in Marlin pull requests are a valuable source to identify feature facets for most features, exceptions being quality and performance or binding time. In contrast, we could barely use them for Bitcoin-wallet. Instead, domain documentation (e.g., Wiki pages, change log, readme files) turned out to be a richer source for facets in the Bitcoin-wallet case compared to Marlin. Consequently, some sources differ strongly in how they can be used for both systems. In the following, we discuss the results for all of our facets.

Rationale. The rationale facet describes why a feature is introduced. Considering the values for this facet, we see that most features in Marlin are driven by customer requests (i.e., users/developers raising an issue or need) and the technical environment (e.g., hardware). Usage contexts, such as being able to react to emergency situa-

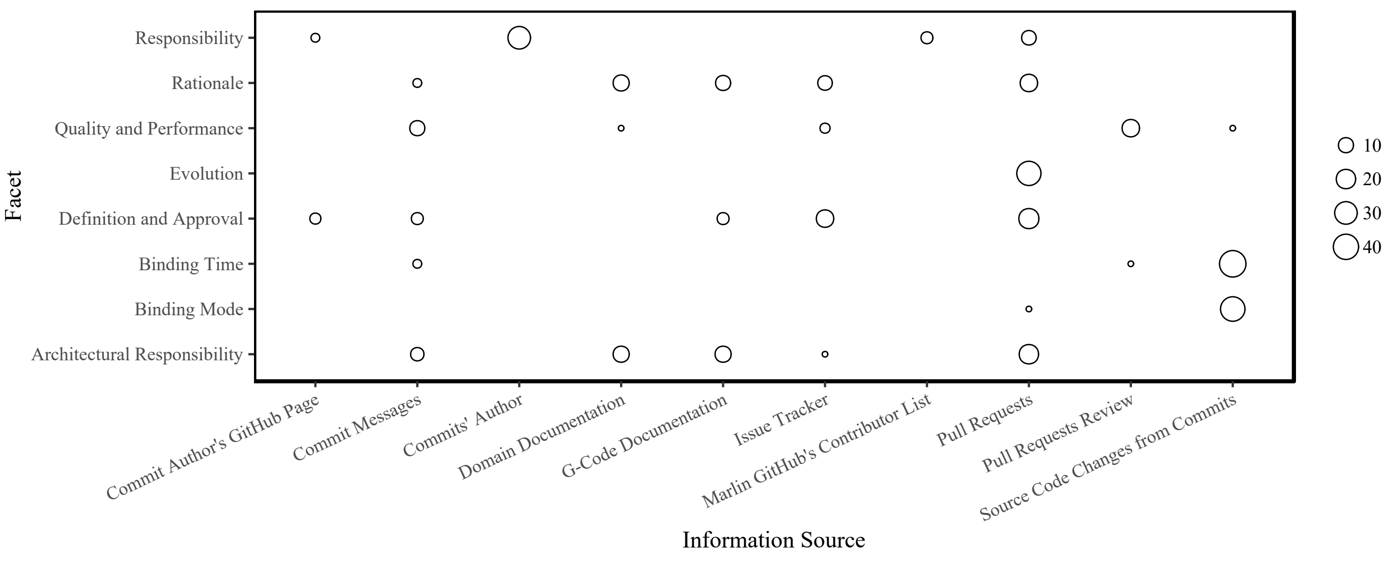

Figure 4: Distribution of identified feature facets for each information source for the Marlin case.

tions, play a minor role for Marlin features. In contrast, many Bitcoin-wallet features are motivated by the usage context. Again, customer requests play an important role. However, the technical environment is not that important. This difference can be explained, as Marlin is much more dependent on the hardware than Bitcoin-wallet.

For both systems, we found no centralized place where we could identify this facet. This may be a characteristic of many open-source systems, as they often do not work with professional requirement management tools, which could centralize such information. Consequently, we relied on commit messages, domain documentation, G-Code documentation, the issue tracker, and pull requests—with none of these sources standing out in particular.

Architectural Responsibility. Architectural responsibility describes to which part of the system a feature belongs. In Marlin, we only identified two values: Features either belong to the application logic or the user interface. An explanation for this is that the 3D-printer firmware does not contain other architectural units, such as a database. Interestingly, we found a similar picture for Bitcoin-wallet. Most features belong to the application logic and some to the user interface. Also, Bitcoin-wallet includes only few other features, which are concerned with the infrastructure of the system. Again, we found no information source specifically useful to identify this facet. Instead, it was necessary to recover the information from commit messages, domain documentation, G-Code documentation, the issue tracker, pull requests, and sometimes even the source code.

Definition and Approval. In both systems, features are usually defined and approved for consideration by customer requests and market analysis (e.g., analyzing the hardware, domain, and use case)—especially based on discussions during Marlin's development process. Competing firmwares or systems play only a minor role. The main information sources in both systems were commit messages and the issue tracker (and the linked pull requests for Marlin). This reflects very well the development process that we found for Marlin and to partly resemble the one for Bitcoin-wallet (cf. Section 4.2). For features not originating from these sources, we had to dig deeper into commit author's GitHub pages, domain documentation, or G-Code documentation.

Binding Time and Mode. As already mentioned, Marlin is mainly implemented with static binding based on the C preprocessor, meaning that the binding time is either at implementation time (mandatory features) or build time (optional features). However, we also found few features that comprise dynamic variability and are changed at link time (e.g., `Homing`). Bitcoin-wallet is very different, as features are either bound at design time or dynamically at runtime (with few exceptions).

To identify the binding time and mode of a feature, we mainly relied on code analysis, directly investigating the implementation of the variability mechanisms. This was particularly necessary, as the variability mechanisms are barely documented and connected to feature or may even be misleading (e.g., Bitcoin-wallet using constants for runtime checks). Consequently, pull requests, domain documentation, domain analysis, and commit messages can be supportive means, but to a rather limited extent.

In the case of Marlin, the identified dynamic variability is interesting, as it is part of mandatory as well as optional features, but is not considered as explicit features by the community. Such variability is used to react to different usage scenarios (cf. `Homing`) or changing inputs of the motherboard's pins. For example, these pins transfer different temperature values that may require a specific reaction or they transfer mechanical controls. To this end, different pins are used, which must be checked individually and at runtime, and represents an essential functionality of Marlin. As a result, we found many interactions with static vari-

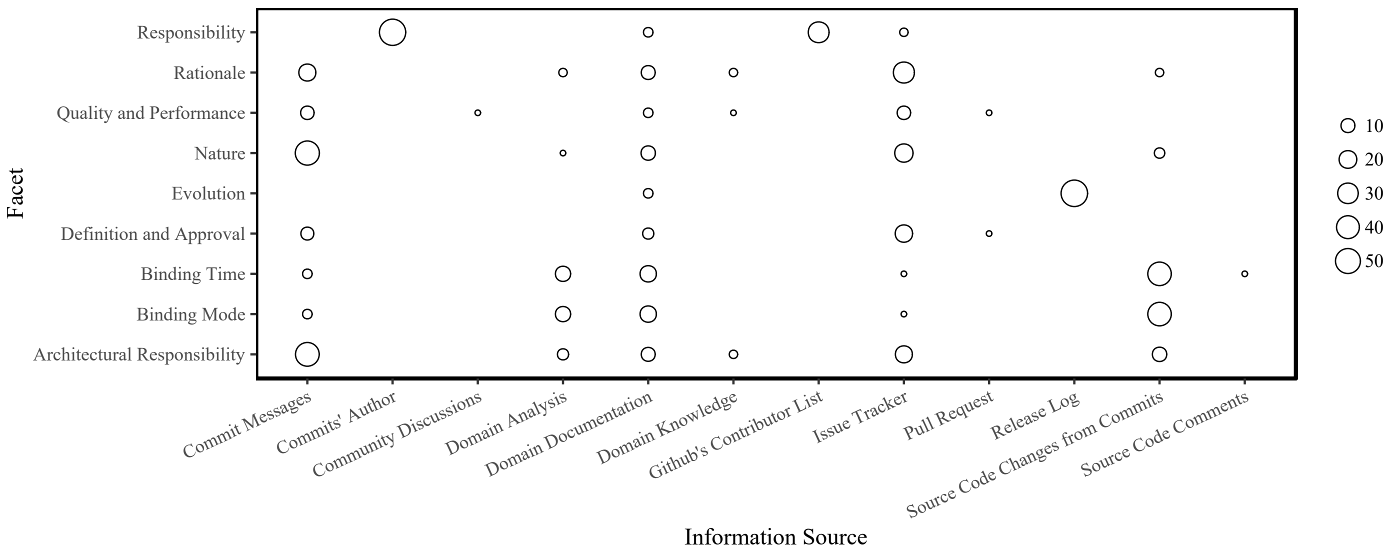

Figure 5: Distribution of identified feature facets for each information source for the Bitcoin wallet case.

ability, for example, for debugging purposes. For us, such interactions of statically and dynamically bound features seem rather interesting. The existing runtime parameters can hardly be replaced with another variability mechanism, as they have to react to changes in a printer's context at runtime. This dependency between hardware and environmental variability in a software may be impossible, or at least impractical, to implement with static variability, or even dangerous if it enables exclusion.

Similarly, while using Bitcoin-wallet, some features appeared to be mandatory, for instance, `Exchange Rates` and its sub-features. Still, unsurprisingly, Bitcoin-wallet comprises more dynamic optionality than Marlin: Android applications are less dependent on the underlying hardware and require dynamic options to allow users to change the system's behavior, as only one version can be provided in the official store.

Arguably, similar situations of mixed and hidden interchanges between static and dynamic binding times occur in many other systems, too, for example, to react to contextual information, changing inputs, or different communication channels. Thus, the question arises if other communities are aware of such issues and communicate them. This awareness seems essential to uncover, understand, and manage feature interactions by the means of dynamic software product line. In Marlin, such issues seem not to be considered or communicated separately, which may result in developers missing them.

Responsibility. Identifying the developer who is responsible for a feature is helpful to ask questions or to assign tasks. As values for this facet, we distinguished between the roles of developers, namely the predominant platform developers and the far fewer application developers—which is especially for Bitcoin-wallet unsurprising, as a single developer is responsible for the whole system and all its features. Unfortunately, the contributor list provided by GitHub is not always suitable to identify this facet. This list does not define roles, but only shows to what extent a developer is involved, which may indicate a certain role. Instead, we relied on the contributors' GitHub pages, commit authors, issue tracker, and pull requests for both systems.

Evolution. While we could utilize different information sources for most facets, evolution is again an exception. Above, we already discussed that evolution in the main branch of Marlin is only characterized by features that are rolled out and the applied development process (cf. Section 4.2). The main information sources for the two systems were the release log and pull requests. This is due to the fact that the release log tracks the integration of new features for Bitcoin-wallet, while it links to the corresponding (required) pull requests for Marlin. Still, it is interesting that these information are not reported in any other source, except for domain documentation resembling the release log in the case of Bitcoin-wallet. However, it is possible that this is due to our focus on the main branch. For features that are in other stages of their development, the evolution facet may be identifiable based on other sources.

Quality and Performance. We described in Section 4.2 that Marlin has a rather strict quality assurance. In Table 1, we see that several non-functional requirements are important for the community. Most commonly seem to be code optimizations, reliability, memory consumption, and safety. It is not surprising to see that Bitcoin wallet has a very different focus when it comes to quality, mainly considering security, regarding that it implements an application to improve the security of money transactions.

As we captured non-functional requirements, we could identify these properties only for a subset of all features. To this end, we utilized commit messages, domain documentation, issue tracker, pull request reviews, and source code changes. Apparently, non-functional requirements are rarely made explicit and mainly discussed in reviews of features that shall be integrated into the system.

Discussion. Overall, we found that different information sources can be helpful for each feature facet. Most of these information sources are only available in modern version control systems, but provide good opportunities to improve automated techniques to recover feature facets. Still, as comparing Marlin and Bitcoin-wallet illustrates, the usability of each information source for a facet depends heavily on its usage, a defined development process, community, and domain of the system.

The question arises to what extent these new sources are reliable. This is similar to using additional information in the code for identifying and locating features. For example, comments and code may not evolve simultaneously [Fluri et al. 2007; Nielebock et al. 2018] and similar situations may occur for some of our sources, such as domain documentations. In contrast, other sources are stored automatically and if they are not manipulated or ambiguous (e.g., deleted, merged, or unclear commit messages), they may be more reliable, as they provide snapshots of the different points in time of the system's development history.

5. Threats to Validity

In the following, we report threats to the internal, external and conclusion validity [Wohlin et al. 2012].

Internal Validity. The main threat to the internal validity is that we, and not the original developers, identified features, their locations, and their facets. Thus, our results may be biased. However, we mitigated this threat with two authors becoming domain experts for each system, for example, by assembling two different kinds of 3D printers (i.e., Delta, Cartesian) for Marlin, which differ in their mechanics and algorithms. We also performed pilot studies and system analyses, during which different authors extensively read documentations (e.g., about G-Code commands) and meta-data (e.g., issue tracker) available in the GitHub repositories. The source code was also analyzed in pairs, which includes cross-checking of the code understanding and of the locations. We plan to validate the data set with the original systems' developers.

External Validity. A threat to the external validity is that we only consider two systems, which may differ from others. Yet, Marlin is a substantial case, and as an embedded system, it shares characteristics with many other embedded systems. In fact, preprocessors are used similarly in almost all open-source and industrial C/C++ systems [Hunsen et al. 2016]. Similarly, Bitcoin-wallet is an Android application that shares common characteristics with others and, thus, should also be representative.

Another factor influencing the external validity is the software-hosting platform used for Marlin and Bitcoin-wallet. Other platforms may utilize other version control systems and additional components that comprise different information sources, store data in another way, or apply other mechanics. However, both systems are hosted on GitHub, which is one of the largest open-source version control systems. Moreover, the available data is usually similar, as such systems are based on the same ideas. Thus, we argue that our results are transferable to other projects.

Conclusion Validity. To enhance the repeatability and reliability of our study, we provide the data set with feature locations and all other data in an online appendix.[1] Consequently, we argue that other researchers can replicate our study, but may derive other results. For example, due to Marlin's evolution they may categorize features differently, or include additional information sources (e.g., developers). Nonetheless, we argue that this is not a threat to the reliability of our study.

6. Related Work

In the following, we discuss other *feature-location datasets*, *experiments on manual feature location*, and related *case studies*. There is also another analysis of the forking techniques of Marlin by Stănciulescu et al. [2015]. The authors aim to understand the pros and cons of cloning based on this analysis and, thus, our goal is different.

Feature-Location Datasets. We are aware of only few data-sets on feature location: Olszak and Jorgensen [2011] developed a tool for feature location, which they apply on multiple systems. The corresponding source code and data is partly available. Ji et al. [2015] annotated feature locations in the source code of the freely available Clafer Web Tools [Antkiewicz et al. 2013]. The authors provide a set of four systems with annotated feature locations in the source code. Martinez et al. [2017] maintain an online catalog of case studies connected to the extractive approach [Krueger 2002] towards software-product-line engineering. The catalog currently includes five academic and open-source systems on which reproducible feature-location studies with available source code have been performed. Such data sets complement ours and we can use them to investigate feature locations and facets in other systems.

Experiments on Manual Feature Location. There have been few experiments and case studies on manual feature location [Krüger et al. 2018a]. Wang et al. [2011, 2013] report three exploratory experiments conducted on four open-source Java systems. Their goal was to understand how developers perform feature location tasks. Overall, they describe distinct phases, patterns, and elementary actions. For evaluating the effectiveness of patterns and actions, the authors rely on junior developers. Similarly, Damevski et al. [2016] performed a field study on developers' behavior when performing feature location tasks. They report the frequency and type of code search tools used, queries, retrieval strategies employed, as well as patterns of developer behavior during feature location.

While these works involve manual feature location, they do not distinct between the nature of a feature and do not aim to recover other facets. Furthermore, they do not include an investigation of information sources for features. Both experiments focus on feature location in GUI-based systems and observe participants' interactions with the GUI, but do not consider preprocessor-based code. Still, as we described, we used similar patterns for our source code analysis as reported by Wang et al. [2013].

Case Studies. Wilde et al. [2003] report experiences of a feature location case study on unstructured FORTRAN code. The authors used two semi-automatic techniques and compare them to manual feature location. Their study reveals that both techniques are effective in locating features, but require considerable adaptation.

Jordan et al. [2015] conducted an industrial in-vivo observation on two experienced software engineers modernizing a COBOL system. They aimed to understand manual searches for feature locations and identify helpful tools. Their results suggest that domain knowledge improves effectiveness and that search tools do not yield relevant results.

Ji et al. [2015] conducted a simulation study using a clone-based software product line on which they applied an embedded feature annotation approach. They located features based on the following sources: Project wikis, commit messages, commit diffs, code, issue trackers, and the original developers. Still, their focus was to show the benefits of embedding feature traces rather than investigating information sources or feature facets.

Krüger et al. [2017] identified and manually mapped features in five cloned Java systems. As information source, they used a code-clone detection tool to identify initial seeds from which they extended their search. The authors' focus was to locate features and compare their results to a fully automated refactoring.

While all these works are related to ours, the goal of our study is complementary. Mainly, we investigated other research questions than most works, or considered other information sources, for example compared to Ji et al. [2015]. Furthermore, our subject systems differ in their development processes and we did not use any feature location technique. While there are some consistent experiences (i.e., domain knowledge improves feature location) some others are quite different among all works (i.e., the usefulness of keyword searches). It seems interesting to investigate those discrepancies.

7. Conclusion

We presented an exploratory study of manually identifying and locating features and their facets in Marlin and Bitcoin-wallet. To this end, we explored and described the development processes of both systems as well as information sources that are useful to locate features, and compared to what extent they can be used to identify their facets. We contribute a data set of feature locations, usable by other researchers to evaluate feature-location techniques or study feature facets, and all our analysis data in an online repository. Among others, locating features in code requires substantial domain knowledge. The same accounts for the corresponding feature facets. We also found substantial differences in the usability of different information sources, considering which source can be used for which facets and to what extent.

As key message, we can summarize our findings as follows: There are more information sources for locating features and identifying their facets in modern systems than are exploited for most corresponding techniques. While a technique based on such different sources seems promising, we are still lacking knowledge to make such a technique usable. Foremost, we need to better understand the relationships between information sources, their integration into development processes, and derive appropriate search strategies that connect the available information.

In future work, we aim to complement our results by further case studies on other systems (and forks of our subject systems) to consolidate our insights into different information sources and development cultures. We plan to do the same for feature facets and the corresponding information sources, also supporting developers to potentially improve their information management. Particularly interesting are comparisons of development processes among various projects to unveil why they are applied and what characteristics may lead to changes. An interview study with the Marlin community and industrial developers may provide further insights into such issues. In the same way, we argue that analyzing how and why such communities combine static and dynamic variability is interesting. Furthermore, we aim to evaluate the features, facets, and locations, for instance, by performing feature-based maintenance and evolution tasks—measuring the benefit of the obtained information and the accuracy of our feature locations. This way, we aim to identify best practices to guide developers in their projects and researchers to scope their work. Finally, our results provide a starting point not only to develop new techniques to recover information about features, but may also extent the scope of existing techniques that we want to match against our results. Improving the support for such artifacts in feature location and information recovery techniques seems essential to facilitate analysis and re-engineering (cf. Section 2) activities.

Acknowledgments

Supported by the ITEA project REVaMP^2 funded by Vinnova Sweden (2016-02804), by the Swedish Research Council Vetenskapsrådet (257822902), and the German Research Council DFG (LE 3382/2-1).

References

Hadil Abukwaik, Andreas Burger, Berima Andam, and Thorsten Berger. Semi-Automated Feature Traceability with Embedded Annotations. In *International Conference on Software Maintenance and Evolution*, ICSME, pages 529–533. IEEE, 2018. doi: 10.1109/ICSME.2018.00049.

Berima Andam, Andreas Burger, Thorsten Berger, and Michel Chaudron. FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces. In *International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, pages 100–107. ACM, 2017. doi: 10.1145/3023956.3023967.

Michał Antkiewicz, Kacper Bąk, Alexandr Murashkin, Rafael Olaechea, Jia H. J. Liang, and Krzysztof Czarnecki. Clafer Tools for Product Line Engineering. In *International Software Product Line Conference*, SPLC, pages 130–135. ACM, 2013. doi: 10.1145/2499777.2499779.

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013. doi: 10.1007/978-3-642-37521-7.

Wesley K. G. Assunção and Silvia Regina Vergilio. Feature Location for Software Product Line Migration: A Mapping Study. In *International Software Product Line Conference*, SPLC, pages 52–59. ACM, 2014. doi: 10.1145/2647908.2655967.

Wesley K. G. Assunção, Roberto E Lopez-Herrejon, Lukas Linsbauer, Silvia R Vergilio, and Alexander Egyed. Reengineering Legacy Applications Into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering*, 22(6):2972–3016, 2017. doi: 10.1007/s10664-017-9499-z.

Thorsten Berger and Jianmei Guo. Towards System Analysis with Variability Model Metrics. In *International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, pages 23:1–23:8. ACM, 2014. doi: 10.1145/2556624.2556641.

Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering*, 39(12):1611–1640, 2013. doi: 10.1109/TSE.2013.34.

Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *International Conference on Software Product Line*, SPLC, pages 16–25. ACM, 2015. doi: 10.1145/2791060.2791108.

Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The Concept Assignment Problem in Program Understanding. In *International Conference on Software Engineering*, ICSE, pages 482–498. IEEE, 1993. doi: 10.1109/ICSE.1993.346017.

Rafael Capilla, Jan Bosch, Pablo Trinidad, Antonio Ruiz-Cortés, and Mike Hinchey. An Overview of Dynamic Software Product Line Architectures and Techniques: Observations from Research and Industry. *Journal of Systems and Software*, 91:3–23, 2014. doi: 10.1016/j.jss.2013.12.038.

Andreas Classen, Patrick Heymans, and Pierre-yves Schobbens. What's in a Feature: A Requirements Engineering Perspective. In *Fundamental Approaches to Software Engineering*, FASE, pages 16–30. Springer, 2008. doi: 10.1007/978-3-540-78743-3_2.

Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, pages 173–182. ACM, 2012. doi: 10.1145/2110147.2110167.

Kostadin Damevski, David Shepherd, and Lori Pollock. A Field Study of How Developers Locate Features in Source Code. *Empirical Software Engineering*, 21(2):724–747, 2016. doi: 10.1007/s10664-015-9373-9.

Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013. doi: 10.1002/smr.567.

Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering*, CSMR, pages 25–34. IEEE, 2013. doi: 10.1109/CSMR.2013.13.

Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do Crosscutting Concerns Cause Defects? *IEEE Transactions on Software Engineering*, 34(4):497–515, 2008. doi: 10.1109/TSE.2008.36.

EIA RS-274-D. Interchangeable Variable Block Data Format for Positioning, Contouring, and Contouring/Positioning Numerically Controlled Machines. Standard, Electronic Industries Association, 1979.

Eduardo Figueiredo, Bruno Carreiro da Silva, Cláudio Sant'Anna, Alessandro F. Garcia, Jon Whittle, and Daltro José Nunes. Crosscutting Patterns and Design Stability: An Exploratory Analysis. In *International Conference on Program Comprehension*, ICPC, pages 138–147. IEEE, 2009. doi: 10.1109/ICPC.2009.5090037.

Beat Fluri, Michael Wursch, and Harald C. Gall. Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes. In *Working Conference on Reverse Engineering*, WCRE, pages 70–79. IEEE, 2007. doi: 10.1109/WCRE.2007.21.

Critina Gacek and Michalis Anastasopoules. Implementing Product Line Variabilities. *SIGSOFT Software Engineering Notes*, 26(3): 109–117, 2001. doi: 10.1145/379377.375269.

Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering*, 21(2):449–482, 2016. doi: 10.1007/s10664-015-9360-1.

Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. Maintaining Feature Traceability with Embedded Annotations. In *International Systems and Software Product Line Conference*, SPLC, pages 61–70. ACM, 2015. doi: 10.1145/2791060.2791107.

Howell Jordan, Jacek Rosik, Sebastian Herold, Goetz Botterweck, and Jim Buckley. Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads. In *International Conference on Program Comprehension*, ICPC, pages 174–177. IEEE, 2015. doi: 10.1109/ICPC.2015.26.

Kyo Chul Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute Carnegie-Mellon University, 1990.

Charles W. Krueger. Easing the Transition to Software Mass Customization. In *International Workshop on Software Product-Family Engineering*, PFE, pages 282–293. Springer, 2002. doi: 10.1007/3-540-47833-7_25.

Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. Finding Lost Features in Cloned Systems. In *International Systems and Software Product Line Conference*, SPLC, pages 65–72. ACM, 2017. doi: 10.1145/3109729.3109736.

Jacob Krüger, Thorsten Berger, and Thomas Leich. Features and How to Find Them: A Survey of Manual Feature Location. In *Software Engineering for Variability Intensive Systems: Foundations and Applications*. LLC/CRC Press, 2018a.

Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, pages 105–112. ACM, 2018b. doi: 10.1145/3168365.3168371.

Jacob Krüger, Jens Wiemann, Wolfram Fenske, Gunter Saake, and Thomas Leich. Do You Remember This Source Code? In *International Conference on Software Engineering*, ICSE, pages 764–775. ACM, 2018c. doi: 10.1145/3180155.3180215.

Jaejoon Lee and Dirk Muthig. Feature-Oriented Variability Management in Product Line Engineering. *Communications of the ACM*, 49(12):55–59, 2006. doi: 10.1145/1183236.1183266.

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in 40 Preprocessor-Based Software Product Lines. In *International Conference on Software Engineering*, ICSE, pages 105–114, 2010. doi: 10.1145/1806799.1806819.

Max Lillack, Stefan Stanciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wasowski. Intention-Based Integration of Software Variants. In *International Conference on Software Engineering*, ICSE, 2019.

Angela Lozano. An Overview of Techniques for Detecting Software Variability Concepts in Source Code. In *Advances in Concep-*

tual Modeling. Recent Developments and New Directions, pages 141–150. Springer, 2011. doi: 10.1007/978-3-642-24574-9_19.

Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *International Systems and Software Product Line Conference*, SPLC, pages 38–41. ACM, 2017. doi: 10.1145/3109729.3109748.

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *European Conference on Object-Oriented Programming*, ECOOP, pages 495–518. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2015. doi: 10.4230/LIPIcs.ECOOP.2015.495.

Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering*, 41(8):820–841, 2015. doi: 10.1109/TSE.2015.2415793.

Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. Commenting Source Code: Is it Worth it for Small Programming Tasks? *Empirical Software Engineering*, pages 1–40, 2018. doi: 10.1007/s10664-018-9664-z.

Andrzej Olszak and Bo Norregaard Jorgensen. Understanding Legacy Features with Featureous. In *Working Conference on Reverse Engineering*, WCRE, pages 435–436. IEEE, 2011. doi: 10.1109/WCRE.2011.64.

Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. Feature-Oriented Software Evolution. In *International Workshop on Variability Modelling of Software-Intensive Systems*, VaMoS, 2013. doi: 10.1145/2430502.2430526.

Leonardo Passos, Jesús Padilla, Thorsten Berger, Sven Apel, Krzysztof Czarnecki, and Marco Tulio Valente. Feature Scattering in the Large: A Longitudinal Study of Linux Kernel Device Drivers. In *International Conference on Modularity*, MODULARITY, pages 81–92. ACM, 2015. doi: 10.1145/2724525.2724575.

Denys Poshyvanyk, Yann-Gael Gueheneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature Location Using Probabilistic Ranking of Methods Based on Execution Scenarios and Information Retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007. doi: 10.1109/TSE.2007.1016.

Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *European Conference on Object-Oriented Programming*, ECOOP, pages 419–443. Springer, 1997. doi: 10.1007/BFb0053389.

Baishakhi Ray and Miryung Kim. A Case Study of Cross-System Porting in Forked Projects. In *International Symposium on the Foundations of Software Engineering*, FSE, pages 53:1–53:11. ACM, 2012. doi: 10.1145/2393596.2393659.

Abdul Razzaq, Asanka Wasala, Chris Exton, and Jim Buckley. The State of Empirical Evaluation in Static Feature Location. *ACM Transactions on Software Engineering and Methodology*, 28(1):2:1–2:58, 2018. doi: 10.1145/3280988.

Martin P. Robillard and Gail C. Murphy. FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code. In *International Conference on Software Engineering*, ICSE, pages 822–823. IEEE, 2003. doi: 10.1109/ICSE.2003.1201304.

Martin P. Robillard and Gail C. Murphy. Representing Concerns in Source Code. *ACM Transactions on Software Engineering and Methodology*, 16(1), 2007. doi: 10.1145/1189748.1189751.

Marko Rosenmüller. *Towards Flexible Feature Composition: Static and Dynamic Binding in Software Product Lines.* PhD thesis, University of Magdeburg, 2011.

Julia Rubin and Marsha Chechik. A Survey of Feature Location Techniques. In *Domain Engineering*, pages 29–58. Springer, 2013. doi: 10.1007/978-3-642-36654-3_2.

Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. Forked and Integrated Variants in an Open-Source Firmware Project. In *International Conference on Software Maintenance and Evolution*, ICSME, pages 151–160. IEEE, 2015. doi: 10.1109/ICSM.2015.7332461.

Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions. In *International Conference on Software Maintenance*, ICSM, pages 213–222. IEEE, 2011. doi: 10.1109/ICSM.2011.6080788.

Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *Journal of Software: Evolution and Process*, 25(11):1193–1224, 2013. doi: 10.1002/smr.1593.

Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajilich, and La Treva Pounds. A Comparison of Methods for Locating Features in Legacy Software. *Journal of Systems and Software*, 65(2):105–114, 2003. doi: 10.1016/S0164-1212(02)00052-3.

Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering.* Springer, 2012.