Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin

Jacob Krüger
Harz University & University of Magdeburg, Germany

Wanzi Gu
Chalmers University of Technology
Sweden

Hui Shen
Chalmers University of Technology
Sweden

Mukelabai Mukelabai
Chalmers | University of Gothenburg
Sweden

Regina Hebig
Chalmers | University of Gothenburg
Sweden

Thorsten Berger
Chalmers | University of Gothenburg
Sweden

ABSTRACT

The notion of features is commonly used to describe, structure, and communicate the functionalities of a system. Unfortunately, features and their locations in software artifacts are rarely made explicit and often need to be recovered by developers. To this end, researchers have conceived automated feature-location techniques. However, their accuracy is generally low, and they mostly rely on few information sources, disregarding the richness of modern projects. To improve such techniques, we need to improve the empirical understanding of features and their characteristics, including the information sources that support feature location. Even though, the product-line community has extensively studied features, the focus was primarily on variable features in preprocessor-based systems, largely side-stepping mandatory features, which are hard to identify. We present an exploratory case study on identifying and locating features. We study what information sources reveal features and to what extent, compare the characteristics of mandatory and optional features, and formulate hypotheses about our observations. Among others, we find that locating features in code requires substantial domain knowledge for half of the mandatory features (e.g., to connect keywords) and that mandatory and optional features in fact differ. For instance, mandatory features are less scattered. Other researchers can use our manually created data set of features locations for future research, guided by our formulated hypotheses.

CCS CONCEPTS

• **Software and its engineering → Software product lines**; **Software reverse engineering**;

KEYWORDS

Feature location, preprocessor, Marlin, case study

ACM Reference Format:
Jacob Krüger, Wanzi Gu, Hui Shen, Mukelabai Mukelabai, Regina Hebig, and Thorsten Berger. 2018. Towards a Better Understanding of Software Features and Their Characteristics: A Case Study of Marlin. In *VAMOS 2018: 12th International Workshop on Variability Modelling of Software-Intensive Systems, February 7–9, 2018, Madrid, Spain.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3168365.3168371

1 INTRODUCTION

The notion of *features* is commonly used to specify, manage, and communicate the behavior of software systems and to support developers in comprehending, reusing, or changing these systems [2]. As such, features are helpful entitites during software development, maintenance, and evolution [4, 27]. However, despite developers being aware of and using features to describe their systems, features and their locations are rarely made explicit in source code. When a system evolves over time, the knowledge of features and their locations quickly fades, which then needs to be recovered. In fact, *feature location* [3, 10, 23, 32] is one of the most common and expensive tasks in software engineering [5, 18, 28, 37].

To recover features and their locations, different (semi-)automated techniques have been proposed [26, 32]. Unfortunately, their accuracy is generally low, they require substantial effort (e.g., calibration towards the project), and often focus on one information source only, such as keywords in source code. To improve manual or automated feature-location techniques, we need to improve the empirical understanding of features. This includes information sources that can be used to recover features and their locations, strategies to exploit these information sources, and characteristics of features. Furthermore, realistic datasets of feature locations are necessary to evaluate and compare corresponding techniques.

Different notions of features exist [4, 6]. The software-product-line engineering (SPLE) community has extensively investigated features as units of variability [2, 7]—so-called optional features. These features are used in variation points, such as preprocessor directives (e.g., # ifdef) or if statements and can therefore be easily located. Their characteristics are also well-understood, due to extensive studies [22]. Yet, this only represents variable (optional) features, while little is known about non-variable (mandatory) features—units of functionality, used for system evolution and maintenance. This notion of features is more common in industrial software engineering [4] and in research on concern location [12, 14, 31].

We strive to improve the empirical understanding of software features and their characteristics. Our goal is two-fold: First, we explore information sources existing in modern open-source projects, aiming to understand to what extent they can be used to identify mandatory features and their locations. Second, we analyze how the characteristics of mandatory features differ from optional ones, aiming to understand whether and how insights gained from studies of optional features apply to mandatory features.

In this paper, we present an initial exploratory study on identifying and locating features in Marlin, a variant-rich open-source embedded firmware for 3D printers. Hosted on Github, it boasts a richness of different information sources that can contain traces or indicators for features. After studying the development culture and processes, we manually explore these sources as entry points for identifying and locating features. We document the features in a feature model and their locations in the source code. We analyze core characteristics, compare mandatory to optional features, and formulate hypotheses that try to explain the differences. Finally, we discuss our next steps to continue this work and to address its current limitations. In summary, we contribute:

- a set of *feature fact sheets* for 43 features that contain information sources and search strategies for each feature;
- a data set of the Marlin source code with annotated features and a corresponding feature model;
- empirical data on characteristics of the features; and
- an online appendix containing all these contributions.[1]

On a final note, we hope that we can raise discussions about the different notions of features. Our findings are an initial assessment of Marlin, to be extended in future work. Yet, our preliminary results already provide insights into the different notions of features and indicate that researchers should carefully distinguish them. For instance, we find different characteristics among mandatory and optional features, which implies that it can be problematic to derive conclusions about feature-based maintenance and evolution, when studying preprocessor directives (optional features), only. Furthermore, we describe different information sources that support locating features. Together with the provided data set, these results can help to improve feature location techniques and to steer developers who need to recover features and their locations.

2 NOTIONS OF FEATURES

Several notions of features exist [2, 4, 6]. We distinguish between features as units of *variability* and as units of *functionality*.

Features as Units of Variability. In SPLE, features are primarily seen as units of variability, given the widespread use of annotation-based variability mechanisms, typically using preprocessors and conditional compilation (e.g., # if or # ifdef) [2, 25]. In Listing 1, we show a code excerpt from Marlin, where the preprocessor macro NOOZLE_PARK_FEATURE represents an optional feature. During the build process, the annotated code is only included if this macro (feature) is enabled. Through code-level dependencies and to foster automated configuration, these optional features are often declared in a variability model [8, 33], which is typically the input for a configuration tool (however, Marlin uses simple configuration files).

In this notion, the locations of features only represent their variable parts; any mandatory code part that also belongs to a feature is not annotated. As such, locating the variable code is easy, but recovering the locations of mandatory parts is difficult and costly.

[1]https://github.com/hui8958/Marlin/tree/MarlinFeatureAnnotations

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
```

Likewise, completely mandatory features may not be represented in the feature model. So, in summary, this notion is useful if features are only used to configure a product line, not if features shall be used, for instance, to plan development, to communicate, to maintain a system, to fix bugs, or to re-engineer a system.

Take the example of re-engineering cloned products into a product line [11, 21, 34] and specifically consider a single feature that is cloned among two variants, and slightly modified in one variant. If the feature is integrated into one platform, only the differences will be annotated (likely, a new feature representing the differences is introduced). The actual location of the whole feature is not annotated and likely needs to be recovered for maintenance and evolution.

Given the availability of many open-source systems with optional features, studies on their code-level characteristics, measuring tangling or scattering degrees, have been conducted [2, 22]. It is often argued that these metrics influence maintenance and evolution effort. However, the exact relationship is still largely unclear, due to the limitation that only optional features are studied.

Features as Units of Functionality. A broader notion of features is to see them as units of functionality, which we adopt in this paper. Here, a feature represents a functionality (or concern) in a system, regardless of whether it is an optional or mandatory functionality in the case of a product line. Such features and their locations are rarely documented, for example with feature-traceability databases [30] or embedded feature annotations [18], wherefore recovering their locations is costly and error-prone [37]. Even automated or semi-automated feature-location techniques require substantial manual effort (e.g., for calibration or for finding so-called seeds from which they can start exploring code) and fall short in accuracy [32]. They also often only take one information source into account (e.g., code comments), while it is not clear which other information sources can be used for systems rich in meta-data, such as open-source projects hosted on GitHub, with potentially relevant information in issue trackers, pull requests, or Wiki pages.

3 STUDY DESIGN

We conduct a case study on Marlin, a configurable 3D-printer firmware with a rich set of information sources.

3.1 Research Questions

We formulate two research questions:

RQ_1 *What information sources help to locate features to what extent?* Automated feature-location techniques usually exploit information sources, such as code or requirements documents. For Marlin, several different information sources, for instance, code, release logs, and pull requests, exist. We systematically analyze these sources to recover both mandatory and optional features and to record information about each feature.

RQ$_2$ *What are differences between mandatory and optional features?* We compare core code-level characteristics of mandatory and optional features. Specifically, we use the most commonly used metrics to characterize features: Lines of feature code, tangling degree, and scattering degree. We formulate hypotheses, aiming to explain our findings.

Answering these research questions provides insights into the Marlin and similar systems. Still, the results need to be complemented by case studies of other systems (or other forks of Marlin) to obtain more detailed insights into information sources, feature-location strategies, and to create larger datasets. Such extensions are also necessary to confirm or refute our formulated hypotheses. Furthermore, our identified features and their locations need to be evaluated, for instance, by performing feature-based maintenance and evolution tasks and measuring the benefit of features and the accuracy of their locations.

3.2 Subject System

Our subject system is Marlin, which reflects two common representations of variants: First, Marlin relies on the C preprocessor [20] to implement variation points in its platform. Thus, optional features are defined as preprocessor macros in the code and can be configured in the files `Configuration.h` and `Configuration_adv.h`. Marlin's build system is based on plain Makefiles, which contain conditionals (e.g., `ifeq`) to select the respective files to be built based on a configuration. Second, Marlin exists in over 4.600 forks by third-party developers that extend and adapt it to their own needs [34] (a.k.a. clone-and-own [11, 29]), thus potentially introducing additional variability compared to the original fork. Our analysis is based on the mainline of Marlin.

3.3 Methodology

Domain Analysis. To identify and locate features, it is crucial to understand and become experts in Marlin's domain. To this end, we construct two 3D printers: A Delta printer—which uses trigonometric functions to move arms up and down to position the printing-nozzle—and a Cartesian printer—which uses Cartesian coordinates and a rail for each axis to move the printing-nozzle. Here, we learn about the hardware components and follow instructions described in the manual. We then install the Marlin firmware onto the printers' motherboards and test some configurations. During this phase, we get an understanding on the functionality of hardware components and how they are connected to the firmware. In particular, we learn which optional features that are defined as preprocessor macros are represented by which hardware. As a result, we also identify hardware commonalities between both printers, which are essentials for 3D-printing. After construction, we create a first version of a feature model with 6 optional and 13 mandatory features based on our understanding of the hardware components.

Pilot Study on Marlin's Ecosystem. After constructing the printers and gaining first insights into the 3D-printing domain, we conduct a pilot study on the Marlin ecosystem. Here, we identify 18 developers that are actively extending Marlin. We study their development behavior for new features based on pull requests and commits. In this phase, we do not identify new features, but aim to determine additional information sources for this purpose.

Manual Feature Location. The previous steps improve our understanding on 3D printing, Marlin, and some optional features. Thus, domain knowledge and the release log, with corresponding pull-requests and commits, are our initial information sources. We then completely manually locate features by performing a systematic code review, relying on information sources we identify in the pilot study. Starting with Marlin's main file, we continue to read comments, g-code documentation, and aim to understand the code.

We annotate the location of each feature—if it is not yet in preprocessor directives (in the case of optional features)—by relying on an embedded feature-annotation approach [18] and a tool to visualize these features and their annotations [1]. The annotations are lightweight: // &begin[<feature name>] and // &end[<feature name>] associate the lines between these comments to a feature, and // line &[<feature name>] annotates single lined source code that is separated from the rest of its feature. We refine the feature model to represent newly identified features.

Documentation. For each identified feature, we create a *feature fact sheet* to document the following information:

- Name of the feature
- Feature's name in preprocessor directives and annotations
- Description of the feature's behavior
- Used information sources
- Applied search strategy
- Release version
- Feature characteristics (LOC, scattering, tangling)
- Pull request with commit links, numbers, names, code change

All feature fact sheets, the corresponding data, and the constructed feature model are publicly available.[1]

3.4 Metrics

To describe each feature and to answer RQ$_2$, we calculate the following metrics per feature, based on its annotations:

(1) *Lines of Code (LOC)* represents the size of a feature based on the number of implemented code lines.
(2) *Scattering Degree (SD)* describes at how many different locations a feature is implemented. Here, we consider non-consecutive lines of code belonging to the same feature as different locations.
(3) *Tangling Degree (TD)* measures the number of other features that are (partly) included within a considered feature. Thus, we count the number of other features surrounded by our annotations or preprocessor directives for the considered feature.

4 RESULTS

We report our results by first describing insights from the pilot study, and then answering and discussing the research questions.

4.1 Pilot Study

In the pilot study, we explore the Marlin Github project, together with its community and development culture. We find that the primary means of communication are issue trackers and pull requests.

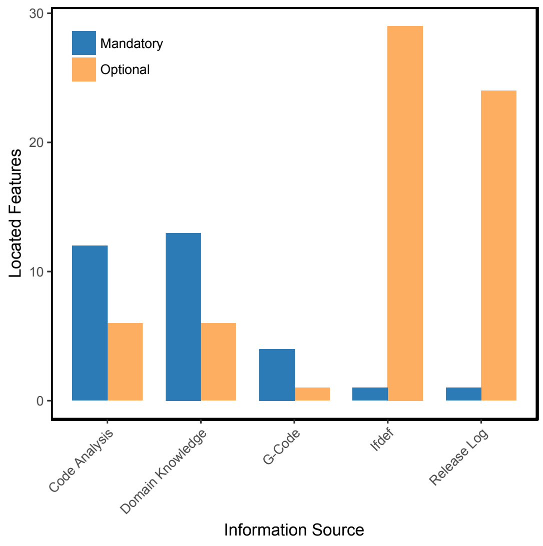


Figure 1: Entry points used to locate features

In the issue tracker, new features, coding standards, potential solutions, and improvements are discussed. Based on the raised issues, tasks are normally self-assigned by developers. To contribute, developers are asked to fork Marlin into an own repository in which they develop their code. When an issue is implemented, the corresponding developer creates a pull request for the changes. Key developers of Marlin then review the source code to avoid bugs or bad formats and, after acceptance, a branch in the main repository is created to merge the changes.

We find that pull requests for new features are linked to the release log, in which developers track development, quality improvements, and bug fixes for each release. Interestingly, the pull requests are labeled and categorized by the developers, for example, as `PR:Bugfix`, `PR:Coding Standard`, and `PR:New Feature`. After analyzing the commits that belong to a pull request, we find that names for new features are derived from the preprocessor directives, for example `PRINTCOUNTER` (cf. Listing 1). Thus, we identify the release log and the corresponding pull requests and commits as information source to identify and locate features.

We also find an additional, domain-specific information source: G-code instructions [13]. G-code is a numerical control programming language that is used for computer-aided manufacturing to operate machine tools. The so-called g-code instructions specify the system's behavior to the machine controller, for example, the direction and speed of movement. Such instructions are directed into corresponding implementations to command the electrical units of 3D printers, as we illustrate in Listing 1. Because g-code instructions and their domain functions are documented, we use them as one of the information sources during manual feature identification.

Interestingly, the Marlin community uses the notion of optional features and structures its communication around them. A unified terminology seems to be in use from the source code up to the tracking systems and release log. In contrast, we find no explicit use of mandatory features in the release log or pull requests. We also learn that Marlin's main file `Marlin_Main.cpp` contains the main logic of 3D-printing. It 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.

4.2 RQ$_1$: Information Sources for Entry Points

Based on the findings in our pilot study, we apply different strategies involving a set of information sources to identify and locate features. The release log documents a number of features, which allows us to identify those pull requests and commits in which a new feature is introduced. Each feature in the release log is linked to a maximum of six pull requests. Given that there are around 4.000 pull requests in Marlin, this documentation turns out to be a tremendous help. Due to the linked commits, we gain an excellent entry point to locate features in code, using commit messages and documented code changes. In total, we analyze 38 pull request and 100 related commits, allowing us to locate 24 optional and 1 mandatory feature. Yet, the release log only contains new features from the latest releases, while older features are not documented, so we consider other sources as well.

For this purpose, we perform a systematic code review, starting from the most important file `Marlin_Main.cpp`. It turns out that most mandatory features in the system are, partially or completely, located in that file. We systematically study the whole file to locate features using our previously collected domain knowledge. From there, we continue to analyze all other files. Note that we also identify features in this step we were unaware of before. Altogether, the systematic code review took 25 hours.

Besides domain knowledge and the release log, we focus on different source-code elements. Here, we consider # ifdef directives that, unsurprisingly, are present for all optional features we identify, and even in one mandatory feature. Another helpful means are g-code commands that are present in four mandatory and one optional feature. Due to the g-code documentation and g-codes' strong connection to hardware, it is fairly easy to understand the behavior that is implemented in these features. Finally, we investigate the remaining source code based on comments and keywords, which helps us to identify and locate twelve mandatory and six optional features. In total, we locate 30 optional and 13 mandatory features in Marlin, as we show in Figure 1.

During code analysis, we find 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 more domain knowledge to scope the keywords. For example, we find the term `feedrate` multiple times in the code and comments. Only our domain knowledge from building the printers helps 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.

Discussion. Due to the existing notion of features being optional, Marlin 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 mandatory features. Besides the actual source code and its elements, mainly domain knowledge helps to identify mandatory features of Marlin. 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,

Table 1: Means and medians of the considered metrics

Feature Type	Mean			Median		
	LOC	SD	TD	LOC	SD	TD
Mandatory	305.46	15.69	11.69	262	12	5
Optional	155.37	7.97	5.1	72.5	5	1

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 [15]. Other domain-specific information sources may be helpful, such as the g-code commands in our study, but require domain knowledge to identify them. Ultimately, we find five complementary information sources helpful, as shown in Figure 1:

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

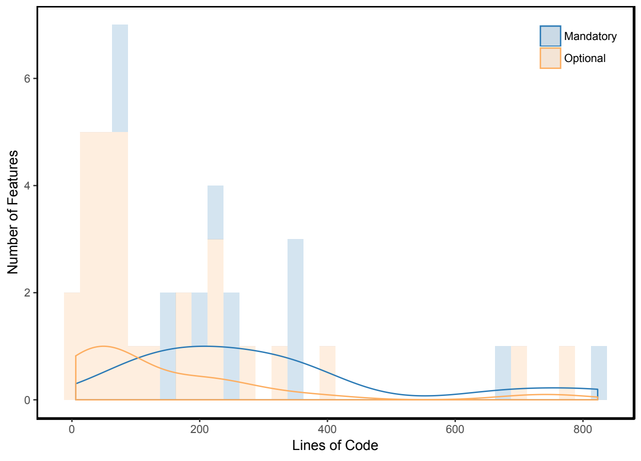
Using these and a combination with other information sources can facilitate identifying and locating both types of features. In particular, we experience that domain knowledge is necessary to identify features and find their locations.

4.3 RQ$_2$: Mandatory vs. Optional Features

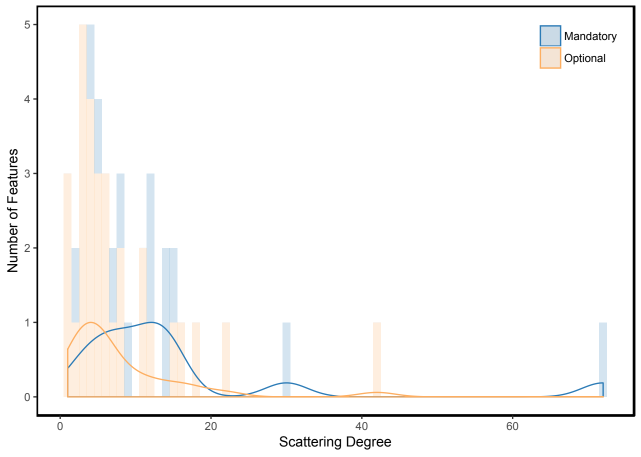
Figure 2 shows the distribution of lines of code (Figure 2a), scattering degree (Figure 2b), and tangling degree (Figure 2c) among optional and mandatory features. The curves display a smoothed relative distribution for which the highest number of features in a bin represents 100%. Additionally, Table 1 shows the mean and median values for these metrics. We see that the mean of each metric is approximately twice as high for mandatory features compared to optional ones. Considering the median, this difference increases further, for instance, the median for mandatory features is approximately 3.6 times the size of the optional ones. This reflects that for all three metrics, most features are represented on the left sides of the distributions in Figure 2. For example, most features have less than 400 lines of code, have a scattering degree below 23, and tangling degree below 10. In each case, some outliers exist for mandatory as well as optional features.

In Figure 3, we compare the lines of code of each feature to its scattering (Figure 3a) and tangling (Figure 3b). Unsurprisingly, we see that smaller code sizes are often connected to less scattering and tangling. However, there are exceptions, for instance a mandatory feature of 214 lines at 75 different locations: `Board` controls pins on the motherboard of a printer and connects it to other hardware, therefore the feature is distributed heavily in the code. For optional features, it seems that a larger size results in more scattering and tangling. In contrast, scattering of mandatory features seems independent to their size but connected with tangling increases.

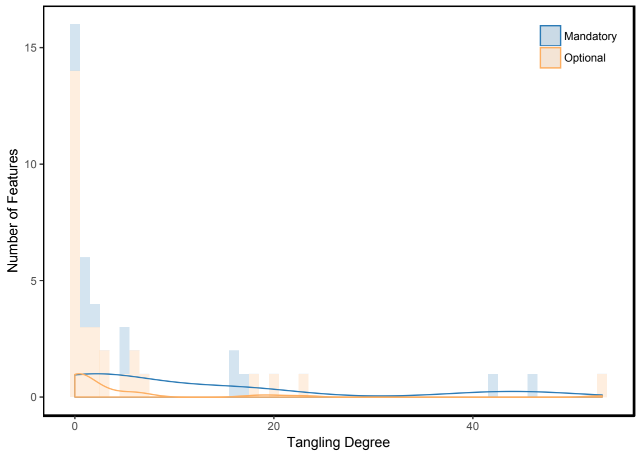
To test if the displayed relations are significant, we use Spearman's Rho and Kendall's Tau significance tests [16]. Both tests are rank correlations and are used to test for monotonic dependencies between two parameters. Neither of both tests requires a normal distribution and the results range from -1 (negative correlation) to 1 (positive correlation). For each test, our *null hypothesis* is that there is no correlation between the parameters. So, we try to discard



(a) Distribution of feature sizes in lines of code.

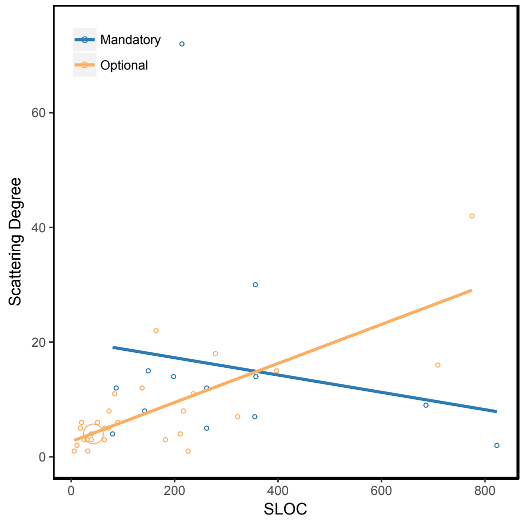


(b) Distribution of scattering degree among features.

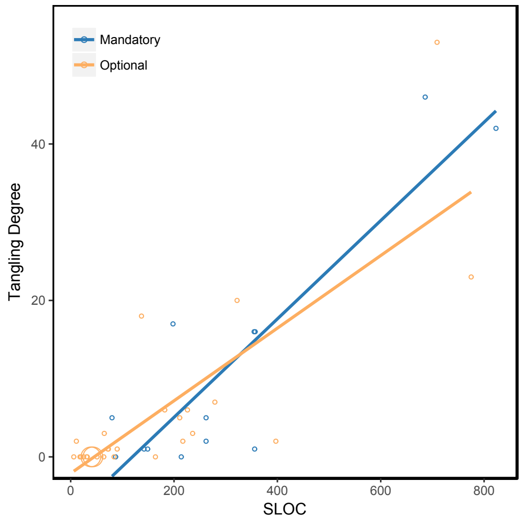


(c) Distribution of tangling degree among features.

Figure 2: Comparing mandatory and optional features. The histograms show the stacked number of features in each bin. The curves approximate the relative distribution for which the bin with the highest number of features represents 100%.



(a) Scattering degree related to lines of code.



(b) Tangling degree related to lines of code.

Figure 3: Relations between scattering, tangling, and lines of feature code for the located features

this hypothesis in favor of the *alternative hypothesis* that there is a correlation. In Table 2, we show the results of each test.

Discussion. Our results show that optional and mandatory features do vary in their characteristics. In addition, we find various significant correlations among optional and mandatory features.

Considering the scattering degree, we find that:

- *Size and scattering of mandatory features are not correlated.*
- *Size and scattering of optional features have a moderate to strong positive correlation.*

Considering these findings, we can reason about the way Marlin evolved over time. Conceptually, we consider two possibilities: On the one hand, mandatory features could be the dominant dimension of decomposition [35] and define the architecture of Marlin. Thus, optional features are later on weaved into the code wherever they are needed. On the other hand, the changeability of the system could be the main interest of developers. Then, we would assume that optional features are implemented cohesive to facilitate replacing and exchanging them (e.g., similar to components).

As the scattering of mandatory features is not correlated to their size, we argue that they are implemented as required. For example, `Board` is scattered in the code because it needs to control hardware at different points. However, even large features with over 800 lines are rather cohesive, indicating that they are implemented as a single unit. In contrast, optional features are more scattered the larger they are. This indicates that they may be integrated into the code afterwards to implement their functionality where it is necessary. Thus, we argue that the first approach is more reasonable and that mandatory features align to the system's architecture.

Considering the tangling degree, we find that:

- *Size and tangling of mandatory features have a strong positive correlation.*
- *Size and tangling of optional features have a moderate positive correlation.*

These correlations are not surprising: A higher number of tangled features of one type should also increase the tangling degree of the other type. However, there are several optional features that are not tangled at all, meaning that they do not affect any behavior of other features. In contrast, almost all mandatory features are tangled, which is reasonable as they provide base functionality of Marlin that is extended by and, thus, interacts with optional features.

Table 2: Results of the significance tests

Correlation	Feature Type	Test	Level	P-value
Scattering	Mandatory	Spearman	-0.062	0.84
		Kendall	-0.039	0.854
	Optional	Spearman	0.663	6.507×10^{-5}
		Kendall	0.54	4.58×10^{-5}
Tangling	Mandatory	Spearman	0.604	0.029
		Kendall	0.47	0.03
	Optional	Spearman	0.78	3.648×10^{-7}
		Kendall	0.624	6.792×10^{-6}

4.4 Additional Remarks

We remark that for two features it is challenging to identify whether they are mandatory or optional, mainly because we do not solely rely on the existence of preprocessor directives, but the actual behavior of features:

- A feature allowing printers to move in arcs is heavily encapsulated in preprocessor directives. But as this is a necessary movement for all 3D printers, we classify it as mandatory.
- A feature to cancel the heat-up phase is not yet implemented, only some empty methods exist. During our study, discussions between developers on how to implement this feature are ongoing. As we cannot extract any data for this feature, we ignore it. Since the same funcationality can be achieved by stopping the printing, this feature is likely optional.

In addition, we observe an interesting situation: Some optional features are not completely wrapped in conditional-compilation directives. In these cases, we find that the non-wrapped code does not affect the system's behavior if the feature is deselected (e.g., dead code). Still, this can potentially be a source for errors, since such false-optional code may change the system's behavior in certain configurations. Further investigations of such false-optional code and its error-proneness, for example in different variants and during system evolution, is necessary.

5 HYPOTHESES AND PROSPECTS

Based on our results, we formulate three hypotheses that we aim to refute or confirm in our future work.

H_1 Mandatory and optional features have different code-level characteristics. In Marlin, mandatory features have twice the size, scattering degrees, and tangling degrees compared to optional ones. Also, their scattering degrees do not correlate to their size, which is the case for optional ones. Considering different implementations, developers, and domains, we may be able to identify correlations that help to classify features. This could improve automatic analysis, for example for feature location and reverse engineering. We will analyze additional systems and also include contributions of other researchers to validate if such correlations always exist.

H_2 Effort and time to locate each type of feature depends on the used information sources. While we did not (yet) precisely measure the necessary efforts and time for each information source, we experienced that they are relatively different for features. Assessing the effort needed to locate and understand features with each information source can help to reduce costs of feature location, scope corresponding techniques, and identify suitable sources. We will conduct controlled experiments to measure the location effort of mandatory features based on different information sources and also apply this methodology on systems without variability.

H_3 Mandatory feature are aligned with the architecture of the system. For Marlin, we find that the scattering degree of mandatory features does not correlate to their size. However this correlation exists for optional features. Also, while mandatory features are twice as scattered as optional ones, this seems to be connected to their actual purpose, for example controlling different hardware adapters. We also look into the development history and optional features are implemented later on and integrated at different points. Considering reuse and evolution, this leads to the question how systems are decomposed in practice: Are mandatory features the core of the system and optional ones are weaved into them? Or are optional features made as reusable as possible and mandatory ones adapted to this structure? In future work we aim to investigate the structure of additional systems, analyze their evolution, and perform empirical studies to verify our hypothesis. Most likely, we will find systems for both decompositions and also their combinations. Here, it is interesting to identify the reasons why systems evolve in a specific way.

6 THREATS TO VALIDITY

The main threat to the *internal validity* is that we, and not the original developers, identify the features and their locations. As a result, feature locations and metrics may be biased. However, we mitigate this threat with two authors becoming domain experts by assembling two different kinds of 3D printers (i.e., Delta, Cartesian), which differ in their mechanics and algorithms. We also perform a pilot study in which the authors extensively read documentation (e.g., about g-code commands), and meta-data (e.g., issue tracker) available in the Marlin Github repository. The source code is also analyzed in pairs, which includes cross-checking of the code understanding and of the locations. We plan to validate the data set with original Marlin developers. Furthermore, for our statistical tests, large sample sizes for mandatory feature are necessary to strengthen the significance tests.

A threat to the *external validity* is that we only consider one system, 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 [17].

7 RELATED WORK

Feature-Location Datasets. We are aware of few data-sets on feature location: Olszak and Jorgensen [26] develop a tool for feature location, which they apply on multiple systems for which the source code and data is partly available. Ji et al. [18] annotate feature locations in the source code of the freely available Clafer Web Tools. The authors provide a set of four system with annotated feature locations in the source code. Martinez et al. [24] maintain an online catalog of case studies connected to extractive SPLE. This includes five academic and open-source systems on which reproducible feature-location studies with available source code are performed. Such data sets complement ours and we can use them to extend our work and test our hypotheses with independently derived data.
Experiments on Manual Feature Location. Wang et al. [36, 37] report three exploratory experiments conducted on four Java open-source systems. Their goal is to understand how developers perform feature location tasks to identify distinct phases, patterns, and elementary actions. For evaluating the effectiveness of patterns and actions, the authors rely on junior developers. Similarly, Damevski et al. [9] perform a field study on developer 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 explicitly distinguish between optional and mandatory features. Furthermore, they do not include an investigation of information sources for mandatory features. Both studies focus on feature location in GUI-based systems and observe participants' interactions with the GUI, but do not consider preprocessor-based code.
Case Studies. Wilde et al. [38] report experiences of a feature location case study on unstructured FORTRAN code. The authors use 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. [19] conduct an industrial in-vivo observation on two experienced software engineers modernizing a COBOL system. They aim to understand manual location searches and identify helpful tools. Their results suggest that domain knowledge improves effectiveness and that search tools do not yield relevant results.

Ji et al. [18] conduct a simulation study using a clone-based product line on which they apply an embedded feature annotation approach. They locate features based on the following sources:

Project wikis, commit messages, commit diffs, code, issue trackers, and the original developers. Still, their focus is to show the benefits of embedding feature traceability rather than investigating information sources or feature characteristics.

Krüger et al. [21] identify and manually map features in five cloned systems. As information source, they use code-clone detection to identify initial seeds from which they extend their search. The authors' focus is 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 investigate other research questions, comparing optional and mandatory features, than most works, or consider other information sources, for example compared to Ji et al. [18]. Furthermore, our subject system differs in its development approach and we do not use any feature location technique.

8 CONCLUSION

We presented an initial exploratory study of manual feature location in Marlin. We explored and described information sources that were useful to locate Marlin's features, and compared the characteristics of optional and mandatory features. We contribute a data set of feature locations, usable by other researchers to evaluate feature-location techniques or study feature characteristics.

Among others, locating features in code required substantial domain knowledge for half of the mandatory features (e.g., to connect keywords). We also found substantial differences in the code-level characteristics of mandatory and optional features, with regard to size, scattering degree, and tangling degree. For instance, mandatory features are less scattered, which we attribute to a better alignment of mandatory features to the system's architecture. We formulated these findings as hypotheses, which need to be confirmed or refuted by studying other Marlin forks or systems.

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 (3382/2-1).

REFERENCES

[1] Berima Andam, Andreas Burger, Thorsten Berger, and Michel Chaudron. 2017. FLOrIDA: Feature LOcatIon DAshboard for Extracting and Visualizing Feature Traces. In *VaMoS*.

[2] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*. Springer.

[3] Wesley Klewerton Guez Assunção and Silvia Regina Vergilio. 2014. Feature Location for Software Product Line Migration: A Mapping Study. In *SPLC*.

[4] Thorsten Berger, Daniela Lettner, Julia Rubin, Paul Grünbacher, Adeline Silva, Martin Becker, Marsha Chechik, and Krzysztof Czarnecki. 2015. What is a Feature? A Qualitative Study of Features in Industrial Software Product Lines. In *SPLC*.

[5] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. 1993. The Concept Assignment Problem in Program Understanding. In *ICSE*.

[6] Andreas Classen, Patrick Heymans, and Pierre-yves Schobbens. 2008. What's in a Feature: A Requirements Engineering Perspective. In *FASE*.

[7] Paul C. Clements and Linda M. Northrop. 2002. *Software Product Lines*. Addison-Wesley.

[8] Krzysztof Czarnecki, Paul Grünbacher, Rick Rabiser, Klaus Schmid, and Andrzej Wąsowski. 2012. Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches. In *VaMoS*.

[9] Kostadin Damevski, David Shepherd, and Lori Pollock. 2016. A Field Study of How Developers Locate Features in Source Code. *Empirical Software Engineering* 21, 2 (2016), 724–747.

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

[11] Yael Dubinsky, Julia Rubin, Thorsten Berger, Slawomir Duszynski, Martin Becker, and Krzysztof Czarnecki. 2013. An Exploratory Study of Cloning in Industrial Software Product Lines. In *CSMR*.

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

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

[14] Eduardo Figueiredo, Bruno Carreiro da Silva, Cláudio Sant'Anna, Alessandro F. Garcia, Jon Whittle, and Daltro José Nunes. 2009. Crosscutting Patterns and Design Stability: An Exploratory Analysis. In *ICPC*.

[15] Beat Fluri, Michael Wursch, and Harald C. Gall. 2007. Do Code and Comments Co-Evolve? On the Relation Between Source Code and Comment Changes. In *WCRE*.

[16] Gregory A. Fredricks and Roger B. Nelsen. 2007. On the Relationship Between Spearman's Rho and Kendall's Tau for Pairs of Continuous Random Variables. *Journal of Statistical Planning and Inference* 137, 7 (2007), 2143–2150.

[17] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2015. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (2015), 449–482.

[18] Wenbin Ji, Thorsten Berger, Michal Antkiewicz, and Krzysztof Czarnecki. 2015. Maintaining Feature Traceability with Embedded Annotations. In *SPLC*.

[19] Howell Jordan, Jacek Rosik, Sebastian Herold, Goetz Botterweck, and Jim Buckley. 2015. Manually Locating Features in Industrial Source Code: The Search Actions of Software Nomads. In *ICPC*. IEEE, 174–177.

[20] Brian W. Kernighan and Dennis M. Ritchie. 1988. *The C Programming Language*. Prentice Hall.

[21] Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *SPLC*.

[22] Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. 2010. An Analysis of the Variability in 40 Preprocessor-Based Software Product Lines. In *ICSE*.

[23] Angela Lozano. 2011. An Overview of Techniques for Detecting Software Variability Concepts in Source Code. In *Advances in Conceptual Modeling. Recent Developments and New Directions*. Springer, 141–150.

[24] Jabier Martinez, Wesley K. G. Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *SPLC*.

[25] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Sarah Nadi, and Rohit Gheyi. 2015. The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *ECOOP*.

[26] Andrzej Olszak and Bo Norregaard Jorgensen. 2011. Understanding Legacy Features with Featureous. In *WCRE*.

[27] Leonardo Passos, Krzysztof Czarnecki, Sven Apel, Andrzej Wąsowski, Christian Kästner, and Jianmei Guo. 2013. Feature-Oriented Software Evolution. In *VaMoS*.

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

[29] Baishakhi Ray and Miryung Kim. 2012. A Case Study of Cross-System Porting in Forked Projects. In *FSE*.

[30] Martin P. Robillard and Gail C. Murphy. 2003. FEAT: A Tool for Locating, Describing, and Analyzing Concerns in Source Code. In *ICSE*.

[31] Martin P. Robillard and Gail C. Murphy. 2007. Representing Concerns in Source Code. *Transactions on Software Engineering Methodologies* 16, 1 (2007).

[32] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*. Springer, 29–58.

[33] Ina Schaefer, Rick Rabiser, Dave Clarke, Lorenzo Bettini, David Benavides, Goetz Botterweck, Animesh Pathak, Salvador Trujillo, and Karina Villela. 2012. Software Diversity: State of the Art and Perspectives. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 477–495.

[34] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *ICSME*.

[35] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. 1999. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *ICSE*.

[36] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2011. An Exploratory Study of Feature Location Process: Distinct Phases, Recurring Patterns, and Elementary Actions. In *ICSM*.

[37] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *Journal of Software: Evolution and Process* 25, 11 (2013).

[38] Norman Wilde, Michelle Buckellew, Henry Page, Vaclav Rajilich, and La Treva Pounds. 2003. A Comparison of Methods for Locating Features in Legacy Software. *Journal of Systems and Software* 65, 2 (2003), 105–114.