Finding Lost Features in Cloned Systems

Jacob Krüger
Harz University of Applied Sciences
Wernigerode, Germany
Otto-von-Guericke-University
Magdeburg, Germany
jkrueger@hs-harz.de

Louis Nell
Harz University of Applied Sciences
Wernigerode, Germany

Wolfram Fenske
Otto-von-Guericke-University
Magdeburg, Germany
wfenske@ovgu.de

Gunter Saake
Otto-von-Guericke-University
Magdeburg, Germany
saake@ovgu.de

Thomas Leich
Harz University of Applied Sciences
Wernigerode, Germany
tleich@hs-harz.de

ABSTRACT

Copying and adapting a system, also known as *clone-and-own*, is a common reuse approach that requires little initial effort. However, the drawbacks of clones are increasing maintenance costs as bug fixes and updates must be propagated. To reduce these costs, migrating cloned legacy systems towards a software product line promises to enable systematic reuse and customization. For both, managing and migrating cloned systems, it remains a challenge to identify and map features in the systems. In this paper, we *i)* propose a semi-automatic process to identify and map features between legacy systems, *ii)* suggest a corresponding visualization approach, and *iii)* assess our process on a case study. The results indicate that our process is suitable to identify features and present commonalities and variability in cloned systems. Our process can be used to enable traceability, prepare refactorings, and extract software product lines.

CCS CONCEPTS

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

KEYWORDS

Software product line, code clone detection, feature location, legacy system, extractive approach, reverse engineering

ACM Reference format:
Jacob Krüger, Louis Nell, Wolfram Fenske, Gunter Saake, and Thomas Leich. 2017. Finding Lost Features in Cloned Systems. In *Proceedings of SPLC '17, Sevilla, Spain, September 25-29, 2017,* 8 pages.
https://doi.org/10.1145/3109729.3109736

1 INTRODUCTION

With cloning (a.k.a. *clone-and-own*), developers copy an existing system and adapt it to new requirements [14, 32, 34]. This approach is often used due to its low initial costs and simplicity [18, 35]. However, each new system results in additional *code clones* [31], requiring careful propagation of changes (e.g., bug fixes) [30]. Especially when knowledge about feature locations deteriorates, mapping features in different systems is a challenging but necessary task to maintain cloned systems and trace requirements [19]. While version control systems can support tracing to some extent [17, 30, 40], they often fall short in this regard. Hence, even with tool support, maintaining cloned systems remains an error-prone and costly task.

As the maintenance costs for such cloned systems increase, companies may consider to migrate these towards a *software product line* [8, 26]. A software product line is described by a set of *features* that implement commonalities and differences of variants [1, 9, 10]. Each feature represents a user-visible functionality and can be reused to customize a system [2]. Thus, changes in a feature must not be propagated into other systems, which can significantly reduce maintenance costs.

Several migration approaches were proposed [17, 27, 45], but the initial step of identifying and tracing features in multiple legacy systems remains challenging [19, 22, 25, 42]. For this purpose, *feature location* [3, 13, 33] techniques were proposed. However, these are mostly designed for a single system [16]. In contrast to other techniques that focus on multiple systems, for example by Xue [45], we focus on a step-wise process and leave the specific design, such as the used analysis techniques, to the developer [17]. This seems necessary when due to vanishing domain knowledge [19] extensive analyses are required.

In this paper, we present an approach to identify and map features in cloned systems. To this end, we introduce a semi-automatic process that is independent of used programming languages or concepts. We focus on small and partly automated steps to reduce the necessary time and effort of our approach. While we cannot fully automate all steps, we can utilize *code-clone detection* [31] and feature location [3, 13, 33] techniques. Another challenge is to visualize the results in a suitable way [16]. To address this issue, we describe a representation we designed to support our proposed process. Finally, we show the feasibility of our approach based on a case study and compare it to the results of automatic refactorings [17].

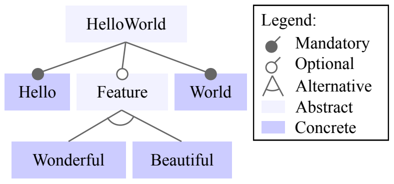

Figure 1: Feature diagram for a HelloWorld product line.

Overall, we aim to support developers to identify common features in cloned systems. Our process can help to improve the management of cloned systems, recover variability information, enable traceability, or migrate them towards a product line.

In detail, our contributions are:

- We propose a semi-automatic, step-wise process to identify and map features in cloned systems. This process helps to analyze, manage, and migrate cloned legacy systems.
- We describe an approach to map and present variability according to our process. This approach is based on previous work for product line potential analysis [16], which we refine to suit our needs.
- We describe an exploratory case study to assess our process and compare our findings to a reference study that is based on the same systems [17]. Overall, we show that our process is feasible and helps to analyze and prepare migrations of cloned legacy systems.

This paper is structured as follows: In Section 2, we introduce the foundations for the understanding of our process. We define the concrete research problem and our process in Section 3. Then, we describe our approach to map features during the process and to present it to the user within Section 4. In Section 5, we present our evaluation. We discuss related work in Section 6 and conclude in Section 7.

2 BACKGROUND

To model the features we identify with our process, we rely on *variability modeling*. We utilize *feature location* and *code-clone detection* concepts to identify and map common code in legacy systems. In this section, we provide an overview on these topics.

2.1 Variability Modeling

Software product lines comprise a set of features that can be combined to derive a customized variant [1, 9]. To manage and represent the resulting variability and dependencies, several variability models have been proposed, for instance, feature models, delta models, or decision models [7, 11, 36, 38]. In this work, we rely on *feature diagrams* [1, 21], which are tree representations of the established feature models [5, 7, 36], to express variability.

In Figure 1, we illustrate a feature diagram for a *HelloWorld* product line. The root (i.e., `HelloWold`) is an abstract feature, which means that it does not implement any functionality but helps to structure the diagram [41]. In contrast, `Hello` is a concrete feature that contains source code. A child feature can either be optional (i.e., `Feature`), which means it need not be selected when its parent is selected, or mandatory (i.e., `Hello`, `World`), which means it must

```
int log(String s) {
 if (s != null) {
  System.out.print(s);
  counter ++;
  return counter;
 }

 return -1;
}
```

(a) Original source code.

```
Type-2   int log(String m) {
Type-2    if (m != null) {
Type-2     System.out.print(m);
Type-1     counter++;
Type-1     return counter;
Type-1    } // comment
Type-3    System.err.print("null arg");
Type-1    return -1;
Type-1   }
```

(b) Cloned source code.

Figure 2: Example for code clones Type-1 to Type-3.

be selected when its parent is selected. Child features can have additional dependencies. For instance, `Wonderful` and `Beautiful` are alternatives, wherefore only one of both can be selected at the same time. Further dependencies between sub-trees can be covered with cross-tree constraints, such as, requires or excludes. Based on the feature model, a configuration of the product line can be derived. A configuration is *valid* if it conforms to all dependencies and only then a variant can be instantiated.

2.2 Feature Location

Feature location aims at identifying code that implements a feature. First, features must be specified as requirements and afterwards mapped to the source code. Several approaches have been proposed, for example, *concern graphs* and *formal concept analysis* [3, 13, 33]. Still, there are shortcomings that hamper their application in practice. Firstly, because feature location is already challenging for one system [22], such approaches are rarely adopted for several systems [16]. This limits their usability for migrating legacy systems towards a product line. Secondly, only for few approaches tools are implemented and those are limited in several ways, for example, they require adaptations to programming languages or do not analyze the source code but documentation of systems [3, 13]. Finally, feature location can only propose feature candidates, but these must be manually evaluated [6, 22, 48]. For these reasons, we consider feature location to support the automation of our process but focus on code-clone detection to map features in multiple legacy systems.

2.3 Code-Clone Detection

Code clones are code artifacts that occur in identical or similar form at several occasions [31]. Clones between two systems can indicate potential features that are implemented in both. Hence, code-clone detection can enhance feature location in this context [16, 45].

As code might be changed after cloning, clones differ in their degree of similarity. We illustrate this in Figure 2, in which Figure 2a represents the original source code and Figure 2b its clone. Roy et al. [31] define four types of code clones:

Type-1 clones are identical and may only differ in whitespaces or comments, which we represent with *Type-1* markers in Figure 2b.

Type-2 clones additionally contain renamed elements, for instance, function or variable names. In Figure 2b, all lines marked with *Type-1* and *Type-2* form such a clone.

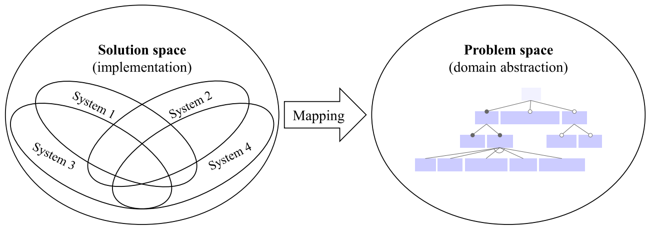

Figure 3: Problem and solution space for cloned systems.

Type-3 clones also include modified, removed, or added statements. The whole clone we show in Figure 2b represents such a clone, as the line marked with *Type-3* is added.

Type-4 clones implement the same functionality while sharing no or only few textual similarity.

Several approaches exist to identify especially Type-1 and Type-2 clones [31]. However, it is problematic to automatically identify the other two types.

Following Bauer and Hauptmann [4], we further separate between to other kinds of code clones. Firstly, clones that appear within the same system and, secondly, clones that appear between systems. We refer to the second kind as *cross-system code clones*, which are essential for our approach as they represent functionality that is reused in different systems, potentially indicating features.

3 MAPPING FEATURES IN LEGACY SYSTEMS

In this section, we define the starting point for our approach. We state the problem based on *problem space* and *solution space* to derive requirements for our process, which we introduce afterwards.

3.1 Problem Statement and Requirements

The *proactive approach* [24] to software-product-line adoption is assumed to require the highest upfront investment but to also promise the most benefits [1, 5, 37]. Proactive development means to design and implement a product line from scratch. Hence, the problem space that defines a system's requirements and behavior is defined first, for example, with variability models [2, 9, 10]. Afterwards, the features and their behavior are implemented, resulting in the solution space.

In contrast, our precondition is that cloned legacy systems exist, each defining its own solution space. From these, a product line can be extracted to enable systematic reuse and to reduce costs. This is referred to as the *extractive* approach [24]. However, the different solution spaces should share common but also contain unique parts, which we depict as a Venn-diagram in Figure 3. The initial step towards a product line is to map these solution spaces in a single problem space, reversing the proactive order.

Our goal in this paper is to describe a process to map existing solution spaces towards a single problem space. For this, we aim to fulfill the following requirements:

Req-1 Our process itself is independent of specifics of the cloned systems, such as, programming paradigms or languages.

Req-2 Our process can be semi-automated to reduce manual analysis efforts.

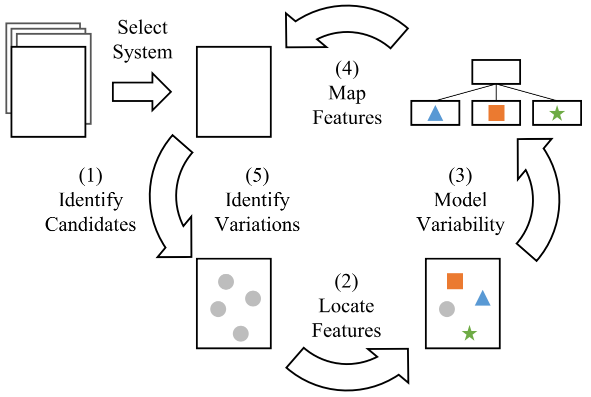

Figure 4: Step-wise process to locate and map features in cloned legacy systems.

Req-3 Our process enables step-wise refinements of the problem space (i.e., variability model), allowing developers to evaluate intermediate results.

Req-4 Our process investigates each system only once to reduce costs and limitations of pairwise comparison [16], namely the high number of necessary comparisons and hidden information in the results.

Based on these requirements, we propose a generally applicable process that can reduce the efforts and errors of mapping variability in cloned systems. Thus, we aim to reduce the *adoption barrier* [8] for product lines and improve the management of cloned systems.

3.2 Mapping Process

Our process, which we depict in Figure 4, is based on the idea proposed by Koschke et al. [23] and integrates steps that developers perform for feature location [42]. In contrast to Koschke et al. [23], we do not assume that an expert designs a model in advance and compares it to the system's architecture. Instead, our goal is to completely identify the variability of systems within their source code and to reverse engineer a corresponding variability model. While we focus on source code in this paper, our process can also be adapted for other artifacts, such as documentations.

As input, our process requires the source code of all systems that shall be analyzed. Initially, we aim to identify and map features only for one selected system. We do this to create a complete variability model and mapping for this base system (s_{base}) that we can later extend. During the process, we refine the variability model to represent all reused parts in the systems and map features to their corresponding code. To do this, we benefit from code-clone detection and feature location techniques. We describe how a corresponding mapping is presented in Section 4.

1. Identify Candidates. Starting with *n* different systems, we first select a base system from which we start our analysis. As we illustrate in Step 1 in Figure 4, we first identify *feature candidates*. These candidates represent artifacts that potentially belong to a feature. For our process, we utilize the fact that several cloned legacy systems exist. Hence, we apply code clone detection to identify cross-system clones between s_{base} and other systems. These cross-system clones are our first candidates for potential features.

2. Locate Features. To this point, we found trivial commonalities between the systems that represent feature candidates. However, we need to analyze if we found actual features and if these are complete, which is why we now evaluate the feature candidates (Step 2 in Figure 4). Several candidates may belong to the same feature but also may not cover all code belonging to this one. To support this task, feature location techniques can be used to identify additional code of a feature. Still, in the end a developer has to manually decide which candidate is part of which feature and if it is completely located [6, 22]. The result of this step is a full mapping of the source code to features.

3. Model Variability. In the third step, we create a variability model that represents the feature dependencies of the analyzed systems. To create this model, approaches to reverse engineer variability models [39] or dependencies, for example, in preprocessor-based systems [29], can be adopted to enable automation. Still, manual effort is necessary for these approaches, mainly because feature dependencies can only partly be extracted from a system's source code. In addition to creating the model, features are also mapped to the base system to enable traceability.

4. Map Features. After the previous three steps, we have a distinct mapping between a variability model and a system's source code. In the fourth step of our process, we map another system to these features and the model. As we know exactly the code we have identified and its position, this step can be fully automated. Still, we have to be aware that lines of code might be removed, changed, or added compared to previous systems, which requires adaptations of existing code-clone detection approaches. The result of this step is a mapping of all previously identified features to a newly selected system's source code.

5. Identify Variations. In a cloned system, existing features might be changed or completely new features are introduced. Hence, we have to identify and assess such *variations* in the newly selected system. Again, this can be supported with approaches for feature location (analyzing the system by itself) and code clone detection (comparing it with another system). Still, at this point we can exclude already identified parts of the source code from our analysis, reducing the manual effort from this point forward.

System-Wise Refinement. The second to fifth step are repeated for any new system, as we show in Figure 4. After each cycle, the new system is mapped to the variability model, which is refined if variations occur. In the end, we can derive a full variability model that represents all features within the legacy systems. This allows developers to analyze the systems and extract reusable features from the source code.

Overall, the results of our process support developers in several ways. Our main objective is to enable the extraction of a software product line. We do this as we identify commonalities and differences in the legacy systems, construct and transfer a variability model, and, thus, prepare refactorings [17]. Furthermore, our process helps to continue developing with cloned systems but improve change propagation due to the implemented mapping [30]. Hence, it is easier to identify which changes must be done in which systems and to assess their impact.

4 VISUALIZATION CONCEPT

During our process, a developer needs support to find, analyze, and map features in the source code. For this reason, a visual representation is necessary. In the context of investigating the reuse potential of cloned systems, Duszynski and others propose to use *occurrences matrices* for analysis and bar charts for visualization [15, 16]. Both techniques are more suitable as the number of systems grows than Venn-diagrams, which become unintelligible. However, while the simple distribution of common and variable code among systems is suitable to evaluate reuse potential we see four shortcomings of occurrence matrices and bar charts:

(1) Systems must be compared pairwise and, thus, multiple times to detect all commonalities. For instance, pairwise comparison can lead to transitive dependencies: A clone is located in $s_{base} \wedge s_1$ and also in $s_{base} \wedge s_2$. Hence, $s_1 \wedge s_2$ also contains this clone, which can only be exploited while comparing these systems or aggregating results. Overall, $n * (n-1)/2$ pairs are necessary for n different systems [16], contradicting the idea of our process.
(2) Occurrences matrices map only atomic elements (i.e., lines of code) that can be unambiguously identified. At the end, the matrices are merged to aggregate information for all systems. As a result, the displayed order of elements can be different from the real one, which is problematic for mapping source code to features and analyzing variations. Furthermore, occurrences matrices are suitable to be used by tools but are rather complicated to analyze for humans.
(3) Only two entities (e.g., files with the same path) are compared. Additional analyses are necessary to identify atomic elements that are moved to a different entity. To address this point, code clone detection seems more promising to compare source code.
(4) Presenting the results in bar charts [16] hides important information about the location and size of specific commonalities. A direct mapping is not possible in this case due to the aggregation of information.

We address these points by using an adapted visualization concept based on occurrence matrices, bar charts, and code clone reports.

More precisely, we implemented a prototype that transforms results of the code clone detection tool *Clone Detective* [20] into the representation we depict in Figure 5. In this view, a specific feature (i.e., `Game`) is displayed in its distribution among files and systems. This representation shows the commonalities and variabilities between different systems in more detail and does not condense information. For improved analysis capabilities, we plan to also transform the views to display a specific class and the distribution of features within it, identical to the presentation of code clone detection tools. Finally, the variability model as well as corresponding source code is mapped towards these views, allowing users to directly access and navigate between them. Hence, it is possible to open and investigate the mapped feature code of each systems directly from this view.

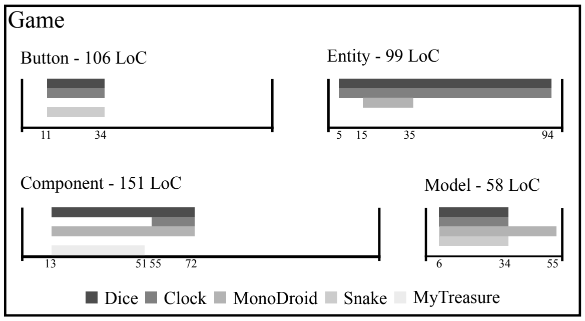

Figure 5: Sketch of the visualization concept to display and analyze commonalities and variability in cloned systems.

5 EVALUATION

To evaluate our process, we performed a case study on a set of cloned systems. In this sections, we describe the tools we used, our research questions and method, and discuss our results. Finally, we review potential threats to validity.

5.1 Tooling

For the code-clone detection during our evaluation, we used *Clone Detective*, which is part of the *ConQAT* framework [12, 20]. This tool uses token-based analysis, is language independent, and finds most type-2 clones [31]. To enable cross-system code clone detection, we simulated that all our subject systems belong to a single project. We manually analyzed the code clones using *Clone Detective*'s user interface. In addition, we used our prototype to transform the output towards our visualization concept. We decided against further tools to focus on the general applicability and tool independence of our process.

5.2 Subject Systems and Reference Study

To evaluate our process, we require a set of cloned systems. While it does not matter in which programming language they are developed, it is necessary that they contain common and custom code. Such custom code is a result of, for instance, adding new functionality to a cloned system or refactoring it. In addition, a documentation or a reference study should be available to evaluate whether the found features are meaningful.

Due to these requirements, we decided to use a set of five systems of *ApoGames*[1] that implement different games for Android and are written in Java. For clarity, we always use shortened versions for feature and file names. This means that we suppress any identifying naming conventions and always refer to these as an entity that exists in several of these games.

In a *reference study* [17], we used these systems to evaluate refactorings that automatically extract code clones into feature modules. For this, we also implemented a rename refactoring to address type-2 clones. We present detailed results for this automatic approach that we can use to compare our mapping against. In Table 1, we summarize the results of this previous study [17]. The focus was

[1] http://apo-games.de/index_android.php, 22.03.2017

Table 1: Results of the automatic migration in the previous study [17].

System	Base #SLOC	Migrated #SLOC	ΔSLOC
Dice	2,504	1,346	1,158
Clock	3,584	2,696	888
MonoDroid	6,483	5,490	993
Snake	2,946	1,786	1,160
MyTreasure	5,322	4,483	839
Common	-	1,779	-
Total	20,839	17,580	-15.6%

the automatic extraction of features based on cloned fields and methods and, at the end, 1,779 SLOC were migrated into modules.

Due to the automation, the final product line contains 15 common and 5 custom (i.e., unique code of each system) features. However, the resulting model contains mainly cross-tree constraints between common and custom features, neglecting dependencies between the common features. We also found that the automatic refactoring partly extracted rather small code fragments of the same class into different modules. For example, we investigated the file `ApoEditor` that was part of all five systems. In the final product line, this file was part of seven common features. However, six of these refinements added only one or two statements to the original source code and two features contained solely extensions to this file. Such a fine granularity, which is the result of automatic refactorings, seems unsuited for modules and it is questionable whether these represent useful features.

Overall, we can use this reference study to compare our mapping and also illustrate the necessity of our process while extracting product lines: Applying automatic refactorings only after identifying features in the source code will result in *i)* a suitable variability model, *ii)* improved scope of automated refactorings, and *iii)* corresponding mappings.

5.3 Research Questions

Based on the previously described subject systems and reference study, we derive the following research questions:

RQ-1: How much of the extracted code in the reference study do we identify? First, we assess to which extent we identify the same code clones and features as our reference study. This way, we aim to investigate how complete we mapped the systems with our process compared to automatic refactorings.

RQ-2: How do the differences between our mapping and the reference study look like? Based on the first research question, we aim to analyze potential differences in the found code clones and features. More precisely, we discuss the granularity of features we identified compared to the automatic approach.

RQ-3: How does a feature model for the subject systems look like? Finally, we will derive a feature model based on our results. We cannot show that the model is correct, due to missing documentation and domain knowledge. Still, based on the results derived during the process, we argue that it represents a suitable structure for the systems.

Table 2: Identified and mapped features.

Feature	Function	#Classes	∑SLOC				
			Dice	Clock	MonoDroid	Snake	MyTreasure
Base	Core functionality	2	275	275	275	199	275
Editor	Create levels	1	61	34	0	27	0
Game	Game objects	8	347	129	118	253	20
Load	Load levels	2	282	282	282	282	267
Menu	Menu	2	295	61	201	93	0
Puzzle Chooser	Select levels	2	121	111	0	121	0
Save	Save levels	1	17	17	17	17	17
Total			1,398	909	893	992	579

5.4 Methodology

We performed the process we described in Section 3 as follows. We analyzed one base system (i.e., *Dice*) by identifying cross-system code clones with all other systems. Based on this, we located features and manually determined which clones belong to them. We derived an initial feature model and refined it while mapping the found features to the other systems, for instance, we changed dependencies from mandatory to optional. Note that we did not investigate variations within the other systems as is emphasized in our process. Depending on our decisions whether specific statements are additions to a feature or not, our results would change. Hence, we may bias our findings due to our personal opinion on a feature's size. In addition, we will reason about necessary efforts and benefits of automatic refactorings. This is why we solely focused on the detected code clones in this evaluation.

To answer our first research question, we measure the lines of code for the identified features. We compare the sizes of common code in each subject system with our reference study. To answer our second research question, we manually investigate the differences between the features created in the reference study and these we identified. We analyze their number, sizes, and distributions in the different systems. To answer our third research question, we model the variability of the systems as a feature diagram. For this, we analyzed which features are present in which systems and manually determined dependencies between them, for instance, based on method calls. To summarize our findings, we discuss benefits of our process and its combination with automatic refactorings.

5.5 Results

In Table 2, we describe the features we identified, the functionality they implement, and their sizes. Overall, we found seven common features that are shared among the systems. These features are mostly part of one or two classes but `Game` is scattered among eight different classes. The source lines of code vary heavily, from 17 SLOC for the `Save` feature up to 347 SLOC for `Game`.

We remark that not all lines of code are part of each system. Instead, subsets exist only in two or three systems. For example, we depict part of the feature `Game` in Figure 5. There, common code exists between up to four systems within the class `Component`. However, the first and last part do only exist in three systems. Thus, the numbers we present in Table 2 provide only an overview of the size but not the structure of features.

Furthermore, we see that the distribution of common code varies heavily between the different systems. In particular, we found that some features are not part of `MonoDroid` and `MyTreasure` at all. We see that `Dice` contains 1,398 SLOC in common features in contrast to `MyTreasure` for which we found only 579 SLOC. The other three systems range from 893 to 992 SLOC in common features.

5.6 Discussion

Regarding **RQ-1**, we found that our results vary from the reference study. We identified that `Dice` and `Clock` contain more common code than was extracted with the automatic refactorings. In contrast, we mapped less feature code within `Snake` and `MyTreasure` with approximately 160 SLOC and 260 SLOC respectively. Overall, we see that we found more feature code within the analyzed base system (`Dice`). Hence, the manual analysis we performed in our process improves localization of common features. Still, because we only mapped the found code in other variants but performed no additional analysis, we missed commonalities between the other systems.

Regarding **RQ-2**, we found that we also identified smaller and unique code artifacts in the base system compared to the reference study. This is not surprising as we manually analyzed it and were not solely focusing on common fields and methods. Considering the sizes, our mapping performs similar to automatic refactoring if the same source code is mapped. While we thought that our process would require far more effort due to the manual analysis, we also required similar analysis in the reference study. There, many fields and methods could only be migrated into modules after renaming, which also required manual assessments. Still, this seems to be a limitation of the used clone detection tool rather than the refactoring itself.

A critical point to answer **RQ-2** is which common code is part of which features. The automatic refactorings created a feature with 619 SLOC that is shared among all systems [17]. However, we identified no feature of this size and especially not one that is part of all systems. When comparing the results, we found that automatic refactorings, not surprisingly, migrated parts of several features we identified into a single one. For instance, one refactored module contained parts of `Game`, `Base`, `Menu`, `Load`, and `Save`. Thus, we argue that it is problematic to identify meaningful features based solely on automatic clone refactoring.

Another difference is that in the reference study common code was also refactored if it appeared multiple times in the same system.

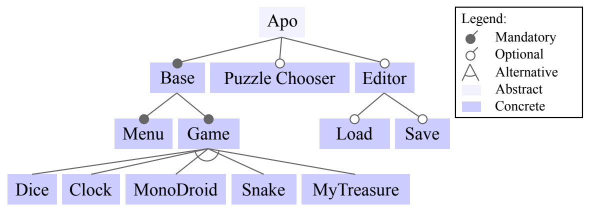

Figure 6: Extracted feature model for ApoGames.

The focus on pure code clones and identical naming resulted in more extracted source code. This was mostly due to system specific methods overwriting or copying parts of existing ones. As explained previously, we did not map this code to specific features because we do not assume that they represent an actual feature. However, this is not a limitation in our process but in our evaluation. To this point, we argue that domain knowledge cannot be fully replaced to decide on the size and scope of features in cloned systems.

Regarding **RQ-3**, we display the final feature model we derived from our analysis in Figure 6. Overall, the investigated systems do contain 3 mandatory and 4 optional features. We mainly derived optional and mandatory dependencies based on the sizes and appearances of features in specific systems, which we show in Table 2. In addition, we investigated which features call each other to derive parent-child dependencies. While the resulting model is not perfect and mainly considers features which share code among systems, we argue that it represents a reasonable structure.

Summary. In summary, we find that automatic refactorings but also our process have advantages and disadvantages. Automatic refactorings ease the extraction of code clones into modules significantly. However, they heavily depend on the used code-clone detection approach and potentially on identical naming to identify feature code. In addition, it seems problematic that they extract all found clones without regarding their scope. Hence, features may not represent a meaningful unit and may contain code artifacts that do not belong to a feature.

We argue that our process provides an initial step to identify and map features before migrating them. Still, the disadvantages are increased costs as extended manual analysis are necessary. An integrated tooling that utilizes both and supports them with additional techniques seems to be a promising approach. This way, meaningful features could be identified with reduced costs.

5.7 Threats to Validity

A potential threat to the validity of our evaluation are the considered subject systems. Firstly, these are rather small, which will rarely be the case in real-world scenarios. However, larger cloned systems that contain different changes and are developed independently are rare. In addition, studies that provide detailed information to compare our findings against do not exist. These points are fulfilled by the *ApoGames* systems [17]. Secondly, the systems were developed with specific naming conventions: Classes have prefixes that identify a system. For example, levels are defined in `ApoClockLevel` in *ApoClock* but `ApoDiceLevel` in *ApoDice*. This convention can impact the results of our code-clone detection tool. Still, we especially used a tool that is capable of handling renaming (i.e., type-2 clones) to some extent and manually assessed the results to limit this threat.

To detect code clones in the systems, we used *Clone Detective*. While our reference study [17] uses another tool, we especially selected a tool that can detect type-2 clones to consider the renaming refactoring preformed there. Other than this, we used the same setting of detecting clones only for more than ten tokens, resulting in the same potential threat: The detection may have missed shorter clones belonging to a feature. As defined in our process, we limited this problem by manually assessing the feature candidates and analyzing potential variations.

Due to our mainly manual analysis, we may have missed features. Still, the evaluation illustrates the potential of our approach and we aim to develop and assess corresponding tooling. A more significant concern is the basic assumption of our approach that domain knowledge is missing. In the context of our evaluation this may result in a different variability model than intended by the developer. We carefully checked our findings and compared them to a reference study to reduce both threats.

6 RELATED WORK

Duszynski et al. [16] present an approach to reverse engineer quantitative information about commonality and variability of cloned systems. This information is visualized using occurrence matrices and bar charts. Although the approach is adaptable to other similarity detection tools, the presented instantiation relies on `diff`, which is more sensitive to formatting, renaming, and reordering than the clone detector we use. Additionally, their visualizations mainly support management decisions, such as cost estimations, whereas our visualizations can also guide developers through the analysis process.

Xue and various co-authors locate features in legacy systems in a "top-down/bottom-up sandwich" fashion. They combine feature model differencing at requirements level with clone differentiation [43], program dependency graphs, and information retrieval techniques at implementation level [44–47]. Their approach requires as input the feature models of all variants, which may constitute a major challenge as the necessary information is often unavailable for legacy systems. By contrast, we locate and map features in only one variant and step by step transfer the mapping to other variants.

Several frameworks of migrating cloned variants were proposed, for example by Rubin et al. [34] and Martinez et al. [28], which abstract conceptual activities, e.g., similarity detection, from concrete techniques, e.g., clone detection. Similarly, the steps of our process are also independent from specific techniques or tools. In addition to presenting a process, we also provide an implementation of that process, and demonstrated its feasibility on a case study.

Ziadi et al. [48] propose an algorithm to identify feature candidates in legacy variants by processing structural models of those variants. After manual filtering of the feature candidates, a feature model can be built. While their algorithm is likely to reduce the amount of manual work compared to our approach, it is programming-language dependent whereas the clone detector we use is not. It would be interesting future work to compare our feature location approach and theirs on the ApoGames.

7 CONCLUSION

Identifying and mapping features in cloned legacy systems is an expensive but important task, for instance, to facilitate the maintenance of these systems or to extract a product line. We proposed a semi-automatic process to reverse-engineer commonalities and variabilities in such systems incrementally. Furthermore, we introduced a visualization approach to support developers during this process. The results of our evaluation show that our process is able to identify meaningful features and can provide the basis for, or improve, other approaches, for instance, product line extraction and automatic refactorings.

In the future, we aim to extend our prototype to support direct mappings to the source code, enable different views, and improve its integration into other tools. Additionally, we want to provide an integrated environment for our whole process to guide users during their analysis. Integrating our automated refactorings that migrate mapped features accordingly can extend our approach. With these tools, we will conduct additional case studies especially in industrial contexts to investigate their practical suitability.

ACKNOWLEDGMENTS

This research is supported by DFG grants LE 3382/2-1, SA 465/49-1, and Volkswagen Financial Services AG.

REFERENCES

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

[2] Sven Apel and Christian Kästner. 2009. An Overview of Feature-Oriented Software Development. *JOT* 8, 5 (2009), 49–84.

[3] Wesley K. G. Assunção and Silvia R. Vergilio. 2014. Feature Location for Software Product Line Migration: A Mapping Study. In *SPLC.* ACM, 52–59.

[4] Veronika Bauer and Benedikt Hauptmann. 2013. Assessing Cross-Project Clones for Reuse Optimization. In *IWSC.* IEEE, 60–61.

[5] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VAMOS.* ACM, 7:1–7:8.

[6] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. 1993. The Concept Assignment Problem in Program Understanding. In *ICSE.* IEEE, 482–498.

[7] Lianping Chen and Muhammad Ali Babar. 2011. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *IST* 53, 4 (2011), 344–362.

[8] Paul C. Clements and Charles W. Krueger. 2002. Point/Counterpoint: Being Proactive Pays Off/Eliminating the Adoption Barrier. *Software* 19, 4 (2002), 28–30.

[9] Paul C. Clements and Linda M. Northrop. 2001. *Software Product Lines: Practices and Patterns.* Addison-Wesley.

[10] Krzysztof Czarnecki and Ulrich W. Eisenecker. 2000. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley.

[11] 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.* ACM, 173–182.

[12] Florian Deissenboeck, Elmar Juergens, Benjamin Hummel, Stefan Wagner, Benedikt Mas y Parareda, and Markus Pizka. 2008. Tool Support for Continuous Quality Control. *Software* 25, 5 (2008), 60–67.

[13] Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. 2013. Feature Location in Source Code: A Taxonomy and Survey. *SMR* 25, 1 (2013), 53–95.

[14] 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.* IEEE, 25–34.

[15] Slawomir Duszynski. 2010. Visualizing and Analyzing Software Variability with Bar Diagrams and Occurrence Matrices. In *SPLC.* Springer, 481–485.

[16] Slawomir Duszynski, Jens Knodel, and Martin Becker. 2011. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *WCRE.* IEEE, 303–307.

[17] Wolfram Fenske, Jens Meinicke, Sandro Schulze, Steffen Schulze, and Gunter Saake. 2017. Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line. In *SANER.* IEEE, 316–326.

[18] Stefan Fischer, Lukas Linsbauer, Roberto Erick Lopez-Herrejon, and Alexander Egyed. 2014. Enhancing Clone-And-Own with Systematic Reuse for Developing Software Variants. In *ICSME.* IEEE, 391–400.

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

[20] Elmar Juergens, Florian Deissenboeck, and Benjamin Hummel. 2009. CloneDetective - A Workbench for Clone Detection Research. In *ICSE.* IEEE, 603–606.

[21] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study.* Technical Report. SEI, CMU.

[22] Christian Kästner, Alexander Dreiling, and Klaus Ostermann. 2014. Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features. *TSE* 40, 1 (2014), 67–82.

[23] Rainer Koschke, Pierre Frenzel, Andreas P. Breu, and Karsten Angstmann. 2009. Extending the Reflexion Method for Consolidating Software Variants into Product Lines. *SQJ* 17, 4 (2009), 331–366.

[24] Charles W. Krueger. 2002. Easing the Transition to Software Mass Customization. In *PFE.* Springer, 282–293.

[25] Jacob Krüger. 2017. Lost in Source Code: Physically Separating Features in Legacy Systems. In *ICSE.* IEEE, 461–462.

[26] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *SPLC.* ACM, 354–361.

[27] Miguel A. Laguna and Yania Crespo. 2013. A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring. *SCICO* 78, 8 (2013), 1010–1034.

[28] Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. 2015. Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach. In *SPLC.* ACM, 101–110.

[29] Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. 2015. Where do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *TSE* 41, 8 (2015), 820–841.

[30] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with Variantsync. In *SPLC.* ACM, 329–332.

[31] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. 2009. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *SCICO* 74, 7 (2009), 470–495.

[32] Julia Rubin and Marsha Chechik. 2013. A Framework for Managing Cloned Product Variants. In *ICSE.* IEEE, 1233–1236.

[33] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering: Product Lines, Languages, and Conceptual Models.* Springer, 29–58.

[34] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2013. Managing Cloned Variants: A Framework and Experience. In *SPLC.* ACM, 101–110.

[35] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing Forked Product Variants. In *SPLC.* ACM, 156–160.

[36] 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. *STTT* 14, 5 (2012), 477–495.

[37] Klaus Schmid and Martin Verlage. 2002. The Economic Impact of Product Line Adoption and Evolution. *Software* 19, 4 (2002), 50–57.

[38] Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. 2006. Feature Diagrams: A Survey and a Formal Aemantics. In *RE.* IEEE, 139–148.

[39] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. 2011. Reverse Engineering Feature Models. In *ICSE.* IEEE, 461–470.

[40] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *ICSME.* IEEE, 151–160.

[41] Thomas Thüm, Christian Kästner, Sebastian Erdweg, and Norbert Siegmund. 2011. Abstract Features in Feature Modeling. In *SPLC.* IEEE, 191–200.

[42] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *SMR* 25, 11 (2013), 1193–1224.

[43] Zhenchang Xing, Yinxing Xue, and Stan Jarzabek. 2011. CloneDifferentiator: Analyzing Clones by Differentiation. In *ASE.* IEEE, 576–579.

[44] Yinxing Xue. 2011. Reengineering Legacy Software Products into Software Product Line Based on Automatic Variability Analysis. In *ICSE.* ACM, 1114–1117.

[45] Yinxing Xue. 2013. *Reengineering Legacy Software Products Into Software Product Line.* Ph.D. Dissertation. University of Singapore.

[46] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. 2010. Understanding Feature Evolution in a Family of Product Variants. In *WCRE.* IEEE, 109–118.

[47] Yinxing Xue, Zhenchang Xing, and Stan Jarzabek. 2012. Feature Location in a Collection of Product Variants. In *WCRE.* IEEE, 145–154.

[48] Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. 2012. Feature Identification from the Source Code of Product Variants. In *CSMR.* IEEE, 417–422.