Migrating Java-Based Apo-Games into a Composition-Based Software Product Line

Jamel Debbiche
Chalmers | University of Gothenburg
Gothenburg, Sweden

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany

Oskar Lignell
Chalmers | University of Gothenburg
Gothenburg, Sweden

Thorsten Berger
Chalmers | University of Gothenburg
Gothenburg, Sweden

ABSTRACT

A software product line enables an organization to systematically reuse software features that allow to derive customized variants from a common platform, promising reduced development and maintenance costs. In practice, however, most organizations start to clone existing systems and only extract a software product line from such clones when the maintenance and coordination costs increase. Despite the importance of extractive software-product-line adoption, we still have only limited knowledge on what practices work best and miss datasets for evaluating automated techniques. To improve this situation, we performed an extractive adoption of the Apo-Games, resulting in a systematic analysis of five Java games and the migration of three games into a composition-based software product line. In this paper, we report our analysis and migration process, discuss our lessons learned, and contribute a feature model as well as the implementation of the extracted software product line. Overall, the results help to gain a better understanding of problems that can appear during such migrations, indicating research opportunities and hints for practitioners. Moreover, our artifacts can serve as dataset to test automated techniques and developers may improve or extend them in the future.

CCS CONCEPTS

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

KEYWORDS

Software product line, Extraction, Case Study, Feature model, FeatureHouse, Apo-Games

1 INTRODUCTION

A software product line allows an organization to implement a family of similar software products based on a common platform [1, 9]. In such a platform, developers systematically manage and reuse features that implement mandatory and optional functionalities of the products. Developers can derive a specific product by configuring (selecting) the features that shall be part of the product, which is then built in an automated step. Software product lines promise several benefits, such as reduced development and maintenance costs, improved quality, and faster time to market [14, 34]. Still, despite such benefits, most organizations fear the initially higher investment to set up a reusable platform [8, 20] and only later extract [15] one out of a set of existing software variants. In practice, such variants often emerge from cloning one product and adapting it to the needs of another customer, referred to as clone-and-own [7, 11, 32].

While the extractive approach of software-product-line adoption is the most common one in practice [6, 12], most techniques (e.g., feature location) that aim to automatically analyze the cloned systems face severe limitations [18, 22, 23, 30]. To overcome such limitations, we need to provide common ground truths that allow to evaluate and compare techniques based on real-world artifacts [22, 33]. As a step in this direction, Krüger et al. [21] have contributed a challenge case that comprises 20 Java and five Android games that a single developer implemented based on the clone-and-own approach. They have challenged the research community to provide additional artifacts (i.e., feature models, feature locations, code smells, architectures, migrated software product lines) that can serve as datasets to evaluate automated techniques for analyzing these games.

In this paper, we report our experiences on tackling the fifth challenge of migrating game variants into a software product line. To this end, the first two authors of this paper analyzed five Java games and migrated three of these (due to time restrictions) into a composition-based software product line implemented with FeatureHouse [2]. We further documented our activities, results, and experiences for each step. Our contributions are:

• We describe the applied analysis and migration process that resulted in the extracted software product line.
• We report the challenges that we experienced during this process as lessons learned.
• We provide a repository\(^1\) with all artifacts we created, namely the feature model and code base for the software product line.

\(^1\)https://bitbucket.org/Jacob_Krueger/splc2019_featurehouse_apo-games_spl
Table 1: Selected subject systems for this study. Asterisks (*) denote games that we analyzed, but did not transform.

<table>
<thead>
<tr>
<th>Name</th>
<th>Year</th>
<th>SLOC</th>
<th>Game Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>* ApoCheating</td>
<td>2006</td>
<td>3,960</td>
<td>Level-based puzzles</td>
</tr>
<tr>
<td>* ApoStarz</td>
<td>2008</td>
<td>6,454</td>
<td>Level-based puzzles</td>
</tr>
<tr>
<td>ApoCarus</td>
<td>2011</td>
<td>5,851</td>
<td>Endless runner</td>
</tr>
<tr>
<td>ApoNotSoSimple</td>
<td>2011</td>
<td>7,558</td>
<td>Level-based puzzles</td>
</tr>
<tr>
<td>ApoSnake</td>
<td>2012</td>
<td>6,557</td>
<td>Level-based puzzles</td>
</tr>
</tbody>
</table>

For simplicity, the repository comprises a FeatureIDE [26] project that others can import. Our results are helpful for researchers and practitioners to better understand how variants can be migrated into a software product line. Moreover, researchers can use our implementation as baseline to evaluate and compare automated techniques or to incorporate more Apo-Games in the future.

## 2 METHODOLOGY

As we tackle the fifth challenge defined by Krüger et al. [21], we migrated cloned variants towards a software product line. Consequently, we also performed parts of other challenges (e.g., feature modeling) and provide corresponding artifacts (i.e., the feature model). In this section, we describe our applied methodology, comprising the subject systems we selected, our preparation for the transformation, the feature recovery process, and our actual transformation process. We remark that we did not follow a specific methodology (e.g., as described by Assunção et al. [3]), but employed and adapted activities that are regularly mentioned in the literature. Finally, we summarize this section by describing the resulting software product line.

### 2.1 Subject Systems

The Apo-Games comprise a set of 20 Java games that have been developed with the clone-and-own approach. Of these games, we selected five as subjects and provide an overview of these five in Table 1. We remark that we selected games that showed the most commonalities while playing them, indicating that they would contribute to a reasonable software product line. As we can see, the games are from different periods and cover between four to over seven thousand source lines of code. While four of them are part of the same sub-domain of games (puzzles), one is a representative for an endless running game. We included this one game from a different sub-domain, because we assumed more overlap for the same type of games, but also aimed to show that we can integrate rather different games into a software product line.

During our analysis, we faced some problems (cf. Section 3), due to which we only transformed three of the games. Moreover, we cleaned the source code to map these features, we found that several techniques did not align to the domain features we identified in our top-down analysis. Namely, But4Reuse suggested features, such as mousebuttonfunction, according to keywords that appear often in the code, but these keywords do not reveal the actual domain features. Due to this mismatch, we decided to rely on a manual transformation, resulting in a feature-wise migration of the games.

### 2.2 Preparation

Before the actual transformation, we analyzed existing tools to identify whether we could rely on one of them. However, we found that most existing techniques (e.g., for automatic feature location [30]) depend on specific artifacts (e.g., source code, models, documentation, git history) or additional tools that must be available to the developer. A particular problem in this regard is that most tools were not useful in our case, as they are discontinued, commercial or provide unsuitable results. For instance, But4Reuse [25] can suggest features and support extractive adoption, but the suggested features did not align to the domain features we identified in our top-down analysis. Namely, But4Reuse suggested features, such as mousebuttonfunction, according to keywords that appear often in the code, but these keywords do not reveal the actual domain features. Due to this mismatch, we decided to rely on a manual transformation, resulting in a feature-wise migration of the games.

While we did not find a technique that we could employ as is for the Apo-Games, we nonetheless adopted the described strategies for our own process. This let to the adoption of the sandwich approach [35] for our feature recovery process (cf. Section 2.3). We used the results, especially the feature mappings, to plan the transformation and to actually migrate variant features. During our analysis, we also decided to use FeatureIDE [26], as it supports various activities and variability mechanisms that we needed. Moreover, FeatureIDE is a plug-in for Eclipse, which allowed us to use other plug-ins more easily. For the variability mechanism, we used a composition-based technique and feature-oriented programming [29] in particular, supporting the physical separation of features [1, 13, 16]. Consequently, we selected FeatureHouse [2], which is directly supported by FeatureIDE and integrates more recent Java versions (i.e., compared to AHEAD [5]).

### 2.3 Feature Recovery

The first step of extracting a software-product line is to detect features and their dependencies in the legacy systems [4], which we defined in a feature model [1, 10]. Moreover, we had to also locate and map features in the source code [18]. To this end, we decided to employ a manual analysis comprising top-down and bottom-up strategies, referred to as sandwich approach [35].

**Top-Down Analysis.** As first step of our top-down analysis, we played each subject game and identified its visible features. We listed the features and performed a pairwise comparison between the games to identify commonalities and variability. In a second step, we reverse engineered class diagrams for each game, using the Visual Paradigm plug-in for Eclipse. This extraction helped to understand that, except for ApoCheating, all subject games share the same architecture.

During the top-down analysis, we identified 33 features throughout all games (i.e., the ones we could not match to But4Reuse’s suggestions). However, when we started to investigate the actual source code to map these features, we found that several technical features were still missing in our documentation. To address

---

1. [https://marketplace.eclipse.org/content/unnecessary-code-detector](https://marketplace.eclipse.org/content/unnecessary-code-detector)
2. [https://marketplace.eclipse.org/content/visual-paradigm-eclipse](https://marketplace.eclipse.org/content/visual-paradigm-eclipse)
As first step of the actual transformation, we migrated the common code base of all games into a FeatureIDE project. This common part was rather small, comprising the main panel and the game engine. While this base could already be compiled for testing, it only showed a black window.

We then incrementally added features of the games into the software product line. A particular problem in this regard was to ensure the correct behavior of the transformed code, as we needed at least one complete game for testing. Moreover, we had to carefully plan which features to implement first, due to their domain and technical dependencies. We partly addressed the dependencies by changing different data structures into more flexible and variable ones. For instance, we changed most arrays into data structures of the List collection to ensure that the games were executable despite missing or disabled features. However, such changes and the remaining dependencies still challenged testing.

During the transformation, we also added variant-specific features that do not contribute to reuse, but only variability. Due to time restrictions, we finally migrated 23 features, which can be used to instantiate three of the legacy games (cf. Table 1), and for which we show their dependencies in Figure 1. Overall, we emphasized variability over reuse, and thus introduced not only features based on clones and variations, but actual domain features [23]. As a result, the software product line allows to configure 56 games.

### Table 2: Statistics of the extracted software product line compared to the original games. Products refers to the games that we could execute based on the implemented features.

<table>
<thead>
<tr>
<th>Feature Model Statistics</th>
<th>Features</th>
<th>Conc. Impl.</th>
<th>Constraints</th>
<th>Products</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>47</td>
<td>42</td>
<td>23</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Source Code Statistics</th>
<th>Legacy Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>SLOC</td>
</tr>
<tr>
<td></td>
<td>SLOC Legacy</td>
</tr>
<tr>
<td></td>
<td>Reduction</td>
</tr>
<tr>
<td>55</td>
<td>13,932</td>
</tr>
<tr>
<td></td>
<td>19,966</td>
</tr>
<tr>
<td></td>
<td>-30.22%</td>
</tr>
</tbody>
</table>

Conc.: Concrete features; Impl.: Implemented features

2.5 The Software Product Line

In Table 2, we provide a brief overview of the software product line we extracted. As aforementioned, we identified and modeled 47 features with 16 cross tree constraints. We show the 23 implemented features and their dependencies as a feature model in Figure 1. In the diagram, we can see that only few abstract features exist for structuring other features. On the positive side, we have only few features that are directly connected to a specific game or game type (e.g., SnakeInteractive). Thus, most of the features we identified and implemented seem well suited for reuse.

Considering the source code, we implemented 55 classes comprising 13,932 SLOC in our software product line. As the three legacy games totaled at 19,966 SLOC, we achieved a reduction of roughly 30%. Some of this reduction is the result of our code cleansing in the beginning. However, this rather shows the positive impact an extractive adoption can have on code quality. Also, composition-based variability mechanisms usually result in additional source code, because new classes are needed to implement feature modules. So, we argue that our software product line resulted in a reasonable complexity and size on implementation level.

Finally, our software product line allows to configure 56 games, not only the three legacy games, showing that a variable platform can immediately increase the product portfolio. To test the correct behavior, we configured, generated, and ran all 56 games successfully. However, we remark that not all configurations result in a fully playable game, yet. This is especially true for games that contain Apoculus game-play, as this variant is not completely migrated nor as configurable as the other two games (ApoNotSoSimple and ApoSNAKE). Due to the changes we employed during the transformation, a code-level comparison of the original games and the instantiated variants is not useful, but we were able to play the original games as variants of the software product line.

### 3 LESSONS LEARNED

During the transformation of the three games into a software product line, we experienced five main challenges.

**Abstraction Level of Features.** While identifying features with the sandwich approach, we found that different abstraction levels can result in varying sets of features. This highlights the importance of combining such analyses. Moreover, this experience underpins that we need to decide on the purpose and extent of a feature model before its actual design, also deciding how to structure features [28]. Despite the mismatch that we experienced with But4Reuse, we also
argue that such tools can support the semi-automated identification of features on the technical level.

Planning of Features. For the migration of a software product line, limited resources are available. We experienced that careful planning and extending analysis activities can considerably simplify the actual transformation of the variants. Most importantly in our experience was to know what dependencies between features and to their artifacts exist. This knowledge helped to get an intuition of the effort that would be necessary during the actual migration.

Updates on the Feature Model. While we implemented features, we experienced that we identified and added new constraints to the feature model. However, this was problematic as, at the beginning, many features were not fully functioning, meaning that many constraints were missing, too. Thus, constant updates to the feature model and the corresponding source code added effort to the extraction process.

Complexity of Superimposition. FeatureHouse uses superimposition to compose various features with the same class and method names, resulting in a customized variant. While this works properly and is a concept related to inheritance, it also poses challenges when the size of a project grows. As different features can have classes with the same name and physically separate related code, it became challenging to understand and identify what classes to change during maintenance and updates [17, 19, 31].

Tracking Evolution. We used a version control system and different branches to document our progress. Still, we found it challenging to understand later on what happened at what point in time, as feature code was scattered and tangled throughout the commit history—which is a good argument to use variation control systems [24]. For instance, two features may be changed to enable a third one (e.g., implementing updates to the feature model) and all changes are part of one commit.

4 THREATS TO VALIDITY

Internal Validity. The major threat to the results of this work is the missing interaction with the original developer. Instead, we relied on code analysis, reading comments, and reverse engineering architectures. Consequently, while we carefully analyzed and checked the results, we cannot ensure that we understood all parts of the Apo-Games perfectly or that another (i.e., the original) developer would derive the same implementation for a software product line. Still, as we were able to instantiate and run the original and 53 more games, we argue that our implementation is reasonable and can serve as a dataset for transformation and analysis techniques or to incrementally add more games.

External Validity. The subject systems are small compared to industrial or established open-source systems. However, they are publicly available and have been truly developed based on the clone-and-own approach, for which only few real-world subject systems exist. In addition, games contribute to more and more software systems, also exhibiting similar development patterns and characteristics as other software [7]. Thus, we cannot overly generalize the results, but they still yield important insights into the extraction of software product lines.

5 CONCLUSION

In this paper, we described a case study during which we migrated three Java-based Apo-Games into a composition-based software product line. For this purpose, we conducted a manual analysis and transformation process, resulting in the following insights:

- Extracting a composition-based software product line is challenging and time consuming, due to the changes that are needed to enable composition.
- During code transformations, we highly recommend to ensure that the software product line can always be tested to ensure the correct transformation of features.
- Incrementally adopting new features facilitates the extraction, as various artifacts (i.e., the feature model) need updates and must be tested.

Besides our lessons learned, we also provide a public repository comprising the extracted software product line and feature model.

In future work, we aim to extend the current artifacts and provide more detailed insights into the migration process. To this end, we want to replicate the extraction to verify our results and improve the validity of our insights. Finally, we plan to improve our dataset so that we can use it as ground-truth to evaluate automated techniques for extracting software product lines.

Acknowledgments. This work is supported by the ITEA project REVaMP² funded by Vinnova Sweden (2016-02804), and the Swedish Research Council Vetenskapsrådet (257822902). We thank Jennifer Horff for valuable comments on this work.
REFERENCES


