Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany

Thorsten Berger
Chalmers | University of Gothenburg
Gothenburg, Sweden

ABSTRACT
Many software systems need to exist in multiple variants to account for varying requirements. Organizations usually implement these variants using clone & own—a strategy that is cheap and readily available, but does not scale with the number of variants, and then requires re-engineering the cloned variants into a configurable platform, typically known as a software product line. Since such a re-engineering can be risky and costly, ideally, organizations could rely on decision support and empirical data to assess the costs, activities, and benefits associated with this re-engineering. Despite decades of product-line research, we are still missing detailed data on such a re-engineering endeavor. Obtaining such details is challenging, since industrial settings are almost infeasible. We address this gap with a study on migrating two cases of cloned variants of open-source Android and Java games. Two teams of student developers re-engineered the clones into software product lines, logging activities and efforts based on a measurement strategy we conceived. Our results indicate that the applied process and its activities are relatively similar among both cases, but the actual efforts of each activity and the problems faced differed substantially. Interestingly, none of the existing re-engineering tools proved particularly useful. We hope that our results support researchers improving re-engineering techniques, and practitioners trying to understand the challenges, costs, and activities involved in re-engineering a software product line.

CCS CONCEPTS
- Software and its engineering → Software product lines: Software reverse engineering; Maintaining software.

KEYWORDS
Software product lines, empirical study, re-engineering, clone & own

1 INTRODUCTION
Software product-line engineering provides concepts, techniques, and processes to systematically manage and reuse a set of artifacts (e.g., source code, models or documentation) based on an integrated, configurable platform [3, 10, 43]. To implement such a platform, an organization needs to adopt concepts, such as features, feature models, and configurators. The platform allows to derive individual variants in an automated process by selecting the desired features. Product-line engineering promises several benefits, such as reduced development and maintenance costs, higher software quality, and faster time to market [23, 43, 50].

However, setting up an integrated platform requires high initial investments that most organizations are reluctant to spend [8, 24, 30]. For this reason, organizations often copy an existing system and customize it to new requirements, typically called clone & own [13, 48]. While clone & own is a simple strategy to reuse and is initially cheaper, maintaining an increasing number of cloned variants does not scale [13, 33, 42, 47, 53]. In particular, propagating new features, updates, and bug fixes increases the costs with each new variant, as it does not only require the developers to synchronize changes, but also to locate the right target systems, adapt the changes to those systems, and to assure the quality to avoid unwanted side effects. A software product line avoids such problems with its integrated platform. So, many organizations re-engineer (a.k.a. extract [25]) a software product line from their cloned systems at some point—the most common adoption scenario in practice [5, 14, 20, 53].

The overall process of product-line engineering, which classifies into domain engineering and application engineering, is reasonably well understood, as well as the corresponding activities (e.g., domain analysis and feature modeling) [3, 10]. Similarly, numerous case studies report experiences of re-engineering a software product line from cloned systems [20, 32, 33, 36, 54]. However, little attention has been paid to systematically collecting and analyzing the characteristics of the activities employed, especially their costs. As such, organizations are challenged when planning and deciding on the concrete activities to perform and prioritize, for example, because they reduce the costs for following activities. This missing knowledge results in uncertainty before re-engineering clones into a software product line, and may prevent organizations from initiating the re-engineering at all.

In this paper, we tackle this problem and report a multi-case study [52] based in action research (interacting and adapting during the case studies) [15] on re-engineering variants of the Apo-Games [31] dataset into software product lines. Our two cases each comprise two distinct sets of the games (i.e., Java and Android) from which two separate teams of two student developers re-engineered individual software product lines. The cases also differed in the variability mechanisms: annotation-based with the
Antenna preprocessor and composition-based with feature-oriented programming [44]. During a time span of six months, we defined a documentation strategy and both teams tracked the activities they performed during the re-engineering. We compare these activities and their characteristics, analyzing commonalities and differences to understand the resulting costs. So, we provide insights into the re-engineering based on systematically collected data, complementing existing research and experience reports, which typically, abstractly describe lessons learned. In detail, we contribute:

- A documentation strategy for analyzing re-engineering projects.
- Details and experiences about the actual re-engineering.
- Details on the data we measured during the re-engineering.
- A comparison of both cases to consolidate activities and costs.

Our results indicate that the activities performed during the re-engineering of a software product line do not substantially differ, regardless of the variability mechanism used. Considering the efforts, we found substantial differences, but these are mostly due to the training that was necessary for each team to familiarize with product-line engineering. Still, our measurements indicate that the majority of effort is invested in the actual migration and quality assurance. Our findings can help researchers to improve existing techniques and scope future research, while practitioners can better assess what activities to plan for, what these can cost, and perhaps even whether a re-engineering is beneficial at all.

2 BACKGROUND

Product-Line Engineering. A software product line enables developers to reuse software based on a configurable platform [3, 10]. This platform allows to derive similar, yet customized, variants based on customer-specified requirements. These requirements are matched against the feature model to derive a valid configuration that defines the desired variant. This configuration is used to automatically generate the customized product.

Cost Models for Software Product Lines. We are interested in understanding the activities and costs of re-engineering a software product line from a set of cloned systems. Several researchers have proposed cost models that shall help to estimate the efforts of adopting a software product line. Unfortunately, as Ali et al. [2] survey, these cost models differ heavily in their granularity and applicability. More precisely, there have been almost no empirical evaluations and most cost models build on experiences with single projects, neither considering any data-driven inputs, nor the re-engineering from clones [26, 30]. So, there is a severe need to understand this process and support developers in estimating its costs.

For example, consider SIMPLE [9] and COPLIMO [7], which may be the best known cost models for product-line engineering. SIMPLE defines five cost functions, such as organizational costs, without specifying any cost factors. This allows for a high-level understanding of the costs, but is intentionally limited to this purpose. In contrast, COPLIMO defines fine-grained cost factors (e.g., lines of code, ratio of design modifications) that consider commonalities and differences between variants. A reason for such discrepancies is missing data on re-engineering activities and their costs, which are needed to derive an empirically based cost model.

3 DESIGN OF THE STUDY

In this section, we report the details of our multi-case study design.

3.1 Research Objectives

The goal of our study was to analyze the activities and costs that are associated with re-engineering a software product line from a set of cloned systems. To this end, we defined three research objectives:

- RO1 Document and compare activities that were performed for the product-line re-engineering.
- RO2 Understand the costs and characteristics of the activities identified during the re-engineering.
- RO3 Analyze the costs of product-line re-engineering, comparing especially the use of annotation-based and composition-based variability mechanisms.

To address these objectives, we conducted a multi-case study [52] combined with action research [15] comprising two cases, as we depict in Figure 1. During the study, three teams have been involved in the conduct (cf. Table 2). Two developer teams (two Master students each, all working in industry) were independently re-engineering software product lines from the cloned variants, and cooperated only for the initial design and while comparing the final results. The coordination team (the authors) advised the other teams in designing the study and measurements, providing feedback on intermediate outcomes, discussing problems, and comparing the results.

3.2 Conduct Literature Survey

In the first phase, both developer teams conducted independent literature surveys on product-line re-engineering, starting from literature and course material provided by the coordination team. During these surveys, the teams familiarized with variability mechanisms, reverse-engineering techniques, cost models, variability modeling, and corresponding tools. The goal of this step was to achieve an understanding of relevant activities and concepts. As a result, both teams obtained the knowledge needed to perform the case studies and to define the remaining methodology.

3.3 Design Measuring and Logging Concept

To gain insights into the efforts of re-engineering software variants into an integrated platform, we needed to measure and track the actual process. For this purpose, both developer teams started to identify what information to track based on their literature survey. The metrics we selected are mainly based on established cost models for product-line engineering, namely SIMPLE [9], COPLIMO [7], and InCoME [41]. Both teams collected similar data that is relevant to describe the activities they would perform, and argued in favor of using a common template and a version control system.
We used these nine types to classify activities, especially those that we discussed the findings of the literature surveys and applied an works on the re-engineering of software product lines to identify (i.e., preprocessor, feature-oriented programming).

To allow us to derive meaningful and comparable insights, we synthesized the results of both groups from the previous step, ensuring • comparable and sufficient knowledge of both groups on product-line re-engineering; • a unified measuring and logging concept; and • that activities were measured in the same granularity. For this purpose, we all met in multiple sessions to unify the methodology. So, even if some steps of the actual re-engineering strategy differ (e.g., due to different variability mechanisms), we can still identify and compare common activities.

Identified Activities for Re-Engineering. We analyzed existing works on the re-engineering of software product lines to identify activities that were employed [4, 28, 36, 49]. These activities were on different levels of granularity (e.g., domain engineering vs. feature modeling) and used various terminologies (e.g., feature modeling vs. variability modeling). To unify the activities and adjust granularity, we discussed the findings of the literature surveys and applied an open-card sorting like method. Finally, we defined nine types of activities (AT) that are commonly mentioned in the literature:

AT\textsubscript{1} Software-product-line training summarizes activities that the developer teams performed to get used to concepts and tools for product-line engineering (including the literature surveys).

AT\textsubscript{2} Domain analysis describes activities that the teams employed to understand the domain of our subject systems, namely Java and Android games.

AT\textsubscript{3} Preparatory analysis includes activities that the teams employed to improve the quality of the legacy systems or to gain data for the following steps.

AT\textsubscript{4} Feature identification is an activity during which the teams aimed to identify what features exist within a legacy system.

AT\textsubscript{5} Architecture identification comprises activities that the teams employed to understand the architecture of the legacy systems and define a new one for their software product lines.

AT\textsubscript{6} Feature location is an activity on which developers spent most of their time, because they still have to manually locate the source code that corresponds to a feature [6, 32, 51].

AT\textsubscript{7} Feature modeling is a core activity in product-line engineering and defines the commonalities and variability of the software product line in terms of its features [11, 39].

AT\textsubscript{8} Transformation summarizes activities connected to the actual re-engineering and implementation of a software product line.

AT\textsubscript{9} Quality assurance includes activities that are connected to testing and fixing the resulting software product line.

We used these nine types to classify activities, especially those that may differ between the two teams, due to the cases’ characteristics. For example, the teams used subject systems from different platforms (i.e., Android, Java) and individual variability mechanisms (i.e., preprocessor, feature-oriented programming).

3.4 Synthesize Methodology

To allow us to derive meaningful and comparable insights, we synthesized the results of both groups from the previous step, ensuring • comparable and sufficient knowledge of both groups on product-line re-engineering;

- a unified measuring and logging concept; and

- that activities were measured in the same granularity.

For this purpose, we all met in multiple sessions to unify the methodology. So, even if some steps of the actual re-engineering strategy differ (e.g., due to different variability mechanisms), we can still identify and compare common activities.

Identified Activities for Re-Engineering. We analyzed existing works on the re-engineering of software product lines to identify activities that were employed [4, 28, 36, 49]. These activities were on different levels of granularity (e.g., domain engineering vs. feature modeling) and used various terminologies (e.g., feature modeling vs. variability modeling). To unify the activities and adjust granularity, we discussed the findings of the literature surveys and applied an open-card sorting like method. Finally, we defined nine types of activities (AT) that are commonly mentioned in the literature:

AT\textsubscript{1} Software-product-line training summarizes activities that the developer teams performed to get used to concepts and tools for product-line engineering (including the literature surveys).

AT\textsubscript{2} Domain analysis describes activities that the teams employed to understand the domain of our subject systems, namely Java and Android games.

AT\textsubscript{3} Preparatory analysis includes activities that the teams employed to improve the quality of the legacy systems or to gain data for the following steps.

AT\textsubscript{4} Feature identification is an activity during which the teams aimed to identify what features exist within a legacy system.

AT\textsubscript{5} Architecture identification comprises activities that the teams employed to understand the architecture of the legacy systems and define a new one for their software product lines.

AT\textsubscript{6} Feature location is an activity on which developers spent most of their time, because they still have to manually locate the source code that corresponds to a feature [6, 32, 51].

AT\textsubscript{7} Feature modeling is a core activity in product-line engineering and defines the commonalities and variability of the software product line in terms of its features [11, 39].

AT\textsubscript{8} Transformation summarizes activities connected to the actual re-engineering and implementation of a software product line.

AT\textsubscript{9} Quality assurance includes activities that are connected to testing and fixing the resulting software product line.

We used these nine types to classify activities, especially those that may differ between the two teams, due to the cases’ characteristics. For example, the teams used subject systems from different platforms (i.e., Android, Java) and individual variability mechanisms (i.e., preprocessor, feature-oriented programming).
Apo-Games have been developed by a single developer based on games. After the case study was conducted, each team separately analyzed their data. We removed unused code to facilitate analyzing the results.

### 3.5 Perform Case Study

After we synthesized the methodology that both developer teams should follow, they conducted the actual multi-case study. We provide an overview of our cases’ characteristics in Table 2.

#### Subject Systems

For our subject systems, we used the Apo-Games dataset [31], which provides 20 Java and five Android games. The Apo-Games have been developed by a single developer based on clone & own in a real-world setting, meaning that they have been published (e.g., in the Google Play Store). By now, they have become an established dataset for evaluating techniques for reverse engineering and product-line re-engineering. Several case studies have been concerned, for example, with recovering feature models [38] and architectures [34] from the games.

For each team, we selected a set of five games from the overall dataset. We aimed to incorporate the same number of systems with similar sizes to make the overall processes comparable. To force both developer teams to work on their own and to compare different variability mechanisms, we selected games that have been implemented with different languages (i.e., Java vs. Android) and employed two different variability mechanisms. Team 1 used a composition-based mechanism, namely feature-oriented programming [44]. In contrast, team 2 re-engineered the games into an annotation-based platform, using the Antenna preprocessor.

#### Multi-Case Study Design

As aforementioned, all teams synchronized on the initial design of the logging concept and while comparing the results. However, the actual cases were conducted completely independent, meaning that our design reflects a multi-case study. We did allow the developer teams to use any tools that were available, as long as they documented and described the usage. Not surprisingly, one of the main used tools has been FeatureIDE [37], which provides capabilities for feature modeling, configuring, implementing, and testing software product lines. Still, both teams also relied on other tools, for example, IntelliJ IDEA for running the Android games and several plug-ins for code analysis.

During the whole process, each team communicated their progress once a week in a short Scrum-like meeting to the coordination team. In such meetings, we evaluated the progress and identified problems. We also conducted additional bi-weekly meetings during which the teams presented their results and progress in detail.

#### 3.6 Log Activities

In parallel to conducting the case studies, both developer teams documented their activities as aforementioned. For this purpose, each team used a separate Git repository in which they committed their activities. The teams updated the templates whenever they had to return to or repeat an activity, for example, the transformation and quality assurance.

#### 3.7 Summarize Data

After the case study was conducted, each team separately analyzed the resulting data and reported their lessons learned, based on which we reported technical insights and our artifacts in previous works [1, 12]. Thus, the first practical experiences we obtained were unbiased from any comparison between the two teams. As the templates were completed, this step involved mainly checking the version-control history to verify the data. Finally, both teams summarized their data by providing overview tables on the single activities, their activity types, descriptions, and efforts.

### Table 1: The final template for logging re-engineering activities with a concrete example from our dataset.

<table>
<thead>
<tr>
<th>Information</th>
<th>Activity type: Preparatory analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activity: Removing unused code</td>
<td></td>
</tr>
<tr>
<td>Activity ID: A10</td>
<td></td>
</tr>
<tr>
<td>Variant IDs: V2, V3, V4, V5</td>
<td></td>
</tr>
<tr>
<td>Start date: 2019-03-11</td>
<td></td>
</tr>
<tr>
<td>End date: 2019-03-12</td>
<td></td>
</tr>
<tr>
<td>Description: Identifying code that is not used in the variants. We removed unused code to facilitate analyzing the variants.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Data</th>
<th>Total hours: 12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commits: 7</td>
<td></td>
</tr>
<tr>
<td>LOC added: 0</td>
<td></td>
</tr>
<tr>
<td>LOC removed: 11,670</td>
<td></td>
</tr>
<tr>
<td>LOC modified: 0</td>
<td></td>
</tr>
<tr>
<td>Files added: 0</td>
<td></td>
</tr>
<tr>
<td>Files removed: 78</td>
<td></td>
</tr>
<tr>
<td>Files modified: 133</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Artifacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: Source code</td>
</tr>
<tr>
<td>Output: Refactored source code</td>
</tr>
<tr>
<td>Tools: Eclipse, UCDetector, IntelliJ</td>
</tr>
</tbody>
</table>

**Activity Description**

Complexity: The activity is of relatively low complexity, thanks to the available tools.

Importance: This activity is very important because failure to detect unused code means the developer will spend time transforming source code that is never used.

Dependencies: N/A

**Logging in the Version Control System.** We performed the whole re-engineering process within version control systems, namely Git repositories. To track activities and analyze their development, the teams documented the corresponding commit identifiers in our template. In addition, they decided to further track activities in commit messages by specifying activities in these.

#### Table 2: Overview of team distribution and subject systems.

<table>
<thead>
<tr>
<th>Teams</th>
<th>Members</th>
<th>PL</th>
<th>VM</th>
<th>Games</th>
<th>Year</th>
<th>SLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev. 1</td>
<td>2 developers</td>
<td>Java</td>
<td>FOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApoCheating</td>
<td>2006</td>
<td>3,960</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApoStarz</td>
<td>2008</td>
<td>6,454</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apolcarus</td>
<td>2011</td>
<td>5,851</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApoNotSoSimple</td>
<td>2011</td>
<td>7,558</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ApoSnake</td>
<td>2012</td>
<td>6,557</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dev. 2</th>
<th>2 developers</th>
<th>Android</th>
<th>Antenna</th>
</tr>
</thead>
<tbody>
<tr>
<td>ApoClock</td>
<td>2012</td>
<td>3,615</td>
<td></td>
</tr>
<tr>
<td>ApoDier</td>
<td>2012</td>
<td>2,523</td>
<td></td>
</tr>
<tr>
<td>ApoMono</td>
<td>2013</td>
<td>2,956</td>
<td></td>
</tr>
<tr>
<td>MyTreasure</td>
<td>2013</td>
<td>6,487</td>
<td></td>
</tr>
<tr>
<td>Dev. 3</td>
<td>2 developers</td>
<td>Java</td>
<td>FOP</td>
</tr>
<tr>
<td>ApoStarz</td>
<td>2011</td>
<td>5,851</td>
<td></td>
</tr>
<tr>
<td>ApoSnake</td>
<td>2012</td>
<td>6,557</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Coord. Authors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dev. 1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Programming Language; Variability Mechanism; Developer; Coordination</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java; FOP; 2 developers; N/A</td>
</tr>
<tr>
<td>Android; Antenna; 2 developers; N/A</td>
</tr>
<tr>
<td>Java; FOP; 2 developers; N/A</td>
</tr>
</tbody>
</table>

| MyTreasure | 2013 | 5,360 |

The final template for logging re-engineering activities with a concrete example from our dataset.
Table 3: Activities each team documented.

<table>
<thead>
<tr>
<th>ID</th>
<th>Activity</th>
<th>Types</th>
<th>Dev. Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>A01</td>
<td>Research on software product lines</td>
<td>AT₁</td>
<td>✔ ✔</td>
</tr>
<tr>
<td>A02</td>
<td>Research on tools</td>
<td>AT₂</td>
<td>✔ ✔</td>
</tr>
<tr>
<td>A03</td>
<td>Running and testing games</td>
<td>AT₃, AT₄</td>
<td>✔ ✔</td>
</tr>
<tr>
<td>A04</td>
<td>Translating comments to English</td>
<td>AT₅</td>
<td>✔ ✔</td>
</tr>
<tr>
<td>A05</td>
<td>Pairwise comparison of variants</td>
<td>AT₆</td>
<td>✔ ✔</td>
</tr>
<tr>
<td>A06</td>
<td>Removing unused code</td>
<td>AT₇</td>
<td>✔ ✔</td>
</tr>
<tr>
<td>A07</td>
<td>Reverse engineer class diagrams</td>
<td>AT₈, AT₉, AT₁₀</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>A08</td>
<td>Reviewing source code</td>
<td>AT₈, AT₁₀, AT₁₂</td>
<td>✔ ✔ ✔</td>
</tr>
<tr>
<td>A09</td>
<td>Create feature model</td>
<td>AT₇</td>
<td>✔ ✔</td>
</tr>
<tr>
<td>A10</td>
<td>Transforming source code to features</td>
<td>AT₈, AT₉</td>
<td>✔ ✔</td>
</tr>
</tbody>
</table>

3.8 Compare Results

Based on the summaries provided by both teams, each team then individually compared the results to each other—described in two independent theses. Afterwards, the coordination team performed an analysis and comparison of the data on its own. For this purpose, we first compared the results without considering the summaries provided by the developer teams. Then, we merged all three analyses to obtain a single representation and discussed our insights.

4 RESULTS & DISCUSSION

In this section, we detail our research objectives. For each research objective, we first present the results of our case study and then discuss their implications.

4.1 RO₁: Activities Performed

Results. We provide an overview of all activities that the developer teams performed in their respective case in Table 3. After comparing and consolidating the activities, we identified ten that we could clearly distinguish from each other. As we can see in Table 3, while all activity types appear at least once, we could not clearly separate all of them, due to their interconnections. Namely, both teams performed domain analysis (AT₂) in parallel to several activities. For instance, familiarizing with the games and their domain immediately leads to the identification of features (AT₄). Similarly, locating features (AT₆) during the code review also led to newly identified features (AT₈). Moreover, both teams employed continuous integration during the transformation process and tried to test the extended software product line after each change. Consequently, it is not possible to make a clear separation between the activity types of transformation (AT₃) and quality assurance (AT₉).

We can see that both teams employed each activity type at least once. However, developer team 1 (FOP) performed two additional preparatory analyses: translating comments to English (A04) and removing unused code (A06). They employed these activities to support their program comprehension and to speed up the transformation. This indicates that, as intended, each team conducted an individual case, and thus the details on their activities differ. In the following, we briefly summarize the activities.

Research on Software Product Lines. During this activity, both developer teams familiarized with the general concepts of product-line engineering. This way, we aimed to ensure a comparable knowledge base and that each team could successfully conduct its case. As this activity is also needed in an industrial context, we decided to fully document the time that was spent on it.

Research on Tools. Both teams also had to familiarize with the tools they wanted to use. Mostly, the teams were concerned with getting used to FeatureIDE and, for the Antenna group that used the Android games, IntelliJ Idea. Still, both teams used—or aimed to use—additional tools and plug-ins. We measured all efforts of setting up the tools, fixing errors in them, and training to use them.

Running and Testing Games. For this activity, each team ran and played their games to see how they work and behave. The goal was to understand the domain and identify user-visible features to obtain an initial set of features that could be compared among the games. Overall, this process resembles a top-down analysis, as both teams started with playing the games instead of reviewing code.

Translating Comments to English. One team decided to translate the available comments from German to English. The goal was to better understand the games’ behavior later on by relying on the original developer’s descriptions. This step assumed that the comments were maintained during development to be helpful [17, 40].

Pairwise Comparison of Variants. To re-engineer cloned variants, pairwise comparisons of the source code, is an established technique to automatically locate code that is common and variable. However, this technique has some limitations, mainly that identified clones do not represent actual features and that this technique does not scale [6, 14, 32]. For these reasons, both teams used the results of this step solely to have starting points for other activities, and to obtain a better understanding of the games’ code.

Removing Unused Code. The games that developer team 1 investigated are a bit larger in size compared to the Android games of team 2. While they seemed to comprise more common code, they also highlighted one problem of clone & own: unused code. The team identified this issue during the previous activity and decided to use an Eclipse plug-in (UCDetector)1 to remove code that was never executed, reducing the code size by almost 40%.

Reverse-Engineer Class Diagrams. To better understand the legacy systems’ structure and guide the development of the software product line, both developer teams decided to reverse engineer class diagrams (using Visual Paradigm,² and IntelliJ Idea). They automatically reverse-engineered the class diagrams and manually merged them. The results indicated a high degree of reuse on class level in each case, for example, by providing generic classes for common game elements, such as the player and enemies.

Reviewing Source Code. After the teams analyzed the games’ domain and structure, they performed a code review (bottom-up analysis). During this activity, they identified additional features, located the source code belonging to each feature, and documented common and variable parts for the transformation. So, we cannot clearly distinguish between feature identification and location.

Create Feature Model. In the previous steps, each developer team built their knowledge about the cloned variants and the implemented features. They used this knowledge to design a feature model for each of the two resulting software product lines. The models helped to define the dependencies between features, guided the transformations for which the teams focused on establishing a testable code base first, and was mandatory for developer team 1 to use feature-oriented programming.

1https://marketplace.eclipse.org/content/unnecessary-code-detector
2https://marketplace.eclipse.org/content/visual-paradigm-eclipse
Transforming Source Code to Features. While most of the previous activities were similar for both teams, the actual re-engineering and testing were quite different, due to the variability mechanisms used. Developer team 1 had to largely change the code architecture and unify the code base to properly introduce feature-oriented programming. Due to time constraints, they could not re-engineer all five legacy systems, but had to stop after three. In contrast, developer team 2 performed pair-wise merges of variants, adding annotations to the variable parts, and only minor structural changes.

Discussion. Despite the different sets of legacy systems, programming languages, tools, and variability mechanisms, both developer teams employed similar activities. This result indicates that the activities and the types that we reported can be considered as generalized abstractions of more fine-grained activities. So, our descriptions can be seen as a general guide on what activities to plan for during product-line re-engineering. Developer team 1 pointed out (i.e., in their thesis) that a more fine-grained analysis of activities (e.g., refactor similar methods into a generic method) could provide more detailed insights into the activities’ characteristics. However, both teams also highlighted that our abstraction level of activities allowed to compare the results. So, we argue that we provide a comprehensible and comparable set of activities and their types for understanding product-line re-engineering.

Most academic publications on product-line adoption, and on re-engineering in particular, report waterfall-like processes. For example, domain and application engineering are often considered strictly separated for the adoption, and other papers clearly distinguish between analysis, detection, and transformation. During our case study, both teams switched regularly between various activities, because they identified new features (AT6) or new code belonging to a feature (AT7) during the code review. An iterative process seems more reasonable, as, for example, building a complete feature model from the beginning is hardly possible. Consequently, constant updates and iterations are necessary, especially if the developers have to familiarize with the systems under consideration.

Insights on RO1:
- Our list of activity types allows to analyze product-line re-engineering, providing a comprehensible abstraction that is independent of specific techniques.
- The typical waterfall-like re-engineering process that is often reported in research seems unreasonable.

Table 4: Overview of the efforts tracked in person-hours.

<table>
<thead>
<tr>
<th>ID</th>
<th>Activity Type</th>
<th>Dev. Team 1</th>
<th>Dev. Team 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AT1</td>
<td>Software-product-line training</td>
<td>16.00</td>
<td>18.00</td>
</tr>
<tr>
<td>AT2</td>
<td>Domain analysis</td>
<td>18.00</td>
<td>22.00</td>
</tr>
<tr>
<td>AT3</td>
<td>Preparatory analysis</td>
<td>49.25</td>
<td>50.00</td>
</tr>
<tr>
<td>AT4</td>
<td>Feature identification</td>
<td>22.25</td>
<td>7.00</td>
</tr>
<tr>
<td>AT5</td>
<td>Architecture identification</td>
<td>2.00</td>
<td>7.00</td>
</tr>
<tr>
<td>AT6</td>
<td>Feature location</td>
<td>50.00</td>
<td>20.00</td>
</tr>
<tr>
<td>AT7</td>
<td>Feature modeling</td>
<td>7.00</td>
<td>10.00</td>
</tr>
<tr>
<td>AT8</td>
<td>Transformation</td>
<td>103.50</td>
<td>180.00</td>
</tr>
<tr>
<td>AT9</td>
<td>Quality assurance</td>
<td>103.50</td>
<td>60.00</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>371.50</td>
<td>496.00</td>
</tr>
</tbody>
</table>

4.2 RO2: Costs of Activities

Results. We experienced that precisely tracking the effort for each activity can be challenging, due to the aforementioned interconnections and iterations. To address this issue, we tracked the version-control history to verify and distribute the effort estimates. Still, we consider especially transformation (AT4) and quality assurance (AT9) together, as both teams performed continuous integration. For this reason, the teams incrementally added new features and tested them, challenging a clear separation of efforts (i.e., team 1 specified only development effort, while stating that 50% is quality assurance). In Table 4, we summarize the efforts that each team documented throughout its case.

Developer Team 1. The team that re-engineering the cloned variants into a feature-oriented software product line spent more than half of its time on the actual transformation and quality assurance (27.86%, each). Most of the remaining effort went into preparatory analyses, such as removing unused code and translating comments, as well as feature location, which is known to be an expensive activity. Other activities required considerably less effort, resulting in accumulated efforts of 371.5 person hours in total.

Developer Team 2. The second team did perform extensive product-line training (18.15%) and domain analysis (16.53%). We experienced that this was mainly due to the nature of the subject games, namely that they have been implemented in Android. Available tools for product-line engineering do not sufficiently support Android, if at all, which required the team to spend a lot of time on fixing and combining tools (e.g., FeatureIDE and IntelliJ IDEA). Nonetheless, most of the effort went into the actual transformation and quality assurance (48.39%).

Discussion. As mentioned above, strictly distinguishing efforts for each activity is hardly possible. Most activities are interconnected and the developers switch between them. So, the activity of reviewing source code may not initiate feature location (AT6), but the developers will immediately search for additional features (AT4) and refine their domain knowledge (AT2). We argue that this indicates that existing tools should not solely focus on a single activity, but combine them to support developers in their natural analysis strategies. Moreover, while we used existing cost models to derive our logging template, their granularity seems to be inappropriate. For instance, SIMPLE simply defines four cost functions that summarize all activities, while COPLIMO defines factors (e.g., ratio of code modified) without considering activities and tooling.

During the case study, both teams experienced that certain system and project characteristics can have unexpectedly high efforts. For example, the missing tool support for Android-based software product lines drastically increased the costs for developer team 2. Similarly, developer team 1 reported that they have been heavily relying on the code comments and that the removal of unused code considerably facilitated their activities. Finally, both teams reported that missing knowledge about product-line engineering and the systems in particular was a challenge. These issues clearly highlight the impact of system characteristics and challenge the community to provide new or improved tools that consider these characteristics to support developers.

To further strengthen this point, we can consider the activities for which we had particularly well working tools: architecture
We already described that the team had to investigate and connect was localizing bugs after integrating new features, which was a considerable effort into this single activity, while developer team 2

The main issue that the team faced was the complexity of directly different analysis strategies, during our discussion we found that this issue is more connected to the actual variability mechanism. For feature-oriented programming (developer team 1), we need to first identify and locate all features in the code to refactor them into meaningful and composable modules. In contrast, annotations (developer team 2) allow to add variability in an ad hoc manner. Not all code of a feature must be located immediately, but it can be stepwise refined and extended with code from other variants. As a result, feature location requires less effort, which is an interesting insight indicating that the efforts for feature location for re-engineering a software product line depend on the variability mechanism that will be introduced. However, the actual transformation may comprise some of this effort, too.

Considering the other activities, the total numbers of person hours spent and also their ratios are comparable among both cases. This is hardly surprising, as most of them are independent of project specifics and are quite similar. For example, both teams performed comparable steps during the feature identification (AT3) and architecture recovery (AT5). Overall, we can see that the main differences between both cases are caused by preparation (e.g., training, domain analysis, code analysis) and feature location. Besides the transformation and quality assurance, these are also the activities that contribute to most efforts.

Overall, while the efforts for both cases are comparable, we argue that re-engineering a software product line based on an annotation-based variability mechanism seems more suitable. This mechanism can be applied ad hoc and requires less knowledge in advance. Both teams made similar statements, experiencing that there is a trade-off between the variability mechanisms: Feature-oriented programming is complex, requires learning, and seems too expensive for smaller, scattered features, such as in the Apo-Games. A preprocessor is simpler to use and annotating code does not require to restructure fine-grained and scattered features into modules. However, developer team 2 also argued that the readability and structure of the code becomes poor, suggesting that a later restructuring into separate classes or changing towards composition could help to address these problems. In summary, an annotation-based variability mechanism seems easier to introduce with less effort, but the developers should still consider the granularity of decomposition of the resulting software product line.

4.3 RO3: Comparing the Cases

Results. In Table 4, we can see the efforts of both cases compared to each other. Overall, developer team 2 spent almost 125 additional hours for their re-engineering, especially in the beginning while researching product-line engineering and during domain analysis. We already described that the team had to investigate and connect multiple tools to implement their software product line in Android. Moreover, we found that developer team 1 did not completely track the training phase, because they did already research the topic before we designed the study. For a more comparable overview, we provide the ratios of effort that was spent on each activity.

Discussion. The efforts for the actual transformation and quality assurance are similar for the composition-based and annotation-based cases (207 to 240 person hours and 55.72% to 48.39%, respectively). However, we remark that due to time restrictions, development team 1 could only migrate three of the intended five variants. The main issue that the team faced was the complexity of directly migrating towards a composition-based variability mechanism. In contrast to annotations, composable modules require larger-scale refactoring, merges, and adaptations of the implementation to enable a configurable platform. A particular problem the team faced was localizing bugs after integrating new features, which was a complex task due to the separation of previously connected code [27, 29]. Moreover, composition-based mechanisms are still not established in practice and require more knowledge, which was also highlighted by the developer teams as one reason for their struggles.

Besides the aforementioned training and domain analysis activities, feature location is also among the activities with considerable differences. Apparently, developer team 1 put 50 person hours (13.46%) of effort into this single activity, while developer team 2 spent only seven person hours (1.41%). While this may be due to different analysis strategies, during our discussion we found that this issue is more connected to the actual variability mechanism. For feature-oriented programming (developer team 1), we need to first identify and locate all features in the code to refactor them

<table>
<thead>
<tr>
<th>Insights on RO3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Re-engineering a software product line into a composition-based variability mechanism is considerably more complex than using an annotation-based variability mechanism.</td>
</tr>
<tr>
<td>• Annotation-based variability mechanisms require less focus on feature location to initiate an integrated platform.</td>
</tr>
<tr>
<td>• Of all other activities, the actual transformation and preparations (e.g., training, domain analysis, code analyses) require the most effort.</td>
</tr>
<tr>
<td>• The efforts of most other activities are comparable for both variability mechanisms.</td>
</tr>
</tbody>
</table>

5 THREATS TO VALIDITY

Internal Validity. The most severe threat to the internal validity is that we could not cooperate with the original developer of the Apo-Games during our experiment. As a result, we performed all analyses and transformations ourselves. This can threaten the results we obtained and the original developer would potentially require less effort to perform the same tasks, due to their knowledge. Still, as both teams started under similar conditions, the results
remain comparable and valuable, showing how developers without previous knowledge about a system could perform.

Another threat to our study is the data collection. There are numerous cost factors that depend also on human factors and project specifics, making it impossible to understand all costs. Similarly, activities are tangled and it is hardly possible to precisely track all costs. We aimed to mitigate these threats by considering existing works beforehand, defining a logging template, and verifying our results with the version-control history. Nonetheless, this threat remains and can bias our results.

**External Validity.** A threat to the external validity of our study is that we used only five subject systems of the Apo-Games for each of our two cases. While this prevents us from over-generalizing our results to the real-world, there exist only few publicly available datasets that have been developed based on clone & own. As such, the Apo-Games have already established themselves as a suitable baseline for reverse and re-engineering research (cf. Section 6). We argue that, while we cannot fully overcome this threat, we could still derive valuable insights in this regard.

Another threat to the external validity is the tooling we relied on during the case study. We aimed to use reliable tools, such as FeatureIDE and InelliJ IDEA, but there have been some problems, especially with the Android dataset. However, we experimented with research tools that have been particularly designed to support the re-engineering of software product lines, but these proved less helpful—if we could get them started at all. As a consequence, there may exist some tools that would have severely simplified our case study, but we could not use them.

6 RELATED WORK

The Apo-Games have been proposed to serve as baseline for reverse and re-engineering software product lines from cloned systems [31]. As such real-world datasets are rare, the systems have already gained a lot of attention in the research community. For example, Lima et al. [34, 35] proposed a technique to automatically recover the architecture of cloned systems and evaluated it on the Apo-Games, which is related to our architecture identification activity. Similarly, Mendonça et al. [38] derived a technique to automatically create Pareto-optimal feature models that can represent the existing variants. Both techniques can be helpful when re-engineering a software product line and can facilitate the activities we defined. However, we tried to avoid collecting additional information on the Apo-Games to avoid biases in our methodology and to have the same conditions for both developer teams. Finally, we reported own studies on the Apo-Games [1, 12, 32], including independent descriptions of the artifacts we created during the case study in this paper. There are significant differences, as we did previously focus solely on the analysis of the systems, including feature identification and modeling. This paper considerably differs, as we focus on all activities, the measured costs, and a comparison between both cases, none of which we did report before.

The ESPLA catalog [36] collects case studies on re-engineering software product lines from cloned systems. Several case studies report experiences and lessons learned, based industrial as well as open-source systems [18–21, 54]. Still, the catalog also highlights the problem that many artifacts are not or only partly available.

Also, we are not aware of any case study that particularly focuses on activities, costs or measuring empirical data. Consider, for example, the paper of Rubin et al. [46], in which the authors report experiences of migrating three sets of industrial systems towards individual software product lines. Similar to Fenske et al. [16], the authors focus on providing automation for refactoring and merging the systems into a common platform. While this is promising, in neither case do the authors elaborate on the activities that are needed to analyze the systems or the resulting costs.

7 CONCLUSION

In this paper, we reported a multi-case study on the activities that are connected to re-engineering a software product line from cloned systems. To this end, we collected empirical data on the activities employed and their costs. We used two sets of five different games from the Apo-Games dataset and varied the programming languages as well as variability mechanisms. Overall, we found:

- We identified a common set of activities employed during both cases of our study, independently of the variability mechanism or tooling. These can serve as a baseline to better understand re-engineering processes and contradict strict waterfall-like processes. Instead, our teams had to regularly switch and iterate through activities.

- Iterations and the interconnections between activities (e.g., the results of one activity are a prerequisite for another) challenge the tracking and documentation of costs. These issues are problematic for cost estimations, as we found that some project characteristics can heavily impact costs, while developers may not take them into account. Considering this situation, we also argue that especially tool support not for single activities, but the whole re-engineering process is needed.

- We found that re-engineering cloned systems into a composition-based software product line is more complex than using annotations. The main reasons for this are refactorings and merges, as well as unfamiliarity with the concept of composition. Most other activities required similar efforts in both cases, with transformation and quality assurance contributing to most costs.

The results can help researchers and practitioners to better understand the activities and costs of re-engineering cloned variants into a software product line. Practitioners can benefit from this, as they can better plan their projects: They can more reasonably decide whether re-engineering is sensible and how to distribute resources.

For researchers, the results further help to open new research directions and refine existing concepts. In future work, we plan to extend our analysis to more studies, including industrial ones. Based on a larger corpus of data, we hope to achieve even more detailed and empirical insights on activities and costs. We plan to refine and extend our tracking concept to also automate the collection of empirical data and to control for additional variables, such as further variability mechanisms, tools, and system sizes. However, our results show that we still need to advance existing tools and provide them in a reusable form.

Acknowledgments. Supported by Vinnova Sweden (2016-02804), the Swedish Research Council Vetenskapsrådet (257822902), and the German Research Council (SA 465/49-3). We thank Jennifer Horkoff for valuable comments on this work.
REFERENCES


