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

Jacob Krüger
Otto-von-Guericke University
Magdeburg, Germany
jkrueger@ovgu.de

Thorsten Berger
Chalmers | University of Gothenburg
Gothenburg, Sweden
bergert@chalmers.se

ABSTRACT

Many software systems need to exist in multiple variants. Organizations typically develop variants using clone & own—copying and adapting systems towards new requirements. However, while clone & own is a simple and readily available strategy, it does not scale with the number of variants, and then requires a costly re-engineering of the cloned variants into a configurable software platform (a.k.a., software product line). Ideally, organizations could rely on decision models or at least on substantial empirical data to assess the costs and benefits of such a re-engineering. Unfortunately, despite decades of research on product lines and platforms, such data is scarce, not least because obtaining it from industrial re-engineering efforts is challenging. We address this gap with a study on re-engineering two cases of cloned variants of open-source Android and Java games. Student developers re-engineered the clones into software product lines, logging their activities and costs. They performed the types of activities typically associated with re-engineering, but the activities were intertwined and done iteratively. The costs were relatively similar among both cases, but the used variability mechanism had a substantial impact. Interestingly, beyond a common diffing tool, no dedicated re-engineering tool was particularly useful. We hope that our results support researchers working on re-engineering techniques and decision models, as well as practitioners trying to assess the costs and activities involved in re-engineering a software platform.

CCS CONCEPTS

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

KEYWORDS

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

ACM Reference Format:
Jacob Krüger and Thorsten Berger. 2020. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *Proceedings of the 14th International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS '20), February 5–7, 2020, Magdeburg, Germany.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377024.3377044

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
VaMoS '20, February 5–7, 2020, Magdeburg, Germany
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-7501-6/20/02…$15.00
https://doi.org/10.1145/3377024.3377044

1 INTRODUCTION

Software product-line engineering provides methods and tools to systematically manage and reuse software artifacts (e.g., source code, models or documentation) based on an integrated and configurable software platform [4, 16, 57]. To develop such a platform, an organization needs to adopt many of those methods and tools—for instance, it needs to identify and manage features [8, 41] (abstract representations of the platform's functionalities), organize these features in a feature model [30, 52], incorporate variation points in the source code [44], adopt a configurator tool [10, 59], reorganize teams [55, 65] or change processes [11]. Once established, the product-line platform allows to derive individual software variants in an automated process by selecting the desired features. Software product-line engineering promises several benefits, such as reduced development and maintenance costs, higher software quality, and faster time to market of new variants [15, 32, 57, 66].

Unfortunately, establishing a product-line platform requires high upfront investments most organizations are reluctant to spend [14, 33, 39]. Instead, organizations typically use a strategy known as clone & own [19, 63]—copying an existing variant and adapting it to new requirements. While clone & own is initially simple and cheap, it does not scale with the number of variants and then causes high maintenance overheads that challenge organizations [19, 43, 56, 62, 69]. In particular, manually propagating changes (e.g., new features, updates or bug fixes) across the cloned variants requires developers to identify the target variant and exact location to apply the change, to adapt the change for the new location, and to quality-assure the adaptation to avoid unwanted side effects. A product line avoids such problems with its integrated platform. Many organizations eventually re-engineer such a platform from their cloned variants—the most common adoption scenario in practice [9, 20, 27, 69], also known as extractive product-line adoption [34].

Re-engineering cloned variants into a product-line platform is challenging. While product-line research focused on building product lines from scratch [4, 16] and on improving already well-engineered platforms, the actual re-engineering [6] is not well understood or supported. While numerous case studies report experiences of re-engineering a software product line from cloned systems [11, 27, 42, 43, 48, 70], little attention has been paid to systematically analyzing the re-engineering process, especially with respect to the activities involved and their costs. As such, organizations are challenged when planning and prioritizing the concrete activities to perform. The lack of qualitative and quantitative empirical data results in organizations being uncertain whether and how to re-engineer clones into a product-line platform.

We present a multi-case study [68] on re-engineering two sets of cloned variants of Java and Android games into two product-line

platforms. Our subject systems are the Apo-Games [40], a curated dataset of open-source games for Java and Android environments, that were cloned over time. Our methodology was action-research-like [21]: Two teams of two student developers each conducted the re-engineering over a period of six months, supervised by the authors who continuously discussed the re-engineering process and any problems arising. To document and track the re-engineering, we conceived a measurement and logging strategy, which involved identifying the types of activities typically associated with product-line re-engineering in the literature. To obtain broader insights, we varied the implementation technique for variation points, with one team using an annotation-based variability mechanism using a preprocessor, and the other team a composition-based variability mechanism using feature-oriented programming (FOP) [5, 58]. We qualitatively and quantitatively compared the teams' documented activities and their characteristics—among others, to measure costs in terms of efforts (person-hours) spent per activity.

In summary, we contribute:

- A logging and measurement strategy for documenting re-engineering projects.
- Qualitative and quantitative empirical data on the re-engineering of our two cases.
- The resulting product-line platforms.[1]
- An online appendix referencing the relevant repositories and more detailed data.[2]

Our results show that the developer teams performed the types of activities typically associated with re-engineering, but the activities were intertwined and performed iteratively. Beyond a common diffing tool, no dedicated re-engineering tool proved particularly useful. Creating a platform using FOP as variability mechanism was more costly and difficult than creating a platform with an annotation-based mechanism. Otherwise, the costs and activities were largely similar among both teams (only the training costs were higher for one team) for the other activities (e.g., domain analysis or feature modeling). The majority of costs was, not surprisingly, still spent on transforming the source code to integrate the variants into a platform. We hope that our findings help researchers to improve existing techniques and scope future research, and practitioners to assess and plan their re-engineering efforts.

2 BACKGROUND

Software Product-Line Engineering. A product line facilitates software reuse in an application domain. It relies on establishing a configurable platform [4, 16] that allows to derive similar, yet customized, software variants. Product-line engineering comprises two main processes. Domain engineering establishes the platform. Organizations typically perform a domain analysis to identify features and define a feature model (a popular variability-modeling notation) [4, 9, 17, 52], which defines the prospective variants of the platform. Thereafter, the platform is implemented, for which organizations can choose among a range of different implementation techniques (variability mechanisms) to define variation points controlled by the respective features. Application engineering derives individual variants. Based on customer-specific requirements, individual features are selected, and a variant is derived by binding the variation points in an automated process. Any functionality that is requested, but missing in the platform, must be implemented.

Cost Models for Software Product Lines. To estimate the costs (e.g., person-hours needed) of adopting a product line, researchers have proposed a variety of cost models. These models differ heavily in the granularity of costs considered and in their applicability in practice [2]. Unfortunately, most of these models were derived from experiences with a single project only, almost no model comes with an empirical evaluation, and no model considers the re-engineering of product lines from cloned variants [2, 35, 39]. So, there is a severe need to improve our empirical understanding of the re-engineering process and its cost, to support organizations striving to adopt a product line from cloned variants.

Consider, for instance, the cost models SIMPLE [15] and COPLIMO [13], which are among the better known cost models for product-line engineering. SIMPLE defines five conceptual cost functions (e.g., costs of reuse) that are assumed to return all costs corresponding to certain activities. These functions allow to understand the costs, but are intentionally limited to this purpose—not providing details on actual parameters (i.e., values of cost factors). In contrast, COPLIMO defines fine-grained cost factors (e.g., lines of code, ratio of design modifications) that consider commonalities and differences between variants. Still, COPLIMO faces the limitations outlined above, as it does not consider re-engineering, does not build on empirical data, and lacks an evaluation. A reason for such discrepancies is missing data on re-engineering activities and their costs, which are needed to derive an empirically grounded cost model.

3 STUDY DESIGN

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

3.1 Research Objectives

The goal of our study was to establish the activities and costs associated with re-engineering a product-line platform from a set of cloned variants. We defined three research objectives:

RO$_1$ Identify and analyze necessary re-engineering activities.
RO$_2$ Measure the costs of the activities identified.
RO$_3$ Perform a cross-case analysis.

To address these objectives, we conducted a multi-case study [68] comprising two cases, as we depict in Figure 1. Our methodology is action-research-like. Action research [21] usually involves solving a real-world problem in a real-world environment (e.g., within a company) with frequent interactions and interventions of researchers. While we simulate the real-world environment, and our developers are student developers with industrial experience, we are inspired by action research to supervise the developers and continuously discuss the re-engineering process and any problems arising.

During the study, three teams were involved in the conduct (cf. Table 2). Two *developer* teams (two Master students each, all with industrial experience) indepsendently re-engineered product-line platforms from the cloned variants. They collaborated only for the design of a logging and measurement strategy, and for comparing the results. The *coordination* team (the authors) advised the other teams in designing the re-engineering methodology and

[1] We presented the platforms' characteristics in two short papers before [1, 18].
[2] https://bitbucket.org/easelab/aporeengineering

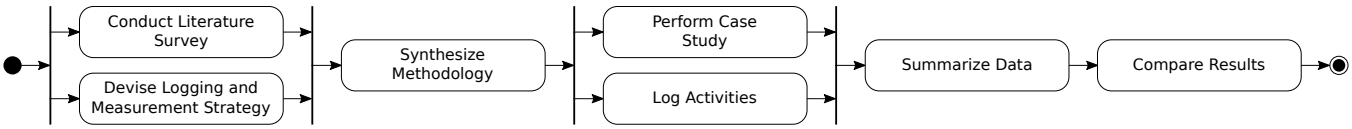

Figure 1: Overview of our study design

measurements, providing feedback on intermediate outcomes, discussing problems, and comparing the results.

3.2 Conduct Literature Survey

The goal of this first step was three-fold: familiarize with prduct-line engineering methods and tools, obtain an overview on the necessary re-engineering activities, and plan the re-engineering. For instance, both developer teams needed to be sufficiently familiar with feature modeling, their assigned variability mechanisms (annotation-based or composition-based), cost models, and tools. To this end, we provided teaching material from a product-line course and some publications on re-engineering [6, 37, 48, 64]. As a result, both teams obtained a comparable level of training, both described the envisioned activities, and provided a rough plan of the re-engineering.

3.3 Devise Logging and Measurement Strategy

To track and measure the activities performed during the re-engineering, both developer teams proposed a logging and measurement strategy (consolidated into one strategy in the next step). They proposed what kind of information to track based on the literature survey, which included cost factors taken from three established cost models for product-line engineering, namely SIMPLE [15], COPLIMO [13], and InCoME [54]. Since none of these models' cost factors are specific to re-engineering, we slightly adapted them to define metrics usable to measure re-engineering. In the remainder, we synonymously use the terms cost and effort, since we measure the former in terms of the latter (i.e., person-hours, explained shortly).

3.4 Synthesize Methodology

To assure that both developer teams' cases are comparable, we consolidated their results from the previous step, ensuring comparable and sufficient knowledge of both groups on product-line re-engineering, as well as a unified measuring and logging strategy employed by both groups.

Identified Activities for Re-Engineering. The identified activities were relatively diverse, on different levels of granularity (e.g., domain engineering versus feature modeling), and used various terminologies (e.g., feature modeling versus variability modeling). To make them comparable, we abstracted these activities into nine activity types (ATs):

- AT_1 *Software-product-line training* summarizes activities that the developer teams performed to get used to methods and tools for product-line engineering (including the literature surveys).
- AT_2 *Domain analysis* describes activities that the teams employed to understand the domain of our subject systems, namely Java and Android gamses.
- AT_3 *Preparatory analysis* includes activities that the teams employed to improve the quality of the legacy systems (e.g., remove unused code or translate comments) or to obtain data (e.g., code clones) for the following activities.
- AT_4 *Feature identification* is an activity during which the teams aimed to identify what features exist within a cloned variant.
- AT_5 *Architecture identification* comprises activities that the teams employed to understand the architecture of the legacy systems, and define a new one for their product-line platforms.
- AT_6 *Feature location* refers to locating the exact code locations (at the class, method, statement, or even sub-statement level) of a feature. Since automated feature-location techniques are typically not accurate enough [12, 42, 67], feature location was a largely manual task for our teams.
- AT_7 *Feature modeling* is a core activity in product-line engineering and models the commonality and variability of the product line in terms of its mandatory and optional features [17, 52].
- AT_8 *Transformation* summarizes activities on the actual code transformation and implementation of the product-line platform.
- AT_9 *Quality assurance* includes activities that are connected to validating the resulting platform (e.g., that the cloned variants can be derived from the platform) and general quality assurance (e.g., unit testing).

We used these nine activity types to classify and compare activities, as well as to map efforts (person-hours) to the individual activity types to measure the re-engineering costs.

Activity Sheets. To document the re-engineering activities, we created a common logging template based on the previous steps, discussing the meaning of each part of the template to achieve a common understanding among the teams. After resolving only few discrepancies, we obtained the template we illustrate in Table 1 with example data of one activity as it was tracked. The template has four sections as follows.

The section **Information** contains the activity's meta-data: activity types, short name, identifier, as well as the original variant on which the activity was done (if applicable), start date, end date, and a short description of the activity.

The section **Data** characterizes the activity by referencing the commits made and providing metrics (cf. Section 3.3), when applicable (e.g., training activities of type AT_1 did not involve code changes or commits). The metrics include the number of person-hours spent, as well as the lines of code and files changed (added, removed or modified) during the activity. The latter resemble the cost factors of modified code (CM) and portion of adapted software (AFRAC) from the cost model COPLIMO.

The section **Artifacts** documents the artifacts that were input and output of the activity (e.g., source code or feature model), as well as the tools used (e.g., the diff tool).

Table 1: Example of an activity sheet following our template.

Information	
Activity type:	Preparatory analysis
Activity:	Removing unused code
Activity ID:	A10
Variant IDs:	V2, V3, V4, V5
Start date:	2019-03-11
End date:	2019-03-12
Description:	Identifying code that is not used in the variants. We removed unused code to facilitate analyzing the variants.
Data	
Total hours:	12
Commits:	7 35351f7035e22907d30828cd82a475d6fd012d75 1397a2c35632c474e361da003d7c8027f3d659e7 ...
LOC added:	0
LOC removed:	11,670
LOC modified:	0
Files added:	0
Files removed:	78
Files modified:	133
Artifacts	
Input:	Source code
Output:	Refactored source code
Tools:	Eclipse, UCDetector, IntelliJ
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

Finally, the section **Activity Description** contains further descriptions: a summary of the complexity and the importance experienced, as well as dependencies to other activities (i.e., activity identifier and a description of the dependency).

Logging in the Version Control System. We performed the whole re-engineering within Git repositories with frequent commits. The teams documented the respective commit identifiers according to our template, and used descriptive commit messages using the documented activity names.

3.5 Perform Case Study

After we defined the re-engineering methodology that both developer teams should follow, they started re-engineering the two cases. Table 2 shows their core characteristics.

Subject Systems. We used the Apo-Games dataset [40], which offers 20 Java and five Android games. They were created by a single developer based on clone & own in a real-world setting, meaning that they are published (e.g., in the Google Play Store) and are used by end-users. Apo-Games is a popular dataset to evaluate re- and reverse-engineering techniques. For instance, researchers reverse-engineered feature models [50] and synthesized platform architectures [46] from the Apo-Games' cloned variants (the games).

Table 2: Team distribution and subject systems.

Team	Environment	VM	Games	Year	LOC
Dev. 1	Java	FOP	ApoCheating	2006	3,960
			ApoStarz	2008	6,454
			ApoIcarus	2011	5,851
			ApoNotSoSimple	2011	7,558
			ApoSnake	2012	6,557
Dev. 2	Android	Antenna	ApoClock	2012	3,615
			ApoDice	2012	2,523
			ApoSnake	2012	2,965
			ApoMono	2013	6,487
			MyTreasure	2013	5,360
Coord.	—	—	—	—	—

Variability **M**echanism; **Dev**eloper Team; **Coord**ination Team

We selected five games per developer team, aiming at the same number of variants with similar sizes to improve comparability. To assure that both teams worked independently without interactions, both teams' games were implemented with different languages (i.e., Java and Android). For broader insights, we also varied the variability mechanism used to define variation points in the prospective platform. Team 1 used a composition-based mechanism, namely feature-oriented programming [58] with the composer tool FeatureHouse [5]. Team 2 re-engineered the games into an annotation-based platform, using the preprocessor tool Antenna.[3] Both tools are integrated in FeatureIDE [49].

Multi-Case Study Design. As we explained above, all teams collaborated to design a unified logging and measurement strategy (cf. Section 3.4), as well as to compare the results (cf. Section 3.8). However, the actual re-engineering was conducted independently by each team.

We allowed the developer teams to use any tool they found useful, but they needed to document the usage. Not surprisingly, the prime tool was FeatureIDE [49], which provides capabilities for feature modeling, configuring, and implementing software product lines. Both teams also used other tools, for example, IntelliJ IDEA for running the Android games and several plug-ins for code analysis.

During the whole case study, each team communicated their process once a week in a short Scrum-like meeting to the coordination team. In these meetings, we evaluated the progress and identified problems. Within additional bi-weekly meetings, the teams presented their results and progress in detail.

3.6 Log Activities

The teams described each re-engineering activity in a separate document (activity sheet) based on our template (cf. Table 1) and updated the documents incrementally when the activity was repeated. Each team used a separate Git repository in which they committed changes and stored their activity sheets. The teams updated the sheets whenever they had to return to or repeat an activity, for example, for the transformation and quality assurance activitity.

3.7 Summarize Data

After conducting the case study, each team separately analyzed the resulting data and reported their lessons learned. For details on the

[3] http://antenna.sourceforge.net

Table 3: Activities performed per developer team.

ID	Activity	Types	Dev. Team	
			1 (FOP)	2 (PP)
A01	Research on software product lines	AT_1	✔	✔
A02	Research on tools	AT_1	✔	✔
A03	Running and testing games	AT_2; AT_4	✔	✔
A04	Translating comments to English	AT_3	✔	–
A05	Pairwise comparison of variants	AT_3	✔	✔
A06	Removing unused code	AT_3	✔	–
A07	Reverse engineer class diagrams	AT_2; AT_5	✔	✔
A08	Reviewing source code	AT_2; AT_4; AT_6	✔	✔
A09	Create feature model	AT_7	✔	✔
A10	Transforming source code to features	AT_8; AT_9	✔	✔

resulting platforms and those lessons learned, see our prior works [1, 18]. As the activity sheets 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.

3.8 Compare Results

Based on the teams' summaries, each team then individually compared the results to each other. Thereafter, we (the coordination team) analyzed and compared the data on our own (without considering the developer teams' summaries). Finally, we consolidated all three analyses into one and synthesized a set of core insights.

4 RESULTS & DISCUSSION

We now present and discuss the results for our research objectives.

4.1 RO_1: Activities Performed

Results. In Table 3, we show all activities the developer teams performed in their cases. Comparing and consolidating the activities revealed ten that we could clearly distinguish from each other.

We observe that the different types of activities we identified are intertwined. Some activities map to multiple types and vice versa. For instance, multiple activities can be seen as domain analysis (type AT_2), and multiple activity types were done in parallel. For example, familiarizing with the games (activity A03) contributes to both domain analysis (AT_2) and feature identification (AT_4). Similarly, locating features (AT_6) during the code review also led to newly identified features (AT_4). Moreover, both teams continuously integrated the changes during the transformation process and tried to test the extended product-line platform after each change. Thus, it does not appear to be feasible to clearly separate transformation (AT_8) and quality assurance activities (AT_9) during re-engineering.

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. In the following, we briefly summarize all activities performed.

During activity **A01, Research on Software Product Lines**, the teams became familiar with product-line engineering in general. This way, we ensured a comparable knowledge base and that each team could successfully conduct its case. As this activity is also needed in an industrial context, we documented the time spent on it.

In activity **A02, Research on Tools**, both teams familiarized with the tools they wanted to use. The prime concern was getting used to FeatureIDE and, for the Antenna group that used the Android games, with 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.

For activity **A03, Running and Testing Games**, 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.

Team 1 documented the activity **A04, Translating Comments to English**, since many code comments were in German. The goal was to better understand the games' behavior through the original developer's descriptions. This step assumed that the comments were maintained during development to be helpful [23, 53].

Activity **A05, Pairwise Comparison of Variants**, refers to diffing the source code of two variants using a diff tool, as has been done in case studies before [24, 27]. While the differences could be immediately wrapped by annotations as a quick way of integrating two variants, further effort is necessary to cluster the differences into features and to refactor code to adhere to a proper and scalable architecture [12, 20, 42, 45]. As such, 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.

Developer team 1 performed activity **A06, Removing Unused Code**, since their games were a bit larger in size compared to the Android games of team 2. While they appeared to have 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[4]) to remove code that was never executed, reducing the code size by almost 40 %.

Activity **A07, Reverse-Engineer Class Diagrams**, aimed to better understand the legacy systems' structure and guide the development of the platform. Both developer teams extracted class diagrams using Visual Paradigm[5] and IntelliJ Idea, respectively. Diffing the diagrams manually indicated a high degree of reuse on the class level in each case, for example, via generic classes for common game elements, such as the player and enemies.

During activity **A08, Reviewing Source Code**, after the teams analyzed the games' domain and structure, they identified additional features, located the source code belonging to each feature, and documented common and variable parts for the transformation. This again illustrates that the two activity types feature identification and location are intertwined [37].

In activity **A09, Create Feature Model**, the developer teams designed a feature model for their respective product line. They used their knowledge about the cloned variants and their implemented features obtained in the previous steps. The models helped to define the dependencies between features and guided the transformations, with both teams aiming to establish a testable code base first.

Activity **A10, Transforming Source Code to Features**, refers to the actual re-engineering and testing, which differed significantly

[4] https://marketplace.eclipse.org/content/unnecessary-code-detector
[5] https://marketplace.eclipse.org/content/visual-paradigm-eclipse

among both teams (since the variability mechanisms differed significantly), despite many of the preceding activities being similar. 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 variants, 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.

A particular challenge expressed by team 1 for this activity were dependencies between features recognized during the transformation. Specifically, integrating a feature potentially required transforming other features first, given its code dependencies. In fact, planning the order of transforming features is an open problem.

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 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 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 detection, analysis, and transformation [3, 6]. During our case study, both teams switched regularly between various activities, because they identified new features (AT_4) or new code belonging to a feature (AT_6) 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.

- *Our identified activity types abstractly represented the actual activities that need to be performed during re-engineering.*
- *These types can be used to classify re-engineering activities.*
- *The activity types were intertwined during the actual activities (e.g., feature identification and location).*
- *Waterfall-like re-engineering processes are not reasonable.*

RO_1: Activities Performed

4.2 RO_2: Costs of Activities

Results. In Table 4, we summarize the efforts that each team documented throughout its case. Since precisely tracking the effort for each activity was challenging, given the interrelations between activities and the iterations, we double-checked those estimates against the version-control history. Still, we estimate the costs of transformation (AT_8) and quality assurance (AT_9) together, as both teams continuously integrated and tested the variants into the platforms. 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).

Table 4: Costs tracked in terms of effort (person-hours).

ID	Activity Type	Dev. Team 1		Dev. Team 2	
		ph	%	ph	%
AT_1	Software-product-line training	16.00	4.31	90.00	18.15
AT_2	Domain analysis	18.00	4.85	82.00	16.53
AT_3	Preparatory analysis	49.25	13.26	40.00	8.06
AT_4	Feature identification	22.25	5.99	22.00	4.44
AT_5	Architecture identification	2.00	0.54	5.00	1.00
AT_6	Feature location	50.00	13.46	7.00	1.41
AT_7	Feature modeling	7.00	1.88	10.00	2.00
AT_8	Transformation	103.50	27.86	180.00	36.29
AT_9	Quality assurance	103.50	27.86	60.00	12.10
Total		371.50		496.00	

Developer Team 1 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 [29, 37, 67]. Other activities required considerably less effort, resulting in accumulated efforts of 371.5 person-hours in total.

Developer Team 2 performed extensive product-line training (18.15 %) and domain analysis (16.53 %), mainly due to the nature of the cloned variants (Android games). The product-line tooling available did not specifically support Android, if at all, so the team had to spend a substantial effort on fixing and combining tools (e.g., FeatureIDE and IntelliJ IDEA). Still, most of the effort went into the actual transformation and quality assurance (48.39 %).

Discussion. Recall that distinguishing the efforts for each activity is challenging, since activities are intertwined and the developers switch between them in iterations. The activity of reviewing source code may not initiate feature location (AT_6), but the developers will immediately search for additional features (AT_4) and refine their domain knowledge (AT_2). This suggests 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 design the template for our activity sheets, the granularity of these cost models seems to be inappropriate. For instance, SIMPLE defines four cost functions that summarize all activities, while COPLIMO defines factors (e.g., ratio of code modified) without considering activities and tooling.

Both teams experienced that certain system and project characteristics can lead to unexpectedly high efforts. For example, the missing tool support for Android-based product lines drastically increased the costs for developer team 2. Similarly, developer team 1 reported that they had 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. This clearly highlights the impact of system characteristics and challenges the research community to provide new or improved tools that consider these characteristics to support developers.

To further emphasize this point, consider the activities for which we had particularly well working tools: architecture identification (AT_5) and feature modeling (AT_7). For recovering the architecture, both teams relied on automated and established tools, considerably reducing the effort. Similarly, FeatureIDE provides a comprehensible feature-model editor and as long as we collected information on all features, the actual modeling was straightforward. In contrast, most other activities are poorly supported, resulting in considerably more effort—among others because the teams performed feature location mostly manually [29, 37, 60] and only supported by the pairwise comparison (AT_3).

- *The interconnection of re-engineering activities challenged precisely documenting and estimating the corresponding costs.*
- *System and project characteristics heavily impacted the costs.*
- *Developers may not be aware of these characteristics at the beginning (e.g., availability of comments, code quality, existing tooling, and domain knowledge).*
- *Some automation (e.g., architecture recovery) was possible, but more automated tools are needed to support more activities.*

RO_2: Costs of Activities

4.3 RO_3: Cross-Case Analysis

Results. In Table 4, we compare the efforts of both cases. Developer team 2 spent almost 125 hours more, 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 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 spent on each activity.

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, developer 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 [36, 38]. 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 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 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 refined step-wise 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 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 similar among both cases. This is hardly surprising, as most of them are independent of project specifics and are relatively similar. For example, both teams performed comparable steps during the feature identification (AT_4) and architecture recovery (AT_5). 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.

Discussion. Overall, while the efforts for both cases are comparable, we argue that re-engineering a 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 refactor 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.

This discrepancy in activity A10 is somewhat surprising, seemingly contradicting techniques that aim to automatically refactor between composition-based and annotation-based variability mechanisms [31] or that flexibly project either representation [7, 51]. Considering these works, team 1 could potentially have re-engineered its variants into a annotation-based platform and automatically refactored it into a composition-based one. Studying and explaining this discrepancy is subject to future work.

- *Re-engineering into composition-based variability was considerably more costly than into annotation-based variability.*
- *Annotation-based variability required less focus on feature location.*
- *Among the other activities, preparation (e.g., training, domain analysis, code analyses) and transformation were most costly.*
- *The costs of most other activities were similar for both variability mechanisms.*

RO_3: Cross-Case Analysis

5 THREATS TO VALIDITY

Internal Validity. The main threat is that we could not cooperate with the original developer of the Apo-Games. 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 his 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 concerns the granularity of our data collection. Since costs often depend on many factors (e.g., human factors and project specifics), and since activities are intertwined, it is challenging to measure the exact costs per activity and isolate these costs from confounding factors. We mitigated these threats by surveying existing cost models, defining a common template for the activity sheets, and verifying our results with the version-control history. Still, while we reduced confounding factors with this strategy, transferring our costs to other projects should be done with care.

External Validity. A threat is that we used only five cloned Apo-Games variants for each of our two cases. While this limits the generalizability of our results, there exist only a few publicly available datasets of software variants developed using clone & own. However, the dataset has been used to evaluate specific re-engineering techniques in the literature. In this light, while our quantitative results hardly generalize to portfolios of cloned software variants with substantially different characteristics (e.g., variant sizes in LOC, or team sizes in number of developers), we believe the qualitative insights also hold for other re-engineering cases.

Our developer teams' selection of tools might have influenced the documented activities and their costs. Our focus was on reliable tools, such as FeatureIDE and InelliJ IDEA, but there have been some problems, especially with the Android dataset. We also experimented with research tools designed for re-engineering of software product lines. However, these proved less helpful—if we managed to start them at all. As such, while better tooling might exist, our results reflect the state of the art that can be achieved with the current tool maturity and landscape.

6 RELATED WORK

The Apo-Games dataset has been proposed as a baseline for reverse- and re-engineering of software product lines from cloned variants [40]. Since real-world datasets are rare, Apo-Games recently gained attention in the research community. For example, Lima et al.'s [46, 47] technique to automatically recover the architecture of cloned variants is evaluated on Apo-Games; and so is Mendonca et al.'s [50] technique to automatically create Pareto-optimal feature models that can represent the existing variants. While both techniques could have supported re-engineering activities, we tried to avoid collecting additional information on the Apo-Games to avoid biases in our methodology, and to start with the same conditions for both developer teams. We published a prior study on the Apo-Games [42], which focused on feature location. We also published descriptions (short papers) of the platforms we created during the present case study [1, 18], also reporting early experiences. In the present paper, we report a full re-engineering of variants into platforms in a case study with two cases, reporting on all activities performed as well as their costs, and we compare the two cases.

Other case studies and case-study collections on re-engineering product lines from cloned variants exist. The ESPLA catalog [48] collects case studies that report experiences and lessons learned, based on industrial and open-source systems [25–28, 70]. The catalog also highlights the problem that many artifacts are not or only partly available. We are also not aware of any case study that studies the activities and their costs costs empirically. Consider, for example, Rubin et al. [61], who report experiences of re-engineering three sets of industrial cloned variants towards product lines; likewise, Fenske et al. [22] provide automation for refactoring and merging the systems into a common platform. While such works are promising, neither elaborates on the activities that are needed to analyze the systems or the resulting costs.

7 CONCLUSION

We reported a case study on re-engineering two cases of cloned variants into product-line platforms. We documented re-engineering activities and measured the costs (in terms of person-hours) needed. For broader insights, we varied the technological environment (standalone Java versus Android) and the variability mechanisms (preprocessor versus feature-oriented programming). We found:

- A common set of re-engineering activities performed in both cases. These can be seen as a baseline of activities needed for re-engineering. Furthermore, many of our activities were intertwined and needed to be performed iteratively with frequent switching among the activities—suggesting that re-engineering does not follow waterfall-like processes.
- A re-engineering challenge arising from iterations and interconnections among activities (e.g., the results of one activity are a prerequisite for another), which suggests that having too specialized tools may not optimally support developers. It also reinforces that tracking and documenting costs in isolation for individual activities is difficult.
- Re-engineering cloned systems into a platform relying on composition-based variability mechanisms is more complex than re-engineering into annotation-based mechanisms. Our experiences suggest that composition-based mechanisms require more training and more effort in modularizing code. Most other activities required similar efforts in both cases, with transformation and quality assurance contributing to most costs.

In future work, we plan to extend our analysis to more cases, especially industrial ones. Based on a larger corpus of data, we hope to obtain more detailed and empirical data on activities and costs, also confirming or refuting our activities and activity types. We plan to refine and extend our logging and measurement strategy to more automatically collect empirical data and to control for confounding factors, such as tool majurity, developer experience, and system sizes. However, our results show that we still need to advance existing tools and provide them in a mature form. Finally, another open research question concerns finding an optimal order of integrating features during re-engineering, which turned out as a challenge specifically for developer team 1, which recognized feature dependencies requiring transforming dependent features first.

ACKNOWLEDGMENTS

Supported by the Swedish Research Council (257822902) and the German Research Council (SA 465/49-3). We thank Jennifer Horkoff for valuable comments.

REFERENCES

[1] Jonas Åkesson, Sebastian Nilsson, Jacob Krüger, and Thorsten Berger. 2019. Migrating the Android Apo-Games into an Annotation-Based Software Product Line. In *SPLC*.

[2] Muhammad S. Ali, Muhammad A. Babar, and Klaus Schmid. 2009. A Comparative Survey of Economic Models for Software Product Lines. In *SEAA*.

[3] Vallabh Anwikar, Ravindra Naik, Adnan Contractor, and Hemanth Makkapati. 2012. Domain-Driven Technique for Functionality Identification in Source Code. *ACM Sigsoft Software Engineering Notes* (2012), 1–8.

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

[5] Sven Apel, Christian Kastner, and Christian Lengauer. 2009. FEATUREHOUSE: Language-Independent, Automated Software Composition. In *ICSE*.

[6] Wesley K. G. Assunção, Roberto E. Lopez-Herrejon, Lukas Linsbauer, Silvia R. Vergilio, and Alexander Egyed. 2017. Reengineering Legacy Applications into Software Product Lines: A Systematic Mapping. *Empirical Software Engineering* 22, 6 (2017), 2972–3016.

[7] Benjamin Behringer, Jochen Palz, and Thorsten Berger. 2017. PEoPL: Projectional Editing of Product Lines. In *ICSE*.

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

[9] 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*.

[10] Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. 2013. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering* 39, 12 (2013), 1611–1640.

[11] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jabier Martinez. 2019. The State of Adoption and the Challenges of Systematic Variability Management in Industry. *Empirical Software Engineering* (2019). Preprint.

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

[13] Barry Boehm, A. Winsor Brown, Ray Madachy, and Ye Yang. 2004. A Software Product Line Life Cycle Cost Estimation Model. In *ISESE*.

[14] Paul C. Clements and Charles W. Krueger. 2002. Point / Counterpoint: Being Proactive Pays Off / Eliminating the Adoption Barrier. *IEEE Software* 19, 4 (2002), 28–31.

[15] Paul C. Clements, John D. McGregor, and Sholom G. Cohen. 2005. *The Structured Intuitive Model for Product Line Economics (SIMPLE)*. Technical Report CMU/SEI-2005-TR-003. Carenegie-Mellon University.

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

[17] 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*.

[18] Jamel Debbiche, Oskar Lignell, Jacob Krüger, and Thorsten Berger. 2019. Migrating Java-Based Apo-Games into a Composition-Based Software Product Line. In *SPLC*.

[19] 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*.

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

[21] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. 2008. Selecting Empirical Methods for Software Engineering Research. In *Guide to Advanced Empirical Software Engineering*. Springer, 285–311.

[22] 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*.

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

[24] Thomas Fogdal, Helene Scherrebeck, Juha Kuusela, Martin Becker, and Bo Zhang. 2016. Ten Years of Product Line Engineering at Danfoss: Lessons Learned and Way Ahead. In *SPLC*.

[25] Paul Grünbacher, Rick Rabiser, Deepak Dhungana, and Martin Lehofer. 2009. Model-Based Customization and Deployment of Eclipse-Based Tools: Industrial Experiences. In *ASE*.

[26] Nili Itzik, Iris Reinhartz-Berger, and Yair Wand. 2015. Variability Analysis of Requirements: Considering Behavioral Differences and Reflecting Stakeholders' Perspectives. *IEEE Transactions on Software Engineering* 42, 7 (2015), 687–706.

[27] Hans P. Jepsen, Jan G. Dall, and Danilo Beuche. 2007. Minimally Invasive Migration to Software Product Lines. In *SPLC*.

[28] Hans P. Jepsen and Flemming Nielsen. 2000. A Two-Part Architectural Model as Basis for Frequency Converter Product Families. In *IW-SAPF*.

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

[30] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. Rep. SEI, CMU.

[31] Christian Kästner, Sven Apel, and Martin Kuhlemann. 2009. A Model of Refactoring Physically and Virtually Separated Features. In *GPCE*.

[32] Peter Knauber, Jesus Bermejo, Günter Böckle, Julio C. S. do Prado Leite, Frank J. van der Linden, Linda M. Northrop, Michael Stark, and David M. Weiss. 2001. Quantifying Product Line Benefits. In *PFE*.

[33] Peter Knauber, Dirk Muthig, Klaus Schmid, and Tanya Widen. 2000. Applying Product Line Concepts in Small and Medium-Sized Companies. *IEEE Software* 17, 5 (2000), 88–95.

[34] Charles W. Krueger. 2001. Easing the Transition to Software Mass Customization. In *PFE*.

[35] Jacob Krüger. 2016. *A Cost Estimation Model for the Extractive Software-Product-Line Approach*. Master's thesis. Otto-von-Guericke-Univerity Magdeburg.

[36] Jacob Krüger. 2018. Separation of Concerns: Experiences of the Crowd. In *SAC*.

[37] Jacob Krüger, Thorsten Berger, and Thomas Leich. 2019. Features and How to Find Them: A Survey of Manual Feature Location. In *Software Engineering for Variability Intensive Systems: Foundations and Applications*. LLC/CRC Press, 153–172.

[38] Jacob Krüger, Gül Çalıklı, Thorsten Berger, Thomas Leich, and Gunter Saake. 2019. Effects of Explicit Feature Traceability on Program Comprehension. In *ESEC/FSE*.

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

[40] Jacob Krüger, Wolfram Fenske, Thomas Thüm, Dirk Aporius, Gunter Saake, and Thomas Leich. 2018. Apo-Games: A Case Study for Reverse Engineering Variability from Cloned Java Variants. In *SPLC*.

[41] Jacob Krüger, Mukelabai Mukelabai, Wanzi Gu, Hui Shen, Regina Hebig, and Thorsten Berger. 2019. Where is my Feature and What is it About? A Case Study on Recovering Feature Facets. *Journal of Systems and Software* 152 (2019), 239–253.

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

[43] Elias Kuiter, Jacob Krüger, Sebastian Krieter, Thomas Leich, and Gunter Saake. 2018. Getting Rid of Clone-and-Own: Moving to a Software Product Line for Temperature Monitoring. In *SPLC*.

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

[45] Max Lillack, Stefan Stanciulescu, Wilhelm Hedman, Thorsten Berger, and Andrzej Wasowski. 2019. Intention-Based Integration of Software Variants. In *41st International Conference on Software Engineering (ICSE)*.

[46] Crescencio Lima, Wesley K. G. Assunção, Jabier Martinez, Ivan do Carmo Machado, Christina von Flach G. Chavez, and Willian D. F. Mendonça. 2018. Towards an Automated Product Line Architecture Recovery: The Apo-Games Case Study. In *SBCARS*.

[47] Crescencio Lima, Ivan do Carmo Machado, Eduardo S. de Almeida, and Christina von Flach G. Chavez. 2018. Recovering the Product Line Architecture of the Apo-Games. In *SPLC*.

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

[49] Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. 2017. *Mastering Software Variability with FeatureIDE*. Springer.

[50] Willian D. F. Mendonça, Wesley K. G. Assunção, and Lukas Linsbauer. 2018. Multi-Objective Optimization for Reverse Engineering of Apo-Games Feature Models. In *SPLC*.

[51] Mukelabai Mukelabai, Benjamin Behringer, Moritz Fey, Jochen Palz, Jacob Krüger, and Thorsten Berger. 2018. Multi-View Editing of Software Product Lines with PEoPL. In *ICSE*.

[52] Damir Nešić, Jacob Krüger, Ştefan Stănciulescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *ESEC/FSE*.

[53] Sebastian Nielebock, Dariusz Krolikowski, Jacob Krüger, Thomas Leich, and Frank Ortmeier. 2019. Commenting Source Code: Is it Worth it for Small Programming Tasks? *Empirical Software Engineering* 24, 3 (2019), 1418–1457.

[54] Jarley Nobrega, Eduardo Santana de Almeida, and Silvio Meira. 2008. InCoME: Integrated Cost Model for Product Line Engineering. In *SEAA*.

[55] Henk Obbink, Jürgen Müller, Pierre America, Rob van Ommering, Gerrit Muller, William van der Sterren, and Jan Gerben Wijnstra. 2000. COPA: A Component-Oriented Platform Architecting Method for Families of Software-Intensive Electronic Products. In *SPLC*.

[56] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with VariantSync. In *SPLC*.

[57] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[58] Christian Prehofer. 1997. Feature-Oriented Programming: A Fresh Look at Objects. In *ECOOP*.
[59] Rick Rabiser, Paul Grünbacher, and Martin Lehofer. 2012. A Qualitative Study on User Guidance Capabilities in Product Configuration Tools. In *ASE*.
[60] Julia Rubin and Marsha Chechik. 2013. A Survey of Feature Location Techniques. In *Domain Engineering*.
[61] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. 2015. Cloned Product Variants: From Ad-hoc to Managed Software Product Lines. *International Journal on Software Tools for Technology Transfer* 17, 5 (2015), 627–646.
[62] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. 2012. Managing Forked Product Variants. In *SPLC*.
[63] Ştefan Stănciulescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *ICSME*.
[64] Daniel Strüber, Mukelabai Mukelabai, Jacob Krüger, Stefan Fischer, Lukas Linsbauer, Jabier Martinez, and Thorsten Berger. 2019. Facing the Truth: Benchmarking the Techniques for the Evolution of Variant-Rich Systems. In *SPLC*.
[65] Frank J. van der Linden, Jan Bosch, Erik Kamsties, Kari Känsälä, and Henk Obbink. 2004. Software Product Family Evaluation. In *SPLC*.
[66] Frank J. van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*. Springer.
[67] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2013. How Developers Perform Feature Location Tasks: A Human-Centric and Process-Oriented Exploratory Study. *Journal of Software: Evolution and Process* 25, 11 (2013), 1193–1224.
[68] Robert K. Yin. 2018. *Case Study Research and Applications: Design and Methods*. Sage.
[69] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. 2006. Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In *SEAS*.
[70] Gang Zhang, Liwei Shen, Xin Peng, Zhenchang Xing, and Wenyun Zhao. 2011. Incremental and Iterative Reengineering Towards Software Product Line: An Industrial Case Study. In *ICSM*.