

# An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse

Anonymous Author(s)

## ABSTRACT

Software reuse lowers development costs and improves the quality of software systems. Two strategies are common: clone & own (copying and adapting a system) and platform-oriented reuse (building a configurable platform). The former is readily available, flexible, and initially cheap, but does not scale with the frequency of reuse, imposing high maintenance costs. The latter scales, but imposes high upfront investments for building the platform, and reduces flexibility. So, each strategy has distinctive advantages and disadvantages, imposing different development activities and software architectures. Deciding for one strategy is a core decision with long-term impact on an organization's software development. Unfortunately, the strategies' costs are not well-understood—not surprisingly, given the lack of systematically elicited empirical data, which is difficult to collect. We present an empirical study of the development activities, costs, cost factors, and benefits associated with either reuse strategy. For this purpose, we combine quantitative and qualitative data that we triangulated from 26 interviews and a systematic literature review. Our study both confirms and refutes common hypotheses. For instance, we confirm that developing for platform-oriented reuse is more expensive, but simultaneously reduces reuse costs; and that platform-orientation results in higher code quality compared to clone & own. Surprisingly, refuting common hypotheses, we find that change propagation can be more expensive in a platform, that platforms can facilitate the advancement into innovative markets, and that there is no strict distinction of clone & own and platform-oriented reuse in practice.

## CCS CONCEPTS

• **Software and its engineering** → *Risk management; Reusability.*

## KEYWORDS

costs, empirical study, clone & own, software product line, platform

## ACM Reference Format:

Anonymous Author(s). 2020. An Empirical Analysis of the Costs of Clone- and Platform-Oriented Software Reuse. In *Proceedings of The 28th ACM Joint ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States*. ACM, New York, NY, USA, 12 pages. <https://doi.org/10.1145/nnnnnnn.nnnnnnn>

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 ACM 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](mailto:permissions@acm.org).

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States

© 2020 Association for Computing Machinery.

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...\$15.00

<https://doi.org/10.1145/nnnnnnn.nnnnnnn>

## 1 INTRODUCTION

Software reuse is one of the most important practices to save development costs, increase the quality, and reduce the time-to-market of software systems [59, 89]. The idea is to avoid re-implementing existing functionality (a.k.a., *features* [1, 7, 82]), but instead to reuse it for new systems (a.k.a., *variants*), which otherwise need to be developed completely anew. Various reuse strategies exist [1, 59], typically classified into: copy & paste, clone & own, and platform-orientation. Copy & paste reuses small code snippets to solve similar or identical problems within the same or another system. In contrast, the other two strategies reuse a whole system or its parts (e.g., components, files, packages or modules) to engineer a new, but similar variant that is customized to individual stakeholder requirements, such as hardware, features, and runtime environments, or non-functional aspects, such as costs, performance, regulations, and energy consumption. We focus on reuse in the large and, therefore, the latter two strategies.

In the **clone & own** strategy (a.k.a., *ad hoc, opportunistic* or *scavenging* reuse; or *cloning in the large*), developers create a copy of an existing system and adapt that copy to new requirements [1, 29, 88]. As a cheap and readily available strategy, organizations typically start with clone & own, which is well-supported by version-control systems, such as Git (branching and merging) and software-hosting platforms, such as GitHub (forking and pull requests) [37, 64, 88]. However, with an increasing number of cloned variants, maintenance easily becomes a challenge, for instance, when developers need to propagate changes, bug fixes or features. Developers easily loose their overview understanding of the variants and are challenged by substantial maintenance overheads [6, 12, 81].

In the **platform-oriented** strategy, developers implement a single code base from which they can derive individual variants, typically using methods and tools known from software product-line engineering (SPLE) [1, 23, 27, 93]. A platform employs variability mechanisms—techniques to implement variation points (e.g., pre-processors or parameterization), typically controlled by features. The features abstractly represent the variability and are typically organized in a feature model [27, 53, 78]; a tree-like structure of features and their constraints. Using, for instance, configurator tools or dedicated build systems, developers derive individual variants (i.e., reuse software) by enabling or disabling features. This strategy is usually advocated for systems with many variants, including highly configurable systems, such as the Linux kernel, which boasts over 15,000 configuration options today, allowing it to run on a range of hardware environments, from small embedded devices to large super-computer clusters. Unfortunately, adopting a platform requires large initial investments into architecting and creating the platform, adopting SPLE methods and tools, as well as training and (re-)organizing development teams, before benefits, such as substantially reduced time-to-market through the automated derivation of new variants as well as reduced maintenance efforts, can be

achieved [63, 85, 93]. In case of migrating from clone & own, an organization needs to re-engineer software not intended for reuse into reusable features, impacting not only the software architecture, but also the business strategy, organizational structure, and processes.

Over the last decades, numerous studies focused on software reuse [3, 4, 10, 14, 17–23, 31–35, 40, 48, 62, 65, 73–76, 86, 90, 92, 93, 95], many of which also describe the migration from clone & own towards a software platform as the most common way to adopt platforms in practice [8, 29]. Such studies indicate that the maintenance burden of clone & own becomes the main problem leading organizations to adopt a platform strategy. Still, to the best of our knowledge, most studies discuss the advantages and disadvantages of either strategy based on personal experiences and educated guesses. The fundamental question remains: **What costs are associated with either reuse strategy and under what circumstances is which strategy preferable?** Answering this question based on empirical data provides a means for organizations to reason more reliably on several practical decisions—for instance, whether to adopt a platform for a new or existing set of variants. Our long-term goal is to establish qualitative criteria, measurement techniques, and decision models to support the decision making of organizations developing many variants of a software system, as well as to enhance their confidence in their decisions.

We present a study of the activities (e.g., bug fixing), costs (i.e., money or developers’ efforts spent [91]), cost factors (i.e., project properties impacting costs), and benefits (i.e., savings) that are associated with the development process of new variants using either strategy—clone & own and platform-oriented reuse. Specifically, we conducted (1) an interview study with 28 practitioners from a large organization that employs both strategies, and (2) a systematic literature review (SLR) [55, 56]. The SLR complements our interview data with an analysis of the insights and data provided in case studies and experience reports. By triangulating from both sources, we provide consolidated quantitative data, distributions (e.g., costs in per cent), and qualitative arguments on development costs, cost factors, and benefits. We combined the two sources to address the problem that costs in software engineering are hard to quantify and assign to specific activities [41, 50, 51, 91]. In fact, few studies report quantitative data, and often only on a subset of the relevant costs. Based on our data, we also discuss the evidence we find for confirming or refuting common hypotheses about software reuse. Our study is of the rare breed that addresses this problem by providing evidence based on empirical data reported for over 100 organizations.

In summary, we contribute:

- Empirical data on the costs and cost factors associated with the activities of clone & own and platform-oriented software reuse.
- A discussion of evidence we find for confirming or refuting common hypotheses on software reuse.
- A replication package with our interview guide and the data we analyzed as an open-source dataset.<sup>1</sup>

Our results show several differences in the activities and costs of clone & own and platform-oriented software reuse that have not been clearly described in research, yet. For instance, our results confirm the common hypotheses that a platform improves quality and reduces development costs even more than clone & own. Still,

<sup>1</sup>Submitted as supplementary material, will be uploaded to zenodo.org

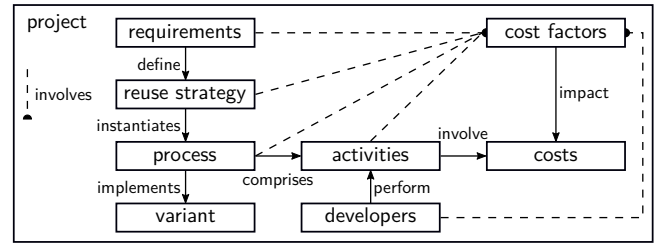


Figure 1: Overview of our conceptual framework.

our results also indicate that clone & own is more advantageous than usually considered in research. Moreover, we identified findings that contradict each other (i.e., are inconclusive) and refute common hypotheses, such as cheaper change propagation in a platform. So, our results help to reason about the costs associated with development activities and compare these between the two strategies, supporting organizations in their decision making. Finally, our data can help researchers and tool vendors to focus their efforts on developing new techniques tackling the most important activities, namely coordination, gaining knowledge about the reusable system, solving ripple effects due to dependencies, and adopting mixed processes for clone & own and platform-oriented reuse.

## 2 METHODOLOGY

In this section, we describe our conceptual framework, research objectives, and methods for eliciting data from our two source.

### 2.1 Conceptual Framework

We provide an overview of our conceptional framework in Figure 1, relating the concepts we are concerned with to each other. A *project* is the scope in which an organization develops a new software variant. Based on a set of *requirements*, the organization defines the *reuse strategy* it employs during the project: clone & own or a platform. The reuse strategy determines a concrete development *process* that ends with the delivery of the specified *variant*. A development process comprises a number of *activities*, such as scoping the variant or bug fixing, performed by the organization’s *developers*. Each activity involves *costs*, for which we consider monetary spendings as well as developers’ effort (e.g., person hours) [91]. Finally, cost factors (a.k.a., cost drivers) represent any project property that impacts the costs of implementing the variant. These factors may be related to, for example, the reuse strategy, specific activities, and developers—and can impact each other (e.g., a higher modularity of a platform may require less design adaptations for a new variant).

### 2.2 Research Objectives

We address three research objectives (RO) in our study:

**RO<sub>1</sub>** *Identify the process and its respective activities that practitioners employ to reuse software for new variants.*

During our collaborations with industrial practitioners, we experienced that the idea of using either pure clone & own or a full-fledged platform rarely applies. These experiences motivated this objective of identifying the processes and activities of software reuse in practice.

**RO<sub>2</sub>** *Identify the costs associated with the activities.*

Based on our data, we aimed to identify the impact of either reuse strategy on the costs of the activities identified. This

**Table 1: Overview of the interviews we conducted.**

ID	Ph.	h	Subjects	Strategy	Data
I <sub>1</sub>	EXP	~0.5	System architect	P	Qual.
I <sub>2</sub>	EXP	~0.5	Software engineer	C&O	Qual.
I <sub>3</sub>	EXP	~0.5	Release engineer	P	Qual.
I <sub>4</sub>	EXP	~0.5	Technical lead	C&O→P	Qual.
I <sub>5</sub>	EXP	~0.5	Technical lead	C&O→P	Qual.
I <sub>6</sub>	EXP	~0.5	Project manager	C&O→P	Qual.
I <sub>7</sub>	EXP	~0.5	2 Software engineers	C&O+P	Qual.
I <sub>8</sub>	PD	>3	Firmware architect	C&O+P	Qual.
I <sub>9</sub>	PD	>3	Software engineer	P	Qual.
I <sub>10</sub>	PD	~1	System architect	P	Qual.
I <sub>11</sub>	PD	~1	Software engineer	C&O+P	Qual.
I <sub>12</sub>	PD	~1	2 Software engineers	C&O+P	Qual.
I <sub>13</sub>	CA	~1	Firmware developer	C&O+P	Qual. & Quan.
I <sub>14</sub>	CA	~1	Software developer	C&O	Qual. & Quan.
I <sub>15</sub>	CA	~1	Technical lead	C&O+P	Qual. & Quan.
I <sub>16</sub>	CA	~1	Technical lead	C&O+P	Qual. & Quan.
I <sub>17</sub>	CA	~1	Software developer	C&O+P	Qual. & Quan.
I <sub>18</sub>	CA	~1	Technical lead	P	Qual. & Quan.
I <sub>19</sub>	CA	~1	System architect	C&O	Qual. & Quan.
I <sub>20</sub>	CA	~1	Technical lead	C&O+P	Qual. & Quan.
I <sub>21</sub>	CA	~1	Software developer	P	Qual. & Quan.
I <sub>22</sub>	CA	~1	System architect	C&O	Qual. & Quan.
I <sub>23</sub>	CA	~1	Software developer	C&O+P	Qual. & Quan.
I <sub>24</sub>	CA	~1	Software developer	C&O+P	Qual. & Quan.
I <sub>25</sub>	CA	~1	Firmware architect	C&O+P	Qual.
I <sub>26</sub>	CA	~1	Software architect	C&O→P	Qual.

Phase: EXPloration; Process Definition; Cost Assessment

C&O: Clone & own; C&O→P: Clone & own migration to platform;

C&O+P: Clone & own and platform; P: Platform

information can help to decide for a reuse strategy and to determine activities that are more challenging to perform.

### RO<sub>3</sub> Analyze the cost factors that impact either reuse strategy.

Finally, we aimed to understand the cost factors impacting the reuse strategies, which helps to identify those that are important to consider for cost estimations and can potentially be altered (e.g., involving more developers) to reduce costs. So, we analyze cost factors also based on economical *benefits* (i.e., reduced costs) that we identified.

Next, we describe how we elicited data to address these objectives.

## 2.3 Collecting Data with Interviews

We conducted interviews with practitioners to collect data regarding our research objectives. In Table 1, we summarize our interviews, whose identifiers (column ID) we use as references.

**Subjects.** Our interview study relied on a close collaboration with a system architect and a product manager (i.e., interviews I<sub>8</sub> and I<sub>9</sub>) at a large organization who have an overview of the organization's development practices and an interest in our study. The organization develops network equipment, including network cameras, print servers, camera servers, scanner servers, and storage servers. Together with our contacts, we sampled 26 subjects with various roles (e.g., software engineers, technical leads, firmware architects) to obtain a broad perspective on their reuse practices. The majority of our subjects is involved in the development of the organization's large portfolio of network cameras. Our subjects reported between three to over 20 years of experiences in one or both reuse

strategies—considering solely their current position, while several reported to have also experiences from previous organizations. We usually interviewed one subject at a time, but twice we involved two subjects in one interview (i.e., I<sub>7</sub>, I<sub>12</sub>) and regularly discussed our objectives with both of our contacts at the same time.

**Interview Design.** We structured our 26 interviews into three different phases: exploration, process definition, and cost assessment. During the first two phases, we conducted unstructured and semi-structured interviews without fixed guidelines, but instead had open discussions, taking notes in parallel. For the third phase, we used our insights to construct a guide for semi-structured interviews. During this phase, we recorded all but one interview (personal preference of the interviewee) and transcribed them later. In addition, we took notes during each interview.

We started with seven *exploratory* interviews that took around half an hour each. During this initial phase, we were interested in understanding the general strategies and policies of reuse at the organization. With our two contacts, we scoped the remaining two phases of our interview study, also considering previous works and our experiences from collaborations with other organizations.

In the *process definition* phase, we conducted five semi-structured interviews. Our goal was to refine our previous insights to construct the reuse processes that are employed at the organization, and to identify cost factors. Finally, we agreed on a consolidated reuse process (i.e., integrating clone & own and platform-oriented reuse) and a set of 10 activities together with our contacts, which is why these interviews took far longer than the intended one hour.

In the last phase, we aimed to quantitatively and qualitatively *assess the costs* and cost factors associated with the activities we identified. We constructed a semi-structured interview guide, which we used for 14 interviews. In each interview, we asked our interviewee to assess the costs of developing a new variant based on their experiences, and to distribute these costs among the activities we identified. Moreover, we asked them to assess the cost factors we collected on a seven-point Likert scale, considering what impact adjusting them would have on costs (i.e., strongly reduces to strongly increases). To this end, we were concerned with any of the two reuse strategies the interviewee worked with, and asked especially those who had experiences with both strategies to explain their perceived differences between both.

**Resulting Data.** In the exploration and process definition phases, we obtained qualitative data (i.e., natural language descriptions) of the reuse processes employed at the organization and the related cost factors (cf. Section 3.1). During the cost assessment phase, we also obtained quantitative data, namely a dataset for each interviewee and their employed reuse strategies—comprising estimates (in per cent) of the cost distribution among the 10 activities (cf. Section 3.2), assessments (Likert scale) on the cost factors' impact (cf. Section 3.3) or both. We could not obtain all data we asked for:

- One interviewee (I<sub>23</sub>) joined running projects, and thus was not confident in estimating the costs of activities.
- One interviewee (I<sub>24</sub>) was not confident in assessing the costs of the first two activities, so we miss these values once for each, clone & own and platform-oriented reuse.
- Two interviewees (I<sub>25</sub>, I<sub>26</sub>) could not provide quantitative data, as they are not involved in variant development, but platform maintenance—for which they reported highly valuable insights.



Overall, we received eight and seven quantitative datasets for clone & own and platform-oriented reuse, respectively. Considering cost factors, we obtained one more dataset for each strategy (due to  $I_{23}$ ). We miss three more values, as interviewees did not feel confident to assess the impact of a cost factor (i.e., team size once for each strategy and the number of teams once for clone & own).

## 2.4 Collecting Data from the Literature

Our SLR focused on qualitatively analyzing the publications identified, without computing or reporting statistics about the publications, which is typically part of SLRs [55, 56]. In Table 2, we show an overview of all publications we selected.

**Search Strategy.** We relied on a manual search that encompassed five sources, given the many problems of automated searches in digital libraries [2]. All publications we identified through the first three sources served as starting set for a snowballing search [96], and we validated our results against related work (fifth source).

*Source<sub>1</sub>: Knowledge.* First, we analyzed publications that we knew and that we deemed potentially relevant for our review. After we applied our inclusion criteria (explained shortly), we selected 15 publications. These publications are marked with a **K** in Table 2.

*Source<sub>2</sub>: Resources.* Clone & own and platform-oriented reuse are core research topics in the field of SPLE. Building on our experience in SPLE, we selected four resources that collect real-world case studies and experience reports on adopting software platforms, usually originating from clone & own: (1) BigLever case-study reports;<sup>2</sup> (2) SPLC Hall of Fame;<sup>3</sup> (3) SEI technical reports on software product lines;<sup>4</sup> and (4) ESPLA catalog [74]. In Table 2, all 24 publications from these sources are marked with an **R**.

*Source<sub>3</sub>: Manual Search.* Using a manual search, we identified the most recent publications related to our topic. We analyzed the last three completed editions (as of July 2019) of relevant conferences (research and industry tracks) and journals through DBLP, namely: 2016-18 ESE, ESEC/FSE, ICSME, IST, JSS, IEEE Software, SPE, SPLC, TOSEM, and TSE.

2017-19 ICSE, ICSR, and VaMoS.  
During this manual search, we found seven publications, marked with an **M** in Table 2.

*Source<sub>4</sub>: Backwards Snowballing.* Using the selected publications, we employed backwards snowballing to identify further publications concerned with our research objectives. We did not limit the snowballing to a specific number of iterations, but continued with any newly identified publication. Still, not all publications were accessible, so we could not check a minority of publications and excluded them from our search. As a result, we identified another 12 relevant publications through backwards snowballing.

*Source<sub>5</sub>: Validating against Related Work.* In our SLR, we also identified SLRs, mapping studies, and surveys that are related to ours. We used these to validate the completeness of our SLR, checking the publications included in the related work against our own sample. We knew the books of Pohl et al. [82] and van der Linden et al. [93], both describing the fundamentals of SPLE as well as providing 11 and 15 industrial case studies as examples, respectively. However,

**Table 2: Overview of the publications included in this study.**

S	Ref	Venue	RM	Organizations (Subjects)	Strategy
V	[45]	CMPSAC'90	MCS	5	C&O+P
K	[14]	AL'92	ER	IBM	P
	[71]	IEEE Soft.'94	MCS	HP	P
K	[42]	IEEE Soft.'95	ER	Matra Cap Systems	C&O
R	[15]	Tech. Rep.'96	CS	CelsiusTech	P
V	[36]	ARES'98	ER	ABB	P
	[84]	JSS'98	IS	83 (109)	P
R	[3]	Book'99	ERs	Nokia, Cummins, HP, Deutsche Bank, US NRO, Philips	P
V	[68]	SPLC'00	ER	LG	C&O→P
R	[20]	Tech. Rep.'01	ER	US NRO	P
V	[34]	JSS'01	QE	4 (4)	C&O
R	[35]	Tech. Rep.'01	ER	Market Maker	P
KR	[23]	Book'02	ERs	Cummins, US NRO, Market Maker, CelsiusTech	P
R	[22]	Tech. Rep.'02	IS	Salion	P
R	[25]	Tech. Rep.'02	ER	DoD-NUWC	P
R	[16]	PFE'03	ER	Salion	P
	[31]	ICSE'03	ER	Alcatel	C&O→P
	[32]	SPE'03	CS	Deutsche Bank	C&O→P
V	[10]	Tech. Rep.'04	ER	Argon	P
	[90]	APSEC'04	CS	Dialect Solutions	C&O→P
R	[11]	Tech. Rep.'05	ER	Engenio	C&O→P
R	[19]	Tech. Rep.'05	ER	TAPO, RCE	C&O→P
K	[82]	Book'05	ERs	HP, Lucent, Siemens	C&O→P, P
KR	[43]	OOPSLA'06	ER	Engenio	C&O→P
	[58]	SPLC'06	CS	Testo AG	C&O→P
	[87]	ISESE'06	IS	Statoil ASA (16)	P
V	[47]	SPLC'07	ER	OTs	C&O→P
R	[49]	SPLC'07	ER	Danfoss	C&O→P
K	[93]	Book'07	ERs	AKVAsmart, Bosch, DNV, Market Maker	P
	[46]	IEEE Soft.'08	MCS	2	C&O
K	[54]	ESE'08	MCS	Apache, Gnumeric	C&O
R	[60]	SPLC'08	ER	HomeAway	P
	[72]	JSS'08	S	57 (57)	P
	[86]	SEAA'08	IS	1 (11)	P
	[48]	CrossTalk'09	ER	Overwatch Systems	C&O→P
R	[69]	ICSEC'09	ER	FISCAN	C&O→P
R	[70]	SPLC'11	ER	FISCAN	C&O→P
R	[80]	SPLC'11	ER	Fujitsu QNET	C&O→P
V	[83]	SPLC'11	ER	ORisk	P
R	[98]	ICSM'11	CS	Alcatel-Lucent	C&O→P
K	[29]	CSMR'13	IS	3 (11)	C&O+P
R	[67]	SPLC'13	ER	US Army	P
K	[92]	Chapter'13	ER	Philips	P
	[5]	SER&IP'14	IS	Google (49)	C&O
R	[21]	CrossTalk'14	ER	Gen. Dyn., Lockheed	P
R	[28]	SPLC'14	ER	US Amry	C&O→P
K	[30]	ESEM'14	IS	Multiple (10)	C&O
R	[39]	SPLC'14	ER	DoD	C&O→P
R	[38]	SPLC'15	CS	DoD	C&O→P
M	[6]	JSS'16	IS	Google, 1 (108)	P, C&O+P
KM	[12]	ESEC/FSE'16	IS	Eclipse, R, node.js (28)	P
KMR	[33]	SPLC'16	ER	Danfoss	C&O→P
M	[77]	SPLC'16	CS	Mitsubishi	P
K	[95]	Tech. Rep.'16	IS	Multiple (59)	C&O
KM	[65]	SPLC'18	ER	TA, HCP	C&O→P
KMR	[75]	SPLC'18	MCS	OSS (6)	C&O→P
M	[40]	ICSE-SEIP'19	ER	Samsung	C&O→P

Source: Knowledge; Resources; Manual search; Validation with related work

Research Method: CS: Case study; ER: Experience report; IS: Interview study;

MCS: Multi-case study; S: Survey; QE: Quasi-experiment

as these have often been published before, we first collected the original publications. Similarly, the paper of Northrop [79] and the book chapter of Krueger and Clements [61] are introductions to SPLE providing nine and four case studies, respectively, that we already included. Justo et al. [52] report an SLR assessing what benefits of software reuse have been transferred to industry. Similarly,

<sup>2</sup><https://biglever.com/learn-more/customer-case-studies/>

<sup>3</sup><https://splc.net/fame.html>

<sup>4</sup><https://www.sei.cmu.edu/publications/technical-papers>

Bombonatti et al. [13] performed a mapping study to identify how non-functional requirements are supported by software reuse and how they influence each other. In their SLR, Mohagheghi and Conradi [76] are concerned with reuse in general and investigate the economic benefits in the context of industrial case studies. Finally, the Software Engineering Institute recently published a catalog of SPLE publications against which we compared our sample [24].

Our study is complementary to these publications. While we used all of them to validate the completeness of our own SLR, none provides empirical evidence (especially not quantitative data) on the costs associated with clone & own and platform-oriented software reuse. From the related work, we included the seven publications that are marked with an **V** in Table 2. We only included data directly from the related work if we could not access an original publication, if there was no original publication, or if the related work provided updated data compared to the referenced publications. Note that we employed snowballing also on these new publications.

**Publication Selection.** We defined four inclusion criteria (IC):

IC<sub>1</sub> Written in English.

IC<sub>2</sub> Describes empirical findings on costs of software reuse.

IC<sub>3</sub> Concerned with clone & own, a platform, both, or the migration between the two strategies.

IC<sub>4</sub> Reports experiences regarding costs, not only estimations.

In addition, we employed two exclusion criteria (EC):

EC<sub>1</sub> Not possible to identify whether software was reused based on clone & own or a platform.

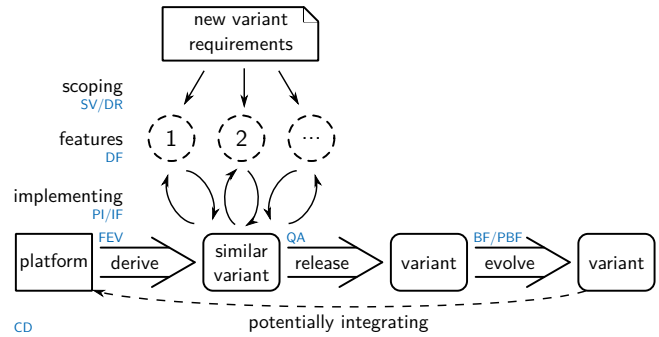
EC<sub>2</sub> Only cites cost effects reported in previous publications, but does not provide new data.

For EC<sub>1</sub>, we classified each publication based on keywords, for example, components, the C preprocessor (both platform strategy) or copying of complete systems or modules (not just copy & paste).

**Quality Assessment.** We analyzed the costs reported in each publication, which were often described as an outcome, insight or byproduct, for example, in experience reports. So, the data we were interested in is often only reported to show the efficiency of employing a reuse strategy, driven by industrial experiences, not systematically elicited, and sometimes not peer reviewed (e.g., technical reports). We decided to not perform a quality assessment, as the publications have different goals and research methods—challenging comparisons; and the assessment would not add benefits to our analysis—intentionally building on practical experiences [55, 56].

**Data Extraction.** For each publication, we extracted bibliographic data, namely the authors, title, publication venue, and publication year. We extracted further data that was relevant for answering our research questions (cf. Table 2): the reuse strategies employed, the research method, the organizations and/or subjects (e.g., for interviews), mentioned cost factors, qualitative insights, and quantitative data. We collected the data in a semi-structured document (provided in our dataset<sup>1</sup>) and used identifiers to trace the data back to its source. To analyze our data, we used an open-card sorting-like method [99] to connect findings on the same reuse strategy and cost factors by identifying synonyms.




**Resulting Data.** We identified the sample of 57 publications we show in Table 2. As expected, few publications (10) are concerned with the costs or benefits of clone & own [46, 66]. In particular, only four publications [34, 42, 45, 80] report five instances of quantified data on clone & own, while most only discuss the advantages and



**Figure 2: The organization's development process with blue abbreviations relating to Table 3 (CD affects all activities).**

disadvantages of either reuse strategy—illustrating the value of our interview data. In fact, the migration towards a platform (23 publications) and the benefits of one (28 publications) are considered much more frequently in our sample of publications.

### 3 RESULTS AND DISCUSSION

In this section, we analyze the results for our research objectives. Within the discussions, we highlight how our data relates to common hypotheses and assumptions (e.g., by Knauber et al. [57]) on clone & own and platform-oriented reuse. We denote refutations as , confirmations as , and inconclusive insights as .

#### 3.1 RO<sub>1</sub>: Development Process

We categorized the activities (cf. Table 3) our interviewees reported in the process-definition phase to design and refine the process we show in Figure 2. This abstracted process represents the general development process for new variants in the organization, integrating clone & own and platform-oriented reuse.

**Results.** A major difference to the reuse processes assumed in research is that neither pure clone & own nor pure platform-oriented reuse is employed, but variations of both to varying extents. Usually, the organization scopes features according to new customer requirements and derives a variant (or even the full platform) that is similar to these requirements to separate it from the platform, which is maintained and updated in parallel. On the separated variant, the responsible developers implement the customer requests until the variant can be released. At that point, the developers and platform engineers have to make a decision: On the one hand, they can keep the variant outside of the platform, which would result in what they refer to as a *long-living clone* (i.e., clone & own) that may never be reintegrated into the platform, due to its divergence. On the other hand, and what is mostly done at the organization, the developers can immediately integrate the changes of the variant back into the platform to incorporate new features, wherefore the variant was only a *short-living clone* (i.e., platform-oriented reuse). So, the major difference between clone & own and platform-oriented reuse at the organization is that the maintenance of long-living clones is done by the development team, while short-living clones are maintained by the platform maintainers. Despite the variations in their development process, both reuse strategies essentially comprise the 10 activities we show in Table 3. We use the identifiers to refer to these activities throughout the remaining paper.

**Table 3: Activities we elicited during our interviews.**

ID	Activity
SV	Scoping the Variant according to customer requirements.
DR	Defining the Requirements to specify what must be implemented.
FEV	Finding an Existing Variant that is most similar to the requirements.
DF	Designing the Features that implement the requirements.
PI	Planning the Implementation of features.
IF	Implementing the Features of the variant.
QA	Quality Assuring the implemented variant.
BF	Bug Fixing the variant.
PBF	Propagating Bug Fixes to other variants and/or the platform.
CD	Coordinating the Development between developers and teams.

**Discussion.** The process we identified is similar to what we experienced and researchers recently reported for other organizations [29, 94] and open-source communities [64, 88]. However, researchers usually still consider the two reuse strategies as completely separated: developers can either apply clone & own or a platform. In contrast, our analysis and such recent publications indicate that a combination of different strategies is regularly used in practice. So, it is problematic to apply findings that are reported, for example, on migrating from clone & own towards a full-fledged software product line, to industry. This finding supports the argument that organizations with a set of similar variants (should) strive towards a platform to some extent (e.g., they may employ clone & own, but support it with automated change propagation [81]).

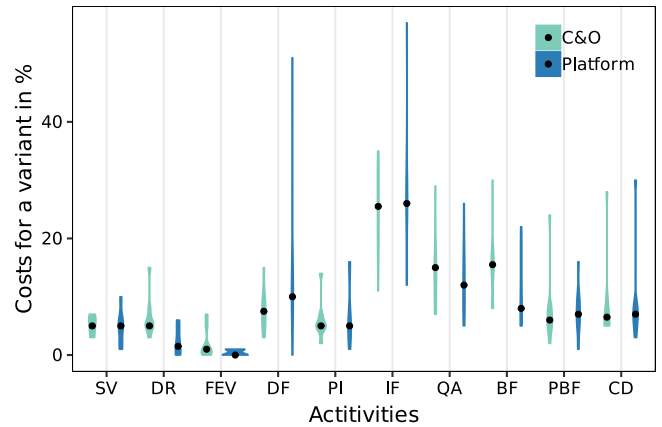
As a result, the question may not be whether to adopt clone & own or a platform, but for what variant should which strategy be employed? For instance, at the organization, short-living clones are rapidly integrated, and thus represent platform-based reuse. They are quite often separated and re-integrated before major changes on the platform happen. The organization usually uses short-living clones to implement well-defined features for established variants. In contrast, some long-living clones are re-integrated only after years or potentially not at all. During our interviews, we found that clone & own is mostly used to advance independently to new markets or test completely new features. However, a problem of this strategy is that new features or variants may be highly valuable and shall be re-integrated into the platform, which becomes far more expensive, due to the long period of co-evolution.

#### RO<sub>1</sub>: Development Process and Activities

*We identified a process (cf. Figure 2) for variant development that integrates clone & own and platform-oriented reuse, comprising 10 activities (cf. Table 3). While both strategies are employed in parallel, they can be differentiated for individual variants.*

### 3.2 RO<sub>2</sub>: Costs of Activities

In the following, we describe the costs that are associated with clone & own and platform-oriented reuse. Our analysis for this research objective was based on qualitative insights and relative, quantitative data, for three reasons: First, it is problematic to identify precise data on software costs, which we mitigated by using qualitative insights. Second, absolute numbers are not representative, as they may be completely out of order for different organizations (e.g., start-ups versus large organizations). To improve our quantification, we combined the results of our interviews (cf. Figure 3)

**Figure 3: Distribution of the costs for developing a variant elicited from our interviews (cf. Table 3).**

with those from our SLR (cf. Figure 4). Finally, we avoid repetitions and can clarify connections and discrepancies between the results. **Results.** We asked our interviewees during the cost-assessment phase to elicit on the distribution of costs among the development activities for a concrete short-living and/or long-living clone based on their experiences. In Figure 3, we display the resulting distributions. As not all interviewees ended up with a sum of 100 % (min 68 %, max 133 %, avg 99.6 %), we normalized their responses to consider the total costs that went into developing a variant. To avoid faulty data, we verified the normalized data and whether we forgot any activity with each interviewee. However, only some of them stated the integration as additional costs after delivering a variant, and we asked them to exclude this activity from their elicitation.

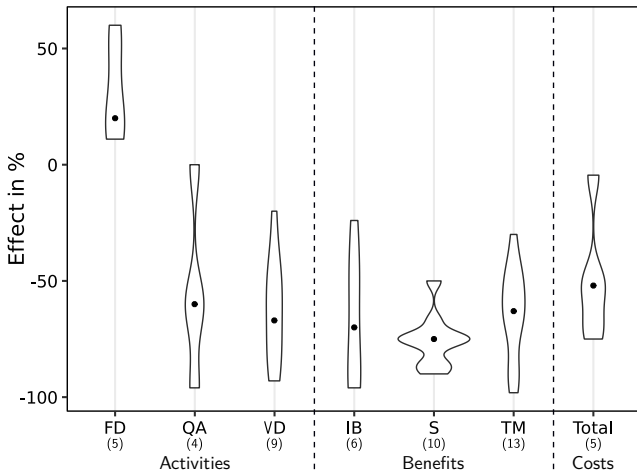
In the literature, we identified numerous qualitative insights on the benefits of employing a platform. We also extracted quantitative data on the effects of platform-oriented reuse on activities, measurable benefits, and the total costs in the form of relative values, which we display in Figure 4. The numbers below each entry correspond to the number of quantitative values, and the dots represent medians. We only show results for which we found at least three values, and did not consider the amount (i.e., in LOC) of reuse itself. So, we display the effect of platform-oriented reuse on the three activities (left side): **feature development (FD)**, **quality assurance (QA)**, and **variant development (VD)**. In the middle, we show the quantified benefits of platform-oriented reuse, namely on **identified bugs (IB)**, **staffing (S)**, and **time-to-market (TM)**. To the right, we show the total cost savings that have been reported for platform-oriented reuse. For simplicity, we summarize values for migrations (comparing to clone & own) and general values (comparing to standalone systems). However, we found that the reported costs and benefits did not differ much, which therefore does not threaten our analysis.

**Discussion.** We structure the discussion according to activities that are related (e.g., to set up development). To this end, we refer to the abbreviations defined in Table 3 and the previous paragraph.

*SV, DR, FEV.* One interviewee stated on selecting a variant (FEV) for development from their platform:

“That’s something that you understand way in the beginning when you get the requirement[s] for the project. You understand now, it’s a derivative of this one, which will be very obvious [...]”





**Figure 4: Impact of platform-oriented reuse on activities' costs (left, RO<sub>2</sub>), reuse benefits (middle, RO<sub>3</sub>), and total development cost (right) that we elicited from our SLR.**

As we can see in Figure 3, this statement aligns to all activities relating to setting up variant development: The costs for defining requirements (DR) and finding a corresponding variant (FEV) are lower and more narrow compared to clone & own, while the initial scoping (SV) is similar for both. This matches with qualitative findings from our SLR, suggesting that platforms can actually improve developers' knowledge about variations [14, 23, 25, 32], that such knowledge is elementary for clone & own [5, 28, 30, 32], and that particularly scoping and selecting an existing variant poses problems in clone & own [29, 32].

*DF, PI, IF, FD, VD.* Clone & own can considerably reduce the costs for developing variants [6, 45, 54, 95]. For instance, Henry and Faller [42] report a cost reduction of 35 %. However, in line with common hypotheses, most data from our SLR indicates that platform-oriented reuse reduces these costs even further [6, 10, 15, 16, 21–23, 25, 83, 86, 87, 92]. In Figure 4, we can see that the nine studies reporting quantitative data suggest median savings of around 67 % for developing a platform variant (VD).

Despite the benefits for developing new variants, we could also confirm that developing features for reuse is typically more expensive than implementing them for single use [10, 71, 92, 92]. Five studies indicate a median increase of 20 %. We can confirm this trend based on our interviews: In Figure 3, we can see that the design (DF) and the implementation (IF) of features are considered to be more expensive for platform-oriented reuse—especially with drastic outliers towards high costs. This supports the argument that platforms will only pay off if features are reused several times. We can best summarize this insight with an interview quote on designing features for reuse:

“For short-living clones, we have to design [...] it to be able to be used by others. A long-lived clone, with that, we can ignore that.”

We found five studies suggesting that organizations can develop a larger number of feature more efficiently with a platform, but they do not provide costs for single features [11, 31, 43, 90]. For instance, Fogdal et al. [33] report that Danfoss could develop over 2,500 features a year instead of under 300 before adopting a platform. The origin of this benefit is unclear, but considering that feature

development is more expensive, it should be caused by other cost factors (e.g., higher quality, more reuse, staffing).

*BF, PBF, QA.* The research community usually argues that propagating bug fixes (PBF) between variants is one of the major drawbacks of clone & own [29, 32, 65]. Indeed, most studies report that bug fixing is a challenging aspect of clone & own [11, 28, 33, 43, 65, 69, 69], particularly as variants co-evolve, which is why fixes must not only be propagated, but also adapted to the new variant. In contrast, platforms are challenging to test in their entirety [6, 12, 86], but are argued to not require change propagation [31, 90].

In contrast, we can see in Figure 3 that our interviewees consider bug fixing (BF) more expensive in clone & own, but the propagation seems more problematic for a platform. One of our interviewees explained this situation as follows:

“Propagate bug fixes, of course, is longer for the short-living clones because we would actually have to do it. For long-living clones, we don't do it at all.”

This indicates that bug propagation may be an important issue, but it requires a re-evaluation by the research community. In particular, it seems necessary to consider that change propagation between clones may not be intended, and thus poses no problem—while change propagation in platforms suffers from feature dependencies (cf. Section 3.3). Still, our SLR (median: -60 %) and interview results confirm reduced costs for quality assuring (QA) a platform.

*CD.* Coordinating the development is a core activity for which we found contradicting reports, considering that researchers usually assume that clone & own provides independence, whereas platforms enforce clearly defined responsibilities and roles. Mostly, coordination and responsibilities (e.g., who owns a feature or variant) are mentioned when problems appear in clone & own [6, 32] or platform-oriented reuse [6, 28, 30, 93]. We identified only one study of Jepsen et al. [49] in which the authors report that a platform facilitates coordination. Our interview data supports this ambiguity: In Figure 3, we can see that coordination is considered as similarly expensive during development for both reuse strategies at a median of 5 % for clone & own and 7 % for a platform.

*Integration.* Considering the integration (or migration) of cloned variations into a platform, we received eight cost estimations from our interviewees (four for either strategy), which align to the results of our SLR. Not surprisingly, developers require considerably less time to fully re-integrate a short-living clone into a platform compared to integrating a long-living clone. In particular, the costs heavily depend on the co-evolution of a separated variant compared to the platform, which we also identified as a major problem in our SLR [11, 28, 30, 43, 54, 65], and the extent of the variant's delta compared to the platform. Again, we can best summarize the problems and costs of re-integrating variants with an interview quote, clearly highlighting the preference of the interviewee towards platform-oriented reuse:

“I think a lot of time is wasted on the long-living clones, because, if you wait one-and-a-half years until you merge, everything [has] changed, maybe. The new Linux kernel, a new version of something else, and then suddenly, your branch is just not working anymore. The longer you wait, the more pain it is. I think you waste a lot of time with those. It's always better to be up-to-date with master.”

This statement also describes causes for the costs of integrating long-living clones, such as understanding the co-evolution, updating old features, as well as fixing outdated dependencies and bugs.

#### RO<sub>2</sub>: Costs of Activities

Our findings support the argument that the success of reuse depends heavily on platform-orientation [72, 84], with our SLR indicating overall median savings of 48 %. The data shows:

- Setting up development is cheaper with a platform.
- Developing features for a platform is more expensive (+20 %), but will pay off due to decreased variant-development costs (-67 %), outperforming clone & own (-35 %).
- A platform increases the quality of the derived software, reducing the costs for quality assurance by 60 %.
- Longer co-evolution between variants (and the platform) increases integration costs.
- Surprisingly, change propagation is more costly for a platform.
- Coordination costs are similar for both strategies.

### 3.3 RO<sub>3</sub>: Cost Factors and Benefits

Finally, we investigated the cost factors impacting both reuse strategies, and their relations. Again, we combined the results from our SLR (cf. Figure 4) with those from our interviews.

**Results.** In Figure 5, we show the responses on the Likert-scale ratings for cost factors that we elicited during our interviews. A positive rating indicates that an increase of the factor (e.g., more reuse for a new variant) is considered to have a positive impact on the costs for a new variant (i.e., reduces costs). In contrast, negative ratings (e.g., a larger delta for the new variant) indicate that our interviewees experienced that an increase of that factor increases costs. We display the assessment separately for clone & own and platform-oriented reuse. For an easier comparison, we show the average values of both strategies in the middle of each cost factor.

**Discussion. Reuse & Delta.** The first two cost factors we analyzed in our interviews are the amount of reusable code and the delta of newly required code for a variant. For both strategies, more reuse reduces costs, while a larger delta increases costs 🍏. Interestingly, the impact of reuse is similar for both strategies, with one particular outlier for clone & own indicating a negative impact:

“Basically, for us it would be more of an effort to remove things, stuff we don’t need compared to just having it there.”

Change propagation and scalability issues have been well investigated for clone & own, in contrast to this problem of removing unwanted features. Finally, it is interesting that a larger delta is considered to have less impact on platform-oriented reuse, which somewhat contradicts our findings that developing new features is more expensive on a platform (cf. Section 3.2) 🍏.

**Developers & Staffing.** A factor that we identified in SLR and interviews is the number of developers involved in development. In Figure 4, we can see that several publications report a decrease in the staff needed to develop a new variant from a platform with a median reduction of 75 % [3, 20, 23, 26, 32, 33, 60, 69, 70, 82]. This supports the established hypothesis that the same number of developers can develop a larger number of variants when using a (main-tained) platform instead of clone & own 🍏. At the organization, we experienced a situation similar to another report [58]:

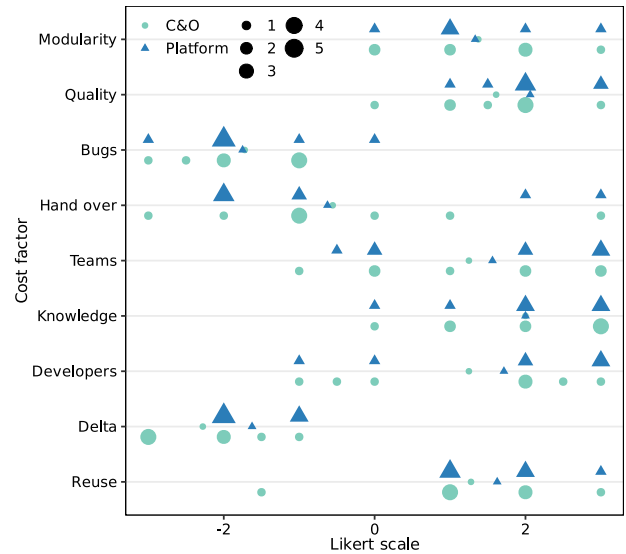


Figure 5: Ratings of cost factors elicited in interviews.

“We had fewer products and fewer developers in the company, the platform was in a horrible state, so you [couldn’t] really use it to release. [It] got more stable, but also the products that were using the platform increased exponentially. Instead of having 10 products on a lousy platform, you have a hundred products on a good platform.”

In both cases, the organization’s potential for growth was large enough to require more staff to address additional customer demands. Still, for developing a specific variant, our interviewees generally consider having more developers (cf. Figure 5) as beneficial. The outliers represent cases in which teams get too large and coordination becomes an issue.

**Knowledge.** The knowledge software developers have about a system is essential for their performance. For developing a set of variants, additional challenges arise, as the developers have to also comprehend variability mechanisms (platforms) or what variants exist (clone & own). Missing knowledge seems to be a primary problem of clone & own and may motivate switching towards a platform. After such a migration, developers must only remember a smaller, structured code base, which may explain this benefit. Still, knowledge is also key to establishing a platform and missing knowledge is a major problem raised in two publications [6, 87]. Our interviewees also highly value knowledge as the factor with the most impact for both reuse strategies (cf. Figure 5). For this reason, they also apply specific policies:

“[We are] growing, so we try to have teams with experienced people together with new people.”

Overall, our results show that knowledge is a primary factor that heavily impacts the costs of software reuse. Unfortunately, we are not aware of a hypothesis or research that is concerned with understanding the impact and differences of knowledge on the two reuse strategies or migrations between them 🍏.

**Teams & Hand Overs.** Related to the previous cost factors, our interviewees indicated that coordination is heavily impacted by clearly defining the size and roles of teams as well as the required hand overs between them:



“I think that the more people you have, it becomes a lot of coordination and also responsibilities [are] not as clear. If you are three people, it’s hard to hide.”

In Figure 5, we can see that the number of teams with defined responsibilities is considered to have a similar positive impact on costs for both reuse strategies. However, the necessary hand overs between teams (e.g., moving a variant from development to quality assurance) is perceived as slightly negative. So, there seem to be no larger differences between both reuse strategies for coordinating in and between teams, aligning to our previous insights 🔄.

**Bugs & Quality.** Software reuse is considered to improve the quality of software, and thus reduce the number of bugs. For clone & own, we found three publications that support this argument [5, 34, 95] and three that argue that quality is a problem [11, 43, 46] 🔄. Still, for platform-oriented reuse, numerous publications argued that it improves the quality compared to other reuse strategies [23, 28, 31, 36, 49, 58, 69–71, 83, 86, 87, 90, 92, 93], including clone & own 🟢. Surprisingly, while the quality of a platform is known to be important for successful reuse, we identified only one paper that stated this as a costly challenge for platform-oriented reuse [58].

We can support both insights with our interview data. For instance, one of our interviewees stated that the organization pushed strongly against clone & own to avoid quality and compatibility issues originating from long-living clones:

“I guess we tried to kill them off because it is a hassle to maintain. [...] If it’s not tested every day, if it’s not daily rebuilt and checked, [...] something is rotting in the code, it’s not being compatible anymore with the platform.”

We can further underpin this with our results for **RO<sub>2</sub>**, indicating that quality assurance is less costly with platform-oriented reuse. Despite these benefits, several interviewees also stated that the path towards this improvement was challenging. The platform was initially of low quality, and thus not trusted. As our interviewees’ assessment (cf. Figure 5) indicates, they consider the quality of the reused system as one of the cost factors with most impact, and it is even more important for platform-oriented reuse 🟢.

The improved quality of a platform is used as an argument that fewer bugs must be fixed, further reducing costs. Indeed, as we can see in Figure 4, several studies report that considerably fewer bugs (median: -70 %) are identified in platforms [3, 20, 21, 23, 33, 71, 80, 83] 🟢. For clone & own, we also found two studies suggesting that this strategy can reduce bugs by 35 % [42] to 66.7 % [80] 🟢. Our interviewees assessed that a change in the number of bugs would have roughly the same effect on development costs for both strategies, as we can see in Figure 5.

**Modularity/Dependencies.** One benefit that we already mentioned is the independence of clone & own [28–30, 90, 95]. This independence allows developers to test variants, implement innovative features or reply faster to customer requests. For example, the organization employs clone & own especially to innovate:

“If it’s a new business we don’t want it [in the platform] because we don’t want to maintain it.”

This is regularly considered to be the main reason to still employ clone & own. In contrast, two publications report that platforms allow to move easier to new markets [23, 58], which is quite surprising. Kolb et al. [58] state that only the platform allowed them to

develop highly complex variants that emerged to new markets. We received an identical response from one of our interviewees:

“I would not be able to have such a complex product if I would not be able to reuse.”

This opposes common hypotheses in research, and we need to better understand what enables organizations to adapt to new and changing markets faster 🔄: independence or a reliable platform?

The independence of clone & own is also considered to free developers of dependencies, but the results of our SLR are quite contradicting in this regard 🔄. For instance, Bowen [14] argues that platforms can help resolve dependencies between clones that are reported as a problem in several publications [5, 6, 48, 95]. This opposes the hypothesis that platforms can pose dependency problems, for which we found little evidence [6, 65]. The main problem may be best explained with the analysis of open-source platforms by Bogart et al. [12], aligning to a statement of one interviewee:

“Since we’re not part of the platform [...], they can sometimes break things they think [...] no one is using [...]. Then, we found out they broke something that we actually use.”

Missing knowledge, community policies or unintended side effects can easily result in misbehaving or missing features, breaking some variants. However, this does not only influence platforms, but also clone & own, where clones are derived from the platform, but intentionally co-evolved. To investigate this issue further, we asked our interviewees about the importance of having a modular structure in the system. They consider modularity to have a positive impact on costs with almost identical impact for clone & own and platform-oriented reuse 🟢.

**Time to Market.** Faster time to market is considered to be a main benefit of software reuse and particularly platforms. Four publications confirm faster time to market for clone & own [5, 30, 46], for example, of around 30 % [80] 🟢. Still, platform-orientation can drastically outperform clone & own [3, 10, 23, 25, 31, 36, 47, 48, 58, 69–71, 80, 86, 87, 92, 93], as new variants can be delivered instantly, if all required features have been implemented (median: 63 %) 🟢.

### RO<sub>3</sub>: Cost Factors & Benefits

Regarding cost factors, our data shows:

- More reuse reduces, while more new code increases costs, with platform-orientation benefiting more than clone & own for both.
- The number of developers developing a variant is a challenging factor to assess correctly for both reuse strategies, but platforms allow to develop more features and variants with the same staff.
- Independent of the reuse strategy, knowledge about the system is an essential cost factor.
- As coordination is challenging, having teams with defined roles is beneficial for both reuse strategies. Still, our results are slightly negative on the impact of hand overs between teams.
- The quality of a platform is essential for its success and results in fewer bugs compared to individual systems (-70 %). Still, both reuse strategies benefit similarly from quality.
- Independence favors clone & own, but can also be achieved with a platform. Surprisingly, both strategies are vulnerable to dependencies and rippling effects, and thus benefit from modularity.
- Clone & own can reduce the time to market by 30 %, but an established platform can lead to a reduction of 63 %.

## 4 THREATS TO VALIDITY

We now discuss the threats to validity of our study, including construct, internal, external, and conclusion validity, following established guidelines and recommendations [55, 97].

**Construct Validity.** We conducted 26 interviews with practitioners who did not use the terminology that has been established in research. To mitigate this threat, we used our exploratory interviews to understand the terms and practices established in the organization. Together with our two contacts, who are familiar with the research terminology, we clarified and unified terms. We used the terms established in the organization to conduct our interviews. Moreover, at least one of the authors was present during all interviews to allow the interviewees to ask about any unclear constructs, and each interview started with an introduction into the purpose and terms of the study.

We extracted data from 57 publications that used different terminology, based on the research focus, domain, and authors. To tackle this issue, we carefully read each publication and extracted keywords to categorize them into one of the two reuse strategy, and to assign our data to the correct costs and cost factors. We aimed to mitigate any threat of wrongly assigning publications or data by using an open-card sorting-like method to unify synonyms. Finally, we aligned this terminology to the one we established for the organization to triangulate the data from both sources.

**Internal Validity.** It is challenging to precisely assign costs to specific activities and cost factors in software engineering, due to, for instance, the tangling of activities (e.g., bug fixing and bug propagation) or cost factors (e.g., developers' knowledge and team composition). Moreover, the costs of developing and particularly reusing software are hard to assign to a specific system, as just copying software has close to zero costs, while the costs of developing features may be distributed among all variants using it. So, costs in software engineering are far more challenging to quantify than in manufacturing, where costs can usually be assigned to a specific product. For those reasons, and because most organizations do not track their costs on a detailed level (e.g., assigning costs to complete projects), quantified data is hard to obtain. We addressed this problem by adopting our research method, triangulating from interviews with experienced software developers and an SLR.

To ensure that we could derive valid results, we used qualitative and quantitative data from two different sources. We conducted the SLR particularly to avoid threats that may be caused by conducting interviews only at one organization. So, we aimed to improve the internal validity of our results by incorporating the experiences of skilled engineers with established research findings. Also, we aimed to improve the completeness of our SLR, verifying our sample against related work. Moreover, we discussed regularly with our contacts to verify that our data and results were sensible. Finally, we discussed our findings with three practitioners from another organization that employs platform-oriented reuse to perform a sanity check based on their experiences.

**External Validity.** An interview study at a single organization is naturally limited in its external validity. However, many software platforms exhibit similar characteristics, whether they originate from open-source projects or industry [44]. Also, many industrial organizations employ similar reuse practices considering clone & own

and platform-oriented reuse [9, 29]. So, while we cannot fully overcome this threat, our results are still important for organizations to understand the costs of their reuse strategies.

We also conducted our SLR to improve the external validity of our study, mitigating the limitations of conducting interviews at a single organization. The SLR includes publications from various domains, levels of maturity, programming languages, countries, and over 100 organizations. So, we argue that the results of our SLR improve the external validity of our study. As we found similar results regarding reuse practices, costs, and cost factors in our interviews, we argue that these are also reliable.

**Conclusion Validity.** We employed interviews and an SLR to mitigate many of the aforementioned threats, and argue that this methodology was appropriate to obtain reliable insights in the costs of software reuse. For this reason, we argue that our findings provide an extensive body of knowledge that provides an intuition and guidance for organizations aiming to establish a platform, and researchers to focus their research efforts. Due to confidentiality reasons, we cannot provide all of our data, for example, on the interview results and the transcripts. Still, we described our methodology in detail and provide a dataset with all artifacts that allow other researchers to replicate our study.

## 5 CONCLUSION

In this paper, we reported the combined results of an interview study conducted at a large organization and an SLR. We described the development process and costs associated with the two main reuse strategies for developing variants: clone & own and platforms. Our analysis is the first to provide systematically elicited and substantial empirical data on these reuse strategies. As a result, we found 18 pieces of evidence that confirm common hypotheses on software reuse. The results suggest that platform-orientation is preferable over clone & own, reducing overall costs, improving quality, and allowing for more variants. Still, we also identified seven inconclusive pieces of evidence, and three that refute established hypotheses:

- Platform-orientation and clone & own are not strictly separated practices, which is often assumed in research.
- Bug propagation is usually considered to be the main problem of clone & own, but our findings indicate that it can be even more challenging and expensive for a platform.
- Clone & own is usually argued to facilitate innovation compared to a platform, but our findings suggest that platforms can actually be essential to successfully advance into innovative markets.

Overall, besides providing the first substantial, evidence-based body-of-knowledge on costs of the two reuse strategies, we obtained important insights that are relevant for practitioners to consider in their decisions, and for researchers to scope their work. For future research, we are especially interested in detailed investigations of our inconclusive and refuting pieces of evidence, and in deriving a decision model for reuse strategies based on empirical evidence.

## REFERENCES

- [1] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. 2013. *Feature-Oriented Software Product Lines*.
- [2] Muhammad Ali Babar and He Zhang. 2009. Systematic Literature Reviews in Software Engineering: Preliminary Results from Interviews with Researchers. In *ESEM*. IEEE.

- [3] Leonard Bass, Paul Clements, and Rick Kazman. 1999. *Software Architecture in Practice*.
- [4] Jonatas Bastos, Paulo da Mota Silveira Neto, Pádraig O’Leary, Eduardo de Almeida, and Silvio de Lemos Meira. 2017. Software Product Lines Adoption in Small Organizations. *Journal of Systems and Software* 131 (2017).
- [5] Veronika Bauer, Jonas Eckhardt, Benedikt Hauptmann, and Manuel Klimek. 2014. An Exploratory Study on Reuse at Google. In *SER&IP*.
- [6] Veronika Bauer and Antonio Vetro. 2016. Comparing Reuse Practices in Two Large Software-Producing Companies. *Journal of Systems and Software* 117 (2016).
- [7] 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*.
- [8] Thorsten Berger, Ralf Rublack, Divya Nair, Joanne Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wasowski. 2013. A Survey of Variability Modeling in Industrial Practice. In *VaMoS*.
- [9] Thorsten Berger, Jan-Philipp Steghöfer, Tewfik Ziadi, Jacques Robin, and Jaber Martinez. 2019. The State of Adoption and the Challenges of Systematic Variability Management in Industry. *Empirical Software Engineering* (2019).
- [10] John Bergey, Sholom Cohen, Lawrence Jones, and Dennis Smith. 2004. *Software Product Lines: Experiences from the Sixth DoD Software Product Line Workshop*. Technical Report. Carnegie Mellon University.
- [11] BigLever. 2005. *BigLever Software Case Study: Engenio*. Technical Report 2005-06-14-1.
- [12] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to Break an API: Cost Negotiation and Community Values in Three Software Ecosystems. In *FSE*.
- [13] Denise Bombonatti, Miguel Goulão, and Ana Moreira. 2017. Synergies and Tradeoffs in Software Reuse - A Systematic Mapping Study. *Software, Practice & Experience* 47, 7 (2017).
- [14] Gregory Bowen. 1992. An Organized, Devoted, Project-Wide Reuse Effort. *Ada Letters* XII, 1 (1992).
- [15] Lisa Brownsword and Paul Clements. 1996. *A Case Study in Successful Product Line Development*. Technical Report. Carnegie Mellon University.
- [16] Ross Buhrdorf, Dale Churchett, and Charles Krueger. 2003. Salion’s Experience with a Reactive Software Product Line Approach. In *PFE*.
- [17] Liangping Chen and Muhammad Ali Babar. 2011. A Systematic Review of Evaluation of Variability Management Approaches in Software Product Lines. *Information and Software Technology* 53, 4 (2011).
- [18] Liangping Chen, Muhammad Ali Babar, and Nour Ali. 2009. Variability Management in Software Product Lines: A Systematic Review. In *SPLC*.
- [19] Paul Clements and John Bergey. 2005. *The U.S. Army’s Common Avionics Architecture System (CAAS) Product Line: A Case Study*. Technical Report CMU/SEI-2005-TR-019. Carnegie Mellon University.
- [20] Paul Clements, Sholom Cohen, Patrick Donohoe, and Linda Northrop. 2001. *Control Channel Toolkit: A Software Product Line Case Study*. Technical Report CMU/SEI-2001-TR-030. Carnegie Mellon University.
- [21] Paul Clements, Susan Gregg, Charles Krueger, Jeremy Lanman, Jorge Rivera, Rick Scharadin, James Shepherd, and Andrew Winkler. 2014. Second Generation Product Line Engineering Takes Hold in the DoD. *CrossTalk – The Journal of Defense Software Engineering* 27, 1 (2014).
- [22] Paul Clements and Linda Northrop. 2002. *Salion, Inc.: A Software Product Line Case Study*. Technical Report. Carnegie-Mellon University.
- [23] Paul Clements and Linda Northrop. 2002. *Software Product Lines - Practices and Patterns*.
- [24] CMU. 2018. *SEI Product Line Bibliography*. Technical Report REV-03.18.2016.0.
- [25] Sholom Cohen, Ed Dunn, and Albert Soule. 2002. *Successful Product Line Development and Sustainment: A DoD Case Study*. Technical Report CMU/SEI-2002-TN-018. Carnegie Mellon University.
- [26] Alejandro Cortiñas, Miguel Luaces, Oscar Pedreira, Ángeles Places, and Jennifer Pérez. 2017. Web-Based Geographic Information Systems SPLE: Domain Analysis and Experience Report. In *SPLC*.
- [27] Krzysztof Czarnecki and Ulrich Eisenacker. 2000. *Generative Programming: Methods, Tools, and Applications*.
- [28] Michael Dillon, Jorge Rivera, and Rowland Darbin. 2014. A Methodical Approach to Product Line Adoption. In *SPLC*.
- [29] 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*.
- [30] Anh Duc, Audris Mockus, Randy Hackbarth, and John Palframan. 2014. Forking and Coordination in Multi-Platform Development: A Case Study. In *ESEM*.
- [31] Christof Ebert and Michel Smouts. 2003. Tricks and Traps of Initiating a Product Line Concept in Existing Product. In *ICSE*.
- [32] D. Faust and Chris Verhoef. 2003. Software Product Line Migration and Deployment. *Software, Practice & Experience* 33, 10 (2003).
- [33] Thomas Fogdøl, 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*.
- [34] William Frakes and Giancarlo Succi. 2001. An Industrial Study of Reuse, Quality, and Productivity. *Journal of Systems and Software* 57, 2 (2001).
- [35] Cristina Gacek, Peter Knauber, Klaus Schmid, and Paul Clements. 2001. *Successful Software Product Line Development in a Small Organization – A Case Study*. Technical Report 013.01/E. Fraunhofer IESE.
- [36] Christopher Ganz and Michael Layes. 1998. Modular Turbine Control Software: A Control Software Architecture for the ABB Gas Turbine Family. In *ARES*.
- [37] Georgios Gousios, Martin Pinzger, and Arie van Deursen. 2014. An Exploratory Study of the Pull-based Software Development Model. In *ICSE*.
- [38] Susan Gregg, Rick Scharadin, and Paul Clements. 2015. The More You Do, the More You Save: The Superlinear Cost Avoidance Effect of Systems Product Line Engineering. In *SPLC*.
- [39] Susan Gregg, Rick Scharadin, Eric LeGore, and Paul Clements. 2014. Lessons from AEGIS: Organizational and Governance Aspects of a Major Product Line in a Multi-Program Environment. In *SPLC*.
- [40] MyungJoo Ham and Geunsik Lim. 2019. Making Configurable and Unified Platform, Ready for Broader Future Devices. In *ICSE*.
- [41] Fred J Heemstra. 1992. Software Cost Estimation. *Information and Software Technology* 34, 10 (1992).
- [42] Emmanuel Henry and Benoît Faller. 1995. Large-Scale Industrial Reuse to Reduce Cost and Cycle Time. *IEEE Software* 12, 5 (1995).
- [43] William Hetrick, Charles Krueger, and Joseph Moore. 2006. Incremental Return on Incremental Investment: Engenio’s Transition to Software Product Line Practice. In *OOPSLA*.
- [44] Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. 2016. Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study. *Empirical Software Engineering* 21, 2 (2016).
- [45] Angelo Incorvaia, Alan Davis, and Richard Fairley. 1990. Case Studies in Software Reuse. In *CMPSAC*.
- [46] Slinger Jansen, Sjaak Brinkkemper, Ivo Hunink, and Cetin Demir. 2008. Pragmatic and Opportunistic Reuse in Innovative Start-Up Companies. *IEEE Software* 25, 6 (2008).
- [47] Paul Jensen. 2007. Experiences with Product Line Development of Multi-Discipline Analysis Software at Overwatch Tectron Systems. In *SPLC*.
- [48] Paul Jensen. 2009. Experiences with Software Product Line Development. *CrossTalk – The Journal of Defense Software Engineering* 22, 1 (2009).
- [49] Hans Jespen, Jan Dall, and Danilo Beuche. 2007. Minimally Invasive Migration to Software Product Lines. In *SPLC*.
- [50] Magne Jørgensen. 2014. What We Do and Don’t Know about Software Development Effort Estimation. *IEEE Software* 31, 2 (2014).
- [51] Magne Jørgensen and Kjetil Moløkken-Østfold. 2004. Reasons for Software Effort Estimation Error: Impact of Respondent Role, Information Collection Approach, and Data Analysis Method. *IEEE Transactions on Software Engineering* 30, 12 (2004), 993–1007.
- [52] José Justo, Fernando Pincirol, Santiago Matalonga, and Nelson Araujo. 2018. What Software Reuse Benefits have been Transferred to the Industry? A Systematic Mapping Study. *Information & Software Technology* 103 (2018).
- [53] Kang et al. 1990. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Technical Report. CMU/SEI-90-TR-21.
- [54] Cory Kapser and Michael Godfrey. 2008. “Cloning Considered Harmful” Considered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering* 13, 6 (2008).
- [55] Barbara Kitchenham, David Budgen, and Pearl Brereton. 2015. *Evidence-Based Software Engineering and Systematic Reviews*. Vol. 4. CRC.
- [56] Barbara Kitchenham and Stuart Charters. 2007. *Guidelines for Performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE-2007-01. Keele University.
- [57] Peter Knauber, Jesus Bermejo, Günter Böckle, Julio do Prado Leite, Frank van der Linden, Linda Northrop, Michael Stark, and David Weiss. 2001. Quantifying Product Line Benefits. In *PFE*.
- [58] Ronny Kolb, Isabel John, Jens Knodel, Dirk Muthig, Uwe Haury, and Gerald Meier. 2006. Experiences with Product Line Development of Embedded Systems at Testo AG. In *SPLC*.
- [59] Charles Krueger. 1992. Software Reuse. *ACM Computing Surveys* 24, 2 (1992).
- [60] Charles Krueger, Dale Churchett, and Ross Buhrdorf. 2008. HomeAway’s Transition to Software Product Line Practice: Engineering and Business Results in 60 Days. In *SPLC*.
- [61] Charles Krueger and Paul Clements. 2013. Systems and Software Product Line Engineering. In *Encyclopedia of Software Engineering*.
- [62] Jacob Krüger and Thorsten Berger. 2020. Activities and Costs of Re-Engineering Cloned Variants Into an Integrated Platform. In *VaMoS*.
- [63] Jacob Krüger, Wolfram Fenske, Jens Meinicke, Thomas Leich, and Gunter Saake. 2016. Extracting Software Product Lines: A Cost Estimation Perspective. In *SPLC*.
- [64] 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).



- [65] Elias Kuitert, 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*.
- [66] Naveen Kulkarni and Vasudeva Varma. 2017. Perils of Opportunistically Reusing Software Module. *Software, Practice & Experience* 47, 7 (2017).
- [67] Jeremy Lanman, Rowland Darbin, Jorge Rivera, Paul Clements, and Charles Krueger. 2013. The Challenges of Applying Service Orientation to the U.S. Army's Live Training Software Product Line. In *SPLC*.
- [68] Kwanwoo Lee, Kyo Kang, Eunman Koh, Wonsuk Chae, Bokyoung Kim, and Byoung Choi. 2000. Domain-Oriented Engineering of Elevator Control Software. In *SPLC*.
- [69] Dong Li and Carl Chang. 2009. Initiating and Institutionalizing Software Product Line Engineering: From Bottom-Up Approach to Top-Down Practice. In *ICSAC*.
- [70] Dong Li and David Weiss. 2011. Adding Value through Software Product Line Engineering: The Evolution of the FISCAN Software Product Lines. In *SPLC*.
- [71] Wayne Lim. 1994. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software* 11, 5 (1994).
- [72] Daniel Lucrédio, Kellyton dos Santos Brito, Alexandre Alvaro, Vinicius Garcia, Eduardo de Almeida, Renata de Mattos Fortes, and Silvio de Lemos Meira. 2008. Software Reuse: The Brazilian Industry Scenario. *Journal of Systems and Software* 81, 6 (2008).
- [73] C. Marimuthu and K. Chandrasekaran. 2017. Systematic Studies in Software Product Lines: A Tertiary Study. In *SPLC*.
- [74] Jabier Martinez, Wesley Assunção, and Tewfik Ziadi. 2017. ESPLA: A Catalog of Extractive SPL Adoption Case Studies. In *SPLC*.
- [75] Jabier Martinez, Xhevahire Tërnavá, and Tewfik Ziadi. 2018. Software Product Line Extraction from Variability-Rich Systems: The Robocode Case Study. In *SPLC*.
- [76] Parastoo Mohagheghi and Reidar Conradi. 2007. Quality, Productivity and Economic Benefits of Software Reuse: A Review of Industrial Studies. *Empirical Software Engineering* 12, 5 (2007).
- [77] Motoi Nagamine, Tsuyoshi Nakajima, and Noriyoshi Kuno. 2016. A Case Study of Applying Software Product Line Engineering to the Air Conditioner Domain. In *SPLC*.
- [78] Damir Nešić, Jacob Krüger, Stefan Stănculescu, and Thorsten Berger. 2019. Principles of Feature Modeling. In *ESEC/FSE*.
- [79] Linda Northrop. 2002. SEI's Software Product Line Tenets. *IEEE Software* 19, 4 (2002).
- [80] Jun Otsuka, Kouichi Kawarabata, Takashi Iwasaki, Makoto Uchiba, Tsuneo Nakanishi, and Kenji Hisazumi. 2011. Small Inexpensive Core Asset Construction for Large Gainful Product Line Development: Developing a Communication System Firmware Product Line. In *SPLC*.
- [81] Tristan Pfofe, Thomas Thüm, Sandro Schulze, Wolfram Fenske, and Ina Schaefer. 2016. Synchronizing Software Variants with VariantSync. In *SPLC*.
- [82] Klaus Pohl, Günter Böckle, and Frank van Der Linden. 2005. *Software Product Line Engineering: Foundations, Principles and Techniques*.
- [83] Gerard Quilty and Mel Ó Cinnéide. 2011. Experiences with Software Product Line Development in Risk Management Software. In *SPLC*.
- [84] David Rine and Robert Sonnemann. 1998. Investments in Reusable Software. A Study of Software Reuse Investment Success Factors. *Journal of Systems and Software* 41, 1 (1998).
- [85] Klaus Schmid and Martin Verlage. 2002. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software* 19, 4 (2002).
- [86] Devesh Sharma, Aybuke Aurum, and Barbara Paech. 2008. Business Value through Product Line Engineering – A Case Study. In *SEAA*.
- [87] Odd Slyngstad, Anita Gupta, Reidar Conradi, Parastoo Mohagheghi, Harald Rønneberg, and Einar Landre. 2006. An Empirical Study of Developers Views on Software Reuse in Statoil ASA. In *ISESE*.
- [88] Ștefan Stănculescu, Sandro Schulze, and Andrzej Wąsowski. 2015. Forked and Integrated Variants in an Open-Source Firmware Project. In *ICSME*.
- [89] Thomas Standish. 1984. An Essay on Software Reuse. *IEEE Transactions on Software Engineering* 5 (1984).
- [90] Mark Staples and Derrick Hill. 2004. Experiences Adopting Software Product Line Development without a Product Line Architecture. In *APSEC*.
- [91] Adam Trendowicz. 2013. *Software Cost Estimation, Benchmarking, and Risk Assessment: The Software Decision-Makers' Guide to Predictable Software Development*. Springer.
- [92] Frank van der Linden. 2013. Philips Healthcare Compositional Diversity Case. In *Systems and Software Variability Management*.
- [93] Frank van der Linden, Klaus Schmid, and Eelco Rommes. 2007. *Software Product Lines in Action - The Best Industrial Practice in Product Line Engineering*.
- [94] Julia Varnell-Sarjeant, Anneliese Andrews, Joe Lucente, and Andreas Stefik. 2015. Comparing Development Approaches and Reuse Strategies: An Empirical Evaluation of Developer Views from the Aerospace Industry. *Information & Software Technology* 61 (2015).
- [95] Robert Walker and Rylan Cottrell. 2016. *Pragmatic Software Reuse: A View from the Trenches*. Technical Report 2016-1088-07. University of Calgary.
- [96] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *EASE*.
- [97] Robert K Yin. 1998. Case Study Research: Design and Method. (1998).
- [98] 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*.
- [99] Thomas Zimmermann. 2016. Card-Sorting: From Text to Themes. In *Perspectives on Data Science for Software Engineering*. Elsevier, 137–141.