

Analysing and supporting software reuse in practice

Veronika Maria Bauer

Institut für Informatik
der Technischen Universität München

**Analysing and supporting software reuse in
practice**

Veronika Maria Bauer

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Prof. Dr. Florian Matthes

Prüfer der Dissertation:

1. Prof. Dr. Dr. h.c. Manfred Broy
2. Prof. Dr. Harald C. Gall,
Universität Zürich, Schweiz

Die Dissertation wurde am 15.06.2016 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 07.11.2016 angenommen.

Abstract

Software reuse, “the use of existing engineering knowledge and artefacts to build new software systems” [1], is considered a key element to reach the goal of delivering high quality software in time and on budget and impacts the entire life cycle of software development, maintenance, and evolution [2, 3]. Research has proposed many conceptual approaches (e.g., [4, 5, 6, 7, 8, 9]). However, due to technological limitations, many of these proposals could at the time not be transferred into practice in a feasible way.

Technological evolution has since enabled to build advanced infrastructures, providing an unprecedented accessibility to knowledge and potentially reusable artefacts ranging from code snippets over libraries, components, to services [10]. This has led to a drastic change in the way software development and software reuse can be approached [11, 12]. To which extent have these developments improved reuse application in practice?

Contributions: This thesis analyses the state of the practice of software reuse by means of two in-depth empirical investigations on software reuse in two large software houses and confirms reports from the literature [13]: finding and adopting an adequate reuse strategy remains a challenge for software producing companies and frequently fails due to underestimation of the effort required for the transition.

Based on a literature study and the empirical results obtained in the investigations, this thesis proposes a Reuse Adoption Support Model (RASM). RASM supports practitioners in selecting an adequate reuse approach with respect to their organizational context. The model incorporates reuse facets derived from a multitude of research papers proposing reuse approaches or reporting their adoption. To show the applicability of RASM, this thesis reports a proof of concept application in a large software organization.

Capturing an organization’s reuse context usually amounts to significant manual effort and might also require extracting information from the source code base of the organization. To increase the feasibility of the latter, this thesis proposes method and tool support to identify potential for reuse, e.g., in the form of semantic redundancies, and an assessment for third-party library reuse. The support provided by the tooling transcends the initial model application phase: by means of iterative application during the adoption, it provides feedback on the effects of the implemented measures and, thus, valuable information for driving the reuse process.

Acknowledgements

Many people knowingly or unknowingly contributed to this work: by exchanging ideas and forming them, giving and requesting feedback, serving as examples and role models, facilitating the logistics, as well as providing moral support during the obligatory failures and sharing the enthusiasm in case of success. All of this helped me to carry this work to completion and I am deeply grateful for it.

In particular, I would like to express my gratitude to my supervisors, Prof. Dr. Dr. h.c. Manfred Broy and Prof. Dr. Harald C. Gall: your continued interest in the topic as well as the support for experimenting with and realising my own ideas was valuable for shaping the work into its current form. Manfred, thank you for providing the opportunity for pursuing my doctorate at your chair. Harald, thank you for the hospitality of your research group over the last two years. I have always felt welcome.

A substantial amount of my research relied on input from practitioners. I would like to express my gratitude to all study participants that offered their valuable time to share their insights into software reuse, giving sincere accounts of their challenges and successes, and providing a reality check for my ideas.

Warm thanks go to my colleagues for the support and energy contributed to our joint work, be it research projects, writing, teaching, reviewing, or debating. It was a pleasure to work with you! In particular, I'd like to thank my Munich and Zurich office companions Fiona and Katja for their encouragement, support, and enthusiasm.

Productive work would have hardly been possible without the support of the secretaries and technical support staff and I would like to thank Silke and Dieter representative for all.

Working on this dissertation has, at some points, been challenging. All the more, I'm grateful for the patience and moral support I received from colleagues and friends. I'm afraid, I was not always easy during that time. Thank you for bearing with me :)

Last but not least, my thanks go out to my family: my parents and siblings for their encouragement and unconditional support, as well as to my aunt Margaret for being my first academic role model. Thank you, Diego, for keeping me sane during the last stretch of completing this work, thank you Miriam, Hannah, and Eva for the laughter and joy you bring to my life!

"It always seems impossible until it's done."

— *Nelson Mandela, 1918-2013*

Contents

I	Introduction	1
1	Analysing and supporting reuse strategies in practice	3
1.1	Software reuse - a current topic for today's software practice?	4
1.2	Problem statement	6
1.3	Goal and research method	7
1.4	Contributions	7
1.5	Outline	10
1.6	Previously published material	11
2	Software Reuse: Terms and fundamentals	13
2.1	Visions and motivation	14
2.2	Reuse philosophies	16
2.3	Reusable entities	18
2.4	Reuse in practice	22
2.5	Position of this thesis	32
II	Evaluating the state of practice of reuse	33
3	Case studies on reuse in industrial practice	35
3.1	Empirical studies on software reuse in practice	36
3.2	Methodology	36
3.3	Case description G	37
3.4	Case description U	38
3.5	Original case study designs	39
3.6	Data collection & analysis procedures	40
3.7	Company reuse placement	40
4	An exploratory case study of software reuse at Google	43
4.1	Study goal and context	44

4.2	Methodology	46
4.3	Study results	47
4.4	Discussion	57
4.5	Threats to validity	58
4.6	Considerations for practitioners	59
4.7	Summary and conclusions	60
5	A case study of software reuse adoption	61
5.1	Challenges of structured reuse adoption	62
5.2	Study design	64
5.3	Adoption of a strategic reuse program	65
5.4	Lessons learned — Adoption attempts	70
5.5	Current research collaboration	72
5.6	Summary and conclusions	74
6	Synthesizing the case studies	75
6.1	Comparing reuse practices	76
6.2	Study goal and research questions	76
6.3	Study design	77
6.4	Analysis Methodology	79
6.5	Study Results	80
6.6	Discussion and relation to state of the art	87
6.7	Threats to validity	95
6.8	Summary and conclusions	96
III	Guiding strategic reuse decisions in practice	99
7	A pragmatic model for guiding reuse adoption in practice	101
7.1	Guiding reuse adoption in practice	102
7.2	Reuse adoption support model	103
7.3	Model overview	103
7.4	Structure of intent	104
7.5	Structure of the reuse facets	105
7.6	Application of RASM	114
7.7	Justification	115
7.8	Company Reuse Placement	118
7.9	Summary	121
8	Applying the decision model in practice	123
8.1	A proof-of-concept application of RASM in practice	124
8.2	Background of model application at U	124
8.3	Model application	125

8.4	Results for case U	127
8.5	Limitations of evaluation	136
8.6	Next steps	136
IV	Methods and tools to detect reuse potential	139
9	Detecting reuse potential in the context of a RASM application	141
9.1	Detecting reuse potential in source code	142
9.2	Discovering unintentional re-implementations	142
9.3	A hybrid approach to discover unintentional re-implementations	147
9.4	Combining clone detection and LSI to detect re-implementations	153
9.5	Cross-project clone detection as guidance for reuse improvement	166
9.6	Conclusion	170
10	A structured assessment model for third-party library usage	171
10.1	Opportunities and risks of third-party library reuse	172
10.2	Assessment model	173
10.3	Assessment process	178
10.4	Tool support	179
10.5	Case study	181
10.6	Related work	186
10.7	Summary and future work	188
V	Conclusion	191
11	Summary and conclusions	193
11.1	Summary of the contributions	193
11.2	Outlook	195
11.3	Conclusions	196
VI	Appendix	197

Part I

Introduction

1 | Analysing and supporting reuse strategies in practice

Software reuse, “the use of existing engineering knowledge and artefacts to build new software systems” [1], is considered a key element to reach the goal of delivering high quality software in time and on budget and impacts the entire life cycle of software development, maintenance, and evolution [2, 3]. Starting from the late 1960’s [14], decades of research efforts have been spent on analyses, methods, tools, and empirical investigations targeting to support practitioners in executing reuse tasks [15, 16, 11].

Many conceptual approaches have been proposed, addressing aspects ranging from the creation and organization of reusable artefacts, e.g. object-oriented techniques and programming languages, components-off-the-shelf (COTS) approaches [4], reuse repositories [5, 6], or software product lines (SPLs) [7, 8], to organizational support and templates e.g. the “experience factory” [17], the “reuse curator model” [18], or the REBOOT approach [5, 19].

Visions were that reuse would soon be effected on an abstract level by means of techniques such as automated code generation or Very High Level Languages (VHLLs) [2]. In addition, planned and strategic reuse programs were advocated as most beneficial, once the high initial efforts were completed. However, due to technological limitations, many of these proposals could at the time not be transferred into practice in a feasible way. For instance, the reuse repositories at the core of several approaches proved as tedious in set-up and maintenance, involving significant manual intervention [12]:

“Component-based software reuse faces an inherent dilemma: in order for the approach to be useful, the repository must contain enough components to support developers, but when many examples are available, finding and choosing appropriate ones becomes troublesome.” [6] — S. Henninger, 1997

This led to a significant research effort in classification and retrieval techniques, that, however, could not overcome the challenge of providing practitioners with an effective way to build, populate, and maintain a dedicated reuse repository that could be adapted to meet changing organizational needs [6]. In addition, organizational challenges soon became apparent: Early-on, researchers vocalize one of the most challenging aspect of planned reuse in practice, the sepa-

ration of those who are investing into reuse and those who profit from this investment, which requires a global focus of management transcending the scope of single projects:

“The traditional unit of analysis and control for software managers is the software project, and subsequently the resulting application system. [...] Yet there is a range of insights that can only be attained through the monitoring and management of the software inventory at the level of the entire firm. [...] Reuse, by its nature, is an activity that spans multiple projects and application systems enterprisewide. To manage such reuse requires monitoring the firm’s software at the level of the organization or enterprise.” [20] — R. D. Banker et al., 1993

Technological evolution, especially the creation of the internet technologies, have since enabled to build advanced infrastructures, providing an unprecedented accessibility to knowledge and potentially reusable artefacts ranging from code snippets over libraries, components, to services [10]. Furthermore, Software Engineering best practices, such as, e.g., requirements engineering, software architecture design, automated quality analyses by means of static analyses, code reviews, and continuous integration focus (to different extents) on creating reuse and avoiding redundancies. Also development paradigms based on the new infrastructure possibilities, such as agile methodologies, test driven development, etc. have significantly changed the way of developing software. This has also lead to a drastic change in the way reuse can be approached in terms of publishing, retrieving, and maintaining reusable entities [11, 12]. Figure 1.1 briefly illustrates this development. At the same time, empirical research has started to quantify the benefits of reuse [21]. In addition, the Open Source community has embraced the potential of reuse, e.g., in terms of software libraries [22], and has demonstrated the benefits thereof that have not remained unnoticed in industrial practice [23]. In this context, the following questions impose themselves:

How do software producing companies nowadays approach reuse? Which reuse approaches do they choose for their development, maintenance, and evolution? Do they proceed in a planned and strategic manner on an abstract level? Is reuse a topic mastered in practice?

1.1 Software reuse - a current topic for today’s software practice?

Industry encounters provide a variety of insights on the state of reuse in practice [25, 26, 27]: Reuse in general is playing a key role in everyday software development, maintenance, and evolution. Furthermore, in some highly specialized domains, Software Product Lines (SPLs) [8] have been successfully adopted, commercially successful, and support a high level of reuse [28].

However, evidence suggests that for a large number of software producing companies, finding and adopting an adequate reuse strategy remains a challenge for several reasons:

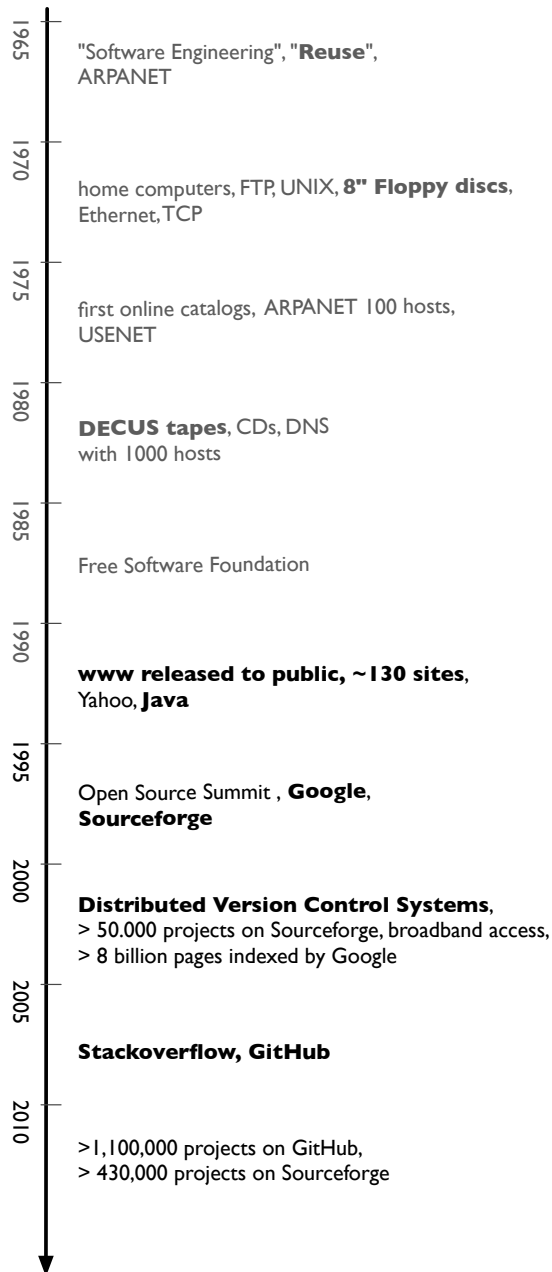


Figure 1.1: A brief overview of the technological development since reuse became a topic of software engineering research. The time-line displays the media and infrastructure available to share reusable entities before the WWW-era and highlights the birth of open source platforms and available tools that now serve as important sources for reusable code and/or knowledge. Based on [24].

First, the idea of software reuse is very intuitive on an abstract level and, thus, tends to mislead practitioners (and particularly management) to underestimate the effort required to adopt a reuse approach, as well as the time-frame in which benefits can be expected [13].

Second, adoption of a reuse approach often-times requires changes not only in the development processes of an organization: most of the times, significant changes in non-technical processes and the organizational structure are necessary to enable reuse success [17]. Identifying and addressing the required changes is likely to cause disruption and scepticism within the organization [29].

Lastly, the characteristics of already existing systems can cause significant challenges for reuse adoption: Usually, the systems at hand tend to incorporate a varied mix of technologies, architectural styles, and programming conventions. Large number of libraries and frameworks (produced within the companies or drawn from company-external sources) contribute to the systems [22, 30, 31]. The systems have grown to considerable size over generations of developers pressured to ship a product to the market. As a result, many implementation decisions have been taken to optimize for a local goal [20]. These developments tend to incite redundancies and inhibit company-wide reuse efforts.

Reuse has usually occurred in a pragmatic and unstructured way with the means at hand, often-times resulting in a significant technical debt [32] manifested in the form of, e.g., undocumented redundant implementations [33], code clones, unmanaged dependencies, and unreflected use of libraries [34, 35, 36]. This way of effecting reuse during a systems' development can entail significant costs during maintenance and evolution. Attempts to adopt more disciplined and structured reuse approaches have often been underestimated in terms of the required effort [13] and have failed to produce benefits in relation to the invested resources.

In order to determine and adopt a reuse strategy, the context factors mentioned above, as well as their effects, need to be understood and considered to reach an informed decision about software reuse at the respective organization. The key questions of this dissertation are therefore:

How can we enable practitioners to select a feasible reuse strategy? Can we capture and improve (deteriorated) reuse situations?

1.2 Problem statement

Practitioners lack support to determine adequate reuse strategies. This results in a mismatch between business goals, context, and employed reuse approaches, causing significant inefficiencies in development and increased efforts in maintenance.

To mitigate this challenge, this work supports practitioners in making an informed decision with respect to software reuse, based on a concrete and structured assessment of their organization's reuse capability. It does so by providing systematic reuse adoption guidance based on own investigations and the available literature.

1.3 Goal and research method

Selecting an adequate reuse strategy that respects company goals and capabilities is crucial for beneficial reuse. This thesis aims to support practitioners in selecting an adequate reuse strategy for their context. To reach this goal, we propose the following steps for this work:

In a first step (Contribution 1), we turn towards analysing current reuse practices. This is a necessity to ensure this work is grounded on the needs of practitioners as well as ensuring the relevance of the subsequent contribution. From a methodological perspective, we select empirical survey research as means of study and conduct two detailed case studies on reuse in practice. The studies are conducted at two large software companies and focus on a detailed understanding of current reuse practices, supporting and inhibiting context factors, benefits and drawbacks. The case studies are complemented by a detailed synthesis study that integrates their results.

In a second step (Contribution 2), this work integrates the results from the first contribution with a literature review to propose a constructive reuse adoption support model for practitioners aiming to design a reuse strategy. The model draws the basis from a classification scheme derived from a literature study and connects the characteristics of the reuse approaches with the context knowledge gained from the empirical studies. When applying the model, potential sources of effort or incompatibilities become apparent at an early stage of the decision making process. In this way, the application prevents an underestimation of efforts and allows for an informed discussion of options. These claims are supported by a proof-of-concept case study in industry.

Lastly, we propose supporting methods and tools (Contribution 3) to support data collection for the reuse adoption support model. In particular, we focus on the identification of missed reuse opportunities, quantifying the potential for reuse in source code, and the chances and risks presented by the third-party libraries that are incorporated in a given system. The proposed methods and tools are validated by means of industry case studies.

1.4 Contributions

The contributions of this dissertation are threefold: First, it investigates the current state of software reuse in two large software producing corporations by means of two in-depth field studies with the purpose of identifying challenges in current software reuse practice. The results highlight different forms of reuse practice, a focus on code reuse, and challenges in selecting and adopting a new reuse approach. In addition, they indicate the need to account for economic goals and company context during the selection process.

By combining literature with own empirical results, this work reaches the conclusion that underestimating the impact of introducing reuse programs is a crucial point of failure in practice. To mitigate this issue, this dissertation provides, as second contribution, a decision support model capturing the essential questions decision-makers have to answer in order to reach an informed decision on the feasibility of introducing a specific reuse approach.

Lastly, this work presents methods and tool support for specific aspects of the reuse decision support. The following sections provide details on each contribution (visualized in Figure 1.2).

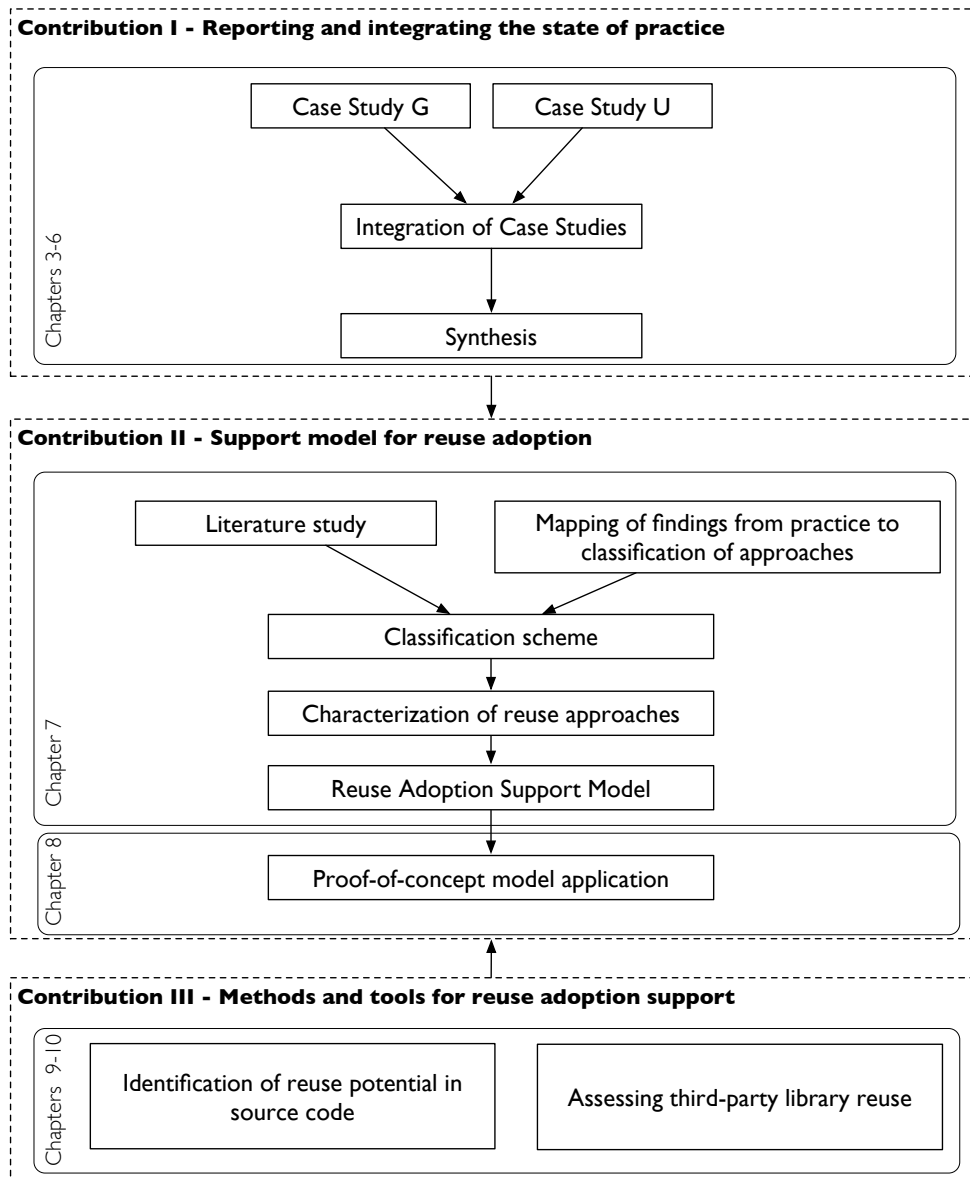


Figure 1.2: Map of the contributions presented in this thesis and their relations. The arrows represent the flow of results: the synthesis of Contribution I is integrated in Contribution II. The methods and tools presented in Contribution III can be used to support the application of the Reuse Adoption Support Model of Contribution II. The rounded boxes provide a mapping of the contributions to the Chapters of this thesis.

Contribution 1. Two detailed case studies and a structured comparison on software reuse in practice in two large software houses

Most research accounts of software reuse in practice focus on narrow aspects and often fall short of providing a contextual details, such as goals and domain constraints, of the organization from which they report. This decreases the value of the reported experiences, as reasoning about success and failure, as well as applicability of a given approach, lacks vital information.

To obtain first hand insights into current reuse practices, we conduct two in-depth case studies on software reuse in two software houses that are highly diverse in domain, age, and culture. We report on the implementation of reuse, influence (context) factors, effects, and challenges. The studies each comprise extensive interviews and a comprehensive on-line questionnaire, capturing the knowledge of 138 experienced software engineers, architects, designers, and managers. Furthermore, we provide a structured approach for integrating evidence from diverse studies and report a comparison of the findings of both studies.

The studies highlight that reuse in practice occurs in many different flavours, however, mostly limited to source code. Partially, the technological potential has been embraced, rendering once infeasible approaches, such as repositories as source for reusable entities, operational. Successful reuse is tightly coupled to the company goals and compatible with in the development culture. Establishing systematic reuse in heterogeneous contexts poses significant challenges and requires structured decision support.

Contribution 2. A support model for guiding software reuse adoption

Over decades, research has proposed numerous approaches and techniques for software reuse. Those proposals range from the "visionary" and abstract to the very concrete and technology-bound. Reports on (un)successful adoption of these research proposals, on the contrary, are limited and suggest difficulties in selecting suitable reuse approaches. This contribution proposes a scheme to systematize the reported evidence and capture the respective characteristics with the goal to enable informed decisions on reuse approaches.

The results of a literature review highlight that a large number of reuse approaches and reusable entities have been proposed by research that could be adopted in practice in a wide range of contexts. However, the set of reported goals and benefits is quite focussed and allows for a stringent selection of reuse approaches in a given business context.

Based on the literature analysis, mentioned above, and the empirical findings obtained in Contribution 1, this contribution develops a first version of a qualitative decision model for stakeholders that need to find a suitable reuse strategy for their software development. This contribution addresses the concern raised in literature and practice that the consequences of introducing structured reuse are notoriously underestimated in practice, causing significant loss in terms of failing reuse programs.

The model includes key factors of the different reuse approaches, such as expected benefits, time horizon to benefit realization, preconditions, drawbacks, etc. and relates them to strategic

aspects of reuse, highlighting potential synergies and/or incompatibilities. In this way, decision makers are guided to address relevant aspects in their considerations.

Our proof-of-concept model evaluation at an organization supported practitioners with identifying critical aspects of their envisioned reuse adoption. In the specific application, the model re-focused the analysis and discussion on aspects that would, otherwise, have been neglected in the often politicized debate around the potential organizational changes. This allowed practitioners to detect incompatibilities between the envisioned reuse strategies and the time frame for benefit realization on the one hand, and the current state of the company on the other hand, at an early stage. As a result, they could target the implementation of beneficial changes in reach of the current company capabilities and, at the same time, in line with the change trajectory towards the envisioned approach. Practitioner feedback supports that applying the model was helpful for performing a thorough up-front assessment of their envisioned reuse strategy.

Contribution 3. Methods and tools for reuse assessment

For several aspects of the adoption support model, we present methods and tools that support the assessment of reuse potential, as well as chances and risks of current reuse. These methods can, on the one hand, be applied to identify potential for reuse during a model application. On the other hand, they can support the implementation of reuse strategy changes by monitoring their effect on the underlying source code and, thus, allowing to identify the effectiveness of the applied measures.

On the technical level, we present two types of methods: on the one hand, we focus on detecting potential for reuse. The respective methods support identifying candidate reusables by detecting missed reuse opportunities in the form of semantic re-implementations, as well as considering the application of clone detection across project boundaries to detect candidates for reusable entities. On the other hand, the contribution provides support for assessing the opportunities and risks present on a project due to the reuse of third-party libraries.

1.5 Outline

Part 1 introduces the concept of software reuse, provides the background for this work, and presents an overview on the state of the art as well as the state of the practice of software reuse reported in literature. Part 2 presents our insights into reuse in industrial software practice. It presents the two case studies as well as the synthesis study that are part of Contribution 1. Part 3 presents the Reuse Adoption Support Model and its proof-of-concept evaluation in industry, Contribution 2. Part 4 presents methods and tools for measuring and monitoring aspects of a company's code reuse practices, Contribution 3. Part 5 summarises and concludes the dissertation.

1.6 Previously published material

Parts of this work have been published in [25, 26, 27, 34, 37, 38, 39, 40, 41].

2 | Software Reuse: Terms and fundamentals

“Reuse is easy to understand but challenging to institute.” [42] — S. Wartik and T. Davis, 1999

This chapter introduces the topic of software reuse. It presents the various definitions and gives an introduction on the visions and motivation that have driven research in the area. Subsequently, the chapter briefly describes the most prominent reuse philosophies and highlights important characteristics of reusable entities. To provide the background to this work, the chapter proceeds with a summary of the state of the art on software reuse in practice, detailing on impact factors enabling or hindering reuse success. Finally, the chapter establishes the position taken by this work.

Contents

2.1	Visions and motivation	14
2.2	Reuse philosophies	16
2.3	Reusable entities	18
2.4	Reuse in practice	22
2.5	Position of this thesis	32

2.1 Visions and motivation

Software reuse is one of the oldest topics in Software Engineering. Proposed by McIlroy [14] as a concept for addressing the “Software Crisis” in the late 1960’s, it has been defined and addressed from a variety of aspects. Krueger, in his survey on software reuse, captures the overall intuition:

“Software reuse is the process of creating software systems from existing software rather than building software systems from scratch.” [2] — C. Krueger, 1992

Subsequent definitions extended the scope to encompass the aspects of knowledge and software artefacts:

“Software reuse, the use of existing software artefacts or knowledge to create new software.” [43] — W. Frakes et al., 1996

2.1.1 Reuse visions

The original vision of software reuse was first published in McIlroy’s 1968 paper on “Mass produced software components” [14]. In the text, he lines out the vision of a structured approach to reuse that enables system development by assembling proven and adequate software entities:

“Software components (routines), to be widely applicable to different machines and users, should be available in families arranged according to precision, robustness, generality and time-space performance.” [14] — M.D. McIlroy, 1968

He, furthermore, identified potential inhibitors to and potential benefits of such enterprise, derived from the experience of reuse in more seasoned and mature engineering domains:

“Existing sources of components [...] lack the breadth of interest or coherence of purpose to assemble more than one or two members of such families, yet software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors.” [14] — M.D. McIlroy, 1968

Early-on, the ideas of reusing finalized components, as well as ‘software bases’ [46] for domain specific and refinable solutions, were picked up and solutions were evaluated on a theoretical level. However, implementation of the propositions remained challenging due to restricted technological options [6].

Apart from its role in realizing high quality software systems on time and within budgets, researchers shared McIlroy’s vision of software reuse serving as means to increased maturity of the software engineering discipline.

“The major contribution of software reuse to software engineering [...] will be to push it into maturity.” [47] — R. Prieto-Díaz, 1994

In this vision, *structured and systematic reuse* is central to the advance in maturity:

“Current trends to institutionalize reuse and to integrate reuse in the software development process further demonstrate that the end and the success of software reuse are close, and that both converge into systematic reuse.” [47] — R. Prieto-Díaz, 1994

During this process, software reuse itself would disappear as an conscious aspect of research and development and become an integral and ingrained part of software production and, thus, a solved problem before the year 2000 [47, 48].

2.1.2 Reuse motivation

The original concerns software reuse should address were a rising demand for software that could hardly be met by industry, the high cost of producing software systems, and the desire to form software engineering such that it resembled other more mature engineering disciplines [14, 46, 48].

“Production of software is costly and error prone, and its main production factor, good programmers, is scarce. Therefore, one strives to solve these problems by ultimately circumventing this costly manual production process.” [46] — R. Mittermeir and W. Rossak, 1987

In this context, reuse received its appeal from observations on the nature of ‘typical’ software products containing large amounts of domain-independent functionality¹:

“Most applications devote less than 10% of their code to the overt function of the system; the other 90% goes into system or administrative code: input and output; user interfaces, text editing, basic graphics, and standard dialogs; communication; data validation and audit trails; basic definitions for the domain such as mathematical or statistical libraries; and so on. It would be very desirable to compose the 90% from standard parts.” [49] — M. Shaw, 1995

Goals Software reuse, thus, seemed like the key concept that could resolve, or at least significantly mitigate, the issues mentioned above: Reuse of existing elements would *improve productivity and time-to-market* by *reducing implementation time*, *increase the quality* of the software products as the likelihood of errors prevailing in artefacts was assumed to be reduced by its prior testing and use, and *maintenance would be eased* by modular composition of the systems resulting from reuse of parts (see, e.g., [11]), seemingly at no (or negligible) cost [50].

This favourable view of software reuse prevails up to this date with researchers and practitioners alike, as the following quotes illustrate:

“To improve software productivity, when constructing new software systems, programmers often reuse existing libraries or frameworks by invoking methods provided in their APIs.” [51] — Zhong et al., 2009

¹This qualitative assessment has been validated quantitatively by more recent research, e.g. by Heinemann et al. in the context of Open Source Development [22].

“[W]e present strong evidence that code reuse is of major importance in OSS development and has contributed to its success. We further show that OSS developers perceive efficiency and effectiveness as the main benefits of code reuse.” [52] — M. Sojer and J. Henkel, 2010

“The reuse of knowledge is considered a major factor for increasing productivity and quality. In the software industry knowledge is embodied in software assets such as code components, functional designs and test cases. This kind of knowledge reuse is also referred to as software reuse.” [53] — Spoelstra et al, 2011

“It has been argued that only software reuse can bring the gain of productivity in software construction demanded by the market.” [22] — Heinemann et al., 2011

“The usage of third-party libraries can decrease development time and cost through reuse of existing pieces of functionality.” [54] — Raemakers et al., 2012

Quantitative studies (summarized in [21]), confirm parts of the claimed benefits in a laboratory setting. However, the body of qualitative studies on the subject of reuse adoption in practice provides a more heterogeneous assessment: studies indicate that in practice substantial up-front investments are required to enable the adoption of advanced software reuse approaches (see Sections 2.2.2 and 2.4.3).

2.2 Reuse philosophies

Literature distinguishes between *ad-hoc* and *structured* reuse philosophies. As this differentiation has a significant impact on goals, methods, applicability, and effort, we detail on both in the subsequent paragraphs.

2.2.1 Ad-hoc reuse

Ad-hoc reuse generally refers to an unplanned approach to reuse that follows no specific, company-wide strategy and, thus, is effected at the discretion of the individual programmers [55]. Reusables ranging from code snippets to libraries are combined in an opportunistic way to reach an increase in terms of development speed. In this style of reuse, de-facto reusables can co-exist with reusables by design without restrictions.

Most of the time, this term is used synonymously for *clone-and-own* reuse which denotes a reuse approach that relies on ad-hoc copying and modifying (proven) solutions for the purpose of increased development pace. It is also known as: *code scavenging* [2], *pragmatic reuse* [56], *opportunistic reuse* [57], and *copy-and-paste* (or *cut-and-paste* or *copy-and-modify*) reuse [58], *pragmatic reuse* [59].

Code snippets are among the reusable entities that can be obtained with the greatest ease. However, also high-level reusables, e.g., components, third-party libraries, and frameworks can

be reused in a pragmatic way. In either case, *clone-and-own* reuse implies taking ownership of reusables by copying them to a new location and adopting them to the specific needs.

Clone-and-own has the undeniable advantages of visible and fast progress, as well as flexibility and independence of consumers [56, 60]. As it has virtually no preconditions on the organizational context, it is applied widely in industry [60]. Depending on the given industrial context, this practice can serve as a disciplined tool [61] used to circumvent technical hindrances of more structured reuse approaches [59, 62]. Nevertheless, it induces a known array of risks, e.g., in the form of error propagations, unmanageable code base volumes, decreased understandability. This leads to high maintenance efforts that decimate the benefits gained by the faster development pace [63].

2.2.2 Structured reuse

Structured reuse, has a long history of being considered a “silver bullet” [64] in research and practice alike, as it is considered superior to pragmatic reuse in terms of the benefits it provides [65, 3, 29]. At its core stands the vision of a well defined and structured base of high-quality artefacts for a specific domain [66] that have been explicitly developed for reuse and can be used to construct new products with minor customization effort. High-level artefacts, e.g. requirements, are considered as first-class reusables [66]. Clear and repeatable processes capture the responsibilities of producers and consumers, lining out assumptions and guarantees, as well as capturing feedback [17]. Maintenance of the reusable entities is usually centralized and the responsibility of the producers. Tool support allows for locating, accessing, and connecting reusable entities.

The most prominent representative of this kind of reuse approaches is the one of Software Product Lines [8]. SPLs reportedly have been adopted successfully in practice [28] and have provided significant benefits [8, 67]. These benefits, however, only follow significant up-front investments and perseverance in adopting the reuse strategy across several organizational units:

“Reuse is a mid-term investment impacting the entire software development process. It must be based on a product strategy which spans several releases or a family of products.” [50] — M. Wasmund, 1994

Successfully introducing structured reuse, therefore, is a non-trivial task that transcends typical organizational boundaries (e.g. of departments) and units of work planning (such as single projects) and, thus, requires a holistic view of the organization.

Research offers some guidance for structured reuse introduction by proposing, e.g., organizational blueprints supporting reuse or incremental adoptions of revised development paradigms [17, 68]; however, these measures often require significant and long-term changes in industry. These deep changes are challenging to effect [69], especially whilst continuing everyday business activities. In addition, management is not necessarily aware of the impact of required transformations and the related efforts [50, 29, 13]. This reportedly leads to unwillingness to continuously support the reuse adoption and to a failure of that venture [13]. Furthermore, incentive conflicts, such as local team priorities or unofficial policies, can significantly hinder the adoption of structured

reuse [18, 70]. These unofficial policies are often backed by short-time evaluation of applied reuse strategies: Dubinsky et al. [60] investigate the reasons that cause companies to reluctantly move away from ad-hoc reuse in the form of code cloning for product line implementation to structured product line engineering approaches. They identify efficiency, low overhead, short-term thinking, and lack of governance as main drivers.

Independently of organizational aspects, creating or obtaining suitable reusable entities is a key challenge of structured reuse: reverse engineering of existing solutions or creation of new entities require for a deep understanding of domain and business context. In addition, they need to find an adequate compromise between specificity and generalization to ensure ease of reuse and avoid significant adaptation efforts [71]. During the life-cycle of the reusables, dependencies and impact of changes in maintenance and evolution need to be considered and carefully managed [72].

The above mentioned challenges highlight that success of structured reuse adoption depends to a significant degree on organizational factors. Long-term management commitment, awareness of human factors, and modification of non-reuse processes according to the specific context of the company are, therefore, critical enabling factors [29, 13].

2.3 Reusable entities

Over the typical software product life-cycle (as, e.g., in [73]), many diverse entities and artefacts are created that could and should be reused in another setting [46, 71]. The potential artefacts range from knowledge [43] and product requirements [74, 75] over code snippets [59] to software frameworks and libraries [76], reference architectures [77] and Software Product Lines [8], encapsulating tested knowledge and proven solutions. Depending on the selected styles of reuse, the set of reused entities varies. Figure 2.1 illustrates this heterogeneity: in the central axis, it shows reusables ordered by degree of abstraction (the increase of abstraction is expressed by the arrow). For the categories *models* and *source code*, examples for possible forms of reuse are given (e.g. by use of generators, in different granularities of source code packaging).

Black- and white-box reuse Literature differentiates between *black-box* and *white-box* reuse. This notion refers to the insights a consumer has on the internals of the reusable, as well as to their possibility to modify them. Subsystems, components, or libraries are typically reused in a black-box fashion² with behaviour modification obtained by parametrization. Textual artefacts, e.g., source code, are usually reused in a white-box fashion, allowing the consumer to inspect and modify its details.

Reusability One crucial aspect of reusable entities is their degree of *reusability*, a measure capturing

“the ease with which the resource can be reused in a new situation.” [15] — Y. Kim
and E. Stohr, 1992

²This, clearly, is not the case if the source code of e.g. a component is copied and integrated in another code base. In this case, the component is reused in a white-box fashion and is represented in textual source code.

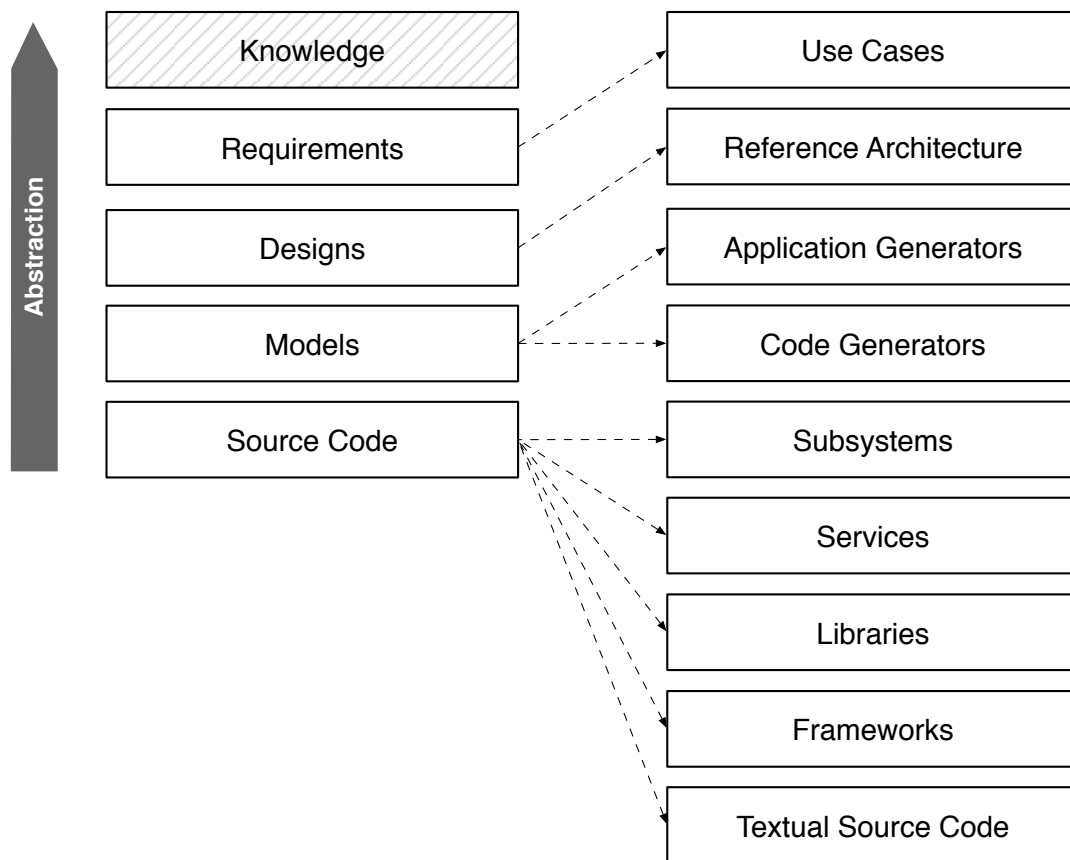


Figure 2.1: Examples for potentially reusable entities created during software development. On the left side of the figure, the different types of artefacts are ordered in terms of their level of abstraction. The arrows connected to the artefact types refer to typical examples of the respective type. Knowledge is listed as it is mentioned in several definitions. However, it is highlighted as it is not a documented type of artefact.

The goal of this measure is to create the basis for a comparison of the applicability of multiple components in a specific system context. Several “-ilities” are considered factors constituting reusability (e.g. adaptability, composability, maintainability, modularity, portability, reliability, understandability) [56].

A significant amount of research has been invested into metrics for and models of reusability, albeit, these research efforts have often been fragmented [11]. This fact has in the past negatively impacted the evaluation and application of these measures in research and practice and is now gradually tackled by recent structuring efforts (e.g. [56]). Furthermore, the proposed reusability metrics map the constituting factors exclusively on technical properties and contexts of the component (e.g. code metrics such as cyclomatic complexity, interface stability), thus ignoring further vital characteristics for practical use (such as e.g. characteristics of the provider).

Other researchers, e.g. Basili [78], have proposed more high level notions of reusability. We briefly summarize them here. According to Basili, one should take note of three aspects in order to understand the reusability of an entity: the characteristics of the reusable, its context, and the process of transforming the reusable to enable its reuse. In terms of characteristics or dimensions of the reusable, the following should be considered: its *type*, its *self-containedness*, and its *quality*. With respect to the context dimension, one needs to consider the original *requirements* as well as the current *solution domains*, which might rely on different assumptions and pursue different goals. For the transformation of the reusable, one needs to consider the *mechanism* by which knowledge-transfer is going to occur, the *type of transformations*, their *integration* in the development process, and the *resulting quality* of the transformation. Whilst this framework stays at a rather abstract level, it illustrates the steps that are performed (implicitly or explicitly) when selecting and adapting a reusable.

Intent From a reuse application position, an additional aspect of reusables becomes apparent, and is illustrated by Figure 2.2: there is a difference between artefacts built for reuse and those reused out of convenience, here denoted as *de-facto* reusables and reusables *by design*. This difference regards the responsibilities of the *producer*, i.e., the unit or person producing a given reusable entity, and the *consumer*, i.e., the unit or person reusing the reusable, of available reusables.

In the case of *de-facto reusables*, the reused entity has not necessarily been built to support reuse at all, in a *consumer-agnostic* way. It is, therefore, reused to speed up the development process, without knowledge about the rationale or inherent assumptions about the intended application context. Furthermore, the producer of the reusable has no liability to assure any specific level of quality or guarantees for the entity. Since consumers usually have no possibility to influence the form of the original reusable, they have to effect all necessary changes and maintain the reused parts in the future.

In the case of *reusables by design*, effort has been invested to support specific consumer requirements. Therefore, this kind of reusable is *consumer-aware*. Consumers can obtain information about the assumptions for the use cases or intended applications of the reusables that can be used *as is* or altered and extended. Furthermore, producers provide a certain set of guarantees for the intended scope of use of the reusable. Maintenance usually is the responsibility of the producer, except for custom extensions added by the consumer.

The intent underlying a reusable is, to some extent, congruent to its scope of use: Textual source code, reused in an ad-hoc way is usually a de-facto reusable, reused in the local scope of a package or a project. Libraries, or some components are reusables by design and, in this role, are built to be used on any scope that requires their specific functionality. Product line platforms, in turn, are de-facto reusables built for a company-wide strategic scope.

Reusables	De-facto	By design - utility	By design - domain specific
Built for reuse	not necessarily	yes	yes
Functionality	any kind	general purpose	domain specific
Motivation for reuse	development efficiency	development efficiency, maintenance gains, quality guarantees	development efficiency, maintenance gains, quality guarantees
Assumptions	unknown	known for basic set of application	known for use case/ functionality
Guarantees	unknown	for basic set of application	for use case/ functionality
Effort invested in reusability	none for de-facto instance of reuse	for basic set of application	for use case/ functionality
Effort invested during reuse	selection, adaptation, maintenance	selection, no maintenance for use-as-is, maintenance for altered versions	selection, no maintenance for use-as-is, maintenance for extended versions
Example	code snippets	general purpose libraries	domain specific component
Focus on consumers	Consumer-agnostic reusable	Consumer-aware reusable	Consumer-aware reusable

Figure 2.2: Types of reusables in practice.

2.4 Reuse in practice

The alleged benefits of software reuse have not failed to attract the attention of industry, where the need for cost reduction and quality improvements is perceived. In addition, the potential of reuse to foster innovation and market penetration due to shorter production cycles promised strategic business advantages.

Benefits have been reported from successful reuse adoptions: lower cost and faster development [79, 80, 81, 27], higher quality [79], standardized architecture [82, 79], and risk reduction [81] by resorting to known artefacts. Whilst these reports are encouraging and provide valuable insights into reuse-conducive factors (see Section 2.4.2), they are outnumbered by accounts of unsolved challenges (see Section 2.4.3).

Reports from literature and practice suggest that adoption of a suitable reuse strategy is challenging: Software reuse takes place in a multi-faceted environment and, thus, incorporates aspects ranging from technical to organizational at different levels of abstraction [1, 80].

Congruence of business goals, context factors, and processes need to be established for reuse to provide the desired economical effects [83]. Options for reuse approaches range from loose to tight reuse [13] that involve different levels of organizational commitment.

In addition, the selection of adopted reuse approaches differs among the various domains of software development: reuse practices in embedded and non-embedded software development differ with component-based approaches and product lines prevailing in embedded systems development, whilst in non-embedded contexts ad-hoc reuse is most frequent [81].

Implementing reuse approaches requires developing and communicating a reuse vision — including what should be reused (e.g. *source code*), for which purpose (e.g. *decreasing development effort*), in which technical realization (e.g. *linking and maintaining globally*) — supported by clear governance rules for development and maintenance. These governance rules should include the responsibilities of consumers and producers.

The remainder of this section details on the two main perspectives of software reuse in practice and reports impact factors affecting the adoption and execution of reuse strategies.

2.4.1 Performing software reuse in practice

When performing software reuse in practice, two main perspectives need to be considered: developing *with reuse* and developing *for reuse*.

Developing with reuse The process of developing *with reuse* can be defined in terms of the constituting actions, as detailed in the taxonomy in Figure 2.3. Starting top-down, the taxonomy reads as follows: When attempting to *reuse*, a developer needs to *retrieve* and *modify* a reusable item. To retrieve candidate reusables, they must first *identify* them, then *evaluate* their suitability for the purpose, and *select* the best fit. Before identifying candidate reusables, a developer needs to *characterize* formally or informally the requirements the potential reusable needs to meet and to *match* existing reusables against this specification.

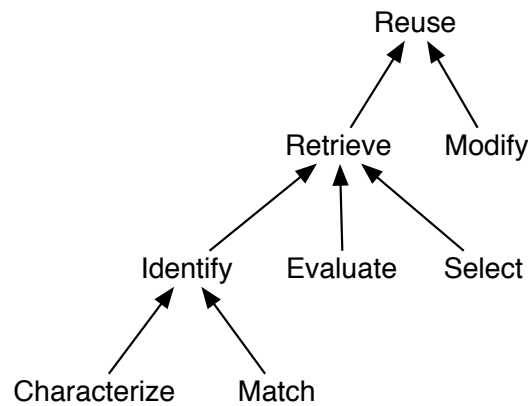


Figure 2.3: Taxonomic definition of Software Reuse from the perspective of the reuse consumer based on [44, 45].

Individual cost-benefit estimation Effecting the actions related to reuse incurs a cognitive effort on the part of the person attempting to reuse. The intuitive evaluation of the potential gain of reuse versus the estimated effort required to effect reuse impacts the decision for or against reuse in a given situation. In this evaluation, however, short time benefits, e.g., faster creation of suitable artefacts, tend to be valued higher than long term costs, e.g., for maintenance, of newly created (and potentially redundant) artefacts. If an organization strives to increase reuse, it should, therefore, invest into enabling factors that decrease the effort required to reuse, such as, e.g., a supporting infrastructure, an adequate communication and development culture, and a shared vision of reuse³.

Developing for reuse When developing *for reuse*, i.e., producing artefacts for the purpose of reuse, an analogous evaluation takes place: the cost of creating an artefact that is of sufficient quality and generality to be reusable for many usually incurs an up-front investment that is higher than creating single solutions for every use. However, this cost amortizes with the number of reuse instances as well as with the saved effort in terms of maintenance. The number of potential consumers as well as the life expectancy of a reusable artefact and its embedding systems are, therefore, key factors to be considered.

Global cost-benefit estimation Predicting which artefacts provide enough benefit to an organization to compensate for the additional effort required for reuse is challenging in the face of emerging requirements and diverging needs of organizational units. The estimate of the effort for producing the respective artefact, as well as creating an environment that enables reuse, needs to be related to a qualified estimate of the cost incurred by neglecting reuse opportunities. Depending on the current organizational context, choosing an ad-hoc position on software reuse that incurs costs at a later point in time (e.g., choosing *clone-and-own* over a structured approach) might be acceptable for business reasons. In another context, deferring costs for inadequate reuse

³For details, see Chapters 3, 6, and 7.

to a later stage of the product life cycle might be detrimental (e.g., if extensive maintenance is costly). In this case, the comparatively higher initial cost of introducing a more advanced reuse approach might be considered acceptable. Adequately weighing the alternatives available with respect to reuse can be challenging and requires a variety of skills and knowledge from an organization, ranging from development to management practices.

2.4.2 Success factors and enablers

Several works have proposed processes for, or reported on, adoption attempts of structured reuse in practice and deduced respective success and failure factors. The following paragraphs give a brief overview on a selection of studies, starting from the 1990's up to 2014.

2.4.2.1 Overview of studies

Basili [78] connects the topics of maintenance and reuse and, in a synthesis of previous research efforts on reuse, derives a theoretical reuse framework. He proposes three reuse-oriented maintenance models and lists three enabling factors: first, an *improvement paradigm* that models organizational learning for process and product improvement, composed of planning, analysis, learning and feedback tasks. Second, introducing or improving reuse requires a *reuse-oriented environment* that supports the necessary alterations to anchor the reuse models in the development processes. Last, *automated support* for artefact management and reuse measurement.

Joos [84] reports on the experience of introducing a systematic reuse process at Motorola in the 1990s that succeeded due to *management support, education of engineers, suitable incentives, and tool support*. The paper provides a rich account on the difficulties encountered as well as the successful solutions for each of them.

Frakes and Fox [1] identify four dimensions (managerial, economic, legal, technical) that need to be addressed when implementing systematic reuse. They find that a *holistic understanding*, as well as an adequate strategy to address the respective dimensions, contribute to reuse success.

Rine [82] statistically analysed the results of a qualitative survey on software reuse to determine predictors for reuse success as well as to establish the relationships between reuse capability, productivity, quality and these predictors. The data is drawn from a questionnaire and interview survey with companies from different domains more than half of the studied companies work in real-time or embedded software production. Subsequently, he uses the results for a theoretical validation. His main result is that

"[S]ome of the success factors [...] have a predictive relationship to software reuse capability. Software reuse capability also had a predictive relationship to productivity and quality." [82] — D. Rine, 1997

Rine proposes the following as the lead indicators of software reuse capability:

"[A] product-line approach, architecture which standardizes interfaces and data formats, common software architecture across the product-line, design for manufacturing approach, domain engineering, management which understands reuse issues, software

reuse advocate(s) in senior management, state-of-the-art tools and methods, precedence of reusing high level software artifacts such as requirements and design versus just code reuse, and trace end-user requirements to the components (systems, subsystems, and/or software modules) which support them. [...] The three most significant non-predictors of software reuse capability are: taking the library, salvage, or junk yard approach to software reuse, the effectiveness and efficiency of the software library or repository, and certification of components to some quality level(s)."[82] — D. Rine, 1997

Whilst many of the predictors are widely recognized as factors enabling different styles of successful reuse, this, as well as subsequent work leading to a reuse reference model [85], do not take into account the highly differentiated goals and contexts of companies. It therefore claims that by complying with the reuse reference model practitioners can obtain high reuse rates and potentially find orientation for improvement. Nevertheless, the initial adoption hindrances remain untouched and so do the assumptions about feasibility and risks of introducing a heavy-weight reuse approach. In addition, library based reuse is discarded too easily: first, it is assumed to happen without any domain analysis and therefore as likely missing the point of providing useful functionality. This, however, is a factor of similar effect for SPL development so it can not be used to discard the approach. Furthermore, the survey data stems from a time in which neither search nor retrieval infrastructure was very developed, causing repositories to be infeasible due to technical limitations. Therefore, these conclusions should be re-assessed with newer studies.

Fichman and Kemerer [18] assess 15 projects within a software company for the reasons that caused reuse adoption processes to shrivel. They identify incompatible incentive structures that valued the success of single projects over contributions to company-wide reuse. Consequentially, the authors propose to establish *incentive-compatible* programs for systematic reuse including effective cost management for the case of reuse failure. In addition, they propose the role of a *reuse curator*.

Morisio et al. [13] report on success factors for adopting or running company-wide reuse programs. They collected quantitative evidence from 24 reuse projects in European companies varying in size, business domain and culture. The authors conclude that success of reuse projects depend on *management commitment, awareness of human factors* and *modification of non-reuse processes* according to the specific context of the company.

Slyngstad et al. [79] conduct a qualitative empirical survey with 16 developers of the IT department of a Norwegian oil company. The authors conclude that the main reuse facilitator was the existence of *component information repositories*. On the contrary, education and experience did not have an impact on the study subject's inclination to reuse.

Lucredio et al. [80] study reuse in the Brazilian software industry by means of a survey with 57 participants from 56 companies that aims to relate characteristics of software development organizations with successful reuse adoption. In their attempt to determine which development and context factors have influence on software reuse success, they align with Rine [82]. The authors assess the impact of twenty-one, divided into four perspectives: organizational factors, business factors, technological factors and processes factors. Assessment is done by relating the

presence or absence of factors to benefits or drawbacks experienced by practitioners introducing or effecting reuse. Of the respondents, 53% described their reuse as successful, i.e. leading to success in their software projects. The authors report their results for three categories of companies: for small organizations, *experience of developers, application domain, tool support, quality assurance, and a systematic reuse process* impacted reuse success. In medium size and large organizations, the application domain was less significant. Reuse was most likely to be successful if it was *institutionalized by means of a dedicated team, developing product families by means of a systematic reuse process, reusing all kinds of available artefacts*. This was *supported by tools for development and configuration management*. In addition, reuse in large organizations benefited from adopting a *quality model for artefacts*. All organization types experienced better success with reuse when being able to draw on *existing artefacts* (as opposed to building them explicitly for reuse).

In a quantitative survey study on software reuse in Open Source development, Sojer and Henkel [52] conclude that *personal networks* as well as *exposure to a variety of projects* lead to developers reusing more code as these factors helped them to discover and access the respective reusables. In addition, the *personal conviction of the benefits* provided by reuse (e.g. the possibility to quickly build a working prototype) incited developers to rely on reuse.

Varnell et al. [81] present a study comparing reuse practices in embedded and non-embedded software development in the aerospace domain. They report that in embedded systems development, component-based approaches prevail, followed by product lines, whilst in non-embedded contexts ad-hoc reuse is most frequent, with component-based approaches and product lines being equally important. For either type of development, participants reported significant *savings in labour hours* due to reuse and the benefit of *risk reduction*. However, a decrease in the number of defects could not be confirmed.

2.4.2.2 Technical success factors and enablers

Technical infrastructure: The presence of an adequate technical infrastructure is reported as a key reuse enabler [79, 80] or even prerequisite [9] to conduct software reuse. Infrastructure ranges from repositories for code and the respective documentation, to support for development, quality assurance, configuration management, and deployment. Particularly, the ease of access to reusable entities provided by tool support facilitates reuse [52, 60].

2.4.2.3 Organizational and social success factors and enablers

Incentives: In an Open Source context, the *personal conviction of the benefits* provided by reuse incited developers to rely on reuse [52].

Congruence: Empirical evidence suggests that the congruence of reuse goals and the selected reuse approach with organizational context factors can significantly improve the success of reuse adoption and practice [83].

Knowledge: *personal networks* as well as *exposure to a variety of projects* reportedly enable Open Source developers to reuse more code as these factors helped them to discover and access the

respective reusables [52]. In a closed source environment, *experience of developers* is reported as success factor in some studies (e.g. [80]), whilst it did not impact reuse success in others (e.g. [79]).

Management: Various sources suggest that sustained management commitment is a key enabler for any advanced reuse program [13, 80, 23]. In particular, this management commitment is needed to drive the *modification of non-reuse processes* as well as to create the *awareness of human factors*, e.g. changes in responsibility [23] and adaptation to new processes [13], that impact organizational change (and address them accordingly).

Process: In several studies, the relevance of a systematic reuse process is reported as success factor [80, 82]. Tailoring of non-reuse processes is reported as enabler [13].

Organizations structure: For medium size and large companies, institutionalized reuse by means of a dedicated workforce is reported as success factor [80, 8].

Artefacts: Reuse of artefacts other than code, as well as reuse of already existing artefacts are reported as enabling successful reuse [80, 82].

Quality assurance: High quality of reusable artefacts is a key reuse success factor as it establishes trust with the users [23]. To achieve this, adequate methods should be adopted (e.g. quality models [80] or code reviews [9]).

2.4.3 Challenges and inhibitors

“Although the benefits can be substantial, software reuse has never reached its full potential. Organizations are not aware of the different levels of reuse or do not know how to address reuse issues.”[53] — W. Spoelstra, 2011

After decades of theoretical and applied research on reuse, researchers concede that reuse has not always met the given expectations [52, 53]. Besides technical challenges that need to be addressed, a substantial number of organizational and human factors have been identified as potential inhibitor to a successful application of advanced reuse practices [13]. Technical factors include creation, retrieval, modification, and maintenance of reusables. However, these topics are strongly linked to human factors, such as cognitive effort [2], program understanding, and motivation, as well as organizational factors, such as business strategy, management commitment, and company culture [86].

Frakes and Fox [1] summarize many of the above mentioned topics in a reuse failure mode model, derived from a questionnaire on (code) reuse answered by 113 people from 29 organizations working in different domains. The authors mention four dimensions (managerial, economic, legal, technical) that need to be addressed when implementing systematic reuse.

The authors derive seven failure modes (no attempt to reuse, part does not exist, part not available, part not found, part not understood, part not valid, part not integrable). According to the authors, the most frequent failure modes are “no attempt”, “not integrable”, “not understood”, while “not found” or “not available” are the least important. In Figure 2.4, we map the failure modes onto the taxonomical definition on reuse proposed by Basili and Rombach [44].

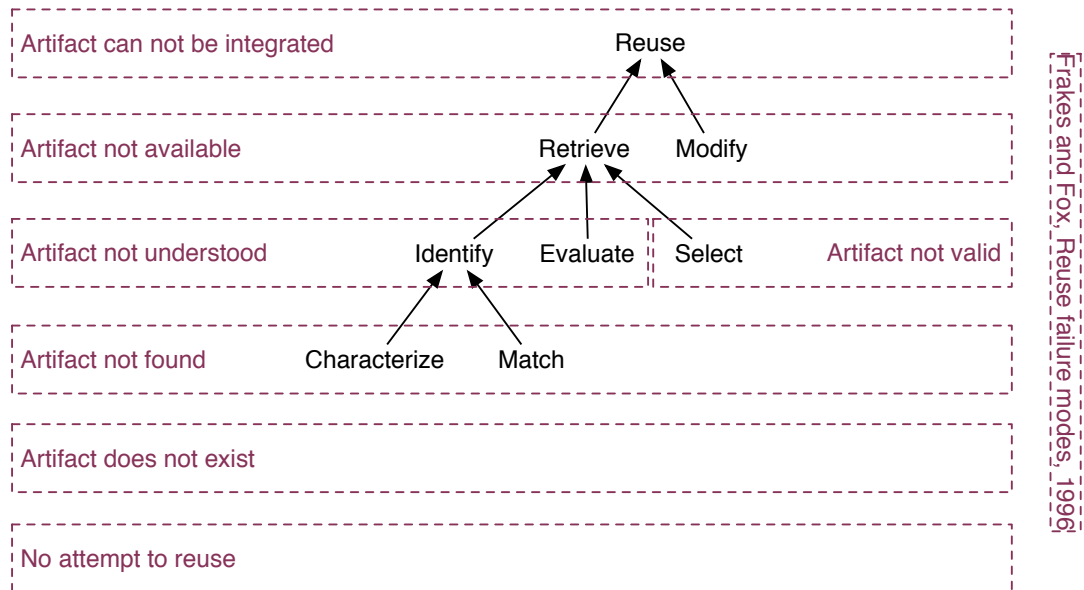


Figure 2.4: Mapping of Frakes and Fox reuse failure modes on the impacted activities of the reuse process.

The figure highlights that, in fact, the failure modes cover all activities in the reuse process, i.e., every activity of this process can be hampered by the occurrence of different failure modes. The modes relate to human (e.g. *no attempt to reuse*) and technical factors (e.g. *artifact can not be integrated*), which often can be due to organizational factors, such as contradictory incentives.

In the following, we detail on the challenges most frequently reported in literature:

2.4.3.1 Technical challenges and inhibitors

Creation and design of reusables: The creation of reusables can be challenging due to several factors. First, determining what reusables should be built by design is non-trivial. It requires a detailed understanding of the envisioned application context to reduce friction when integrating reusables [87]⁴. Second, providers must strike a critical balance: on the one hand, a reusable should encapsulate a specific functionality in order to be coherent, understandable, and clearly fit a defined task. On the other hand, it should be as generic as possible to allow being reused in numerous different contexts with little adaptation effort [46].

Technical incompatibility is a strong inhibitor to reuse, denoting problems of interoperability due to incompatible platforms, paradigms, and technologies [49, 88, 59]. Technical incompatibilities can decrease or annihilate the possibilities of extracting or combining existing parts [89].

⁴According to Greenfield et al. [87], this lack of knowledge about the final context is an enormous challenge when providing reusables that are consumed in an ad-hoc way.

Storage and retrieval of reusables: Companies aiming for internal reuse repositories are still facing the challenge of populating and classifying them, which requires a considerable upfront investment and often proves infeasible [12, 25].

Also in the context of Open Source software, challenges in retrieval of reusable entities are one core inhibitor to successful and widespread adoption of reuse in practice [46, 88, 68, 59]. Whilst originally the challenge of retrieval lay in locating and accessing catalogues of reusable entities [46], it is nowadays the number of potential reusable entities that challenges developers aiming to reuse [90, 91].

Technical Infrastructure: On a technical level, the development infrastructure used by a company can significantly impact the way reuse can be approached [9, 25]: the absence of a supporting infrastructure de-facto renders structured reuse impossible, as it hinders developers to access and retrieve reusables in a coordinated and controlled way [60]. Furthermore, advanced infrastructures can mitigate the risk of errors introduced into reusables [27].

2.4.3.2 Organizational challenges and inhibitors

Many of the technical challenges mentioned above are challenging on the conceptual level. However, they tend to be exacerbated due to the organizational context that embeds them. The following organizational hindrances are particularly prominent in literature:

Organizations structure: the quality of inter-unit relationships has a significant impact on a successful outcome of reuse adoption. Competition, overlapping or unclear responsibilities, priority conflicts, and lack of coordination of reuse activities diminish the likelihood of success of a reuse program [29].

Inertia: product-centric organizations tend to promote a focused view on development. Managers and developers are usually assessed based on the success of their isolated projects, incentivising local optimization that counteract reuse on a company-wide scale [87, 18, 29].

Knowledge: Adoption of advanced reuse is a global topic that requires a clear positioning of the organization [70] and research into current methods and techniques for reuse (which tends to be neglected [29]).

Measurement: Introducing central reuse requires significant resources and collaboration across different organizational units. Without measurement and adequate compensation, this might lead to unwillingness to cooperate [29].

Management: Introducing the required governance strategies for creation, maintenance, and decommission of reusable items can be challenging in the face of heterogeneous preferences and process weaknesses [87]. Adjusting the context, thus, causes additional overhead that tends to be underestimated in the initial planning [29, 13], endangering reuse success.

Economic: Investing into the reusability of software or supporting infrastructure imposes non-negligible costs onto projects and requires firm and long-term support from management to resolve restrictive resource constraints [29, 84, 82].

Disincentives: One of the strongest disincentives is lack of quality of the entities provided for reuse [29]. This inhibitor needs to be overcome by means of transparent quality assurance and

clear governance lining out assumptions and guarantees that hold for entities designed for reuse. In addition, the criteria that are applied to assess developers and managers have an impact on their motivation to engage into reuse [18, 9].

In addition, the *cognitive distance* [2], i.e., the challenge to understand in detail the artefacts others produced, that needs to be overcome increases with the type of artefact that is reused. The *cognitive load* of artefact understanding is further impacted by the quality of the reusable.

Legal constraints and risks: Incompatibilities of licenses as well as liability issues have reportedly inhibited incorporating potential reusables of third parties [92, 27, 25]. In addition, the potential risks presented by unstable providers can exceed the perceived usefulness of the artefacts offered [34].

2.4.3.3 Social challenges and inhibitors

The, arguably, most famous challenge to reuse on a social level is the “not invented here” syndrome [87]. It encompasses a number of aspects that lead to resistance against reusing entities that have been provided by other parties. These aspects can be related to trust and transparency, e.g. doubts about the precise functionality of a reusable, the process under which it was produced and its quality (and the quality of the respective documentation) [79], the reluctance to take on dependencies on other parties, as well as to the (sometimes significant) cognitive effort of understanding reusables provided by others⁵ or the need for compromise in terms of functionality.

In addition, personal preferences for specific solutions paired with “engineering pride”, the desire to find beautiful or better solutions to challenging technical problems, can motivate individuals to avoid reuse. This tendency, is present in industrial software development, as well as in Open Source communities [52], and can be strongly exacerbated in contexts in which the lack of supporting infrastructure increases the cost of finding and accessing reusables.

Some reuse approaches (component based) promote a certain presence of guided redundancy to allow for exchangeability of components depending on specific system requirements, so the creation of new solutions could be beneficial in this case. However, the resulting reusables need to clearly communicate to which requirements they correspond; a factor that often-times is neglected and, thus, causes assumptions about the given application context to remain implicit.

Summary Challenges and Inhibitors As the studies presented above show, conducting reuse is by no means trivial. Challenges are present on technical as well as organizational levels. To address the issues relevant for a particular reuse application, it is advisable to thoroughly analyse and understand the given goals and their context.

2.4.4 Reuse approaches in practice

The following paragraphs briefly introduce reuse approaches that have reportedly been adopted in practice.

⁵Understanding a reusable in this case encompasses: understanding the exact functionality, the assumptions about the application context, the conformance to non-functional requirements, etc.

Clone-and-Own is the most frequent realization of pragmatic reuse and denotes a reuse approach that relies on copying and, potentially, modifying of (proven) solutions for the purpose of effective development. It is also known as: *code scavenging* [2], *ad hoc reuse* [55], *opportunistic reuse* [57], and *copy-and-paste* (or *cut-and-paste* or *copy-and-modify*) reuse [58], *pragmatic reuse* [59]. As it has virtually no preconditions on the organizational context, it is applied widely in industry [60]. Depending on the given industrial context, this practice can serve as a disciplined tool [93, 61, 60]. On other occasions, clone-and-own is the only feasible reuse mechanism at disposal due to, e.g., organizational restrictions or absence of supporting technology. However, they incur the risk of inducing errors as well as significantly increasing maintenance efforts [63]. On the conceptual level, the task of finding working code examples among the vast amount of available source code can be a time-consuming challenge [94].

Inner Source As empirical studies have shown, Open Source projects heavily build on code reuse on the basis of libraries, reaching reuse rates between 30 and 90% [52, 22]. Open Source development relies on transparency, self-selection of tasks, asynchronous communication, and quality assurance. *Inner Source*⁶ attempts to transfer this reuse-inducing⁷ development style from the Open Source community to industry [23]⁸. The key benefits of *Inner Source* lie in the full access of developers to the seed project's source code and the shared responsibility for reusable assets. This transparency and availability serve as a key enabler for reuse. Literature reports instances of successful *Inner Source* e.g. at Hewlett-Packard, Alcatel-Lucent, Philips Healthcare, IBM, and SAP. Pointers to the respective material and a summary of the studies are provided in [9].

Component-based reuse aims to build software systems out of interchangeable components [98], potentially provided by third-party vendors [99], enabled by separation of implementation and interfaces, with a possibility of extension via well-defined extension points [99].

Adoptions of *component-based* reuse have been reported [100], particularly in the domains of embedded software development [101]; however, most of the proposed methods and techniques designed to support the approach are currently lacking validation of benefits and accounts of application in practice [56, 102, 103].

Service Oriented Architectures (SOA) can be seen as a related to component-based reuse in the sense that services and components aim to encapsulate clearly defined functionality, usable in a black-box fashion via explicit interfaces, enabling composition of systems and the exchange of components or services [104]. However, services are distinct from components in their focus on business-relevant functionality delivered at a higher granularity, whilst components tend to provide more technically relevant functionality [105, 106].

⁶Research on the topic dates back only to 2002 and has not yet been very extensive. However, in recent years, the topic has attracted more attention from the research community. [9]

⁷One of the key goals of *Inner Source* is reuse. However, it comprises more than just mechanisms for reuse. In addition, its principles and development practices, as well as the advanced tool support, clearly enable code-based reuse.

⁸Literature knows the phenomenon also as *Progressive Open Source* and *Controlled Source* [95], *Corporate Source* [96], *Corporate Open Source* [97], and *Internal Open Source* [96].

Software Product Lines (SPLs) aim to "reduce the overall engineering effort required to produce a collection of similar systems by capitalizing on the commonality among the systems and by formally managing the variation among the systems" [107]. Implemented successfully, SPLs reportedly lead to very high reuse rates and enable rapid creation and delivery of new product variants [8]. However, adopting a product line approach demands significant maturity of an organization's process and development capabilities. Commercially successful product line implementations are showcased in the *Product Line Hall of Fame*⁹.

2.5 Position of this thesis

As lined out above, literature offers a wide variety of definitions of and around software reuse. In this thesis, we position ourselves as follows:

For the definition of **software reuse**, we follow Frakes and Fox and define it as

"the use of existing engineering knowledge and artefacts to build new software systems." [1] — W. Frakes and C. Fox, 1996

However, with respect to **reusable entities**, we only consider work products that originate in the software development cycle. This, particularly, means that non-persisted (e.g., documented or formalized) *knowledge* and *experience* are in general not considered *reusables*.

We refer to the roles involved in the creation and integration of reusables as follows: individuals or entities providing reusables are called **producers**, whilst individuals or entities relying on reusables for construction of their systems are called **consumers**.

This work does not promote any particular reuse approach; on the contrary, it is committed to study the circumstances in which a specific approach can be applied with benefit in practice. As a consequence, this work, in a first step, attempts to provide a deeper understanding on software reuse through its empirical studies of software reuse in practice. In a second step, it aims to support practitioners in understanding the options different reuse approaches provide for their context, and in choosing accordingly.

⁹<http://www.splc.net/fame.html>

Part II

Evaluating the state of practice of reuse

3 | Case studies on reuse in industrial practice

Anecdotal evidence as well as empirical studies indicate that, despite years of research on many aspects of the phenomenon, adopting and practising reuse remains challenging (see Chapter 2). Unfortunately, little is known about which proposed practices and approaches have been successfully transferred to practice, adopted and proven useful, how reuse is in fact effected in current practice, and what are currently the main challenges.

This chapter presents the overall design of two empirical studies that have been conducted to capture the experience of practitioners in industrial software development and maintenance with respect to reuse. Parts of this chapter were published in [25, 27, 41].

Contents

3.1	Empirical studies on software reuse in practice	36
3.2	Methodology	36
3.3	Case description G	37
3.4	Case description U	38
3.5	Original case study designs	39
3.6	Data collection & analysis procedures	40
3.7	Company reuse placement	40

3.1 Empirical studies on software reuse in practice

Reuse is essential for industrial software development and therefore widely applied. However, anecdotal evidence as well as empirical studies, e.g. [60, 13], indicate that, despite years of research on many aspects of the phenomenon (see Chapter 2), reuse is not yet a solved problem. Unfortunately, little is known about which proposed practices and approaches have been transferred to practice, adopted (and possibly proven useful), how reuse is in fact effected in practice, and what are currently the main challenges. This, however, is a necessary precondition for evaluating the practical success and the impact of research in the field of reuse, as well as for choosing and adapting the focus of future research to practitioners' needs.

To mitigate the mentioned issue, we conducted two empirical case studies (C1 and C2, presented in Chapters 4 and 5) on reuse in practice that were designed to complement each other in terms of the characteristics of the participating companies. Furthermore, we integrate the results of the two cases in a study (CI) presented in Chapter 6.

Common to the three Chapters is the goal of developing a more comprehensive picture of the circumstances under and forms in which reuse is conducted and how it is managed. We attempt, on the one hand, to establish the goals and motivations that drive the decisions for or against specific reuse strategies, processes, and policies. On the other hand, we compare the reuse decisions to their technical realization in terms of reusables and reuse approaches and observe the effects. Additionally, we capture the organizational factors that could influence the way reuse is effected in an organization. We capture the aspects of our goal in two conceptual blocks: understanding the way reuse is effected, and identifying challenges and opportunities of reuse that occur in practice.

Outline This chapter first presents the studies and their context, and then characterizes the participating companies. Parts of this chapter were published in [27, 25, 41].

3.2 Methodology

This section introduces the two cases that we subject to comparison in the present study. The two companies under study are named U and G. We detail their context and outline the respective case study designs previously conducted and their main results. Table 3.1 summarises the context details of both companies¹.

Both studies featured an extensive on-line questionnaire containing mainly closed-questions and a number of semi-structured 1-2 hours interviews always conducted by two researchers (one conducting the interview, one scribe). The complete original questionnaires for both cases are available in [41]. The interview guide is included for reference in the Appendix, Table 12.1.

We conducted each study within a period of 3-4 months. Figure 3.1 shows the relationship between the studies: The study at G (conducted in 2013) preceded the study at U (conducted

¹Please note that factors in the categories *development context* and *reuse characteristics* reflect tendencies and the state of the companies at the moment of the studies. Both companies continuously strive to improve their craft, so this table does not necessarily reflect their situation at the time of reading.

in 2015) and influenced its research design.² The integration of the two studies was performed on completion of case U.

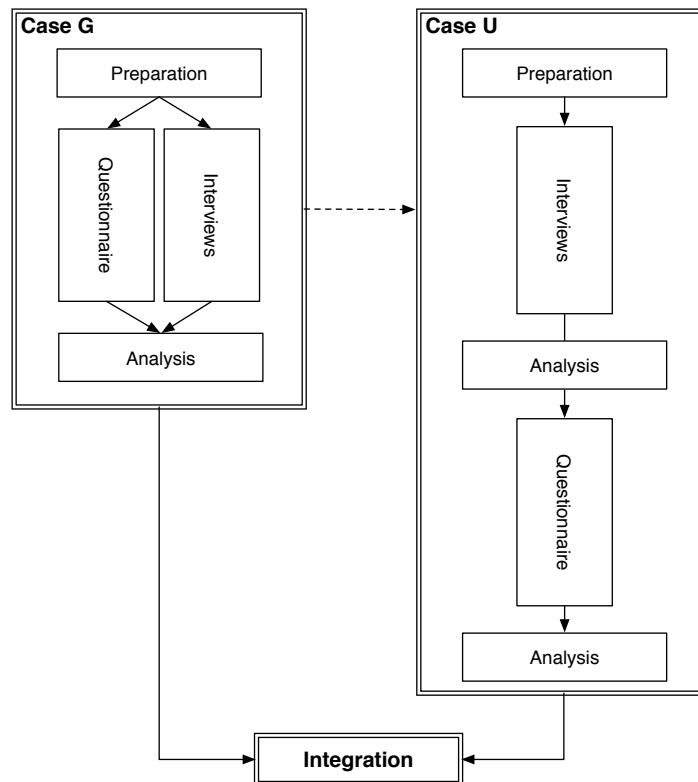


Figure 3.1: Study setup: case G, case U, and case integration.

3.3 Case description G

Company G is a multinational corporation, specialised on Internet-related services and products. The company structure supports flat hierarchies and multi-project assignments for engineers. Development follows a homogeneous process with advanced tool support centred around collective code ownership and agile practices [108]. Developers at G work on multiple projects at the same time, they are organised in small teams, and develop software with several programming languages (mainly C++, Java, Python, and JavaScript). Reuse is mandated for a small set of utility functionalities; however, reusing existing code in an adequate way is considered best practice and fostered by the development style and organizational incentives. The reuse goals for the company are faster development of new features, lower maintenance costs and consistency.

²Experience from the first study at G lead us to first conduct the interviews in case U in order to focus the questionnaire to the most relevant parts for U.

Study demographics We interviewed 10 engineers and collected 39 responses to a 45-minute on-line questionnaire. The participants originated from more than 25 different teams distributed worldwide and held varying organisational roles (developers, maintainers, managers, as well as any combination of the three roles). Their experience ranged from <1 to 20+ years in their current role (time at the company: <1 to 10+ years). By means of qualitative data analysis, we extracted the context of reuse, involving roles, responsibilities, and reuse practices, i.e. reused artefacts and reuse realizations. We collected current issues, success factors, and ideas for improvement.

3.4 Case description U

U is a national software producing company providing technical information services and business information products to their clients. The company was founded in the 1960s, emerged as a service provider and gradually moved to providing stand-alone software products and services. Currently, U counts around 6000 employees. The company structure is hierarchical, structured along market segments. The products have historically grown over decades and contain a broad mix of technologies. Software development is very heterogeneous across departments and teams, ranging from waterfall processes to tailored Scrum approaches. Also the level of development tool support, testing practices, and code ownership is highly diverse. As a result, products are integrated on a binary level. Developers usually work on specialized topics of a single product and tend to be responsible for the respective subsystems (Subsystem code ownership, see [108]). Reuse is mandated for an internal utility platform providing domain-independent functionality to products. The company's reuse goals are: consistent extension of the .NET framework, consistent integration of existing products, lower maintenance costs.

Study demographics

We study the current practice of reuse at U by means of an exploratory study consisting of an interview phase with 20 participants, followed by questionnaire phase with 69 respondents. We report on the state of practice of reuse, comprising success factors, challenges and ideas for improvement.

We drew interview participants from each of U's product and support development departments and all levels of the hierarchy. The participants worked at U between 15 and more than 30 years. Even though the company is mainly based in one area, the teams are distributed. For the data collection and analysis, we proceeded as for case G.

Questionnaire participants were invited by a newsletter and a post on a company news portal. Respondents came from 10 of the 13 departments. 44% worked at U for at most 10 years, 20% for 11-20 years, and 36% for more than 20 years. 15% reported their role as manager. The respondents' job focus was mainly on development (78%), and architecture (13%). Respondents at U usually work within one product area and are organised in product departments over several hierarchical units. They are developing software most frequently in C# and SQL. In addition, they use Java and C++.

3.5 Original case study designs

In preparation of the first case study, we conducted a literature review to derive the original concepts for interview guides and questionnaire. To meet our research objective, we opted for a combination of interviews and questionnaire as they compensate each other's weaknesses: interviews provided us with a highly detailed account on reuse practices, highlighting particularities of the company context, as well as raising new ideas and concerns. However, they were expensive and time-intensive to conduct for both parties. Therefore, we chose to complement the interviews with an on-line questionnaire that was designed to capture responses from a wide range of participants.

Before rolling out the study in either of the cases, we piloted and revised the interview guide and questionnaires to remove ambiguities, increase understandability, and, in case of the questionnaires, to ensure technical performance.

Semi-structured interviews We conducted semi-structured interviews with developers, maintainers, and managers at G and U. In case G, our interview guideline was based on a pre-study with a scope similar to the questionnaire. In U, we added company specific aspects (e.g., related to the development and release practices) to the interview guideline and iteratively refined it during the course of the study to accommodate new aspects impacting U's reuse practices (e.g., the cultural heterogeneity of the different development units).

In both cases the aim of the interviews was to obtain detailed insights into reuse application in different development teams and projects, as well as its implications regarding non-technical aspects such as company culture and interpersonal skills. We, therefore, selected participants from different departments of both companies. Each interview lasted between one and 2 hours and was conducted by two researchers, one leading the conversation with the participant while the other created the transcript and asked clarification questions.

Online questionnaire To gain a comprehensive overview of reuse at the companies, we developed an on-line questionnaire for each case³.

For each reuse aspect, several multiple choice questions were asked. Furthermore, we invited the participants to contribute additional information in the form of free text. We asked the participants to provide their main job focus, their level of experience in their current role, the time spent working at the company and the type of project they were working on. Taking part in the questionnaire took approximately 30–40 minutes. Participation was optional.

³When conducting the first study, we noticed technical limitations of the platform and room for improvement in the resolution of our scales. After conducting the first study, we improved the questionnaire for the next by migrating to a professional platform and adjusting the resolution of the scales. In addition, we had to take into account the different company contexts and philosophies. Lastly, we incorporated results, such as success factors or challenges, from the first study.

3.6 Data collection & analysis procedures

After performing the interviews, we processed the transcripts by applying techniques from grounded theory, which support inductive content analysis. To extract the important information, we coded the transcripts⁴ twice: first, we went through a phase of initial coding [109] to separate the transcripts into statements, assign them with codes, and triage them to focus on the ones relevant to reuse in practice. Based on the relevant codes, we build up emergent categories. In another, focused, round of coding [109], we pruned the categories to the most significant ones and created relationships between them. The coding process resulted in clusters of categories connected with each other, containing the relevant statements. In the case of U, we set out with a collection of potential codes obtained from the study at G. However, we adapted and pruned the collection to accommodate new information related to the new organizational context⁵.

3.7 Company reuse placement

Given their reuse context, the companies clearly differ in their reuse capabilities. Based on the Company Placement provided by the Reuse Adoption Support Model (RASM) in Chapter 7, company G scores in the category *advanced* reuse capabilities, while company U scores in the category *basic* reuse capabilities⁶. The lessons that can be drawn from the presented cases are, thus, of different nature.

⁴Coding means “categorising segments of data with a short name that simultaneously summarises and accounts for each piece of data”. [109]

⁵For instance, the roles present at the companies differed noticeably. Also the notions of teams, products and projects required a mapping between the cases.

⁶The available categories are *basic*, *intermediate*, *advanced*.

Table 3.1: Characterization of the participating companies

	Company U	Company G [27]
Company settings		
Established	1960s	1990s
Overall staff*	~6000	~40000
Software staff*	~1000	>2000
Software production*	client product portfolio	online product portfolio
Application domain*	business information systems	online services
Type of software*	business	business
Organisation of development units	hierarchical, strong separation in departments	flat hierarchies, peer-driven, interconnected
Scope	national	international
Development context		
External requirements for release cycles	yes	no
Development style	heterogeneous	homogeneous
Code ownership	strong	collective
Code reviews	rarely	mandated
Development infrastructure	local	central
Source code repositories	several local ones	one central
Product assembly	binary integration	continuous source integration
Developer focus	dedicated aspects of single products	multiple aspects of multiple projects
Staff experience of sample*	high	medium
Reuse characteristics		
Reuse approach*	ad-hoc (loose*) in transition to structured (tight*)	tool-supported ad-hoc (loose*)
Current reuse scope	department	company
Global requirements engineering for reuse	limited, grass-root	limited, tool-based
Global incentives for reuse	no	yes
Co-ordination of reuse	within department	on-demand
Co-ordination overhead for reuse	significant	low
Reuse consumer**	within department	all
Reuse producer**	within department	all
Pool of available artefacts for reuse	limited	significant
Dedicated personnel for reuse	yes for basic, domain independent functionality	yes for basic, domain independent functionality
Reuse tool support	low	advanced
Accessibility of reusable artefacts	mixed	good
Formal reuse assessment	no	no
Motivation for reuse	high	high
Satisfaction with current reuse benefits	mixed	positive
Study data		
Total number participants	89	49
Participant average time in company	11-20 years	1-3 years

* adapted from [13], ** adapted from [3].

Table 3.2: The empirical studies in numbers

Study code	# companies	I participants	I duration	Q invited	Q answers	Q duration
C1	1	10	1	600	39	40 minutes
C2	1	20	1-2	275	69	30 minutes

4 | An exploratory case study of software reuse at Google

This chapter presents an exploratory case study conducted at Google. It was motivated by the question of how one relatively young, yet large and renowned, software company tackles the topic of software reuse. The case study relies on interviews and an online questionnaire to capture the way reuse is effected on a daily basis at Google. Our results indicate that in the presence of a homogeneous and quality-conscious development culture, supported by advanced tool support, software reuse in an Inner Source style can be practiced in a large scale. Furthermore, this reuse approach aligns well with the company context and the strategic needs. Therefore, is considered as beneficial by the participants. Parts of this chapter are published in [27].

Contents

4.1	Study goal and context	44
4.2	Methodology	46
4.3	Study results	47
4.4	Discussion	57
4.5	Threats to validity	58
4.6	Considerations for practitioners	59
4.7	Summary and conclusions	60

This section presents an exploratory study of software reuse at Google. It was conducted in the time from September 2012 to September 2013. The goal of the study was to shed light on the current practice of reuse at Google. We study the current practice of reuse at Google by means of an online questionnaire with 39 participants and interviews with 10 participants. We report on the state of practice of reuse, comprising success factors, challenges and ideas for improvement. Based on our results, we provide a list of considerations for implementing reuse, prerequisites and open issues.

We found that reuse was code-centered and effected mostly on demand. Depending on the provenance of the reusable entities, reuse was addressed as follows: Reuse of third-party code was subject to strict regulations and processes, whilst for reuse of internal code opportunity-driven behaviour was predominant¹. Key enabler for success with this implementation of reuse were the high quality of the source code and the supporting infrastructure.

While this case study shows that state-of-the-art development infrastructure can significantly mitigate the disadvantages of opportunistic code reuse, it also highlights remaining challenges: in the face of a daunting amount of source code, finding and identifying suitable candidates for reuse remains difficult and can incite rewriting of functionality from scratch. Furthermore, the web of dependencies created by opportunistic reuse was perceived as a noticeable downside. Overall, the disadvantages experienced due to opportunistic reuse were deemed acceptable by participants, given the concrete benefits of faster development pace and decreased maintenance effort they experienced.

For the following aspects of reuse, participants desired improvements: detecting and preventing redundant implementations of functionality, advanced change propagation during maintenance of reusable entities, as well as a framework to assess the adequacy of reuse instances.

Outline The study is structured as followed: Section 4.1 describes our study and Section 4.3 summarizes our results. In Sections 4.4 we discuss our findings and detail on threats to validity in Section 4.5. Section 4.6 lists our considerations for practitioners before Section 4.7 concludes the chapter.

4.1 Study goal and context

This section first describes the context and the goal of our study. Then it details on the methodology and data collection procedures.

4.1.1 Study goal and research questions

Following the GQM goal template [110], we formalize the goal of this study as follows:

The goal of this study is to analyze *reuse practices* for the purpose of *characterization and understanding* with respect to their *effectiveness* from the viewpoint of *software development professionals* in the context of *Google*.

¹An exception to this are so-called *core libraries* that cover essential functionality for most products and are globally mandated for reuse.

From this overall goal, we derive the following subgoals: *understanding current reuse practices* and *identifying challenges and opportunities* occurring in practice. We address our subgoals by the following research questions.

4.1.1.1 Understanding current reuse practices

This category gives insights in the current state of reuse practices at Google.

RQ3.1: Which roles are involved in reuse practices? We investigate which roles are involved in reuse, as well as their relationships, responsibilities, and motivation for performing reuse.

RQ3.2: What reuse practices are applied and how often are they used? We assess which (and to what extent) reuse practices and reuse activities are effected and how they are supported by tools and infrastructure. This includes as well the entities that are reused, namely knowledge and artefacts.

RQ3.3: Which measures are taken to assess the adequacy of reuse? We analyze which measures exist to assess adequacy of reuse as well as the strategies for reuse improvement and how they are applied.

4.1.1.2 Identifying challenges and opportunities occurring in practice

This category identifies problems, challenges, and opportunities that are relevant in practice. The results of these research questions indicate practical solutions and open issues.

RQ3.4: What are problems, challenges, success factors? We investigate which problems and challenges occur in practice and elaborate the benefits of and the success factors for reuse.

RQ3.5: What do software professionals consider as potential improvement? We collect ideas for improvements of reuse practices from the viewpoint of the software professionals. These might not (yet) be realizable or feasible but provide a ranking of the research directions in terms of relevance.

4.1.2 Study context and subjects

Our participants at Google are usually working on multiple projects, are organized in small teams, and are developing software with several programming languages, most frequently C++, Java, Python, and JavaScript. They were drawn from more than 25 different teams, had an experience between less than one year and more than 20 years in their current role. The participants worked at Google between less than one year and more than 10 years, with the large majority between one and 3 years.

39 participants answered the questionnaire. Their self-assessed experience level ranged from 3 to 8, with a median of 4, on a scale from 1 to 10, with most experience denoted with 10. Ten participants took part in the interviews. Their experience level ranged from 4 to 6, with a median of 5, on the same scale.

For the organizational roles of the participants see Tables 4.1 and 4.2.

Table 4.1: Roles of participants - questionnaire

Technical Lead	Developer	Maintenance	Manager
2	21	2	1
	6		
1			
6			

Table 4.2: Roles of participants - interview

Technical Lead	Developer	Maintenance	Manager
	2		1
7			

4.2 Methodology

Our sources of research evidence are the two complementary parts of survey research: interviews and questionnaires. In an informal pre-study, we collected a range of aspects, such as *reuse management*, *legal aspects*, *sources of reusable artefacts*, *reuse strategies*, *reuse adequacy* or *extent of reuse*, from the literature. These aspects serve as the basis for the questionnaire and the interview guideline.

Online questionnaire To gain a comprehensive overview of reuse at Google, we developed an online questionnaire (see Appendix 12.1). For each reuse aspect, several multiple choice questions were asked. Furthermore, we invited the participants to contribute further information, which was not covered by the questions. In the end of the questionnaire, we asked the participants to provide information on their level of experience in their current role, the time spent working at Google or the type of project they were working on. In total, the questionnaire contained 43 questions. Taking part in the questionnaire took approximately 20–30 minutes. Participation was optional.

We randomly selected 600 candidates for participation, based on an internal directory of all software developers at Google and sent the invitations via email. Our sample included developers from Google offices around the world, covering all types of projects and technologies. In the end, 39 developers took part in the questionnaire.

The responses of the online survey were analyzed with descriptive statistics and visualizations. As a result, we could establish a weight for the results obtained in the interviews.

Semi-structured interviews Complementary to the questionnaire, we conducted semi-structured interviews with software developers. We developed an interview guideline based on our pre-study to support the interviewer in structuring the conversation. The scope of the interviews was similar to the questionnaire. However, the aim of the interviews was to obtain detailed insights into reuse application in different development teams and projects, as well as its implications regarding non-technical aspects such as company culture and interpersonal skills. We, therefore, selected 10 participants with different responsibilities with respect to reuse from 9 different development teams. Each interview lasted one hour and was conducted by two re-

searchers, one leading the conversation with the participant while the other created the transcript and asked clarification questions.

Data collection & analysis procedures For data collection and analysis, we proceeded as described in Section 3.6.

4.3 Study results

To answer the research questions, we extract our information from the questionnaire and the interview data. We refer to outcomes of software development activities as artefacts. Artefacts provided or available for reuse are called reusables.

Reuse activities: During coding, we identified a set of activities related to reuse and use them to structure the results of RQs 2 and 4. The activities are: *publish* - reasons to create and publish reusables, *find* - where and how to find reusables, *understand* - means to properly understand reusables, *select* - selecting the right reusable, *adapt* - adapting the reusable, *integrate* - technically integrating the reusable. These activities encompass and extend the ones proposed by Karlsson [5].

RQ1: Which roles are involved in reuse practices?

From the interviews, we identified the following organizational and processual roles involved with reuse at Google.

Organizational roles

Engineer: Engineers are responsible for software development and maintenance at Google. Their motivation for reuse depends on the direct benefit they can obtain from it. Benefit means, for example, faster completion of features, visibility and impact of work, expressed for instance by a high amount of users for a reusable. Senior Engineers are asked to share their experience and advise (e.g. in library design discussions). Engineers are directly involved into reuse: they can introduce libraries, invest time in reusability, propose features to libraries, and share their own artefacts.

Manager, Technical Lead: As far as reuse is concerned, the managing roles have several responsibilities: firstly, they can decide to push or suppress reuse as they see fit. Secondly, they serve as coordinating role within and between teams to ensure that reuse options are identified on a larger level, e.g. on a feature granularity. In this way, reuse serves as a means to achieve consistency over the range of products. Thirdly, managing roles are responsible to mandate certain reuse decisions on different scales from teams to product areas. Lastly, the managing roles are also responsible to ensure legal compliance of reuse.

Team: A development team at Google usually consists of about 5 to 6 engineers, working on focussed tasks. They are coordinated by technical and product managers. The motivation to reuse is strongly dependent on the individuals in the team.

Processual roles

Producer: Every engineer takes the role of a producer internally as (nearly) all code is available to other internal projects, (about 41% of the participants share artefacts also externally, i.e. Open Source). Typically, the producers or current implementers of a piece of code have ownership over it and are contacted by engineers desiring a change. Ownership is lived as responsibility for the artefact, allowing for others to effect changes.

Apart from the default producer, there are designated producers, which provide central libraries and components, oftentimes taking the role of reuse champions, motivating and pushing reuse strategies at different scales. Their responsibilities and motivation differ as follows: producers serve as the gatekeeper to the library or component, ensuring that only new and valuable functionality is adopted. Furthermore, they are responsible to upgrade all consumers if they effect changes in the artefact. In order to ensure adoption of a reusable by the consumers, producers attempt to perceive common needs as early as possible, e.g. from usage patterns or legacy code. Furthermore, producers are lobbying the use of core libraries and components to improve the quality in the code base (since “introducing new stuff causes problems”).

Consumer: As a consumer, the engineer is responsible to integrate the reusable. If integration requires changes to someone else’s code, the consumer is responsible for contacting the code owner to find a solution. If the integration breaks other people’s code, the engineer is responsible to either fix the problem or to roll back the change within a fixed tolerance window.

Reviewer: Tightly integrated with the software development cycle at Google is the role of the reviewer, which is again taken by each engineer. Every piece of code submitted to the central code repository is subject to a review cycle. It is the reviewer’s responsibility to assess the quality of a proposed solution and to suggest better options, if available. In the context of reuse, this implies proposing reusables and assessing the adequacy of a reuse action.

Legal: The legal department is the main point of reference with respect to licensing issues, induced by third-party artefact reuse. Also, sharing artefacts with the Open Source community needs to be confirmed.

Third-party: Third-party describes all entities external to Google, whose artefacts are reused by Google in its software development. Most of the time, these are Open Source projects providing libraries for specific problems.

RQ2: What reuse practices are applied and how often are they used?

We follow a top-down structure to report the results of this research question: We first report processual results, then results concerning the individual reuse activities (*Publish, Find, Understand, Select, Adapt, Integrate*), and finally results that concern the reusable itself.

Reuse practices & processes

In our study, we found that, apart from a set of regulations regarding third party reuse, at Google there is no centrally controlled mandate for organizing reuse. Reuse processes and strategies can

be initiated by engineers as well as managers. According to the questionnaire, reuse is generally more ad-hoc ² (47%) than strategical ³ (37%). Comparing different types of development goals, reuse is more strategical for product development (24%) than for prototype development (13%) or tool development (13%). The interviews revealed that it is up to the engineers and managers to decide how much effort to invest into reuse. The scope of effort ranges from individuals investing their 20%-time ⁴ for reuse, to entire teams dedicating 30–40% of their time to it. Mostly, reuse takes place on demand: teams focus on completing features, and refactor for reusability when the need for reuse occurs. Furthermore, except for important artefacts, reuse knowledge is often in peoples' heads. The exception to this general philosophy are the teams providing the utility libraries for the entire company. These *core libraries* and components are reused by the entire company. Therefore, they are carefully designed by experienced engineers, and the scope of their functionality is guarded by the respective team. New functionality can be proposed by everyone but is only adopted if it provides significant value to the company and is not yet contained in similar form in the library. In particular, one core team uses the concept of an incubator, a dedicated place where engineers can place potential reusables. Over the time, their usage is monitored and if the reusable obtains a sufficient usage, the team will invest in improving its reusability and, over different maturity stages, incorporate it into the library.

As far as third party libraries and frameworks are concerned, there is no central coordination for their selection. However, if multiple options for large libraries exist, the core teams decide on one option to ensure homogenous use. The policies for introducing a third party library are as follows: the engineer introducing a library is responsible to ensure that no library for the same functionality is imported yet and to ensure with the legal department that the license is compatible. If these prerequisites are met, the engineer is responsible for integrating the library in a dedicated part of the code base, ensuring it is ready to use and keeping it updated. Third party library usage is governed by several rules, e.g. the libraries are only allowed to be integrated by linking; copy and paste from third party code is forbidden. Third-party reusables are frequently used in projects: more than half of the participants in the questionnaire have introduced at least one or two, with 10% having introduced more than five⁵.

Reuse activities

Publish: The interviews indicate that the decision to create and provide a reusable is mainly value-driven. We found two main reasons why a reusable should be created: either there is a common need for a specific functionality or the functionality itself provides substantial value (even if it is not commonly needed). However, there is no explicit process for creating and

²From the questionnaire: “Ad-hoc reuse means that developers are allowed to reuse any available artefact which seems suitable for the task at hand.”

³From the questionnaire: “Strategical reuse implies that reuse is driven by specific organizational goals. Usually guidelines or policies describe which reuse is adequate for a given situation.”

⁴ At the time of the study, Google encouraged their employees to spend about 20 percent of their time experimenting with their own ideas.

⁵Introducing here could mean “importing to third party base at Google” as well as “starting to use a library that was imported by another engineer”.

publishing reusables; the decision whether to create a reusable is up to the developer/team. Usage patterns in legacy code can provide indicators for common needs and serve as basis for creating reusables.

Within Google, most of the code is available for everyone via one central repository: 68% of the participants of the questionnaire stated that all artefacts are available for other projects and 41% stated that only some artefacts are available for other projects. To promote reusables, we found several ways: there are dedicated mailing lists, internal web pages, Google+, newsletters and user guides. Table 4.3 shows the top three ways of sharing artefacts: via a common repository (97%), via packaged libraries (34%), and via tutorials (31%). These numbers support the above findings.

Table 4.3: Which are your top-three ways of sharing artefacts?⁶

Answer	#Answers	Percentage
Common repository	31	97%
Packaged libraries	11	34%
Tutorials	10	31%
Blogs	6	19%
Email	3	9%
I do not share artefacts	1	3%
Other	1	3%
None of the above	0	0%

Find: The questionnaire indicates that the most common source for finding reusables are internal repositories (87%), closely followed by colleagues (38%). Furthermore, resources on the web are used to find third party libraries and ideas. External build automation tools were not used.

Furthermore, the questionnaire suggests that the most frequently used way to retrieve reusables is the internal code search engine, which provides (read) access to the complete code within the Google main repository (see Table 4.4). Code search is followed closely by “communicating with colleagues”. The interviews detailed on the communication between the engineers: the communication culture is very direct, so asking for advice on what to reuse is happening regularly. Furthermore, the reviewers will suggest reuse if they recognize reimplementations or suboptimal solutions in the submitted code. Lastly, user mailing lists are available for all bigger libraries and are used for questions on reusables. Focussed support for finding reusables, such as code recommenders or code completion are not widely used for finding reusables.

Understand: The interviews indicate that code search provides code snippets, which are used to understand reusables. As most participants of the questionnaire find reusables by code search, this provides a means to properly understand reusables by usage examples. The questionnaire indicates that all sorts of code documentation and tutorials are among the most often used to properly understand a reusable (see Table 4.5): reviewing interface documentation (72%),

Table 4.4: Which are your preferred ways to find reusables? Please indicate the top three.

Answer	#Answers	Percentage
Code Search	30	77%
Communicating with colleagues	25	64%
Web search	19	49%
Browsing repositories	16	41%
Browsing documentation	9	23%
Other	3	8%
Code completion	2	5%
Code recommenders	1	3%
Tutorials	1	3%

searching example usages in blogs and tutorials (64%), reviewing implementations (64%), and reading guidelines (51%).

Table 4.5: What do you do to properly understand and adequately select reusable artefacts?

Answer	#Answers	Percentage
I review interface documentation	28	72%
I look for example usages on blogs and tutorials	25	64%
I review implementations	25	64%
I read guidelines	20	51%
I explore third-party products	11	28%
Other	4	10%
I participate in trainings for third-party technologies/artefacts	2	5%
Nothing	0	0%

Select: The interviews as well as the questionnaire only give some hints for this activity. The interviews indicate that there is a difference between reusables originating from an internal or an external source; Internal reusables are usually preferred, since it is more difficult to reuse external code due to processual and legal issues. Usually, external libraries or code are only used if there is confidence in the library/code and only for specialized tasks. Depending on the project type, the selection criteria for external reuse may vary: there are some projects where the footprint needs to be as low as possible, and there are some projects where run-time performance is most important. In general, one important selection criterion is that the documentation of the reusable needs to be good. This is also supported by the questionnaire (see Table 4.5), as the 4 top-most answers concern documentation.

Adapt: When deciding whether to integrate a reusable, engineers assess the effort and the possibilities of integration. If major changes need to be effected on the reusable, the engineer is required to discuss them with the respective owner before copying or modifying it.

Integrate: Usage of software libraries are the most frequent reuse mechanisms employed at Google, followed by frameworks, design patterns, and code scavenging. Component-based development does occur, but significantly less (see Table 4.6). Usage of software libraries occurs by linking/calling and includes-linking to the Google-internal code base.

Table 4.6: Which of the following possibilities of reuse do you employ most? Please indicate the top three.

Answer	#Answers	Percentage
Software libraries	32	89%
Software frameworks	19	53%
Design patterns	13	36%
Code scavenging (copy, paste, modify)	12	33%
Component-based development	8	22%
Architecture reuse	5	14%
Product lines	1	3%
Application generators	1	3%
None	0	0%
Other	0	0%

Reusables

The data from the questionnaire implies that in the software development at Google, a variety of development artefacts is being reused (see Table 4.7) on different levels of granularity, however with a strong preference of libraries (see Table 4.9) and a focus on general utility. There is also a significant amount of reuse of domain-specific functionality (see Table 4.8) and on a small granularity, such as classes.

The interviews provided detailed insight on the nature of reusables: the majority of the reused artefacts are provided internally. A lot of them are provided by the different projects and available, yet normally not curated, for reuse. Furthermore, there is a *core* of utility and infrastructure libraries that are reused by entire Google. These libraries are well documented, maintained by dedicated teams and provide among others basic infrastructure for the programming languages used at Google. Open Source libraries also belong to the set of reusables. Their contributions are also curated for maintainability.

Apart from code and other development artefacts, we found that reuse of knowledge/ideas is prevalent. It occurs if problems need to be solved again, but the existing solution is not reusable due to fundamentally differing programming paradigms, for example. In these cases, engineers rely on “tested knowledge” gained from experience or their co-workers.

Furthermore, tools, training documents and examples are reused across the company.

Table 4.7: Which are the top-three types of artefacts you reuse?

Answer	#Answers	Percentage
Source code	37	97%
Code in binary form	12	32%
Style guides	11	29%
UI Designs	10	26%
Requirement docs. / Use cases	5	13%
Architecture documentation	5	13%
Prototypes	2	5%
Informal design models	2	5%
Own, domain specific design models	2	5%
Semiformal design models (UML)	0	0%
Formal design models	0	0%
Other	0	0%

Table 4.8: What is the scope of the reused artefacts?

Answer	#Answers	Percentage
Domain-independent general functionality	27	77%
Domain-specific functionality	18	51%
Product-specific functionality	9	26%
Other	0	0%

Table 4.9: What granularity do the reused entities typically have?

Answer	#Answers	Percentage
complete libraries	31	84%
one or more classes	18	49%
coarse-grained, e.g. entire frameworks	13	35%
fine-grained, e.g. single methods/functions	11	30%
small code sections	8	22%
Other	0	0%

RQ3: Which measures are taken to assess the adequacy of reuse?

According to our questionnaire, the majority of engineers attempt to implement sustainable design decisions (76%) to ensure reusability of their artefacts. This applies especially to the “core” and infrastructure libraries. To improve the code quality and avoid duplicate solutions built by engineers, the library teams inspect legacy code for unusual usage patterns. These serve as sources for requirements of new reusables.

83% of our participants effect unit tests and 71% effect system tests to ensure a certain level of quality of their artefacts. We also found individual solutions to mitigate negative effects of reuse “hacks”, such as cloning: one engineer ensures traceability by adding the original location in the internal directory to copied code to enable traceability in cases of bugs and need for clarification, despite the presence of a system wide clone detection.

Overall, reuse adequacy is not monitored following a structured process. However, to determine the impact of a reusable, the library teams measure the adoption of reusables by counting the number of users. Effected reuse manifests itself by calls to the reusable, which is treated as the current metric for successful reuse. Currently, some engineers are working on a more elaborated reuse metric. Moreover, to handle the problem of dependency explosion and limit unwanted reuse, build visibility rules and access rules were introduced.

RQ4: What are problems, challenges, success factors?

In the questionnaire and the interviews, we asked the engineers about challenges and benefits they experienced with reuse, as well as factors making reuse beneficial to them.

Issues and challenges: The factor considered most *disruptive to the reuse process* was difficulties in finding artefacts (56%). This ranged before the difficulty of adapting the artefact to project needs (53%) and licensing issues (44%). The “not invented here” phenomenon is listed (34%) on the fourth place. Accessing the artefacts was hardly considered an issue (6%). 13% of the engineers did not experience difficulties disrupting the reuse process within their teams.

Dependency explosion was considered the most severe *issue attributed to reuse* (52%). 39% considered the ripple effects caused by changes in reused artefacts as problematic, while 35% linked reuse to a decrease of code understandability. Loss of control was mentioned by 29%, while 26% did not experienced issues caused by reuse.

73% of the engineers agreed that the *absence of reuse* in their projects led to duplicate implementations. 64% attributed increased development effort to insufficient reuse. Inconsistencies (48%) and high maintenance effort (45%) were also considered as negative consequences.

The interviews revealed more details on issues with respect to the reuse activities, management and philosophy.

Publish One significant challenge for library providers is to create the “right” reusables. To this end, they need to identify common needs before engineers start to create own solutions. The availability of all code for reuse poses a challenge, as also unmaintainable solutions might be reused. Structuring reusables is a difficult challenge when publishing them. The engineers need to find a suitable level of abstraction for the reusable and classify it accordingly so that

others can identify it as reuse candidate. Some teams provide infrastructure to address this issue; however, the solution is not widely known yet.

Find The cost of searching for reusables, composed of the time needed to look up and assess candidates as well as the probability to find nothing or to not be able to integrate, is still high. As a result, engineers create their own solutions.

Understand If understanding whether a reusable fulfills the current need is too difficult, developers will create their own solutions. This happens especially when reusables are too abstract and thus the engineers cannot understand them anymore.

Adapt Challenges occurring during adaption concern the usability of library interfaces, as well as the overhead required to adapt a reusable. Low usability might cause users to employ libraries in unintended ways. The overhead for adaption is currently underestimated, especially for copy-paste reuse.

Integrate Incompatibilities in programming paradigms currently truncate reuse possibilities at the level of ideas. Another issue is bad modularization of libraries, which increases the size of the product binaries.

Management Reuse is challenging for management at different levels: despite management support being important, reuse cannot be simply induced by a manager. Especially under time pressure, reuse will not be as beneficial as intended. Operationally, the management of dependencies remains a challenge, as changes run the risk of breaking multiple projects. This is a challenge, as the implementation of reuse must not block anyone in accomplishing their work.

Philosophy A challenge with respect to company philosophy is to strike the right balance with respect to reuse: on the one hand, a lot of people need to be motivated and educated for reuse. On the other hand, excessive reuse should be prevented.

Success factors and benefits: According to our questionnaire, the main benefits of reuse experienced by the engineers were: higher development pace (91%), less maintenance effort (69%), and higher code quality (47%). Furthermore, they attributed the availability of new functionality (41%), higher consistency(38%) and regular bug fixes (34%) as beneficial to reuse.

The questionnaire indicates that high quality of reusable artefacts is the most important success factor of reuse at Google (68%). It is followed closely by supporting infrastructure and tools (65%) and adequate abstractions (58%). Homogeneous development culture, as well as dependency management are also considered important (each 32%). Surprisingly, the direct communication culture (19%) or the presence of suitable incentives (6%) were considered as significantly less important. In the interviews, in contrast, both aspects were pointed out by several engineers.

In the interviews, we found an overall sensitivity to code quality with the engineers. This expresses itself through a variety of applied constructive and analytical quality assessment methods, such as “serious review cycles”, patterns and guidelines, maturity levels for some components, as well as a solid testing infrastructure. These are partially mandated and employed independently of reuse. However, their presence seems to facilitate beneficial reuse as they provide confidence in the quality of the reusables as well as a safety net for effecting changes.

Engineers stressed that the supporting infrastructure, especially the code search platform, as well as the continuous integration approach and the communication culture were success factors. They particularly enable finding, understanding and integration of reusables. Furthermore, the selection of adequate reuse mechanism as well as a suitable abstraction level are important. Engineers agreed that success factors for reusables, besides overall good quality, were an intuitive usability as well as high stability.

In terms of management, the low organizational overhead for engineers to initiate reuse encourages its adoption. In contrast, the library providers named thorough planning, involving senior engineers, and a strict error handling to ensure high quality of the reusables as one of their main success factors.

With respect to the company culture, the engineers saw the networking and communication attitude as a success factor. In particular, they mentioned the 20%-time, read access to the code-base, open criticism culture, “dog fooding”⁷, and extensive profiling of products to continuously improve performance.

RQ5: What do engineers consider as potential improvement?

In the interviews and the questionnaire, we found wishes for improvement in three categories: culture, technical support, and methods.

Culture: Despite reuse being an established tool in development, engineers wish for more reuse spirit with managers and fellow engineers. Furthermore, they wish for a dual development strategy, interleaving feature production phases with phases focussing on code quality.

Technical support: Most of the wishes for potential improvement of technical infrastructure/-support address the three reuse activities *Publish*, *Find*, and *Adapt*:

Publish Participants in the interview wished for a tool that automatically makes code reusable. Furthermore, they wished for a shared place where common utility functions can be published. The questionnaire indicates that the architecture of reusables or libraries should be improved to meet reuse needs: 41% of the participants of the questionnaire answered to the question “In your opinion, what would be the three most important actions to make reuse beneficial in your company” that reusables need to be bundled more coherently in terms of functionality and 38% that libraries should be split to provide more specific functionality.

Find Most wishes expressed in the interviews concern finding the *right* reusable: They wish for natural language queries for code search, a better keywording mechanism, a better discovery tool, and a pattern and components catalogue that contains reusables. Furthermore, an oracle, that looks at a piece of code and tells whether it already exists is wished for. The questionnaire indicates that discovery is still an issue, since 45% answered to the question “In your opinion, what would be the three most important actions to make reuse beneficial in your company” that available artefacts shall be listed in a *marketplace* to ease the discoverability of useful functions, and 21% answered that libraries should be merged to ease the discoverability of already implemented functionality.

⁷“Dog fooding” refers to using own products in one’s daily work, thus finding problems immediately.

Adapt In the interviews, the participants said that it should be easier to change code (more than just refactoring) and that protocols should be provided for the usage of functions.

Furthermore, some wishes expressed in the interviews concern the assessment of reuse: Engineers wished for a method to determine how many times specific methods are used in different projects. The questionnaire further indicates that structured rules for dependency management (21%), clear strategic decisions for interface support (28%), and maturity levels for reused artefacts (17%) would make reuse more beneficial.

Method: The interviews further indicate that different levels of abstraction is still an issue as the participants wished for homogeneous abstractions and a programming language with an ideal abstraction mechanism.

Remarks

Overall, we noticed a decisively reuse-friendly atmosphere during our interviews. The participating engineers considered it a significant and beneficial part of their development practice. This impression was backed up by a closing question from the questionnaire: 62% of the participants stated that the current state of reuse was “just right”. 21% opinionated that there “should be more of it to leverage the full potential”. None of the participants wished the effort spent into reuse to be decreased. 18% opted for “other”.

4.4 Discussion

Our study found that a considerable share of reuse in the considered environment is ad-hoc and opportunistic, guided by few strict principles. The major motivational factor for reuse is the short-term reduction in development effort for new features. Due to the development infrastructure and collective code ownership paradigm, all code is, in principle, reusable by everyone. In essence, much of the reuse occurs “along the way”, during the feature-driven evolution of the code base. If code parts evolve that can be used in multiple places, commonly used abstractions are extracted. An exception to this general tendency can be observed for the company’s *core libraries*, which provide generic, product-independent functionality for which a common need among the developer community is known to exist or anticipated for the future. The library providers are responsible to manage the evolution of libraries according to consumers’ needs, keeping the libraries’ structure coherent and providing a certain quality level. This is to a large extent supported by both the shared code base and the open communication culture.

A noteworthy finding of the study is that there is no explicit common notion for what constitutes adequate reuse. Besides legal constraints on the reuse of external artefacts, few commonly mandated rules or guidelines for developers exist. The existing rules are either imposed from the reuse champions, or originate from developers addressing a reuse issue. Apart from these rules, the judgement about the adequacy of reuse, involving factors like the maturity of the reused artefacts and the degree of entangledness between the own code and the reused artefacts, lies in the responsibility of individual teams or developers.

The study shows that despite of decades of research in software reuse, fundamental challenges to integrate reuse seamlessly into the developer workflow remain. The company uses a shared code base and provides a powerful code search environment that allows searching in the entire code base. Nevertheless, the major obstacles to reusing artefacts mentioned by the developers were the identification of suitable reusables and the adaptation of reusables to specific needs. The most challenging issues involved in the creation of reusable artefacts were the a priori identification of commonly needed functionality and the structuring and publishing of reusable artefacts.

Despite the identified challenges, a considerable amount of reuse does occur at Google. The main factors for successful reuse are the high quality of the reusable artefacts, induced by comprehensive code reviews, as well as the development infrastructure (in particular the shared code base and code search) and the open communication culture among the developers. Reviews of code changes identify missed reuse opportunities, allowing the developer to rework the code accordingly, thus increasing the extent of reuse.

On the other hand, the study found that while the open development culture and shared code base at Google undoubtedly fosters reuse of code, it also poses a significant risk: Immature code may be reused inadvertently, resulting in quality and maintenance problems. This risk is mitigated by the obligatory code reviews and build visibility rules.

As most prevalent open issues for effective reuse the study identified the improvement of tool support for creating and finding reusable artefacts. The participants suggested tooling for an automated publishing of code as a reusable artefact. Ideas for the improvement of search capabilities included natural language queries for code, a component catalogue and a tool that can identify functionally similar code given a code snippet.

4.5 Threats to validity

Internal validity

Self-selection bias: Most participants displayed a favorable attitude towards reuse. Since participation in our study was optional, it is possible that only engineers considering reuse as beneficial volunteered to take part. The tendency of the answers seems to confirm this bias.

Selection of participants: The participants of the interviews were sampled by convenience through personal contact in just one company. This might have introduced a bias. To mitigate, we sampled the participants from different teams and different roles. The participants of the questionnaire were sampled in an automated way. Therefore, we could not avoid including absent engineers in our sample. No overlap occurred between interview and questionnaire participants. In this way, we obtained the viewpoint of 49 different people within the organization.

External validity

Sample of participants: We sampled all our study participants from Google. We are aware that this greatly impacts the generalizability of our results. However, since we aim to report on applied and scalable reuse practices, we still consider our results as valid. Although the response

rate to our questionnaire was low, we consider the answers as valuable since the set of participants reflected the distribution of teams and products within Google.

Construct validity

Limitations of research methods: To compensate the limitations of our research methods, we employed multiple methods to collect the data, namely interviews and questionnaires.

Interpretation of the interviews: To ensure the correct selection and categorization of the statements, the interviews were always conducted by two researchers to ensure the correct understanding of the information. The coding and triaging of the data was always performed in discussion by three researchers.

4.6 Considerations for practitioners

Our results indicate that certain reuse practices can be applied beneficially at a large scale. This section distills these practices and includes some remarks on the prerequisites to implement them. Furthermore, it highlights open issues.

Quality matters. Our results provide the following insights to practitioners: Most importantly, the *quality of reusables* is crucial for motivating developers to reuse them. Alongside this concern follows the *quality of the documentation* available for the reusables. This enables developers to quickly establish whether the reusable meets their needs. A prerequisite for achieving the desired high quality is a quality-conscious mindset with engineers and managers, which translates into investments in constructive and analytical quality measures, e.g. reviews, continuous quality assessment and thorough testing.

Invest in infrastructure and automation. Effective reuse requires a suitable *supporting infrastructure*. This impacts especially finding, integrating, and publishing reusables. Automated indexing of the code base, a powerful search engine, as well as the possibility to link to reusables provide the basis for efficient reuse.

Control organizational overhead. *Low organizational hurdles* together with *suitable incentives* make it easier for developers to initiate reuse, as it decreases the additional effort for reuse. Nevertheless, a *systematic strategy* and process for the central reusables is important to strike the balance between including new functionality and keeping the reusable maintainable and usable.

Culture matters, too. Reuse requires *trust* between the different parties involved. We found that an *open communication and criticism culture*, supported by reviews and design discussions, help to establish the necessary confidence.

Open issues

In our study, the engineers pointed out issues that would further improve reuse: Producers need support to *uncover required functionality* as early as possible, (ideally before developers create their own solutions) to avoid multiple implementation of the same functionality. Despite the presence of a powerful search engine, it is still hard to select the best candidate from a huge

list of options. Support for *natural language queries* was seen as one option to alleviate this issue. For frequently used reusables, even minor changes might impact all consumers and entail major rework effort. To reduce the burden of modifying reusables, there should be a feasible and scalable way to automatically *effect and propagate changes*. The effort spent on reuse might not pay off immediately but will amortize after some time. Therefore, there should be *adequate incentives* for developers and managers to do tasks from which they do not benefit directly but are necessary to leverage reuse properly. Reuse entails a trade-off between functionality that is easily accessible and loss of control. If employed in an inadequate way, reuse can become a risk to projects. Therefore, it is important to devise structured approaches, meaningful metrics and tools to *assess the adequacy of reuse*.

4.7 Summary and conclusions

We performed an exploratory study on reuse in practice with the goal to provide practitioners with examples of scalable reuse practices.

To this end, we interviewed 10 engineers at Google and collected the opinion of 39 engineers via a comprehensive online questionnaire. By means of qualitative data analysis, we extracted the context of reuse at Google, involving roles and responsibilities, as well as reuse practices, reused artefacts and reuse mechanisms. We furthermore collected the current issues as well as success factors and ideas for improvement. Reuse at Google is performed in an ad-hoc manner on an inter-project scale with the goal of decreasing development time for features. There is no structured approach to assess the adequacy of reuse. The main unit of reuse is code, which is integrated in a library style. This is enabled by a capable support infrastructure that turns the largest part of the code base into a searchable reuse repository. Furthermore, pervasive automated testing increases the confidence of modifying reusables and ensures that code incorporating reusables behaves as expected. The quality of the reusables, the supporting infrastructure, as well as the communication culture are seen as clear success factors. The biggest challenges to reuse are finding the right reusables from the vast amount of functionality, adapting the reusables to meet current needs, and licensing issues. For the future, the participants wished for more reuse spirit within the company, as well as better support to address the problems in finding and adapting suitable reusables. Furthermore, they wished for support to find the best candidates for new reusables as well as to assess the reuse effort.

Based on our results we invite practitioners to invest in the quality of reusable artefacts and in reuse supporting infrastructure. Furthermore, organizational overhead should be kept low. Lastly, a culture of openness and trust seems to support reuse.

Research addresses current open issues of the industry. However, the low adoption poses questions of usability and scalability. As future work, we propose to collect detailed accounts of research results potentially applicable to the challenges pointed out in our study. Our results reflect the reuse practices at Google. While they provide a detailed account of success factors and challenges, data from further companies is needed to assure generalizability.

5 | A case study of software reuse adoption

Adoption of reuse approaches in practice can pose multiple challenges. Research-industry collaborations are considered a suitable vehicle to mitigate adoption difficulties and to validate the applicability of scientific results. However, they do not always live up to the expectations of either of the partners. Unfortunately for researchers and practitioners alike, insights from failed adoption initiatives and co-operations are often difficult to obtain. This hinders discussions on lessons learned during the adoption process and delays improvements. This chapter aims to mitigate this by presenting lessons learned from interviews we conducted with practitioners in the context of a study on software reuse in industry. The participating company had already undertaken two failed attempts to adopt an advanced reuse approach. In our study, we identified tacit assumptions that were related to the encountered difficulties and present the lessons learned from the adoption approach. Furthermore, we report strategies that helped us to overcome the scepticism caused by a previous unsuccessful guided collaboration. Parts of this work are published in [25].

Contents

5.1	Challenges of structured reuse adoption	62
5.2	Study design	64
5.3	Adoption of a strategic reuse program	65
5.4	Lessons learned — Adoption attempts	70
5.5	Current research collaboration	72
5.6	Summary and conclusions	74

5.1 Challenges of structured reuse adoption

Understanding and improving the adoption of research results in practice is an ongoing concern in the software engineering community [111]. To this end, one key challenge needs to be overcome: research does not operate under the same operational constraints as industry. As a result, it is usually difficult for researchers to “eat their own dog-food” and test their approaches under realistic circumstances. Furthermore, assumptions on significant context factors that need to be present for a successful adoption of the proposed approaches might remain implicit, with potential detrimental effects to adoption efforts. This is especially critical when proposing measures requiring significant and long-term changes in industry, such as large organizational restructuring [17]. Adoption of a structured reuse approach is a prime example for this case: approaches usually require substantial changes on the technical as well as the organizational level. These deep changes are very difficult to achieve [69]. Whilst abstracting a problem from its context is a necessity of scientific thought [112], as researchers we run the risk of overestimating the applicability and the potential benefits of our results if we fail to explicate a clear relation to the assumed and envisioned application context.

Providing approaches that are applicable in practice is a challenge for researchers; successfully adopting these approaches might be even harder from an industrial perspective. Research-industry collaborations seem a suitable and popular vehicle to address and mitigate these challenges for both sides [113, 69, 114, 115]. However, they are not free from challenges: one of the most obvious differences between academia and industry becomes apparent in the goals they typically bring to a collaboration: researchers wish for high quality data to validate their approaches, practitioners wish for a suitable, affordable, and fast solution for a specific issue [115, 116]. Consequently, also successful technology adoption might have a different meaning for each party [27].

Structured reuse, by means of reusable components [14], Software Product Lines (SPL) [8], or the more pragmatic Inner Source [23] philosophy, have a long history of being considered a “silver bullet”[64] in research and practice alike. Virtually every paper on software reuse starts with a mantra-like declaration of abstract benefits (improved quality, decreased cost, decreased time to market) [65, 3, 29] and proceeds by adding several new aspects on how to achieve them. Often, the feasibility is demonstrated by the application of the approach in a small number of case studies. Whilst, from a researcher’s perspective, this is a reasonable practice, it poses several problems for industry adoption: For instance, often-times the exact conditions under which the proposed solution is supposed to work are not mentioned in detail. Neither does the reader learn about all the preconditions and assumptions that make a solution applicable in the first place. Lastly, there is a tendency to advocate structured approaches as superior independently of a company’s strategic context, business goals, or domain.

Several works have reported on adoption attempts of structured reuse in practice and proposed respective success and failure factors: Joos [84] reports on the experience of introducing a systematic reuse process at Motorola in the 1990s that succeeded due to management support, education of engineers, suitable incentives, tool support). Frakes and Fox [1] present a reuse failure mode model, derived from a questionnaire on (code) reuse answered by 113 people from

29 organizations. The authors mention four dimensions (managerial, economic, legal, technical) that need to be addressed when implementing systematic reuse. Lynex and Layzell [29] assess the management and organizational issues raised by the introduction of reuse programs in industry. They collect inhibitors to adoption gained from experiences from reuse projects reported in the literature, provide reasoning for causes and present possible solutions. Fichman and Kemerer [18] examine the extent to which the introduction of a formal and systematic reuse program was advanced in one large organization. They found that reuse was prevalent on an informal, local, scope but neglected on an inter-project, systematic, level due to an incentive conflict with respect to team priorities such as completing a project on time and on budget. Sherif and Vinze [70] report on barriers to reuse adoption, concluding that individual declination towards reuse was caused mainly by the organizational stance on reuse adoption. Morisio et al. [13] report on success factors for adopting or running company-wide reuse programs, collecting evidence from 24 reuse projects in European companies varying in size, business domain and culture. The authors identify underestimation of the required effort as main driver for failure and conclude that success of reuse projects depend on management commitment, awareness of human factors and modification of non-reuse processes according to the specific context of the company. Dubinsky et al. [60] investigate the reasons that cause companies to reluctantly move away from ad-hoc reuse in the form of code cloning to structured product line approaches. They identify efficiency, low overhead, short-term thinking, and lack of governance as main drivers.

Challenges that can occur when attempting to adopt structured reuse became apparent to us during a study on reuse in practice. To establish how reuse is currently effected in practice, we started a number of research-industry studies, of which 2 are currently completed. In their context, we so far interviewed approximately 30 practitioners from well established software companies (head count ≥ 5000 , more than a decade of experience in the market). Especially in one company, issues during the adoption of research approaches were a consistent theme across departments and hierarchy. We therefore decided it worthwhile to extract and report lessons learned.

Goal and contribution: Our goal is to create awareness of challenges of adopting structured reuse in practice. In this chapter, we report practitioner accounts on several stages of a reuse adoption attempt, collect harmful patterns in the form of tacit and implicit assumptions and interpret them to identify lessons learned. Practitioners and researchers thus can consider this information and use it to counteract some of the challenges when driving a similar adoption attempt.

Outline: Section 5.2 introduces the context of the adoption situation of the given company. Section 5.3 details on the two previous adoption attempts. The findings are interpreted in Section 5.4. Section 5.5 reflects on the options of researchers and practitioners to improve collaboration in the context of research adoption. Section 5.6 concludes the chapter.

5.2 Study design

In the context of evaluating the state of the practice of reuse in industry, we so far completed two exploratory studies with two companies. At the current state of our research, we conducted 35 hours of interviews and collected 138 questionnaire responses. This chapter collects impressions from one specific company at which we conducted around 20 one-hour interviews and obtained 69 questionnaire responses. We report the impressions of our interview participants on the adoption of a structured reuse approach to their development practice.

In the following, we detail on the company context, summarized in Table 3.1, and line out the case study design and the main results. In addition, we provide pointers to supplementary material.

Company characteristics: We can not disclose the names of the company and the involved partners in the previous research collaboration. However, we line out the characteristics of the company in order to give the reader an understanding of the circumstances: U is a national software house providing technical information services and business information products to their clients. The company was founded in the 1960s, emerged as a service provider and gradually moved to providing stand-alone software products and support services. Currently, U counts around 6000 employees. The company structure is hierarchical, structured along market segments within one specialized domain. The products have historically grown independently over decades and contain a broad mix of technologies and various conventions in terms of architectural styles. In addition, many product varieties have been created to address niche markets, resulting in several hundred different products. This heterogeneity, furthermore, leads to a range of different required release dates.

Software development and maintenance is very heterogeneous across departments and teams, ranging from waterfall processes to tailored Scrum approaches. Also the level of development tool support, testing practices, and code ownership is highly diverse. As a result, product parts are integrated in a "big bang" style to prepare the respective releases. Developers usually work on specialized topics of a single product and tend to be responsible for the respective subsystems (Subsystem code ownership, see [108]).

Participants: We study the current practice of reuse at U by means of an exploratory study consisting of an interview phase with 20 participants, followed by questionnaire phase with 69 respondents.

To gain insight into contextual and strategic factors of reuse adoption, we drew interview participants from each of Us product and support development departments and all levels of the hierarchy between senior developers (including architects and user experience designers), higher, and top management (including the CTO and board members of the development department). The participants worked at U between 15 and more than 30 years. By means of qualitative data analysis, we extracted the context of reuse, involving roles, responsibilities, and reuse practices, i.e. reused artefacts and reuse realizations. We collected current issues, success factors, and ideas for improvement.

Questionnaire participants were invited by a newsletter and a post on a company news portal. Respondents came from 10 of the 13 departments. 44% worked at U for at most 10 years, 20% for 11-20 years, and 36% for more than 20 years. 15% reported their role as manager. The respondents job focus was mainly on development (78%), and architecture (13%). Respondents at U usually work within one product area and are organised in product departments over several hierarchical units. They are developing software most frequently in C# and SQL. In addition, they use Java and C++. The participants are software development professionals and managers with an experience ranging from 5 to 30 years. We selected them from a range of positions, hierarchy levels, and departments. Their participation was optional.

Methodology: As means of data collection, we used semi-structured interviews and online questionnaires.

The study contained semi-structured 1-2 hours interviews always conducted by two researchers and an extensive online questionnaire containing mainly closed-questions. We analyzed the interviews by means of inductive content analysis. To extract relevant concepts, we coded the transcripts [109] and triaged the emerging concepts for relevance w.r.t. reuse. We conducted the study during 3 months in the time frame from September 2012 to February 2015.

Supplementary material in the Appendix (Section 12.1) shows the topics and sample questions of the interviews. For this chapter, we focus on the interviews as the theme of technology adoption and a failed research collaboration mostly emerged during these sessions. We, therefore, extracted the related statements from the transcripts and present them in the following.

Goal of our collaboration: From the academic perspective, the goal for our research collaboration was to assess the state of reuse in practice. From the industrial perspective, the goal was to collect a neutral and honest account on the current perception of the reuse strategy, including improvement points.

5.3 Adoption of a strategic reuse program

At the beginning of our study, we found the following situation with respect to reuse adoption: several years ago, management had decided to move from a range of independent products in the same or similar market segments to a more integrated version. This was a business decision based on feedback from clients and market requirements. In the process of pursuing this goal, a need for unification between products became apparent. To address this, the need for a structured reuse approach came into focus with the goal to improve software production, reducing errors, and providing customers and users with a homogeneous product. Two adoption attempts had taken place: one without guidance by researchers and one in collaboration with academia.

Reuse goals and targeted approach: The company's reuse goals are: consistent extension of the .NET framework used by their products, consistent integration of existing products, lower maintenance costs. These goals should be reached by a structured company-wide reuse approach based on a shared platform, providing building blocks for products. The target vision of the reuse approach contained elements of SPL engineering and characteristics of Inner Source development.

Current state of reuse: At the point of writing, reuse is mandated for an internal utility platform providing domain-independent functionality to products. In U, code is reused in a very heterogeneous way and mostly retrieved from colleagues on the basis of personal contact or, occasionally, by searching the web. In addition to code, style guides are reused.

5.3.1 Tacit assumptions and their consequences on adoption

At the company, many of the success factors mentioned in the literature for a transition to more strategic reuse were present: there was top management commitment, reuse was institutionalized by means of a visible workforce unit, the reason for reuse was founded on business needs, reuse was established as an explicit goal for developers, and a reuse champion was appointed [13]. However, in practice, we identified a number of assumptions tacitly held by key stakeholders of the adoption process which, from the beginning, significantly decreased the chances of a successful outcome.

In the following sections, we report the assumptions and their effects. We denote the roles which held the assumption in the following way: *HM* for *higher management*, *TM* for *top management*, *SD* for *senior developers*, followed by the number of the assumption.

In boxes, we contrast the respective assumptions with the participants' retrospective comments on their effect¹. We denote the roles which provided the statements in the following way: *HM* for *higher management*, *TM* for *top management*, *SD* for *senior developers*, followed by the number of the statement. Furthermore, we distinguish between *product (P)* and *base platform (B)* participants.

5.3.2 Unguided adoption attempt — company-internal

Several measures were taken with the goal to establish a structured internal reuse approach: on the one hand, top management decided to build a generic base platform for all products the company was currently producing. For this task, a new department was founded and the use of that platform was mandated for all products. Furthermore, reuse became part of the developer goals with the intent of creating a reuse culture. Lastly, a designated reuse manager was appointed with the goal to identify potentially reusable entities and to build up a network of contributors.

Assumption 1, HM

“Successful transition to the new approach will not require deep organizational change.”

Based on this assumption, the department for base development was treated like any other product department. It was provided with a small to medium volume of resources (the weight between product development and base development is roughly 10:1) and assigned with a consulting status. Consequentially, recommendations for strategies or change, e.g. with respect to

¹The quotes were translated into English. They are no longer verbatim, however we attempted to stay as closely as possible to the original meaning.

product architecture, were not binding for the products. As a consequence, significant energies were lost in the attempt to convince products to actively participate in the change. One senior developer summarized this by

SD1-B: “We have a rather political company culture, involving a lot of lobbying.”

Furthermore, the company’s heterogeneous development culture was not considered in terms of its impact on the reuse strategy. This meant that a platform approach should be adopted despite lacking central access points for source code and binding governance rules. Furthermore the infrastructure for code repositories, building, and testing was different in every department. Coordination was furthermore complicated by differing release time requirements of products. Lastly, the company relied on hierarchical communication paths and hindered exchange across the departments. Mending this was considered a key requirement for improvement by participants of product and base departments:

SD2-BP: “Increasing networking between the departments is a necessity.”

Assumption 2, TM/SD

“Products will see the benefit of the platform and, therefore, will contribute.”

From the beginning, top-management was relying on active participation from products to contribute to the platform. This should, on the one hand, ensure the usefulness of the platform content, and on the other hand, educate every product department in the use of the platform. The contributing product developers, therefore, should serve as multipliers. One of the tacit assumptions, however, was that middle management would provide resources when planning their efforts without further incentive. Senior product developers summarized the situation as follows:

SD3-P: “Top-management had a rather abstract interest in reuse. Middle management rather saw the cost than the potential. Workforce had varied views. We enjoyed [building for reuse] and could refer to the management goals. However, the architects were not enabled to create new structures.”

The obvious discrepancies reportedly discouraged other developers from participating in the reuse initiative:

SD4-B: “Middle management occasionally even went out to punish developers that invested time in reuse contributions. [...] This quickly made many others lose interest [to participate].”

Assumption 3, TM/HM

“It is apparent to everyone how to do reuse.”

Reuse was never precisely defined, despite the platform use being mandated and despite reuse being explicated as a developer goal. The vagueness of the reuse initiative even led to negative effects:

SD5-P: "During that time there was a massive focus on code reuse (also in the sense of copy and paste). Frequently, this lead to inadequate solutions for a given problem."

Consequentially, neither the scope of the platform and its intended use, nor the mechanisms for reuse on an individual level were explicitly discussed. Instead, product needs and product commonalities were assumed to be understood. This lead to a skewed implementation of the basis platform,

SD6-P: "Bigger departments and early collaborators dictated and shaped the platform to their needs."

as well as to a insufficiently focussed functionality.

SD7-P: "The basis unit is trying to do too much."

As a consequence, products reported a significant overhead in their deliverables, loss in performance of the product, and issues with dependencies.

SD8-B: "Products sit on a large bunch of code they don't really need."

Relying on the multiplier model, the base development department assumed that extensive training in platform use was not necessary and that providing assistance, whenever asked for it, would be sufficient.

Assumption 4, HM

"One reuse manager can fully organize and coordinate reuse."

The purpose of the reuse manager was to collect candidates for reusable entities from all development departments. In addition, the role served as contact for anyone with inquiries or willingness to contribute to reuse. However, coordination was only intended as a part-time activity.

SD9-B: "Reuse coordination was always difficult. It stopped completely after I moved to a new role."

In the absence of a centralized code infrastructure, this identification and extraction of candidate reuse entities was a largely manual and infeasible process. The heterogeneity of the infrastructure as well as closed code repositories made the task more difficult.

SD10-P: "Source code analyses won't work here."

As a workaround, the reuse manager set up a wiki structure where reusable components could be registered with a description and a contributor contact. However, contributions remained scarce.

Effects of unguided adoption attempt

After bringing the reuse initiative under way, top-management returned to everyday business. Since reuse was institutionalized to some extent, the platform was built and introduced. However, acceptance varied greatly among the products.

HM1-B: “If users don’t like a platform, they will evade it and build their own things next to it. [...] This becomes apparent when changes in the platform have no effect in some of the products.”

Furthermore, the voluntary co-operation between departments did not work as expected.

SD11-P: “Middle management enforced decisions that directly opposed the strategy of top management.”

Summing up, the expectations towards the reuse efforts were only partially met.

5.3.3 Guided adoption attempt — research collaboration

To mend the weaknesses of the previous reuse adoption attempt, management initiated a research collaboration on the topic.

With interviews within the development organization, the researchers assessed the state of the current situation and proposed a new approach, based on principles of Open Source practices.

Assumption 5, HM

“A research collaboration will provide us with a tailored solution.”

As improvement, they proposed a model for reuse that required a significant restructuring of the entire organization. This idea was encountered with scepticism across all organizational ranks.

TM1-BP: “The project and the idea were good in theory. However, [the researchers] did not take into account the company culture. [...] We were unable to adopt these ideas and maybe we were scared by the required effort and organizational risk. [...] They did not consider how to connect research to the company needs.”

Assumption 6, HM/SD

“Collaborating researchers will align their goals with the organization.”

Furthermore, the attitude of the researchers caused significant irritation. Practitioners perceived them as disinterested towards the organizational risk as well as the long term applicability of the proposed approach.

SD12-P: “The solution they proposed simply was unadoptable for us at that point in time: too risky, too expensive, unclear how we should even get there!”

This eroded trust and goodwill on the company side:

SD13-P: “[The researchers] were pushing their approach, no matter if it was suitable for us or not. They went around the house interviewing people, but nothing really came from that.”

SD14-B: “They cared for their results and not so much for our needs.”

Consequently, the case confirmed the assumption in practitioners that research results were no match for them. As a result, the collaboration was not continued.

SD15-B: “We expected more from this collaboration.”

Effects of guided adoption attempt

Years later, the memory of this experience is still lingering in parts of the organization, resulting into tangible scepticism towards research(ers). On the one hand, this resulted in warnings from our industry contacts, such as:

SD16-B: “Someone has done research on this context before, there is burned soil.”

On the other hand, scepticism was outspoken:

SD17-P: “Don’t give us yet another tool.”

SD18-P: “Maybe this worked for someone else, but it won’t work for us.”

5.4 Lessons learned — Adoption attempts

In this section, we first sum up the core points of the encountered difficulties during research adoption. Then, we propose our lessons learned on how to avoid/mitigate these.

5.4.1 Company-internal adoption difficulties

In the first reuse adoption attempt, several points caused the initiative to miss its goals: the understanding of the organizational, conceptual, and technical preconditions and requirements of the chosen approach was not deep enough. In this case, this led to an underestimation of the necessary change, as well as to overly optimistic expectations regarding the benefits of the enterprise. In the end, the heterogeneity of the organizational context with respect to infrastructure, development processes, applicable mechanisms, communication culture, release dates etc. significantly reduced the benefits obtained from the platform approach.

Assumption 1 reveals that the company management fell for a well-known reuse myth: “reuse is for free” [50]. This misunderstanding highlights a typical danger of research adoption, as research papers as well as industry targeted publications often-times focus on the assumed benefits and general applicability of the approaches they propose. In this way, risks and necessary preconditions might be overlooked in the plan for adoption. In addition, the assumption expresses the desire to obtain a quick solution without investing significant resources. This desire partially prevented a thorough study of the organizational impact that a consistent application of the adoption was bound to have if it was meant to be successful. This aspect highlights the difficulty of motivating investments in research adoptions whose benefit can not be easily quantified.

Lesson 1

A detailed assessment of the organizational factors is crucial to make a realistic estimate in terms of effort required to adopt a structured reuse approach.

Assumption 2 shows that top management did not recognize the incompatibility of the current company and development structure and the planned adoption approach, which led to an overly optimistic assessment on the effort required for a successful outcome. As a consequence, the

adoption process stagnated when the company returned to everyday business. This highlights the need for top management to actively monitor the change process and compensate for the frictions induced by the change until the goal is reached.

Lesson 2

Management commitment needs to exceed the initiation phase of a structural reuse adoption and provide long term support to reach the expected benefits.

Assumption 3 highlights that the concrete implications of the approach were not understood in detail. One possible reason is that the concept to be adopted, strategical reuse, seemed so intuitive on an abstract level that the lack of methodology prescription on how to effect the adoption did not become apparent until it was too late. This highlights the need for a concrete adoption plan, including a selection of desired methods.

Assumption 4 provides a further example of how an abstract intuition for research adoption forms a mismatch with the details of a business environment. As a result, management did not assess the feasibility and practicality of the measure.

Lesson 3

A rough and intuitive plan is not sufficient for structured reuse adoption. Instead the aimed for approach needs to be understood in detail to plan a successful adoption.

5.4.2 Research-industry adoption difficulties

Assumption 5 highlights the (mis)understanding of a research collaboration as a consultancy service. Whilst research collaborations *can* take that form, this often-times does not match the goals of the participating researchers, who (as in the present case) might look for feedback on or validation of a new approach. As a result, the approaches proposed by researchers might not always be the best match for the current situation of the industrial partner. Furthermore, the assumption shows the expectation that research approaches fit to a company without adjustment to the specific situation. Since research strives for general applicability, these two aspects are bound to collide if no explicit measures are taken. In this case, a part of the previous collaboration could, e.g., have consisted in a joint tailoring effort with industry delivering their concrete context and researchers providing support for an adoption strategy. Lastly, the assumption suggests that the goal for the cooperation might have been too ambitious in terms of scope.

Lesson 4

It is important that researchers and practitioners alike be clear about their agendas, openly discussing goals and restrictions of their share of the adoption project to detect incompatibilities early on.

Assumption 6 points to an expectation that is valid to a certain degree: if a collaboration is to be mutually beneficial, both parties might need to be flexible in terms of their goals. However, from the practitioners' perspective, their research partners did not show that kind of courtesy. The lack of interest in preconditions and context required for the adoption of the proposed approach caused frustration, since the suggested benefits seemed out of reach to practitioners. The attempt of pushing an approach onto a company for which it was not beneficial caused a lot of damage for the given and also future research-industry collaborations. It confirmed to several involved individuals the view of academia as theory-obsessed and practically irrelevant, a factor that increased scepticism towards software engineering research in general.

Lesson 5

Sensitivity to the operational circumstances and restrictions of practitioners' work context is relevant to establish a trustful and beneficial relationship that is required to reach the collaboration goals.

5.4.3 Threats to validity

Reuse adoption was one of several aspects of our study. A dedicated study would have likely produced a more detailed picture. Nevertheless, the theme re-occurred reliably throughout the interviews so we consider the aspects as a useful starting point for further investigations.

This chapter reports impressions from a single case study in a single company. Therefore, the results can not be considered general. However, several aspects coincide with experiences from other industry projects.

The participants were selected by a company insider. We invited to participate in the study by giving several talks within the company. This introduces a self-selection bias, which we tried to mitigate by selecting the volunteers according to their department and their position to obtain a varied picture.

The section on guided adoption is limited to the practitioners' perspective. Our scope was to highlight the difficulties encountered from their perspective and we did not want to adopt the position of an arbiter. Therefore, we did not contact the researchers of the previous collaboration.

5.5 Current research collaboration

Since the research adoption is still an active topic, the company agreed to participate in our study. The motivation on their part was to reflect on the adoption attempts of their reuse strategy, to evaluate the current state of how reuse is conducted across the different departments, and to use this information as the basis for improvement strategies. Our goals were data collection for a broader view on software reuse in current practice. In this case, the goals of industry and research aligned rather well and the scope of the collaboration was clear and of little risk. Being

aware of the past research collaboration experience of this company, we invested strongly into communication, transparency, and expectation management. We could establish a close link between the company and research contacts, which keeps alive the cooperation and supports a regular exchange of findings.

5.5.1 Research perspective

Establishing a trustful collaboration after a bad research adoption experience can be challenging. The following points helped us to overcome most of the encountered obstacles:

Transparency of goals and scope: From a research perspective, we were transparent about our goals and requirements with respect to results, publishing, and our position towards the company. We clearly communicated our minimum deliverables for success. At the same time, we carefully sourced the interests and agenda of the company stakeholders and clarified the expected deliverables as well as the legal restrictions on our research. Together with company contacts, we discussed the realizability of these goals and clarified potential legislative, moral, or scientific objections.

Knowledge about the previous failure: In the context of initiating the collaboration, we investigated the history of previous research adoptions and research-industry collaborations to find out when they took place, who was involved, which goals and results were achieved and which issues were encountered. This prepared us for potential apprehensions and helped us to address them whenever they surfaced.

Critical distance to research approaches: Lastly, we could gain trust by displaying a flexible stance on our own research approaches. We reflected on the (implicit) preconditions that our solutions presupposed e.g. on the tool level, infrastructure, culture, goals and mapped this to the context of the specific company. In this way, we could reach a common understanding of potential next steps for the collaboration.

5.5.2 Industry perspective

After two adoption attempts without satisfactory outcome, the company reflected thoroughly on the weaknesses of their previous strategies. In the current adoption attempt, the company has taken several measures to improve the process. The following aspects helped them to proceed:

Create organizational prerequisites: During reflection it became clear that the company did not yet fulfil several of the prerequisites that were necessary for a successful research adoption. For instance, the hierarchical communication structure was found to impair adoption: first, it caused a local prioritization of efforts, which counter-acted the strategy. Second, the lack of exchange between departments prevented the necessary distribution of knowledge and the homogenization of processes. To mitigate these issues, the company instantiated cross-department exchange forums for their technical experts. Via these forums, the company is guided through the changes needed to prepare a successful adoption. Management also realized that the adoption process needed a champion with decision power if the adoption process should be kept alive along every-

day business. As a consequence, it provided the platform department with decision rights to push measures necessary for adoption.

Building and instantiating a long term adoption strategy: To ensure a more successful adoption, the company hired an expert that had significant experience in leading similar adoption processes in larger software companies. The expert is now building up a concrete long term strategy to ensure beneficial adoption. At the same time, the company entered another research collaboration with the goal to obtain a complimentary and neutral viewpoint on the current situation of the adoption process.

5.6 Summary and conclusions

In the chapter, we reported the experience of a company attempting to adopt a structured reuse approach. We identified six assumptions that negatively impacted the adoption process and discussed their implications. We interpreted the findings and derived lessons learned. In particular, we found that selecting a suitable approach for the current context and given goals was difficult since many factors need to be accounted for. In addition, the preconditions for research approaches often remained unclear. We conclude that to execute a successful adoption, companies need support on how to guide change to experience benefits in the long run.

Research collaborations can potentially help with the adoption process. However, in our study we found that conflicts in short and long term goals as well as missing transparency of expectations erode the trust needed for a successful collaboration.

Lastly, we detailed on aspects that helped our current research-industry cooperation to overcome some of the challenges from researcher and practitioner perspective.

6 | Synthesizing the case studies

Decades of research have proposed reuse methods and techniques and partially studied their adoption. Have these visions been successfully adopted in practice?

In this chapter, we take a first step towards an answer by comparing and integrating the outcomes of the two in-depth empirical studies on software reuse presented in Chapters 4 and 5. We conclude that in current practice: source code remains the only significant reusable, Inner Source approaches can be adequate and successfully adopted on a large scale, and organizational and technological heterogeneity significantly impairs adoption of any type of reuse transcending clone-and-own. Parts of this work are published in [41].

Contents

6.1	Comparing reuse practices	76
6.2	Study goal and research questions	76
6.3	Study design	77
6.4	Analysis Methodology	79
6.5	Study Results	80
6.6	Discussion and relation to state of the art	87
6.7	Threats to validity	95
6.8	Summary and conclusions	96

6.1 Comparing reuse practices in two large software-producing companies

To be able to support practitioners in experiencing successful reuse, we need to deepen our understanding of the current state of reuse in practice. In addition, we need to integrate existing evidence and investigate if reported issues of previous studies [5, 19, 117, 13, 21, 100, 22, 60, 81] are still current in today's professional software development.

The goal of this chapter is to take a first step towards this investigation. To this end, we systematically integrate insights from two recent in-depth industrial case studies on reuse practices. We compare their results and relate them to previous work with the aim to identify open issues potentially relevant for a broader scope of companies.

Specifically, we compare observed reuse practices, effects and influence factors in two large and diverse software producing companies (denoted as G and U in the following¹). We collected information from a total of 138 professional software developers by means of an extensive online questionnaire (108 respondents) and interviews (30 participants, 35 hours of interviews). The study at G precedes the study at U and was designed as an exploratory study on the state of software reuse in practice. Based on the results at G, the study at U was designed to contrast the outcomes for a different, yet more common, type of company and development context. These differences in context also required an adaptation of the study design. Therefore, at U, we first conducted the interviews to determine which topics of the first study were applicable. In a second step, we rolled out the questionnaire.

In this study we present the result of our integration, detailing thoroughly on study design, methodology, company contexts, and the respective findings.

Outline Section 6.2 lines out the study goal and derives our research questions. Sections 6.3 and 6.4 present the research design and methodology used for the integration of the studies presented in Chapters 3 to 5. Section 9.4.6 reports our results which are subsequently discussed and related to previous research in Section 6.6. Section 6.7 details the limitations of the integration before Section 6.8 proposes future work and concludes the Chapter.

6.2 Study goal and research questions

We compare software reuse implementations in two large companies: we look for commonalities and differences in the way the companies perform reuse, on the problems and benefits experienced and the factors which tend to inhibit or enable it. Formally, we define the study goal according to [110]:

The goal of the study is to *characterise* reuse practices
for the purpose of *comparing them*

¹Whilst the identity of Google is disclosed in [27], company U preferred to remain anonymous.

with respect to *their realisation and effects*
 from the viewpoint of *software professionals*
 in the context of two large *software producing companies*.

Effects refer to positive (*benefits*) and negative (*difficulties*) consequences of the presence or absence of the effected reuse approach. From our goal, we derive two main research questions:

RQ1. Which reuse practices are applied in the two cases?: We assess which (and to what extent) reuse practices and reuse activities are conducted and how they are supported by tools and infrastructures. From our point of view, *reuse practices* refer to how reuse is organised and implemented in a given company context. Consequently, the term encompasses several aspects: the entities that are reused (namely knowledge and artefacts), the process that is followed to create, obtain, and reuse them, as well as the way in which they are reused on the technical level.

RQ2. What are the effects of the respective approaches reported by in the two cases?: We investigate which problems and challenges occur in practice and capture the perceived benefits of and the success factors for reuse. We are especially interested to see whether our findings confirm the consensus of the literature, and whether the two companies report different aspects.

6.3 Study design

6.3.1 Methodological differences

Although the two investigations presented above shared goals, research methods and research questions, their implementation was not identical and, thus, necessitated methodological fine-tuning to perform the comparison. The following paragraphs highlight the differences of the cases and their consequences.

Timing. The investigation at G preceded the one at U and the comparison of the two studies was not planned at that time. When designing case U, we built partially on results of case G, including, e.g., items that were relevant as additional answer options in some questions (see, e.g., SFB3 in Table 6.2).

Study design. In case G, interviews and questionnaire phases were run simultaneously: The questionnaire sourced information on a large scale, whilst the interviews delivered a high level of detail to interpret the questionnaire. In case U, the interviews were conducted first. They delivered a very detailed view of the reuse practices and served as a filter to tailor the questionnaire to U's context. As a result, some questions do not have a direct counterpart in both studies (see, e.g., FAR3 in Table 6.2).

Scales. The scales used in the questionnaires differ: in case G questions could be answered only by multiple choice and free text. In case U, a different approach was used: The responses were given either on a four- or five-point Likert scale or as free text. For each question in U, we

report the question code, followed by a tuple (L<scale level 4 or 5>, <semantics of lowest bound of scale>, <semantics of highest bound of scale>). Example: FAR1 (L4, never, always). Last, the wording of the questions is not always identical² (see e.g. question CHR1 in Table 6.2).

Diffusion. In G, 600 participants were randomly selected from a database of employees and invited via personalized email. In U, due to legal restrictions, we were unable to invite participants directly. Instead, we published a link in a company internal newsletter of the development departments and an entry in a company internal developer news portal. In addition, we invited our company contact and interview partners to spread the word.

6.3.2 Selected material for comparison

Tables 6.1 and 6.2 contain the questions we used for the comparison between the two cases. They were selected since they had matching counter-parts in both studies. The response options are reported in Section 9.4.6 (see Tables 12.3 and 12.4). To interpret and discuss the findings (Section 10.5.2), we draw on parts of the interview data. Due to the differences reported above (and verifiable also on the supplementary material and on Tables 6.1 and 6.2), we rely both on reported guidelines to qualitatively compare case studies [118] and quantitative methods to analyse and compare the surveys' questions [119]: we report our analysis methodology in the following section.

Table 6.1: Questions selected for comparison for RQ1

Question ID	Question text U	Question text G
Extent of code reuse (ECR)		
ECR2	What type of functionality do you reuse and which organisational unit provides it? — L5, no use, always use	What type of functionality do you reuse?
ECR3	What is the scope of functionality that you reuse? — L5, no use, always use	What is the scope of the reused artefacts?
Finding artefacts (FAR)		
FAR1	How often do you use the following ways to find reusable artefacts? — L4, never, always	What are your top-three ways to search for reusable artefacts?
Reused artefacts (RAF)		
RAF1	How often do you reuse the following artefacts? — L4, never, always	Which are the top-three types of artefacts you reuse?
RAF2	What are your sources for obtaining reusable artefacts? — L4, never, always	What are your standard sources for reusable artefacts?
Technical realization of reuse (TRR)		
TRR1	How often do you use the following techniques to realize reuse? — L4, does not apply, strongly applies	Which of the following possibilities of reuse do you employ most? Please indicate the top three.
TRR2	Which granularity do the reused entities have? — L4, does not apply, strongly applies	What granularities do the reused entities typically have?

²Note that the questions in case U were originally expressed in German and translated for this work.

Table 6.2: Questions selected for comparison for RQ2

Question ID	Question text U	Question text G
Challenges, effects, and context factors of reuse (CHR)		
CHR1	How often do the following aspects negatively impact reuse in your team? —L4, never, always	In your opinion, are there difficulties disrupting the reuse process in your team?
CHR2	How often do potential disadvantages of reuse occur in your project? —L4, never, always	In your opinion, are there problems caused by reuse in your project?
CHR3	How often do the following problems occur due to lack of reuse in your project? —L4, never, always	In your opinion, are there problems caused by the absence of reuse in your project?
Success factors and benefits (SFB)		
SFB1	How often are the potential benefits of reuse realized in your project? —L4, never, always	Which benefits of reuse do you experience in your project?
SFB3	How important are the following factors to increase the benefits from reuse? — L4, unimportant, very important	In your opinion, what would be the three most important actions to make reuse beneficial in your company?
SFB4	In your opinion, which factors contribute to successful reuse projects in your company? — free text	In your opinion, what are the three most important key factors to make reuse beneficial in your company? *
Reuse in everyday development practice (RED)		
RED1	How much do you agree with the following statement* on reuse on your daily work? — L4, does not apply, strongly applies	not present in G, taken from Reuse failure modes, Frakes and Fox [1].
RED4	How much do the following statements* regarding the implementation of reuse apply to your organizational unit? — L4, does not apply, strongly applies	not present in G, taken from the organizational part of reuse maturity models, e.g. [120].
Finding artefacts (FAR)		
FAR3	How much do the following statements* apply regarding the accessibility and modifiability of company internal source code? — L4, does not apply, strongly applies	Success factor derived from G [27].

The statements are reported in the Appendix Table 12.4 next to the respective question code.

6.4 Analysis Methodology

From the analysis methodology perspective, the different scales are the most relevant issue for the comparison. We address it by applying an aggregation on the Likert scales of U and a scale conversion procedure to answers from survey G to make them fully comparable with those of survey U. Subsequently, we apply regular hypothesis testing. To explain why data conversion was needed we will refer as example to the question on *RAF1* from Table 6.1.

On survey U, the question *RAF1* was formulated on the following way: *How often do you reuse the following artefacts?* Participants could select on a 4 points Likert scale the frequency of usage of that item (values: never, occasionally, regularly, always). We aggregate answers on points 3 and 4 and label them *Category H: regular or high usage*. Similarly, we aggregate answers on points 1 and 2 and label them as *Category L: irrelevant or low usage*. Table 12.2 in the Appendix provides the aggregations used for the other scales types.

On survey G, question *RAF1* was formulated differently: *Which are the top-three types of artefacts you reuse?* For such type of questions participants could select up to three items, for others they did not have any limit. However, this was not enforced by the software: as a

consequence, some participants could exceed the number of available choices and selected up to four items, however most participants selected only one or two options³. Thus we believe it is reasonable to apply the following conversion procedure to make the answers of survey G comparable to those of survey U:

- We compute for each item the frequency of selection assuming that such a selection is equivalent to *Category H regular or high usage*: in fact participants are asked to select the *top three used artefacts*.
- Accordingly, we assume that when the item is not selected, this is equivalent to *Category L irrelevant or low usage*: we are confident that this is a quite straightforward assumption, because enforcement on the three options was not applied and some participants in fact exceed that number of selected options.

With such conversion the data structure is identical to that of survey U, where for each item a contingency table is assigned. Therefore, we apply the χ^2 test [119] on each of the resulting contingency tables to check whether the proportions in *H* and *L* differ significantly (with $\alpha = 0.05$). If the test is rejected (i.e. $pvalue \leq 0.05$) then we assign usage of item *i* to the category *H* or *L*, depending on which has the highest number of answers. When interpreting the findings of our analysis, we relate the statistical analysis to the findings of the interviews.

6.5 Study Results

Figures 6.4 to 6.5 summarize the study results for RQ1, Figures 6.6 to 6.7 for RQ2. Figure 6.2 explains how to read the graphical representation. For each item, the statistical significance of the answer tendency according to the χ^2 test (low or high relevance or likelihood) is represented. Rectangles denote statistical significance for the respective item, full circles denote an inconclusive answer (due to an even distribution or a non-significant skew), empty circles denote missing data. The underlying statistical data is available in the Appendix, Tables 12.3 and 12.4.

Figure 6.1 represents synthetically the main findings:

We found that reuse in both companies focused mainly on one artefact type, i.e. source code, thus not leveraging further reuse possibilities proposed by state of the art techniques. In the presence of an elaborate development tool-support and a quality-gated central repository, this infrastructure is more relevant for access and retrieval than personal contact (and can be seen as an instance of a successful reuse repository implementation); without this infrastructure, personal recommendations and contacts are important for pointers and access to potential solutions.

We found clear benefits (in terms of development speed and less maintenance efforts) when software reuse was effected in a homogeneous, ad-hoc, tool-supported way, and at a comparatively high level of granularity. In contrast, benefits did not materialise when reuse was effected in a

³This applies to all questions affected by this issue, i.e.: *FAR1*, *RAF1*, *TRR1* from Table 6.1, and *SFB3* and *SFB4* from Table 6.2. For the latter two, categories *H* and *L* are not about high or low frequency, but concern high or low relevance.

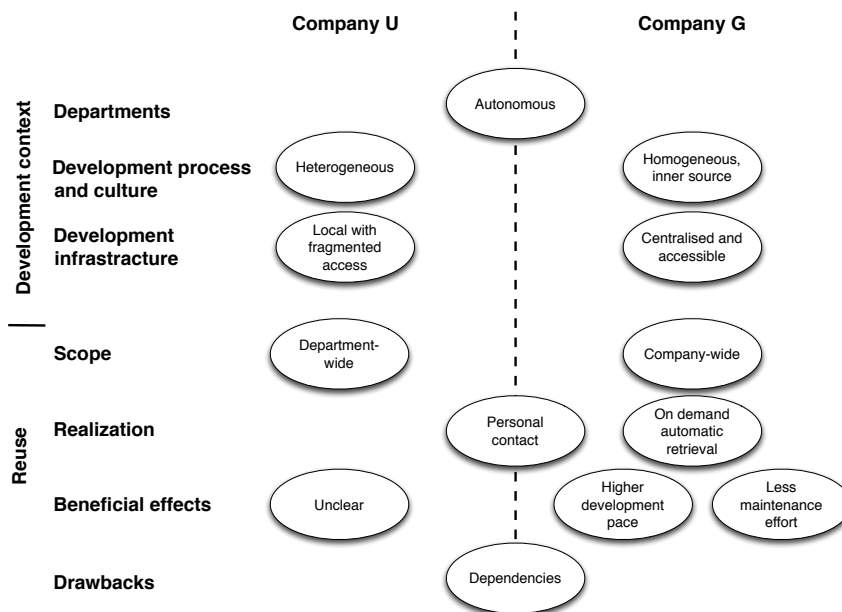


Figure 6.1: Summary of results, according to authors' data analysis and interpretation.

heterogeneous way, and tool support was mostly absent. In both cases, these characteristics reflected the development process and culture of the company.

Finally, we can report some improvements in terms of reuse implementation due to technical advances, as well as one instance of inner source practices that enabled reuse. However, many of the organizational challenges remain and need to be addressed in order to establish reuse approaches that are beneficial to companies.

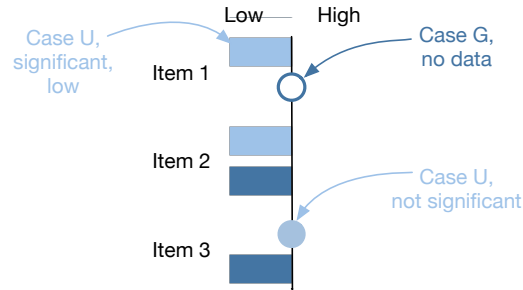


Figure 6.2: Example of the graphical representation of the results.

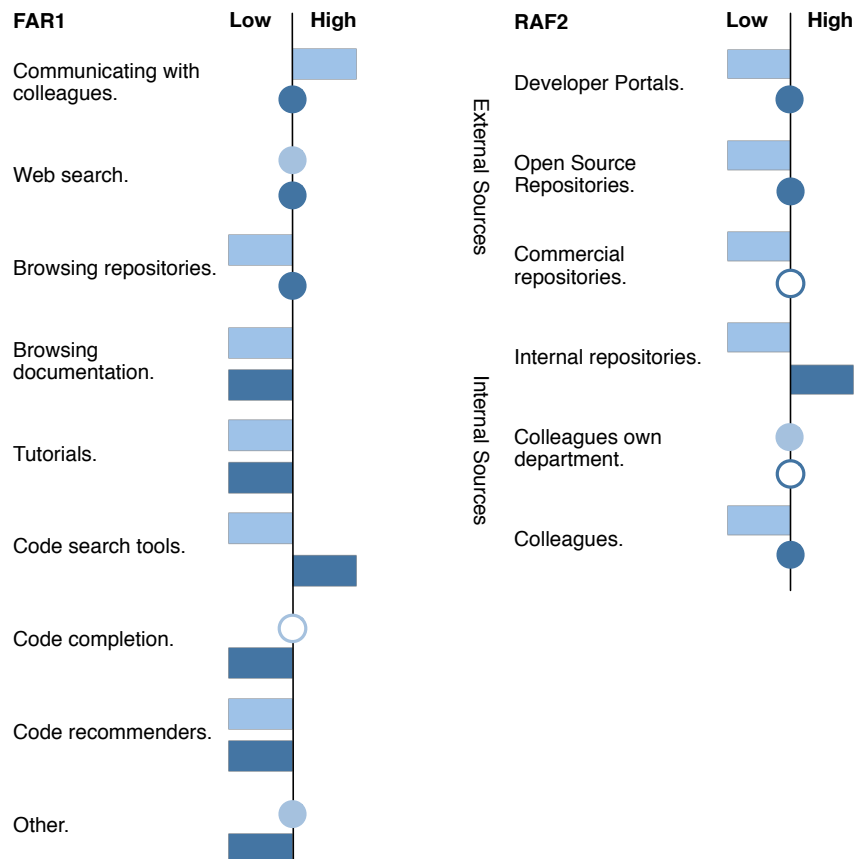


Figure 6.3: RQ1 - Sources of reusable entities and way of access, questions FAR1 and RAF2.

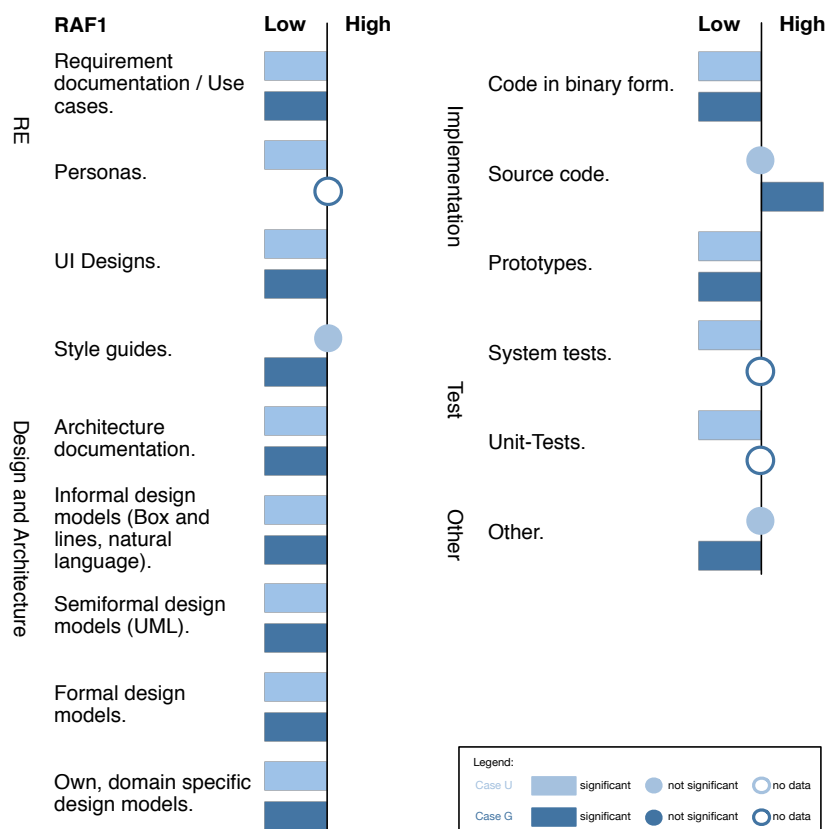


Figure 6.4: RQ1 - Reused entities, question RAF1.

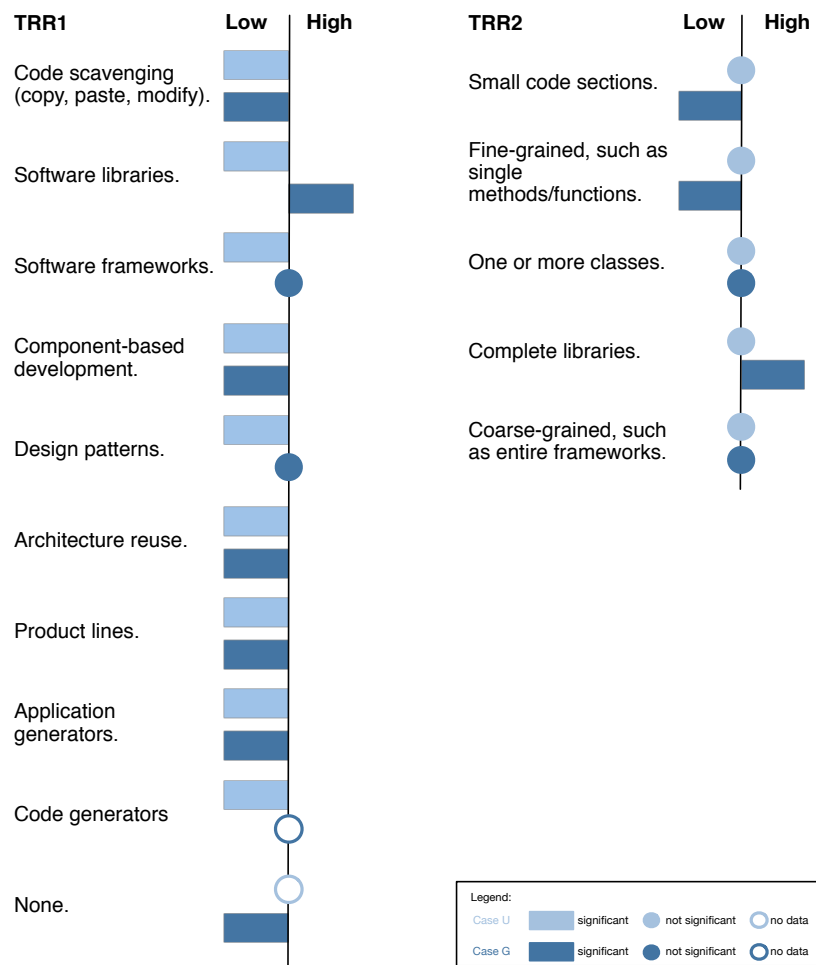


Figure 6.5: RQ1 - Technical realization of reuse, questions TRR1 and TRR2.

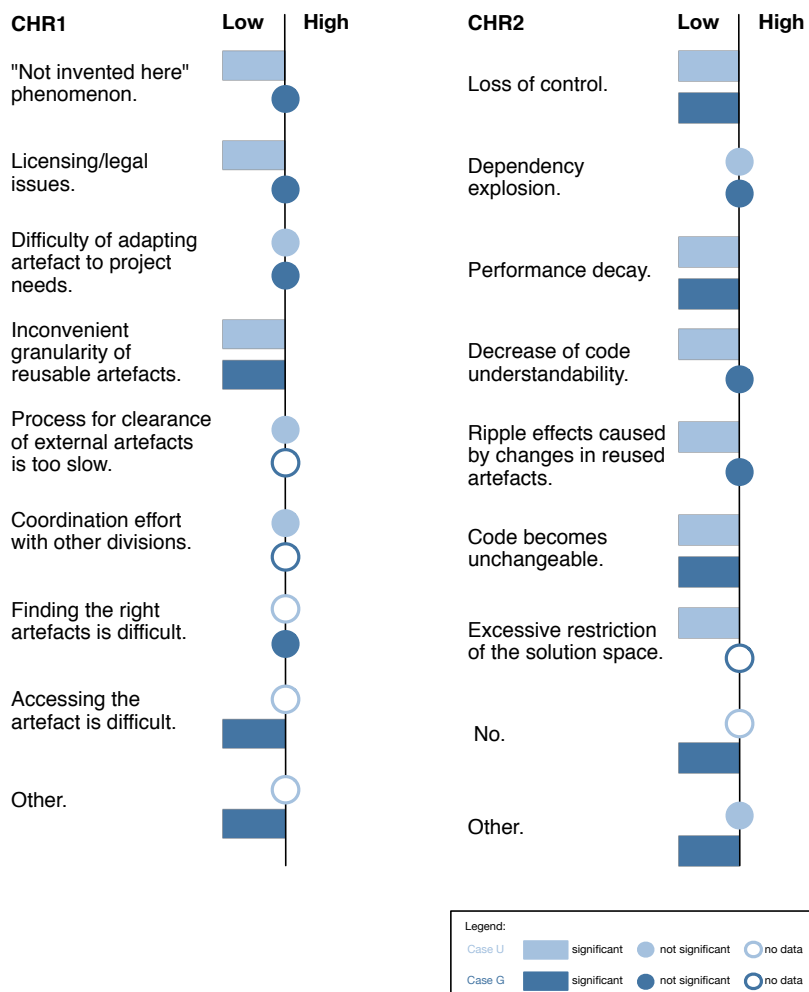


Figure 6.6: RQ2 - Inhibitors to reuse and issues due to reuse, questions CHR1 and CHR2.

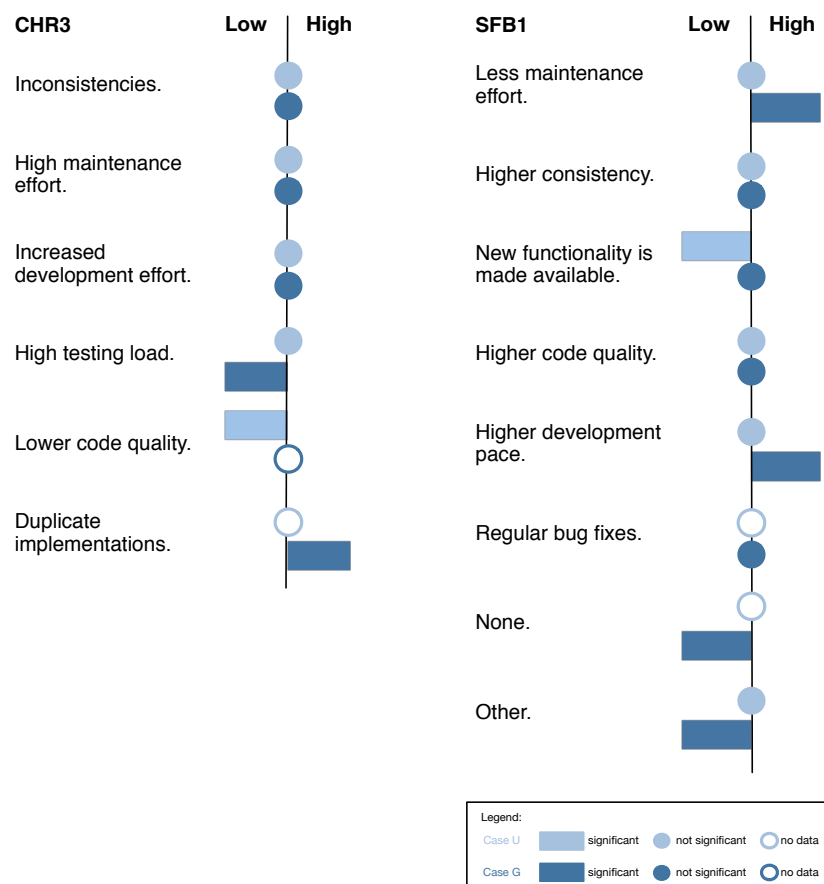


Figure 6.7: RQ2 - Issues of absence of reuse and benefits of reuse, questions CHR3 and SFB1.

6.6 Discussion and relation to state of the art

In this section, we discuss and interpret the findings of the data comparison for each research question, providing explanations thanks to the material of the interviews. In addition, we relate the results to the positions found in the literature⁴.

We structure the paragraphs as follows: *Comparison agreement* (i.e. aspects in common between the two companies, according to the methodology applied to analyse the results), *comparison differences* (i.e. diverging aspects in the two companies, according to the methodology applied to analyse the results), *interview data* (if appropriate), *interpretation* of the findings, *related literature* in support or that contrast the findings. The *id's* reported with the questionnaire items refer to their label in the respective data table given in the context of each RQ.

6.6.1 RQ1 — Comparing reuse practices

The survey questions related to this research question are reported in Table 6.1. See Table 12.3 for the responses, as well as the comparison and statistical values.

Reused artefacts (RAF1, Figure 6.4)

Comparison agreement: According to question RAF1 (L4, never, always), we observe that in both cases the majority of potentially reusable artefacts are *not* reused. In particular, no artefacts from the Requirement Engineering, Design and Architecture, and Test and Deployment phases are reused.

Comparison differences: In case G, *source code* (id=25) is the only response of statistical high relevance. In contrast, in case U no artefact is reused frequently with statistical significance (however the responses indicate that about half of respondents reuse *source code* (id=25) and *style guides* (id=34)).

Interview data: The interview data reflects the findings of the questionnaire: Source code is the clear reusable entity in G and also mentioned frequently as reusable in U.

Interpretation: These findings indicate that much of the potential for reuse is unleveraged in the two companies. Literature proposes reuse on a much wider scale from models to requirements (see, e.g., [2, 121, 122]). Reasons for this might be that artefacts of earlier development stages are not available, accessible, or considered useful.

Related literature: When comparing these findings to the ones reported by [13], we see that this focus tends to be typical for companies with a loose reuse approach. In companies aiming for a more advanced reuse approach, despite lacking the prerequisites in terms of process and tool support, ad-hoc code reuse is used as best effort to create, e.g., SPLs [60]. Our findings

⁴For the selection of studies, we started out from well-known sources, such as textbooks (e.g. [5]) and journal summary papers on reuse adoption and results in practice (e.g. [21]) as well as reports on research projects (e.g. the REBOOT [19] and ESPRIT [117] projects) on industrial accounts on reuse and manually followed the references, as well as searching the citing papers. In addition, we searched the last 10 years of proceedings of the main venue ICSR for current publications related to reuse in practice (e.g. [100, 22]). Additionally, we included known papers from different venues that contribute to the question (e.g. [60]). Lastly, we performed an unstructured search on Google scholar to find articles related to software reuse and adoption and practice (e.g. [81]).

on U supports this observation. Also a more recent on-line survey [81] confirms the presence of ad-hoc code reuse. Furthermore, it reports moderate reuse of requirements, which, however, we can not confirm in our two cases.

Extent of code reuse (ECR2 and ECR3)

Comparison agreement: The results for questions ECR2 and ECR3 (L5, no use, always use) show that *domain-independent general purpose functionality* (id=9) are highly relevant with statistical significance in both companies.

Comparison differences: In case G, *product-specific functionality* (id=11) are excluded as extent of code reuse. In contrast, in case U, although not in a significant way, *domain-specific* (id=10) and *product specific functionality* are mentioned by more than half of respondents, with all types of functionality highly reused (from id=1 to id=8).

Interview data: For case G, interviews confirm a core of basic functionality, on which systems are built. For case U, interviews report of a reuse approach that is arranged along multiple layers of general purpose functionality, but also domain specific reusable entities. Considerable leeway is given to single departments with respect to their local design decisions.

Interpretation: In U, reuse of more specialised functionality might indicate an opportunity for a more structured tight reuse approach, e.g., in the form of a product line.

Related literature: In literature, reuse of utility functionality is well covered, especially in the form of Open Source libraries and frameworks [12, 22]. In commercial scenarios, product line and component-based approaches suggest a similar behavior [8]. Work on *Inner Source*, furthermore, suggests that well defined domain specific functionality can be a suitable and valuable entity for reuse [23].

Finding and accessing reusables (FAR1, Figure 6.3)

Comparison agreement: In both cases, the results for questions FAR1 (L4, never, always) show that a number of retrieval options are currently considered irrelevant with statistical significance: *code recommenders* (id=16), *browsing documentation* (id=17), and *tutorials* (id=18). Web search (id=12) is reported in both cases by about half respondents, yet not significant.

Comparison differences: For case U, *communicating with colleagues* is the most important (and the only significant) way to find reusable artefacts. In contrast, within G *code search tools* are in this position, while considered irrelevant in company U.

Interview data: The interviews in G confirm the high usage of the code search tools, but also stress the communication (synchronously and asynchronously) and trust among engineers. In U, interviews as well as one of the answers to FAR3 (L4, does not apply, strongly applies), indicate that code available in U can not readily be searched and accessed across departments. Therefore, retrieval and accessibility are limited by lack of technical infrastructure.

Furthermore, in both interview rounds, the concepts of *reuse producers* and *reuse consumers* emerged: in case G, due to the development process and infrastructure, all developers assume both roles, drawing on, as well as feeding into, a global pool of reusables; in U, on the contrary, the producer role is limited locally, because a dedicated group of developers takes care of the common platform, whereas the remaining developers provide reusable code either within their departments or not at all.

Interpretation: These findings highlight the importance of three enabling factors of reuse: trust between colleagues, automated support for artefact retrieval, and technical accessibility of artefacts. In the absence of infrastructure, personal communication is crucial with the disadvantages of being slow and costly. Communication is still important in the presence of infrastructure; however, it is more concerned with the goal of clarification. The key access point shifts to the tool support asynchronously available to each developer.

Despite the reliance on tools, reactive support systems are not yet widely used to improve reuse.

Related literature: The mentioned enabling factors are, amongst others, considered preconditions for the so-called *Inner Source* approach [23], as well as the development of SPLs [8]. Technical support, such as *code recommenders* (id=16) are one example of research that should contribute to these three aspects [90] and might, in principle, target the right goals; however, these tools are not yet used widely (only one respondent per case declared to use them). This could be an indication that from a research perspective, more work needs to be done to adapt research work to the reality of practitioners, especially in terms of usability and scalability [123, 124].

Sources of artefacts (RAF2, Figure 6.3)

On the item level, we observe no agreement for question RAF2 (L4, never, always). However, there is a tendency in both cases towards company-internal artefact sources.

Comparison differences: The only statistically significant source in case G is that of *internal repositories* (id=37). In U, all but one source (*colleagues from own department*, id=39), are of low usage.

Interview data: In G, interviews further stress the importance of the central repository for reuse success.

In U, interview data indicates that, in the absence of technical access, one of the main sources of reusable artefacts might be the code that developers have previously written themselves. Also, the fact that developers stay in their department for many years and acquire specialist knowledge might impact their willingness to rely on and trust the work of others.

Participants from both companies mentioned the business and legal risks of reuse across company borders. Licensing was mentioned as significant inhibitor to reusing Open Source code, and the reliability and robustness of an external commercial software provider as high risk to reusing proprietary reusables.

Interpretation: In the case of G, the internal infrastructure and repository seem to provide adequate support for reuse across the company. In U, this kind of reuse is hampered by a combination of specialist knowledge and organizational and technical separation.

In both cases, internal sources preferred over external ones due to the potential risks entailed to the latter.

Related literature: Whilst the web is considered a huge repository in literature [12], this is scarcely reflected in the context of both companies: legal restrictions, security policies, and domain specificity prevent a significant exchange of reusable artefacts across company boundaries. Access to reusable entities is, thus, mediated by personal networks of developers. In addition,

we can confirm that the potential risks imposed by dependencies on third parties [34] impact the decision of companies with respect to third-party reuse.

Technical reuse realization (TRR1 and TRR2, Figure 6.5)

Comparison agreement: For question TRR1 (L4, does not apply, strongly applies), in both cases *code scavenging* (id=42), *component-based development* (id=45), *architecture reuse* (id=47), *product lines* (id=48), and *application generators* (id=49) are *not* considered relevant.

Comparison differences: In case U, all offered ways to technically realize reuse are marked as not relevant with statistical significance. In case G, realising reuse by means of *software libraries* (id=43) is of statistically significant high relevance.

Interview data: In case G, the interview data is largely consistent with the survey, although some instances of code scavenging were mentioned. For U, in contrast to the survey, the interview data suggests application of code scavenging (id=42), as well as some libraries (id=43) and framework-based reuse (id=44) (see also question TRR2).

IP and RL: see TRR2

Comparison agreement: Question TRR2 (L4, never, always) addresses the granularity of the reused artefacts. There is no common findings between the cases.

Comparison differences: *Complete libraries* (id = 55) are of high relevance in case G. In contrast, reuse in case U takes place on all levels of granularity, however, none of the corresponding answer is statistically more relevant than the other.

Interview data: -

Interpretation: Generally, G realizes reuse homogeneously on a higher abstraction level than U, where reuse is effected in a very heterogeneous way. Furthermore, the selection for U indicates a conflict to the results of the interviews and the responses of TRR1: reusing small code entities (snippets and single classes) suggests the presence of code scavenging. However, it is possible that the respondents do not have a strong preference for any of the reuse methods or do generally not reuse code as much (see RAF1).

Related literature: The realisation of reuse reported in case U aligns with the findings of Fichman and Kemerer [18]: in a study with 15 software developing teams, they found that reuse was prevalent on an informal, local, scope but neglected on an inter-project, systematic, level. The authors identified as root cause to the failure an incentive conflict with respect to team priorities such as completing a project on time and on budget. The case of U, furthermore, also confirms findings by Dubinski et al. [60]: in the absence of supporting technical infrastructure and processes, developers resort to primitive reuse mechanisms to model complex reuse approaches.

Work on Inner Source highlights the pivotal role of technical infrastructure for reuse, especially when effected in a loose and ad-hoc way [23]. Case G confirms these findings: reuse is conducted in an informal and ad-hoc way. However, supported by an advanced infrastructure (that required significant resources and management commitment, confirming findings of, e.g., [13, 80]), a viable company-wide development process, and suitable organizational incentives, reuse takes place in a large scale and across project boundaries.

Summary of RQ1

Case G exhibits a homogeneous approach to reuse, progressing in an inner source style that allows for ad-hoc and opportunity driven development. Case U exhibits a very heterogeneous approach in which many different styles co-exist. From this perspective, both companies reflect their development processes in the way they effect reuse.

Both cases focus on code reuse (G in the form of libraries, U with no predominant granularity). This entails that the large potential present in additional development artefacts remains unleveraged. The frequent reuse of general purpose functionality is confirmed.

Automated and tool-supported access to and retrieval of reusables is considered as key factor for effective reuse. In G, the impact of the infrastructure clearly shows in the finding and retrieving actions of reuse. In U, their absence restricts reuse to a local scope.

In both cases, the sources of reusables are mainly within the companies. The reported reasons for this were business risks in terms of security or robustness of the provider, as well as licensing aspects.

6.6.2 RQ2 — Comparing effects and context factors

The questions relevant for this RQ are reported in Table 6.2. The responses, the comparison, and the statistical values are reported in Table 12.4.

Inhibitors (CHR1, Figure 6.6)

Comparison agreement: Question CHR1 (L4, never, always) reports disrupting factors to the reuse process. There is no statistically significant inhibitor of high relevance. Respondents in both cases agree that *Inconvenient granularity of reusable artefact* (id=60) does impact them.

Comparison differences: The "not invented here" phenomenon (id=57) is reported little and is rated as irrelevant in case U. Questions FAR3 and RED1⁵ (both: L4, does not apply, strongly applies) indicate that difficulties in retrieving and accessing artefacts significantly disrupt the reuse process in U.

Interview data: At G, participants report the perceived ease of creating needed functionality from scratch over understanding existing solutions as inhibitor. Also, they mention the considerable (cognitive) effort involved in selecting suitable candidates from a plethora of potential results.

In U, the interviews disclose a further inhibitors to reuse: organizational and technical separation of departments, as well as the absence of a thorough global reuse strategy that takes into account different life cycle characteristics of system parts. This leads to the creation of unsuitable artefacts⁶.

Interpretation and related literature: At G, the reported negative connotation with the required cognitive effort for selection and adaptation could be seen as a more subtle instance of the "not invented here" (NIH) syndrome [87]. At U, the difficulties in access across project

⁵Due to differences in the questionnaires, some of the items present in CHR1 for case G are contained in FAR3 and RED1 in case U. Therefore, we include them in this paragraph.

⁶Unsuitable, e.g., on the conceptual level by incompatible abstractions and decompositions that increase the effort of reusing artefacts, but also on the business level, causing significant investments in low-return artefacts and eroding management trust.

boundaries are one of the main inhibitors to reuse, aligning with the observations of [60]. The lack of availability of reusables provided by other parties could, potentially, explain the perceived absence of NIH.

Difficulties due to reuse (CHR2, Figure 6.6)

Comparison agreement: Question CHR2 (L4, never, always) reports on difficulties encountered *due to* reuse. None of the presented options was of high frequency with statistical significance, nor did the respondents highlight significant other issues. The only difficulty in common in both cases, but not in a statistically significant way, is that of *dependency explosion* (id=67).

Comparison differences: In case G around one third of the respondents reported *ripple effects caused by changes in reused artefacts* (id=70) and a *decrease in code understandability* (id=69) as negative consequences of reuse (no statistical significance).

Interview data: In G, the complexity of the technical dependency structure was considered a challenge, however mitigated by the infrastructure and offset by the experienced gains. In U, participants mentioned a variety of harmful dependencies that they linked to negative aspects of reuse: technical ones (e.g. unstable interfaces, versioning dependencies), as well as organizational ones (e.g. diverging release cycles, delays due to coordination efforts).

Interpretation: Difficulties around reuse arise on several levels ranging from technical to organizational. In the context of organizational heterogeneity, non-technical dependencies impose a variety of challenges that inhibit beneficial reuse.

Related literature: Previous work suggests that additional rework due to reuse might not be a significant overhead [79]. Also, organizational heterogeneity is known as a challenge in the context of establishing development practices exceeding the scope of separate organizational entities [23]. Our findings support both of these suggestions.

Difficulties due to lack of reuse (CHR3, Figure 6.7)

Comparison agreement: Question CHR3 (L4, never, always) reports the negative consequences due to the *lack of* reuse. Both cases report regular occurrences⁷ of *inconsistencies* (id=75), *high maintenance effort* (id=76), and *increased development effort* (id=77).

Comparison differences: The only factor of significant relevance in case G is the occurrence of *duplicate implementations* (id=80). In U, no factor is of statistical significance.

Interview data: The item *duplicate implementations* is missing in case U; however, the interviews indicate multiple instances of this issue. In both cases, unwanted redundancies were not yet tracked systematically (or tracked at all).

Interpretation: Both companies to some degree incur the typical drawbacks associated with lack of reuse. However, the only aspect of significance is the one of duplicate implementations (which, arguably, entails some of the other drawbacks).

Related literature: Research has been addressing discovering and tracking redundancies in the form of code clones [125, 126, 63] and re-implementations [127, 39, 40]. At this point, several industrial tools exist that support structural (as opposed to semantic) detection approaches on an industrially viable scale [128]. Therefore, this issue might be mitigated within a reasonable timeframe.

⁷Around 50% of the participants report these occurrences; however, they are not statistically significant.

Benefits (SBF1, Figure 6.7)

Comparison agreement: For question SBF1 (L4, never, always), there is no statistically significant agreement.

Comparison differences: Only case G reports statistically significant high occurrences of the benefits *higher development pace* (id=104) and *less maintenance effort* (id=100).

Interview data: Participants in case G consider their reuse realisation as beneficial, i.e. fulfilling the goals behind their reuse approach. In case U, participants indicate that the aimed-for benefits of the given reuse strategy have not yet materialized as expected.

Related literature & Interpretation: Generally, improved code quality is one of the benefits associated with reuse [3]. We could not directly confirm this in our studies: In case G, participants already considered the produced artefacts as high quality and, thus, would not attribute this characteristic to reuse in particular. Instead, they considered their code quality as one of the main enablers of reuse. On the other hand, for G, we can confirm the gain in development speed [79, 80] and the decrease in maintenance effort [8].

In case U, the heterogeneity of development did not allow a clear assessment of the quality of the reused code. With respect to development speed and maintenance effort, our data provided no clear results.

Success factors (SBF4)

Comparison agreement: Question SBF4 was multiple choice in case G and free text in case U. Therefore, we can not provide a comparison based on our statistic test, but instead report the findings for each case separately.

Comparison differences: Respondents in case G report two statistically significant relevant success factors: *the high quality of artefacts* (id=132) and *supporting infrastructure and tools* (id=134)⁸. More than 50% of the respondents also mentioned *adequate abstractions* (id=129) as success factor. The remaining options (*direct communication culture* (id=130), *suitable incentives* (id=131), *well-defined process for reuse* (id=133), *stricter rules for dependency management* (id=135), *homogeneous development culture* (id=136)) for success factors were reported as significant and low relevance⁹.

In the case U, the free text responses reflected a negative tendency. However, we added the success factors from case G as potential improvements for case U (question SFB3 — L4, unimportant, important). All but two of these factors were reported as significant and high relevance by the respondents of case U.

Interview data: The interviews in G indicate that the accessibility of artefacts as well as the "safety net" and immediate feedback provided by an extensive tool support increase the inclination to build on existing solutions. In addition, automation is seen as the only feasible way to draw reusable artefacts from a large code base. Last, the benefits were tangible to

⁸Since the difference between these two answers is only one response, we consider the second item also as highly relevant.

⁹Note that the respective question in case G asks for the top 3 success factors, but this was not enforced by the software. As a result, most participants selected only one or two options, whilst others exceeded the number and selected up to four items. Since for this question none of the other options approaches even moderate frequency, the ranking seems comparable to the frequencies in U.

developers. This further motivated them to reuse during development. In U, the interview participants stress the negative effects of the heterogeneous development culture. As a result, they saw the need for one or more reuse champions that lobby a homogeneous development and reuse strategy across departments.

Interpretation: We consider this a noteworthy finding, as it indicates that developers are more willing to trust existing artefacts if they can thoroughly inspect and validate them, and they have faith in the process (and quality assurance) by which those artefacts were created. In addition, the potential improvements at U indicate the need for changes in the reuse and development processes.

Related literature: Parts of these findings coincide with literature: Kruger [2] considers abstraction the "essential feature of any reuse technique" and stresses the importance of an "integration framework" for reuse. Joss [84] reports management support, education of engineers, suitable incentives, tool support¹⁰ as relevant success factors for introducing structured reuse. Several studies (e.g. [13, 70]) suggest that, besides conceptual difficulties, the adoption of a reuse approach is significantly driven or inhibited by the organisational commitment towards the adoption process. Whilst in case G the most significant reported success factors were of technical nature (indirectly enabled by the organization), the results also align with [52], reporting the belief in benefits as motivator and success factor for reuse. In case U, the most important improvements included the mentioned organisational aspects. In terms of the definitions of reuse maturity and "good" reuse stated in the literature (see [120] for an exemplary reuse maturity model), case G challenges conventional academic assumptions: Whilst reuse is ingrained in the organisation, thorough planning and formal reuse assessment are not. However, due to their trust in their code and engineering quality, as well as their elaborate development support infrastructure, developers at G implemented a beneficial version of opportunistic ad-hoc reuse that matches exactly the company goals.

Summary of RQ2

For this research question, we found no statistically significant inhibitors or negative effects. However, technical incompatibilities and organizational heterogeneity as well as dependencies were identified as factors challenging beneficial reuse. Furthermore, participants in G report the challenge of identifying the right reusables from a large number of candidates and adapting them to meet current needs.

In terms of difficulties encountered due to lack of reuse, both cases agree on occurrences of duplicate implementations.

In the homogeneous and tool-supported context of G, a significant increase in development speed and a significant decrease of maintenance efforts are reported as benefits of reuse. In U, these effects have not been observed.

¹⁰It might be noteworthy to consider the differences in "tool support" w.r.t. the drastic advances of the technologies and paradigms used for programming and reuse. In a component repository, as e.g. proposed by the REBOOT approach in the 1990s (see [5, 19]), this relates to a basic protocol for repository and configuration management, whilst in today's setting, this refers to advanced code search and recommendation systems, central build and testing infrastructure etc. (see e.g. [27])

In G, the quality of the reusables and the supporting infrastructure are seen as clear success factors. Participants in U considered a tight network of personal connections across departments and reuse champions as crucial preconditions to successful reuse adoption.

6.7 Threats to validity

When studying development practices in specific companies, it is very challenging to generalise results even to contextually similar companies. Our study is not immune to this threat to the external validity, however we provided a detailed contextualisation of the two companies, which should serve as a framework for further studies to compare findings with ours. In the long run, such a contextualisation framework should help to provide a sensible generalisability of results. Regarding the internal threats, we identify three main issues:

Self-selection bias: The participation in our studies was optional and might have led to a biased selection of participants: in case G, most participants of interviews and questionnaire displayed a favorable attitude towards reuse, so it is possible that only engineers considering reuse as beneficial volunteered to take part. In case U, on the contrary, a significant number of participants vented their disappointment with the current state of reuse. In addition, the departments that, during the interviews, appeared least concerned with improvements did not participate at all in the questionnaire. To mitigate this threat, we attempted to select our interview participants in both cases from as many departments and as many different positions on reuse as possible.

Selection of participants: For each study, the participants of the interviews were sampled by convenience through personal contact in the respective company. This might have introduced a bias in the results. To mitigate this threat, we sampled the interview participants from different organizational units and different roles. We could not control the selection of the questionnaire participants. In case G, the respondents matched the expected distribution of departments. In case U, several departments did not contribute.

Heterogeneity of sample: The sample of the participants of the two studies differs in terms of the time spent at the respective company. This could potentially influence the knowledge of reuse practices and, thus, bias the responses. This sampling difference follows from two characteristics in which the studied companies differ: age of the company (G < 20 years, U > 50 years) and turnover of staff on projects (at G, developers moved between projects frequently, whilst at U staying with the same products for more than 10 years was not uncommon). However, we believe that this does not affect our findings: first, at G, reuse practices are homogeneous and streamlined with the development process. Newcomers are trained extensively to adhere to the given development practices (including the reuse practices). Therefore, we are confident that our participants at G were fully familiar with and aware of the reuse practices at their company.

Limitations of research methods: Our interpretation of the answers from the questionnaire at G rely on the assumption that non-selected items in multiple choice questions are considered equivalent to rate those options as irrelevant or of low usage. This assumption might not be completely true for questions in which participants were asked to select up to three items. How-

ever, the exact number of selected items was not software-enforced. As a result, respondents typically selected between one and three items and sometimes exceeded the number and selected four items or more. At U, the questionnaire design prevented this complication.

Subjectivity in responses: When designing the questionnaires, we aimed to capture the responses by means of precise measures. However, as we frequently asked participants about their experiences (e.g. on the perceived maintenance effort without reuse) and their agreement, we could not assume that they were equipped with the necessary tooling to provide objective measurements as responses. As a result, we resorted to more abstract, yet subjective, answer options (e.g. high, medium, low). Whilst these can only provide a tendency, this is a typical procedure for this kind of study (see, e.g., [80]) and, nevertheless, captures the perceived benefits/drawbacks of reuse in the respective cases. We, therefore, consider this aspect a minor threat to the validity of our conclusions.

Study design: As mentioned in Section 3.2, the study setup differs between the two cases: in case G, interviews were conducted during the same phase as the questionnaire. In case U, the interviews preceded the questionnaire. In this way, we could focus the content of the questionnaire and reduce it in size. We consider the change in design a minor threat to validity of our conclusions, as we retained the questions needed for the comparison.

Timeliness of second study: Despite our best efforts, the studies could not be conducted in a more narrow timeframe. However, we consider the impact of this delay as minor for the following reasons: The company studied in the first case is still developing in the same way (Inner Source), focussing on code reuse and trying to compensate drawbacks of the approach by more elaborate tool support. Since the company is stable and continuous in their approach, the data is still accurate. Therefore, we assume that the comparison is valid from this perspective.

6.8 Summary and conclusions

In this study, we reported on the comparison of two in-depth case studies on software reuse in industrial practice, integrating data from 138 professional developers of two companies, G and U.

The comparison highlights that reuse in practice occurs pragmatically in different flavours, however, mostly limited to source code. The technological potential has been partially embraced, rendering operational once infeasible approaches, such as repositories as source for reusable entities.

Successful reuse is tightly coupled to the company goals and ingrained in the development culture, also in terms of management and tool support. Perceived business success of reuse seems more determined by coherence between culture and approach than by the structuredness of the adopted approach.

In the homogeneous and coherent reuse setting, clear benefits for development and maintenance are reported. These benefits did not materialize in the heterogeneous setting.

Establishing any kind of systematic reuse in heterogeneous company and development contexts poses significant challenges and requires structured decision support. Further work is needed

to support companies in heterogeneous contexts to identify and install the required preconditions of suitable reuse approaches.

Part III

Guiding strategic reuse decisions in practice

7 | A pragmatic model for guiding reuse adoption in practice

Fragmented research and lacking guidance pose a significant challenge for practitioners aiming to select a suitable reuse approach or to improve their current practices. The resulting lack of understanding, resources, and planning often leads to a failure of reuse adoption. The work of this chapter aims to counteract this phenomenon by presenting a pragmatic model for guiding strategic reuse decisions.

Contents

7.1	Guiding reuse adoption in practice	102
7.2	Reuse adoption support model	103
7.3	Model overview	103
7.4	Structure of intent	104
7.5	Structure of the reuse facets	105
7.6	Application of RASM	114
7.7	Justification	115
7.8	Company Reuse Placement	118
7.9	Summary	121

7.1 Guiding reuse adoption in practice

As literature [13, 83] as well as the results presented in Chapter 5 suggest, underestimation of required adoption efforts, as well as inadequate combinations of reuse approach and company context, are two of the influence factors most detrimental to the success of any reuse adoption. Many other negative consequences, e.g., lacking support of management, unwillingness to provide the required human and technical resources, and overestimation of benefits, follow from these factors.

Whilst research has provided a multitude of reuse maturity [129, 130, 131] and reference models (e.g., [53]), they tend to focus on abstract organizational criteria and, frequently, only implicitly suggest which kind of reuse adoption compliance with their models will facilitate. Consequentially, concrete preconditions for different reuse approaches also remain obscure to practitioners. In addition, most of the proposed models lack empirical validation [132].

As far as reuse adoption is concerned, proponents of specific reuse approaches, such as SPLs or component-based reuse, have proposed mechanisms to probe an organization's adoption readiness for the respective approach (e.g., [98, 9]). However, in their isolation, these probing mechanisms do not support practitioners to tailor their reuse process from multiple approaches, potentially combining beneficial techniques that fit the organization's current need for improvement. In addition, potential downsides of the respective approaches are not reported with the same prominence as potential benefits, potentially raising unrealistic expectations in terms of return on investment.

To enable practitioners to make informed strategic reuse decisions, this chapter proposes a pragmatic model for guiding strategic reuse decisions across different reuse approaches. The model aims to incorporate existing probing mechanisms as well as to enable extension with further approaches.¹

The model elements have been derived from literature and validated by a model application with an industrial partner (see Chapter 8).

Use Cases: We currently envision the following two use cases for the application of our reuse adoption support model (RASM): On the one hand, it provides an early validation of a selected reuse strategy to reduce the risk of failure of a reuse adoption and the subsequent damages to the organization. It addresses this goal by enabling a rough congruence check between the motivation, goals, and time frame set by the adopting organization and the intent, characteristics, and adoption experiences reported of the selected reuse approach.

On the other hand, the model aims to support organizations that wish to improve their reuse practices. It addresses this goal by enabling a capability check of the organization with an array of reuse approaches by mapping the given context of the organization onto the prerequisites reported for the given reuse approaches. In this way, organizations can find the approach that is most in reach, in that sense "optimal", given their current capabilities. In addition, this assessment can provide the basis for reuse process improvement.

¹In the scope of this thesis, we limit the model instantiation to *Inner Source*.

7.2 Reuse adoption support model

Goal: The presented reuse adoption support model (RASM) aims to address the challenge mentioned above by supporting practitioners during the decision for and the adoption process of specific reuse approaches. In particular, it provides structured input for a feasibility assessment and, thus, highlights aspects that tend to be neglected in the usual decision process. Even in cases in which no given reuse approach is immediately feasible for a company, RASM allows to identify improvement points that can be used for an iterative long term planning of a reuse improvement effort.

RASM is a taxonomical model as it aims to structure the complexity of software reuse but does not currently support automatic predictions about the cost or return on investment of any given reuse approach. Rather it produces an immediate overview on the topics that need to be addressed by the organization attempting the adoption. Some of the topics can be supported by additional analyses (e.g. clone detection, library usage measurement, discovery of re-implementations) given an enabling infrastructure on the organization's part.

Intended audience: With this model, we target practitioners that plan to introduce a reuse approach to their organization or improve their current reuse practices.

Intended use: At the current state of the model, we suggest that a reuse expert, potentially external to the organization guides the application of the support model for two reasons: on the one hand, discussion on software reuse is often highly political (see Chapter 5) and benefits from neutral input. On the other hand, the model needs to be tailored to accommodate company-specific details.

7.3 Model overview

RASM contains two main components (see Table 7.1): the element *Reuse approach* captures the characteristics of a given approach as reported in the respective literature. This encompasses the *intent* inherent to the approach (motivation, goals, and scope, see Table 7.3), as well as its *realization* and *prerequisites* in terms of associated reusables, practices, tools, and organizational aspects. The realization of the approach, i.e., the way the approach is expected to manifest itself by its proponents, is mainly drawn from research literature describing the given approach. Contrarily, the prerequisites for adopting an approach frequently stem from papers reporting and analysing instances of adopting companies and their experience with the process. Consequentially, the model element reuse approach encompasses *adoption experiences*, such as profiles of the reported adopters, experienced benefits and challenges, as well as key adoption factors and risks reported in literature.

The component *company profile*, on the other hand, captures a *context snapshot* of current software reuse and software development at the organization that is applying the model. In addition, the element characterizes the *intent* of the *adoption initiative*, as well as a potentially pre-selected candidate reuse approach.

<i>Reuse Adoption Support Model</i>			
Reuse Approach		Company profile	
Characteristics		Context snapshot	
	Intent		
	Reuse facets		Reuse facets
Adoption reports		Adoption initiative	
	Intent		Intent
	Reuse facets		

Table 7.1: Structure of RASM - Overview

As the model aims to support judging the congruence between reuse approaches and organizations, the information captured by the two main components needs to be relatable to each other. We enable this by introducing two structures in the model that facilitate the description of reuse approaches and company profiles: *Intent* and *Reuse facets*.

Each reuse approach is created with a specific *intent*, i.e., a motivation to address a particular *issue*, aiming for specific *benefits*, and targeting a particular *scope* (see Table 7.3). The same can be said about reuse adoption initiatives (notwithstanding variations in details) and the model captures this, as displayed in Table 7.1.

The realization of a reuse approach tends to reflect in several aspects of software development. We term these aspects *reuse facets*. Literature suggests variations of the following categories: *artefacts*, *practices*, *tools*, and *organization* (see Table 7.4). Again, the same categories also suit the purpose of capturing a relatable instance of an organization's *context*, as well as describing known *adopters* and *key adoption factors*. Therefore, the context snapshot and the key adoption factors are structured accordingly. Table 7.1 displays the model elements that are instances of reuse facets. Section 7.5 further details on the structure of reuse facets.

7.4 Structure of intent

Intent, i.e., the purpose of an approach or adoption initiative, is represented in RASM on the one hand in the goals of the reuse approach, on the other hand in the motivation of the company's adoption initiative. The purpose of the intent element of RASM is to make explicit what benefits a particular approach can deliver. This gives organizations the opportunity to select the approach that is best to alleviate their current issues.

In particular, intent separates benefits into the following categories: *economic* and *organizational*. Both categories contain more fine-granular goals, such as, increasing competitiveness by means of shorter time to market, or improve resource alignment and knowledge transfer within an organization. In addition, intent also incorporates the organizational scope and the envisioned time to benefits. Table 7.3 gives an overview on the detailed structure of the intent element.

During instantiation, each aspect of intent is populated with the purpose and benefits that the given approaches target. In this way, an organization can understand which approach can deliver which benefits. This information then can be related to the outcome of the reuse facet assessment to come to an informed decision for a particular reuse approach.

7.5 Structure of the reuse facets

The concept of reuse facets form the core element of the model. They capture the different facets related to software reuse as described in the breadth of the reuse literature², as well as facets emerging from the empirical studies presented in the previous part of this dissertation. The main studies considered for building up the structure of the reuse facets are listed in Table 7.9. Reuse facets are hierarchical structures, composed of a number of elements. The goal of each element is to explicitly model the context of its parent elements. This fine grained structure provides the foundation without which no reasonable selection of a reuse approach can be effected [133].

Based on the literature, we identify four types of facets that are highly related to software reuse in practice: the type of *artefacts* that are at disposal for reuse or that are envisioned to be created for that purpose, the *practices* by means of which an organization currently develops its software and implements software reuse, the *tools* that support the development and reuse actions, and the characteristics of the encompassing *organization*. As noted by Rothenberger et al. [83], congruence between these facets and the selected reuse approach is a crucial factor for successful reuse.

Evidence on reuse influence factors tends to be scattered. However, we are aware of one study [80] that validates a postulated set [82] of success factors and relates them to the results of two additional studies [13, 1]. Therefore, we encode the state of evidence on the influence factors and highlight the factors within our reuse facets as displayed in Table 7.2.

Influence	Rationale	Symbol
confirmed	all studies confirm, more than one	↑↑
potential	one study indicates or majority of studies	↑
controversial	conflicting evidence or no clear majority	⚡
none	all studies decline, more than one	∅

Table 7.2: RASM - Encoding of the current evidence of Reuse Influence Factors according to [80]. The encoding is used to highlight the stage of validation of the respective elements in Tables 7.5 to 7.8.

7.5.1 Artefacts

Artefacts are a central facet of software reuse: they denote the assets available for reuse and are a key characteristic of each reuse approach. In the literature, the following aspects of artefacts are reported as relevant impact factors for software reuse [80]: the *origin of a reusable asset*, i.e., whether it originates from within an organization or not, the *kind of reused asset*, e.g., requirements documents, design documents, or source code of different types. In addition, the *technical compatibility of reusable assets* is reported as factor to be considered during selection of a reuse approach and the planning of the adoption [89]. This factor captures key aspects, such

²These facets have been collected from technical research papers, such as [80, 82, 1] as well as industry research papers, such as [13, 9, 89, 23].

as the architectural compatibility (to prevent or identify *Architectural Mismatch* that gravely affects the reusability of artefacts [88, 49]) or intent, scope of use and the related non functional aspects of the given reusables.

Table 7.5 illustrates the dimensions of the reuse facet *artefact*. The symbols express the degree to which the influence of a specific factor has been validated and confirmed by quantitative measurements from several sources.³

As the highlights, the *kind of reused artefacts* has hardened as factor for reuse success in the way that reuse of high abstraction artefacts, such as requirements and design documents, tend to indicate higher potential for benefits due to reuse [80, 82]. Due to a wide range of sources, also the reuse of source code is reported as beneficial (see, e.g., [9, 60, 59] as well as Chapter 4). For source code reuse, the reuse of packaged code, e.g., libraries, frameworks, and services, is regarded as more beneficial to long term reuse success than reuse of source text by means of *clone-and-own* reuse. Nevertheless, depending on the intent of an approach and an adoption initiative, also source text can be an appropriate reusable.

According to literature, *origin of reuse assets* likely influences the outcome of a reuse initiative [80]. Work presented in this thesis confirms this (see Chapters 4, 5 and 10): the reputation of the provider of a reusable asset has a significant impact on the trust and reuse willingness of the consumer. Furthermore, company external sources bring a series of legal issues that need to be considered on the consumer's side.

Technical compatibility of assets has been highlighted to facilitate reuse across units within an organization [89]. Absence or insufficient compatibility, on the other hand, quickly reduce the benefits that could, in theory, be obtained by reusing a functionally suitable and available asset, as the cognitive and organizational effort required to adapt the asset quickly outstrip the benefit. Since this detail is likely overlooked in the decision process for a reuse approach, we include it in the model.

7.5.2 Practices

Reuse approaches usually prescribe a certain set of practices, e.g., domain analysis, scoping and variability management in the context of SPLs, or peer review, automated unit testing, and continuous integration in the context of *Inner Source*.

These practices are effected in different phases of the software product life-cycle. For this reason, the reuse facet *practices* contains the *life-cycle phases* as part of its subcategories together with *documentation* and *quality assurance* as typical cross-cutting themes. In addition, the selected *reuse mechanism* as well as the *reuse process* with associated practices are explicitly listed. Lastly, the subcategory *development process compatibility* completes the facet. Its presence is motivated by the observation that certain reuse approaches, e.g., SPLs or *Inner Source*, require software development and reuse practices to be compatible in order to reach the envisioned benefits [8, 23].

³The values are resolved in Table 7.2.

The practices an organization employs to develop and maintain software, as well as to effect reuse, can have a significant impact on software reuse. The reuse facet *practices* captures these aspects and makes them comparable to the practices recommended or required by the given reuse approaches.

Table 7.6 depicts the factors of the reuse facet and provides a mapping to their source in literature and the empirical evidence of their impact.

The practices in the life-cycle phases are the standard activities of Software Engineering. In addition, the facet contains the items *reuse mechanism*, *systematic reuse process* [80], and *development process compatibility* [89] which are relevant for reuse adoption:

The element *reuse mechanism* captures the means by which software reuse is implemented in a given approach or a given organization. Of the various given options, e.g., copy-paste-modify, compile-time linking, product family development is the only one for which quantitative empirical evidence exists to support its benefits [80]. Nevertheless, the other options are widely found in practice and considered appropriate in the given context [60, 59, 9, 27].

Another element, *systematic reuse process*, has also been shown to positively impact software reuse success across several studies [80]. Depending on the situational context of further studies [13, 82], *software reuse measurement* has shown diverging results, whilst *configuration management of reusable assets* and *process maturity/quality models usage* have a mostly positive effect.

Corresponding to the element technical compatibility of artefacts in the facet *artefacts*, the element *development process compatibility* captures an important facet that has proven dangerous to ignore when attempting a reuse adoption on a company-wide scale [25]: Even in the presence of most technical and organizational prerequisites, heterogeneity in the development practices and release scheduling has annihilated the potential benefits of the selected approach. In another case, the consideration of the element has lead the stakeholders to reconsider a potentially hazardous adoption [89].

7.5.3 Tools

Depending on the reuse approach, the supporting *infrastructure* and *development tools* have a pivotal influence on a successful adoption and practice [9, 27]. Especially in the context of *Inner Source* reuse, numerous types of tools, such as *version control*, *build servers*, *bug trackers*, and *testing frameworks* are reported to enable and facilitate developing according to the approach [23]. In a wider context, the use of *CASE tools* has a mostly positive effect on reuse, whilst a generic use of *repository systems* produces mixed outcomes [80].

The *technical compatibility* of toolsets is an aspect derived from the technical compatibility [89]. As highlighted by [9], *homogeneity* of toolsets greatly facilitate the collaboration of developers on shared reusable items.

Table 7.7 depicts the elements of the facet.

7.5.4 Organization

The reuse facet *organization* captures essential aspects of the context of an organization. Based on impact factors in the literature, the categories *business context*, *development context*, and *skills* were derived.

Table 7.8 summarizes the structure of the facet.

The element *business context* captures aspects such as the *culture* of the organization (e.g., expressed in hierarchical structures, autonomy of managers and developers, etc.) and the *support* in terms of budget and human resources an organization provides for software reuse. Both aspects have found to impact greatly the success of reuse adoption [13, 9]. Furthermore, the element captures more strategic aspects, namely the *application domain*, the *type of developed software*, and *legal constraints*. *Application domain* has been established as a key impact factor to software reuse, whilst the *type of developed software* is a debatable factor [80]. *Legal constraints* were found to be impacting decisions regarding reuse as a mismatch with business goals prevented, e.g., relying on certain types of reusable assets [34, 27, 25].

With respect to an organization's *development context*, some approaches have specific prerequisites in terms of compatible *software development approaches* [9], whilst others suggest specific forms of structuring the workforce of a software organization [17, 8]. Overall, the quantitative empirical evidence of the general impact of these two factors is contradictory and requires further investigation [80]. In addition, *process compatibility* between the units of an organization targeted for reuse adoption has been established as an important impact factor [89, 25].

Lastly, the organizational and individual *skills* have been reported to impact the success of reuse adoption [70]. On the *organizational* side, *software reuse education* is an empirically controversial factor. *Project team experience*, on the other side, has been identified as a potential impact factor, depending on the respective organization context [80].

The *individual skills* of *managers* and *developers* are partly mentioned in the literature describing software reuse approaches (e.g. [9]). The required skill sets include technical skills, as well as change management and social skills.

<i>Intent</i>	Constituents
Motivation	current issues economic organization
Goals	economic benefits competitiveness quality organizational benefits knowledge transfer resource alignment
Scope	organizational units single division multiple divisions all divisions time to benefits short term medium term long term

Table 7.3: Factors and constituents of the *Intent* element of the RASM.

<i>Reuse facets</i>	Constituents
	artefacts practices tools organization

Table 7.4: *Reuse facets* of the RASM.

<i>Artefacts</i>	Constituents
Kind of reused assets [80] ↑	requirements ↑ use cases workflows design ↑ architecture templates feature models UI templates source text ↑ snippets classes (sub)systems packaged source code ↑ components libraries frameworks services
Characteristics of candidate reusables [9]	pre-existing initial runnable implementation initial architecture fully specified fully implemented created for reuse business value domain independent functionality domain specific functionality product group specific functionality scope of use stakeholders potential users
Technical compatibility of assets [89]	architecture compliance scope of use NFRs aligning purpose modularity
Life expectancy	reusable reusing products
Origin of artefacts [80] ↑	company internal within department internal third party company external Open Source commercial provider private third parties

Table 7.5: Factors and constituents of the *Artefacts* facet of the RASM. The symbols encode the stage of validation of the factors (see Table 7.2).

<i>Practices</i>	Constituents
reuse practices [80, 9, 41, 8, 59]	<p>mechanism</p> <ul style="list-style-type: none"> copy-paste-modify hard copy duplication compile-time linking branching service composition product derivation by means of variation points ↑ <p>process</p> <ul style="list-style-type: none"> ad-hoc selection and integration of reusables opportunity-driven selection and integration of reusables strategic selection and integration of reusables systematic company-wide reuse process ↑ reuse measurement configuration management of reusables ↑ quality model usage ↑
requirements engineering practices [8, 9]	<ul style="list-style-type: none"> identification of commonalities and variations between products integration of different stakeholder needs strategic prioritization tracing
design practices [8]	<ul style="list-style-type: none"> feature modeling creation of reference architecture
development practices	<ul style="list-style-type: none"> iterative incremental serial contribution management
quality assurance practices [8, 9, 27]	<p>reviews</p> <ul style="list-style-type: none"> architecture reviews peer review of source code architecture compliance <p>test</p> <ul style="list-style-type: none"> automated unit tests automated integration tests system tests
maintenance practices	<ul style="list-style-type: none"> task assignment regression testing prioritization of tasks cost estimation
release and deployment practices [9, 89]	<ul style="list-style-type: none"> continuous integration frequent releases configuration management
documentation practices	<ul style="list-style-type: none"> code comments concise and consistent naming in code tracing to requirements and tests descriptive documentation of functional purpose descriptive documentation of extra functional guarantees and limitations
homogeneity of practices [89, 9]	<ul style="list-style-type: none"> reuse practices development practices maintenance practices quality assurance practices documentation practices coordinated product release schedules coordinated integration schedules

Table 7.6: Factors and constituents of the *Practices* facet of the RASM. The symbols encode the stage of validation of the factors (see Table 7.2).

<i>Tools</i>	Constituents
development tools [9, 27, 41]	version control CASE tools ↑ IDEs code recommender code search engine issue tracker build server supporting continuous integration
infrastructure tools [9, 27, 41]	documentation wiki Q&A forums list archives communication developer mailing lists user mailing lists IRC channels
homogeneity of toolset [9, 89]	homogeneity

Table 7.7: Factors and constituents of the *Tools* facet of the RASM. The symbols encode the stage of validation of the factors (see Table 7.2).

<i>Organization</i>	Constituents
business context [80, 9, 39]	culture management communication support top management middle management resources human resources budget application domain ↑ type of developed software ↵ legal constraints life cycle duration strategy reuse vision
development context [89, 80]	software development approach ↵ traditional lean workforce software organization ∅ product development reusables development total homogeneity practices tools reuse vision culture
skills [80, 9, 8]	organizational project team experience ↑ maturity software reuse education ↵ individual managers developers
reuse-related roles [27, 9, 134, 135]	consumer roles producer roles coordinator roles
project management [134]	process management process alignment between reusables development team and using/contributing units process for developing shared projects project planning long-term vision of shared asset coordination of contributions monitoring and steering quality assurance of contributions tracking of feature evolution human issues manage transparency

Table 7.8: Factors and constituents of the *Organization* facet of the RASM. The symbols encode the stage of validation of the factors (see Table 7.2).

7.6 Application of RASM

To apply RASM in practice, we envision the following steps: first, we ensure the model is instantiated with the reuse approaches of interest. Second, together with the organization, we determine the potential benefits they hope to realise by means of the reuse adoption. Third, together with the organization, we determine the potential adoption effort required by the given reuse approaches. Last, we compare the options available to the organization: for each approach, we compare the potential benefits it covers with the estimated effort required for adoption. In this way, an organization can base the decision for a specific approach on a structured comparison that accounts for the given context.

7.6.1 Instantiation for reuse approaches

In its current form, RASM contains elements derived from a variety of reuse approaches and studies on reuse impact factors. When preparing a model application, for each element of type intent and reuse facet, an instantiation is required.

To give an example with respect to the reuse facets, this means that within the four categories all subcategories applicable for the given approach are filled with more detail whilst all irrelevant (or non-reported) subcategories are marked as *not applicable* (or *no data*). In this way, the comparability of the different reuse facet instances is ensured.

With respect to the given reuse approaches, it would be sufficient to instantiate the respective intents and reuse facets once and, in that process, to create a knowledge base that is at disposal for future application of the RASM. As more research evidence on reuse impact factors, reuse approach adoption, or new reuse approaches, becomes available, it can be inserted in the knowledge base and also used to refine the model.

For approaches that have not yet been part of an instantiation of the model, we use the intent and reuse facet structures to build up instantiations, capturing *required*, *recommended*, *neutral* and *discouraged* realization aspects as reported in the experience reports in the literature. *Required* implies that, according to literature, constituents need to be present in a specific form, otherwise the adoption can not succeed. *Recommended* implies that, according to literature, complying with the suggested expression of the given constituent facilitates adoption and increases the benefits that the approach can provide. *Neutral* marks aspects that are compatible with the given approach but, according to literature, have no enabling effect on adoption and reuse success. *Discouraged* highlights practices that, according to literature, should be avoided because they counteract the intent of the approach and can hinder its success.

7.6.2 Determining the potential adoption benefits

The goals of most reuse adoptions tend to be abstract, such as, decrease of time-to-market or decrease of development and maintenance effort. Virtually every reuse approach claims to fulfil these benefits; however, from a detailed perspective, the degree and the means by which this is achieved differ greatly, impacting the applicability with respect to a given organizational context.

To capture which concrete benefits each approach can provide, RASM, in the intent structure, decomposes the abstract goals into more concrete aspects. During an instantiation, the intent of an approach is populated with the evidence of the obtained benefits reported in the literature (*strong, medium, low realization*).

During the assessment, an organization is then offered the possibility to highlight which of the addressed aspects is currently causing issues and with which severity (*low, medium, high*). Based on this assessment, an organization can compare what it could potentially gain from adopting a given approach and prioritize accordingly.

7.6.3 Determining the potential adoption effort

Depending on their current situation, the effort required for an adoption of a given reuse approach varies significantly between different organizations. To account for this fact, RASM collects the correspondence between the prerequisites of reuse approaches and a company profile as follows:

Given the RASM instantiation for the desired approaches, organizations should fill in their current position for each constituent of each facet and compare their values with the ones required by each approach. This comparison is projected on a three-point scale, capturing agreement, discrepancies, or conflicts between company profile and approach realization. In this way, RASM provides a direct overview on open issues. These issues then can be prioritized in different ways: to obtain an estimate of the total effort required for adoption, all elements marked as *conflict* or *discrepancy* as well as *required* can be counted. This assessment can, subsequently, be compared to the list of issues mitigated by the given approach to allow for a balanced decision on adoption.

Once the decision for adoption has been taken, priorities might change: to avoid long delays to success of new measures, an organization might be interested into quick wins, i.e., steps that can be performed with moderate effort and still drive the adoption process towards its goal. In this case, the elements classified as *discrepancy* and *required* can provide a suitable starting point.

Likewise, adopting organizations can assess the potential value of elements marked as *discrepancy* and *recommended*: if any of these could be resolved with little effort, they might be candidates for easing adoption or improving the benefits obtained by means of the reuse approach.

7.7 Justification

Proposing a new Reuse Adoption Support Model requires justification with respect to three questions: *Is there a need for a new model? What is the difference to already existing models? What justifies the content of the model?*

As the motivation of this Chapter has already answered the first question, the remainder of this Section proposes justifications for the remaining two.

7.7.1 Creation of a Reuse Assessment Support Model

Previous work on software reuse has proposed a variety of models to measure organizations readiness for adoption, as well as their reuse maturity. Instances of these propositions are the Reuse Assessment Models, e.g., by Wartik and Davis [42], Reuse Reference Models, e.g., by Nada and Rine [85], as well as Reuse Maturity Models, e.g., by Koltun and Hudson [130], Davis [131], Garcia et al. [120].

The following points of criticism are found towards these models: in general, only a small amount of the proposed models were industrially validated [132]. Usually, there are no reports of application (e.g. [130]), it remains unclear which reuse approaches they are supposed to support (e.g. [42, 130]) or they are targeted only to one approach (e.g. SPLs [85]). In addition, the models remain highly abstract in terms of their categories, which makes it hard to deduce actions for adoption.

Consequently, we find the following justification for creating a new model: With RASM, we provide an integrated view of reuse across numerous different approaches and models, and enable practitioners to find the best fit for their current situation instead of aiming for unrealistic ideals. We strive for comparability of different approaches, clear overview of the reuse facets and the intent covered by the given approaches. RASM supports to determine a detailed delta between a given company profile and the given reuse approaches. It, thus, allows to assess the potential benefits and efforts for an informed decision, and to deduce a concrete trajectory for adoption. RASM can be integrated with more specific assessment models (e.g. Rehesaar's Capability Assessment Model for Component Reuse [98]) that can be applied once an organization has selected an adequate approach.

7.7.2 Reuse facet dimensions

The deduction of the reuse facets followed a detailed literature review to understand the different reported dimensions of reuse. As can be seen in Table 7.9, the structuring dimensions overlap on key points (mainly technical aspects of reusables and the importance of organizational aspects) but vary noticeably in details between the different sources⁴. To ensure that our model is able to capture highly divergent reuse approaches, we included in the reuse facets structure all the dimensions mentioned in the literature. In addition, we included the detailed factors, summarized by the dimensions in the original sources, in the reuse facets.

⁴For the sake of illustration, we selected literature that covers the broadest variation in terms of dimensions.

		Selected literature												
		Frakes and Fox [1]	Prieto-Diaz [136]	Lucrecio et al. [80]	Sherif and Vinze [70]	Stol et al. [9]	Morisio et al. [13]	Koziolek et al. [89]	Pohl et al. [8]	Basili et al. [133]	Davis [131]	Koltun and Hudson [130]	Rine and Nada [85]	Bauer et al. [27, 25, 41]
Reported reuse dimensions	<i>Artefacts</i>					✓			✓					✓
	Performance		✓											
	Technical Compatibility							✓						✓
	Technical	✓	✓									✓	✓	✓
	Technological			✓					✓		✓		✓	✓
	Quality												✓	✓
	Asset Development Factors										✓			✓
	Application Development Factors										✓			
	Scope								✓					
	<i>Practices</i>					✓			✓	✓			✓	✓
	Process			✓			✓		✓	✓	✓		✓	✓
	Process Compatibility							✓					✓	✓
	Measurement									✓			✓	✓
	<i>Tools</i>					✓							✓	✓
	<i>Organizational</i>			✓	✓	✓			✓	✓	✓		✓	✓
	Managerial	✓	✓				✓				✓	✓	✓	✓
	Legal	✓										✓		✓
	Economic	✓	✓						✓	✓		✓		✓
	Business			✓			✓		✓				✓	✓
	Cultural		✓			✓						✓	✓	✓
	Technology transfer		✓											✓
	Individual				✓									
	Commitment						✓							✓
	Human factors						✓							
	Reuse understanding						✓							
	Structure								✓					✓
	Dedicated workforce			✓					✓					
Domain								✓						
Strategy							✓	✓	✓					

Table 7.9: This table presents an overview of the different reuse facets reported in literature. Facets in bold face are reported as structuring **reuse dimensions** in the respective papers. Facets in bold face and italics denote the final *reuse facets* selected for the model. The additional facets are reported as influence factors and included for a more comprehensive justification of the heterogeneity of the reuse facets in the model.

7.8 Company Reuse Placement

Motivated by our experience with reuse in practice, we aimed to compare the reuse proficiency of our cases on a high level: practitioners were interested, on the one hand, to get a compact and brief estimate of areas of their current reuse performance that might be problematic. On the other hand, they were curious to compare their placement with the one of other organizations and to see how close those were to the state of the art. From the research perspective, we were interested to see whether the differences we observed could be captured in a light-weight placement approach. Based on the detailed structure of RASM, we, thus, derived a company reuse placement that acknowledges the realization of reuse within an organization according to the four reuse facets presented above.

7.8.1 Placement structure

Within each facet, an organization can be placed in one of the three categories *basic*, *intermediate*, or *advanced reuse capability*. The placement follows according to two categories per facet: the *support for reuse* currently provided by the realization of the factors within the organization and the *homogeneity* of this realization across the organization. The placement reflects the general tendency of the given organization at a high level of abstraction. Local deviations are possible.

For the reuse facets *practices*, *tools*, and *organization*, the general support for reuse is expressed on an ordinal scale (*neutral*, *supporting*, *integrated*). They capture the *degree* to which the factors' realization at a given organization integrate with reuse support. For the reuse facet artefacts, the *type* of artefact was selected as discriminatory factor, as it is the most characteristic reuse support factor according to literature. The three values for artefact type are *low level source code* (e.g., snippets), *high level source code* (e.g., libraries), and *all types of artefacts* (e.g., including requirements, architecture templates).

The homogeneity of the facet realization is expressed by the attribute distribution of facet. The following values determine the company placement: *local* expression of the realization is characteristic for basic reuse capability, a *distributed* realization for intermediate reuse capability, and a *global* realization for advanced reuse capability.

7.8.2 Placement example

Figure 7.1 presents the structure and, for illustration, places the cases G and U (see Part II), as well as another case in progress, I, in the structure.

Company context The context factors, as well as the reuse implementation of G and U are known from Chapter 6. We briefly describe the context of T here: I is an international company, with a headcount of approximately 4000 employees. The organization has grown over a decade and incorporated other organizations. Software of T is developed in a distributed manner. Consequentially, an elaborate development infrastructure is established that enforces a rather homogeneous development process, including governance for quality assurance (in the form of code reviews and tests).

Reuse at T is approached in a strategic way; however, due to the heterogeneity in development practices that had to be overcome due to the previous integration of other companies, reuse practices have been devised on a basic level of process and tool support. The products of T indicate potential for the development of a product line; however, the effort to acquire the necessary skills to build them according to the state of the art currently is judged too high. Instead, mechanisms such as duplication of source code into variation folders or *clone-and-own* reuse are employed. They bring benefits to the organization in terms of development speed. Their effect on maintenance, so far, has not been assessed.

Placement description Within the facet artefacts, all three cases display reuse of source code on different levels of abstraction. Therefore, the companies are placed in the categories *basic* and *intermediate* for the type of artefact. In the distribution of artefact use, the three cases differ: in U, the distribution of artefact use is mostly local. In I, the distribution is distributed over several products. In G, the distribution of artefact use is homogeneous on a global scale.

With respect to the facet practices, U largely displays practices that are neutral to reuse. In addition, practices at U are heterogeneous as most divisions follow their own local development styles. This gives U the placement *basic* for both attributes. At T and G, practices are in place that support reuse, e.g. a clear governance of development and quality assurance practices and a shared vision of reuse. This places T and G in the category *intermediate* for support. In the case of I, these practices are spread to several organizational units; however, some departments are still in the progress of adopting them (*intermediate* placement). At G, the practices are followed globally (*advanced* placement).

As far as tools are concerned, U has no significant infrastructure that supports reuse. Also, most departments so far have their particular local tool-chains and configurations. This positions U in the category *basic*. At I, the available infrastructure supports reuse and is used by most teams. This positions T with *intermediate*. At G, infrastructure dedicated to facilitate reuse exists and is in use globally, positioning G in the category *advanced*.

On the organizational level, support for reuse at U is neutral (*basic*) to supporting on a distributed level (*intermediate*). At I, there is support for reuse from an organizational perspective; however, it remains unclear from which positions in the organization it is driven. This places T in the *intermediate* category. At G, reuse is supported globally (*advanced*), but not a top priority (*intermediate*).

Summing up, U generally falls within the category *basic reuse capability*, T tends towards *intermediate reuse capability*, and G is mostly placed within *advanced reuse capability*. This placement largely coincides with the benefits reported from the companies (see Figure 7.1, bottom): At U, only a limited improvement for development and maintenance is reported. T self-reports moderate improvements for development and limited improvements for maintenance. G reports significant increase of development speed as well as less maintenance burden. With respect to code quality, U reports limited improvements. T and G report a moderate effect; with G claiming high quality code rather as enabling factor to code reuse instead of a result thereof.

As we study further companies, we are interested to observe their placement and reported benefits as a first step towards a reuse capability benchmark.

Reuse dimensions	attributes	Basic reuse capability	Intermediate reuse capability	Advanced reuse capability
artefacts	type of artefact <i>Company placement</i>	low level source code <i>U, T</i>	high level source code <i>G, U, T</i>	all types of artefacts
	distribution of artefact use <i>Company placement</i>	local <i>U</i>	distributed <i>T</i>	global <i>G</i>
practices	support for reuse <i>Company placement</i>	neutral	supporting <i>T, G</i>	integrated
	distribution of practices <i>Company placement</i>	local <i>U</i>	distributed <i>T</i>	global <i>G</i>
tools	support for reuse <i>Company placement</i>	neutral	supporting <i>T</i>	integrated <i>G</i>
	distribution of tools <i>Company placement</i>	local <i>U</i>	distributed <i>T</i>	global <i>G</i>
organization	support for reuse <i>Company placement</i>	neutral <i>U</i>	supporting <i>T, G</i>	integrated
	scope of support <i>Company placement</i>	local <i>U</i>	distributed <i>U, T</i>	global <i>G</i>
Company placement		7U, 1T	2U, 8T, 3G	5G
Reported benefits	limited effect	<i>U</i>	moderate effect	significant effect
	increased development speed	<i>U</i>	<i>T</i>	<i>G</i>
	decreased maintenance effort	<i>U, T</i>		<i>G</i>
higher quality code	<i>U</i>	<i>T, G</i>		

Figure 7.1: Company Reuse Placement according to the four reuse facets.

7.9 Summary

In this Chapter, we presented RASM, a pragmatic model for reuse adoption support in practice. It constructively addresses the challenge of selecting an adequate reuse approach for adoption in practice by providing structures that relate the intent and realization of reuse approaches to a given company profile and adoption initiative. The model is based on a literature study and, on the one hand, allows a detailed assessment of congruence between reuse approaches and a company context by means of its intent and reuse facet structures. On the other hand, it provides a fast estimate of an organization's reuse capabilities by means of its company placement. Both methods allow practitioners to identify areas that are critical for improving their reuse capabilities.

8 | Applying the decision model in practice

This Chapter presents a proof of concept of model application in practice with company U. The goal of U was to prepare an introduction of Inner Source to support shared company-wide development of reusables. In this case, their goal of the research collaboration was to identify discrepancies between the approach and their current company profile. The outcome of the assessment should form the base for a detailed roadmap for Inner Source adoption. For this case study, we instantiated RASM for Inner Source and conducted a full day workshop at U. In the course of the workshop, participants worked with the model instantiation and derived next steps for each reuse facet. Direct participant feedback and the results of an anonymous on-line survey suggest that RASM supported tackling the complex issue of Inner Source adoption by structuring the topic in meaningful facets and ensuring that important aspects were explicitly considered.

Contents

8.1	A proof-of-concept application of RASM in practice	124
8.2	Background of model application at U	124
8.3	Model application	125
8.4	Results for case U	127
8.5	Limitations of evaluation	136
8.6	Next steps	136

8.1 A proof-of-concept application of RASM in practice

Goal The goal of this chapter is to study the application of the reuse adoption support model (RASM), presented in Chapter 7, in practice. In particular, we aim to understand whether an application of RASM can support practitioners in adopting new reuse practices or assessing potential improvements of their current reuse practices.

We expect this support to occur in the following form: the model application should help practitioners to structure the assessment and discussion of reuse options, to identify critical points for adoption that potentially would have been overlooked by a less structured adoption process, and to provide the basis for an informed and actionable adoption decision. Furthermore, we assume that the results of the application provide value for the operationalization, once the decision for an adoption has been taken.

As the model is designed to be tailored and applied to various contexts and by various roles, we address our goal by a partial model application in specific real-world industry scenarios that are described in the remainder of this chapter.

8.2 Background of model application at U

This section describes in brief the main characteristics of the organization at which the model is applied, the overall goals that drive the given organization's reuse practices, and the specific context and goals of our case. It, furthermore, details on how we instantiated the model, how we performed the intervention, and how we partially evaluated the model's usefulness.

Organizational context As described in detail in Chapter 3, Section 3.4, U is a national company of 6000 employees that work in a hierarchical environment and build on products historically grown since the 1960s. As a result, on the organizational side, numerous development styles have emerged in the different product groups. On the product side, a broad mix of technologies and styles need to be integrated and pose a challenge for future development. In addition, U's systems tend to reach a life time of 20+ years in the market, thus requiring significant effort for maintenance.

Reuse goals As a result of their products' longevity, U is interested in saving maintenance costs. Furthermore, U wants to increase collaboration between business units to create shared reusable assets and prevent redundant implementations of domain-specific utility functionality. To reach these goals, upper management has decided to introduce *Inner Source*.

Case description The present study followed up onto a previous one, described in Chapter 5, and, driven by a fresh management initiative, had the goal to support the former platform team to learn from the results and prepare the adoption of *Inner Source*.

In a first iteration, the data collected during the previous study was re-analysed for relevant learnings for the platform team. In a workshop with 10 members (1 architect, 2 managers, 7 senior developers) of the platform team, we triaged the learnings and identified topics that required a future intervention.

In particular, the need for a structured and detailed discussion of the *Inner Source* strategy became apparent. This motivated the instantiation of RASM for *Inner Source* and U.

Case study goals With this intervention, U aimed to identify aspects of *Inner Source* adoption relevant for their company. This coincides with one of the two use cases of RASM, namely an early validation of the selected strategy to guide the adoption of a reuse approach.

From the research perspective, we aimed to validate whether the model is capable to highlight conflicts between the current state of the company and the envisioned approach. In addition, we were interested to see whether it could support practitioners in identifying missing items, structuring the discussion on the topic, prioritizing the identified open issues, and developing concrete next steps.

Study description In this case study, we instantiated the model for *Inner Source* and U. A tabular representation is included for reference in the Appendix, Section 12.2.

In a one-day workshop, we invited our ten participants¹ to work with the model to improve their understanding of *Inner Source*.

Evaluation process To validate the usefulness of RASM, we asked the participants for the following: first, we invited them to write down all strengths, weaknesses, opportunities, and threats of *Inner Source* adoption within their company that occurred to them during the input talk. We then clustered these points for a ground truth of relevant points.

Second, during the hands-on session, we asked them to highlight new and relevant factors in the tables, which we collected afterwards.

Last, we provided the participants with a short on-line questionnaire, containing two Likert batteries with a total of nine items, assessing the perceived helpfulness and accessibility of the model, and one free text form for additional comments. Figures 8.3 and 8.4 report the Likert items and the selected answers.

8.3 Model application

In this section, we describe at first how we instantiated RASM for *Inner Source* and then provide the details of the study execution at U.

8.3.1 Instantiation for Inner Source

For our case at U, we instantiated the model for the *Inner Source* approach. The instantiation strongly relies on the available literature on the topic (chiefly on [9, 23, 97, 135, 134], as well as the work presented in Chapters 3 to 6)². The mentioned literature was coded according to the themes of the model and the respective categories were filled.

To make the information accessible to the workshop participants, we created a tabular structure for the elements *intent*, *artefacts*, *practices*, *tools*, and *organization*. Figure 8.1 shows an

¹Three of the participants of the first workshop were not present. However, three additional participants joined.

²The resulting tables of the instantiation are included in the Appendix, Section 12.2, for reference.

excerpt of the tools facet table. The table consists of an *approach-specific* part and a *company-specific* part.

The approach-specific part is listed on top of the Figure and contains the following information: on the left hand side, the *aspects* of the facet are listed together with the respective *constituents*. The next column details on the requirements of the *approach* with respect to the given constituent and provides a brief explanation. If needed, these details can be enriched by highlighting additional *implications* of the constituent onto other factors in the model. The column *values approach* provides an assessment of the relevance of the respective factor to the success of adopting the approach.

The company-specific part or the table (at the bottom of the Figure) structures the information of the comparison as follows: the column *values company* captures the current state of the constituent at the organization. When comparing the content of the column with the value given by the approach, three values can emerge: *conflict*, i.e., the given factor is not present or currently contradicting the required form at the adopting company, *discrepancy*, i.e., the factor is present but does not yet align with the form required by the approach, or *agreement*, i.e., the form in which the factor exists at the company coincides the with the form required by the approach. This categorization explicates the potentially challenging aspects of the adoptions.

The two last columns capture whether the respective element is *new* to the participants or has already been considered in the adoption planning and whether it is considered *relevant*. This column is added as additional prioritization mechanism as it draws attention to factors that have so far been overlooked and might require additional attention.

To create the instantiation, the approach-specific part of the model was filled in by researchers, based on the literature available. In particular, the assessments of *values approach* incorporate the *adoption experiences* that are reported in the literature. Contrarily, the company-specific part was left blank to be filled in by the participants.

8.3.2 Study execution at U

At U, we used the RASM instance for *Inner Source* in a workshop that was structured as follows (see Figure 8.3.2):

Initially, we collected the participants' associations and thoughts on the *Inner Source* initiative and summarized their main points. We then gave an introductory talk on *Inner Source*, presented an overview on RASM, and introduced them to the tabular representation of the model instance.

Subsequently, we invited the participants form groups of two or three people, select one of the four reuse facets, and manually evaluate the respective factors. In addition, they should highlight for each factor whether they had so far considered it in their company internal discussion of *Inner Source* and whether they considered the factor as relevant. Figure 8.1 shows an excerpt of the setup of the tool facet. During this hands-on session, the researcher was present to guide the participants on demand.

Facet	Tools	Constituents	Details approach	Implications on	Values Approach
					<i>Inner Source</i>
	development tools				
		version control	The source code of an Inner Source project is accessible openly in the respective version control system to allow potential users and contributors full access.		mandatory

Values Company	Conflict <i>Risk for adoption</i>	Discrepancy <i>Potential risk</i>	Agreement <i>Potential enabler</i>	New element?	Relevant element?
Which development tools are currently in use in our company?					

Figure 8.1: Excerpt of the tabular representation of the RASM tool facet for *Inner Source* used during the workshop. The graphic shows the columns and the first row of the table.

After the hands-on session, each group summarized their findings and proposed actions and presented them to all participants for discussion. At the end of the workshop, the group and their managers selected the next steps from the proposed actions.

As follow-up to the workshop, we prepared a short on-line questionnaire to collect anonymous feedback from the participants (see Figures 8.3 and 8.4).

8.4 Results for case U

In this section, we report the results of the workshop with respect to our research goals. We structure this section into results that we obtained via observation or direct feedback during the workshop, results gained from evaluating the participants' assessment of the reuse facets of the instantiation, and results obtained from the anonymous post-workshop feedback questionnaire.

8.4.1 Observations from the workshop

Introductory session: During the introduction of the workshop, preceding the input talk, it became clear that the *Inner Source* adoption was considered an extremely controversial topic by the participants: on the one hand, strong advocates of the initiative were present:

"We really should do this. It already happens successfully on a really small scale with contributors from other departments."

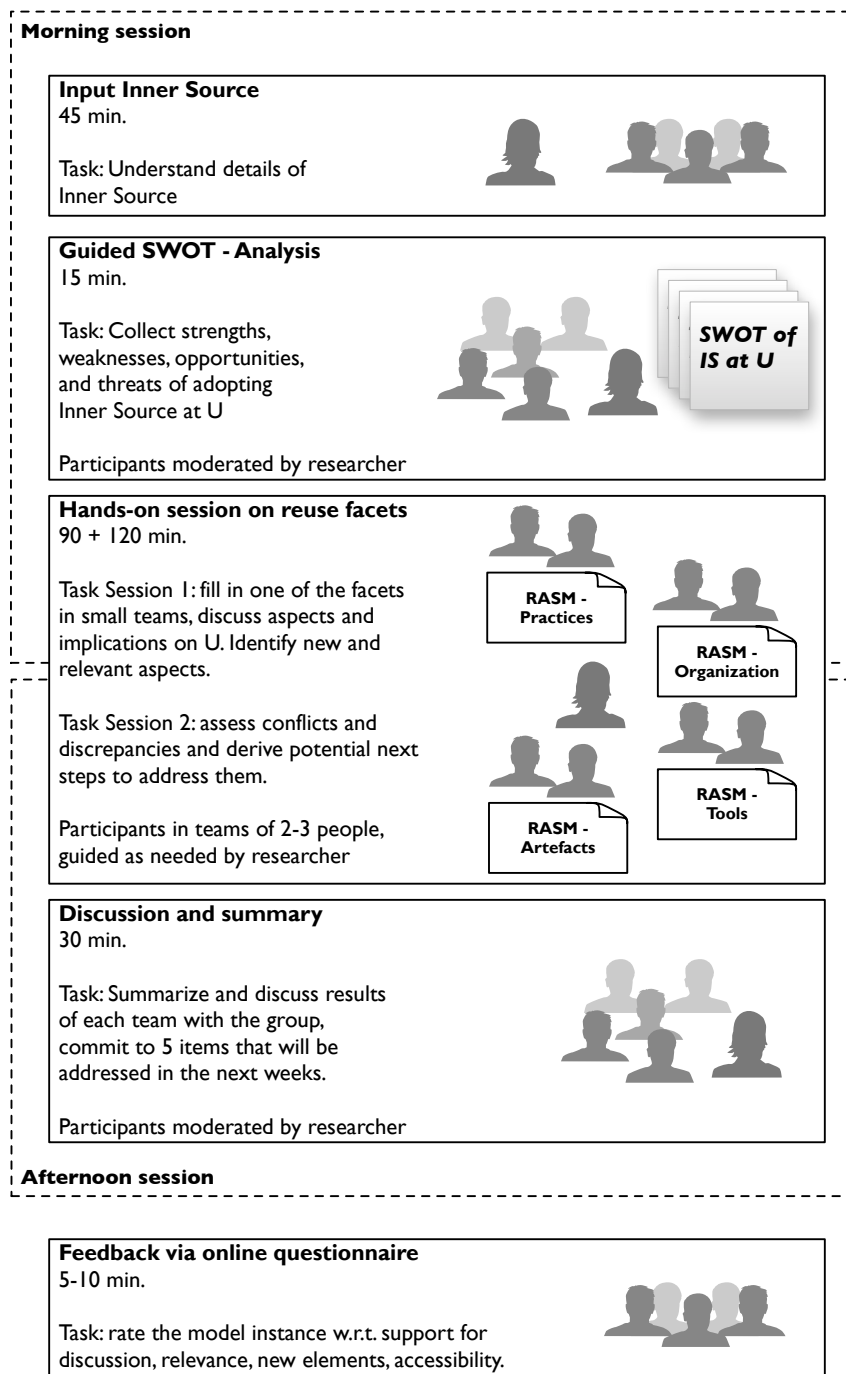


Figure 8.2: Details of the RASM application for *Inner Source* at U. The elements in the dashed boxes present the schedule of the one-day workshop. The single elements detail on the time given to participants for the particular tasks, the concrete tasks that were effected, and the roles of the participants and the researcher during the given tasks. The feedback questionnaire was provided on-line and could be filled in until a week after the workshop.

On the other hand, outspoken sceptics voiced their concerns:

“In this company, this will never work. In Open Source, people contribute because they are enthusiastic about a technology, they want to be part of it. With our technologies, we will never achieve this identification.”

Both sides were emotional about the topic and motivated to see their viewpoint confirmed by the workshop. However, so far, either of them was focussed on a particular and narrow aspect of the approach (e.g., the perceived unattractiveness of the technologies or the presumed incompatibility of *Inner Source* with the present corporate culture, vs. a small number of successful instances of cross-department development efforts).

Discussing their positions within the team lead both groups to consider each other’s points of argument.

Guided Strengths-Weaknesses-Opportunities-Threats analysis: Following the input talk, we collected the participants’ statements on the strengths, weaknesses, opportunities, and threats they associated with the adoption process. Even though they identified positive aspects for all reuse facets, the risks and threats were more dominant. In particular, the present organizational stance towards the shared development entailed by *Inner Source* was seen as most significant risk for success of *Inner Source* adoption:

“We can get the infrastructure and processes in place, but if we cannot obtain strong commitment from top and middle management, this will never be a success.”

In addition, the unwillingness to distribute clear responsibilities and announce concrete goals for the transition surfaced as significant threat to the adoption.

“We have been talking about this for months now, but still no one is really responsible and assigned to drive the topic.”

As a result, the investigations into *Inner Source* had not been structured and detailed so far.

Reuse facets hands-on session: During the hands-on session, we observed the following: Generally, the participants agreed with the different reuse facets. However, they were surprised at the number of factors that, according to literature, should be considered when adopting *Inner Source*.

During the course of the workshop, the participants used the structure of the respective reuse facet to guide their discussion. They proceeded in a top-down way and discussed the meaning of each factor with respect to their organization. Whenever needed, they addressed the researcher with clarification questions or to explain their reasoning on particular points.

The rigid structure forced participants to move away from their “favourite” topics with respect to *Inner Source* and look at other, usually neglected, aspects. In several cases, especially for the management roles, this meant facing the complexity of the topic:

“I realized that, so far, the discussion was on the buzzword level, blind to the complexity of the topic. We completely avoided looking into the organizational aspects.”

Reaching a shared assessment for the respective factors occasionally proved a challenge for the groups. In this application, especially the facets *artefacts* and *practices* diverged significantly in their expression within different parts of the organization. Thus, participants found assigning a

single assessment value for several of their factors seemed too coarse to reflect the organization's context.

8.4.2 Evaluation of facets and categories

During the workshop, participants were asked to discuss the factors of each reuse facet of *Inner Source*, as presented to them by the model. Based on their comparison of the factor expression required by the approach with their assessment of the factor at U, they rated the factors as *conflict*, *discrepancy*, or *agreement* with the current state of the practice at U. Furthermore, they marked the factors as *new* and/or *relevant*, if appropriate.

Table 8.1 summarizes the results of their assessment: in total, the participants discussed 140 aspects of *Inner Source* adoption. Of these aspects, 35 were rated as conflicts, 35 as discrepancy, and 46 as agreement, accounting for 116 assessed factors³. Of these 116 factors, 70 require action within the adoption process (60%). 105 out of the 116 factors were rated as *relevant* (90%) and 15 were rated as *new*. With respect to the different facets, the results varied. We detail on them in the following paragraphs.

Facet	Factors	Conflict	Discrepancy	Agreement	New	Relevant
Artefacts	39	0	9	24	0	26
Practices	45	13	10	11	5	34
Tools	13	7	2	4	4	13
Organization	43	15	14	7	6	32
Total	140	35	35	46	15	105

Table 8.1: Summary of the compliance assessment between U and *Inner Source* based on the tabular instantiation of RASM. The table displays the number of factors (constituents) of each facet, as well as the number of conflicts, discrepancies, and agreements between the company values and *Inner Source*. In addition, the number of new and relevant factors is reported.

Artefacts The facet artefacts showed the most agreements of the four reuse facets. Out of 39 factors, 24 were considered as agreements and 9 as discrepancy. The factors seen as potential risks included the availability of design artefacts, such as architecture and UI templates, that should ensure technical compatibility and consistency of the developed artefacts. Furthermore, the existence of seed projects was marked as discrepancy: a number of small of candidate projects could be identified, but their overall business value, scope of use across departments, and their potential user/contributor base were not entirely clear. Since these are key factors for the adoption of *Inner Source* [9], these discrepancies should be addressed.

As the risks identified on the artefact level were minor and, thus, few urgent actions could be derived, participants considered the facet as useful check-list for reference when considering an artefact for *Inner Source* development.

³Participants could leave factors blank if they did not consider them relevant.

Practices The assessment of the facet practices revealed a high number of challenges to *Inner Source* adoption, expressed in 13 conflicts, 10 discrepancies, and 11 agreements. Particularly, the heavy use of *clone-and-own* reuse from snippets to larger system parts, as well as incompatible development styles and (partial lack of) quality assurance practices led to conflicts. In addition, the integration of the different stakeholder needs and the lack of a contribution management were seen as significant challenge for adoption. The strong heterogeneity of development, maintenance, and quality assurance practices accounted for the discrepancies.

According to the participants, mending the identified risks on an organization-wide scale was out of their direct reach. However, they identified the development of a set of practices for their *Inner Source* implementation as one next step.

Tools On the tool level, participants identified 7 conflicts, 2 discrepancies, and 4 agreements. However, they remarked that several of the conflicts, e.g., the lack of a central and accessible version control system, had already be identified independently and were being worked on. In summary, the participants of this group identified supporting infrastructure elements that they expect can be adopted without significant effort.

Organization For the facet organization, the participants assessed 36 out of 43 factors and identified 15 conflict, 14 discrepancies, and 7 agreements. The conflicts were assessed for the factors management culture, resources, homogeneity of practices, toolsets, reuse vision, and department culture. Furthermore, the alignment between the conventional software development and *Inner Source* development, the creation of appropriate roles and processes, and the quality assurance for contributions were considered important, yet not yet compatible with the current state of the organization.

Discrepancies occurred for the factors related to community building, e.g., the recruiting and management of core contributors, the project management roles, the confidence of developers and managers with the transparency induced by *Inner Source* development, as well as the required support by middle management.

Estimating the adoption effort Table 8.1 displays the total number of conflicts, discrepancies, and agreements between U and the RASM instantiation for *Inner Source*. However, to reach a qualitative estimate of the adoption effort, these should be weighted according to their relevance. Table 8.2 provides this overview by separating conflicts and discrepancies of the single reuse facets according to their criticality (required or recommended for adoption).

The result of this split provides the following information: in terms of required constituents, U's company profile currently is in conflict with 11 (mostly in the facets practices and organization) and has discrepancies with five constituents (with a majority in organization). In terms of recommended constituents, U is in conflict with 23 (with all but artefacts concerned) and has discrepancies with 21 (distributed across all reuse facets).

Two insights can be drawn from this information: first, U is currently missing a number of key requirements that are necessary for adopting *Inner Source*. Most of them are lacking in the organizational facets, followed by practices. This indicates that incompatibilities with the organizational culture might be the most significant threat to the adoption. Second, U's context is currently clearly at odds with a large number of aspects that would support the adoption and

ensure better benefit realization of the approach. However, half of the missing recommended factors have been marked as discrepancies and, thus, could be addressed with moderate effort.

Prioritizing the adoption steps A third of the required constituents for *Inner Source* adoption currently missing at U fall in the category discrepancy (see Table 8.2). This indicates that these constituents could be reached by means of moderate efforts and might provide tangible progress to stakeholders. Mending the discrepancies of required constituents, thus, might be strategically beneficial to advance the adoption.

Facet	Factors	Conflict		Discrepancy	
		required	recommended	required	recommended
Artefacts	39	0	0	1	5
Practices	45	2+3*	7	1	8
Tools	13	1	6	0	2
Organization	43	5	10	3	6
Total	140	11*	23	5	21

Table 8.2: Summary of the compliance assessment between U and *Inner Source* based on the tabular instantiation of RASM. The table displays the number of factors (constituents) of each facet, as well as the number of conflicts, discrepancies, and agreements between the company values and *Inner Source*. In addition, the number of new and relevant factors is reported. (*Three elements were discouraged and in conflict. They are summarized with conflict and required.)

Summary of the facet evaluation The results of the model application provides the following insights: first, the majority of the contained factors were relevant to the participants. In addition, the application supported them to identify some new aspects in all but one facets. Second, by means of the model application, participants identified a significant number (50% of the presented factors fall in one of the two categories) of aspects conflicting with or diverging from the context requirements of *Inner Source*.

In summary, these findings suggest that RASM can contribute in identifying relevant aspects that need to be resolved for a successful adoption of a reuse approach.

8.4.3 Participant feedback workshop

Following up on the workshop, we invited the participants to fill in an anonymous questionnaire, reflecting their experience with the model application. The questionnaire covered two aspects: the perceived usefulness of the model instance for the participants as well as the ease of use of the tabular representation.

Each aspect was covered by one Likert battery, providing four to five answer options and featuring a four-point Likert scale with an additional no-answer option (disagree, moderately disagree, moderately agree, agree, no answer). In addition, the participants were invited to provide additional comments on the model in a free-text form.

We report the questions and their answer options in Figures 8.3 and 8.4. Eight of the ten workshop participants participated in the questionnaire. All responses were complete. The no-answer option was not used.

Perceived usefulness As reflected by Figure 8.3, the participants' attitude with respect to the perceived usefulness of the model was positive: In particular, the participants strongly appreciated the *structuring* of the topic, the identification of *new aspects*, and the identification of aspects *critical to the adoption in the given context*. In addition, the majority of participants reported that relevant discussions were initiated during the process of applying the model and that it helped them identify concrete next steps to advance the adoption.

During the further course of *Inner Source* adoption, the majority of the participants are planning to use the model instance as reference.

The positive impressions are backed by the free text feedback that considered the model as providing a *“good overview on topic”*, *“structured view on topic”*, and appreciated the support for discussion *“model structured complexity of topic and tamed it for discussion”*, *“topics initiated important debate and highlighted relevant aspects”*. Summing up, participants considered themselves more informed after the workshop: *“This definitely helped filling in the blanks.”*

Perceived accessibility Figure 8.4 relates the answers with respect to the accessibility of the model. In general, participants felt the need for qualified support when completing the assessment of the reuse facets. The free text answers reinforce this impression:

“Without help, completing the model would be too difficult.”

In particular, two participants indicated that additional care is required to prevent misunderstandings when translating information from the academic realm to a particular case in practice: *“A glossary would have helped as, for several terms, we have a specific notion within our organisation that did not coincide with the model terminology.”*, *“The model was too academic and should have been more specific to U.”*

Summary of the participant feedback Overall, participants appreciated the structure and the degree of detail of the model instantiation as it grounded the fierce debate, supported them to manage the complexity of the topic, and covered neglected aspects. To improve the model in the future, two participants wished for an up-front tailoring of the model.

8.4.4 Summary of the model application

Benefits Based on the participant feedback and observations from the workshop, we consider the goals of the model application to be met: participants gained new and relevant insights for their adoption process, and derived facilitating steps to improve the current state of practice in their organization. The structure of the model supported them in the face of the complexity of the topic and might be a useful reference for future decisions.

Challenges From the observations, we realized that a lack homogeneity with respect to a facet poses a challenge for filling in the model in its current form. In the case of U, most frequently, these items were marked as discrepancy, as they were not matching the requirements of *Inner*

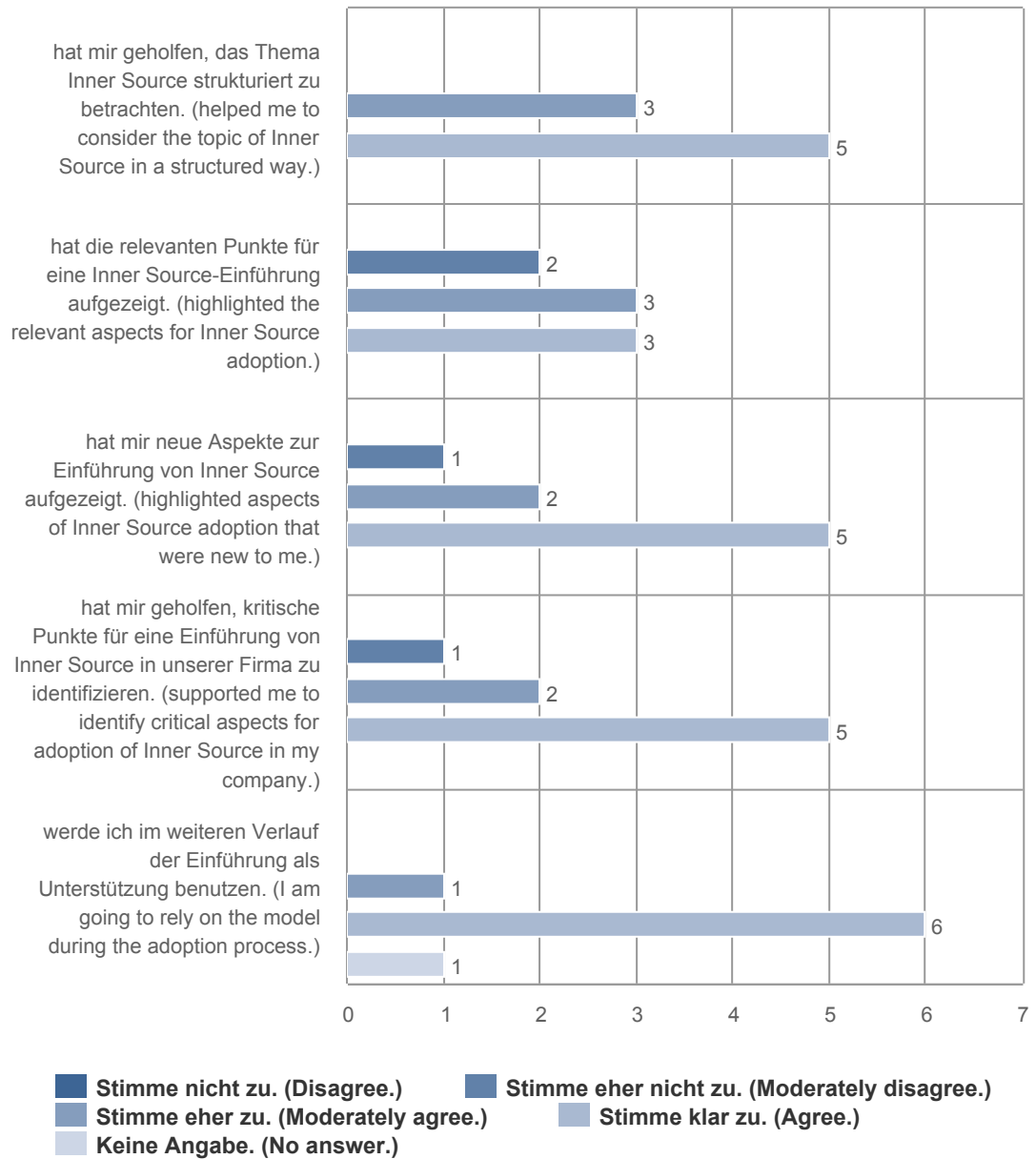


Figure 8.3: Responses for the question *model assessment*. Question text: “The model for Inner Source...”

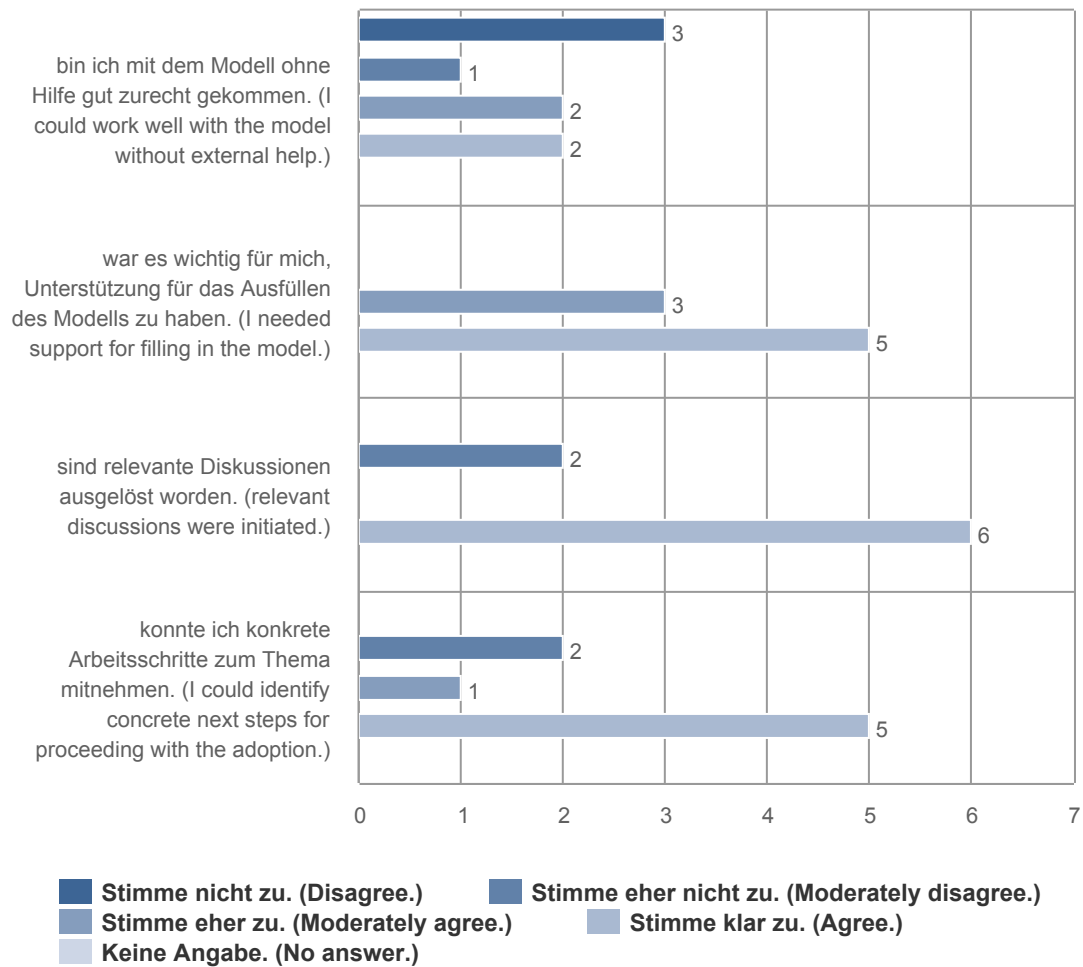


Figure 8.4: Responses for the question *model accessibility*. Question text: “When applying the model during the workshop...”

Source with respect to all software development departments. For future application, this aspect needs further investigation.

The answers to the questionnaire reinforced the need for support when working with the model. In particular, finding a common language with each organization should be facilitated, e.g., by providing a detailed set of definitions with the model.

8.5 Limitations of evaluation

The presented evaluation serves as a proof of concept to inspect how a RASM can support practitioners to systematize their discussion, evaluation, and planning of an adoption of a software reuse approach. As the approach was already decided on by higher management, this evaluation did not effect a full effort-benefit comparison but focused on the identification of potential issues and critical points for adoption.

A case-based evaluation cannot give conclusive evidence in terms of quantitative results in terms of saved effort or expenses.

Nevertheless, it provides indication that the purpose of the model, grounding a complex discussion by means of structure, as well as making research results available to practitioners, addresses a challenge faced in practice. The practitioner accounts, furthermore, suggest that an interactive application of the model has the potential to rationalize discussions on reuse adoption, as well as highlighting aspects that have so far been neglected.

8.6 Next steps

As we have seen in this proof of concept, a single instance application of the model can provide a first realistic/objective global assessment of the effort required to adopt a particular reuse approach. However, for a successful adoption, this is only the first step: organizations need to derive concrete actions, plan for the required changes, assign responsibilities, and follow up on the adoption process.

Operationalization Whilst many of the indicated changes fall within the organizational domain (and, therefore, within a different range of instruments for management), also the technical and code-centric aspects of reuse adoption require attention: for instance, a change of reuse strategy might entail discouraging *clone-and-own* reuse practices and, instead, promoting the use of specific libraries. To manually assess the compliance with this new strategy, as well as its effect on, e.g., code quality, is infeasible. Therefore, tool support is required that supports practitioners with measurement of reuse aspects in the source code. This topic is addressed in more detail in Part IV.

Further instantiations To support the approach selection use case of RASM, a knowledge base of instantiations for further reuse approaches needs to be created and maintained. Ideally, also the interactions and dependencies between different expressions of (groups) reuse factor constituents would be integrated in the model to provide more detailed guidance for adoption.

Further applications This proof-of-concept evaluation has focused on the identification of discrepancies and conflicts that need do be addressed during *Inner Source* adoption at U. In future applications of RASM, we aim to apply the full model and, thus, effect the complete comparison of potential adoption benefits with estimated adoption efforts.

Part IV

Methods and tools to detect reuse potential

9 | Detecting reuse potential in the context of a RASM application

The reuse adoption support model presented in Chapter 7 requires a significant number of details to be captured on the development context and capabilities of a company. One of these is the identification of suitable entities for reuse that should either be created or prepared for reuse. Whilst many of these aspects are difficult to capture in a quantitative way, several questions relevant to software reuse and potential improvements can be answered by analysing the software products of the respective company.¹ For instance, a prevalence of cloning might indicate a lack of suitability of given artefacts or a lack of support in the current infrastructure to find or access entities for reuse. If a counter strategy, e.g., introducing a new infrastructure, targets these issues to support more advanced reuse practices, these aspects could be tracked by automated measurements and, thus, related to the degree of success achieved by a new reuse strategy. The following chapter presents methods, approaches, and tools developed to quantify different aspects of reuse and deliver a more reliable base for the assessment of reuse potential or risks. Parts of this Chapter were published in [37, 39, 40].

Contents

9.1	Detecting reuse potential in source code	142
9.2	Discovering unintentional re-implementations	142
9.3	A hybrid approach to discover unintentional re-implementations .	147
9.4	Combining clone detection and LSI to detect re-implementations	153
9.5	Cross-project clone detection as guidance for reuse improvement	166
9.6	Conclusion	170

¹The applicability, however, depends on the technical infrastructure at disposal.

9.1 Detecting reuse potential in source code

Potential for software reuse can be assessed on many different abstraction levels, ranging from system requirements to source code. In the context of companies that seek to improve their current reuse practices from a pragmatic level, starting to unearth candidates for reusables or instances of missed reuse opportunities can serve as an important starting point for a new strategy. Arguably, both indicators manifest themselves in the form of code redundancies.

On this note, the techniques proposed by the software clone research community offer significant potential. However, they are usually limited to the scope of single projects and find application mostly within the assessment of code quality. In addition, conventional clone detection tends to fall short when attempting to capture missed reuse opportunities in the form of semantic re-implementations. Arguably, these are frequent in practice and, therefore, a method to discover them could significantly improve the insights about the potential for reuse in a code base. During the assessment of adequate reuse approaches with RASM, these insights are of significant value: they provide an objective base for identifying useful candidate reusables, as well as supporting monitoring the changes induced by a change in reuse strategy.

Clearly, automated detection mechanisms for reuse potential have their limitations: they can rely on structural or semantic similarity, but either of these measures can result in incomplete results, due to potentially significant variations in structure or vocabulary of the analysed system. Nevertheless, applying automated detection mechanisms to detect reuse potential can provide interesting findings that manual analysis could not yield feasibly.

Sections 9.2, 9.4, and 9.5 address the limitations of current work in the following way: Section 9.2 presents a novel, heuristic, approach to detect semantic re-implementations and Section 9.4 in a case study on an industrial system provides indicators for the applicability of the approach in practice. Section 9.5 lines out how conventional clone detection, ported to a cross-project scope, could improve the detection of candidate system parts for reuse.

9.2 Discovering unintentional re-implementations

Unintentional re-implementation of existing functionality is an issue frequently reported in practice. It causes increased efforts for development and maintenance and indicates the lack of a suitable reuse strategy. However, instances are hard to find with existing approaches. For practitioners, this increases maintenance risks, such as inconsistent bug fixing, and hinders quality improvement efforts. For researchers, this hinders a reliable quantification of the issue. Insights could inform decisions on candidate reusables and tool support for reuse.

We propose a pragmatic approach combining identifier-based concept location with static analysis to detect candidate re-implementations between two sets of source code. We present initial results from applying the approach to detect re-implementations of utility functionality present in libraries within a sample of Java projects. Parts of this work have been published in [39, 40].

9.2.1 Background

An abundance of valuable software assets is present in companies' code repositories, via Open Source libraries, and commercial component markets. Nevertheless, developers tend to re-implement existing functionality [137, 27], missing out on the benefits of reuse opportunities. Furthermore, this can result in the creation of "Simions" [138], independent re-implementations of existing functionality that do not share a common origin in terms of code. Simions have long term negative effects in the form of increased development and maintenance efforts.

Re-implementations can happen easily for various reasons: (1) the scale of development in large projects makes staying up to date with reusable entities challenging. This entails a certain amount of parallel implementation efforts. Despite the higher probability of a required functionality being available, the increasing effort for searching, understanding and adapting a reusable incites implementing functionality from scratch [27, 137]. (2) The use of established protocols might impose specific implementation steps that are duplicated [139]. (3) To achieve business goals, duplicate implementations might be necessary at times. (4) Evolution of libraries might make parts of the code obsolete [137]. (5) Low API usability prevents users from finding the implementations realizing a specific concept [140].

Discovering re-implementations is difficult in theory and practice: first, semantic equivalence checking is well studied (e.g. [141]) and a generally undecidable problem. Approaches to detect re-implementations therefore are constrained to resort to approximations. Second, previous work [33, 138] concludes that existing approaches, such as clone detection or random testing approaches [142], do not provide satisfactory results to detect re-implementations in practice.

Consequently, research so far is unable to realistically quantify the size of the problem. Practitioners, on the other hand, miss an approach providing support to avoid new and discover existing re-implementations [27]. Therefore, we need a new approach to discover missed reuse opportunities in the form of (unintentional) re-implementations.

Summing up, we can state the following: Re-implementations of existing functionality happen in practice and entail negative effects, such as increased costs for development and maintenance. However, we are lacking a comprehensive approach to discover them. As a result, the extent of the phenomenon remains unclear. Furthermore, practitioners lack support to address the issue.

In the following, we present a novel approach to discover re-implementations between software libraries and a system's source code. To this end, we establish a broader definition of similarity, based on the concepts embodied in the identifiers. We implement our approach for Java systems and provide a calibrated set of parameters for it. We report on a proof of concept evaluation, detecting re-implementation of library functionality in three Java systems.

9.2.2 Related work

Prior work has addressed cases of semantic code duplication. Our notion of re-implementations is related as follows:

Semantic clones [143] are code fragments with isomorphic program dependence graphs, and therefore structurally similar. Their behaviour can, but does not need to, be functionally

similar. **Accidental clones** [139] are code fragments of different origin that are syntactically similar due to the adherence to a specific protocol. This does, however, not imply behavioural similarity. **Type-4 clones** [125], **“wide miss” clones** [127], and **Simions** [138] refer to the same phenomenon: behaviourally similar code fragments that have no common origin. Unlike cloned code, these fragments are likely to differ greatly in their structure [138].

In the following, we present approaches that aim to detect or avoid unintentional re-implementations.

9.2.2.1 Detecting similar implementations

Closest to our approach is the work by Marcus and Maletic [127]: they aim to interactively detect *high-level concept clones* by computing the similarity of source code documents (that can be of the granularity of files or methods) and clustering of the results. The similarity is computed with Latent Semantic Indexing, LSI [144]. The clustering can be enhanced by using structural information. Determining relevant high-level concepts is done by the user. In a case study, the authors uncover simions of a list within one system. Our approach differs in scope and techniques: We aim to find simions between a corpus of libraries and one or more systems. Since the libraries determine the relevant concepts for the analysis, we need a pragmatic way to extract their key concepts from their source code. For this, we choose TF-IDF which is robust across systems and does not require extensive tuning. Furthermore, we take use the program structure to restrict the vocabulary used by the analysis.

Al-Ekram et al. [139] report empirical findings on *accidental cloning* across software systems. Their approach detects structurally similar code fragments caused by usage patterns required by specific technologies. However, the authors state that their approach is likely to miss re-implementations that are fundamentally different in structure.

Jiang and Su [142] propose random testing to automatically mine functionally equivalent code fragments. The source code is randomly cut in chunks. Two chunks are considered equivalent if they produce the same output for the same random input data. The study reports promising results for the test systems, namely a Linux Kernel and a sorting benchmark. These systems are written in C and, due to their functionality, do not require functionality, such as string processing, which is prevalent in average systems. Deissenboeck et al. [33] found that reproducing Jiang and Su’s experiment on Java code yielded unsatisfactory results. Apart from challenges induced by the different requirements of the technical platform, they consider their definition of equivalence problematic: first, it does not account for side effects. This causes code fragments to be pronounced equivalent that a programmer would deem fundamentally different. Second, independently of the given input, most code chunks did not produce any output or yielded exceptions. By their definition, these chunks are equivalent. In practice, this information is of little value.

Kawrykow and Robillard [137] propose an approach to mine Java systems for methods “imitating” library methods available to these systems. Their goal is to replace functionality implemented in the client code by calls provided by the library. They abstract method bodies to program elements and perform a matching between the available library methods and the client

methods. Whilst they cater to the important use case of replacing obsolete client methods by library methods, the notion of equivalence on the method level is still too restrictive for our task: the re-implementations we are looking for might be present in different code structures and would therefore be missed by the approach.

9.2.2.2 Detecting similar applications

Using API calls to find relevant code has been proposed and applied before, e.g. in the context of code search [145, 146, 147] and rapid prototyping [148]. However, we do not know of this technique being used to track and quantify simions in existing software systems. Despite the differing context of the work, the successful use of API calls as indicators for relevant code encourages us to exploit this idea for our goal.

Teyton et al. [149] support the process of library migration by mining function mappings from projects that have already completed the transition between two given libraries. This approach pragmatically overcomes the challenges of establishing a notion of “similarity” in terms of the program constructs themselves and is, therefore, immune against differences in structure and vocabulary. However, in our context, historical data is not applicable.

9.2.2.3 Preventing re-implementations

Thung et al. [150] address the problem of duplicate implementation in a constructive way: by analyzing repositories to determine which APIs are used together, they provide recommendations of potentially useful APIs for a given project. Their goal is to inform developers of existing APIs before they re-implement the respective functionality. Our work complements this approach by discovering already existing re-implementations that could be replaced by libraries.

Code recommenders, proposed by [151, 91, 152, 153, 154], address re-implementations by recommending code snippets or applicable library methods depending on the current development context. Similar to code recommenders, enhanced code completion [155, 156] aims to ease discovery of existing functionality to the developer. These approaches do not support detection of already existing re-implementations. However, the techniques employed to generate recommendations include code structure and identifier analyses.

9.2.2.4 Concept location

The use of identifiers is a common strategy for concept location [157]. Methods from text retrieval, TR, (such as Term Frequency-Inverse Document Frequency, TF-IDF [158], or Latent Semantic Indexing, LSI [144]) are used to extract concepts given by e.g. use cases from source code. Recently, these approaches have been enhanced by adding static and/or dynamic program information. A study by Basset and Kraft [159] further suggests that structural term weighting can improve TR based concept location. To the best of our knowledge, the mentioned techniques have not been applied to our case.

9.2.2.5 Clone detection

Code clones are a popular reuse mechanism in Open Source and industrial software development practice [52, 160, 60]. The mechanism provides short term benefits for development but is linked to negative consequences for maintenance (e.g., higher testing efforts, bug propagation, faults due to inconsistent changes, security violations [63, 161]). Consequentially, significant research effort has been invested into approaches and tools for clone detection [162, 63, 163, 164, 165, 166]. State of the art clone detectors produce good results in discovering identical code fragments (*Type-1*), syntactically identical fragments (*Type-2*), as well as syntactically similar fragments that have been moderately altered by adding, removing, or changing the order of statements (*Type-3*) [167, 162, 163]. However, discovering semantic re-implementations, also known as *Type-4 clones*, remains a challenge [168, 167]. Research has identified the need to improve the results of clone detection for better applicability [169, 170, 171]. In [170], the authors propose the concept of *structural clones*, that incorporate several *simple clones* for better insights. Our ACD is following this proposition; our study, however, has a different scope. In [171] the authors apply clustering with LSI to clone detection results to support maintenance with relations between clusters. Again, the scope of our study differs.

9.3 A hybrid approach to discover unintentional re-implementations

The following section presents our approach. At this stage of our work, we focus on discovering re-implementations of well established concepts available in open source libraries.

Our simion detection proceeds as follows (see Figure 9.1): it takes as input the so-called “concept library”, a curated collection of libraries from which the concepts are mined, as well as the “study object”, consisting of one ore more software systems in which we look for simions. The identifiers of concept library and study object are extracted and analyzed in a preprocessing phase to learn the specific concepts present in their source code. The preprocessed identifier information is then used in the matching phase to compute the likelihood of two code entities implementing equivalent functionality.

By resorting to identifiers, we overcome the problem of restricting similarity to a syntactic level. As studies have shown, identifiers are valuable sources for capturing programmers’ intent [172, 173]. Therefore, we assume that two code fragments that contain identifiers belonging to the same concept might provide the same functionality and could, therefore, be potential re-implementations. This assumption is strengthened by Haiduc and Marcus [174].

Whilst the intuition behind this approach is quite simple, relying solely on identifiers risks to clutter the results with false positives: the same identifiers occur when defining a specific functionality as well as when using it. To mitigate this, we only consider identifiers present in *declarations* of methods, fields, and classes.

Our approach abstracts functionality provided by identifiers on a per-file basis. We opt for this granularity to capture concepts spread over several methods. During the preprocessing phase, we assign a set of “significant” identifiers to each source code file. We deem those identifiers as significant that best² capture the concepts of the respective file. Based on this information, we compute a similarity score for all files within the study object and the files of the concept library³. The similarity score is computed as follows: $\frac{\sum_{i \in I_{b \cap s}} v(i)}{\sum_{i \in I_b} v(i)}$, with I_b denoting the relevant identifiers of a concept file and $I_{b \cap s}$ denoting the overlapping relevant identifiers of a concept and a study object file. $v(i)$ denotes the weight assigned to the given identifier i . We implemented our prototype on top of ConQAT⁴, an Open Source software quality analysis tool.

9.3.1 Calibration of the approach

The quality of the obtained results depends significantly on the processing (and the quality) of the identifiers. In this section, we describe the variation points and the steps of calibrating the parameters of our approach.

²“Best” is determined by the characteristic identifiers contained in the corpus of libraries.

³Depending on the context, not all concepts need to be searched for. Instead, the analysis can be run for specific concepts present in the library, such as “Collections”, “I/O” etc.

⁴www.conqat.org

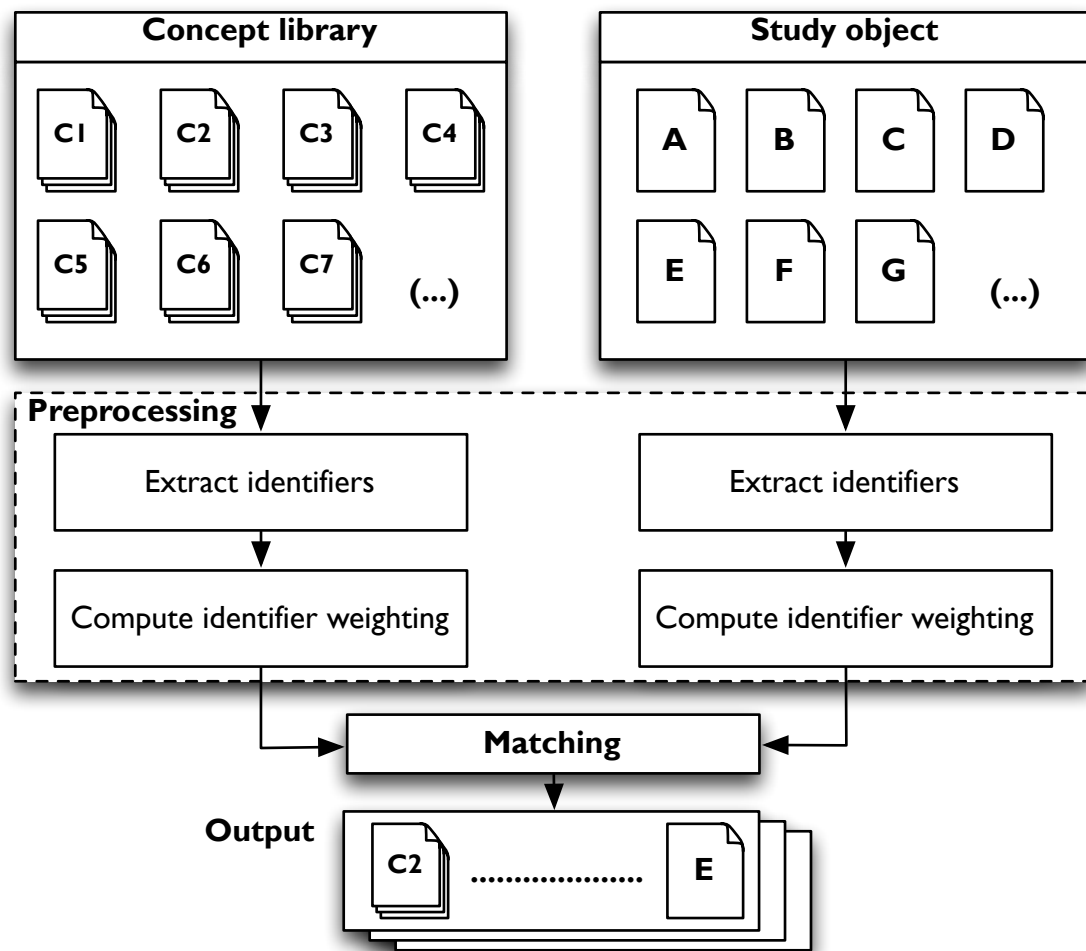


Figure 9.1: This figure illustrates our approach: we extract relevant identifiers for each concept and compute the best matches within the study object. In the preprocessing step, different approaches can be taken to select and prepare the identifiers that are subsequently used.

There are two steps in the process that allow for variation: (1) identifier extraction and (2) identifier ranking. For both steps, we present the potential options. Then, we describe the setup of the calibration process, the tested parameter configurations, and the resulting set of parameters used for our proof-of-concept evaluation.

Identifier extraction The first step of the preprocessing phase is the extraction of the identifiers. We assume that preselection of identifiers guided by the program structure would improve the precision of our findings. To quantify the effect of this step we included this decision in our calibration process. Second, we tested the impact of splitting⁵ and partitioning the identifiers on our results.

⁵We used CamelCase as well as non alpha-numeric characters as indicators for splitting. Furthermore, we applied an English word stemmer and removed trailing digits.

Identifier ranking The second step of the preprocessing phase assigns a weight to the identifiers extracted for each file. Pragmatically, one could count the absolute frequencies to identify the most relevant concepts in a file. However, this approach risks to overshadow important concepts. For this reason, we compare the effect of ranking concepts according to their absolute identifier frequency to using the TF-IDF metric. Furthermore, we test the impact of several threshold values for TF-IDF.

9.3.1.1 Calibration setup

To calibrate our approach, we considered the specific use case of discovering potential re-implementations of well known “Collections” concepts in the Qualitas Corpus. We buildt up our concept library by curating these concepts from well known Open Source libraries, such as Apache Commons, Trove⁶ and Guava⁷. By manually assessing the library implementations, we found that indeed the vocabulary used to represent the concepts in the identifiers was very similar.

Study Object To test the suitability of a configuration, we need a way to measure the quality of the result we obtain. Since we can not manually establish the number of re-implementations of a given concept within the 112 systems present in the Qualitas Corpus⁸, we injected deliberate re-implementations into the corpus by inserting the files of the Guava Collections into the Qualitas Corpus⁹. In this way, we obtain a known set of expected hits for our approach. Nevertheless, determining the quality of the result remains challenging. Manually validating the presence of all expected Guava files in the results is infeasible. Furthermore, results yielding the same number but different files can not be differentiated in quality. To address these challenges, we set up the experiment as shown in Figure 9.2: for each configuration, we run our analysis once. Then, we randomly sample 10 files from the Guava Collections and probe the result set to find out 1) whether they are included and 2) in which position of the result set they occur. This probing step is repeated 100 times, each time with a different random sample.

Configurations We account for the mentioned variation points in the following way: The selection of the identifiers is done either by extracting all identifiers present in the current file or extracting only identifiers present in declarations of classes, methods, and variables.

Partitioning of the identifiers refers to employing splitting techniques and providing, in addition, substring representations of the identifiers. Take as example the identifier `arrayStackItem`. The ordered substring representations would yield: `{array, stack, item, arrayStack, stackItem, arrayStackItem}`. This variation is either on or off. The configuration for TF-IDF varies from counting the identifier frequency to using TF-IDF with the threshold values of 5, 10, and 15.

To compare the results for each configuration, we computed the following metrics per probing step: the *average hit count*, establishing how many of the files in our probing set are present in

⁶<http://trove.starlight-systems.com/>

⁷<https://code.google.com/p/guava-libraries/>

⁸We used the Qualitas Corpus version 20130901r and the JRE 1.6.0.

⁹For this setup, we removed the Guava Collections from the concept library.

the results, and the *position*, denoting the rank of the files in the result set. The final performance of a configuration is rated by averaging the average hit count and the position values for all the probing steps.

9.3.1.2 Calibrated configuration

The calibration procedure yielded the following configuration as the most suitable: *preselecting* the identifiers according to the program structure, *partitioning* the identifiers, and using *TF-IDF* with a *threshold value of 5*. Consequently, we run our proof of concept evaluation with these settings.¹⁰

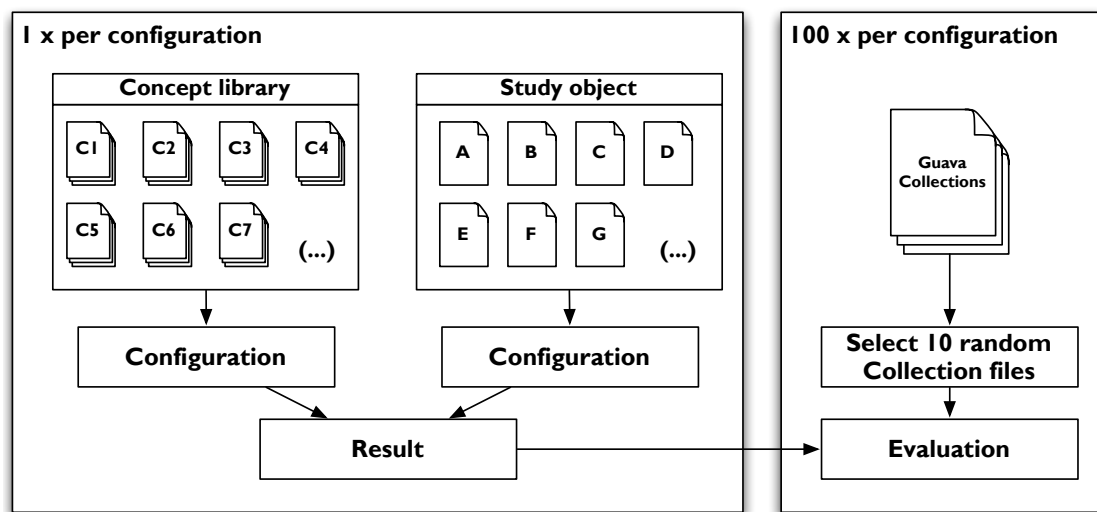


Figure 9.2: This figure visualizes our evaluation process for detecting re-implementations of Collection functionality. To establish a base-line of known duplicates, we injected the Collection implementation of Guava into the Qualitas Corpus and measured the detection rates for these known “re-implementations”.

9.3.1.3 Taxonomy of the results

From the calibration results, we built up a taxonomy of findings by manually assessing the first 250 results. The code fragments matched the following categories:

A code fragment is a *potential re-implementation* if it implements or extends functionality contained by or equal to the concept we searched for. Clearly, our approach can not prove the fragment’s functional equivalence. However, it can point developers to candidate points in order to establish whether they really present missed reuse opportunities. Potential re-implementations can have a varying degree of similarity to the concept implementation in the library. We therefore differentiate between *perfect match*, where study object and concept library implement the same

¹⁰Comparing our calibrated approach to [127] would be interesting. However, it is unclear if their system is available.

functionality, *similar match*, where study object and concept library implement similar aspects of the same concept, and *concept match*, where study object and concept library implement different aspects of the same concept.

If a code fragment merely wraps library calls for a specific concept, our approach will still include it in the result. However, we consider this case as a *concept application* and do not classify it as missed reuse opportunity.

False positives unrelated to the considered concept are seen as *bad matches*. For situations where we can not fit a result in any of these categories, we label them as *undefined*.

Result classification The distribution of the calibration findings is as follows: 195/250 potential re-implementations (out of which 11/195 perfect matches, 86/195 similar matches and 98/195 concept matches), 30/250 concept applications, 18/250 bad matches, and 7/250 undefined. The ranking of the results intuitively presented the potential re-implementations with higher values than the other categories.

We, furthermore, assessed the 18 bad matches found within the 250 results. The majority (16/18) of bad matches occurred due to similarities in the vocabulary employed by different concepts, in our case string manipulation and iterations over collections. The remaining bad matches were applications of the concept contained e.g. in implementations wrapping the usage of a library.

9.3.2 Proof of concept evaluation

Our proof of concept evaluation provides a first answer to the following question: Which re-implementations do we find within our study objects?

To answer this question, we select three Open Source Java projects, analyze them and manually examine the source code indicated as re-implementation by our approach. We restrict the search for re-implementations again to the “Collections” concept. Furthermore, we limit the assessment to the first 30 results for each system.

Setup The concept library used for this evaluation contains the Apache, Trove and Guava collections. Our study objects are the Apache projects “MyFaces” and “Tomcat”, present in the Qualitas Corpus, and the “Spring IO” framework[175]. The systems provide functionality related to web applications with Java. Due to this specialization, we expect them to use given collection implementations. Therefore, re-implementations of this concept would be missed reuse opportunities.

Results The inspection of the analysis results yielded the following re-implementations: a perfect match of *IteratorEnumeration* and a concept match for *MapEntries* in MyFaces, a *Null-Comparator* perfect match and a similar match for a *UnmodifiableMap* in the Spring framework, and a perfect match for the *ArrayStack* implementation, a perfect match for the *Entry* implementation and a similar match for a *HashMap* implementation in Tomcat. For all three systems, the perfect matches ranked within the first five positions of the findings.

9.3.3 Threats to validity

The preliminary character of our investigation entails a number of threats to validity. Firstly, the concept library and the study objects are currently very specific. It remains to be seen how well the approach performs on a larger and less carefully curated collection of libraries and systems. Secondly, we calibrated our approach for a well known concept, encompassing a clear vocabulary. This characteristic might not be necessarily given for other concepts. It remains to be seen if our approach can provide helpful results nevertheless. Possibly, it could be enhanced by sourcing domain knowledge from the implementors or including ontologies.

Determining the equivalence of implementations by automatic ranking as well as manual inspection remains challenging. Therefore, neither the weighting function nor the manual assessment for determining the quality of the results can be perfectly reliable.

9.3.4 Summary

We presented a pragmatic approach to detect re-implementations. Due to a wider notion of similarity, which is based on the concepts contained in the source code, it is able to find potential duplicates that likely would be missed by established approaches such as clone detection. Our preliminary results look promising and encourage us to follow up with extended evaluations. In this way, we aim to quantify the extent of re-implementations in software systems as well as to support practitioners to avoid or remove missed reuse opportunities.

9.4 Combining clone detection and Latent Semantic Indexing to detect re-implementations

The previous section has presented a novel approach and a proof of concept to detect re-implementations in practice. The results were promising but indicated a typical applicability issue of static analyses: the sheer number of results is overwhelming and can not be addressed in a feasible way. Furthermore, results tend to be cluttered at first and require extensive triaging before being perceived as actionable by practitioners. These findings motivate the following work: we analyse whether intersecting the results of a conventional clone detection with the results obtained by our IR-based approach can mitigate these issues and produce a focussed and high-quality result set that is considered actionable by practitioners.

This section presents a refined version of the approach presented in Section 9.2 and the results of a case study conducted with an industry partner. We show that the combination of the approaches indeed lead to valuable results for practitioners.

The intuition guiding our approach is the following: first, code clones and re-implementations are instances of redundancies. Second, Type-4 clones could potentially have evolved from lower clone types, diverging from the original to an extent that only minimal syntactical similarities remain, yet retaining semantic similarities. Third, approaches for detection provide an unmanageable number of results. Cross validation of the findings of clone detection with LSI and vice versa could yield a more focussed result set for each approach, as the findings of the syntactical level are confirmed or rejected by the semantic level. As a result, we expect a significant increase in the precision of the findings. Last, each technique is known to miss certain types of relevant redundancies, e.g., renamed copies [127], structurally diverging fragments [138]. A combination of both could compensate for this shortcoming [127].

Our findings suggest that (1) latent semantic indexing and clone detection complement each other, (2) aggregated clone detection can be a better indicator for re-implementations than LSI, and (3) the combination of the techniques provides high quality result sets which were considered relevant and actionable by practitioners. Parts of this work were published in [40].

9.4.1 Study goal and research questions

The goal of our study is to provide an answer to the following question: **Can LSI and clone detection complement each other to better detect re-implementations?**

For our study, we derive the following research questions:

RQ1. Do LSI and CD produce different results?: We assess whether the two analyses produce different result sets for the same system, compare the overlaps and additional hits each of them provides, and study the characteristics of the findings. Additionally, we want to determine whether there are re-implementations that can be found only by either of the approaches and would, thus, warrant a combined application to reach a more detailed understanding when assessing a system.

RQ2. Does intersecting the result sets of both analyses improve the quality of the results?: LSI as well as clone detection (static analysis in general) in practice suffer from the fact that they produce an enormous number of findings. Assessing all of them is infeasible for practitioners. For this research question, we explore whether intersecting the result sets of our analyses produces a more focused and high-quality result set.

RQ3. Are the results of the combined analyses relevant for practitioners?: Since re-implementations are difficult to track, evidence is scarce on their relevance. With this research question, we would like to explore whether our findings are of practical use, i.e., actionable for application, for practitioners.

9.4.2 Study object and subjects

Study Object Our industrial study object is a Java enterprise application managing the data exchange between several back office systems, management tools and offline capable end user applications on different devices and platforms. It is written in Java and has been under development for the last decade, growing to >150.000 LOC distributed over >2900 classes and 1194 files. The average file size is 130 LOC, the median file size 60 LOC, the minimum file size 5 LOC, the maximum 2030 LOC.

Based on expert opinions, we suspected the presence of re-implementations. However, we did not have any kind of reference links or baseline at our disposal. For the analysis, we excluded generated (thus unmaintained) system parts.

Study Subjects For RQ3, we asked two system experts to rate the re-implementations according to their relevance for action. One of the experts is the system architect responsible for the entire study object. He has 18 years of work experience, 14 of which were spent at the current company. The other expert is a Dev-Ops engineer working on parts of the system. His work experience at the company amounts to 3 years, his overall work experience to 16. Their participation was motivated by the interest of gaining new insights about the study object.

9.4.3 Implementation of an LSI-based approach

To conduct our study, we implemented an LSI-based approach to detect potential semantic re-implementations by means of combining IR techniques and structural code information.¹¹ The implementation is inspired by previous work [39], as well as by the proposal by [127].

Intuition Our implementation proceeds as follows (see Figure 9.3): it takes as input a body of source code in which we look for re-implementations. We extract and preprocess the identifiers to learn the specific concepts present in the source code. The identifier information is extracted and then used to compute a similarity score in the pairwise comparison of files. This is based on the assumption that the higher the score, the higher the likelihood of the files implementing

¹¹ In [176], we conduct a comparison of two IR techniques, LSI and TF-IDF, for our case. Results suggest that for typical software systems LSI is the better choice, therefore, for our present study, we use LSI, instead of TF-IDF, as IR technique. Contrarily, in the study presented in Section 9.2, the size of the data set indicated the use of TF-IDF.

equivalent functionality. As a result, we obtain pairs of files with an associated similarity score. For further details, see [39, 176].

Identifiers By resorting to identifiers, we overcome the problem of restricting similarity to a syntactic level. As studies have shown, identifiers are valuable sources for capturing programmers' intent [172, 173, 174]. Therefore, we assume that two code fragments that contain identifiers belonging to the same concept might provide the same functionality and could, therefore, be potential re-implementations. During preprocessing, identifiers are split, stemmed, and partitioned.¹²

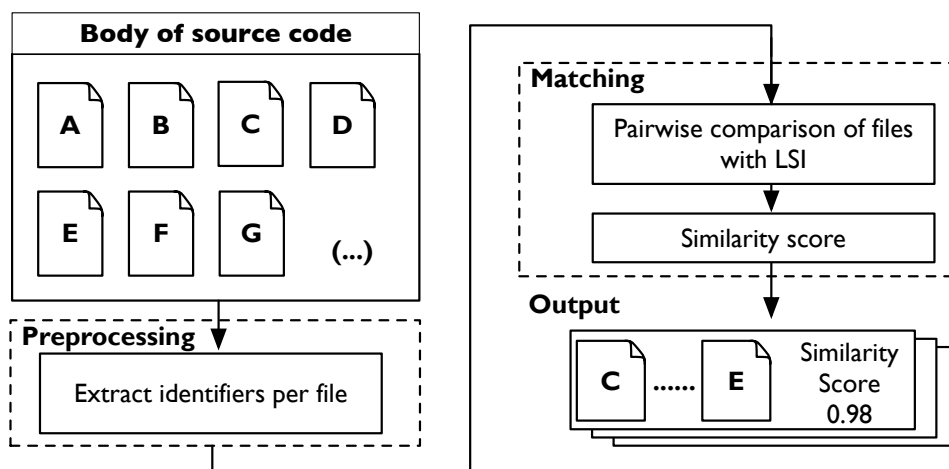


Figure 9.3: Overview of the LSI-based approach. During the analysis, we extract relevant concepts by means of identifiers present in each file and compute the best matches between them.

Our implementation abstracts functionality provided by identifiers on a file basis. We opt for this granularity to capture concepts spread over several methods¹³.

Similarity Based on previous work [176], we use LSI to compute the similarity score of the given pairs of files.

Code structure Whilst the intuition behind this approach is quite simple, relying solely on identifiers risks to clutter the results with false positives: the same identifiers occur when defining a specific functionality and during its use. To mitigate this, we only consider identifiers present in *declarations* of methods, fields, and classes, as well as contained parameters. Previous work [39] suggests that this improves precision in locating the implementations of a concept.

Further exclusions Due to the nature of object oriented programming languages, classes implementing an interface or classes inheriting from a parent share a significant part of their vocabulary. Since these are not re-implementations, we filter these instances from the results.

¹²For details, see [39].

¹³In an Object Oriented context, the assumption of concepts being captured by classes, often represented in files, seems reasonable.

LSI parameters	CD parameters
Global weight: ENTROPY	Minimum length
Local weight: LOGTERMFREQUENCY	in statements: 10
Similarity measure: COSINE	Length of gaps: unbounded
Factors: $0.2 * N$ (N=number of classes)	Number of gaps: unbounded

Table 9.1: Parameter setting for both techniques

9.4.4 Study setup

Initially, we run our LSI-based approach and the ConQAT clone detector [177], developed by our group and proven in numerous industry projects, on our study object. The clone detection identifies clones from identical (Type-1) to near-miss (Type-3), with a minimum length of 10 statements for each continuous cloned region. For LSI, we select our configuration according to the benchmark provided by [178]. Table 9.1 summarizes the parameter settings of our study.

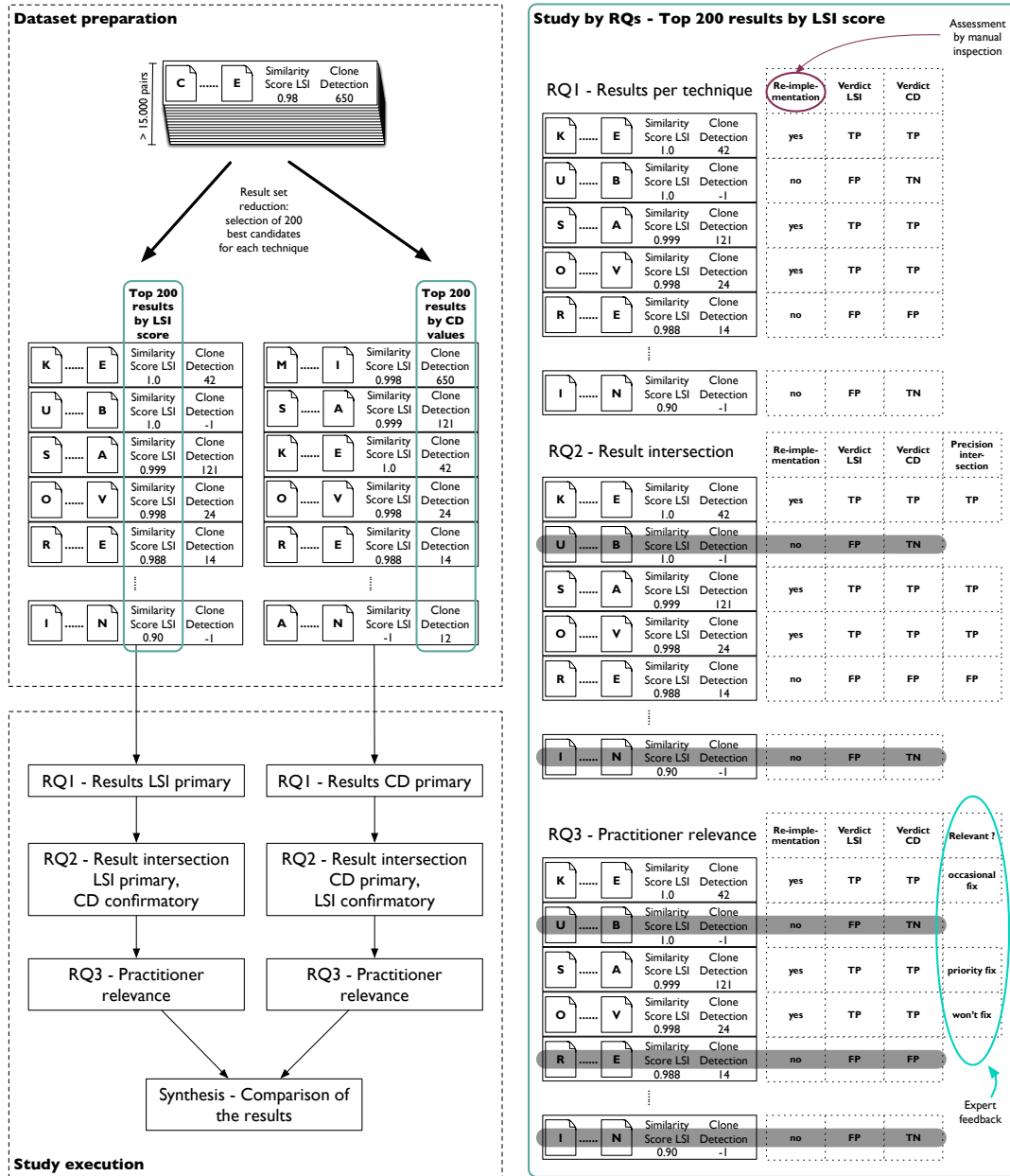
We combine the results of both analyses in result tuples of the following form (see Figure 9.4(a)): $\langle \textit{File A}, \textit{File B}, \textit{Similarity score LSI}, \textit{Number of cloned statements} \rangle$. Files A and B denote the files that have been identified as containing potential re-implementations by either or both of the techniques. The similarity score of LSI represents the similarity value as identified by the technique and ranges from 0.0 to 1.0. The clone detection values capture the number of cloned statements shared between the classes for clones that were within the threshold as mentioned above. In the cases in which one of the two techniques did not produce a result for a given pair, its value is set to -1.

Since the total number of re-implementations in the system is unknown, we had no baseline values at our disposal. Consequentially, we had to analyze the results manually. However, each of our setups produces a significant number of results (>15000 hits). For feasibility reasons, we thus had to limit the number of result pairs. We proceeded as illustrated in Figure 9.4(a): for our approach as well as for clone detection, we sorted the set of result pairs in descending order and extracted the top 200 result pairs for manual inspection.

The two result sets obtained in this way are referenced according to their *primary* technique, i.e. the technique used for the ranking. The second technique is considered as *confirmatory*, as its value for any given result pair is also known. In this way, we set up two branches of comparison: our LSI-based approach vs. the ConQAT clone detection (result set LSI-CD), and the ConQAT clone detection vs. our LSI-based approach (result set CD-LSI). In this way, we can assess the best results for each approach and compare whether the other approach confirms or rejects the findings.

9.4.5 Study execution

In the following paragraphs, we describe the steps of the study execution for each research question. A visual representation of the study execution is given in Figure 9.1.



(a) Dataset construction and study design.

(b) Detailed study execution on an exemplary LSI-CD dataset.

Figure 9.4: This figure provides a high level (left hand side) as well as a detailed view (right hand side) of the study design.

Setup for RQ 1 To perform the study, we manually compare the best 200 reported clone pairs with the best 200 re-implementation pairs found by our approach using LSI. We, thus, compute for each primary technique the *precision at rank 200* and validate whether the confirming technique also reports a similarity value for the same result pairs.

Producing rankable, file-based, similarity values from the clone detection results is not straightforward: it requires a mapping between a purely syntactical view, transcending class and file scopes, to a semantic one, restricted to the file level. We, therefore, had to determine what could be the best indicator for re-implementations in terms of clone detection results. Faced with the choice between ranking by maximal clone length and the sum of cloned statements, we opted for the latter for the following reasons: Starting from a pair of files, we assume that the more cloned portions of code exist between these files, the higher the odds that they might be (or contain) re-implementations of one another. For large clones, this might be obvious. However, in the case of re-implementations, it is likely that structural similarity has decayed significantly (if it ever existed) and, thus, only small cloned fragments exist. In an ordering based on clone size, these fragments would rank very low and would likely be missed as interesting indicators. An aggregation, therefore, would push files with many small shared clones to a higher position. We, thus, decided to aggregate the number of cloned statements on a file-basis and rank the pairs in a descending order. We refer to this measure as *Aggregated Clone Detection, ACD*.

For each pair resulting in one of the two result sets, we first compared the two constituents to validate whether they were indeed re-implementations and marked them accordingly.¹⁴ If they were, we were interested whether the pair was found by both approaches. We define *found* for our approach as an LSI *similarity score of* >0.5 and as *present* in the clone detection results. If the pair was found only by either of the approaches, we analyzed the reasons. In case of resulting pairs that were not re-implementations, we also marked them accordingly, and provided a rationale for exclusion. On this basis, we assess the performance of both techniques with respect to their number of true positives, true negatives, false positives and false negatives. In addition, we count the number of mutual agreements of both techniques on each result set.

Setup for RQ 2: Based on the mutual agreement of the two analyses, we assess the quality of the overlapping results for each branch of comparison.

Setup for RQ 3: To gain an understanding of the practical applicability of our results, we provided system experts with a list of all true positives provided by both analyses. We invited the experts to assess our findings by indicating the relevance of the positive hits by expressing whether the finding would entail action on their part. Specifically, they ranked the findings according to *priority fix*, *occasional fix*, *won't fix*. At the time of writing, we revisited these assessments and found that priority fixes had partially been resolved already, resulting in the category *fixed*.

¹⁴We considered pairs as re-implementations if they shared at least enough semantic and contextual overlap to warrant a unified implementation.

9.4.6 Results

We report on the results of our study according to our RQs.

RQ1. Do LSI and ACD produce different results?: The following paragraphs detail first on the comparison branch LSI-based approach vs. ACD and then on ACD vs. LSI-based approach (recall Figure 9.4 for the study design). For each primary technique, we report the precision at rank 200. For the confirmatory technique, we report the number of confirmed and missed true positives, as well as shared false positives.

Results of LSI-based approach vs. ACD

Out of the top 200 results of applying our approach with LSI, we obtained the following results (summarized in column LSI-CD of Table 9.2): 65/200 pairs were re-implementations. This leaves a large number of incorrect hits in the result set, which we inspected further: we found that the largest share (99/200 pairs) of unrelated pairs were included due to interface implementations that led to common identifiers but no re-implementations. A smaller share (18/200 pairs) was due to a system convention in handling exceptions, whilst an even smaller share (18/200) was due to accidental clones¹⁵ that happened to share identifiers.

Primary findings: LSI in this setting produced 65/200 correct hits (marked as true positives TP in the upper half of Table 9.3) and 135/200 incorrect hits (marked as false positives FP). Within the first 100 top hits, LSI has a density of positive results of 50%. However, when excluding the bad matches due to interface implementations and exception handling, the number of false positives decreases to 18. A filtering of these types of false positives suggests itself even more strongly when considering the range of the LSI similarity scores for the top 200 result pairs: currently, they start at 1.0 and only decrease to 0.96. This suggests that a considerable share of true positives might fall below the threshold set for manual inspection. Overall, the precision at rank 200 of the unfiltered result of LSI is 33%.

Recall of ACD: In comparison, the clone detection produces the following results on the same pairs: it correctly identifies 49/65 of the present re-implementations and correctly discards 125/135 unrelated pairs. However, it misses 16/65 correct pairs, with no or below the threshold overlap of cloned statements. Furthermore, it incorrectly identifies 10/135 unrelated results as hits. This mostly happens in the presence of accidental cloning.

Summary: In this setting, LSI provides 16 pairs of re-implementations that were not retrieved by the clone detection. 8 of these pairs fell below the threshold of the clone detection, the other 8 were structurally differing re-implementations that partially stemmed from strongly modified type-3 clones.

¹⁵Accidental clones [139] are code fragments of different origin that are syntactically similar due to the adherence to a specific protocol.

Table 9.2: Profile of the top 200 result pairs for LSI and ACD.

Re-implementation	LSI	ACD
pairs assessed	200	200
yes	52	89
yes - deprecated	13	15
yes - total	65	104
no	5	3
no - accidental clones	13	93
no - exception	18	0
no - interface	99	0
no - total	135	96

ACD vs. LSI-based approach

Out of the top 200 results of applying Clone Detection and aggregating the results, we obtained the following results (summarized in column CD-LSI of Table 9.2): 104/200 pairs were re-implementations. This leaves nearly 50% of incorrect hits (96/200) in the result set, which we inspected further: we found that accidental clones induced by specific system conventions were almost exclusively responsible for this phenomenon (93/96 pairs).

Primary findings: ACD in this setting produced 104/200 correct hits (marked as true positives TP in the lower half of Table 9.3) and 95/200 incorrect hits (marked as false positives FP), mostly due to accidental clones. Interface implementations did not impact the results and the exception handling usually was small enough in terms of statements as to fall under the threshold of 10 statements set for the analysis. The sum of aggregated cloned statements of the top 200 result pairs ranges from 615 to 21 within a result pair. Overall, the precision at rank 200 for ACD is 52%.

Recall of LSI: In comparison, LSI produced the following results on the same pairs: it correctly identifies 67/104 of the present re-implementations and correctly discards 74/95 unrelated pairs. However, it misses 38/104 correct pairs, that often have retained structural similarity, but have been extensively renamed after duplication. Furthermore, it incorrectly identifies 21/95 unrelated pairs as relevant, a phenomenon occurring in the presence of accidental clones that share particular identifiers, such as for GUI or database functionality.

Summary: In this setting, the clone detection provides 37 pairs of re-implementations that were missed by LSI. 21 of these pairs were relatively small cloned portions of code in large classes that otherwise were not related. 16 pairs were cloned but extensively renamed.

RQ2. Does intersecting the result sets of both analyses improve the quality of the results? : As shown in Table 9.3, in both comparisons the secondary approach confirms a large portion of the results considered as true positives by the primary approach. In terms of error in the intersection, the intersection *LSI-based approach* & *ACD* with 83% true positives provides a better, albeit smaller, set of results than *ACD* & *LSI-based approach* with 76% true positives. We conclude that both intersections produce a manageable set of high-quality results.

Table 9.3: Hits of top 200 result pairs for LSI (LSI-ACD) and hits of top 200 result pairs for aggregated clone detection (ACD-LSI).

Results LSI primary - ACD confirmatory			
LSI TP	65	ACD TP	49
LSI TN	0	ACD TN	125
LSI FP	135*	ACD FP	10
LSI FN	0	ACD FN	16
LSI FP cleansed	18		
Agreement on 59 pairs, 49 TP, 10 FP			
Results ACD primary - LSI confirmatory			
LSI TP	67	ACD TP	104
LSI TN	75	ACD TN	0
LSI FP	21	ACD FP	96
LSI FN	38	ACD FN	0
Agreement on 88 pairs, 67 TP, 21 FP			

Furthermore, they significantly outperform the precision of the result sets obtained with the isolated techniques.

RQ3. Are the results of the combined analyses relevant for practitioners?: As can be seen from Table 9.2, some of the re-implementations found by our analysis are marked as *deprecated*. We applied this marker to result pairs that were re-implementations with at least one constituent marked as deprecated in the source code. The respective files were usually versions of older releases kept in the system for the sake of backward compatibility. In this context, they were unlikely to be actionable hits, so we excluded them from the result sets provided to the system experts for assessment.

Practical relevance of results - LSI-based approach vs. ACD

For this setup, we provided system experts with 52 result pairs, and asked them to mark for each finding if it should induce an action. 7/52 pairs were scheduled for fixes immediately, 34/52 pairs were set to be fixed given an occasion, i.e. someone working on the respective piece of code should inspect and resolve the finding, and 11/52 pairs were discarded as they were duplicates due to deliberate reasons, e.g., architecture decisions and system conventions. Summing up, 41/52 pairs were considered actionable.

Practical relevance of results - ACD vs. LSI-based approach

As for the first setup, we provided system experts with the appropriate set of re-implementations, in this case 89 pairs. 12/89 pairs were scheduled for fixes immediately, 15/89 pairs were scheduled for a priority fix, 51/89 pairs were set to be fixed given an occasion. 12/89 pairs were discarded as they were duplicated deliberately. In total, 78/89 pairs were considered actionable by the system experts.

Table 9.4: Practitioner rating of non-deprecated re-implementations.

Practitioner verdict	LSI-CD	CD-LSI
practitioner - fixed	7	12
practitioner - priority fix	0	15
practitioner - occasional fix	34	51
practitioner - no fix	11	12
practitioner - actionable	41/52	78/89

9.4.7 Discussion

In this section we answer our starting question by combining the findings of our research questions, reported above. We discuss the lessons learned from performing the study and present potential mitigation to the encountered challenges.

Can LSI and clone detection complement each other to detect re-implementations?

With our study, we intended to investigate whether a combination of LSI and clone detection could be feasible for detecting re-implementations. Based on our results, we argue that this is the case: first, each technique found instances of re-implementations that were missed by the other. Second, each technique could to some degree compensate the weaknesses of the other: when analyzing the top 200 clone detection results, LSI could exclude a number of accidental clones that were semantically unrelated. When analyzing the top 200 LSI results, clone detection correctly excluded findings based on interface implementations.

Lessons learned

When evaluating the results, we identified several strengths and weaknesses of the two techniques. We relate the essence of our experience in the following paragraphs.

Aggregated clone detection

An experiment by Juergens et al. [168] suggests that the results of clone detection per se do not promise particular potential for the detection of semantic re-implementations. Therefore, the clone detection seemed an ideal baseline candidate for our goal of identifying additional benefit of IR for detection of re-implementations. However, to perform the study, we faced the challenge to find a common granularity as basis for our results. This resulted in the technical decision to aggregate the number of cloned statements on a file basis. This decision proved more consequential than we expected.

True positives A large number of re-implementations present in our study object were portions of code that were initially copied and changed significantly over time to an extent that even Type-3 capable detectors would miss the big picture. In a standard setting of clone detection, only small fragments would have been captured that would have been lost in the vast number of findings. The aggregation on a class basis captured this historical context more comprehensively by rating pairs sharing many small cloned fragments significantly higher than by ranking them based on their largest shared clone. In terms of functionality, the findings included utility functionality, data structures, and domain specific redundancies.

False positives As is to be expected for a clone detector, accidental clones constituted a large part of the false positives in the primary clone detection result set. However, this was offset by considering the LSI value for the respective pairs.

False negatives As expected, re-implementations that were lacking structured similarity were missed by clone detection.

LSI-based approach

Previous [39, 176] and related work (see Section 10.6) have indicated the suitability of IR techniques, particularly LSI, for our goal. However, we identify the challenge that, contrarily to clone detection, the LSI scores hardly decrease over the inspected range of pairs. We are, thus, likely to have covered only a fraction of interesting result pairs. This issue could have been mitigated by removing the large number of false positives due to system conventions. However, we decided against this measure to avoid biasing the data set. Future studies should consider these learnings.

True negatives LSI proved beneficial to downrate the accidental clones present in the result set of the clone detection.

False positives Our results show that filtering identifiers based on code structure is insufficient for LSI to produce good results. System conventions need to be known to remove pairs that, on an identifier basis, seem re-implementations, but, on inspection, prove functionally unrelated.

False negatives tended to occur for result pairs that shared re-implementations that were comparatively small with respect to the overall size of the given files. In this case, the large number of unrelated identifiers caused LSI to miss these instances.

Potential applications

We identify two potential applications of our findings:

Use Case 1 - a focussed view on re-implementations Our results indicate a favorable ordering in combining the two techniques when aiming for a focussed result set of re-implementations: First by aggregated clones, then by LSI values. Interestingly, this order produces relevant findings in the following way: huge cloned segments, are by default ranked in a high position. In addition, the many small remaining fragments, originally stemming from the same code but diverged so much that they fall out of the type-3 detection, are promoted due to the aggregation on class level and, thus, ranked high enough for top results. LSI mostly confirms them.

Potentially, this combination could be used to improve the prioritization of clone detection findings by removing a large portion of unwanted results. This hypothesis is backed by the assessment of practitioners that considered 88% of the positive results of this set as actionable.

Use Case 2 - identifying semantic re-implementations Our results indicate that LSI can identify additional pairs of re-implementations that are missed even by aggregated clone detection. However, as seen by the results of our study, system conventions and programming language details should be considered to remove a large share of unwanted hits from the candidate result set.

Both use cases can be useful for determining the shape of re-implementations in the context of assessing reuse potential during RASM application.

9.4.8 Threats to validity

Human Error Manual assessment is possibly flawed as the judgements made on the resulting pairs rely on our subjective judgment. To mitigate this, we applied researcher triangulation. Furthermore, to avoid bias towards our approach, we opted for a conservative assessment, rating result pairs as negative hits when in doubt.

Study Object We selected the study object since expert opinion suggested the presence of re-implementations. However, the total number of re-implementations in the system is unknown, so we cannot report the recall of our results. Also, we can not project the experienced density to other systems.

Scope of analysis Our study focused on a file granularity. Therefore, the re-implementations we found are bound in size by this limitation. Whilst this is common for a number of quality analyses (e.g., clone detection, size measures, etc.), different techniques need to be applied to find re-implementations on a more abstract level.

External validity Our study bases on one study object and cannot provide results that are generalizable. Nevertheless, we believe that this study provides clear indications that a combination of the two techniques improves their performance for the detection of re-implementations. Further work is needed for additional evidence.

9.4.9 Summary

In this section, we investigated whether a combination of LSI and clone detection could improve the results each technique by itself produced for re-implementations. We evaluated the combination of both techniques in a case study on an industrial system. The results suggest that (1) latent semantic indexing and clone detection complement each other, (2) aggregated clone detection can be a better indicator for re-implementations than LSI, and (3) the combination of the techniques provides high quality result sets which were considered relevant and actionable by practitioners.

The study highlighted important limitations of both techniques: for the LSI-based approach, the quality of the results was severely impacted by the false positives due to identifiers shared, e.g., by interface implementations and exception handling. This suggests that results could be improved by extending the pre-processing or filtering unwanted types of matches from the result set. For the clone detection, accidental clones accounted for most of the false positives, but could mostly be eliminated by LSI.

Overall, we found that in isolation Aggregated Clone Detection outperformed the LSI-based approach with a precision of 52% vs. 33%. However, intersecting the results of each technique with the confirmatory results of the other greatly improved the precision of the respective result sets: the intersection *ACD&LSI* yielded a precision of 76% true positives and the precision

of the intersection $LSI \mathcal{E} ACD$ significantly increased to 83%. In addition, both intersections significantly reduced the size of the result sets, producing manageable sets of high-quality results that were considered actionable by practitioners. We, thus, conclude that a combination of the two techniques can improve the precision of the detection of re-implementations. Nevertheless, replication on a constructed dataset with a known number of re-implementations is needed to measure the recall of the proposed technique.

9.5 Cross-project clone detection as guidance for reuse improvement

Code clones are known to introduce significant maintenance costs and decrease the quality of a code base [63]. On the other hand, code clones indicate the need for equal or similar functionality in several places [179]. Whilst determining the clone coverage for single projects is already widely researched, less is known on the extent and characteristics of cross-project clones. In this Section, we present results on cross-project clones and propose an approach on how clone detection could be used as guidance for reuse improvement. Parts of this work was published in [37].

9.5.1 Assessing cross-project clones for reuse optimization

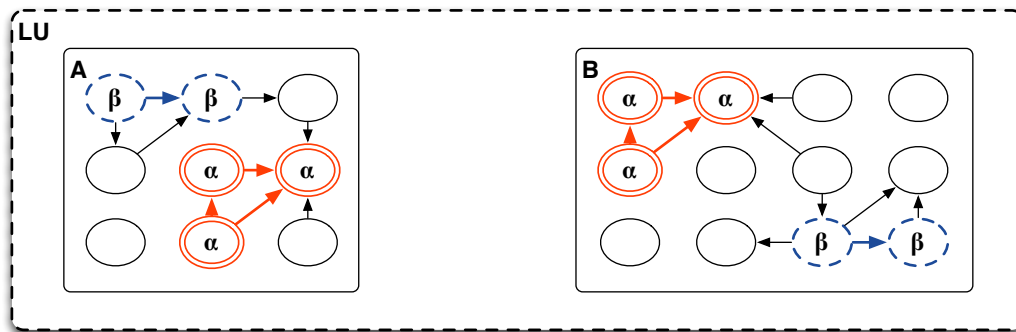
Organizational structures (e. g., separate accounting, heterogeneous infrastructure, or different development processes) often restrict systematic reuse among projects within companies. As a consequence, code is often copied between projects which increases maintenance costs and can cause failures due to inconsistent bug fixing. Assessing cross-project clones helps to uncover organizational obstacles for code reuse and to leverage other ways of systematic reuse. Furthermore, knowing how strongly clones are entangled with the surrounding code helps to decide if and how to extract them to commonly used libraries. We propose to combine cross-project clone detection and dependency analyses to detect (1) what is cloned between projects, (2) how far the cloned code is entangled with the surrounding system and (3) what are candidates for extraction into common libraries.

9.5.2 Motivation

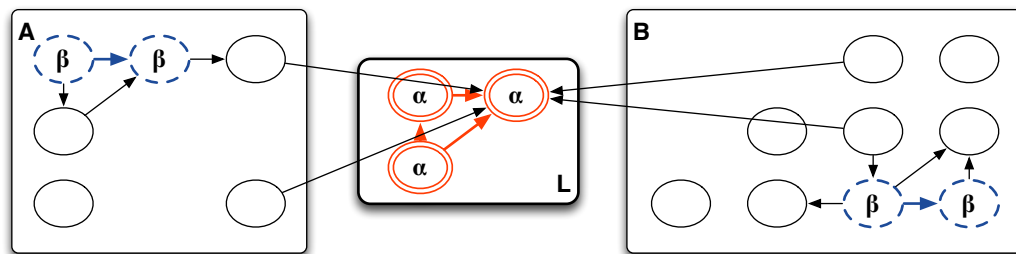
Companies developing software systems for a specific application domain are often confronted with recurring implementation tasks. Since a lot of domain knowledge is manifested in the code, it suggests itself to reuse successful solutions accross project boundaries. However, organizational factors can restrict structured reuse among projects. For example, separate accounting of charges, different development models or heterogeneous infrastructure can prevent teams to create commonly accessible, reusable, software modules.

Consequently, cloning is a popular way to reuse successful implementations across project boundaries [60]. Especially if a developer implemented a solution in a previous project, (s)he is likely to copy and adapt it. According to [180] as well as to own experience, these cross-project clones (CPCs) tend to be larger portions of code, ranging from several methods to files or entire subsystems. This “simple” way of overcoming reuse difficulties is known to have negative consequences on maintenance costs and software quality in the long term [?].

We propose to combine cross-project clone detection (CPCD) with static dependency analyses to assess and manage this aspect of code reuse with the goal to provide input for organizational and maintenance decisions concerning reuse.



(a) Cross-project clone detection and dependency analysis



(b) Goal: Extraction of candidates in library

Figure 9.5: The figure illustrates the three steps of our idea. A and B denote two software systems. LU denotes the logical union of the systems during clone detection. Circles represent entities of the systems, e.g. functions, classes, or files. The systems contain two clone classes: the double-lined (α) and the dashed (β) group. Arrows denote the dependencies between entities belonging or connected to clones. L denotes an extracted library, containing (α).

9.5.3 Idea

We would like to answer the following questions: To which amount does cross-project cloning (CPC) exist among a specific set of projects (within a company, a division, of a certain technology etc.)? Do cross-project clones (CPCs) implement specific functionality? Are there CPCs that would qualify as candidates for library creation? If yes, which of the CPCs could be extracted with reasonable effort? What are the organizational reasons that give rise to CPC?

First, we run a clone detection on all systems of interest, treated as one logical unit (*LU* in Figure 9.5(a)). We filter the findings to CPCs. The result already provides relevant information: the extent of cloning within systems compared to the extent of cloning across systems and the total extent of CPC. Therefore, we can quantify CPC at this point.

Next, we aim to detect clone classes potentially implementing coherent functionality. We attempt this by identifying clones that form weakly connected components in the dependency graph. We argue that this is reasonable as own experience as well as existing work (c.f. [180])

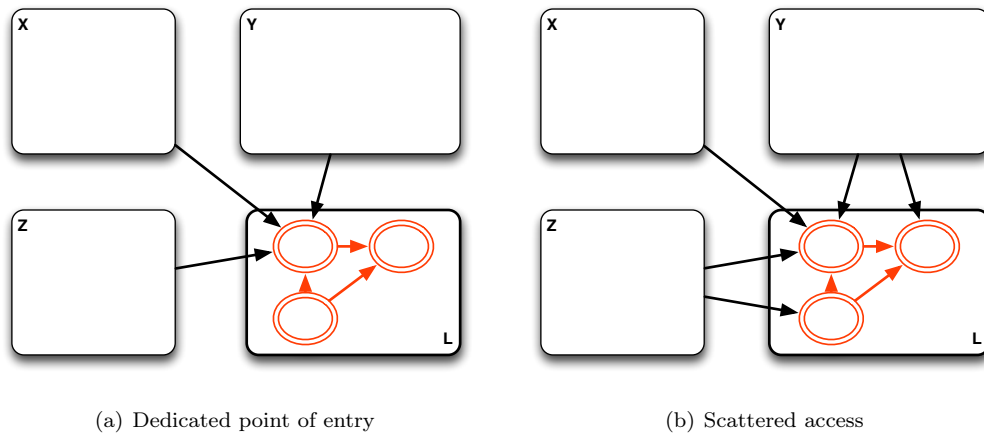


Figure 9.6: Different usage patterns of cloned regions.

have indicated CPCs to be rather large, potentially comprising several functions, classes, or even entire system components. We expect most of these large chunks of code to be reused in a library-style manner, i. e., being called from the surrounding system parts, but not calling back into the system. Based on these assumptions, we propose to run a static dependency analysis to identify clones that are candidates for extraction.

To achieve a sufficient precision, the analysis works on a method granularity. For each clone class, we assess all instances, as usage patterns might differ. For each clone, we start from its location in the code. On the dependency graph, we trace all outgoing calls until we reach the border of the cloned region. We then assess the number of calls from the cloned region into uncloned system code. Ideal extraction candidates do not call into the system (see clone α in Figure 9.5). Clones with a large number of calls into system code are less desirable for extraction (see clone β in Figure 9.5). In practice, we expect calls into the system to happen and propose to use a threshold value for the number considered acceptable¹⁶.

From the clone-system border in the dependency graph, we trace the calls from the system into the cloned region. In this way, we can collect usage profiles of the clones. Figure 9.6 shows two examples of potential usage patterns. Comparing the calling patterns within the same clone class allows to deduce requirements for an extracted library.

The last step is to assess the functionality implemented by the extraction candidates. The process of functionality assessment serves as input for library creation. Figure 9.5(b) illustrates the goal of our idea: the candidate has been extracted into a new library, which both of the systems call.

9.5.4 Usage scenarios

Reuse assessment: Being able to quantify the extent of cloning between projects enables to understand dimension and causes of reuse by cloning. For example, the existing technical infras-

¹⁶Arguably, very prominent but strongly connected clone groups should also be assessed manually.

structure might hamper sharing code between teams. Improving the development infrastructure would enable developers to use structural reuse methods more often.

Identify demand for certain libraries: To make a library appealing for developers, it should provide a common set of helpful solutions for a certain class of problems (such as I/O, printing or DB access). Being able to identify cloned code helps to characterize the reused functionality. Thereby, the demand for solutions for a class of problems can be determined. For example, if I/O handling code is among the most frequently cloned functionality, a commonly accessible library providing I/O functionality would offer developers an alternative to cloning code between projects.

Assess effort for code extraction: Knowing how far cloned code is entangled with the surrounding system helps to estimate the complexity to separate it by creating a library. If, e.g., cloned code is strongly interwoven in several projects, extracting the code might require exorbitant resources compared to the resulting savings in maintenance effort.

9.5.5 Related work

Cross-project clone detection Mende et al. [126] propose CPCD to support the grow-and-prune model [181] for Software Product Lines management. Schwarz et al. [182] provide evidence of a significant amount of CPC for the Smalltalk ecosystem. Furthermore, they propose scalable clone-detection techniques for CPCD. Krinke et al. [180] assessed the copying and cloning between projects of the GNOME Desktop Suite, studying the flow of code between the different projects. Al-Ekram et al. [139] investigate cloning “by accident”, a consequence of projects implementing identical code portions. Instances could be candidates for inclusion into libraries.

Higher-level clones Basit and Jarzabek introduce structural clones [170], logical groups of simple clones. We follow this idea to better identify code portions with coherent functionality that are candidates for extraction into libraries.

Assessment of reuse In earlier work [34, 38] we proposed an assessment model for usage of third-party libraries as well as code reuse within organizations. CPCD will be integrated as one measure for the maturity of code reuse.

9.5.6 Summary and next steps

Identifying potential for reusable entities is an important step towards beneficial reuse. In this section, we proposed a method based on cross-project clone detection to detect clusters of source code clones that have found application in several projects and, thus, are of value for reuse. We suggest a pragmatic ranking of the identified clusters based on their degree and type of dependencies on the embedding source code.

In a next step, we aim to implement the proposed idea, building on the clone detection and dependency analyses provided by the ConQAT¹⁷ toolchain. To assess the benefit for development and maintenance in practice, we plan to apply our idea on a set of projects of an industrial

¹⁷www.conqat.org

partner. Upon completion, the analysis can be applied during an assessment phase with RASM and as monitoring instrument during the adoption of the selected approach.

9.6 Conclusion

In this chapter, we have presented methods and tools that support the identification of candidate reusables in the form of missed reuse opportunities and cross-project code clones. Such assessment is informative during an application of RASM, as it provides estimates of the cost of lacking reuse that are based on the source code.

Missed reuse opportunities With respect to missed reuse opportunities, we addressed the aspect of identifying potential re-implementations. Their presence indicates that reuse is not adequately effected in an organization. In particular, they represent a cost factor for development and maintenance that is not easily discovered without tool support.

As we have seen in Section 9.2, addressing re-implementations can be challenging even in the presence of tool support due to the daunting number of results. This number of findings discourages practitioners and can freeze improvement attempts.

As a consequence, in Section 9.4, we proposed a refined approach that combines clone detection with LSI to provide more reliable and focused findings. We evaluated this refined approach in an industrial case study. In the context of reuse adoption and improvement, the results of the case study are encouraging: It provides a compact set of missed reuse opportunities that, according to practitioners, are relevant to them.

Candidate reusables Re-implementations can indicate the need of a shared reusable. Another, more concrete, indicator for a candidate reusable is the presence of cross-project clones: their existence is proof for the need of (near-)identical functionality in multiple places, even across projects. This suggests a certain domain independence, which is beneficial for a reusable as, in this case, it can serve a wide range of stakeholders. In this chapter, we proposed an approach to detect cross-project-clones and rank them according to their suitability for extraction into a shared reusable.

During future applications of RASM, we aim enhance the assessment by applying the presented methods and tools.

10 | A structured assessment model for third-party library usage

Most organizations employ third-party software for specialized parts of their systems. Whilst the legal aspects of these uses are usually addressed with due diligence, the selection process and the assessment of potential and risk of the respective pieces of software tend to be unstructured with unclear governance. Furthermore, the intensity and location of use are rarely monitored. As a consequence, organizations lack an overview of their de-facto employed third-party software and are unaware of the compliance with respect to the intended use. Furthermore, they have no objective basis on how the different characteristics of those entities impact their quality goals. However, these impacts can affect the success of a reuse strategy and, therefore, should be considered in the course of a reuse assessment. The effects of unmonitored library reuse often surface during maintenance. To support practitioners with an earlier risk assessment, this chapter provides a structured assessment approach for third-party libraries that captures characteristics of third-party software and their impacts on maintenance activities. Apart from supporting an initial reuse assessment with RASM, the approach supports monitoring the changes of a new reuse strategy, e.g., the proliferation or removal of certain libraries within specific system parts. In addition, it can be extended to measure and monitor the use of company-internal third-party libraries. This chapter presents the instantiation of the assessment support for specific and detailed aspects of reuse. Parts of this work have been published in [26, 34].

Contents

10.1 Opportunities and risks of third-party library reuse	172
10.2 Assessment model	173
10.3 Assessment process	178
10.4 Tool support	179
10.5 Case study	181
10.6 Related work	186
10.7 Summary and future work	188

10.1 Opportunities and risks of third-party library reuse

Reuse with software libraries plays a central role in modern software development—instead of writing a complete software system from scratch, significant parts of its building blocks are reused from third-party libraries. Especially for widely-used platforms like Java, a considerable amount of reusable libraries with a large variety of functionality is available in code repositories on the Internet. In a recent study on reuse in Java open source projects we found that for almost half of the projects the amount of reused code exceeded the amount of newly developed code [22].

However, library reuse comes at a cost: Included libraries can significantly impact the maintainability of a software system. Often, projects use a number of different libraries and the code is highly entangled with their APIs. This poses multiple risks to the evolution of the software. First, libraries continuously evolve. New releases provide added functionality and bug fixes. In many cases, it is desirable to migrate a software system to the latest stable release of a library, especially in case of critical bugs such as security flaws. However, migration can cause considerable maintenance effort, as backward-compatibility may not always be ensured. Second, a library might be still unstable and introduce bugs into the software, which could be difficult to find and hard to fix. Third, the provider's support or maintenance of a library might be discontinued, such that the library consumer can no longer expect fixes for critical bugs. Finally, the license of a library or the legal constraints in a project may change, forcing the project to stop employing the library. Again, a potentially costly replacement with a different library or an own implementation of the reused functionality is required. Unfortunately, most existing approaches that aim at assessing the maintainability of a software system primarily focus on the project's own code. The included libraries are often disregarded, missing important aspects affecting maintainability.

Problem A plethora of external software libraries form a significant part of modern software systems. Consequently, these systems contain a considerable fraction of code developed and maintained by third parties. Therefore, external libraries and their usage have a significant impact on the maintenance of the including software. Unfortunately, third-party libraries are often neglected in quality assessments of software, leading to unidentified risks for the future evolution of the software.

Contribution Based on industry needs, we propose a structured approach for the systematic assessment of third-party library usage in software projects. It can be applied to support specific maintenance decisions as well as to monitor the project's state of reuse over time. The approach is supported by a comprehensive assessment model relating key characteristics of software library usage to development activities. The model defines how different aspects of library usage influence the activities and, thus, allows to assess if and to what extent the usage of third-party libraries impacts the development activities of a given project. Furthermore, we provide guidance for executing the assessment in practice, including tool support for a pre-selection of important libraries and multiple automated static code analyses. We evaluate the approach with a case study involving an industrial software system of 3.5 MLOC including about 90 external libraries.

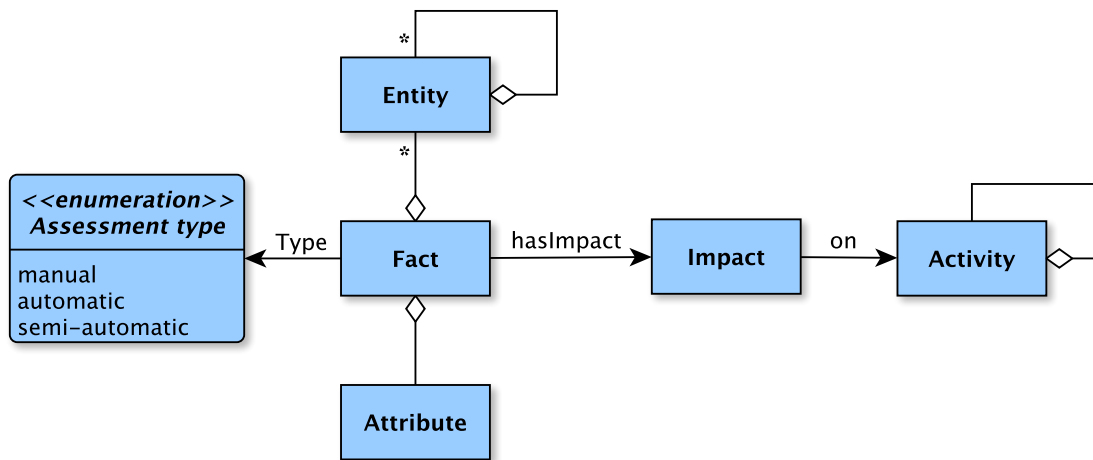


Figure 10.1: The meta-model of the assessment model.

Outline The remainder of the chapter is organized as follows: Section 10.5 introduces the industry partners of this work. Section 10.2 presents the assessment model, whilst Section 10.3 details the assessment process. Section 10.4 describes the tool support provided for the analysis. Section 10.5 presents design and results of the case study. The following Sections 10.5.2 and 10.5.3 discuss results and threats to validity. Section 10.6 gives an overview on related work before Section 10.7 summarizes the chapter and identifies future work.

10.2 Assessment model

The proposed assessment model is inspired by activity-based quality models [183] that offer a soundly structured and precise way of expressing quality factors and their mutual dependencies. In this model, we follow the terminology coined by Kitchenham et al. where *entities* “are the objects we observe in the real world” and *attributes* are “the properties that an entity possesses” [184]. We extend this by allowing an attribute to be attached to multiple entities and adapted the meta-model from [183], as shown in Figure 10.1. This is necessary as several relevant library attributes emerge only from the relation between entities. Figure 10.2, which shows the concrete instantiation of the meta-model, contains examples: for instance, the attribute ENTANGLEDNESS is attached to the entities System and Library as it models a characteristic that involves both.

Entities are structured in a hierarchical manner to foster completeness. The combination of one or more entities and an attribute is called a *fact*. Facts are expressed as [Entities | ATTRIBUTE]. A fact has an *assessment type*, which can be a manual assessment by an expert, an automatic analysis, or semi-automatic as a combination of the above. To express the impact of a fact, the model relates the fact to a development *activity*. This relation can either be positive, i. e., the fact eases

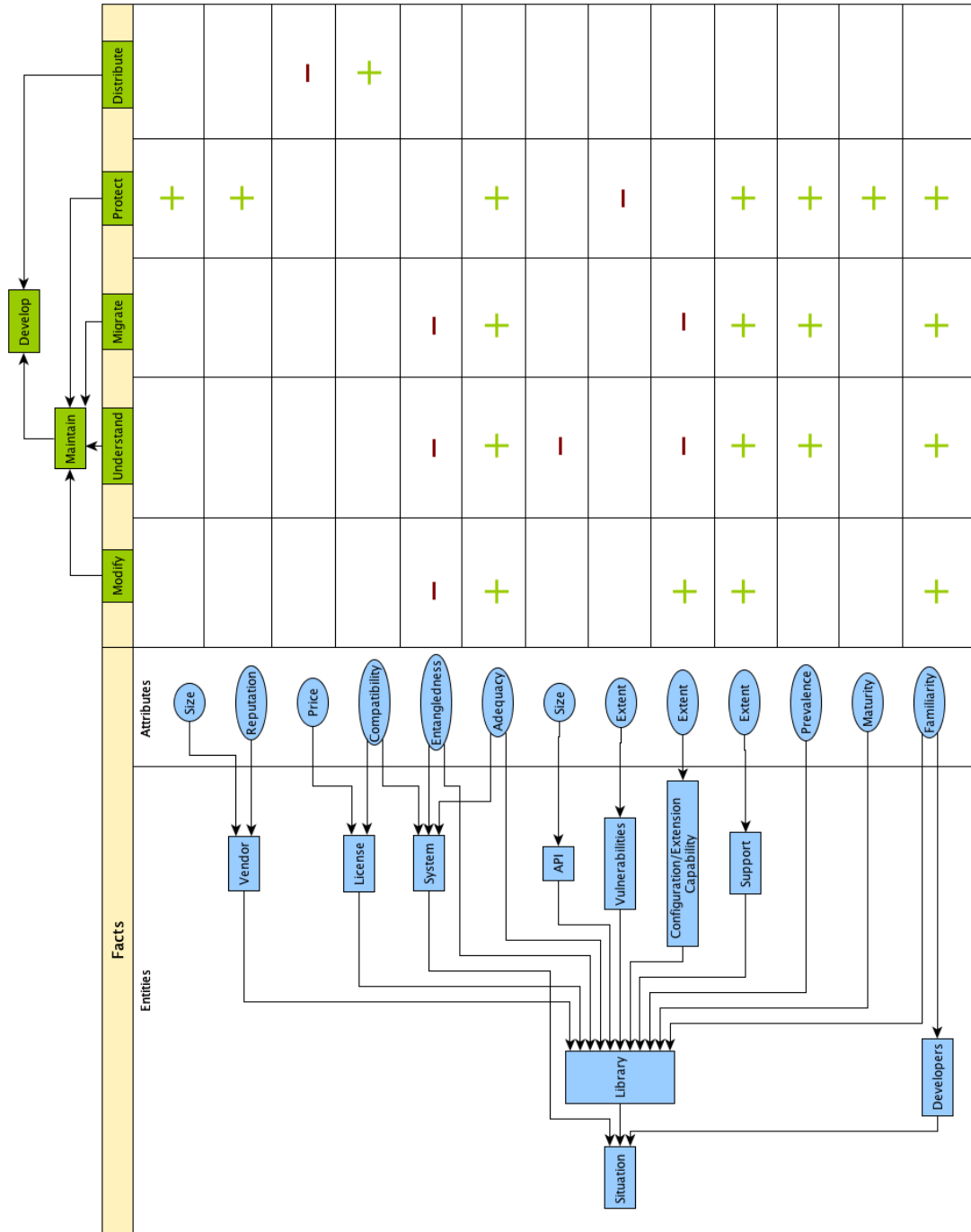


Figure 10.2: Instantiated assessment model with facts, development activities, and impacts between them.

the affected activity, or negative, i. e., the fact impedes the activity. In Figure 10.2, for example, the ENTANGLEDNESS of System and Library has a negative impact on all maintenance activities whereas the protection against security-relevant attacks as well as the legal aspects of the distribution of the system are not affected by this fact. *Impacts* are expressed as [Entity | ATTRIBUTE] $\xrightarrow{+/-}$ [Activity]. Each impact is backed by a *justification*, which provides the rationale for its inclusion in the model.

So far, the model constitutes a *Definition Model* in the sense of [185] as it defines quality aspects and their relations. To become an assessment model that can be used to assess a specific situation, it needs to be enriched with metrics and a measurement method. We provide this with the three-value ordinal scale $\{low, medium, high\}$ that is used to *quantify* facts. The assessment of the facts is mainly a manual activity performed by an expert who bases the judgement on a set of metrics that may be determined automatically. Table 10.2 shows the metrics used in our model.

To *assess* the impact on the activities, we use the three-value scale $\{bad, satisfactory, good\}$. If the relation between a fact is positive, there is a straight-forward mapping from $low \rightarrow bad$, $medium \rightarrow satisfactory$, $high \rightarrow good$. If the fact [Library | PREVALENCE], for example, is rated high, the effect on the activity *migrate* is good as the impact relation is positive [Library | PREVALENCE] $\xrightarrow{+}$ [Migrate] (see Figure 10.2) as a high prevalence of a library usually gives rise to alternative implementations of the required functionality. If the impact relation is negative, the mapping is turned around: $low \rightarrow good$, $medium \rightarrow satisfactory$, $high \rightarrow bad$. A high [Library, System | ENTANGLEDNESS], for example, results in a bad effect on the activity *understand* as the relation is negative: [System, Library | ENTANGLEDNESS] $\xrightarrow{-}$ [Understand]. A high entangledness tends to cause difficulties to clearly understand how a library is to be used. Especially scatteredness of method calls could hamper understanding.

The assessment of a single library thus results in a mapping between the activities and the $\{bad, satisfactory, good\}$ scale. To aggregate the results, we simply count the occurrences of each value at the leaf activities. Hence, the assessment of a library finally results in mapping from $\{bad, satisfactory, good\} \rightarrow \mathbb{N}_0$. We deliberately do not use an aggregation that results in a single number to avoid comparing apples with oranges.

Activities

With one exception, the activities included in the model are typical for maintenance. The following paragraphs briefly describe each of them.

Modify: Modifying a system means to change it to e. g., fix a bug or add new functionality.

Understand: Understanding a system is critical for all maintenance tasks. Developers need to understand the structure and functionality of a system before they can start to extend or change it.

Migrate: Migration is the process of exchanging a library with a newer version or a different library. When migrating a software system to another library, it is usually important that the new library comprise the same functionality as the old one.

Protect: Often, security issues surface during the productive usage of a software system. It is important to fix these during maintenance, e.g., by migrating the system to an updated library version.

Distribute: Distributing a software system refers to any kind of proliferation, such as providing the system open source on the Internet or shipping the product to a commercial customer. This activity is not directly linked to maintenance. However, third-party libraries impact it.

Metrics

The model quantifies each fact with one or more metrics. The list of metrics, their description and assignment to facts are shown in Table 10.2. As an example, to quantify the extent of vulnerabilities of a library, we measure the number of known critical issues in the bug database of the library. Some of the facts cannot be measured directly, as they depend on many aspects. For instance, the maturity of a library cannot be captured with a single metric but must be judged by an expert. We do not employ an automatic, e.g., threshold-based, mapping from metric values to the {low, medium, high} scale but fully rely on the experts capabilities.

Impacts

Impacts define how facts influence activities. A justification for each impact provides a rationale for the impact which increases confirmability of the model and the assessments based on the model. Note that a fact might have a positive impact on one activity whilst negatively impacting another one. To ensure readability, we give one example of an impact per activity. The complete list can be found in the Appendix 12.3.

[Extension/Configuration Capability | EXTENT] $\xrightarrow{+}$ [Modify] A high extension capability of an external library positively impacts modifications, such as adding new features, because it increases the chances that they can be accomplished with the same library.

[Extension/Configuration Capability | EXTENT] $\xrightarrow{-}$ [Understand] Whilst high extension capability positively impacts modifications, it hinders understanding. The reason is the high complexity caused by the flexibility of extension mechanisms which make it harder to understand how to use the library.

[Extension/Configuration Capability | EXTENT] $\xrightarrow{-}$ [Migrate] A high capability for extension and configuration inhibits migration as it is less likely that alternatives provide the same flexibility.

[Vendor | REPUTATION] $\xrightarrow{+}$ [Protect] The reputation of the library vendor positively influences protection of a system, as a renowned vendor can be expected to provide critical updates in a timely manner.

[License, System | COMPATIBILITY] $\xrightarrow{+}$ [Distribute] Characteristics of third-party libraries also impact the distribution of a system, e.g., low compatibility of licenses can block the distribution of a system.

Fact	Metric	Description
[Support EXTENT]	Expert assessment	Availability of support and training
[Vendor SIZE]	Headcount Sales Volume	The number of contributors is determined e. g., from the vendor's website The sales volume is determined e. g., from the vendor's website
[Vendor REPUTATION]	Expert assessment	Reputation as perceived by an analyst/domain expert
[License PRICE]	Price	Price for a redistribution license
[Library PREVALENCE]	#Books #Google Hits #Job Advertisements	Number of search results on Amazon Number of search results on Google Number of search results on jobpilot.de
[Library MATURITY]	Expert assessment	Development status (e. g., development, inactive, stable)
[API SIZE]	#API types	Number of types listed in the Javadoc
[Vulnerabilities EXTENT]	#Known critical issues	Number of security issues in the bug database
[Extens./conf. capabilities EXTENT]	Expert assessment	Availability and characteristics of extension and configuration features
[Library.System ENTANGLEDNESS]	#API calls #Distinct API methods %Affected classes Scatteredness of API calls	Number of API method calls Number of distinct API methods called Fraction of classes in the system with API method calls Degree of scatteredness of API calls regarding the package structure
[Library.System ADEQUACY]	Expert assessment %API Utilization	How well does the actual use correspond to the intended use Fraction of API methods that are actually used
[Library.Developers FAMILIARITY]	Avg. #years of experience	Developers are interviewed or their CVs are consulted
[License.System COMPATIBILITY]	Expert assessment	Analysis of license terms, e. g., by legal expert

Figure 10.3: Facts and associated metrics of assessment model

10.3 Assessment process

Our assessment process provides guidance to operationalize the model for assessing library usage in a specific software project. When assessing a real-world project, the sheer number of libraries require a possibility to address the *most relevant* libraries first. Therefore, the first step of the process structures and ranks the libraries according to their entangledness with the system. This pre-selection directs the effort of the second step of our process: the expert assessment of the libraries. The last step describes how to generate an assessment report from the aggregated results.

Ranking and pre-selection

In the first step of our assessment process, a set of automatic static analyses are run on the project to be assessed. Their goal is to objectively determine the degree of entangledness between external libraries and the system. This step ensures the applicability of our approach in practice, as extracting these values by hand is unrealistic for real world projects. Furthermore, the extracted metrics help to identify the most significant candidates for the expert assessment and therefore decrease the effort of the system review.

Ranking

To rank the libraries according to their entangledness with the system, we extend work presented in [26] and determine the following values for all libraries: The number of *total method calls* to a library allows to rank all external libraries according to the strength of their direct relations the system. The value is computed for the entire system hierarchy to allow a drill down from system level to class level. This way, the point of impact can be explored precisely. This is relevant for cases, in which participants of the assessment wish to understand in detail where a library affects their system. The number of *distinct method calls* to a library adds information about the implicit entangledness of libraries and system. It allows to understand how difficult a migration could be. The granularity of the computed value is the same as for the total number of method calls. The *scatteredness of method calls* to a library describes whether the usage of the library is concentrated to a specific part of the system, or scattered across it. The value is computed based on the package structure of the system (for further details see Section 10.4). The *percentage of affected classes* gives a complementary overview about the impact a migration could have on the system. This value is independent from the system structure.

The results of these analyses are presented as an HTML document, which provides the possibility to inspect the raw data, as well as detailed insights via the drill-down. The data for each metric is presented in a tabular way with the libraries ordered descendingly according their scores.¹ Figure 10.4(a) shows an excerpt of this representation for our study object, picturing the total and distinct method calls.

¹For further details on the visualization, please refer to [26].

Pre-selection

Under ideal circumstances, all libraries should be assessed in full detail – however, we are aware that in practice the sheer number of libraries and the limited resources will make this unrealistic. Therefore, we propose to use the results of the automated analyses as the basis for a pre-selection of libraries: the union of the first N libraries in each category are candidates for detailed inspection and subjected to an expert assessment².

Expert assessment

Our model guides the expert during the assessment process. The automated analyses have provided the information which can be extracted from the source code. The expert now needs to evaluate the remaining metrics. For this, he or she requires detailed knowledge about the project and its domain. Furthermore, detailed information about the libraries needs to be researched.

In practice, some library characteristics, such as incompatible licenses or security issues might be an exclusion criterium for libraries. If the expert is aware of such criteria, we advise them to assess them for all the libraries of the project. Following the metric evaluation, the expert needs to map the results to the scale *{low, medium, high}*. According to the impact relationships in Figure 10.2, these values map to *{bad, satisfactory, good}*.

The goal of our approach is to assess the impact each library has on the activities in our model. Therefore, for each library we count the number of *{bad, satisfactory, good}* scores it has received for each single activity. In the same way, the overall score of a library is computed.



















Report generation

Subsequent to the assessment, a report can be generated from our model. The structure is the following: a chapter is dedicated to each external library. The expert enters the description of the library and fills in the sections generated from the model categories. At the end of each chapter a table summarizes the assessment for the respective library according to the model in Figure 10.2. At the end of the report, we include an overview which shows the aggregated results for all the libraries in the project. For each library, one row holds the scores per activity in the final table. Also the overall score of the library is reported. Table 10.3 shows an example. By offering the detailed chapters with the assessment of all libraries as well as the aggregated view on the system, we cater to the needs of experienced as well as inexperienced recipients, who may be consultants, project managers, or developers. As it highlights potential areas of concern, the report is supposed to serve as basis for reuse and maintenance decisions.

10.4 Tool support

The assessment model includes five metrics that can be automatically determined by static analyses of the software system to be assessed. Four of these metrics are used in the pre-selection

²The number N of libraries in the assessment can be determined based on the resources available to conduct the assessment.

Library	Modify	Understand	Migrate	Protect	Distribute	Overall
Library	 G:2  S:2  B:1	 G:2  S:1  B:2	 G:1  S:0  B:3	 G:0  S:2  B:0	 G:0  S:0  B:1	 G:5  S:5  B:7

Legend: G: # of *good* impacts, S: # of *satisfactory* impacts, B: # of *bad* impacts

phase of our approach to rank the libraries with regards to their significance to the project. The tool support for the assessment is implemented in Java on top of the open source software quality assessment toolkit ConQAT³, a modular toolkit for creating quality dashboards which integrate the results of multiple quality analyses. The current implementation is targeted at analyzing the library usage of Java systems but could be adapted to other programming languages with a library reuse concept and for which a parser API in Java is available.

The analysis requires the source and byte code of the project as well as the included libraries as input. The output is a set of HTML and data files showing the metric values for each library in a tabular fashion. The analysis traverses the abstract syntax tree (AST) for each class in the project and determines all method calls to external libraries. For each library, it determines the following five metrics (see also Table 10.2):

- Number of API method calls
- Number of distinct API method calls
- Percentage of affected classes
- Scatteredness of the API
- Percentage of API utilization

The number of total and distinct API method calls as well as the percentage of affected classes are aggregated during the AST-traversal. The scatteredness metric requires more computation: it expresses the degree of distribution of API calls over the system structure. API calls within one package are considered as *local*. We would expect local calls for specific functionality, e. g., calls to networking or image rendering libraries. These would be expected to be concentrated to small parts of the system. Contrarily, libraries providing cross cutting functionality such as logging would be expected to be called from a large portion of the system, therefore exhibiting a high scatteredness value. We compute scatteredness as the sum of the distances between all pairs of package nodes in the package tree with calls to a specific API. The distance of two nodes in the package tree is given by the sum of the distance from each node to their least common ancestor. It is important to note that since the scatteredness metric depends on the system structure (i. e., the depth of the package tree) its values cannot be compared in a meaningful way across different software systems. The percentage of API utilization is computed as fraction between the number of distinct API methods called and the total number of API methods in the

³<http://www.conqat.org/>

library. The complete tool support is available as a ConQAT extension and can be downloaded as a self-contained bundle including ConQAT⁴.

10.5 Case study

The goal of the case study is to show the applicability of our approach on a real-world software system. To this end, we performed a case study on a system of azh Abrechnungs- und IT-Dienstleistungszentrum für Heilberufe GmbH, a customer of CQSE GmbH.

Industry partners

The CQSE GmbH was founded in early 2009 as a spin-off of the competence center for Software Quality and Maintenance at the chair for Software & Systems Engineering of the Technische Universität München. It provides consulting services for software quality assurance. In particular, it helps customers in applying novel techniques like clone detection and architecture conformance analysis to ensure long-term maintainability of their software systems.

The azh Abrechnungs- und IT-Dienstleistungszentrum für Heilberufe GmbH is one of the largest provider for billing and IT-services for professional health care providers in Germany. With 550 employees they provide support for 20,000 customers.

Analyzed system

The analyzed system is a distributed billing application with a distinct data entry component running on a J2EE application server which is accessed from around 350 fat clients (based on Java Swing). The system's source code comprises about 3.5 MLOC. The system's files include 87 Java Archive Files (JARs).

Study procedure

We executed our assessment approach (see Section 10.3) on the study object and recorded our observations during the process. We presented our results to the stakeholders in the company and qualitatively captured their feedback. For this we used the following guiding questions:

- Does the report contain the *central* libraries?
- Does the assessment conform to the stakeholders' intuition?
- Are important aspects missing in the assessment?
- Were parts of the assessment result surprising?

⁴<http://www4.in.tum.de/~ccsm/library-usage-assessment/>

10.5.1 Results and observations

The pre-selection step revealed that out of the 87 JAR files included by the project files, the system's source code directly calls methods from 47. The extent of entangledness between the system and these libraries differs significantly, as illustrated in Figure 10.4(b). For some libraries, only one method is called while for others, there are several thousand method calls indicating the difference in importance for the project. Also the degree of scatteredness varies significantly, as shown in Figure 10.4(c).⁵

The pre-selection step to determine the most *important* libraries returned a set of 20 JARs. In two cases we conceptually merged several individual JARs into one logical library as they originated from the same in-house project, which resulted in 10 *logical* libraries for further analysis. We then determined for each library all metrics of our assessment model (see Table 10.2) and evaluated each library. The aggregated result of the library usage assessment for the studied software system is shown in Table 10.5.1.

After a feedback cycle with the CQSE consultant we learned that Spring would have been expected as a central library but was missing in our results. The reason for this is that Spring is a dependency injection framework, in which the framework code mainly calls the user's code and thus the source code does not contain many API method calls. This is a threat to the internal validity of our analysis (see Section 10.5.3). After the consultant's feedback, we added Spring to the list of libraries and performed a detailed assessment for it. The result is highlighted in grey in Table 10.5.1.

The complete assessment process for a single library took around 20-60 minutes. Without pre-selection, even in the best case, the assessment of all 87 included libraries would have required 29 person hours. In contrast, the assessment of the *significant* libraries identified by pre-selection decreased the effort to approximately 5 person hours.

Interpretation

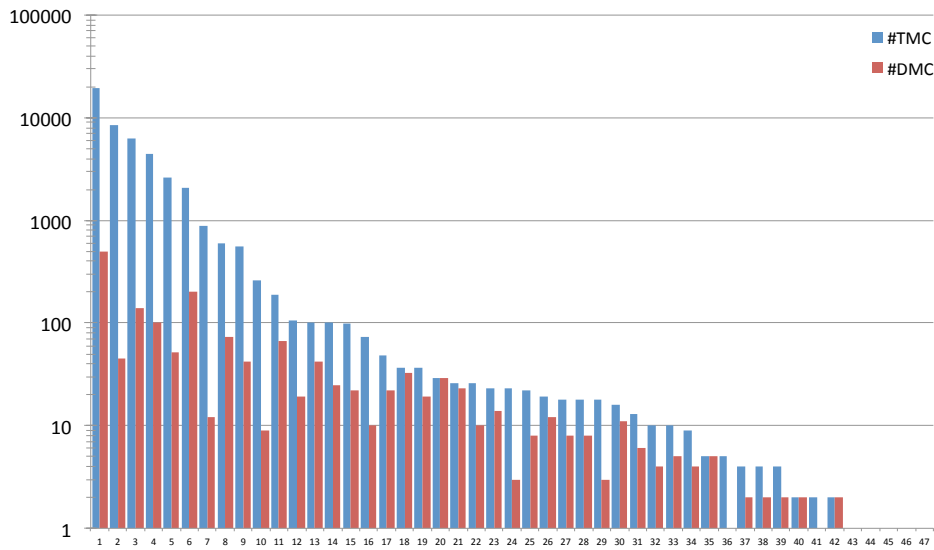
The results in Table 10.5.1 show a mature style of external library usage: most assessed libraries include significantly more positive than satisfactory or negative scores. This is, amongst others, due to a good choice of libraries as no immature or insufficiently supported libraries are included. Another reason for the good results is the high familiarity of the developers with the libraries that helps to overcome potential problems. Also, the support for most libraries is excellent, which positively influences most activities.

The libraries with the best scores are JFormDesigner and Jasper-reports, with 22 and 20 good scores respectively. The libraries introducing most risk are Drools (9×bad), Jai_codec (9×bad) and azh-library1 (10×bad). Depending on the activities, there is a lot of variation concerning the scores of the libraries. The libraries score very heterogeneously for *Modify*, *Understand* and *Migrate*. Contrarily, *Protect* and *Distribute* are well supported. The only exception is Jasper-report as the vendor's commitment to an open-source distribution model is not entirely clear.

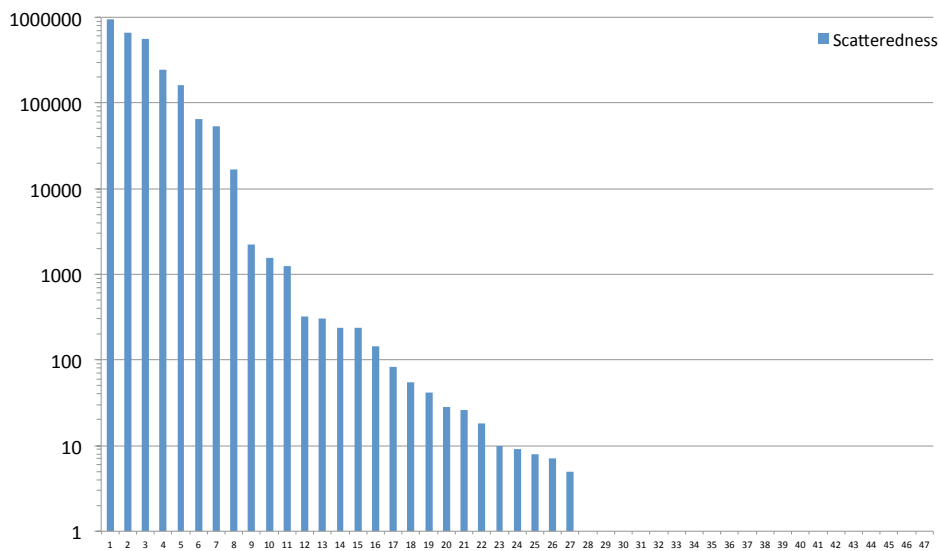
⁵Note that the long tail of libraries with only one method call or scatteredness of 1 or 0 is represented by the blanks in Figures 10.4(b) and 10.4(c), as they are not visible due to the log-scale.

API Dependence Tree (Main)				
API Dependence Tree				
Element	azh-library1-client	log4j	azh-library1-server	
[-]	19381(493)	8567(45)	6246(140)	4425(101)
[+]	19381(493)	8567(45)	6246(140)	4425(101)
[+]		2927(14)	1263(27)	806(51)
[+]	11(6)	376(27)	117(34)	56(24)
[+]		16(4)		50(13)
[+]		77(13)	129(15)	27(8)
[+]		74(6)	26(5)	1(1)
[+]		267(14)	69(11)	232(34)

(a) Excerpt of the automated analysis results, showing the distribution of total and distinct method calls over the system decomposition.



(b) The distribution of total vs. distinct method calls.



(c) The distribution of scatteredness.

Figure 10.4: Result distribution for total and distinct method calls and scatteredness for each JAR file used by the system.



Figure 10.5: Aggregated view of the assessment results

Stakeholder feedback

The consultant of the CQSE reported that he perceives the automated pre-selection process as highly beneficial because it allows a selection of *central* libraries based on quantitative data. As not all libraries can be assessed in a typical audit this selection is essential; however, up to now, it was often based on *educated guesses* and opinions of the architects and developers. The metric-based pre-selection helps to make the selection more objective. This is particularly important as quality audits often stir emotions and, hence, all aspects of the audit must withstand fierce criticism and may not appear to be subjective.

Regarding the questions formulated in Section 10.5, the CQSE consultant replied that the report contains the central libraries with the exception of the Spring framework, as discussed above. The results conform to the consultant's intuition as he did not expect the system to depend on inadequate libraries. However, one aspect missing in the assessment is a peculiarity of the Drools library: Drools is not only a library but also defines its own rule description language. Central parts of the assessed system are implemented in this language. Hence, Drools must be considered more central than *normal* libraries. This is not directly reflected in the assessment. The analysis returned some surprising results w.r.t. to the centrality of the libraries. For example, the imaging library JAI was not considered by the consultant before it was pointed out by the analysis.

The software engineer of azh reported that the most important libraries of the project were selected. In general, the results conformed to the developer's intuition. However, in some cases more details of the assessment would be helpful. According to the engineer, all important issues were evaluated in our model. In addition, explicit statements about the future viability of the libraries or alternatives to the previously used libraries would be interesting. Parts of the assessment result were surprising, in particular that the libraries JFormDesigner and azh-library2 were among the most central libraries. Overall, the positive outcome of the assessment corresponded to the engineer's perception of the state of library usage on the project.

10.5.2 Discussion

The system analyzed in the case study employs a considerable number of software libraries. While some of them play a central role and thus have a significant impact on the maintenance and further evolution, others are of lesser importance. Our ranking step in the assessment process was able to identify these *important* libraries, allowing to focus further investigation on their usage adequacy. Supported by the assessment model, we identified a number of metrics for each of the identified libraries leading to a comprehensive usage assessment per library. The assessment shows which development activities are influenced and thus allows to identify problematic library usage at a glance. However, proposing *one-size-fits-all* solutions for addressing the issues uncovered by the analysis does not seem adequate; the possible actions are highly dependent on the project context and the maintenance strategies. Our findings rather serve the purpose to create awareness and provide the information for decision making.

An interesting question is what is considered as an *external* library. For instance a library produced by a different department within the same company may be either considered internal or external. Another central issue is what entity is considered a library. The technical structuring imposed by JAR files may not map to what is considered a logical reusable library. More relevant are factors like provider and release cycle. For an assessment with the proposed model, this has to be defined depending on the specific context in a project. During the discussion of the results, it became apparent that weights for facts would be useful to tailor the model to a specific project context.

10.5.3 Threats to validity

Internal validity

The assessment model was created based on CQSE's experience with software quality assessment and software development. We do not know how complete the model is in terms of aspects influencing library usage. However, due to the experience from CQSE GmbH in assessing technology usage in diverse and large industrial projects we are confident that we covered the most central influence factors.

The ranking of the libraries regarding their significance for the including project is based on static analysis metrics taking into account method calls to external libraries. This means that libraries that are indirectly used via other libraries are not considered in the assessment. However, the automated analysis produces a dependency graph of all JAR files including transitively referenced JARs. This allows an assessor to manually identify additional central libraries for detailed assessment.

In addition, reuse can also occur by other means such as subclassing which is typically used in frameworks using a dependency injection mechanism (e.g., Spring). Moreover the analysis cannot detect method calls via the Java Reflection mechanism. In the future, parts of these limitations could be addressed with further static and dynamic analyses.

External validity

Our case study was restricted to a single commercial software system written in Java. We do not know how our findings transfer to software built on other programming ecosystems besides Java. Especially w.r.t. the availability of third-party libraries, we expect major differences.

We are convinced that both the assessment model and the assessment approach have a good applicability to other programming platforms for which a rich variety of reusable libraries exists. More extensive case studies are required to provide evidence this for hypothesis.

10.6 Related work

We relate our approach to the fields of analysis of third-party library usage, architecture analysis and software quality assessment.

Analysis of third-party library usage

In [26], we proposed an approach to determine the degree of dependence between a software project and its third-party libraries in order to support decision making in various use cases during software maintenance. The focus of this previous work was to quantify library reuse while in this work we present an assessment model which defines what constitutes *adequate* reuse with respect to third-party libraries.

Klatt et al. [36] suggested an approach to identify the impact of evolving third-party components on long-living software systems. They use a white-box impact analysis which requires access to the third-party source code and combined it with data from bug trackers and quality analyses on the third-party code. In contrast to our approach, the authors do not provide an explicit model defining the impacts of library usage characteristics to development activities.

Kotonya and Hutchinson [186] suggested an approach that helps developers understanding the impact of change in commercial off-the-shelf (COTS) software components employed in a project. Contrarily to our approach, they rely on a COTS component-oriented development process and focus on the more specific use case of change impact analysis.

Raemaekers et al. [54] proposed an approach to automatically assess the risk imposed by third-party library usage in software projects. They measure the usage frequency of third-party libraries in a corpus of open source and commercial software systems. The risk assessment is based on the assumption that an *uncommon*, i. e., infrequently used, libraries expose a higher risk compared to a library that is frequently employed by software projects. In contrast to our approach, the authors have a very general heuristic for assessing the risk of library usage. We provide a comprehensive model taking multiple factors of libraries and their usage into account.

Lämmel et al. [31] analysed API usage in 1,476 open source Java projects. They determined the *API usage footprint* of the projects, in terms of the number of included libraries and the number of (distinct) API methods called from the projects' code. Contrarily to our approach, the authors focus on the extent of API usage and do not take into account other characteristics of the library and its usage.

Architecture analysis

Architecture analysis approaches aim at evaluating a software system with regards to its internal structure.

The *software architecture analysis method (SAAM)* proposed by Kazman et al. [187] is an approach for a scenario-based evaluation of software architectures. The method involves describing activities that have to be supported by the software system, prioritizing them, and assessing how well the architecture facilitates them.

The *architecture tradeoff analysis method (ATAM)* [188] is an approach for evaluating a system's architecture with respect to competing quality characteristics (e. g., modifiability vs. performance). The goal is to mitigate risks of architectural decisions, ideally early in the development cycle. Potential architectural alternatives are analyzed and a risk mitigation is used to drive refinements of the architectures.

Thus, architecture analysis approaches offer a general framework for the evaluation of principal architectural decisions. In contrast, the proposed library usage assessment approach provides a detailed model of the influence of library usage on maintenance activities.

Software quality assessment

Software quality assessment methods supported by an explicit quality model, such as Quamoco [189] or Squalo [190], use the same principal approach as our method. Attributes of the system and their impacts on quality characteristics are explicitly modeled according to a well-defined meta-model. The model is operationalized in the automated assessment which analyzes a concrete software system with regards to the modeled quality attributes. However, these models do not take into account aspects originating from third-party library usage. In contrast, our approach is specifically targeted at these aspects and is thus complimentary to these approaches.

10.7 Summary and future work

Despite their pivotal role in modern software development and their impact on maintenance, third-party libraries tend to be overlooked in quality assessments of software systems. We presented a structured approach to assess the adequacy of library usage in software projects. Based on industrial experience, we provided a lightweight assessment model as well as an assessment process, including tool support and guidance for pre-selecting candidates for inspection. We reported results of a case study applying our approach to an industrial system. The results indicate that our approach gives a comprehensive overview on the external library usage of the analyzed system. It outlines which maintenance activities are supported to which degree by the employed libraries. Furthermore, the semi-automated pre-selection allowed for a significant reduction of the time required by the expert assessment.

Based on the results of the case study, we consider this analysis as adequate to cover third-party library reuse during a reuse assessment with RASM. In addition, an iterative application provides insights into the effectiveness of the selected reuse strategy with respect to library reuse.

Future work

Currently an impact between an attribute and an activity is either positive or negative. As a future extension, we plan to extend our meta-model with weights for impacts. This allows for a more fine-grained modeling of the individual impact relationships. Moreover, we want to include custom aggregation functions to allow for a custom weighting in order to emphasize the impact of individual attributes regarding the overall assessment result. A further goal is to increase the automation of the current assessment.

Library reuse rates over time To further benefit from the approach presented in this chapter, we envision the following extension: the analyses could be applied to the version history of the studied systems. In this way, reuse trends could be monitored and linked to further quality

indicators or external events. This would lead to detailed insights on the interplay of reuse with the given environment:

First, this extended analysis would enable practitioners to monitor the acceptance and effects of a change in their reuse strategy on the level of library reuse. Changes, such as the removal or proliferation of a specific technology, as well as the emergence of new usage patterns could be assessed on a fine grained scale. The results of this more detailed information could support a more thorough analysis of reuse issues: anomalies, as well as significant changes in reuse patterns could be linked to a specific point in time and, thus, serve as input for a root cause analysis. Identifying potential root causes, however, could be a first step towards understanding systemic aspects of undesired reuse; an information highly relevant to shape the future reuse strategy.

A second aspect of a more detailed understanding of usage changes over time regards the budgeting for reusable entities: often, in practice, providing reusables is considered a cost and quantifying the obtained benefits remains challenging. Capturing the proliferation (e.g., in a scenario aiming to replace several copies of code with a reusable library) of a technology can provide a more solid understanding of the long term benefits, e.g., with respect to maintainability. On this basis, the approach could provide a better estimate of the value of a particular reusable. Consequentially, an organization could adjust the compensation for creating and maintaining the given artefact.

Last, a continuous monitoring of library use could support project managers to identify future problems that one or more projects are accumulating. One example of this is migration debt that can accumulate to an unmanageable extent [191].

Assessing internal library usage In a commercial software development setting, organizations frequently create proprietary libraries that, arguably, share many of the risk factors mentioned in this Chapter. Usually, these libraries are created by dedicated company-internal third parties. Consequently, applying the approach to internal software libraries could yield the same benefits as for the Open Source libraries mentioned above. Given a suitable development infrastructure and deployment mechanisms within a given organization, we expect a smooth transfer of the approach to internal libraries.

Part V

Conclusion

11 | Summary and conclusions

Software reuse is an intriguing idea, promising substantial benefits for adopters. This leads to high interest of organizations in effecting reuse in one way or another. However, certain low-entrance types of reuse (e.g., clone and own) tend to lead to increased development velocity at the cost of causing significant overhead in maintenance. Organizations, therefore, might aim to introduce more advanced reuse approaches, e.g. Inner Source, or Software Product Lines.

Unfortunately, these approaches demand a significant level of skills and context factors from an organization. These factors are not always obvious from research work which usually focusses more on specific aspects of a technology than on a full picture for adopters. Furthermore, the hurdle of adoption is different for each organization, depending on the given context and capabilities.

As a result, many organizations struggle to identify and adopt their desired reuse approach. In particular, understanding the details and gauging the effort required for the adoption, e.g., in terms of organizational change, time and resources, is challenging. This leads to unrealistic expectations in terms of return on investment, as well as the time to benefits. As a consequence, resources tend to be short, management support wavers, and the adoption initiative fails.

From the research side, little support is offered so far to prevent this: research on reuse is fragmented, guidelines for adoption focus on single aspects of approaches and technologies, and context requirements often remain implicit. Consequently, gaining a deeper understanding of proper reuse approaches suitable for adoption in the particular context poses a significant challenge for practitioners.

As stated in Section 1.3, our work aimed to counter these issues and support practitioners to explore adequate options with respect to software reuse and, as a result, to derive realistic expectations and plans for adoption.

11.1 Summary of the contributions

In this work, we addressed our goal from three different angles, providing empirical evidence of current software reuse practice, a constructive support model for reuse adoption, as well as tool support.

In a first step (Contribution 1), we aimed for a deep understanding of the state of reuse in current software development practice. To obtain this knowledge, we conducted two detailed case

studies at two large software houses, analysing current reuse practices, supporting and inhibiting context factors, benefits and drawbacks. In addition, we integrated the results of the two cases in a formal synthesis.

The synthesis highlights that reuse in practice occurs in many different flavours, however, mostly limited to source code. Partially, the technological potential present in elaborate development infrastructures and tool support has been embraced: as a consequence, once infeasible approaches, such as repositories as source for reusable entities, can now be operationalized in a more beneficial way. Generally, successful reuse was tightly coupled to the company goals and compatible with in the development culture. Establishing systematic reuse in heterogeneous contexts posed significant challenges. As a result of these studies, the remainder of this work is grounded on the needs of practitioners. Thus, we ensured the relevance of the subsequent contributions.

The second contribution constructively addresses the challenge of selecting an adequate reuse approach for adoption in practice by proposing the Reuse Adoption Support Model (RASM). Based on literature and the results of the first contribution, RASM incorporates a classification scheme that is able to capture a range of diverse reuse approaches and to relate their requirements to the characteristics of an adopting organization. In this way, RASM supports a structured discussion of reuse approach alternatives, ensuring at the same time that important details are considered. As a result, adopting organizations are guided through the complexity of reuse approaches, discover the aspects that are relevant and critical in their context, and obtain an objective assessment on the overall feasibility as well as the detailed kind of efforts required for a successful adoption. A proof-of-concept evaluation at one company confirms the assumed benefits of applying RASM in practice.

At this point in time, the application of the model requires a substantial manual effort. However, some of the facets of reuse can be supported by automation, e.g., assessing the prevalence of copied code, estimating redundant implementations, measuring the use of libraries, and benchmarking the maintenance quality of software. Therefore, as third contribution, we proposed supporting methods and tools to support data collection for the RASM. In particular, we focus on the identification of missed reuse opportunities, quantifying the potential for reuse in source code, and the chances and risks presented by the third-party libraries that are incorporated in a given system. The tool support can enhance the application of RASM in two ways: when applied during the assessment of the company profile, it provides information on the current state of code reuse as well as on potential candidates for reusable artefacts. After the model application and during the adoption of a reuse approach, the tooling can provide insights into the effects of the approach on the source code level: by repeated analysis, changes, e.g., of the use of *clone-and-own* reuse or the proliferation of shared reusables, become visible and can be matched against the expected outcomes.

Academic key findings From an academic perspective, the key contributions on the empirical side are, on the one hand, current evidence on the state of the practice with respect to software reuse, and, on the other hand, a systematic method for synthesizing the outcomes of methodologically differing case studies. On the methodological side, we provided a model suited for

accommodating a range of reuse approaches and systematically relating them to the context of a particular company, thus providing a vehicle for increased technology transfer from academia to practice.

Practice key findings From a practice-oriented perspective, the key contributions of this dissertation are, first, current evidence on otherwise inaccessible insights on reuse from other organizations. This counteracts the prevalent folklore that tends to surround software reuse in practice. Second, with RASM, this dissertation presents a solution to the challenging problem of selecting an adequate reuse approach for a given organization. The model supports the comparison of multiple approaches, a detailed matching of approaches and organization based on the organization's goals and capabilities, a general placement of organizations in an extensible benchmark of software reuse capabilities, as well as a light-weight method for comparing alternative approaches based on expected adoption effort and obtained benefits. Last, method and tool support enables for a partial automation of the data collection for applying the model and monitoring the transition to the new approach.

11.2 Outlook

With RASM, this dissertation has provided a framework for building up a software reuse approach knowledge base that supports practitioners with structured guidance for assessing their reuse adoption plans. It has contributed an instantiation of RASM for *Inner Source* as well as initial processes for applying the model in practice, with promising results. Nevertheless, further aspects need to be researched to ensure more instances of reuse success in practice.

Extension of RASM To reach its full potential, a substantial amount of work needs to follow to extend RASM: first, the characterization and classification of further reuse approaches needs to continue to build up a data base that can support matching organizations with their optimal reuse approaches. Furthermore, the model application process needs to be refined. In this context, the accessibility of the model instantiations could be improved to allow for an automated evaluation of the application results.

Second, from a conceptual angle, the dependencies and impacts between the reuse facet constituents should be studied further: a clear understanding of their relation to the envisioned benefits would facilitate a more fine-granular planning of the adoption. In addition, milestones with clear benefits could be targeted, which would allow for earlier realizations of benefits.

Third, from an empirical point of view, we aim to study in greater detail the effects of applying RASM in practice. In particular, we aim to see how well RASM is suited to support practitioners continually during the adoption and improvement phase of their reuse initiative. To this end, we aim to apply RASM in different contexts and to synthesise the respective findings.

Future challenges in software reuse Selecting an adequate reuse approach is the first step of an organization towards success with software reuse. However, a number of challenges remain that need to be addressed during adoption and everyday reuse practice, some of them transcending the borders of software engineering with other fields:

First, despite of significant improvements, technical challenges persist. As we have seen, e.g., in Chapter 4, the presence of a state-of-the-art development infrastructure can not alleviate all issues related to finding and identifying candidates for software reuse: the options available in a large code base are overwhelming and identifying the best one can appear infeasible, inciting rewriting code from scratch. Additional work is needed to address this challenge.

Second, software reuse (and, in particular, its adoption) depends as much on social and organizational matters as it does on technical ones (and, frequently, reuse initiatives fail due to these “soft” factors, see, e.g., Chapter 5). However, these aspects tend to be out of the expertise of trained computer scientists. Consequently, the aspects of organizational change, motivation, incentives, and hindrances should be studied from an interdisciplinary angle, integrating relevant research from the respective fields with the knowledge of software engineering.

Last, reuse adoption processes can take several years to be completed. As, over this time, they can be influenced by a large number of factors, it is challenging to present rigorous results on the effects of single interventions. All the more, studies on software reuse in practice should ensure to carefully report the context of their cases to enable future research to integrate the pieces of evidence into meaningful insights.

11.3 Conclusions

Despite continuous advances, software reuse is not yet a problem solved in practice. Due to its multifaceted nature, organizations aiming to perform it successfully need to take into account numerous complex details. During reuse adoption, the available options need to be understood in detail to ensure an informed decision and allow for realistic estimates in terms of time, resources, and effort. These aspects are tightly related to the particular context of an organization and, thus, an approach that provides a detailed assessment of costs and benefits to be expected, as well as suggestions for a step by step improvement, is of significant value.

Part VI

Appendix

12 | Appendix

Contents

12.1 Appendix for Chapters 4, 5, and 6 — Analyzing and comparing reuse practices	200
12.2 Appendix for Chapter 8 — Instantiation of RASM for Inner Source	206
12.3 Appendix for Chapter 10 — Rationales for impacts of the library usage assessment model	217

12.1 Appendix for Chapters 4, 5, and 6 — Analyzing and comparing reuse practices

The questionnaire material of the case studies is included in [41]. Here, we include the interview guide used for both studies and the supplementary tables containing the scale aggregations and response data for Chapter 6.

12.1.1 Interview guide

Table 12.1 presents an overview of the interview topics as well as sample questions from the interview guide.

12.1.2 Scale aggregations

Table 12.2 provides the details of how the aggregation of the Likert scales was computed for each question. The column *Question ID* refers to the ID of the questions. *Scale U* encodes the type of the Likert scale for the respective question in the questionnaire of U. L4 encodes a four point scale; L5 encodes a five point scale. Together with the scale code, we report the two boundary values of each scale. The column *aggregation* shows how the aggregation of the respective scale was computed: P1 to P4 (or P5, where applicable) encode the possible options that could be selected. Their sum represents the aggregation of the number of participants that selected the respective options. For instance, for the question FAR1, we aggregated the number of participants that selected option P1 or P2 in code Low (L) and the number of participants that selected option P3 or P4 in code High (H). Next to the aggregation, we report the semantic value of the aggregated value.

12.1.3 Result of comparison RQ 1 and RQ 2

Tables 12.3 and 12.4 report the results of the comparison for RQs 1 and 2. They contain the following information:

Question ID refers to the code of the question in the questionnaire (for the respective questions see Tables 6.1 and 6.2).

Response options lists the possible answers.

Answ low/high represent the count of participants that selected the low/high end of the Likert-scale (case D) or marked a selection (case G) for the respective item, according to the analysis methodology described in Section 6.4. This data is reported to facilitate replication of the analysis by third parties.

Pval chi.square reports the p-value of the χ^2 test.

Verdict expresses if the vote for the item tended to the low or the high category. Blank fields in the tables represent information that was missing in the respective questionnaire.

Table 12.1: Interview guide: topics and sample questions for cases G and U

<p>Interview topics</p> <p>Economic, social, conceptual, and technical aspects of reuse</p> <p>What are current goals of reuse? Where do you see potential?</p> <p>Are there current issues? If yes, which?</p> <p>Is there need for support? If yes, which? (Tools, processes, SE practices, ...)</p> <hr/> <p>Reuse assessment</p> <p>When do you consider reuse as successful? Generally (fulfilling company goals)? From your specific perspective?</p> <p>How do you assess the success of reuse, the fulfilment of reuse goals?</p> <p>How do you decide on how reuse should be done?</p> <p>How do you proceed to implement the selected way of reuse? Steps? Support?</p> <p>In which phases do you expect/target reuse benefits? In which form should benefits occur?</p> <p>Which business goals do you aim to support with (internal/external) reuse?</p> <p>What are the current product goals and requirements?</p> <p>In which way is reuse currently effected?</p> <p>What kind of reuse do you aim for?</p> <p>What are preconditions/requirements/challenges on the process/conceptual/technical/organisational/communication levels?</p> <hr/> <p>Experiences with current reuse</p> <p>What works currently in terms of reuse? Why?</p> <p>What did not work wrt. reuse so far? Why?</p> <p>What were the biggest mistakes committed w.r.t. reuse?</p> <p>How can these mistakes be mitigated/corrected?</p> <hr/> <p>Planning and conflicts of interest</p> <p>Reuse as a source of conflict within company, local vs. global optimization</p> <p>How is reuse planning done locally (department/team/group) and globally (company-wide)?</p> <p>Balance of resources between products and basis?</p> <p>What is harder: providing or maintaining reusable entities?</p> <p>What do you consider essential for a professional stance wrt. reuse?</p> <p>How do you address reuse and evolution of entities?</p> <p>How important is tool support? For which parts of the process?</p> <hr/> <p>Product line adoption</p> <p>Starting points, goals, trigger for decision</p> <p>Process, issues and challenges: on which level? How addressed?</p> <p>Successful? How could success be validated?</p> <p>Which methods/strategies were effective? What would you do differently next time?</p> <hr/> <p>Product line evolution</p> <p>Challenges? Key points? Success criteria and factors?</p> <p>What (in terms of content) should be in the platform? In initial phase? In further evolution?</p> <p>How do you proceed to coordinate the different stakeholders? Requirements engineering? Sources of requirements for platform? Process? How are requirements persisted? How is a decision reached?</p> <p>Reference architectures: relevant? present? in which shape? Are deviations acceptable?</p> <p>What needs to happen to achieve the acceptance and use of an internal framework? How is knowledge transferred?</p> <p>How do you proceed with a common platform? What about governance, guarantees, compensations? Resources?</p> <hr/> <p>Development context</p> <p>How important is homogeneity of process, tool infrastructure, quality assurance, goals?</p> <hr/> <p>Reuse from external sources</p> <p>How important is the provenance of reused code? Security? Certification? Accountability, liability? Type of usage?</p> <p>How do you procure/inspect/maintain (clone-and-own/external/central)? Who is responsible, has an overview, assesses external entities and their usage? Are there rules/limitations for the use of external entities?</p> <hr/> <p>Improvements</p> <p>What is your most important wish for improvement wrt. reuse?</p>
--

Table 12.2: Scale aggregations for questionnaire U. *Question ID* refers to the respective question, *Scale U* relates the type of the scale (*L4* for a 4 point Likert scale, *L5* for a 5 point Likert scale) and reports the extreme values of the given scale. *Aggregation* illustrates how, for the given scale, the values were aggregated in the categories *Low* and *High*. $P<n>$ denotes the number of responses at the given point of the Likert scale.

Question ID	Scale U	Aggregation	
		Low	High
Extent of code reuse (ECR)			
ECR2	L5, no use, always use	P1: no use	P2+P3+P4+P5: use
ECR3	L5, no use, always use	P1: no use	P2+P3+P4+P5: use
Finding artefacts (FAR)			
FAR1	L4, never, always	P1+P2: irrelevant, low usage	P3+P4: regular, high usage
Reused artefacts (RAF)			
RAF1	L4, never, always	P1+P2: irrelevant, low usage	P3+P4: regular, high usage
RAF2	L4, never, always	P1+P2: low usage	P3+P4: high usage
Technical realization of reuse (TRR)			
TRR1	L4, does not apply, strongly applies	P1+P2: irrelevant, low usage	P3+P4: regular, high usage
TRR2	L4, does not apply, strongly applies	P1+P2: irrelevant, low frequency	P3+P4: regular, high frequency
Challenges, effects, and context factors of reuse (CHR)			
CHR1	L4, never, always	P1+P2: never, occasionally	P3+P4: regularly, always
CHR2	L4, never, always	P1+P2: never, occasionally	P3+P4: regularly, always
CHR3	L4, never, always	P1+P2: never, occasionally	P3+P4: regularly, always
Success factors and benefits (SFB)			
SFB1	L4, never, always	P1+P2: never, occasionally	P3+P4: regularly, always
SFB3	L4, unimportant, important	P1+P2: unimportant, slightly important	P3+P4: important, very important
SFB4	free text	—	—
Reuse in everyday development practice (RED)			
RED1	L4, does not apply, strongly applies	P1+P2: does not apply, applies slightly	P3+P4: applies, strongly applies
RED4	L4, does not apply, strongly applies	P1+P2: does not apply, applies slightly	P3+P4: applies, strongly applies
Finding artefacts (FAR)			
FAR3	L4, does not apply, strongly applies	P1+P2: does not apply, applies slightly	P3+P4: applies, strongly applies

Table 12.3: Responses and values relevant for RQ1

Question ID	Response options	U answ low	U answ high	U pval chi.square	U verdict	G answ low	G answ high	G pval chi.square	G verdict
1	ECR2 Serialization (e.g. XML).	14	38	<0.001	HIGH	17	19	0.74	-
2	ECR2 Networking.	19	35	0.03	HIGH	19	17	0.74	-
3	ECR2 Persistency.	16	37	<0.001	HIGH	19	17	0.74	-
4	ECR2 Visualization/GUI.	11	44	<0.001	HIGH	24	12	0.05	LOW
5	ECR2 Architecture (e.g. rich client, plugin).	16	38	<0.001	HIGH	21	15	0.32	-
6	ECR2 Algorithms	18	38	0.01	HIGH	NA	NA	NA	NA
7	ECR2 User Interfaces	17	38	0.01	HIGH	NA	NA	NA	NA
8	ECR2 General utility.	NA	NA	NA	NA	12	24	0.05	HIGH
9	ECR3 Domain-independent general functionality.	9	46	<0.001	HIGH	8	27	<0.001	HIGH
10	ECR3 Domain-specific functionality.	24	33	0.23	-	17	18	0.87	-
11	ECR3 Product-specific functionality.	23	33	0.18	-	26	9	<0.001	LOW
12	FAR1 Web search.	27	32	0.52	-	20	19	0.87	-
13	FAR1 Browsing repositories.	46	12	<0.001	LOW	23	16	0.26	-
14	FAR1 Communicating with colleagues.	17	43	<0.001	HIGH	14	25	0.08	-
15	FAR1 Code search tools.	52	6	<0.001	LOW	9	30	<0.001	HIGH
16	FAR1 Code recommenders.	58	1	<0.001	LOW	38	1	<0.001	LOW
17	FAR1 Browsing documentation.	50	11	<0.001	LOW	30	9	<0.001	LOW
18	FAR1 Tutorials.	50	8	<0.001	LOW	38	1	<0.001	LOW
19	FAR1 Other	2	0	0.16	-	36	3	<0.001	LOW
20	FAR1 Code completion.	NA	NA	NA	NA	37	2	<0.001	LOW
21	RAF1 System tests	44	14	<0.001	LOW	NA	NA	NA	NA
22	RAF1 Unit-Tests	41	18	<0.001	LOW	NA	NA	NA	NA
23	RAF1 Personas	50	7	<0.001	LOW	NA	NA	NA	NA
24	RAF1 Code in binary form	36	18	0.01	LOW	26	12	0.02	LOW
25	RAF1 Source code	30	29	0.90	-	1	37	<0.001	HIGH
26	RAF1 Informal design models (Box and lines, natural language)	42	12	<0.001	LOW	36	2	<0.001	LOW
27	RAF1 Semiformal design models (UML)	52	4	<0.001	LOW	38	0	<0.001	LOW
28	RAF1 Formal design models	45	9	<0.001	LOW	38	0	<0.001	LOW
29	RAF1 Own, domain specific design models	44	13	<0.001	LOW	36	2	<0.001	LOW
30	RAF1 Requirement documentation / Use cases	44	16	<0.001	LOW	33	5	<0.001	LOW
31	RAF1 Architecture documentation	45	11	<0.001	LOW	33	5	<0.001	LOW
32	RAF1 Prototypes	51	8	<0.001	LOW	36	2	<0.001	LOW
33	RAF1 UI Designs	38	21	0.03	LOW	28	10	<0.001	LOW
34	RAF1 Style guides	28	30	0.79	-	27	11	0.01	LOW
35	RAF1 Other	2	0	0.16	-	38	0	<0.001	LOW
36	RAF2 Developer Portals.	39	22	0.03	LOW	25	14	0.08	-
37	RAF2 Internal repositories.	43	17	<0.001	LOW	5	34	<0.001	HIGH
38	RAF2 Commercial repositories	51	7	<0.001	LOW	NA	NA	NA	NA
39	RAF2 Colleagues own department.	26	36	0.20	-	NA	NA	NA	NA
40	RAF2 Colleagues.	42	19	<0.001	LOW	24	15	0.15	-
41	RAF2 Open Source Repositories.	50	11	<0.001	LOW	25	14	0.08	-
42	TRR1 Code scavenging (copy, paste, modify).	47	10	<0.001	LOW	24	12	0.05	LOW
43	TRR1 Software libraries.	37	21	0.04	LOW	4	32	<0.001	HIGH
44	TRR1 Software frameworks.	36	19	0.02	LOW	17	19	0.74	-
45	TRR1 Component-based development.	38	18	0.01	LOW	28	8	<0.001	LOW
46	TRR1 Design patterns.	48	9	<0.001	LOW	23	13	0.10	-
47	TRR1 Architecture reuse.	47	9	<0.001	LOW	31	5	<0.001	LOW
48	TRR1 Product lines.	51	3	<0.001	LOW	35	1	<0.001	LOW
49	TRR1 Application generators.	51	3	<0.001	LOW	35	1	<0.001	LOW
50	TRR1 Code generators	50	6	<0.001	LOW	NA	NA	NA	NA
51	TRR1 None.	NA	NA	NA	NA	36	0	<0.001	LOW
52	TRR2 small code sections.	28	27	0.89	-	29	8	<0.001	LOW
53	TRR2 fine-grained, such as single methods/functions.	24	29	0.49	-	26	11	0.01	LOW
54	TRR2 one or more classes.	27	26	0.89	-	19	18	0.87	-
55	TRR2 complete libraries.	27	28	0.89	-	6	31	<0.001	HIGH
56	TRR2 coarse-grained, such as entire frameworks.	34	21	0.08	-	24	13	0.07	-

Table 12.4: This table contains the responses relevant for RQ2

Question ID	Response options	U				G			
		answ low	answ high	pval chi.square	verdict	answ low	answ high	pval chi.square	verdict
57	CHR1 "Not invented here" phenomenon.	39	18	0.01	LOW	21	11	0.08	-
58	CHR1 Licensing/legal issues.	49	12	<0.001	LOW	18	14	0.48	-
59	CHR1 Difficulty of adapting artefact to project needs.	33	29	0.61	-	15	17	0.72	-
60	CHR1 Inconvenient granularity of reusable artefacts.	41	21	0.01	LOW	25	7	<0.001	LOW
61	CHR1 Process for clearance of external artefacts is too slow.	33	25	0.29	-	<i>Interview data</i>			
62	CHR1 Coordination effort with other departments.	30	32	0.80	-	<i>Interview data</i>			
63	CHR1 Other.	3	6	0.32	-	27	5	<0.001	LOW
64	CHR1 Finding the right artefacts is difficult.	<i>See FAR3 and RED1.</i>				14	18	0.48	-
65	CHR1 Accessing the artefact is difficult.	<i>See FAR3 and RED1.</i>				30	2	<0.001	LOW
66	CHR2 Loss of control.	46	14	<0.001	LOW	22	9	0.02	LOW
67	CHR2 Dependency explosion.	35	27	0.31	-	15	16	0.86	-
68	CHR2 Performance decay.	43	18	<0.001	LOW	27	4	<0.001	LOW
69	CHR2 Decrease of code understandability.	50	10	<0.001	LOW	20	11	0.11	-
70	CHR2 Ripple effects caused by changes in reused artefacts.	44	17	<0.001	LOW	19	12	0.21	-
71	CHR2 Code becomes unchangeable.	44	17	<0.001	LOW	28	3	<0.001	LOW
72	CHR2 Excessive restriction of the solution space.	47	13	<0.001	LOW	<i>Interview data</i>			
73	CHR2 No.	NA	NA	NA	NA	23	8	0.01	LOW
74	CHR2 Other.	3	2	0.66	-	30	1	<0.001	LOW
75	CHR3 Inconsistencies.	32	29	0.70	-	17	16	0.86	-
76	CHR3 High maintenance effort.	30	31	0.90	-	18	15	0.60	-
77	CHR3 Increased development effort.	34	27	0.37	-	12	21	0.12	-
78	CHR3 High testing load.	29	32	0.70	-	23	10	0.02	LOW
79	CHR3 Lower code quality.	43	16	<0.001	LOW				
80	CHR3 Duplicate implementations.	<i>Interview data</i>				9	24	0.01	HIGH
81	FAR3 I can readily read the source code available within the company.	45	14	<0.001	LOW	<i>Interview data</i>			
82	FAR3 I can effect required changes independently.	44	14	<0.001	LOW	<i>Interview data</i>			
83	FAR3 The integration of existing code requires little effort from my side.	42	14	<0.001	LOW	<i>Interview data</i>			
84	FAR3 The original developer of reused code is responsible for maintaining it.	14	44	<0.001	HIGH	<i>Interview data</i>			
85	RED1 The quality of artefacts is acceptable for reuse.	34	28	0.45	-	<i>Interview data</i>			
86	RED1 Reusable assets are classified in a comprehensive way.	55	6	<0.001	LOW	<i>Interview data</i>			
87	RED1 In everyday work I attempt to reuse artefacts.	5	58	<0.001	HIGH	<i>Interview data</i>			
88	RED1 When I want to reuse an artefact, it already exists in the company.	48	14	<0.001	LOW	<i>Interview data</i>			
89	RED1 Reusable assets are accessible with acceptable effort.	44	19	<0.001	LOW	<i>Interview data</i>			
90	RED1 Existing artefacts are found easily.	54	9	<0.001	LOW	<i>Interview data</i>			
91	RED1 Existing artefacts are understandable.	39	22	0.03	LOW	<i>Interview data</i>			
92	RED1 Existing artefacts match the required functionality.	42	21	0.01	LOW	<i>Interview data</i>			
93	RED1 Functionally matching artefacts can be integrated with ease.	42	21	0.01	LOW	<i>Interview data</i>			
94	RED4 Reuse is the responsibility of individual developers.	11	51	<0.001	HIGH	<i>Interview data</i>			
95	RED4 Planning for reuse happens in a grassroots fashion.	20	42	0.01	HIGH	<i>Interview data</i>			
96	RED4 Reuse is initiated and coordinated by a small group of people.	34	28	0.45	-	<i>Interview data</i>			
97	RED4 Providing reusable assets is a shared initiative.	46	18	<0.001	LOW	<i>Interview data</i>			
98	RED4 Dedicated personal is assigned to provide reusable assets.	41	23	0.02	LOW	<i>Interview data</i>			
99	RED4 Development and provision of reusable artefacts is coordinated across departments and departments.	51	13	<0.001	LOW	<i>Interview data</i>			

Table 12.6: This table contains the responses relevant for RQ2 — continued from previous page

Question ID	Response options	U answ low	U answ high	U pval chi.square	U verdict	G answ low	G answ high	G pval chi.square	G verdict
100	SFB1 Less maintenance effort.	33	26	0.36	-	10	22	0.03	HIGH
101	SFB1 Higher consistency.	31	29	0.80	-	20	12	0.16	-
102	SFB1 New functionality is made available.	38	22	0.04	LOW	19	13	0.29	-
103	SFB1 Higher code quality.	37	23	0.07	-	17	15	0.72	-
104	SFB1 Higher development pace.	36	24	0.12	-	3	29	<0.001	HIGH
105	SFB1 Regular bug fixes.	NA	NA	NA	NA	21	11	0.08	-
106	SFB1 None.	NA	NA	NA	NA	32	0	<0.001	LOW
107	SFB1 Other.	2	2	1	-	32	0	<0.001	LOW
108	SFB3 Suitable abstractions.	14	46	<0.001	HIGH			<i>see SBF4.</i>	
109	SFB3 Direct communication culture.	9	51	<0.001	HIGH			<i>see SBF4.</i>	
110	SFB3 Suitable incentives.	19	41	0.01	HIGH			<i>see SBF4.</i>	
111	SFB3 Higher quality of artefacts.	5	57	<0.001	HIGH			<i>see SBF4.</i>	
112	SFB3 Well-defined process for reuse.	14	47	<0.001	HIGH			<i>see SBF4.</i>	
113	SFB3 Supporting infrastructure and tools.	9	53	<0.001	HIGH			<i>see SBF4.</i>	
114	SFB3 Stricter rules for dependency management.	16	43	<0.001	HIGH	23	6	<0.001	LOW
115	SFB3 Homogeneous development culture.	17	44	<0.001	HIGH			<i>see SBF4.</i>	
116	SFB3 None of the above.	NA	NA	NA	NA	27	2	<0.001	LOW
117	SFB3 Other.	2	4	0.41	-	25	4	<0.001	LOW
118	SFB3 Clear strategic decisions for interface support.	7	54	0	HIGH	21	8	0.02	LOW
119	SFB3 Introduce maturity levels for reused artefacts.	22	40	0.02	HIGH	24	5	<0.001	LOW
120	SFB3 Bundle code more coherently in terms of functionality, e.g. into dedicated libraries.	21	40	0.02	HIGH	17	12	0.35	-
121	SFB3 Split libraries to provide more specific functionality.	25	35	0.2	-	18	11	0.19	-
122	SFB3 Merge libraries to ease the discovery of already implemented functionality.	NA	NA	NA	NA	23	6	<0.001	LOW
123	SFB3 List available artefacts in a "marketplace" to ease the discovery of useful functionality.	9	54	0	HIGH	16	13	0.58	-
124	SFB3 Announce the release of new artefacts.	8	55	0	HIGH	25	4	<0.001	LOW
125	SFB3 Developers could broadcast requests for specific functionality.	10	50	0	HIGH	28	1	<0.001	LOW
126	SFB3 Existing code could be consolidated and prepared for reuse.	16	46	0	HIGH	23	6	<0.001	LOW
127	SFB3 Structured and company-wide requirements engineering.	14	49	0	HIGH			<i>Interview data</i>	
128	SFB3 Focus on usefulness for the respective customer.	22	39	0.03	HIGH	NA	NA	NA	NA
129	SFB4 Adequate abstractions.			<i>Free text</i>		13	18	0.37	-
130	SFB4 Direct communication culture.			<i>Free text</i>		25	6	<0.001	LOW
131	SFB4 Suitable incentives.			<i>Free text</i>		29	2	<0.001	LOW
132	SFB4 High quality of artefacts.			<i>Free text</i>		10	21	0.05	HIGH
133	SFB4 Well defined reuse process.			<i>Free text</i>		25	6	<0.001	LOW
134	SFB4 Supporting infrastructure and tools.			<i>Free text</i>		11	20	0.11	-
135	SFB4 Dependency management.			<i>Free text</i>		21	10	0.05	LOW
136	SFB4 Homogeneous development culture.			<i>Free text</i>		21	10	0.05	LOW
137	SFB4 Other.			<i>Free text</i>		29	2	<0.001	LOW

12.2 Appendix for Chapter 8 — Instantiation of RASM for Inner Source

This Section presents the tabular representation of the instantiation for RASM for Inner Source used during the workshop at U. For reasons of readability, we include only the approach-related parts. The highlighted areas encode the assessment of the participants: D stands for discrepancy, C for conflict between the company values and the requirements of *Inner Source*. We include the material in the order *intent, artefacts, practices, tools, and organization*.

<i>Intent</i>	<i>Constituents</i>		<i>Details approach</i>	<i>Implications on</i>	<i>Values Approach</i>			
			<i>How does Inner Source address the goals?</i>		<i>Are the respective goals covered by Inner Source?</i>			
Motivation	current issues	economic						
			high costs of maintenance					
			business need for shared assets					
			cost pressure					
			market pressure for timely releases					
		organization						
			unbalanced/lack of resources					
			unclear requirements					
			missing access to skills					
			communication challenges					
		organizational improvement						
Goals								
economic benefits	competitiveness							
				Can be improved by directing development and maintenance efforts as needed across divisions. Familiarity of developers with systems, processes, practices, and tools reduce friction when dispatching them to new projects. Further supported by frequent release cycles.		yes		
			shortened time to market					
			flexibility for mass customization	Contributors can extend the shared assets as needed.	Coordination, code ownership and governance. Challenge: find suitable mechanisms to allow for customization that benefits users and does not disrupt them.	yes		
			feasibility for niche markets	Contributors can extend the shared assets as needed.		possible		
			decrease in development effort	Reuse of shared assets decreases the effort that would be otherwise needed for creating several instances of the functionality.		yes		
		quality						
			consistency	Inner Source can help to establish consistent use of shared assets and processes, leading to better end quality of the products.		yes		
			decrease in maintenance effort	Reuse of shared assets decreases the effort that would be otherwise needed for maintaining several instances of the functionality.		yes		
			continuous QA	Inner Source comes with a set of practices that aim to ensure high quality of the source code, e.g., peer reviews, transparent access to source code, ownership roles that support contributions as well as quality gates.		yes		
			frequent releases	Inner Source encourages frequent releases to bring patches and new functionality to users early. In this way, a fast feedback cycle is established between contributors and users.		yes		
			increase modularity of products	Inner Source encourages modular design of projects to facilitate distributed contributions, maintenance, understandability, and flexibility for reuse.		yes		
			increased quality consciousness	The transparency induced by Inner Source development practices increases the consciousness of developers for the code they produce. The meritocratic aspect can motivate to provide better solutions and avoid hacks.		yes		
		organizational benefits	knowledge transfer					
					shared best practices	Inner Source encourages the creation of homogeneous development practices. The approach requires a set of best practices that, subsequently, need to be embedded in the organization's context.	Challenge: requires a thorough debate; risks opposition to change.	yes
					shared expertise	Specialists from across the company can contribute their expertise to any given Inner Source projects, thus letting the company as a whole benefit from their knowledge.	Challenge: requires management to allow participation of experts in Inner Source.	yes
					support for innovation	Initiators are enabled to provide an experimental first running solution of a new idea to potential users. Depending on their interest, the project then can be expanded through contributions.		yes
					ease of collaboration	Availability of Inner Source products, as well as frequent feedback cycles, reduces the friction in collaboration.		yes
					awareness of existing solutions	Stakeholders, contributors, and users from multiple divisions distribute the knowledge of existing solutions.		yes
				resource alignment				

		Inner Source project thrives from the participation of a wide range of stakeholders. Their collaboration inevitably leads to focussed communication across divisions on common needs and thus can create a network of expertise and trust.	Requires infrastructure for (asynchronous) communication to technically enable access to information, as well as management support to facilitate exchange and engagement across organizational boundaries.	yes
	communication across divisions			
	information sharing	As source code and all related documents are openly accessible, any interested person is enabled to get insights into the project. This is the first step to encourage potential users and contributors to consider investing into the initiative.		yes
	organization-wide prioritization of efforts invested in reusables	Integrating multiple stakeholders into a project gives a more global perspective on the needs of the development organization. In this way, resources can be pooled and directed in an efficient way.		yes
	stake-driven effort direction	Creation and maintenance of shared reusable is driven by concrete needs as they arise. Avoids binding resources to prediction as well as upfront reusable creation. Beneficial in combination with information sharing.	Requires transparency and information flow to reach full potential: potential users and contributors must be able to find out about existence of initiative and project.	yes
	independence of organizational units	Inner Source enables units to integrate their own needs in the project, keeping the organizational effort lower than in silo-style software development. This can result in faster time to market.	Requires feedback cycle with project owner.	yes
	shared maintenance burden	Inner Source encourages community-based debugging of a project. This means that the burden of maintenance can be shared between divisions, thus increasing the motivation of contributors.	Requires active and functional community of stakeholders and contributors.	yes
Scope				
organizational units				
	single division	Inner Source development can, in principle, be adopted for any organizational scope.		in principle
	multiple divisions	However, the full advantages and benefits can only materialize when the boundaries of single organizational units are transcended.		yes
	all divisions			yes
time to benefits				
	short term	Time to benefits largely depends on the organization's readiness to adopt Inner Source. This readiness is captured in the alignment between the requirements of Inner Source and the current state of the organization in the <i>dimensions artefacts, practices, tools, and organizational structure</i> . Usually, Inner Source requires a paradigm shift in the adopting organizations. Therefore, full benefits in the short term are unlikely. Some soft benefits, e.g., constructive debate on development practices, can occur. Also, within a pilot project, first benefits might become visible.		n/a
	medium term	Depending on the starting point of the organization, Inner Source can provide substantial benefits in the medium to long term.		yes
	long term			yes

Facet Artefacts	Constituents		Details approach	Implications on	Values approach - Recorded <i>Inner Source</i>
				granularity in which reuse can be effected and reuse rates achievable.	
Kind of reused assets			Captures the type of entity that is reused.		
	requirements	use cases workflows			n/a n/a
C	design	architecture templates feature models		technical compatibility variability management	recommended n/a
C		UI templates		consistency	n/a
	source code	source text	snippets	maintenance, quality assurance, reuse approach	n/a
			classes	maintenance, quality assurance, reuse approach	recommended
			(sub)systems	maintenance, quality assurance, reuse approach	recommended
		packaged code	components	maintenance, quality assurance, reuse approach	recommended
			libraries	maintenance, quality assurance, reuse approach	recommended
			frameworks	maintenance, quality assurance, reuse approach	recommended
			services	maintenance, quality assurance, reuse approach	recommended
Characteristics of candidate reusables			Seed product	startup cost of approach.	
C	pre-existing		"seed" product—a shared asset—that is of significant value to the organization, or at least of high perceived value should exist to attract potential users and contributors. One central challenge is to identify an appropriate initial software domain and matching product, create a suitable architecture and utility basis set.		recommended
		initial runnable implementation	need for an initial basic architecture and implementation		recommended
		initial architecture	planning, analysis and design are largely conducted by the initial project founder, and are not part of the general OSS development life cycle.		recommended
		fully specified	requirements and features of the seed product need not be fully known at the outset so that the project can benefit from organization-wide input and continuously evolve		discouraged
		fully implemented	Raising a community of contributors can be hard for product purely in maintenance phase.		discouraged
	created for reuse		Seed product can be created for reuse or for solution for local need.		n/a
C	business value		Seed product should have significant value to the organization	management support, resources, incentive	recommended
		domain independent functionality	Should not be commodity software that already exists, as this risks to waste resources and might seem pointless to potential contributors and users.		discouraged
		domain specific functionality	Niche functionality that plays a key role (and, as such, a large number of stakeholders) in the organization domain.		recommended
		product group specific functionality	Niche functionality that has a key role in the organization product portfolio and offers a competitive advantage for several stakeholders.		recommended
C	scope of use		Across departments		recommended

			An Inner Source project must be needed by several stakeholders (i.e., individuals, teams, or projects that productize the shared asset) so that members from across an organization can contribute their expertise, code and resources. Also broadens the expertise available to project. However, it is essential to recognize and accommodate the tension between cultivating a general, common resource on the one hand, and the pressure to get specific releases of specific products out on time.	support, management, resources	recommended
C	stakeholders				
	potential users		Business value and variety of stakeholders help to establish a sufficiently large pool of users and contributors to establish a vibrant Inner Source project.	process compatibility	recommended
Technical compatibility of assets					
	architecture compliance		Captures the degree to which architecture compliance of artefacts enable reuse	reusability	recommended
	scope of use		Identify discriminatory value for many stakeholders, avoid feature creep	reusability, quality assurance, maintenance	recommended
	NFRs		Captures the non-functional requirements guaranteed by artefacts	reusability, maintenance, incentive	recommended
	aligning purpose		different product groups have different needs, and that groups can benefit from other groups' (specialist) contributions	maintenance, incentive, quality assurance, management	recommended
	modularity		"architecture for participation," facilitates understanding and contributions as well as reuse, allows many developers to work on different parts of the same product simultaneously, keeping merging overheads low. It is important to find a balance between functionality and simplicity as losing 'conceptual integrity' may result in a component which is no longer easy to use or whose architecture imposes too many restrictions on the client-application.		recommended
Life expectancy					
	reusable		of reusables	maintenance, responsibility, resources	recommended
	reusing products		of reusing products	stakeholders, maintenance	recommended
Origin of artefacts					
	company internal				required
C		within department	The reused artefacts come from within the reusing department.	maintenance	required
		internal third party	The reused artefacts come from within the company but from outside of the reusing department.	maintenance, management, resources	required
	company external				n/a
		Open Source	The reused artefacts are provided by an Open Source project external to the company.	legal, maintenance	n/a
		commercial provider	The reused artefacts are provided by an commercial third party external to the company.	legal	n/a
		Private third parties	The reused artefacts are provided by private third parties, e.g., on forums on the internet.	legal	n/a

Facet Practices	Constituents		Details approach	Implications on	Values approach - Recorded Inner Source
reuse practices					
	mechanism				
C		copy-paste-modify			discouraged
		hard copy duplication			discouraged
C		compile-time linking			recommended
		branching			recommended
D		service composition			recommended
		product derivation by means of variation points			n/a
	process				
		ad-hoc selection and integration of reusables			neutral
		opportunity-driven selection and integration of reusables			neutral
		strategic selection and integration of reusables			recommended
C		systematic company-wide reuse process			recommended
		reuse measurement			n/a
		configuration management of reusables			recommended
		quality model usage			n/a
requirements					
engineering practices					
D	identification of commonalities and variations between products		Emerging, not necessarily determined upfront.	Requires additional process for managing variability.	n/a
			Multiple stakeholders are needed to keep an Inner Source project alive. Their different viewpoints can broaden the expertise available to the project. However, their needs have to be reconciled. For conflicting contributions, the product owner is responsible for taking a decision.		required
			Prioritization in Inner Source happens on a need-basis. If an issue or requirement is urgent enough, concerned developers can create a solution and propose it for integration. Elicitation strategies need to be adjusted accordingly.		recommended
	integration of different stakeholder needs		Proposals for additions and improvements can be collected in a backlog and linked to an implementation via e.g. a tracker.		recommended
	strategic prioritization				
	tracing				
design practices					
	feature modeling		Not explicitly part of the Inner Source practices.		n/a
	creation of reference architecture		Initial architecture is created by product owner.		recommended
development practices					
	iterative		reinvention of existing solutions to overcome their limitations, e.g., by reworking prototypes. Risks: reinvention might be limited by availability of required resources.	Quality, Maintenance	recommended
	incremental		focus on running code over perfect solutions. Maintain a high standard of quality nevertheless.		recommended
	serial		starting from complete specification, developing the artefact top-down.		discouraged
C	contribution management		Contributions from motivated developers are a key mechanism to keep an Inner Source project alive. Nevertheless, the product owner and core developers need to manage contributions to ensure a high quality standard of the system. One way this is commonly addressed in Open Source is a staged contribution process: Contributors propose their patches to the core team that either accepts it and includes it to the code base or asks for concrete improvements. To avoid frustrated contributors, the core team should respond quickly and be constructive.		recommended
quality assurance practices					
	reviews		High quality of a reusable is a key element for user and contributor motivation to get involved.		

	architecture reviews	Review of architectural designs, e.g., by product owner or core developers. Peer review, performed based on self-selection by other interested developers. Goal: to ensure that any code that is checked in is of good quality, does not contain hacks, and will not lead to an undesirable path of evolution for the respective unit of the project. Risks: transparency could be out of developer/manager comfort zone.		recommended
C	peer review of source code			required
D	architecture compliance	Ensured by the role of the product owner/"benevolent dictator".		recommended
	test			
D	automated unit tests	Since Inner Source is centered around shared development of a system (often build with continuous integration), breaking the build is a serious issue. Therefore, changes must be tested before committing them to the shared repository. Support Continuous Integration by providing automatic feedback on the fitness of the entire code base.		required
C	automated integration tests system tests			recommended n/a
maintenance practices				
	task assignment regression testing prioritization of tasks cost estimation			
release and deployment practices				
D	continuous integration	New contributions are integrated in the code base and tested frequently (e.g. at least daily). As a result, development can respond quickly to failures and user feedback and users can profit from improvements.		recommended
	frequent releases configuration management	Collect feedback from user basis by providing frequent releases. For risk averse users, provide stable version, for curious users, provide beta version.		recommended
documentation practices				
C	code comments concise and consistent naming in code tracing to requirements and tests	Focus in Inner Source is on the code. It should be of high quality, including descriptive comments of functionality and purpose, appropriate naming, etc..		recommended recommended n/a
C	descriptive documentation of functional purpose descriptive documentation of extra functional guarantees and limitations	Information on functionality and limitations can be found in archived developer communications on mailing lists and wikis.		recommended recommended
homogeneity of practices				
D	reuse practices development practices maintenance practices quality assurance practices	homogeneous reuse, development, maintenance, documentation, and quality assurance practices ensure transparency and induce trust in the resulting artefacts.		recommended recommended recommended
	documentation practices			n/a
	coordinated product release schedules coordinated integration schedules	A number of case studies on Inner Source adoption reports on coordination practices for integration and release. However, these practices need to be tailored to the specific context of the adopting organization.		recommended recommended

Facet Tools	Constituents		Details approach	Implications on	Values approach - Recorded <i>Inner Source</i>
development tools					
C	version control CASE tools		The source code of an Inner Source project is accessible openly in the respective version control system to allow potential users and contributors full access.		required
		IDEs			required
		code recommender			neutral
		code search engine			recommended
		issue tracker			recommended
C		build server supporting continuous integration			recommended
infrastructure tools					
D	documentation	wiki	Wiki pages for each Inner Source project give information about its purpose, functionality, scope, etc. to potential users and contributors. In Q&A forums or wikis, frequent questions regarding the project are collected and answered for future reference.		recommended
		Q&A forums			recommended
C		list archives	The archives contain all mailing list communication to document development decisions and their rationale for future reference. They often contain detailed and specific information that can be highly relevant to future contributors and users.		recommended
C	communication		Inner Source heavily relies on communication media to persist discussions and decisions regarding the development of a project. In this way, newcomers can trace design and implementation rationales even after a project has changed ownership.		
		developer mailing lists	Developer mailing lists enable asynchronous communication between the contributors of an Inner Source project. In addition, it persists the discussions for future reference. Communication on the lists are public and available in archives.		recommended
		user mailing lists	User mailing lists serve to inform users of new features and planned changes in the project.		recommended
		IRC channels	IRC channels allow for direct real-time communication between contributors as well as users. They can be used for one-to-one communication, as well as broadcast medium for key developers to have, e.g. Q&A sessions.		recommended
homogeneity of toolset					
C	homogeneity		set of common and compliant development tools aims to make contributing easy: it enables a smooth transition from using and observing a project to active participation. Also, it avoids undesirable behaviours such as tediously replicating the original shared asset's source code into different repositories, likely causing significant merging problems later. Risk: within large organizations, often a wide range of different tools are in use and cherished. Therefore, a transition to a new unified toolset can become a political issue. Standardized toolsets can be bundled and provided, e.g., by means of a software forge.		recommended

Facet	Organization	Constituents		Details approach	Implications on	Values approach - Recorded
						<i>Inner Source</i>
business context						
C	D	culture				
		management		Management needs to be comfortable with sharing resources and responsibility, allowing/encouraging developers to work according to the principles of Inner Source.		recommended
D	D	communication		Communication becomes visible and accessible in Inner Source, as it is conducted electronically and archived in a form that is accessible company-wide.		required
		support				
D	D	top management		Top management support is vital to ensure the required resources are made available for company-wide transition.		required
		middle management		Middle management support is required to ensure that community building, as well as contributions for development and maintenance of shared assets are possible.		required
C	C	resources				
		human resources		Inner Source requires human resources to be realized: developers need the allowance to participate in projects and full time positions are required for the core and support teams.		required
D	D	budget		Inner Source requires additional budget for, e.g., a suitable infrastructure.		required
		application domain				
D	D	type of developed software		Inner Source is independent of the type of developed software.		
		legal constraints		If the adopting organization operates under specific legal constraints, Inner Source practices can and should be tailored to fit the specific organization's needs.		
D	D	life cycle duration		Inner Source is in principle independent of the typical life cycle duration. However, to provide best value, functionality of high value to the organization should be developed as shared asset.		
		strategy				
D	D	reuse vision		Inner Source projects aim to provide high quality shared assets for reuse and extension to the organization. It focuses on running code and high modularity to provide immediate benefits as well as ease of contribution and extension.		recommended
		development context				
D	D	software development approach		Inner Source is in principle independent of given development approaches at the organization. Given the required infrastructure and resources, it can co-exist with other approaches. Nevertheless, in settings with significant differences, management needs to explicitly guide the interaction between different processes.		
		workforce software organization		The size of the workforce of the adopting organization has a potential impact on how Inner Source can be performed. Especially, it determines how much responsibility can be taken on by a support unit within the organization. Tailoring of roles, responsibilities, and processes needs to balance potential skews of resources.		
C	C	product development				
		reusables development				
C	C	total				
		homogeneity				
C	C	practices		Inner Source works best in a context that features homogeneous development and		recommended

C	confidence	tools		maintenance practices, a homogeneous toolset, a shared reuse vision, and a compatible organizational culture. If these are not (yet) given, adopting organizations need to create the conditions for Inner Source to be successful (e.g., by fulfilling the mandatory preconditions w.r.t. infrastructure, processes, practices, and organizational aspects).	recommended	
		reuse vision			recommended	
		culture			recommended	
D	skills	developers		Developers need to feel at ease with the transparency induced by Inner Source. While for some this transparency will bring the sense of reward for their contributions, others might feel threatened by the potential scrutiny coming over their contributions. Guidance should be offered to support sceptics to participate. Managers need to feel comfortable to share responsibility with others that they can not directly influence. They need to adjust to transparency and meritocracy.	recommended	
		managers			recommended	
C	organizational	project team experience		Can facilitate the creation of the seed project of an Inner Source initiative. Organizational maturity with respect to homogeneous practices and tools can facilitate the adoption of Inner Source.	recommended	
		maturity			recommended	
	individual	managers		Managers need to feel comfortable to share responsibility with others that they can not directly influence. They need to adjust to transparency and meritocracy.		
			change management		required	
			coordination		required	
			reuse understanding		required	
			social competence		required	
		developers			Developers need to feel at ease with the transparency induced by Inner Source. While for some this transparency will bring the sense of reward for their contributions, others might feel threatened by the potential scrutiny coming over their contributions. Guidance should be offered to support sceptics to participate.	
			technical competence			required
			social competence			required
		reuse understanding	required			
reuse-related roles						
	consumer					
	producer roles					
D	coordinator roles	chief architect		Strategic responsibility for project. Should give shape to architecture of the shared asset, but, arguably, should also take part in implementation to remain grounded in the system details. "benevolent dictator", works closely with Liaison.	required	
		core contributors		Individuals that by merit of their contributions have become central contributors to the Inner Source project. Often, they take on the responsibility for a specific part of the system. Extend the project in accordance with architect. "trusted lieutenants"	required	
		contributor		Contribute to the Inner Source project, but can not directly modify the codebase. The respective contributions are reviewed by the chief architect or the core contributors for fitness in terms of functionality, quality, architecture compliance, and generality.	required	
C		liaison		Overall responsibility for Inner Source Project, communicates with internal customers, interfaces with the core team. Works closely with Chief Architect.	recommended	

C	release advocate		Responsible for specific release, works closely with users to assess impact of changes and ensure a smooth transition		recommended
	delivery advocate		Assigned to organizational units that are new customers of a shared asset. Ensures that the unit adopts the required tools and practices and produces contributions in line with the target vision, quality, and architecture of the project.		recommended
D	feature advocate		Responsible to see a feature to completion.		recommended
	support team		Helps business units with operational tasks, such as writing documentation, release notes, and release management tasks.		recommended
	project manager		Responsible for release planning, process compliance, progress monitoring.		recommended
project management					
	process management				
C	process alignment between Inner Source development team and using/contributing units		Alignment between the Inner Source development style and contributing/using projects needs to be modeled to prevent damage to products (e.g., troubles with integration or missing a delivery deadline)	Challenge: Attempting Inner Source without properly aligning surrounding processes can significantly reduce the benefits that can be obtained.	required
	process for developing Inner Source projects		Clear guidelines on how Inner Source development should be implemented (e.g., how contributions are selected and integrated, which roles are implemented, which responsibilities are assigned to whom, how Inner Source interfaces with the rest of the organization...)	Switching parts of the development to Inner Source can lead to conflicts with general organizational processes. These friction points need to be addressed in a timely manner to prevent failure (e.g., loss of acceptance and support for Inner Source initiative).	required
	project planning				
D	long-term vision of shared asset		Core team should have a long term vision of shared assets. Requires detailed understanding of needs of contributing products.	Information spreading to contributors and users is key.	recommended
	coordination of contributions		Core team should coordinate and synchronize different contributors to avoid redundant work occurring, e.g., by contributions that are too specific for general utility. Depending on size of project and criticality of asset, this might require a full time project manager.	Distinguish general development from customer-specific changes to ensure the project stays viable for stakeholders.	required
	monitoring and steering				
C	quality assurance of contributions		Core team needs to review and triage contributions.	Depending on the staffing, this can present a bottleneck. A functioning and large community as well as stakeholders in contributing units are needed to alleviate this burden.	required
	tracking of feature evolution		Core team needs to track contributions to specific features to identify issues (delay, insufficient quality, etc.) early on.		recommended
	human issues				
	manage transparency		Developers might feel pressured due to the increasing transparency (insecurity, averseness to scrutiny, worry about their job security, consciousness of embarrassing state of code base etc.). Furthermore, they might need to develop new styles of communication and contribution.	Willingness to contribute might be largely affected by these aspects. To overcome them, the Inner Source initiative needs to ensure an appropriate style of communication and a guided entry path for newcomers to help potential contributors adjust to the new development paradigm.	recommended

12.3 Appendix for Chapter 10 — Rationales for impacts of the library usage assessment model

This document contains all impacts of our assessment model of external library usage, presented in Chapter 10. The impacts are grouped by activity and are followed by a brief rationale justifying their inclusion in the model.

Impacts define how facts influence activities. A justification for each impact provides a rationale for the impact which increases confirmability of the model and the assessments based on the model. Note that a fact might have a positive impact on one activity whilst negatively impacting another one.

Modify [Extension/Configuration Capability | EXTENT] $\xrightarrow{+}$ [Modify] A high extension capability of an external library positively impacts this activity because it increases the chances that a modification can be accomplished with the same library.

[System, Library | ENTANGLEDNESS] $\xrightarrow{-}$ [Modify] High entangledness of an external library with a system negatively impacts modifications as it might cause a many parts of the system to be affected.

[System, Library | ADEQUACY] $\xrightarrow{+}$ [Modify] Adequate usage of a library within a system eases the activity modify because it increases the chances of consistent and correct use.

[Support | EXTENT] $\xrightarrow{+}$ [Modify] Good support for a library eases modifications as developers are able to obtain help with their task if needed.

[Developers | FAMILIARITY] $\xrightarrow{+}$ [Modify] Familiarity of developers with a specific library eases modifications because the developers already know the particularities of the library. Therefore, modification are likely to be completed faster and with fewer errors.

Understand [Extension/Configuration Capability | EXTENT] $\xrightarrow{-}$ [Understand] Whilst high extension capability positively impacts modifications, it hinders understanding. The reason is the high complexity brought about by the flexibility of extension mechanisms which make it harder to understand how to use the library.

[System, Library | ENTANGLEDNESS] $\xrightarrow{-}$ [Understand] Entangledness also tends to cause difficulties to clearly understand how a library is to be used. Especially scatteredness of method calls could hamper understanding.

[System, Library | ADEQUACY] $\xrightarrow{+}$ [Understand] A high adequacy of use eases understanding, as the library is employed as expected, in coherence with given documentation.

[API | SIZE] $\xrightarrow{-}$ [Understand] The larger an API the harder to understand it becomes. The sheer size of the interface makes it difficult to get an overview of the provided functionality.

[Support | EXTENT] $\xrightarrow{+}$ [Understand] A large extent of support for a library facilitates understanding of a software system as developers can obtain qualified assistance in using the library.

[Library | PREVALENCE] $\xrightarrow{+}$ [Understand] A high prevalence of a library supports understanding a software system because it often implies a good availability of additional documentation such as web blog articles or forum discussions.

[Developer | FAMILIARITY] $\xrightarrow{+}$ [Understand] A high familiarity of developers with a library eases understanding a software system as less concepts need to be learned from scratch.

Migrate [Library | PREVALENCE] $\xrightarrow{+}$ [Migrate]

A high prevalence of a library positively influences migration, as it usually gives rise to alternative implementations of the required functionality.

[System, Library | ENTANGLEDNESS] $\xrightarrow{-}$ [Migrate] If a system is highly entangled with a library, migration to a different library will be more difficult as more different code sections have to be changed.

[System, Library | ADEQUACY] $\xrightarrow{+}$ [Migrate] If a system's usage of library is in line with the library's intended usage, migration is less difficult since finding alternatives for or removal of unintended uses can be problematic.

[Support | EXTENT] $\xrightarrow{+}$ [Migrate] A high extent of support eases migration of a library since developers are able to obtain direct assistance for this task from the library's provider.

[Developer | FAMILIARITY] $\xrightarrow{+}$ [Migrate] A developer who is highly familiar with a library can more easily migrate a system to a different library compared to a developer who doesn't know the library.

[Extension/Configuration Capability | EXTENT] $\xrightarrow{-}$ [Migrate] A high capability for extension and configuration inhibits migration as it is less likely that alternatives provide the same flexibility.

Protect [Vendor | REPUTATION] $\xrightarrow{+}$ [Protect]

The reputation of the library vendor positively influences protection of a system, as a renowned vendor can be expected to provide critical updates in a timely manner.

[Vendor | SIZE] $\xrightarrow{+}$ [Protect] A large vendor is expected to be more capable of providing fixes to critical security flaws and thus protecting an including software system is supported.

[Vendor | REPUTATION] $\xrightarrow{+}$ [Protect] A vendor with a good reputation is expected to be more capable of providing fixes to critical security flaws and thus protecting an including software system is supported.

[Library, System | ADEQUACY] $\xrightarrow{+}$ [Protect] An adequate usage of a library improves protection, as the security measures can be used as intended.

[Vulnerabilities | EXTENT] $\xrightarrow{-}$ [Protect] Security flaws in a library inhibit protection of an including software system as the security issue of the library may affect the security of the overall system.

[Support | EXTENT] $\xrightarrow{+}$ [Protect] Support positively impacts protection, as developers can rely on experts to ensure the security of their application.

[Library | PREVALENCE] $\xrightarrow{+}$ [Protect] High prevalence of a library might help to protect a system, as the library is assumed to be mature, and potential fixes are expected to be provided rapidly.

[Library | MATURITY] $\xrightarrow{+}$ [Protect] A high maturity positively affects the ability to protect the overall software system as more mature libraries are expected to exhibit less security-related problems.

[Developer | FAMILIARITY] $\xrightarrow{+}$ [Protect] The familiarity of developers with a library helps to protect the system because developers are aware of the security mechanisms and how to use them.

Distribute [License, System | COMPATIBILITY] $\xrightarrow{+}$ [Distribute]

Third party libraries impact the distribution of a system as low compatibility of licenses, as well as high license fees [License | PRICE] $\xrightarrow{-}$ [Distribute], can block the distribution of a system and therefore need to be taken into account.

[License | PRICE] $\xrightarrow{-}$ [Distribute] A high license price of a library makes distribution more difficult as the inclusion of the library affects the cost to produce the software and thus impacts the (commercial) distribution.

[License | COMPATIBILITY] $\xrightarrow{+}$ [Distribute] The compatibility of the license with the system is an essential precondition to enable distribution of the software.

List of Figures

1.1	A brief overview of the technological development since reuse became a topic of software engineering research. The time-line displays the media and infrastructure available to share reusable entities before the WWW-era and highlights the birth of open source platforms and available tools that now serve as important sources for reusable code and/or knowledge. Based on [24].	5
1.2	Map of the contributions presented in this thesis and their relations. The arrows represent the flow of results: the synthesis of Contribution I is integrated in Contribution II. The methods and tools presented in Contribution III can be used to support the application of the Reuse Adoption Support Model of Contribution II. The rounded boxes provide a mapping of the contributions to the Chapters of this thesis.	8
2.1	Examples for potentially reusable entities created during software development. On the left side of the figure, the different types of artefacts are ordered in terms of their level of abstraction. The arrows connected to the artefact types refer to typical examples of the respective type. Knowledge is listed as it is mentioned in several definitions. However, it is highlighted as it is not a documented type of artefact.	19
2.2	Types of reusables in practice.	21
2.3	Taxonomic definition of Software Reuse from the perspective of the reuse consumer based on [44, 45].	23
2.4	Mapping of Frakes and Fox reuse failure modes on the impacted activities of the reuse process.	28
3.1	Study setup: case G, case U, and case integration.	37
6.1	Summary of results, according to authors' data analysis and interpretation.	81
6.2	Example of the graphical representation of the results.	82
6.3	RQ1 - Sources of reusable entities and way of access, questions FAR1 and RAF2.	82
6.4	RQ1 - Reused entities, question RAF1.	83
6.5	RQ1 - Technical realization of reuse, questions TRR1 and TRR2.	84
6.6	RQ2 - Inhibitors to reuse and issues due to reuse, questions CHR1 and CHR2.	85
6.7	RQ2 - Issues of absence of reuse and benefits of reuse, questions CHR3 and SFB1.	86
7.1	Company Reuse Placement according to the four reuse facets.	120

8.1	Excerpt of the tabular representation of the RASM tool facet for <i>Inner Source</i> used during the workshop. The graphic shows the columns and the first row of the table.	127
8.2	Details of the RASM application for <i>Inner Source</i> at U. The elements in the dashed boxes present the schedule of the one-day workshop. The single elements detail on the time given to participants for the particular tasks, the concrete tasks that were effected, and the roles of the participants and the researcher during the given tasks. The feedback questionnaire was provided on-line and could be filled in until a week after the workshop.	128
8.3	Responses for the question <i>model assessment</i> . Question text: “ <i>The model for Inner Source...</i> ”	134
8.4	Responses for the question <i>model accessibility</i> . Question text: “ <i>When applying the model during the workshop...</i> ”	135
9.1	This figure illustrates our approach: we extract relevant identifiers for each concept and compute the best matches within the study object. In the preprocessing step, different approaches can be taken to select and prepare the identifiers that are subsequently used.	148
9.2	This figure visualizes our evaluation process for detecting re-implementations of Collection functionality. To establish a base-line of known duplicates, we injected the Collection implementation of Guava into the Qualitas Corpus and measured the detection rates for these known “re-implementations”.	150
9.3	Overview of the LSI-based approach. During the analysis, we extract relevant concepts by means of identifiers present in each file and compute the best matches between them.	155
9.4	This figure provides a high level (left hand side) as well as a detailed view (right hand side) of the study design.	157
9.5	The figure illustrates the three steps of our idea. A and B denote two software systems. LU denotes the logical union of the systems during clone detection. Circles represent entities of the systems, e.g. functions, classes, or files. The systems contain two clone classes: the double-lined (α) and the dashed (β) group. Arrows denote the dependencies between entities belonging or connected to clones. L denotes an extracted library, containing (α).	167
9.6	Different usage patterns of cloned regions.	168
10.1	The meta-model of the assessment model.	173
10.2	Instantiated assessment model with facts, development activities, and impacts between them.	174
10.3	Facts and associated metrics of assessment model	177
10.4	Result distribution for total and distinct method calls and scatteredness for each JAR file used by the system.	183
10.5	Aggregated view of the assessment results	184

List of Tables

3.1	Characterization of the participating companies	41
3.2	The empirical studies in numbers	41
4.1	Roles of participants - questionnaire	46
4.2	Roles of participants - interview	46
4.3	Which are your top-three ways of sharing artefacts?	50
4.4	Which are your preferred ways to find reusables? Please indicate the top three.	51
4.5	What do you do to properly understand and adequately select reusable artefacts?	51
4.6	Which of the following possibilities of reuse do you employ most? Please indicate the top three.	52
4.7	Which are the top-three types of artefacts you reuse?	53
4.8	What is the scope of the reused artefacts?	53
4.9	What granularity do the reused entities typically have?	53
6.1	Questions selected for comparison for RQ1	78
6.2	Questions selected for comparison for RQ2	79
7.1	Structure of RASM - Overview	104
7.2	RASM - Encoding of the current evidence of Reuse Influence Factors according to [80]. The encoding is used to highlight the stage of validation of the respective elements in Tables 7.5 to 7.8.	105
7.3	Factors and constituents of the <i>Intent</i> element of the RASM.	109
7.4	<i>Reuse facets</i> of the RASM.	109
7.5	Factors and constituents of the <i>Artefacts</i> facet of the RASM. The symbols encode the stage of validation of the factors (see Table 7.2).	110
7.6	Factors and constituents of the <i>Practices</i> facet of the RASM. The symbols encode the stage of validation of the factors (see Table 7.2).	111
7.7	Factors and constituents of the <i>Tools</i> facet of the RASM. The symbols encode the stage of validation of the factors (see Table 7.2).	112
7.8	Factors and constituents of the <i>Organization</i> facet of the RASM. The symbols encode the stage of validation of the factors (see Table 7.2).	113

7.9	This table presents an overview of the different reuse facets reported in literature. Facets in bold face are reported as structuring reuse dimensions in the respective papers. Facets in bold face and italics denote the final <i>reuse facets</i> selected for the model. The additional facets are reported as influence factors and included for a more comprehensive justification of the heterogeneity of the reuse facets in the model.	117
8.1	Summary of the compliance assessment between U and <i>Inner Source</i> based on the tabular instantiation of RASM. The table displays the number of factors (constituents) of each facet, as well as the number of conflicts, discrepancies, and agreements between the company values and <i>Inner Source</i> . In addition, the number of new and relevant factors is reported.	130
8.2	Summary of the compliance assessment between U and <i>Inner Source</i> based on the tabular instantiation of RASM. The table displays the number of factors (constituents) of each facet, as well as the number of conflicts, discrepancies, and agreements between the company values and <i>Inner Source</i> . In addition, the number of new and relevant factors is reported. (*Three elements were discouraged and in conflict. They are summarized with conflict and required.)	132
9.1	Parameter setting for both techniques	156
9.2	Profile of the top 200 result pairs for LSI and ACD.	160
9.3	Hits of top 200 result pairs for LSI (LSI-ACD) and hits of top 200 result pairs for aggregated clone detection (ACD-LSI).	161
9.4	Practitioner rating of non-deprecated re-implementations.	162
12.1	Interview guide: topics and sample questions for cases G and U	201
12.2	Scale aggregations for questionnaire U. <i>Question ID</i> refers to the respective question, <i>Scale U</i> relates the type of the scale (<i>L4</i> for a 4 point Likert scale, <i>L5</i> for a 5 point Likert scale) and reports the extreme values of the given scale. <i>Aggregation</i> illustrates how, for the given scale, the values were aggregated in the categories <i>Low</i> and <i>High</i> . <i>P<n></i> denotes the number of responses at the given point of the Likert scale.	202
12.3	Responses and values relevant for RQ1	203
12.4	This table contains the responses relevant for RQ2	204
12.6	This table contains the responses relevant for RQ2 — continued from previous page . . .	205

Bibliography

- [1] W. B. Frakes and C. J. Fox, “Quality improvement using a software reuse failure modes model,” *Software Engineering, IEEE Transactions on*, vol. 22, no. 4, pp. 274–279, 1996.
- [2] C. Krueger, “Software reuse,” *ACM Computing Surveys*, vol. 24, no. 2, pp. 131–183, 1992.
- [3] W. Lim, “Effects of reuse on quality, productivity, and economics,” *IEEE Software*, vol. 11, no. 5, pp. 23–30, 2002.
- [4] D. Batory and S. O’Malley, “The design and implementation of hierarchical software systems with reusable components,” *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 4, pp. 355–398, Oct. 1992. [Online]. Available: <http://doi.acm.org/10.1145/136586.136587>
- [5] E.-A. Karlsson, *Software reuse: a holistic approach*. John Wiley & Sons, Inc., 1995.
- [6] S. Henninger, “An evolutionary approach to constructing effective software reuse repositories,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 2, pp. 111–140, 1997.
- [7] L. Bass, P. Clements, S. Cohen, L. Northrop, and J. Withey, “Product line practice workshop report,” Software Engineering Institute, Tech. Rep., 1997.
- [8] K. Pohl, G. Böckle, and F. J. van der Linden, *Software product line engineering: foundations, principles and techniques*. Springer Science & Business Media, 2005.
- [9] K.-J. Stol, P. Avgeriou, M. A. Babar, Y. Lucas, and B. Fitzgerald, “Key Factors for Adopting Inner Source,” *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2014.
- [10] A. Mili, R. Mili, and R. T. Mittermeir, “A survey of software reuse libraries,” *Annals of Software Engineering*, vol. 5, pp. 349–414, 1998.
- [11] W. B. Frakes and K. Kang, “Software reuse research: Status and future,” in *IEEE Transactions on Software Engineering*, vol. 31, no. 7, 2005, pp. 529–536.
- [12] O. Hummel and C. Atkinson, “Using the web as a reuse repository,” in *Reuse of Off-the-Shelf Components*. Springer, 2006, pp. 298–311.
- [13] M. Morisio, M. Ezran, and C. Tully, “Success and failure factors in software reuse,” *Software Engineering, IEEE Transactions on*, vol. 28, no. 4, pp. 340–357, 2002.
- [14] M. McILROY, “Mass produced software components,” in *NATO Software Engineering Conference Report*, 1968.
- [15] Y. Kim and E. A. Stohr, “Software reuse: Issues and research directions,” in *Twenty-Fifth Hawaii International Conference on System Sciences*, 1992.

- [16] H. Mili, F. Mili, and A. Mili, "Reusing software: Issues and research directions," in *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 21, no. 6, 1995.
- [17] V. Basili, G. Caldiera, and H. Rombach, "The experience factory," *Encyclopedia of software engineering*, 1994.
- [18] R. G. Fichman and C. F. Kemerer, "Incentive compatibility and systematic software reuse," *The Journal of Systems and Software*, vol. 57, pp. 45–60, 2001.
- [19] G. Sindre, R. Conradi, and E. Karlsson, "The reboot approach to software reuse," *Journal of Systems and Software*, vol. 30, no. 3, pp. 201–212, 1995.
- [20] R. D. Banker, R. J. Kauffman, and D. Zweig, "Repository evaluation of software reuse," *Software Engineering, IEEE Transactions on*, vol. 19, no. 4, pp. 379–389, 1993.
- [21] P. Mohagheghi and R. Conradi, "Quality, productivity and economic benefits of software reuse: a review of industrial studies," *Empirical Software Engineering*, no. 12, pp. 471–516, 2007.
- [22] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the Extent and Nature of Software Reuse in Open Source Java Projects," in *ICSR'11*, 2011.
- [23] K.-J. Stol and B. Fitzgerald, "Inner Source – Adopting Open Source Development Practices in Organizations: A Tutorial," in *IEEE Software*, 2015.
- [24] D. P. Management, "Timeline: Digital technology and preservation."
- [25] V. Bauer, "Challenges of structured reuse adoption — Lessons learned," in *Profes 2015*, 2015.
- [26] V. Bauer and L. Heinemann, "Understanding API Usage to Support Informed Decision Making in Software Maintenance," in *CSMR 2012*, 2012.
- [27] V. Bauer, J. Eckhardt, B. Hauptmann, and M. Klimek, "An Exploratory Study on Reuse at Google," in *SER&IP's*. ACM, 2014.
- [28] D. M. Weiss, P. Clements, K. Kang, and C. Krueger, "Software product line hall of fame," in *Software Product Line Conference, 2006 10th International*, Aug 2006, pp. 237–237.
- [29] A. Lynex and P. J. Layzell, "Organisational considerations for software reuse," *Annals of Software Engineering*, vol. 5, pp. 105–124, 1998.
- [30] J. Businge, A. Serebrenik, and M. G. J. van den Brand, "Eclipse API Usage: The Good and The Bad," in *Sixth International Workshop on Software Quality and Maintainability*, 2012.
- [31] R. Lämmel, E. Pek, and J. Starek, "Large-scale, AST-based API-usage analysis of open-source Java projects," in *SAC'11*, 2011.
- [32] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers*, vol. 82, pp. 25–46, 2011.
- [33] F. Deissenboeck, L. Heinemann, B. Hummel, and S. Wagner, "Challenges of the dynamic detection of functionally similar code fragments," in *CSMR'12*, 2012. [Online]. Available: <http://doi.ieeecomputersociety.org/10.1109/CSMR.2012.38>
- [34] V. Bauer, L. Heinemann, and F. Deissenboeck, "A Structured Approach to Assess Third-Party Library Usage," in *ICSM'12*, 2012.
- [35] S. Raemaekers, A. van Deursen, and J. Visser, "Exploring Risks in the Usage of Third-Party Libraries," in *Sixth International Workshop on Software Quality and Maintainability*, 2012.

- [36] B. Klatt, Z. Durdik, H. Koziolok, K. Krogmann, J. Stammel, and R. Weiss, "Identify impacts of evolving third party components on long-living software systems," in *CSMR'12*, 2012.
- [37] V. Bauer and B. Hauptmann, "Assessing cross-project clones for reuse optimization," in *IWSC*, 2013.
- [38] V. Bauer, "Facts and fallacies of reuse in practice," in *CSMR 2013*, 2013.
- [39] V. Bauer, T. Völke, and E. Jürgens, "A Novel Approach to Detect Unintentional Reimplementations," in *ICSME'14*, 2014.
- [40] V. Bauer, T. Völke, and S. Eder, "Combining clone detection and latent semantic indexing to detect reimplementations," in *IWSC 2016*, 2016.
- [41] V. Bauer and A. Vetrò, "Comparing reuse practices in two large software-producing companies," *accepted for publication in the Journal of Systems and Software*, 2016.
- [42] S. Wartik and T. Davis, "A phased reuse adoption model," *Journal of Systems and Software*, vol. 46, no. 1, pp. 13–23, 1999.
- [43] W. Frakes and C. Terry, "Software reuse: metrics and models," *ACM Computing Surveys (CSUR)*, vol. 28, no. 2, pp. 415–435, 1996.
- [44] V. R. Basili and H. D. Rombach., "Support for comprehensive reuse." in *IEEE Software*, 1991.
- [45] J. Llorens, *Software Reuse Fundamentals*, The Reuse Company, 2005.
- [46] R. T. Mittermeir and W. Rossak, "Software bases and software archives: alternatives to support software reuse," in *Proceedings of the 1987 Fall Joint Computer Conference on Exploring technology: today and tomorrow*. IEEE Computer Society Press, 1987, pp. 21–28.
- [47] R. P. Díaz, "The Disappearance of Software Reuse," in *IEEE*, 1994.
- [48] H. Gall, M. Jazayeri, and R. Kloesch, "Research directions in software reuse: Where to go from here?" in *SSR*, 1995.
- [49] M. Shaw, "Architectural issues in software reuse: It's not just the functionality, it's the packaging," in *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. SI. ACM, 1995, pp. 3–6.
- [50] M. Wasmund, "Reuse facts and myths," in *Proceedings of the 16th International Conference on Software Engineering*, ser. ICSE '94. Los Alamitos, CA, USA: IEEE Computer Society Press, 1994, pp. 273–. [Online]. Available: <http://dl.acm.org/citation.cfm?id=257734.257786>
- [51] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "Mapo: Mining and recommending api usage patterns," in *ECOOP 2009–Object-Oriented Programming*. Springer, 2009, pp. 318–343.
- [52] M. Sojer and J. Henkel, "Code Reuse in Open Source Software Development: Quantitative Evidence, Drivers, and Impediments," *Journal of the Association for Information Systems*, 2010.
- [53] W. Spoelstra, M. Iacob, and M. van Sinderen, "Software reuse in agile development organizations: a conceptual management tool," in *SAC 2011*, 2011.
- [54] S. Raemaekers, A. van Deursen, and J. Visser, "An analysis of dependence on third-party libraries in open source and proprietary systems," in *CSMR'12*, 2012.
- [55] R. P. Díaz, "Status report: Software reusability," *IEEE Software*, vol. 10, no. 3, pp. 61–66, 1993. [Online]. Available: <http://doi.ieeecomputersociety.org/10.1109/52.210605>
- [56] D. Hristov, O. Hummel, M. Huq, and W. Janjic, "Structuring software reusability metrics for component-based software development," in *Proceedings of Int. Conference on Software Engineering Advances (ICSEA)*, 2012.

- [57] M. B. Rosson and J. M. Carroll, "The reuse of uses in smalltalk programming," *ACM Transactions on Computer-Human Interaction*, vol. 3, pp. 219–253, 1996.
- [58] B. M. Lange and T. G. Moher, "Some strategies of reuse in an object-oriented programming environment," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '89. New York, NY, USA: ACM, 1989, pp. 69–73. [Online]. Available: <http://doi.acm.org/10.1145/67449.67465>
- [59] R. Holmes and R. J. Walker, "Systematizing Pragmatic Reuse Tasks," *ACM Trans. Softw. Eng. Methodol.*, 2012.
- [60] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki, "An Exploratory Study of Cloning in Industrial Software Product Lines," in *CSMR 2013*, 2013.
- [61] C. J. Kapser and M. W. Godfrey, "'Cloning considered harmful' considered harmful: patterns of cloning in software," *Empirical Software Engineering*, 2008.
- [62] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants," in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, 2014.
- [63] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 2009, pp. 485–495.
- [64] T. Ravichandran and M. A. Rothenberger, "Software reuse strategies and component markets," *COMMUNICATIONS OF THE ACM*, 2003.
- [65] G. W. Hislop, "Analyzing existing software for software reuse," *Journal of Systems and Software*, 1997.
- [66] W. B. Frakes and S. Isoda, "Success factors of systematic reuse," *Software, IEEE*, vol. 11, no. 5, pp. 14–19, 1994.
- [67] P. Clements and L. Northrop, "Software product lines: practices and patterns," 2002.
- [68] J. Llorens, J. Fuentes, R. Prieto-Diaz, and H. Astudillo, "Incremental Software Reuse," *ICSR*, 2006.
- [69] P. Diebold and A. Vetrò, "Bridging the gap: Se technology transfer into practice: Study design and preliminary results," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '14. New York, NY, USA: ACM, 2014, pp. 52:1–52:4. [Online]. Available: <http://doi.acm.org/10.1145/2652524.2652552>
- [70] K. Sherif and A. Vinze, "Barriers to adoption of software reuse. a qualitative study." *Information and Management*, vol. 41, pp. 159–175, 2003.
- [71] H. Gall and R. Klösch, "Reuse engineering: software construction from reusable components," in *Computer Software and Applications Conference, 1992. COMPSAC'92. Proceedings., Sixteenth Annual International*. IEEE, 1992, pp. 79–86.
- [72] A. R. Yazdanshenas and L. Moonen, "Fine-grained change impact analysis for component-based product families," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 119–128.
- [73] I. Jacobson, G. Booch, and J. Rumbaugh, *The unified software development process*. Addison-Wesley Reading, 1999.

- [74] M. Luckey, A. Baumann, D. M. Fernandez, and S. Wagner, "Reusing Security Requirements Using an Extended Quality Model," in *SESS*, 2010.
- [75] C. Palomares, X. Franch, and C. Quer, "Requirements Reuse and Patterns: A Survey," in *REFSQ*, 2014.
- [76] L. Heinemann, "Effective and efficient reuse with software libraries," Ph.D. dissertation, Technische Universität München, 2012.
- [77] R. Cloutier, G. Muller, D. Verma, R. Nilchiani, E. Hole, and M. Bone, "The concept of reference architectures," *Systems Engineering*, vol. 13, no. 1, pp. 14–27, 2010.
- [78] V. R. Basili, "Viewing maintenance as reuse-oriented software development," *Software, IEEE*, vol. 7, no. 1, pp. 19–25, 1990.
- [79] O. P. N. Slyngstad, A. Gupta, R. Conradi, P. Mohagheghi, H. Rønneberg, and E. Landre, "An empirical study of developers views on software reuse in statoil asa," in *ACM/IEEE international symposium on Empirical software engineering*, 2006.
- [80] D. Lucrédio, K. dos Santos Brito, A. Alvaro, V. C. Garcia, E. S. de Almeida, R. P. de Mattos Fortes, and S. L. Meira, "Software reuse: The brazilian industry scenario," *Journal of Systems and Software*, vol. 81, no. 6, pp. 996 – 1013, 2008, agile Product Line Engineering. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0164121207002221>
- [81] J. Varnell-Sarjeant, A. A. Andrews, and A. Stefik, "Comparing reuse strategies: An empirical evaluation of developer views," in *COMPSACW 2014*. IEEE, 2014, pp. 498–503.
- [82] D. C. Rine, "Success factors for software reuse that are applicable across domains and businesses," in *Proceedings of the 1997 ACM symposium on Applied computing*. ACM, 1997, pp. 182–186.
- [83] Marcus A. Rothenberger and Kevin J. Dooley and Uday R. Kulkarni and Nader Nada, "Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices," in *IEEE Transactions on Software Engineering*, 2003.
- [84] R. Joos, "Software reuse at motorola," in *IEEE Software*, vol. 11, no. 5, 1994.
- [85] N. Nada and D. C. Rine, "Three empirical evaluations of a software reuse reference model," *Annals of Software Engineering*, vol. 10, no. 1-4, pp. 225–259, 2000.
- [86] T. A. Standish, "An essay on software reuse," *Software Engineering, IEEE Transactions on*, no. 5, pp. 494–497, 1984.
- [87] J. Greenfield, K. Short, S. Cook, and S. Kent, *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.
- [88] D. Garlan, R. Allen, and J. Ockerbloom, "Architectural mismatch or why it's hard to build systems out of existing parts," in *Proceedings of the 17th International Conference on Software Engineering*, ser. ICSE '95. New York, NY, USA: ACM, 1995, pp. 179–185. [Online]. Available: <http://doi.acm.org/10.1145/225014.225031>
- [89] H. Koziolok, T. Goldschmidt, T. de Gooijer, D. Domis, S. Sehestedt, T. Gamer, and M. Aleksy, "Assessing software product line potential: An exploratory industrial case study," *Empirical Software Engineering*, 2015.
- [90] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *Software, IEEE*, vol. 27, no. 4, pp. 80–86, 2010.

- [91] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel, "Identifier-based context-dependent api method recommendation," in *CSMR'12*, 2012.
- [92] M. Torchiano and M. Morisio, "Overlooked aspects of cots-based development," *Software, IEEE*, vol. 21, no. 2, pp. 88–93, 2004.
- [93] J. R. Cordy, "Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation," in *Proceedings of the IEEE 11th International Workshop on Program Comprehension*, 2003.
- [94] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 664–675.
- [95] J. Dinkelacker, P. Garg, D. Nelson, and R. Miller, "Progressive Open Source," in *ICSE' 02*, 2002.
- [96] R. Goldman and R. P. Gabriel, *Innovation happens elsewhere: Open source as business strategy*. Morgan Kaufmann, 2005.
- [97] V. K. Gurbani, A. Garvert, and J. D. Herbsleb, "A case study of a corporate open source development model," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 472–481.
- [98] H. Rehesaar, "Capability assessment for introducing component reuse," in *ICSR'11 - Top Productivity through Software Reuse*. Springer, 2011, pp. 87–101.
- [99] J. Hopkins, "Component primer," *Communications of the ACM*, vol. 43, no. 10, pp. 27–30, 2000.
- [100] R. Land, D. Sundmark, F. Lueders, I. Krasteva, and A. Causevic, "Reuse with software components - a survey of industrial state of practice," in *ICSR'09*, 2009.
- [101] M. Morisio, C. B. Seaman, V. R. Basili, A. T. Parra, S. E. Kraft, and S. E. Condon, "Cots-based software development: Processes and open issues," *Journal of Systems and Software*, vol. 61, no. 3, pp. 189–199, 2002.
- [102] C. Ayala, Ø. Hauge, R. Conradi, X. Franch, and J. Li, "Selection of third party software in off-the-shelf-based software development—an interview study with industrial practitioners," *Journal of Systems and Software*, vol. 84, no. 4, pp. 620–637, 2011.
- [103] T. Vale, I. Crnkovic, E. S. de Almeida, P. A. da Mota Silveira Neto, Y. C. Cavalcanti, and S. R. de Lemos Meira, "Twenty-eight years of component-based software engineering," *Journal of Systems and Software*, vol. 111, pp. 128 – 148, 2016. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0164121215002095>
- [104] A. Göb, "Soa und softwarequalität," Dissertation, Technische Universität München, 2013.
- [105] S. Jones, "Toward an acceptable definition of service [service-oriented architecture]," *IEEE Software*, 2005.
- [106] O. Vogel, I. Arnold, A. Chughtai, E. Ihler, T. Kehrer, U. Mehlig, and U. Zdun, *Software-Architektur: Grundlagen - Konzepte - Praxis*. Spektrum Akademischer Verlag Heidelberg, 2009.
- [107] C. W. Krueger, "Software product line reuse in practice," in *Application-Specific Systems and Software Engineering Technology, 2000. Proceedings. 3rd IEEE Symposium on*. IEEE, 2000, pp. 117–118.
- [108] I. Nordberg, M.E., "Managing code ownership," *Software, IEEE*, vol. 20, no. 2, pp. 26–33, Mar 2003.

- [109] K. Charmaz, *Constructing grounded theory: A practical guide through qualitative analysis*. Pine Forge Press, 2006.
- [110] V. Basili, G. Caldiera, and H. Rombach, “The Goal Question Metric Approach,” *Encyclopedia of Software Engineering*, vol. 1, 1994.
- [111] T. Gorschek, C. Wohlin, P. Carre, and S. Larsson, “A model for technology transfer in practice,” *Software, IEEE*, vol. 23, no. 6, pp. 88–95, Nov 2006.
- [112] E. W. Dijkstra, “On the role of scientific thought,” in *Selected Writings on Computing: A personal Perspective*, ser. Texts and Monographs in Computer Science, 1982. [Online]. Available: http://dx.doi.org/10.1007/978-1-4612-5695-3_12
- [113] M. T. Baldassarre, D. Caivano, and G. Visaggio, “Empirical studies for innovation dissemination: Ten years of experience,” in *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, ser. EASE '13. New York, NY, USA: ACM, 2013, pp. 144–152. [Online]. Available: <http://doi.acm.org/10.1145/2460999.2461020>
- [114] A. Sandberg, L. Pareto, and T. Arts, “Agile collaborative research: Action principles for industry-academia collaboration,” *Software, IEEE*, vol. 28, no. 4, pp. 74–83, July 2011.
- [115] C. Wohlin, A. Aurum, L. Angelis, L. Phillips, Y. Dittrich, T. Gorschek, H. Grahn, K. Henningsson, S. Kagstrom, G. Low, P. Rovegard, P. Tomaszewski, C. van Toorn, and J. Winter, “The success factors powering industry-academia collaboration,” *IEEE Softw.*, vol. 29, no. 2, pp. 67–73, Mar. 2012. [Online]. Available: <http://dx.doi.org/10.1109/MS.2011.92>
- [116] P. Rodríguez, P. Kuvaja, and M. Oivo, “Lessons learned on applying design science for bridging the collaboration gap between industry and academia in empirical software engineering,” in *Proceedings of the 2Nd International Workshop on Conducting Empirical Studies in Industry*, ser. CESI 2014. New York, NY, USA: ACM, 2014, pp. 9–14. [Online]. Available: <http://doi.acm.org/10.1145/2593690.2593694>
- [117] S. Hallsteinsen and M. Paci, *Experiences in software evolution and reuse: twelve real world projects*. Springer Science & Business Media, 1997, vol. 1.
- [118] D. Cruzes, T. Dybå, P. Runeson, and M. Höst, “Case studies synthesis: a thematic, cross-case, and narrative synthesis worked example,” *Empirical Software Engineering*, pp. 1–32, 2014. [Online]. Available: <http://dx.doi.org/10.1007/s10664-014-9326-8>
- [119] A. Agresti, *An introduction to categorical data analysis*. New York: Wiley, 1996. [Online]. Available: http://www.worldcat.org/search?qt=worldcat_org_all&q=0471113387
- [120] V. C. Garcia, D. Lucrédio, A. Alvaro, E. S. De Almeida, R. P. de Mattos Fortes, and S. R. de Lemos Meira, “Towards a maturity model for a reuse incremental adoption.” in *SBCARS*. Citeseer, 2007, pp. 61–74.
- [121] L. Heinemann, “Facilitating reuse in model-based development with context-dependent model element recommendations,” in *Recommendation Systems for Software Engineering (RSSE), 2012 Third International Workshop on*, June 2012, pp. 16–20.
- [122] M. Luckey, A. Baumann, D. Méndez, and S. Wagner, “Reusing security requirements using an extended quality model,” in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Secure Systems*. ACM, 2010, pp. 1–7.
- [123] E. Murphy-Hill and G. C. Murphy, “Recommendation Delivery,” in *Recommendation Systems in Software Engineering*. Springer, 2014, ch. 9, pp. 223–242.

- [124] S. Proksch, V. Bauer, and G. C. Murphy, "How to build a recommendation system for software engineering," in *Software Engineering*. Springer, 2015, pp. 1–42.
- [125] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *SCHOOL OF COMPUTING TR 2007-541, QUEEN'S UNIVERSITY*, vol. 115, 2007.
- [126] T. Mende, F. Beckwermert, R. Koschke, and G. Meier, "Supporting the grow-and-prune model in software product lines evolution using clone detection," in *CSMR 2008*, 2008.
- [127] A. Marcus and J. I. Maletic, "Identification of High-Level Concept Clones in Source Code," in *ASE*, 2001.
- [128] L. Heinemann, B. Hummel, and D. Steidl, "Teamscale: Software quality control in real-time," in *Proceedings of the 36th ACM/IEEE International Conference on Software Engineering (ICSE'14)*, 2014.
- [129] J. K.S and R. Vasantha, "A new capability maturity model for reuse based software development process," *IACSIT International Journal of Engineering and Technology*, vol. 2, no. 1, 2010.
- [130] P. Koltun and A. Hudson, "A reuse maturity model," in *44th Workshop on Institutionalizing Software Reuse*, 1991.
- [131] T. Davis, "The reuse capability model: a basis for improving an organization's reuse capability." in *Second International Workshop on Software Reusability*, 1993.
- [132] K. K. Mandava, B. Ravi Kiran *et al.*, "A systematic mapping study on value of reuse," *International Journal of Computer Applications*, vol. 34, no. 4, pp. 37–44, 2011.
- [133] *Aligning Organizations Through Measurement*. Springer, 2014.
- [134] M. Höst, K.-J. Stol, and A. Oruđević-Alagić, *Software Project Management in a Changing World*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, ch. Inner Source Project Management, pp. 343–369. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-55035-5_14
- [135] V. K. Gurbani, A. Garvert, and J. D. Herbsleb, "Managing a corporate open source software asset," *Communications of the ACM*, vol. 53, no. 2, pp. 155–159, 2010.
- [136] R. Prieto-Diaz, "Making software reuse work: an implementation model," *ACM SIGSOFT Software Engineering Notes*, vol. 16, no. 3, pp. 61–68, 1991.
- [137] D. Kawrykow and M. P. Robillard, "Improving api usage through automatic detection of redundant code," in *ASE' 09*.
- [138] E. Jürgens, F. Deissenboeck, and B. Hummel, "Code Similarities Beyond Copy & Paste," in *CSMR*, 2010.
- [139] R. Al-Ekram, C. Kapser, R. Holt, and M. Godfrey, "Cloning by Accident: An Empirical Study of Source Code Cloning Across Software Systems," in *ISESE*, 2005.
- [140] D. Ratiu and J. Jürgens, "Evaluating the reference and representation of domain concepts in apis," in *ICPC*, 2008, pp. 242–247.
- [141] G. Cousineau and P. Enjalbert, "Program Equivalence and Provability," in *MFCS*, 1979.
- [142] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," in *ISSTA*, 2009. [Online]. Available: <http://doi.acm.org/10.1145/1572272.1572283>
- [143] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *ICSE' 08*.

- [144] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," 1990.
- [145] S. Chatterjee, S. Juvekar, and K. Sen, "Sniff: A search engine for java using free-form queries," in *FASE*, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-00593-0_26
- [146] C. McMillan, M. Grechanik, and D. Poshyvanyk, "Detecting similar software applications," in *ICSE*, 2012, pp. 364–374.
- [147] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *Software Engineering, IEEE Transactions on*, vol. 38, no. 5, pp. 1069–1087, 2012.
- [148] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *ICSE*, 2012.
- [149] C. Teyton, J.-R. Falleri, and X. Blanc, "Automatic discovery of function mappings between similar libraries," in *WCRE*, 2013.
- [150] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *WCRE*, 2013, pp. 182–191.
- [151] E. Duala-Ekoko and M. P. Robillard, "Using structure-based recommendations to facilitate discoverability in apis," in *ECOOP*, 2011.
- [152] O. Hummel, W. Janjic, and C. Atkinson, "Code conjurer: Pulling reusable software out of thin air," *IEEE Software*, vol. 25, no. 5, pp. 45–52, 2008.
- [153] F. McCarey, M. Ó. Cinnéide, and N. Kushmerick, "Rascal: A recommender agent for agile reuse," *Artif. Intell. Rev.*, vol. 24, no. 3-4, pp. 253–276, 2005.
- [154] Y. Ye and G. Fischer, "Information delivery in support of learning reusable software components on demand," in *IUI*, 2002, pp. 159–166.
- [155] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *ESEC/SIGSOFT FSE*, 2009.
- [156] R. Robbes and M. Lanza, "Improving code completion with program history," *Autom. Softw. Eng.*, vol. 17, no. 2, pp. 181–212, 2010.
- [157] A. Marcus and S. Haiduc, "Text retrieval approaches for concept location in source code," in *ISSSE'11*.
- [158] G. Salton and C. Buckley, "Term-weighting approaches in automatic text retrieval."
- [159] B. Basset and N. A. Kraft, "Structural Information Based Term Weighting in Text Retrieval for Feature Location," in *ICPC'13*.
- [160] B. Hauptmann, V. Bauer, and M. Junker, "Using Edge Bundle Views for Clone Visualization," in *IWSC'12*, 2012.
- [161] F. Gauthier, T. Lavoie, and E. Merlo, "Uncovering access control weaknesses and flaws with security-discordant software clones," in *Proceedings of the 29th Annual Computer Security Applications Conference*. ACM, 2013, pp. 209–218.
- [162] R. Koschke, *Survey of research on software clones*. Internat. Begegnungs-und Forschungszentrum für Informatik, 2007.

- [163] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, 2009.
- [164] N. Gode and R. Koschke, "Incremental Clone Detection," in *CSMR*. IEEE, 2009.
- [165] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt, "Index-based Code Clone Detection: Incremental, Distributed, Scalable," in *ICSM*. IEEE, 2010.
- [166] V. Bauer, L. Heinemann, B. Hummel, E. Juergens, and M. Conradt, "A Framework for Incremental Quality Analysis of Large Software Systems," in *ICSM*. IEEE, 2012.
- [167] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and Evaluation of Clone Detection Tools," *TSE*, 2007.
- [168] E. Juergens, F. Deissenboeck, and B. Hummel, "Clone Detection Beyond Copy & Paste," in *IWSC*, 2009.
- [169] D. Steidl and S. Eder, "Prioritizing Maintainability Defects Based on Refactoring Recommendations," in *ICPC*, 2014. [Online]. Available: <http://doi.acm.org/10.1145/2597008.2597805>
- [170] H. A. Basit and S. Jarzabek, "A Data Mining Approach for Detecting Higher-level Clones in Software," *IEEE TSE*, 2007.
- [171] R. Tairas and J. Gray, "An Information Retrieval Process to Aid in the Analysis of Code Clones," *Empirical Software Engineering*, no. 1, 2009.
- [172] B. Caprile and P. Tonella, "Restructuring Program Identifier Names," in *ICSM*, 2000.
- [173] F. Deissenboeck and M. Pizka, "Concise and consistent naming," *Software Quality Journal*, vol. 14, no. 3, pp. 261–282, 2006. [Online]. Available: <http://dx.doi.org/10.1007/s11219-006-9219-1>
- [174] S. Haiduc and A. Marcus, "On the Use of Domain Terms in Source Code," in *ICPC*, 2008.
- [175] Pivotal Software, Inc., "The spring io platform."
- [176] V. Bauer, T. Voelke, and S. Eder, "COMPARING TF-IDF AND LSI AS IR TECHNIQUE IN AN APPROACH FOR DETECTING SEMANTIC RE-IMPLEMENTATIONS IN SOURCE CODE," Technische Universität München, Tech. Rep., 2015.
- [177] E. Juergens, F. Deissenboeck, and B. Hummel, "CloneDetective - A workbench for clone detection research," in *ICSE*, May 2009.
- [178] S. Eder, H. Femmer, B. Hauptmann, and M. Junker, "Configuring Latent Semantic Indexing for Requirements Tracing," *RET*, 2015.
- [179] E. Juergens, "Why and how to control cloning in software artifacts — doctoral dissertation," *Technische Universität München*, 2011.
- [180] J. Krinke, N. Gold, Y. Jia, and D. Binkley, "Cloning and copying between gnome projects," in *7th IEEE Working Conference on Mining Software Repositories*, 2010.
- [181] D. Faust and C. Verhoef, "Software product line migration and deployment," *Software Practice and Experience*, 2003.
- [182] N. Schwarz, M. Lungu, and R. Robbes, "On how often code is cloned across repositories," in *34th International Conference on Software Engineering*, 2012.
- [183] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, and J. Girard, "An activity-based quality model for maintainability," in *ICSM'07*, 2007.

-
- [184] B. Kitchenham, S. Pfleeger, and N. Fenton, "Towards a framework for software measurement validation," *Software Engineering, IEEE Transactions on*, vol. 21, no. 12, pp. 929–944, 1995.
- [185] F. Deissenboeck, E. Juergens, K. Lochmann, and S. Wagner, "Software quality models: Purposes, usage scenarios and requirements," in *WOSQ '09*, 2009.
- [186] G. Kotonya and J. Hutchinson, "Analysing the impact of change in COTS-based systems," *COTS-Based Software Systems*, pp. 212–222, 2005.
- [187] R. Kazman, L. Bass, M. Webb, and G. Abowd, "SAAM: A method for analyzing the properties of software architectures," in *ICSE'94*, 1994.
- [188] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson, and J. Carriere, "The architecture tradeoff analysis method," in *ICECCS'98*, 1998.
- [189] S. Wagner, K. Lochmann, L. Heinemann, M. Kläs, A. Trendowicz, R. Plösch, A. Seidl, A. Goeb, and J. Streit, "The quamoco product quality modelling and assessment approach," in *ICSE'12*, 2012.
- [190] K. Mordal-Manet, F. Balmas, S. Denier, S. Ducasse, H. Wertz, J. Laval, F. Bellingard, and P. Vaillegues, "The squal model—A practice-based industrial quality model," in *ICSM'09*, 2009.
- [191] S. Raemaekers, A. van Deursen, and J. Visser, "Measuring software library stability through historical version analysis," in *SQM*, 2012.