University of Magdeburg

School of Computer Science

Master's Thesis

A Cost Estimation Model for the Extractive Software-Product-Line Approach

Author:

Jacob Krüger

February 16, 2016

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
Dipl.-Inf. Wolfram Fenske
M.Sc. Jens Meinicke
Otto-von-Guericke-University Magdeburg
Department of Technical and Business Information Systems

Prof. Dr.-Ing. Thomas Leich
METOP GmbH
Affiliated Institute to the Otto-von-Guericke-University Magdeburg

Krüger, Jacob:
A Cost Estimation Model for the Extractive Software-Product-Line Approach
Master's Thesis, University of Magdeburg, 2016.

Abstract

Companies are more and more forced to customize their software products for different customers. In practice they often clone an existing system and adapt it to the customer's requirements. At some point, a set of similar but separated variants emerged. In such scenarios software product lines promise benefits, for example, reduced maintenance effort, improved quality, and customizability. In most cases, the extractive approach is applied, with which legacy systems are re-engineered to implement re-usability. However, introducing new development processes into a company is risky and might not pay off. Thus, cost estimations are applied to predict whether and when the change is beneficial. Existing cost models for software product lines focus on development from scratch. As a result, they are not suited for the extractive approach.

In this thesis we introduce a semi-automatic cost estimation approach for the extraction of software product lines. We use existing cost models to derive economic descriptions that consider and explain benefits of legacy systems. They support experts during their estimation. From those descriptions we derive a calculation-based cost model for the extractive approach. We investigate possibilities to automatically extract information from legacy systems to support our cost model. This reduces the analysis effort and provides reliable data. In our evaluation we use forks of an open-source project to describe a fictional business case. We conduct interviews with experts and compare the estimations with case studies to validate our cost model. We show that our approach provides a practical useful methodology and calculates reasonable results.

Acknowledgements

I would like to thank my advisors Prof. Gunter Saake, Prof. Thomas Leich, Dipl.-Inf. Wolfram Fenske and M.Sc. Jens Meinicke for their support during this thesis. Their constructive input helped me to considerably improve quality of content and writing.

My thanks to everyone at the METOP GmbH for their support during the last years. In particular, I thank Stephan Dassow and Andy Kenner for numerous productive discussions on my thesis.

Finally, I would like to thank my family and friends for encouragement and moral support.

Contents

List of Figures **ix**

List of Tables **xi**

1 Introduction **1**
1.1 Contribution 3
1.2 Outline 4

2 Background **7**
2.1 Clone-and-Own 7
2.2 Software Product Lines 8
2.3 Software Cost Estimation 12
2.4 Summary 15

3 An Overview of Cost Models for Software Product Lines **17**
3.1 Cost Model Requirements 17
3.2 Combination of Cost Estimations 18
3.3 Overview 20
3.3.1 Categorization 22
3.3.2 Discussion 24
3.3.3 Threats to Validity 26
3.4 Summary 26

4 Economic Descriptions for the Extractive Approach **29**
4.1 SIMPLE 29
4.2 Economic Descriptions 31
4.2.1 Costs to Develop Assets 34
4.2.2 Costs for New Products 35
4.2.3 Costs for Maintenance and Evolution 36
4.2.4 Organizational Costs 37
4.3 The Cost Factors in the Cost Curve 38
4.4 Summary 42

5 A Semi-Automatic Cost Model **43**
5.1 COPLIMO 43

5.2 Adapted Calculations 49
5.2.1 Costs to Develop Assets 50
5.2.2 Costs for New Products 52
5.2.3 Costs for Maintenance and Evolution 52
5.3 Cost Estimation and the Software-Product-Line Development Process . 53
5.4 Automating the Cost Estimation 55
5.4.1 Extracting Information from Legacy Systems 55
5.4.2 Extraction Scenarios 56
5.4.3 Computing the Shared Code 58
5.5 Summary 60

6 Evaluation **61**
6.1 Evaluation of Cost Models 61
6.2 Marlin Forks 64
6.3 Business Case 66
6.4 Cost Model Estimations 68
6.4.1 Stand-Alone Development 69
6.4.2 Proactive Software-Product-Line Development 70
6.4.3 Extractive Software-Product-Line Development 72
6.4.4 Clone-and-Own Development 74
6.5 Discussion 75
6.5.1 Interpretation and Hypotheses 76
6.5.2 Interviews 78
6.5.3 Comparison with Case Studies 82
6.6 Threats to Validity 84
6.7 Summary 85

7 Conclusion **87**
7.1 Contributions 87
7.2 Future Work 89

A Appendix **91**
A.1 Interview Protocol Expert A 91
A.2 Interview Protocol Expert B 92

Bibliography **95**

List of Figures

1.1 Software-product-line adoption strategies 2

2.1 Clone-and-own development 8
2.2 Software-product-line development process 9
2.3 General cost estimation process 13
2.4 Improvement of software cost estimation accuracy 14

4.1 Hypothetical cost curves for proactive software-product-line development 32
4.2 Cost curves of the proactive software-product-line approach with risk . 33
4.3 Impact of experience on the average development costs 33
4.4 Hypothetical cost curves for extractive software-product-line development 34
4.5 Calculating the costs for new products 36
4.6 Fixed and variable costs for the extractive approach 40
4.7 Simplified dependency between fixed and variable costs 40
4.8 Possible scenarios for varying investments for the extractive approach . 41

5.1 Calculating the adoption costs 50
5.2 Cost estimation process for the extractive approach 54
5.3 Venn-diagram of possible overlaps between four products 57
5.4 Overlapping code clones 59

6.1 Venn-diagrams of intersections between Marlin forks 66
6.2 Effort for proactive software-product-line and stand-alone development 76
6.3 Comparison of the development and life-cycle efforts for the scenarios . 77
6.4 Development effort comparison 78
6.5 Maintenance effort comparison 79

List of Tables

3.1 Cost models for software product lines 22
3.2 Comparison of software-product-line cost models 23

5.1 Coherence between SIMPLE cost functions and COPLIMO parameters 48
5.2 Overview of parameters used for the cost estimation 49

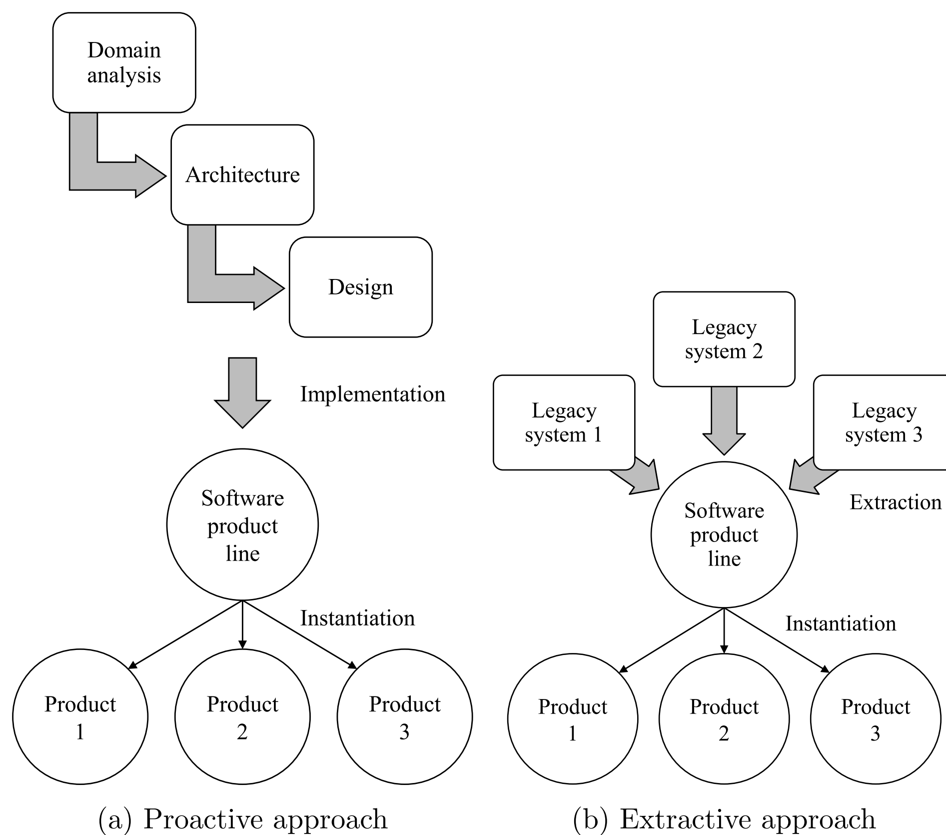
6.1 Questionnaire for the experts 64
6.2 Overview of the used Marlin forks 65
6.3 Sizes of the intersections between the Marlin forks 65
6.4 Ratings of the COPLIMO parameters for the business case 69
6.5 Effort for stand-alone development of the variants 70
6.6 Ratios of unique and reusable code for the products 71
6.7 Cost estimations for the remaining products 72
6.8 Estimated effort to extract the intersections 73
6.9 Estimated effort to extract the variants 74
6.10 Summary of the estimated efforts in person-months 75
6.11 Interviewed experts 79
6.12 Selection of case study results 82

1. Introduction

Customers more and more demand customized software products [Pohl et al. 2005, pp. 4-7; Fischer et al. 2014]. Thus, software companies are forced to set up multiple variants of the same product that are adapted for different customers [Schmid and Verlage 2002]. This led to the adoption of product lines, a systematic approach to manage and reuse similar products, for software engineering [Krueger 2002a; Pohl et al. 2005, p. 7; Apel et al. 2013, p. 7]. Using software product lines instead of stand-alone systems promises multiple benefits besides mass customization, such as, reduced development effort for new products, faster time-to-market, decreased maintenance effort, increased product quality, and improved cost estimations [Knauber et al. 2002; Pohl et al. 2005, pp. 9-13; Clements and Northrop 2006, p. 17; Apel et al. 2013, pp. 8-10; Martinez et al. 2015].

Companies often face time limitations, fear the additional risks and costs of systematic reuse, are unaware of suited approaches, or start with a single innovative product and, thus, still use single-system development even if product lines would be beneficial [Knauber et al. 2000; Krueger 2002a; Schmid and Verlage 2002; Tang et al. 2010; Dubinsky et al. 2013]. Therefore, instead of starting their development with a software product line (the proactive approach), companies apply unsystematic reuse approaches, such as, clone-and-own, to derive customized variants for new customers [Knauber et al. 2000; Apel et al. 2013, p. 41; Dubinsky et al. 2013; Fischer et al. 2014; Martinez et al. 2015]. Those strategies often become more expensive in development, maintenance, and customization than software product lines as soon as the number of products increases and the same changes must be implemented in multiple variants [Rubin et al. 2012; Apel et al. 2013, p. 41]. Facing growing efforts, a company might consider to change to a systematic reuse technique [Apel et al. 2013, p. 41]. Thus, in industrial practice, re-engineering the existing variants into a software product line is more common than proactively developing one [Schmid and Verlage 2002; Pohl et al. 2005, p. 201; Duszynski et al. 2011; Berger et al. 2013; Koziolek et al. 2015].

For the introduction of software product lines, Krueger [2002a] defines several adoption strategies. We illustrate two of them in Figure 1.1. The *proactive approach* is used to develop a product line, instead of single systems, from the start. To do this, the domain is analyzed to derive an architecture and design for similarities and variability. Afterwards, the product line is implemented. This is often assumed as the most cost saving strategy but requires high up-front investments [Clements 2002; Schmid and Verlage 2002; Pohl et al. 2005, p. 9; Berger et al. 2013; Apel et al. 2013, p. 9]. In contrast, the *extractive approach* is based on existing products (referred to as legacy systems) [Fenske et al. 2014]. From those products similarities are identified and fragments are reused to develop a product line. For companies that use clone-and-own, the proactive approach implies that all existing products are developed anew. The more reasonable attempt is to extract a product line from the legacy systems. Still, this task requires effort and, thus, results in additional costs.



(a) Proactive approach (b) Extractive approach

Figure 1.1: Software-product-line adoption strategies based on Krueger [2002a]

Before introducing a software-product-line approach a company must estimate resulting costs and savings. Otherwise, it cannot justify that the paradigm change provides benefits [Martinez et al. 2015]. There are several cost estimation approaches such as algorithmic models or expert judgment [Boehm 1984]. Each approach has its advantages and drawbacks, which is why combinations and comparisons of multiple estimation strategies have been proposed [Boehm 1984; Jørgensen 2007; Jørgensen et al. 2009].

For software-product-line engineering multiple cost models exist [Khurum et al. 2008; Ali et al. 2009; Charles et al. 2011; Blandón et al. 2013; Heradio et al. 2013]. However, they focus on the proactive approach and rarely consider the reuse of legacy systems [Koziolek et al. 2015]. Thus, for the extractive approach it is only possible to use models that only describe cost factors but do not provide algorithms [Ali et al. 2009]. A company that wants to extract a product line from its legacy systems must rely on experts. Expert estimations are based on the individual intuition, knowledge, and experiences, which can lead to varying results [Jørgensen 2004, 2007; Jørgensen et al. 2009]. It is difficult to justify experts' predictions to customers or run multiple scenarios [Jørgensen et al. 2009]. In contrast, algorithmic cost models provide a comprehensible and replicable method.

Adapting existing cost models for the extractive approach is important to support companies in their decisions. Ali et al. [2009] state several attributes that must be considered, for instance, only a subset of estimation approaches regards the life-cycle and not only the adoption process. The number of models that define or are adoptable to changing use-case scenarios is highly limited. Additionally, as stated by Blandón et al. [2013], most approaches lack in documentation and tool support, wherefore they are difficult to apply in industrial practice.

1.1 Contribution

In industrial practice it is common to extract a software product line from legacy systems. The transition is less risky than proactive development, as artifacts can be reused and the products are already at the market. Still, success is not granted. Thus, an organization must consider the costs and savings of a software product line to make a reasonable decision. Cost models can be used to estimate the extraction effort and predict the break-even point of the investment. However, existing approaches focus on proactive development of software product lines. Especially, automated extraction of information from the legacy systems is not considered. A cost model that focuses on the extractive approach and provides solutions for the determination of parameters is missing.

Goal of this Thesis

Our goal for this thesis is to introduce a cost estimation approach for the extraction of software product lines from legacy systems. According to Boehm [1984], Jørgensen [2007], and Jørgensen et al. [2009], the combination and comparison of multiple cost predictions is more reliable than using only one of them. Thus, we want to support judgment- and calculation-based estimations. For this reason, we focus on three tasks:

1. We investigate and discuss benefits of legacy systems on the development costs of software product lines.

2. We propose a cost model that considers the benefits we identified.

3. We propose a solution to automate the extraction of information for our cost model from legacy systems.

Methodology

To achieve our goal, we apply the following methodology. We start by describing the use-case scenario and requirements for our approach. Then, we use a literature review to identify existing cost models for software product lines. Based on our requirements we derive attributes to determine models we can adopt for the extractive approach. We use the cost factors provided by the models and additional literature to discuss economic impacts of legacy systems on the re-engineering process towards product lines. From this, we derive how the cost factors of existing cost models can be adapted. We match the identified adaptions to the algorithm-based COPLIMO [Boehm et al. 2004]. From parameters that are considered within the models, we derive information that can be extracted from legacy systems to support the estimation process. In particular, we focus on automatic data collection approaches, which can reduce the analysis effort while increasing the accuracy of cost predictions. We apply our approach to a fictional business scenario with four variants of the open-source 3D-printer firmware *Marlin*[1]. We discuss our estimations within qualitative interviews with experts. Additionally, we compare the results from our scenario with benefits that are reported in industrial case studies of software-product-line economics.

Results

During the interpretation of our results we show that the estimations of our cost model are reasonable. With the responses of our interviews we further prove that our approach is usable for the industrial practice and provides acceptable results. However, we also identify shortcomings and possible extensions. Especially, we find additional parameters that a company might want to consider. We were not able to conduct or repeat a case study, but we compare our estimations with those reported from practice. This provides no evidence but clues that our cost model calculates reliable estimations. From those results, we conclude that we address our goal and provide a suited cost model for the extractive software-product-line approach.

1.2 Outline

The structure of this thesis is as follows.

In Chapter 2 we introduce the necessary background for understanding our work. We describe the concept of *clone-and-own* as an unsystematic reuse strategy. Next, we provide an overview on *software product line engineering* with focus on thee extractive approach. Finally, we introduce the basics of *software cost estimations*.

[1]MarlinFirmware/Marlin: https://github.com/MarlinFirmware/Marlin [08.02.2016]

Within Chapter 3 we describe requirements for our approach. We provide an overview of existing cost models and apply attributes that help to classify them. We use this classification to restrict the number of models and select those that are suited for our adoptions.

In Chapter 4 we first describe SIMPLE [Böckle et al. 2004; Clements et al. 2005], a framework-like cost estimation approach. Based on its cost factors, we discuss economics for the extractive approach. In particular, we focus on differences in contrast to the proactive approach.

We begin Chapter 5 with descriptions of COPLIMO [Boehm et al. 2004] and its relation to SIMPLE. We then derive a calculation based cost model that considers the economic benefits we presented in Chapter 4. Finally, we propose solutions to automatically extract information from legacy systems. This information can be used as input for our approach.

In Chapter 6 we evaluate our cost model with a fictional business case based on real-world software. We assess our estimations and discuss them with experts to show the usability of our approach.

We conclude this thesis in Chapter 7 and present topics for further research.

2. Background

In this chapter, we introduce the basic concepts that are necessary for the understanding of this thesis. We begin with a description of the *clone-and-own* approach in Section 2.1. Afterwards, we provide information on *software product lines* with focus on the extractive approach to software-product-line adoption and variability mining in Section 2.2. Finally, we explain in Section 2.3 possibilities and difficulties of *software cost estimations*.

2.1 Clone-and-Own

Software reuse is an approach of developing systems from already existing artifacts rather than from scratch [Krueger 1992]. With the *clone-and-own* approach a legacy system is cloned and modified to match new customer needs [Fenske et al. 2014; Stanciulescu et al. 2015]. We illustrate an according example in Figure 2.1. The initial requirements of the first client led to the development of a first product. For other customers, that have similar but also new requirements, a suited system is cloned and adapted. This reuse strategy is considered as a fast and simple way of using existing products to develop new variants [Rubin et al. 2012; Fischer et al. 2014]. After cloning, the systems are separated, which is why they can be individually customized and assigned to different development teams [Schmid and Verlage 2002; Yoshimura et al. 2006a; Rubin et al. 2012]. In conclusion, clone-and-own is an opportunistic and ad-hoc strategy to derive customized products from existing ones [Clements and Northrop 2006, p. 11; Martinez et al. 2015; Stanciulescu et al. 2015]. In industrial practice, software product lines are often not deployed in the beginning because companies face strict schedules or fear the high upfront investments [Krueger 2002b; Pohl et al. 2005, p. 9; Apel et al. 2013, p. 41; Dubinsky et al. 2013]. Instead, due to its benefits, cloning is often applied [Schmid and Verlage 2002; Rubin et al. 2012; Dubinsky et al. 2013; Apel et al. 2013, p. 41; Fischer et al. 2014].

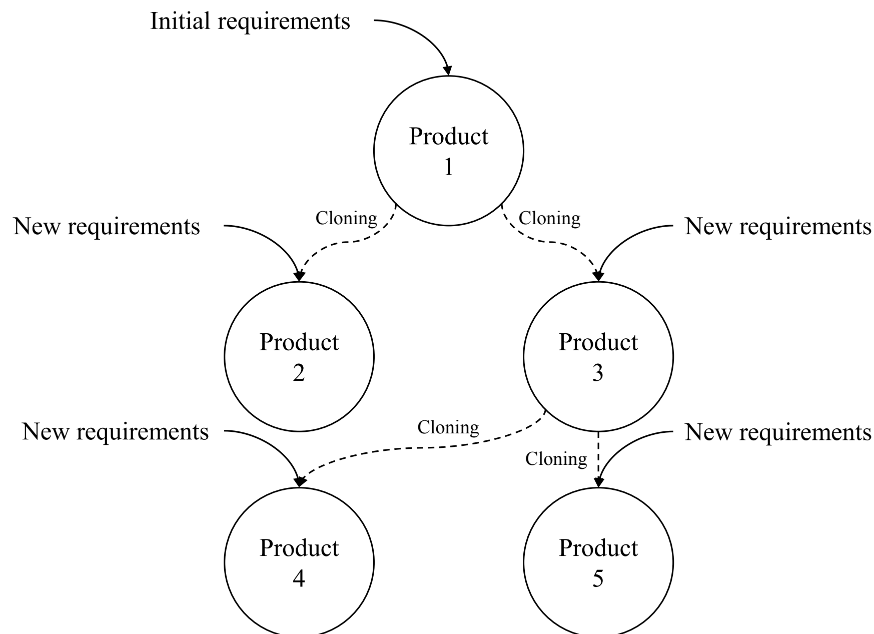


Figure 2.1: Clone-and-own development

2.2 Software Product Lines

Managing variants with clone-and-own is sometimes considered as a product-line approach [Krueger 2004; Dubinsky et al. 2013]. As we explained, the systems are developed ad-hoc and without re-usability. In contrast to this, Clements and Northrop [2006, pp. 12] emphasize that software product lines are designed for reuse and require only the maintenance of a single code base. This is a clear separation to cloning, in which the systems are developed without planning for re-usability [Fenske et al. 2014]. Moreover, cloned variants contain identical code that is not part of a common base.

A software product line describes a set of similar systems that share, and are developed from, common *assets* [Northrop 2002; Clements and Northrop 2006, p. 5; Apel et al. 2013, p. 8]. Assets are reusable artifacts that describe and implement a *feature* [Northrop 2002; Clements and Northrop 2006, p. 14]. Features describe functionalities that fulfill a requirement and can be selected and combined to build customized variants [Krueger 2006; Apel and Kästner 2009; Apel et al. 2013, p. 18]. Contrary to the clone-and-own approach, developing a software product line is separated in two tasks, which we illustrate in Figure 2.2 [Czarnecki and Eisenecker 2005, pp.20-21; Pohl et al. 2005, pp. 20-21; Apel et al. 2013, pp. 19-22]. *Domain engineering* describes the process of developing assets for the software product line. Thus, commonalities and variations between the variants must be defined and realized. The result is not a single system but a set of reusable artifacts. The goal of *application engineering* is to develop a customized variant for a customer. Therefore, the assets provided from the domain implementation are reused and combined into a product. The necessary flexibility to customize products is called *variability* [Pohl et al. 2005, p. 8].

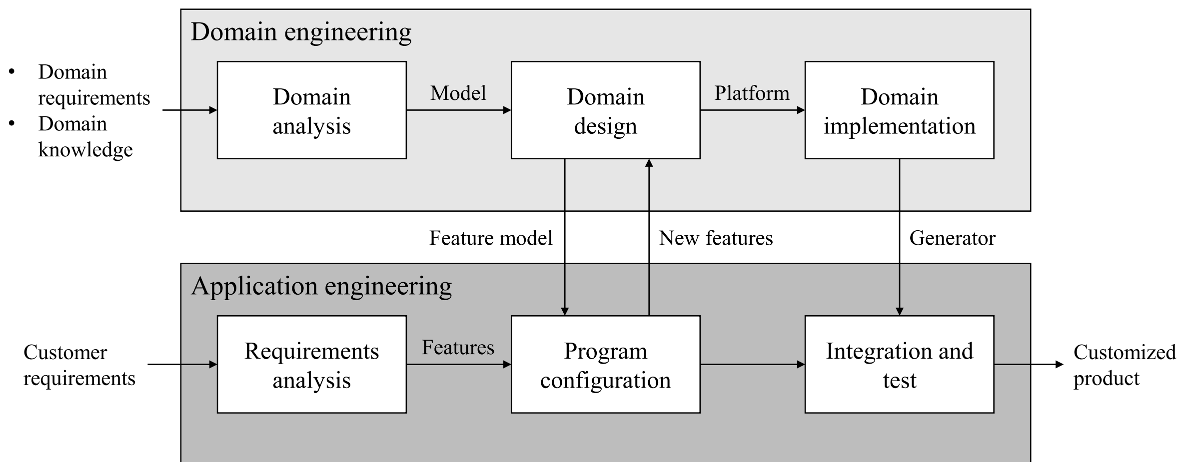


Figure 2.2: Software-product-line development process, adapted from Czarnecki and Eisenecker [2005, p. 21]

More and more customers expect products that suit their needs and, as a result, the importance of customization increases [Northrop 2002; Schmid and Verlage 2002; Pohl et al. 2005, p. 4]. With the growth of variants a company develops, the complexity and maintenance effort of unsystematic reuse approaches, such as clone-and-own, become more expensive. At this point, an organization might consider to migrate towards software-product-line engineering, a systematic reuse and customization strategy [Apel et al. 2013, p. 8; Martinez et al. 2015].

For the adoption of software product lines in a company, Krueger [2002a] defines three approaches:

- Proactive - A new product line is completely designed and implemented from scratch.
- Reactive - The software product line is not completely planned from the begging. Instead, some, or only one, variants are developed for systematic reuse. New features and assets are added later to extend and refine the scope.
- Extractive - A set of existing and similar legacy systems are used to develop a product line. Therefore, the common and different parts are re-engineered into reusable assets.

The proactive approach is considered as the optimal strategy to introduce a software product line [Clements 2002; Schmid and Verlage 2002; Berger et al. 2013]. However, the extractive approach is more common in industrial practice [Schmid and Verlage 2002; Pohl et al. 2005, p. 201; Duszynski et al. 2011; Koziolek et al. 2015]. Berger et al. [2013] showed in an empirical study that 50% of the participating companies used the extractive approach at least once while proactive development was applied by

35.3%. The reason for this is that companies mostly use cloning to derive customized variants (compare with Section 2.1). There are several disadvantages of the clone-and-own approach compared to software product lines. With the growth of variants, the amount of duplicated code increases. Especially, tracing and implementing changes in all reused clones is a challenging task [Yoshimura et al. 2006a; Rubin et al. 2012; Dubinsky et al. 2013]. Many activities, for instance, bug fixes, documentation, or testing, must be repeated several times [Dubinsky et al. 2013]. Thus, the maintenance of all variants becomes more complex. To overcome such problems, companies might consider to migrate towards a software product line.

For the extractive approach, the following tasks are essential [Krueger 2002a; Apel et al. 2013, pp. 41-42; Martinez et al. 2015]:

- Similarities and differences between the legacy systems are analyzed to identify features. This can be done manually or with the support of *feature location* techniques [Dit et al. 2013; Rubin and Chechik 2013; Assunção and Vergilio 2014]. After suited features are selected and their dependencies are described, it is possible to derive a *feature model* that displays the variability for the software product line [Czarnecki and Eisenecker 2005, p. 7].
- The identified features that shall be part of the product line are re-engineered into reusable assets. It may be possible to reuse parts of the legacy systems and extract their functionalities. Otherwise, the features must be implemented anew.
- The variability of the assets, described by the feature model, is implemented with a suited technique, for example, aspect-oriented programming [Kiczales et al. 1997].

Finally, an extracted software product line is developed. Instead of extracting all features at once, it is possible to only consider a subset of the legacy systems [Apel et al. 2013, p. 42]. For instance, older products or those that have few commonalities with others, can be excluded to reduce the re-engineering effort

Variability Mining

Variability mining provides semi-automatic tool support to analyze legacy systems [Kästner et al. 2014]. Thus, it can provide reliable data for our cost estimations. Variability mining covers the process of locating features in legacy systems to their implementation [Lozano 2011; Kästner et al. 2014]. *Feature location* has the goal to identify code fragments that implement a feature. For this task multiple approaches exist, see Dit et al. [2013], Rubin and Chechik [2013], or Assunção and Vergilio [2014] for recent surveys. However, current techniques have several shortcomings, which we are going to describe in the next paragraphs.

First, feature location is a challenging problem even for a single product [Kästner et al. 2014]. Thus, few approaches work on multiple legacy systems [Duszynski et al. 2011].

For the extractive approach this is problematic. It is necessary to analyze several variants and find their commonalities. Adopting a single product into a software product line provides no benefit. Therefore, we only consider feature location for multiple legacy systems. Some examples for such techniques are the approaches by Yinxing [2012], Ziadi et al. [2012], or Martinez et al. [2015].

Second, the number of implemented techniques is additionally limited [Dit et al. 2013; Assunção and Vergilio 2014]. Moreover, tools only work on specific programming languages, or do not report code statements but documentation or bug artifacts [Dit et al. 2013; Assunção and Vergilio 2014]. However, to estimate costs it is necessary to determine the sizes of commonalities. Some approaches require additional input information, such as design models, or do not even work on code level [Dit et al. 2013; Assunção and Vergilio 2014]. This makes it difficult to apply them on unfamiliar systems and also provides not the size of commonalities.

Finally, tools provide only semi-automated support due to the complexity of feature location [Biggerstaff et al. 1993; Kästner et al. 2014]. They suggest possible candidates but it still requires expertise and knowledge about the legacy systems to derive features [Assunção and Vergilio 2014; Kästner et al. 2014]. Especially, for multiple legacy systems the results must be matched. Only this way the commonalities between them can be identified and separated.

Code Clone Detection

Code clone detection can be used to automatically locate reusable artifacts in multiple legacy systems [Duszynski et al. 2011; Yinxing 2012]. The number of available approaches and implementations for code clone detection is higher than for feature location and variability mining [Bellon et al. 2007; Roy et al. 2009]. Still, the tools differ in multiple facets, for example, availability, language support, or granularity of the comparison [Roy et al. 2009]. Furthermore, the approaches detect different types of code clones. Therefore, the results of several tools for the same systems might differ. Roy et al. [2009], for example, name four code clone types based on their reviewed literature:

1. Identical code copies that only differ in white space or comments.
2. Additional changes of identifiers, literals, or types.
3. Further variations within the clones, such as, added, removed, or modified statements.
4. The code fragments are implemented differently but perform the same computation.

With the help of an appropriate code-clone-detection tool, similar code between variants can be determined automatically. Those commonalities indicate possible features.

Additionally, their size can be extracted. Thus, *software cost estimations* for software product lines can be supported. It is possible to automatically provide more accurate data about the proportion of reusable assets and unique artifacts.

2.3 Software Cost Estimation

Software cost estimation describes the process in which the costs to develop or enhance a software system are predicted [Leung and Fan 2002]. Especially, the benefits, risks, and costs, for a projects are compared. The results of cost estimations are important to evaluate different development approaches, such as, software-product-line or stand-alone engineering, and decide for one [Böckle et al. 2002; Yoshimura et al. 2006a; Dubinsky et al. 2013; Koziolek et al. 2015]. During the project life-cycle the estimates can be used for the following tasks [Boehm 1984; Boehm et al. 2000a; Leung and Fan 2002; Jørgensen 2007]:

- Before a project it is possible to estimate whether the promised benefits are worth the effort. This calculation can also be done within the project's duration to decide whether it needs to be stopped or adapted.
- Projects and tasks in a project can be prioritized to determine their importance for the organization.
- It is possible to assign resources more accurately to a project and its tasks.
- The effort that results from changes during the project can be estimated.
- The estimations can be compared with the actual costs to check if the project proceeds as planned.

There are a number of different approaches to estimate the costs of software development. Each of them has its own advantages and disadvantages that often complement each other, wherefore they should be combined and compared [Boehm 1984; Jørgensen 2007; Jørgensen et al. 2009]. According to Boehm [1984], acceptable cost estimation methods are:

1. *Algorithmic models* support the estimation with calculations based on parameters that represent cost drivers.
2. *Expert judgment* is based on human expertise and experiences to estimate the costs for a project.
3. *Analogies* use existing data of similar completed projects for the cost estimation.
4. *Top-down* approaches estimate the costs for the whole project and afterwards distributes them among individual components.

5. *Bottom-up* strategies first estimate the costs for each component individually and aggregate the results to calculate the overall effort.

As described by Heemstra [1992] and illustrated in Figure 2.3, cost estimations follow a general structure. First, in the *sizing stage*, the size of the new product is estimated. This can be done with different metrics, for example, lines of code. In the following *productivity stage*, an effort estimation is derived from the assumed size. With cost drivers, specifics of the software or development approach are considered. Afterwards, the predicted costs and correlating resources are distributed among project phases. Those phases are, for example, requirements analysis, product design, and implementation. Finally, *sensitivity and risk analysis* are applied to evaluate uncertainties of the projects and the reliability of the estimations. Based on historical data, the whole process can be *validated and calibrated*. Algorithmic cost models support the sizing and productivity stage, while the remaining tasks must be done manually.

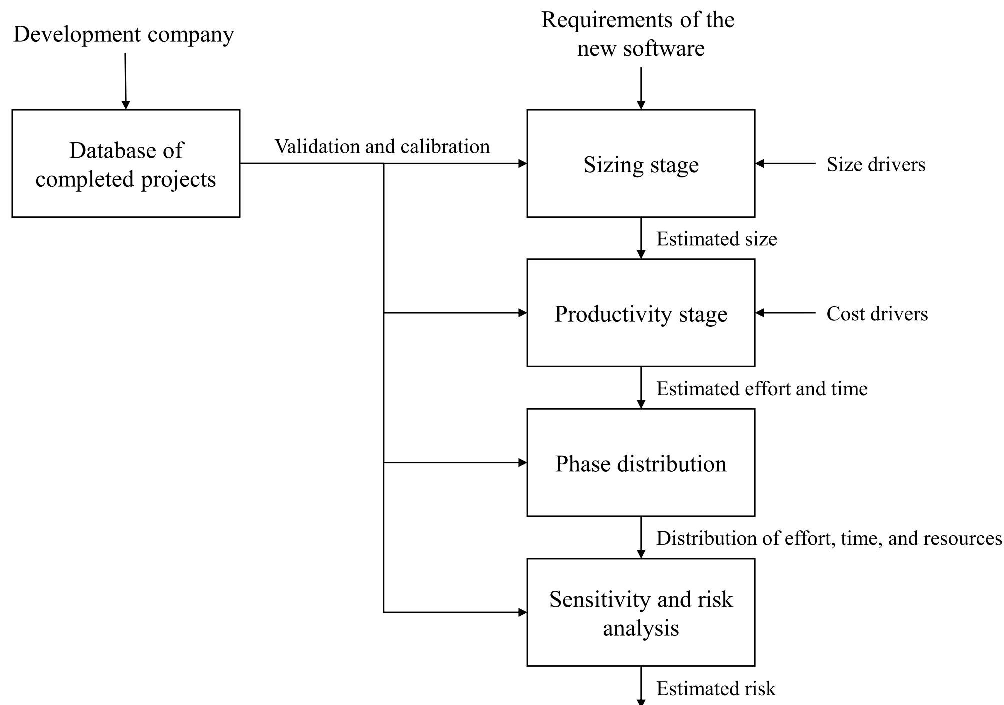


Figure 2.3: General cost estimation process, adapted from Heemstra [1992]

Limitations of Software Cost Estimations

Estimating the effort for a project is an important, but also challenging, task in software development. Following, we briefly describe some limitations for cost estimations, that can lead to distortions.

First, the greatest challenge is the *time-point and uncertainty* of the prediction. The estimation shall help to decide whether a project will be beneficial for an organization.

Therefore, the prediction is often necessary before the development and specification start [Heemstra 1992]. But at this point, a lot of important information, for example, concrete approaches, sizes, or requirements, are missing. As a result, the reliability of the estimation decreases. The corresponding lack of knowledge represents uncertainties, which can only be clarified during the development process. This results in better estimations during the project life-cycle, as illustrated in Figure 2.4 [Boehm 1984]. After the implementation the costs can be accurately determined. In contrast, the estimations during the conception can heavily vary from the real value.

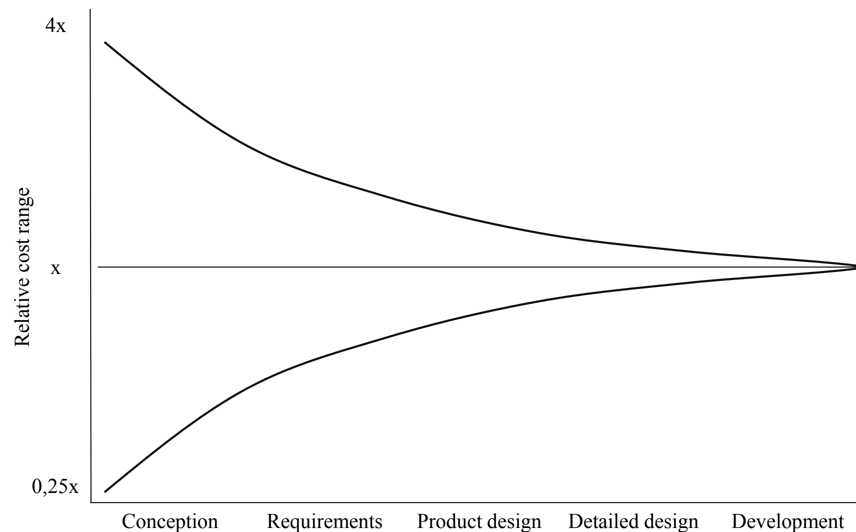


Figure 2.4: Improvement of software cost estimation accuracy during project phases, adapted from Boehm [1984]

Second, because of differences in the way software is developed, models have to be more specific and cannot address all domains [Heemstra 1992; Boehm et al. 2000a]. For example, different approaches could be necessary for object-oriented and procedural programming paradigms. Also, the development process itself influences the factors that must be considered. Keeping this in mind, it is crucial to use an appropriate cost model, which can be a hard task [Leung and Fan 2002].

Third, a lot of models and also experts rely on size metrics, such as, lines of code or function points, to determine the effort that is necessary to develop a software [Leung and Fan 2002]. However, such metrics must be considered carefully. Depending on the programming language, the size of functionalities can differ and requires adaptions on the cost model. Also, a reliable statement about code sizes can only be made after the development [Leung and Fan 2002]. Thus, it is difficult to apply concrete efforts to size metrics.

Fourth, the knowledge of experts is the most frequently used method for cost estimations [Moløkken and Jørgensen 2003]. However, it can become difficult to find a person with the right knowledge and training, or to repeat the estimation to simulate different

scenarios [Leung and Fan 2002]. It is also possible that an expert uses only his own knowledge as basis for calculations and ignores more experienced employees [Heemstra 1992]. Cost models do not provide a full solution for the dependency on experts. They still require experience and knowledge to calibrate their parameters to the company's situation [Jørgensen 2007; Jørgensen et al. 2009].

Finally, historical data from similar projects supports the estimation and decision processes. This data also could help to determine the importance of specific factors. Unfortunately, such data is not widely available [Heemstra 1992; Leung and Fan 2002; Northrop 2002].

In conclusion, software cost estimation provides methods to estimate the costs, risks, and benefits, for a project in advance. Based on the gained information, a company can make reasonable decisions whether a new product or development approach can be accepted. Still, there remain problems with effort estimations, independently from the used method. Those must be considered in industrial practice and also during the development of new cost models.

2.4 Summary

In this chapter, we presented the clone-and-own approach and described its problems. Afterwards, we described software product lines and their benefits. We focused on the extractive approach, which is the most common in practice, and variability mining, which can support automatic extractions of information. Finally, we introduced software cost estimation and its problems.

3. An Overview of Cost Models for Software Product Lines

In this chapter, we first define a use-case and requirements for our approach. Afterwards, we provide a short overview of existing cost models for software product lines. Based on our requirements we identify models that can be adapted for the extractive approach to software-product-line adoption. We further compare and discuss those models to select approaches we use as foundation for our work.

3.1 Cost Model Requirements

Cost models address different scenarios in which they are applied [Böckle et al. 2004]. For example, the approaches we discuss in this chapter focus on the introduction of software product lines. For our model we define the use-case as follows:

> A company owns a number of products that were developed by cloning. For some reasons, such as, to decrease the costs to develop new products, reduce maintenance effort, or improve customizability, the organization considers to change the development approach. Thus, it wants to evaluate the possibility to extract the legacy systems into a software product line.

From this description we derive assumptions for our model. Those do not apply in the context of proactive software-product-line engineering. First, the organization developed a number of products for the same domain, for instance, database systems. Thus, we assume that the variants contain similarities in design, model, and code. Second, the company is aware of its market situation. As a set of products exists, the organization has reliable information about their current and further success. Therefore, we assume that the company only wants to know whether a product line would result in cost savings. The success of the systems themselves is of minor interest. Finally, the extraction

of assets is a bottom-up approach [Koziolek et al. 2015]. We think that the same method is also reasonable for a corresponding cost model (compare with Section 2.3).

Before the organization decides to extract a software product line, an analysis of the costs and benefits is necessary. A company will only change its development process if cost savings can be achieved within a reasonable duration. The following questions are of interest to determine whether and when the organization can benefit from re-engineering its variants into a product line:

Q-1 How much does the extraction of the legacy systems into a software product line cost?

Q-2 How much does the development of new products within the software product line cost?

Q-3 How much does the software-product-line approach save during the maintenance per period as compared to continuing clone-and-own development?

Answering these questions enables an organization to make a reasonable decision about its future development approach. By answering the first one (*Q-1*), we determine the effort it requires to change the development process in a company. The other two questions consider the savings that adopting software-product-line engineering promise. With the second question (*Q-2*) the company determines the development costs for new products. Those should be lower compared to the clone-and-own approach due to improved reuse and customizability [Knauber et al. 2002; Pohl et al. 2005, pp. 9-13; Clements and Northrop 2006, p. 17; Apel et al. 2013, pp. 8-10; Martinez et al. 2015]. Still, the main reason for extracting a software product line are reduced maintenance efforts, as the code base is reduced in contrast to stand-alone systems [Yoshimura et al. 2006a; Rubin et al. 2012; Dubinsky et al. 2013]. Thus, answering the third question (*Q-3*) is crucial. Finally, the additional effort (*Q-1*) is accounted to the savings (*Q-2*, *Q-3*) to determine when the change will pay off. This is compared to the company's goals to make a reasonable decision. Therefore, our approach must provide answers in the form of cost estimations.

To enable users to apply our approach, we also have to describe its economic foundations. Those descriptions also support judgment-based estimations, which we further compare to cost models in the next section.

3.2 Combination of Cost Estimations

As we described in Section 2.3, software cost estimation can be applied in several ways. In this section, we discuss shortcomings of judgment-based estimations and cost models. Both approaches require manual work, for example, to determine information [Jørgensen et al. 2009]. The main difference is, that cost models define a set of input data and provide an equation to calculate the efforts. In contrast, for judgment-based estimations

information are selected individually and the efforts are determined by personal opinion. Finally, we describe the benefits of combining those two approaches and the structure of our model.

Shortcomings of Judgment-Based Cost Estimations

Estimations done by experts are common and frequently used in practice [Jørgensen 2004]. However, there are some problems that must be considered.

First, while the estimation process itself can be analytical and structured, the quantification of the efforts is in most cases based on intuition and too optimistic [Jørgensen 2007; Jørgensen et al. 2009]. The experts can also be influenced by unnecessary or unusable information [Jørgensen 2004; Koziolek et al. 2015]. For example, a stakeholder could assume to low development costs or rely on information about efforts in another domain that may vary significantly. Therefore, it is hard to justify the results to a client.

Second, it is problematic to find an expert who has the required knowledge and provides reliable estimation results [Heemstra 1992; Jørgensen 2004]. Especially, the lack of information of estimators is problematic. For example, the project manager might have the necessary knowledge about the company and domain while he is not familiar with the development approach that shall be used.

Third, as described in Section 2.3, estimations in early steps of the development process are rather inaccurate [Boehm 1984]. Therefore, many companies want to run a considerable amount of analysis to evaluate different scenarios, for example, a worst case. However, this is a difficult and time consuming task with judgment-based estimations while cost models can be adjusted to simulate different situations [Jørgensen et al. 2009].

Summarized, judgment-based cost estimations have the following shortcomings. As they are based on *intuition* it is difficult to justify the results. The expert might have a *lack of information*, which distorts his estimations or makes them less reliable. Due to *inaccuracies*, the company might run multiple scenarios, which is time consuming for humans.

Shortcomings of Cost Models

While cost models can overcome some of the problems of judgment-based estimation they also have shortcomings.

First, cost estimation models must be calibrated to an organizations situation. This is difficult, as the model can include too much or too little information and, thus, calculates unreliable results [Jørgensen 2007; Jørgensen et al. 2009]. Additionally, in most cases it is too complicated to formally describe all relevant information. With increasing complexity, due to more parameters, a model also becomes harder to comprehend.

Therefore, important contextual information or domain knowledge, that an expert can consider, can be missing [Jørgensen 2004, 2007].

Second, personal barriers and relationships are not included in any model [Jørgensen et al. 2009]. However, they may have a huge influence on the development effort and, therefore, should be regarded. Estimation models are also difficult to adjust to changing situations or a lack of information [Jørgensen 2007].

Third, in industrial practice, cost models are not as accepted as the calculations of experts. As a result, companies do not put much effort into training and and the proper usage of cost models. In some cases, the use of cost models is only applied to disguise judgment-based estimations [Jørgensen et al. 2009].

In conclusion, the following shortcomings apply to cost models. *Calibrating the parameters* to a company's situation is a challenging task and relies on human judgment. Cost models cannot display *personal barriers and relationships* or changing situations in those. Due to the *lack of acceptance*, models can hardly be applied solely.

Supporting Judgment-Based Estimations with Models

Judgment-based estimations and cost models have shortcomings. Both approaches require experts to determine input parameters. Additionally, estimators might apply a model for calculations and to justify their results. A cost model also cannot replace instincts or knowledge about culture and politics in an organization and its surrounding [Clements et al. 2005; Ali et al. 2009].

Analyzing multiple case studies, Jørgensen [2004, 2007] concluded, that experts perform better under specific circumstances. However, he found no proof that cost models are always worse nor better. The combination of judgment- and model-based estimations is considered as best way and often performs better than each approach solely [Boehm 1984; Jørgensen 2007; Jørgensen et al. 2009].

In conclusion, it is reasonable, to develop a cost model that supports judgment-based estimations. Thus, our approach includes two levels. The first level describes *economics* for the extractive software-product-line approach. It provides a structured guidance for the identification of important costs and supports the understanding of the whole model. This is necessary because we cannot provide formulas for every cost factor. The second level introduces our *cost model* that can be used or further adapted. In particular, we explain possibilities to gather information from the existing legacy systems. That information can be used in our model and also independently, for example for judgment-based estimations.

3.3 Overview

To select models on which we can base the two levels of our approach, we analyzed existing cost models. In this section, we first describe our search process and its results. Afterwards, we derive attributes from our previous descriptions to categorize and identify models that are suitable for the extractive approach.

We searched within the following online databases: Google Scholar, ACM Digital Library, IEEE Xplore Digital Library, SpringerLink and ScienceDirect. As search term we applied:

> (”software product line“ OR ”software product family“) AND (”costs“ OR ”cost estimation“ OR ”cost model“ OR ”investment“)

With this term we exclude papers that do not focus on the cost estimation for the software-product-line approach. Out of the results, we considered a paper for our analysis if it describes a cost model or provides an overview of those. In a first step we analyzed the surveys by Khurum et al. [2008], Ali et al. [2009], Charles et al. [2011], Blandón et al. [2013], and [Heradio et al. 2013]. We found that only Khurum et al. [2008] and Heradio et al. [2013] describe a systematic search process, which is not focused on, but includes cost models for software product lines. While Ali et al. [2009], Charles et al. [2011], and Blandón et al. [2013] do not describe how they found and selected the models, they define attributes and compare them. Thus, we cannot evaluate their completeness but use the provided categories.

After we extracted the cost models described in these surveys, we matched them with the other articles we found. In particular, we focused on the period since 2013, which is not covered by the surveys. Therefore, we also analyzed the references and citations of the papers to identify more current approaches. Overall, we identified 14 different models: Withey [1996], Poulin [1997], Schmid [2003], ABC analysis [Cohen 2003], SIMPLE [Böckle et al. 2004; Clements et al. 2005], Peterson [2004], COPLIMO [Boehm et al. 2004], SoCoEMo-PLE (2) [Lamine et al. 2005a,b], Ganesan et al. [2006], qCOPLIMO [In et al. 2006], Wesselius [2006], InCoME [Nobrega et al. 2008], Heradio et al. [2012], Tüzün and Tekinerdogan [2015].

According to Mili et al. [2000] and Schmid [2002] (cited by Ali et al. [2009]), each cost model has its own perspectives on software-product-line cost estimation. For a first overview, we focused on two attributes. The *underlying model* describes whether the cost model is based on other approaches for software cost estimations. Cost models that adapt a reliable model might have a solid basis. Approaches that are reused may be better suited for adaptions or are accepted in practice. However, this is no exclusion or inclusion but a beneficial criterion. Second, the *scope* of a cost model for software product lines covers either only the adoption or the complete life-cycle [Ali et al. 2009]. As we explained in Section 3.1, extracting a software product line is connected with additional costs. Only during the life-cycle savings can appear. Thus, we excluded models that only focus on the adoption process from our further investigations.

In Table 3.1 we display all approaches with the according underlying model and scope. We can see that half of the models were developed anew. Five approaches are based on Mili et al. [2000], COCOMO II [Boehm et al. 1995, 2000b], or COQUALMO [Chulani et al. 1999], which are cost models that do not focus on software product lines. Of all models, only Poulin, SIMPLE, and COPLIMO are reused themselves. Excluding the cost models that do not consider the life-cycle, only six remain.

Approach	Underlying model	Scope
Withey		Adoption
Poulin		Adoption
SoCoEMo-PLE (2)	Mili et al., Poulin	Life-cycle
Schmid		Life-cycle
ABC analysis		Adoption
SIMPLE		Life-cycle
InCoME	Mili et al., SIMPLE	Life-cycle
Ganesan et al.	SIMPLE	Adoption
Tüzün and Tekinerdogan	SIMPLE	Adoption
Peterson		Adoption
COPLIMO	COCOMO II	Life-cycle
qCOPLIMO	COPLIMO, COQUALMO	Life-cycle
Heradio et al.	COPLIMO	Adoption
Wesselius		Adoption

Table 3.1: Cost models for software product lines. Gray cells highlight models that are excluded because they do not consider the product-line life-cycle.

3.3.1 Categorization

Based on the requirements we defined in Section 3.1 and Section 3.2 we derive additional attributes that describe suitable cost models. They can be described with the following questions:

- Which *cost estimation approach* (Est) does the model apply?
- Are different *scenarios* (Scen) defined?
- Are *market and risk attributes* (M/R) considered?
- Which *evaluation approach* (Eval) is used for the model?
- Is the model used in *case studies and tools*?

Following, we describe each of the corresponding attributes and why we consider them important. In Table 3.2 we illustrate the values for the remaining six cost models.

Cost Estimation Approach

As we stated in Section 2.3, several methods for cost estimations exist. Some cost models for software product lines provide only a framework and describe cost drivers

or functions [Ali et al. 2009]. While those approaches can be adapted to many situation, they highly depend on experts' knowledge. Thus, we categorize such models as judgment-based. We consider other approaches, that define parameters and equations as algorithmic models. From our illustration in Table 3.2 we can conclude that the distribution of both cost estimation methods is equal. As we described in Section 3.2, combining judgment- and calculation-based estimations is beneficial. Thus, we want to pick a suitable model from each category. On the one hand, we can derive economic descriptions to explain the considered costs. On the other hand, we can adopt a cost model for the extractive approach to provide concrete calculations.

Scenarios

Some cost models define different use-case scenarios in which they can be applied [Mili et al. 2000; Schmid 2002, cited by Ali et al. 2009]. This indicates that a model is adaptable to changing situations, for example, the extraction of a software product line. We can see in Table 3.2 that this attribute correlates with the applied cost estimation approach. This indicates, that the judgment-based models provide more flexibility because they provide frameworks. In contrast, the algorithmic models are less suitable for adoptions but provide concrete equations for specific scenarios.

Market and Risk Attributes

Other attributes defined by Mili et al. [2000] and Schmid [2002] (cited by Ali et al. [2009]) are the consideration of market attributes and risks. They consider, whether the products that are developed will be successful or not. However, we assume that a company that wants to extract a software product line is aware of its market situation (compare with Section 3.1). Thus, we do not want to include this consideration in our approach. As we can see in Table 3.2 the market is only directly included by Schmid.

Model	Est	Scen	M/R	Eval	Case studies and tools
SoCoEMo-PLE (2)	AM	✗	✗	✗	✗
Schmid	JB	✓	✓	FE	✗
SIMPLE	JB	✓	✗	FE	[Nolan and Abrahão 2010] [Tang et al. 2010] [Koziolek et al. 2015]
InCoME	JB	✓	✗	CS	✗
COPLIMO	AM	✗	✗	FE	[Chen et al. 2006]
qCOPLIMO	AM	✗	✗	FE	✗
AM - Algorithmic cost model				CS - Case study	
JB - Judgment-based estimation				FE - Fictional experiment	

Table 3.2: Comparison of software-product-line cost models

Evaluation

There is no methodology to evaluate cost models for software product lines [Ali et al. 2009]. As we illustrate in Table 3.2, out of the six models, only InCoME was applied in an industrial case study while in four cases fictional data was used. Additionally, we found no evaluation for SoCoEMo-PLE, which makes a validation of its usability difficult.

Case Studies and Tools

We analyzed the citations of the six cost models to determine whether they are applied in case studies or tools. We think that applying an approach in practice or tools, either during the evaluation or later on, is a clue for its practical usability. Thus, we favor cost models that fulfill this criterion. We see in Table 3.2 that only SIMPLE and COPLIMO were used in additional case studies or tools.

3.3.2 Discussion

Following, we briefly describe the six cost model that consider the life-cycle. In addition, we discuss all models based on the identified attributes.

SoCoEMo-PLE 2

SoCoEMo-PLE (*Software Cost Estimation Model for a Product Line Engineering Approach*) [Lamine et al. 2005a] considers four investment cycles (component-, domain-, application-, and corporate engineering) that represent different viewpoints in the organization. In each cycle, suitable stakeholders determine input parameters to calculate investments, periodic costs and periodic benefits. The model is based on parameters from Poulin [1997], Mili et al. [2000], and COCOMO II [Boehm et al. 1995, 2000b]. An extended model called SoCoEMo-PLE 2 [Lamine et al. 2005b] also regards *commercials off-the-shelf* components. Those are standard software systems and can be included in a software product line.

We faced a number of complications with this model. First, the approach is complex and depends on the integration of a great number of different stakeholders, which increases the effort. Second, Lamine et al. [2005a,b] did not provide any evaluation of their model. Thus, we cannot judge the results and practical usability. Finally, the detailed description of the approach, provided in a master‘s thesis, is not available in English or German. Thus, the meaning and units of some parameters are unclear to us. In conclusion, we decided not to use SoCoEMo-PLE.

Schmid

Schmid [2003] introduces a cost model that consists of three levels. In the first level, several project parameters are selected. They are categorized into *development constraints*, *software attributes* and *market attributes*. Those values are used to determine

the costs, benefits and risks. The second level takes the *time-value of money* concept into consideration. This means, that earned or spend money is more worth in the presence than in the future [Drake and Fabozzi 2009]. Finally, the third level also considers re-usability alternatives. In particular, the scope of the product line and, thus, the company's portfolio are determined.

In conclusion, Schmid [2003] describes an approach that is usable for different scenarios. However, it requires a huge amount of information about the market and organization. If a company uses the extractive approach, its products are already successful. The goal is to achieve cost savings during maintenance rather than decisions about potential new software. We think that this makes the extensive analysis applied in this model an overhead of effort. Thus, we decided against using this model as basis four our own approach.

SIMPLE

The *Structured Intuitive Model for Product Line Economics* (SIMPLE) [Böckle et al. 2004; Clements et al. 2005] is an abstract approach that helps to identify the costs and benefits of a software product line but not calculating them. While this requires a lot of human effort for the estimation itself, the model is usable for developers and managers independently of their experience in software-product-line engineering. As a result, the model is adaptable for any product-line engineering approach. For example, Böckle et al. [2004] define a scenario for the extractive approach.

SIMPLE is the basis for other models, integrated in simulation frameworks, company specific tools, for instance, at Rolls-Royce, and used in industrial practice [Nolan and Abrahão 2010; Tang et al. 2010; Koziolek et al. 2015]. Additionally, this model is well documented and structured [Blandón et al. 2013]. As it fulfills our requirements but does not provide detailed calculations, we adapt SIMPLE for the economic descriptions of our model. We present a detailed discussion of SIMPLE in Section 4.1.

InCoME

The *Integrated Cost Model for Product Line Engineering* (InCoME) [Nobrega et al. 2008] is a combination of SIMPLE and the approach of Mili et al. [2000]. Like SIMPLE, it is an abstract model that supports the identification of costs and benefits. InCoME also includes several viewpoints on the organization, such as, domain engineering, product engineering or component engineering. With a simulation model, different business scenarios can be evaluated.

The additional layers increase the complexity of the estimation process. Stakeholders have to make multiple predictions for all possible scenarios and viewpoints. This might improve the company's decision for or against product-line engineering. However, it also increases the analysis effort. In our opinion, this approach does not provide benefits over SIMPLE for our economic descriptions. Therefore, we decided against InCoME.

COPLIMO

The *Constructive Product Line Investment Model* (COPLIMO) [Boehm et al. 2004] is a calculation-based estimation approach. Its parameters and equations are derived from COCOMO II [Boehm et al. 1995, 2000b]. COPLIMO is separated into two parts. First, the adoption costs for the software product line are estimated. This is based on different parameters that are categorized as *relative costs of writing for reuse* and *relative costs of reuse*. Second, life-cycle costs are estimated based on the amount of annually maintained code. In particular, COPLIMO considers the experience of developers with the software and the distribution of unique and reusable code in a product.

COPLIMO is a one of the well-structured and documented cost models [Blandón et al. 2013]. Other approaches and an estimation tool are based on it [Chen et al. 2006]. Thus, we think that COPLIMO provides an understandable and adaptable base for our cost model. However, the model describes no scenarios and its parameters and equations must be adjusted for the extractive approach. We provide a detailed description in Section 5.1.

qCOPLIMO

The goal of qCOPLIMO [In et al. 2006] is to take software quality into consideration. It extends COPLIMO with COQUALMO [Chulani and Boehm 1999], another COCOMO II derivative. As a result, additional parameters, for example, test effectiveness and defect costs, are introduced. We could not find case studies or tools in which qCOPLIMO is applied. Thus, we are not able to evaluate whether the possible estimation improvements justify the increased complexity. For this reason, we decided to remain with COPLIMO and, if necessary, introduce quality factors later.

3.3.3 Threats to Validity

For our overview and categorization, we are aware of two threats to validity.

First, we did not describe a full systematic literature review. Thus, it might be difficult to replicate our results. Especially, other researchers could define cost models differently from us. For example, they may consider only algorithmic models, or include software-product-line scoping approaches. However, we think that we provide a reasonable overview which is also based on corresponding surveys.

Second, other authors may consider a different set of requirements and attributes to select cost models. Thus, they could find other models better suited for adaptions towards the extractive approach. Still, we described and explained our selection such that the reader can derive his own conclusions.

3.4 Summary

In this chapter, we described the use-case scenario for our model. Based on this, we derived requirements our approach. Afterwards, we summarized existing cost models

for software product lines. For this purpose, we used a literature review. In particular, focused on the period not covered by surveys. Based on the requirements, we defined criteria to identify models which we can adapt for our approach and briefly discussed them. Finally, we chose SIMPLE for the economic (Chapter 4) and COPLIMO for the calculation level (Chapter 5) of our approach. Both models are described in detail in the following chapters.

4. Economic Descriptions for the Extractive Approach

As we described in the last chapter, our approach includes two levels. In this chapter, we focus on the economic level. Therefore, we first introduce SIMPLE [Böckle et al. 2004; Clements et al. 2005] as the basis of our descriptions. Afterwards, we describe how economics for software product lines are generally displayed in cost curves. We then adapt the functions and illustrations for the extractive approach. Finally, we match our descriptions with basic economic models.

4.1 SIMPLE

The *Structured Intuitive Model for Product Line Economics* (SIMPLE) provides an overview and descriptions of costs that should be considered in software-product-line engineering. The goal is to enable developers but also the management of a company to understand the estimation process. For this purpose, SIMPLE does not provide calculations for the cost functions it defines. Instead, the estimators in an organization must decide how to determine the efforts. For example, they could rely on experts or implement a cost model.

SIMPLE defines a general scenario of an organization's situation as follows:

> "An organization has n product lines, each comprising a set of products, and s1 standalone products. It wants to have m product lines, each comprising a (perhaps different) set of products, and s2 stand-alone products. Along the way, the organization intends to add k products or delete d products."
>
> [Böckle et al. 2004]

Other practical scenarios can be derived from this description. For example, Böckle et al. [2004] propose alternatives that consider the exclusion of products from an existing product line. Depending on the scenario, the costs and their coherence vary. For the estimation process, SIMPLE provides cost functions. However, those only support the identification of costs but do not provide implementations. Thus, either an additional model or judgment-based estimations are used. The cost functions can be separated into *basic*, which describe the development, and *evolution*, which considers the maintenance.

Basic Cost Functions

SIMPLE defines four basic costs that occur during software-product-line development:

1. C_{org} represents the costs of introducing the software-product-line approach in an organization. For example, this includes efforts for training, reorganization, and process improvement.
2. C_{cab} considers the costs of building the core asset base. Tasks, such as, commonality and variability analysis, introduction of development environments, and design of the software architecture, are considered within this function.
3. C_{unique} describes the effort of developing new functionality that is not part of the asset base.
4. C_{reuse} represents all costs that occur when assets are reused for a new product. For example, costs that arise during the identification, integration, or testing of components.

The costs of developing a software product line, can be estimated with Equation 4.1. The number of distinct products that will be built is described with n.

$$C_{SPL} = C_{org} + C_{cab} + \sum_{i=1}^{n} (C_{unique}(product_i) + C_{reuse}(product_i)) \tag{4.1}$$

To decide for or against a product-line approach, its estimated costs are compared to those of building the software separately. For that reason, SIMPLE defines the function $C_{prod}(product)$ which returns the cost to develop one product without reuse. Afterwards, the savings or losses for a software product line can be estimated with Equation 4.2.

$$C_{savings} = \sum_{i=1}^{n} C_{prod}(product_i) - C_{SPL} \tag{4.2}$$

Finally, the return on investment (ROI) is calculated. Therefore, the savings of a product line are compared to the necessary investments, as shown in Equation 4.3.

$$ROI = \frac{C_{savings}}{C_{org} + C_{cab}} \tag{4.3}$$

The basic cost functions only consider the development of a software product line. However, life-cycle costs are also considered in SIMPLE.

Evolution Cost Function

In SIMPLE, the maintenance costs are described as the effort of releasing new versions. They are summarized in the costs for evolution (C_{evo}) as illustrated in Equation 4.4. A new cost function C_{cabu} is introduced to represent efforts to adapt the asset base. This is necessary as each update in a product may require adaptions in assets. In addition, every change can also have side effects on other products, which must be considered. Thus, for each variant a company must consider the costs for changes in the asset base, the product unique code, and for reusing the updated assets. Again, the maintenance costs for the product line are compared to those of single-system development.

$$C_{evo} = \sum_{i=1}^{n} (C_{cabu}(product_i) + C_{unique}(product_i) + C_{reuse}(product_i)) \tag{4.4}$$

In conclusion, SIMPLE provides a set of cost functions. However, they only support the identification of costs for software product lines. The implementation of the functions is left to the user.

4.2 Economic Descriptions

In this section, we discuss the cost functions provided by SIMPLE with regard to the extractive software-product-line approach. Therefore, we use the economic curves that display the effort reduction for the proactive adoption shown in Figure 4.1 as basis and extend them. They illustrate the basic assumptions of the existing cost models: The effort to initiate a software-product-line is higher than for single-system development [Knauber et al. 2002]. Those investments are compensated through reduced development costs for new variants.

The shown curves are highly simplified. The greatest restriction is the assumption, that the development of a new product requires always the same effort. Thus, both curves are linear. However, new software variants differ, for example, in size, functionality, and complexity. In many cases the organization will also reuse existing artifacts with unsystematic approaches, such as, clone-and-own. This can reduce the effort for single system development. Therefore, the costs for one product can be significantly higher or lower than those of another. For the software product line, adding new assets increases the dependencies and interactions. The rising complexity and testing effort can lead to higher costs.

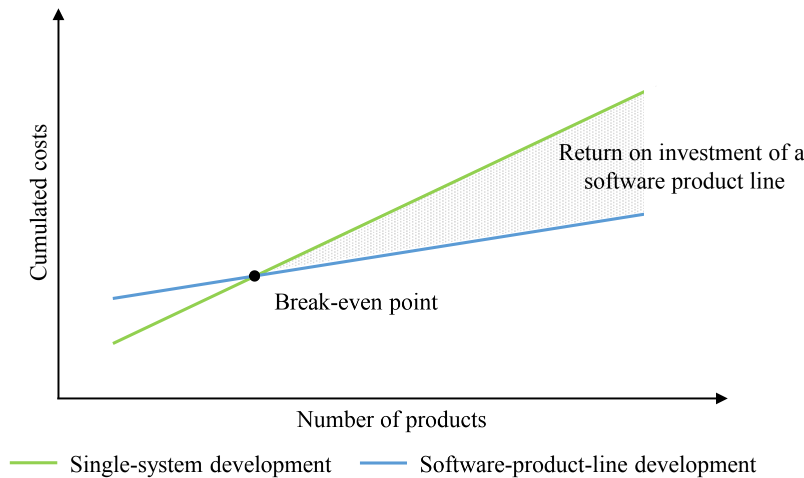


Figure 4.1: Hypothetical cost curves for proactive software-product-line development

At the moment a product line is planned, its specific costs and benefits cannot be exactly measured and the organization is taking risks [Schmid and Verlage 2002; Tang et al. 2010]. Those uncertainties span a range for possible cost curves as is shown in Figure 4.2. We conclude from this diagram that the calculation of the break-even point is difficult. In particular, some factors and their impact, for example, technological innovations or political changes, are hard to predict. Still, they can have huge impact on the development effort. Thus, the time point after which the software product line pays off might vary from the calculated plan. Considering such risks, a company must also be aware of the fact that the software product line could fail. This would not only result in additional costs but provide no benefit at all. Running several scenarios, for example, a worst case, provides clues about those possibilities. Conducting estimations for multiple situations by experts is expensive. Cost models provide a faster and less costly method [Jørgensen et al. 2009].

The linear curves in Figure 4.1 also disregard effects of gained experience and knowledge. Developers need less effort with familiar domain and software. For example, Tüzün and Tekinerdogan [2015] analyzed several surveys and found that experience influences the costs of developing a product as single system and within a product line. They also confirmed this with an industrial case study. In Figure 4.3 we illustrate possible cost curves that regard those effects. As we can see, the average costs of developing a new product decrease over time and are not constant. However, this is often assumed by existing cost models. Another observation is, that with an increasing number of developed products the positive influences of gained experience decrease [Tüzün and Tekinerdogan 2015]. Therefore, the curves are regressive. This can be expected, as the effort of developing a new variant cannot decrease without limit. The costs for new functionalities and customization remain. Experience effects do not only apply to the adoption but also during the remaining life-cycle and influence other attributes, such as, time-to-market or the return on investment, as well. For the extractive approach

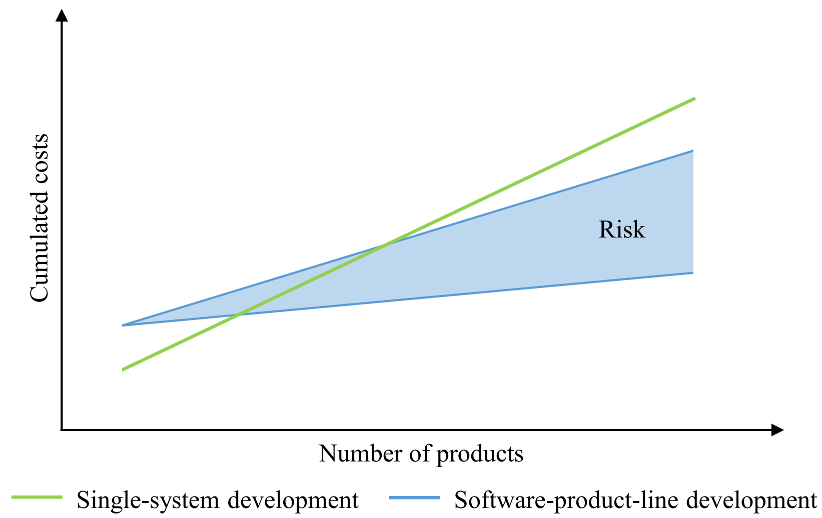


Figure 4.2: Cost curves of the proactive software-product-line approach with risk, adapted from Schmid and Verlage [2002]

we also have to consider that the developers already gained experiences for the existing products.

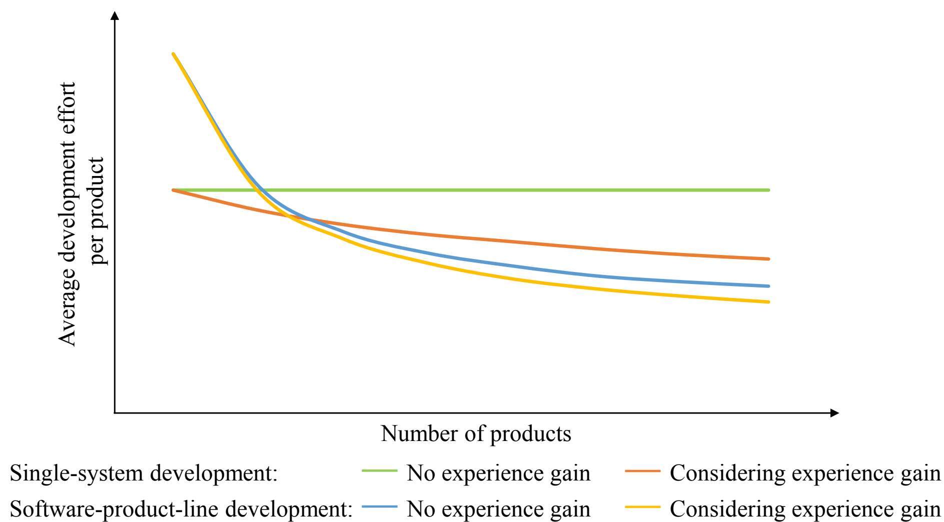


Figure 4.3: Impact of experience on the average development costs for single-system and software-product-line development, adapted from Tüzün and Tekinerdogan [2015]

Summarized, linear cost curves exclude effects, for example, risks and gained experience, and apply average estimations for all products. Those restrictions make economic models easier to understand and use. For this reason, we also use the simplified curves shown in Figure 4.1 during our following discussions. However, we have to consider that the described effects exist.

4.2.1 Costs to Develop Assets

When the extractive approach is applied, a company owns a set of legacy products. From those, a software product line shall be extracted by reusing existing artifacts. Those re-usable fragments can be re-engineered and do not need to be developed anew. Instead of the initial effort of the proactive approach, with domain analysis and design, there is normally a smaller adoption barrier [Krueger 2002b; Böckle et al. 2002; Schmid and Verlage 2002]. Thus, extracting a software product line from legacy systems requires less investment than developing one from scratch. Hence, the proactive approach is not reasonable for our use case.

Specifically, as we illustrate in Figure 4.4, the costs that were spent to develop the legacy systems must be excluded from the cost estimation. The products already exist and are now re-engineered into a product line. Thus, the adoption results only in additional costs. Therefore, the first question an organization needs to answer is (compare with Section 3.1):

Q-1 How much does the extraction of the legacy systems into a software product line cost?

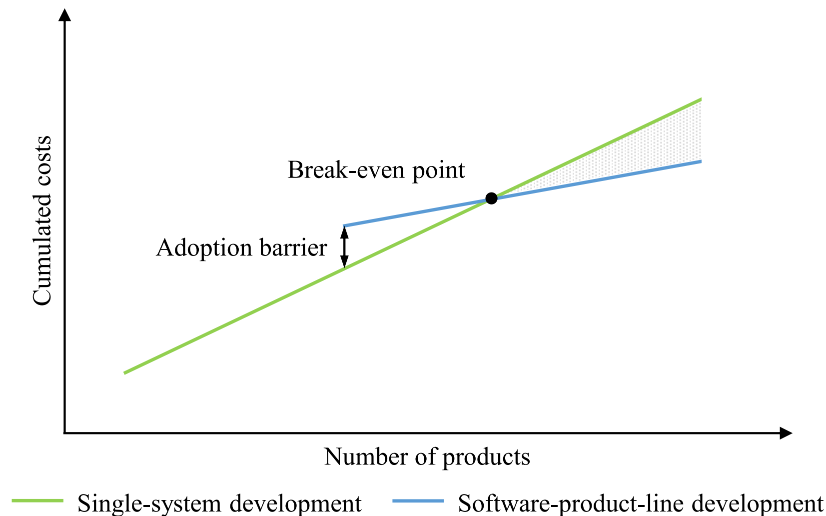


Figure 4.4: Hypothetical cost curves for extractive software-product-line development

While a lot of code already exists, variability must still be modeled and implemented. Tools for feature location or code-clone detection support this process. They provide necessary information about commonalities and differences between the legacy systems [Duszynski et al. 2011]. However, the location and extraction of feature still requires manual work [Biggerstaff et al. 1993; Yoshimura et al. 2006b; Kästner et al. 2014]. Another benefit is the experience of the developers with the software [Tüzün and Tekinerdogan 2015]. They are familiar with code, documentation, and design, which helps them to extract assets. Furthermore, the domain analysis can be reduced or even skipped, as

the products are already at the market. Because of those benefits we assume that the extraction of the asset base should be less expensive than developing it with the proactive approach. This is also supported by Krueger [2002b] who states that the extractive approach enables an organization to switch faster towards a software product line.

The estimation of adoption costs also supports the decision which legacy systems or features should be re-engineered. In some cases, it may be too expensive to extract a whole product, due to its size, complexity, and dependencies. Instead of adapting those variants into the software product line they can remain stand-alone. This is also an alternative if the company owns versions that are either outdated or have significant quality problems. Thus, an organization must decide which functionalities of the legacy systems shall be migrated into a product line. In extreme cases, it is possible to either extract all or only the core features. For the first scenario every functionality becomes an asset. This requires a lot of investment and initial effort as features must be located, re-developed into assets, and tested. However, the benefits are a decrease in code size that must be maintained as well as high variability and customizability. In the second scenario only the core features, which are shared among all legacy systems, are extracted into assets. The product specific functions remain stand-alone. As a result, the adoption costs are lower, but the flexibility of the developed software product line decreases. However, it is possible to extract all possible subsets of features. This can help an organization to balance the initial investment and required variability. One practical example is to extract only the features that are required for the new variant. Afterwards, those feature can be used as base for a product line that is incrementally expanded. A cost model for the extractive approach must consider those possibilities. In our approach we achieve this with a bottom-up strategy. Thus, a company can estimate the costs for every feature separately and decide which are suitable for a product line.

4.2.2 Costs for New Products

Identical to the adoption in the proactive approach, an organization wants to know after how many new products they reach the break-even point (see Figure 4.4). This leads to the second question we defined in our requirements (see Section 3.1):

Q-2 How much does the development of new products within the software product line cost?

In this context, it is important that we consider the costs for a new variant with individual functions and customization. Building the exactly same software product frequently requires almost no effort [Apel et al. 2013]. In Figure 4.5 we illustrate the estimation process for the development of new variants in a software product line. Based on the product requirements the company identifies features that must be developed (C_{unique}). The features can either be left stand-alone or integrated into the asset base. Finally, the assets for the new product are selected and the customized variant is instantiated. At this point, the company considers the costs for reuse (C_{reuse}). In COPLIMO [Boehm

et al. 2004], the individual efforts for assets are not calculated. Instead, products are described with unique and reusable code parts. This approach summarizes the costs for one variant but does not allow estimations for new and extracted assets. Thus, we have to adapt COPLIMO to apply the described methodology in our cost model.

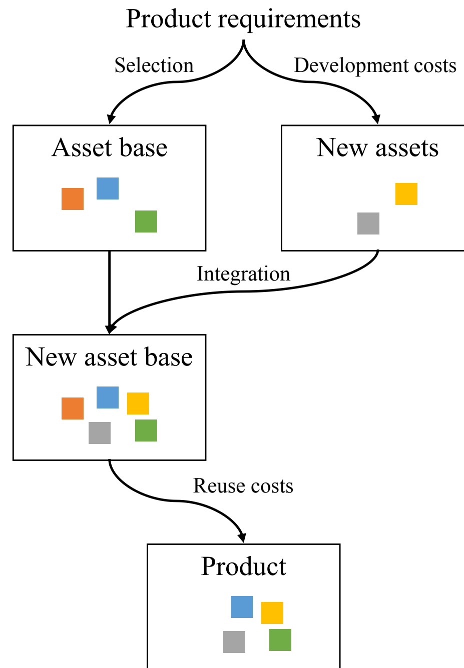


Figure 4.5: Calculating the costs for new products

Another assumption about software product lines is that more features can be developed with the same effort compared to stand-alone systems [Knauber et al. 2002]. While the integration of a new feature is more expensive, the derivation of products is cheaper. In most cases, one new asset also does not only provide one but a whole set of new variations. Still, due to this explosion of possible configurations it becomes more difficult to manage the variability [Deelstra et al. 2005]. Therefore, it is more likely that an organization implements the software product line only for the variants it will actually sell.

4.2.3 Costs for Maintenance and Evolution

In many cases the reason to extract a software product line is not the reduced effort to develop new variants. Often, an organization might not plan new products at this point but faces increasing maintenance costs and wants to reduce them [Apel et al. 2013, p. 41]. Thus, it is essential for our model to answer our third question (see Section 3.1), which displays this situation:

Q-3 How much does the software-product-line approach save during the maintenance per period as compared to continuing clone-and-own development?

For this purpose, we determine the cost difference in maintenance between single-system and software-product-line development. We not only have to consider the number of variants but also the time [Cohen 2003; Boehm et al. 2004; Krueger 2004]. It is essential to determine the savings that appear in a period of time in order to estimate when the investment in a product line is compensated. An organization might decide against an approach that only pays off after more than five years. Some companies cannot afford to make investments that do not pay off quickly. Others might not plan for such a long period or do not know whether the products will provide the same revenue. For example, a competitor could decrease the income because the users switch and the organization might plan to stop the development of their own variants.

An important point to consider is the quality of the legacy systems. They might contain errors in design and implementation. A company can remove them during the adoption phase which increases the investment. Alternatively, the problems can be fixed during the life-cycle. However, implementing variability increases the complexity of the code and can lead to new flaws [Fenske and Schulze 2015]. Removing errors then can become more difficult. From the existing products a company could identify quality parameters to estimate the maintenance costs. Such information can be the complexity, test-ability, maintainability, or design flaws.

As we showed in Chapter 3, most existing cost models focus only on the adoption of software product lines but disregard the costs and savings that occur during the life-cycle. However, product lines are often used to decrease the maintenance effort. Thus, it is necessary to further investigate costs that occur due to additional variability, necessary tests, complexity, and product scoping [Schackmann and Lichter 2006].

4.2.4 Organizational Costs

To successfully apply the software-product-line approach in a company, some organizational measures must be considered. Estimating the according costs and benefits is difficult. Often they cannot be measured directly. For example, the required effort is depending on the organizations situation as well as the knowledge and productivity of its employees. Developing a cost model that displays the influences of human relationships is not only expensive but also exploding in complexity [Jørgensen 2007]. Thus, we summarize activities that must be considered during the estimation. We do not provide specific parameters for all of them in our calculations.

Software-product-line engineering requires other processes than single-system development. In particular, to enable reuse among several variants developers must apply common processes. Identical programming and documentation styles are necessary [Poulin 1994]. Thus, it is important to define, unify, document, and follow, structured processes and work-flows [Dikel et al. 1997; Jones and Soule 2002; Northrop 2002; Böckle et al. 2002; Pohl et al. 2005, p. 9; Mansell 2006]. This process improvement provides benefits even without the usage of product lines, but an estimation of the costs and savings it will provide is difficult [Jones and Soule 2002].

In addition to the refinement of its processes, a company may also require a set of new tools [Gacek et al. 2001]. The costs of these tools can be calculated. However, their occurring benefits are hard to measure. If the organization does not develop a product line for the first time, it can already have a suitable tool-chain that does not need to be changed. Examples for important tools are:

- A development environment that supports the chosen language and variability methods.
- An instantiation tool that supports the customization and adoption of the products.
- A version control system to manage changes in the development artifacts.

Another aspect is to convince and train the different stakeholders, for example, managers and developers [Northrop 2002; Böckle et al. 2002]. They must understand the software product line approach, development processes, tools, and the infrastructure. Especially, the transition from developing a single system at a time to a software product line is a major change and is often a completely new concept for the employees [Knauber et al. 2000; Mansell 2006]. The communication between the developers and their dedication are essential because they are no longer working on separated systems, but on the same code base [Mansell 2006]. Thus, the integration of changes and extensions in the product line must be coordinated. Training the stakeholders for those tasks requires effort. However, the costs decrease when the participants already gained experience with software-product-line engineering or the systems they developed [Gacek et al. 2001; Tüzün and Tekinerdogan 2015]. In particular, this applies for the extractive approach. The understanding of the developers for the product-line approach and the products is included as parameter in our cost model.

As described, the costs and benefits of introducing the software product line significantly depends on the organization's situation and its employees. To decide, which development tools and how much training are necessary, requires expert knowledge and analysis. Still, we derive from our descriptions that the organizational costs are an investment. They are spent to introduce the software-product-line approach. In addition, those investments have impact on the life-cycle efforts. Thus, the organizational costs are essential to answer all questions we defined in Section 3.1.

In the next section, we take a detailed look on the cost functions in connection to economic cost curves. In particular, we focus on the impacts of investments on the development of new products and maintenance.

4.3 The Cost Factors in the Cost Curve

In economics it is often assumed that all costs can be reduced to the following two categories [Viner 1932]:

- *Fixed costs* (C_f) summarize all costs that are necessary to initiate, and remain constant during, the production. For example, they include costs for machinery or buildings. However, those costs are only fixed for the period that is considered. Later on additional investments may be necessary.
- *Variable costs* (C_v) summarize the costs to create a number (n) of products. For example, they represent wages and the costs for resources that are required for each unit.

The fixed costs represent the investments that are necessary to build the software product line [Pohl et al. 2005, p. 9]. Variable costs include all efforts for a new variant. Equation 4.5 describes the basic cost function (top) in comparison to the calculation of the SIMPLE model (bottom). On the one hand, we can see that the costs for organizational activities and the asset base are not influenced by the number of products that are developed. Therefore, they represent the fixed, respectively adoption, costs that are necessary to create a software product line. On the other hand, the costs for reused and unique code parts depend on the number of products that are developed and, thus, belong to the variable costs.

$$\begin{aligned} C = &\overbrace{C_{org} + C_{cab}}^{C_f} + \overbrace{\sum_{i=1}^{n} (C_{unique}(product_i) + C_{reuse}(product_i))}^{C_v * n} \\ C_{SPL} = &C_{org} + C_{cab} + \sum_{i=1}^{n} (C_{unique}(product_i) + C_{reuse}(product_i)) \end{aligned} \tag{4.5}$$

In Figure 4.6 we apply the fixed (C_f) and variable (C_v) costs to the extractive software-product-line approach. We see that the fixed costs illustrate the adoption barrier. Thus, they represent the gap between the single-system and starting software-product-line development. Afterwards, separated development of new variants is not applied anymore. This is illustrated as dashed line. The accumulated variable costs for a number of products are represented by the area below the cost curve for software-product-line development.

An organization can spend different amounts of money for its fixed costs. For example, it can reduce or extend the training of its employees or use open-source instead of proprietary software. The effort that is spent during those tasks has impact on the costs that will occur while developing new products and during the life-cycle [Boehm 1984]. In Figure 4.7, we illustrate a possible scenario on how the fixed and variable costs are correlated. If we increase our initial investment, the depending variable costs describe a regressive curve. Accordingly, changing the fixed costs by the same amount (Δ_f) leads to different savings (Δ_v). This means, that the marginal utility of one invested unit decreases with the amount already spent.

If we consider the fixed and variable costs as economic goods, this representation displays an *indifference curve*. Indifference curves were first introduced in economics by

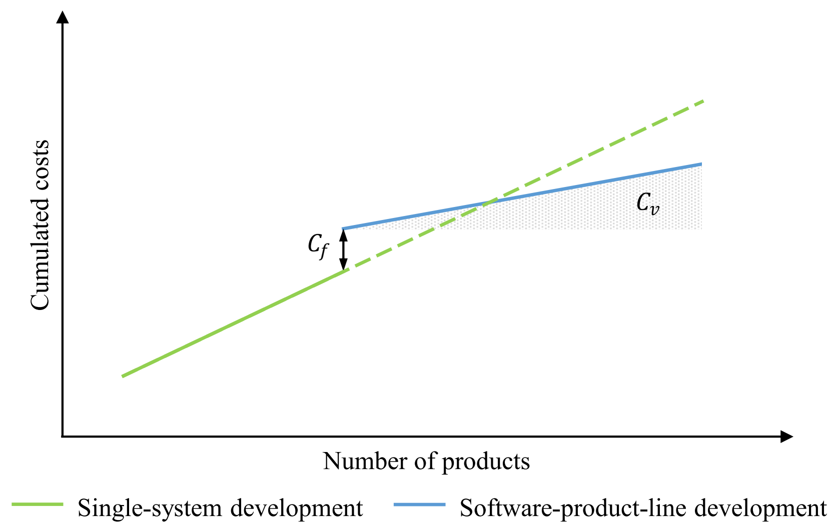


Figure 4.6: Fixed (C_f) and variable (C_v) costs for the extractive software-product-line approach

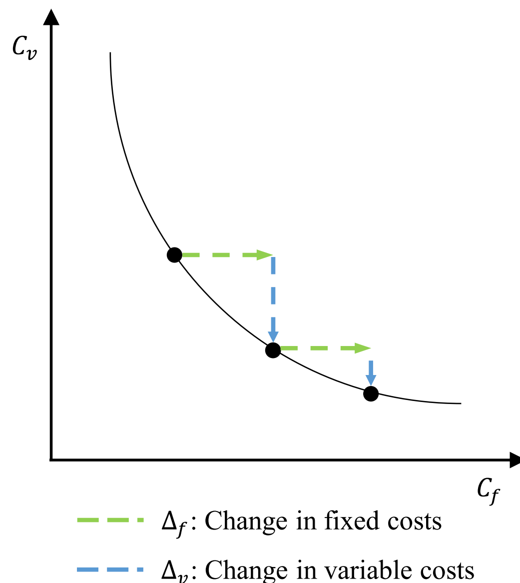


Figure 4.7: Simplified dependency between fixed and variable costs

Pareto [1906] (German translation by Reiß [1992, pp. 213-215]) and describe how two goods can be exchanged with each other. However, the model is difficult to apply in the margins, where one good is almost fully substituted with the other. For example, in our case, we cannot completely replace the variable costs by more initial investments. Thus, if we assume that the fixed costs require money as input and define a certain quality as output, we can describe the following problem a company faces:

> What is the optimal ratio between investment and resulting quality?

An organization that wants to extract a software product line faces uncertainties and unreliable data, which makes a prediction difficult [Schmid and Verlage 2002]. Another influence is the selected software-product-line implementation technique. A heavyweight strategy requires more investment then a lightweight one but also decreases the necessary effort for new variants [McGregor et al. 2002]. Thus, it is problematic to find a reliable ratio for a company. Instead, an organization might run several scenarios to consider uncertainties or different migration strategies. For example, costs can be estimated for the extraction of the whole, or only parts, of the legacy systems. In Figure 4.8 we illustrate cost curves for product lines with different investments (Δ_f). Due to the resulting variable costs (Δ_v), the break-even point changes. Thus, some strategies are not appropriate for a company. Those are marked red in Figure 4.8. For example, high investments need more new products before the decreased development costs pay off. In contrast, low fixed costs might not provide enough benefits for new products or the life-cycle. In conclusion, a company must be careful to make a reasonable decision.

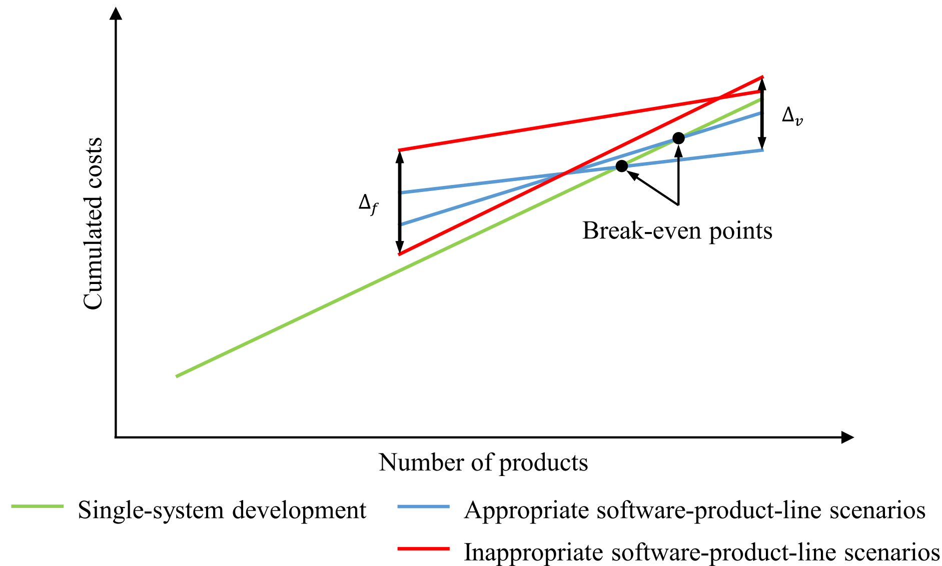


Figure 4.8: Possible scenarios for varying investments for the extractive approach

If we assume that C_f^* describes the optimal fixed costs for an organization, the following scenarios are possible:

1. The organization reduces the initial effort as they are not able or willing to make such a high investment. This will decrease the competence of their employees and the quality of the product line. Thus, the costs for developing new software and the maintenance will increase.

2. The organization increases the initial investment. While the quality and competence of the employees improves further, those benefits cannot compensate the additional investment in an acceptable period of time.

In reality, we can hardly know the optimal investment and an organization must also consider its limited resources. It is also possible that reducing the investment, which a cost model recommends, has no negative influence. Also, increasing it could lead to huge benefits. The conclusion we can derive from the economic model is that in software engineering, the fixed costs influence the variable and life-cycle costs. Thus, a company must be careful when adopting a software product line.

4.4 Summary

Existing cost models for software product lines rarely consider the extractive approach [Koziolek et al. 2015]. We used the functions provided by SIMPLE to categorize the costs that must be considered. We then developed an economic model for the extractive approach. In particular, we focused on the first task we defined in Chapter 1 and, thus, discussed benefits that can occur due to legacy systems.

As we described, some adaptions are necessary to customize existing cost models for the extractive software-product-line approach. There are also factors that are difficult to estimate correctly. For instance, costs and benefits of necessary process adaptions are hard to predict. Still, based on existing literature and cost models we considered the most important factors.

In the next chapter we introduce our implementations for the cost functions. Thus, we introduce a cost model based on the descriptions we provided here.

5. A Semi-Automatic Cost Model

As we described in the previous chapter, SIMPLE [Böckle et al. 2004; Clements et al. 2005] only provides descriptions of the cost factors an estimator must consider. However, it is not clarified how the costs can be calculated. For this, it is possible to apply COPLIMO [Boehm et al. 2004; Blandón et al. 2013]. In order to combine both approaches, it must be clear, how they are connected. Thus, we begin this chapter with a detailed overview of COPLIMO and match it with SIMPLE. Afterwards, we introduce our cost model for the extractive software-product-line approach. To do this, we adapt the calculations of COPLIMO based on our economic description given in the previous chapter. We then present an overview that shows the connections of our model to software-product-line engineering. Finally, we discuss which parameters of our cost model can be determined automatically and how.

5.1 COPLIMO

The *Constructive Product Line Investment Model* (COPLIMO) is an algorithmic cost model for software-product-line engineering. It is well documented and is adapted in other models and tools [Chen et al. 2006; Blandón et al. 2013]. Thus, we think that COPLIMO provides a good basis for our own cost model. A shortcoming in regard to the extractive approach is that costs are estimated for whole products. Thus, the efforts for the assets or extraction are not separated. Heradio et al. [2012] identify similar simplifying assumptions for COPLIMO:

- All products are of equal size.
- Every product reuses all core assets.
- The distribution of code that is unique and reused with or without modifications is equal within all products.

- COPLIMO uses effort multipliers provided by COCOMO II [Boehm et al. 1995, 2000b] to estimate development costs. However, those multipliers are applied to the whole product line and not on single assets.

For the extractive approach those simplifications are problematic. Developing the product line starts not with a single product but a set of features that are extracted from several legacy systems. Thus, the costs of each feature and not only whole products are of interest. However, it is not trivial to adopt this cost model for the extractive approach. For our cost model we modify the calculations according to our economic descriptions.

Following, we introduce COPLIMO based on Boehm et al. [2004]. The development cost estimation for a software product line is based on two parameters proposed by Poulin [1997]. These parameters are the *relative cost of writing for reuse* and *relative cost of reuse* that represent the additional effort of writing and using re-usable software.

Relative Cost of Writing for Reuse

The *relative cost of writing for reuse (RCWR)* factor is composed of three cost drivers. Each of them is a factor itself. Thus, a value of 1.0 means no additional effort while a higher value represents additional costs.

- *Development for reuse (RUSE)* describes the effort that is necessary to implement software reuse. For a single software-product-line, the factor is considered as 1.15, which means that the effort is 15% higher than for a stand-alone product.
- *Required reliability (RELY)* regards the additional effort of checking that each variant in the software product line works.
- *Degree of documentation (DOCU)* describes the necessary documentation for a project and focuses on the demands during the life cycle.

For each of this cost drivers, a set of effort multipliers is defined by Boehm et al. [2004]. An estimator has to rate the required degree of each cost driver and apply the according multiplier to the model. Finally, the relative cost of writing for reuse parameter is calculated by multiplying all cost drivers (see Equation 5.1).

$$RCWR = RUSE * RELY * DOCU \tag{5.1}$$

The resulting value is later used to consider the additional costs for the asset base.

Relative Cost of Reuse

There are two factors that describe the *relative costs for reuse (RCR)* within COPLIMO. On the one hand, an asset can be reused without changes (*black-box*, *bb*). In that case, it is only necessary to select and add the module into a new product as well as documenting the process. These tasks are combined into the *assessment and assimilation (AA)* factor that ranges from 0 to 8%, depending on the complexity. For black-box reuse no further parameters are necessary. Thus, the relative effort for reusing a black-box asset is described as shown in Equation 5.2.

$$RCR_{bb} = AA \tag{5.2}$$

On the other hand, not all assets can be reused without changes (*white box*, *wb*). Thus, the calculation of the *relative cost of reuse* becomes more complex. In addition to the assessment and assimilation factor the following estimations are used:

- *Design modified (DM, in %)* describes how much of the features design must be adapted to integrate it into the new system. This value must be guessed subjectively.
- *Code modified (CM, in %)* includes the amount of source code that is changed to integrate the asset.
- *Integration required for adapted software (IM, in %)* estimates the necessary effort to integrate the reused feature into a product and test it in relation to a stand alone approach.
- *Software understanding (SU, in %)* describes how well the software is written. Therefore, subjective opinions of the structure, clarity, and self-descriptiveness of the code are determined and compared with rankings described by Boehm et al. [2004].
- *Programmers relative unfamiliarity (UNFM, multiplier)* estimates how good the programmers understand the software. For example, if they have never worked on it, a 1.0 multiplier is additionally applied.

Next, the so called *adoption adjustment modifier (AAF)* is calculated (see Equation 5.3). This value summarizes the relative modification to a products size.

$$AAF = (0.4 * DM) + (0.3 * CM) + (0.3 * IM) \tag{5.3}$$

Depending on the *AAF* result, the *adoption adjustment multiplier (AAM)* (see Equation 5.4), which is the white box relative cost of reuse equivalent, can be determined.

A high AAF value (>50) indicates more changes on the reused code. Because the resulting efforts are nonlinear, the equation is adapted [Selby 1988, cited by Boehm et al. [2004]].

$$RCR_{wb} = AAM = \begin{cases} \frac{AA+AAF*(1+(0.02*SU*UNFM))}{100} & \text{, if } AAF \leq 50 \\ \frac{AA+AAF+(SU*UNFM)}{100} & \text{, if } AAF > 50 \end{cases} \quad (5.4)$$

The resulting values are later used to determine the costs to reuse assets for new products.

Effort Estimation

Before estimating the costs for a software product line, COPLIMO requires those for single-system development. The predicted efforts are used for comparison and as basis for further calculations. For single-systems, COPLIMO assumes that all products have nearly the same size (*PSIZE*). The effort, in person-months, for a number (n) of standalone products (*PMNR(n)*), is estimated with Equation 5.5.

$$PMNR(n) = n * A * PSIZE^B * \prod_{j=1}^{m} EM_j \quad (5.5)$$

A and B represent factors from COCOMO II and describe a project's complexity and the company's situation. For example, they can include process maturity and team cohesion, which both include organizational costs. *EM* are additional effort multipliers and cost drivers.

Based on the result, the costs for the first instance of the software product line (*PMR(1)*) are estimated. The effort of developing the product without reuse, is split among unique (*PFRAC*), with-box reusable (*AFRAC*) and black-box reusable (*RFRAC*) parts. The additional costs of writing for reuse (*RCWR*) is applied to the reusable fragments as illustrated in Equation 5.6. Thus, the result includes investments for the asset base.

$$PMR(1) = PMNR(1) * (PFRAC + RCWR * (AFRAC + RFRAC)) \quad (5.6)$$

For new variants in a software product line, the first step is to estimate the *equivalent product size* (*EKSLOC*). This parameter includes the sizes of unique code and for necessary adaptions to reuse assets. Therefore, the corresponding relative cost of reuse (*RCR*) are applied. The size of code that must be implemented anew is calculated as shown in Equation 5.7.

$$EKSLOC = PSIZE * (PFRAC + (RFRAC * AAM) + (AFRAC * AA)) \quad (5.7)$$

Based on this size prediction, the effort for new variants (*PMR(n)*) is estimated with Equation 5.8.

$$PMR(n) = A * EKSLOC^B * \prod_{j=1}^{m} EM_j \tag{5.8}$$

Until this point, COPLIMO calculates the efforts of single-system and software-product-line development. It considers the additional investment for the asset base in the first product-line instance. To consider maintenance costs, the following calculations are used.

Software-Product-Line Life-Cycle

For the product-line life-cycle, COPLIMO applies the *annual change traffic (ACT)* factor. This parameter represents the relative amount of code that is modified each year. Furthermore, COPLIMO assumes that the former introduced factors and the annual change traffic remain almost constant. As for the development process, we also need a comparison value for development without reuse. At first, the amount of code that must be maintained per year (*AMSIZE*) is calculated. For a single product, its complete code is used, as described in Equation 5.9. In contrast, for a product line it is necessary to differentiate between unique, black- and white-box parts (see Equation 5.10). These three values are added to estimate the complete amount of code that must be maintained.

$$AMSIZE = PSIZE * ACT(1 + \frac{SU}{100} * UNFM) \tag{5.9}$$

$$\begin{aligned} AMSIZE_{Unique} =& PSIZE * PFRAC * ACT(1 + \frac{SU}{100} * UNFM) * n \\ AMSIZE_{bb} =& PSIZE * RFRAC * ACT(1 + \frac{SU}{100} * UNFM) \\ AMSIZE_{wb} =& PSIZE * AFRAC * ACT(1 + \frac{SU}{100} * UNFM) * \\ & (1 + \frac{AAF}{100} * (n-1)) \end{aligned} \tag{5.10}$$

Finally, the maintenance effort for l years and n products are calculated. Thereto, Equation 5.11 is applied.

$$PM(n,l) = l * n * (A * AMSIZE^B * \prod_{j=1}^{m} EM_j) \tag{5.11}$$

In contrast to SIMPLE, COPLIMO provides an algorithmic cost model. Thus, it does not only support the identification of possible costs but also their calculation. Following, we describe how both models are connected.

Connecting SIMPLE and COPLIMO

Existing cost models for software-product-line engineering differ not only in their attributes (compare Chapter 3) but also in the parameters that are used and how those are named [Ali et al. 2009; Heradio et al. 2013]. Therefore, it can be difficult to match the calculations and concepts of one approach with those of another, even if they represent the same costs [Ali et al. 2009]. Still, we can connect the COPLIMO parameters to the SIMPLE cost functions as follows.

- C_{org}: Organizational costs can be considered with the scaling factors A and B. In addition, COPLIMO rates how good the developers understand the software they develop.
- C_{cab}: The costs of developing the reusable core asset base are represented in the *relative cost of writing for reuse* and the effort to build the first variant within the software-product-line. However, the costs for assets are not estimated separately.
- C_{unique}: In COPLIMO the unique software parts are considered as the number of new lines of code for a product.
- C_{reuse}: The costs for reusing are considered within the *relative cost of reuse*, which summarizes different parameters. In contrast to SIMPLE, COMPLIMO differentiates between black-box and white-box reuse.
- C_{evo}: COPLIMO considers the evolutionary costs within its life-cycle model. For the calculation, the code size that must be maintained in each year is required.

In Table 5.1 we illustrate the connections between both cost models. We see that the focus of COPLIMO are the development of assets and their reuse. For those costs the most parameters can be applied and, thus, estimated more detailed.

SIMPLE	COPLIMO
C_{org}	SU, UNFM, A, B
C_{cab}	RUSE, RELY, DOCU, PFRAC, RFRAC, AFRAC
C_{unique}	PFRAC
C_{reuse}	AA, DM, CM, IM, SU, UNFM, AAM, RFRAC, AFRAC
C_{evo}	AMSIZE, ACT, SU, UNFM

Table 5.1: Coherence between SIMPLE cost functions and COPLIMO parameters

The combination of judgment- and calculations-based cost estimation improves the results [Boehm 1984; Jørgensen 2007; Jørgensen et al. 2009]. In the previous chapter we introduced our economic descriptions. The following section provide the second level of our approach, the cost model. It is based on COPLIMO.

5.2 Adapted Calculations

In this section, we describe how we adapted COPLIMO for the extractive software-product-line approach. As we stated, the organizational costs are considered in several parameters within COPLIMO. Thus, we will not separately describe cost estimations for those.

Within this section, we use a number of parameters from COPLIMO and COCOMO II [Boehm et al. 1995, 2000b]. While some of them remain as they are described within those two models, we adjust others. In Table 5.2 we provide an overview with descriptions of all parameters used within our model. We explained some of them for the COPLIMO in the previous section. The parameters we adapt for our model are beneath the dividing line. We introduce them in detail when they are applied.

Parameter	Description
A	Calibration coefficient of COCOMO II
B	Exponential scale factor of COCOMO II
EM	Effort multipliers of COCOMO II
AA	Assessment and assimilation factor
AAM	Adaptation adjustment multiplier
$AMSIZE$	Code size that is annually maintained
$RCWR$	Relative costs of writing for reuse factor
$RUSE$	Development for reuse cost factor
$DOCU$	Degree of documentation cost factor
$RELY$	Required reliability cost factor
$EKSLOC$	The equivalent code size to instantiate a product
$PM(asset_{adoption})$	The effort to extract an asset from the legacy systems in person-month
$PM(asset_{new})$	The effort to develop a completely new asset in person-month
$PM(product)$	The effort to develop a product in person-month
$aSize$	The size of an asset
$uniqueSize$	The size of code that is unique for a product and is not reused
$RCWR_{adoption}$	RCWR factor for the extraction of assets
$RUSE_{adoption}$	RUSE factor for the extraction of assets
$DOCU_{adoption}$	DOCU factor for the extraction of assets
$p_{applied}$	Number of products that use an asset

Table 5.2: Overview of parameters used for the cost estimation. New parameters are displayed beneath the dividing line.

In Section 4.2 we provided economic descriptions for the extractive approach. Following, we use those explanations to derive a suitable cost model from COPLIMO.

5.2.1 Costs to Develop Assets

As most existing cost models for software product lines, COPLIMO compares the efforts of developing a set of variants stand-alone or with a product line. To simplify the calculations, constant sizes and distribution of reused and unique code for all products are assumed. Such simplifications can lead to distorted and inaccurate estimations [Heradio et al. 2013]. A bottom-up approach, which estimates the efforts for assets and then aggregates costs for products, can improve the predictions. Moreover, as we described in our requirements, following a bottom-up strategy is better suited for the extractive approach. Therefore, we have to apply two steps, illustrated in Figure 5.1, to calculate the costs for the adoption barrier.

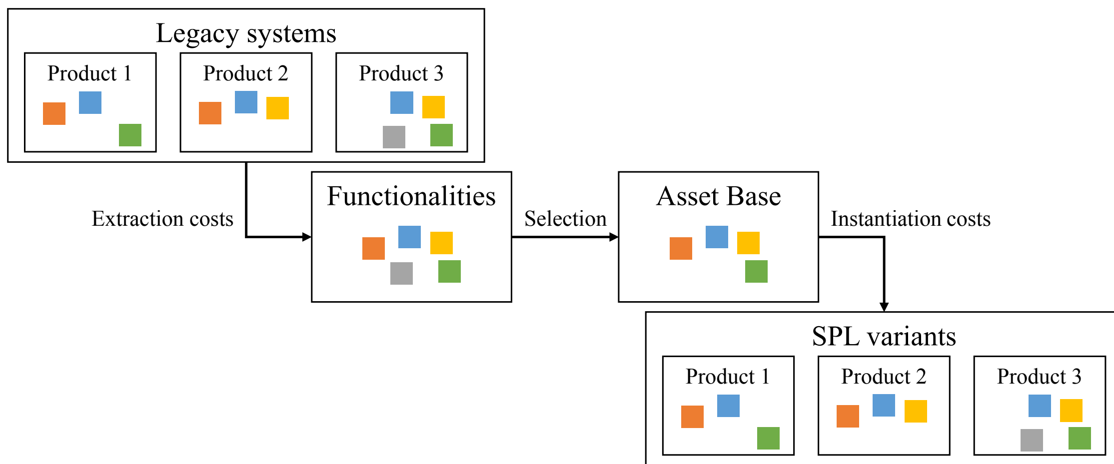


Figure 5.1: Calculating the adoption costs

First, we estimate the effort to extract each functionality into an asset. The results can be used to simulate several scenarios. For example, only extracting the features that are shared among all variants. Thus, a company can decide which parts of the legacy systems are reasonable to re-engineer [Boehm 1984]. Second, we estimate the costs to instantiate the considered legacy systems from the extracted software product line. The accumulated costs for those two tasks describe the adoption effort. Following, we introduce the respective formulas in our model.

Costs for Extracting Assets

In COPLIMO, the costs of developing the asset base are included in the estimation for the first instance of the new software product line. For the extractive approach, in contrast, we already have a set of existing products and have to decide which functionalities out of them shall be extracted. Therefore, instead of considering new products as whole, we estimate the effort of developing each asset independently. To do this, we assume that every asset can be seen as an individual project. Therefore, we apply the adapted COPLIMO formula for developing a product in a product line on asset level, as we show in Equation 5.12. We can estimate the whole costs for all assets by

summing them up. A, B and the additional effort multiplier remain the same as in the original estimation and can be adjusted for each asset, if necessary. In smaller projects, for example, a single feature, B is often applied as one [Boehm et al. 1995]. For a first estimation without details about every asset, this assumption is especially useful. Several sizes can be summed up before the effort calculation. This might distort the results but simplifies the process because it requires less analysis.

$$PM(asset_{adoption}) = A * aSize^{B} * RCWR_{adoption} * \prod_{j=1}^{m} EM_j \tag{5.12}$$

An asset itself is a separated implementation and does not benefit from the reuse of other assets. Thus, we do not split the code size into black-box, white-box, or unique parts. We adapt the relative costs of writing for reuse for the existing artifacts as shown in Equation 5.13. An estimator has not only to consider the additional variability that must be implemented but also the additional effort of merging different variants.

$$RCWR_{adoption} = RUSE_{adoption} * RELY * DOCU_{adoption} \tag{5.13}$$

We assume that the reliability must be considered as if there exist no legacy systems. We think that this is appropriate as different code parts might be merged and additional variability is added. Therefore, it is necessary to evaluate the assets as completely new products. In contrast, the factors of developing for reuse and additional documentation should be reduced depending on re-usability of the existing artifacts.

Costs of Instantiating a Product

To estimate the effort of instantiating a product, we adapt the COPLIMO formula that calculates the effort for each product that benefits from systematic reuse. Therefore, we need the amount of additional code required to integrate all assets and develop the unique product parts ($uniqueSize$). As we show in Equation 5.14, we can consider black-box (AA) and white-box (AAM) reuse depending on the adaptions that are necessary to use an asset. The necessary parameters can be calculated as in COPLIMO and described in Section 5.1. For the existing products we assume that most of their parts can also be reused and do not need to be developed anew. Therefore, it seems better fitted to apply the calculation for white-box reuse instead of unique development. The size we determine at this point ($EKSLOC$), refers to the amount of code that must be adapted or developed anew to instantiate a product.

$$EKSLOC = uniqueSize + \sum_{i=1}^{n}(aSize_{bb,i} * \frac{AA}{10}) + \sum_{i=1}^{n}(aSize_{wb,i} * AAM) \tag{5.14}$$

Afterwards, we use Equation 5.15, provided by COPLIMO, to estimate the effort to instantiate a variant from the product line.

$$PM(product) = A * EKSLOC^B * \prod_{j=1}^{m} EM_j \tag{5.15}$$

By accumulating all efforts to extract assets and instantiate the legacy systems from the software product line, we estimate the adoption costs.

5.2.2 Costs for New Products

To estimate the costs of developing new products, we first have to consider the scenario that the organization decides to develop a new asset. This can occur if additional variants are planned that can reuse functionality from the new product. In this case, we first use Equation 5.16 to estimate the effort to develop the new asset. This time, the relative costs of writing for reuse are calculated with the original COPLIMO estimates, as we cannot reuse any existing artifacts.

$$PM(asset_{new}) = A * aSize^B * RCWR * \prod_{j=1}^{m} EM_j \tag{5.16}$$

Afterwards, we can use Equation 5.14 and Equation 5.15 as previously explained. We can also directly apply them if no new asset is planned. In those cases, the estimation is equal to COPLIMO.

In COPLIMO we found no consideration of development effort for new assets. Instead, we would have to apply the same formulas as for the first software-product-line instance and combine them with the estimation for following products. Otherwise, we only can estimate the effort of developing variants that already benefit from systematic reuse but do not implement new assets. We think that this is a shortcoming of the COPLIMO model because the necessity of such adjustments might not be obvious to everybody. We address this problem by estimating the costs for new assets separately. Only afterwards, we calculate the costs to instantiate the new variant. In addition, the individual costs for an asset can be distributed among several products. For example, if a company develops two new systems that both require the same new asset, the project plan can be adapted accordingly.

5.2.3 Costs for Maintenance and Evolution

The maintenance costs for a software product line are independent from the development approach. However, COPLIMO's assumption that all features are reused in all products can lead to distortions. In particular, the maintenance costs assume additional integration code for white-box assets for every variant. In reality, those assets may only be used in some products. Estimations based on assets instead of whole products improve the accuracy [Heradio et al. 2012]. For this reason, our model considers that assets are only reused by some variants. As a result, we might assume a smaller

amount of code that must be maintained and, thus, our cost estimations can be lower than with COPLIMO.

As in COPLIMO, we first estimate the code size that must be maintained every year (*AMSIZE*). For each asset, we identify the unique, black-, and white-box parts. In Equation 5.17 we provide the according equations. In contrast to COPLIMO, we do not estimate the size of adoptions for white-box reuse for all products. Instead, we use only the number of variants that really apply the feature ($p_{applied}$). Thus, we consider that some assets are not used in all products and require no additional code for the integration.

$$\begin{aligned} AMSIZE_{Unique} &= size * ACT * (1 + \frac{SU}{100} * UNFM) \\ AMSIZE_{bb} &= size * ACT * (1 + \frac{SU}{100} * UNFM) \\ AMSIZE_{wb} &= size * ACT * (1 + \frac{SU}{100} * UNFM) * (1 + \frac{AAF}{100} * p_{applied}) \\ AMSIZE &= AMSIZE_{Unique} + AMSIZE_{bb} + AMSIZE_{wb} \end{aligned} \tag{5.17}$$

Finally, we can estimate the maintenance effort for one year by using Equation 5.18. The number of different features that exist within the product line is represented with n. This includes assets and unique product parts.

$$PM_n = A * (\sum_{i=1}^{n} AMSIZE)^B * \prod_{j=1}^{m} EM_j \tag{5.18}$$

With this approach, we do not consider the additional variability effort during the life-cycle. To include, for example, increasing complexity, further investigations on their effects are necessary [Schackmann and Lichter 2006].

In conclusion, our cost model considers the cost functions of SIMPLE we described in Chapter 4. We derived corresponding equations that fulfills our requirements from Section 3.1. In the next section we connect our model to software-product-line development.

5.3 Cost Estimation and the Software-Product-Line Development Process

Until now, we described the economic basics for the extractive approach in Chapter 4 and used them to derive a cost model in the previous section. In this section, we match our approach with the software-product-line engineering process (compare with Section 2.2). Therefore, we add an additional dimension that integrates our cost factors

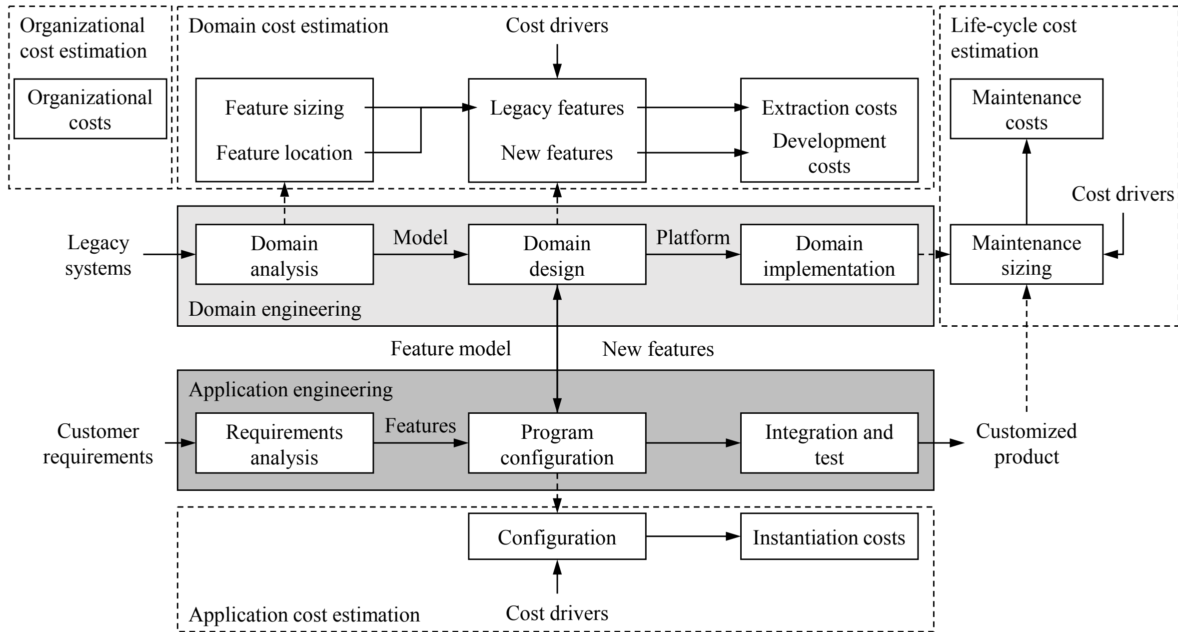


Figure 5.2: The cost estimation process for the extractive approach with our cost model. Dashed lines illustrate connections and related information with the software-product-line development process from Czarnecki and Eisenecker [2005, p. 21]. Cost drivers represent parameters our cost model can apply.

in four clusters. We apply a similar nomenclature to Czarnecki and Eisenecker [2005, p. 21], illustrated in Figure 5.2. Following, we briefly describe each cluster and the corresponding parts of our work.

We described the *organizational cost estimation* in Section 4.2.4. Organizational costs consider efforts that are necessary to introduce a software-product-line approach in an organization, for instance, process refinements or training. Due to their complexity it is difficult to estimate them.

The *domain cost estimation* summarizes all efforts that are necessary to develop the *domain implementation*. For the extractive approach, this especially includes the analysis of legacy systems to identify common functionalities and measure their sizes. Based on this data, we can estimate the effort of extracting a product line into reusable assets. Additionally, the costs for new features are considered in this cluster. We focused on the according economic descriptions in Section 4.2.1 and derived equations in Section 5.2.1.

Within the *application cost estimation*, we focus on the costs to instantiate a variant from the software product line. Based on customer requirements or the legacy systems, product configurations are derived. Those define the amount of new code and the number of white- or black-box reused assets, which are used for the cost estimation. We described this in Section 4.2.2 and provided according equations in Section 5.2.2.

Finally, with the *life-cycle cost estimation* we predict the annual maintenance effort. Therefore, our model requires the size of the *domain implementation* and unique parts

of all *customized products*. We described the economic basis in Section 4.2.3 and calculations in Section 5.2.3.

In conclusion, our approach matches with the software-product-line development process. We consider the domain and application engineering as cost drivers. In particular, our model focuses on the costs to develop the asset base. In addition to the development process, we also consider efforts for introducing and maintaining a product line. Also, our model considers data, that are part of the development process. For example, analyzing the legacy systems provides information about possible features and their sizes. In the next section we investigate possibilities to automatically extract data for our cost model from existing variants.

5.4 Automating the Cost Estimation

Determining reliable values for parameters of a cost model is a challenging task and requires effort [Jørgensen 2007; Jørgensen et al. 2009]. For the extractive approach a set of existing products shall be re-engineered. Thus, analyzing the legacy systems can provide reliable data. In addition, providing methods which do this automatically reduces the effort for the estimation. In this section, we investigate possibilities to gather information from existing legacy systems. Especially, we describe the problems of automatic analysis of multiple software variants. Finally, we propose an approach to compute the amount of shared code which enables us to estimate costs for different scenarios. However, as stated by Heemstra [1992] and Koziolek et al. [2015] it is not advisable to only rely on those values. Additional investigations and interviews with the participating developers are still necessary.

5.4.1 Extracting Information from Legacy Systems

Legacy systems contain data that can be used for our cost model. Following, we briefly investigate the analysis of *source code comments*, *re-usability*, and *code similarities*. Because the first two possibilities are rather difficult to apply automatically and require manual work, we later focus on the identification of common code fragments.

Assessments on the degree and quality of source code comments in legacy systems can be used to adjust the documentation level for the adoption process ($DOCU_{adoption}$). However, approaches to measure comment quality are rare [Steidl et al. 2013]. Analyzing such inline documentation requires tools that are designed for the specific programming language, comment style, and a quality rating. For instance, Khamis et al. [2010, 2013] propose a tool to evaluate only *Javadoc* comments. Still, considering several legacy systems makes it difficult to use automatic analysis, due to different documentation styles. Therefore, comments that describe similar parts must be merged and validated manually. Moreover, additional documentations, such as, models or analog descriptions, must also be included. This requires a lot of manual effort. Thus, we decided not to focus on the extraction of source code comments from legacy systems.

Determining the re-usability of the source code can also support our estimation process. If code that belongs to an asset is already easy to reuse, the re-engineering effort reduces ($RUSE_{adoption}$). However, there exist numerous guidelines and metrics for the estimation of re-usability [Poulin 1994]. Deciding which ones we can apply is difficult. The metrics must be suitable for the individual project and accepted by the customer. This can only be determined with manual analysis and interviews. Another negative impact on the re-usability have *code smells*, which represent flaws in design and implementation Fowler et al. [2006, pp. 75-88]. If such shortcomings exist within legacy systems and are also migrated into a software product line, their negative impact can increase [Fenske and Schulze 2015]. Re-usability metrics and design flaws must also be merged for several legacy systems. Thus, automatic analysis is difficult and we decided against further investigations.

The reduction of the source code size has the biggest impact on the cost savings of software product lines [Cohen 2003; Boehm et al. 2004]. Therefore, we focus on possibilities to estimate the distribution of code duplicates among the legacy systems. For our cost model, it would be ideal to automatically detect all features and their sizes over all variants. This would enable us to estimate the extraction effort of each asset. However, a fully automatically approach for this task is seems not realistic [Biggerstaff et al. 1993; Kästner et al. 2014]. It still requires experts to verify that code and features match. In addition, further code fragments that belong to the same feature must be identified manually. Also, as we stated in Section 2.2, not all variability mining and feature location approaches use source code as input, they depend on a specific programming language, can only work on a single system, or do not generate the output we need. Instead, we apply code-clone detection to determine the common sizes between legacy systems. Following, we first describe different extraction scenarios. Afterwards, we describe an approach to automatically determine the common source code size.

5.4.2 Extraction Scenarios

Our approach to identify reliable data does not extract features but computes the sets of identical code occurring between the source code of multiple variants. In Figure 5.3 we illustrate the possible intersections between four variants. From this, we derive two complementary scenarios:

1. All possible assets are extracted.

2. Only the core assets are extracted.

Estimating the effort for those two scenarios already provides a lower and upper bound for the adoption costs. Overcoming the adoption barrier of other scenarios requires some effort between those two. As we already stated, the amount of extracted assets in each case influences the savings during the maintenance phase. In practice, a company might consider to reuse only the necessary parts of the new product in the first place

(compare to Section 4.2.1). Therefore, we think that the two scenarios can provide a first impression and others should be defined in collaboration with the company. Thus, the effort of analysis can be reduced while achieving more reliable estimations. It is possible to automatically run multiple scenarios by changing parameters.

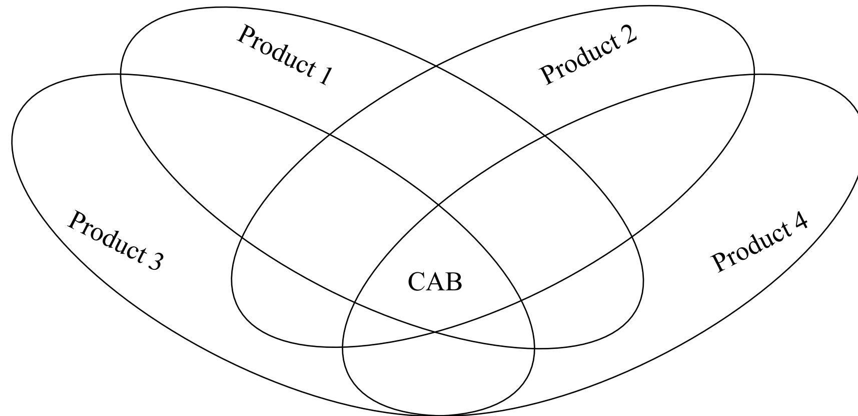


Figure 5.3: Venn-diagram of possible overlaps between four products. The code that is shared among all variants is marked as core asset base (CAB).

The exact number of features is not predictable as several of them can be part of an overlapping code block. For simplifications we assume that all intersections can be extracted into individual features. Therefore, the maximum number of possible assets between n variants is the same as all areas in the Venn-diagram and can be calculated with Equation 5.19 [Duszynski et al. 2011].

$$Intersections = 2^n - 1 \tag{5.19}$$

As we illustrate in Figure 5.3, this already leads to a huge number of possible asset areas, even if we do not consider all separate features within the intersections. We assume that in reality it is most likely that not all combinations will occur. In practice, systems are often derived with clone-and-own. Thus, they share common parts with the system they are cloned from and differentiate from other cloned variants. In contrast, considering n variants, we assume that at least $n+1$ meaningful assets can be extracted.

While those scenarios can define an upper and lower bound, the practical approach for a company may be to extract only the features that are necessary for the new product. Copying the assets into a separate variant represents the clone-and-own approach. If the assets are redesigned for systematic reuse within a software product line, it is possible to incrementally add other products and new functionality, while reducing the adoption costs. In such a case, it must be decided from which legacy systems functionalities are cloned. For a cost estimation our model can still be applied.

Following, we describe an approach to determine the sizes of the intersections. The results can be used as input for our cost model and to display several scenarios.

5.4.3 Computing the Shared Code

We are aware of some tools that may compute the similar code sizes for a set of variants, for instance, the *Bauhaus tool suite* [Raza et al. 2006; Mende et al. 2008], *ConQAT* [Bauer and Hauptmann 2013], the approach by Al-Msie'deen et al. [2013], or *BUT4Reuse* [Martinez et al. 2015]. However, we were not able to use them properly, due to installation or availability issues. Additionally, it is difficult to automatically extract all assets and their sizes [Biggerstaff et al. 1993; Kästner et al. 2014]. The task requires manual analysis to locate features, especially for several legacy systems. Instead, we propose a simplified method based on code-clone detection to compute the sizes of intersections between them. Yoshimura et al. [2006a] use the same approach for cost estimation in a case study but they also do not provide their tool or algorithm.

Duszynski et al. [2011] propose an approach to detect code clones between a number of variants. They compare files with the same name from different projects using the diff algorithm. As a result, they compute the number of code lines that are identical and different between a set of variants. However, the *diff* algorithm that is used, does not identify code clones that are moved to another part of the file and also counts single line clones [Duszynski et al. 2011]. Those can hardly belong to a feature. We think that the following adaptions on the clone detection produce more suitable results:

- Use a code-clone-detection tool that is able to identify clones that were moved within files.
- Only identify code clones with a minimum length of code lines as features should contain more than a few lines.

However, it is challenging to find a suitable code-clone-detection tool. As we stated, we were not able to use approaches designed for cross project analysis [Raza et al. 2006; Mende et al. 2008; Bauer and Hauptmann 2013; Martinez et al. 2015]. Thus, we searched for a clone-detection tool that is not designed for, but can still be used on multiple products.

Based on the work of Roy et al. [2009] we tried two tools that are freely available and usable for academical evaluation: *NICAD* [Roy and Cordy 2008; Cordy and Roy 2011] and *Simian*[1]. Both have also been used for the identification of cross-project code clones [Krinke et al. 2010; Svajlenko et al. 2013]. However, *NICAD* requires individual implementations for every programming language and knowledge about *TXL* [Cordy et al. 1991]. Simian, instead, can work language independently and is usable without further modifications. Thus, we decided to use Simian for clone detection.

As we extracted the information with our proposed approach we also found some problems with Simian. First, overlapping code clones are reported multiple times. For example, in Figure 5.4 we illustrate a part of the comparison of the *Sd2Card.cpp* files

[1] Simian - Similarity Analyser: http://www.harukizaemon.com/simian/ [08.02.2016]

```
01  bool Sd2Card::readData(uint8_t* dst, ui
02    // wait for start block token
03    uint16_t t0 = millis();
04    while ((status = spiRec()) == 0XFF)
05      if (((uint16_t)millis() - t0) > SD_
06        error(SD_CARD_ERROR_READ_TIMEOUT)
07        goto fail;
08      }
09    }
10    if (status != DATA_START_BLOCK) {
11      error(SD_CARD_ERROR_READ);
12      goto fail;
13    }
14    // transfer data
15    spiRead(dst, count);
16
17    // discard CRC
18    spiRec();
19    spiRec();

20    chipSelectHigh();
21    return true;
22
23   fail:
24    chipSelectHigh();
25    return false;
26  }
```

```
01  bool Sd2Card::readData(uint8_t* dst, ui
02    // wait for start block token
03    uint16_t t0 = millis();
04    while ((status = spiRec()) == 0XFF)
05      if (((uint16_t)millis() - t0) > SD_
06        error(SD_CARD_ERROR_READ_TIMEOUT)
07        goto fail;
08      }
09    }
10    if (status != DATA_START_BLOCK) {
11      error(SD_CARD_ERROR_READ);
12      goto fail;
13    }
14    // transfer data
15    spiRead(dst, count);
16
17  #if ENABLED(SD_CHECK_AND_RETRY)
18    {
19      uint16_t calcCrc = CRC_CCITT(dst, c
20      uint16_t recvCrc = spiRec() << 8;
21      recvCrc |= spiRec();
22      if (calcCrc != recvCrc) {
23        error(SD_CARD_ERROR_CRC);
24        goto fail;
25      }
26    }
27  #else
28    // discard CRC
29    spiRec();
30    spiRec();
31  #endif
32    chipSelectHigh();
33    return true;

34  fail:
35    chipSelectHigh();
36    return false;
37  }
```

```
01  bool Sd2Card::readData(uint8_t* dst, ui
02    // wait for start block token
03    uint16_t t0 = millis();
04    while ((status = spiRec()) == 0XFF)
05      if (((uint16_t)millis() - t0) > SD_
06        error(SD_CARD_ERROR_READ_TIMEOUT)
07        goto fail;
08      }
09    }
10    if (status != DATA_START_BLOCK) {
11      error(SD_CARD_ERROR_READ);
12      goto fail;
13    }
14    // transfer data
15    spiRead(dst, count);
16
17  #if ENABLED(SD_CHECK_AND_RETRY)
18    {
19      uint16_t calcCrc = CRC_CCITT(dst, c
20      uint16_t recvCrc = spiRec() << 8;
21      recvCrc |= spiRec();
22      if (calcCrc != recvCrc) {
23        error(SD_CARD_ERROR_CRC);
24        goto fail;
25      }
26    }
27  #else
28    // discard CRC
29    spiRec();
30    spiRec();
31  #endif
32    chipSelectHigh();
33    return true;

34  fail:
35    chipSelectHigh();
36    return false;
37  }
```

Figure 5.4: Overlapping code clones in three different versions of *Sd2Card.cpp* from different *Marlin* forks. The additional code fragments are marked with blue, while their absence is highlighted with red.

from three different forks of the open-source 3D-printer firmware *Marlin*[2] with *KDiff3*[3]. As we can see, the middle and right files are complete clones in this part, while in the one on the left a code fragment is missing. Additional cloned lines in the middle and on the right are marked with blue. The corresponding gap in the left file is illustrated with red. Analyzing all product variants at once led to the problem that overlapping and included clones were reported multiple times. In the shown case, Simian identifies one clone from line 1 to 15 for all three variants and another one from line 1 to 37 for the two identical files. Instead, excluding the overlapping part would be ideal. Thus, the second code clone would only range from line 17 to 37. Then, the size for each intersection describes only the unique clones.

A second problem occurs because all files are compared pairwise [Duszynski et al. 2011]. Thus, if one project contains multiple clones of the same code fragment, but with different sizes, they are not summarized. In such cases, each identified clone is reported separately and some of them several times. For this reason, computing the code clones for all intersections separately does also not provide the unique number of cloned lines.

Still, Simian can identify code clones between multiple products. We apply the following approach to compute the unique size for each intersection:

1. Calculate the source-lines-of-code for each legacy product individually.

[2] MarlinFirmware/Marlin: https://github.com/MarlinFirmware/Marlin [08.02.2016]
[3] KDiff3 - Homepage: http://kdiff3.sourceforge.net/ [08.02.2016]

2. Run Simian on all legacy products to identify all code clones between all possible sets of products.
3. Extract the overall size of each intersection.
4. Compute the size of unique source-lines-of-code for each intersection by removing overlapping clones.

Besides the difficulties in the reported code clones, this approach still has some other problems. We only calculate the size for intersections but we would still need to verify whether we identified possible features. Additionally, while Simian can be adjusted to ignore minor changes in the source code, such as literal renaming, it cannot detect all clone types [Roy et al. 2009]. In particular, clones of type three and four cannot be identified. Thus, the analyzed systems might have more functionality in common than assumed. Still, our method computes an approximation of identical code between variants, which can be used in our cost model.

5.5 Summary

In this chapter we introduced COPLIMO and connected it to the cost functions of SIMPLE. We then derived adaptions for the equations to suit the extractive approach. We showed how our equations are connected to the cost estimation processes and to product-line engineering.

Afterwards, we discussed possibilities to extract information that can be used during cost estimation with our model. Most metrics must be carefully discussed in collaboration with the organization that applies the model, strongly depend on the used programming language, or require non-code artifacts. Thus, we focused on the identification of code clones. To do this, we adapted the approach proposed by Duszynski et al. [2011] to compute the size of identical code between a set of existing legacy systems.

6. Evaluation

In this chapter, we evaluate our cost model we introduced in the previous chapter. However, evaluating cost models is problematic. Several approaches are used but no defined methodology exists [Ali et al. 2009]. Thus, we first introduce an evaluation process for our approach. In this process, we describe a fictional business case based on forks of an open-source project. Afterwards, we apply COPLIMO [Boehm et al. 2004] and our approach on the data. We derive hypotheses based on our economic descriptions in Chapter 4 to evaluate the results. Then, we conduct qualitative interviews with experts to rate the usability and identify problems. Additionally, we compare our estimations with reports about case studies for the extractive software-product-line approach. Finally, we discuss possible threads to validity for our model and the evaluation process.

6.1 Evaluation of Cost Models

The evaluation of cost models is a challenging task. The best case scenario would be to use it several times in practice, analyze the results, identify problems, and adjust the model. Additionally, expert estimations and alternative models would be applied to compare the performances. This way, a cost model could be evaluated in several scenarios. After each iteration refinements and adoptions are applied to improve the approach. However, case studies are not only time consuming but it is also problematic to find suitable companies. The organization must face the situation for which the cost model is designed, must be willing to use it, and allow insight in their data. Particularly, the publication of business data is critical for most organizations [Yoshimura et al. 2006a]. Also, the results gained out of a single case study or with limited information cannot be generalized or transferred to other cases [Ali et al. 2009]. It is difficult to conclude the usability of a model for all or at least some situations by its successful use in single case studies.

In literature, many cost estimation models are not validated on real case studies but historical data [Jørgensen and Shepperd 2007]. This leads to the question, whether the data is even usable for the tested approach. For example, Jørgensen and Shepperd [2007] analyzed several cost estimation studies. They identified that for the usage of data its availability is more important than its usability for the cost model. In addition, they found that the quality is often neither known nor discussed. This can also be explained with the lack of available data [Heemstra 1992; Leung and Fan 2002; Northrop 2002; Knauber et al. 2002; Khurum et al. 2008]. One reason for this is, that most companies handle their information confidential [Yoshimura et al. 2006a; Koziolek et al. 2015].

Experts' knowledge is often used for the validation of cost models. However, their expertise and the reliability of the estimations are often unclear [Bolger and Wright 1994; Jørgensen 2007]. Still, experts can provide important insight into the usability of cost models.

Evaluation in Software-Product-Line Economics

Due to the stated problems, cost models for software product lines are also evaluated with different approaches. For example, Khurum et al. [2008] compared 19 papers about software-product-line economics. Out of those, only six were evaluated with case studies and eight used simplified or fictional data. They also identified a lack of suitable data and replicable studies. How reliable the evaluations are, is questionable because there is no defined and fixed methodology [Ali et al. 2009].

Existing articles often do not provide enough information to replicate the cost estimation with another model. For example, Koziolek et al. [2015] describe cases in which software product lines are extracted. However, they neither provide exact results nor the information our approach requires. Thus, we cannot use this data as we can only guess the parameters for a cost model. We encountered similar problems with other experiments [Clements et al. 2001; Cohen et al. 2002; Mansell 2006; Yoshimura et al. 2006a; Nobrega et al. 2008].

Proposal for Evaluation

We could not apply a case study for our cost model due to time limitations and a lack of suitable participants. While there exist a number of other approaches to evaluate cost models, there is no fixed and reliable methodology [Ali et al. 2009]. Khurum et al. [2008] propose a strategy for a systematic evaluation. At the beginning, the approach is applied in controlled experiments on replicable data. The results are discussed with experts and used for fine tuning of the cost model. Afterwards, the model is suitable for practical evaluation. Thus, industrial case studies can be applied.

Based on this proposal, we evaluate our model with an experiment. We use an existing open-source project with multiple variants to extract realistic information and use them to describe a fictional business case. Then, we estimate the costs for single-system and software-product-line development. We apply our model for the extractive approach

and COPLIMO for the proactive one. Thus, we want to evaluate whether the results resemble our economic descriptions in Chapter 4. We discuss the results in three steps to examine the usability of our approach and possible shortcomings.

First, we compare the estimates of the cost models and check whether the differences are in an expected range. From our economic descriptions in Section 4.2 we derive the following hypotheses.

H-1 If the selected software and business case are suitable, the estimated costs for the proactive approach are lower than for stand-alone development.

With this hypothesis we test, whether our business case and the *Marlin*[1] forks are suitable for our evaluation. Therefore, we compare the costs of developing the existing variants as single-systems and with a proactive software-product-line. If the common asset base results in savings, a product-line is a reasonable scenario for our business case.

H-2 The estimated adoption costs for the extractive approach are lower than for the proactive one.

For the proactive approach the software product line is developed from scratch. In contrast, the extraction benefits from the reuse of existing legacy systems. Thus, the adoption barrier for the extractive approach is lower.

H-3 Extracting only the features that can be reused in the new product reduces the development effort in contrast to stand-alone development.

A company can apply several extraction scenarios. One of those is to only extract the features that can be reused in the new product. Suitable assets are re-engineered to make them reusable. Because not all functionalities must be developed anew, the costs to develop the product decrease.

Second, we discuss the results with experts. We assess the usability of our approach in industrial practice and the credibility of our estimations. Therefore, we use qualitative interviews based on the open-ended questions we present in Table 6.1 [Ballou 2008]. With questions one and two we determine how cost estimation for software product lines is applied in practice and whether the cost models can support this process. With the remaining questions we identify possible shortcomings of our approach. In particular, the experts might provide useful insight into possible improvements we can integrate in further work. We used the sub-questions to focus the discussion on the usability and reliability of our model. However, we do not discuss them separately but within question two and three.

[1] MarlinFirmware/Marlin: https://github.com/MarlinFirmware/Marlin [08.02.2016]

Interview questions	
1	How would you approach the cost estimation for the described business case?
2	Do you think that COPLIMO and our approach are suitable for the scenarios?
2.a	Are the results of the models reliable/acceptable in your opinion?
2.b	Does the automated extraction of data provide benefits for your estimation process?
3	Do the cost factors and parameters match with the costs you used or think are important?
3.a	Which parameters should be more detailed or better adjusted?
3.b	Which cost factors are missing?
3.c	Which parameters are unnecessary or distorting?
4	Did you identify problems with our model?
5	Do you have suggestions for improvements?

Table 6.1: Questionnaire for the experts

Third, we identify reports about case studies in which companies migrate towards a software product line. We compare described costs and estimations with our results. Due to the small number of studies and the lack of provided information those comparisons are not reliable. In particular, we cannot assess how well the situations match with our business case. Still, we can evaluate whether our estimations describe similar trends to those that are reported from industrial practice. For a company a positive perspective is often more important than accurate values to decide for a software-product-line approach [Clements et al. 2005; Koziolek et al. 2015]. However, knowing the costs more precisely enables an organization to improve its planning.

As basis for our evaluation, we use forked open-source software. We describe the selected variants in the next section.

6.2 Marlin Forks

Clone-and-own is an unsystematic but simple alternative to software product lines to develop customized products [Krinke et al. 2010; Ray and Kim 2012; Martinez et al. 2015; Stanciulescu et al. 2015]. In the area of open-source development this can be achieved with software forking [Ray and Kim 2012]. Stanciulescu et al. [2015] analyzed the open-source 3D-printer software *Marlin*. They found, that most of the existing forks are only used to adapt the configuration files. Still, they identified a number of forks that implement new functionalities and have differences in their source code. We picked three of those forks and the master to simulate several variants of existing software for our business case.

In Table 6.2 we summarize the information of the selected variants. Beside *Marlin*, we use two forks, *jcrocholl*[2] and *thinkyhead*[3], that were formally forked. Both are commits ahead and behind of the original software wherefore we assume that they include new features but are also missing others. Instead of cloning the *Marlin* with the provided

[2]jcrocholl/Marlin: https://github.com/jcrocholl/Marlin.git [08.02.2016]
[3]thinkyhead/Marlin: https://github.com/thinkyhead/Marlin.git [08.02.2016]

forking option, *RichCattell*[4] was cloned ad-hoc. Thus, it is an informal fork, which results in missing commit statistics. However, this variant also includes adaptions to the original source code [Stanciulescu et al. 2015]. We identify the commits we selected by providing the first seven characters of the commit hash.

User	Commit ID	Commits		Description
		ahead	behind	
Marlin	5a54204	0	0	Original software
jcrocholl	c295a14	48	2171	Inactive fork
thinkyhead	aad712d	38	29	Active fork
RichCattell	c3bffc2	N/A	N/A	Inactive informal fork

Table 6.2: Overview of the used Marlin forks

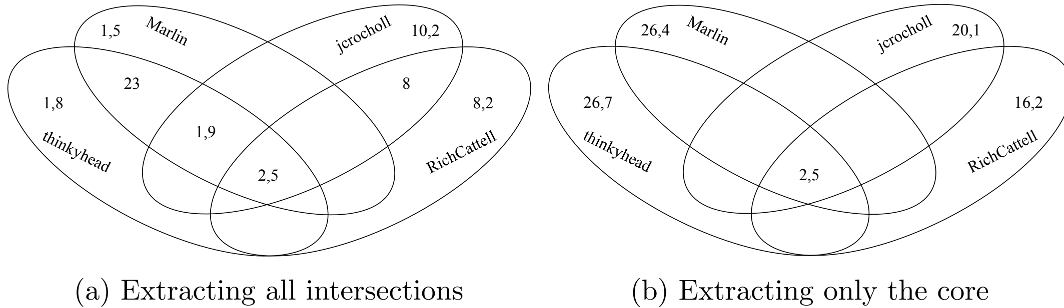
As we stated in Section 5.4.3, it is a challenging task to identify similarities between a set of variants. Using the approach we proposed there, we computed the values presented in Table 6.3. We used the clone-detection tool *Simian*[5] with a threshold of at least 30 identical lines of code and *C++* as programming language. We analyzed all *.cpp* and *.h* files in the main directory. We excluded sub-directories because they contain only example configurations or fonts. In the second column of Table 6.3 we display the overall sizes of identical code for each intersection. This still includes the size for clones that were identified multiple times. For example, the lines of code for *Marlin* include those of all intersections between *Marlin* and the other projects. By subtracting the according values, we calculate the unique sizes we illustrate in column three.

Intersection	Source lines of code	
	Overall	Unique
Marlin	28944	1517
jcrocholl	22618	10166
thinkyhead	29222	1795
RichCattell	18732	8199
Marlin, thinkyhead	27427	22964
RichCattell, jcrocholl	10533	7989
Marlin, jcrocholl thinkyhead	4463	1919
Marlin, RichCattell jcrocholl, thinkyhead	2544	2544

Table 6.3: Sizes of the intersections between the Marlin forks

[4]RichCattell/Marlin: https://github.com/RichCattell/Marlin.git [08.02.2016]
[5]Simian - Similarity Analyser: http://www.harukizaemon.com/simian/ [08.02.2016]

In Figure 6.1 we illustrate the Venn-diagrams for the two scenarios we described in Section 5.4.2. We show the extraction of all intersections in Figure 6.1a. The second scenario, in which only the core assets are extracted is shown in Figure 6.1b. As we assumed, in reality not every intersection contains unique code. However, there are huge similarities between the *Marlin* and *thinkyhead* forks. That can be expected, as both are close in commits as we show in Table 6.2. The other two projects contain more unique code but still contain some similarities. This can be explained with the informality of the *RichCattell* and inactivity of the *jcrocholl* fork. Therefore, the two forks share less code with the other two. The forks might not be best suitable to adapt them all into the same software product line but should still benefit from reuse. Bardo et al. [1996] (cited by Poulin [1997]) also showed that in industrial practice the amount of reuse can vary substantially for each product. Thus, we assume that the forks represent a realistic scenario. Finally, the size of the assets shared among all variants is relatively small in comparison to the total product sizes. Therefore, we do not consider the scenario illustrated in Figure 6.1b for our evaluation.



(a) Extracting all intersections (b) Extracting only the core

Figure 6.1: Venn-diagrams of intersections between Marlin forks. The values are stated in KSLOC. Empty intersections contain no similarities for (a) or are not considered in the scenario (b).

Based on this information, we derive a fictional business case in the next section. We use the values described here for the cost estimation in Section 6.4.

6.3 Business Case

To introduce a software product line, a company will first assess its profitability. Therefore, *business cases* are used. A business case describes a scenario, for example, the introduction of a product line, and assesses the investments and benefits [Brugger 2009]. With the provided information the management can make a reasonable decision. For our fictional business case, we define the following situation:

> A company owns four software systems that were developed with the clone-and-own approach. Over time, maintenance required more and more effort. Because a new variant is planned, the company wants to assess the migration towards a software product line. Thus, an expert is assigned to determine the development and life-cycle costs for several scenarios.

Based on our evaluation proposal in Section 6.1, we consider four possible scenarios for our business case. Each scenario provides cost estimations that we need to test our hypotheses. Furthermore, the results are the basis for our interviews.

S-1 Developing the new variant from scratch.

Developing a new variant from scratch, while a similar product can be reused, is rather theoretical. Still, software-product-line engineering is normally compared to stand-alone development. The estimations also provide the historical costs for the legacy systems. In addition, we need this scenario to test our hypotheses *H-1* and *H-3*.

S-2 Developing the new variant with a proactive software-product-line approach.

We derive this scenario from our hypotheses *H-1* and *H-2*. However, it is unreasonable to develop a software product line from scratch if legacy systems can be reused. Thus, this approach is rather theoretical.

S-3 Developing the new variant within an extracted software-product-line.

This scenario is applied, if a company wants to migrate its legacy systems completely into a product line. Thus, the situation displays a use-case for our cost model and the estimations are necessary to test hypothesis *H-2*.

S-4 Developing the new variant by extracting only the necessary features from legacy systems.

In this scenario only the features that can be reused in the new product are extracted. Thus, the code can be copied and modified only for one new product. For our business case we assume that this resembles the clone-and-own approach. Still, our model is also suitable to estimate costs for this scenario. With the results, we can test hypothesis *H-3*.

Following, we briefly describe the values of the parameters for COPLIMO and our model. Those values are also fictional and do not display a real world situation. However, as we apply the same parameters in both models, the estimations are comparable.

Parameter Values

As legacy systems we use the *Marlin* forks described in the previous section. Thus, we apply the sizes shown in Table 6.3 for asset and unique code. For simplifications we assume that the ratio between white- and black-box reuse is fifty-fifty for all assets. We define the new product with an overall size of 25 KSLOC, which is approximately the average of the *Marlin* forks. We assume, that 10.5 KSLOC of the common code between the legacy systems can be reused. The reusable assets are part of the core and the intersection between the *jcrocholl* and *RichCattel* forks.

In Table 6.4 we summarize the values for the parameters of COPLIMO. We apply values within the ranges defined by Boehm et al. [2004]. In the last column we provide a short explanation about the meaning of each parameter. We defined current and set-point values for *RUSE*, *DOCU*, and *RELY*. For current parameters we use low multipliers to simulate single systems [Boehm et al. 2004]. The set points are the minimum cost drivers for a successful software product line [Boehm et al. 2004]. For example, the *RUSE* factor for one product line is 1.15. As a result, *DOCU* is at least on a very high rating, which is 1.23.

We further base the relative cost of writing for reuse ($RCWR_{adoption}$) for the adoption phase on the difference between current value and set point. Using only those gaps implies, that the whole existing code can be reused the way it is. Only the additional variability must be implemented. However, depending on the re-usability of the code and the software-product-line implementation technique, additional re-engineering is necessary. We found case studies that report of only a few percentages up to a duplication of size for extracted features [Alves et al. 2005; Kästner et al. 2007, 2008; Couto et al. 2011]. For our business case, we apply an additional effort of 0.2 for both factors, $RUSE_{adoption}$ and $DOCU_{adoption}$, that represent the costs of extracting an asset from the legacy code.

In the following section we use the described values and the *Marlin* forks from Section 6.2 to estimate costs for the defined scenarios. Because we use a fictional business case, the results do not represent a real world situation. Still, the estimations of both COPLIMO and our model are comparable as we apply the same parameters.

6.4 Cost Model Estimations

In this section, we estimate the efforts for the described scenarios. First, we calculate the costs to develop the new product as stand-alone software. Afterwards, we estimate the effort to develop a software product line from scratch and by reusing the legacy systems. Finally, we calculate the costs when only the suitable features for the new product are extracted. For each case we also calculate the annual maintenance costs. We round our results to the first decimal place.

Parameter		Rating	Description
A		2.94	For the calibration coefficient we use the standard COCOMO II value.
B		1	We apply a linear scaling factor.
EM		1	We do not consider additional effort multipliers.
RUSE	Current Set point	1.00 1.15	Reuse must be improved from single system to software product line.
DOCU	Current Set point	0.81 1.23	The Documentation must also be improved.
RELY	Current Set point	1.00 1.1	Due to the complexity of product lines, additional validations are necessary.
SU		30	We apply a moderate level for the software understanding.
UNFM		0.2	We assume that the developers are mostly familiar with the software.
AA		2	We apply a basic assessment and assimilation factor.
DM CM IM		10 20 35	The necessary changes for white-box reuse are within the defined ranges.
ACT		0.2	For the annual change traffic we use the same value as Boehm et al. [2004] in their evaluation.
AAF		20.5	The amount of modification is calculated using the parameters above.
AAM		0.25	The assessment and assimilation modifier is calculated using the parameters above.

Table 6.4: Ratings of the COPLIMO parameters for the business case, for details see Boehm et al. [1995, 2004]

6.4.1 Stand-Alone Development

For the stand-alone development (*S-1*) we use the equations provided by COPLIMO. First, we estimate the development costs for the new product as we illustrate in Equation 6.1. To develop the system stand-alone and without reuse, it will require 73.5 person-months. In Table 6.5 we show the efforts of stand-alone development for all legacy-systems. We need them to test our first hypothesis (*H-1*) and because they represent the historical costs that were already spend (compare with Section 4.2.1).

$$
\begin{aligned}
PMNR(n) &= n * A * PSIZE^B * \prod_{j=1}^{m} EM_j \\
PMNR(1) &= 1 * 2.94 * 25^1 * 1 \\
PMNR(1) &= 73.5 \text{ PM}
\end{aligned}
\tag{6.1}
$$

Product	New product	Marlin	jcrocholl	thinkyhead	RichCattell
Effort in PM	73,5	85	66,4	85,8	55

Table 6.5: Effort for stand-alone development of the variants

The life-cycle costs (*PM(n,l)*) of the five stand-alone products for one year is estimated as we describe in Equation 6.2. In this equation, we apply the average size of all products to estimate the overall maintenance effort. Thus, the product size (*PSIZE*) is 24.9 KSLOC.

$$
\begin{aligned}
AMSIZE &= PSIZE * ACT * (1 + \frac{SU}{100} * UNFM) \\
AMSIZE &= 24.9 * 0.2 * (1 + \frac{30}{100} * 0.2) \\
AMSIZE &= 5.3 KSLOC \\
PM(n,l) &= l * n * (A * AMSIZE^B * \prod_{j=1}^{m} EM_j) \\
PM(5,1) &= 1 * 5 * (2.94 * 5.3^1 * 1) \\
PM(5,1) &= 77.9 \text{ PM}
\end{aligned}
\tag{6.2}
$$

To maintain each of the five products individually, the company has to spent 77.9 person months per year.

6.4.2 Proactive Software-Product-Line Development

To estimate the costs of developing the software product line proactive (*S-2*), we again use the calculations provided by COPLIMO. In Table 6.6 we display the ratios of unique (*PFRAC*) and reusable code (*AFRAC* and *RFRAC*) for each product. We apply the code size for each variant independently, instead of predicting the average. By using such details, we can provide a better estimation on which products are more expensive than others.

To fulfill time limitations, a company will most likely use the new variant as basis for the software product line. In Equation 6.3 we show the according calculations. The effort to develop this first product-line instance (*PMR(1)*) is 90.7 person-month, which is an increase by 17.2 in contrast to the stand-alone approach.

Project	Unique	Reuse
New product	0.58	0.42
Marlin	0.05	0.95
jcrocholl	0.45	0.55
thinkyhead	0.06	0.94
RichCattell	0.44	0.56

Table 6.6: Ratios of unique and reusable code for the products

$$
\begin{aligned}
PMR(1) &= PMNR(1) * (PFRAC + RCWR * (AFRAC + RFRAC)) \\
PMR(1) &= 73.5 * (0.58 + (1.15 * 1.23 * 1.1) * (0.21 + 0.21)) \\
PMR(1) &= \ 90.7 \text{ PM}
\end{aligned}
\tag{6.3}
$$

Estimating the effort for additional products is more complicated. Normally, COPLIMO applies the same values for each variant and assumes that all assets are already developed after the first instance. For an estimation without any reliable information, this approach requires few effort. However, we have more details, especially about the amount of code that can be reused for each variant. Thus, we apply the $RCWR$ factor to the parts of the new code ($PFRAC$) that shall be developed as assets and adapt the reuse ratios accordingly, as we show in Table 6.7. We display the adapted calculations for the equivalent code size in Equation 6.4. $REUSE$ represents the percentage of code that will be developed as new assets in the variant. Accordingly, we adapt the values for $RFRAC$ and $AFRAC$.

$$
\begin{aligned}
EKSLOC =& PFRAC * PSIZE + \\
& (REUSE) * PSIZE * RCWR + \\
& AFRAC * PSIZE * \frac{AA}{10} + \\
& RFRAC * PSIZE * AAM
\end{aligned}
\tag{6.4}
$$

We summarize the estimations for all products in Table 6.7. Each product is added to the product line in the order they are listed. For example, when adding the *Marlin* project only the 2.5 KSLOC of the core assets are white- and black-box reusable. Thus, we apply the $RCWR$ factor to the 24.9 KSLOC that can be reused by other products but are not implemented as assets. The effort for *Marlin* exceeds that of the new product. Afterwards, all features are extracted and the remaining variants benefit from that. Thus, the overall effort to develop all variants from scratch within a software product line is 303 person-months.

For the COPLIMO life-cycle model, we first calculate the overall size of code that is unique ($SIZE_P$) and can be white- ($SIZE_A$) or black-box ($SIZE_R$) reused over all variants with the values in Table 6.7. Afterwards, we estimate the size of code that must be maintained per year as we show in Equation 6.5.

Project	PFRAC	REUSE	AFRAC	RFRAC	EKSLOC	Effort in PM
New Product	0.58	0.42	0.21	0.21	25	90.7
Marlin	0.05	0.86	0.045	0.045	40.7	119.7
jcrocholl	0.45	0	0.275	0.275	13	38.2
thinkyhead	0.06	0	0.47	0.47	7.9	23.2
RichCattell	0.44	0	0.28	0.28	10.6	31.2

Table 6.7: Cost estimations for the remaining products

$$\begin{aligned} AMSIZE =& (SIZE_P + SIZE_R) * ACT * (1 + \frac{SU}{100} * UNFM) + \\ & SIZE_A * ACT * (1 + \frac{SU}{100} * UNFM) * [1 + \frac{AAF}{100} * (N-1)] \\ AMSIZE =& (36.2 + 17.7) * 0.2 * (1 + \frac{30}{100} * 0.2) + \\ & 17.7 * 0.2 * (1 + \frac{30}{100} * 0.2) * [1 + \frac{20.5}{100} * (5-1)] \\ AMSIZE =& 18,3 \text{ KSLOC} \end{aligned} \tag{6.5}$$

Finally, we calculate the effort for one year by multiplying the COCOMO II calibration coefficient (A). Thus, we get an effort of 53.8 person-month to maintain the product line every year. Because the model assumes that the used parameters remain stable, the efforts for the following periods are estimated with the same amount.

6.4.3 Extractive Software-Product-Line Development

To calculate the effort of extracting all assets (*S-3*), we first have to adapt the relative costs of writing for reuse. As basis for the calculation we use the difference between the current $RUSE$ and $DOCU$ parameters described in Table 6.4 and the set points. To consider necessary changes in the existing code of assets, we add 0.2 to both parameters. Thus, the adopted relative cost of writing for reuse is approximately 0.24. As we stated in Section 5.2.1, the $RELY$ factor is the same as for the proactive approach to consider necessary tests for the new development approach. In Equation 6.6 we illustrate the calculation for the intersection between all variants, which represents the core assets.

$$\begin{aligned} PM(asset_{adoption}) =& A * aSize^B * RCWR_{adoption} * \prod_{j=1}^{m} EM_j \\ PM(CAB) =& 2.94 * 2.5^1 * [(1.15 - 1 + 0.2) * 1.1 * \\ & (1.23 - 0.81 + 0.2)] * 1 \\ PM(CAB) =& 1.8 \text{ PM} \end{aligned} \tag{6.6}$$

Intersection	KSLOC	Effort in PM
Marlin, thinkyhead	23	16.1
RichCattell, jcrocholl	8	5.6
Marlin, jcrocholl thinkyhead	1.9	1.3
Marlin, RichCattell jcrocholl, thinkyhead	2.5	1.8

Table 6.8: Estimated effort to extract the intersections

In Table 6.8 we summarize the efforts to migrate each intersection into assets. Extracting all features requires 24.8 person-months.

Additionally, we need the costs to develop the new product and instantiate the legacy systems from the product line. In each case, we first estimate the amount of code that must be developed or adapted for a product ($EKSLOC$). In Equation 6.7 we illustrate the size prediction for the new product.

$$\begin{aligned} EKSLOC &= unqiqueSize + \sum_{i=1}^{n}(aSize_{bb,i} * \frac{AA}{10}) + \sum_{i=1}^{n}(aSize_{wb,i} * AAM) \\ EKSLOC &= 14.5 + (\frac{10.5}{2}) * \frac{2}{10} + (\frac{10.5}{2}) * 0.25 \\ EKSLOC &= 16.9 \text{ KSLOC} \end{aligned} \tag{6.7}$$

Based on this result, we can estimate the development effort (*PM(product)*) as illustrated in Equation 6.8. According to our model, the development effort for the new product is 49.7 person-months.

$$\begin{aligned} PM(product) &= A * EKSLOC^{B} * \prod_{j=1}^{m} EM_j \\ PM(product) &= 2.94 * 16.9^{1} * 1 \\ PM(product) &= 49.7 \text{ PM} \end{aligned} \tag{6.8}$$

For the existing variants we apply the unique code size as white-box reuse. Thus, we want to consider necessary changes to integrate variability. We display the effort to instantiate each variant in Table 6.9. For the overall costs we sum up all efforts of developing the assets and products. Thus, for our example our model assumes that it requires 141.5 person-months to extract a full software product line and develop the new product.

Product	KSLOC		EKSLOC	Effort in PM
	Unique	Reuse		
New Product	14.5	10.5	16.9	49.7
Marlin	1.5	27.4	6.5	19.1
jcrocholl	10.2	12.4	5.3	15.6
thinkyhead	1.8	27.4	6.6	19.4
RichCattell	8.2	10.5	4.4	12.9

Table 6.9: Estimated effort to extract the variants

For the life-cycle costs, we again first estimate the amount of code that is maintained per year. Therefore, we sum up the according sizes of each asset and product unique part. In Equation 6.9 we illustrate the calculation for the core asst.

$$\begin{aligned}
AMSIZE_{bb} &= size * ACT * (1 + \frac{SU}{100} * UNFM) \\
AMSIZE_{bb} &= \frac{2.5}{2} * 0.2 * (1 + \frac{30}{100} * 0.2) \\
AMSIZE_{bb} &= 0.3 \\
AMSIZE_{wb} &= size * ACT * (1 + \frac{SU}{100} * UNFM) * (1 + \frac{AAF}{100} * p_{applied}) \\
AMSIZE_{wb} &= \frac{2.5}{2} * 0.2 * (1 + \frac{30}{100} * 0.2) * (1 + \frac{20.5}{100} * 4) \\
AMSIZE_{wb} &= 0.5
\end{aligned} \tag{6.9}$$

Our model predicts the annual size of maintained code with 16.8 KSLOC. Therefore, maintenance requires an effort of 49.4 person-months per year. This is slightly lower than the estimation by COPLIMO because we consider how many products use an asset (compare with Section 5.2.3).

6.4.4 Clone-and-Own Development

For the fourth scenario (*S-4*), in which we only extract the features that are necessary for the new product, we can use our previous calculations. We can use the costs to develop the new product and extract the required assets from our estimations for the full extractive approach. The accumulated effort is 57.1 person-months.

The life-cycle effort for the legacy systems, we use the same calculations as in the first scenario (*S-1*). For the new variant, we apply its maintenance costs and those for necessary assets from scenario three (*S-3*). With this approach, we estimate an effort of 79.1 person-months. This is slightly higher than with stand-alone development because we assumed, that the cloned code is extracted into assets.

6.5 Discussion

In this section, we discuss the results of our business case we calculated in the last section. Therefore, we analyze our estimates with the approach we proposed in Section 6.1:

1. We compare the estimates and test our hypotheses for a first interpretation of the usability of our business case and cost model.

2. We discuss our results with experts that have knowledge about cost estimations and software product lines to validate the industrial practicability and identify possible improvements.

3. We compare the benefits our model predicts, with those reported in industrial case studies.

In Table 6.10 we provide an overview of the efforts we estimated for the three scenarios. We can see, that the models estimate the reported benefits of reduced development and maintenance effort for the software product lines [Knauber et al. 2002; Pohl et al. 2005, pp. 9-13; Clements and Northrop 2006, p. 17; Yoshimura et al. 2006a Apel et al. 2013, pp. 8-10; Martinez et al. 2015]. The historical costs that the company has already spend are highlighted in gray. We later apply the trend average in cost curves to illustrate a possible scenario for the future. In both software-product-line approaches, we used the variants that only benefit from the systematic reuse and do not add new features or are extracted (*S-2*: jcrocholl, thinkyhead, RichCattell; *S-3*: New Product).

Artifact	Estimated effort			
	Stand-alone *S-1*	Proactive SPL *S-2*	Extractive SPL *S-3*	Clone-and-Own *S-4*
New Product	73.5	90.7	49.7	49.7
Marlin	85	119.7	19.1	N/A
jcrocholl	66.4	38.2	15.6	N/A
thinkyhead	85.8	23.2	19.4	N/A
RichCattell	55	31.2	12.9	N/A
Assets	N/A	N/A	24.8	7.4
Sum	365.7	303	141.5	57.1
Maintenance per year	77.9	53.8	49.4	79.1
Trend average	73.1	35.6		57.1

Table 6.10: Summary of the estimated efforts in person-months.

Following, we interpret and discuss the results presented in Table 6.10. We argue, that our model is usable in practice and provides reasonable estimations.

6.5.1 Interpretation and Hypotheses

In Figure 6.2, we illustrate the cost curves for the proactive software-product-line approach and single-system development, based on Table 6.10. We can see, that systematic reuse pays off after four developed products. While the first and second variant require more effort, the remaining products benefit from the assets. The overall savings for development with a software-product line are approximately 17.4%. Thus, we assume that our business case and the *Marlin* forks represent a suitable product-line scenario. This confirms our first hypothesis:

H-1 If the selected software and business case are suitable, the estimated costs for the proactive approach are lower than for stand-alone development.

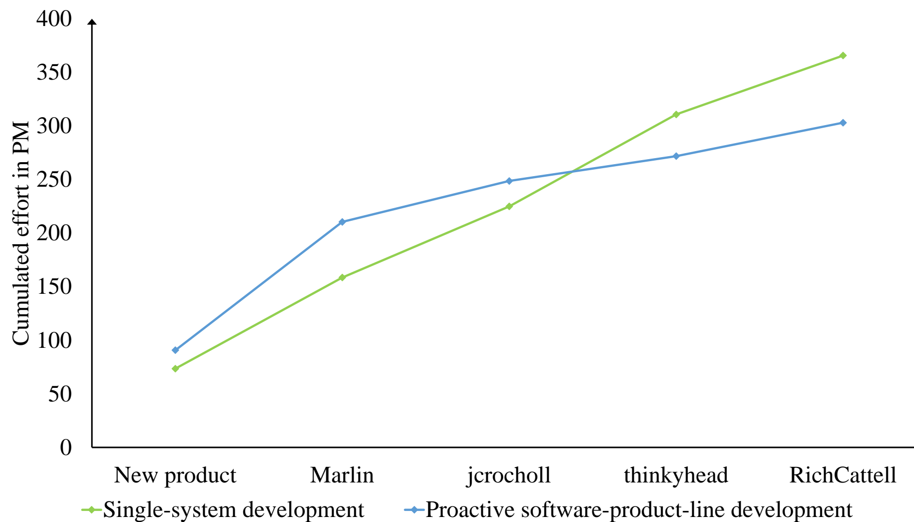


Figure 6.2: Effort for proactive software-product-line and stand-alone development for the business case

In Figure 6.3, we compare the estimated development effort for the new product and the maintenance costs of each scenario. As we can see, the cost models follow our descriptions in Chapter 4. First, developing a new variant from scratch costs approximately half as much as extracting a whole product-line. Savings can only occur due to savings in the life-cycle or because multiple new products can benefit from the systematic reuse. Second, the adoption costs for the proactive approach are estimated with 303 person-months. In contrast, extracting a product line requires only 141.5 person-month. Thus, we confirm our second hypothesis:

H-2 The estimated adoption costs for the extractive are lower than for the proactive approach, as we can reuse existing artifacts.

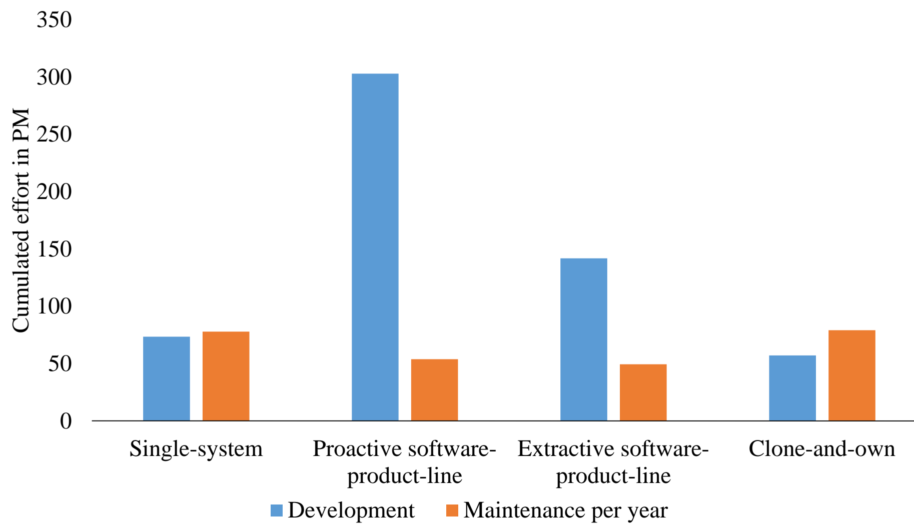


Figure 6.3: Comparison of the development and life-cycle efforts for the scenarios

Summarized, we think that our model estimates reasonable costs for the extraction of software product lines. The predictions are higher than developing one stand-alone product and also lower than a proactive product line.

The life-cycle costs for systematic reuse are lower than for the single-system or clone-and-own approaches. However, they provide lower development efforts. Thus, a company which possesses legacy systems and must consider strict schedules or resource limitations will perhaps not extract a product line. An organization will only consider this as possible scenario, if it is facing huge maintenance problems or is convinced that the product line provides benefits in the near future. Normally, a company may clone an existing variant and modify it for new customers, as this is the least expensive development method. In contrast to beginning from scratch, this represents savings of around 22.3% for our business case. Thus, we can confirm our third hypothesis:

H-3 Extracting only the features that are required for the new product reduces the development effort in contrast to starting from scratch.

We estimate the efforts for this hypothesis with our cost model. Because we think that the described results are reasonable, we assume that our model calculates reasonable values. Moreover, we think, that our approach is also suitable for the incremental extraction of software product lines.

The benefits of extracting a software product line from legacy variants can only occur during the life-cycle or when a greater number of new products shall be integrated. We compare the development efforts for all approaches in Figure 6.4. The curves resemble those we derived from our economic descriptions in Section 4.2.1. According to our predictions, the extraction pays off after three products in contrast to stand-alone development and five for clone-and-own respectively. As we stated, the proactive

software product line is of theoretical nature. Our estimations confirm this. The development of a new product-line is only beneficial if more than seven new variants will be implemented.

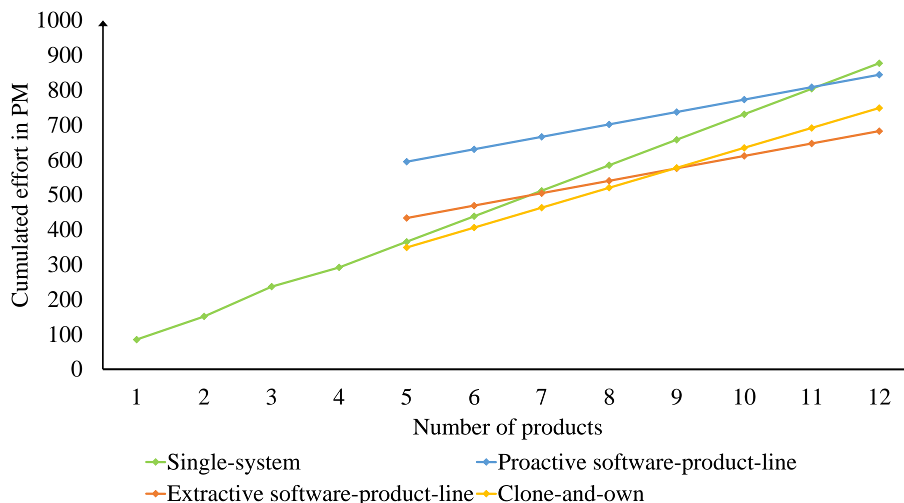


Figure 6.4: Accumulated development effort for the business scenarios in comparison. Every mark represents the development of a product and the four at the beginning are the existing legacy systems.

In Figure 6.5 we illustrate the cost curves for the life-cycle of each scenario. As we can see and expected, the proactive development is again the worst scenario. It would only pay off after ten years. Planning for such a long period is hardly usable in practice. Companies can scarcely predict what their product portfolio requires after that time. In contrast, extracting a software product line pays off after four years. This is still a long duration but with each new variant the savings will increase, wherefore it might be a good alternative.

Conclusion

In conclusion, the trends in our cost model are the same as in other estimation approaches for software product lines. We also showed that our model follows the economic descriptions we introduced in Chapter 4. Thus, we assume that our model predicts reasonable results.

We further support this statement in the next section. Therefore, we applied qualitative interviews with experts.

6.5.2 Interviews

For an industrial evaluation we interviewed two experts, who have experiences with cost estimations and software-product-line development. In Table 6.11 we summarize the information about them. They work in different positions at a German software

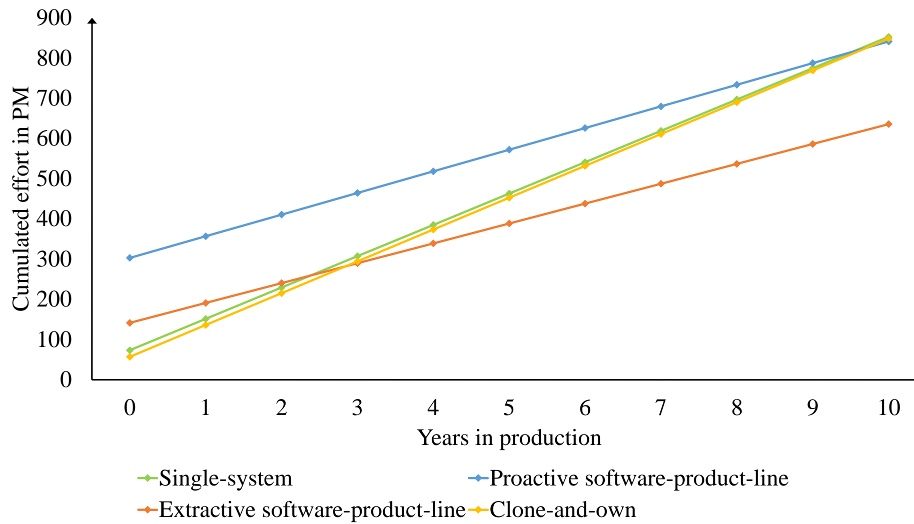


Figure 6.5: Accumulated maintenance effort for the business scenarios in comparison. Each mark represents one year, beginning with the development efforts.

development and consulting company. For privacy reasons, we refer to them with literals. We provided the described business case, cost estimations, and questionnaire, to each of them.

Expert	Position	Interview protocol
A	Principal IT-consultant	Section A.1
B	Division manager applied computer science	Section A.2

Table 6.11: Interviewed experts

The experts' answers are translated and approved by them. Following, we discuss their responses.

1. How would you approach the cost estimation for the described business case?

From the responses, we conduct that the interviewed experts rely on different estimation approaches. For example, for stand-alone products, *Estimator A* uses historical data for his prediction. This reflects an analogy (see Section 2.3) [Boehm 1984]. In contrast, for the proactive approach, he uses COPLIMO. In particular, he states that he would not consider the extraction of the whole product line only for developing a new variant. Rather, he identifies the legacy system which matches the new requirements the best and refines it. This represents a clone-and-own approach and our fourth scenario (*S-4*).

To estimate the effort for a software product line, *Estimator B* applies a manual method and, thus, relying on his judgment. He uses a transparent approach so that it is explainable to the management. In a first step, he analysis the existing files for commonalities

to identify candidate features. Then, he determines which parts are often and recently changed. For the features that are constantly modified in several variants, which therefore require multiple changes, he estimates the extraction costs. Finally, he compares his estimations with the approach currently in practice at the company.

In conclusion, we think that our approach can support the methods of both estimators. We propose the usage of different scenarios, which can be matched with the one *Estimator A* would apply in the first place. *Estimator B* applies a structured approach, that is similar to ours. In contrast to him we do not consider the change frequency to identify possible features. We think that this can be a suitable addition to automatically identify features whose extraction provides benefit.

2. Do you think that COPLIMO and our approach are suitable for the scenarios?

Both estimators stated, that COPLIMO, as well as our approach, are suitable for the described scenarios and provide reasonable results. They also think that the automatic extraction of information from legacy variants is an important task, which can reduce the analysis effort and time. According to *Estimator B*, it might be problematic that COPLIMO does not clearly separate the costs for the assets and the actual products. In this regard, he thinks that our model is an improvement. He also states, that it might be to detailed for some projects, as currently "the reported benefits outweigh the costs multiple times" for software product lines.

3. Do the cost factors and parameters match with the costs you used or think are important?

In the opinion of both estimators the considered parameters are reasonable. For *Estimator A*, "the quantification of a company's real situation is problematic". This is a major challenge for any cost model [Jørgensen 2007; Jørgensen et al. 2009]. However, no estimation approach can overcome this difficulty (compare with Section 2.3) but we try to address it by also supporting judgment-based methods (see Section 3.2). This might be supported with the statements of *Estimator B*, who thinks that our model provides a more detailed and systematic approach.

Still, there are some parameters, that should be addressed in addition. Besides the previously described change frequency, both estimators think that a more detailed consideration of risks is necessary. We address this in Section 4.2 and with our cost model a company is able to run multiple scenarios to consider risks. An additional focus on this topic or adding a suitable parameter to our calculations might be important. Especially, because considering risks is a challenging task even for experts as stated by *Estimator B*.

Another factor that we may have to address in more detail are the organizational costs (see Section 4.2.4). In particular, impacts of the shift in development and processes require more regard. *Estimator B* proposes to additionally consider change-management

and controlling. Switching towards software product lines is a challenge and the organizational change can be high. Resulting costs must be estimated and assigned to make reasonable and strategical decisions. As we described, organizational costs are difficult to include into a cost model. Further industrial investigations are necessary. Still, our model can only benefit from including such costs to our economic descriptions.

4. Did you identify problems with our model?

The main problem for *Estimator A* are the previously stated risks a company faces when introducing a software product line. He proposes further investigations on the impacts that the changing development approach has. The modeling of variability, increasing complexity, and resulting side effects are important to consider. Extending our approach in this regard, provides additional details. However, it is problematic to integrate all those aspects within a cost model and its complexity would also increase (compare with Section 3.2). Still, we should extend our economic descriptions in this regard.

A problem addressed by *Estimator B* is the usage of lines-of-code as size metric. As he states, “applying them on a project with multiple programming languages is difficult” because they might require different amounts of code for the same functionality. We described this problematic in Section 2.3 and a solution to overcome this, can be to include other size metrics, such as, *function points*. This is possible with COCOMO II [Boehm et al. 1995, 2000b] and thus may be integrated in our approach as well. However, all size metrics have the problem that they are only used as predictions until the products are developed [Leung and Fan 2002]. We focused on lines-of-code because we can extract approximations from the legacy systems and they are applied in COPLIMO.

Estimator B also stated that our detailed model “might convey the feeling of reliability and viability”. We are not sure on how we can address this, except repeating, that cost estimations can only predict possibilities. No model can estimate the required effort completely accurate and only while the development process and life-cycle, reliable values can be determined [Boehm 1984].

5. Do you have suggestions for improvements?

Summarizing the previous answers, we can conclude some possible improvements. First, our model can support several and incremental scenarios. A focus on those increases the practical usability. Second, there are some factors, for example, organizational costs, that require more attention. Even if they cannot be added to the calculations, a suitable framework is helpful. Finally, both estimators state that tool support, not only for the extraction itself but also the cost estimations, is necessary. Thus, an implementation of our approach would be helpful.

Conclusion

Based on the responses, we further support the statement that our cost model estimates reasonable values. In addition, the experts stated that our approach is suitable for industrial practice. We also found additional parameters and costs that can be considered

in further work. The main point we have to address is the development of a suitable tool.

In the next section we provide additional clues that the estimations of our model are reasonable. Therefore, we analyzed case study reports that describe the transition from legacy-systems towards software product lines.

6.5.3 Comparison with Case Studies

We found a small number of case studies that deal with cost estimations for software-product-line development. However, the information we identified do not allow a repetition of the cost estimation process. Instead, we briefly describe some of the reported results and compare them to our estimations. The comparison of these values is not reliable and can only provide clues about the accuracy of our cost model. In particular, we are missing important details, for example, about code sizes, re-usability, or the organizational situation. Often it is also not clear whether a proactive or extractive development approach was applied.

Koziolek et al.

Koziolek et al. [2015] analyzed four industrial product sets about the potential to extract a software-product-line. In two cases they were able to describe a business case with SIMPLE [Böckle et al. 2004; Clements et al. 2005]. In Table 6.12 we summarize some information they provide about the analyzed applications.

Case	Systems	Number of products	Lines of code
1	Stand-alone personal computer tools	4 products and 2 product-families of 10	300 K - 2 M
2	Desktop applications	2 products and a product-family of 5	10 K - 100 K

Table 6.12: Selection of case study results from Koziolek et al. [2015]

For both product sets, it was estimated, that developing the core asset base (C_{cab}) requires 150% of the effort for developing an individual product. As our model focuses on the extractive approach, we cannot apply values for developing a new core asset base in contrast to new variants. Thus, we compare the costs of extracting the assets from the legacy systems and instantiating the products from the software product line. This way, we consider the re-usability of existing artifacts. For our business case, the effort to extract all assets costs on average 152.3% of the instantiation costs for the legacy systems within the software product line. Those values are close to each other but might be difficult to compare due to the different approaches.

For the first case, Koziolek et al. [2015] also predicted that the costs of reuse (C_{reuse}) are approximately 12% of those to develop all products individually. The estimations of our model and business case predict a similar value with about 15.9% for all variants.

A positive return on investment was predicted for both cases with different scenarios. The first set of systems was predicted to pay off after three years and the second after seven years at the latest. As we illustrated in Figure 6.5 the extraction of the software product line for our business case would also pay-off after three years. Thus, our model predicts a similar duration.

Clements et al. and Cohen et al.

Clements et al. [2001] report a case study for a software product line of systems for spacecraft command and control. They estimated 18.2% of development and 27.8% of maintenance savings for each new variant. However, for the first new system the company reported a 50% reduction of development effort. Cohen et al. [2002] describe the development for a software product line. For the cost estimation they used the approach by Cohen [2003]. The practical savings for the software-product-line development were about 50% in contrast to stand-alone development.

In our business case, we estimate savings of 32.7% for developing the new product with an extracted product line instead of a single system. This is higher than the estimation done by Clements et al. [2001], but lower than the real benefits in both studies. However, this can be explained with the degree of reuse. Cohen et al. [2002] predicted that 70% of the code for each new product is provided by the asset base. In our scenario, the new product reuses considerably less parts of the assets. Thus, the 32.7% our model predicts seem to be a reasonable value.

Mansell

Mansell [2006] reports case studies about the introduction of software product lines in five small- and medium-sized companies. Their cost estimations of investments and savings are highly varying for each organization. On average they calculated a return on investment of 3.07 for the development. The return on investment is defined as the ratio between savings and investment [Böckle et al. 2004]. For our business case the question is, which estimates we use for those two values. We assume that the savings are the difference between developing the new product stand-alone (*S-1*) and in an incremental (*S-4*) software product line. As investments we use the costs of extracting features. Thus, considering only the assets necessary for the new variant we calculate a return on investment of 2.2. This is lower than the average estimated by Mansell [2006]. However, they state that there is a high variation of costs and savings. Thus, the return on investment also varies. Therefore, we assume that our estimation is reasonable.

McGregor et al. and Pohl et al.

It is often reported, for example, by McGregor et al. [2002] or Pohl et al. [2005, pp. 9-10], that software product lines pay off after approximately three new variants are developed. Because of the used product-line development approach, the benefits can occur later [McGregor et al. 2002]. Also, previously applied reuse strategy and the degree of similarity between variants can result in a later pay-off. For our business case, we estimated the break-even point for three products if compared with single-system development and five for clone-and-own. This matches with the practical reports.

Summary

In the case studies and practical reports, we found clues that our cost model predicts reasonable values. However, it is difficult to compare the estimations because we cannot validate the organization's situation and have too few information. Thus, this interpretation must be considered carefully. Nonetheless, we think that the described studies support our previous results. We conclude, that the estimations of our cost model are credible.

6.6 Threats to Validity

In our evaluation we applied a controlled experiment and analyzed the results based on proposed hypotheses, interviews with experts, and comparison with case studies. Thus, we aimed to consider both, internal and external validity [Siegmund et al. 2015]. In this section we distinguish on threats to our cost model itself and to our evaluation.

Threats to our Cost Model

Like any model, our approach can also only consider a set of parameters. Those might be unnecessary or important factors are missing. Especially, an organization's situation and human factors are difficult to consider [Jørgensen 2007]. This can highly disturb the estimation. We described in Section 2.3 other general problems of cost models that also apply for our approach. For example, the usage of lines of code as size measure and dependency on experts' knowledge can threaten the usability.

Our approach is based on existing cost models and reports. In most cases, economic approaches and studies are rarely evaluated in a structured way [Khurum et al. 2008; Ali et al. 2009]. It is difficult to validate their usability and reliability. Thus, we can also hardly ensure the quality of our approach.

There are several benefits that we do not consider in our model but that can be important for companies. For example, we do not consider the time-to-market or product quality. If a company focuses on those, our model must be refined and adapted. Defining an extendable framework would be helpful.

Threats to the Evaluation

Our business case does not represent an industrial case study but a fictional scenario. While it is a controlled experiment, we have no other cost model that considers the extractive software-product-line approach. Additionally, we have no reliable value for the effort that would occur in reality. Thus, it is problematic to evaluate the internal validity [Siegmund et al. 2015].

For our business case we use forked open-source software which can significantly differentiate from industrial systems [Stanciulescu et al. 2015]. For instance, the motivations of the developers are different. Due to this contrast, our scenarios must be considered

with care. However, we apply all cost models on the same projects. Thus, comparing the results should be reliable.

As we already stated, the comparison of our estimations with reported case studies is difficult. A replication of the estimation process with our model is not possible, because the data is only partly available. Thus, we cannot determine values for our parameters, such as the number of products, code, size, similarities, or reuse potential.

Discussing the results with experts provides important and practically relevant input. However, each interview presents a subjective opinion influenced by personal experiences and knowledge. In addition, we only have a small and regional selection of experts wherefore the results are not representative.

By combining a controlled experiment with the assessment of experts that participate in industrial projects, we consider internal and external validity [Siegmund et al. 2015]. However, to evaluate the practical usability of our approach, case studies are necessary. Only by comparing estimated and real efforts in practical projects can assess the reliability and improvements of a cost model. Thus, multiple case studies that focus on external validity are better suited.

6.7 Summary

In this chapter, we described an evaluation method for our approach based on the proposal of Khurum et al. [2008]. Thus, we defined a fictional business case for a controlled experiment. On this, we applied COPLIMO and our approach. We estimated the efforts for four different scenarios. In our evaluation, we first compared and discussed the results. We tested hypotheses about the outcome that we derived from our economic descriptions in Chapter 4. In a second step, we interviewed experts to evaluate whether our approach is of use for the industrial practice and provides a reliable method. Additionally, we determined benefits, shortcomings, and improvements for our approach. In a last step, we compared our results with reported estimations and benefits from case studies. Finally, we summarized the threats to validity of our cost model and evaluation.

In conclusion, the estimations of our cost model were in the range we expected. Additionally, the interviewed experts confirmed its usability in industrial practice. They think that the estimation of our model are in a credible. In the case studies we also found clues that the estimations are reasonable. Thus, we conclude that our approach is appropriate for the extractive software-product-line approach. In addition, the costs can be estimated more precise than with other cost model, such as COPLIMO.

7. Conclusion

Software product lines can be used to reduce the effort of developing, managing, and customizing multiple variants. Before those benefits can be achieved, upfront investments are necessary [Pohl et al. 2005, p. 9; Apel et al. 2013, p. 9]. Thus, companies must plan in advance whether it saves costs to introduce a product line in contrast to other reuse strategies or stand-alone development. To estimate the efforts of a project, multiple possibilities, such as, judgment- or model-based approaches, exist [Boehm 1984]. For software-product-line engineering, research focuses on proactive development, which is considered as ideal strategy [Clements 2002; Schmid and Verlage 2002]. However, in industrial practice the extractive approach, which reuses legacy systems, is more relevant [Schmid and Verlage 2002; Pohl et al. 2005, p. 201; Duszynski et al. 2011; Berger et al. 2013; Koziolek et al. 2015].

In this thesis, we introduced a cost model that focuses on the extraction of software product lines from legacy systems. We used two levels to support judgment- and model-based cost estimations. On the first level, we described benefits of legacy systems on the development of product lines with an economical point of view. From this we derived a cost model that can calculate concrete values and supports multiple scenarios. Additionally, we investigated possibilities for automatic extraction of information from legacy systems.

7.1 Contributions

In Chapter 3 we presented an overview of existing cost models for software product lines. Based on requirements we described for our cost model, we derived criteria to identify approaches that were suitable for adaptions towards the extractive approach. With further analysis we identified two models, SIMPLE [Böckle et al. 2004; Clements et al. 2005] and COPLIMO [Boehm et al. 2004], that we used as basis for our work.

We derived economic descriptions for the extractive approaches from the SIMPLE model in Chapter 4. Therefore, we refined the provided cost functions and investigated benefits

of legacy systems for the development of software product lines. Thus, we provide support for judgment-based cost estimations. Finally, we matched our descriptions with economic theories.

Within Chapter 5 we described our cost model. First, we introduced COPLIMO and compare the considered efforts with those of SIMPLE to identify connections. Then we refined the calculations for the extraction approach and our economic descriptions. Additionally, we separated the costs to consider single assets instead of full products. Afterwards, we connected our cost model to the software-product-line development process. In the second part, we investigated possibilities to automatically extract information from legacy systems. We focused on the code size and similarities between the variants as they have the biggest impact and suitable tools exist. With those information, the cost estimation is based on reliable data and the analysis effort can be reduced.

We presented our evaluation in Chapter 6. We first described why cost models are difficult to validate, especially, without industrial case studies. We proposed a method based on the idea of Khurum et al. [2008]. Afterwards, we introduced the open-source software we used and our fictional business case. Based on those, we applied our cost model and calculated the efforts for different scenarios. Our discussion consisted of three steps. First, we discussed the results and tested hypothesis we derived from our economic descriptions in Chapter 4. Second, we interviewed experts to validate the usability of our approach. Finally, we investigated industrial case studies and reports to identify clues about the reliability of our cost model. We closed with a summary of threats to validity for our approach and evaluation.

Results

In our first evaluation step, we tested different hypotheses we defined from Chapter 4. With those, we checked whether the estimations of our model are in a range we can expect for the extractive approach. We showed, that the adoption costs are lower than for proactive development. We expected this, because legacy systems are reused during the extraction. In contrast, the overall effort, including the historical costs, is considerably higher than for a product line that is developed from the beginning. This is logical because the extraction is a re-engineering process that only increases the effort. Extracting only the features necessary for the new products represents a clone-and-own or incremental product-line approach. Because we reuse existing artifacts, we assumed that the costs for the new product must be lower than with stand-alone development. Our model also predicts this.

Second, we discussed our business case and estimations with experts. In our interviews, both experts approved that our model is suitable for practical usage. Also, they state that the estimations are within a reasonable range. Finally, we identified additional parameters we can consider in the future.

Finally, we compared our results with cost estimations and reports in industrial case studies. This comparison must be interpreted critical and carefully. We were not able

to repeat any of the reported estimations due to a lack of data. Thus, we used relative coherence between different costs, which is not reliable. However, we think that we were able to identify some clues that our cost model's estimations are reasonable. For example, we found that the costs for reuse and the break-even point are similar in our business case to some studies.

Based on those results, we think that our approach is suitable to support cost estimations for the extractive software-product-line approach. Unfortunately, we were not able to conduct an industrial case study, due to time limitations and missing participants.

7.2 Future Work

During our work, we identified several topics for further research.

Evaluation of our Cost Model

To improve and further evaluate our cost model, practical case studies are necessary. As we stated in Section 6.1, such studies are necessary for a realistic evaluation. In each step the results must be interpreted and discussed with experts and participants to validate the impact of multiple cost factors. Thus, the parameters and calculation we provide in Section 5.2 can be adjusted to improve the accuracy or identify extensions.

Refining and Implementing our Cost Model

As the experts described, our approach can be extended with additional parameters and more detailed descriptions. However, they also stated that the most important task is to consider multiple scenarios. Therefore, tool support is essential. An implementation of our cost model can also be combined with feature models and thus increase the comprehensibility.

Evaluation Methods for Software Economics

During our work, we found a lack of structured evaluation methods for software cost estimations. While a set of industrial case studies might be the best validation, it is questionable whether the results are comparable for different models and companies. Thus, developing systematic evaluation approaches for cost models can help to determine more reliable results and identify problems before introducing them into practice.

Software-Product-Line Economics

For software and especially in software-product-line engineering the amount of case studies is rare. In our opinion, industrial case studies and surveys are necessary to further investigate the economic impacts of software product lines. On the one hand, they can help to identify cost factors and benefits during software development. Thus, it is possible to estimate and proof under which circumstances product-line development is useful. On the other hand, detailed data makes it possible to derive and apply new or adapt existing cost models. Additionally, such information enables researchers to evaluate and compare cost estimation approaches on a reliable basis.

Variability Mining

As we stated in Section 5.4, there are approaches to identify commonalities and evaluate re-usability potential. The interviewed experts clearly stated that variability mining is important to provide reliable data and decreases the analysis effort. However, there are two major challenges that may be addressed in further research. First, most of the approaches do not consider multiple legacy systems. As the extraction of software product lines is only beneficial with a set of similar products, this situation requires more attention. Second, there is a lack of suitable and usable tools. To simplify the transition towards systematic reuse strategies and practical application, further research on automated analysis of multiple legacy systems is necessary.

A. Appendix

A.1 Interview Protocol Expert A

Question 1:

How would you approach the cost estimation for the described business case?

Answer:

For the first scenario I would rely on experience and historical data. Based on the reported costs for previous products an estimation can be derived. In scenario two I would apply COPLIMO. However, developing a software product line for a domain from scratch while suitable products already exist can only be considered as a theoretical possibility. The same counts for scenario three if the goal is to develop only a new product. In this case I would not consider a migration of all legacy systems into a product line. This is often not suitable as the shift in the companies processes would be immense. Instead, I think using scenario four, in which only the necessary artifacts from one or more variants are cloned, is appropriate.

Question 2:

Do you think that COPLIMO and our approach are suitable for the scenarios?

Answer:

As I said, COPLIMO is suitable for the development of a new software-product line. For the migration of legacy systems the approaches presented in your cost model are suitable. Especially, the automatic extraction of information is important. This is the start and premise for any reliable for any cost estimation. Providing clues about the overlapping between the existing variants helps to identify reusable artifacts.

Question 3:

Do the cost factors and parameters match with the costs you used or think are important?

Answer:

Overall, the cost factors and parameters that are considered in the models are suitable. However, the quantification of a company's real situation is problematic. It is not clear, which impact each value has.

Question 4:

Did you identify problems with our model?

Answer:

I think that a consideration of uncertainties a company faces when extracting a software product line is missing. I would like a more detailed view on the risks, such as, side effects of a complete extraction, resulting complexity, or migration and modeling efforts, for the extractive approach.

Question 5:

Do you have suggestions for improvements?

Answer:

For a first impression of the benefits of a software product line the model is to complex. This could be helped by a increased focus on incremental scenarios. Your model already supports this as it is possible to estimate efforts for the extraction of subset of features. Providing detailed support for different migration scenarios, for example, only one or two variants, can improve your model. Comparing those different possibilities is important for industrial practice to consider which legacy systems are suitable for adaptions. Additionally, this would also support estimations for the clone-and-own approach. However, as I said, your model already is able to do this, but a focus on those scenarios would considerably increase its practical utility.

A.2 Interview Protocol Expert B

Question 1:

How would you approach the cost estimation for the described business case?

Answer:

To predict whether a software product line provides benefits for a company, I analyze the existing legacy systems. First, I determine which files are similar over all variants. Thus, I can approximate the size of common and variable parts. Second, I consider the historical and current change frequency of similar files. Those are candidates for extraction into features to decrease the code size that must be maintained. Finally, I would estimate the costs for extraction and maintenance of a software product line with those candidates. By comparing them to the effort of the development approach currently in use by the company, a reasonable decision is possible.

This is a simple and transparent approach. With the help of visualization it can be easily explained to the management. Unfortunately, there is a lack of suitable tools and historical data that can support this process.

Question 2:

Do you think that COPLIMO and our approach are suitable for the scenarios?

Answer:

In my opinion, the results of both models are reasonable. For COPLIMO I see the problem that the estimations for re-engineering and new product are difficult to separate. For a company, this separation is essential, as it must pay the costs for extracting the product line itself. The effort that is unique for the new product is assigned to the customer. I think that COPLIMO is not detailed enough in this regard.

Thus, I think that your model is suitable for the extractive approach, as it supports the distinction into costs for the migration and the new product. For current practice your model might be to detailed. Currently, the reported benefits outweigh the costs multiple times. However, in the future more companies with smaller software products might consider to switch towards product lines. In those cases, the details provided by your model are necessary. A benefit is the automatic data extraction which supports and simplifies the most expensive task. Without tools, it is not reasonable to provide such details as the effort drastically increases with the number of legacy systems. Using the lines of code as a basis for the cost estimation is the pragmatic approach. However, you support experts with details on possible cost drivers and calculations. Thus, they are able to adapt the model to their needs.

Question 3:

Do the cost factors and parameters match with the costs you used or think are important?

Answer:

The cost factors provided in the models are suitable for cost estimations of software product lines. Especially, they are more detailed and systematic then judgment-based estimations. Still there are factors, for example, the change frequency I stated, that could be integrated.

The consideration of organizational costs is important. Estimating those is a challenging task and I think a algorithmic model can hardly display them. Still, your descriptive level might point them out more detailed to support experts. For example, I would consider the shift in the companies development processes, tasks of the change-management and controlling, or the risk of failure. However, even for experienced estimators such parameters are difficult to put into numbers.

Question 4:

Did you identify problems with our model?

Answer:

The details in your approach might convey the feeling of reliability and viability. Additionally, while lines of code are often used and are a pragmatic way to deal with cost estimations, they represent a problematic measurement. For example, applying them on a project with multiple programming languages is difficult. The different paradigms, complexities, and code sizes for the same tasks can hardly be displayed.

Question 5:

Do you have suggestions for improvements?

Answer:

Concluding my previous statements, I think that the semi-automated data extraction is helpful. In my opinion, fully automatizing this process is impossible. Still, I would appreciate more detailed and extended scenarios and possibilities for modifications, as well as tools support. Additionally, focusing more on the identification of organizational costs would also be beneficial.

Bibliography

Ra'Fat Al-Msie'deen, Abdelhak D. Seriai, Marianne Huchard, Christelle Urtado, and Sylvain Vauttier. Mining Features from the Object-Oriented Source Code of Software Variants by Combining Lexical and Structural Similarity. In *International Conference on Information Reuse and Integration*, pages 586–593. IEEE, 2013. (cited on Page 58)

Muhammad S. Ali, Muhammad A. Babar, and Klaus Schmid. A Comparative Survey of Economic Models for Software Product Lines. In *Euromicro Conference on Software Engineering and Advanced Applications*, pages 275–278. IEEE, 2009. (cited on Page 3, 20, 21, 23, 24, 48, 61, 62, and 84)

Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and Evolving Mobile Games Product Lines. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, pages 70–81. Springer, 2005. (cited on Page 68)

Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology*, 8(5):49–84, 2009. (cited on Page 8)

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 1 edition, 2013. (cited on Page 1, 2, 7, 8, 9, 10, 18, 35, 36, 75, and 87)

Wesley K. G. Assunção and Silvia R. Vergilio. Feature Location for Software Product Line Migration: A Mapping Study. In *International Software Product Line Conference: Companion Volume for Workshops, Demonstrations and Tools - Volume 2*, pages 52–59. ACM, 2014. (cited on Page 10 and 11)

Janice Ballou. Open-Ended Question. In Paul J. Lavrakas, editor, *Encyclopedia of Survey Research Methods*, pages 547–549. Sage Publications, 2008. (cited on Page 63)

Tim Bardo, David Elliot, Tony Krysak, Mike Morgan, Shuey Rebecca, and Will Tracz. CORE: A Product Line Success Story. *Crosstalk: The Journal of Defense Software Engineering*, 9(3):24–28, 1996. (cited on Page 66)

Veronika Bauer and Benedikt Hauptmann. Assessing Cross-Project Clones for Reuse Optimization. In *International Workshop on Software Clones*, pages 60–61. IEEE, 2013. (cited on Page 58)

Stefan Bellon, Rainer Koschke, Giuliano Antoniol, Jens Krinke, and Ettore Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering*, 33(9):577–591, 2007. (cited on Page 11)

Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *International Workshop on Variability Modelling of Software-Intensive Systems*, pages 7:1–7:8. ACM, 2013. (cited on Page 1, 2, 9, and 87)

Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The Concept Assignment Problem in Program Understanding. In *International Conference on Software Engineering*, pages 482–498. IEEE, 1993. (cited on Page 11, 34, 56, and 58)

Mario G. Blandón, Paulo P. Cano, and Clifton Clunie. Cost Estimate Methods Comparison in Software Product Lines (SPL). *Revista Universitaria RUTIC*, 1(2):18–24, 2013. (cited on Page 3, 21, 25, 26, and 43)

Günter Böckle, Jesús B. Muñoz, Peter Knauber, Charles W. Krueger, Julio C. S. do Prado Leite, Frank J. van der Linden, Linda M. Northrop, Michael Stark, and David M. Weiss. Adopting and Institutionalizing a Product Line Culture. In Gary J. Chastek, editor, *Software Product Lines*, pages 49–59. Springer, 2002. (cited on Page 12, 34, 37, and 38)

Günter Böckle, Paul C. Clements, John D. McGregor, Dirk Muthig, and Klaus Schmid. Calculating ROI for Software Product Lines. *IEEE Software*, 21(3):23–32, 2004. (cited on Page 5, 17, 21, 25, 29, 30, 43, 82, 83, and 87)

Barry W. Boehm. Software Engineering Economics. *IEEE Transactions on Software Engineering*, SE-10(1):4–21, 1984. (cited on Page 2, 3, 12, 14, 19, 20, 39, 48, 50, 79, 81, and 87)

Barry W. Boehm, Bradford K. Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost Models for Future Software Life Cycle Processes: COCOMO 2.0. *Annals of Software Engineering*, 1(1):57–94, 1995. (cited on Page 21, 24, 26, 44, 49, 51, 69, and 81)

Barry W. Boehm, Chris Abts, and Sunita Chulani. Software Development Cost Estimation Approaches - A Survey. *Annals of Software Engineering*, 10(1-4):177–205, 2000a. (cited on Page 12 and 14)

Barry W. Boehm, Chris Abts, Bradford K. Clark, Ellis Horowitz, A. Winsor Brown, Donald Reifer, Sunita Chulani, Ray Madachy, and Bert Steece. *Software Cost Estimation with COCOMO II*. Prentice Hall, 1 edition, 2000b. (cited on Page 21, 24, 26, 44, 49, and 81)

Barry W. Boehm, A. Winsor Brown, Ray Madachy, and Ye Yang. A Software Product Line Life Cycle Cost Estimation Model. In *International Symposium on Empirical Software Engineering*, pages 156–164. IEEE, 2004. (cited on Page 4, 5, 21, 26, 35, 37, 43, 44, 45, 46, 56, 61, 68, 69, and 87)

Fergus Bolger and George Wright. Assessing the Quality of Expert Judgment: Issues and Analysis. *Decision Support Systems*, 11:1–24, 1994. (cited on Page 62)

Ralph Brugger. *Der IT Business Case*, chapter Business Case - Grundlagen, pages 11–31. Springer, 2 edition, 2009. (cited on Page 66)

Oliver Charles, Markus Schalk, and Steffen Thiel. Kostenmodelle für Softwareproduktlinien. *Informatik-Spektrum*, 34(4):377–390, 2011. (cited on Page 3 and 21)

Yu Chen, Gerald C. Gannod, and James S. Collofello. A Software Product Line Process Simulator. *Software Process: Improvement and Practice*, 11(4):385–409, 2006. (cited on Page 23, 26, and 43)

Sunita Chulani and Barry W. Boehm. Modeling Software Defect Introduction and Removal: COQUALMO (COnstructive QUALity MOdel). Technical Report USC-CSE-99-510, Center for Systems and Software Engineering, 1999. (cited on Page 26)

Sunita Chulani, Barry W. Boehm, and Bert Steece. Bayesian Analysis of Empirical Software Engineering Cost Models. *IEEE Transactions on Software Engineering*, 25 (4):573–583, 1999. (cited on Page 21)

Paul C. Clements. Being Proactive Pays Off. *IEEE Software*, 19(4):28–30, 2002. (cited on Page 2, 9, and 87)

Paul C. Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 5 edition, 2006. (cited on Page 1, 7, 8, 18, and 75)

Paul C. Clements, Sholom G. Cohen, Patrick Donohoe, and Linda M. Northrop. Control Channel Toolkit: A Software Product Line Case Study. Technical Report CMU/SEI-2001-TR-030, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2001. (cited on Page 62 and 83)

Paul C. Clements, John D. McGregor, and Sholom G. Cohen. The Structured Intuitive Model for Product Line Economics (SIMPLE). Technical Report CMU/SEI-2005-TR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2005. (cited on Page 5, 20, 21, 25, 29, 43, 64, 82, and 87)

Sholom G. Cohen. Predicting When Product Line Investment Pays. Technical Report CMU/SEI-2003-TN-017, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2003. (cited on Page 21, 37, 56, and 83)

Sholom G. Cohen, Ed Dunn, and Albert Soule. Successful Product Line Development and Sustainment: A DoD Case Study. Technical Report CMU/SEI-2002-TN-018, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2002. (cited on Page 62 and 83)

James R. Cordy and Chanchal K. Roy. The NiCad Clone Detector. In *International Conference on Program Comprehension*, pages 219–220. IEEE, 2011. (cited on Page 58)

James R. Cordy, Charles D. Halpern-Hamu, and Eric Promislow. TXL: A Rapid Prototyping System for Programming Language Dialects. *Computer Languages*, 16(1): 97–107, 1991. (cited on Page 58)

Marcus V. Couto, Marco T. Valente, and Eduardo Figueiredo. Extracting Software Product Lines: A Case Study Using Conditional Compilation. In *European Conference on Software Maintenance and Reengineering*, pages 191–200. IEEE, 2011. (cited on Page 68)

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 6 edition, 2005. (cited on Page 8, 9, 10, and 54)

Sybren Deelstra, Marco Sinnema, and Jan Bosch. Product Derivation in Software Product Families: A Case Study. *Journal of Systems and Software*, 74(2):173–194, 2005. (cited on Page 36)

David Dikel, David Kane, Steve Ornburn, William Loftus, and Jim Wilson. Applying Software Product-Line Architecture. *Computer*, 30(8):49–55, 1997. (cited on Page 37)

Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature Location in Source Code: A Taxonomy and Survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2013. (cited on Page 10 and 11)

Pamela P. Drake and Frank J. Fabozzi. *Foundations and Applications of the Time Value of Money*. Wiley, 1 edition, 2009. (cited on Page 25)

Yael Dubinsky, Julia Rubin, Theodore Berger, Slawomir Duszynski, Matthias Becker, and Krzysztof Czarnecki. An Exploratory Study of Cloning in Industrial Software Product Lines. In *European Conference on Software Maintenance and Reengineering*, pages 25–34. IEEE, 2013. (cited on Page 1, 7, 8, 10, 12, and 18)

Slawomir Duszynski, Jens Knodel, and Martin Becker. Analyzing the Source Code of Multiple Software Variants for Reuse Potential. In *Working Conference on Reverse Engineering*, pages 303–307. IEEE, 2011. (cited on Page 1, 9, 10, 11, 34, 57, 58, 59, 60, and 87)

Wolfram Fenske and Sandro Schulze. Code Smells Revisited: A Variability Perspective. In *International Workshop on Variability Modelling of Software-Intensive Systems*, pages 3–10. ACM, 2015. (cited on Page 37 and 56)

Wolfram Fenske, Thomas Thüm, and Gunter Saake. A Taxonomy of Software Product Line Reengineering. In *International Workshop on Variability Modelling of Software-Intensive Systems*, pages 4:1–4:8. ACM, 2014. (cited on Page 2, 7, and 8)

Shannon Fischer, Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants. In *International Conference on Software Maintenance and Evolution*, pages 391–400. IEEE, 2014. (cited on Page 1 and 7)

Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 19 edition, 2006. (cited on Page 56)

Peter Gacek, Cristina amd Knauber, Klaus Schmid, and Paul C. Clements. Successful Software Product Line Development in a Small Organization: A Case Study. Technical Report 013.01/E, Fraunhofer IESE, Kaiserslautern, Germany, 2001. (cited on Page 38)

Dharmalingam Ganesan, Dirk Muthig, and Kentaro Yoshimura. Predicting Return-On-Investment for Product Line Generations. In *International Software Product Line Conference*, pages 13–24. IEEE, 2006. (cited on Page 21)

Fred J. Heemstra. Software Cost Estimation. *Information and Software Technology*, 34 (10):627–639, 1992. (cited on Page 13, 14, 15, 19, 55, and 62)

Ruben Heradio, David Fernandez-Amoros, Luis Torre-Cubillo, and Alberto Perez Garcia-Plaza. Improving the Accuracy of COPLIMO to Estimate the Payoff of a Software Product Line. *Expert Systems with Applications*, 39(9):7919–7928, 2012. (cited on Page 21, 43, and 52)

Ruben Heradio, David Fernandez-Amoros, Jose A Cerrada, and Ismael Abad. A Literature Review on Feature Diagram Product Counting and its Usage in Software Product Line Economic Models. *International Journal of Software Engineering and Knowledge Engineering*, 23(8):1177–1204, 2013. (cited on Page 3, 21, 48, and 50)

Hoh P. In, Jongmoon Baik, Sangsoo Kim, Ye Yang, and Barry W. Boehm. A Quality-Based Cost Estimation Model for the Product Line Life Cycle. *Communications of the ACM*, 49(12):85–88, 2006. (cited on Page 21 and 26)

Lawrence Jones and Albert Soule. Software Process Improvement and Product Line Practice: CMMI and the Framework for Software Product Line Practice. Technical Report CMU/SEI-2002-TN-012, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2002. (cited on Page 37)

Magne Jørgensen. A Review of Studies on Expert Estimation of Software Development Effort. *Journal of Systems and Software*, 70(1):37–60, 2004. (cited on Page 3, 19, and 20)

Magne Jørgensen. Forecasting of Software Development Work Effort: Evidence on Expert Judgement and Formal Models. *International Journal of Forecasting*, 23(3): 449–462, 2007. (cited on Page 2, 3, 12, 15, 19, 20, 37, 48, 55, 62, 80, and 84)

Magne Jørgensen and Martin Shepperd. A Systematic Review of Software Development Cost Estimation Studies. *IEEE Transactions on Software Engineering*, 33(1):33–53, 2007. (cited on Page 62)

Magne Jørgensen, Barry W. Boehm, and Stan Rifkin. Software Development Effort Estimation: Formal Models or Expert Judgment? *IEEE software*, 26(2):14–19, 2009. (cited on Page 2, 3, 12, 15, 18, 19, 20, 32, 48, 55, and 80)

Christian Kästner, Sven Apel, and Don Batory. A Case Study Implementing Features Using AspectJ. In *International Software Product Line Conference*, pages 223–232. IEEE, 2007. (cited on Page 68)

Christian Kästner, Sven Apel, and Martin Kuhlemann. Granularity in Software Product Lines. In *International Conference on Software Engineering*, pages 311–320. ACM, 2008. (cited on Page 68)

Christian Kästner, Alexander Dreiling, and Klaus Ostermann. Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features. *IEEE Transactions on Software Engineering*, 40(1):67–82, 2014. (cited on Page 10, 11, 34, 56, and 58)

Ninus Khamis, René Witte, and Juergen Rilling. Automatic Quality Assessment of Source Code Comments: The JavadocMiner. In Christina J. Hopfe, Yacine Rezgui, Elisabeth Métais, Alun Preece, and Haijiang Li, editors, *Natural Language Processing and Information Systems*, pages 68–79. Springer, 2010. (cited on Page 55)

Ninus Khamis, Juergen Rilling, and René Witte. Assessing the Quality Factors Found in In-Line Documentation Written in Natural Language: The JavadocMiner. *Data & Knowledge Engineering*, 87:19–40, 2013. (cited on Page 55)

Mahvish Khurum, Tony Gorschek, and Kent Pettersson. Systematic Review of Papers About Economic Solutions for Product Lines. In *International Workshop on Management and Economics of Software Product Lines*, pages 277–284. IEEE, 2008. (cited on Page 3, 21, 62, 84, 85, and 88)

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Object-Oriented Programming*, pages 220–242. Springer, 1997. (cited on Page 10)

Peter Knauber, Dirk Muthig, Klaus Schmid, and Tanya Widen. Applying Product Line Concepts in Small and Medium-Sized Companies. *IEEE Software*, 17(5):88–95, 2000. (cited on Page 1 and 38)

Peter Knauber, Jesus Bermejo, Günter Böckle, Julio C. S. do Prado Leite, Frank J. van der Linden, Linda M. Northrop, Michael Stark, and David M. Weiss. Quantifying Product Line Benefits. In Frank J. van der Linden, editor, *Software Product-Family Engineering*, pages 155–163. Springer, 2002. (cited on Page 1, 18, 31, 36, 62, and 75)

Heiko Koziolek, Thomas Goldschmidt, Thijmen de Gooijer, Dominik Domis, Stephan Sehestedt, Thomas Gamer, and Markus Aleksy. Assessing Software Product Line Potential: An Exploratory Industrial Case Study. *Empirical Software Engineering*, pages 1–38, 2015. (cited on Page 1, 3, 9, 12, 18, 19, 23, 25, 42, 55, 62, 64, 82, and 87)

Jens Krinke, Nicolas Gold, Yue Jia, and David Binkley. Cloning and copying between GNOME projects. In *Mining Software Repositories*, pages 98–101. IEEE, 2010. (cited on Page 58 and 64)

Charles W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, 1992. (cited on Page 7)

Charles W. Krueger. Easing the Transition to Software Mass Customization. In Frank J. van der Linden, editor, *Software Product-Family Engineering*, pages 282–293. Springer, 2002a. (cited on Page 1, 2, 9, and 10)

Charles W. Krueger. Eliminating the Adoption Barrier. *IEEE Software*, 19(4):29–31, 2002b. (cited on Page 7, 34, and 35)

Charles W. Krueger. Towards a Taxonomy for Software Product Lines. In Frank J. van der Linden, editor, *Software Product-Family Engineering*, pages 323–331. Springer, 2004. (cited on Page 8 and 37)

Charles W. Krueger. New Methods in Software Product Line Development. In *International Software Product Line Conference*, pages 95–99. IEEE, 2006. (cited on Page 8)

Sana B. A. B. Lamine, Lamia L. Jilani, and Henda H. B. Ghézala. A Software Cost Estimation Model for Product Line Engineering: SoCoEMo-PLE. In *International Conference on Software Engineering Research and Practice*, pages 649–655. CSREA Pres, 2005a. (cited on Page 21 and 24)

Sana B. A. B. Lamine, Lamia L. Jilani, and Henda H. B. Ghézala. Cost Estimation for Product Line Engineering Using COTS Components. In Henk Obbink and Klaus Pohl, editors, *Software Product Lines*, pages 113–123. Springer, 2005b. (cited on Page 21 and 24)

Hareton K. N. Leung and Zhang Fan. Software Cost Estimation. In *Handbook of Software Engineering and Knowledge Engineering Volume II: Emerging Technologies*, pages 307–324. World Scientific Publishing, 2002. (cited on Page 12, 14, 15, 62, and 81)

Angela Lozano. An Overview of Techniques for Detecting Software Variability Concepts in Source Code. In Olga Troyer, Claudia Bauzer Medeiros, Roland Billen, Pierre Hallot, Alkis Simitsis, and Hans Mingroot, editors, *Advances in Conceptual Modeling. Recent Developments and New Directions*, pages 141–150. Springer, 2011. (cited on Page 10)

Jason Mansell. Experiences and Expectations Regarding the Introduction of Systematic Reuse in Small- and Medium-Sized Companies. In Timo Käköla and JuanCarlos Duenas, editors, *Software Product Lines*, pages 91–124. Springer, 2006. (cited on Page 37, 38, 62, and 83)

Jabier Martinez, Tewfik Ziadi, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Bottom-Up Adoption of Software Product Lines: A Generic and Extensible Approach. In *International Conference on Software Product Line*, pages 101–110. ACM, 2015. (cited on Page 1, 2, 7, 9, 10, 11, 18, 58, 64, and 75)

John D. McGregor, Linda M. Northrop, Salah Jarrad, and Klaus Pohl. Guest editors' introduction: Initiating software product lines. *IEEE Software*, 27(3):16–21, 2002. (cited on Page 41 and 83)

Thilo Mende, Felix Beckwermert, Rainer Koschke, and Gerald Meier. Supporting the Grow-And-Prune Model in Software Product Lines Evolution Using Clone Detection. In *European Conference on Software Maintenance and Reengineering*, pages 163–172. IEEE, 2008. (cited on Page 58)

Ali Mili, Senta F. Chmiel, Ravi Gottumukkala, and Lisa Zhang. An Integrated Cost Model for Software Reuse. In *International Conference on Software Engineering*, pages 157–166. ACM, 2000. (cited on Page 21, 23, 24, and 25)

Kjetil Moløkken and Magne Jørgensen. A Review of Surveys on Software Effort Estimation. In *International Symposium on Empirical Software Engineering*, pages 223–230. IEEE, 2003. (cited on Page 14)

Jarley P. Nobrega, Eduardo S. d. Almeida, and Sílvio R. L. Meira. Income: Integrated Cost Model for Product Line Engineering. In *Software Engineering and Advanced Applications*, pages 27–34, 2008. (cited on Page 21, 25, and 62)

Andy J. Nolan and Silvia Abrahão. Dealing with Cost Estimation in Software Product Lines: Experiences and Future Directions. In Jan Bosch and Jaejoon Lee, editors, *Software Product Lines: Going Beyond*, pages 121–135. Springer, 2010. (cited on Page 23 and 25)

Linda M. Northrop. SEI's Software Product Line Tenets. *IEEE Software*, 19(4):32–40, 2002. (cited on Page 8, 9, 15, 37, 38, and 62)

Vilfredo Pareto. *Manuale di Economia Politica*. Societa Editrice, 1906. (cited on Page 40)

Dale R. Peterson. Economics of Software Product Lines. In Frank J. van der Linden, editor, *Software Product-Family Engineering*, pages 381–402. Springer, 2004. (cited on Page 21)

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 1 edition, 2005. (cited on Page 1, 2, 7, 8, 9, 18, 37, 39, 75, 83, and 87)

Jeffrey S. Poulin. Measuring Software Reusability. In *International Conference on Software Reuse: Advances in Software Reusability*, pages 126–138. IEEE, 1994. (cited on Page 37 and 56)

Jeffrey S. Poulin. The Economics of Software Product Lines. *International Journal of Applied Software Technology*, 3(1):20–34, 1997. (cited on Page 21, 24, 44, and 66)

Baishakhi Ray and Miryung Kim. A Case Study of Cross-system Porting in Forked Projects. In *International Symposium on the Foundations of Software Engineering*, pages 53:1–53:11. ACM, 2012. (cited on Page 64)

Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In LuísMiguel Pinho and Michael González Harbour, editors, *Reliable Software Technologies*, pages 71–82. Springer, 2006. (cited on Page 58)

Winfried Reiß. *Mikroökonomische Theorie: Historisch fundierte Einführung*. Oldenbourg, 2 edition, 1992. (cited on Page 40)

Chanchal K. Roy and James R. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *International Conference on Program Comprehension*, pages 172–181. IEEE, 2008. (cited on Page 58)

Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, 2009. (cited on Page 11, 58, and 60)

Julia Rubin and Marsha Chechik. A Survey of Feature Location Techniques. In Iris Reinhartz-Berger, Arnon Sturm, Tony Clark, Sholom G. Cohen, and Jorn Bettin, editors, *Domain Engineering: Product Lines, Languages, and Conceptual Models*, pages 29–58. Springer, 2013. (cited on Page 10)

Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing Forked Product Variants. In *International Software Product Line Conference*, pages 156–160. ACM, 2012. (cited on Page 1, 7, 10, and 18)

Holger Schackmann and Horst Lichter. A Cost-Based Approach to Software Product Line Management. In *International Workshop on Software Product Management*, pages 13–18. IEEE, 2006. (cited on Page 37 and 53)

Klaus Schmid. An Initial Model of Product Line Economics. In Frank J. van der Linden, editor, *Software Product-Family Engineering*, pages 38–50. Springer, 2002. (cited on Page 21 and 23)

Klaus Schmid. Integrated Cost- and Investmentmodels for Product Family Development. Technical Report 067.03/E, Fraunhofer IESE, Kaiserslautern, Germany, 2003. (cited on Page 21, 24, and 25)

Klaus Schmid and Martin Verlage. The Economic Impact of Product Line Adoption and Evolution. *IEEE Software*, 19(4):50–57, 2002. (cited on Page 1, 2, 7, 9, 32, 33, 34, 41, and 87)

Richard W. Selby. Empirically Analyzing Software Reuse in a Production Environment. In Will Tracz, editor, *Software Reuse: Emerging Technology*, pages 176–189. IEEE, 1988. (cited on Page 46)

Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on Internal and External Validity in Empirical Software Engineering. In *International Conference on Software Engineering*, pages 9–19. IEEE, 2015. (cited on Page 84 and 85)

Stefan Stanciulescu, Sandro Schulze, and Andrzej Wasowski. Forked and Integrated Variants in an Open-Source Firmware Project. In *International Conference on Software Maintenance and Evolution*, pages 151–160. IEEE, 2015. (cited on Page 7, 64, 65, and 84)

Daniela Steidl, Benjamin Hummel, and Elmar Juergens. Quality Analysis of Source Code Comments. In *International Conference on Program Comprehension*, pages 83–92. IEEE, 2013. (cited on Page 55)

Jeffrey Svajlenko, Chanchal K. Roy, and Slawomir Duszynski. ForkSim: Generating Software Forks for Evaluating Cross-Project Similarity Analysis Tools. In *International Working Conference on Source Code Analysis and Manipulation*, pages 37–42. IEEE, 2013. (cited on Page 58)

Antony Tang, Wim Couwenberg, Erik Scheppink, Niels A. de Burgh, Sybren Deelstra, and Hans van Vliet. SPL Migration Tensions: An Industry Experience. In *Knowledge-Oriented Product Line Engineering*, pages 1–6. ACM, 2010. (cited on Page 1, 23, 25, and 32)

Eray Tüzün and Bedir Tekinerdogan. Analyzing Impact of Experience Curve on ROI in the Software Product Line Adoption Process. *Information and Software Technology*, 59:136–148, 2015. (cited on Page 21, 32, 33, 34, and 38)

Jacob Viner. Cost Curves and Supply Curves. *Zeitschrift für Nationalökonomie*, 3(1): 23–46, 1932. (cited on Page 38)

Jacco H. Wesselius. Strategic Scenario-Based Valuation of Product Line Roadmaps. In Timo Käköla and Juan C. Duenas, editors, *Software Product Lines*, pages 53–89. Springer, 2006. (cited on Page 21)

James Withey. Investment Analysis of Software Assets for Product Lines. Technical Report CMU/SEI-96-TR-010, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 1996. (cited on Page 21)

Xue Yinxing. *Reengineering Legacy Software Products into Software Product Line*. PhD thesis, National University of Singapore, Department of Computer Science, 2012. (cited on Page 11)

Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. Defining a Strategy to Introduce a Software Product Line Using Existing Embedded Systems. In *International Conference on Embedded Software*, pages 63–72. ACM, 2006a. (cited on Page 7, 10, 12, 18, 58, 61, 62, and 75)

Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. Assessing Merge Potential of Existing Engine Control Systems into a Product Line. In *International Workshop on Software Engineering for Automotive Systems*, pages 61–67. ACM, 2006b. (cited on Page 34)

Tewfik Ziadi, Luz Frias, Marcos Aurélio Almeida da Silva, and Mikal Ziane. Feature Identification from the Source Code of Product Variants. In *European Conference on Software Maintenance and Reengineering*, pages 417–422. IEEE, 2012. (cited on Page 11)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 16.02.2016