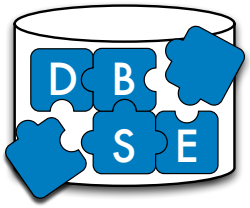
University of Magdeburg

School of Computer Science



Databases
and
Software
Engineering

Bachelor Thesis

Leveraging Code Clone Detection for the Incremental Migration of Cloned Product Variants to a Software Product Line: An Explorative Study

Author:

Konstantin Tonscheidt

April 21, 2015

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake

Institute of Technical and Business Information Systems

Dipl.-Inf. Wolfram Fenske

Institute of Technical and Business Information Systems

Dr.-Ing. Thomas Thüm

Institute of Software Engineering and Automotive Informatics - Technische Universität Braunschweig

Tonscheidt, Konstantin:
Leveraging Code Clone Detection for the Incremental Migration of Cloned Product Variants to a Software Product Line: An Explorative Study
Bachelor Thesis, University of Magdeburg, 2015.

Abstract

Clone-and-own is a pattern that is used often in software engineering to quickly generate product variants. Quick code generation and lack of change propagation in such product sets can lead to high development cost, a problem that is addressed in software product line engineering. In order to benefit from software product line engineering, however, the implementation artifacts have to be migrated, which is an expensive and risky venture in itself that may disrupt the development on the original products. To address this issue, we porpose a new approach that is based on exploiting already existing research on clone analysis and refactoring towards the end of a well-defined, stepwise process that allows further development of the products during the migration. By conducting a case study on a set of clone-and-own created products, we show that clone analysis can be a valuable tool in software product line migration and identify obstacles to be addressed by future research.

Contents

List of Figures **x**

List of Tables **xi**

List of Code Listings **xiii**

1 Introduction **1**

2 Background **3**
2.1 Clone-and-own . . . 3
2.1.1 Advantages of and reasons for using clone-and-own . . . 4
2.1.2 Disadvantages of clone-and-own . . . 4
2.1.3 Product variant generation via clone-and-own . . . 4
2.2 Software Product Lines . . . 5
2.2.1 What are software product lines (SPLs)? . . . 5
2.2.2 SPL Implementation Techniques . . . 7
2.3 Code Clones . . . 8
2.3.1 Different Types of Clones . . . 9
2.4 Refactoring . . . 11
2.4.1 Common Refactorings for Code Clone Removal . . . 11
2.4.2 Refactoring Feature-Oriented Software Product Lines . . . 12
2.4.3 Software Product Line Migration . . . 12

3 Case Study Concept **13**
3.1 SPL Migration Context . . . 13
3.2 Migration Paradigms . . . 14
3.2.1 Bottom-up: By clone size . . . 15
3.2.2 Bottom-up: By variant . . . 15
3.2.3 Top-down: Starting with the Feature Model . . . 15
3.3 SPL Refactoring . . . 15
3.3.1 Clone Analysis Exploitation . . . 16
3.3.2 Feature Awareness of Code Clones . . . 16
3.3.3 Use of Refactorings . . . 16

4 Implementation **19**

4.1 Migrating towards a FeatureIDE SPL 19
4.2 Clone Analysis 20
4.2.1 CPD 20
4.2.2 Presentation of Results 22

5 Evaluation **25**
5.1 The Case Study's Projects 25
5.2 The Case Study 26
5.2.1 Migration towards a simple SPL 26
5.2.1.1 Evaluation 27
5.2.2 Refactoring by Clone Size Heuristic 27
5.2.2.1 Evaluation 29
5.2.3 Refactoring by Product Variant 29
5.2.4 Refactoring by Predefined Features 30
5.2.4.1 Evaluation 31
5.2.5 Obstacles to Migration 31
5.2.6 Importance of Preparational Refactorings 32

6 Conclusion **35**

A Appendix **37**

Bibliography **41**

List of Figures

2.1 Cost of clone-and-own approach versus SPLE based on Pohl et al[PBVDL05] 5

2.2 An example feature model for a program that calculates the factorial of a number. Created with FeatureIDE 6

3.1 First step of the migration; by selecting feature x from the model, product x should be generated 14

4.1 The result of migrating the example Java projects "XMLExamples1" and "XMLExamples2" with default settings. 21

4.2 Clone Analysis View of our FeatureIDE extension. 23

4.3 Eclipse marker indicating starting lines of detected clones. 23

5.1 The clone analysis view of our plugin, showing the results for the unaltered case study projects. 28

5.2 An excerpt of the "ApoButton.java" implementation of the ApoClock feature, as displayed in the Eclipse Java editor. Clones are marked at the left margin. 30

5.3 The feature model after extracting the feature Button from the original products. 31

5.4 Two different Rectangle classes being used in different implementations of"ApoEntity.java". 32

A.1 A histogram of the length of clones within the original products. 38

A.2 A histogram of the length of clones with occurrences in two different original products. 38

A.3 A histogram of the length of clones with occurrences in three different original products. 39

A.4 A histogram of the length of clones with occurrences in four different original products. 39

A.5 A histogram of the length of clones with occurrences in five different original products. 40

List of Tables

5.1 Metadata of the case studies projects and results from clone analysis. . 25

5.2 Number of clones shown, after removing those without occurrences in the selected product. 30

List of Code Listings

2.1 A naive implementation of the factorial function using CPP to create variablity. 7

2.2 Factorial.java - Composition result of features Base and CatchNegatives 7

2.3 Factorial.java - Base 8

2.4 Factorial.java - Feature CatchNegatives 8

2.5 Factorial method - Original 9

2.6 Factorial method with altered whitespaces and comments 9

2.7 Factorial method with altered identifier names, whitespaces, comments 10

2.8 Factorial method with an additional if-then statement, as well as altered comments and whitespaces 10

2.9 Recursive implementation of the factorial method 10

5.1 Imports of ApoSnake and ApoDice's ApoEntity.java 28

5.2 Imports of ApoClock, MyTreasureAndroid and ApoMonoAndroid's ApoEntity.java 28

1. Introduction

Software development is expensive due to the need of skilled personnel and project management cost. To reduce development cost, clone-and-own, i.e. using existing code artifacts as a basis, then altering them to fit the new requirements, is a commonly used strategy [RC13]. This is usually cheaper than starting a new project from scratch, since less code has to be (re-)written. If it is done repeatedly, however, over the years several similar projects will exist with a huge amount of common lines of code. As a result of those projects being independent of each other, traceability and propagation of changes between them is limited, which means changes have to be manually adapted to each project and cost for software maintenance rises[GFGP06].

Software Product Line Engineering (SPLE) manages the differences and exploits the commonalities in software product variants[ABKS13]. With *features*, functionality units defined by the domains requirements, being a central concept in feature-oriented SPLE. *Software Product Lines (SPL)* are software systems that consist of reusable components from which product variants can be created. There are two different approaches to the implementation of SPL: annotation-based and composition-based SPL[ABKS13]. Annotation-based SPL have a single code base which is modified by a preprocessor to generate different product instances. Composition-based SPLs aim to physically separate features, ideally with each feature being implemented in a single module. They use different advanced programming concepts like plugin architectures, aspect-oriented programming, or extend programming languages by feature-oriented constructs for variant composition. Using the taxonomy of Fenske et al, we call the process of creating an equivalent SPL from a set of existing products SPL migration[FTS14].

Migrating several existing products into a SPL is a complex and often expensive venture. There are approaches to automatically migrating to annotation-based SPLs, but migration to composition-based SPLs has not been examined as extensively. The automation of SPL migration is a very complex problem, since having domain knowledge is essential to properly identify, define, and implement features. There are tools that

support SPL development[TKB+14] and migration[DKB11] but supervision, manual editing and refactoring is still required. Ensuring the further development and maintenance on products that are being migrated poses a challenge, since changes have to be implemented in the original product and/or the unfinished SPL. As migration effort grows with size and amount of the original projects, a systematic approach is necessary.

Code Clones (CCs) are a concept to describe similarity of code. Roy et al distinguish between four different kinds of CCs, three of which are based on textual similarity[RCK09]. Based on the assumption, that clone-and-own created product variants will contain a lot of CC we propose a new approach to SPL migration. As a first step, we create a token SPL from variant products. Then we exploit similarities between the product variants that result from their clone-and-own heritage, by using tools for detection and analysis of CCs to find refactoring opportunities. If this constitutes a reliable way to detect leverage points for migration, it would be a first step towards a well-defined, incremental SPL migration process.

Goal of this Thesis

With the ultimate goal of facilitating the migration of several product variants into a single SPL, this thesis examines how existing tools for detecting and analyzing CCs can be deployed in SPL migration to a composition-based SPL. Operating on a set of 5 products that have reportedly been developed by application of clone-and-own, we will gather first experiences with a new approach and explore the possibilities and limits of clone analysis as a means to SPL migration. Based on this experience and the results of our exploration we will then identify strengths and weaknesses of the approach as well as main leverage points for future research.

Structure of the Thesis

Following this introduction, we will first give background information about various terms and topics that are relevant to our research. Then we will describe in detail how we are going to conduct the case study, define some use cases we are going to test and describe our expectations. The subsequent chapter will describe the tool we implemented to test our approach and conduct the case study with. Following that we will describe how we conducted the case study and will afterwards evaluate its results. In the end we will give a conclusion, where we summarize our results and try to answer our research question, how existing tools for CC analysis and presentation can be used to support migrating a set of clone-and-own-created products into a compositional SPL.

2. Background

This chapter deals with the basics of clone-and-own as an approach to creating software systems and describes characteristics of systems created by its application. Furthermore, SPLs will be explained, as well as feature-oriented software development. After that, code clones and refactoring are introduced. Then we present some example refactorings that are commonly used to remove code clones.

2.1 Clone-and-own

Clone-and-own is a software development pattern which is applied in the context of product variant generation[RKBC12, AJB+14, DB07]. As the name implies, it involves duplicating software artifacts like source code, binaries, libraries, or whole software systems. Any software artifact duplication, from copying a single line to the duplication of a software system in order to create a basis for a new project, is an application of clone-and-own. Common examples for this are copy and pasting of methods and the forking of projects[RKBC12]. The copied content may remain as is, but often has to be modified to accomodate the new context. Since copying source code and files usually is a trivial task, this is a fast way to create software functionality in the circumstance of having access to the sources of similar software[DB07]. As such, clone-and-own is an example of code reuse, albeit a very simple one. Although there are certain advantages to it, it results in duplicate code, which is commonly regarded as a code smell[KG06a, SKR08].

For the purpose of this thesis we are going to distinguish between different magnitudes of clone-and-own. Duplicating lines, methods, classes or files within single software systems is clone-and-own on a small scale. This is in contrast to the forking of whole systems or copying a number of such systems resources for the purpose of creating a new software system, which is clone-and-own on a large scale. In this thesis, we examine software system variants created by large scale clone-and-own.

2.1.1 Advantages of and reasons for using clone-and-own

There are several reasons for using the clone-and-own approach in software development. Benefits of clone-and-own can include improvement of program comprehensibility, reduced test effort, and ensurance of source system stability[KG06a]. If, for instance, a developer wants to use a utility function at some point, which exists but cannot be called (for reasons of scope, or due to only being implemented in a separate project), he can copy the code directly to where it is going to be used. This would be an application of small scale clone-and-own. Its advantages are the speed of implementation and the fact, that the copied code might already be tested or proven[KG06a].

2.1.2 Disadvantages of clone-and-own

In the long run, cloning can lead to a wide range of problems[KG06b]. If copied code has to be changed due to a bug report or new requirements, each clone needs to be identified and possibly changed in a similar way. This is potentially time consuming and error-prone, and if clones are missed it can lead to inconsistencies in the project[JDHW09]. Furthermore, the cloned artifact may contain errors that are propagated by the cloning. Thus, creating duplicate code is generally discouraged, because it comes with the risk of lowering code quality[MNK+02] and can lead to higher maintenance costs[GFGP06]. The simplicity of code generation also comes with the danger of inflating a projects code size, which can further increase maintenance effort[Bak95].

2.1.3 Product variant generation via clone-and-own

Projects can be generated via clone-and-own by duplicating a project directory or forking from a project repository (common functionality of version control systems). This large scale clone-and-own is used to create new variants of an existing project[RKBC12]. For example, a customer may want a product with similar functionality as an already implemented one. Clone-and-own is a fast way to realize the new project, as cloning the old project contents quickly creates a code base for the new project. The benefits are similar to that of small scale clone-and-own: implementation speed and previous knowledge about the copied parts' behavior. Product variants are often created by clone-and-own despite the disadvantages of the approach, because it is thought to be the fastest method to reuse code[RKBC12]. A major drawback to this variant creation process is that, especially for high numbers of variants, structural and contentual similarities between variants are not exploited for development and maintainance of the product variants as well as they could be in comparison with other development paradigms. There may, for example, be an enhancement introduced in project A that would also be useful for project B. Since the structure of those parts relevant to the enhancement might have changed in project B, implementing this enhancement for it is not a trivial task[YGM06]. With a large number of product variants, this becomes more of a factor and leads to high development costs if not addressed. Figure 2.1 illustrates that development for more than three software systems (product variants) is more cost efficient when developed as a software product line.

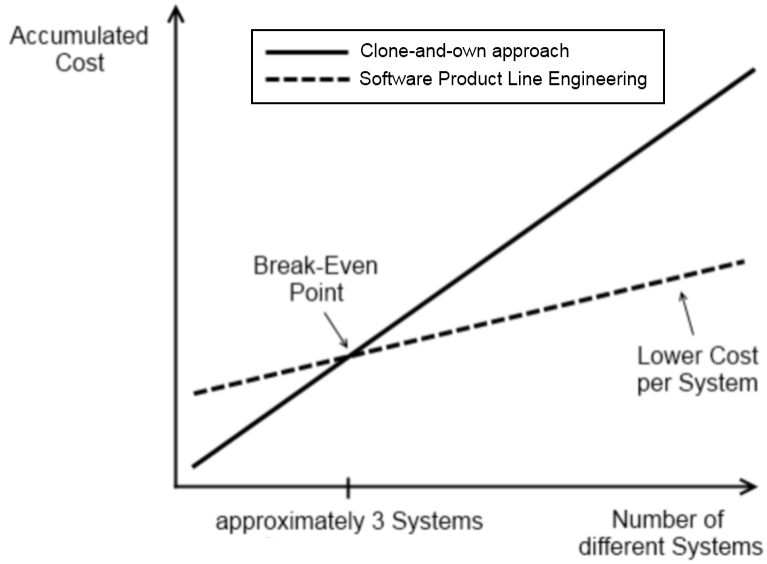


Figure 2.1: Cost of clone-and-own approach versus SPLE based on Pohl et al[PBVDL05]

2.2 Software Product Lines

In this section we give a short introduction into the topic of Software Product Lines and SPLE including motivation, common practices and implementation aspects. For large sets of product variants, as may have been created by large-scale clone-and-own, this is a cost efficient development paradigm. A reuse framework needs additional up-front investment, which for more than 3 variants is worth it, since its cost are outweighed by the resources that are saved due to reuse, as shown in Figure 2.1[ABKS13].

2.2.1 What are SPLs?

Product lines, as a realization of mass customization have been successfully used in domains like automobile industry for decades[CN98]. The general idea is to implement reusable parts, from which the products then can be created, instead of developing every product from scratch. This concept of systematic reuse can be applied to software systems and promises benefits such as reduced time to market, lower development cost and improved code quality[ABKS13].

The conventional approaches of software development are individual software, where one product is tailor-made specifically for one customer, and standard software, which is created with a broad set of features and then sold ”off the shelf”[PBVDL05]. The SPL approach is a paradigm to produce mass customizable software. As such, SPLs are a middle course between individual and standard software, allowing generation of a large number of software variants with little of effort.

Clements and Northrop define SPL like this: “A software product line is a set of software-intensive systems sharing a common mangaged set of features that satisfy the

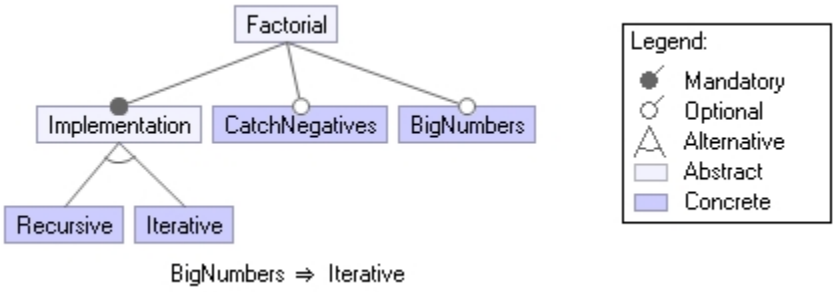


Figure 2.2: An example feature model for a program that calculates the factorial of a number. Created with FeatureIDE

specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."[CN01]. In other words, a SPL is a set of software-related systems from which a number of different product variants can be created. Furthermore, all of those product variants have to belong to a single domain. In feature-oriented SPLE, variants are generated from a set of implementation artifacts. Features in this context can be functional or nonfunctional software requirements, they are increments of functionality that represent requirements and are the variable parts that differentiate product variants from each other[ABKS13]. The feature implementation depends on the reuse framework, which also determines how the reusability and product generation is implemented in the SPL.

In feature-oriented SPLs each distinct and valid feature selection (configuration) evaluates to a different product variant[ABKS13]. A feature model describes which feature combinations are valid. There are a number of different representation forms for feature models, such as logic expressions or graphs[ABKS13]. In this thesis we will mostly use a combination of a tree graph and optional logic constraints, based on the work of Kang et al[KCH+90]. In this model notation we refer to compound features as parent features and primitive features (the leaf features) as child features. Figure 2.2 illustrates how such a simple feature model can look like. For the given feature model any combination of features needs to contain exactly one out of *Recursive* and *Iterative* to be valid. The features *CatchNegatives* and *BigNumbers* implement optional additional functionality. "BigNumbers implies Iterative" is the single constraint of this model. This means, that if a configuration contains the feature *BigNumbers* it must also contain *Iterative* in order to be valid.

Developing a set of products as a SPL has some disadvantages: A framework that enables the reuse of components has to be planned and implemented, before artifacts can be reused. This requires an up-front investment. So until a certain number of products are implemented, a SPL is developed more slowly and at a higher cost than would be the case for multiple products from scratch. In the long-term, the reuse platform often yields improved change propagation and thus reduction of maintenance effort, can significantly reduce code complexity and facilitates evolution of the product

```
1  int factorial(int number)
2  {
3      #ifdef RECURSIVE
4      if(number == 0)
5          return 1;
6      else
7          return number * faculty(--number);
8      #endif
9
10     #ifdef ITERATIVE
11     int result = 1;
12     while(number-- > 1)
13         result= result * number;
14     return result;
15     #endif
16 }
```

Listing 2.1: A naive implementation of the factorial function using CPP to create variablity.

```
1 int factorial(int number)
2 {
3   if(number<0)
4         return -1;
5     int result = 1;
6         while(number-- > 1)
7     result = result * number;
8     return result;
9 }
```

Listing 2.2: Factorial.java - Composition result of features Base and CatchNegatives

range[PBVDL05]. This results in overall less expensive development for SPL with high numbers of product variants, as illustrated in Figure 2.1.

2.2.2 SPL Implementation Techniques

There are two different types of SPL implementation techniques, both of which have their benefits and drawbacks compared to each other: annotation-based and composition-based implementation [ABKS13].

Annotation-based SPLs are implemented in a single code base, in which variability is realized through preprocessor annotation. An example of this would be the #ifdef directive of CPP, the preprocessor of the C programming language, which removes code of deselected features before the project is built. Listing 2.1 shows a code snippet which is annotated by these #ifdef statements. Unless *RECURSIVE* is defined by "#define RECURSIVE" before line four, the lines four to seven will be removed before code compilation. The same applies to *ITERATIVE* and the lines 11-14. This makes it

```
1 int factorial(int number)
2 {
3     int result = 1;
4         while(number-- > 1)
5     result = result * number;
6     return result;
7 }
```

Listing 2.3: Factorial.java - Base

```
1 int factorial(int number)
2 {
3   if(number<0)
4         return -1;
5     original(number);
6 }
```

Listing 2.4: Factorial.java - Feature CatchNegatives

possible to easily switch between the different functionality without having to maintain a project for each implementation. The main advantages of this approach are its language independence and ease of implementing fine grained variability [ABKS13]. There are several downsides including sensitivity to subtle errors, obfuscated source code, restrictions concerning reuse and a lack of separation of concerns [ABKS13].

Composition-based SPLs consist of physically separated feature implementations. The product is created by assembling implementations of selected feature modules using language-specific constructs, such as *Feature Oriented Programming (FOP)*, aspect oriented programming (AOP), or implementation paradigms like plugins [ABKS13]. FeatureHouse is a tool that can be used for composition as part of FOP. It relies on feature structure trees, that describe the language-specific structure of software artifacts[AKL13]. A specific example of how FeatureHouse works is shown in Listing 2.2, which is the result of merging Listing 2.3 with Listing 2.4. Here the factorial function is refined by returning -1 as an error code when it is called with a negative integer as parameter. The modular structure of composition-based SPLs simplifies code reuse [AK09] and documentation. Drawbacks include feature interactions, phenomenons such as the "Tyranny of Dominant Decomposition" and "Multidimensional Separation of Concerns" [TOHSJ99]. Furthermore, some of the constructs for assembly require an unreasonably big overhead for small extensions. Example tools for composition-based SPL development in Java are FeatureHouse (FOP) and AspectJ (AOP).

2.3 Code Clones

CCs are code fragments with a certain degree of similarity [RCK09]. The majority of industrial software systems contains between 5% and 20% code clones [MLM96, KG06b]. Most are created as a result of copy and pasting source code (see Section 2.1) but they

can also be created accidentally [RCK09]. While CCs are regarded as a code smell [FBBO99] and cloning can become a serious problem in industrial software systems [AVMDP02, Bak95], not all clones are harmful to a software system [KG06b].

2.3.1 Different Types of Clones

To describe different types of code clones we use the taxonomy by Roy and Cordy [RCK09]. They distinguish between 4 types of code clones that differ in their degree of similarity.

Type I clones, also known as *exact clones*, are fragments of code that are the same letter for letter, except for comments and whitespaces.

Example: For this code fragment from the factorial implementation

```
1 int factorial(int number)
2 {
3     int result = 1;
4         while(number-- > 1)
5     result = result * number;
6     return result;
7 }
```

Listing 2.5: Factorial method - Original

an exact clone could look like this:

```
1     int factorial(int number){
2     int result=1; //we shall return this
3     while(number-- > 1) result=result*number;
4     return result; //done
5     }
```

Listing 2.6: Factorial method with altered whitespaces and comments

We can see, that after comments and whitespaces (including newlines) are removed, both fragments are exactly the same.

Type II clones include all Type I clones, but they also include code fragments that are the same except for identifier(fields, methods, constants,...) names.

Example: For the original code fragment from the first example (Listing 2.5) a Type II clone could look like this:

The code fragments look more different now, but still have the same syntactic structure. They differ only in the names the method and variables.

Type III clones, again, include all Type II clones in addition to fragments of code, in which statements have been changed, added or removed.

```
1     int facIt(int n){
2     int retVal=1; //we shall return this
3     while(n-- > 1) retVal=retVal*n;
4     return retVal; //done
5     }
```

Listing 2.7: Factorial method with altered identifier names, whitespaces, comments

Example:

This is a Type III clone of the snippet original code fragment from the first example (Listing 2.5):

```
1     int factorial(int number){
2     if(number<0) return -1; //we added this line
3     int result=1; //we shall return this
4     while(number-- > 1) result=result*number;
5     return result; //done
6     }
```

Listing 2.8: Factorial method with an additional if-then statement, as well as altered comments and whitespaces

Here a condition and a statement were inserted, it is a Type III clone of the original statement with a difference of two statements. Except for that, the structure of the code is the same.

Type IV clones are code fragments with "semantic similarity". For instance a recursive implementation of the factorial function and one without recursion that is implemented using a loop are a Type IV clone.

Example: We take the same snippet as in the previous examples (Listing 2.5) for comparison.

A Type IV clone of the snippet could look like this:

```
1 int factorial(int number)
2 {
3      if(number == 0)
4          return 1;
5      else
6          return number * faculty(--number);
7 }
```

Listing 2.9: Recursive implementation of the factorial method

The snippets implement a very similar functionality and thus are Type IV clones. Nevertheless, they are clearly different on textual and syntactical level [RCK09].

There are many different approaches to code clone detection. Detection of Type I clones can be as simple as using the linux *diff* command, which compares data based on its string representation. To detect higher level clones, more sophisticated algorithms, such as token-based comparison are necessary, which are able to detect clones of Type I-III. There are no algorithms that can reliably detect Type IV clones [RCK09]. To detect higher level clones, more sophisticated algorithms, such as token-based comparison are necessary, which are able to detect clones of Type I-III. There are no algorithms known that can reliably detect Type IV clones [RCK09].

2.4 Refactoring

After providing a short explanation of refactoring in general, we will expand on specific refactorings concerning SPLs and CCs.

Software refactorings are alterations to the source code that do not change the software's behaviour [FBBO99, Opd92]. A common example of that is the renaming of an identifier, such as the name of a class or variable, but more complex refactorings can also lead to structural changes. As software is continually changed and new functionality is added, complexity of the code may increase. If this leads to the code becoming less and less maintainable, it is called code decay [OVMMW99]. This is an indicator that the code should be refactored in order to ensure that it remains manageable.

2.4.1 Common Refactorings for Code Clone Removal

Code smells are a set of code patterns that can indicate a need to refactor the smelling code. They were first described by Fowler & Beck, who listed a number of smells and suggested possible refactorings to remove them [FBBO99]. The first smell on their list is Duplicate Code, for which they, depending on the circumstances, suggest a combination of the *Extract Method*, *Pull Up Field*, *Form Template Method*, *Substitute Algorithm*, and *Extract Class* refactorings [BFB99].

The *Extract Method* refactoring moves a piece of functionality into a new method. It is replaced by a call to the so-created method. This can remove unneccessary complexity and, especially in combination with *Rename Method*, improve readability of long code fragments.

The *Pull Up Method* refactoring is usually applied, when two or more similar fields or methods exist in classes with a common ancestor. By moving the field or method to the ancestor, all but that one occurence can be removed, without a change to functionality.

Extract Class can be used to extract duplicate functionality from different classes. The former duplicates then delegate to the functionality of the new class.

Extract Superclass creates a superclass into which any common functionality of class level clones is then introduced. This should be used instead of *Extract Class*, if it is semantically sensible.

2.4.2 Refactoring Feature-Oriented Software Product Lines

Since traditional refactoring only aims to directly improve the structure of source code, interactions with features and product generation in FOP is not well defined. To accurately describe particular SPL specific refactorings, we will define some terms in the following paragraphs. The terminology is based on the work of Schulze et al and intended to ensure feature-awareness of refactorings [STKS12]

*Variant-Preserving Refactoring*s are changes to the feature implementations and/or the feature model, which do not change the validity of feature combinations in the feature model. Additionally, all configurations must remain compilable, if they were compilable before, and their behaviour must not change.

Pull Up Field to Parent Feature moves a field from a set of child features to the parent feature, making it available in all features and removing the original fields from the child features.

The refactoring *Pull Up Method to Parent Feature* involves moving a method that occurs identically in two or more child features to their parent feature and replacing references of the old methods with those of the new one.

2.4.3 Software Product Line Migration

Migration in software development is the process of transforming software systems into a new environment, without changing its functionality [WZ07]. Its aim is to keep the project managable, or to accommodate it to new requirements. A set of cloned variants may be migrated to a SPL, so the development can benefit from code reuse, lowering its cost.

A *Variant-Preserving Migration* is the process that transforms a number of related software products into a SPL. For each initial product, it must be possible to create a product with the same behaviour by generating it from the SPL that is the migration result [FTS14].

3. Case Study Concept

In this chapter we outline the case study's context, outline what we intend to do, and what results we expect. To measure how code clone analysis can assist in SPL migration, we will evaluate several scenarios that are relevant in the context of our migration approach. By our design, the SPL migration is a manual process with a focus on the refactoring of the features as its largest part. To ease this refactoring process we aim to discern those metrics, out of the ones measured by clone analysis, that help most with refactoring. The case study is conducted in the context of an approach to SPL migration that is explained in the following.

3.1 SPL Migration Context

This thesis addresses a specific SPL migration approach, which aims to migrate sets of software systems, specifically ones created by clone-and-own, into a *composition-based* SPL.

As the first step, we create a very simple SPL, in which each of the original products is implemented in a single feature and all features, except for the root feature, are alternatives, as shown in Figure 3.1. In comparison to a complete migration including domain analysis, creation of a sensible feature model and implementation of the newly defined features, we expect the effort to be neglegible, since neither domain analysis, nor feature modeling, nor implementation of new features is required for the creation. The general idea works like this:

1. Create a SPL with a feature model containing only an abstract *Base* feature.
2. Add one feature for each original software system as child feature of *Base* to the feature model.
3. Make all child features of *Base* alternatives.

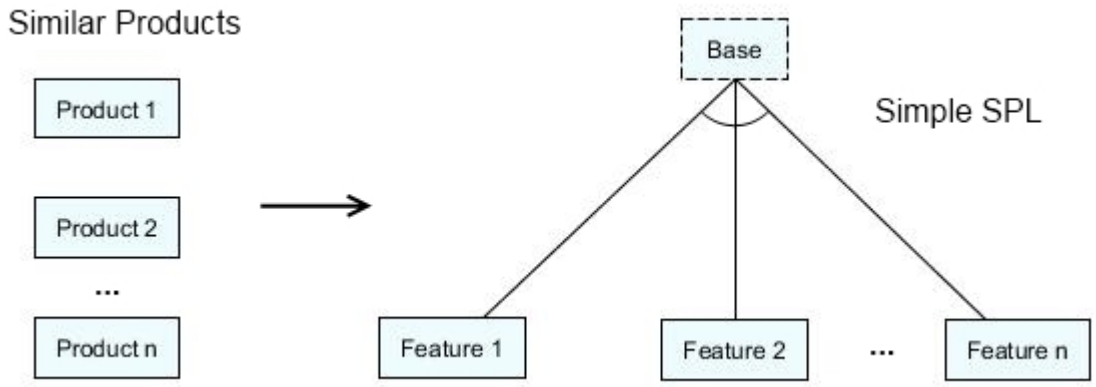


Figure 3.1: First step of the migration; by selecting feature x from the model, product x should be generated

4. Copy implementation artifacts from each original software system to the directory associated with its feature.

The result is a SPL where only configurations consisting of one of the original software systems are valid (compare Figure 3.1). The composition of those unaltered resources with an empty *Base* feature results in the original software system.

Using that simple SPL as a basis the main part of the migration, developing a sensible feature model and the necessary configurations, is then realized by stepwise refactoring of the SPL. The main advantage of this approach is that the impact on production is reduced since products can be generated after each step of the migration process. Other approaches, such as simple applications of the extractive and reactive approaches as described in the work of Krueger and Charles[Kru02] interrupt further development and product generation during migration. Our approach also guarantees conservation of the original semantics as an implication of refactoring conserving semantics. With this approach to SPL migration we argue that supporting the refactoring process directly benefits the migration, because it makes up the main part of the migration. Furthermore, since the original software systems are created by clone-and-own, it is reasonable to assume that the project artifacts in which they are implemented contain a significant amount of code clones. We put forward the theory, that clone detection and removal techniques can be used to aid the migration process, especially if the clone analysis results are processed to be made feature-aware.

To support the refactoring and thus, ultimately, the migration we implement postprocessing of clone analysis results to make them feature-aware and improve the presentation with respect to its use in three different refactoring approaches.

3.2 Migration Paradigms

With the goal of refactoring a SPLs feature model to consist of only semantically sensible, well defined features we consider three different approaches of getting there from a

feature model of variants. We implement support for these refactoring approaches and then evaluate if, and how clone analysis was useful in the SPL refactoring process.

3.2.1 Bottom-up: By clone size

To reduce the time spent by a developer evaluating clone analysis results we evaluate the clones with a simple heuristic:

*clone value = length * occurrences*

The value of a clone is the product of its length measured in lines of code and the number of features the clone occurs in. This is an approximation of total clone length, which should indicate high refactoring potential. More precisely, we expect this to be effective in reducing total code size. In an ideal scenario, all occurences can be replaced by a single implementation, reducing the total code length by roughly *clone length * (occurences - 1)* lines.

3.2.2 Bottom-up: By variant

Due to reasons such as diverging familiarity of developers with code of different variants it might be a good idea to start extracting features from a single variant. To support this we remove all clones that do not have an occurence within a specified feature from the analysis results. This should reduce time spent on finding clones belonging to the relevant variant and thus benefit the refactoring process. It can be combined with the clone size heuristic to further reduce the effort.

3.2.3 Top-down: Starting with the Feature Model

If senior developers or software architects exist that have a high level of knowledge of the original software systems, their domain requirements and implementation, it might be possible to engineer parts of the feature model from domain knowledge.

Using a point in the code that is known to implement the feature as a starting point, it could prove useful to easily identify clones, which occur at this point. This could for example be accomplished by marking cloned code in the editor and providing easy access to the clone's other occurrences.

3.3 SPL Refactoring

There are several aspects to SPL refactoring, which we elaborate on here, especially mentioning how we intend to address them and what results we expect. First we will explain, how we plan to exploit clone analysis in favor of SPL refactoring, introducing new categories for feature-aware clones in the process. Then we specify the refactorings we plan to use in order to remove the clones we found, giving some examples using the FeatureHouse[AKL09] software composition tool and finally discuss possible problems that might arise in the process.

3.3.1 Clone Analysis Exploitation

Clones, as they are detected by common clone analysis algorithms, usually exist in a file- and a string-based form. That usually means their occurrences are returned as lines of text or combinations of file names and line indices, often with attached meta information such as the length of the clone in characters, lines or tokens. While the results of clone analysis might already prove to be of some use, they are not directly indicating refactoring opportunities yet. The main problems inhibiting the use of the raw analysis results are, that they lack feature-awareness and a reliable way to evaluate a clones usefulness. In order to improve the refactoring significantly we process the clone analysis results to compute information about the clones that aids in refactoring as directly as possible by addressing those probems.

3.3.2 Feature Awareness of Code Clones

After running a clone analysis on the SPLs feature implementations, we compute additional information from the clone analysis results and the feature artifacts. While clone analysis results, depending on individual implementations, may contain additional information such as total number of clones, their length and location, they do not contain information about the features they occur in, which is relevant to SPL refactoring. For the purpose of evaluating refactoring potential of clones, we define inter-feature and intra-feature clones[FS15].

Intra-feature code clones are clones with all occurrences being located in a *single* features code.

Inter-feature code clones are clones that have occurrences within code of *two or more* different features.

We expect inter-feature code clones to be significantly more likely to indicate SPL refactorings. Since they occur in at least two different features, refactorings to remove the clone have to involve code extraction to a common ancestor feature. Intra-feature clones on the contrary may indicate opportunities for refactoring, but since those do not affect the feature model, they do not serve the purpose of our migration context. In order to exploit this, we determine for each clone, whether it is inter- or intra-feature, display this and make it possible to filter the clones by the characteristic.

3.3.3 Use of Refactorings

Research by Schulze et al[STKS12] suggests that removal of code clones is mainly done by application of the *Pull Up Method to Parent Feature* refactoring, when the refactoring needs to be variant preserving. In order to get closer to a goal of a sensible feature model, however, we have to introduce new features. Since the distinction between Pull Up Method and Pull Up Field may conflict with the clone based approach to refactoring, we define *Pull Up Code*, as a variant-preserving SPL refactoring, which is composed of a positive number of *Pull Up Field* and *Pull Up Method* refactorings.

Especially at the start of the migration, we expect the creation of new common parent features to be necessary frequently to enable us to pull up code from the features without altering the other variants. Later on, this will no longer be possible, as any feature may only have a single parent. From that point, to remain variant-preserving it is necessary to create a combination of constraints and optional features to which code then has to be moved via *Move Field Between Features* and *Move Method Between Features* refactoring. The constraints being created in a way that the optional feature must be selected, if any feature originally containing an occurrence of the refactored clone, would be selected in the configuration and must not be selected otherwise.

4. Implementation

To test the effectiveness of the changes proposed in the previous chapter we implemented them in a tool to conduct the study with. The tool is implemented as an Eclipse plugin, extendig the FeatureIDE environment.

FeatureIDE is a framework that supports feature-oriented software development in the Eclipse IDE and is used mainly in research and teaching of feature-oriented software development (FOSD)[TKB+14]. Its functionality includes support for feature modeling as well as use of feature-oriented programming (FOP) and other established SPL implementation techniques. FOP-based FeatureIDE projects contain a feature model, a set of configurations, and a set of feature implementations. Feature models are implemented in XML, they can be modified per API and viewed and modified in a specially tailored editor.

Our implementation is divided into two parts: The first is a migration tool to create an initial SPL as described before. The second is an extension to a clone analysis tool which implements the concept of improving clone analysis results by adding feature-awareness.

4.1 Migrating towards a FeatureIDE SPL

As explained in the previous chapter, the case study is conducted in the context of migration, with the first step being the creation of a SPL in which each original product is implemented in a feature (as shown in Figure 3.1). For the FeatureHouse composer, which we are using, and regular Java projects this can be reached by these specific steps:

1. Create a new FeatureIDE project.
2. Add one feature for each original software system as child feature of *Base* to the projects feature model.

3. Make all child features of *Base* alternatives.
4. Create a configuration for each original product in which only its corresponding feature is selected.
5. Copy the original project sources to the directory associated with its feature.

Our prototypical implementation is limited to original projects that are recognized by Eclipse as Java projects, and the resulting SPL has to use FeatureHouse as composer.

Starting the migration is done by selecting the original projects from the workspace and selecting the corresponding option from the context menu. Then the new project's name, as well as the directories for configuration files and the composed output can be chosen in a simple wizard with 2 pages. Once the input is confirmed, the tool first creates a new FeatureIDE project, which is core functionality of the FeatureIDE plugin. It then adds a feature per original project to the feature model and sets them as alternatives. After that, it creates the configuration files with the original project's names and the ".config" extension. Finally, it creates a feature folder for each project and copies the project's artifacts to the folder of their associated feature.

An example output of the migration of two example projects "XMLExamples1" and "XMLExamples2" is shown in Figure 4.1. Since the migration was performed with default settings, the created FeatureIDE project is called "migratedSPL" and the configuration files and feature implementations are located at "/configs" and "/features", respectively. In this example, the "XMLExamples2.config" configuration file was selected and its product generated at the default location "/src".

4.2 Clone Analysis

While a project like the one from the previous example Figure 4.1 already is a SPL, it does not yet provide any of the benefits of SPLs. Its feature model does not allow customisation or reuse beyond the already existing projects, and common functionality is not implemented once for the SPL but once per original project. Refactoring this, so reuse of its components is possible and each functionality is implemented only once without altering its projects semantically is the main challenge (compare Section 3.1). To support this refacoring process, we use clone analysis.

Our contribution is a postprocessing step after the clone analysis, which adds feature-awareness and a custom view. The latter is intended to hide less interesting information and enable its user to quickly find clones that are likely to be relevant to certain migration approaches.

4.2.1 CPD

We use the CPD ("Copy/Paste Detector") tool for clone analysis, which is part of the PMD code analysis suite[1]. CPD uses string-based pattern matching algorithms to

[1] http://pmd.sourceforge.net/pmd-4.3.0/cpd.html

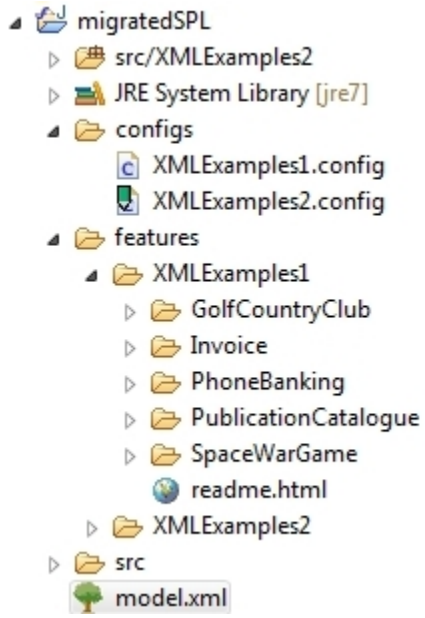


Figure 4.1: The result of migrating the example Java projects "XMLExamples1" and "XMLExamples2" with default settings.

detect code clones. Since it is implemented in Java, we can easily integrate it in an Eclipse plugin that extends the functionality of FeatureIDE.

CPD allows configuration through passing of parameters. Since all, but the required *minimumtilesize* parameter are already set to sensible values, we do not need to adjust much. Minimumtilesize describes the minimum size a clone needs to have to be listed in the results. We set this to 30. Other than that, the only thing we pass the tool are the source files we want to analyze.

The results of the analysis of CPD are returned in the form of an iterator over *Match* objects, where each *Match* represents a code clone. *Match* objects contain attributes that belong to the clone, such as its length in tokens and lines. Per occurrence of the clone, there is one *TokenEntry* object contained in the *markSet*. This is the full list of variables for *Match* objects:

tokenCount *int* the length of the clone fragment in string tokens.

lineCount *int* the length of the clone fragment in lines.

code *String* the cloned code fragment.

markSet $Set < TokenEntry >$ a set of TokenEntries, where each TokenEntry represents an occurrence of the clone and contains the following values:

tokenSrcID *String* the path to the file containing this occurrence.

beginLine *int* the index of the line at which the occurrence starts in the file.

This is the raw clone analysis data that we use as basis for our clone analysis assisted refactoring. To improve this further, we mainly implement enhancements to the presentation and add feature-awareness.

4.2.2 Presentation of Results

The default output of CPD, a .txt or .xml file with a list of clones, is hard to comprehend due to its size. For the projects we observe in our case study those results are longer than tenthousand lines. Since the clones are listed in the order they were detected it is no easy task to find those clones that indicate the best refactoring opportunities. To make it easier to comprehend, we hide some of the information (such as the full code fragment). Additionally, we provide an interactive interface that allows sorting and filtering the results by relevant properties. Relevant properties are mostly the clone length and occurrence count, which can be used to evaluate the clones size. Other important properties include information about the clone's occurrences locations in the code, for example if a clone occurs in more than one feature and in which one(s) it occurs.

To add feature-awareness to the clone analysis results of CPD, we exploit the fact that the implementations of different features is located in different directories. Since we know the full location path for each occurrence of a clone as well as the feature directory of our SPL project, we can determine the feature in which each occurrence is located. If all occurrences are located in the same features directory, it is an intra-feature clone. Otherwise it is an inter-feature clone. Since we know which feature is associated with which directory, we are able to determine the features affected as well as the number of features affected by a clone without much effort.

Furthermore, the result of CPD is a number of "Matches". To clarify the information we are presenting, we rearrange the it and calculate some additional properties. *lineCount* and *tokenCount* are presented in columns titled "Lines" and "Tokens". We count the number of different files referenced in the TokenEntries *tokenSrcID*s and present it in a column "Files". Clones can be expanded to show their "Occurrence"s, for which the additional information of the path to the containing file's location, as well as the line at which it starts is presented. This is the detailed list of attributes we present:

Clone the set of all occurrences of a fragment of code that occurs at least twice in the files we analyze. The properties of a clone are:

- **Lines** the length of the cloned code in lines.
- **Tokens** the length of the cloned code in tokens.
- **Files** the number of different files in which the clone occurs.
- **Type** the clone type. This can be inter-feature or intra-feature (see Section 3.3.2).

CloneOccurrence the particular occurrences of a cloned code fragment. The properties of a CloneOccurrence are:

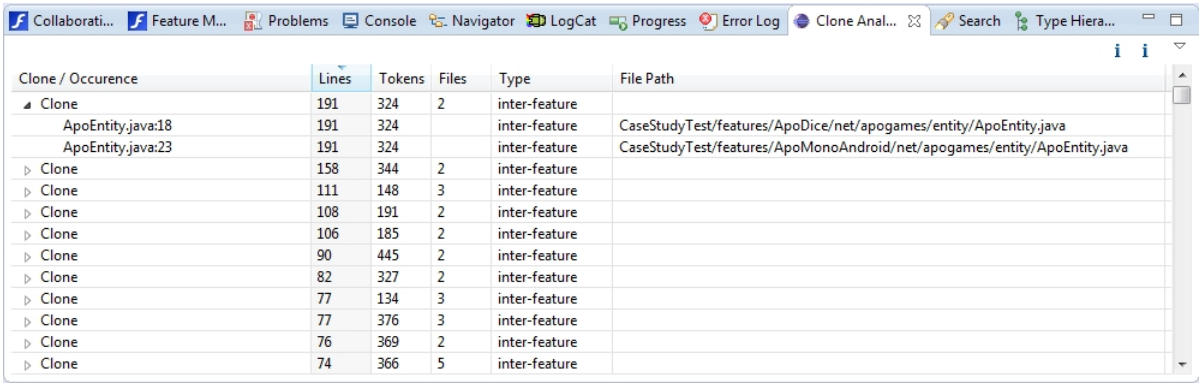


Figure 4.2: Clone Analysis View of our FeatureIDE extension.

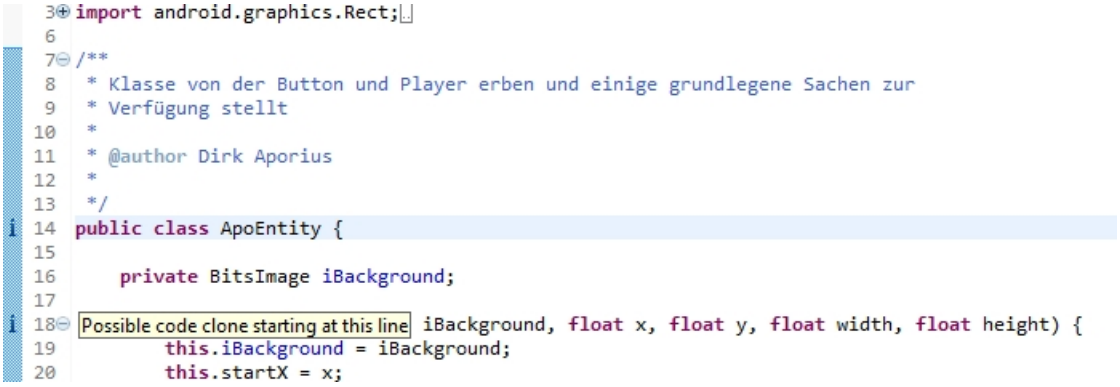


Figure 4.3: Eclipse marker indicating starting lines of detected clones.

File Path the path to file in which this occurrence is located.

Line the index of the line at which the cloned fragment starts.

After rearranging the data and adding the information we gained post-processing the clone analysis results, we display it in a table in an Eclipse view, as shown in Figure 4.2. By default, the table is sorted descending by clone length in lines, since we assume long clones to indicate good refactoring opportunities. Furthermore, it can be filtered by features, removing clones from the list that do not occur in selected features. Other than that, we designed the view so the clones can be navigated by meta information, while hiding the actual cloned code. As the clones may get very long, we hide the actual cloned code to make it easier to gain an overview about the clones.

We also use Eclipse markers to indicate code clones while browsing through code files, as shown in Figure 4.3. This makes it easy to spot clones in the code one is working on. It can also be used to navigate to the marker of a clone by means of the Eclipse problems view, which lists the markers.

5. Evaluation

In order to determine effective ways of supporting the specific migration approach described in Section 3.1, we implemented several approaches and conducted a case study to determine their usefulness. First we detail the basic conditions of the case study. Then we evaluate the degree of effectivity of several approaches and elaborate on the conclusions we reached. Finally we hypothesize on what other measures might be useful in the context of clone analysis and SPL migration.

5.1 The Case Study's Projects

The case study itself is conducted on a set of games implemented in Java for Android phones. The project sources were kindly provided by Dirk Aporius.

Project	Classes	Total LOC	Cloned Lines
ApoSnake	19	3975	2450
ApoClock	28	4933	2342
ApoMonoAndroid	24	7919	2680
ApoDice	19	3470	2216
myTreasureAndroid	27	7240	2222

Table 5.1: Metadata of the case studies projects and results from clone analysis.

There are five projects in total, developed through application of large-scale clone-and-own. Based on their AndroidManifests, the projects were implemented for use with different Android versions. The number of classes, physical lines of code as well as the number of cloned lines are shown in Table 5.1 for each project. This is a small case study, given the number of classes and lines of code. As we expected from clone-and-own created product variants, they contain a significant amount of clones, which is indicated by the cloned lines. The project source files that were provided by Dirk Aporius also included a library project, BitsEngineAndroid, which was developed by

Mark Wiedenhoeft and is used by all of the five projects. Not counting the BitEngine, the projects Java source files have a total size of 27537 lines.

Common points of the projects are mainly classes implementing graphic elements positioning, movement, rendering and similar functions. Graphs showing more detailed clone analysis results of the case studies projects are listed in the appendix. They contain additional information especially about the number and length of clones that occur in a particular number of projects. The presence of a significant amount of clones with occurrences in more than one project confirms that our approach has a chance for success with these projects.

5.2 The Case Study

We conducted the case study manually on a FeatureIDE project, that was created by migration as explained in Chapter 3. Since composition of Android products with FeatureHouse is not currently supported in FeatureIDE, we verified the integrity of the refactorings we made by manually composing the products with FeatureHouse (compare Section 5.2.5). In order to cover several different approaches to migration, we forwent a complete migration of the products in favor of sampling several different approaches.

Following a description of our actions and observations, we evaluate them as results of the case study. With the approach of migration through refactoring in mind, we evaluated clone analysis as a tool to refactor sets of clone-and-own created product variants into composition based SPLs. With this explorative study, we provide first experiences with the approach while identifying obstacles and opportunities for further research.

5.2.1 Migration towards a simple SPL

In the concept we described the first step of our migration approach, creating a simple SPL from the original products where each product is implemented in a single feature, expecting that it can be done for arbitrary product sets with only marginal effort. We did, however, meet some problems during the process described in Section 3.1.

We first implemented this for simple Java projects, specifically projects that do not use any libraries or resources other than the JRE and are implemented in ".java" files. Upgrading this version to support migration of projects such as we use in our case study, however, we encountered several problems. The biggest issue was that FeatureIDE did not support composition of Android projects with the FeatureHouse composer.

Obstacles that we encountered when using our migration tool to migrate the case studies projects include conditions that are imposed on Android projects, such as presence of certain XML and project property files, which were not present at the correct locations after composition. After manually correcting the setup of the project we met a problem of unidentified origin. The Android virtual machine returned the error message "Conversion to Dalvik format failed with error 1".

Another possible obstacle is posed by the BitsEngineAndroid, a library that is used by the projects. The five different projects use three different versions of the library. Implementing the SPL in a single project means, that only one version of the library can be present on the build path at the same time. This can be adressed by replacing the referenced file when building a project. Replacing all of them with the newest version of the library seemed to work, but without more information on the library version's compatibility we did not consider this a solution.

While the existence of these obstacles is important to note, overcoming them immediately is not necessary in the context of this case study, as our focus is on the analysis of the clone detection tools, which does not require the project setup to be in perfect condition. We chose to work around them by manually composing the products with FeatureHouse, while still using FeatureIDE functionality for feature modeling and presentation of the clone analysis data.

5.2.1.1 Evaluation

In this important first step of the migration approach, we encountered the biggest problem during our case study. Properties of the original projects that are attributed to their nature as Android projects prevented us from successfully migrating them into a FeatureIDE SPL project as we originally planned. While we did not manage to implement this migration step for the projects from our case study, however, we did show that it is possible for simple projects. Thus we emphasize that, although this is a setback, it does not compromise the approach to SPL migration. As it is not as trivial as we originally expected, we believe that the obstacles we identified under these circumstances warrant some attention for those who intend to implement this approach themselves.

The main source of the obstacles encountered during this step of the case study was dealing with the properties of Android projects. We infer that the difficulty of the first step of creating a simple SPL is contingent on the implementation circumstances and details of the original product variants. Certain properties of product variants will affect the complexity of this step more than others. For the application to real world systems, finding out which ones most likely lead to problems as well as testing general and specific solutions might be prudent.

5.2.2 Refactoring by Clone Size Heuristic

In Chapter 3 we elaborated on the approach of using a simple heuristic to indicate the refactoring potential of a clone. In simple terms we consider the biggest clones first in the hope of drastic code size reductions.

In Figure 5.1 you can see the clone analysis results of the unaltered case study projects, sorted descending by our heuristic ("size" column). On closer inspection, the `ApoEntity` class is present in all five projects with only very little diverging content. The original `ApoEntity` implementations have a total length of 2175 lines (435 lines x5). By moving

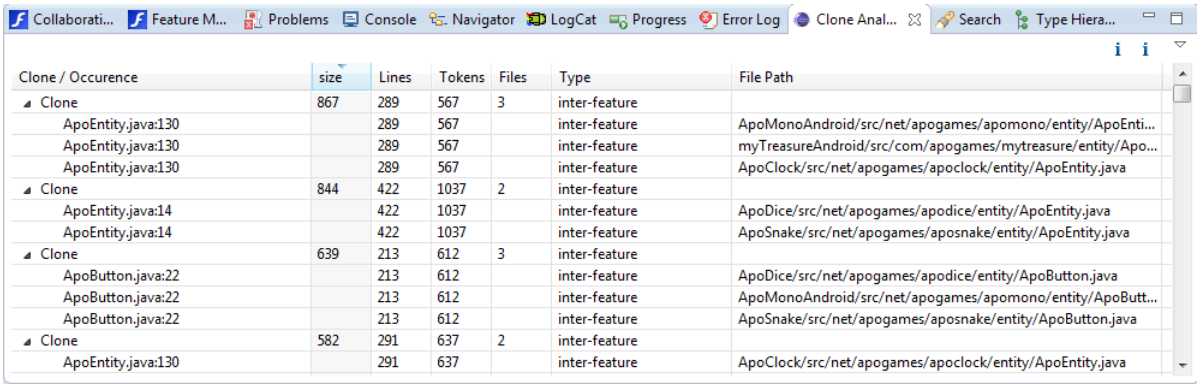


Figure 5.1: The clone analysis view of our plugin, showing the results for the unaltered case study projects.

the cloned code to the Base feature, we managed to reduce the total code size by 1378 lines (Base: 372 lines, refinements: 5x 85 lines). During this refactoring we noticed some interesting properties of the original code, on which we elaborate in the following paragraphs. Later on we also refactored the implementation of the class with the second biggest clones, `ApoButton`. By doing so, we reduced the size of its implementation from 1256 lines (240 lines x 4 + 296 lines) to 586 lines (Base: 196 lines, refinements: 4 x 67 lines + 122 lines).

There is one incidence of a method name being changed in a peculiar way. The name of the getter method for the private boolean variable "bVisible" is "isVisible" in the projects ApoClock and myTreasureAndroid, while in the other three projects it is "isBVisible". Other getters for other boolean variables ("bUse", "bSelect", ...) use the latter naming pattern ("isBUse", "isBSelect", ...) in all projects. This can lead to problems with the migration, since the composition of methods as done by FeatureHouse is based on the method's signature. Methods with different names would be treated as two completely different methods, which can be a source of errors in the long run.

```
1 import android.graphics.Rect;
2 import net.gliblybits.bitsengine.core.BitsImage;
3 import net.gliblybits.bitsengine.render.BitsGraphics;
```

Listing 5.1: Imports of ApoSnake and ApoDice's ApoEntity.java

```
1 import net.gliblybits.bitsengine.utils.BitsRect;
2 import net.gliblybits.bitsengine.graphics.opengl.BitsGLImage;
3 import net.gliblybits.bitsengine.graphics.opengl.BitsGLGraphics;
```

Listing 5.2: Imports of ApoClock, MyTreasureAndroid and ApoMonoAndroid's ApoEntity.java

Another noticable difference is the usage of different library classes in the ApoEntity implementations, shown in Listing 5.1 and Listing 5.2. In the projects ApoSnake and

ApoDice the classes `Rect`, `BitsImage`, and `BitsGraphics` are used in places where the other projects use `BitsRect`, `BitsGLImage`, and `BitsGLGraphics` respectively. That the implementations are still very similar suggests that those classes have similar interfaces.

In fact, if not for those two differences that occur between the products, the implementations of the ApoEntity class would be the same (An example of this can be found later on in this chapter, in Figure 5.4). Especially the different class usage has a big impact on how much of the code can be transferred to a common ancestor feature. While FeatureHouse supports refining classes and methods by adding implementation artifacts in their bodies, the different classes lead to different method heads (`public BitsRect getRec()` instead of `public Rect getRec()`). This in turn means, that although in most cases the method bodies were identical, this similarity could not be exploited to reduce code size.

5.2.2.1 Evaluation

Using the size heuristic we implemented, we managed to identify very large clones in the classes `ApoEntity` and `ApoButton`. Following this lead, we noticed that those two classes are almost the same to the letter (except for the differences mentioned above). Moving the common parts into a common base feature, we were able to reduce the total code length by 797 lines for ApoEntity and 670 for ApoButton. Considering the total size of all features was 27537 lines, this is a considerable improvement for minimal effort.

Taking the longest continuous code clones and extract the cloned code to a common ancestor feature is an effective way of quickly reducing code size. We have shown that clone analysis can be effective in identifying large clones for us so we can pick the figurative low-hanging fruits. By its nature, this will have diminishing returns, which is why it might be prudent to switch to a different approach once the best refactoring opportunities have been taken.

5.2.3 Refactoring by Product Variant

In the previous chapter we mentioned that one might intend to prioritize a specific variant during migration. As the clones are feature-aware due to our postprocessing, we can support the option of hiding clones that do not occur in selected feature or project implementations.

To show the impact of this feature-aware filtering, we measured how far the number of shown clones is reduced when applying the filter for each of the original products. The results are presented in Table 5.2. For different products, the clone selection is reduced to an amount between half to one-third of the original list, depending on the product.

We then started migrating, prioritizing clones that occur in the ApoClock implementation. The first thing to note is that for the projects of our case study, the biggest clones (in the classes `ApoButton` and `ApoEntity`) occur in all original products. Others, such as a clone of 90 lines in the classes `ApoClockComponent` and `ApoMonoComponent`

Filter	# of Clones
ApoSnake	297
ApoClock	311
ApoMonoAndroid	381
ApoDice	274
myTreasureAndroid	304
no filter	808

Table 5.2: Number of clones shown, after removing those without occurrences in the selected product.

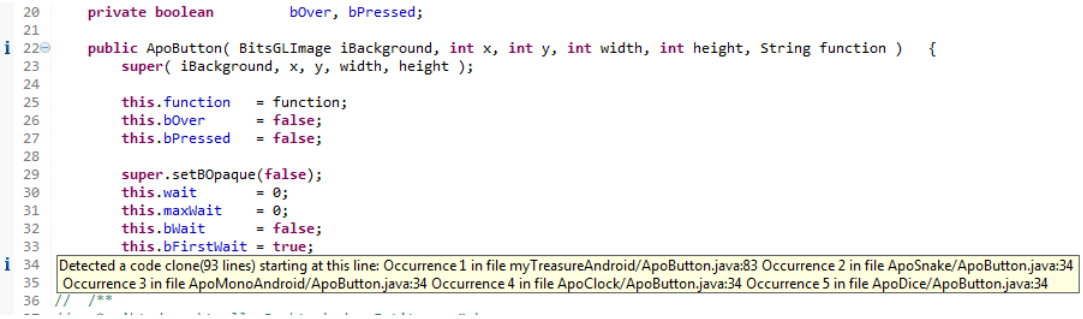


Figure 5.2: An excerpt of the "ApoButton.java" implementation of the ApoClock feature, as displayed in the Eclipse Java editor. Clones are marked at the left margin.

indicate a refactoring opportunity, but are too small to catch the eye when viewing clone results without the filter.

All in all we managed to show that migration with priority on a specific product variant can be supported by feature-aware clone analysis.

5.2.4 Refactoring by Predefined Features

The third and final migration approach we are considering in the case study is selective extraction of features known to be implemented in the original projects. To assist this, we use Eclipse markers to indicate the presence of clones in the editor. This way a known implementation of such a feature can be used as a starting point, with the markers making developers aware of code that is likely related to the feature. Additionally, we made information about the clone's length and location available through the marker's tooltip.

We determined that "Button" is a sensible feature in the domain "Android games", or more general even "Android applications" that has reuse value and is known to be implemented by at least one original product. For this use case, we started at the button implementation in the ApoClock project's `ApoButton` class. Figure 5.2 shows a section of this class, including two markers. The tooltip suggests that large parts of `ApoButton`'s functionality is the same in all of our five projects, which is a strong

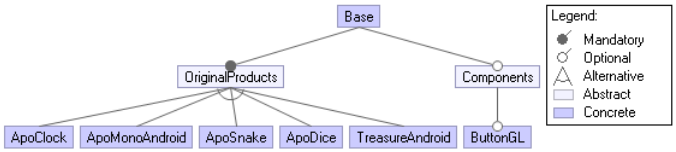


Figure 5.3: The feature model after extracting the feature Button from the original products.

indicator that we can refactor this to extract a feature implementation that can be reused by potential new products.

During the extraction we encountered the same problems we mentioned in Section 5.2.2 in the context of refactoring `ApoEntity`. In the class `ApoButton`, the same classes were imported as in ApoEntity, in the same product variants (Listing 5.1, Listing 5.2). Also, the implementation of `ApoButton` in myTreasureAndroid contains some additional elements that are rendered. As this is specific functionality, we did not extract it into the general Button implementation. From what we gathered during an analysis of the different classes, we inferred that one of them, Listing 5.2, is the more recent implementation and extracted this version into a new feature "Button". The feature model shown in Figure 5.3 is the result of this extraction from the unaltered original products. To assure that the refactoring is product-preserving, retaining the behavior of the existing products, we added the Button feature to the configurations of the ApoClock, ApoMono and ApoTreasure products, from which the implementation was extracted.

5.2.4.1 Evaluation

Given a starting point in the `ApoButton` implementation of the ApoClock project, we could use the markers in the source editor to quickly find related sources. As those had a high degree of similarity to the original implementation, we successfully extracted a sensible feature from our case studies projects with little effort. In regard to supporting SPL migration, we believe markers have a lot of potential since they indicate relations between the project implementations that would not otherwise be visible. Having this information available while browsing the code should also prove useful, if development of a product is done while the migration is in process.

5.2.5 Obstacles to Migration

While conducting the case study we came across several obstacles that impaired the migration process. In addition to the problems we already described in the specific migration approaches, we will describe general problems we encountered here and then elaborate on possible causes and solutions.

A big obstacle to the migration is the particular naming standard used in our case studies projects: The majority of classnames are prefixed with the project name. For

```
381    public Rect getRec() {
382        return new Rect((int)this.getX(), (int)this.getY(), (int)(this.getWidth() + this.getX()), (int)(this.getHeight() + this.getY()));
383    }
```

```
381    public BitsRect getRec() {
382        return new BitsRect((int)this.getX(), (int)this.getY(), (int)(this.getWidth()), (int)(this.getHeight()));
383    }
```

Figure 5.4: Two different Rectangle classes being used in different implementations of“ApoEntity.java”.

example the classes implementing the game menu in projects ApoSnake and ApoDice are called `ApoSnakeMenu` and `ApoDiceMenu` respectively. This leads to several problems. First of all it hampers clone detection. The clone analysis library we use tolerates different names of identifiers, as characteristic for type II clones. Different classnames however can lead to fragmentization of clones that would otherwise correctly be parsed as a whole, as well as shorten some clones so far that they become too short to be shown in the results.

In addition to that, FeatureHouse only refines classes and methods if their names and signatures respectively are identical. With this naming standard however, classes that are semantically and structurally almost identical will be treated as unrelated.

Another obstacle, although unrelated to the clone analysis, was the lack of automation for the refactorings. We had to manually move, delete and copy code fragments and rename classes and methods, which is a potential source of errors. Beyond that, when extracting features as in Section 5.2.4, we had to manually edit the product configurations, so the functionality of the original products remains unaltered. Ideally, we would like to use automatic refactorings similar to the ones implemented in the Eclipse IDE (e.g. “Extract Feature..” analogous to “Extract Class..”), that automatically create features and/or move code and/or rename features/classes/identifiers as necessary to perform the refactoring.

5.2.6 Importance of Preparational Refactorings

While conducting our case study we observed that a big part of the problems we encountered could have been prevented if the projects had been refactored in beforehand with the migration in mind.

One example for this are classes such as “ApoSnakeMenu” and “ApoDiceMenu”, which both implement a menu in their respective projects. Should we try to extract a new feature “Menu” from this, we have to refactor both classes so they have the same name. We expect this to carry less risk and take less effort to do before migrating the separate projects into a SPL since there is no danger of interactions and IDE support is more likely to be available (e.g. Eclipse renaming).

Another example can be found in different implementations of the `ApoEntity` class, which is almost identical throughout the projects but for two different sets of library classes being imported. As can be seen in Figure 5.4, two different classes are used to implement rectancles. Since type II clones only tolerate different names of identifiers,

not of classes, this splits a long code clone in two parts. Refactoring the code so that implementation detail is hidden behind an interface or a facade would improve the clone analysis results and most likely allow even further reduction of code size per refactoring.

6. Conclusion

The process of creating an equivalent SPL for a set of existing products is called SPL migration. SPL migration is usually an expensive and expansive venture that can lead to disruptions in the original projects workflow or development of redundant code. In order to facilitate SPL migration of clone-and-own created products that allows continuous development of products during migration, we proposed a two-step approach. The first step is creating a SPL which lists each of the original product variants as a feature. The second step is the iterative refactoring of this SPL, until the desired result is achieved. A key point of this is the use of clone analysis tools during the second step to pinpoint code segments that can be refactored at low cost with great profit.

To evaluate the use of clone analysis in this context and the approach in general, we conducted an explorative case study on a set of five android Java projects that were created by application of clone-and-own. We implemented the first step of our approach for basic Java projects, but it transpired that the automation of this step is not trivial to implement for Android projects. The assumption that the effort of the initial migration to a token SPL is far outweighed by the second step holds, but we expect the implementation of this first step for arbitrary projects to effect the applicability of the approach in practice as it could significantly reduce the up-front investment costs of the SPL migration.

For the second step, we implemented postprocessing for a clone analysis tool, then integrated the results into an interactive tool that supports the iterative refactoring and tested it for varying use cases. Our results suggest that clone analysis data is a valuable resource in the context of feature-oriented refactoring, especially since the product variants we investigated had such a high number of clones, with between 30% and 64% of the product variants being part of clones. Big clones indicate opportunities to quickly reduce the total code size by removing redundant implementations. Very short clones did not reveal good refactoring opportunities as the similarities they indicate are too

fine-grained, but they can be displayed to assist in gaining an overview of similar parts in different variants.

In our case study, we assumed that clone-and-own created product variants will contain many clones, which was correct for our case study's projects. We performed a preliminary analysis of the initial variants that asserted the presence of a high percentage of clones in the implentation. However, some clone-and-own created product variants might contain a significantly smaller amount of clones or, on the contrary, product variants that were not created by clone-and-own might contain a sufficient number of clones if they were developed under similar constraints (purpose, naming conventions, use of software design patterns). The degree of clone occurrence in a set of products might be a more precise way to discern the applicability of our approach for a given set of products.

While some of the clones we found could be refactored immediately, other clones reveal largely equal implementations that can not be consolidated into a single feature. We concluded that preparational refactorings can increase the total amount of clones detected as well as the percentage of clones that belong to the former category. Some of the latter were caused by the use of different classes from external libraries that lead to variations in method signatures, which cause the FeatureHouse composer to recognize them as entirely different methods. Hiding the actual implementation using software design patterns (e.g. Facade), or extracting those different implementations into two different features might be possible solutions for this problem but devising a universal solution to this requires further research.

As we sampled different approaches to the refactoring step, a full migration of product variants should be done to identify problems that occur later on during the refactoring process. We noticed, especially when we identified refactoring possibilities with the clone size heuristic, that as we refactor the code indicated by the biggest clones, returns of this approach diminish. Such further study could answer the following questions: How much of the initial variants is going to be migrated once we reach the point where clone analysis does no longer provide sensible refactoring opportunities? Do the remaining parts require a different approach to refactor them further, or are the goals of maintainability and reusability already reached?

We proposed a new way of SPL migration, tested several specific approaches in a case study, managed to validate the general applicability of the migration approach and identified several challenges and questions that warrant further research. Thus we met the goal we set for this thesis and conclude this explorative case study to be a success.

A. Appendix

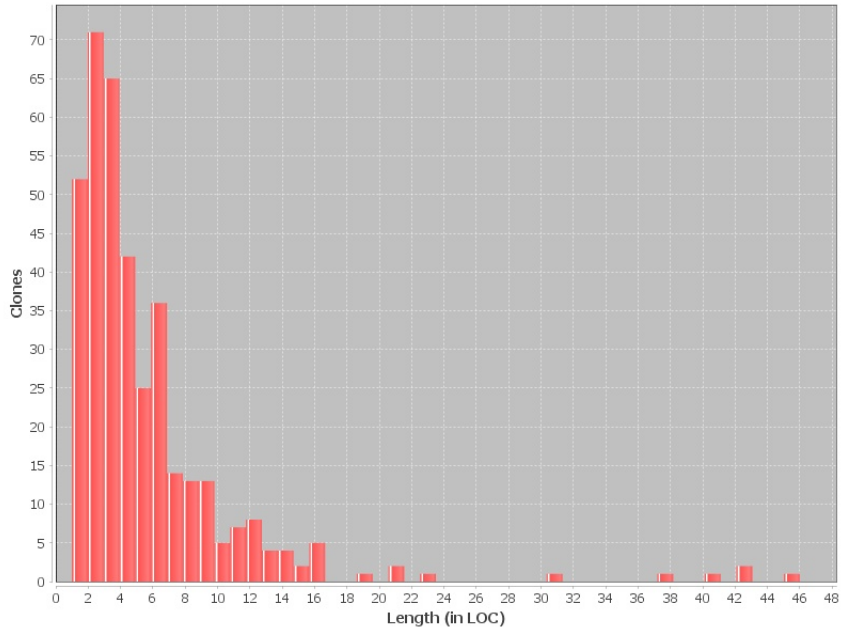


Figure A.1: A histogram of the length of clones within the original products.

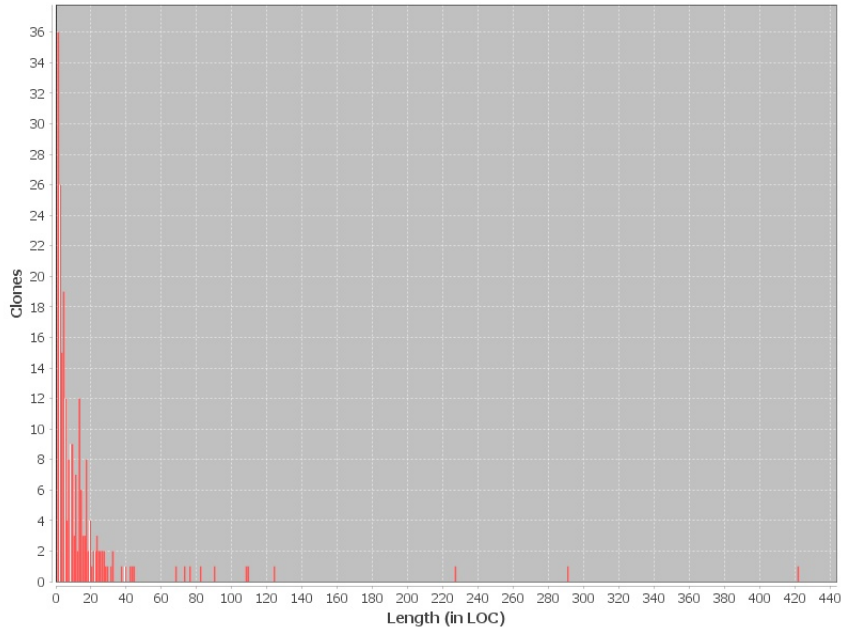


Figure A.2: A histogram of the length of clones with occurrences in two different original products.

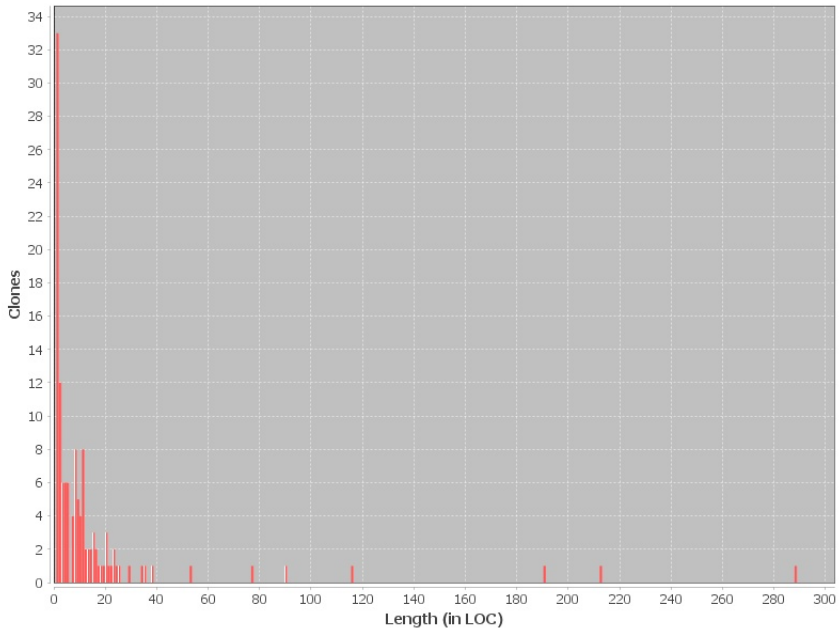


Figure A.3: A histogram of the length of clones with occurrences in three different original products.

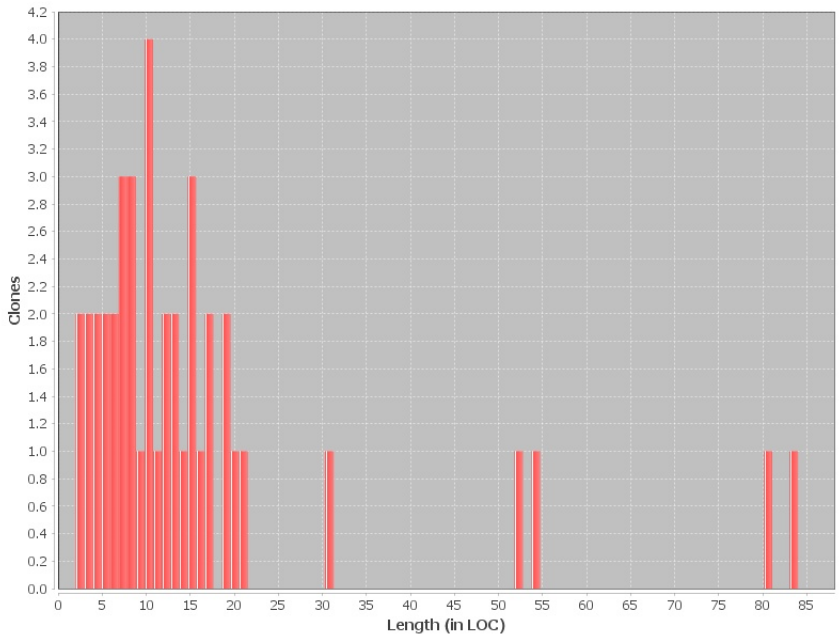


Figure A.4: A histogram of the length of clones with occurrences in four different original products.

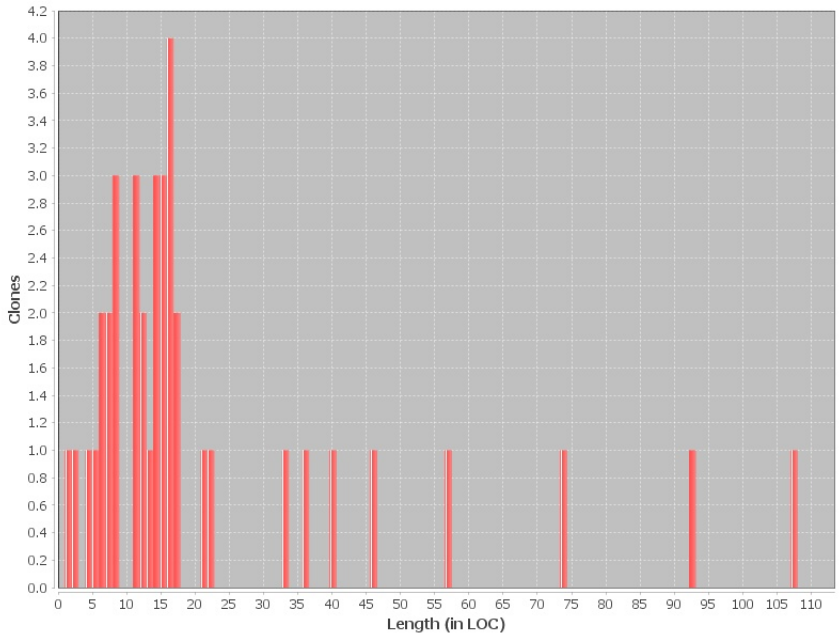


Figure A.5: A histogram of the length of clones with occurrences in five different original products.

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013. (cited on Page 1, 5, 6, 7, and 8)

[AJB+14] Michał Antkiewicz, Wenbin Ji, Thorsten Berger, Krzysztof Czarnecki, Thomas Schmorleiz, Ralf Lämmel, tefan Stănciulescu, Andrzej Wąsowski, and Ina Schaefer. Flexible product line engineering with a virtual platform. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 532–535. ACM, 2014. (cited on Page 3)

[AK09] 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)

[AKL09] Sven Apel, Christian Kastner, and Christian Lengauer. Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering*, pages 221–231. IEEE Computer Society, 2009. (cited on Page 15)

[AKL13] Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The FeatureHouse experience. 39(1):63–79, 2013. (cited on Page 8)

[AVMDP02] Giuliano Antoniol, Umberto Villano, Ettore Merlo, and Massimiliano Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755–765, 2002. (cited on Page 9)

[Bak95] Brenda S Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995. (cited on Page 4 and 9)

[BFB99] Kent Beck, Martin Fowler, and Grandma Beck. Bad smells in code. *Refactoring: Improving the design of existing code*, pages 75–88, 1999. (cited on Page 11)

[CN98] Michael A Cusumano and Kentarō Nobeoka. *Thinking beyond lean: how multi-project management is transforming product development at Toyota and other companies*. Simon and Schuster, 1998. (cited on Page 5)

[CN01] Paul Clements and Linda Northrop. Software product lines: practices and patterns. 2001. (cited on Page 6)

[DB07] Mark Dalgarno and Danilo Beuche. Variant management. In *The British Computer Society. 3rd British Computer Society Configuration Management Specialist Group Conference. Oxford*, page 6. Citeseer, 2007. (cited on Page 3)

[DKB11] Slawomir Duszynski, Jens Knodel, and Martin Becker. Analyzing the source code of multiple software variants for reuse potential. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 303–307. IEEE, 2011. (cited on Page 2)

[FBBO99] Martin Fowler, Kent Beck, John Brant, and William Opdyke. *Refactoring: Improving the Design of Existing Code*. 1999. (cited on Page 9 and 11)

[FS15] Wolfram Fenske and Sandro Schulze. Code smells revisited: A variability perspective. In *Proceedings of the Ninth International Workshop on Variability Modelling of Software-intensive Systems*, page 3. ACM, 2015. (cited on Page 16)

[FTS14] Wolfram Fenske, Thomas Thüm, and Gunter Saake. A taxonomy of software product line reengineering. In *VaMoS*, page 4, 2014. (cited on Page 1 and 12)

[GFGP06] Reto Geiger, Beat Fluri, Harald C Gall, and Martin Pinzger. Relation of code clones and change couplings. In *Fundamental Approaches to Software Engineering*, pages 411–425. Springer, 2006. (cited on Page 1 and 4)

[JDHW09] Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *Proceedings of the 31st International Conference on Software Engineering*, pages 485–495. IEEE Computer Society, 2009. (cited on Page 4)

[KCH+90] Kyo C Kang, Sholom G Cohen, James A Hess, William E Novak, and A Spencer Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical report, DTIC Document, 1990. (cited on Page 6)

[KG06a] Cory Kapser and Michael W Godfrey. " cloning considered harmful" considered harmful. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 19–28. IEEE, 2006. (cited on Page 3 and 4)

[KG06b] Cory J Kapser and Michael W Godfrey. Supporting the analysis of clones in software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):61–82, 2006. (cited on Page 4, 8, and 9)

[Kru02] CharlesW Krueger. Easing the transition to software mass customization. In *Software Product-Family Engineering*, pages 282–293. Springer, 2002. (cited on Page 14)

[MLM96] Jean Mayrand, Claude Leblanc, and Ettore M Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 244–253. IEEE, 1996. (cited on Page 8)

[MNK+02] Akito Monden, Daikai Nakae, Toshihiro Kamiya, Shin-ichi Sato, and K-i Matsumoto. Software quality analysis by code clones in industrial legacy software. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 87–94. IEEE, 2002. (cited on Page 4)

[Opd92] William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992. (cited on Page 11)

[OVMMW99] Magnus C Ohlsson, Anneliese Von Mayrhauser, Brian McGuire, and Claes Wohlin. Code decay analysis of legacy software through successive releases. In *Aerospace Conference, 1999. Proceedings. 1999 IEEE*, volume 5, pages 69–81. IEEE, 1999. (cited on Page 11)

[PBVDL05] Klaus Pohl, Günter Böckle, and Frank Van Der Linden. Software product line engineering. *Springer*, 10:3–540, 2005. (cited on Page ix, 5, and 7)

[RC13] Julia Rubin and Marsha Chechik. A framework for managing cloned product variants. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 1233–1236. IEEE Press, 2013. (cited on Page 1)

[RCK09] 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 2, 8, 9, 10, and 11)

[RKBC12] Julia Rubin, Andrei Kirshin, Goetz Botterweck, and Marsha Chechik. Managing forked product variants. In *Proceedings of the 16th International Software Product Line Conference-Volume 1*, pages 156–160. ACM, 2012. (cited on Page 3 and 4)

[SKR08] Sandro Schulze, Martin Kuhlemann, and Marko Rosenmüller. Towards a refactoring guideline using code clone classification. In *Proceedings of the 2nd Workshop on Refactoring Tools*, page 6. ACM, 2008. (cited on Page 3)

[STKS12] Sandro Schulze, Thomas Thüm, Martin Kuhlemann, and Gunter Saake. Variant-preserving refactoring in feature-oriented software product lines. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, pages 73–81. ACM, 2012. (cited on Page 12 and 16)

[TKB+14] Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. Featureide: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014. (cited on Page 2 and 19)

[TOHSJ99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M Sutton Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software engineering*, pages 107–119. ACM, 1999. (cited on Page 8)

[WZ07] Andreas Winter and Jörg Ziemann. Model-based migration to service-oriented architectures. In *International Workshop on SOA Maintenance and Evolution*, pages 107–110. CSMR, 2007. (cited on Page 12)

[YGM06] Kentaro Yoshimura, Dharmalingam Ganesan, and Dirk Muthig. Assessing merge potential of existing engine control systems into a product line. pages 61–67, 2006. (cited on Page 4)

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

Magdeburg, den