Analysis and Removal of Code Clones in Software Product Lines

Dissertation

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von: Diplom Informatiker Sandro Schulze

geb. am 04.06.1980 in Osterburg (Altmark)

Gutachter:

Prof. Dr. Gunter Saake,
Prof. Dr. Ina Schaefer,
Prof. Dr. Michael W. Godfrey

Ort und Datum des Promotionskolloquiums: Magdeburg, 18.01.2013

University of Magdeburg

School of Computer Science

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Dissertation

Analysis and Removal of Code Clones in Software Product Lines

Author:

Sandro Schulze

October 22, 2012

Advisors:

Prof. Gunter Saake

Otto-von-Guericke University of Magdeburg

Prof. Ina Schaefer

TU Braunschweig

Prof. Michael W. Godfrey

University of Waterloo (Ontario, Canada)

Schulze, Sandro:
Analysis and Removal of Code Clones in Software Product Lines
Dissertation, University of Magdeburg, 2012.

Abstract

Software maintenance is the main driver of total costs in the lifecycle of long-living software systems. Code clones, that is, the replication of code fragments across the system, decrease maintainability: It increases the code size and hinders manual code change, inspection, and analysis. Intensive research has been spent in the last two decades to determine the nature of clones, specifically why and where they occur as well as whether they impair the maintenance of software systems. While recent studies expressed doubt on the general harmfulness of clones, it is commonly accepted that the awareness of existing code clones in software system is indispensable in any case.

Recently, *software product line engineering* gained momentum since it provides a systematic approach for reuse amongst a set of similar programs, commonly referred to as *software product lines (SPL)*. An SPL allows the programmer to manage a set of programs by describing variabilities and commonalities between them in terms of *features*. In this context, a feature is an increment in end-user visible functionality. As a result, a particular program can be derived by selecting the desired features and subsequently composing all corresponding assets.

The goal of this thesis is to bridge the gap between both research areas, reengineering & maintenance (where code cloning belongs to) and software product lines. We argue, that SPLs evolve even more than single software systems and thus, maintenance becomes even more complex. Hence, it is important to figure out specialities of SPLs regarding software reengineering. In this thesis, we focus mainly on code clone analysis and removal.

First, we present results from empirical studies that emphasize the existence of clones in SPLs. More specifically, we provide insights why clones occur in SPLs and point out differences between compositional and annotative SPLs.

Second, we propose variant-preserving refactorings for compositional software product lines as mean for code clone removal. We present particular refactorings in a catalogue-like manner and demonstrate their applicability by means of a case study.

Acknowledgements

Writing a dissertation is a long road to go with many dead-ends and junctions. Hence, it is not easy to follow the right way straight to the final destination and more than once, this way is characterized by throwbacks and privations. But it also means a development of scientific and social skills in a way only few professions provide. However, I would have been never able to follow this path and making this invaluable experiences without many people that accompanied and supported me in different ways. First and foremost, I want to thank Thekla and my lovely son Richard. Their support, love, and sympathy gave me the power to make my way and overcome times of uncertainty and doubts. Furthermore, I want to thank my parents for believing in me and granting me an invaluable independence already in young years.

During my life as a researcher, many people supported me as teacher, mentor, or friend. First, I want to thank Sven Apel, for both, being a friend and a mentor. He actually draw my interest in scientific research, back in 2006, and accompanied me all the way. He helped me to understand research in its entirety, by controversial discussions, constructive criticism, and by showing me different perspectives on certain research topics. In that, he made me a lot the researcher that I am today.

Second, I want to thank my advisor Gunter Saake, who gave me the opportunity of pursuing an academic career. During the time in his research group, he supported me with guidance but also with high degree of freedom, which essentially contributed to my development as a researcher.

I further want to thank my external advisors Ina Schaefer and Mike Godfrey. Especially in the final stage of my thesis, Ina provided me with valuable feedback and a different perspective on my research. Furthermore, I am grateful to her for the chance of being a researcher in future and for taking off the load in the crucial phase of my dissertation. Mike not only agreed to be part of my committee and took the long journey from Canada to attend the PhD defense, which is a great honor for me. He also provided useful feedback and tips for improving the final version of this dissertation.

Furthermore, I want to express my gratitude to the (current and former) members of the Database Research Group at the University of Magdeburg, in particular, Christian Kästner, Marko Rosenmüller, Martin Kuhlemann, Norbert Siegmund, Janet Siegmund, Thomas Thüm, Mario Pukall, Andreas Lübcke, and Martin Schäler. It was always a pleasure to work in such a creative, productive and friendly atmosphere.

A special thank goes to the Metop research institute, namely Thomas Leich and Dan Klingenberg, who not only showed me the industrial side of life but also accompanied me in my very early days as a researcher.

Finally, I want to thank a couple of colleagues for collaborations and discussing a broad range of research topics. In particular, I like to thank Elmar Juergens, Mike Godfrey, Jan Harder, Nils Göde, the PASED gang (namely, Miguel Ferreira, Mauricio Aniche, Dimitris Athanasiou, Daniele Romano, and Martin Beck), and Jörg Liebig.

Contents

List of Figures

List of Tables

1. Introduction

Code clones have been recognized to be the most intrinsic and worst code smell in software systems [Fow00]. Indeed, a multitude of studies account for the existence of code clones in such systems (e.g., [Bak95, BYM$^+$98, RC10]). Generally, they are used in a *copy,paste&adapt* fashion to reuse existing part of the source code. Recently, *Software Product Lines (SPLs)* have been proposed as a more structured approach for reusing source code artifacts (as well as non-code) amongst similar, variable software systems. To this end, different languages, paradigms, and implementation approaches have been proposed that partially overcome problems of current approaches for implementing highly variable and customized software systems. This thesis focuses on analysis of software product lines with respect to code clones. In particular, we investigate whether clones exist and how to characterize them in software product lines, depending on the respective implementation approach. Furthermore, we propose a first approach for code clone removal in SPLs by means of refactoring.

1.1 Overview

Replicated code fragments, commonly referred to as *code clones*, have been subject to intensive research for over two decades. Since they play a pivotal role in the process of software maintenance, considerable effort has been expended to analyze when and how code clones negatively influence software quality and maintenance. Most commonly, researchers report about inconsistent changes and propagating and introducing errors as the main drawbacks of code clones for software quality (e.g., [JDHW09]). Additionally, increased code size and multiple modifications for one change request impede maintenance of the software systems as well. In contrast, recent studies express doubt on the longstanding sentiments about the harmfulness of clones. In particular, they show that code cloning is used as kind of implementation concept such as *templating* or *forking* and that clones are relatively stable with respect to changes [KG06, GH11].

```
class Stack {
  int pop() {/*...*/}
  #ifdef Undo
  int backup;
  void undo() {/*...*/}
  #endif
  #ifdef Peak
  int peak() {/*...*/}
  #endif
  void push(int v) {
    #ifdef Undo
    backup=peak();
    #endif
    /*Common Code*/
  }
}
```

(a) annotative

Feature BaseStack

```
class Stack { ...
  void push(int v) {/*...*/}
  int pop() {/*...*/}
}
```

Feature Peak

```
refines class Stack {
  int peak() {/*...*/}
}
```

Feature Undo

```
refines class Stack { ...
  int backup;
  void undo() {/*...*/}
  void push(int v) {
    backup=peak();
    original(v);
  }
}
```

(b) compositional

Figure 1.1: Examples for (a) annotative and (b) compositional implementation approach for software product lines

However, while code clone research mainly focuses on general purpose (monolithic) software systems, software development changed from single programs to program families in recent past. To this end, software product line engineering provides means to develop a set of related systems from a common code base. The different programs (also called *variants*) that are part of the resulting SPL can be described by their commonalities and variabilities in terms of *features*. Consequently, a particular variant of a software product line can be derived by selecting the respective features. Although it is still a quite new way of developing software systems, the product line approach has been adopted by industrial as well as open source systems and it is expected to increase in the future [GLA+09, JB09].

Different approaches exist for implementing software product lines that go beyond the often used copy-and-branch approach. In this thesis, we mainly focus on two categories: *annotative* and *compositional* implementation techniques. A prominent example for the annotative approach is the *C preprocessor (CPP)*. The CPP is widely adopted in industry to express variability in source code. To this end, the respective code fragments are annotated with constructs such as `#ifdef` or `#endif`. In Figure 1.1a, we show an example of cpp usage for introducing variability in a simple stack implementation for features *Undo* and *Top*. For generating a concrete stack program, the desired features can be selected, usually specified as a configuration file or command line parameter. Afterwards, the preprocessor includes the code belonging to the selected features for compiling the final program (while excluding all other features).

Although the CPP provides a simple but powerful and language-independent way of introducing variability in software systems, it has received a lot of criticism in the literature, culminating in terms such as "#ifdef hell" and "#ifdef considered harmful" [SC92, LST$^+$06a]. This criticism mainly relies on several studies that claim negative effects of the CPP on code quality, maintainability, and readability [Fav95, KS94, Fav97, EBN02, AVRGC08]. In fact, the preprocessor breaks with the fundamental and widely accepted concept of separation of concerns.

Despite this heavy and contiguous criticism, the CPP is the mechanism of choice in practice for expressing variability in software systems. For instance, HP implemented the software for their printers as a software product line with approximately 2 000 features using the C preprocessor [PO97, Ref09]. Another example from the open source domain is the Linux kernel that is under continuous development since three decades and make use of the CPP for variability, meanwhile consisting of more than 5 000 features [TSSPL09, SLB$^+$10]. Besides the widely used C preprocessor, proprietary tools for product line development such as *pure::variants* or *Gears* even provide their own preprocessor [BPSP04, Kru02].

However, in academia not much attention is given to annotative approaches (and the C preprocessor in particular). Instead, academic research focusses on *compositional* approaches to implement software product lines. A prominent approach is *Feature-Oriented Software Development (FOSD)* that aims at modularizing features for efficient composition and reuse [AK09]. Different implementation techniques for FOSD exist such as *components* [Sam97, SGM02], *collaboration-based* design [SB02], *Aspect-Oriented Programming (AOP)* [KLM$^+$97], *Feature-Oriented Programming (FOP)* [Pre97], or *generative programming* [CE00]. In this thesis, we mainly focus on FOP as compositional approach of our choice. The pivotal idea of FOP is to modularize a feature into a cohesive unit called *feature module*[1]. Then, a concrete program can be generated by composing these features, based on a user-specific feature selection (i.e., configuration). In Figure 1.1 b, we show an example for feature-oriented software product lines using FEATUREHOUSE, a compositional approach based on superimposition [AKL09]. As with the annotations, we use the Stack SPL example, encompassing two features *Peak* and *Undo*. Furthermore, we added the feature *BaseStack*, representing the least common code between all variants of the SPL (equivalent to the code that is not annotated in Figure 1.1 a. For each feature, we can add classes, methods, or fields but also extend existing ones. To generate a variant, a stakeholder selects the features of interest, which are then composed to the final program.

However, independent of the chosen implementation approach, software product lines exhibit certain characteristics that are similar to those of standalone systems. First and foremost, even SPLs undergo the process of software evolution. Hence, a continuous process of change takes place, indicated by adding, deleting, and changing source code. Unfortunately, only little is known how this process influences maintainability or quality of product lines and how does this diverge compared to standalone software systems. In particular, and most of our interest, no work exists that investigates the role of code clones in software product lines. We argue, that analyzing software product lines

[1]More precisely, every artifact that belongs to a certain feature (code *and* non-code artifacts) is encapsulated by this feature module.

regarding its maintenance and reengineering opportunities and requirements is crucial for their durability and efficiency (regarding further development) in the same way as for single software systems. Since code clones are a very well-known phenomenon that is widely considered to potentially hamper the maintainability of software systems, we think that this is a good starting point to put a stronger emphasis on reengineering in software product lines.

1.2 Contribution

The goal of this thesis is twofold. First, we provide insights on code cloning in software product lines. In particular, we emphasize to what extent code clones occur in SPLs and whether differences exist regarding the implementation approach of SPLs. Furthermore, we provide some characteristics of these clones as a first step towards managing such clones proactively or even avoiding them in future. Second, we present a first approach of how to remove code clones in software product lines by applying refactorings. While this is a common and well-explored approach in standalone programs, refactoring is a non-trivial task in the presence of variability. In particular, we present how to find clone refactoring candidates and how to take variability into account during the refactoring process.

Characterization of Code Clones in SPLs

We present empirical studies to show that code clones occur quite frequently in software product lines, independent of the actual implementation approach. Based on our analysis, we specifically show that code clones in feature-oriented SPLs mainly occur between alternative features. Moreover, our study reveals that these clones encompass complete syntactical units such as conditional branches or even methods. Both characteristics indicate a certain potential for code clone removal, though we can say nothing about the harmfulness of these clones.

Afterwards, we show code clones exist in annotated SPLs as well, though less frequent and with different characteristics. In particular, our case study partly confirms that the occurrence of such clones is a matter of granularity regarding the preprocessor annotations used to express variability. Beyond that, we observed that these clones mainly occur within one feature.

Code Clone Removal in FOP

Refactoring software product lines is different, because we have to ensure the unchanged behavior of *all* variants instead of only one single program. To this end, we propose the notion of *variant-preserving refactoring*, which takes the dimension of features explicitly into account and propose exemplary refactorings in a catalogue-like manner. Furthermore, we suggest which information can and must be used to identify candidates for code clone removal amongst all detected clones. To demonstrate the applicability, we present a small case study, where we apply the proposed refactorings to remove code clones in feature-oriented software product lines.

Research Questions

Since research on code clones is a very broad area with many different facets, there is a non-negligible risk to get lost in space. To stay focused, we guide this thesis, especially the sections encompassing our empirical studies, by four research questions. These questions are of special importance to guide forthcoming research on clones in SPLs in future and thus serve as a common basis for furture research activities. We present these research questions in the following.

RQ 1: *Do code clones exist in software product lines?*
Although there is a large body of knowledge on code clones in software systems, no work exists that investigates clones from a product line perspective. Hence, with this first question we put emphasis on whether clones even exist in software product lines. In particular, we are interested whether new mechanisms and concepts such as *refinements* in FOP may overcome certain limitations and thus render code clone occurrences to be meaningless. In case of the C preprocessor, it is of special interest for us to what extent code clones exist within preprocessor annotations, which is a different view compared to former studies on code clones in C systems (e.g., [MLM96]).

RQ 2: *Can we observe certain patterns of cloning that are specific to software product lines?*
Previous research on ode clones investigated certain patterns for both, relevant as well as incidental code clones. For instance, an example for the latter are getter and setter methods or blocks of initializing variables, which are common in software systems but do not represent interesting or harmful clones. In contrast, a pattern for relevant code clones could be similar code fragments in sibling classes that have a common superclass. In software product lines, we have an additional dimension, that is, *features*. Taking this dimension into account, we aim at investigating whether there are certain, recurring patterns how clones disseminate between features. In particular, we are interested in the relation between features that share code clones. This gives us a first idea where clones occur, whether certain feature groups (i.e., features that are related to each other) are more prone to clones and maybe whether we can abstract these clones away. Additionally, we aim at investigating on which granularity clones mainly exist, that is, do they occur on arbitrary level or rather on block level such as loops, methods, or even whole classes or features.

RQ 3: *Is it possible to judge on the harmfulness of code clones in software product lines? And if so, how?*
The harmfulness of clones is a topic that gained momentum in the recent past and that is controversially discussed amongst researchers. Furthermore, it is a very important and central issue in code clone research, because it has a direct effect how to manage detected code clones. Numerous studies exist that argue in favor and against the harmfulness of clones and which take different information such as evolution or the occurrence of bugs into account. While we take only limited information into account for answering this question, we primarily want to know at least whether detected clones in software product lines are avoidable and thus a result of bad design or missing abstraction.

Although we know that this will not answer the research question in its entirety, we argue that this could be a first step on reason about detected clones in software product lines.

RQ 4: *Is it possible to remove code clones from a product line point of view?* Refactoring in software product lines is a non-trivial task due to the presence of variability. Furthermore, it is open whether and how code clones occur in software product lines. So, what can we say about their removal? With this question we aim at analyzing under which circumstances code clones can be removed (which is different to the question whether they *should* be removed). Moreover, we want to figure out whether there are criteria that allow for code clone removal, independent of the implementation approach.

1.3 Outline

In the following, we present the overall structure of this thesis

In Chapter 2, we provide the background on code clone research, encompassing detection, analysis, and management of clones. Hence, the reader is familiar with the main concepts and terminology of code clone research, which is important for further chapters of this thesis.

Beside code clones, *software product lines* are an important aspect of this thesis. We introduce the main concepts such as Software Product Line Engineering (SPLE), variability modeling or implementation approaches for SPLs in Chapter 3. For the latter, we introduce FOP and the C preprocessor in more detail.

After laying the foundations for this thesis, we present theoretical thoughts on clones in SPLs in Chapter 4. To this end, we address four criteria, that are important in the context of software product lines and provide reasoning on code clones (and their occurrence) with respect to these criteria. Specifically, we aim at discussing how these criteria influence code clones and how this may differ between compositional and annotative approaches.

The next two chapters encompass our case studies on code clones in the SPLs. In Chapter 5, we present a case study on clones in feature-oriented SPLs. Particularly, we point out to what extent clones occur in such SPLs, whether they are specific to FOP, and how such clones are related to features. Additionally, we focus on differences regarding the development process, that is, whether the analyzed SPLs are developed from scratch or refactored from legacy applications. In Chapter 6, we analyze code clones in CPP-based product lines, especially regarding their occurrences within preprocessor annotations. Within this analysis, we investigate whether the discipline of annotations effects the amount of code clones.

Chapter 7 complements the two previous chapters by providing insights on code clone removal in software product lines using refactoring. Initially, we discuss refactoring in software product lines in general and why and how it is different from refactoring single software systems. Subsequently, we propose exemplary refactorings for feature-oriented software product lines in the fashion of Fowler et al. [Fow00]. Next, we present a case study, where we apply these refactorings to remove code clones in an exemplary product line. Finally, we discuss our results and the generalizability of our approach for CPP-based SPLs.

In Chapter 8, we summarize this thesis and point out our contributions along with the research questions, we posed in Section 1.2. In Chapter 9, we list ideas and suggestions for future work, based on the results of this thesis.

Parts of this thesis are based on previous work, published in [SAK10, SJF11, STKS12].

2. Software Clones – Detection, Analysis, and Management

Software clones, that is, the replication of code fragments also known as *code clones*, have been subject of intensive research since over two decades. Originally, the aim was to detect plagiarism in student projects [Gri81, Jan88]. Since then, a lot of research has been done to investigate *how* and *why* code clones occur. However, there are still open issues in code clone research. First and foremost, there is not even yet a clear definition of *what* a clone actually is. The most common definition is given by Ira Baxter, who defined code clones as follows [Kos07]:

> *Clones are segments of code that are similar according to some definition of similarity.*

To get an idea of what is a clone, we introduce different types of clones, as detected by current tools, in Section 2.1. In this context, *clone detection* is the process of finding code fragments that are similar to each other. Within this thesis, we mainly focus on *syntactical* similarity. We give an overview of existing clone detection techniques and their respective notion of similarity in Section 2.2. Since the sheer detection of clones provides only little information, they are usually analyzed further. This process is called *clone analysis* and aims at a deeper understanding of clones. Amongst others, questions such as how clones are used, how they evolve, or how they can be removed are of interest during the analysis phase. We describe the current state in clone analysis together with clone detection in Section 2.2.

Finally, the treatment of clones, called *clone management*, is an important aspect in code clone research. Basically, we distinguish between two approaches for clone management: First, *code clone removal*, which aims at removing the clones, usually by means of refactoring. Second, *code clone controlling*, which leaves the clones in the system but provides means to developers and managers to keep track of them. To decide which approach is appropriate for particular clones, the *harmfullness* of clones plays a pivotal role. However, this topic is discussed controversially amongst researchers and not yet solved. In Section 2.3, we describe the current state of clone management.

2.1 Types of Clones

As already mentioned, the definition of code clones is somewhat vague regarding the similarity between two or more code fragments. As a result, a categorization, which is widely accepted, has been proposed in the literature [Kos07, RC07] that distinguishes between different types of clones, according to their similarity. In the following, we explain these different clone types by means of a taxonomy, which summarizes similar approaches of other researchers [DBF+95, MLM96, Kon97, BMD+99a, BKA+07].

```
 1 class ... {
 2
 3     public void search() {
 4         /*...*/
 5         for(int j=0;j<(urep.members).size();j++) {
 6             vaux=(Vertex) (urep.members).get(j);
 7             vaux.representative=vrep;
 8             (vrep.members).add(vaux);
 9         }
10     }
11     /* more source code...*/
12 }
```

```
 1 class ... {
 2
 3     public void search() {
 4
 5         for(int j=0;j<(urep.members).size();j++) {
 6             vaux=(Vertex) (urep.members).get(j);
 7             vaux.representative=vrep;
 8             (vrep.members).add(vaux);
 9         }
10     }
11     /* more source code...*/
12 }
```

Figure 2.1: Example of *Type-I* clones, taken from the Graph Product Line (GPL) [LHB01]

2.1.1 Type-I Clones

Code fragments that are (almost) identical are called *Type-I* clones. Only minor differences regarding formatting such as comments or whitespaces are allowed. As a consequence, such clones can be detected by simple text processing tools such as the Unix DIFF tool or even String comparison. For instance, in Section 2.1 we show two code fragments, which are identical except for a missing comment in Line 4 of the second fragment. Using the DIFF tool for detecting clones results in the following output:

```
 1 protected void createTank(){
 2     Super().createTank();
 3     int x, y;
 4     x = GAME_WIDTH * 2 / 3 / 3;
 5     y = (int) (2.5 * x);
 6     menu.add(Sprach.TANKB,
 7         loadImage("choice22.png",x,y),
 8         loadImage("choice02.png",x,y),
 9         2);
10 }
```

```
 1 protected void createTank()
 2 {
 3     Super().createTank();
 4     int x, y;
 5     x = GAME_WIDTH * 2 / 3 / 3;
 6     y = (int) (2.5 * x);
 7     menu.add(Sprach.TANKA,
 8         loadImage("choice11.png",x,y),
 9         loadImage("choice01.png",x,y),
10         0);
11 }
```

Figure 2.2: Example of *Type-II* clones, taken from the TankWar SPL

```
4c4
<
--
> /*...*/
```

The result indicates that there was only a change in Line 4 (`4c4`), which is an comment that has been added (or removed, respectively).

2.1.2 Type-II Clones

While Type-I clones are easy to detect with simple tools, they are not very common. Instead, a common pattern of cloning is *Copy&Paste-and-Modification*, which leads to *Type-II* clones. These clones diverge more than Type-I clones so that even differences in names of identifiers, literals, types, layout, or comments are included in this type of clones. In Figure 2.2, we show two code fragments that are *Type-II* clones due to different modifications. First, there are differences regarding the formatting of the code, because the code fragment on the right-hand side has the opening bracket of method `createTank` on a new line. This may be due to programmer preferences, but also due to programming guidelines. Second, both code fragments differ in a constant (Line 6) and two literals (Line 7 and 8). This is a typical example for reusing code by *Copy&Paste-and-Modification*, for example, because similar or even the same functionality is needed in a different context. (e.g., a different class). For detecting *Type-II* clones, parameterized string matching algorithms can be used, which is why these clones are also called *parameterized* or *p-match* clones [Bak92, KKI02].

2.1.3 Type-III Clones

Type-III clones go even one step further than *Type-II* clones in the way that they additionally allow changing, adding, or deleting statements. Since deleting a statement from one code fragment can be also interpreted as adding to the corresponding (cloned) statement[1], we treat both terms (*deleting* and *adding* statements) synonymously. In Figure 2.3, we show a *Type-III* clone with both, a deleted as well as a

[1]Because the information, which code fragment is a copy of the other, is usually not available.

changed statement. In detail, the top code fragment in Figure 2.3 contains a method call in Line 7, which is deleted in the bottom code fragment. Furthermore, the top code fragment contains a method call of the variable `textField` in Line 8, whereas the bottom code fragment contains a value assignment of variable `textField` (Line 7). Since deleting statements results into gaps (when comparing two similar code fragments), *Type-III* clones are also referred to as *gapped clones*, where the missing statements are called *gaps* [UKKI02b, Kos07, RC07].

```
 1 class PC {
 2     /*...*/
 3     if (option.equals(Sprach.Name)) {
 4         this.setStatus(GameManager.TANK_SELECTED);
 5         this.gameManager.setStatus(GameManager.TANK_SELECTED);
 6         this.name = textField.getText();
 7         this.requestFocus();
 8         textField.setVisible(false);
 9         menu = null;
10     }
11 }
```

```
 1 class Handy {
 2     /*...*/
 3     if (option.equals(Sprach.Name)) {
 4         this.setStatus(GameManager.TANK_SELECTED);
 5         this.gameManager.setStatus(GameManager.TANK_SELECTED);
 6         this.name = textField.getText();
 7         textField=null;
 8         menu = null;
 9     }
10 }
```

Figure 2.3: Example of *Type-III* clones, taken from the TankWar SPL

In contrast to the previously presented clone types, for *Type-III* clones no clear definition of similarity exists. The reason is that there is no precise borderline to what extent two code fragments are allowed to diverge (or overlap, from an inverse point of view) and still can be considered as clones. In practice, users of clone detection tools can usually specify a *similarity threshold* to determine this borderline [Bak95, LLMZ06, RC08, JDHW09]. However, this threshold has a huge impact on the detection result (especially regarding meaningless clones) and has to be chosen carefully.

2.1.4 Type-IV Clones

We introduce this category just for completeness, though this type of clones does not fall into the category of syntactical clones, which we focus on. Indeed, *Type-IV* clones can be syntactically different: The cloning relation for his clone type is based on the *semantic similarity* between two or more code fragments and thus they are also called *semantic clones*. While some approaches assume that *Type-IV* clones still exhibit a certain syntactical similarity [GJS08], we rely on the notion of Juergens et al., where two (semantic) clones must be *behaviorally equivalent* [JDH10]. Hence, for the same

```
int a, b, c;

c = a * b;
```

```
int x, y, z;

int temp = x;
z=0;
while (temp > 0) {
    z = z + y;
    temp = temp - 1;
}
while (temp < 0) {
    z = z - y;
    temp = temp + 1;
}
```

Figure 2.4: Example of *Type-IV* clones, taken from [JDH10]

initial values (i.e., input), two code fragments have to compute the same result (i.e., output) to be *Type-IV* clones.

In Figure 2.4, we show two code fragments that are behaviorally but not syntactically equal and thus *Type-IV* clones by our definition. There is an ongoing debate on whether it is useful to detect semantic clones. On the one hand, semantic clones could be the result of intentional obfuscation of pieces of code, which have been copied. By obfuscating the copied code, the clone producer (e.g., a developer) can hide the fact of cloning, for example, for plagiarism or licensing issues. On the other hand, it is discussed controversially whether semantic clones have an effect on software maintenance.

The previously defined clone types do not exist in isolation. Rather, they are interrelated, based on their (sometimes vague) definition. For instance, a *Type-I* clone is always a *Type-II* clone, but not vice versa. In Figure 2.5 we show the relation amongst all four clone types using a Venn diagram.

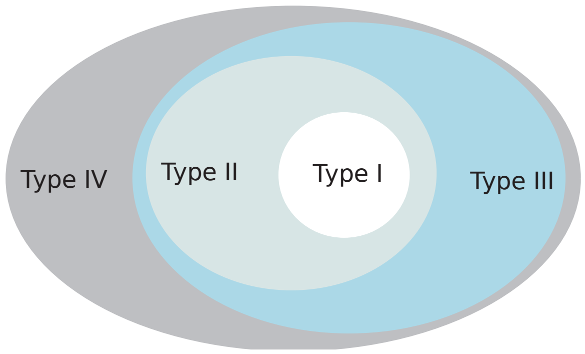


Figure 2.5: A Venn diagram, illustrating the relation between the different clone types

2.1.5 Beyond Code Clones

Recently, clone researcher put their focus on other artifacts that are different from source code. Nevertheless, all of these *non-code artifacts* are related to source code or to the overall software development process. Although this topic is beyond the scope of this thesis, we give a short overview about existing work within this field of research. For instance, Juergens et al. analyzed cloning in requirements specifications [JDF+10]. Since these specifications are often a starting point for the implementation of a software system, code clones in such specifications could lead to code clones in the software systems later on. Another use case are models used for software development. For instance, in model driven development (*MDD*), these models are used to generate the final source code. Hence, it is more useful to detect the clones on the model itself instead of the generated code. Examples for this use case are clones in formal models or in graph-based models such as Matlab/Simulink [DHJ+08, PNN+09]. Finally, some approaches exist that address code clones in UML sequence diagrams and domain models [LMZS06, Sto10].

2.2 Detection and Analysis of Clones

Clone detection and analysis, though two distinct steps, are tightly coupled regarding clone processing in its entirety. First, clone detection is obviously the most fundamental step, because it produces a vast amount of data, indicating code clone occurrences. The user (i.e., a developer) has only few possibilities for configuration of this step, mainly targeting clone length, clone type, or similarity threshold. However, for assessing and managing clones, the information of the clone detection step is too coarse-grained. Hence, a more specific and detailed view on the data, produced by clone detection, is necessary, which is the main purpose of code clone analysis. Amongst others, filtering out accidental or uninteresting clones, focussing on specific clone types or making statistical analyses regarding clone granularity are possible points of interest during code clone analysis. Based on the result of the analysis, further steps, associated with *clone management*, can be performed such as code clone visualization or removal. We illustrate the whole process of clone detection, analysis, and management in Figure 2.6.

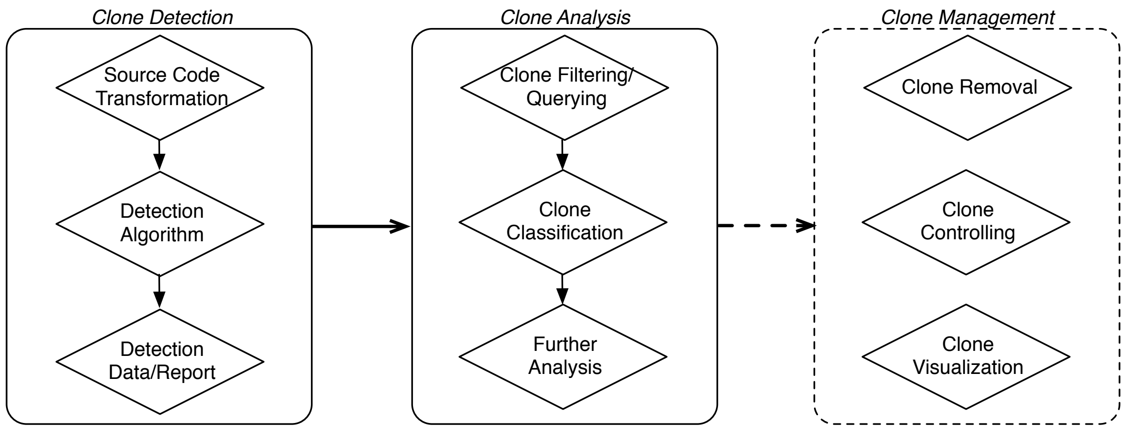


Figure 2.6: Overview of the whole clone detection process

In the following, we will give a detailed overview of the particular steps, performed during clone detection as well as clone analysis, indicated by the diamonds in Figure 2.6. In the same way, we elaborate on clone management steps in Section 2.3.

First, we provide information on how to build a clone relationship and how the accuracy of clone detection can be determined, because we rely on this information in the remaining parts of this section.

2.2.1 When is a Clone a Clone?

For each clone detection tool, it is inevitable to reliably build a clone relationship to decide whether code fragments are code clones or not. Determining such a relation between two code fragments is straightforward by using the definition of similarity, described in the previous section. However, often multiple copies of a code fragment exist and have to be detected as corresponding code clones. Such corresponding clones are often encompassed as *clone class* (and sometimes referred to as clone group or clone set). While each clone detection tool may have a slightly different definition for such a relationship, all of these relations are binary and require some relational properties to be fulfilled. In the remainder of this thesis, we rely on the clone relationship defined of Kamiya et al., which is defined as an *equivalence relation* [KKI02]. Hence, a clone relationship between two code fragments must fulfill the following properties:

- *Reflexivity:* Given a code fragment A, a clone relationship must exist to itself, denoted as $A \sim A$.
- *Symmetry:* Given two code fragments A and B, a clone relation must exist bidirectional, i.e., if A is a clone of B then B is a clone of A: $A \sim B \to B \sim A$.
- *Transitivity:* Given three code fragments A, B, and C. If a clone relation between A and B as well as between B and C exists, then a clone relation must exist between A and C as well: $A \sim B \wedge B \sim C \to A \sim C$.

As a result of this definition, a clone class is an equivalence class where a clone relation exists between any code fragments that belong to this class. Furthermore, the properties reflexivity and symmetry exist for each type of code clones (i.e., *Type-I* to *Type-IV*), while transitivity only holds for *Type-I* and *Type-II* clones. We want to illustrate this fact by a small example, which we show in Figure 2.7.

Within our example, the code fragments A and B form a *Type-III* clone pair, because two lines have been added to code fragment B (in Line 6 and 11, respectively). In the same way, the code fragments B and C form a clone pair of the same type. However, if we consider code fragments A and C, we observe that they differ in four lines (Lines 6, 9, 12, 15 have been added to C), which corresponds to a similarity of 75%. Depending on the specified similarity threshold, it is absolutely possible that these fragments are not detected as clones. Hence, the three code fragments do not form one clone class, but two different clone classes (A, B) and (B, C). This observation basically results into two implications: First, transitivity is not considered for *Type-III* and *Type-IV* clones, because this property can not be guaranteed by these clone types. Second, the code fragment B occurs in two clone classes. Hence, if we measure the amount of clones, we have to take this into account by considering this code fragment once only (if not done by the detection tool).

```
int x, y, z;

int temp = x;
z=0;
while (temp > 0) {
    z = z + y;
    temp = temp - 1;
}
while (temp < 0) {
    z = z - y;
    temp = temp + 1;
}
```

(a) code fragment *A*

```
int w, x, y, z;

int temp = x;
z=0, w=0;
while (temp > 0) {
    w++;
    z = z + y;
    temp = temp - 1;
}
while (temp < 0) {
    w--;
    z = z - y;
    temp = temp + 1;
}
```

(b) code fragment *B*

```
int w, x, y, z;

int temp = x;
z=0, w=0;
while (temp > 0) {
    w++;
    z = z + y;
    temp = temp - 1;
    printNumbers(w,z);
}
while (temp < 0) {
    w--;
    z = z - y;
    temp = temp + 1;
    printNumbers(w,z);
}
```

(c) code fragment *C*

Figure 2.7: Example for non-transitivity of *Type-III* clones

Precision/Recall

Besides the demand for a clone relation between code fragments, it is also important to determine the quality of the clone detection. That is, given a set of detected clones, how much of them are really clones and how much have been detected by mistake? To this end, *precision* and *recall*, originally established in information retrieval, are used as measurements to determine the *accuracy* of a clone detection tool. Both measurements are defined as follows:

Definition 1. *Given a set of candidate clones $CC_{detected}$, detected by an arbitrary clone detection tool. Furthermore, the set of code clones, which really exist, is denoted as CC_{exist}. Then,* precision *and* recall *are defined as:*

$$Precision = \frac{CC_{exist} \cap CC_{detected}}{CC_{detected}} \quad (2.1) \qquad Recall = \frac{CC_{exist} \cap CC_{detected}}{CC_{exist}} \quad (2.2)$$

Precision can be considered as a measure for the quality of the clone detection. By the given definition in Equation 2.1, it decreases the more *false positive* candidate clones (i.e., spuriously detected clones) are within $CC_{detected}$. In contrast, recall is a measure that refers to the completeness of the clone detection result. By definition (cf. Equation 2.2), the more candidate clones are detected that truly exist in a software system (i.e., *true positives*), the higher is the recall and vice versa.

Different studies exist that use these two measures for comparison and evaluation of clone detection techniques and tools [BKA+07, RC09, URSH11, ZR12]. However, such studies suffer always from the limitation that it is nearly impossible (and usually subjective) to decide whether a code clone has been detected correctly or not. For instance, Bellon et al. use a sampling approach across all clone detection techniques they investigate [BKA+07]. As a result, they obtain a reference data set about existing code clones, which are subsequently compared with the result of each tool under investigation [BKA$^+$07]. Similar approaches are suggested by Uddin [URSH11] and Zibran [ZR12]. Nevertheless, with the aforementioned approach, it is still possible that certain clones are missing or false positives are included in the reference data set. A possible solution to overcome this problem has been proposed by Roy et al., who creates and injects artificial code clones into existing source code using a mutation-based approach [RC09]. Alternatively, humans have to review and assess clones (as kind of an oracle) and agree on them via majority vote, which is tedious and very likely to be subjective as well.

Finally, it is worth to mention that there is usually a trade-off between precision and recall, meaning that if you increase the one measure it is likely that the other measure decreases. For instance, increasing precision usually means to allow less parameterization of clones and thus, only identical or almost identical (i.e., only minor differences) code clones are detected, which holds for *Type-I* and to some extent for *Type-II* clones. Hence, code clones with more profound differences are not detected and thus recall decreases. In contrast, if we allow a more flexible parameterization to increase the recall, we may detect code clones that are false negative and thus precision decreases.

2.2.2 Clone Detection Techniques

For more than two decades, clone detection is an active field of research. During this time, numerous techniques have been proposed to detect clones. While at the beginning the techniques were rather simple yet sufficient such as text-based comparison of code fragments, recent techniques are more sophisticated by exploiting different representations of source code to gather as much information as possible to detect clones. In the following, we give an overview of existing clone detection techniques and emphasize their main characteristics, categorized by the type of information they use to detect. For a more comprehensive overview and comparison, we refer to existing work on that topic [BKA$^+$07, RCK09].

Text-Based Clone Detection

Text-based clone detection techniques are probably the most simplistic ones, because they do not use any specific language mechanisms. In fact, a simple text-processing tool such as the Unix DIFF tool can be used to detect code clones in software systems. As a result, these techniques are *language-independent* and thus provide a flexible and lightweight approach to detect code clones across programming language boundaries. On the contrary, this technique is limited to detect only identical clones (*Type-I*) and clones with minor changes such as different formatting style or comments (*Type-II*, in parts).

Basically, this technique compares the source code under investigation line-by-line. Additionally, a normalization is performed on the source code before comparison. During this normalization step, formatting such as white spaces or line breaks but also comments are removed so that the code fragments have the same textual representation for the actual comparison. No further transformations are applied to the source code. For instance, the statement

```
if (he_says_yes == true && she_says_yes == true) married = 1;
```

is normalized as follows:

```
if(he_says_yes==true&&she_says_yes==true)married=1;
```

Although the aforementioned steps are similar for all existing text-based approaches, there are differences in the concrete realization and algorithms used for the actual detection of clones. For instance, Johnson, who did early research on text-based clone detection, proposed to use *fingerprints* on strings of the underlying source code [Joh93, Joh94a]. To this end, he specifies the number of lines that are considered as one entity (window range) and afterwards, a hash is computed over these lines. To cover the whole source code, a sliding window technique is used with the specified window range. Finally, the hash values are compared to find identical code fragments (i.e., substrings).

Another approach of Ducasse et al. uses dot plots (or scatterplots) for detecting areas of cloned code in a software system. [DRD99]. A dot plot is a two-dimensional visualization of source code with both axes (x and y) containing the source code entities under comparison. Here, lines are used as entities for comparison and in the case that two lines are similar, a dot is drawn at the intersection of bot lines (i.e., the (x, y)-coordinate). Finally, Ducasse uses a pattern matching algorithm to detect the actual clones.

A complementary approach is introduced by Marcus and Maletic, who use latent semantic indexing to find duplicated code [MM01]. In contrast to other text-based techniques, they do not normalize identifiers and comments but rather use these syntactical units to identify higher level clones such as *Abstract Data Types (ADT)*.

Finally, recent approaches even extend the original text-based techniques by more sophisticated transformations such as syntactic pretty printing [RC08, URSH11, ZR12]. The main extension of these approaches is that they use lightweight parsing mechanisms to implement pretty printing and other source code transformations. Hence, these approaches are in the tension between token-based and text-based techniques rather than purely text-based.

Token-Based Clone Detection

Compared to the text-based technique, in token-based clone detection the whole program (i.e., the source code) is transformed into a *token stream*. To this end, lexical analysis is applied to the original source code. Afterwards, the token stream is searched for similar (sub)sequences of tokens of maximum length. Then, by mapping such token sequences to their corresponding fragments in the source code, the actual code clones can be identified.

Similar to the text-based technique, the transformation is similar across all token-based approaches, whereas the algorithms used for searching the token sequence are different. Initially, Baker proposed an approach, complemented by the tool *Dup*, where

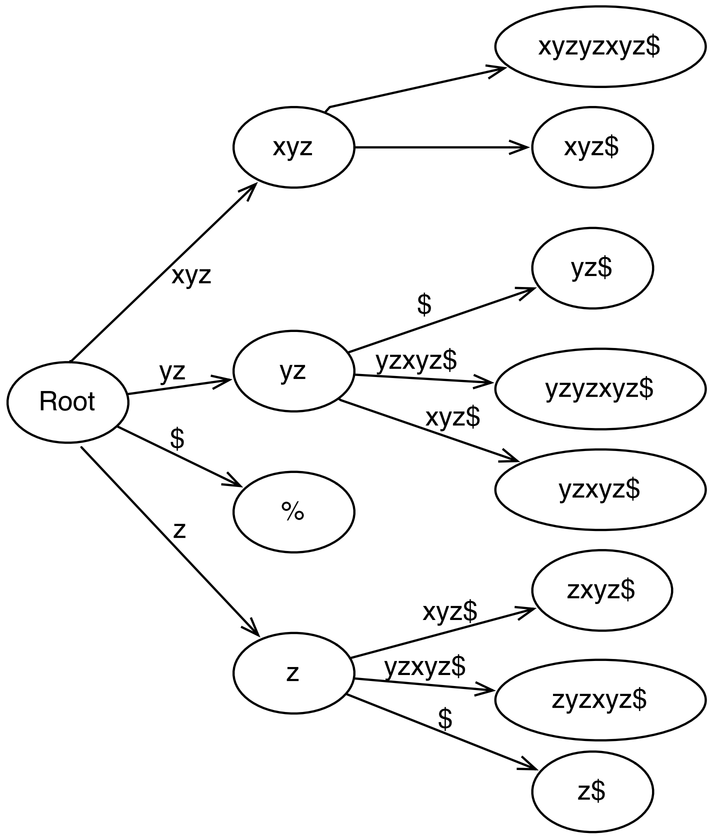


Figure 2.8: Suffix tree for the string `xyzyzxyz$`

the tokens are categorized in parameter tokens (e.g., identifiers) and non-parameter tokens [Bak92, Bak95, Bak96]. Then, for non-parameter tokens a hash function is computed for each line, whereas the parameter tokens are *generalized*. The latter means, that, for instance, all identifiers are encoded in the same way (within the token sequence) and thus differences between identifiers are omitted while searching for identical subsequences. For example a simple expression such as `x=y*z` is encoded as `P=P*P`, where the same placeholder (`P`) is used for all identifiers. Afterwards, a suffix tree, that is a tree structure where suffixes with a common prefix share the same set of edges, is built from the parameterized token sequence. Finally, clone detection is performed by searching for two (or more) suffixes in the tree that have a common prefix, which is obviously a clone. In Figure 2.8, we show an exemplary suffix tree for the string `xyzyzxyz$`.

Other approaches even extend Baker's technique by providing more source code normalization facilities or by using suffix arrays instead of trees to optimize memory consumption [BJ07, KKI02, KYU$^+$09]. For instance, Kamiya et al. present the prominent clone detection tool *CCFinder*, which can handle different normalizations that go beyond identifiers and literals [KKI02].

Finally, Li et al. use a data mining technique called *frequent (sub)sequence mining* to detect similar sequences within the token stream [LLMZ06].

Overall, token-based clone detection is more powerful than text-based in terms of clones that can be detected. Besides detection of *Type-I* and *Type-II* clones, it is possible to detect even some kind of *Type-III* clones. To this end, detected clones of the first two types have to be concatenated in case that the gap between these clones does not exceed a specified threshold such as a certain number of lines between both. However, the downside of this technique is that it is no longer language-independent, because lexical analysis and tokenization requires language-specific information.

Tree-Based Clone Detection

This kind of clone detection is mainly characterized by the fact, that a tree-based representation, which contains detailed syntactical information, is exploited for searching for similar code fragments. In the following, we distinguish between two kinds of source code representation: *Abstract Syntax Tree (AST)* and *Program Dependence Graph (PDG)*.

AST-based Clone Detection

For this kind of clone detection, the program under investigation is parsed and a syntax tree is created that contains all syntactical information of this program. Afterwards, different algorithms can be implemented to search for similar structures (i.e., subtrees) in the AST, which indicate the occurrence of code clones in the corresponding code fragments. In the AST, concrete names or values of identifies, variables, methods and more are abstracted away and thus even more code clones (specifically with similar structure) can be detected, compared to the techniques we introduced before.

Early work on this technique has been done by Baxter et al. [BYM$^+$98]. Within their approach, encompassed in the tool *CloneDr*, they have a compiler generator for producing the AST. Afterwards, subtrees of the AST are assigned to hash buckets (based on a hash function) and only subtrees in the same bucket are compared using a tree matching algorithm.

Yang et al. proposed the tool *cdiff* that uses a dynamic programming approach to find syntactical differences in subtrees [Yan91]. Furthermore, Wahler et al. use data mining techniques to find exact as well as parametrized clones [WSWF04]. In particular, they convert the AST to XML, enriched with all meta information about the program structure. Afterwards, they apply frequent itemset mining, a data mining technique to detect recurring patterns in a large amount of data (items). In this case, an itemset is a subtree of the AST and finding two or more similar itemsets indicates the occurrence of code clones in the corresponding source code.

An approach to detect clones on a higher level of abstraction is proposed by Evans et al. [EFM09]. His approach, called structural abstraction, allows even for variations of complete subtrees of the AST instead only considering variations of single tokens. Hence, more clones, including *Type-III* clones, can be detected with this approach.

Finally, different approaches exist that combine syntax trees and suffix trees. For instance, Koschke et al. propose an approach, where the subtrees of an AST are serialized [KFF06, FFK08]. Afterwards, a suffix tree of this serialized stream of AST nodes is created and used for clone detection. Furthermore, Jiang et al. proposed *anti-unification* to detect similar code fragments in Erlang programs [LT10].

PDG-based Clone Detection

A PDG is a graph-based program representation, where the nodes represent expressions and statements while the edges represent the control and data flow of the corresponding program [FOW87]. Hence, this representation provides a more abstract view on a certain program that an AST (e.g., lexical order of statement/expressions is not considered). For clone detection, algorithms are applied to a PDG in order to find isomorphic subgraphs, which indicate similar code fragments in the corresponding program.

For instance, Komondoor et al. propose to use program slicing to find isomorphic subgraphs within a PDG [KH01]. Another approach has been proposed by Krinke, who uses a matching path algorithm to find similar subgraphs of maximum length [Kri01]. Further approaches for PDG-based clone detection have been proposed by Gabel [GJS08] and Higo [HK09, HYNK11].

Metric-Based Clone Detection

The idea behind metric-based clone detection is to compute and compare certain software metrics such as cyclomatic complexity to find similar code fragments. The metrics can be computed on the actual source code but also on other representations of the underlying program such as the AST or PDG. To compare metrics of different code fragments, the corresponding metrics are structured into vectors and then compared using a specified distance measure such as Euclidean Distance.

Mayrand et al. proposed a metric-based technique to detect clones on function level [MLM96]. To this end, they collect different metrics such as from expressions or layout (e.g., comments, blank lines) but also of the control flow of functions. These metrics are computed for each function of the analyzed program and afterwards compared with each other. Based on the comparison, Mayrand et al. propose an 8-point scale for code clone assessment, ranging from *exact copy* to *distinct control flow*.

Furthermore, Kontogiannis proposed two approaches for metric-based clone detection [KDB$^+$95, KDM$^+$96]. The first approach directly compares metrics, computed for *begin – end* blocks. For each of these blocks, five widely accepted metrics (e.g., fanout, cyclomatic complexity) are computed and stored in 5-dimensional vectors. Afterwards these vectors are compared to identify similar code fragments. The second approach uses *Dynamic Programming* techniques to compare *begin – end* blocks. To this end, a *feature set* (e.g., *use* and *define* relations of variables) is created for each statement of a block. Then, the feature sets are compared to identify similar *begin – end* blocks.

A novel approach of Jiang et al., implemented within the tool DECKARD, compute characteristic vectors on ASTs [JMSG07]. Afterwards, they apply Locality Sensitive Hashing (LSH) to build cluster of similar characteristics vector. To this end, they use the Euclidean Distance as distance measure. Recently, Ngyuen at all presented the tool *JSync* for clone evolution and management [NNP$^+$11]. Within this tool, they use a metric-based technique that computes structural characteristics vectors on subtrees of the AST, which are compared to find similar code fragments.

Finally, some approaches exist that detect duplicated web pages or clones in web documents with metric-based techniques [DLDPF02, LM03].

2.2.3 Clone Analysis

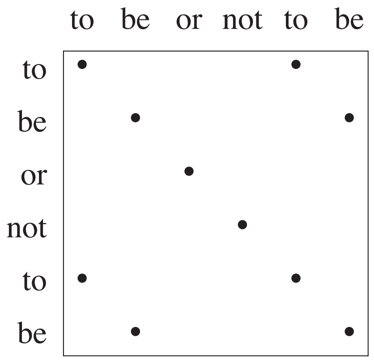
Clone detection is fundamental but only limited regarding the information it provides. In fact, the main contribution of clone detection, regarding the overall process (detection, analysis, management), is documenting or reporting the existence of clones. However, for making decisions whether clones are intentional or accidental, for finding common patterns, for characterizing and classifying clones or even to judge on their harmfulness, clone analysis is an important and inevitable step.

Many approaches for the analysis of clones exist, each with different purposes such as analyzing the evolution or harmfulness of detected clones, e.g., [GK11, SBS$^+$10, BKZ11]. Since some of these analyses are beyond the scope of this thesis, we only provide details about analyses that focus on reengineering opportunities (or necessities), because this is of special interest of this thesis. In particular, we present analysis approaches that use visualization techniques or propose a classification/categorization of clones.

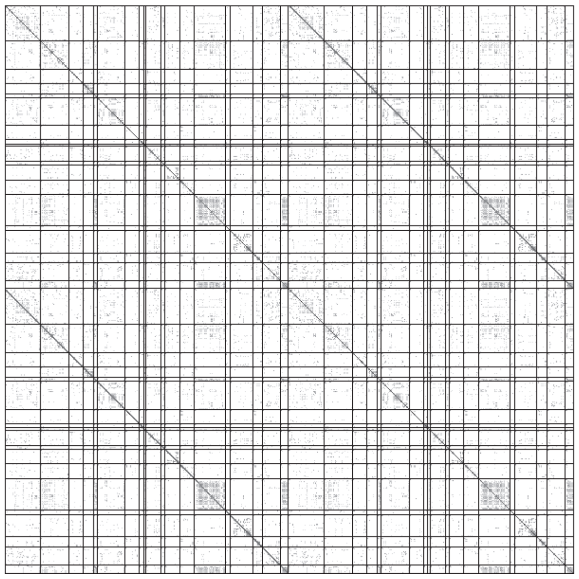
Code Clone Visualization

When a stakeholder (e.g., developer or project leader) has to manage the huge amount of data, produced by clone detection, she feels ”lost in space”, because handling large amount of textual data is impractical. As a solution, applying different visualization techniques to the clone detection results can aid stakeholders to get a more intuitive overview of the detected clones. As a result, the post-processing of clones such as making decisions about code clone removal are supported. However, different visualizations support different tasks such as quality assessment/improvement or compliance checking. In the following, we present visualization techniques commonly used for code clone analysis.

A common visualization technique for code clones are scatterplots [Bak95, DRD99, UKKI02a, RDL04, Cor11]. Basically, scatterplots are derived from *dotplots*, proposed by Church and Helfman to visualize self-similarity and design patterns [CH93, Hel96]. A scatterplot is a two-dimensional matrix where the axes contain the entities that are visualized such as tokens, files, or sub systems. If a similarity between two entities exists, a dot is made on the intersection of these entities. In Figure 2.9 a) we show a simplified example of a scatterplot for a prominent phrase of Shakespeare. The phrase is shown on both axes/dimensions in a word-by-word fashion. Afterwards, dots are made at each intersection of identical words. In particular, scatterplots provide a good overview of *hot spots*, that is, regions where the amount of clones is high, but also of other clone patterns. Cordy even provide zooming facilities to switch between the visualized entities [Cor11]. Hence, it is possible to obtain an overview of the whole



(a) Simple example for scatter-plot creation



(b) Scatterplot for large-scale system

Figure 2.9: Two examples of a dotplot/scatterplot taken from [Hel96]: a) for a famous phrase of Shakespeare and b) for a large system with million lines of code

system, but also to zoom into particular files that contain a high amount of clones. For instance, with such a zooming functionality, it is possible to detect higher-level clone patterns such as a clone relation between certain files or subsystems. However, this technique can only give an overview about existence of clones and partly, about their types, but not on reengineering opportunities. Furthermore, for large-scale systems this visualization may be inappropriate, because it is tedious or impractical to focus on certain regions of clones, which we illustrate with Figure 2.9 (b).

Johnson proposed to use Hasse Diagrams to visualize redundancy in source code files [Joh94b]. With such a diagram, the relation between code clones of one clone class and their corresponding files is visualized explicitly. Furthermore, information about file size is given implicitly by the size of the nodes. Additionally, he presented an approach for navigating through code clones using web pages [Joh96]. However, both approaches provide information on clone class level, while higher level information, such as about architectural or hierarchical characteristics, is missing.

Another prominent technique are *TreeMaps* [RDL04, JDH09, HG11], which can be used to encode different information about code clones and the related source code. In Figure 2.11, we show an exemplary TreeMap taken from the ConQAT tool. Each rectangle represents a file of the analyzed program together with information on its position, relatively to the whole system. Furthermore, the size of each rectangle can provide information on the size of the file [JDH09, HG11] or on detailed information about the code clones itself [RDL04]. Additionally, coloring the rectangles can be used to indicate whether a file contains many clones or not, which enables an easy detection

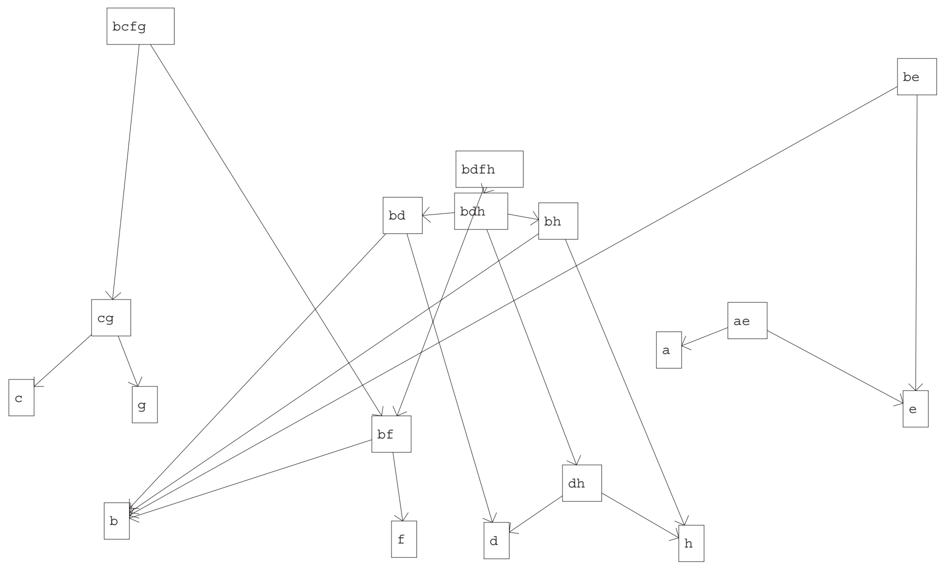


Figure 2.10: Exemplary Hasse Diagram taken from [Joh94b]

of hot spots [JDH09, HG11]. Beyond that, in the tool *Cyclone* it is possible to switch between different levels of granularity such as subsystems or files [HG11]. Nevertheless, this technique is limited in that it provides a good overview about code clones in systems, but not about the relationship between clone classes or cloned files.

Finally, a *SeeSoft view* is a visualization techniques that reveals the relation between clones and source files. In Figure 2.12, we show an example of such a view from the ConQAT clone inspection view. Within this view, each file is represented as rectangle and each clone as a bar within this rectangle, indicating its size and position [JDH09]. Additionally, code clones that belong to the same clone set have the same color. As a result, the stakeholder receives an overview of clones and how they are scattered throughout the system.

Reengineering-Based Code Clone Analysis

While clone visualization aids stakeholders to get an overview of the actual clone situation in their systems and derive follow-up tasks such as clone removal, it can not provide detailed information how to fulfill these tasks. Hence, a more detailed analysis is necessary to investigate the detected clones and to expose useful information to manage them. In the following, we give an overview of existing approaches for clone analysis with a specific focus on reengineering opportunities.

First, Mayrand [MLM96] and Balazinska [BMD$^+$99a] independently introduced approaches for the classification of function clones. Mayrand et al. propose to classify clones regarding their types of differences between the particular code fragments and to what extent they diverge. To this end, they take function names, layout, expressions and control flow into account to classify the function clones. This classification results into eight categories that are not necessarily focussed on reengineering but can be used

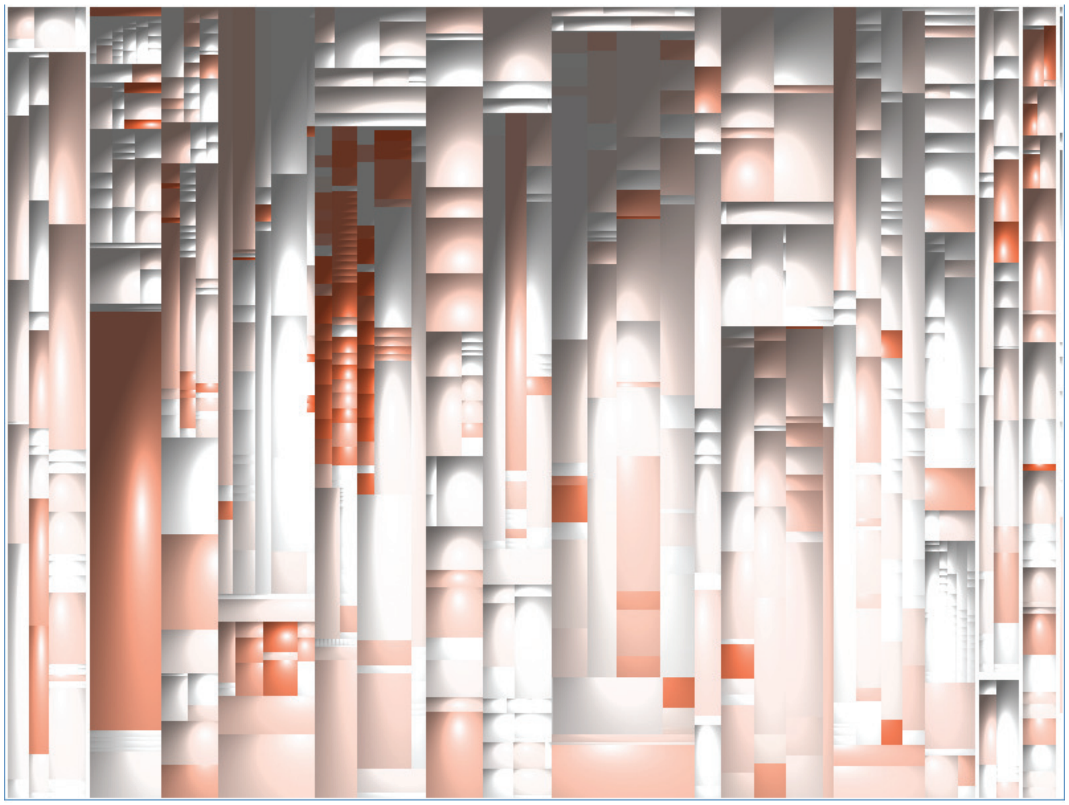


Figure 2.11: Example for a TreeMap as created by ConQAT

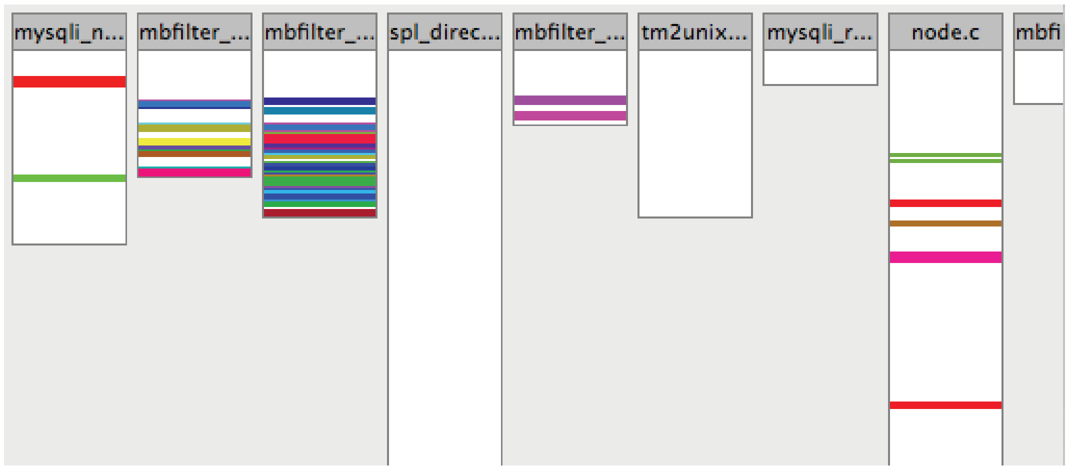


Figure 2.12: Example for a SeeSoft view, taken from ConQAT's clone inspection view

for this purpose to some extent. Balazinska provides a fine-grained code clone schema with an explicit focus on reengineering opportunities. The core idea is to classify function clones by the differences between corresponding method copies. First, she provides a schema of 17 categories, representing different reengineering opportunities such as source code transformation or source code restructuring. Additionally, these categories can be classified into four groups with respect to the observed differences: identical, differences in only one token, differences in a sequence of tokens, and differences regarding attributes of a method (e.g., modifiers, list of thrown exceptions). In subsequent work, she used this classification for redesigning programs and to build reengineering systems for code clone removal [BMD$^+$99b, BMD$^+$00]. However, both approaches are limited in that they analyze clones only on method/function level.

A more comprehensive taxonomy of clones is proposed by Kapser, who applies filtering as a special case of code clone classification [KG05]. Compared to the aforementioned approaches, this one goes beyond simple function clones. More precisely, Kapser also considers partial function clones and other syntactical blocks such as enumerations or macros for this classification. To this end, eight types of regions (in files) are defined and extracted for each source file. Afterwards, detected clone pairs are mapped to these regions. Although this approach aims at the comprehension of code clones and is focussed on reasoning about clones, the information provided by this analysis can be used for reengineering as well.

Basit et al. detects higher level clones to unify the resulting clone classes by generic design solutions such as Template Meta-Programming [BJ07]. To this end, they apply a data mining technique called *market basket analysis*. Basically, they consider each file to be a basket and assign code clones and clone classes to a corresponding file. By applying their analysis technique they can detect *clone patterns* such as files that share a huge amount of clones, indicating that these files are possible targets for unification.

Koni-N'Sapu proposed a classification based on the inheritance hierarchy [KN01]. To this end, they define different scenarios in which code clones can occur such as between sibling classes or between a class and its direct superclass. Then, they assign refactorings to each scenario, which are suitable to remove the detected clones. However, they do not classify the clones themselves such as type or level of granularity (e.g., function clone, conditional clone) for a cloned fragment.

Higo et al. propose a metric-based approach to analyze whether clones can be refactored or not [HKI08]. First, they extract clones, from the detection result, that form a certain syntactical block such as methods, loops, or try-catch blocks. Then, they compute different metrics for these extracted clones to determine whether these clones can be unified or not. The metrics reflect different information of the clones such as their position regarding the class hierarchy or the coupling between the code clone and its surrounding code (i.e., its context). Finally, they relate these metrics to possible refactoring patterns.

Finally, we also proposed an approach for clone classification to guide refactoring of clones, based on location and granularity [SKR08]. For the location, we defined a metric that takes the distance of the files, containing the clones of a clone class, into account. Additionally, we classified the code clones by their syntactical structure such as methods or loops. Based on these two dimensions we assess the possibility of removing these clones by application of certain refactorings such as *Extract Method* [Fow00].

Basically, the classification approaches above provide only information on whether reengineering opportunities for clones exist (and why). However, these approaches do not provide information that indicate whether code clones *should* be removed or not. This topic is covered in the next section.

2.3 Clone Management

As third and last part of a holistic clone detection process (cf. Figure 2.6), *clone management* is the phase, where the information of clone analysis is used to perform certain actions on detected clones (or not). While the term clone management is often used to describe the fact that clones are reactively managed without removing them, we use a slightly different terminology. For us, clone management encompasses *any* activity that takes place on clones after they have been analyzed. In particular, we distinguish between *code clone removal*, which encompasses reengineering activities, and *code clone controlling*, which encompasses conservative activities such as clone tracking. Before we explain these two approaches of clone management, we first discuss the harmfulness of clones, because this is essential for the decision *how* to treat detected clones.

2.3.1 Software Clones – Friend or Foe?

Ironically, code clones are both, blamed for the drawbacks they come along with but also referred to as useful technique for developing software. Hence, it is far away from trivial to judge on the harmfulness on clones. In the following, we give an overview of potential drawbacks associated with clones, mainly addressed by empirical studies. We also acknowledge work that argues in favor of clones as reliable engineering technique.

Maintainability

Software maintenance is a pivotal part of the software lifecycle since it causes up to 80% of the total costs for software development. Hence, keeping a software system maintainable is crucial. However, since the early days of code clone research, it is a common assumption that clones have a negative effect on different aspects of software maintainability. First, code clones can lead to bloated code, which hinders software maintenance due to increased code size [Bak95, Joh94a]. Additionally, the copied code is maybe not needed in its entirety and thus cloning contributes to ”dead” code. For instance, a cloned fragment contains different conditional branches, but not all of them are needed in the new context. Hence, this unused code may lead to less optimized code (regarding data or control flow) or even cause wrong behavior in case that a branch is executed by accident. Another drawback is the increased maintenance effort due to code cloning. If a code fragment is copied multiple times across the code base and changed later on, *all* corresponding clones probably have to be changed as well. Obviously, this is an increase in maintenance effort, because all clone have to be detected and checked whether the change has to be propagated or not. In case that several clones have to be changed synchronously, it is even possible that each change differs, for example, due to different naming schemes. Hence, also if the corresponding copies have been created by copy-and-paste, they can not be maintained in the same way, which increases the effort further.

Inconsistent Changes

As software evolves over time, code clones do so as well. A potential risk that arises from this evolution is that in case of maintenance activities (i.e., adding/modifying/removing code) not all corresponding clones are known or detected. As a result, the code is modified inconsistently, which may cause (semantic) errors such as unwanted or wrong behavior. However, most studies, which investigated the evolution and co-change of code clones, reveal that inconsistent changes only rarely occur. Initially, Kim et al. introduced *code clone genealogies*, which are clone classes with the evolution as an additional dimension [KSNM05]. Their study reveals that long-living clones that have been changed consistently are hard to refactor. Saha et al. extended this study (more systems, advanced clone detection) and observed that approximately 70% of the clone groups never change at all [SAZ$^+$10]. Furthermore, numerous studies state that clones, in case that they are not changed consistently, evolve independently rather than inconsistently [KG06, ACDP07, BSI$^+$09, TCADP10, LW10]. A common pattern that has been observed for such evolution is the *replicate-and-specialize* pattern of Kapser and Godfrey [KG06]. In contrast, Krinke [Kri07] partially rejected the aforementioned results by replicating and extending the study of Kim et al. [KSNM05]. In particular, he observed that only half of the clone genealogies changed consistently and that mostly no late propagation is performed for inconsistently changed clones.

Hence, in contrast to the common assumption, the evolution (and corresponding maintenance activities) of clones is mostly consistent and thus does not necessarily contribute to their harmfulness.

Clone Stability

Clone stability describes how often code clones change over time. The assumption is that if code clones change only occasionally during the evolution of a system, they are less harmful and not much effort has to be spent on their management. So far, different studies exist that aim at investigating *how* stable code clones are. Hotta et al. proposed a measure called *modification frequency* that is used to determine how frequently duplicated code is changed over time compared to non-duplicated code [HSHK10]. Basically, this measure counts the number of modifications of duplicated/non-duplicated code between consecutive revisions. Afterwards, the values can be compared to investigate which kind of code is more prone to changes. As a result of a case study, Hotta et al. conducted, they neither could observe that copied code is more prone to changes nor that this code requires more effort of maintain, as often argued in the literature. In a follow-up study, Göde and Harder [GH11] partially replicated and extended the study of Hotta et al [HSHK10]. They not only confirmed the findings of Krinke but also presented some reasons for the fact, that cloned code is more stable than non-cloned code. Essentially, the quite frequently observed deletion of clones (without the clear intent of code clone removal) is the main reason, which makes cloned code more stable. Recently, Mondal et al. performed a comprehensive study on clone stability [MRR$^+$12]. They state that clone stability depends on the type of clones and the used programming language. For instance, *Type-I/II* clones are rather unstable compared two *Type-III* clones. Furthermore, their study reveals that clones in C# are more stable than in Java or C systems. While the last mentioned study raises doubt on the stability of clones, most studies confirm that clones are not likely to change much over time and thus are not harmful from an evolutionary or maintainability point of view.

Error Propagation and Introduction

Another drawback, often referred to in connection with the harmfulness of clones, is the probability of error introduction or propagation due to copying code. While introducing an error may occur due to inconsistent changes, the propagation of errors takes place, if a code fragment, containing an error is replicated and thus, the error is spread over the system. During a study on clone stability, Bakota et al. investigated different version of Mozilla Firefox [BFG07]. As one result of their study, they found different evolution patterns and related them to several errors, indicating that code clones (and their evolution, respectively) have an effect on errors. Barbour et al. investigated the relation between late propagation and harmfulness of clones [BKZ11]. To this end, they defined eight different types of late propagation and compare them with other common evolution patterns. The analysis data reveal, that late propagation of clones (i.e., inconsistently changed clones are re-synchronized in a later version) is more fault-prone than consistent clone evolution and that certain types of late propagation are particularly harmful. In contrast, Göde and Koschke state that code clones change only rarely and only few of these clones undergo unintentional inconsistent changes [GK11]. They prove their hypothesis by an analysis of the frequency and risks of code clone changes in long-living and well-managed software systems.

In conclusion, we argue that although particular studies show evident for harmfulness of clones regarding errors, these studies do not allow for an generalization of this harmfulness. Rather, each system is specific on its own and thus the contained code clones are so as well.

Code Reliability

While the aforementioned aspects cover a rather negative point of view on code clones, there is also recent work that argues in favor of clones by means of *code reliability* and *trust.* In other words, by copy&paste code, the developer reuses functionality that has been tested and used before without any bugs. Hence, the assumption is, that this code is less error-prone and thus more reliable to use than implementing everything from scratch. Kapser and Godfrey revisited the common belief that code clones are harmful by default [KG06]. Furthermore, they provided substantial work on different patterns, which give insights *why* and *how* cloning takes place [KG08]. For instance, the *forking* pattern indicates that cloning is used as a starting point for developing functionality that is assumed to evolve independently in future. As a reason for that pattern, Kapser and Godfrey mention hardware or platform variation and provide examples such as the Linus SCSI driver subsystem [KG08]. Another pattern is *templating*, which supports the reuse of code if technical limitations hinder an appropriate abstraction. For instance, in COBOL this kind of cloning is an accepted and well-understood development practice,

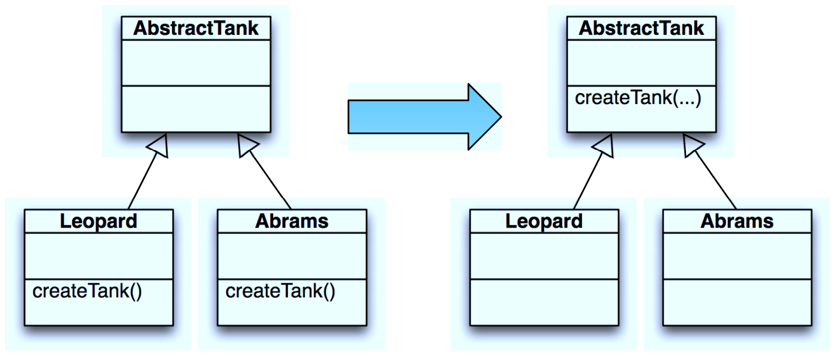


Figure 2.13: Class diagram for exemplary *Pull Up Method* refactoring

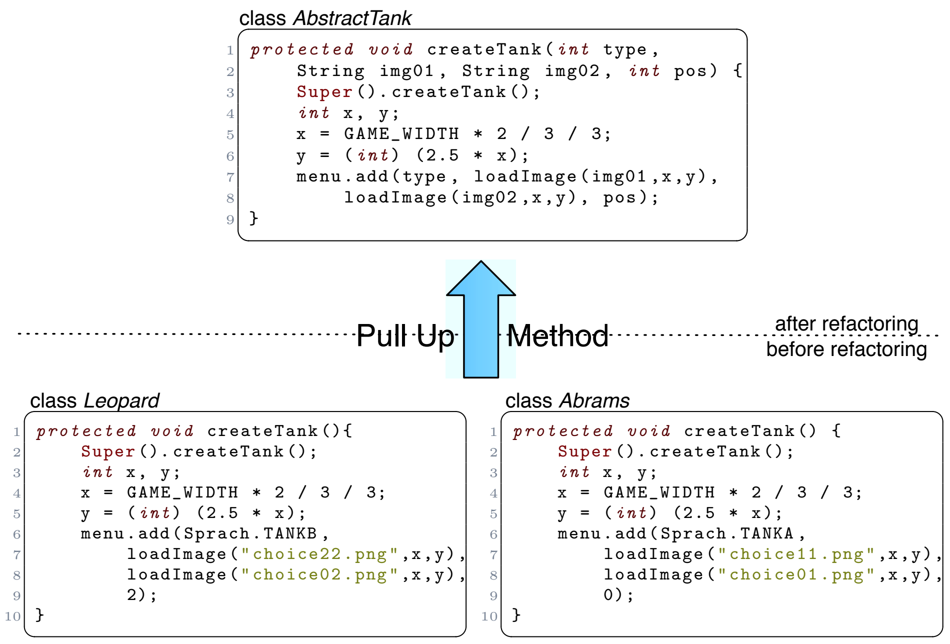
which is also confirmed by Cordy's work on practical barriers [Cor03]. Overall, there may be different situations where code cloning is the only solution for a certain problem and thus should be evaluated by keeping different concerns in mind.

2.3.2 Code Clone Removal

One possibility to deal with detected (and analyzed) clones is code clone removal, which can be seen as *corrective* clone management approach. The core idea is to remove as many clones as possible persistently by applying different techniques such as generics or program transformations. One approach that is widely referred to in relation with code clone removal is *program refactoring* [Opd92, Fow00]. Such program refactorings enable developers to change (ideally to improve) the structure of a system while its external, visible behavior remains untouched. Hence, for a given, predefined and stable input, a program produces the same output, before *and* after applying refactorings.

Generally, refactorings come into play if a decay in a system's design is observable, usually due to software evolution. To this end, several *code smells* have been proposed in the past, indicating an indispensable need for applying certain refactorings [Fow00]. Amongst those code smells, code clones have been voted to be *"number one of the stink parade"* [Fow00]. Although recent studies demand for a more diversified treatment of code clones as bad smells, refactoring is still a popular mean for removing clones. We want to illustrate that with an example, where the *Pull Up Method* refactoring is applied to remove. In Figure 2.13, we show a class diagram, indicating that the two classes `Leopard` and `Abrams` have a common superclass, `AbtractTank`, and a method `createTank`, respectively. Furthermore, in Figure 2.14, we show the code for the method `createTank` for both classes.

The idea behind the *Pull Up Method* refactoring is to move a method that is similar or identical in two or more subclasses to their common superclass. In our example, the method `createTank` in the two subclasses are *Type-II* clones and thus a potential target for the refactoring. As a result, we can remove these clones by pulling (and unifying) the method `createTank` to class `AbstractTank`, following the steps proposed by Fowler [Fow00]. Because the original methods differ in some constants, we have



```
protected void createTank(int type,
    String img01, String img02, int pos) {
    Super().createTank();
    int x, y;
    x = GAME_WIDTH * 2 / 3 / 3;
    y = (int) (2.5 * x);
    menu.add(type, loadImage(img01,x,y),
        loadImage(img02,x,y), pos);
}
```

```
protected void createTank(){
    Super().createTank();
    int x, y;
    x = GAME_WIDTH * 2 / 3 / 3;
    y = (int) (2.5 * x);
    menu.add(Sprach.TANKB,
        loadImage("choice22.png",x,y),
        loadImage("choice02.png",x,y),
        2);
}
```

```
protected void createTank() {
    Super().createTank();
    int x, y;
    x = GAME_WIDTH * 2 / 3 / 3;
    y = (int) (2.5 * x);
    menu.add(Sprach.TANKA,
        loadImage("choice11.png",x,y),
        loadImage("choice01.png",x,y),
        0);
}
```

Figure 2.14: Code example for *Pull Up Method* refactoring

to take this into account during refactoring. Consequently, these constants become parameters in the "new" method `createTank` in class `AbstractTank` (cf. Figure 2.14).

In the context of code clone research, early work on refactoring for clone removal has been done by Baker [Bak92] and Baxter [BYM+98], respectively. They proposed (different) approaches for macro extraction to remove clones in procedural languages. However, with the rise of the object-oriented paradigm, advanced approaches where needed and have been developed. For instance, Ducasse et al. proposed to use clone analysis to guide clone removal using refactorings, resulting into two variants of the well-known *Extract Method* refactoring [RDG99]. Substantial work on object-oriented refactoring for code clone removal has also been done by Balazinska [BMD+99b, BMD+00]. In particular, she proposed an in-depth analysis of clones to figure out similarities and differences between them and how this information can be used to automate the refactoring process [BMD+99b]. Specifically in the last decade, numerous approaches (and tools) have been proposed that focus on code clone removal by applying certain refactoring patterns, depending on the granularity, localization and type of detected clones. In the following, we introduce the most common refactoring patterns for code clone removal.

Pull Up Method (PUM): This refactoring, as already explained in our example, can be applied to replace similar methods in different subclasses by a more generalized method in the common superclass. Hence, it is especially appropriate for removing *Type-I/II* clones on method level [HKKI04b, YHK+05, SKR08, LBC+11, ZR11].

Extract Method (EM): With this refactoring, we can extract a certain piece of functionality (e.g., a for-loop) into a new method. Afterwards, the extracted code fragment is replaced by a call to the newly created method. Hence, if different similar code fragments on block level exist, this refactoring can be used to extracting them into one method and thus to remove all clones [HKKI04a, JH06, ZR11].

Move Method (MM): This refactoring is eligible, if a developer wants to move a method from one class to another, e.g., to increase cohesion. Afterwards, all calls to the relocated method have to be delegated to the class containing the method. In special cases, this refactoring is of interest for code clone removal. For instance, if two (or more) classes `A` and `B` contain a similar method, these two methods could be merged within one class (`A` or `B`) to eliminate redundancy [LBC$^+$11].

Form Template Method (FTM): In the presence of *Type-III* clones, clone removal becomes difficult due to the gaps. For instance, for applying PUM or MM refactoring to remove *Type-III* clones on method level, a developer probably has to handle different method calls or missing statements. Hence, unification becomes cumbersome or even impossible. As a solution, the *Form Template Method* refactoring can be applied to encapsulate these differences in methods with the same signature. As a result, the original methods become even more similar (*Type-I/II*) and it is possible to apply other refactorings such as PUM or MM to remove redundancy [SKR08, LBC$^+$11].

Extract Superclass (ES): If two or more classes share common functionality (e.g., methods, fields), this refactoring can be applied to create a superclass to merge this common functionality. In the context of clone removal, this is especially useful in the presence of conceptual clones on class level.

Extract Class (EC): Sometimes, two or more classes share a similar method, but are conceptually totally different so that the *Extract Superclass* refactoring is not applicable. Yet to remove the cloned methods, with this refactoring the developer creates a new class and moves the similar method to this class (only one of the similar methods). Afterwards each of the original methods is replaced by a delegation to the new class (and the respective method) [YHK$^+$05].

Although refactoring is a well-established technique for code clone removal, it is not always possible to apply refactoring for different reasons such as limitations of the host language. To this end, other approaches such as generative programming, design patterns or a change of the programming paradigm have been proposed in the past [BRJ05, JL06, BMD$^+$99b, MHQB05].

Finally, it is important to point out that code clone removal is a double-edged sword. This means, that there is a trade-off between whether code clone removal is possible and whether it makes sense. Especially the latter is influenced by different criteria. Besides the already mentioned harmfulness of clones, business, monetary or legacy issues may influence the decision in favor or against code clone removal. For instance, Cordy states that practical barriers may outweigh maintenance costs during the decision process of software maintenance activities [Cor03]. For example, in the financial industry the risk of introducing errors due to code clone removal outweighs the potential savings of having less redundancy. Consequently, code clone removal approaches are far from industrial adoption in certain domains. Furthermore, estimating the costs of code clone removal compared to the (maybe decreased) maintenance costs is not a trivial task. We conclude, that the decision on whether to remove clones or not can only be made for each system separately. Additionally, approaches for evaluating the usefulness of clone removal have to be developed, which take *all* criteria into account.

2.3.3 Code Clone Controlling

For one reason or another, code clone removal is not always possible. Nevertheless, it is important to raise awareness that code clones actually exist in a system. To this end, different approaches exist, which support developers but also managers to trace, manage, or assess existing clones. We denote these activities as *code clone controlling* and present an overview on different approaches in the following.

Toomim et al. proposed an approach called *linked editing* that aims at editing corresponding clones simultaneously to guarantee consistent changes, implemented as an extension of the XEmacs editor [TBG04]. Basically, the developer has to link code fragments manually. Then, an algorithm is applied in order to detect commonalities and differences between these code fragments. Hence, if later one of these code fragments is changed, the developer is asked whether to change the linked code fragment as well or not. Similar approaches for other Integrated Development Environments (IDEs) such as Eclipse exist as well(e.g., [DER08]).

While the linked editing approach has to be considered as reactive, it is more useful to pursuit a *proactive* approach by integrating code clone controlling facilities directly in the IDE. As a result, it would be possible to manage clones over their whole lifecycle. Furthermore, with such an integration, clone managing becomes part of the overall development process and thus raises the awareness of clones. Unfortunately, only few approaches exist that partly support this vision. Hou et al. sketch an initial design space for such an approach [HJJ09b]. Amongst others, they point out that a common clone model is inevitable to provide important functionalities such as clone capturing, visualization, and (simultaneous) editing. Furthermore, they report about the endeavor to initiate such an integrated clone management process based on their tool *CnP*, which aims at capturing copy&paste activities during the development process [HJJ09a].

Furthermore, feedback on clones by the developer is a viable approach, providing other developers with information such as *why* this code clone has been created and whether it should be considered harmful or not. One possibility to provide such feedback immediately are annotations. In modern IDEs, such as Eclipse, support exists for parsing such annotations and presenting them to the developer. Up to now, such annotations are only provided byJSync, where the developer can annotate code clones that should be ignored in future [NNP$^+$11].

Finally, certain approaches for code clone visualization (cf. Section 2.2.3) are useful for clone controlling. For instance, visualizations can aid programmers and managers to determine certain regions with a high clone ratio or subsystems that share a superior amount of clones.

3. Software Product Lines

Within this thesis, we investigate code clones in *SPLs*, with a particular focus on certain patterns, on the amount of clones and reengineering opportunities. Hence, it is crucial to know the fundamentals about SPLs. In this section, we introduce fundamental terms and concepts, which are important in the context of software product lines. Particularly, we introduce software product lines engineering, provide details on domain and application engineering, and introduce variability modeling. Furthermore, we outline two different implementation approaches for software product lines, because this is a recurring topic in subsequent sections of this thesis.

3.1 Software Product Line Engineering

Originally, software has been developed individually, meaning that one software system is implemented to solve a certain problem or satisfy the requirements of a particular stakeholder. However, in the recent past, the importance of software as well as the demand for customization increased, specifically due to the pervasiveness of software[1]. Consequently, developing each piece of software from scratch is not efficient anymore, specifically with respect to the development costs or time-to-market. As in other industries, a shift from developing a single software system to developing a whole *family of software systems (or products)* began, resulting into *software product lines* .

As defined by Clements and Northrop, a software product line is *"a set of software-intensive systems that share a common, managed set of features"* [CN01]. A *feature*, in this context, is an increment in functionality that is visible to the end-user. Furthermore, features can be used to distinguish between the different programs, also called *variants*, of a software product line [CE00]. Finally, the products of a software product line share commonalities (i.e., features that are common in all variants) but also exhibit variabilities (i.e., features that are only part of particular variants). Especially by

[1]With this term, we acknowledge the fact, that meanwhile, software is an integral part of our day-to-day life, because it is almost everywhere.

sharing common features, a software product line provides reuse opportunities and thus fulfills the demands for a efficient software development process for the mass market. To meet these new requirements, the development process has to be also adapted so that it efficiently supports the reuse of *assets*. Furthermore, this process has to tackle the problem of developing multiple systems in parallel, each tailored to the requirements of a specific stakeholder or problem. Hence, a dedicated development process for software product lines has been created, which is called *SPLE* [PBVDL05]. A key concept of SPLE is the distinction between *Domain Engineering (DE)* and *Application Engineering (AE)*, which we introduce in the following subsection.

3.2 Domain vs. Application Engineering

To get the most and best out of an software product line approach, it is inevitable to focus on a certain *domain*, for which the particular systems of an SPL should be created. Hence, the respective domain has to be analyzed and the common and variable assets have to be defined. The process, encompassing these steps, is called *domain engineering*.

As a first step, during *domain analysis*, the requirements for products of the respective domain are analyzed. To this end, requirements that currently exist but also those that may be important in the future are taken into account. Based on the requirements, commonalities and differences amongst products of the software product line are determined and described by means of *features* [KCH$^+$90].

Based on this domain analysis, the software product line is designed, which means that an architecture for the whole SPL is defined. A specific characteristic of such an architecture is that it must reflect all intended products as well as commonalities and differences amongst them. Hence, relations between particular features must be made explicit. As a result, the design of an software product line must be variable to some extent so that different products can have a different realization.

As a last step, the reusable assets of the software product line have to be implemented according to the previously defined design. In particular, this implementation must also reflect the distinction between the reusable assets (e.g., features). Different approaches for product lines implementation exist such as components, C preprocessor or FOP (cf. Section 3.4).

While domain engineering focusses on the analysis, design, and implementation of the whole domain, encompassed by an software product line, *application engineering* mainly concerns about product derivation. The first step consists of a requirements analysis as known from traditional software development processes. Additionally, it is checked whether the requirements align with existing features in the software product line, defined during domain analysis. If certain requirements are not covered by any feature, a new one is introduced and can be even integrated into the SPL to increase reusability of the software product line for future products.

If the features of the software product line meet the requirements, the *variant configuration* stage is entered. The features, meeting the requirements, are selected and mapped to the corresponding (reusable) assets, defined during domain engineering. Again, if certain assets have not been designed and implemented during the initial domain engineering process (i.e., because of missing or unknown requirements), this can be done during this step and also be added to the SPL. Finally, the selected assets are used to *generate* the final variant/product.

3.3 Variability Modeling

During the process of domain analysis (cf. Section 3.2), the *scope* of the software product line is determined in terms of features. Furthermore, commonalities and differences between the particular variants are identified. To specify the dependencies between features in a software product line and to model commonalities and differences between variants, a developer usually uses a *variability model*. Different approaches exist *how* to model the variability in SPLs such as *Feature Models (FMs)* [KCH$^+$90], grammars [CHE05], or propositional formulas [Bat05].

With feature models (and grammars), we distinguish between two types of features: *compound features* and *primitive features*. Compound features serve the purpose to group other (sub)features that are related to a common concept. For instance, a feature *Operating System* could be used to group the sub features *Mac OS, Linux,* and *Windows*. Furthermore, sub features can be compound features, primitive features, or a mix of both kind of features. In contrast to compound features, primitive features do not contain any sub features. Additionally, we can define different constraints on sub features, for both, compound and primitive features. In particular, a single feature can be *optional* or *mandatory*. Furthermore, feature groups (e.g., sub features of a compound feature) can be either in an *Or-group* or in an *Alternative-group* [CE00].

In the following, we explain *feature models* more detailed, because these models are of special interest for further chapters of this thesis and thus detailed knowledge is inevitable.

Feature Model

A feature model is a specific type of a variability model, used to specify valid combinations of features. Due to its hierarchical, tree-like structure, we refer to compound features as *parent features* and analogously, to primitive features was *child features*. In Figure 3.1, we show an exemplary feature model for a simple stack product line (Stack SPL).

The feature *DataStructure* is the only mandatory feature, which means that a stakeholder must select this feature for *each* variant. Furthermore, this feature is a compound feature that is used to group the two primitive features *Array* and *LinkedList*. These primitive features form an Alternative-group, that is, they are mutually exclusive and thus only one of these features can be selected for a certain variant. The semantics of this restriction is obvious; for a certain stack implementation (i.e., a variant of this SPL), either an array or a linked list can be used as underlying data structure, but not

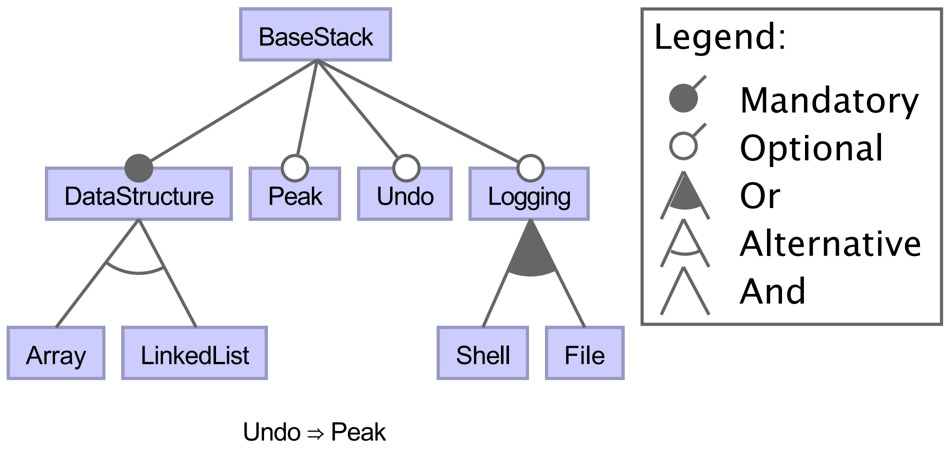


Figure 3.1: A feature model for a Stack product line (Stack SPL)

both at the same time. All other features are optional and thus may or may not be selected for a particular stack variant. Additionally, the features *Shell* and *File* form an Or-group, indicating that zero, one, or both features can be selected.

Beside these "simple" constraints, a feature model may contain *cross-tree constraints*, usually specified by arbitrary propositional formulas [Bat05]. Developer can use these constraints to specify more complex dependencies, especially between more than two features or features that do not exhibit a parent-child relationship. For instance, our exemplary feature model in Figure 3.1 contains an "implies" cross-tree constraint (*Undo* ⇒ *Peak*), indicating that a stakeholder can only select feature *Undo* if she selects feature *Peak*.

To generate a certain variant of the SPL, a stakeholder selects the desired features that should be contained in th final program (called *configuration*). Afterwards, the corresponding assets such as source code files are used to generate the program. How this generation takes place depend on the implementation approach that has been chosen for the software product line such as composition or preprocessor compilation (cf. Section 3.4).

As already stated, compound features are mostly used for the purpose of grouping sub-features. Consequently, they have rather an auxiliary role and do not represent a "feature", that is, they do not provide an increment in functionality. Nevertheless, no policy exist that prohibits compound features to add an increment in functionality to a software product line. To this end, Thüm et al. introduced the notion of *abstract features* [TKES11]. By definition, an abstract feature is a feature that does not contribute to any program variant in terms of functionality. In contrast, a compound feature that contains source code may add functionality to particular variants and thus is *not* an abstract feature. The necessity as well as advantages and disadvantages of this distinction is widely discussed, but out of the scope of this thesis. However, we rely on the notion of abstract features within this thesis to indicate that a feature does not contain any reusable assets such as source code files and serves for grouping purposes exclusively.

3.4 Implementation Approaches for SPLs

While variability modeling defines the scope of an SPL during the domain analysis phase, the main concern of the *domain implementation* phase is the actual development of the reusable assets, defined in the previous phase. To this end, numerous approaches exist that we distinguish into two categories: *compositional* and *annotative* implementation approaches [KAK08].

The core idea of compositional approaches is to separate and modularize the particular, reusable assets such as source code artifacts. The assets that are part of each variant of an software product line are considered as the common code base, which provides the core functionality of the SPL. In contrast, the variable assets are encapsulated in distinct modules (following the rules that are defined by the variability model) and each module provides an extension of the core functionality. By this modularization, compositional approaches are considered to improve certain quality aspects of software development such as program comprehension, reusability, or maintenance [Par79, Big98, AMGS05, AK09]. Furthermore, this modularization allows the creation of products by simply composing these modules, where the composition mechanism depends on the concrete language or tool used for compositional implementation of SPLs. Amongst others, frameworks [JF88], components [HC01], C++ templates [CE00], collaboration-based design [VN96], mixins [BC90, SB02], *AOP* [KLM$^+$97], *FOP* [Pre97, BSR04, AKL09], *Delta-Oriented Programming (DOP)* [SBB$^+$10, SD10], or traits [DNS$^+$06, BDS10] are concepts and languages to achieve the aforementioned modularization.

In contrast, annotative approaches do not focus on separation and modularization of reusable assets. Instead, source code that belongs to a certain feature is marked just-in-place by *annotations*. Additionally, specific configuration files such as *makefiles* in C/C++ can be used to express variability on file level. Examples for annotative approaches are the C preprocessor CPP [LAL$^+$10, LKA11] and CIDE, which uses colors to annotate code fragments that belong to a certain feature [KAK08].

Although both approaches are used with the same goal in mind, they represent two opposite sides of the same idea. As a result, there is an ongoing debate which approach to use for product line implementation. While annotative approaches, especially preprocessor-based ones, are mainly used in industry, compositional approaches gain momentum in academia. However, both approaches have advantages and disadvantages with respect to different criteria such as granularity, language independence, or separation of concern. While it is out of scope of this thesis to compare and assess both approaches in general, we reflect some of these criteria in the tension of code clones in Chapter 4. Moreover, we provide a closer look on feature-oriented programming and CPP-based software product lines, since we mainly focus on these approaches and their relation to code clones within this thesis.

3.4.1 Feature-Oriented Programming

FOP is a compositional approach, which gained momentum in the recent past. The fundamental idea of FOP is to decompose a program into its common and variable parts, called *features* [Pre97, BSR04]. To this end, all assets that belong to a certain feature are modularized into cohesive units called *feature modules*. As a result, FOP provides a clear separation of features and thus improves maintainability and traceability in terms of source code provenance (i.e., *which* code fragment belongs to which feature) [AMGS05, AK09].

While a feature is a visible increment in functionality within the domain model, a feature module is the counterpart on implementation level. Basically, a feature-oriented program consists of a base program, encapsulating the commonalities of an SPL, and each feature module extends this base program by adding some functionality, similar to what has been proposed for the development of program families [Par76]. Furthermore, FOP follows the fundamental concept of *Separation of Concerns (SoC)*, because each feature module is a cohesive unit and can be considered and implemented independent of other feature modules [TOHS99]. To generate a certain program, also called *variant*, of the SPL, the user first of all selects the respective features. Afterwards, the corresponding feature modules are composed in a step-wise manner [BSR04]. In Figure 3.2, we show an example of the abstract composition process for two variants of the Stack SPL. For the variant *Stack-arr*, the feature *Array* refines the base program (i.e., *BaseStack*) in a first step, resulting into a modified (intermediate) program. Then, the feature *Peak* refines the modified program in a second step, which results into the final program. In the same way, we can generate the variant *Stack-list* by stepwise composing the respective features.

Although the general process of composing feature modules is similar for all feature-oriented languages and tools, the concrete composition mechanisms is distinct for each approach. For instance, FEATUREHOUSE is a tool that uses superimposition to compose feature modules [AKL09]. In Figure 3.3, we show a code snippet from our Stack SPL implemented with FEATUREHOUSE. In feature *BaseStack*, the methods `push` and `pop` are initially declared. Furthermore, in feature *Peak*, the method `peak` is added. Finally, feature *Undo* adds the field `undoStore` and the method `undo`. Additionally, this feature refines the already defined method `pop`, which is indicated by the keyword `original`.

All the aforementioned fields and methods are implemented within the same class, though for different features. This reflects a main characteristic of feature-oriented programming: the decomposition of classes. The reason is that a feature is usually scattered across different classes. To realize this interaction between classes and features on implementation level and to decompose classes across features, FOP relies on the concepts of *collaboration-based design* and *mixins* [SB02, VN96, BC90]. With collaboration-based design, an object (or class) can have different *roles* in different collaborations. In the context of FOP, a *collaboration* is the implementation of a feature.

For better illustration, we show an exemplary *collaboration diagram* of our Stack SPL in Figure 3.4. The diagram consists of three different classes (`Stack, Node, Element`) and features (*DataStructure, Array, LinkedList*). Each class can play a role with respect to a certain feature. In FOP, a role is characterized as an increment in functionality such as extending or adding a method. For instance, class `Stack` has a role for each feature of

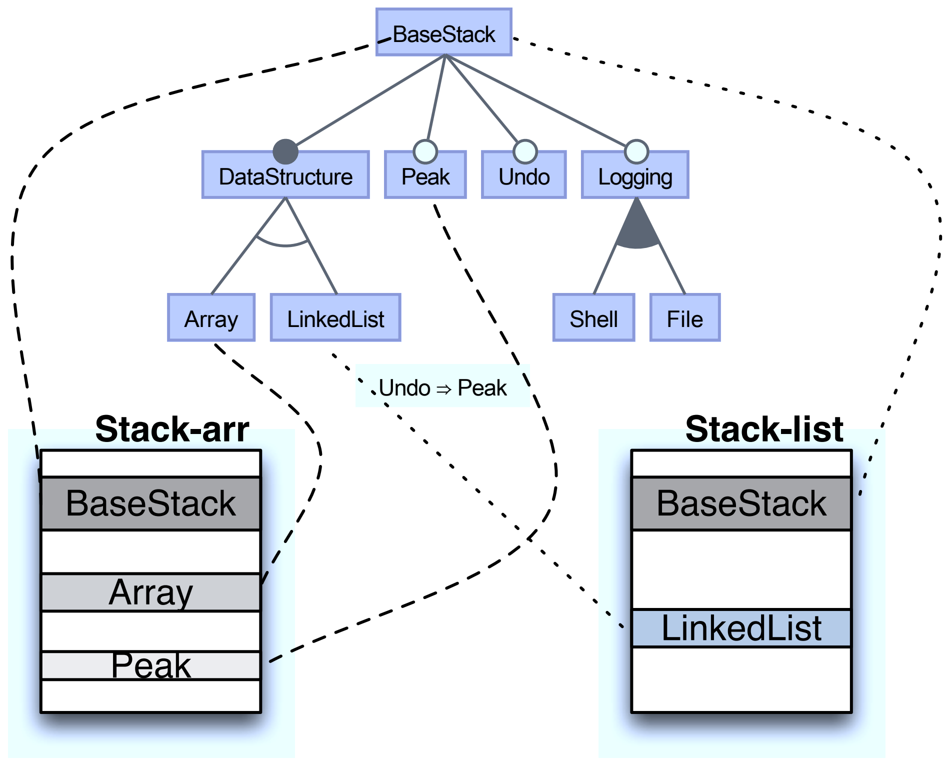


Figure 3.2: Two possible variants of the Stack SPL, generated by composing the feature modules *BaseStack,Array,Peak* and *BaseStack, LinkedList*, respectively.

Feature *BaseStack*

```
class Stack { ...
    void push(int v) {/*...*/}
    int pop() {/*...*/}
}
```

Feature *Peak*

```
class Stack {
    int peak() {/*...*/}
}
```

Feature *Undo*

```
class Stack { ...
    int undoStore;
    void undo() {/*...*/}
    int pop() {
        undoStore = peak();
        original();
    }
}
```

Figure 3.3: Feature-oriented implementation of the Stack product line features *Peak* and *Undo* using FeatureHouse.

the diagram (*DSStack, AStack, LLStack*). In contrast, class `Element` has only two roles (*AElem, LLElem*) because it does not participate in feature *DataStructure*. Likewise, in the exemplary implementation of our Stack SPL, class `Stack` has a role in feature *Peak*, where it implements a new method `peak()` and thus, extends the functionality (cf. Figure Figure 3.3).

3.4.2 The C Preprocessor

The C preprocessor CPP is probably the most common annotative approach to implement variability in software systems. Originally, the CPP tool has been designed to

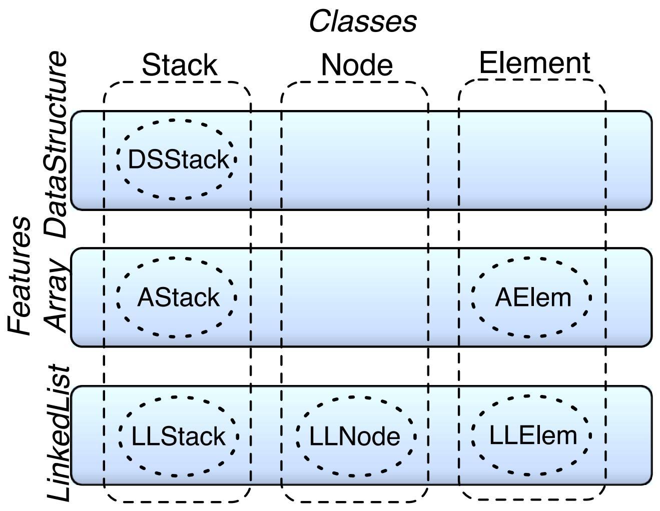
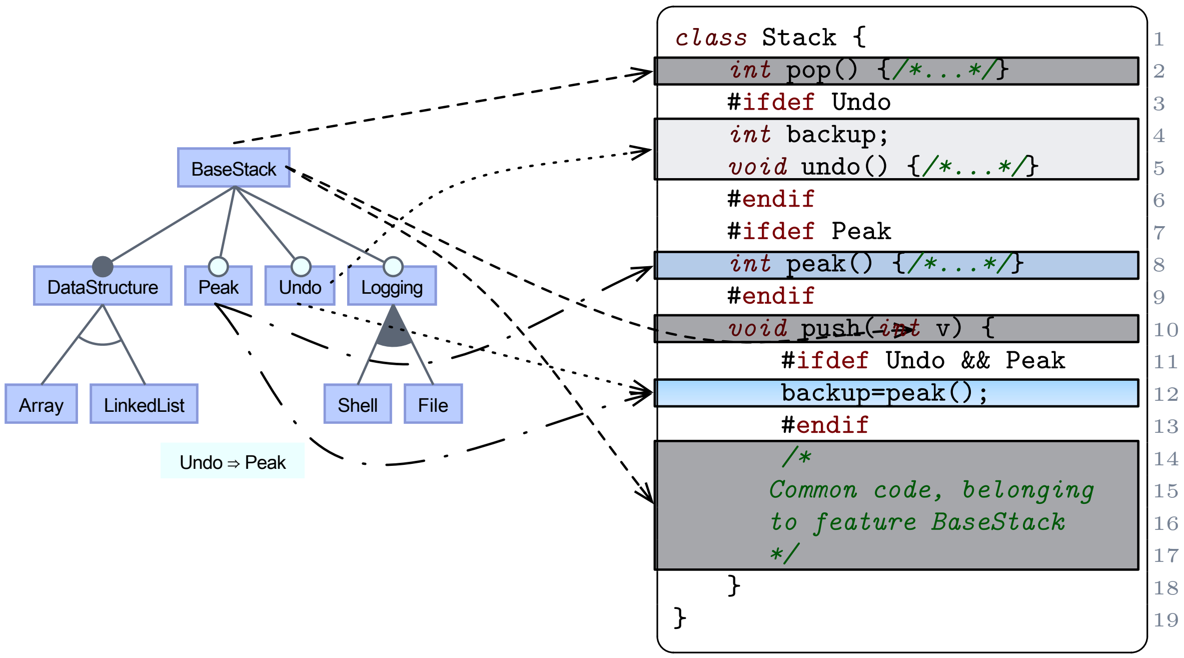


Figure 3.4: Collaboration diagram of the Stack SPL with three classes (dashed rectangles) and three features (horizontal rectangles).

support meta programming. It consists mainly of three different mechanisms: file inclusion (`#include`), macro definition (`#define`), and conditional compilation (`#ifdef`). In this thesis, we only focus on conditional compilation, because this is the mechanism actually used to express variability in programs. Generally, expressing variability using conditional compilation is very flexible, because it allows annotations on every level of granularity and provides a high expressiveness [KAK08]. Furthermore, the CPP is just a simple text processing tool and thus independent of the programming language which it is used with. Both, flexibility and expressiveness and the language-independence are mainly responsible for the predominant usage of the CPP tool for implementing variability, especially in industry. Another reason for its success is the fact that the CPP can easily be integrated into existing processes and implementations, even a posteriori. Nevertheless, researcher criticize the CPP for its negative effects on maintainability, readability, and code quality [Fav95, Fav97, EBN02], which is reflected by terms such as *"#ifdef considered harmful"* or *"#ifdef hell"* [SC92, LST$^+$06b].

In contrast to compositional approaches, the source code that belongs to a certain feature is *not* modularized into one cohesive unit. Instead, the corresponding code is annotated just-in-place using *#ifdef*. In Figure 3.5, we show this for three features of our Stack SPL. For each of the features, different code fragments are annotated, all contained in class `Stack`. For instance, the code for the feature *BaseStack* is scattered across the whole class. Furthermore, code can even belong to more than one feature. In our example, the statement in Line 12 is part of feature *Peak* and *Undo*, which is also reflected by the expression `Undo && Peak`.

To decide whether such an annotated code fragment belongs to a certain variant, each annotation contains a *boolean expression*, which is evaluated. In case that this expression is *true*, the corresponding code is part of the final program. This evaluation takes place in a preprocessing step before the actual compilation.



```
class Stack {
    int pop() {/*...*/}
    #ifdef Undo
    int backup;
    void undo() {/*...*/}
    #endif
    #ifdef Peak
    int peak() {/*...*/}
    #endif
    void push(int v) {
        #ifdef Undo && Peak
        backup=peak();
        #endif
        /*
       Common code, belonging
       to feature BaseStack
       */
    }
}
```

Figure 3.5: Exemplary Mapping between feature model and annotations using the C preprocessor for features *BaseStack,Peak,* and *Undo* of the Stack SPL

4. Reasoning About Code Clones in Software Product Lines

In Chapter 2, we discussed different reasons for the occurrence of code clones. Beside external reasons such as ad-hoc code reuse or time constraints, limitations of the programming paradigm itself may be a source of code clones. For instance, procedural programming languages may cause clones due to a lack of appropriate reuse mechanisms such as inheritance. Furthermore, in some languages such as COBOL, code replication is an accepted concept for templating. But even in object-oriented languages, existing mechanisms for abstraction such as inheritance or generics are not always sufficient for *expressing variability* in programs and thus contribute to code cloning. We illustrate how expressing variability may cause code clones by means of our Stack product line example in Figure 3.3.

Composing the three features of this example, we generally can create four different variants: *Stack*, *Peak* $\bullet$ *Stack*, *Undo* $\bullet$ *Stack*, and *Undo* $\bullet$ *Peak* $\bullet$ *Stack*, where $\bullet$ denotes feature composition. While we used feature-oriented programming for implementing the example in Figure 3.3, we can express the same variability with object-oriented programming as well. We show the respective listings in Figure 4.1, consisting of four separate classes, one for each variant.

For an object-oriented implementation, we use simple inheritance without any code clone activity to generate the first three variants (cf. Figure 4.1, a) – c)). However, implementing the fourth variant, `UndoPeakStack`, requires to inherit from two classes, `PeakStack` and `UndoStack`. Unfortunately, multiple class inheritance is not supported in many OOP languages such as Java. Moreover, even if it is supported, multiple inheritance often suffers from the *diamond problem*, which makes multiple inheritance complex and error-prone [Sin94, MA09]. Hence, code cloning occurs in order to reuse functionality without multiple inheritance in class `UndoPeakStack`.

```
class Stack { ...
    void push(int v) {/*...*/}
    int pop() {/*...*/}
}
```

a) *Stack*

```
class PeakStack extends Stack{
    int peak() {/*...*/}
}
```

b) *Peak • Stack*

```
class UndoStack extends Stack { ...
    int undoStore;
    void undo() {/*...*/}
    void pop(int v) {
        undoStore=v;
        /* ..... */
    }
}
```

c) *Undo • Stack*

```
class UndoPeakStack extends PeakStack { ...
    int undoStore;
    int peak() {/*...*/}
    void undo() {/*...*/}
    void pop(int v) {
        undoStore=v;
        /* ..... */
    }
}
```

d) *Undo • Peak Stack*

Figure 4.1: Object-oriented implementation of *Stack* with features *Peak* and *Undo*

With approaches that explicitly support product-line development and thus expressing variability at the implementation level, we can particularly overcome such limitations. For instance, with our feature-oriented implementation in Figure Figure 3.3 or its annotative counterparts (cf. Chapter 1, Figure 1.1 a), we overcome the aforementioned limitations of multiple inheritance due to class refinement, thus avoiding code replication.

However, even compositional and annotative approaches, which support the programmer in expressing variability, have limitations that may contribute to code cloning as well. To evaluate the existence and effect of code clones in software product lines, we discuss the key mechanisms these approaches by example of FOP and the C preprocessor. In particular, we analyze how they possibly foster code cloning. Moreover, we provide a conceptual analysis that is independent of a particular language, and identify (conceptual) limitations of FOP and the CPP that may tempt the programmer to introduce code clones. To this end, we focus on four criteria related to modularization and expressiveness, how they are supported by both approaches and how this relates to code cloning. The criteria are separation of concerns, granularity of extensions, alternative features, and restructuring features.

Having knowledge on limitations regarding the criteria and its effect on code cloning can help to deal with clones once they occur or even to avoid code clones in advance, e.g., by rethinking the design or coding practices of languages for both, compositional and annotative approaches.

4.1 Separation of Concerns

Separation of Concerns (SoC) is a fundamental concept in software engineering that goes back to the times of Parnas and Dijkstra [Par72, Dij76]. The idea of this concept is to divide a software system into small and preferably cohesive parts. These parts, in turn, have to be semantically meaningful to a certain stakeholder, meaning that they encompass a certain *concern* of the domain. In the context of software product lines, features represent concerns that are important in software product line engineering.

Feature-oriented programming supports the separation of concerns by decomposing a program into its (end-user visible) features. As a result, a feature-oriented program consist of multiple cohesive units, each representing a certain feature, which also increases the overall *traceability* of features in the program. However, not all concerns can be decomposed and treated in a modular fashion at the implementation level. In particular, *crosscutting concerns* are inherently tangled with and scattered across other concerns and thus hard to modularize [KLM$^+$97]. Basically, we distinguish between two kinds of crosscutting concerns (or crosscuts): *homogeneous* and *heterogeneous* [CAB04].

```java
package BasicGraph;
class Graph {
    Vector nv = new Vector (); Vector ev = new Vector ();
    Edge add(Node n, Node m) {
        Edge e = new Edge(n, m);
        nv.add(n); nv.add(m); ev.add(e);
        e.weight = new Weight();
        return e;
    }
    Edge add(Node n, Node m, Weight w) {
        Edge e = new Edge(n, m);
        nv.add(n); nv.add(m); ev.add(e);
        e.weight = w;
        return e;
    }
    void print () {
        for(Edge edge : ev) { edge.print(); }
    }
}
class Edge {
    Node a, b;
    Color color = new Color();
    Weight weight;
    Edge(Node _a, Node _b) { a = _a; b = _b; }
    void print () {
        Color.setDisplayColor(color);
        a.print(); b.print();
        weight.print();
    }
}
class Node {
    int id = 0;
    Color color = new Color();
    void print () {
        Color.setDisplayColor(color);
        System.out.print(id);
    }
}
class Color { static void setDisplayColor(Color c) { ... } }
class Weight { void print() { ... } }
```

Figure 4.2: Excerpts of the Graph Product Line implementation with homogeneous and heterogeneous crosscutting concerns highlighted.

In Figure 4.2, we explain both categories by means of a program, implementing a simple graph structure. The program consists of the core program, *Graph*, and two features *Weight* and *Color*. The code fragments, belonging to both additional features are highlighted with light red (feature *Weight*) and light blue (feature *Color*). In our example, the implementation of *Color* represents a homogeneous crosscut, because the same piece of code extends the basic functionality of *Graph* at multiple places. In particular, the classes `Edge` (Line 22 and 26) and `Node` (Line 33 and 35) are extended with identical code fragments. Obviously, such kind of clones inevitably lead to code clones [Ape10]. In contrast, a heterogeneous crosscut extends a program with different pieces of code in multiple places. In our example in Figure 4.2, feature *Weight* represents such kind of concern, because it extends the base program with different code fragments at different places (Line 7, 10-15, 23, 28). Other common examples for crosscutting concerns is *logging* functionality (homogeneous), usually scattered across the whole program or the *transaction* functionality (heterogeneous) in database management systems that is tangled with other concerns such as query facilities or index structures.

It is always difficult to separate crosscutting concerns. Different studies indicate that FOP is good for heterogeneous crosscuts [ALS08]. In contrast, encapsulating homogeneous crosscuts into feature modules is impossible or at least cumbersome. Especially, if these crosscuts are wrapped around by code, belonging to other concerns, only complicated workarounds, such as *hook methods*, have to be used for decomposition [KAB07, Ape10]. While this may reduce redundancy (which is inherent to homogeneous crosscuts), it decreases understandability and thus decreases maintainability likewise. Hence, we argue, that FOP, though providing good mechanisms for separation of concerns, could be prone to code clones in the presence of homogeneous crosscutting concerns.

But what about annotative approaches, first and foremost the C preprocessor? Inherently, the CPP does not support separation of concerns, but is known to even break with this concept, which is commonly referred to as a drawback of the CPP [KA09]. For instance, with the CPP, a programmer can annotate a code fragment with different feature variables (e.g. `#ifdef (A || B) && C`) and thus a distinct assignment of source code and features does not exist. Usually, variable code fragments are marked just-in-place by annotating them with `#ifdef` and `#endif` directives. Hence, code related to a feature (or concern) is scattered across the whole code base and tangled with other feature code. From this perspective, the CPP even contributes to crosscutting and thus could be prone to clones as well, much like feature-oriented programming. However, in certain situations, the CPP could yet avoid clones due to its high flexibility. For instance, assume that two features *Foo* and *Bar* add the same code fragment to the base code. With FOP, this code would exist separately in the respective feature modules and thus, per definition, it would be a code clone. In contrast, with CPP we can annotate the respective code fragment with the following boolean expression: `#ifdef Foo || Bar`. As a result, the annotated code fragment is selected for all variants that contain feature *Foo* or *Bar* or both features.

4.2 Granularity of Extensions

A feature extends existing program structures introduced by other features. Extensions can be carried out at different levels of granularity [KAK08]. For instance, in our *Stack* product line in Figure 3.3, we extend method `pop` in feature *Undo* by a statement at the beginning of the method. Furthermore, we extend class `Stack` by method `peak` in feature *Peak* in the same example. While FOP (and other compositional approaches) works fine for coarse-grained extensions, it has limitations when realizing fine-grained extensions such as extensions at statement level [MLWR01, KAK08]. For instance, extending a program by adding statements in the middle of an existing method is not possible with FOP without cumbersome boilerplate code. Furthermore, extending a program by adding a parameter to a method's signature is only possible with workarounds as well with FOP [KAK08].

Feature *BaseStack*

```
1  class Stack { ...
2    void push(Object elem) {
3      if (elem == null)
4        return;
5      elemData[size++] = elem;
6    }
7  }
```

a) Original declaration of method **push**

Feature *Synchronized*

```
1  class Stack { ...
2    void push(Object elem, Transaction txn) {
3      if (elem == null || txn == null)
4        return;
5      Lock l = txn.lock(elem);
6      elemData[size++] = elem;
7      l.unlock;
8      fireStackChanged();
9    }
10 }
```

b) Fine-grained extension by introducing redundancy using FeatureHouse

Feature *BaseStack*

```
1  class Stack { ...
2    void push(Object elem) {
3      if (elem == null || h1())
4        return;
5      h2();
6    }
7    boolean h1() {
8      return false;
9    }
10   void h2() {
11     elemData[size++] = elem;
12   }
13 }
```

Feature *Synchronized*

```
1  class Stack {
2    Transaction txn = new Transaction();
3
4    boolean h1() {
5      return txn == null;
6    }
7
8    void h2(Object elem) {
9      Lock l = txn.lock(elem);
10     original();
11     l.unlock;
12   }
13 }
```

c) Fine-grained extension with hook methods using FeatureHouse

Figure 4.3: Two implementation approaches for fine-grained extensions in Stack SPL.

We illustrate this dilemma whether to introduce clones or use workarounds instead by means of an example. Suppose we want to extend our Stack product line by the feature *Synchronized* to manage concurrent access on the data structure. In Figure 4.3 a we show the initial implementation of method **push** in the feature *BaseStack*. For the the support of concurrent access, we have to introduce several fine-grained extensions, which affect the implementation of method **push**, initially declared in feature *BaseStack*. In particular, this method has to be refined by an additional parameter (Line 2), an additional condition (Line 3), and additional statements that wrap around an existing statement (Line 5–7) (cf. Figure 4.3 b; highlighted code).

With the implementation in Figure 4.3 b, the additional functionality of feature *Synchronized* is realized by completely rewriting the initially declared method **push**. While this is a viable solution, it inherently introduces code clones compared to Figure 4.3 a. Of course, this is only a minor problem in our example regarding the amount of cloned code. However, it may become a severe problem if hundreds lines of code have to be copied (and maintained synchronously afterwards). In Figure 4.3 c, we show an alternative implementation strategy that avoids code cloning. To this end, the *hook methods* `h1` and `h2` are introduced in the implementation of feature *BaseStack*. These hook methods serve as placeholder for the fine-grained extensions, introduced by feature *Synchronized*. While this approach prevents code clones, it has other drawbacks. First, for introducing the new feature *Synchronized*, we have to change the implementation of feature *BaseStack* and thus separation of concerns is compromised. Second, we introduce a structured overhead, which may obfuscate the source code and make it difficult to understand [KAK08]. Hence, practitioners may be reasonable to use the first implementation strategy that introduces code clones but retain modularity and understandability of the source code.

With the CPP, the aforementioned problems of realizing fine-grained extension do not exist. The reason is that the CPP enables a developer to mark arbitrary code fragments with preprocessor annotations. In Figure 4.4 a, we show an example of how to realize the feature *Synchronized* with preprocessor annotations. In this example, all code fragments belonging to feature *Synchronized* are annotated by `#ifdef`s, independent of their granularity. For instance, in Lines 3–5, we annotate a single parameter in the signature of method `push` that belongs to feature *Synchronized*. Similarly, we annotate a particular condition of an `if` statement in Line 8–10. According to Liebig et al., we call such fine-grained extensions *undisciplined annotations* [LKA11]. While such annotations provide flexibility and expressiveness for implementing variability, they have several drawbacks [SC92, BM01, LKA11]. First, undisciplined annotations obfuscate source code, as indicated by our example in Figure 4.4 a. Second, such annotations hinder automated analysis and transformation of the program they are used with. The reason is that undisciplined annotations can not be mapped to elements of the AST, which is often necessary for such automated tasks. Finally, fine-grained annotations are prone to introduce syntax errors. For instance, a developer annotates a single parameter but forgets to annotate the corresponding comma that is used to separate this parameter from the other one. Another source of error is the case where an opening and a closing bracket are part of different annotations, which is hard to capture with undisciplined annotations.

Alternatively, a developer can use *disciplined* annotations, which is even recommended in coding guidelines such as for Linux. Disciplined annotations align with the underlying program structure and thus overcome the aforementioned problems of undisciplined annotations. In Figure 4.4 b, we show an implementation of the *Synchronized* feature with disciplined annotations. In contrast to the undisciplined implementation, the respective code is annotated on method level and thus aligns with the underlying structure of the program. As a result, two alternative methods `push` exist, one method with support for concurrent access and one method without. Unfortunately, the code

```
class Stack {
 void push(Object elem
#ifdef SYNC
 , Transaction txn
#endif
 ){
    if (elem==null
#ifdef SYNC
   || txn==null
#endif
   )
   return;
#ifdef SYNC
   Lock l=txn.lock(elem);
#endif
   elementData[size++] = elem;
#ifdef SYNC
   l.unlock();
#endif
   fireStackChanged();
 }
}
```

(a) Fine-grained extension using undisciplined annotations

```
class Stack {
#ifdef SYNC
 void push(Object elem, Transaction txn) {
   if (elem==null || txn==null)
     return;
   Lock l = txn.lock(elem);
   elementData[size++] = elem;
   l.unlock();
   fireStackChanged();
 }
#else
 void push(Object elem) {
   if (elem==null)
     return;
   elementData[size++] = elem;
   fireStackChanged();
 }
#endif
}
```

(b) Fine-grained extension using disciplined annotations

Figure 4.4: Two possibilities of fine-grained extension with the C preprocessor, by (a) using undisciplined and (b) disciplined annotations

that is common for both methods is not encapsulated separately. Hence, the addressing fine-grained extensions with disciplined annotations inherently introduces code clones.

In summary, we argue that for both implementation approaches, FOP and CPP, fine-grained extensions can be expressed without code clones. However, since these non-redundant implementations have several drawbacks, it may be reasonable to introduce code clones to gain other advantages such as automated program analysis or higher understandability of the source code.

4.3 Alternative Features

To describe the dependencies between features and valid variants within an software product line, different constraints exist, which we can express within the feature model. In this context, *alternative features* (also called *Alternative-Group*) describe the constraint that out of a set of sibling features (in the feature model) only one feature can and *must* be selected for a particular variant. It is in the nature of alternative features, that they implement functionality that is similar to some extent regarding each other. This, in turn, may lead to code fragments that are replicated to implement the common parts between alternative features. We illustrate this fact by means of the *Graph Product Line (GPL)*. In Figure 4.5, we show listings of the alternative features *BFS* and *DFS*, implementing the algorithm for breadth-first search and depth-first search, respectively. Apparently, both features share a large, identical portion of code. In fact, they only differ in Line 18, where a method is called (`bfSearch` and `dfSearch`,

Feature *BFS*

```java
public class Graph
{
    public void search(WorkSpace w)
    {
        VertexIter vxiter = getVertices();
        if (vxiter.hasNext() == false) return;
        while (vxiter.hasNext())
        {
            Vertex v = vxiter.next();
            v.init_vertex(w);
        }
        for (vxiter=getVertices();vxiter.hasNext();)
        {
            Vertex v = vxiter.next();
            if (!v.visited)
            {
                w.nextRegionAction(v);
                v.bfSearch(w);
            }
        } //end for bfsSearch
    }
}
```

Feature *DFS*

```java
public class Graph
{
    public void search(WorkSpace w)
    {
        VertexIter vxiter = getVertices();
        if (vxiter.hasNext() == false) return;
        while (vxiter.hasNext())
        {
            Vertex v = vxiter.next();
            v.init_vertex(w);
        }
        for (vxiter=getVertices();vxiter.hasNext();)
        {
            Vertex v = vxiter.next();
            if (!v.visited)
            {
                w.nextRegionAction(v);
                v.dfSearch(w);
            }
        }
    }
}
```

Figure 4.5: Code clones between features *BFS* and *DFS* in *GPL*

respectively) that obviously contains the core functionality in that both features differ. However, because both features are alternative it is not possible to have this common code in only one feature and extend it (via refinements) by the other feature. Of course, other transformations are possible to extract the common code and thus to remove the code clones. However, this is neither trivial nor always possible. We provide more details on such code clone removal in Chapter 7. Although this is a desperate example, implemented by someone else, it illustrates the problem of alternative features causing code clones very well.

With CPP, in contrast, we can avoid code clones by exploiting the capability of expressing fine-grained extensions. Instead of annotating the whole class `Graph` for each feature, as done in our feature-oriented implementation in Figure 4.5, we only annotate the part of the implementation that differs. In Figure 4.6, we show the implementation of the alternative search algorithms in GPL with preprocessor annotations. Instead of replicating the common code for each alternative feature, we only have to annotate the lines of code that differ between these features. Note that this is similar to FOP used with hook methods instead of replication. In particular, we annotate the method call for `bfSearch` with the preprocessor variable *BFS* (cf. Line 18–19) and the call for `dfSearch` with the preprocessor variable *DFS* (cf. Line 20–21). Furthermore, due to the possibility of fine-grained extensions, we could even express alternative features that differ only in partial statements or conditions. Hence, we argue that annotation-based SPLs are not prone to clones caused by alternative features.

```
public class Graph
{
    public void search(WorkSpace w)
    {
        VertexIter vxiter = getVertices();
        if (vxiter.hasNext() == false) return;
        while (vxiter.hasNext())
        {
            Vertex v = vxiter.next();
            v.init_vertex(w);
        }
        for (vxiter=getVertices();vxiter.hasNext();)
        {
            Vertex v = vxiter.next();
            if (!v.visited)
            {
                w.nextRegionAction(v);
\#ifdef BFS
                v.bfSearch(w);
\#elseif DFS
                v.dfSearch(w);
            }
        }
    }
}
```

Figure 4.6: Implementing alternative features *BFS* and *DFS* with annotations

4.4 Feature Evolution

As any software system, SPLs are subject to software evolution. In the context of an SPL, evolution takes place in different ways: First, existing features are extended or changed due to new requirements and thus grow over time. Second, features are added to introduce new functionality or extract and separate existing one from other features. Third, features can be merged if they contain similar functionality that is reasonable to encompass within one feature. Alternatively, a feature could be too large (encompassing to much functionality) and thus has to be split into two or more features. All these kinds of evolution may lead to code clones.

Although adding/modifying code is not problematic with respect to clones on first sight, it may suffer from similar problems as in standalone programs. That is, a programmer extends the code without knowing that similar (or even identical) code already exists, for instance in a sibling feature. Vice versa, a programmer could be aware of a code fragment (e.g., a particular algorithm) in another feature that she reuses by copy&paste in the feature currently under change.

Similar problems may arise when a new feature is introduced. For instance, consider the search algorithms (BFS and DFS) of our Graph Product Line (cf. Figure 4.5). Suppose we want to add our own search algorithm as a new (alternative) feature, reusing as much as possible of the existing algorithms. Hence, we copy parts of one of existing features (which already exhibit identical parts) and modify only what is different within our algorithm. This may be even a common way, because an appropriate abstraction such as unifying the common code in a distinct feature is not easy to find or even not exists.

Moreover, there may be the point, for example after partly uncontrolled or unorganized evolution, where an SPL contains features that encapsulate very similar functionality. Thus, it is worth to unify such features by merging them into one unique feature. During this process, we expect that most of the corresponding code fragments are merged into unique code fragments. However, there may be also parts that slightly differ and thus can only be merged by introducing redundancy. Especially, if the differences are at a fine grain, redundancy is almost unavoidable (cf. Section 4.2). But, also the opposite direction, that is, a feature encompasses more functionality than it was designed for, may lead to clones. As a result of its growth, the feature has to be split, and the corresponding code has to be extracted from the existing feature into the new ones using CUT&PASTE. However, since this code is often tightly coupled with other parts of the original code (i.e., its context), it might be unavoidable to reuse parts of this code by COPY&PASTE, which may cause code clones.

Overall, we argue that these evolutionary clones are very similar to those in standalone programs and may occur independent of the implementation approach. However, currently it is open to what extent such evolutionary changes (or even patterns) occur and how often they really lead to clones.

4.5 Summary

Although software product lines make use of enhanced reuse facilities amongst similar software products, there is a certain risk of code clones. In this chapter, we discussed several criteria related to modularity and flexibility of SPLs that may lead to clones. Furthermore, we pointed out that the implementation approach makes a difference regarding the proneness of clones.

Criteria	Feature-Oriented Programming	C Preprocessor Annotations
Separation of Concerns	good, moderate risk of clones (→)	no separation of concerns, moderate risk of clones (→)
Granularity	coarse-grained extensions, high risk of clones (↗)	fine-grained extensions, low risk of clones (↘)
Alternative Features	possibly overhead, high risk of clones (↗)	efficient solution, low risk of clones (↘)
Feature Evolution	not directly supported, moderate risk of clones (→)	not directly supported, moderate risk of clones (→)

Table 4.1: Overview of different criteria and their influence on code clone occurrence in FOP and the CPP.

In Table 4.1, we provide an overview of how the discussed criteria may affect clone occurrences, taking the respective implementation approach into account. Generally, we argue that preprocessor-based software product lines are less prone to code clones due to their high flexibility and. Especially, for fine-grained extensions and alternative features, this makes a huge difference regarding the risk of clones. However, this expressiveness is a double-edged sword that may affect characteristics of the source code such as readability or error-proneness. Hence, we also pointed out that it is sometimes even worth to introduce clones to increase the aforementioned characteristics. While this chapter contains a conceptual discussion (with some real-world examples) to reason about clones in SPLs, we present case studies on that topic in the next chapters.

5. Code Clones in Feature-Oriented Software Product Lines

In the previous chapter, we discussed factors that may influence the occurrence of clones, depending on the respective implementation approach. However, while this discussion was at a conceptual level, it is inevitable to investigate practical systems to underpin the assumptions summarized in Table 4.1. Hence, in this chapter, we shed light on the issue of code cloning in FOP empirically. To this end, we present an empirical analysis on ten different feature-oriented SPLs. In particular, we describe the setup, the methodology and results of our analysis. Furthermore, we discuss the results and threats to validity. Overall, we make the following contributions in this chapter:

- By means of a case study on ten different feature-oriented SPLs, we analyze the number and characteristics of clones in FOP.
- We explore and discuss whether code clones occur independently of the fact that an SPL has been developed from scratch or refactored from a legacy application.
- We initiate a discussion on the role of code clones in FOP.

5.1 Experimental Setup

In the following, we present some preliminary considerations and the setup of our empirical analysis.

Research Questions

In Chapter 1, we posed four research questions that are of paramount interest in this thesis and which we want to answer with the help of our empirical analysis. However, to address specific characteristics of FOP, which may also influence the occurrence and nature code clones, we concretize the aforementioned research questions to gain deeper insights of code clones in feature-oriented SPLs. As a result, the following research questions arise, which we want to answer with our case study on clones in FOP:

RQ1 *Do code clones exist in feature-oriented SPLs?* This question corresponds to the research question in Chapter 1. Answering this question provides information to what extent code clones exist in feature-oriented SPLs at all.

RQ2 *Is FOP prone to introduce FOP-specific clones especially in the context of SPLs?*

Since FOP partly relies on other language mechanisms such as inheritance, there may be clones that are either caused by these mechanisms than by those, specific to FOP. Hence, we have to distinguish between *FOP-specific* and other code clones. To this end, we also have to define *what* actually makes a code clone FOP-specific.

RQ3 *If clones exist, does the development process of the SPL (e.g., from scratch or by refactoring legacy applications) influence their occurrence?*

Generally, two approaches exist for the overall process of product-line development: proactive and reactive/extractive. While the former implies that an software product line is implemented from scratch, the latter describes the fact that the initial versions of a certain SPL is refactored from existing legacy applications. In the case of FOP, the reactive approach can be considered as *software migration*, because it includes a shift from one programming paradigm to another (e.g., from OOP to FOP). Since this may cause considerable changes of the underlying source code, we focus on the influence of the development process on code clone occurrences with this question.

Subject Systems

The subjects of our analysis are ten feature-oriented SPLs of different size ranging from 150 to 45 000 SLOC[1]. While this is rather small, compared to industrial applications, the subject systems encompass the largest SPLs available in FOP at the time of the case study. All SPLs were developed with FOP tools based on Java, namely FeatureHouse [AKL09] and AHEAD [BSR04]. Furthermore, the selected feature-oriented SPLs stem from different domains such as database systems, editors, and games. We list relevant information in Table 5.1. The programs in the upper half of Table 5.1 are implemented from scratch, whereas the others are refactored from legacy applications. Furthermore, we provide some information on authorship, code size, and the domain. Note that we consider the whole code base of the feature-oriented product lines for our

[1]SLOC is an acronym for source lines of code, a common metric that refers to the length of the source code excluding comments and blank lines.

Name	Programmer	# SLOC	# FM	Description
GPL[1]	R. Lopez-Herrejon (UT Austin)	1 929	28	graph and algorithm library
GUIDSL[2]	D. Batory (UT Austin)	11 527	29	graphical configuration tool
Notepad[3]	A. Quark (UT Austin)	1 012	13	graphical text editor
PKJab[4]	P. Wendler (U Passau)	3 305	8	instant messaging client
TankWar[5]	L. Lei et al. (U Magdeburg)	4 933	38	shoot 'em up game
EPL[6]	R. Lopez-Herrejon (UT Austin)	149	11	arithmetic expression evaluator
Berkeley DB[7]	C. Kästner (U Magdeburg)	45 000	100	transactional storage engine
MobileMedia[8]	C. Kästner (U Magdeburg)	4 227	47	multimedia management
Violet[9]	A. Kampasi (UT Austin)	7 194	88	graphical model editor
Prevayler[10]	J. Liu (UT Austin)	5 270	6	persistence library

FM: feature modules

Table 5.1: Overview of the analyzed SPLs

analysis rather than certain variants. As a result, we are able to detect code clones across the boundaries of individual features, which are of interest for our analysis. For information on dependencies and relations amongst features, a feature model exists for each of the considered SPLs. All SPLs can be downloaded from the Web[2] or are contained as examples within FeatureIDE[3].

Clone Detection

We performed clone detection on the selected SPLs using the clone detection tool *CCFinder*, which uses the token-based approach [KKI02]. Within the tool, the user can specify different parameters such as minimum clone length. Guided by a former study that used CCFinder [BDET05], we set the minimum clone length to 50 tokens, which corresponds to five lines of code. This value also corresponds to commonly chosen clone length of other studies. This way, we omit meaningless code clones such as getter and setter methods, which occur incidentally and thus have no value for our analysis. Afterwards, we merged corresponding code clones to *clone classes* based on the detection results, as usual in clone detection (e.g., [KKI02, BDET05]). To this end, we exploited the equivalence relation that exist between corresponding clone pairs, guaranteed by CCFinder. As a result, we can treat these clones as a unit for further analysis steps or even for their removal. Finally, we performed some minor transformations on the clone classes[4] such as removing comments or white spaces.

5.2 Code-Clone Analysis Process

To analyze our subject systems regarding code clone occurrences, we set up an analysis process that consists of three steps: clone detection, syntactical classification, and feature-related classification. In the following, we explain each step in detail.

[2]http://www.fosd.de/fh

[3]http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/

[4]Whenever an action is performed on a clone class in the following, this action affects all of its member clones.

5.2.1 Clone Detection

This is the first step of our analysis process and thus provides the initial input for the further analysis. Hence, the quality of the clone detection result is crucial for the result overall analysis result. For instance, too many false positive clones make the following analysis steps a tedious task, because we would have to filter out all the false positives to improve the results by only considering significant or meaningful clones. In contrast, a too restrictive clone detection could leave out interesting clones (false negatives) and thus would lead to a result that is only partially correct.

To achieve a "good" clone detection result (i.e., a result that is neither at the one nor at the other end of the aforementioned possibilities of bad results), the tool CCFinder provides different parameters that can be specified. Besides the minimum clone length, we already mentioned, we had to set the following parameters:

- **p-match:** With this parameter, we can specify whether we want to perform a *parameterized match* during the clone detection and thus detect *Type-II* clones as well. Without p-match, all variables and identifiers are replaced by a special token and thus differences between these units are neglected. For instance, the statements `return A+B;` and `return X+Y` are both transformed into `return $+$`, where $ is the special token that is used to replace identifiers. Hence, the two statements would be detected as a clone pair. For our case study, we include such parametrized clones that occur due to *copy,paste&adapt*. Hence, we performed clone detection *without* p-match, which causes parameterization of certain elements (as described above).

- **shaper:** During clone detection, meaningless or uninteresting clones such as sequences of variable initialization or getter/setter methods are detected as well, especially, if we allow parameterization to some extent. Furthermore, clones may be detected that overlap the boundaries of certain syntactical blocks, which is not desirable either. For instance, similar code fragments that start at the end of a method and end in the middle of another one usually neither provide no useful information about the cloning process nor indicate intentional clones. Hence, it is desirable to detect clones at the block level, which are more significant. The shaper parameter of CCFinder supports filtering clones at a certain block level during the preprocessing phase. For instance, with *hard shaper*, only token sequences that are enclosed by a block (e.g., method, loops) are considered as clones. In contrast, without any shaper, any token sequence is considered as a clone as long as it is compatible with the other parameters. For our purposes, we used the *soft shaper*, which prevents that clones are split by the most outer block. For instance, if a code clone exceeds the boundary of a `for` loop but not the boundaries of its enclosing method, it is still considered as clone for further treatment.

- **token size**: This parameter specifies the minimum number of *different* token and thus inherently affects the clone detection result. As an illustrative example, considering the following sequence of variable initialization: `A = 1; B = 1 + 2; C = 1 + 2 + 3;`. This sequence consists of your different types of token, namely, *identifier, integer literal, "=", ";".* If we choose this parameter to be greater than four, this sequence would not be considered as code clone, even if multiple similar or identical sequences exist. In general, less interesting clones such as the aforementioned sequences of variable initialization consists of only few different token. Consequently, we used the default value, proposed by CCFinder, which is 12, to reject such code clones beforehand.

As a result of clone detection, we obtain a list of clone pairs. As a kind of post-processing step, we merge corresponding clone pairs to clone classes. For instance, two clone pairs $cp(A, B)$ and $cp(B, C)$ are merged to clone class $cc(A, B, C)$ by exploiting the equivalence relation that is created between corresponding clones by CCFinder. These clone classes are subject for further treatment within our clone analysis process.

5.2.2 Syntactical Classification

The result of the initial clone detection is the input for further analysis to filter clones of interest. The first step of this process is *syntactical classification*, where we classify the clone classes by their syntactical category. That is, we map the detected clones to the elements of the underlying AST to determine whether a clone relates to certain syntactical elements such as statements or methods. Subsequently, we filter those clone classes that we classified into one of the following categories with the obvious meanings: *IfStatement*, *ForStatement*, *WhileStatement*, *DoStatement*, *MethodDeclaration*, and *TypeDeclaration*. We do this for two reasons. First, these categories indicate enclosing blocks (e.g., `for` loops) that usually encapsulate a coherent piece of functionality. For instance, a method may implement a certain algorithm or a loop is used to iterate over a certain data structure, including some additional actions. Such clones usually result from explicit cloning activity, either to reuse specific code in another context by COPY&PASTE or for reusing general functionality such as commonly accepted algorithms (e.g., Fibonacci). Additionally, such clones may be potential candidates for removal actions, either by introducing abstractions or by extract them into an external library.

The second reason for this filtering process is that these categories provide good refactoring opportunities for code clone removal. Note that this does not address the fact whether a clone class *should* be removed (e.g., because it is considered harmful), but whether it is possible to remove a clone class by means of well-established refactorings. Code clones that do not fit into enclosing blocks impede such a removal, because a) they are difficult to extract and b) they rather occur by accident than on purpose. For instance, considering code clones that partly belong to a loop and partly to its surrounding method. Removing such clones comes with different pitfalls (e.g., validity/accessibility of local and global variables) and is only possible, if at all, with cumbersome workarounds. In contrast, considering clones that are enclosed by the mentioned blocks, provide a vast amount of refactorings that are possibly applicable.

For instance, we could remove them by applying *Extract Method* or *Pull Up Method* refactorings [Fow00], tailored to SPLs.

As a result of this classification, we obtain a set of clone classes that adhere to the aforementioned syntactical blocks. We investigate these clone classes further within the final analysis step.

5.2.3 Feature-related Classification

In the last step of our clone analysis process, we investigate whether the detected clones are FOP-specific or not. In other words, we figure out to what extent existing clones occur due to the feature-oriented nature of the software product line. To this end, we need a criterium that enables us to decide whether code clones are FOP-specific or not. Next, we give a definition for FOP-specific clones, on which we rely during our feature-related classification.

Definition 2 (FOP-specific clone class)**.** *A clone class, that is, a set of corresponding code clones, is FOP-specific, if and only if, at least two of these corresponding clones are located in different features.*

We argue that it is reasonable to distinguish FOP-specific clones based on their distribution over features, because features are distinct entities in FOP compared to other programming paradigms. Hence, if code clones occur between different features, they are presumably caused by limitations of feature-oriented language mechanisms or by the feature-oriented development process. In the first case, it may be impossible to unify clones by abstraction (e.g., locate replicated code in a distinct feature), while in the latter case programmers are not aware of such opportunities for abstraction. Moreover, if code clones occur only within one feature, either in a single or in different classes, we can not clearly determine whether these clones are caused by FOP or not. Since FOP partly relies on OOP, these clones may occur for the same reasons as in usual, standalone software systems. Hence, we refer to all code clones that do not adhere to Definition 2 as *OOP-specific clones*.

Beyond the mere classification of FOP-specific clones, this step also encompasses a closer look at the features that contain code clones. More precisely, we are interested in the relation of features that share corresponding clones. This, in turn, provides insights in the nature of FOP-specific clones, specifically, whether certain patterns (with respect to the features) exist regarding code clone occurrences. In Section 4.3, we already discussed the possibility of clones between alternative features. However, other relations between features exist as well, such as parent-child relationship or cross-tree constraints. In particular, we focus on the following categories during this analysis step:

- *Common Parent Feature (CPF):* This category encompasses all clone classes, where the corresponding code clones occur in sibling features that have a common, direct parent feature.

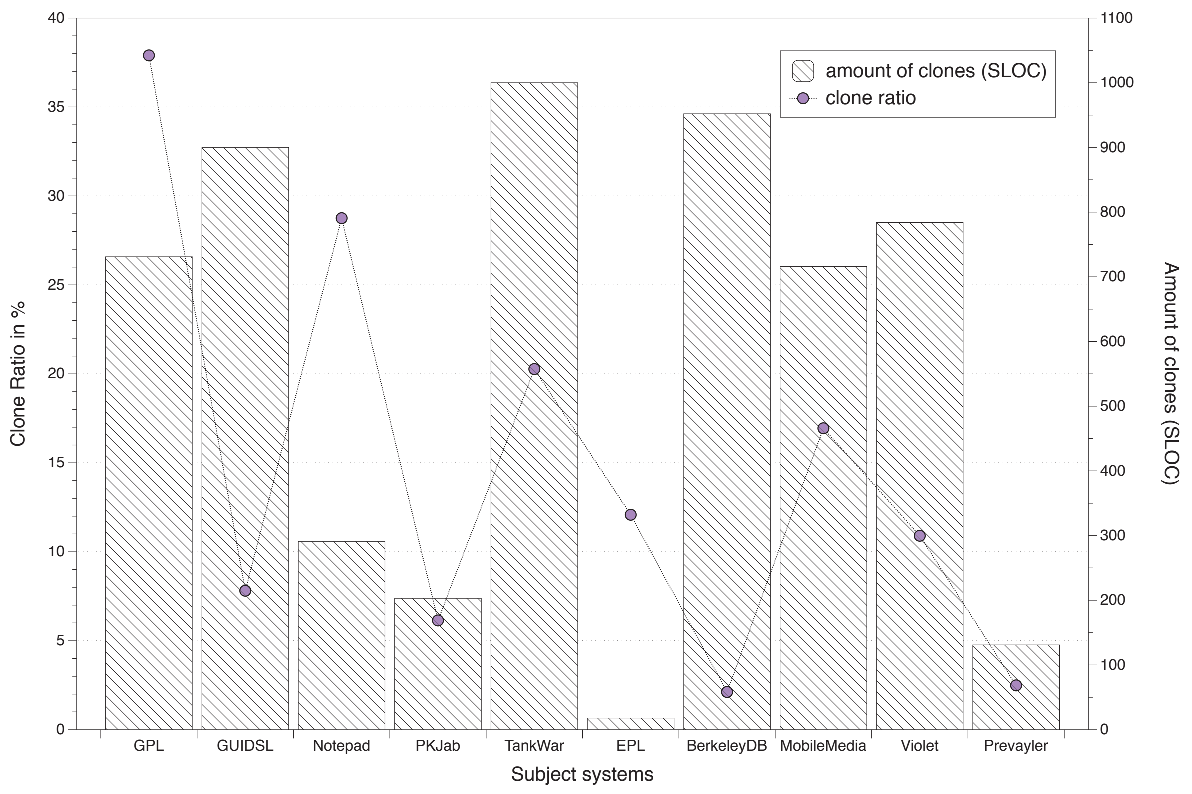


Figure 5.1: Overall clone ratio and amount of clones.

- *Alternatives (A):* This category represents a sub-category of the previous category. All clone classes that are classified into category *CPF* and, beyond that, where all sibling features belong to an alternative group are part of this category.

- *Common Dependency Feature (CDF):* This category encompasses all clone classes, where the corresponding code clones are located in features that are related with at least one common feature. This relation could be a parent-child relationship or a cross-tree constraint. For instance, if two features A and B share corresponding code clones of clone class $\mathrm{CC}_{\mathrm{AB}}$ and the cross-tree constraints $A \Rightarrow C$ and $B \Rightarrow C$ exist, the clone class is part of this category.

Beside the identification of recurring patterns for code clone occurrences, we use the aforementioned categorization to identify code clones that are potential refactoring candidates. We explain this in detail in Chapter 7. Finally, we also investigate how many features are affected by code clones within this analysis step. As a result, we obtain insights whether clones disseminate throughout the whole SPL or only in certain features (e.g., related to the GUI).

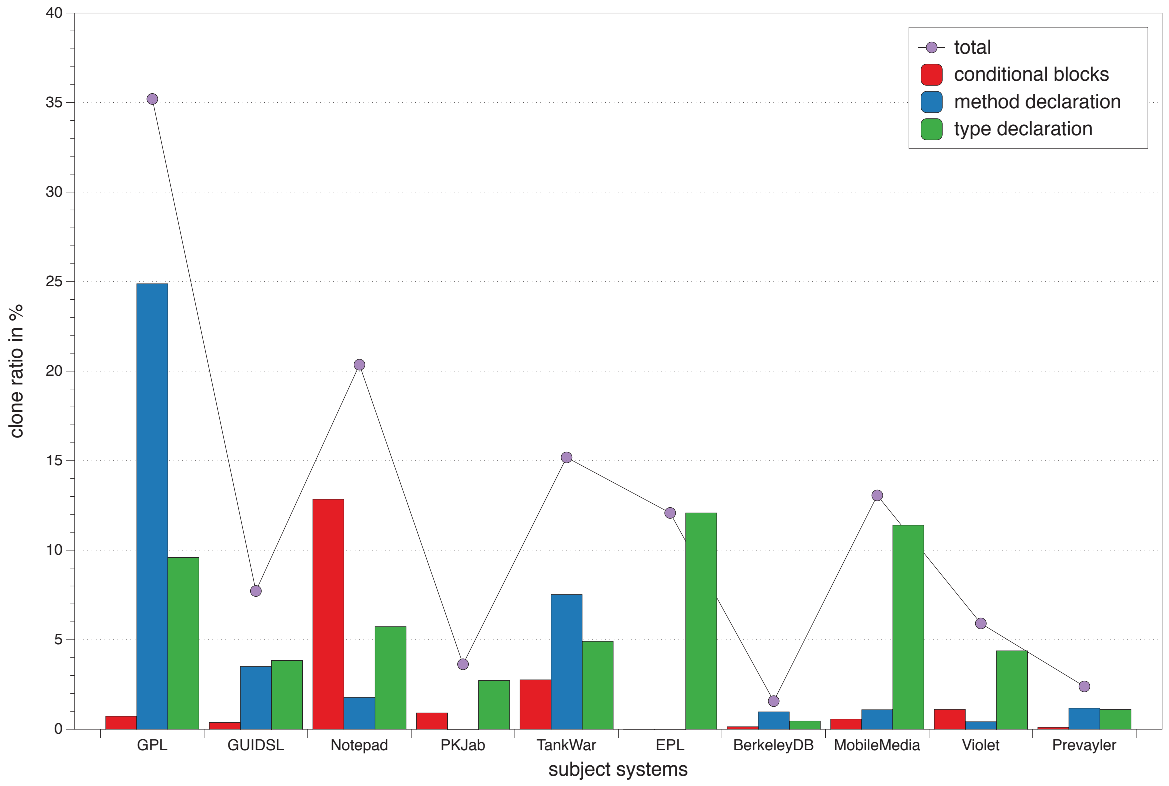


Figure 5.2: Block clone ratio – overall and separated by syntactical blocks.

5.3 Results

Next, we present the results of our code clone analysis on ten feature-oriented SPLs. We structure this section according to the analysis steps, presented in the previous section. Additionally, we examine the differences that may result from the development process of our SPLs (from scratch or refactoring). For discussion and interpretation of the results, we refer to Section 5.4.

A common measure to determine the amount of clones relative to the whole source code is the *clone ratio*. Consequently, we use this measure as well to relate the amount of clones to the total amount of code (SLOC) for each SPL. Additionally, we calculated the percentage of the average and the standard deviation ($a \pm s$) on the clone ratio of all considered SPLs.

Amount of code clones.

The results of our initial clone detection reveal that there is a significant amount of clones in feature-oriented SPLs (cf. Figure 5.1). Regarding all considered SPLs, 15 ± 10 % of the overall code are clones. We observed considerable differences regarding the clone ratio of the particular SPLs that ranges from 2 % to 37 %, which is also reflected by the relatively high standard deviation. Beyond this, we noticed that two of the smallest SPLs (*GPL* and *Notepad*) exhibit the highest clone ratio values with 37 % and

28 % respectively. By contrast, the two largest SPLs (*BerkelyDB* and *GUIDSL*) are amongst those with the lowest clone ratio value.

Block-level clones.

With our first analysis step, we aimed at detecting clones that are encompassed by an enclosing, syntactical block. We show the results in Figure 5.2, where we summarized those clones as *conditional blocks* that map to the following AST nodes: `IfStatement, ForStatement, WhileStatement`. Our data reveal that the clone ratio decreases in comparison to the initial clone detection in almost all SPLs, indicating that we filtered out uninteresting clones within this step for nearly every product line. Nevertheless, there is still a huge amount of clones that are enclosed by syntactical blocks, indicated by a total amount of code clones of 12 ± 9 %. Furthermore, we observed that the clone classes, filtered out by the syntactical classification, mainly fall into the categories *MethodDeclaration*, and *TypeDeclaration* (cf. Figure 5.2, green and blue bars), except for Notepad SPL. Particularly, we noticed the high amount of code clones in category *TypeDeclaration*, which means that whole classes have been cloned.

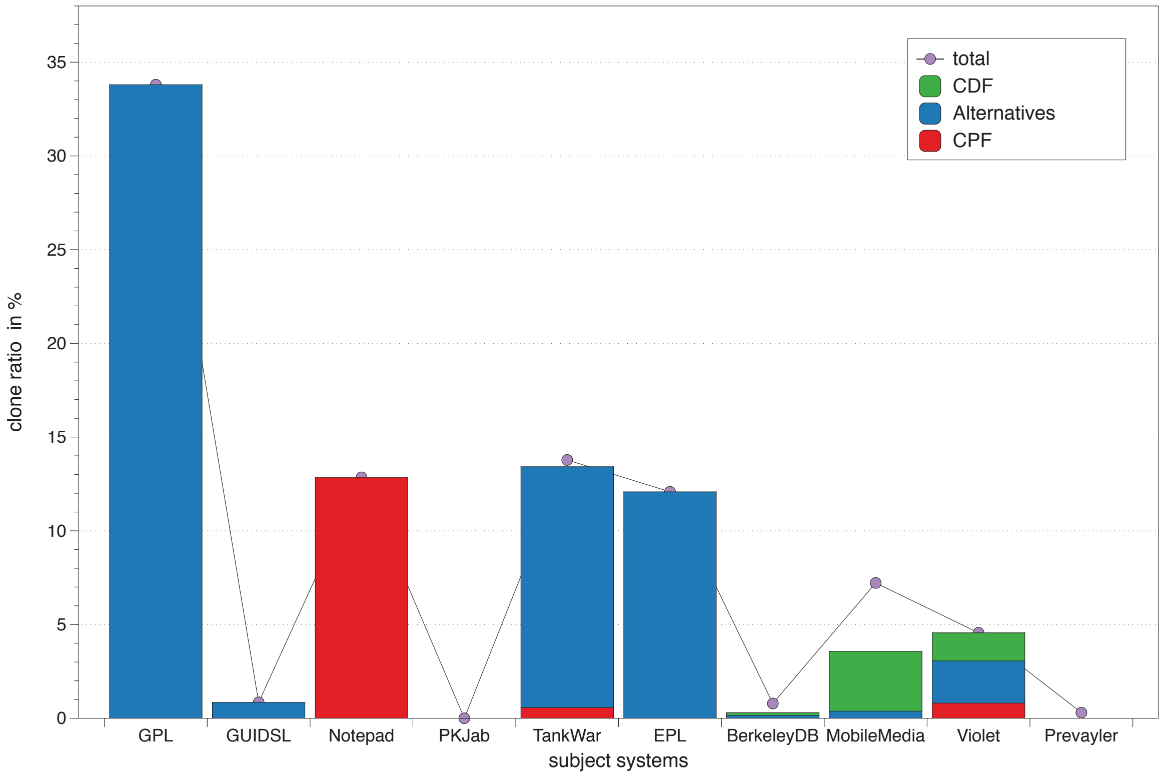


Figure 5.3: Ratio of FOP-specific clones – overall and distinguished by feature relations.

FOP-specific clones.

The data resulting from the last analysis step (cf. Figure 5.3) reveal that a considerable amount of code clones exist that are FOP-specific (by our definition). Nevertheless, we observed that four feature-oriented SPLs (*GUIDSL*, *PKJab*, *Berkeley DB*, and *Prevayler*) contain (almost) no FOP-specific clones. We assume that this results from the fact, that these SPLs exhibit the lowest clone ratio even in the initial clone detection (cf. Figure 5.1) and that the existing clones are OOP-specific. Generally, we observed that the clone ratio is considerably lower than the clone ratio after syntactical classification for all SPLs except of *GPL*. Regarding all SPLs, the amount of clones is 9 ± 9 %, which reveal that there is a high diversity between the clone ratio of the several SPLs. Actually, four SPLs exhibit a clone ratio greater than 10 %, whereas the clone ratio of the remaining SPLs is less than 8 %. Finally, our data reveal that FOP-specific clones are mostly distributed over alternative features with a common parent feature (blue bar in Figure 5.3). Only the *Notepad SPL* exhibit clones in features that have common parent feature but are not alternative, indicated by the red bar in Figure 5.3. Finally, in three SPLs (*MobileMedia*, *BerkeleyDB*, and *Violet*), clones are contained in features that have a common dependency feature instead of a common parent feature.

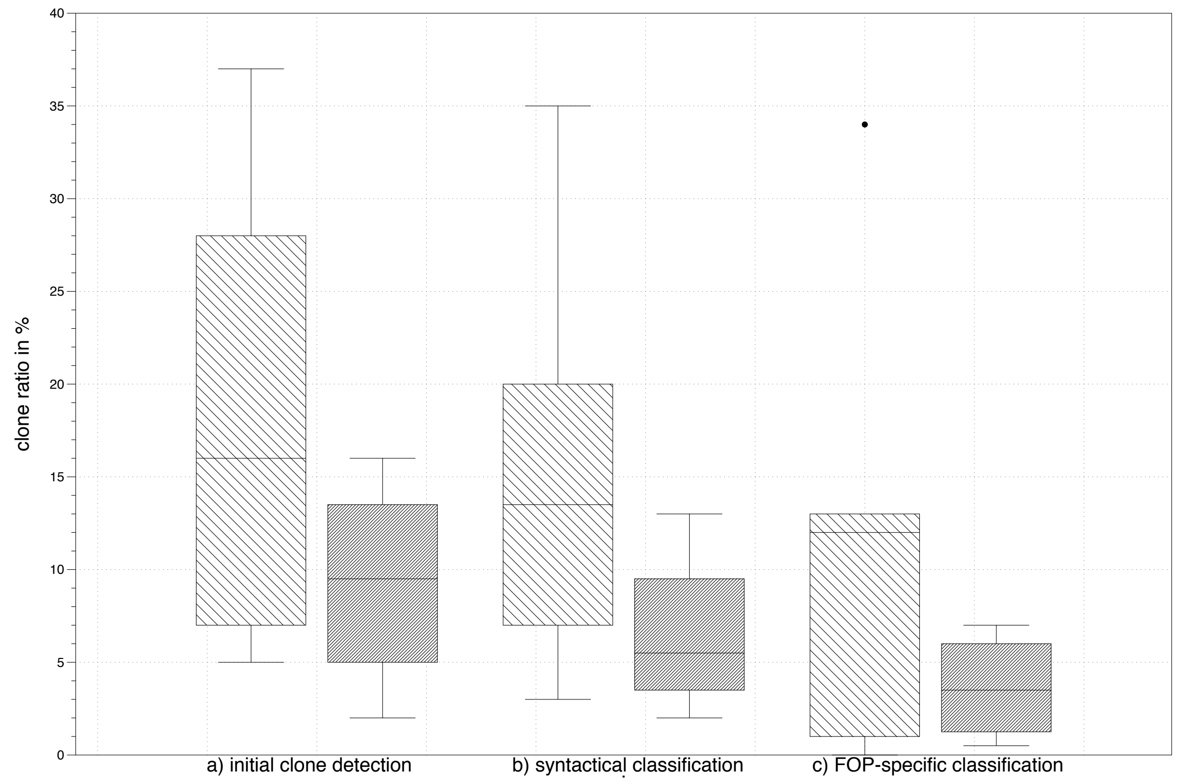


Figure 5.4: Comparison of SPLs from scratch vs. refactored from legacy for all three steps of our code clone analysis process.

From scratch vs. Refactoring.

Beside the amount of clones per system, we were also interested whether the development process has an influence on code clone occurrences. Hence, we measured the amount of clones for each analysis step by taking the development process into account.

We show the results by means of box plots in Figure 5.4. The main observation is that the amount of clones in SPLs developed from scratch is considerably higher than in SPLs decomposed from legacy applications for all steps of our clone analysis process. For instance, the amount of clones after the initial clone detection for SPLs developed from scratch is 19 ± 12 % (cf. Figure 5.4 a)). Even after the last analysis step, we detected a considerable amount of clones for these systems (12 ± 12 %, cf. Figure 5.4). In contrast, the amount of clones for software product lines refactored from legacy applications is significantly lower, ranging from 10 ± 5 % after the initial clone detection to 5 ± 4 % after FOP-specific classification. Beyond the differences, caused by the development process, our data reveal a high diversity amongst the software product lines developed from scratch. This diversity (i.e., large differences between the several SPLs regarding amount of clones) is visually indicated by the relatively large range of the box plots in Figure 5.4.

5.4 Discussion

In the following, we interpret and discuss the results of our case study with regard to the research questions we posed in Section 5.1.

Do code clones exist in feature-oriented SPLs?

Based on the results of our analysis, we conclude that a considerable amount of code clones actually exist in feature-oriented SPLs. Beyond that, we observed that there are significant differences, regarding the amount of clones between the analyzed SPLs in general. In a few of them the amount is negligible. In addition, some of the smallest SPLs exhibit the highest amount of clones. However, considering the overall result of the clone detection, we can not confirm a correlation between SLOC metric and clone ratio.

Furthermore, most of the detected clones occur at the block level, meaning that they are inside a syntactical block such as *conditionals* or *methods*. The reason could be that feature-oriented SPLs have a rather coarse-grained nature due to the decomposition into modules. More precisely, fine-grained extensions are exposed by cloning at a coarse grain rather than using workarounds such as hook methods to avoid code replication. However, while this may explain the occurrence of FOP-specific clones at the block level, we are not sure, whether OOP-specific clones at this level occur for the same reason.

Is FOP prone to introduce FOP-specific clones in SPLs?

Our results indicate that there are FOP-specific clones in the analyzed SPLs. An interesting observation is that the majority of these FOP-specific clones occur between alternative features. As already discussed in Chapter 4, such features are often semantically related, because of similar concepts they implement (e.g., *BFS* and *DFS* in the Graph SPL). Thus, alternative features are presumably more prone to clones, which is finally confirmed by our study. Moreover, we observed that most of the FOP-specific clones, independent of the feature constraints, occur due to fine-grained extensions. As discussed likewise in Chapter 4, this is a major disadvantage of compositional approaches and bypassing such extensions inevitably leads to boiler-plate code. Alternatively, code cloning is a more straightforward and clean solution. Indeed, our case study indicates that code cloning is a viable approach to handle fine-grained extensions.

Generally, both observations coincide with the limitations of FOP as analyzed by us and other researchers before (see Chapter 4). However, since we also detected a vast of block-level clones, there may be a considerably high potential for code clone removal, specifically for clones between alternative features. We put emphasis on this topic in Chapter 7.

Nevertheless, we also detected clones that are FOP-specific by our definition, which do not occur in alternative features. Considering our data, we can not clearly infer why these clones occur between these features. Different reasons such as idioms or crosscutting concerns are possible but we can not confirm this entirely. But even without these clones it is a matter of fact that feature-oriented SPLs contain FOP-specific clones that follow certain patterns regarding feature relations and granularity.

Is the development process of the SPL crucial for code cloning?

Our results show differences between the analyzed SPLs that can be ascribed to the development process (from scratch vs. refactoring). In detail, the SPLs developed from scratch contain a significant higher amount of clones than the SPLs decomposed from legacy applications. The SPLs decomposed from legacy applications were developed originally object-oriented and finally, were decomposed (semi-)automatically [KAK09]. For simplifying this decomposition process, complex group constraints have been avoided, and thus most of the features are optional. Based on our results, this is a reasonable explanation for the different amount of clones between SPLs developed from scratch and refactored from legacy. Furthermore, the fact that the corresponding legacy applications have not been designed with variability in mind could also influence the amount of clones.

Finally, the fact that the programmers of the SPLs developed from scratch were may be not capable to exploit all concepts and mechanisms of FOP (as often observed with new programming paradigms) could be a reason for the higher amount of clones. Hence, they may have introduced clones unnecessarily or missed to factor out clones where it was possible with the mechanisms of FOP.

How to deal with clones in feature-oriented SPLs?

During our analysis, we particularly looked at the refactoring potential of the detected clones. The corresponding data reveal that a large portion of the overall detected clones exhibits characteristics that indicate refactoring opportunities. One interesting observation we made is that a huge amount of clones between alternative features are across method declarations. These clones can be refactored by pulling them up to a common parent feature. We will have a closer look to concrete refactorings in Chapter 7.

Beside refactoring, other possibilities exist for managing clones that we did not consider in our analysis, such as clone tracking [DER07] or linked editing [TBG04]. The idea of both approaches is that the detected clones remain in the code but information on their existence is used for their management (e.g., for changing code clones simultaneously).

5.5 Threats to Validity

Although we designed and executed our case study carefully, some threats to validity remain, which we address in the following.

Single FOP language.

Although FOP is a general paradigm, it depends to some extent on the mechanisms of the underlying language. As a result, different FOP languages exist (e.g., for C++ [ALRS05] and Java [BSR04, AKL09]) that may lead to different implementations for feature-oriented SPLs. In this paper, we focused only on FOP languages based on Java, so that the results of our analysis are comparable. However, the classification we made along with our analysis is also valid for other languages (e.g., C++ or C#). Although no empirical evaluation for other languages yet exists, we assume that at least our feature-related analysis is independent of the programming language (or paradigm), because the decomposition of source code into features is independent of the underlying language.

Selected SPLs.

A major problem with case studies is that the selected programs may be biased. To address this problem, we selected SPLs from different domains, of different size, and implemented by different programmers. Nevertheless, one problem remains: all of the analyzed SPLs are prototypical implementations from academia. Hence, there is a lack of comparable results of industrial SPLs, which is also caused by the fact that such systems do not exist for FOP. Nevertheless, the considered SPLs have been implemented by different authors and for other purposes than analyzing them for code clones.

Classification of FOP-specific clones.

During our analysis, we proposed a classification of FOP-specific clones based on the relation of the affected features. However, we detected clones for which we can neither infer why these clones occur nor if they are FOP-specific indeed. One possibility is that these clones are contained in features that implement homogeneous crosscutting concerns. Since this kind of concerns occurs in OOP programs as well, the respective clones may be not purely FOP-specific. Beyond that, our condition for FOP-specific clones may lead to inaccurate results regarding the classification within respective analysis step. Hence, we should refine this condition to be more restrictive in our classification of what an FOP-specific clone is. However, we defined a lower bound with our definition of what an FOP-specific code clone is, which can be used as a base for future work.

5.6 Related Work

Many studies exist on code clones in object-oriented software systems. Some of them focus only on whether code clones exist or not (e.g., [Bak95, BYM$^+$98, Kri01]) whereas others analyze code clones with respect to their effects (e.g., [LLMZ06, MNK$^+$02]), their removal (e.g., [BMD$^+$00]) or other peculiarities such as identifying crosscutting concerns (e.g., [BDET05]). However, all of these studies are limited to OOP (and, to a minor fraction, functional programming). By contrast, our work focuses on clone detection and analysis of peculiarities of FOP and SPLs, which has not been considered so far. We open a new field for code clone research activity. Additionally, we related the causes for FOP-specific code clones to the limitations of FOP, which can initiate discussions on FOP language design.

Related work that focusses on general drawbacks of FOP exists as well. Kästner et al. addressed the problem of fine-grained extensions during SPL development regarding compositional and annotative implementation approaches [KAK08]. They figure out common problems of compositional approaches in the presence of fine-grained extensions, which can only be solved by tricky workarounds. As an alternative, they propose *virtual* separation of concerns, which overcomes the problems by supporting both, fine-grained *and* coarse-grained extensions.

Another problem that has been analyzed by Kästner et al. is the *optional feature problem* [KAuR$^+$09]. This problem is quite common during SPL development and describes the mismatch that two features that are optional in the *problem space* (e.g., in the variability model) depend on each other in their implementations (e.g., features *Logging* and *Transaction* in a DBMS). The authors analyzed common solutions, independent of the underlying programming language or implementation approach. Amongst others, multiple feature implementations is a solution, which inherently ears to code clones. Hence, while they not focussed on code cloning in general, they acknowledged that cloning is a solution to solve this problem. Additionally, they provide two case studies to provide insights on the impact of the optional feature problem and how it is generally solved.

Regarding clone detection or analysis in software product lines, only few related work exists. Chiba et al. recently proposed family polymorphism to counter clones in FOP [TC12]. In Particular, the reimplemented the MobileMedia SPL, added features and analyzed it with respect to code clones. Afterwards, they refactored the source code using SuperGluonJ, which provides inheritance between features. The main difference to our work is that we focussed on detailed insights of how and why code is cloned in feature-oriented SPLs. In contrast, they addressed the problem of how to overcome limitations of FOP that foster code cloning.

Mende et al. propose clone detection for supporting the evolution of SPLs [MBKM08]. However, in their work they consider SPLs, realized by object-oriented, preprocessor-based languages such as C++ and thus, the individual features are separated only *virtually*, i.e., by textual elements such as *#ifdef*. In our work, we consider feature-oriented SPLs where the features are separated into modules and we show that clone detection for such SPLs is applicable as well.

5.7 Summary

Code clones are a well-established and investigated field of research in software engineering. Different techniques for their detection and analyses to understand reasons for and impact of code clones exist. Though, it is far away from obvious how to treat clones. However, while variable software systems, also referred to as software product lines, gain more and more momentum, no work exist that investigates the occurrence of code clones amongst them. We addressed this issue for feature-oriented software product lines.

To this end, we presented a case study on code clones in feature-oriented software product lines. First, we formulated research questions regarding the causes and removal of code clones specific to feature-oriented SPLs. Particularly, we were interested whether clones occur in FOP, whether they are specific to FOP (and to what extent), and whether the development process of SPLs has an effect on code clones. Second, we conducted an empirical analysis on ten different SPLs to answer these questions. We presented our analysis process, which consists of the initial clone detection and two analysis steps to identify FOP-specific clones at a coarse-grained level. The results of our case study give strong evidence that a considerable amount of FOP-specific exist in our analyzed product lines. Moreover, we investigated that especially alternative features are a root cause for such clones and that the amount of clones is significantly higher in SPLs that are developed from scratch.

However, there still some questions we could not answer so far. Hence, we consider our case study rather as a starting point for a new direction of research for code clones. For instance, it is totally unclear whether our results are generalizable to other compositional approaches such as AOP. Moreover, we argue that is is worth to quantify more detailed which causes are crucial for FOP-specific clones and which are not. Finally, it is open how to asses and manage code clones in feature-oriented (or, more generally, compositional) software product lines. We pick up this question in Chapter 7, where we propose an approach for removing clones, which is one possible solution to that question.

6. Code Clones in CPP-Based Software Product Lines

In Chapter 5, we investigated the role of code clones in compositional software product lines, more precisely, in feature-oriented SPLs. However, in industrial applications, compositional product-line techniques gain only minor attention. Rather, annotative approaches are widely used to develop industrial software product lines, where the *C preprocessor (*CPP*)* is probably the most prominent and commonly used tool. The CPP is a powerful text processing tool tightly coupled with the C programming language [Ker88]. Due to its token-based nature, the cpp is language-independent and provides easy mechanisms to express variable source code. To this end, the CPP provides three different mechanisms: *macros* (`#define`), *file inclusion* (`#include`), and conditional compilation (e.g., by `#ifdef`). In the following, we solely focus on conditional compilation, because this mechanism is primarily used to express variability on source code level. Note that we use `#ifdef` as a placeholder for all possibilities of conditional inclusion (i.e.,`#ifndef, #if, #elif, #else` and `#endif`) and that we refer to all of them as *preprocessor directives* or *annotations*. Preprocessor annotations are discussed controversially due to the fact that they can be used in two ways: *disciplined* and *undisciplined*.

In this chapter, we investigate the effect of preprocessor usage on occurrences of code clones. Since a C/C++ program with conditional compilation can be considered as software product line (or variable software system) , we inherently provide insights of how and why clones occur in such preprocessor-based SPLs and how it may differ compared to feature-oriented SPLs. First, we take a closer look at preprocessor annotations and their distinction in disciplined and undisciplined ones. We present examples and possible advantages and disadvantages of both categories. Second, we present a code clones analysis process, which is the "skeleton" of our empirical analysis. Third, we discuss the results of our analysis on clones in CPP-based product lines.

With the results and discussions presented in this chapter, we extend existing studies by a large empirical study that investigates the relation between code clones and preprocessor annotations. Hence, we provide a new perspective on clones in C systems, which has not been considered before. As a result, we provide new insights into *why* code clones occur in annotative SPLs. Furthermore, the information of our analysis is useful to support the decision whether to remove clones or not.

6.1 A Discipline of Annotations

Preprocessor annotations, such as `#ifdef`, can be used to generate different variants (with different functionality) of a program [KAK08]. Each annotation consists of a boolean expression that is evaluated by the CPP tool to determine whether the conditional code is included in a certain program or not. Usually, such an expression represents a *feature*, an increment in user-visible functionality [LAL+10].

Because preprocessor annotations can be used at every level of granularity, they provide a high expressiveness to the programmer in terms of variability. Conversely, this flexibility makes it a root for poor code quality, caused by the missing structure of the CPP tool. Amongst others, the CPP is considered to be error prone and to impair readability and maintainability of the code [EBN02, Str95, Fav97, SC92]. Especially, if used in a fine-grained manner that does not align with the underlying program structure, it is commonly accepted that annotations contribute to unstructured, tangled source code with the mentioned negative effects [LKA11, EBN02, KAK08, KAT+09].

```
 1 class Stack {
 2  void push(Object o
 3 #ifdef SYNC
 4  , Transaction txn
 5 #endif
 6  ){
 7     if (o==null
 8 #ifdef SYNC
 9    || txn==null
10 #endif
11    )
12    return;
13 #ifdef SYNC
14    Lock l=txn.lock(o);
15 #endif
16    elementData[size++] = o;
17 #ifdef SYNC
18    l.unlock();
19 #endif
20    fireStackChanged();
21  }
22 }
```

(a) undisciplined

```
 1 class Stack {
 2 #ifdef SYNC
 3  void push(Object o, Transaction txn) {
 4    if (o==null || txn==null)
 5      return;
 6    Lock l = txn.lock(o);
 7    elementData[size++] = o;
 8    l.unlock();
 9    fireStackChanged();
10  }
11 #else
12  void push(Object o) {
13    if (o==null)
14      return;
15    elementData[size++] = o;
16    fireStackChanged();
17  }
18 #endif
19 }
```

(b) disciplined

Figure 6.1: Two examples for undisciplined and disciplined annotation usage.

In Figure 6.1, we show two code fragments containing conditional code that is undisciplined and disciplined, respectively. Undisciplined annotations (Figure 6.1 a, Lines

3–5 and 8–10) are made on arbitrary syntactical units, such as parameters or branch conditions, and do not align with the overall code structure. By contrast, disciplined annotations (Figure 6.1 b) are mapped to corresponding syntactical units such as functions or statements and thus align with the code structure. As a result, it is commonly accepted that disciplined annotations alleviate the drawbacks of annotations on source-code quality [EBN02, KA09, BM01]. However, we and others observed that disciplined annotations may lead to code clones [LKA11].

The distinction between disciplined and undisciplined annotations inevitably raises the question where we have to draw the borderline between both types of annotations. Distinguishing preprocessor annotations by their discipline has been originally proposed by Liebig et al., who provide the following definition [LKA11]:

Definition 3. *Disciplined Annotations*

In C, annotations on one or a sequence of ***entire functions*** *and* ***type definitions*** *(e.g.,* `struct`*) are disciplined. Furthermore, annotations on one or a sequence of* ***entire statements*** *and annotations on* ***elements inside type definitions*** *are disciplined. All other annotations are undisciplined.*

We use this definition within this paper because it is reasonable and suitable for our purposes. Furthermore, by this definition it is possible to map preprocessor annotations to elements of an AST. This, in turn, enables at least semi-automated analyses and refactoring that take the variability of the software system into account [LKA11]. In Figure 6.2, we provide further examples for both kinds of annotations. For instance, the listing in Figure 6.2 a contains annotations of alternative function parameter. This is obviously an undisciplined, both, due to fine granularity and our definition, and even though it is well-formatted it is obfuscating to some extent. For more details on drawbacks and advantages of disciplined as well as undisciplined annotations we refer to [LKA11].

6.2 Experimental Setup

In this section, we present the research questions, the overall design, and the subject systems of our empirical study on code clones in CPP-based SPLs.

Research Questions

Similarly to our empirical study in Chapter 5, we want to refine our initial research questions into more detailed questions, dedicated to the actual implementation techniques and its characteristics (here: preprocessor annotations). Another point is that beside the overall research questions, the correlation between the discipline of annotations and the amount of code clones is of special interest for us. Keeping both issues in mind, we formulate the following research questions.

```
  need_redraw = check_timestamps(
#ifdef FEAT_GUI
    gui.in_use
#else
    FALSE
#endif
  );
```

(a) annotated function parameter

```
  int n = NUM2INT(num);
#ifndef FEAT_WINDOWS
  w = curwin;
#else
  for (w = firstwin; w != null;
       w = w->w_next, --n)
#endif
    if (n == 0)
      return window_new(w);
```

(b) annotated head of `for`-loop

```
void tcl_end() {
#ifdef DYNAMIC_TCL
  if (hTclLib) {
    FreeLibrary(hTclLib);
    hTclLib = NULL;
  }
#endif
}
```

(c) annotated conditional branch

```
typedef struct {
  typebuf_T save_typebuf;
  int typebuf_valid;
  struct buffheader save_stuffbuff;
#if USE_INPUT_BUF
  char_u *save_inputbuf;
#endif
} tasave_T;
```

(d) annotated statement

Figure 6.2: Examples for undisciplined (a and b) and disciplined (c and d) annotations.

RQ 1 *To what extent do code clones occur in annotated* `#ifdef` *blocks?*

Several studies reveal the existence of code clones in C programs. However, none of these studies analyzed how much of the detected code clones occur in annotated code (i.e., code that is enclosed by preprocessor annotations). We aim at answering this question to better understand to what extent preprocessor usage causes code clones, independent of their discipline. Furthermore, we aim at investigating the relations of the variable blocks that contain the code clones such as `#ifdef ...#else ...` blocks, which correspond to alternative features in FOP.

RQ 2 *Are there differences between disciplined and undisciplined annotations regarding code clone occurrence?*

This question is motivated by the observation that disciplined annotations may come at the expense of introducing code clones [LKA11]. Consequently, we evaluate whether this observation is accidental or may depend on the discipline of annotations. Answering this question may also affect the evaluation of code clones regarding their harmfulness. For instance, if a code clone is introduced to overcome undisciplined annotations, should this clone considered harmful? As a direct consequence, this information may also support refactoring decisions in terms of code clone removal.

Study Design

We perform a code clone analysis supplemented by clone detection and source code analysis. The detailed process is described in Section 6.3. As a result, we collect information on the amount of code clones, *#ifdef code* (i.e., code that is annotated by `#ifdefs`), and *#ifdef clones* (i.e., code clones that are enclosed by `#ifdefs`). Then, we compute different measures based on these results.

First, we compute the code clone and *#ifdef* coverage. Coverage denotes the part of the source code that is covered by code clones or `#ifdef` blocks, respectively. These measures provide us with general information on the systems and whether it is worth to investigate these systems further.

Second, we compute the *#ifdef* clone coverage to investigate how much of the overall code contains *#ifdef* clones. Additionally, we compute the ratio of *#ifdef* clones compared to (a) all detected code clones (*#ifdef-clone/clone ratio*) and (b) the total amount of annotated code (*#ifdef-clone/#ifdef ratio*). With these measures, we can determine whether there are other factors that are likely to cause *#ifdef* clones.

Finally, we compute all measures for the subject systems, separated by their actual discipline of annotations and thus can compare both categories.

	undisciplined			**disciplined**	
program	# SLOC	description	program	# SLOC	description
CHEROKEE	47 983	Web server	BERKELEYDB	160 283	database system
GNUPLOT	67 854	plotting tool	DIA	121 117	diagramming software
LYNX	111 994	Web browser	GHOSTSCRIPT	491 703	postscript interpreter
PHP	471 604	program interpreter	LIGHTTPD	37 380	Web server
PRIVOXY	24 784	proxy server	MINIX	54 627	operating system
SENDMAIL	85 094	mail transfer agent	PARROT	84 222	virtual machine
TCL	122 460	program interpreter	PYTHON	331 014	program interpreter
VIM	233 426	text editor			

Table 6.1: Overview of analyzed C programs.

Subject Systems

For our case study, we use fifteen software systems written in the C programming language. We selected programs of different size and domains to have a representative sample. Furthermore, to evaluate our second research question, we split the sample in two comparable groups: Seven systems are categorized as disciplined and eight systems undisciplined. The criterium for the separation of our subject systems is the amount of undisciplined annotations compared to the overall amount of annotations. As a result of this criterium, the systems that we identified to be "undisciplined" contain 12 % undisciplined annotations on average. The classification is based on a recent study of Liebig et al., who analyzed the discipline of annotations in C programs [LKA11]. Based on this analysis, we defined a threshold of more than 10 % undisciplined annotations as inclusion criterion for undisciplined systems. Apart from that, both groups are comparable regarding size and domains. In Table 6.1, we give an overview of the analyzed programs.

6.3 Code Clone Analysis Process

In this section, we describe the overall process of analyzing the chosen software systems.The process consists of three steps, as we illustrate in Figure 6.3: *clone detection, source code analysis,* and *code clone analysis*. The first step provides information on code clone occurrences in form of a clone report. In the second step, we analyze the source code for occurrences of preprocessor annotations. Finally, we merge the information of step one and two to figure out where code clones and annotations overlap. Next, we provide a more precise description of each step.

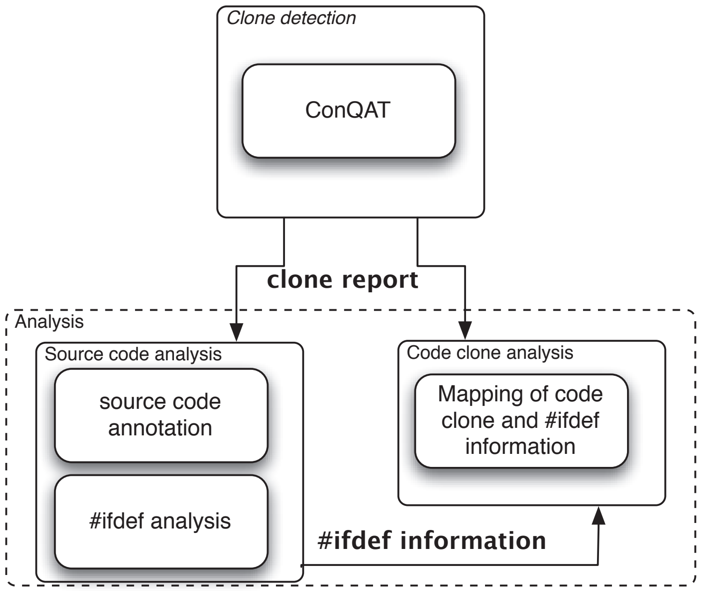


Figure 6.3: Overview of the code clone analysis process.

Clone Detection.

For clone detection, we use ConQAT[1], a token-based clone detection tool. We decided in favor of ConQAT (and against CCFinder) for two reasons. First, ConQAT can detect *Type-III* clones, which is important for our analysis, because such clones may occur within disciplined annotations, as indicated by our example in Figure 6.1. Second, ConQAT already merges corresponding clones to clone classes and, beyond that, provides visualization to gain first insights after the clone detection. To ensure comparability between our empirical study in Chapter 5 and in this chapter, we selected comparable settings. For instance, for both studies, we defined the same minimum clone length

[1]https://www.conqat.org

(that is, five lines) and enabled parameterization of certain syntactical elements. Furthermore, ConQAT as well as CCFinder use the token-based technique. However, for some parameters such as gap ratio, which can only be specified in ConQAT, we could not determine a counterpart.

Initially, the source code is transformed into a token sequence from which comments and white spaces are removed. Afterwards, ConQAT performs normalization on the token sequence, which can be divided into two parts. First, statements are created from the token sequence since it leads to better clone detection results. For instance, by merging tokens to statements, the detection tool ignores clones that begin or end in the middle of a statement. In a second normalization step, tokens are normalized by user-defined rules, which eliminates differences between the specified syntactical units such as identifiers or constants. For instance, we set up the normalization in such a way that differences between literals values of the same type (e.g. boolean or int) are ignored for the actual clone detection. However, we do not normalize differences between identifiers to preserve high precision.

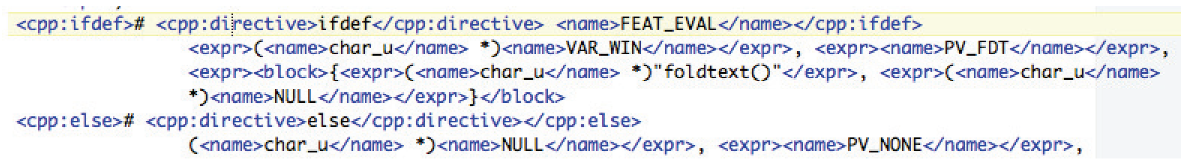
Finally, the clone detection is performed on the normalized token sequence. In a nutshell, a suffix tree is built on the token sequence and then, the algorithm searches for all identical or similar substrings in the tree. The user can influence the clone detection result by specifying different parameters such as the minimum clone length. For our purposes, we selected a minimum clone length of eight statements. Furthermore, we performed a gapped clone detection, so that gapped clones are detected as well. Therefore, we have to specify the *gap ratio*, a measure that determines the maximum number of gaps between two code clones. We selected a gap ratio of 0.25, which means for a clone pair of eight statements that two statements of at least one clone may have been deleted, added, or changed.

The result of the clone detection is subject to post-processing such as filtering out self-overlapping clone groups. Furthermore, remove clones that we detect in generated code. Finally, a clone report is generated, containing information on all source files as well as on all clone groups and its corresponding code clones, which can be used for further usage. For a more detailed description of the clone detection algorithm, we refer to [JDHW09].

Source Code Analysis.

With this step, we aim at extracting information about code fragments that are enclosed by preprocessor annotations in the source code. For obtaining this information, we have to analyze the source code that was subject to clone detection in the previous step. To this end, we annotate corresponding source files using *src2srcml*[2], a source code markup language that annotates the source code in an xml-like fashion without breaking its overall structure [MCM02]. We show an example in Figure 6.4.

[2]http://www.sdml.info/projects/srcml/



```
<cpp:ifdef># <cpp:directive>ifdef</cpp:directive> <name>FEAT_EVAL</name></cpp:ifdef>
                <expr>(<name>char_u</name> *)<name>VAR_WIN</name></expr>, <expr><name>PV_FDT</name></expr>,
                <expr><block>{<expr>(<name>char_u</name> *)"foldtext()"</expr>, <expr>(<name>char_u</name>
                *)<name>NULL</name></expr>}</block>
<cpp:else># <cpp:directive>else</cpp:directive></cpp:else>
                (<name>char_u</name> *)<name>NULL</name></expr>, <expr><name>PV_NONE</name></expr>,
```

Figure 6.4: Example for C source code, annotated with *src2srcml*.

Afterwards, we detect `#ifdef`s in the annotated code using *XPath*, an XML language that can be used to navigate through an XML document. As a result, we obtain all occurrences of `#ifdef` annotations, identified by their absolute position (i.e., line number) in the source file in form of a *preprocessor tree*. The tree structure has the advantage that nested `#ifdef blocks` occur as child nodes of their surrounding annotations in the tree. Note that we get this information for *complete* `#ifdef` blocks, that is, code fragments that are enclosed by annotations such as `#ifdef` and `#endif`. Finally, the results of this analysis can be used for code clone analysis.

Code Clone Analysis.

With our final analysis step, we aim at merging the information of the previous steps to detect clones that occur specifically in preprocessor annotations. Thus, we map the detected code clones to the detected `#ifdef` annotations, based on their absolute position in the source file. We illustrate the mapping algorithm in Figure 6.5.

```
proc $mapClones(cg, pa)$
Input: cg = list of clone groups, pa = list of annotations
for each $cg_i \in cg$ do
  get all code clones from $cg_i$
  for each $clone$ do
    $fa \subset pa$ = all annotations for the file containing the clone
    if $(a \in fa)$ is within $clone$ then
      $clone'$ = new clone with the position of $a$
      create new ifdef clone group with $clone'$ (if this is the first $clone$ of $cg_i$)
      otherwise add $clone'$ to existing ifdef clone group (for $cg_i$)
    end if
  end for
end for
Result: list of #ifdef clone groups
```

Figure 6.5: Algorithm for mapping annotations to code clones.

Our algorithm has two inputs: A list of all clone groups from the clone report and a list of preprocessor annotations together with the information where they can be found (i.e., the file and the line number) from the preprocessor tree. For the mapping, we consider the code clones of each clone group separately. Then, we compare the position of each clone with the positions of all preprocessor annotations that have been found in the file containing the clone. This results into three possible cases:

1. One or more `#ifdef` blocks are enclosed by the code clone (i.e., the code clone is larger than the preprocessor annotation).

2. The code clone is enclosed by the `#ifdef` block (i.e., the preprocessor annotation is larger than the clone)

3. Both, the code clone and the `#ifdef` block overlap only in parts.

For our purposes, only the first case represents a match and thus an *#ifdef clone*. We omit the second case, because it is possible that only a small part of a preprocessor annotation is a clone, which can also be caused by some idiomatic code. This poses the risk, that the code clone is meaningless. For the same reason, we also omit the third case. In the third case, incomplete clones such as statements at the end of a conditional branch may occur, which also pose a high risk to be meaningless or even incidental. For instance, suppose a code clones is identified in Lines 100–120 in File *foo.c*. Furthermore, a preprocessor annotation has been detected in Lines 115–123. As a result of our mapping algorithm, we detect that both, code clone and preprocessor annotation, overlap in Lines 115–120. However, since this is an arbitrary code fragment of both, the clone and the annotation, it is unlikely that it has been created intentionally. As a consequence, taking such partial *#ifdef* clones into account would lead to less accurate results.

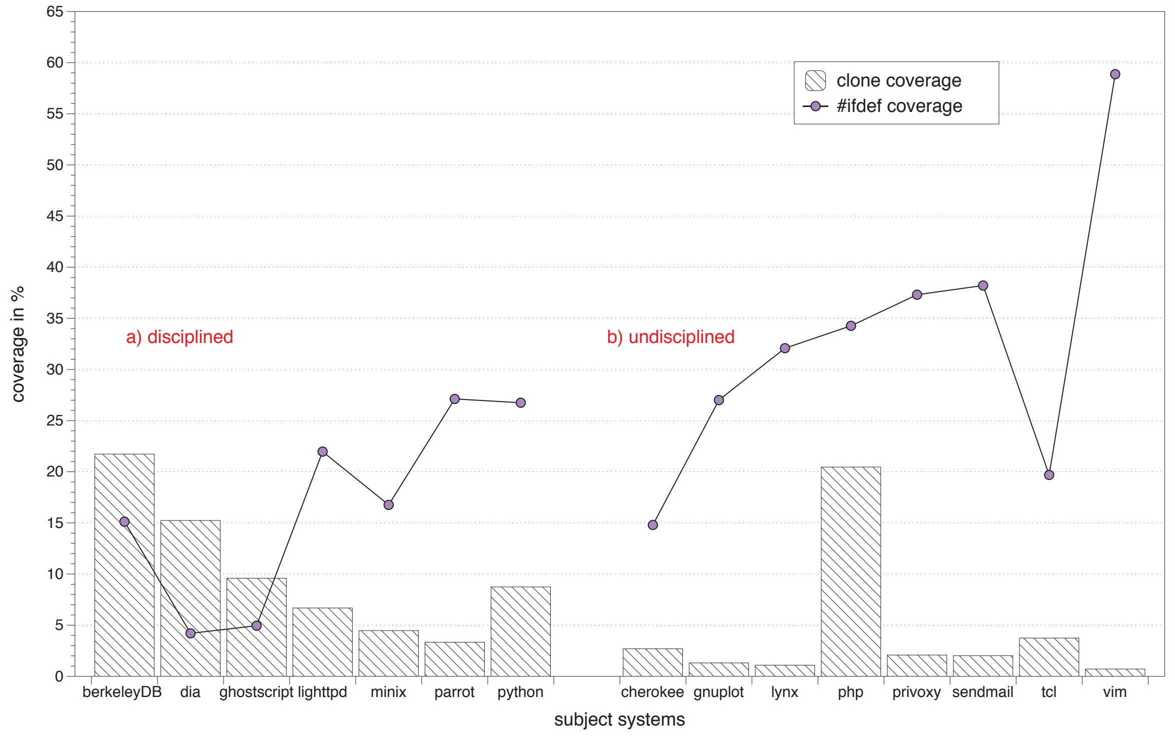


Figure 6.6: Clone and `#ifdef` coverage of the analyzed systems

Additionally to the aforementioned restrictions, the `#ifdef` block must have a length of at least five source lines of code, including the lines containing the annotations. This is threshold has been used by others [BDET05] and worked well for us, too. In case of a match, we create a new *#ifdef clone*. We do this for all clones of a clone group. Finally, all corresponding ifdef clones are merged to an *#ifdef clone group*.

6.4 Results

Next, we present the results of our empirical study. Besides presenting the data that we collected, we also relate different measures to investigate whether factors, other than the discipline of annotation, bias our results, primarily with regard to our second research question.

First of all, we present the results of step one (clone detection) and step two (#ifdef analysis). We observed that, in all systems, annotated code as well as code clones exist. In undisciplined systems (Figure 6.6, right side), the coverage of annotated code is 32, 7 %, on average, whereas it is 16, 7 % on average for the disciplined systems (Figure 6.6, left side). Furthermore, our data reveal that disciplined systems exhibit a considerably higher clone coverage (10 % on average, 8, 7 % on median), compared to the undisciplined systems (4, 3 % on average, 2 % on median). Most notably, we observed that six of eight undisciplined systems have a clone coverage less than 3 %. Nevertheless, all systems contain code clones as well as `#ifdef` annotated code in a reasonable amount and thus are further investigated.

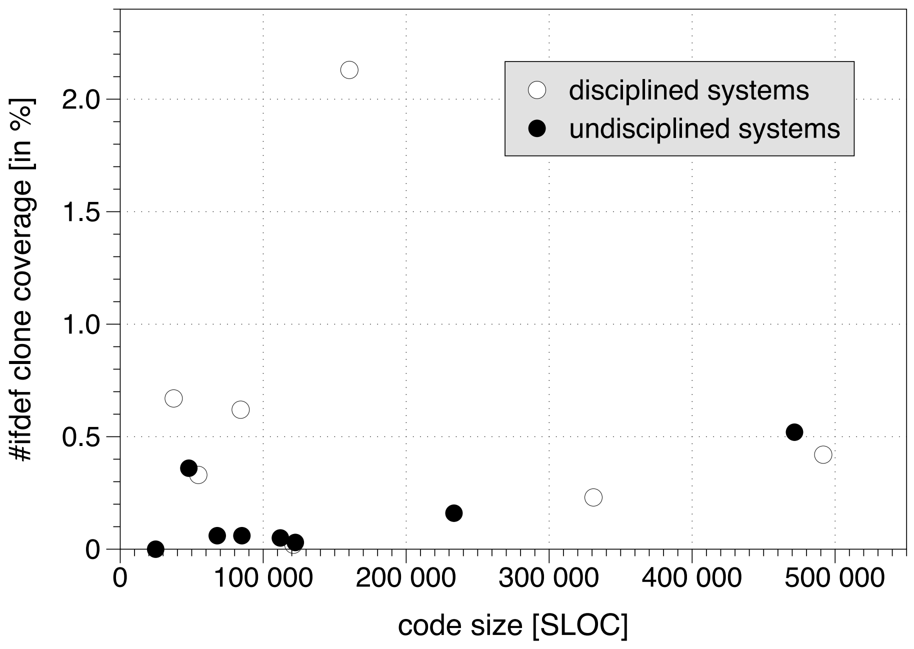


Figure 6.7: Analysis results for #ifdef clones in relation to code size (correlation coefficient: 0,64)

To evaluate RQ 1, we measured the *#ifdef* clone coverage for all our subject systems and show the results in Figure 6.7. The respective scatter plot indicates that only few code clones exist in annotated code. In fact, fourteen (out of fifteen) systems have an *#ifdef* clone coverage lower than 1 % and only four of them have a coverage greater than 0.5 %. Only one system (*BerkeleyDB*) has an *#ifdef* clone coverage higher than 2 %.

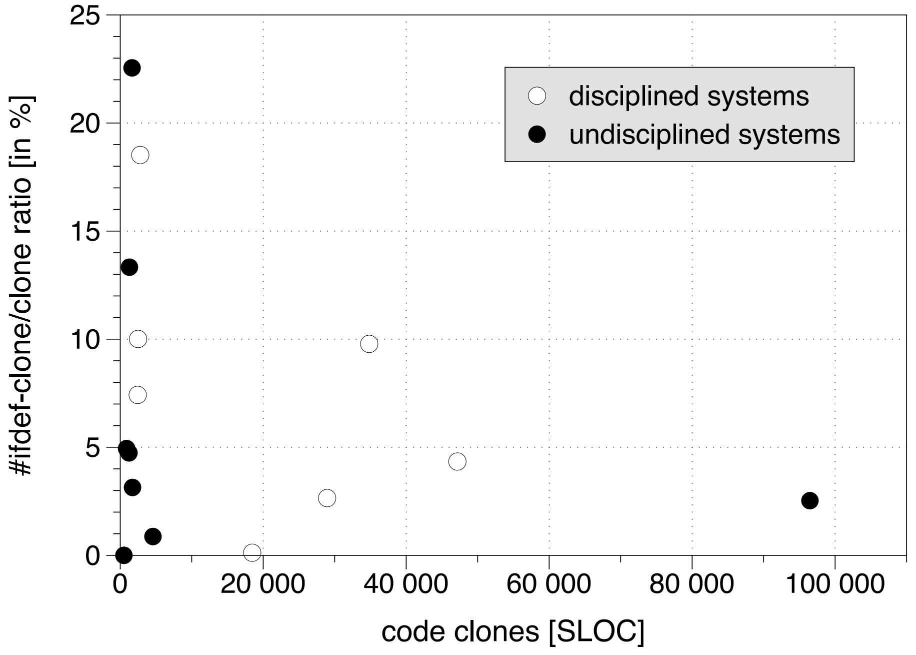


Figure 6.8: Analysis results for #ifdef clones in relation to total amount of clones (correlation coefficient: 0,77)

Because we observed considerable differences between disciplined and undisciplined system regarding overall code clone and *#ifdef* coverage (cf. Figure 6.6), we also computed further measures to evaluate whether certain correlations exist. First, we computed the *#ifdef*-clone/total-clone ratio and related it to the overall amount of code clones for each of the systems. With this measure, we wanted to know, whether a correlation exists between the total amount of clones and the actual clones that occur in preprocessor annotations. We show the result in Figure 6.8. Our data reveal, that only a minor fraction of all detected code clones occur within *#ifdef* blocks, independent of the actual amount of clones.

In the same way, we computed the *#ifdef*-clone/*#ifdef*-code ratio and related it to the total amount of annotated code. Our data, which we show in Figure 6.9, reveal that, similarly to the previous measure, only a small fraction of the overall *#ifdef* code contains code clones (Figure 6.9). Interestingly, we observed that systems with the highest amount of *#ifdef* code contain only few *#ifdef* clones. By contrast, amongst the systems with a rather small amount of *#ifdef* code ($< 25K$ SLOC), in particular, systems *#ifdef* code contains up to 14 % *#ifdef* clones.

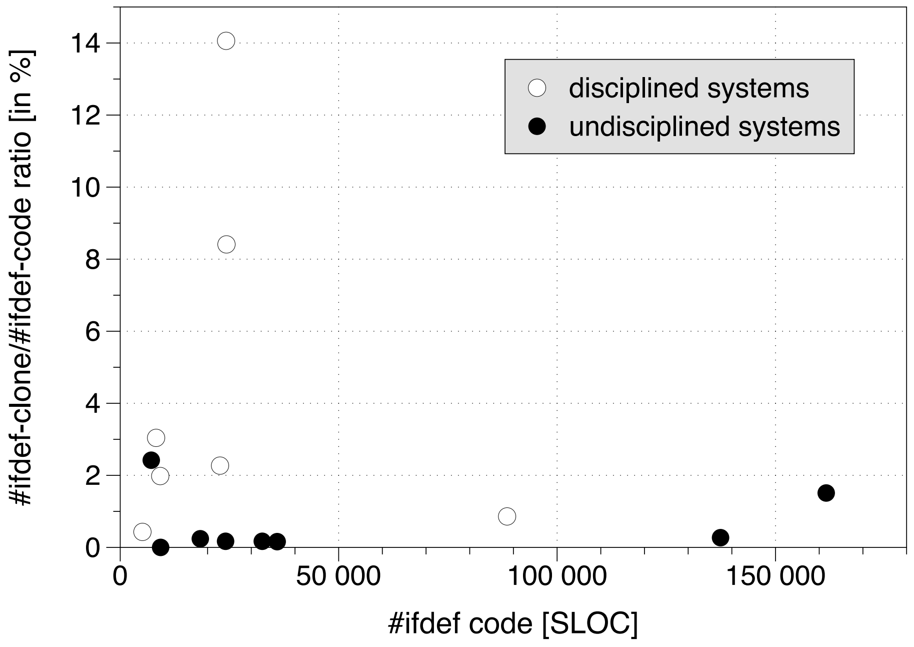


Figure 6.9: Analysis results for #ifdef clones in relation to #ifdef code (correlation coefficient: 0,34)

With RQ 2, we aim at investigating whether there are differences between disciplined and undisciplined systems regarding the amount of *#ifdef* clones. To this end, we compare the amount of *#ifdef* clones in disciplined and undisciplined systems. Although *#ifdef* clone coverage is rather low, the scatter plot in Figure 6.7 reveals that the amount of *#ifdef* clones is considerably higher in disciplined systems. We observed that five of the eight undisciplined systems have a coverage close to zero and the average *#ifdef* clone coverage is $0,15\,\%$. By contrast, four of the seven disciplined systems have an *#ifdef* clone coverage of approximately $0,5\,\%$ or higher ($0,63\,\%$ in average). Furthermore, we observed that annotated code in disciplined systems is more likely to contain clones than in undisciplined systems (cf. Figure 6.9). Although the latter systems contain more annotated code, in six systems this code contains nearly no code clones. In contrast, our data reveal that out of the six system that contain $2\,\%$ or more code clones in *#ifdef* blocks, five are disciplined systems. All of these observations are confirmed by a medium or even high correlation coefficient we computed, using the method of Pearson.

6.5 Discussion

We interpret and discuss the results with respect to the research questions, proposed in Section 6.2. Furthermore, we present and discuss some observations regarding the characteristics of *#ifdef clones*.

To what extent do code clones occur in annotated `#ifdef` blocks?

Based on our analysis and the respective results, we found that the amount of code clones in preprocessor annotation is rather small (with minor exceptions). This especially holds for the systems with a high amount of undisciplined annotations, where all measured data indicate that *#ifdef* clones are negligible. While this may render code clones in preprocessor annotations to be of minor interest, we found two reasons for this result.

First, due to our very strict definition of *#ifdef* clones (only complete annotated blocks are considered), this result can be interpreted as a lower bound. Consequently, a more fine-grained investigation of *#ifdef* clones, where we also consider code clones of the two cases we omitted so far (partial overlapping clones and annotations and annotations, which entirely surround detected clones) may lead to an increased amount of clones. However, we have to take care that we still obtain a good precision and thus define reasonable thresholds or definitions in which case we consider a code clone to be an *#ifdef* clone.

Second, a larger case study with more systems can also support us in investigating the role and nature of *#ifdef* clones in cpp-based programs.

Are there differences between disciplined and undisciplined annotations regarding code clone occurrence?

Our results indicate that *#ifdef* clone coverage is considerably higher for disciplined systems compared to undisciplined systems. This, in turn, confirms the assumption that disciplined annotations come at cost of code clones. Furthermore, our data reveal that other factors such as code size and amount of code clones or annotated code do not correlate with the amount of *#ifdef* clones.

To evaluate whether the observed differences between disciplined and undisciplined systems are significant or rather occurred randomly, we conducted a significance test. We applied an adapted version of the Mann-Whitney-U test: Instead of providing the significance level of the test, we check whether the calculated U values are significant according to a table specifically designed for small sample sizes [AF96, Giv08]. The results of the Mann-Whitney-U test reveal that the differences for the *#ifdef/#ifdef*-clone ratio as well as the *#ifdef* clone coverage are significant. First, for the *#ifdef/#ifdef*-clone ratio, we obtained the following results: $U = 6$, $p < 0.01$. Second, for the *#ifdef* clone coverage our test produced the following results: $U = 12$, $p < 0.05$. Since both significance levels are smaller than 0.05, we can assume that the differences we observed are significant and not caused randomly.

Revisiting the overall target of this empirical study, that is, investigating the effect of preprocessor annotations on code clones, our study results indicate that disciplined annotations increase the amount of code clones compared to undisciplined one. However, due to the small *#ifdef* clone coverage, the effects of these clones may be not as negative as for undisciplined annotations. Consequently, from our point of view, the benefits of disciplined annotations such as tool support [LKA11] outweigh the drawbacks of code cloning, especially considering the rather low amount of clones in annotated code.

Further observations

Beside analyzing clones from a quantitative point of view, we also reviewed the detected clones to find out more about their characteristics. In particular, we where interested how *#ifdef* clones disseminate over files and features and on which granularity they occur.

Regarding the features, which contain corresponding clones, we observed that nearly all clones of a clone class occur within the same feature, meaning that the preprocessor variable of the different #ifdef blocks is identical. This observation holds for both, disciplined and undisciplined systems. Furthermore, even if clones occur between different features, they had no obvious dependencies such as alternative relationships. Rather, we can consider these features as optional without any explicit dependencies regarding the preprocessor itself.

Beyond features, we also analyzed the dissemination of corresponding clones with respect to the source files. As a result, we made different observations; while clone classes (i.e., two or more corresponding clones) in undisciplined systems are located mostly in the same file, in disciplined systems such clones are mostly scattered across multiple files. Currently, we cannot explain why this is the case. However, together with our first observation regarding the features, the latter observation implicitly indicates that features are scattered across multiple files as well, maybe even more than in the undisciplined systems. This could be a possible, though not verified, reason for the differences we observed.

Finally, we observed that the detected *#ifdef* clones occur at any level of granularity, instead of at the block level only such as functions. Furthermore, the clones are rather short, mainly with a length of 5 to 10 lines of code. This may be caused by the fact that the CPP inherently supports fine granularity, even if used in a disciplined way. Although this makes it hard to reason about *why* these clones occur, we assume that cloning is tightly connected with the respective features (or preprocessor variables). That is, functionality is reused at different places but within the same feature, even at fine-grained level.

6.6 Threats to Validity

Single Programming Language.

Although the CPP tool is language-independent and thus used with several programming languages, we only considered C programs in our study, for three reasons. First, for the selected systems we had prior knowledge about the discipline of annotations, based on the study of Liebig et al. [LKA11]. Second, with the decision for one specific language, we can prevent that certain mechanisms of different languages influence our analysis results such as the amount of (*#ifdef*) code clones. Finally, the cpp tool has been used with C programs for a long time and thus the systems are mature and different case studies exist. Overall, we assume that this decision may limit generalizability, but does not affect our findings.

Selected Software Systems.

There is a risk that the selected systems bias the study results. To mitigate this effect, we selected systems of different size and of different domains as far as this was possible.

Clone Detection.

Both initial clone detection as well as *#ifdef* clone detection have been performed automatically, based on certain input parameters. Due to the large code amount, it is infeasible to check each clone regarding the precision of clone detection. However, we randomly selected samples (for code clones *and #ifdef* clones) from each subject system for a manual review process. All of these samples were true code clones, that is, we detected no false positives. Furthermore, for clone detection, we selected parameters that are commonly accepted to avoid that meaningless or even false code clones are detected. Beyond that, our mapping of clones to preprocessor annotations may influence our results. However, since we used a rather strict mapping, in which we omitted certain cases, we argue that the precision is quite high. Hence, we obtain a pessimistic yet reliable result for the amount of *#ifdef* clones.

Study evaluation.

During the interpretation of our results, we made several observations regarding our research questions. To strengthen our results, we computed statistical measures, namely Pearson's correlation coefficient and a significance test. The main problem of the statistical evaluation is the quite small sample set represented by the fifteen subject systems. Although the statistic results indicate the correctness of the relations we observed, we have to be careful with the conclusions we draw. Hence, a larger case study with more systems is necessary. Nevertheless, our study results provide first insights on the relation between preprocessor annotations and code clones.

6.7 Related Work

Analyzing CPP-based systems has been a major concern research from different perspectives, including general drawbacks of the CPP as well as specifically code clones. For instance, Ernst et al. conducted a large empirical study that investigates the usage of preprocessor annotations and its implications [EBN02]. In their work, they highlight advantages of disciplined preprocessor use and why this fails regularly. However, they mainly focused on the usage of macro definitions such as `#define`; conditional inclusion is just mentioned partly. Recently, Liebig et al. presented comprehensive results regarding preprocessor usage, on which we partially base our paper. Particularly, they analyzed preprocessor annotations in the context of software product lines by means of a large empirical study. First, they defined several metrics for measuring system properties such as granularity or types of extensions [LAL+10]. Second, they analyzed the discipline of annotations in cpp-based programs [LKA11]. In this context, they give a definition for disciplined annotations and conducted a case study on how disciplined and undisciplined annotations are used.

Amongst the several case studies on code clones, some of them specifically focus on clones in C programs. Initially, Mayrand et al. presented an experiment on function clones in C programs [MLM96]. Within their work they propose a taxonomy for function clones as well as different notions of similarity, based on metrics. Recently, Roy et al. conducted a large case study on code clones in open source systems (C & Java) [RC10]. They propose different metrics such as clone density, clone location, and clone size for comprehensive insights on cloned code and verified their results manually. However, both code clone analyses concentrate only on function blocks (and different metrics related to them) and thus they do not consider preprocessors, which is a new perspective of our work.

6.8 Summary

The C preprocessor is criticized to be responsible for different severe problems in cpp-based softwares systems such as introducing subtle errors, impede automated analysis, or increased effort for maintenance. Amongst others, the fine-grained usage conditional inclusion via `#ifdef`, also known as *undisciplined annotations*, is considered to be a main reason for the aforementioned problems. However, this is only the peak in an ongoing discussion on advantages and disadvantages of the CPP. In this chapter, we contribute to this discussion by analyzing to what extent the discipline of preprocessor annotations is responsible for code clones occurrences. To this end, we conducted an empirical study on fifteen open source C systems. First, we presented a clear distinction between disciplined and undisciplined annotations, taken from Liebig et al. [LKA11]. We related both kind of annotations to each other and provided examples for both categories.

Second, we provided information on our case study setup and the actual analysis process. Within the study setup, we formulated two research questions to guide our analysis and presented our subject systems. Next, we proposed an analysis process for our case study, which consists of three steps: clone detection, source code analysis, and code clone analysis. We provided detailed information for each of these steps. Particularly, we presented an algorithm for mapping code clones to preprocessor annotations.

Third, we presented and discussed the results of our analysis. We stated, that preprocessor annotations contain only a minor amount of code clones. Hence, we assume that preprocessor annotations may have only a minor effect on the amount of code clones. Beyond that, we observed significant differences between systems with disciplined and undisciplined annotations. Our results indicate that systems with disciplined annotations are more prone to code clones than systems with undisciplined annotations. Additionally, we observed that the detected clones do not follow a certain pattern, neither with respect to their granularity nor with respect to the features. Specifically regarding the affected source files, we investigated differences between undisciplined (mostly in one file) and disciplined (,mostly between multiple files) systems. Summarizing our results, we conclude that our study provides sound findings to verify the common belief that disciplined *#ifdefs* are prone to code clones.

However, due to the small amount of *#ifdef* clones, we argue that it is probably more beneficial to manage these clones instead of remove them (and probably introduce undisciplined annotations). To evaluate this claim, further studies are necessary to evaluate the harmfulness of these clones. Besides evaluating the harmfulness of *#ifdef* clones, there are more questions that remain unanswered: For what types of annotations do code clones occur in particular? Is there an overall relation between code clones and variability in cpp-based systems? How would our results change for different types of clones (e.g., only type-II clones or partial *#ifdef* clones) and more subject systems? In future work, we will focus on these questions to gain more insights on the relation between code clones and preprocessor annotations. Beyond that, we aim at providing a bigger picture of clones in the context of systems with high variability.

7. Removing Clones in Software Product Lines

In the previous two chapters, we presented case studies to analyze code clones in feature-oriented and preprocessor-based software product lines, respectively. Although we observed differences between the underlying implementation approaches (compositional vs. annotative) within our analysis, we detected code clones in both of them. Especially in feature-oriented SPLs, a considerable amount of clones occur. However, it is still open how to proceed with the detected clones. In Chapter 2, we discussed two possibilities: *code clone removal* and *code clone controlling*. In this chapter, we focus on the former.

A common approach to remove code clones is the application of refactorings. Initially, refactoring has been defined by Opdyke as a maintenance process that changes the internal structure of a program but does not affect its external behavior [Opd92]. Based on this work, Fowler presented numerous refactorings in a catalogue-like manner [Fow00]. While these refactorings are mainly designed for object-oriented programming, their applicability is limited for software product lines. Hence, we and other argue that refactoring in SPLs goes beyond traditional (object-oriented) refactoring approaches and thus, has to be revisited for product lines [BTG10, KKAS11, STKS12].

In this chapter, we address this gap in research by proposing an extension of existing refactoring techniques, specifically for *feature-oriented programming (FOP)*, a compositional approach for product line development. While we mainly focus on code clone removal as application scenario in this chapter, even more use cases for refactoring in SPLs exist. Particularly, we argue that SPLs evolve even more than stand-alone programs, and thus, maintenance of SPLs is an important issue. For instance, new or changed requirements may induce a change to the feature model or source code of an SPL. Furthermore, poor code quality caused by improper design decisions may require changes to the SPL. Commonly, most of these changes require an appropriate refactoring support.

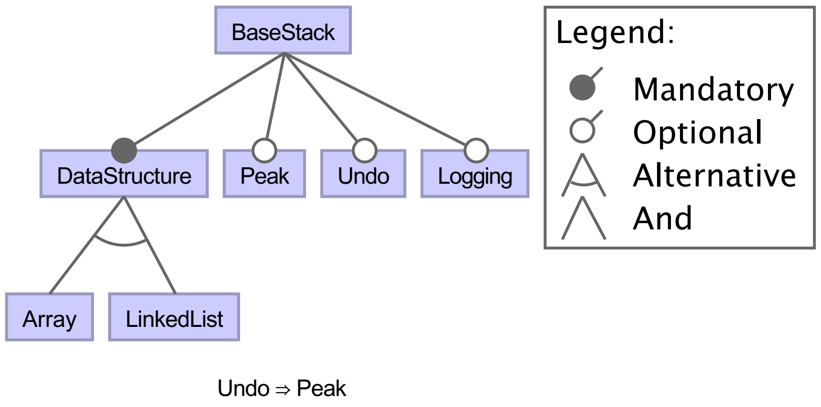


Figure 7.1: A feature model specifying valid feature combinations of the Stack product line.

Overall, we make the following contributions within this chapter:

- We point out limitations of traditional object-oriented refactorings for SPLs.
- We propose a definition for refactoring of SPLs in a *variant-preserving* way.
- We provide exemplary refactorings for feature-oriented SPLs in a catalogue-like manner.
- We provide a case study on the *TankWar* SPL, where we remove detected clones by applying our feature-oriented refactorings.
- Additionally, we discuss the generalizability of our definition and the actual refactoring towards annotative SPL implementation techniques.

7.1 Limitations of Traditional Refactoring

In this section, we provide information about the limitations of traditional (object-oriented) refactoring regarding its application in feature-oriented software product lines. To this end, we review the relation between features and collaborations and point out how this affects refactoring of SPLs. Moreover, we present the different dimensions and corresponding challenges that result from feature semantics. To illustrate the main facts, we provide examples using the Stack SPL. As a reminder, we show the feature model of this SPL, already known from Chapter 3, in Figure 7.1.

7.1.1 Features and Collaborations

Implementing a software product line with FOP is mainly characterized by decomposing a program into modular implementation units. The core idea of this decomposition is that all artifacts (e.g., source files, config files) that belong to a certain feature are

Feature *BaseStack*

```java
class Stack { ...
    void push(Element e) {/*...*/}
    int pop() {/*...*/}
}
```

Feature *Array*

```java
class Stack {
    ArrayList stack = new ArrayList();

    void push(Element e) {
        stack.add(0,e);
    }
    /*...further implementation...*/
}
class Element {
    /* initial implementation of class Element*/
}
```

Feature *LinkedList*

```java
class Stack {
    LinkedList stack = new LinkedList();

    void push(Element e) {
        stack.push(e);
    }
}

class Node {
    /* ...further implementation...*/
}
class Element {
    /* implementation of class Element*/
}
```

Figure 7.2: Feature-oriented implementation of the Stack product line containing features *BaseStack, Array,* and *LinkedList*.

modularized into one cohesive unit. This unit is called a *feature module* and we can map this module directly to its corresponding feature in the feature model. Afterwards, selected features are composed to generate a certain variant of the SPL. In this context, each features realizes an increment in functionality by adding new structures to the current program such as classes or extend existing structures such as classes or methods.

In Figure 7.2, we show an exemplary feature-oriented implementation for our Stack SPL. Feature *BaseStack* is the base implementation of the SPL where class `Stack` is initially declared. Moreover, features *Array* and *LinkedList* realize alternative data structures. Both features extend the functionality by overriding the original method `push` and adding the class `Element`. Additionally, the class `Node` is initially declared in feature *LinkedList*.

An important characteristic of FOP is the relation between features and classes. Generally, a set of classes contributes to the implementation of features. These classes have two interesting attributes: First, they communicate and thus have a *collaborating* nature. Second, they can be considered to be *orthogonal* with respect to feature modules. Due to this relation(s) between features and classes, FOP is technically similar to *collaboration-based design* [SB02, VN96]. In fact, we can consider feature modules to be a specific representation of collaborations.

In the context of collaborations, classes play different *roles* in different feature modules. That is, all functionality that a class contributes to a certain feature is encapsulated by the role in the corresponding feature module. We illustrate this relation by an exemplary *collaboration diagram* of our Stack SPL, which corresponds to the code listings in Figure 7.2. The diagram, which we show in Figure 7.3, consists of three different classes (`Stack, Node, Element`) and features (*BaseStack, Array, LinkedList*). Each class can play a role with respect to a certain feature or not. In FOP, a role is characterized as an increment in functionality such as extending or adding a method. For instance, class `Stack` has a role for each feature of the diagram (*BStack, AStack, LL-Stack*). In contrast, class `Element` has only two roles (*AElem, LLElem*) in features *Array* and *LinkedList*, where it reimplements the method `push`, respectively. Likewise, in the exemplary implementation of our Stack SPL, class `Node` has a role in feature *LinkedList*, where the initial implementation of this class takes place and thus extends the functionality (cf. Figure 7.2).

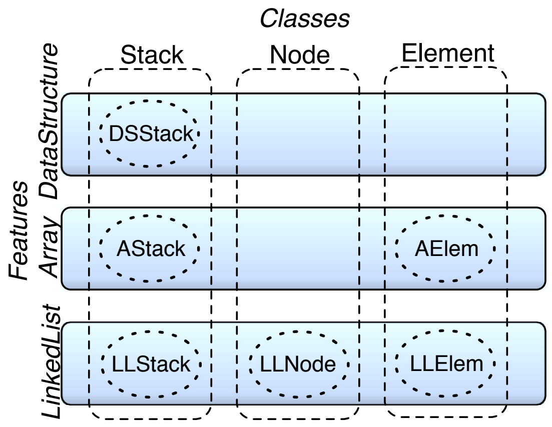


Figure 7.3: Collaboration diagram of the stack SPL with classes `Stack, Element,` and `Node` features *BaseStack, Array,* and *LinkedList* (excerpt).

7.1.2 A New Dimension of Refactoring

Next, we figure out how collaboration-based design affects refactoring of feature-oriented software product lines. Originally, refactoring is applied for the purpose of improving the structure of source code. This, in turn, requires at least a vague idea of what makes source code good or bad. For object-oriented programs, Fowler proposed a comprehensive overview of *code smells* (e.g., design flaws, replicated code) and how to remove these smells by application of refactorings [Fow00]. Although such an overview of code smells does not exist for FOP, many situations such as evolution or increasing maintainability require refactoring. Particularly, we argue that we can even adopt code smells for FOP as in traditional refactoring. For instance, there may be a method in one feature that is more frequently used by another feature. Hence, it would be useful to move this method into the latter feature.

Unfortunately, it is not as easy to adopt object-oriented refactoring techniques for feature-oriented SPLs[1]. One reason is the aforementioned relation between classes and features in FOP, which introduces an additional dimension for refactoring. In Figure 7.4, we show the resulting dimensions that have to be considered for refactoring. Each dimension implies certain challenges and risks, which we explain in the following.

[1]We use the terms feature-oriented program and feature-oriented SPL synonymously throughout this paper.

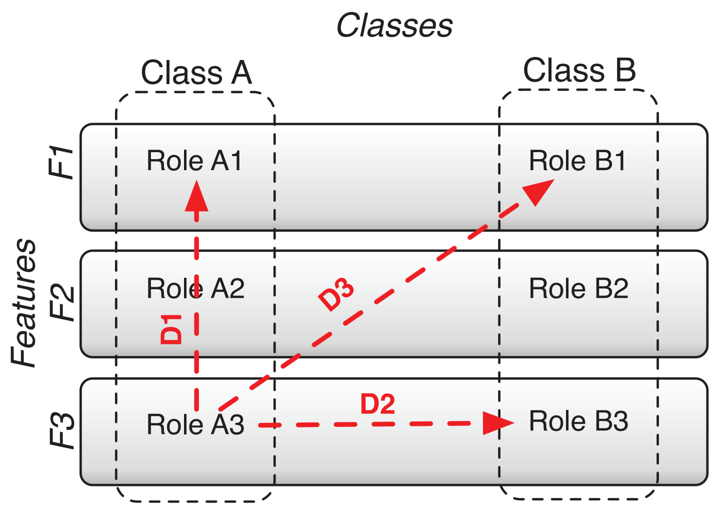


Figure 7.4: Different dimensions of refactoring in feature-oriented programming.

D1 – Intra-Class, Inter-Module

For this dimension of refactoring, we have to take different features into account (here: *F1,F2,F3*). Furthermore, the class that is involved in refactoring, is decomposed and represented by different roles, according to the respective features. Both, multiple features and roles, have an effect on the refactoring process. First and foremost, we have to consider feature dependencies and constraints and how they affect certain refactorings. For instance, in our Stack SPL, the features *Array* and *LinkedList* are alternative. If we would now move a method of class `Element` from *Array* to *LinkedList*, all variants with the feature *LinkedList* retain their behavior. However, the method `push` is not available in variants that contains feature *Array*. This, in turn, may change the behavior of these variants or even lead to serious errors such as dangling references.

Furthermore, we have to deal with the different roles when applying refactorings. For instance, if we rename a method or field of one role, we have to ensure that we rename corresponding syntactical element in other roles as well. This, in turn, requires knowledge whether corresponding elements exist between roles. One possibility to obtain that knowledge is to temporarily compose the affected roles to one class (e.g., roles *A1, A2, A3* to class `A`, cf. Figure 7.4), gathering the knowledge from the composed class and use it on the actual implementation level. While this approach is similar to *Refactoring Feature Modules (RFM)* [KBA09], it is tedious and time-consuming and thus not applicable at all. Recently, a native, feature-oriented compiler called FUJI has been proposed, which provides an AST for complete features that can be used to obtain the required information.

D2 – Inter-Class, Intra-Module

This dimension covers refactorings that (physically) take place within one feature, but may affect different classes (and roles implicitly). Consequently, we don't have to consider feature dependencies and constraints necessarily, as in dimension D1. Nevertheless, we have to take other features, or more precisely, certain roles in other features, into account for two cases. First, if a refactoring (e.g., renaming) affects only one class (and thus one role), we face the same challenges as with dimension D1, mentioned above. Second, a refactoring may affect different roles in *different* classes, which is a more complex and non-trivial task. For instance, assume that we want to move a method `foo` from role *A3* to role *B3*. Additionally, this method refines the original method `foo`, declared in role *A1*. As a result of the supposed refactoring, two problems may occur. First, feature *F1* is not part of a certain variant and thus the method `foo` is not initially declared and cannot be refined by our refactored method. Second, even if feature *F1* exists, we have to add some kind of delegation to the refactored method `foo` in over to point to the original method. Furthermore, other methods that call the refactored method have to be updated as well. All these problems may cause serious errors or make such a refactoring even inapplicable.

D3 – Inter-Class, Inter-Module

The refactorings of this dimension are probably the most complex ones, because they combine the challenges and problems of the aforementioned dimensions. In particular, we have to consider feature dependencies and constraints, because multiple features may be affected by the refactoring process. Furthermore, it is possible that different roles, even across different classes, are involved in a certain refactoring.

According to traditional refactoring, where the preservation of the external behavior is a mandatory requirement, for refactoring in FOP, preserving the variability as well as the unchanged behavior of each variant is mandatory. We comprise both aspects within the term *variant-preserving* refactoring, which we introduce in the next section.

7.2 A Notion of Variant-Preserving Refactoring

In this section, we give a concrete definition for the term variant-preserving refactoring. Furthermore, we introduce some specific aspects that have to be considered for refactoring in FOP.

7.2.1 Definition

As indicated in the previous section, refactoring of feature-oriented programs can involve two parts of the product line. First, the feature model that reflects the domain knowledge (*problem space*), including feature dependencies, and, second, the source code that implements the functionality of the SPL (*solution space*). We define a *variant-preserving refactoring* as follows:

Definition 7.1. *A change to the feature model or the implementation of features or both is called* **variant-preserving refactoring** *if the following two conditions hold:*

1. *Each valid combination of features remains valid after the refactoring, whereas the validity is specified by the feature model.*

2. *Each valid combination of features that was compilable before can still be compiled and has the same external behavior after the refactoring.*

With the first condition, we address refactorings that are applied to the feature model. In particular, this condition ensures that all combinations of features specified as valid *before* any refactoring activity (before-edit version) of the feature model are also valid after one ore more refactorings (after-edit version). Note that by our definition new variants are allowed as long as they do not affect existing feature combinations. For instance, assume that we extend our Stack SPL by adding an optional feature *Sync*, which manages concurrent access on the stack (cf. Figure 7.5). AS a result, the number of possible variants, which we can generate from this SPL, has bee doubled (from 12 to 24). Nevertheless, this refactoring is variant-preserving, because all feature combinations (i.e., variants) that have been valid before the refactoring are still valid after the refactoring. With the second condition, we address refactorings that operate on the implementation of features. Following our definition, such refactorings must not lead to changed behavior of any program of the SPL.

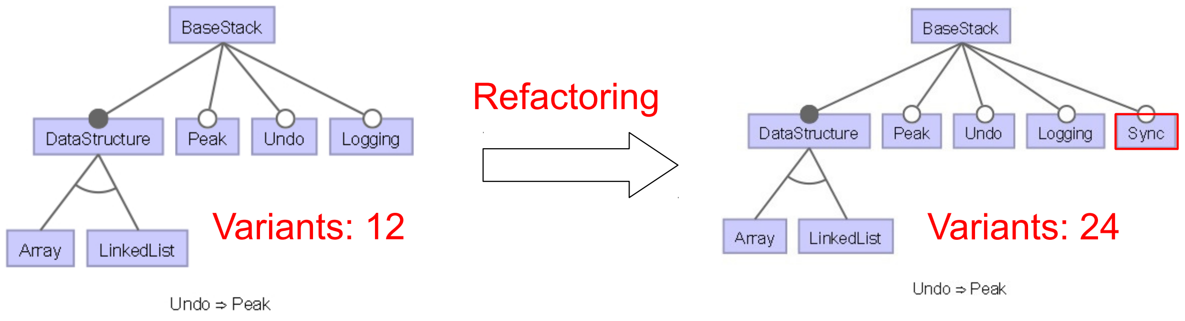


Figure 7.5: Example for feature model refactoring by adding an optional feature

In the remainder of this chapter, we focus mainly on the second condition, that is, source code refactorings. For a more comprehensive overview of feature model refactorings, we refer to existing work such as [TBK09, BTG10].

7.2.2 Aspects of Refactoring in FOP

Although the definition of variant-preserving refactoring given above is sufficient to cope with the specific FOP characteristics, we identified three specific aspects that require additional treatment. In the following, we explain these cases and discuss, how they affect the application of refactoring in FOP.

Root Feature as a Target.

The first case we consider is the refactoring of a code fragment to the root feature (e.g., feature *BaseStack* in Figure 7.1) of a feature-oriented SPL. Theoretically, using this feature as the target feature for a certain refactoring is often possible, because the root feature is selected per default for each variant of the product line. However, in practice this procedure comes at costs of intrinsic implications.

First, using the root feature as target feature decreases the cohesion of the source code that implements a certain feature. Typically, FOP aims at implementing a certain feature in a cohesive unit, i.e., the corresponding feature module. However, taking off code fragments from a feature module (and moving it to the root feature) breaks with the modular implementation and thus may decrease the cohesion within the feature module. Specifically, this refactoring is a kind of generalization, because the refactored code is moved to the most general (and meaningless) feature of the SPL.

Furthermore, preserving the variability can be difficult in certain cases if the root feature is the target feature for refactoring due to possible conflicts of existing and refactored code fragments.

Intra- vs. Inter-Feature refactoring.

Another aspect that must be considered thoroughly are the features that are involved in the actual refactoring process. We differentiate between two kinds of refactoring: First, if the refactoring affects more than one feature we call this an *inter-feature refactoring*. Amongst others, moving a method from one feature to another one is an example for such a kind of refactoring. Second, if the refactoring affects only one or more classes of the same feature, we call this an *intra-feature refactoring*. For that kind of refactoring, extracting a method (affects one class) or moving a method between classes in *one* feature are exemplary refactorings. As a result of this distinction, we can establish a relation between the different possible dimension of refactoring in Figure 7.4 and the two kinds of refactoring mentioned above: While inter-feature refactoring takes place along the dimensions *D1* and *D3*, intra-feature refactoring takes place along dimension *D2* (including refactoring within one class).

This has the following implications: For inter-feature refactoring, the programmer has to ensure that the actual refactoring does not violate the conditions of Definition 7.1. Consequently, if we want to restructure the SPL across feature boundaries, we have to apply FOP-specific refactorings (as introduced in the following section) to preserve the variability of the SPL.

For intra-feature refactoring, we can apply the traditional, object-oriented refactorings and add only additional checks (taking all features into account) after the refactoring to detect and update incorrect references. We can do this, because intra-feature refactoring *does not* affect the variability directly and thus is inherently variant-preserving. Nevertheless, we have to take feature semantics into account to some extent. First, if we check the pre-conditions (i.e., conditions that have to be fulfilled to apply a refactoring), we must take into account the relation between features or roles to decide whether the refactoring is applicable or not, respectively. Second, after the application of an intra-feature refactoring (using traditional refactoring techniques), we possibly have to update references (to fields or methods) in more features than the target feature. For instance, assume that we want to move a method `foo()` from class `A` to a class `B` for Feature *F3* in our example in Figure 7.4. Consequently, we can apply the *Move Method* refactoring proposed by Fowler [Fow00]. In case that the corresponding role *B3* does not exist, we have to introduce it, that is, creating the class `B` for Feature *F3*. Additionally, we may have to update references to the original method in class `A` to the new class `B` to avoid dangling method references.

Combining source code and (feature) model refactoring

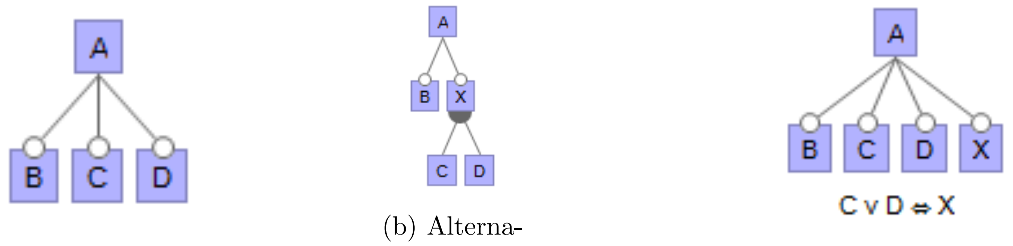
One characteristic of FOP is the physical separation of concerns into feature modules and a concrete mapping of these modules to features of the feature model. Due to this tight connection, we may face situations, where a refactoring is suitable that affects both, the source code *and* the feature model. In such a case, we have to ensure that both conditions of our definition are fulfilled. In the following we address this aspect by the example of code clone removal. A general pattern to do this, is that we replace the replicated code in multiple locations by a single reusable code fragment.

For example, in the simple case that an SPL always requires one of two alternative features, and both features introduce the same method, then we can remove all cloned instances of the method and introduce it only once in the root feature. With this modification, we eliminate cloning and the method is always available from the root feature. Obviously, we cannot move every cloned code fragment into the root feature. If it is valid to select none of the features containing cloned code, moving code to the root feature would bloat the code base of variants in that none of these features is selected. Additionally, it can be considered as violation of separation of concerns.

A general solution is to move cloned code into a newly created feature that is selected *if and only if* at least one of the features containing cloned code is selected. Consider the feature model in Figure 7.6 (a) and assume that some code between features C and D is cloned. In this case, we could create a new parent feature X for C and D and move the cloned code there as illustrated in Figure 7.6 (b). Alternatively, we can create a new feature X somewhere else in the feature model and use a cross-tree constraint (X equals C or D) to enforce the previous semantics as in Figure 7.6 (c). Of course, we can also search the feature model for existing features that would meet the condition, instead of creating a new one. Note that both transformations of the feature model preserve all existing variants and do not create new variants (called feature model refactoring) [TBK09].

The pattern of moving cloned code to a single new location works uniformly for different kind of clones: cloned types, cloned methods... etc. However, sometimes alternative for refactoring may be available that do not affect the feature model. While it is an open issue how to decide in such a case, which refactoring is more suitable, we argue, that it may be less complex if a refactoring affects only one space.

As conclusion, we argue that refactoring feature-oriented SPLs includes both, FOP-specific as well as traditional refactorings, depending on the features involved in the



(a) Original

(b) Alternative #1

(c) Alternative #2

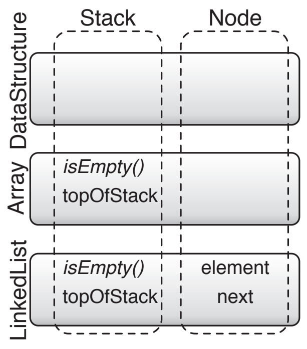
Figure 7.6: Exemplary feature model edits for code clone removal

refactoring process. Furthermore, in specific cases, the solution as well as implementation space is affected by variant-preserving refactoring. The challenging task is to identify which refactoring approach to chose in which situation and how to deal with additional mechanics that go beyond traditional refactorings.

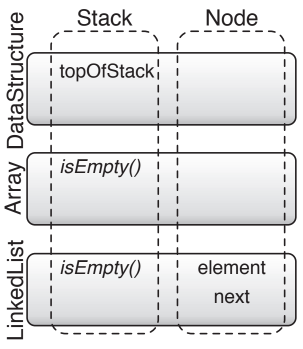
7.3 A Catalogue for FOP Refactoring

In this section, we present four refactorings for feature-oriented software product lines in a catalog-like manner. Note that we focus on inter-feature refactorings (i.e., dimensions D1 and D3 in Figure 7.4) within our catalogue, because these refactorings require considerable changes with respect to the original object-oriented refactorings. Consequently, the presented refactorings are rather extensions of the object-oriented ones proposed by Fowler [Fow00] by taking feature semantic into account. Furthermore, we present the FOP refactorings in the same way as Fowler so that programmers can easily understand their mechanics and application (assuming that they have knowledge of the original refactorings). We are also inspired by Monteiro, who did this in a similar way for aspect-oriented programming [MF05]. Finally, we introduce the terminology we use for the description of refactorings. First, a *target feature* refers to a feature that is target of a refactoring, for example, where a method is moved to. Likewise, a *source feature* in the context of refactorings is a feature, where the code fragments, which are subject to refactoring, are originally located. Third, we refer to program elements, that are subject to refactoring as *candidates*. For instance, a *candidate field* is a field that is subject to refactoring.

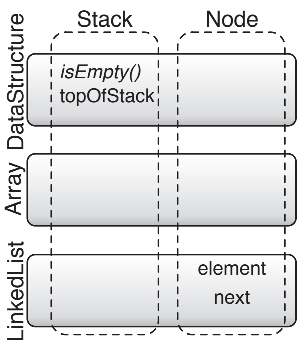
The first refactorings we present are an adaptation of the *Pull Up Field* and *Pull Up Method* refactoring of Fowler [Fow00, p. 320 and p. 322, respectively]. Originally, these refactoring describe how to move methods or fields between subclasses and its superclass by exploiting the *inheritance hierarchy*. In contrast, for FOP we consider features as source and target of the refactoring rather than classes (although the latter are also taken into account). Consequently, we exploit the *refinement hierarchy* instead of inheritance hierarchy for the application of these refactorings.



(a) Before refactoring



(b) After *Pull Up Field*



(c) After *Pull Up Method*

Figure 7.7: Collaboration diagram with features *DataStructure, Array,* and *LinkedList* (a) before refactoring, (b) after *Pull Up Field* refactoring, and (c) after *Pull Up Method* refactoring. On the right side, we show excerpts of the implementation of features *Array, LinkedList* (before refactoring) and feature *DataStructure* (after refactoring). The refactored code fragments are marked by an dotted rectangle.

7.3.1 Pull Up Field to Parent Feature

Typical situation: A feature has sub features, that are similar to some extent. Specifically, certain fields could be duplicates. Hence, the programmer can reduce duplication by generalizing these fields.

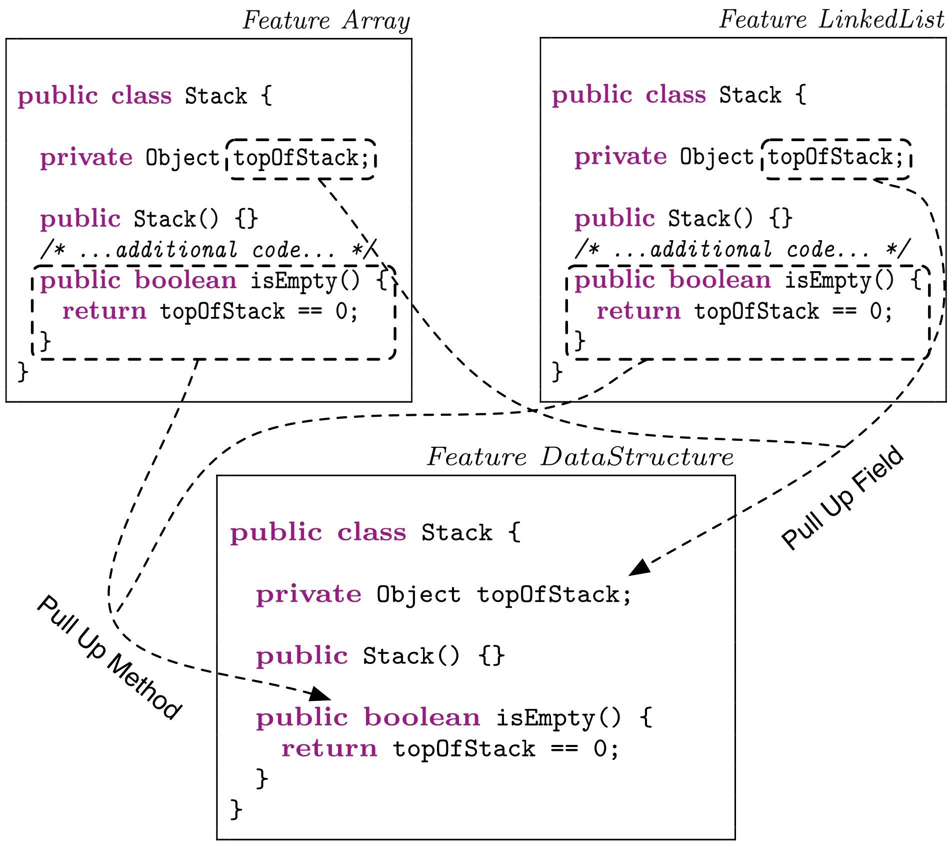
Recommended action: Move fields that are identical (e.g., type and initialization) across sub features and used in the same way in the respective features (e.g., referenced by the same methods) to the common parent feature.

Pre-conditions:

- The features, containing the field(s) that are subject to refactoring, must have the same, direct parent feature.
- The respective fields must reside in the same class (e.g., class `Stack` in our example)

Mechanics:

1. Check whether the candidate fields are used in the same way (e.g., qualifier).
2. Check whether the pre-conditions, mentioned above, are fulfilled.
3. In the case, that the class containing the candidate fields has no role in the target feature (i.e., is not implemented in the target feature), create the class for the target feature.
4. If the candidate fields have different names, rename the respective fields so that they all have the name that should appear after the refactoring in the target feature.
5. Create a new field in the respective class of the target feature.



```
public class Stack {

  private Object topOfStack;

  public Stack() {}
  /* ...additional code... */
  public boolean isEmpty() {
    return topOfStack == 0;
  }
}
```

```
public class Stack {

  private Object topOfStack;

  public Stack() {}
  /* ...additional code... */
  public boolean isEmpty() {
    return topOfStack == 0;
  }
}
```

```
public class Stack {

  private Object topOfStack;

  public Stack() {}

  public boolean isEmpty() {
    return topOfStack == 0;
  }
}
```

Figure 7.8: Excerpts of the implementation of features *Array, LinkedList* (before refactoring) and feature *DataStructure* (after refactoring). The refactored code fragments are marked by an dotted rectangle.

6. If the original fields were private and the refactoring is applied along dimension D3 (cf. Figure 7.4), we have to change the access modifier of the new field to `protected` so that the sub features can access it.

Example: In Figure 7.7 and Figure 7.8, we illustrate this refactoring with a collaboration diagram and provide an exemplary implementation of our Stack SPL, respectively. The two alternative features *Array* and *LinkedList* each contain a field `topOfStack` in class `Stack` that represents the first element of the stack. Both, the candidate fields as well as the source features fulfill the (pre-)conditions. Hence, we can apply the pull up field refactoring as described above with feature *DataStructure* as the target feature (cf. Figure 7.1). First, we have to introduce a role for class `Stack` in the target feature by creating this class in the respective feature. Then, we create a new field in this newly created class. Afterwards, we delete the original fields in feature *Array* and *LinkedList*. In Figure 7.8, we show the corresponding code fragments for the features involved in the refactoring. Since the refactoring takes place along dimension D1, we do not have to change the access modifier of the field. The reason is that we move the field to another feature but it remains within the same class `Stack`. Hence, it is accessible within the target feature after composition, even though it is private. For a comprehensive overview of access modifiers in feature-oriented programming, we refer to the work of Apel et al. [AKL+12].

7.3.2 Pull Up Method to Parent Feature

Typical situation: Methods that are (semantically or syntactically) identical occur in different (sibling) features.

Recommended action: Move the candidate method(s) up to the parent feature of the two source features, which contain the identical method.

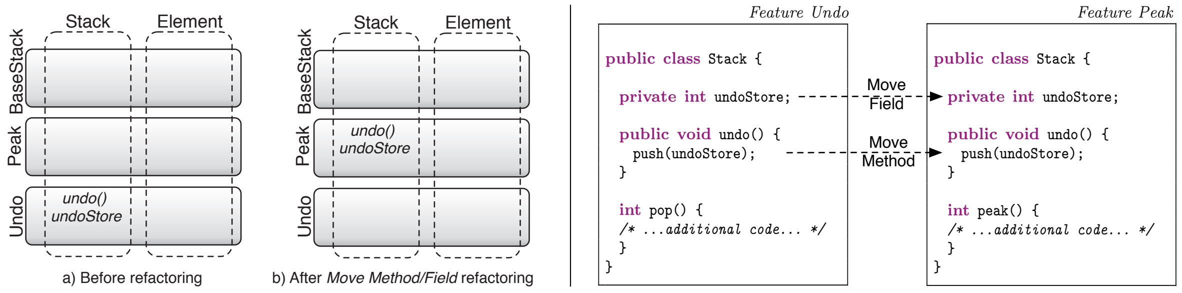
Pre-conditions:

- The features, containing the method(s) that are subject to refactoring, must have the same, direct parent feature.
- The respective methods must be implemented by the same class (e.g., class `Stack` in our example)

Mechanics:

1. Identify the identical methods.
2. Check whether the pre-conditions, mentioned above, are fulfilled.
3. If the candidate methods have different signatures, change the signatures so that they comply with the one you want to have in the target feature.
4. In the case that the class containing the candidate methods has no role in the target feature create the class for the target feature.
5. Create a new method in the corresponding class in the target feature and copy the body of one of the candidate methods to it (and adjust the new method if necessary). If fields are accessed directly within the method (i.e., not passed as parameter), move the field(s) to the target feature using *Pull Up Field to Parent Feature*. Check for dangling references regarding the moved field and update them.
6. Delete the remaining methods in the sub-features.

Example: Similar to the *Pull Up Field* refactoring, we illustrate this refactoring by means of our Stack SPL. The two features *Array* and *LinkedList* contain a method `isEmpty()` (in class `Stack`) that checks, whether the current stack is empty or not. Both methods are identical and the two features have a common parent feature, namely *DataStructure* (cf. Figure 7.1). Hence, the pre-conditions for the refactoring are fulfilled. First, we have to create class `Stack` in the target feature. Next, we create a new method in class `Stack` of feature *DataStructure* and copy the body of one of the methods of the two candidate features. Since the original method(s) reference a field, we also have to apply *Pull Up Field*. We show the corresponding collaboration diagrams in Figure 7.7 and an exemplary implementation in Figure 7.8. Finally, we delete the original methods in features *Array* and *LinkedList*, respectively.



```
public class Stack {

  private int undoStore;

  public void undo() {
    push(undoStore);
  }

  int pop() {
  /* ...additional code... */
  }
}
```

```
public class Stack {

  private int undoStore;

  public void undo() {
    push(undoStore);
  }

  int peak() {
  /* ...additional code... */
  }
}
```

Figure 7.9: On the left side, we depict the corresponding collaboration diagram before and after the *Move Method* and *Move Field* refactoring. On the right side, we show excerpts of the implementation of features *Undo* (before refactoring) and feature *Peak* (after refactoring).

7.3.3 Move Method Between Features

Next, we present FOP-specific adaptations of the *Move Method* and *Move Field* refactoring, respectively. Specifically, we focus on moving a method/field along dimension D1 (cf. Figure 7.4), because this refactoring affects only one class. For dimension D3, the refactorings would be more complex, because we have to consider access modifiers and accessors of fields that are moved across classes. Hence, we limit our considerations to dimension D1 only and set aside the discussion on how to move units along dimension D3 for future work.

Typical situation: A method uses or is used by another feature more than by the one it is currently contained in (e.g., indicated by caller/callee relations). Hence, by moving this method to the feature where it is referenced more frequently (or even exclusively), we can increase feature cohesion. Alternatively, we can apply this refactoring to move a method, which is replicated throughout the source code, to a common feature without decreasing the overall variability of the SPL. However, for applying this refactoring in the latter case, we have to check feature constraints to guarantee that the refactoring is variant-preserving.

Recommended action: Move the method from its current role to the role of the target feature.

Pre-conditions: The method must be available for all variants as before the refactoring. For instance, consider the generic feature model in Figure 7.6 (a), where feature *A* is the parent feature of the optional features *B,C,D*. Furthermore, we assume that in feature *C* a method `foo` exists that references another method `bar` in feature *B*. If we now, for example, move method `bar` to feature *D*, we must ensure, that this method is still accessible for method `foo` in feature *C* for *all* variants that contain this feature. Otherwise, this refactoring would violate our definition for variant preservation, and thus, leads to changed behavior of certain variants.

Mechanics:

1. Examine the source feature for code fragments (e.g., methods, fields) that are only used by the source method. Decide whether they are about to be moved as well.
2. Check whether sub-features or parent features of the source feature exist that contain other declarations of the source method. If there are such features this may hinder to move the source method (e.g., if these features occur in variants without the target feature).
3. If the class that contains the source method has no role in the target feature create the class for the target feature.
4. Declare the method in the target feature (in the respective class).
5. Move the code from the source method to the target. Adjust it if necessary (e.g., references to fields/methods).
6. Decide whether to turn the source method into a delegating method or delete it at all. The latter is especially useful if you have only few or no references to the source method.
7. In case that you delete the source method, replace references to this (outdated) method with references to the target method.

Example: As an example, we apply the refactoring to move a method from the optional feature *Undo* to the optional feature *Peak* of our Stack SPL. In Figure 7.9, we show the corresponding collaboration diagrams and code fragments that are affected by this refactoring. In particular, we move the method `undo()` from feature *Undo* to feature *Peak*. Although this may be questionable from a design point of view, with this example we are able to appropriately illustrate the mechanics of the refactoring. The pre-condition is fulfilled, because feature *Undo* requires feature *Peak* (indicated by a cross-tree constraint in Figure 7.1). Hence, all variants that contain *Undo* contain *Peak* as well. Furthermore, class Stack has already a role in feature *Peak* and thus we do not have to introduce it. Consequently, we declare the method `undo` in feature *Peak* and move the code from the method in the source feature to the target feature. At this point we have to care about moving local objects from *Undo* to *Peak*, namely the field `undoStore` (cf. Figure 7.9). Since we have to move this field as well, we apply *Move Field Between Features* refactoring. You find the respective explanation for this refactoring below. Afterwards, we can proceed by deleting the original method and replace the references of the method `undo()`. Note that we do not depict these last steps in Figure 7.9.

7.3.4 Move Field Between Features

Typical situation: As described above, this refactoring usually supplements the *Move Method* refactoring, to move fields (i.e., member variables) that are referenced within a method that has been moved. In specific cases, this refactoring may also reapplied to move a field that is used by another feature (or a certain role in it respectively) more than by the role on which it is defined.

Recommended action: Move the field from its current role to the role of the target feature.

Pre-conditions: The field must be available for all variants as before the refactoring.

Mechanics:

1. If the class that contains the source method has no role in the target feature, create the class for the target feature.
2. Create a field in the target feature (within the respective class) and provide getter/setter methods to access this field (if necessary).
3. Delete the field from the source feature (to say, the respective role).
4. Replace references to the source field with references to the respective accessor method on the target (e.g., the getter method).

Example: In our example in Figure 7.9, we have to move the field `undoStore` from feature *Undo* to feature *Peak*, because the method that uses the field is moved as well (using the *Move Method Between Features* refactoring). To this end, we create the field in class `Stack` in feature *Peak*. Since the field is not used outside its new role, we do not provide a getter method. Consequently, there are no references that we have to replace either. Finally, we have to delete the original field in the source feature *Undo*.

Although the presented refactorings encompass only a small subset of possible feature-oriented refactorings, they provide us with sufficient facilities to remove clones. In the next section, we present a case study that shows the applicability of these refactorings for removing clones.

7.4 A Case Study on FOP Refactoring – The TankWar Experience

In Chapter 5, we presented an empirical study on code clones in feature-oriented software product lines. However, we omitted the question how to deal with the detected, FOP-specific clones. In this section, we present a case study, where we apply the proposed feature-oriented refactorings to remove code clones in the *TankWar* product line. We provide details on the SPL subject to refactoring and on the methodology we used for removing clones. Afterwards we present and interpret the results of our case study. Finally, we will discuss our results in a broader sense with a specific focus on generalizability and automation of refactorings.

7.4.1 Case Study Setup and Methodology

The TankWar product line.

TankWar is a shoot 'em up game, running on PC and mobile phone, that has been developed as SPL by students of the university of Magdeburg. The game was developed as product line because it must adhere to strong portability requirements [AMC+05]. For instance, TankWar has been developed for PC and mobile phone, which have different constraints regarding memory or display. Even between mobile phones, there can be considerable differences, e.g., a modern smartphone has more memory than a five year old mobile phone. As a result, the developer must be able to tailor the game in order to achieve the best game quality. In Figure A.1 (in the Appendix), we show the feature model of the TankWar product line where features such as *Image* and *Sound* are specific for different platforms.

Several reasons influenced our decision to select TankWar as subject to our case study. First, it has a considerable size (approx. 5000 SLOC) and consists of 38 features. Although this is still a toy example, compared to software systems of industrial size, it is one of the largest SPLs available for FOP. Second, our code clone analysis results revealed that TankWar contains a considerable amount of code clones (total: $\sim 20\%$, FOP-specific: $\sim 14\%$). Hence, this SPL contains a sufficient amount of potential refactoring candidates. Furthermore, we observed that a majority of these clones exist between alternative features, which inherently have a common parent. Additionally, most of these clones occur on method or constructor level. Hence, we expect that removing these clones by refactoring is very promising.

Methodology

In the following, we point out how we got from clone classes to refactoring candidates. Note that we performed most of the steps manually, because tool support for automating the refactoring of feature-oriented SPLs is yet not existent.

From clone analysis, we had already the information on size of clone and clone classes and on the dependencies of the features that contain corresponding clones. Based on these dependencies, we already could estimate *which* refactoring may be applicable to a clone class. For instance, if corresponding clones occurred between alternative features on method level, the *Pull Up Method* refactoring could be applicable. However, to be really sure that a possible refactoring is variant-preserving, we supplemented our manual analysis with tool support to check the defined pre-condition, especially regarding feature dependencies. We obtained the information, which we needed to check this pre-conditions (regarding the features), from *FeatureIDE*[2], an Eclipse plug-in for feature-oriented software development. FeatureIDE provides not only an editor to create feature models but also capabilities to formulate queries to a feature model to gather knowledge about feature relations and dependencies. The query is implemented as a method within FeatureIDE and can be expressed in natural language as question such as *Given a Feature X, which features are in all variants that contain feature X?*. Hence, providing the features that contain the candidate methods (for refactoring) as input for the query, we obtain a list of features that are possible refactoring targets. Due to these capabilities, we can check the pre-conditions such as common parent feature automatically by only providing the features that contain the candidates for refactoring.

After this step, we have a set of fourteen clone classes, which are subject to refactoring. For each of these clone classes, we apply the refactoring manually, according to the mechanics described in Section 7.3. Although this is a tedious task, it is manageable

[2]http://www.fosd.net/featureide/

due to the small amount and the small size of the SPL. After each refactoring, we check whether the variants, that are affected by the refactoring, are still compilable and whether their behavior changed or not.

7.4.2 Results and Interpretation

Next, we provide details on the refactored clone classes, the refactorings we applied and discuss the results and observations we made during the refactoring process. As mentioned above, fourteen clone classes emerged that contain potentially refactorable clones. We list these clone classes in Table 7.1, together with their syntactical category (*SC*), the features containing the code clones (*CF*), the target feature for the refactorings (*RF*), and the applied refactorings (if possible). During the actual refactoring process, we investigated four clone classes (#11 – #14) that are ”not refactorable” for the following reasons: Three of them (#11 – #13), consist of *type-II* clones that are distributed over different sub features of the feature *Tools*, which means that refactoring would be only possible with some workarounds. Since this leads to complicated code and, in this special case, to increased code size, we excluded these clone classes from the refactoring process. The fourth clone clone class (#14) consist of *type-III* clones with notable differences so that a refactoring is not applicable. Beyond this, the clones are scattered over features that have neither a common parent nor other dependencies that are essential for the application of refactorings.

For the remaining ten clone classes that are subject of our refactoring process, we make the following initial observations. Obviously, most of the clones exist between alternative features that separate platform-dependent functionality. In addition, the clone classes fall only into three different syntactical categories (*IfStatement, MethodDeclaration, TypeDeclaration*), which coincides with our observation in Section 5.3 that almost all clone classes fall into one of these categories. Moreover, we made the following observations. First, all member clones, i.e., clones of a single clone class, have a common, direct parent feature. Second, clone classes with syntactical category *TypeDeclaration (TD)* in fact contain replicated methods or constructors as code clones. Hence, we treat them like clone classes of category *MethodDeclaration (MD)* for the refactoring process. Third, we observed that seven clone classes consist of *type-I* clones and three of *type-II* clones.

For the actual refactoring process, we applied the proposed refactorings supplemented by two additional refactorings. The first additional refactoring can be considered as *Pull Up Constructor Body to Parent Feature.* In fact, this refactoring can be considered as a special case of the *Pull Up Method to Parent Feature* refactoring, where the constructor method of a class is subject to refactoring. Furthermore, we applied the *Extract Method* refactoring in its original form to extract replicated code fragments into methods in preparation of a *Pull Up Method* refactoring. We argue that applying this refactoring is possible, because their is no difference between extracting a method in OOP and FOP.

During the application of the refactorings mentioned above, we made the following observations. Initially, we could apply the refactorings to all of the ten clone classes

CC	SC	CF	RF	Refactorings
# 1	TD	Leopard, Abrahams,...	Tanks	Extract Method, Pull Up Method
# 2	MD	PC, Handy	Platform	PUM
# 3	IS	PC, Handy	Platform	EM, PUM
# 4	MD	PC, Handy	Platform	PUM
# 5	MD	PC, Handy	Platform	PUM
# 6	MD	PC, Handy	Platform	PUM
# 7	TD	PC, Handy	Platform	PUC
# 8	MD	Re_PC, Re_Handy	Record	PUM
# 9	TD	Re_PC, Re_Handy	Record	PUC
# 10	IS	TankWar, Tools	TankWar	EM
# 11	TD	Bomb, Freeze, ...	–	–
# 12	TD	Bomb, Freeze, ...	–	–
# 13	TD	Bomb, Freeze, ...	–	–
# 14	IS	Handy, Re_Handy	–	–

CC: clone class; SC: syntactical category; CF: feature(s), containing the clones; RF: feature, the clones are refactored to; EM: Extract Method refactoring; PUM: Pull Up Method refactoring; PUC: Pull Up Constructor Body refactoring

Table 7.1: Overview of clone classes removed by refactorings

and consequently, remove the code clones. For three clone classes, we had to apply the *Extract Method* refactoring in advance, either for extracting the identical part of the clones (#1) or for extracting if statements into methods (#3, #10). For two clone classes (#7, #9) we had to replace a value by a variable, which we initialized for each of the clones separately. Finally, we applied the *Pull Up Method* refactoring to all clone classes to remove the clones.

After the code clone removal process, we analyzed the TankWar product line again, according to the methodology introduced in Section 5.2. This lead to the following results. The amount of code clones has been decreased throughout all analysis steps. Regarding the initial clone detection, the amount of clones decreased from 20 % (cf. Figure 5.1) to 12 %. For the syntactical classification, the amount of code clones is 50 % lower in the refactored SPL (7 %) compared to the original one (15 %). Finally, we achieved a vast decrease of the amount of FOP-specific clones. In the refactored SPL, only 4 % FOP-specific clones exist, which is three times lower than in the original SPL (12 %). Hence, we conclude that code clone removal through refactorings is a promising approach to remove FOP-specific clones from feature-oriented SPLs.

7.5 Discussion

In this chapter, we proposed feature-oriented refactorings and presented a case study to demonstrate their applicability for code clone removal. Nevertheless, two issues remain that we could not address: supporting the (automated) process of refactorings and generalizability, especially regarding refactoring in preprocessor-based software product lines. In the following, we discuss both issues.

Automating FOP-specific refactorings.

An important aspect for the applicability of the proposed refactorings is to what extent the refactoring process can be automated. For the refactorings, presented in this paper, we identified three steps that would benefit from such automation: detecting possible refactoring candidates, checking the pre-conditions, and changing the source code. In the following, we mainly focus on step one and three, because we addressed the second step already in Section 7.4.

The first step is of particular interest for the *Pull Up* refactorings, because it is impossible to identify identical or similar code fragments manually. However, different approaches exist that can be used to guide the user to such code fragments. First of all, *clone detection*, that is, the detection of replicated code fragments, can be used to determine syntactically identical or similar code fragments. Second, with recent approaches it is even possible to detect semantically identical (but syntactically different) code fragments [JDH10]. However, for the latter kind of similar code fragments, it is still open how to automate the syntactical unification of such code fragments.

Automating the actual refactoring, especially the propagation of source code changes, requires an appropriate representation of the underlying source code such as an *abstract syntax tree (AST).* Since generating such as AST for each variant of an SPL is impossible, an AST for the whole SPL, which contains all information on variability, is required. Recently, FUJI, an extensible compiler for feature-oriented programming in Java, has been proposed [AKL^{+}12]. Amongst others, FUJI provides an AST for each feature of the product line and thus is promising with regard to automating the refactoring of feature-oriented SPLs. Beyond that, recent approaches for annotation-based approaches provide a variability-aware AST that could be used for refactoring (especially for automating the corresponding step), which is one of our future tasks [KGR^{+}11].

Generalizability.

While we focussed on compositional SPL implementation techniques, specifically on FOP, in this chapter, we are also concerned with code clones in preprocessor-based software product lines within this thesis.

Regarding our refactoring approach, the question is whether we can apply it to the annotation-based approach as well. From our point of view the answer is twofold: First, given our definition for variant-preserving refactoring we argue that it holds for annotation-based approaches, specifically the CPP, as well. The reason is that we can also build a feature model for annotative SPLs to determine which combinations of features are valid and which are not. Since our definition is based on the valid combination of features, it can be applied to annotation-based SPLs as well.

However, applying the FOP-specific refactorings, that we propose in this chapter, to annotation-based SPLs may require additional or changed mechanics. For instance, in C programs, variability is also expressed using configuration files that determine whether certain source files may be compiled together or not. Since this is an additional variability mechanism, it may influence the actual refactoring process and the preservation of variants. Furthermore, other preprocessor directives for file inclusion (*#include*) and macros (*#define*) exist that complicate refactorings.

Beside this, the main problem of annotative software product lines is, that the CPP, used to express variability is *not* part of the actual host language. Thus, parsing and building an AST for such product lines does not provide us with information about variability. However, this information is necessary (beside the information about feature dependencies) to perform automated analysis in general and to decide whether a refactoring is applicable or not in particular [LKA11]. Moreover, the actual refactoring is executed on the AST rather than on the source code representation within an IDE. Recently, Kästner et al. proposed a variability-aware parser that parses unprocessed code within the TypeChef project [KGR+11]. Amongst others, this parser provides an AST that contains all variability information by hosting the preprocessor annotations as special node within the tree. Hence, it is possible to use this AST for different kind of automated analyses, in particular, for refactoring. Nevertheless, we argue that it is still a long road to go until sufficient refactoring support, mainly ”out-of-the-box” for annotative SPLs will be available.

We conclude, that our proposed refactorings are only partly generalizable and that we more work has to be done to fully generalize variant-preserving refactoring across SPL implementation techniques. Nevertheless, we argue that our approach can be used as a starting point.

7.6 Related Work

A variety of work has been done on refactoring software product lines. In this section, we account for the most important work and figure out how it differs from the approach we presented in this chapter.

Refactoring of feature models.

Similar to us, Alves et al. identified shortcomings in traditional refactorings when applied to SPLs due to missing support of configurability [AGM+06]. As a solution, they propose a set of feature model refactorings to improve the feature model of an SPL and, beyond that, to merge multiple programs into one SPL. Likewise, Thüm et al. present an approach that supports reasoning about feature model edits [TBK09]. To this end, they classify changes to a feature model into refactoring, generalization, specialization, and arbitrary edits. Additionally, they formalize their approach using propositional formulas. All the aforementioned approaches have in common that they focus on refactoring of feature models. In contrast, we focus on refactoring the underlying source code of an SPL with our approach. Although we take feature models into account, we use their information for evaluating pre-conditions only. Finally, we focus on feature-oriented SPLs while the work mentioned above is independent of the SPL implementation technique.

Refactoring of feature-oriented SPLs.

Liu et al. propose a theory for feature-oriented refactoring that relates code to algebraic refactoring [LBL06]. With their theory, they focus on decomposing programs into features. Specifically, they address the problem, that features may have different implementations in different variants of the SPL. In contrast to our work, where we address the refactoring of source code in existing feature-oriented SPLs, they address how to refactor an object-oriented legacy application into features. Kuhlemann et al. propose *refactoring feature modules (RFM)* for refactoring in feature-oriented SPLs [KBA09]. The core idea of RFM is to encapsulate information that is necessary for performing an object-oriented refactoring in a separate feature module and make it explicit in the corresponding feature model. Additionally, Kuhlemann et al. formalized their approach by means of an algebraic model [KKAS11]. In contrast to their approach, where refactorings are features in a concrete SPL, we aim at a more interactive approach. In particular, we aim at integrating the proposed refactoring into an IDE such as Eclipse so that they can be used on any SPL, depending on the implementation technique only. Finally, Borba et al. recently proposed a theory that covers both, refactoring of feature models and source code, based on product line refinement [BTG10]. With their general, language-independent formalization they provide SPL properties (in terms of refinement) that support evolution of software product lines. In contrast to their work, we specifically focus on FOP and introduce first refactorings in a more practical way by describing rather the actual mechanisms than the underlying theory. However, integrating our ideas with their theory could be beneficial and is left for future work.

7.7 Summary

Refactoring of source code is a pivotal task regarding the quality and longevity of source code. While it is well-understood for conventional, stand-alone systems, only few work exists for SPLs. In this chapter, we presented how existing, object-oriented refactorings can be extended and applied to feature-oriented SPLs, while preserving all variants of the SPL. To this end, we introduced the notion of variant-preserving refactoring and proposed concrete refactorings for feature-oriented SPLs.

To show the applicability of our refactorings, we presented a case study on code clone removal in one feature-oriented SPL by means of our refactorings. As a result of these refactorings, we could remove a large portion of FOP-specific clones. However, there are some limitations that we observed during our case study. As a matter of fact, all of the removed clones occurred in features with a common, direct parent feature and most of them where alternative features. Hence, we can make no clear statement on removing clones caused by fine-grained extensions or crosscutting concerns. Furthermore, we can not estimate how difficult it is to apply our refactorings on features that have more complex dependencies that go beyond alternatives and parent-child relationships. Moreover, the detected clones where mostly identical (*Type-I*) or had only slight differences (*Type-II*). However, we found FOP-specific clones (*Type-II* and *Type-III*) that where not refactorable at all. In particular, one reason was that the application of refactorings implied complicated workarounds that outweigh the benefits of code clone removal. For instance, in one case the refactoring would have lead to multiple methods (instead of two) and finally, even more lines of code that introduced by code clones. This observation lead us to the assumption that there is a border line where the extraction (of clones) is difficult or not beneficial for maintainability anyway.

Beyond the removal of code clones we also discussed the possible automation of our refactorings, which highly depends on appropriate tool support. Finally, we discussed the generalizability of our approach. Specifically, we focussed on annotative SPLs using the CPP and to what extent variant-preserving refactorings make sense and are applicable for such software product lines.

As next steps, we suggest to extend this work by more refactorings for feature-oriented and annotative SPLs. Moreover, we put some effort on providing tool support for these refactorings so that they can be applied and pre-conditions can be checked automatically. Additionally, merging our ideas with existing theories and formalizations for feature-oriented refactoring is an important future task, because it allows us to prove the variant-preserving nature of refactorings. Finally, extending our refactoring approach to annotative SPLs is a challenging but promising task that we intend to pursue in the future.

8. Conclusion

Software Product Lines provide facilities for efficiently managing thousand of (software) products at once by means of variabilities and commonalities. Such an approach comes with different advantages such as fast time-to-market and reuse at large-scale. Hence, it plays a pivotal role for commercial success of software development. Consequently, SPLs gain momentum in both, academia as well as industry.

In research, major work on product lines encompasses implementing, testing, and verification. Furthermore, evolution of SPLs, specifically of the problem space (e.g., variability models) is subject of research. In contrast, reengineering & maintenance (where clone detection and analysis belongs to) has not been subject of intensive research so far. However, we argue that software product lines evolve similar to single software systems or even more. As a result, maintenance and code quality become a problem. Due to the complexity of industrial SPLs, caused by different variability spaces, feature semantics etc., it is a challenging task to counter this evolutionary decays with common approaches. Consequently, new approaches, tailored to the specific characteristics and mechanisms of SPLs have to be investigated to avoid such problems. With this thesis, we bridge this gap by tailoring clone analysis and removal to software product lines. In a broader sense, we aim at encouraging other researcher to put emphasis on this field of research.

Our main contribution is to provide insights on code clones in SPLs (compositional and annotative) and how to remove them by the application of refactorings. We guided our analysis by four research questions, which we presented in Chapter 1. To conclude our work, we go back to this questions and present the main answers in this chapter.

Do code clones exist in software product lines?

Although this question appears to be superfluous, it is of great importance for this thesis. We posed it to emphasize the fact, that *nothing* is known about code clones in SPLs, even not whether they exist. Furthermore, it is absolutely unclear to what extent compositional approaches such as FOP may avoid code clones due to new reuse mechanisms such as refinements. Hence, we aim at provide initial insights on both issues by answering this question.

To this end, we first provided some theoretical discussion on modularity and expressiveness in SPLs and how they may foster code clones. As a result, we not only provided a structured review of possible advantages and drawbacks of certain implementation approaches, but also provided an overview *why* code clones could occur in SPLs.

Then, we provided case studies to complement our theoretical reflection with facts. As overall result, we detected code clones, specific to product lines (FOP-specific and *#ifdef* clones), in both, compositional as well as annotative software product lines. However, we also observed considerable differences. Comparing both implementation approaches, it turned out that feature-oriented SPLs contain a considerable higher amount of clones than the preprocessor-based SPLs. Interestingly, this result confirms with the assumption, which results from the discussion in Chapter 4. Furthermore, with our case study on the latter SPLs, we provide evidence for the general assumption that the discipline of annotations is related to the amount of clones. In particular, we detected more code clones in disciplined systems that is, systems, which contain no or only very few undisciplined annotations.

Regarding feature-oriented software product lines, we provided a first definition for what renders a clone to be caused by feature orientation. Based on this definition, a considerable amount of *FOP-specific* clones exist, as revealed by our analysis. Another interesting observation we made is the fact that the amount of clones depends on the actual development process. More precisely, our case study clearly indicates that feature-oriented SPLs developed from scratch contain more code clones than SPLs that have been refactored form legacy applications.

Eventually, we could confirm the existence of code clones in SPLs and provide reasons, why these clone occur.

Can we observe certain patterns of cloning that are specific to software product lines?

By providing patterns of clones, we gain insights on *where* clones and probably *why* they occur. Furthermore, such patterns may be useful to be aware of clones in other systems and for reasoning about alternative solutions, which avoid code cloning.

Within this thesis, we presented pattern for both, feature-oriented and preprocessor-based software product lines. The term pattern in this context indicates that code clones are accompanied by certain, recurring observations with regard to problem and solution space. For feature-oriented product lines, we found patterns regarding the implementation and the feature model. For the first, we observed that the detected clones occur mostly on block level, that is, they are encompassed by either loops, conditionals, methods, or even whole classes. This, in turn, clearly indicates that a coherent piece of functionality has been copied and that there is a relatively high potential for removing such clones. Additionally, we figured out that code clones occur mainly between sibling features, in particular within an alternative group. This observation has different implications from our point of view. First, this relation between features may be a reason for corresponding clones, because such features usually implement similar concepts. Second, programmers can use this pattern to be aware of clones during development and to think about possible alternative solutions, which avoid code replication. For instance, as we have demonstrated it is possible to unify such clones in the common parent feature. Hence, we argue that having knowledge on the aforementioned patterns is beneficial during implementing and reviewing feature-oriented software product lines.

For preprocessor-based software product lines, we observed different patterns than for feature-oriented. Regarding variable code blocks (i.e., features), we observed that corresponding clones occur nearly always within one feature. In terms of preprocessor annotations, this means, that the boolean expression such as `#ifdef A && B` is identical of the blocks that contain the clones. Interestingly, this observation is contrary to our observations of FOP. Moreover, we detected a certain pattern regarding files, which contain clones. For undisciplined systems, corresponding clones occur mainly in one file. In contrast, for disciplined systems, we observed also a considerable amount of clones that occur between multiple files.

Overall, we presented some interesting patterns, which provide insights on where and why clones may occur. Additionally, these patterns are useful to raise the awareness of clones and to support programmers in reason about certain clones during implementation.

Is it possible to judge on the harmfulness of code clones in software product lines? And if so, how?

Usually, different means such as evolution of clones, inconsistent changes or correlations of clone occurrences with software metrics are used for assessing their harmfulness. In this thesis we did not address this explicitly for two reasons. First, at least for feature-oriented software product lines no history information such as code repositories exist and thus, we had no access to evolutionary information about these systems. While this information is available for the annotative SPLs, it is a complex task to figure out changes of *#ifdef* clones over multiple version. We come back to this topic in Chapter 9. Second, metrics that take the variability into account simply do not exist. Hence, we even could not relate the detected clones to certain metrics that *potentially* indicate their harmfulness.

However, we observed other indicators that may serve as arguments in favor and against the refactoring of code clones. In our study on feature-oriented software product lines, we detected a vast amount of clones, which occur in alternative features (and the same class) and on a rather coarse-grained level (regarding the syntactical elements they are enclosed by). In our exemplary case study on refactoring, we have shown for the TankWar SPL, that such clones are good refactoring candidates. The main reason is that for alternative features a more abstract feature exist, that is, the parent feature. Hence, by pulling these cloned fragments up in the feature model, they reside in a more ”abstract” unit, which can be accessed by each of the original features. Without any other information for assessing the harmfulness, we argue that in such cases a refactoring is recommended to make use of abstraction. However, we also observed borderline cases, where a refactoring is cumbersome, due to certain workarounds, and may introduce a unnecessarily high amount of very small methods.

For the preprocessor-based software product lines, which we analyzed in Chapter 6, we made different observations. The detected *#ifdef* clones mainly occur on a fine-grained level and do not encompass a self-contained piece of functionality. This, in turn, makes it hard to justify any refactoring, regardless of the fact that refactoring in the presence of conditional compilation is a difficult task. We argue, that refactoring for removing these clones is not recommended, also for another reason: Most of the detected clones occur within one feature. Furthermore, to some extent (at least for all undisciplined systems we analyzed) these clones even occur in the same file. Hence, these clones exhibit some kind of *locality* which is advantageous if we consider controlling such clones instead of removing them. Finally, in certain cases code clones prevent from undisciplined annotations, which may obfuscate the source code and make it hard to understand. Although all the aforementioned reasons to not provide tangible information on the harmfulness of clones, we argue that in the case of annotative SPLs, code clone controlling is the more preferable solution compared to code clone removal.

Is it possible to remove code clones from a product line point of view?

For removing clones, refactoring is a commonly applied technique. Unfortunately, software product lines pose additional challenges for applying refactorings, mainly caused by the variability (e.g., features) that has to be taken into account. This not only increases the complexity of such refactorings but also limits the reuse of existing (object-oriented) refactoring in large parts. Within this thesis, we figured out the different dimensions that can occur during refactoring of feature-oriented software product lines. Based on these dimensions, we came up with the notion of *variant-preserving* refactoring, which explicitly takes the variability of SPLs into account. Based on this notion, we could provide exemplary refactorings, which are based on existing OO refactorings and tailored specifically to FOP. We applied these refactorings to remove clones for one exemplary product line. As a result, we could reduce the amount of FOP-specific clones significantly. Hence, we have shown that code clone removal is a viable approach, at least for feature-oriented software product lines, to remove clones. Nevertheless, we argue that our work is just a starting point. We proposed only few refactorings that were sufficient to remove the clones we detected. But there may be other clones, which require a different treatment and different refactorings.

Furthermore, we argue that one of the main contributions regarding code clone removal is the notion of variant-preserving refactoring. Although we address feature-oriented SPLs with this notion, it is generalizable regarding both, other implementation approaches such as annotative SPLs but also other compositional approaches such as

AOP. By providing this notion, we raised awareness of the challenges of refactoring software product lines and lay a foundation for further reasoning on refactoring SPLs. Additionally, based on this notion, we can provide concrete refactorings (even derived from existing ones), tailored to the specific characteristics of SPLs.

However, we also reached limitations in answering this question, first and foremost, regarding preprocessor-based software product lines. Thus, we could not provide means to remove clones from CPP-based SPLs. Although we argue that in our case study, the detected clones should not be removed anyway, there may be cases where such a removal is beneficial or even inevitable. We addressed this issue in Section 5.4, by provide a discussion on generalizing our approach for refactoring feature-oriented SPLs. Specifically, we figured out a possible way to cope with refactoring in annotative SPLs by providing information on variability directly in the AST.

9. Future Work

The underlying thesis contains two main contributions: empirical studies and thus first insights on code clones in software product lines and a variant-preserving refactoring of product lines. For both areas, we propose suggestions for further work. Additionally, we propose ideas for work on evolution of software product lines.

The Role of Clones in Compositional and Annotative SPLs

With respect to code clones, we suggest to gain more insights on differences between code clones in compositional and annotative implementation approaches. Although we provided first insights for both approaches (by means of FOP and the CPP), a deeper investigation and a comparison is still open. This not only includes a larger empirical studies with more SPLs, but also to directly compare the results to assess the role of clones in the respective approaches, preferably with regard to certain criteria such as frequency, granularity, or harmfulness.

Especially the latter is a topic on its own, which we want to address in further work. But how to evaluate the harmfulness of clones in software product lines? Currently, we pursuit different ideas, commonly used in code clone research before. In particular, we want to analyze the evolution of clones to find out, whether negative effects such as inconsistent changes occur. Another idea is to develop measures for the quality and maintainability of software product lines. In presence of such measures, we then can explore whether a correlation between the measures (e.g., negative values thereof) and detected clones exist and thus assess the harmfulness of code clones.

Finally, we suggest to extend our current study to other paradigms and techniques such as Aspect-Oriented or Delta-Oriented Programming (AOP/DOP). As a result, we can draw a bigger picture of code clones in SPLs. Furthermore, we can compare different compositional approaches with regard to code clone occurrences to possibly find out the most essential reasons for cloning in such SPLs.

Towards Automated Refactoring in SPLs

One of the key results of this thesis is the definition of *variant-preserving refactoring* for SPLs. Additionally, we provided concrete refactoring techniques and *how* and *when* we can apply them to remove code clones. We suggest to put considerable effort in further work on this topic in two directions.

First, the initial idea of refactoring for FOP should be transferred to other compositional approaches. To this end, the original notion of variant-preserving refactoring must be defined so that we can abstract from concrete variability mechanisms such as refinements or aspects. In particular, we suggest to investigate to what extent a more formal definition, similar to the refinement theory of Borba [BTG10], is suitable for this purpose.

Second, we suggest to extend the existing set of refactorings and provide tool support for FOP. Usually, refactoring is guided by bad design, also called *code smells*. It is inevitable to provide code smells for feature-oriented SPLs as well, either by reviewing existing SPLs or by defining measures that aid in detecting code smells automatically. Furthermore, to efficiently apply the proposed refactorings, tool support is crucial to support the developer. Currently, we and others are working on this issue, by providing a tool that checks the applicability of a certain refactoring as well as execute the refactoring. Nevertheless, this is a challenging task and further work can help to achieve a broader range of different approaches.

Third, adapting the ideas of variant-preserving refactoring to annotative SPLs is a challenging, yet, beneficial issue for further work. We suggest to rethink our notion in the light of preprocessor annotations to provide variability-aware refactoring. Based on such a notion and current approaches on variability aware analysis of CPP-based systems [KGR$^+$11, KATS12], tool support for (automated) refactoring in the presence of preprocessor annotations would be the ultimate goal.

Evolution of Variable Software Systems

Software evolution plays a pivotal role in software engineering, because it affects software quality and maintainability. While software product lines evolve as well, only few work exists that analyzes the evolution of SPLs. Specifically, the evolution of systems that use the C preprocessor such as Linux, have a long history of evolution. We suggest to concentrate on this issue as well in future work. While first approaches exist that analyze (and understand) the evolution of the variability model of such systems [LSB$^+$10, PCW12], it is interesting to gain insights on how variability in the implementation space (i.e., the source code) evolves. Particularly, we suggest to focus on co-changes between preprocessor annotations (and their corresponding preprocessor variable) and what this could indicate with regard to the overall product-line architecture. Additionally, such an evolution analysis could provide insights to what extent variability evolves over time and to what extent it is stable.

A. Appendix

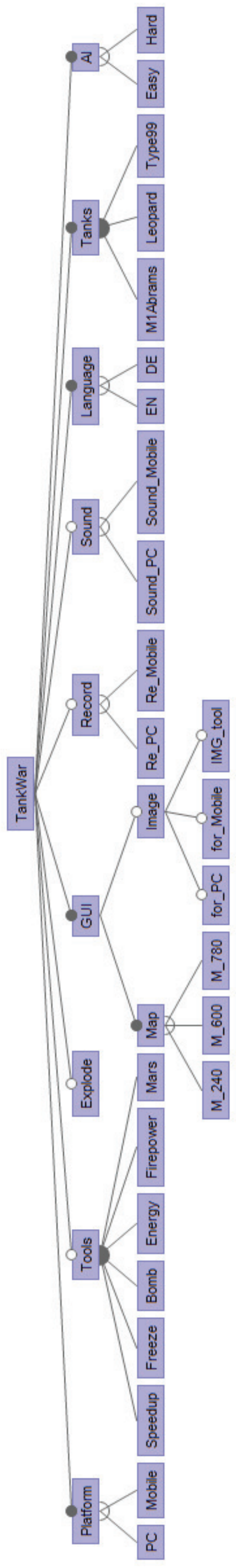


Figure A.1: Feature model of the *TankWar* product line

Bibliography

[ACDP07] L. Aversano, L. Cerulo, and M. Di Penta. How Clones are Maintained: An Empirical Study. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 81–90. IEEE Computer Society, 2007. (cited on Page 28)

[AF96] T. Anderson and J. Finn. *The New Statistical Analysis of Data*. Springer-Verlag, 1996. (cited on Page 85)

[AGM+06] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. Refactoring Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 201–210. ACM Press, 2006. (cited on Page 111)

[AK09] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009. (cited on Page 3, 39, and 40)

[AKL09] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE Computer Society, 2009. (cited on Page 3, 39, 40, 58, and 69)

[AKL+12] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. Access Control in Feature-Oriented Programming. *Science of Computer Programming*, 77(3):174–187, 2012. Feature-Oriented Software Development (FOSD 2009). (cited on Page 103 and 110)

[ALRS05] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 125–140. Springer-Verlag, 2005. (cited on Page 69)

[ALS08] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. *IEEE Transactions on Software Engineering (TSE)*, 34(2):162–180, 2008. (cited on Page 48)

[AMC+05] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 70–81. Springer-Verlag, 2005. (cited on Page 106)

[AMGS05] G. Antoniol, E. Merlo, Y.-G. Guéhéneuc, and H. Sahraoui. On Feature Traceability in Object Oriented Programs. In *Proceedings of the International Workshop on Traceability in Emerging Forms of Software Engineering (TEFSE)*, pages 73–78. ACM Press, 2005. (cited on Page 39 and 40)

[Ape10] S. Apel. How AspectJ is Used: An Analysis of Eleven AspectJ Programs. *Journal of Object Technology (JOT)*, 9(1):117–142, 2010. (cited on Page 48)

[AVRGC08] B. Adams, B. Van Rompaey, C. Gibbs, and Y. Coady. Aspect Mining in the Presence of the C Preprocessor. In *Proceedings of the AOSD Workshop on Linking Aspect Technology and Evolution (LATE)*, pages 1:1–1:6. ACM Press, 2008. (cited on Page 3)

[Bak92] B. Baker. A Program for Identifying Duplicated Code. *Computing Science and Statistics*, 24:49–57, 1992. (cited on Page 11, 19, and 31)

[Bak95] B. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 86–95. IEEE Computer Society, 1995. (cited on Page 1, 12, 19, 22, 27, and 70)

[Bak96] B. Baker. Parameterized Pattern Matching: Algorithms and Applications. *Journal of Computer and System Sciences*, 52(1):28–42, 1996. (cited on Page 19)

[Bat05] D. Batory. Feature Models, Grammars, and Propositional Formulas. In H. Obbink and K. Pohl, editors, *Software Product Lines*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005. (cited on Page 37 and 38)

[BC90] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311. ACM Press, 1990. (cited on Page 39 and 40)

[BDET05] M. Bruntink, A. Deursen, R. Engelen, and T. Tourwe. On the Use of Clone Detection for Identifying Crosscutting Concern Code. *IEEE Transactions on Software Engineering (TSE)*, 31(10):804–818, 2005. (cited on Page 59, 70, and 82)

[BDS10] L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines Using Traits. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 2096–2102. ACM Press, 2010. (cited on Page 39)

[BFG07] T. Bakota, R. Ferenc, and T. Gyimothy. Clone Smells in Software Evolution. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 24–33. IEEE Computer Society, 2007. (cited on Page 29)

[Big98] T. Biggerstaff. A Perspective of Generative Reuse. *Annals of Software Engineering*, 5:169–226, 1998. (cited on Page 39)

[BJ07] H. A. Basit and S. Jarzabek. Efficient Token Based Clone Detection with Flexible Tokenization. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 513–516. ACM Press, 2007. (cited on Page 19 and 26)

[BKA+07] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering (TSE)*, 33(9):577–591, sept. 2007. (cited on Page 10, 16, and 17)

[BKZ11] L. Barbour, F. Khomh, and Y. Zou. Late Propagation in Software Clones. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 273–282. IEEE Computer Society, 2011. (cited on Page 22 and 29)

[BM01] I. Baxter and M. Mehlich. Preprocessor Conditional Removal by Simple Partial Evaluation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 281–290. IEEE Computer Society, 2001. (cited on Page 50 and 75)

[BMD+99a] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring Clone based Reengineering Opportunities. In *International Software Metrics Symposium (METRICS)*, pages 292–303. IEEE Computer Society, 1999. (cited on Page 10 and 24)

[BMD+99b] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Partial Redesign of Java Software Systems Based on Clone Analysis . In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 326–336. IEEE Computer Society, 1999. (cited on Page 26, 31, and 32)

[BMD+00] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Advanced Clone Analysis to Support Object-Oriented System Refactoring. In *Proceedings of the Working Conference on Reverse Engineering*

(WCRE), pages 98–107. IEEE Computer Society, 2000. (cited on Page 26, 31, and 70)

[BPSP04] D. Beuche, H. Papajewski, and W. Schroder-Preikschat. Variability Management with Feature Models. *Science of Computer Programming*, 53(3):333–352, 2004. (cited on Page 3)

[BRJ05] H. Basit, D. Rajapakse, and S. Jarzabek. An Empirical Study on Limits of Clone Unification Using Generics. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 109–114, 2005. (cited on Page 32)

[BSI$^+$09] N. Bettenburg, W. Shang, W. Ibrahim, B. Adams, Y. Zou, and A. Hassan. An Empirical Study on Inconsistent Changes to Code Clones at Release Level. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 85–94. IEEE Computer Society, 2009. (cited on Page 28)

[BSR04] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004. (cited on Page 39, 40, 58, and 69)

[BTG10] P. Borba, L. Teixeira, and R. Gheyi. A Theory of Software Product Line Refinement. In *Proceedings of the International Colloquium on Theoretical Aspects of Computing (ICTAC)*, pages 15–43. Springer-Verlag, 2010. (cited on Page 91, 97, 112, and 122)

[BYM$^+$98] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 368–377. IEEE Computer Society, 1998. (cited on Page 1, 20, 31, and 70)

[CAB04] A. Colyer, R. A., and G. Blair. On the Separation of Concerns in Program Families. Technical Report Technical Report COMP-001-2004, Computing Department, Lancaster University, 2004. (cited on Page 47)

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000. (cited on Page 3, 35, 37, and 39)

[CH93] K. W. Church and J. I. Helfman. Dotplot: A Program for Exploring Self-Similarity in Millions of Lines of Text and Code. *Journal of Computational and Graphical Statistics*, 2(2):153–174, 1993. (cited on Page 22)

[CHE05] K. Czarnecki, S. Helsen, and U. Eisenecker. Formalizing Cardinality-Based Feature Models and Their Specialization. *Software Process: Improvement and Practice*, 10(1):7–29, 2005. (cited on Page 37)

[CN01] P. Clements and L. Northrop. *Software Product Lines – Practices and Patterns.* Addison-Wesley, 2001. (cited on Page 35)

[Cor03] J. Cordy. Comprehending Reality - Practical Barriers to Industrial Adoption of Software Maintenance Automation. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 196–205. IEEE Computer Society, 2003. (cited on Page 30 and 32)

[Cor11] J. Cordy. Exploring Large-Scale System Similarity Using Incremental Clone Detection and Live Scatterplots. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 151–160. IEEE Computer Society, 2011. (cited on Page 22)

[DBF$^+$95] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley. The Development of a Software Clone Detector. *International Journal of Applied Software Technology*, 1(3/4):219–236, 1995. (cited on Page 10)

[DER07] E. Duala-Ekoko and M. Robillard. Tracking Code Clones in Evolving Software. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 158–167. IEEE Computer Society, 2007. (cited on Page 69)

[DER08] E. Duala-Ekoko and M. P. Robillard. Clonetracker: Tool Support for Code Clone Management. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 843–846. ACM Press, 2008. (cited on Page 33)

[DHJ$^+$08] F. Deissenboeck, B. Hummel, E. Jurgens, B. Schatz, S. Wagner, J.-F. Girard, and S. Teuchert. Clone Detection in Automotive Model-based Development. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 603–612. IEEE Computer Society, 2008. (cited on Page 14)

[Dij76] E. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976. (cited on Page 46)

[DLDPF02] G. Di Lucca, M. Di Penta, and A. Fasolino. An Approach to Identify Duplicated Web Pages. In *Proceedings of the International Computer Software and Applications Conference (COMPSAC)*, pages 481–486. IEEE Computer Society, 2002. (cited on Page 22)

[DNS$^+$06] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. P. Black. Traits: A Mechanism for Fine-Grained Reuse. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(2):331–388, 2006. (cited on Page 39)

[DRD99] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 109–118. IEEE Computer Society, 1999. (cited on Page 18 and 22)

[EBN02] M. Ernst, G. Badros, and D. Notkin. An Empirical Analysis of C Preprocessor Use. *IEEE Transactions on Software Engineering (TSE)*, 28(12):1146–1170, 2002. (cited on Page 3, 42, 74, 75, and 87)

[EFM09] W. Evans, C. Fraser, and F. Ma. Clone Detection via Structural Abstraction. *Software Quality Journal*, 17:309–330, 2009. (cited on Page 20)

[Fav95] J. Favre. The CPP Paradox. In *Proceedings of the European Workshop on Software Maintenance*, 1995. (cited on Page 3 and 42)

[Fav97] J. Favre. Understanding-In-The-Large. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 29–38. IEEE Computer Society, 1997. (cited on Page 3, 42, and 74)

[FFK08] R. Falke, P. Frenzel, and R. Koschke. Empirical Evaluation of Clone Detection Using Syntax Suffix Trees. *Empirical Software Engineering*, 13(6):601–643, 2008. (cited on Page 20)

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, July 1987. (cited on Page 21)

[Fow00] R. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000. (cited on Page 1, 7, 26, 30, 62, 91, 94, 99, and 100)

[GH11] N. Göde and J. Harder. Clone Stability. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 65–74. IEEE Computer Society, 2011. (cited on Page 1 and 28)

[Giv08] L. Giventer. *Statistical Analysis for Public Administration*. Jones and Bartlett Publishing, second edition, 2008. (cited on Page 85)

[GJS08] M. Gabel, L. Jiang, and Z. Su. Scalable Detection of Semantic Clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 321–330. IEEE Computer Society, 2008. (cited on Page 12 and 21)

[GK11] N. Göde and R. Koschke. Frequency and Risks of Changes to Clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. IEEE Computer Society, 2011. (cited on Page 22 and 29)

[GLA$^+$09] D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew. Verifying Architectural Design Rules of the Flight Software Product Line. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 161–170. Carnegie Mellon University, 2009. (cited on Page 2)

[Gri81] S. Grier. A Tool that detects Plagiarism in Pascal Programs. In *Proceedings of the SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '81, pages 15–20. ACM Press, 1981. (cited on Page 9)

[HC01] G. Heineman and W. Council. *Component-Based Software Engineering*. Addison-Wesley, 2001. (cited on Page 39)

[Hel96] J. Helfman. Dotplot Patterns: A Literal Look at Pattern Languages. *Theory and Practice of Object Systems*, 2(1):31–41, 1996. (cited on Page xi, 22, and 23)

[HG11] J. Harder and N. Göde. Efficiently Handling Clone Data: RCF and Cyclone. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 81–82. ACM Press, 2011. (cited on Page 23 and 24)

[HJJ09a] D. Hou, P. Jablonski, and F. Jacob. CnP: Towards an Environment for the Proactive Management of Copy-and-Paste Programming. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 238–242. IEEE Computer Society, 2009. (cited on Page 33)

[HJJ09b] D. Hou, F. Jacob, and P. Jablonski. Exploring the Design Space of Proactive Tool Support for Copy-and-Paste Programming. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 188–202. ACM Press, 2009. (cited on Page 33)

[HK09] Y. Higo and S. Kusumoto. Enhancing Quality of Code Clone Detection with Program Dependency Graph. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 315–316. IEEE Computer Society, 2009. (cited on Page 21)

[HKI08] Y. Higo, S. Kusumoto, and K. Inoue. A Metric-Based Approach to Identifying Refactoring Opportunities for Merging Code Clones in a Java Software System. *Journal of Software Maintenance and Evolution*, 20(6):435–461, 2008. (cited on Page 26)

[HKKI04a] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring Support Environment Based on Code Clone Analysis. In *Proceedings of the International Conference on Software Engineering and Applications (SEA)*, pages 222–229, 2004. (cited on Page 32)

[HKKI04b] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Refactoring Support Based on Code Clone Analysis. In F. Bomarius and H. Iida, editors, *Product Focused Software Process Improvement (PROFES)*, volume 3009 of *Lecture Notes in Computer Science*, pages 220–233. Springer, 2004. (cited on Page 31)

[HSHK10] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto. Is Duplicate Code More Frequently Modified Than Non-Duplicate Code in Software Evolution?: An Empirical Study on Open Source Software. In *Proceedings of the Proceedings of the Joint Workshop on Software Evolution and International Workshop on Principles of Software Evolution (IWPSE-EVOL)*, pages 73–82. ACM Press, 2010. (cited on Page 28)

[HYNK11] Y. Higo, U. Yasushi, M. Nishino, and S. Kusumoto. Incremental Code Clone Detection: A PDG-based Approach. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 3 –12, oct. 2011. (cited on Page 21)

[Jan88] H. Jankowitz. Detecting Plagiarism in Student Pascal Programs. *The Computer Journal*, 31(1):1–8, 1988. (cited on Page 9)

[JB09] H. P. Jepsen and D. Beuche. Running a Software Product Line: Standing Still is Going Backwards. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 101–110. Carnegie Mellon University, 2009. (cited on Page 2)

[JDF$^+$10] E. Juergens, F. Deissenboeck, M. Feilkas, B. Hummel, B. Schaetz, S. Wagner, C. Domann, and J. Streit. Can Clone Detection Support Quality Assessments of Requirements Specifications? In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 79–88. ACM Press, 2010. (cited on Page 14)

[JDH09] E. Juergens, F. Deissenboeck, and B. Hummel. CloneDetective - A Workbench for Clone Detection Research. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 603–606. IEEE Computer Society, 2009. (cited on Page 23 and 24)

[JDH10] E. Juergens, F. Deissenboeck, and B. Hummel. Code Similarities Beyond Copy amp; Paste. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 78–87. IEEE Computer Society, 2010. (cited on Page xi, 12, 13, and 110)

[JDHW09] E. Juergens, F. Deissenboeck, B. Hummel, and S. 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 1, 12, and 79)

[JF88] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988. (cited on Page 39)

[JH06] N. Juillerat and B. Hirsbrunner. An Algorithm for Detecting and Removing Clones in Java Code. In *Proceedings of the International Workshop on Software Evolution through Transformations (SeTra)*, pages 63–74, 2006. (cited on Page 32)

[JL06] S. Jarzabek and S. Li. Unifying Clones with a Generative Programming Technique: A Case Study. *Journal of Software Maintenance and Evolution*, 18(4):267–292, 2006. (cited on Page 32)

[JMSG07] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 96–105. IEEE Computer Society, 2007. (cited on Page 21)

[Joh93] J. H. Johnson. Identifying Redundancy in Source Code Using Fingerprints. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 171–183. IBM Press, 1993. (cited on Page 18)

[Joh94a] J. Johnson. Substring Matching for Clone Detection and Change Tracking. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 120–126. IEEE Computer Society, 1994. (cited on Page 18 and 27)

[Joh94b] J. H. Johnson. Visualizing Textual Redundancy in Legacy Source. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pages 32–41. IBM Press, 1994. (cited on Page xi, 23, and 24)

[Joh96] J. H. Johnson. Navigating the Textual Redundancy Web in Legacy Source. In *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, page 16. IBM Press, 1996. (cited on Page 23)

[KA09] C. Kästner and S. Apel. Virtual Separation of Concerns – A Second Chance for Preprocessors. *Journal of Object Technology (JOT)*, 8(6):59–78, 2009. (cited on Page 48 and 75)

[KAB07] C. Kästner, S. Apel, and D. Batory. A Case Study Implementing Features Using AspectJ. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 223–232. IEEE Computer Society, 2007. (cited on Page 48)

[KAK08] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 311–320. ACM Press, 2008. (cited on Page 39, 42, 48, 49, 50, 70, and 74)

[KAK09] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 157–166. ACM Press, 2009. (cited on Page 68)

[KAT+09] C. Kästner, S. Apel, S. Trujillo, M. Kuhlemann, and D. Batory. Guaranteeing Syntactic Correctness for All Product Line Variants: A Language-Independent Approach. In *Proceedings of the International Conference on Objects, Models, Components and Patterns (TOOLS)*, pages 175–194. Springer-Verlag, 2009. (cited on Page 74)

[KATS12] C. Kästner, S. Apel, T. Thüm, and G. Saake. Type Checking Annotation-Based Product Lines. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(3), 2012. to appear; submitted 8 Jun 2010, accepted 4 Jan 2011. (cited on Page 122)

[KAuR+09] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 181–190, 2009. (cited on Page 70)

[KBA09] M. Kuhlemann, D. Batory, and S. Apel. Refactoring Feature Modules. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 106–115. Springer-Verlag, 2009. (cited on Page 95 and 112)

[KCH+90] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 36 and 37)

[KDB+95] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo. Pattern Matching for Design Concept Localization. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 96–103. IEEE Computer Society, 1995. (cited on Page 21)

[KDM+96] K. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein. Pattern Matching for Clone and Concept Detection. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, 3:77–108, 1996. (cited on Page 21)

[Ker88] B. W. Kernighan. *The C Programming Language.* Prentice Hall, 2nd edition, 1988. (cited on Page 73)

[KFF06] R. Koschke, R. Falke, and P. Frenzel. Clone Detection Using Abstract Syntax Suffix Trees. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 253–262. IEEE Computer Society, 2006. (cited on Page 20)

[KG05] C. Kapser and M. Godfrey. Improved Tool Support for the Investigation of Duplication in Software. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 305–314. IEEE Computer Society, 2005. (cited on Page 26)

[KG06] C. Kapser and M. W. Godfrey. "Cloning Considered Harmful" Considered Harmful. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 19–28. IEEE Computer Society, 2006. (cited on Page 1, 28, and 29)

[KG08] C. Kapser and M. Godfrey. "Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software. *Empirical Software Engineering*, 13:645–692, 2008. (cited on Page 29)

[KGR$^+$11] C. Kästner, P. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM Press, 2011. (cited on Page 110, 111, and 122)

[KH01] R. Komondoor and S. Horwitz. Using Slicing to Identify Duplication in Source Code. In P. Cousot, editor, *Static Analysis*, volume 2126 of *Lecture Notes in Computer Science*, pages 40–56. Springer, 2001. (cited on Page 21)

[KKAS11] M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. An Algebra for Refactoring and Feature-Oriented Programming. Technical Report FIN-006-2011, Otto-von-Guericke University Magdeburg, 2011. (cited on Page 91 and 112)

[KKI02] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Code. *IEEE Transactions on Software Engineering (TSE)*, 28(7):654–670, 2002. (cited on Page 11, 15, 19, and 59)

[KLM$^+$97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997. (cited on Page 3, 39, and 47)

[KN01] G. G. Koni-N'Sapu. A Scenario Based Approach for Refactoring Duplicated Code in Object Oriented Systems. Diploma thesis, University of Bern, Switzerland, 2001. (cited on Page 26)

[Kon97] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 44–54. IEEE Computer Society, 1997. (cited on Page 10)

[Kos07] R. Koschke. Survey of Research on Software Clones. In R. Koschke, E. Merlo, and A. Walenstein, editors, *Duplication, Redundancy, and Similarity in Software*, number 06301 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2007. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. (cited on Page 9, 10, and 12)

[Kri01] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 301–309. IEEE Computer Society, 2001. (cited on Page 21 and 70)

[Kri07] J. Krinke. A Study of Consistent and Inconsistent Changes to Code Clones. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 170–178. IEEE Computer Society, 2007. (cited on Page 28)

[Kru02] C. W. Krueger. Easing the Transition to Software Mass Customization. In *Proceedings of the International Workshop on Software Product-Family Engineering (PFE)*, volume 2290 of *Lecture Notes in Computer Science*, pages 178–184. Springer-Verlag, 2002. (cited on Page 3)

[KS94] M. Krone and G. Snelting. On the Inference of Configuration Structures from Source Code. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 49–57, 1994. (cited on Page 3)

[KSNM05] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. An Empirical Study of Code Clone Genealogies. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 187–196. ACM Press, 2005. (cited on Page 28)

[KYU+09] S. Kawaguchi, T. Yamashina, H. Uwano, K. Fushida, Y. Kamei, M. Nagura, and H. Iida. SHINOBI: A Tool for Automatic Code Clone Detection in the IDE. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 313–314. IEEE Computer Society, 2009. (cited on Page 19)

[LAL+10] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines.

In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. ACM Press, 2010. (cited on Page 39, 74, and 88)

[LBC$^+$11] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon. Automated Scheduling for Clone-Based Refactoring Using a Competent GA. *Software: Practice and Experiences*, 41(5):521–550, 2011. (cited on Page 31 and 32)

[LBL06] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 112–121. ACM Press, 2006. (cited on Page 112)

[LHB01] R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, pages 10–24. Springer, 2001. (cited on Page xi and 10)

[LKA11] J. Liebig, C. Kästner, and S. Apel. Analyzing the Discipline of Preprocessor Annotations in 30 Millions Lines of C Code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM Press, 2011. (cited on Page 39, 50, 74, 75, 76, 77, 86, 88, and 111)

[LLMZ06] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Transactions on Software Engineering (TSE)*, 32(3):176–192, march 2006. (cited on Page 12, 19, and 70)

[LM03] F. Lanubile and T. Mallardo. Finding Function Clones in Web Applications. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 379–386. IEEE Computer Society, 2003. (cited on Page 22)

[LMZS06] H. Liu, Z. Ma, L. Zhang, and W. Shao. Detecting Duplications in Sequence Diagrams Based on Suffix Trees. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 269–276. IEEE Computer Society, 2006. (cited on Page 14)

[LSB$^+$10] R. Lotufo, S. She, T. Berger, K. Czarnecki, and A. Wąsowski. Evolution of the Linux kernel variability model. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 136–150. Springer, 2010. (cited on Page 122)

[LST$^+$06a] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. *SIGOPS Operating Systems Review*, 40(4):191–204, April 2006. (cited on Page 3)

[LST$^+$06b] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A Quantitative Analysis of Aspects in the eCos Kernel. In *Proceedings of the SIGOPS European Conference on Computer Systems*, pages 191–204. ACM Press, 2006. (cited on Page 42)

[LT10] H. Li and S. Thompson. Similar Code Detection and Elimination for Erlang Programs. In M. Carro and R. Peña, editors, *Practical Aspects of Declarative Languages*, volume 5937 of *Lecture Notes in Computer Science*, pages 104–118. Springer, 2010. (cited on Page 21)

[LW10] A. Lozano and M. Wermelinger. Tracking Clones' Imprint. In *Proceedings of the International Workshop on Software Clones (IWSC)*, pages 65–72. ACM Press, 2010. (cited on Page 28)

[MA09] D. Malayeri and J. Aldrich. CZ: Multiple Inheritance Without Diamonds. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 21–40. ACM Press, 2009. (cited on Page 45)

[MBKM08] T. Mende, F. Beckwermert, R. Koschke, and G. Meier. Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection. In *Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR)*, pages 163–172. IEEE Computer Society, 2008. (cited on Page 70)

[MCM02] J. Maletic, M. Collard, and A. Marcus. Source Code Files as Structured Documents. In *Proceedings of the International Workshop on Program Comprehension (IWPC)*, pages 289–292. IEEE Computer Society, 2002. (cited on Page 79)

[MF05] M. P. Monteiro and J. M. Fernandes. Towards a Catalog of Aspect-oriented Refactorings. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 111–122. ACM Press, 2005. (cited on Page 100)

[MHQB05] E. R. Murphy-Hill, P. J. Quitslund, and A. P. Black. Removing Duplication From java.io: A Case Study Using Traits. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 282–291. ACM Press, 2005. (cited on Page 32)

[MLM96] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 244–253. IEEE Computer Society, 1996. (cited on Page 5, 10, 21, 24, and 88)

[MLWR01] G. Murphy, A. Lai, R. Walker, and M. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 275–284. IEEE Computer Society, 2001. (cited on Page 49)

[MM01] A. Marcus and J. Maletic. Identification of High-Level Concept Clones in Source Code. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 107–114. IEEE Computer Society, 2001. (cited on Page 18)

[MNK$^+$02] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *Proceedings of the International Symposium on Software Metrics*, pages 87–96. IEEE Computer Society, 2002. (cited on Page 70)

[MRR$^+$12] M. Mondal, C. Roy, M. Rahman, R. Saha, J. Krinke, and K. Schneider. Comparative Stability of Cloned and Non-cloned Code: An Empirical Study. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1–8. ACM Press, 2012. (cited on Page 28)

[NNP$^+$11] H. Nguyen, T. Nguyen, N. Pham, J. Al-Kofahi, and T. Nguyen. Clone Management for Evolving Software. *IEEE Transactions on Software Engineering (TSE)*, PP(99):1–19, 2011. accepted for publication. (cited on Page 21 and 33)

[Opd92] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992. (cited on Page 30 and 91)

[Par72] D. L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972. (cited on Page 46)

[Par76] D. Parnas. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering (TSE)*, SE-2(1):1– 9, 1976. (cited on Page 40)

[Par79] D. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering (TSE)*, SE-5(2):128–138, 1979. (cited on Page 39)

[PBVDL05] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005. (cited on Page 36)

[PCW12] L. Passos, K. Czarnecki, and A. Wasowski. Towards a Catalog of Variability Evolution Patterns: The Linux Kernel Case. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 62–69. ACM Press, 2012. (cited on Page 122)

[PNN+09] N. H. Pham, H. A. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen. Complete and Accurate Clone Detection in Graph-based Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 276–286. IEEE Computer Society, 2009. (cited on Page 14)

[PO97] T. Pearse and P. Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, pages 270–277. IEEE Computer Science, 1997. (cited on Page 3)

[Pre97] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In M. Aksit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer, 1997. (cited on Page 3, 39, and 40)

[RC07] C. Roy and J. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-541, Queen's University at Kingston, 2007. (cited on Page 10 and 12)

[RC08] C. Roy and J. Cordy. NICAD: Accurate Detection of Near-Miss Intentional Clones Using Flexible Pretty-Printing and Code Normalization. In *Proceedings of the International Conference on Program Comprehension (ICPC)*, pages 172–181. IEEE Computer Society, 2008. (cited on Page 12 and 18)

[RC09] C. Roy and J. Cordy. A Mutation/Injection-Based Automatic Framework for Evaluating Code Clone Detection Tools. In *Proceedings of the International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 157–166. IEEE Computer Society, 2009. (cited on Page 16 and 17)

[RC10] C. Roy and J. Cordy. Near-miss Function Clones in Open Source Software: An Empirical Study. *Journal of Software Maintenance and Evolution*, 22(3):165–189, 2010. (cited on Page 1 and 88)

[RCK09] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Science of Computer Programming*, 74(7):470–495, 2009. Special Issue on Program Comprehension (ICPC 2008). (cited on Page 17)

[RDG99] M. Rieger, S. Ducasse, and G. Golomingi. Tool Support for Refactoring Duplicated OO Code. In *ECOOP Workshops*, pages 177–178. Springer, 1999. (cited on Page 31)

[RDL04] M. Rieger, S. Ducasse, and M. Lanza. Insights Into System-Wide Code Duplication. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 100–109. IEEE Computer Society, 2004. (cited on Page 22 and 23)

[Ref09] J. Refstrup. Adapting to Change: Architecture, Processes and Tools: A Closer Look at HP's Experience in Evolving the Owen Software Product Line. In *Proceedings of the International Software Product Line Conference (SPLC)*, 2009. keynote presentation. (cited on Page 3)

[SAK10] S. Schulze, S. Apel, and C. Kästner. Code Clones in Feature-Oriented Software Product Lines. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 103–112. ACM Press, 2010. (cited on Page 7)

[Sam97] J. Sametinger. *Software Engineering with Reusable Components*. Springer-Verlag, 1997. (cited on Page 3)

[SAZ$^+$10] R. Saha, M. Asaduzzaman, M. Zibran, C. Roy, and K. Schneider. Evaluating Code Clone Genealogies at Release Level: An Empirical Study. In *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*, pages 87–96. IEEE Computer Society, 2010. (cited on Page 28)

[SB02] Y. Smaragdakis and D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-based Designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11:215–255, 2002. (cited on Page 3, 39, 40, and 93)

[SBB$^+$10] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 77–91. Springer, 2010. (cited on Page 39)

[SBS$^+$10] G. Selim, L. Barbour, W. Shang, B. Adams, A. Hassan, and Y. Zou. Studying the Impact of Clones on Software Defects. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 13–21. IEEE Computer Society, 2010. (cited on Page 22)

[SC92] H. Spencer and G. Collyer. #ifdef Considered Harmful, or Portability Experience with C News. In *Proceedings of the USENIX Technical Conference*, pages 185–197. USENIX Association Berkeley, 1992. (cited on Page 3, 42, 50, and 74)

[SD10] I. Schaefer and F. Damiani. Pure Delta-Oriented Programming. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 49–56. ACM Press, 2010. (cited on Page 39)

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 2nd edition, 2002. (cited on Page 3)

[Sin94] G. B. Singh. Single versus Multiple Inheritance in Object Oriented Programming. *SIGPLAN OOPS Messenger*, 5(1):34–43, 1994. (cited on Page 45)

[SJF11] S. Schulze, E. Juergens, and J. Feigenspan. Analyzing the Effect of Preprocessor Annotations on Code Clones. In *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*, pages 115–124. IEEE Computer Society, 2011. (cited on Page 7)

[SKR08] S. Schulze, M. Kuhlemann, and M. Rosenmüller. Towards a Refactoring Guideline Using Code Clone Classification. In *Proceedings of the International Workshop on Refactoring Tools (WRT)*, pages 6:1–6:4. ACM Press, 2008. (cited on Page 26, 31, and 32)

[SLB$^+$10] S. She, R. Lotufo, T. Berger, A. Wasowski, and K. Czarnecki. Variability Model of the Linux Kernel. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 45–51. University of Duisburg-Essen, 2010. (cited on Page 3)

[STKS12] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. Variant-Preserving Refactoring in Feature-Oriented Software Product Lines. In *Proceedings of the International Workshop on Variability Modeling in Software-intensive Systems (VaMoS)*, pages 73–81. ACM Press, 2012. (cited on Page 7 and 91)

[Sto10] H. Stoerrle. Towards Clone Detection in UML Domain Models. In *Proceedings of the European Conference on Software Architecture (ECSA)*, pages 285–293. ACM Press, 2010. (cited on Page 14)

[Str95] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, New York, NY, USA, 1995. (cited on Page 74)

[TBG04] M. Toomim, A. Begel, and S. Graham. Managing Duplicated Code with Linked Editing. In *Proceedings of the IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC)*, pages 173–180. IEEE Computer Society, 2004. (cited on Page 33 and 69)

[TBK09] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 254–264. IEEE Computer Society, 2009. (cited on Page 97, 99, and 111)

[TC12] F. Takeyama and S. Chiba. Feature-Oriented Programming with Family Polymorphism. In *Proceedings of the International Workshop on Variability & Composition (VariComp)*, pages 1–6. ACM Press, 2012. (cited on Page 70)

[TCADP10] S. Thummalapenta, L. Cerulo, L. Aversano, and M. Di Penta. An Empirical Study on the Maintenance of Source Code Clones. *Empirical Software Engineering*, 15:1–34, 2010. (cited on Page 28)

[TKES11] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. Abstract Features in Feature Modeling. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 191–200. IEEE Computer Society, 2011. (cited on Page 38)

[TOHS99] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 107–119. ACM Press, 1999. (cited on Page 40)

[TSSPL09] R. Tartler, J. Sincero, W. Schröder-Preikschat, and D. Lohmann. Dead or Alive: Finding Zombie Features in the Linux Kernel. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 81–86, 2009. (cited on Page 3)

[UKKI02a] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. Gemini: Maintenance Support Environment Based on Code Clone Analysis. In *International Software Metrics Symposium (METRICS)*, pages 67–76. IEEE Computer Society, 2002. (cited on Page 22)

[UKKI02b] Y. Ueda, T. Kamiya, S. Kusumoto, and K. Inoue. On Detection of Gapped Code Clones Using Gap Locations. In *Proceedings of the Asia-Pacific Software Engineering Conference (APSEC)*, pages 327 – 336. IEEE Computer Society, 2002. (cited on Page 12)

[URSH11] M. Uddin, C. Roy, K. Schneider, and A. Hindle. On the Effectiveness of Simhash for Detecting Near-Miss Clones in Large Scale Software Systems. In *Proceedings of the Working Conference on Reverse Engineering (WCRE)*, pages 13–22. IEEE Computer Society, 2011. (cited on Page 16, 17, and 18)

[VN96] M. VanHilst and D. Notkin. Using Role Components in Implement Collaboration-Based Designs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 359–369. ACM Press, 1996. (cited on Page 39, 40, and 93)

[WSWF04] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. Clone Detection in Source Code by Frequent Itemset Techniques. In *Proceedings of the Working*

Conference on Source Code Manipulation and Analysis (SCAM), pages 128–135. IEEE Computer Society, 2004. (cited on Page 20)

[Yan91] W. Yang. Identifying Syntactic Differences Between Two Programs. *Software: Practice and Experiences*, 21(7):739–755, 1991. (cited on Page 20)

[YHK$^+$05] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. On Refactoring Support Based on Code Clone Dependency Relation. In *International Software Metrics Symposium (METRICS)*, pages 10–16. IEEE Computer Society, 2005. (cited on Page 31 and 32)

[ZR11] M. Zibran and C. Roy. A Constraint Programming Approach to Conflict-Aware Optimal Scheduling of Prioritized Code Clone Refactoring. In *Proceedings of the Working Conference on Source Code Manipulation and Analysis (SCAM)*, pages 105–114. IEEE Computer Society, 2011. (cited on Page 31 and 32)

[ZR12] M. Zibran and C. Roy. IDE-based Real-Time Focused Search for Near-Miss Clones. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1–8. ACM Press, 2012. (cited on Page 16, 17, and 18)

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

Magdeburg, den