Code Clones in Feature-Oriented Software Product Lines

Sandro Schulze
University of Magdeburg, Germany
sanschul@iti.cs.uni-magdeburg.de

Sven Apel
University of Passau, Germany
apel@uni-passau.de

Christian Kästner
Philipps University Marburg, Germany
kaestner@informatik.uni-marburg.de

Abstract

Some limitations of object-oriented mechanisms are known to cause code clones (e.g., extension using inheritance). Novel programming paradigms such as feature-oriented programming (FOP) aim at alleviating these limitations. However, it is an open issue whether FOP is really able to avoid code clones or whether it even facilitates (FOP-related) clones. To address this issue, we conduct an empirical analysis on ten feature-oriented software product lines with respect to code cloning. We found that there is a considerable number of clones in feature-oriented software product lines and that a large fraction of these clones is FOP-related (i.e., caused by limitations of feature-oriented mechanisms). Based on our results, we initiate a discussion on the reasons for FOP-related clones and on how to cope with them. We show by means of examples how such clones can be removed by applying refactorings.

Categories and Subject Descriptors D.2.3 [*SOFTWARE ENGINEERING*]: Coding Tools and Techniques; D.2.9 [*SOFTWARE ENGINEERING*]: Management; D.2.13 [*SOFTWARE ENGINEERING*]: Reusable Software

General Terms Design, languages, measurements

Keywords Code clones, software product lines, feature-oriented programming, refactoring

1. Introduction

Code cloning, that is, the replication of code fragments in source code, is known to be a serious and common problem in object-oriented programming (*OOP*) [6, 10, 35]. Numerous studies have shown that code clones occur frequently in software systems, ranging from 7 % to 23 % [6, 10, 21, 26, 32]. Although recent studies discuss the harmfulness of code cloning controversially, it is widely accepted that code clones have a negative effect on the software structure, in terms of a decrease of maintainability and the introduction of errors [10, 23, 35]. The causes for code cloning have been widely discussed and include, amongst others, shortcomings of the host language, which force the programmer to introduce code clones without any alternative.

Recently, novel programming paradigms such as *feature-oriented programming (FOP)* gained attention, which aim at overcoming certain limitations of OOP. In the context of FOP, a feature is an increment in program functionality. The functionality encompassed by a feature is encapsulated in a cohesive unit called *feature module*. Feature modules can be used to create a *software product line (SPL)*. Typically, an SPL consists of a set of features.[1] A user-defined subset of these features is composed to synthesize a tailored product. In comparison to OOP, FOP aims at a stronger cohesion and reusability of feature code. However, so far, no studies have been conducted to explore the presence and effects of code clones in feature-oriented software product lines. Based on our experience with SPL development, we assume that, although FOP has the potential to avoid OOP-related code clones, it may introduce FOP-related clones.

We want to shed light on the issue of code cloning in FOP and raise a number of research questions: Do code clones exist in feature-oriented SPLs? Is FOP prone to introduce FOP-related clones, especially in the context of SPLs? Does it matter whether the SPL was developed from scratch or refactored from a legacy application? And finally, what can we do against code clones in feature-oriented SPLs?

To answer these questions, we perform a code clone analysis of ten feature-oriented SPLs, six implemented from scratch and four refactored from legacy applications. We use the established clone detection tool *CCFinder*[2] [24] to obtain the code clones for the respective product lines. In particular, we make the following contributions:

- We perform a conceptual analysis of code clones in FOP. We identify limitations of FOP that are likely to introduce code clones. For example, the coarse-grained granularity of extensions in FOP is a limitation that may lead to code clones. We show that a considerable number of (FOP-related) code clones actually exist in the SPLs we analyzed.
- By means of a case study on ten different non-trivial feature-oriented SPLs, we analyze the amount 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 legacy code.
- Based on the results of our analysis, we discuss possibilities of removing clones by means of refactoring without breaking the validity of the underlying SPL. We show that code clones can be removed (in parts) through the application of refactorings that are tailored to feature-oriented SPLs.
- We initiate a discussion on the existence and effect of code clones in FOP.

GPCE'10, October 10–13, 2010, Eindhoven, The Netherlands.

[1] The terms feature and feature module are used synonymous for the remaining paper, because there is a 1:1 mapping between problem and solution space in FOP.

[2] http://www.ccfinder.net

2. Background

2.1 Code Clones

We give a short overview of research on code clones to lay the foundation for the remaining sections. Code clones are argued to have a negative effect the software structure. Amongst others, increased maintenance costs and an increased probability of bugs are the most devastating consequences [37, 43].

Code clones are classified into four categories: *type-I* (identical), *type-II* (similar), *type-III* (statements are removed, added or changed) and *type-IV* (semantically similar) clones. In Figure 1, we give an example for a *type-II* clone pair, which we detected in one of our case studies, the *graph product line (GPL)* [34]. The two code fragments differ only in the variable names `urep` and `vrep` (Lines 4–8 and Lines 10–15). Furthermore, corresponding code clones can be merged to *clone classes* (e.g., by establishing an equivalence relation between them). Such clone classes are useful for code clone analysis or removal since they allow to treat related clones as a unit.

Several approaches for the detection of clones exist, in particular text-based [6, 17], token-based [24, 32], tree-based [10, 31] and metric-based [30, 35] clone detection. For detailed information on clone detection techniques and tools we refer to Roy et al. [44].

The result of clone detection can be used for further processing. Generally, two approaches exist: code clone removal and code clone management, of which the latter implies that the clones remain in the program. For code clone removal, on which we concentrate in this paper, we carry out an analysis of the detected clones to obtain information on the clones, useful for their removal (e.g., identifying *clone classes* or determining the clone type). Then, clones are removed using *refactorings*, that is, restructuring the program in a behavior-preserving way [18]. There are several approaches, in which different refactorings are applied manually or (semi-)automatically to remove clones [7, 19].

```
1  class Graph { /*...*/
2    public Graph Kruskal {
3      // some code
4      for(int j=0;j<(vrep.members).size();j++) {
5        vaux = (Vertex) (vrep.members).get(j);
6        vaux.representative = urep;
7        (urep.members).add(vaux);
8      }
9      /*...*/
10     for(int j=0;j<(urep.members).size();j++) {
11       vaux = (Vertex) (urep.members).get(j);
12       vaux.representative = vrep;
13       (vrep.members).add(vaux);
14     }
15   }
16 }
```

Figure 1. An example of a *type-II* code clone, taken from the *graph product line (GPL)* [34]

2.2 Feature-Oriented Software Product Lines

An SPL is a set of software-intensive systems that share a common, managed set of features [13]. In this context, a feature is an increment in functionality that implements a stakeholder's requirement [9, 42]. To generate a concrete program from an SPL, the user selects the desired features – typically a subset of all features of the SPL – and a generator generates the tailored program (a.k.a. *variant*). Different variants of an SPL contain common as well as different features.

The variability of an SPL is described by a *feature model* [25]. A feature model defines the features of an SPL and their dependencies. In Figure 2, we depict a graphical feature model of a

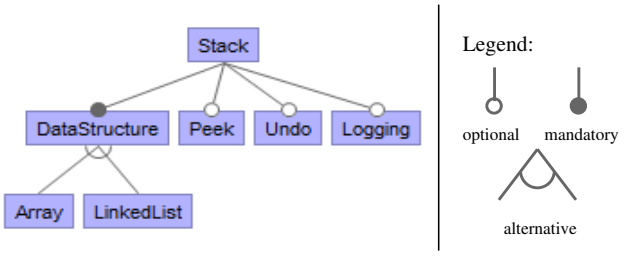


Figure 2. Feature model of the *Stack* SPL

Stack product line. The root of the model represents the SPL itself, whereas the remaining nodes represent individual features. Features can be optional or mandatory and, beyond that, can be grouped as alternative features. Optional and alternative features are used to express variability in an SPL whereas mandatory features express commonalities across all variants.

There are different approaches to implement SPLs, e.g., frameworks [22] or conditional compilation [27, 41]. Here, we use FOP, a language-based approach (a.k.a *compositional approach*) that gained much attention in the past and that aims at the modularization of a program by decomposing it according to *Features*. Several FOP languages and tools exist, such as AHEAD [9], FeatureHouse [5] or FeatureC++ [3], which provide various mechanisms for the modularization and composition of features. The core idea is that the functionality encompassed by a feature is implemented by a cohesive unit called *feature module*. As a result, we obtain a clean relationship between the feature and its implementation unit. The increment of functionality, introduced by a feature, is realized by adding new structures such as classes or methods and refining existing ones, such as extending a method.

Feature *Stack*

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

Feature *Peek*

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

Feature *Undo*

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

Figure 3. Feature-oriented implementation of *Stack* with features *Peek* and *Undo*

In Figure 3, we show three feature modules of our *Stack* product line. Feature *Stack* is the basic feature of our product line, in which class `Stack` is initially declared. The two features *Peek* and *Undo* refine this class to add functionality (indicated by `'refines class...'`). Feature *Peek* introduces a new method to copy the topmost element of the stack. Feature *Undo* introduces a method `undo` and extends the existing method `push` by an additional statement at the beginning so that the topmost element is assigned to a backup variable. Keyword `original` invokes method `push` of the original class `Stack`. We use operator '•' to denote the composition of features to a program. Although, in our example, each feature contains only one class, typically multiple classes and class refinements belong to a feature.

3. Reasoning about Code Clones in Feature-Oriented SPLs

Limitations of (object-oriented) programming languages are discussed as a reason for code clones [8, 43]. Mechanisms such as inheritance or generics are not always sufficient for reusing functionality or *expressing variability* in programs and thus contribute to code cloning. By contrast, novel programming paradigms such as FOP provide mechanisms to overcome these limitations. We illustrate the problem of expressing variability with our *Stack* product line example.

Composing the three features shown in Figure 3, we can generate four different, meaningful combinations of programs (variants): *Stack*, *Peek* • *Stack*, *Undo* • *Stack*, and *Undo* • *Peek* • *Stack*. Implementing these four variants in OOP requires four separate classes, one for each program, which we show in Figure 4. Creating the three programs `Stack`, `PeekStack`, and `UndoStack` can be done by using simple inheritance without any code clone activity. However, the creation of the fourth program, `UndoPeekStack`, would require to inherit from two classes, `PeekStack` and `UndoStack`. Since this is not possible in many OOP languages (*diamond problem*), code has to be cloned contemporary to reuse functionality in class `UndoPeekStack`. By contrast, with our feature-oriented implementation in Figure 3 we overcome these limitations by means of class refinements and thus no code clones occur. Hence, we conclude that FOP *can* help to avoid code clones.

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

```
class PeekStack
  extends Stack{
  int peek() {/*...*/}
}
```

```
class UndoStack
  extends Stack { ...
  int backupPush;
  void undo() {/*...*/}
  void push(int v) {
    backupPush=v;
    original(v);
  }
}
```

```
class UndoPeekStack
  extends PeekStack { ...
  int backupPush;
  int peek() {/*...*/}
  void undo() {/*...*/}
  void push(int v) {
    backupPush=v;
    original(v);
  }
}
```

Figure 4. Object-oriented implementation of *Stack* with features *Peek* and *Undo*

At the same time, FOP may have limitations that contribute to code cloning as well, such as alternative features. To evaluate the existence and effect of code clones in feature-oriented SPLs, we discuss the key mechanisms of FOP and analyze how they possibly foster code cloning. We begin with a conceptual analysis independent of a particular FOP language, and identify (conceptual) limitations of FOP that may tempt the programmer to introduce code clones. Having knowledge on such weaknesses can help to deal with clones or even to avoid code clones in advance (e.g., by rethinking the design of the FOP language). Furthermore, we use the results of our conceptual analysis for structuring our empirical analysis.

Separation of concerns. Features are concerns relevant in SPL engineering. As stated by Parnas [40], it is imperative to separate concerns. But some concerns (or rather their implementations) are inherently tangled with and scattered across other concerns, commonly referred to as *crosscutting concerns* [29]. Hence, it is difficult to encapsulate them in separate modules. Although FOP is able to modularize certain kinds of crosscutting concerns using feature modules, it has weaknesses to separate others, especially homogeneous crosscutting concerns [4]. Homogeneous crosscutting concerns extend a program at several points by the same piece of code [15]. While studies show that this kind of crosscutting concern occurs rarely compared to others, it still may introduce a comparatively large number of code clones [2].

Granularity of extensions. A feature extends existing program structures by other features. Extensions can be carried out at different levels of granularity [27] (e.g., extending classes or methods). For instance, in our *Stack* product line in Figure 3, we extend the method `push` in feature *Undo* by a statement at the beginning of the method. While FOP (and other compositional approaches) work fine for coarse-grained extensions, it has limitations when realizing fine-grained extensions, such as extensions at statement level [27, 39]. For instance, extending a program by adding statements in the middle of an existing method is only possible with FOP with cumbersome boilerplate code. As an example, we show in Figure 5 two methods of two features, breadth-first search *(BFS)* and depth-first search *(DFS)*, of *GPL* [34]. Both methods differ only in the called method in Line 18 and thus, these methods cannot be easily factored out in a common piece of code. Furthermore, extending a program by adding a parameter to a method's signature is not possible either with FOP [27]. Such limitations are likely to cause code clones across features since the respective methods have to be reimplemented in features with only minor changes, as can be seen in Figure 5.

Alternative features. In particular, alternative features, i.e., features that are mutually exclusive, may lead to code clones. A main characteristic of alternative features is that they are often similar, differing only in some points. Structured reuse of the common code of such features, e.g., by class refinements, is not possible without further workarounds (e.g., additional features, boilerplate code) due to their alternative nature. Consequently, the respective functionality has to be implemented for each feature separately. For instance, the two features *BFS* and *DFS* of Figure 5 are alternative features. Hence, the same interface has to be implemented in both features separately. As a result, both features share a large portion of identical code and differ only in one particular statement.

Restructuring features. As usual (OO) programs, SPLs are subject to software evolution. This means that features are extended due to new requirements and thus grow over time. This may lead to the point at which a feature contains more functionality than it was designed for. As a result, the feature has to be split and the corresponding code has to be extracted from the existing feature into the new one using *Cut&Paste*. However, since this code is often tightly coupled with the remaining code of the existing feature, it might be unavoidable to reuse parts of this code by *Copy&Paste*, which may cause code clones.

All of the discussed limitations may introduce code clones. We refer to them as *FOP-related clones*, because they originate from the limitations and concepts of FOP. However, since FOP is based on OOP, other code clones, originating from the latter, may occur as well in feature-oriented SPLs. For instance, the presence of replicated code fragments within a class or a file may not be specific to FOP. Hence, we refer to these code clones as *OOP-related clones* in the remainder of the paper.

4. Empirical Analysis of Ten Feature-Oriented SPLs

To explore to what extent code clones exist in feature-oriented SPLs, we conducted an analysis on ten different feature-oriented SPLs. In this section we describe the setup, the methodology and results of our analysis. Furthermore, we discuss the results and threats to validity.

Feature *BFS*

```
1  public class Graph
2  {
3    public void search(WorkSpace w)
4      {
5          VertexIter vxiter = getVertices();
6          if (vxiter.hasNext() == false) return;
7          while (vxiter.hasNext())
8          {
9              Vertex v = vxiter.next();
10             v.init_vertex(w);
11         }
12         for (vxiter = getVertices(); vxiter.hasNext();)
13         {
14             Vertex v = vxiter.next();
15             if (!v.visited)
16             {
17                 w.nextRegionAction(v);
18                 v.bfSearch(w);
19             }
20         } //end for bfsSearch
21     }
22 }
```

Feature *DFS*

```
1  public class Graph
2  {
3      public void search(WorkSpace w)
4      {
5          VertexIter vxiter = getVertices();
6          if (vxiter.hasNext() == false) return;
7          while (vxiter.hasNext())
8          {
9              Vertex v = vxiter.next();
10             v.init_vertex(w);
11         }
12         for (vxiter = getVertices(); vxiter.hasNext();)
13         {
14             Vertex v = vxiter.next();
15             if (!v.visited)
16             {
17                 w.nextRegionAction(v);
18                 v.dfSearch(w);
19             }
20         }
21     }
22 }
```

Figure 5. Code clones between features *BFS* and *DFS* in *GPL* (only Line 18 differs)

4.1 Prerequisites

The subjects of our analysis are ten feature-oriented SPLs of different size (150 to 45000 SLOC[3]). All SPLs were developed with FOP tools based on Java, namely FeatureHouse [5] and AHEAD [9]. Furthermore, the selected feature-oriented SPLs stem from different domains such as database systems, editors, and mobile games. We list them in Table 1. The programs in the upper half of Table 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. We consider the whole code base of the feature-oriented SPLs rather than certain variants, because this way 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.[4]

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

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

We performed clone detection on the selected SPLs using the token-based clone detection tool *CCFinder* [24]. We decided to use CCFinder because of its high recall and a relatively high precision (i.e., only few false positive code clones are detected) [11]. As result, we can ensure that we do not miss any code clones in the analyzed SPLs. Within CCFinder, the user can specify different parameters such as minimum clone length. Guided by a former study that used CCFinder [12], we set the minimum clone length to five lines of code. 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. Consequently, 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[5] such as removing comments or whitespaces. We list the results of the whole analysis in Table 2 and explain the different parts of this Table in the remaining section.

name	SLOC	FM	domain
GPL[1]	1 929	28	graph and algorithm library
GUIDSL[2]	11 527	29	graphical configuration tool
Notepad[3]	1 012	13	graphical text editor
PKJab[4]	3 305	8	instant messaging client
TankWar[5]	4 933	38	shoot 'em up game
EPL[6]	149	11	arithmetic expression evaluator
Berkeley DB[7]	45 000	100	transactional storage engine
MobileMedia[8]	4 227	47	multimedia management
Violet[9]	7 194	88	graphical model editor
Prevayler[10]	5 270	6	persistence library

developed by [1]R. Lopez-Herrejon (UT Austin), [2]D. Batory (UT Austin), [3]A. Quark (UT Austin), [4]P. Wendler (U Passau), [5]L. Lei et al. (U Magdeburg), [6]R. Lopez-Herrejon (UT Austin)

refactored by [7]C. Kästner (U Magdeburg), [8]C. Kästner (U Magdeburg), [9]A. Kampasi (UT Austin), [10]J. Liu (UT Austin)

FM: feature modules; SLOC: source lines of code

Table 1. Overview of the analyzed SPLs

4.2 Code Clone Analysis Methodology

The code clone analysis process consits of three steps: clone detection, syntactical classification, and feature-related classification. In the following, we explain each step and present the main results. The clone ratio, given at several points in this (and the following) subsection is always related to the total amount of code (SLOC) for each SPL.

Clone detection. In Table 2 (a), we show the results of the clone detection. The result of the initial clone detection process shows that the considered SPLs exhibit large portion of clones. For instance, in *TankWar*, we detected 1000 SLOC of clones, which are 20 % of the total code base. Nevertheless, there may be clones which occur incidental and thus are meaningless for our analysis (e.g., a sequence of variable declarations). Furthermore, at this point, it is still unclear whether the clones are FOP-related or OOP-related. Hence, we perform a more detailed analysis to gain more information on the detected clones.

Syntactical classification. First of all, we classify the initial clone classes by their *syntactic category*, that is, their relation to cer-

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

	(a) clone detection		(b) syntactical classification						(c) feature-related classification			
	$\mathrm{SLOC}_{\mathrm{clones}}$	$\mathrm{CR}_{\mathrm{cd}}$	IS	FS	WS	MD	TD	$\mathrm{CR}_{\mathrm{sc}}$	CPF/A	CDF	$\mathrm{SLOC}_{\mathrm{clones}}$	$\mathrm{CR}_{\mathrm{FOP}}$
GPL	731	37	0	14	0	480	185	35	652/652	0	652	34
GUIDSL	900	7	20	0	24	403	443	7	98/98	0	98	1
Notepad	291	28	130	0	0	18	58	20	130/0	0	130	12
PKJab	203	6	18	12	0	0	90	3	0/0	0	0	0
TankWar	1000	20	132	4	0	371	242	15	662/634	0	680	13
EPL	18	12	0	0	0	0	18	12	18/18	0	18	12
BerkeleyDB	952	2	65	0	0	436	207	2	69/69	67	357	<1
MobileMedia	716	16	24	0	0	46	482	13	16/16	135	305	7
Violet	784	11	56	24	0	30	315	6	220/162	108	328	5
Prevayler	131	2	6	0	0	62	58	2	0/0	0	16	<1

CR: clone ratio (in %) for clone detection (cd), syntactical classification (sc), and feature-related classification (FOP); IS: IfStatement; FS: ForStatement; WS: WhileStatement; DS: DoStatement; MD: MethodDeclaration; TD: TypeDeclaration; CPF/A: clones with common parent feature/portion of clones from alternative features; CDF: clones with common dependency feature;

Table 2. Statistics of clone detection and analysis

tain syntactical elements (e.g., statements, expressions, ...). Afterwards, we select the clone classes that we classify into one of the following categories with the obvious meanings: *IfStatement*, *ForStatement*, *WhileStatement*, *MethodDeclaration*, and *TypeDeclaration*. We do this for two reasons: First, these categories indicate enclosing blocks (e.g., `for` loops) that encapsulate a semantically coherent piece of functionality. Hence, such blocks may result from explicit cloning activity (e.g., by *Copy&Paste*) rather than occur incidental. Second, these categories provide good refactoring opportunities for code clone removal, for instance, by applying *Extract Method* or *Pull Up Method* refactorings, tailored to SPLs [18]. All clone classes with different syntactic categories are filtered out and not considered for further analysis.

We list the results of this classification in Table 2 (b). For each category, we give the number of clones (SLOC). In the last column, we provide the clone ratio for each of the considered SPLs (column $\mathrm{CR}_{\mathrm{sc}}$). For instance, *TankWar* has the following number of code clones: 371 SLOC related to method declarations, 242 SLOC related to type declarations, and 132 SLOC related to `if` statements. Overall, this product line has a clone ratio of 15 % considering only syntactical clones of interest, i.e., there are some clones that are not related to the considered syntactical categories.

Feature-related classification. In the second step of our analysis, we identify the nature of code clones, that is, whether they are FOP-related or not. This step is based on the syntactical classification and thus only syntactical clones of our interest are considered. For determining their nature, we have to consider corresponding code clones as a whole. Thus, we analyze the clone classes created after clone detection and selected for further analysis by syntactic classification. First of all, we define the following condition: A clone class is FOP-related, if its member clones affect *at least two features*, that is, the corresponding clones must occur in at least two different features. Code clones of a clone class that affect only one feature, occur within one class or between different classes of a single feature. Since this kind of cloning happens in a usual OOP program as well, we assume that these clone classes are OOP-related. These clones are relevant too, but outside the scope of this paper.

After the classification, we analyzed the FOP-related clones to gain information on the causes of cloning as well as to identify possible refactorings applicable for code clone removal. Specifically, we analyzed whether the detected clones cover features that have a common parent, are mutually exclusive, or are in a parent-child relationship. We discuss the concrete correlation between feature relationship, obtained by this analysis step, and refactoring in Section 5.

We show the result of the whole second analysis step in Table 2 (c). The respective columns contain the results of our analysis. In column $\mathrm{SLOC}_{\mathrm{clones}}$, we list the total number of clones between multiple features. Column *CPF/A* indicates the SLOC (of clones) that occur in features with a common, direct parent feature and how many of these clones occur in alternative features. Furthermore, we list how many clones (using the SLOC metric) occur in features that are in a parent-child relationship (column *CDF*). We describe such dependencies in Section 5 in detail. Finally, we list the clone ratio of the remaining, FOP-related clones in column $\mathrm{CR}_{\mathrm{FOP}}$.

For instance, *TankWar* contains 680 lines of FOP-related code clones (column $\mathrm{SLOC}_{\mathrm{clones}}$), which are 13 % compared to the whole code size (column $\mathrm{CR}_{\mathrm{FOP}}$). The first part of column *CPF/A* indicates that 662 lines of code clones exist in features with a common, direct parent feature. The second part of this column indicates that 634 lines of code clones exist in alternative features. Furthermore, no code clones exists in features with a parent-child relationship (column *CDF*). In the following subsection we have a closer look at the results.

4.3 Results

During our analysis, we collected various data. Next, we describe our results from Table 2. For a discussion and interpretation of the results refer to Section 4.4. We structure our description according to our analysis steps and examine the differences that may result from the different development process of our SPLs.

We calculated the percentage of the average and the standard deviation ($a \pm s$) on the clone ratio of all considered SPLs. Next, we state the results for our three analysis steps, that is, clone detection, syntactic classification and feature-related classification. Additionally, we set our focus on how the results depend on the development process of the analyzed SPLs.

Number of code clones. The results of our initial clone detection reveal that there is a significant number of clones in feature-oriented SPLs (cf. Table 2 (a)). 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*) have the highest clone ratio values with 37 % and 28 % respectively. By contrast, the two largest SPLs (*Berkely DB* and *GUIDSL*) are among those with the lowest clone ratio value.

Refactorable clones. With our first analysis step, we aimed at detecting clones that could be target to refactorings because of their syntactic characteristics. The data (cf. Table 2 (b)) reveal that there is still a huge number of clones that may be removable indicated by a total clone ratio of $12 \pm 9\,\%$. In addition, we observed the clone ratio decreases in comparison to the initial clone detection in almost all SPLs. This means that not all code clones exhibit syntactical characteristics that are useful for refactoring by our means. We observed further that the clone classes, filtered out by the syntactical classification, mainly fall into the three categories *IfStatement, MethodDeclaration*, and *TypeDeclaration*. Particularly, we noticed the high number of code clones in category *TypeDeclaration*, which means that whole classes have been cloned.

FOP-related clones. The data resulting from the last analysis step (cf. Table 2 (c)), reveal, that there are code clones that are FOP-related by our definition. Nevertheless, we observed that four feature-oriented SPLs (*GUIDSL*, *PKJab*, *Berkeley DB*, and *Prevayler*) contain (almost) no FOP-related clones. We assume that this results from the fact, that these SPLs have the lowest clone ratio even in the initial clone detection (cf. Table 2 (a)) and that the existing clones are OOP-related. Generally, we observed that the clone ratio is considerable lower than the clone ratio after syntactical classification for all SPLs except of *GPL*. Regarding all SPLs, the ratio of FOP-related clones is $9\pm9\,\%$ and of OOP-related clones is$3 \pm 2\,\%$. This reveals that the analyzed SPLs contain more FOP-related clones and that there is a high diversity of the FOP-related clone ratio of the several SPLs. Actually, four SPLs have an FOP-related clone ratio greater than 10 %, whereas the clone ratio of the remaining SPLs is less than 8 %. Finally, we observed that the FOP-related clones mostly have a common parent feature and are distributed over alternative features (column *CPF/A* in Table 2 (c)). Only in three SPLs (*MobileMedia*, *BerkeleyDB*, and *Violet*), clones are contained in features that are in a parent-child relationship (instead of having a common parent feature).

From scratch vs. refactored. Considering all of the data we collected, we observed one peculiarity: Throughout all analysis steps, the number of clones in SPLs developed from scratch is higher than in SPLs decomposed from legacy applications. This is also indicated by the number of clones that is higher for SPLs from scratch ($19 \pm 12\,\%$ in Table 2 (a), $12 \pm 12\,\%$ in Table 2 (c)) than for SPLs from legacy applications ($10\pm5\,\%$ in Table 2 (a), $5\pm4\,\%$ in Table 2 (c)). The SPLs refactored from legacy applications were developed originally object-oriented and finally, were decomposed manually or automatically [28]. Beyond that, the data reveal that the diversity of clone ratios between the individual SPLs from scratch is very high, which indicates that there are even differences amongst those SPLs. Finally, we observed that both kinds of SPLs contain FOP-related as well as OOP-related clones.

4.4 Discussion

Next, we discuss the results of our case study regarding the research questions raised in the introductory section.

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

Is FOP prone to introduce FOP-related clones in SPLs? Our results indicate that there are FOP-related clones in the analyzed SPLs. An interesting observation is that the majority of these FOP-related clones occur between alternative features. This observation coincides with the limitations of FOP analyzed by us and other researchers before. But this observation indicates that there is a high potential for code clone removal, because we can possibly extract the respective clones into a separate but shared feature (cf. Section 5). In contrast, we also detected clones that are FOP-related, which do not occur in alternative features. Considering our data, we can not clearly infer why these clones occur. But even without these clones it is a matter of fact that feature-oriented SPLs contain FOP-related clones.

Does the development process of the SPL influence code cloning? Our results show differences between the analyzed SPLs that can be ascribed to the development process (from scratch vs. refactored). The SPLs developed from scratch contain a significant higher number of clones than the SPLs refactored from legacy applications. In Particular, we observe a relation between FOP-related clones and alternative features. Indeed, the SPLs from scratch have a considerable amount of alternative features while the decomposed SPLs have not. We conclude that (a) alternative features especially lead to code clones and (b) the SPLs from scratch contain a higher number of this kind of features. This observation is supported by the fact that the SPLs refactored from legacy applications were not designed with variability in mind and thus contained no or only few alternatives before decomposition. As a result, the variability was mostly introduced by optional features during the decomposition process.

Another reason may be that the programmers of the SPLs from scratch were not capable to exploit all concepts and mechanisms of FOP (as often observed with new programming paradigms). 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 is that a huge number of clones between alternative features are across method declarations. These clones can be refactored by pulling them up to the common parent feature. We will have a closer look to concrete refactorings in Section 5.

However, our data do not reveal information on the concrete number of actual refactorable clones. For instance, two cloned methods may be similar except of one statement, that differs in the call to another method as in Figure 5. As a result, it will be hard to apply a refactoring such as *Pull Up Method* to these clones. This requires a more detailed analysis, which we present exemplarily for one of the considered SPLs in Section 5. Besides refactoring, other possibilities exist for managing clones, we did not consider in our analysis, such as clone tracking [16] or linked editing [46]. 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. Apart from that, we believe that code clone removal, if applicable, is the most suitable practice for dealing with clones.

4.5 Threats to validity.

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 Java, C++, Haskell, and UML) that may lead to different implementations for feature-oriented SPLs. In this paper, we focused only on FOP languages based on Java, so the results of our analysis are comparable. These FOP languages are considered to be most ma-

ture and a sufficient amount of case studies exist compared to other FOP languages. However, the classification we made along with our analysis is in theory also valid for other languages, e.g., C++ or C#. Although no empirical evaluation for other languages yet exists, we assume that our analysis is independent of the underlying language.

Selected SPLs. A major problem with case studies is that the selected programs may be biased an thus the results are not generalizable. In addition, only few SPLs exist that are implemented using FOP (based on Java), which might reduce the validity of our study. To address this problem, we considered all feature-oriented SPLs that we were able to locate for our study. Beyond that, the selected SPLs are from different domains and of different size. Nevertheless, one problem remains: All of the analyzed SPLs are prototypical implementations from academia. Hence, there is a lack of comparable results of SPLs with industrial strength, which is also caused by the fact that such systems do not exist for FOP yet. Nevertheless, the considered SPLs have been implemented by different authors and for other purposes and *prior* to and independent of our analysis. Considering the SPLs refactored from legacy applications it is worth to mention that these SPLs were refactored without code clones in mind. Hence, we can definitively exclude that the results of our empirical analysis are biased because of code clone awareness.

Classification of FOP-related clones. During our analysis, we proposed a classification for FOP-related 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-related 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-related. Beyond that, our condition for FOP-related clones is rather a criterion that can be used to omit clones that are clearly OOP-related. As a result, this condition can not ensure that FOP-related clones are caused by FOP exclusively. Hence, we should refine this condition to be more restrictive in our classification of what an FOP-related clone is. However, we defined a lower bound with our definition of what an FOP-related code clone is, which can be used as a base for future work.

5. Removal of (FOP-related) Code Clones

During our empirical analysis, we looked for refactoring opportunities for removing the detected (FOP-related) clones. In this section, we discuss how the analysis results can be used for deriving concrete refactorings. Subsequently, we apply these refactorings exemplarily to one of our case studies, the *TankWar* product line.

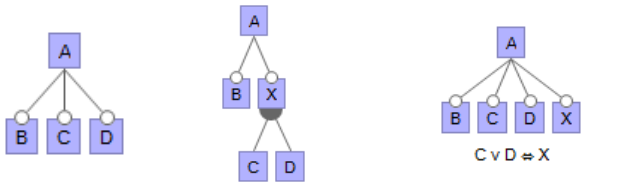
5.1 Extracting Code Clones from Features

There are different possibilities to remove FOP-related clone classes. The general pattern is that we want to 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.

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 (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 (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 (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) [45].

The pattern of moving cloned code to a single new location works uniformly for different kind of clones: cloned types, cloned methods and others.



(a) Original (b) Alternative #1 (c) Alternative #2

Figure 7. Feature model edits for code clone removal

5.2 Exemplary Refactoring of FOP-related Code Clones

To evaluate the applicability of refactorings to FOP-related clones, we performed code clone removal by manually applying refactorings for one of our case studies: *TankWar*. TankWar is a shoot 'em up game, running on PC and mobile phones, that was developed by students of the University of Magdeburg. We selected this product line, because it has a high clone ratio, a relatively high number of FOP-related clones, and it is of medium size (approx. 5 000 SLOC). The game was developed as a product line because it must adhere to strong portability requirements, as common for mobile games [1]. 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 and computing power 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 6, we show the feature model of the TankWar product line in which features such as *Image* and *Sound* are specific for different platforms. Furthermore, we can see several alternative feature groups and, according to our analysis (cf. Table 2 (c)), these features contain a large portion of code clones. Hence, we expected that removing these clones by refactoring is very promising.

Following our analysis results, we identified fourteen clone classes, which contain potentially refactorable clones. We list these clone classes in Table 3, 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). After a first review of these clone classes, we declared four clone classes (#11 – #14) as "not refactorable" for the following reasons: Three of them (#11 – #13), contained in different features below the feature *Tools*, consist of *type-II* clones (i.e., they differ in constants, variable names, etc.) 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

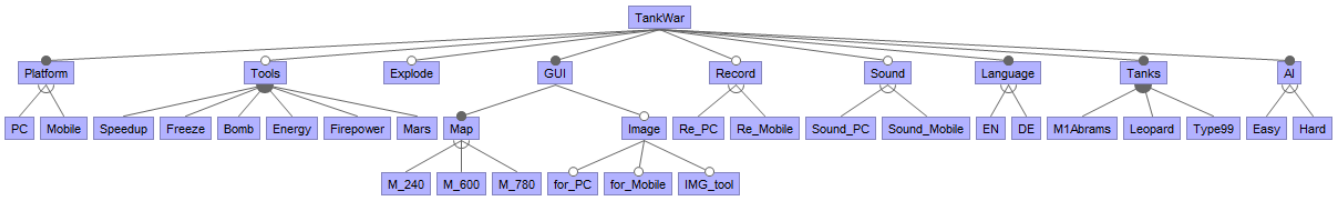


Figure 6. Feature model of the *TankWar* product line

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

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 3. Overview of clone classes removed by refactorings

clone class (#14) consist of *type-III* clones with notable differences such that a refactoring was not applicable. Beyond this, the clones of these four clone classes are scattered across 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 where finally subject of our refactoring process, we made the following 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 syntactic categories (*IfStatement, MethodDeclaration, TypeDeclaration*), which coincides with our observations in Section 4. Furthermore, we made some observations, which we did not list in the table due to space restrictions. First, all member clones (i.e., clones of a single clone class) have a common, direct parent feature. Second, clone classes with syntactic category *TypeDeclaration (TD)* contained 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 consist of *type-II* clones.

For the actual refactoring process, we had to tailor certain object-oriented refactorings to FOP. We call such refactorings *feature-oriented refactorings*.[6] For instance, we tailored the *Pull Up Method* refactoring such that it operates on the level of features instead of classes. As a result, the application of this refactoring in the context of feature-oriented SPLs means that the respective method is moved from the current feature to its parent feature. In the same way, weadapted the *Pull Up Constructor Body* [18]. In contrast, we used the *Extract Method* refactoring in its original form, 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 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 (feature-oriented) 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 4. This lead to the following results: The number of code clones has been decreased throughout all analysis steps. Regarding the initial clone detection, the number of clones decreased from 20 % (cf. Table 2 (a)) to 12 %. For the syntactic classification, the number of code clones is reduced by the half to 7 %. Finally, we achieved a vast decrease of the number of FOP-related clones. In the refactored SPL, only 4 % FOP-related clones exist, which is three times lower than in the original SPL (12 %). We conclude that code clone removal through refactorings is a promising approach to remove FOP-related clones from feature-oriented SPLs.

5.3 Discussion

First, a large portion of FOP-related clones can be removed by refactoring. This fact raises the question whether these clones occur due to *Copy&Paste* activities rather than limitations of FOP. But even in this case it is unclear why the programmer copied the code. For instance, it is possible that the programmer was not aware of proper (feature-oriented) refactorings or mechanisms that can avoid the resulting code clones. As a result, these clones can be considered FOP-related as well to some extent.

Secondly, we observed that there are some limitations for the removal of FOP-related clones. On the one hand, all of the removed clones occurred in features with a common, direct parent feature and most of them were alternative features. Hence, we can make no clear statement on removing clones caused by fine-grained extensions or crosscutting concerns. On the other hand, the detected clones where mostly identical (*type-I*) or had only slight differences (*type-II*). In the case, that the clones have notable (syntactic) differences, it may be difficult to remove them.

Third, we found FOP-related clones (type-II and type-III) that were not refactorable at all. One reason was that the application of refactorings implied complicated workarounds that outweigh the benefits of code clone removal. This observation led us to the assumption that there is a border line at which the extraction (of clones) is no longer beneficial for maintainability.

However, we are confident that code clone removal is possible for a considerable fraction FOP-related clones and that refactorings are a viable approach to remove them. Nevertheless, more research

[6] Note that this term is used in analogy to aspect-oriented refactorings [38] and different from the term used by Liu et al. [33], where it describes the process of decomposing (object-oriented) programs into features.

is necessary to find out reasons, characteristics, and solutions for code clones and their removal in feature-oriented SPLs.

6. Related Work

The different topics addressed in this paper such as clone detection/removal or feature-oriented programming, have been subject to intensive research in the past.

Many studies exist on code clones in object-oriented software systems. Some of them focus only on whether code clones exist or not [6, 10, 31] whereas others analyze code clones with respect to their effects [32, 37], their removal [7] or other peculiarities, e.g., identifying crosscutting concerns [12]. However, all of these studies are limited to OOP (and, to a minor fraction, functional programming). In contrast, our work focuses on clone detection and analysis of particularities 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-related code clones to the limitations of FOP, which can initiate discussions on FOP language design.

Beyond FOP, clone detection in SPLs is rare as well. Mende et al. propose clone detection for supporting the evolution of SPLs [36]. 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 syntactical 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.

Because of the negative effects of code clones, their removal has been subject of research in the past. Balazinska et al. propose advanced code clone analysis that supports the application of object-oriented refactorings for code clone removal [7]. Higo et al. propose a metric-based approach to identify refactoring opportunities [20] for code clones. Although we focus on refactorings for code clone removal as well, our work is different in the way that we propose refactorings tailored to FOP, which is in line with aspect-oriented refactorings [14, 38]. Specific details of such refactorings are outside the scope of this paper; for details, see [33].

7. Conclusion

Code clones have negative effects on software systems. We analyzed the existence of code clones in feature-oriented software product lines. To this end, we formulated research questions, regarding the causes and removal of code clones, specific to feature-oriented SPLs. We conducted an empirical analysis on ten different SPLs to answer these questions. Additionally, we performed a conceptual analysis on limitations of FOP and how these limitations contribute to code clones. Based on the results of the empirical analysis, we removed FOP-related code clones of an exemplary product line by the application of refactorings.

We observed, that code clones exist in feature-oriented SPLs and that a considerable number of these clones is FOP-related (in the context of SPLs). Furthermore, we found that these clones are caused by limitations of feature-oriented SPLs such as alternative features. Another interesting observation we made is that the development process is crucial for the number of FOP-related code clones. More precisely, we found that feature-oriented SPLs, developed from scratch, contain a significantly higher number of FOP-related clones than SPLs refactored from legacy applications. Finally, we observed that refactoring is a viable approach to remove FOP-related clones.

However, there still some questions we could not answer so far. More research is necessary to quantify which causes are crucial for FOP-related clones and which are not. Furthermore, it is open whether clones are common to SPLs in general. For instance, in future work we will analyze how other mechanisms for dealing with variabilty such as #ifdefs influence the occurence of code clones (compared to FOP). Nevertheless, our work and the results can serve as input for a discussion on code clones in feature-oriented SPLs and in SPL engineering in general.

Acknowledgments

We thank Thomas Thüm for giving insights to and the extension of FeatureIDE. Schulze's work has been funded in part by the German Federal Ministry of Education and Science (BMBF) through the Research Programme under Contract No. FKZ:13N10817. Apel's work is supported in part by DFG project #AP 206/2-1. Kästner's work is supported in part by the European Research Council (ERC grant ScalPL #203099).

References

[1] V. Alves and et al. Extracting and Evolving Mobile Games Product Lines. In *Proc. of the Int. Software Product Line Conf.*, pages 70–81. Springer-Verlag, 2005.

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

[3] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proc. of the Int. Conf. on Generative Programming and Component Engineering*, pages 125–140. Springer Verlag, 2005.

[4] S. Apel, T. Leich, and G. Saake. Aspectual Feature Modules. 34(2): 162–180, 2008.

[5] S. Apel, C. Kästner, and C. Lengauer. FeatureHouse: Language-Independent, Automated Software Composition. In *Proc. Int. Conf. on Software Engineering*, pages 221–231. IEEE Computer Society, 2009.

[6] B. S. Baker. On Finding Duplication and Near-Duplication in Large Software Systems. In *Proc. of the Work. Conf. on Reverse Engineering*, pages 86–95. IEEE Computer Society, 1995.

[7] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. Advanced Clone-Analysis to Support Object-Oriented System Refactoring. In *Proc. of the Work. Conf. on Reverse Engineering*, pages 98–107. IEEE Computer Society, 2000.

[8] H. Basit, D. Rajapakse, and S. Jarzabek. Beyond Templates: A Study of Clones in the STL and some General Implications. In *Proc. Int. Conf. on Software Engineering*, pages 451–459. ACM Press, 2005.

[9] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Soft. Eng.*, 30(6):355–371, 2004.

[10] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone Detection Using Abstract Syntax Trees. In *Proc. of the Int. Conf. on Software Maintenance*, pages 368–377. IEEE Computer Society, 1998.

[11] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo. Comparison and Evaluation of Clone Detection Tools. *IEEE Trans. Soft. Eng.*, 33(9):577–591, 2007.

[12] M. Bruntink, A. Deursen, R. Engelen, and T. Tourwe. On the Use of Clone Detection for Identifying Crosscutting Concern Code. volume 31, pages 804–818. IEEE Computer Society, 2005.

[13] P. Clements and L. Northrop. *Software Prodcut Lines: Practices and Patterns*. Addison Wesley, 2006.

[14] P. Cole, L.and Borba. Deriving Refactorings for AspectJ. In *Proc. Int. Conf. on Aspect-Oriented Software Development*, pages 123–134. ACM Press, 2005.

[15] 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.

[16] E. Duala-Ekoko and M. Robillard. Tracking Code Clones in Evolving Software. In *Proc. Int. Conf. on Software Engineering*, pages 158–167. IEEE Computer Society, 2007.

[17] S. Ducasse, M. Rieger, and S. Demeyer. A Language Independent Approach for Detecting Duplicated Code. In *Proc. of the Int. Conf. on Software Maintenance*, pages 109–118. IEEE Computer Society, 1999.

[18] M. Fowler. *Refactoring – Improving the Design of Existing Code*. Addison Wesley, 2000.

[19] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. Aries: Refactoring Support Environment based on Code Clone Analysis. In *Proc. of Int'l Conf. on Soft. Eng. and Appl. (SEA)*, pages 222–229. ACTA Press, 2004.

[20] Y. Higo, S. Kusumoto, and K. Inoue. A Metric-Based Approach to Identifying Refactoring Opportunities for Merging Code Clones in a Java Software System. *J. Softw. Maint. Evol.*, 20(6):435–461, 2008.

[21] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proc. Int. Conf. on Software Engineering*, pages 96–105. IEEE Computer Society, 2007.

[22] R. Johnson and B. Foote. Designing Reusable Classes. *J. of Object-Oriented Progr.*, 1(2):22–35, 1988.

[23] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter? In *Proc. Int. Conf. on Software Engineering*, pages 485–495. IEEE Computer Society, 2009.

[24] T. Kamiya, S. Kusumoto, and K. Inoue. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. volume 28, pages 654–670. IEEE Press, 2002.

[25] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[26] C. Kapser and M. W. Godfrey. Supporting the Analysis of Clones in Software Systems: A Case Study. *J. Softw. Maint. Evol.*, 18(2):61–82, 2006.

[27] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in Software Product Lines. In *Proc. Int. Conf. on Software Engineering*, pages 311–320. ACM Press, 2008.

[28] C. Kästner, S. Apel, and M. Kuhlemann. A Model of Refactoring Physically and Virtually Separated Features. In *Proc. of the Int. Conf. on Generative Programming and Component Engineering*, pages 157–166. ACM Press, 2009.

[29] G. Kiczales et al. Aspect-Oriented Programming. In *Proc. of the Eur. Conf. on Object-Oriented Programming*, pages 220–242, 1997.

[30] K. Kontogiannis. Evaluation Experiments on the Detection of Programming Patterns Using Software Metrics. In *Proc. of the Work. Conf. on Reverse Engineering*, pages 44–55. IEEE Computer Society, 1997.

[31] J. Krinke. Identifying Similar Code with Program Dependence Graphs. In *Proc. of the Work. Conf. on Reverse Engineering*, pages 301–309. IEEE Computer Society, 2001.

[32] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Soft. Eng.*, 32(3):176–192, 2006.

[33] J. Liu, D. Batory, and C. Lengauer. Feature Oriented Refactoring of Legacy Applications. In *Proc. Int. Conf. on Software Engineering*, pages 112–121. ACM Press, 2006.

[34] R. Lopez-Herrejon and D. Batory. A Standard Problem for Evaluating Product-Line Methodologies. In *Proc. of the Int'l Conf. on Generative and Component-Based Soft. Eng. (GCSE)*, pages 10–24. Springer-Verlag, 2001.

[35] J. Mayrand, C. Leblanc, and E. Merlo. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. In *Proc. of the Int. Conf. on Software Maintenance*, pages 244–254. IEEE Computer Society, 1996.

[36] T. Mende, F. Beckwermert, R. Koschke, and G. Meier. Supporting the Grow-and-Prune Model in Software Product Lines Evolution Using Clone Detection. In *Proc. of the Eur. Conf. on Software Maintenance and Reengineering*, pages 163–172. IEEE Computer Society, 2008.

[37] A. Monden, D. Nakae, T. Kamiya, S. Sato, and K. Matsumoto. Software Quality Analysis by Code Clones in Industrial Legacy Software. In *Proc. of the Int Symposium on Software Metrics*, pages 87–96. IEEE Computer Society, 2002.

[38] M. Monteiro and J. Fernandes. Towards a Catalog of Aspect-Oriented Refactorings. In *Proc. Int. Conf. on Aspect-Oriented Software Development*, pages 111–122. ACM Press, 2005.

[39] G. Murphy, A. Lai, R. Walker, and M. Robillard. Separating Features in Source Code: An Exploratory Study. In *Proc. Int. Conf. on Software Engineering*, pages 275–284. IEEE Computer Society, 2001.

[40] D. L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[41] P. W. Pearse, T. T.and Oman. Experiences Developing and Maintaining Software in a Multi-Platform Environment. In *Proc. of the Int. Conf. on Software Maintenance*, pages 270–277. IEEE Computer Society, 1997.

[42] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proc. of the Eur. Conf. on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[43] C. Roy and J. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-451, School of Computing, Queen's University at Kingston, 2007.

[44] C. K. Roy, J. Cordy, and R. Koschke. Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach. *Sci. Comp. Prog.*, 74(7):470–495, 2009.

[45] T. Thüm, D. Batory, and C. Kästner. Reasoning about Edits to Feature Models. In *Proc. Int. Conf. on Software Engineering*, pages 254–264. IEEE Computer Society, 2009.

[46] M. Toomim, A. Begel, and S. Graham. Managing Duplicated Code with Linked Editing. In *Proc. of the Symposium on Visual Languages - Human Centric Computing*, pages 173–180. IEEE Computer Society, 2004.