Measuring and Improving Code Quality in Highly Configurable Software Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

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

von Dipl. Inf. Wolfram Fenske

geb. am 02.10.1981 in Staßfurt

Gutachterinnen/Gutachter

Prof. Dr. Gunter Saake
Prof. Dr. Sven Apel
Prof. Dr. Ina Schaefer

Eingereicht:
Magdeburg, den 16.03.2020

Verteidigt:
Magdeburg, den 06.07.2020

Fenske, Wolfram:
Measuring and Improving Code Quality in Highly Configurable Software Systems
Dissertation, University of Magdeburg, 2020.

Abstract

Highly configurable software systems (also known as *software product lines*) are software systems that can be configured to fulfill multiple sets of requirements. To achieve this configurability, the code base of highly configurable software systems must be variable, so that well-defined pieces of the implementation can be selectively compiled into a given product or left out of it.

In this thesis, I have investigated how variability at the implementation level affects the quality of the code base. Specifically, I argue that variability adds new design challenges that assessments of internal quality should take into account. To this end, I propose *variability-aware code smells,* an adaptation of the well-known code smell idea that treats variability as a first class concept. I show that such smells exist *in the wild* and present evidence that they negatively affect software development. Moreover, I propose a concept and tool support to detect variability-aware code smells in C programs that use preprocessor directives (a. k. a. `#ifdef`s) as the variability mechanism. Preprocessor-based variability is widely used but also often criticized as a cause for hard-to-understand code and subtle bugs. For this reason, I analyzed the impact of preprocessor-based variability, finding that it has *no* systematic effect on maintainability in terms of the frequency or extent of code changes.

In addition to studying the effect of variability on code smells, this thesis also addresses the effect of variability on a closely related topic, *refactoring.* To this end, I created a taxonomy of reengineering activities for highly configurable software systems and classified existing work according to this taxonomy. Moreover, I propose two *variant-preserving refactorings* for reducing code replication in highly configurable software systems, and embed these refactorings in a novel process to migrate product families that were created through clone-and-own to a software product line.

In summary, this thesis contributes new insights into the impact of variability at the implementation level. It demonstrates how different design choices for implementing variability can have either positive or negative effects on important aspects of software development, such as understandability or maintainability. Moreover, it presents novel evidence that static source code configurability based on C preprocessor directives does *not* increase code change proneness. Finally, this thesis adds clarity to future discussions about refactoring techniques for highly configurable software systems and contributes new refactorings to improve the code quality such systems.

Inhaltsangabe

Hoch-konfigurierbare Software-Systeme (auch bekannt als *Software-Produktlinien*) sind Software-Systeme, die konfiguriert werden können, um unterschiedlichen Anforderungen gerecht zu werden. Um diese Konfigurierbarkeit zu erreichen muss die Code-Basis hoch-konfigurierbarer Software-Systeme variabel sein, sodass beim Kompilieren eines gegebenen Produkts wohldefinierte Teile der Implementierung selektiv ein- oder ausgeschlossen werden können.

In der vorliegenden Doktorarbeit habe ich untersucht, wie Variabilität auf der Implementierungsebene die Qualität der Code-Basis beeinflusst. Im Speziellen behaupte ich, dass Variabilität neue Herausforderungen für den Entwurf mit sich bringt, die bei der Bewertung der internen Qualität berücksichtigt werden sollten. Zu diesem Zweck schlage ich *variabilitätsgewahre Code Smells* vor, eine Adaptierung der bekannten Code-Smell-Idee, welche Variabilität in den Mittelpunkt der Betrachtung rückt. Ich zeige, dass derartige Smells in realen Software-Projekten vorkommen und bringe Belege dafür vor, dass sie die Software-Entwicklung negativ beeinflussen. Des Weiteren stelle ich ein Konzept und Werkzeugunterstützung vor, um variabilitätsgewahre Code-Smells in C-Programmen, die Präprozessordirektiven (sogenannte `#ifdef`s) als Variabilitätsmechanismus nutzen, zu detektieren. Präprozessor-basierte Variabilität ist weit verbreitet, wird aber auch häufig als Ursache schwer verständlichen Codes und subtiler Bugs kritisiert. Aus diesem Grund habe ich die Auswirkung von Präprozessor-basierter Variabilität untersucht, mit dem Ergebnis, dass *keine* systematischen Auswirkungen auf die Wartbarkeit im Sinne von häufigeren oder umfangreicheren Änderungen vorliegen.

Zuzüglich zur Untersuchung der Auswirkungen von Variabilität auf Code-Smells adressiert diese Doktorarbeit die Auswirkungen von Variabilität auf ein eng verwandtes Thema, *Refactoring.* Zu diesem Zweck habe ich eine Taxonomie von Reengineering-Aktivitäten für hoch-konfigurierbare Software-Systeme erstellt und bestehende Arbeiten gemäß dieser Taxonomie klassifiziert. Weiterhin schlage ich variantentreue Refactorings zur Reduktion von Code-Replikation vor und bette diese Refactorings in einen neuartigen Prozess ein, mit dessen Hilfe Produktfamilien, die durch *Clone-and-Own* entstanden sind, in eine Software-Produktlinie überführt werden.

Zusammenfassend liefert diese Doktorarbeit neue Erkenntnisse zum Einfluss von Variabilität auf die Implementierungsebene. Sie zeigt, wie unterschiedliche Entscheidungen bei der Implementierung von Variabilität sowohl positive als auch negative Auswirkungen auf wichtige Aspekte der Software-Entwicklung, wie zum Beispiel Verständlichkeit oder Wartbarkeit, haben können. Zusätzlich liefert sie neue Belege,

dass statische Quellcode-Konfigurierbarkeit basierend auf C-Präprozessordirektiven die Änderungsanfälligkeit *nicht* erhöht. Abschließend erleichtert diese Doktorarbeit zukünftige Diskussionen über Refactoring-Techniken für hoch-konfigurierbare Software-Systeme und steuert neue Refactorings bei, um die Code-Qualität solcher Systeme zu verbessern.

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AOP	Aspect-Oriented Programming
API	Application Programmer Interface
AST	Abstract Syntax Tree
DOP	Delta-Oriented Programming
FM	feature model
FOP	Feature-Oriented Programming
FOR	Feature-Oriented Refactoring
FOSD	Feature-Oriented Software Development
LOC	non-blank, non-comment lines of code
OOP	Object-Oriented Programming
OR	odds ratio
PDG	Program Dependence Graph
SPL	Software Product Line
SPLE	Software Product Line Engineering
UML	Unified Modeling Language
VCS	Version Control System
VSoC	Virtual Separation of Concerns

1. Introduction

Refactoring is the act of changing the internal structure of software without changing its external behavior [107]. It is an important technique in software engineering, especially in object-oriented programming. Refactoring has two major use cases: First, refactoring is used as a maintenance technique to improve the internal quality of a software system. In this context, *code smells* help identify pieces of code that suffer from poor structure and would benefit from refactoring [107]. Second, refactoring supports major reengineering activities, such as adapting an existing piece of software to a new architecture. In this second context, refactoring ensures that code retains its original behavior even though it undergoes major changes.

Refactoring has been researched in depth and corresponding tool support is standard in current IDEs [249]. Moreover, research has addressed the detection and correction of code smells (e. g., [227, 88, 196, 256, 258, 257]) and extensively investigated their negative impact on software development (e. g., [169, 1, 170, 338, 380, 124]). Yet despite the maturity of refactoring and code smell research, the state of the art falls short when dealing with the variability of *highly configurable software systems*. A highly configurable software system represents not just a single program but a set of related programs – a *program family* [61, 53, 13]. The commonalities and differences of the members of such a program family are communicated in terms of *features,* that is, increments in functionality that are important to some stakeholder. When an individual product of the product family is compiled, only the code that belongs to the product's features is included whereas other code, which belongs to unwanted features, is excluded. The possibility to selectively include or exclude *feature code* (i. e., code that implements a particular feature) requires a *variability mechanism*. By combining a *variability mechanism*, such as conditional compilation, a plug-in architecture or *Feature-Oriented Programming (FOP)* [13], with a standard programming language, parts of the code base come variable. For example, pieces of feature code can mutually exclude each other.

Variability in the code base creates two important design challenges, and in this thesis, I argue that if these challenges are not handled appropriately, the code quality of highly configurable software systems is at risk. First, the use of variability mech-

anisms has to be structured in the same way that classes and methods in a standard programming language have to be structured. However, it is unclear which structures are good and which ones are bad because corresponding research and experience is still lacking. In particular, existing code smells are not helpful because they are oblivious to variability. Thus, even if developers of highly configurable software systems wanted to refactor, they would not know *what* to refactor. Second, variability at the code level causes existing refactorings to fail and introduce behavior-altering changes. This problem has been recognized in the literature [330] and some progress has been made [328, 329, 205, 174, 175]. Nevertheless, the design and automation of refactorings for highly configurable software systems remains a challenge. Thus, even if developers of highly configurable software systems knew *what* to refactor, they would not know *how*. The goal of my thesis is to address both of these challenges and thereby advance the state of refactoring for highly configurable systems. To this end, I formulated two research questions.

1.1 Research Questions

My objective is to investigate how variability in the code base affects code smells and refactoring. I formulate two research questions and accompanying sub-questions to guide me toward this objective. To distinguish these research questions, which overarch the whole thesis, from subordinate research questions, which are specific to an individual chapter, the chapter-specific research questions are referred to as RQ 1, RQ 2, and so on, whereas the thesis-wide research questions are referred to as $\mathrm{RQ_T}$ 1 and $\mathrm{RQ_T}$ 2, where *T* stands for *thesis*. In particular, this thesis addresses the following questions:

$\mathbf{RQ_T}$ 1. *How do variability mechanisms affect the code smell concept?* This question targets the *what?* of refactoring for highly configurable software systems. In particular, I want to identify patterns of encoding variability that lead to maintenance problems. Knowledge about harmful patterns enables developers to avoid them in the future and write more reliable and understandable code. Moreover, this knowledge can serve as a guide for research into corresponding corrective refactorings. To achieve this goal, I address the following sub-questions.

$\mathbf{RQ_T}$ 1.1 *How do established code smells change when variability is involved?*

$\mathbf{RQ_T}$ 1.2 *Are there styles of encoding variability that have negative effects on code comprehension or maintenance?*

$\mathbf{RQ_T}$ 1.3 *How can these harmful styles of encoding variability be detected automatically?*

My second research question targets the *how?* of refactoring for highly configurable software systems. This question is difficult to formulate since the *what?* of refactoring is also the subject of my investigations. To overcome this difficulty, I focus on a known use case of large-scale refactoring: the *migration* of a clone-and-own product family into a highly configurable software system. It is common practice in the industry to create new variants of a software product by copying an existing product and adapting it to fit a new set of requirements [91, 245, 79, 317, 349]. This practice, known as *clone-and-own,* is cheap and easy at first, but it incurs excessive maintenance costs in the long run because code is replicated at a massive scale.

A possible way to reduce these costs is to migrate the product family, that is, to reengineer all products in such a way that code replication is reduced and systematic reuse is increased. During migration, it is crucial to preserve the behavior of every product in the product family, which makes migration an ideal use case for refactoring. Several migration approaches have been proposed in the literature (e. g., [6, 211, 368]), but the details of refactoring the actual source code remain unclear. Therefore, I study refactoring for highly configurable software systems in the context of migration. This way, I address not only the *what?* of refactoring but also the *how?*, and make sure that the refactorings I explore serve a relevant use case. To this end, I formulate the following research question.

RQ$_\mathbf{T}$ 2. *How can clone-and-own product families be migrated in a variant-preserving manner?* My goal with this question is to explore new refactorings that can be used to increase systematic reuse in a highly configurable software system in a safe, behavior-preserving manner. In particular, I ask the following sub-questions.

RQ$_\mathbf{T}$ 2.1 *Which refactorings for highly configurable software systems have been proposed in the literature?*

RQ$_\mathbf{T}$ 2.2 *How can code replication in clone-and-own product families be reduced in a variant-preserving manner?*

RQ$_\mathbf{T}$ 2.3 *What are the limitations of a refactoring-based migration approach?*

In answering these research questions, this thesis makes a number of contributions. I outline these contributions next.

1.2 Contributions

This thesis contributes to the research on highly configurable software systems in four important ways: The first contribution is a catalog of *variability-aware code smells,* which are code smells that explicitly take variability into account. I describe how the appearance of these smells changes depending on the variability mechanism and propose a metrics-based approach and tool support to detect them. Moreover, I provide initial evidence that certain variability-aware code smells occur frequently in real-world systems and negatively affect code comprehension. Complementing these negative patterns, I additionally provide examples of helpful implementation patterns, that is, patterns for encoding variability that reduce the complexity in a highly configurable software system. These patterns, both negative and postive, open up new directions for future studies. Moreover, they raise awareness to the implications of different solutions to variability-related problems and help practitioners increase internal software quality.

The second contribution of this thesis is an in-depth study of the relationship between C preprocessor (CPP) directives and maintainability. This contribution is especially relevant since preprocessors in general, and the CPP in particular, are a commonly used variability mechanism in industry [13]. I provide evidence that using CPP directives as a means to implement fine-grained static source code configurability has little to no negative effects regarding the frequency or extent of changes to the code base. These results partly explain the continued, wide-spread

use of CPP-based variability in practice. Moreover, these findings challenge the frequent critique of the CPP and emphasize the importance of unbiased, evidence-based software engineering research.

The third contribution of this thesis is a taxonomy of reengineering approaches that target highly configurable software systems. This taxonomy brings clarity to discussions about refactoring and other kinds of reengineering techniques for configurable software. In particular, both practitioners and researchers will find it easier to name the exact reengineering problem they need to solve and match it to the available solutions.

Finally, the fourth contribution of this thesis is a process based on clone detection and variability-aware refactorings for migrating clone-and-own product families to a highly configurable software system. The methodology and refactorings I propose will help practitioners make the difficult transition from clone-and-own development to a systematic reuse approach. In addition to their use in a migration context, my refactorings are also applicable in other settings where code replication is a problem. Thus, they will be useful for a wide range of maintenance tasks in highly configurable software systems.

1.3 Structure of the Thesis

Apart from this introductory chapter, my thesis consists of two background chapters, three chapters comprising the main contributions, and a final chapter discussing the conclusion and future work. In detail, the structure is as follows:

- **Chapter 2** provides background knowledge on code smells and refactoring for traditional (non-configurable) software systems.
- **Chapter 3** contains the background on highly configurable software systems and clone-and-own, which are two alternative paradigms for developing software program families. Moreover, Chapter 3 provides the fundamentals on code clones that will be necessary to follow my migration approach.
- In **Chapter 4**, I introduce the concept of variability-aware code smells, present a corresponding detection approach, and discuss the results of a survey and a case study that validate the concept of variability-aware code smells.
- Building on the variability-aware code smell concept, **Chapter 5** reports on an empirical study of the relationship between preprocessor-based variability and maintainability. The corresponding data was mined from twenty open-source software systems written in C and was examined both quantitatively as well as qualitatively.
- In **Chapter 6**, the topic switches from code smells to refactoring. In particular, I present a taxonomy of reengineering and refactoring approaches for highly configurable software systems. Moreover, I discuss two novel refactorings for highly configurable software systems and present a migration approach for clone-and-own product families that is based on these refactorings.
- **Chapter 7** concludes the thesis and discusses ideas for future work.

2. Code Smells

Martin Fowler and Kent Beck introduced *code smells* in their much-cited *refactoring* book as a way to describe structural weaknesses in the source code of object-oriented software systems (Fowler et al. [107], p. 75). My concept of *variability-aware code smells*, which I present in Chapter 4, is based on their idea. Moreover, this thesis presents an approach to detect variability-aware code smells automatically (see Chapter 4) and to remove them with the help of refactoring (see Chapter 6). In this chapter, I present the necessary foundations of code smells, code smell detection, and refactoring.

2.1 Defining Code Smells

Code smells are intricately connected to software *refactoring,* which makes it impossible to explain one without the other. Hence, I start with Fowler's definition of refactoring before continuing to explain what code smells are:

> Refactoring is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. (Fowler et al. [107], p. 53)

Unfortunately, there is no equally succinct definition of "code smells" as there is for refactoring. Instead, Fowler et al. describe code smells rather vaguely, for example as "certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring" (Fowler et al. [107], p. 75) or as "surface indications that usually corresponds to a deeper problem in the system" [106]. To make up for this lack of a definition, I have identified the following four points to highlight the key properties of code smells.

First, smelly code is *not* necessarily faulty code. In other words, source code with a smell usually behaves correctly but is written in a way that makes it hard to understand or change. However, Fowler et al. also argue that a bug report indicates

that refactoring is required because the code was not clear enough to see the bug beforehand (Fowler et al. [107], p. 59).

Second, code smells only hint at underlying problems, they are not the problems themselves [106, 55]. Trifu and Marinescu use illnesses as an analogy [363]: An illness is indicated by one or several symptoms, and is cured through appropriate treatment. In software engineering, the illness is a higher-level design problem, the symptoms are code smells, and the treatment is refactoring. An important implication of this analogy is that the goal of refactoring is *not* to eliminate code smells but to remove the underlying design problem. The code smells disappear simply as a side effect.

Third, code smells are something that is readily recognizable in the source code. For instance, Fowler claims that a Java method with more than a dozen lines of code makes his "nose twitch" [106]. The smell in this instance is called Long Method (Fowler et al. [107], p. 76f.) and indeed, overly long and complex methods are easily spotted in the source code. However, other smells require a more intimate knowledge of the software system in question. For example, the smell Parallel Inheritance Hierarchies (Fowler et al. [107], p. 83) can only be detected if the inheritance relationships of the involved classes are known. So it is true that code smells are "quick to spot", but only by developers who are familiar with the source code and its history.

Finally, code smells are not precise; they are only heuristics. Opdyke writes, "These structural abnormalities or structural similarities don't always mean that you'd want to apply a refactoring, but often they do." (Fowler et al. [107], p. 384). In other words, the presence of code smells warrants closer inspection but in some cases, smelly code is just fine (see also [106, 55]).

Related Concepts

Code smells are not the only attempt at identifying bad practices in software development. Anti-patterns, linting tools, and style guides are only some of the concepts that pursue similar goals. In this section, I explain these related concepts and highlight similarities to and differences from code smells.

Anti-patterns. Design patterns codify known, good solutions to recurring problems in software development [37, 108]. Brown et al. inverted this idea and created the concept of *anti-patterns,* which codify known, *bad* solutions to recurring problems [42].

Brown's catalog groups anti-patterns into three categories, *software development*, *software architecture*, and *software project management* (see Chapters 5, 6 and, 7, respectively). Anti-patterns in the latter two categories refer to high-level issues. For example, the Vendor Lock-In anti-pattern describes software that is difficult to evolve because it depends too much on proprietary third-party architectures. Problems of this kind have nothing to do with way the source code is structured. Hence, only software development anti-patterns, which describe problems on the implementation level, are related to code smells.

Although some researchers make no distinction between software development anti-patterns and code smells (e. g., [274, 281]), others see a cause-and-effect relationship (e. g., [258, 172, 170]): Anti-patterns are the reasons for a flawed design and code smells are the source code manifestations of the flawed design. Consider the BLOB anti-pattern as an example. A BLOB is a design in which one class performs all the processing and other classes degenerate to mere data holders (Brown et al. [42], Chapter 5). The class that monopolizes the processing will have the LARGE CLASS smell because it is "trying to do too much" (Fowler et al. [107], p. 78). It will also smell of FEATURE ENVY (Fowler et al. [107], p. 80) because its computations are mostly based on data that belongs to other classes. Those other classes, in turn, will have the DATA CLASS smell (Fowler et al. [107], p. 86). In summary, anti-patterns, and in particular, software development anti-patterns, lead to poor software designs and code smells can be symptoms of such designs.

Linting tools and bug patterns. Long before code smells and anti-patterns were proposed, tools have existed that check for suspicious, potentially erroneous programming constructs, such as unused variables and infinite loops. The first of these tools was LINT, a static code analyser for C programs [146]. Since then, many "LINTING tools" followed, for instance, FINDBUGS and CHECKSTYLE for JAVA [139, 50], ESLINT for JAVASCRIPT [90], and COVERITY for C/C++, FORTRAN, and RUBY [355]. Hovemeyer and Pugh, the creators of FINDBUGS, demonstrated that checking for unused variables and similar suspicious patterns is a cheap and effective means to prevent bugs [139]. Consequently, they call them "bug patterns".

Code smells and linting tools share some similarities, but they also have differences. The similarities are that both pursue the goal of preventing bugs, and they do so with the help of suspicious patterns in the source code. The difference is that bug patterns serve to identify concrete, localized problems, whereas code smells serve to identify higher level design flaws and are concerned with larger program elements. For example, the bug pattern *uninitialized variable* simply means that the programmer forgot an initializing statement. The fix usually involves a single line of code. By contrast, a smell like LARGE CLASS indicates the whole class design is flawed. Fixing such a design usually involves creating one or more helper classes and shifting around many lines of code.

Style guides. Companies, but also open-source projects, may specify *style guides*, guides that define how source code should or should not be written (e. g., Airbnb [3], NPM [269], and the LINUX kernel project [210]). Many style guides focus on formatting questions, for example, whether to use DOS- or UNIX-style line endings (cf. [269]). However, some style guides also include LINT-like bug patterns and other higher-level issues. For example, Airbnb advises programmers, "Be descriptive with your naming." [3]. The NPM' style guide, in turn, forbids the use of `null` or `0` as `boolean` values [269]. Just like code smells, such rules focus on the internal quality of software, which is why some researchers extend Fowler's traditional code smell catalog with selected style guide rules (e. g., Saboury et al. [319]). Thus, I conclude that there is an overlap, but generally, style guides and code smells are not the same.

Code clones. Code clones are source code fragments that occur in the same or similar form in multiple locations [312]. They usually result from copying a piece of source code and pasting it elsewhere, possibly adapting the copy afterwards [152].

Under the name Duplicated Code, Fowler et al. included code clones at position one in their catalog (Fowler et al. [107], p. 76). Nevertheless, the academic literature continues to treat code clones and code smells as separate topics, likely because code clone research predates the work of Fowler et al. by several years (e. g. [24, 145, 180, 236, 36]). I adhere to this separation in this thesis because in this thesis, I mainly consider code clones in the context of cloning entire programs, a topic that I explain in detail in the next chapter (see Section 3.6) as well as in Chapter 6.

Smells in other paradigms and specialized domains. Building on the pioneering work of Fowler et al., Brown et al. and Riel for object-oriented software, numerous researchers transferred the smell concept to new programming paradigms and to specialized domains. I summarize their work in the next paragraphs.

The literature contains several catalogs of code smells for programming paradigms besides *Object-Oriented Programming (OOP)*. Specifically, smells for *Aspect-Oriented Programming (AOP)* [102, 260, 294, 346], and for *Delta-Oriented Programming (DOP)* [329] have been discussed. Hermans et al., in turn, pioneered code smell and refactoring research for spreadsheet applications [131, 130, 129, 128]. Moreover, code smells and refactoring have been explored for functional programming [320] and databases [9].

Garcia et al. argue that bad design also affects higher levels of abstraction, not just the implementation level. To characterize such high-level problems, they proposed a catalog of *architectural smells* [110, 111], and further researchers have extended this initial catalog with additional smells [10, 254].

Complementing smell catalogs for other paradigms and higher levels of abstraction, various authors explored code smells for specific domains. In particular, Deursen et al. discovered that test code has is own specific set of code smells [67]. These so-called *test smells* are now an established subfield in the wider field of code smell research (e. g., [66, 310, 34, 367]). Others investigated code smells for Android and iOS apps (e. g. [284, 125, 262, 167, 126, 222, 122, 45]). These studies cover a variety of topics, including automatic code smell detection [284, 167, 126] and the impact of code smells on performance and energy consumption [125, 262, 45].

My work on variability-aware code smells is similar to the aforementioned work as it also transfers the smells concept from OOP to other paradigms (*Feature-Oriented Programming (FOP)* and procedural programming) and considers smells in a non-traditional setting (highly configurable software systems). I discuss this point further in Chapter 4.

Summary

Code smells are symptoms of design problems that make a software system hard to understand, change, or evolve. These symptoms are reflected in poorly structured source code, for example, in methods that are overly long or in inheritance hierarchies that are difficult to change. The remedy is to restructure the code through

refactoring, which solves the design problems and, as a side effect, eliminates the code smells.

There are several concepts that are related code smells, including software development anti-patterns and lint-like bug patterns. I have discussed how these concepts are similar to and different from code smells.

2.2 A Code Smell Example: Long Method

Having defined the code smell concept in the previous section, I will now make the concept more palpable by explaining the Long Method smell (Fowler et al. [107], p. 76f.) in detail. This smell is of special importance to my thesis as it forms the basis for my variability-aware code smell Annotation Bundle, which I describe in Chapter 4. In this section, I discuss the essential properties of a Long Method in OOP and outline how to refactor a Long Method into a short one.

In Listing 2.1, I show a Java method that Martin, the author of the programmer's guide *Clean Code*, uses as an example for a Long Method (Martin [232], Chapter 3). Martin cites a number of reasons why this method is hard to understand. For instance, there are several nested `if` statements, e. g., on Lines 6 and 9, some of them doubly nested. Inside these `if` statements, many low-level activities take place, such as string buffers that are filled (e. g., on Lines 12–14 and 21–23) and wiki pages that are fetched (e. g., on Lines 7–8 and 17). It is hard to see how all these low-level activities fit together and which part of the method's overall task they solve.

Several refactorings can turn a Long Method into something that is easier to understand. Particularly complex cases can be refactored into their own class using Replace Method with Method Object (Fowler et al. [107], p. 135ff.). In less severe cases, such as the one in Listing 2.1, Extract Method (Fowler et al. [107], p. 110ff.) is the recommended refactoring. Specifically, Martin applied Extract Method multiple times to turn the original method into the much shorter variant shown in Listing 2.2. For brevity, I omitted the extracted helper methods from the listing and only show the top-level method. Compared to the original, it is easier to get an intuition of what the refactored method does. Although not all details are clear, one can understand that a web page, which is passed as a parameter, can either be a test page or not. If it is, some setup and teardown code is included and the page is rendered as HTML; otherwise, the page is rendered immediately. The old method name, `testableHtml`, hardly revealed that this was the sequence of actions that the method would perform. For this reason, the method was also renamed. The new name, `renderPageWithSetupsAndTeardowns`, reflects more clearly what the method actually does.

In summary, a Long Method is overly complex because it implements all the steps toward fulfilling a big task right in place. The solution is to identify the steps that belong to a certain subtask and refactor them into a helper method using Extract Method. That way, a Long Method is turned into an easily understandable, shorter method.

```
1  public static String testableHtml(PageData pageData,
2                          boolean includeSuiteSetup) throws Exception {
3      WikiPage wikiPage = pageData.getWikiPage();
4      StringBuffer buffer = new StringBuffer();
5      if (pageData.hasAttribute("Test")) {
6          if (includeSuiteSetup) {
7              WikiPage suiteSetup =
8                  PageCrawlerImpl.getInheritedPage(SuiteResponder.SUITE_SETUP_NAME, wikiPage);
9              if (suiteSetup != null) {
10                 WikiPagePath pagePath = suiteSetup.getPageCrawler().getFullPath(suiteSetup);
11                 String pagePathName = PathParser.render(pagePath);
12                 buffer.append("!include -setup .")
13                       .append(pagePathName)
14                       .append("\n");
15             }
16         }
17         WikiPage setup = PageCrawlerImpl.getInheritedPage("SetUp", wikiPage);
18         if (setup != null) {
19             WikiPagePath setupPath = wikiPage.getPageCrawler().getFullPath(setup);
20             String setupPathName = PathParser.render(setupPath);
21             buffer.append("!include -setup .")
22                   .append(setupPathName)
23                   .append("\n");
24         }
25     }
26     buffer.append(pageData.getContent());
27     if (pageData.hasAttribute("Test")) {
28         WikiPage teardown = PageCrawlerImpl.getInheritedPage("TearDown", wikiPage);
29         if (teardown != null) {
30             WikiPagePath tearDownPath = wikiPage.getPageCrawler().getFullPath(teardown);
31             String tearDownPathName = PathParser.render(tearDownPath);
32             buffer.append("\n")
33                   .append("!include -teardown .")
34                   .append(tearDownPathName)
35                   .append("\n");
36         }
37         if (includeSuiteSetup) {
38             WikiPage suiteTeardown =
39                 PageCrawlerImpl.getInheritedPage(SuiteResponder.SUITE_TEARDOWN_NAME, wikiPage);
40             if (suiteTeardown != null) {
41                 WikiPagePath pagePath = suiteTeardown.getPageCrawler().getFullPath(suiteTeardown);
42                 String pagePathName = PathParser.render(pagePath);
43                 buffer.append("!include -teardown .")
44                       .append(pagePathName)
45                       .append("\n");
46             }
47         }
48     }
49     pageData.setContent(buffer.toString());
50     return pageData.getHtml();
51 }
```

Listing 2.1: Example of a Long Method *(Reproduced from Martin [232], Chapter 3)*

2.3 Negative Effects of Code Smells

Code smells, anti-patterns, and related concepts are of interest because they are believed to be detrimental to software development. Specifically, they are believed to impede program comprehension, increase maintenance effort, and make faulty changes more likely. In this section, I summarize the respective research findings.

The initial publications I reviewed for this section stem from the systematic literature review of code smell research by Zhang et al. [388], and from the excellent discussions of related work by Sjøberg et al. [338] as well as Hall et al. [124]. Through snowballing

```
1  public static String renderPageWithSetupsAndTeardowns(PageData pageData,
2                          boolean isSuite) throws Exception {
3      boolean isTestPage = pageData.hasAttribute("Test");
4      if (isTestPage) {
5          WikiPage testPage = pageData.getWikiPage();
6          StringBuffer newPageContent = new StringBuffer();
7          includeSetupPages(testPage, newPageContent, isSuite);
8          newPageContent.append(pageData.getContent());
9          includeTeardownPages(testPage, newPageContent, isSuite);
10         pageData.setContent(newPageContent.toString());
11     }
12     return pageData.getHtml();
13 }
```

Listing 2.2: Long Method example from Listing 2.2 after refactoring *(Reproduced from Martin [232], Chapter 3)*

and following up on more recent work of the eminent research groups in the field, I extended this initial set of publications to a total of twenty studies. Although this process is no substitute for a systematic literature review, I argue that it yielded a representative overview of the relevant findings.

Note that in the present chapter, I discuss the negative effects of all code smells *except* Duplicated Code. The overview of the negative effects of Duplicated Code (code clones) follows in the next chapter, when I explain large-scale cloning as means to develop program families.

2.3.1 Change and Fault Proneness

Many studies of the negative effects of code smells focus either on change proneness [169, 274, 309], on fault proneness [62, 143, 202, 124, 218, 231, 319], or on both [273, 170]. In this context, change proneness refers to changes happening either more frequently or more extensively. Both forms of change proneness are undesirable because statistically, frequent changes statistically increase the likelihood of faults [84, 123], while more extensive changes require more maintenance effort [86, 255, 338]. Fault proneness, in turn, corresponds to the frequency with which a piece of code contains a fault (a. k. a. "bug"). Since identifying faults directly is impossible in the general case, fault proneness is usually approximated by counting the number of bugfixes that a piece of code has undergone (see [103, 339] for details about the corresponding technique).

Nearly all studies agree that the presence of one or several code smells in a piece of code is statistically associated with an increase in change proneness [169, 170, 274, 273, 309] and fault proneness [202, 62, 273, 170, 218, 319]. Furthermore, there is evidence that Java classes that are *related* to a class with a smell are more fault-prone than classes without such a relationship [231, 143]. Here, "related" means that a class calls methods of a smelly class or is frequently changed together with a class containing smells.

Even though the aforementioned studies suggest that code smells are unequivocally harmful, the evidence is only clear if no distinction is made between smells. However,

a look at the detailed results reveals that individual smells often have negative effects only in some software systems but not in others (for examples, see [169, 170, 309, 319]). For instance, Saboury et al. investigated twelve smells in five JavaScript regarding fault-proneness [319]. Among these twelve smells, not a single one was consistently associated with increased fault proneness in all five systems.

There is further work that casts a doubt on the harmfulness of code smells [124, 273]. In particular, Hall et al. showed that Java classes exhibiting certains smells are sometimes *less* fault-prone than classes without smells [124]. Olbrich et al., in turn, found that after controlling for differences in size, the apparently negative effects reversed, meaning that smelly classes were actually *less* change- and fault-prone than classes without smells. Research on object-oriented metrics already highlighted the importance of controlling for size when trying to predict change or fault proneness [87, 390]. This is also important for code smells, especially for smells such as Large Class or Long Method, which are related to large code size. For these reasons, the study I present in Chapter 5 went to great lengths to control for the confounding effects of size, as well as other potential confounding factors.

2.3.2 Program Comprehension

Deligiannis et al. ran two similar controlled experiments in which the effect of God Classes on program comprehension was measured [64, 65]. Resultes were mixed: God Classes negatively affected program comprehension only in the first experiment [64] but had no effect in the second one [65].

A controlled experiment involving the Blob and Spaghetti Code anti-patterns found that neither anti-pattern affects program comprehension when it occurs in isolation [1]. However, confronted with code in which both anti-patterns were present, the participants had more difficulties comprehending the code and gave less accurate answers.

2.3.3 Maintenance Effort

Instead of using change- or fault-proneness as proxies, several researchers chose to measure maintenance effort directly in controlled experiments [64, 65, 337, 338]. Two such experiments, which involved university students, found that the presence of God Class smells increases maintenance effort and leads to solutions that are less complete and less correct [64, 65]. In another experiment, professional developers were given maintenance tasks on industrial software systems [337, 338]. After controlling for confounding variables (e. g., differences in experience between individual developers) and for file size, only two out of the twelve smells under investigation were significantly correlated with effort. Notably, one of the two significant correlations – the correlation for Refused Bequest – was *negative*. In other words, if Refused Bequest was present, effort *decreased*. As a secondary finding, this study concluded that predictions of maintenance effort based on file size alone are more accurate than predictions based on smell information alone.

A recent trend in academia is to focus on co-occurring smells instead of smells that occur in isolation (see, e. g., [1, 383, 272, 285]). The hypothesis is that if two or more smells occur close to each other (e. g., in the same class), their respective negative

effects aggravate. Regarding program comprehension, the experiment of Abbes et al. [1] (already mentioned in the previous section) supports this hypothesis. Further evidence comes from Oizumi et al., who showed that certain combinations of co-occurring code smells indicate design problems with high accuracy whereas a code smell that occurs alone rarely corresponds to an actual design problem [272].

2.3.4 Subjective Perception of Developers

To enrich the quantitative evidence with qualitative insights, developers have been asked about their subjective opinions on the harmfulness of code smells [381, 382, 282]. According to these studies, developers generally perceive Duplicated Code as a design problem [382], as well as smells related to long or complex code (e. g., Complex Class and Long Method) [382, 282]. Other smells (e. g., Refused Bequest) are only considered problematic if the smell intensity is high [282], and a final group of smells (including, e. g., Lazy Class) is seen as harmless.

Yamashita and Moonen identify thirteen factors that professional developers deem important for maintenance and reason how well they align with code smells [381]. They find that code smells cover nine factors at least partially but not the remaining four. For example, several smells, including the God Class smell, are related to the "simplicity" factor, but no smell is related to a factor such as "appropriate technical platform".

2.3.5 Biases in Surveyed Studies

The literature discussed in this section contains several biases that threaten the validity of the findings on the negative effects of code smells. First, code smells come from an OOP background, so it is no surprise, that all the work on their negative effects has been performed on software written in an OOP language. However, there is a notable bias towards one language in particular, Java. Except for Saboury et al., who studied JavaScript systems, all the other studies used Java subject systems. The results may be similar for other OOP languages that resemble Java, such as C++, or C#, but we cannot be certain.

Second, as several authors already pointed out, the choice of studied code smells is imbalanced [388, 124]. For example, the negative effects of Large Class and its siblings (God Class, Brain Class, and the Blob) were covered by fifteen studies (namely [1, 64, 65, 143, 169, 170, 202, 218, 231, 274, 273, 282, 309, 337, 338]). By contrast, only six studies have considered Feature Envy (see [62, 202, 231, 282, 337, 338]) and none have investigated the negative effects of Parallel Inheritance Hierarchies. As a result, there are some smells whose harmfulness has been researched thoroughly but for many others, the empirical evidence is thin or missing entirely.

A final threat is the choice of analyzed systems. Only four studies analyzed industrial systems (see [272, 337, 338, 381]) but the vast majority analyzed open-source systems (see [1, 62, 124, 143, 170, 169, 202, 231, 218, 274, 273, 282, 309]). A partially mitigating factor is that many studies included Eclipse (see [169, 170, 309, 62, 124, 202, 231, 282]). Although open-source, Khomh et al. point out that Eclipse is close in size and complexity to many industrial software systems, and is partly developed by a commercial company, IBM [170].

2.3.6 Summary

Researchers have investigated whether code smells make source code more difficult to change and evolve, more fault-prone, or harder to understand. Moreover, developers' opinions on the usefulness of code smells as indicators of maintainability have been collected. The evidence suggests that if we do not distinguish between specific types of smells, then smelly code is harder to maintain and contains more faults than code without smells. But if we look at individual smells, the findings can hardly be generalized. A smell that has negative effects in one system can have no effects or even positive ones in another system.

Other factors besides code smells may affect maintenance effort and fault proneness, but not all studies consider them. Those that ignore potentially confounding factors report medium to large negative effects of code smells. By contrast, studies that control for these factors report small negative effects, no effects, or even small positive effects.

There are few findings on the effects of code smells on program comprehension. The findings that are available indicate that smelly code is harder to understand, especially if a piece of code suffers from multiple smells. However, more work is needed to draw definitive conclusions.

From a developer's point of view, not all code smells are equally relevant. While developers generally pay attention to smells that are related to long or complex code, other smells are only seen as problematic if the smell intensity is high, and yet others are seen as unimportant. Moreover, maintainability encompasses many different factors, and not all of them are addressed by code smells.

Research into the negative effects of code smells is biased in several ways. Specifically, most studies consider open-source systems written in Java and concentrate on a small number of well-known smells. To achieve a comprehensive picture of the harmfulness of code smells, more studies are needed, which should cover further programming languages, different smells, and focus on software developed in industrial settings.

2.4 Detecting Code Smells

As discussed in the previous section, code smells may negatively affect software development. It is therefore useful to detect them, either manually or automatically. The literature mentions a number of arguments that speak against manual smell detection. Not only is manual detection time-consuming, unrepeatable, and does not scale [227, 257], it is also highly subjective [224, 223, 332, 275, 140, 276]. For example, experiments showed that developers who know a certain piece of code well tend to notice fewer smells than others who are new to that code [223, 224]. Furthermore, there are certain smells whose detection requires extensive context information, such as knowledge about inheritance relationships, third-party libraries, or historical developments [223, 224, 275, 276]. Moverover, different developers may employ different heuristics to detect a given code smell [332, 140]. Collaborative smell detection – developers work in groups to detect smells – has been shown to lead

to more consistent, less subjective results [275, 276]. However, it is still very time-consuming because the process remains manual.

Due to the subjectivity and inefficiency of manual code smell detection, tool automation is desirable. In this section, I describe the corresponding approaches, thus laying the groundwork for Chapter 4, in which I present my detection approach for variability-aware code smells. In particular, I describe detection techniques based on object-oriented metrics but also summarize work using machine learning, historical and lexical information, and visualizations.

2.4.1 Detection with Object-Oriented Metrics

Marinescu, both alone and in collaboration with Lanza, worked extensively on detecting code smells using object-oriented metrics [227, 229, 228, 230, 196]. As Lanza and Marinescu explain, a metric is "the mapping of a particular characteristic of a measured entity to a numerical value" (Lanza and Marinescu [196], Chapter 2). In the context of code smell detection, the measured entities are program elements (e. g., classes and methods), and the characteristics of those entities are, for example, the number of instance variables in a class or the number of statements within a method.

To determine whether an entity is smelly, Marinescu uses *detection strategies*. A detection strategy consists of *rules* that are combined with boolean operators, such as *or* and *and*. A rule, in turn, is a comparison between the metric value of an entity and a *threshold*. As an example, consider the detection strategy for the God Class smell depicted in Figure 2.1. According Marinescu, a God Class has three properties [227, 196]: First, it accesses an unusually high amount of data from foreign classes. Second, its methods are unusually complex. Thirdly, it lacks cohesion. All three properties are reflected in the detection strategy, which is a conjunction of three rules, each of them relying on a different metric. Specifically, the metrics are *access to foreign data (ATFD)*, *weighted method count (WMC)*, and *tight class cohesion (TCC)*.

The rules in the detection strategy in Figure 2.1 use two kinds of thresholds, *absolute* and *relative* ones. The rule $ATFD > FEW$ contains an example of an absolute threshold value. In particular, *FEW* is a constant that equates to 4 (Lanza and Marinescu [196], p. 18). The constant *ONE THIRD*, used in the third rule, is another absolute threshold. In contrast to these absolute values, the threshold *VERY HIGH* in the rule $WMC \geq VERY\ HIGH$ is relative. A formal definition of *VERY HIGH* is given elsewhere (Lanza and Marinescu [196], p. 15), but in a nutshell, it means the following: Given the *WMC* values of all classes in a software system, a *VERY HIGH* value is an outlier, that is, a value that is much larger than the average value of the *WMC* metric. Tying it all together, this detection strategy defines a God Class as a class that (1) accesses more than 4 attributes from other classes, (2) has a *WMC* value that is much larger than the average *WMC* value of other classes, and (3) has a cohesion value, as measured by *TCC*, that falls below $\frac{1}{3}$.

Similar to the God Class detection strategy from Figure 2.1, Marinescu formulated detection strategies for a total of eleven code smells [229, 196] and implemented a corresponding detection tool [229, 228, 226, 196]. The tool detects code smells

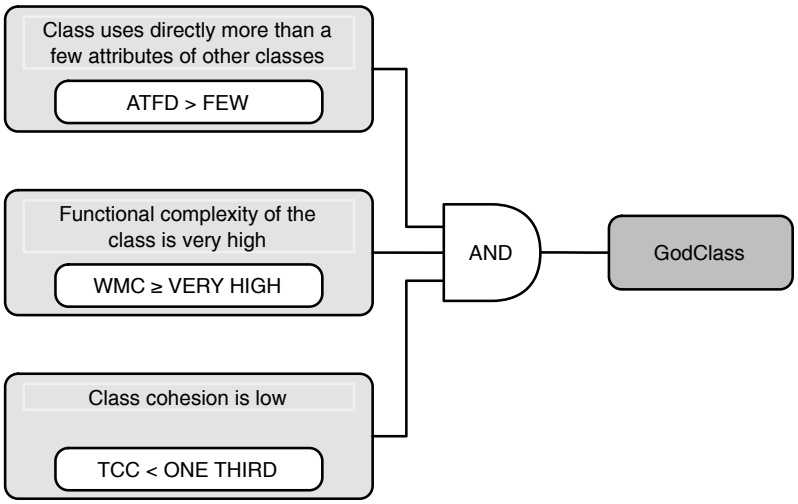


Figure 2.1: Detection strategy for a God Class *(Taken from Lanza and Marinescu [196], p. 81)*

with a high precision (70 % on average); recall however, was not reported [228]. To summarize, Marinescu has shown that detecting code smells automatically using object-oriented metrics is both feasible and scalable. Moreover, he has proven that this approach is generalizable to a variety of smells. Yet, certain smells remain that can only be detected by taking additional sources of information into account, such as evolutionary and textual information. I discuss the corresponding work in the following section.

Fundamentally, Moha's smell detection approach, Decor [257], is similar to Marinescu's. Decor, too, relies on object-oriented metrics that are compared to absolute and relative thresholds [257]. Nevertheless, there are two notable enhancements. First, Decor can also simple lexical information. For instance, it can detect classes with an unclear purpose by checking whether the class name contains "weasel words", such as "Make" or "Exec" [257]. Second, detection strategies can refer to other detection strategies. Thus, anti-patterns, which arise from the interaction between several code smells, can be described very succinctly in a Decor detection strategy.

Others languages and frameworks for code smell detection have been proposed, either with an explicit code smell focus [342] or more generally aimed at reverse engineering [362, 173]. Specifically, the FindSmells tool allows users define their own code smell detection strategies in a domain-specific language [342]. Although the language makes smell detection flexible, it lacks some of the interesting features that Moha's Decor approach already possessed. Others, in turn, propose frameworks for reverse engineering software systems, using Prolog-like [362] or SQL-like query languages [173], respectively. These frameworks have many use cases, and code smell detection is just one of them. Consequently, the exemplified detection strategies are trivial and contribute little to the field compared to Marinescu's and Moha's work.

Finally, Srivisut and Muenchaisri propose eight code smells and corresponding detection strategies for AOP [347, 346]. The novelty of their work lies in the metrics, which are specifically aimed at AOP. The relevance of their new smells, however,

seems questionable: In their validation on four different systems, only one smell occurred frequently whereas the other seven occurred only once or twice in a given system, or not at all.

2.4.2 Machine Learning

Designing detection strategies by hand and deciding on appropriate thresholds requires substantial expertise. To overcome this difficulty, various researchers combine metrics-based detection approaches with machine learning. The basic idea is to automatically learn the characteristics of smelly code from training data instead of manually encoding these characteristics into a detection algorithm. To this end, various machine-learning techniques that have been explored, including decision trees [184, 19], *support vector machines (SVMs)* [220, 219], and genetic programming [252, 168, 334, 167]. The results frequently exceed what traditional metrics-based detectors can achieve. For instance, Arcelli Fontana et al. report detection accuracies of 95 % in terms of F-measure [19]. Apart from detection itself, machine learning has also been applied to *rank* detection result, that is, to make the most severe smell instances appear at the top of the results list. *Bayesian belief networks* [171] as well as random forests [20] performed well in this context.

2.4.3 Additional Sources of Information

Not all code smells can be detected reliably using only object-oriented metrics. For example, the Shotgun Surgery smell (Fowler et al. [107], p. 80) is intrinsically characterized by the way that code is changed: It forces developers to make many scattered modifications in order to change a single piece of functionality. Scattered changes cannot be detected with object-oriented metrics, and further smells exist whose characteristics are equally hard to describe with object-oriented metrics alone. For this reason, several smell detection approaches take further sources of information into account. In particular, Palomba et al. mine co-change information from version control repositories to detect change-centric smells [283]. Other work extracts textual information from the source code (e. g., variable and method names) to find smells related to low cohesion or high coupling [33, 286]. Chatzigeorgiou et al., in turn, adapted page ranking algorithms known from web search engines to identify classes that are highly central to a system's design (i. e., God Classes) [49]. Others combine object-oriented metrics with distance measures and *Abstract Syntax Tree (AST)* analyses [105, 365, 366]. Further work considers the role that a class plays in a design pattern and uses this role information to reduce the amount of false-positives during smell detection [225]. Finally, Ligu et al. use a technique similar to mutation testing to detect instances of the Refused Bequest (Fowler et al. [107], p. 87) smell [207].

2.4.4 Visually Aided Detection

Humans have a great capacity to process visual data, which allows them to examine large amounts of information at once. This capacity has been exploited for code smell detection. Many visual code smell detection approaches are fundamentally visualizations of object-oriented metrics of a software system (e. g., [335, 195, 196,

68]). The elements of the analyzed system (e. g., classes or methods) are represented as graphical objects (e. g., boxes), and different metric values are expressed by varying the width, height, and color of these objects. Additional data is encoded in the layout, for example by placing similar objects close together. Finally, filtering techniques help users focus on objects of interest. By combining all of these ideas, clever visualizations make program elements with a code smell stand out as visual anomalies that will quickly catch the user's eye.

Apart from using visualizations to detect smells, visualizations can also help summarize and rank the results of traditional code smell detectors [88, 351]. For instance, Steinbeck proposes a combination of tree maps and heat maps to help developers focus on the most severe smell instances first and, thus, prioritize their maintenance efforts [351].

2.4.5 Summary

Purely manual code smell detection is both time-consuming and subjective. As a solution, semi- and fully automatic code smell detection approaches have been proposed. Many approaches to detect code smells rely on object-oriented metrics. The basic assumption behind metrics-based smell detection is that smelly program elements exhibit unusual metric values. Consequently, it is possible to detect smells by identifying elements with exceptional (combinations of) metrics. Designing a corresponding detection strategy involves selecting suitable metrics and deciding on appropriate threshold values. In early work, these decisions were made by experts who designed and implemented detection strategies manually. More recent work uses machine learning for this purpose, and often achieves better results than expert-designed detection strategies.

In addition to object-oriented metrics, other sources of information can be beneficial for code smell detection. For instance, mining changes from version control systems, as well as extracted textural information through information retrieval techniques have been shown to be effective.

Finally, researchers have explored the use of visualizations for code smell detection. Corresponding work ranges from interactive, graphical smell detection approaches to visualizing the distribution of code smells in a system.

2.5 Refactoring

In the previous sections, I have explained code smells in depth and also mentioned the term *refactoring* a number of times, but without explaining it in detail. In the next section, I illustrate the usefulness of refactoring with the help of an example. Afterwards, I will explain how it is possible to ensure that refactorings change a program's structure without also changing its behavior. Finally, I discuss tool support and summarize research findings on current refactoring practice.

2.5.1 A Refactoring Example

In this section, I give an example of how refactoring can be used. Specifically, I explain how refactoring helps remove Duplicated Code from a fictitious banking

application written in Java (see Listing 2.3). The application contains two classes representing bank accounts, `OwnAccount` and `ForeignLoan` (see Listing 2.3 (a)). `OwnAccount` is for accounts that are managed by the bank using the application; `ForeignLoan`, in turn, is for keeping track of loans that a customer has taken out with a foreign bank. A comparison of these classes reveals two problems: First, some of the code in `ForeignLoan` is identical to code in `OwnAccount` (highlighted in green Listing 2.3 (a)). Other code is conceptually similar, but named differently (highlighted in red). Second, despite their conceptual relation, the classes lack a common superclass. Consequently, it is impossible to write generic code that works on objects of both classes.

```
1  class OwnAccount {
2    double balance;
3    String accountNumber;
4    double overdraft;
5
6    double getBalance() { return balance; }
7    String getAccountNumber() {
8      return accountNumber;
9    }
10   double getOverdraft() { return overdraft; }
11 }
```

```
1  class ForeignLoan {
2    double balance;
3    String accountNo;
4    String bankCode;
5
6    double getBalance() { return balance; }
7    String getAccountNo() {
8      return accountNo;
9    }
10   String getBankCode() { return bankcode; }
11 }
```

(a) Before refactoring: Identical class members highlighted in green; nearly identical members highlighted in red.

```
1  class OwnAccount {
2    double balance;
3    String accountNumber;
4    double overdraft;
5
6    double getBalance() { return balance; }
7    String getAccountNumber() {
8      return accountNumber;
9    }
10   double getOverdraft() { return overdraft; }
11 }
```

```
1  class ForeignLoan {
2    double balance;
3    String accountNumber;                    ❶
4    String bankCode;
5
6    double getBalance() { return balance; }
7    String getAccountNumber() {              ❷
8      return accountNumber;                  ❶
9    }
10   String getBankCode() { return bankCode; }
11 }
```

(b) After Rename: Naming differences removed (changes highlighted in yellow).

```
1  abstract class Account {
2    double balance;                          ❷
3    String accountNumber;                    ❸
4
5    double getBalance() { return balance; }  ❷
6    String getAccountNumber() {              ❸
7      return accountNumber;
8    }
9  }
```

```
1  class OwnAccount extends Account {         ❶
2    ~~double balance;~~                      ❷
3    ~~String accountNumber;~~                ❸
4    double overdraft;
5
6    ~~double getBalance() { ... }~~          ❷
7    ~~String getAccountNumber() { ... }~~    ❸
8    double getOverdraft() { return overdraft; }
9  }
```

```
1  class ForeignLoan extends Account {        ❶
2    ~~double balance;~~                      ❷
3    ~~String accountNumber;~~                ❸
4    String bankCode;
5
6    ~~double getBalance() { ... }~~          ❷
7    ~~String getAccountNumber() { ... }~~    ❸
8    String getBankCode() { return bankCode; }
9  }
```

(c) Refactoring complete: Common code pulled up to a new superclass.

Listing 2.3: Example of an Extract Superclass refactoring

To solve these problems, an Extract Superclass refactoring is performed, whose purpose is to extract common functionality from two unrelated classes into a common superclass (Fowler et al. [107], p. 336). The refactoring happens in two phases. In the first phase, naming differences are eliminated so that similar code becomes actually identical code. This is achieved with a series of Rename refactorings (see Listing 2.3 (b)). Then, in the second phase, a new superclass is created and all the identical code is moved into this superclass with the help of Pull Up refactorings (see Listing 2.3 (c)).

In Listing 2.3 (b), I show the result of the first phase, renaming. `OwnAccount`, shown on the left remains unchanged. In `ForeignLoan`, however, shown on the right, two members underwent Rename refactorings to make the names identical to the ones in `OwnAccount` (see yellow highlights). A Rename refactoring encompasses two kinds of changes. First, the name of the program element (e. g. a field or method) must be changed at the place where the element is defined. Second, the whole program must be checked for references to the program element, and these references must be changed accordingly. Consequently, renaming `accountNo` to `accountNumber` involves one change on Line 3, where the field is defined, and a second change on Line 8, where the field is referenced (see the changes marked with ❶ in Listing 2.3 (b)). Renaming the `get` method from `getAccountNo` to `getAccountNumber`, in turn, only requires a single change (see the change marked with ❷) because there is no code calling this method. (Of course, if other code in the application contained such calls, further changes would be necessary.)

After eliminating the naming differences in phase one of the Extract Superclass refactoring, the second phase commences. In this phase, three refactorings are executed, whose purpose is to create a new superclass and to pull up duplicated code from the subclasses to that superclass. The resulting code is depicted in Listing 2.3 (c). Changes are highlighted in yellow, and the marks ❶, ❷, and ❸ indicate which changes belong together. In detail, refactoring ❶ creates a new superclass, `Account` (see upper part of Listing 2.3 (c)). The class is initially empty, and `extends` clauses are added to both `OwnAccount` and `ForeignLoan` to make them subclasses. Refactoring ❷, a Pull Up refactoring, moves the field `balance` and its `get` method, `getBalance`, to the superclass. To this end, a single copy of the corresponding field and method definition is created in the superclass, and the original definitions are deleted from `OwnAccount` and `ForeignLoan`. In the same way, the final refactoring (see mark ❸), moves the field `accountNumber` and its `get` method to the superclass.

As a result, Extract Superclass has consolidated code clones in `OwnAccount` and `ForeignLoan`. Moreover, it made the conceptual relationship between the two classes explicit, thus opening up further reuse potential in the rest of the code base.

2.5.2 Ensuring Behavior Preservation

One of the central properties of refactorings is *behavior preservation*. According to Griswold, who proposed the concept, behavior preservation means that given the same input, a program will compute the same output before and after refactoring (Griswold [119], p. 65). In their seminal survey of refactoring, Mens and Tourwé state more formally that a refactoring must perserve both the syntactic as well as the

semantic correctness of the program [249]. Syntactic correctness is easily checked with a parser: the refactored program must simply remain compilable. The difficult part is ensuring semantic correctness because it is generally impossible to decide whether an arbitrary program change preserves the semantics of the program [120]. Consequently, formal proofs of semantics preservation have only been constructed for a few specific program transformations in specific programming languages, but not for popular general purpose languages such as C++ and Java [249]. Fowler et al. avoid formal proofs altogether and essentially recommend to frequently run tests during refactoring (Fowler et al. [107], p. 89). However, most approaches in the literature try to provide stronger guarantees while, at the same time, avoiding the complexities of formal proofs [249]. The core idea is to rely on a conservative notion of behavior preservation, meaning they accept that some valid refactorings have to be ruled out as potentially incorrect (see [249], as well as Opdyke [278], p. 6f. for a concrete example). The approach I use in my thesis (see Chapter 6), called *precondition* checking, is one of these conservative approaches.

Preconditions. In order to automate refactorings, Griswold and Opdyke propose to split a refactoring into two phases, the checking of *preconditions* and the actual source code transformation [119, 278]. In this context, preconditions are the properties that must hold so that the subsequent source code transformation will be behavior-preserving. As an example, recall the Rename refactoring in Listing 2.3, which renames `ForeignLoan`'s `accountNo` field into `accountNumber`. The only precondition when renaming a variable is that the new name must not collide with the name of an existing variable (Opdyke [278], p. 60f.). If the name were already used, the change would either produce a name clash and render the code uncompilable; or worse, the renamed variable might be shadowed by a variable from another scope, thus changing the program's behavior. In the example in Listing 2.3, the check for name collisions is simple because first, none of `ForeignLoan`'s methods introduce local variables and because second, there are no super- or subclasses that might contain conflicting variable definitions. As a consequence, the check for name collisions only needs to verify that none of the existing fields are already named `accountNumber`. As this is not the case, the refactoring can proceed.

Precondition checks commonly rely on static analyses of the program's AST and the inheritance relationships between classes [249]. In certain cases, these analyses are easier to perform on a *Program Dependence Graph (PDG)* [100], which is a graph representation of the data and control dependencies within a program (Griswold [119], p. 59). However, PDG-based precondition checks are not strictly more powerful than checks on an AST, which is why I do not use them in my thesis.

Other approaches. Roberts observed that static analyses are sometimes too restrictive and envisioned analyses based on runtime information as a solution [306]. Kataoka et al. later showed that such an approach is feasible [163]. Mens et al. adapted work on graph transformations to overcome the limitations of Opdyke's overly conservative precondition approach [246, 248, 251] and constructed formal correctness proofs for several refactorings [248]. Others used program slicing [374] to guarantee behavior preservation of Extract Method refactorings [194, 193]. Finally, Sands proposed an improvement theory for functional programs, showing that refactorings derived from this theory increase a program's efficiency [320].

2.5.3 Tool Support

In the previous section, I discussed research into ensuring behavior preservation. The goal of this research is to automate refactoring and thus relieve developers from the burden of manually checking preconditions and changing the source code. The resulting tools, called *refactoring engines,* are available in many current IDEs and for many mainstream programming languages, such as Java, C++, and PHP.

Not all languages are supported equally well. For example, the Eclipse IDE currently provides more than twenty refactorings for Java [83], but only six refactorings for C/C++ [82]. ReSharper, a popular Visual Studio plug-in, is only slightly better, providing eight C/C++ refactorings [301]. As these examples show, developers working in some languages benefit from extensive tool support whereas others, who work in a different language, are mostly forced to refactor manually.

Beyond refactoring engines in IDEs, which are intended for interactive use, there are also tools to refactor a software system fully automatically (e. g., [261, 47, 4, 127, 189, 188]). These tools are useful when a code base has to undergo many similar refactorings or when the same refactorings are applied repeatedly. For example, Akers et al. discuss a tool that automatically refactors a C++ system from a component style to a CORBA-like framework [4]. *Refactoring feature modules,* in turn, combine refactorings with generative programming [189, 188]. Among other use cases, refactoring feature modules make it possible to generate variants of a software system with different *Application Programmer Interfaces (APIs).*

Finally, there is a category of tools that bridges the gap between code smell detection and refactoring (e. g., [250, 365, 366, 265, 264, 33]). For instance, the Stench Blossom plug-in for Eclipse adds unobtrusive hints to the IDE's editor window to alert the programmer to the presence of code smells [265, 264]. Programmers can either ignore the hints and continue coding, or they can click on them to receive advice on corresponding refactorings. The tool I present in Chapter 6 of my thesis takes up some of Stench Blossom's ideas. Specifically, my tool also highlights refactorable code in the editor and uses tooltips to provide additional information.

2.6 Summary

This chapter contained the background on code smells and refactoring for traditional software systems. In short, code smells are patterns in the source code that indicate design deficiencies. Not only do code smells make source code hard to understand, change, and extend, they also increase the likelihood of faulty changes. Developers should therefore be aware of code smells, and as a first step to raise this awareness, code smells must be detected. In this chapter, I have discussed several approaches to automate the detection of code smells, with a specific focus on approaches based on object-oriented metrics.

The design deficiencies indicated by code smells can be corrected through refactoring, which was the second major topic of this chapter. To refactor a program means to change its internal structure without changing its behavior. Refactoring is challenging, because even simple changes must obey numerous rules to avoid unwanted side-effects. Precondition checking is a frequently used technique to address this

challenge. Based on this technique, tool support for refactoring has been developed, which relieves the developer from the tedious and error-prone process of executing refactorings manually. I summarized the corresponding research in this chapter.

In my thesis, I transfer code smells and refactoring to *highly configurable software systems.* Such systems are special in that they correspond not just to a single program, but to a range of similar, yet distinct programs. Configurability affects code smells and refactoring in various ways, but before I can discuss this in detail, I have to explain the fundamentals of highly configurable software systems in the following chapter.

3. Highly Configurable Software Systems

This chapter shares material with the VaMoS '15 paper "Code Smells Revisited: A Variability Perspective" [96], the SANER '17 paper "Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line" [95], as well as the GPCE '17 paper "How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness" [98].

Traditionally, one software system corresponds to exactly one software product. However, sometimes, there is a need for multiple software products that satisfy similar requirements but also exhibit certain differences. Such a group of related software products is called a *software product family* [288] and one way to build a product family efficiently is to design it as a *highly configurable software system* (a. k. a. *Software Product Line (SPL)*) [53, 61, 13]. The Linux kernel is a well-known, successful example of a highly configurable software system. Thanks to being highly configurable, customized Linux kernels run on systems ranging from smartphones to supercomputers, with platforms as varied as Intel's x86 architecture, PowerPC, and ARM. This thesis is concerned with the code quality of highly configurable software systems, and in this chapter, I provide the necessary background on how such systems are built. In particular, I explain the development process for highly configurable software systems, *Software Product Line Engineering (SPLE)*, elaborate on modeling and implementation aspects, and, most importantly, discuss the implications of configurability for the quality of the source code.

In the second half of this background chapter, I focus on an alternative approach to building software product families. This alternative is called *clone-and-own* [91, 245, 79, 317, 349] and it plays a central role in my study of refactoring that I present in Chapter 6. In a nutshell, clone-and-own means that an existing software product is copied and adapted to fit a new set of requirements. Since clone-and-own involves copying entire code bases, it introduces *code clones* [312] at a massive scale, with

corresponding negative consequences [79]. For example, if a bug is discovered in a cloned piece of code, the bugfix has to be propagated to all the products that contain the clone. In the best case, this involves a lot of repetitive work; in the worst case, some products are forgotten and a bug that was supposed to be fixed continues to persist [147, 79]. As I show in Chapter 6, code clones are not only a source of problems, they can also be exploited to facilitate the transition from clone-and-own to SPLE. Therefore, at the end of the present chapter, I explain the background of clone-and-own development and code clones, with a particular focus the negative effects and on techniques to detect and remove clones.

3.1 Software Product Line Engineering

Software is now used much more pervasively than it used to be several decades ago. The earliest software products were developed to perform scientific simulations, decipher encrypted texts, and solve other mathematical problems. While these early software products were often deployed on just a few machines, the availability of ever cheaper computers soon turned software into a mass product. This means that software products for word processing, graphics editing, and many other tasks were developed once and deployed on thousands or even millions of machines. Today, software is part of many products that we use on a regular basis. This encompasses not only software on desktop computers, but also on smartphones, cars, or coffee machines.

As the use of software has grown over time, the way it is developed has changed, too. The early software products and also the mass-produced ones were developed to fulfill a fixed set of requirements. If those requirements changed, the product was either evolved and enhanced, or a new product was developed from scratch. But with the advent of *mass customization*, which continues to replace mass production in many industries, including the software industry [185], software had to become more flexible. Consequently, much of today's software is developed in a way that severals sets of requirements, sometimes even competing ones, can be satisfied. Such software systems allow the creation of multiple, distinct software products, each of which is customized to best serve a specific purpose. For example, Google's Web browser, Chrome, is available for three different desktop operating systems (Windows, Linux, and Mac OS X), and also as a mobile app for Android and iOS devices. While the desktop variants are controlled by mouse and keyboard, and are optimized for high performance, the mobile variants offer touch operation and are more energy efficient. Hence, Google Chrome is not a single software product, but a *software product family* [288, 61, 13].

The members of a software product family differ in the features that they provide but more importantly, they also share a lot of commonalities. Hence, a product family should not be treated as separate products because that would waste a lot of reuse opportunities. Instead, a product family should be treated as a whole so that each common feature is developed, maintained, and evolved only once. The developers of Google Chrome and of many other software systems (e. g., Linux, Eclipse, MySQL, to name only a few) have managed to do that because these software systems are developed as *highly configurable software systems.*

According to Clements and Northrop, a highly configurable software system (also called *Software Product Line (SPL))* is "a set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [53]. This definition mentions a central concept for highly configurable software systems, the concept of a *feature*. A feature is an increment in functionality that is relevant to a particular stakeholder [13]. The exact nature of what a feature encompasses varies. It ranges from things that matter to users, such as the option to pay by credit card in an online shop SPL, to technical aspects that only matter to developers, such as a workaround for a bug in a specific operating system, but also includes non-functional properties, such as the amount of memory and processing power consumed. In the context of my thesis, a feature can be anything that falls in this broad spectrum of meanings. Generally, features characterize the commonalities and variations of the products (also referred to as *instances* or *variants*) of an SPL.

Highly configurable software systems offer a number of benefits, including shorter time to market, reduced cost, and improved quality (Clements and Northrop [53], p. 17). Whether or not these benefits materialize depends on the ability to develop what Clements and Northrop call the *core assets* – reusable artifacts that can be configured and combined in flexible ways. Designing and implementing these core assets poses challenges that traditional software development processes fail to address. For example, variations between products must be planned, release cycles of many products have to be synchronized, and developers have to reason about potential side effects when changing a reused artifact. For these reasons, Clements' and Northrop's definition states that an SPL is developed "in a prescribed way". This prescribed way is known as *Software Product Line Engineering (SPLE)*, and in the following sections, I explain the domain and application engineering levels of SPLE, discuss how commonalities and variations are modeled, and describe how configurability is implemented on the source code level.

3.2 Domain and Application Engineering

In this section, I explain SPLE using the terminology of *Feature-Oriented Software Development (FOSD)*, an approach to SPLE that puts features at the center of every stage of the development process [15]. Not all highly configurable software systems are developed strictly according to SPLE, which is why SPLs are sometimes considered a subcategory within the broader category of highly configurable software systems [336]. I do not make this distinction in this thesis and instead agree with Sincero et al., who argues that many of the benefits attributed to SPLs can be had even if the development process deviates from SPLE in certain aspects [336].

An overview of the feature-oriented SPLE process is depicted in Figure 3.1. As illustrated in the overview, development tasks in SPLE are structured according to two orthogonal dimensions. The first dimension distinguishes between *problem space* and *solution space,* and the second dimension distinguishes between *domain engineering* and *application engineering* (Apel et al. [13], Chapter 2). Roughly speaking, problem space relates to identifying and organizing requirements, that is,

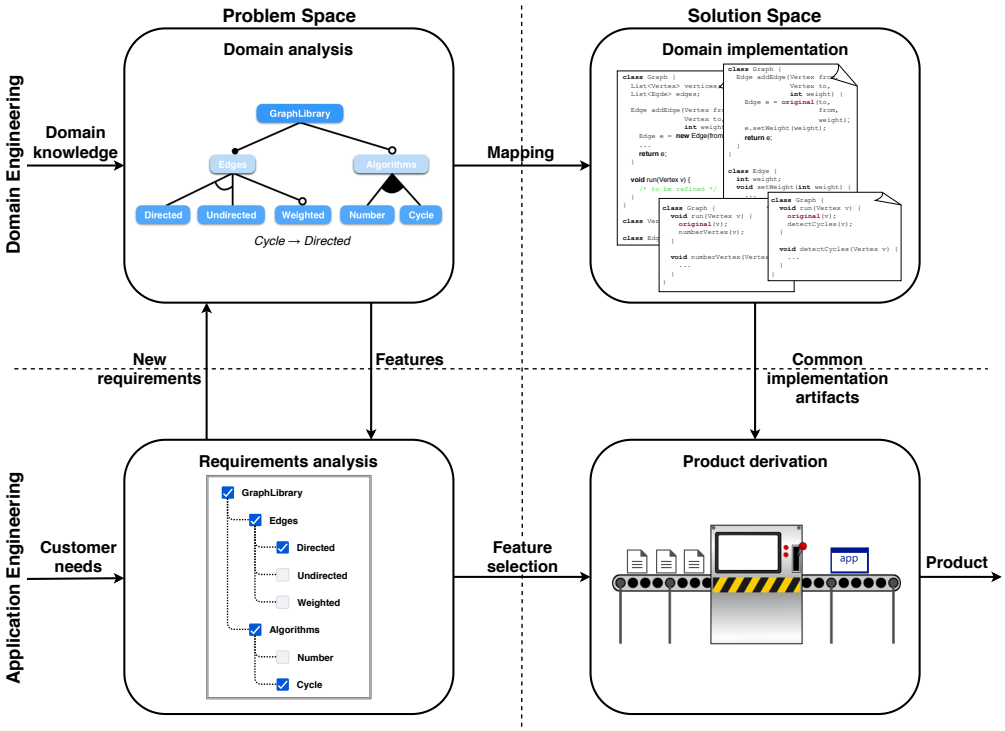


Figure 3.1: Overview of a software product line engineering process *(Adapted from Apel et al. [13])*

the problems that the products of the SPL shall solve. Solution space, in turn, is about solving these problems by implementing source code, models and other artifacts, from which products can be built. Regarding the second dimension, domain engineering focuses on all products as a whole, whereas application engineering focuses on an individual product. To summarize, domain engineering is "development *for* reuse" and application engineering is "development *with* reuse" (Apel et al. [13], Chapter 2). Combined with each other, these two dimensions yield the four groups of tasks that are illustrated in Figure 3.1: (1) domain analysis, (2) domain implementation, (3) requirements analysis, and (4) product derivation. I describe these groups of tasks next.

Domain analysis encompasses all domain engineering tasks that belong to problem space. The goal is to identify all the requirements that the SPL shall satisfy, frame them as features, and organize these features so that commonalities and variabilities become apparent. To this end, domain analysis draws on the information elicited during requirements analysis, which is the first phase of application engineering. Information also flows in the opposite direction as requirements analysis draws on the features that domain analysis has already uncovered. This exchange, illustrated by the arrows connecting domain analysis and requirements analysis in Figure 3.1, shows that domain and application engineering are not entirely separate but benefit from communication and feedback.

A crucial task during domain analysis is to *scope* the domain, that is, to decide what the products in the SPL should and should not be able to do. If the scope is too narrow, possibly useful future products are excluded and reuse potential is wasted.

Conversely, if the scope is too large, the SPL becomes overly complex because it tries to foster reuse among products that actually have little in common.

When the requirements have been collected and the scope is clear, the features of the SPL are identified. This way, the similarities and differences of the requirements and, consequently, of the products in the SPL are documented. The next step is to establish how the features relate to each other, for example, whether features mutually exclude (are alternative to) each other, whether they can co-exist, or whether certain features are specializations of other, more general features. These relationships are typically expressed in feature models, which I cover in more detail in Section 3.3.

Domain implementation relates to all the tasks of domain engineering in solution space. They are depicted in the upper right part of Figure 3.1. The goal is to implement the features that domain analysis has uncovered in the form of source code, models (e. g., class diagrams), database schemas, and so on. The resulting implementation artifacts constitute the reusable *core assets* of the SPL. Core assets are combined with each other in many different ways to create a product. To make these combinations possible, core assets must offer customization options – *variation points.* There are multiple mechanisms to implement such variation points. I cover the mechanisms that are most relevant to this thesis in Section 3.4.

After domain analysis and domain implementation, domain engineering is complete. The domain of the product line is scoped, the relationships between features are documented, and a collection of reusable artifacts has been created. Based on these features and artifacts, it is now possible to derive concrete software products. This is the focus of the application engineering level of SPLE (see lower part of Figure 3.1), which consists of requirements analysis (in problem space) and product derivation (in solution space).

Requirements analysis entails eliciting requirements of an individual product by interviewing customers, reading documents, and so on, much in the way that requirements analysts do in traditional software engineering [141]. A key difference is that a requirements analyst in SPLE can make use of the knowledge from domain analysis. In the ideal case, the analyst only has to map requirements to the corresponding features, thus creating a *configuration* that represents the product. In other cases, the analyst may uncover a requirement for which no corresponding feature exists. If this happens, there are basically three options (Apel et al. [13], Chapter 2): The first option is to decide that the requirement is out of scope of the SPL and to not build the product. The second option is to change the scope, which involves going back to domain engineering, introducing the necessary features and implementing the corresponding assets. Such a scope change breaks the usual separation between domain and application engineering and instead requires that the two interact. In Figure 3.1, this interaction is symbolized by the two arrows connecting domain analysis and requirements analysis, one of them facing downward, the other one upward. Finally, the third option to deal with a requirement without a corresponding feature is to build the product in a hybrid fashion, using a combination of SPLE and traditional software engineering. Based on the reusable assets, a product is built that is as close as possible to what the customer wants.

Subsequently, this product is enhanced by implementing the missing requirements, but without feeding the enhancements back into the SPL code base.

Product derivation encompasses the solution space tasks of application engineering, and it commences when the requirements of the product are clear. In this step, the core assets that implement the required features are gathered, the variation points are configured to match the product at hand, and finally the product is *generated*. As indicated by the assembly line pictogram in the lower right part of Figure 3.1, the ultimate goal in FOSD is to automate product derivation, for example by using generative programming techniques. However, depending on the variability mechanism, product derivation can also involve writing glue code, that is, variant-specific code that ties together the reused core assets, as well as adding functionality that is unique to the product being derived.

3.3 Modeling Variability

One important task during domain analysis is to model the commonalities and differences between products of the SPL. There are several variability modeling languages to achieve that, which differ in terms of expressiveness and in the way the variability model is represented. Linux, for example, uses the KConfig language, which describes the variability model in textual form and offers advanced concepts such as *tristate* features.[1] A tristate feature can be configured in one of three ways: built into the kernel (state `y`), not included at all (state `n`), and included as a dynamically loadable module (state `m`). In contrast to KConfig, many other modeling languages only allow boolean features: a features is either included or it is excluded, without any third option. I use one of these simpler languages throughout my thesis, FODA *feature models (FMs)*, which result from a process called *feature-oriented domain analysis (FODA)* [150]. I chose FODA FMs because they visualize the relationships between features in an easy-to-understand tree structure, because they are sufficiently expressive for my needs, and because they are commonly used in the SPL literature. In the remainder of my thesis, I will simply refer to these models as *feature models (FMs)*.

In Figure 3.2, I show the FM for an exemplary product line of graph algorithms, the GraphLibrary SPL (adapted from Lopez-Herrejon and Batory [214]). FMs have a tree structure, which puts features in a parent-child relationship. In a nutshell, parent features express more general concepts, whereas child features represent specializations (or refinements) of their respective parent features. In Figure 3.2, the root feature is *GraphLibrary*, and we can imagine that it encompasses general functionality of the domain, such as providing classes for vertices and edges. Its children, *Edges* and *Algorithm*, cover more specific aspects, namely different kinds of edges (e. g., directed or undirected) as well as various graph algorithms.

While visualizing generalization / specialization relationships between features is one goal of FMs, they also define the set of *valid configurations* of an SPL. A *configuration* is a subset of the available features. Given a specific configuration, we say that a feature is *selected* if it is part of that configuration and *not selected* otherwise.

[1] https://www.kernel.org/doc/Documentation/kbuild/kconfig-language.txt

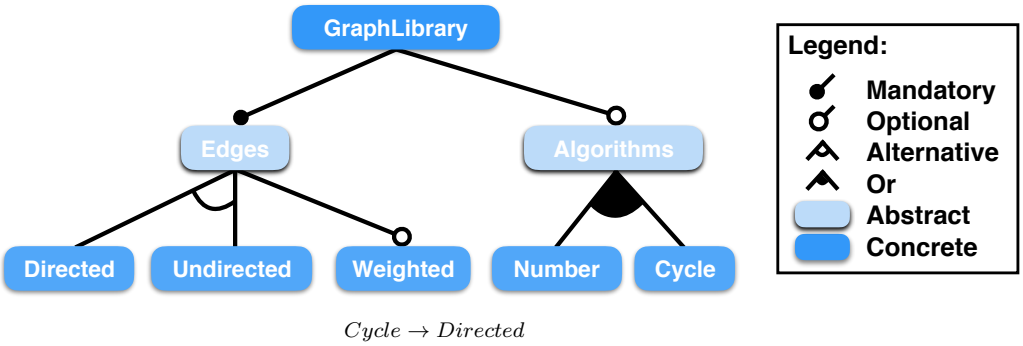


Figure 3.2: FODA feature model of the GraphLibrary product line

In every domain, there will be constraints as to which feature combinations are meaningful or not. For example, a graph cannot be directed and undirected at the same time. An FM expresses these domain constraints, thereby defining which configurations are *valid*. First, a valid configuration must include the root feature, called *GraphLibrary* in the example FM in Figure 3.2. Second, a child feature can only be selected if its parent is selected. For example, it would be invalid to select the *Cycle* feature without selecting the base functionality from *GraphLibrary*. Finally, given that a parent feature is selected, there are several rules, which I will cover next, that govern which of its children can be selected. Assuming that a configuration satisfies all these rules, we say that the configuration is a *valid configuration*. A valid configuration corresponds to a product (a. k. a. *instance*) of the SPL.

The rules that govern whether or not a child feature can be selected are as follows. First, a child feature can either be *mandatory* or *optional*. A mandatory feature *must* be selected, an optional feature *can* be selected (but it does not have to be). Second, child features can form an *alternative group* or an *or-group*. Features in an alternative group mutually exclude each other, meaning that *exactly* one of them must be selected. In an or-group, by contrast, *at least* one feature must be selected (but more are allowed). Third, any additional restrictions not expressible through these means can be added as a *cross-tree constraint*, a propositional logic formula that is written below the feature diagram. To illustrate how these rules are used to model the GraphLibrary SPL, consider Figure 3.2. The FM expresses that every graph library must include edges (feature *Edges* is mandatory), but it is possible to leave the algorithms out (feature *Algorithms* is optional). Moreover, edges can either be directed or undirected, but not both (features *Directed* and *Undirected* form an alternative). Furthermore, if *Algorithms* are selected, then at least one of the concrete algorithms must be selected as well (features *Number* and *Cycle* form an or-group). Lastly, selecting the *Cycle* algorithm requires *Directed* edges (see the cross-tree constraint below the feature tree).

A final property of the FM in Figure 3.2 is that it includes both *concrete* and *abstract* features. Concrete features (shown in a darker shade of blue) correspond to one or several implementation artifacts, which is the usual case. The algorithms *Number* and *Cycle* are some of the concrete features in the example FM. Abstract features (shown in a lighter shade of blue), by contrast, lack corresponding implementation artifacts [359]. Their purpose is to improve the clarity of a model by grouping

features. In the example FM, the abstract feature *Algorithms* groups the *Number* and *Cycle* algorithms together.

3.4 Implementing Variability

After identifying the features and modeling their relationships during domain analysis, they must be implemented during domain implementation. Most implementation work happens in a general purpose programming language, such as C or Java, and much of the resulting source code will be *base code,* that is, code that is common to all products. However, there is a second kind of code in a highly configurable software system beyond base code. It is called *feature code*. Feature code is the code that implements a configurable feature, and it must be possible to include or exclude such code depending on a product's configuration. To this end, the underlying programming is combined with a *variability mechanism.*

Different *variability mechanisms* exist, which can be subdivided into *annotation-based* and *composition-based* mechanisms [155, 13]. Annotation- and composition-based variability mechanisms differ in terms of modularization and *separation of concerns* [287, 72, 73], and these differences have a fundamental effect on how the variable code base is structured. Since the structure of code is at the heart of the code smell concept (see Chapter 2), I hypothesize in my thesis that different variability mechanisms also lead to different code smells. I come back to this point in Section 3.4.3 of this chapter and explore it further in Chapter 4.

3.4.1 Composition-Based Mechanisms

Composition-based mechanisms *physically separate* concerns, which means that all artifacts (code and non-code) that belong to a certain feature are modularized in one cohesive unit [155, 13]. In my thesis, I focus on *Feature-Oriented Programming (FOP)* as a representative of composition-based mechanisms. In FOP, this unit is called a *feature module*, and directly corresponds to a feature in the FM [297, 32]. There are several tools that support the implementation of SPLs using FOP, such as Ahead [32] and FeatureHouse [16]. Beyond FOP, other composition-based variability mechanisms have been explored, for instance, component frameworks, plug-in architectures, and *Aspect-Oriented Programming (AOP)* [333, 22, 340, 117]. Moreover, *Delta-Oriented Programming (DOP)*, an extension of FOP, has been proposed that allows more complex refinements [322]. In my thesis, though, I use FOP as the composition-based variability mechanism and specifically its FeatureHouse [16] flavor. I made this choice because FOP has a sound formal foundation [18] and a clear focus on physical separation of concerns, and because tool support (e. g., the Eclipse-based FeatureIDE [358]) and open-source case studies[2] are available.

In Listing 3.1, I show a selection of the feature modules of the FOP implementation of my GraphLibrary SPL (see Listing 3.2 for the FM). For brevity, parts of the implementation have been omitted, which is indicated by `...` in the listing. The feature module *GraphLibrary*, depicted in part (a) of the listing, *introduces* the fundamental classes, such as `Graph`, `Edge`, and `Vertex` (see Lines 1–15, 16, and 17).

[2]See http://spl2go.cs.ovgu.de/ for a selection of SPL case studies.

```
1  class Graph {                    Feature GraphLibrary
2    List<Vertex> vertices;
3    List<Edge> edges;
4
5    Edge addEdge(Vertex from, Vertex to,
6                 int weight) {
7      Edge e = new Edge(from, to);
8      ...
9      return e;
10   }
11
12   void run(Vertex v) {
13     /* to be refined by Algorithms */
14   }
15 }
```

```
16 class Vertex { ... }
```

```
17 class Edge { ... }
```

(a)

```
1  class Graph {                    Feature Weighted
2    Edge addEdge(Vertex from, Vertex to,
3                 int weight) {
4      Edge e = original(to, from, weight);
5      e.setWeight(weight);
6      return e;
7    }
8  }
```

```
9  class Edge {
10   int weight;
11   void setWeight(int weight) {...}
12 }
```

(b)

```
1  class Graph {                    Feature Number
2    void run(Vertex v) {
3      original(v);
4      numberVertex(v);
5    }
6
7    void numberVertex(Vertex v) {...}
8  }
```

(c)

```
1  class Graph {                    Feature Cycle
2    void run(Vertex v) {
3      original(v);
4      detectCycles(v);
5    }
6
7    void detectCycles(Vertex v) {...}
8  }
```

(d)

Listing 3.1: FOP-based implementation of the GraphLibrary product line

A feature module is said to *introduce* a program element (e. g., a class, method, or field) if no other feature module has previously defined the respective program element. Along with the class `Graph`, the method `addEdge(Vertex, Vertex, int)` is introduced, whose responsibility is to attach a new edge to a given `Graph` object, as well as the method `run(Vertex)` (see Lines 5–10 and 12–14, respectively). The body of `run(Vertex)` is initially empty but in combination with the feature modules in part (c) and (d) of the listing, `run(Vertex)` serves as a hook for running algorithms.

The feature module for feature *Weighted* is depicted in Listing 3.1 (b). This module *refines* the classes `Graph` and `Edge` that were introduced by *GraphLibrary*. A *refinement* in FOP adds new fields or methods to an existing class or overrides existing methods. Refinement is similar to inheritance in *Object-Oriented Programming (OOP)*, with the key difference that inheritance extends a subclass relative to its superclasses, whereas refinement makes extensions in the same class, relative to the class's introduction and refinements in other feature modules. Specifically, Lines 10–12 in Listing 3.1 (b) add the field `weight` and the method `setWeight(int)` to the class `Edge`. Lines 2–7, in turn, constitute a method refinement. Specifically, method `addEdge(Vertex, Vertex, int)`, originally introduced by *GraphLibrary* (see Lines 5–10 in Listing 3.1 (a)), is refined. First, the refined method invokes the original behavior by way of an `original` call (see Line 4 in Listing 3.1 (b)). Afterwards, the original behavior is extended by executing two additional statements (see Lines 5 and 6). As this example shows, an `original` call is similar to a `super` call in an object-oriented language, with the difference that a `super` call refers to

a method in a superclass, whereas an `original` call refers to a method in the same class but from another feature module.

The feature modules for the algorithms, *Number* and *Cycle*, are depicted in the lower part of Listing 3.1, in subfigures (c) and (d), respectively. Both feature modules refine the class `Graph` and are implemented in a similar fashion. On Line 2, they refine the `run(Vertex)` method. To this end, they use an `original` call on Line 3 followed by a regular method call on Line 4 that invokes the respective algorithm (`numberVertex(Vertex)` or `detectCycles(Vertex)`). Hence, the `run(Vertex)` method that *GraphLibrary* introduces as an empty hook is refined so that it will run the algorithms that are selected in a given product.

Based on these feature modules, it is possible to derive products of the GraphLibrary SPL. Product derivation in FOP is more complex than just compiling each file with a regular compiler. The reason is that a class in FOP does not necessarily reside in a single file. Instead it may be divided into multiple introductions and refinements. Hence, before compiling a class for a given product, all the relevant introductions and refinements must be *composed* first. (Note that not just classes need to be composed but also artifacts written in other languages, such as XML documents.) The composition idea is borrowed from function composition in mathematics and also uses the symbol • as the composition operator. As an example, the composition of *GraphLibrary* and *Number*, written as *Number* • *GraphLibrary*, is shown in Listing 3.2. The tool performing the composition is called a *composer* and its output depends on the chosen composition technique. FeatureHouse, the composer that was used to produce the code in Listing 3.2, uses *superimposition* [16] as the composition technique.[3] As is shown in Listing 3.2, the result of superimposing *Number* on *GraphLibrary* encompasses all the code from *GraphLibrary*, such as the definition of class `Graph` including its fields `vertices` and `edges` (see Lines 1–3) and the method `addEdge(Vertex, Vertex, int)` (see Lines 5–9). On Lines 13 and 16, we see code from the *Number* feature. Remember from Listing 3.1 (a) and (b) that both *GraphLibrary* and *Number* define a class named `Graph`. Superimposition has connected these definitions with each other by adding the `numberVertex(Vertex)` method to the composed `Graph` class. Similarly, superimposition has merged the `run(Vertex)` methods that both *GraphLibrary* and *Number* define. On Lines 11–14, the resulting method definition is shown, which closely resembles the defintion from `Number` except that the `original` call was replaced with the method body from the *GraphLibrary* feature module. Hence, the composed `run(Vertex)` method behaves exactly as expected: First, it executes the original behavior of *GraphLibrary* and afterwards, it invokes the behavior of *Number*.

In certain situations, the *composition order* is important, that is, the order in which feature modules are composed. For example, the correctness of the precondition checks for the refactorings I propose in my thesis (see Chapter 6, especially Section 6.4) depends on the composition order. As an illustration, consider Listing 3.3, where two variants of the `run(Vertex)` method of class `Graph` are depicted that result from different composition orders. In particular, the compo-

[3]Note that the actual output of FeatureHouse is more complex than the code shown in Listing 3.2. To ease presentation, I show a simplified but functionally equivalent version. Details about the way that FeatureHouse implements superimposition can be found elsewhere [16].

```
1  class Graph {
2    List<Vertex> vertices;
3    List<Edge> edges;
4
5    Edge addEdge(Vertex from, Vertex to, int weight) {
6      Edge e = new Edge(from, to);
7      ...
8      return e;
9    }
10
11   void run(Vertex v) {
12     /* to be refined by Algorithms */
13     numberVertex(v);
14   }
15
16   void numberVertex(Vertex v) { ... }
17 }
```

Feature module *Number* superimposed on feature module *GraphLibrary,* producing the composition *Number* • *GraphLibrary.*

Listing 3.2: Composition of two feature modules of the GraphLibrary product line

sition *Cycle* • *Number* • *GraphLibrary* is shown in Listing 3.3 (a) and the composition *Number* • *Cycle* • *GraphLibrary* is shown in Listing 3.3 (b). In Listing 3.3 (a), `numberVertex(Vertex)` is executed *before* `detectCycles(Vertex)` (see Lines 6 and 7) because feature *Number* is composed *before* feature *Cycle.* In Listing 3.3 (b), by contrast, the order of execution is reversed (see Lines 6 and 7) because feature *Number* is composed *after* feature *Cycle.* Consequently, two products based on the same code base and the same configuration can exhibit different behaviors depending on the composition order. Therefore, analyses and transformations of FOP-based SPLs must take the composition order into account to produce correct results. In FeatureIDE, the default composition order is the level order of the features in the feature diagram. However, other composition orders, including fully customized ones, are possible.

For simplicity, the previous examples only showed the composition of a few features, but the process can be easily extended to an arbitrary number of features. In this

```
1  class Graph {
2    ...
3
4    void run(Vertex v) {
5      /* to be refined by Algorithms */
6      numberVertex(v);
7      detectCycles(v);
8    }
9
10   ...
11 }
```

(a) *Cycle* • *Number* • *GraphLibrary*

```
1  class Graph {
2    ...
3
4    void run(Vertex v) {
5      /* to be refined by Algorithms */
6      detectCycles(v);
7      numberVertex(v);
8    }
9
10   ...
11 }
```

(b) *Number* • *Cycle* • *GraphLibrary*

Listing 3.3: Effects of different of composition orders in FOP

way, given all the features in a product's configuration, the code base for that product is generated by composing the corresponding feature modules. The composed code base is then compiled in the usual fashion to produce an executable product.

3.4.2 Annotation-Based Mechanisms

As explained in the previous section, composition-based variability mechanisms such as FOP modularize all the code that implements a specific feature into one cohesive unit. In contrast to this physical separation of concerns, annotation-based mechanisms provide a *virtual separation* of concerns [155, 13]. Virtual separation of concerns means that all features are implemented in a single code base, and annotations are used to mark code fragments that correspond to a given feature. In order to create a specific product, annotated code is optionally excluded or modified by a preprocessor before it is passed on to the compiler [13]. Hence, *conditional compilation* is a central idea of annotation-based variability mechanisms: In contrast to non-annotated code, which is always compiled into a product, annotated code is only compiled if a certain condition is met during preprocessing.

There are many preprocessors that provide conditional compilation. One of the oldest and most well-known preprocessors is the C preprocessor, CPP [166]. Beyond that, there are many preprocessor with similar capabilities, such as XVCL or the JAVA preprocessors JPP and ANTENNA [30, 144, 279, 11]. Most preprocessors employ textual annotations, but other approaches have been explored as well. CIDE, for instance, uses colors to annotated variable code fragments [153], which has the advantage that annotations do not clutter the code base. Among these choices, I focus on the CPP in my thesis as the representative of annotation-based variability mechanisms because it continues to be widely-used in industrial and open-source projects [13].

The CPP, originally invented for the C programming language [166], is a general-purpose, text-based preprocessor. The way CPP preprocesses text is controlled by directives, for instance for macro definition (`#define`) or file inclusion (`#include`). More important to my thesis are the conditional compilation directives, namely `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif`. In a nutshell, feature code is annotated with these directives so that the preprocessor can conditionally exclude this code from compilation. Whether to include or exclude feature code depends on whether the conditional expression of an `#if` (or `#elif`) is `true` or `false`. I refer to such expressions as *feature expressions*. The most common form of a feature expression is to test whether a name, which I call a *feature constant*, is defined as a preprocessor macro or not. In fact, this form is so common that directives of the form `#if defined(X)` can be abbreviated to `#ifdef X`. Hence, I will synonymously use `#ifdef` in my thesis to refer to any of the CPP's conditional compilation directives. Simple feature expressions can be combined to form more complex ones using logical operators (such as `&&` or `||`). Apart from testing whether a feature constant is defined, a conditional compilation directive can also involve an arithmetic expression. For example, the GNU C library, GLIBC, contains the following expression: `defined __GNUC__ && __GNUC__ >= 2`.[4] The first part of the expres-

[4] See Line 54 in `libio/oldstdfiles.c` at revision `58b587c1f8` in git://sourceware.org/git/glibc.git

sion tests whether the feature constant `__GNUC__` is defined. If so, the second part, an arithmetic expression, tests whether `__GNUC__`'s value is greater than or equal to 2. Thus, the expression ensures that only a GNU C compiler of at least version 2 is used to compile the respective piece of feature code.

I illustrate the use of CPP conditional compilation with the help of the "Hello world" program shown in Listing 3.4. Depending on how this code is compiled, three distinct program variants can be created. The first variant simply prints "Hello world!"; the second additionally asks the user to guess a number and always congratulates him or her for guessing right; the third variant also asks for a number but always claims the user guessed wrong. The inclusion of the corresponding feature code (Lines 4, 8–9, 11, and 13) depends on the feature constants `GUESS_POS` and `GUESS_NEG`. Specifically, if `GUESS_POS` or `GUESS_NEG` are defined, Lines 4 and 8–14 are preprocessed further. If, on the contrary, neither feature constant is defined, all of these lines are excluded. It is possible to *nest* `#ifdef`s, in which case the processing of the inner `#ifdef` depends on how the outer `#ifdef` is evaluated. In Listing 3.4, the `#ifdef` spanning Lines 10–14 is nested. Considering this `#ifdef` in isolation suggests that the program will always respond negatively (i. e., execute the statement on Line 13) if `GUESS_POS` is undefined. However, this is not entirely correct because this `#ifdef` is nested within the `#ifdef` starting on Line 7. If neither `GUESS_POS` nor `GUESS_NEG` are defined when this outer `#ifdef` is preprocessed, the inner `#ifdef` will be skipped entirely.

```
1  #include <stdio.h>
2  int main(int argc, char **argv) {
3  #if defined(GUESS_POS) || defined(GUESS_NEG)
4    int x;
5  #endif
6    printf("Hello world!\n");
7  #if defined(GUESS_POS) || defined(GUESS_NEG)
8    printf("What is my favorite number? ");
9    scanf("%d", &x);
10 #  ifdef GUESS_POS
11   printf("Yes, %d is my favorite number!\n", x);
12 #  else
13   printf("No, %d is my favorite number!\n", x+1);
14 #  endif
15 #endif
16   return 0;
17 }
```

Listing 3.4: A "Hello world" program with annotation-based variability

For completeness I want to mention that highly configurable systems frequently combine preprocessor-based variability with *build system variability*. Build system variability essentially means that certain source files are either compiled into the product or not depending on a given configuration (Apel et al. [13], Chapter 5). There are multiple possibilities to implement this form of conditional compilation. One possibility is to generate build scripts that are tailored to a specific configuration. The GNU AUTOTOOLS[5] framework is an example of such an approach.

[5] https://www.gnu.org/software/automake/manual/html_node/index.html

Another possibility is the use of *conditionals* of the Makefile language.[6] Such conditionals work similarly to `#ifdef`s but instead of controlling the inclusion or exclusion of lines of source code, conditionals control whether or not a Makefile rule is evaluated, and thus, whether or not a file is compiled into the product. Build system variability is a coarse-grained variability mechanism that is only applicable if an entire file consists of feature code. Some researchers propose it as the sole variability mechanism [350] but this is rare. More commonly, systems take the Linux route and combine the coarse-grained variability offered by the build system with the fine-grained variability of preprocessor annotations [40, 70].

3.4.3 How Variability Mechanisms Affect the Shape of Feature Code

In this section, I discuss how annotation-based and composition-based variability mechanisms affect the shape of the code in different ways. To this end, I complement the composition-based implementation of the GraphLibrary SPL shown in Listing 3.1 with the annotation-based implementation shown in Listing 3.5. Although cpp directives are more common in C or C++ code, I chose Java as the host language to keep both implementations as close to each other as possible.

```
1  class Graph {
2    List<Vertex> vertices;
3    List<Edge> edges;
4
5    Edge addEdge(Vertex from, Vertex to,
6                 int weight) {
7      Edge e = new Edge(from, to);
8  #ifdef Weighted
9      e.setWeight(weight);
10 #endif
11     ...
12     return e;
13   }
14
15   void run(Vertex v) {
16 #ifdef Number
17     numberVertex(v);
18 #endif
19 #if defined(Cycle) && defined(Directed)
20     detectCycles(v);
21 #endif
22   }
```

```
23 #ifdef Number
24   void numberVertex(Vertex v) {...}
25 #endif
26
27 #ifdef Cycle
28 #  ifdef Directed
29   void detectCycles(Vertex v) {...}
30 #  endif
31 #endif
32 }
```

```
33 class Vertex { ... }
```

```
34 class Edge {
35 #ifdef Weighted
36   int weight;
37   void setWeight(int weight) {...}
38 #endif
39   ...
40 }
```

Listing 3.5: Annotation-based implementation of the GraphLibrary product line

A comparison of Listing 3.1 and Listing 3.5 reveals that the FOP-based implementation looks considerably different from the cpp-based implementation. Both implementations are functionally equivalent, so these differences must be rooted in the chosen variability mechanism. Indeed, the literature has identified a number of aspects by which variability mechanisms can be distinguished, including *scattering,*

[6] https://www.gnu.org/software/make/manual/html_node/Conditionals.html

tangling, and the *granularity* of feature code [343, 89, 32, 158, 155]. In the following paragraphs, I explain these aspects in detail. Moreover, I outline how they affect the shape of feature code because the shape is important to the way in which code smells appear.

Scattering and tangling. Taking the feature *Weighted* as an example, we observe that the FOP-based implementation encapsulates the corresponding feature code in a single place, the feature module (see Listing 3.1 (b)). The same feature code is also present in the CPP-based implementation but it is *scattered* over multiple locations (see Lines 9, 38, and 39 in Listing 3.5). Moreover, feature code in the CPP-based implementation is *tangled* with the base code and with other feature code (e. g., on Lines 20 and 29). Scattering and tangling of feature code are typical for annotation-based variability mechanisms and are often seen as an obstacle to traceability and modular reasoning about the implementation of a feature [343, 89, 32, 155].

Interestingly, the composition-based implementation in Listing 3.1 exemplifies another kind of scattering: The implementation of the `Graph` class is scattered across all four feature modules in the listing. Hence, a programmer would have to inspect four files and mentally compose them to understand the `Graph` class in its entirety. Inheritance in OOP has a similar effect: In order to fully understand a subclass, the developer also has to take its superclasses into account. This reasoning lead to speculations that inheritance makes programs harder to understand [200] and should be taken into account as a software quality metric [51]. However, empirical studies of this matter are inconclusive, with some suggesting that deep inheritance hierarchies are detrimental to software quality [28, 46] and others suggesting they are not [356, 87] or only in certain cases [352]. Consequently, it is open to debate whether FOP's scattering of program elements (e. g., classes or methods) is problematic.

Code obfuscation. Apart from scattering and tangling, annotation-based variability mechanisms are criticized for *obfuscating* the source code of the underlying programming language (e. g., [343, 89, 213]). There are signs of obfuscation in Listing 3.5. Specifically, 14 out of 40 lines of code consist of preprocessor directives. Since these directives are interleaved with the source code, understanding the program may be difficult. Code obfuscation as well as scattering and tangling are the reasons why "`#ifdef` [is] considered harmful" by some researchers [343] and why others even speak of "`#ifdef` hell" [213].

Granularity. The *granularity* of variability is a further important difference between annotation- and composition-based mechanisms [158]. Composition-based variability mechanisms provide only coarse-grained variability. Specifically, introductions in FOP can add new classes, fields, and methods, and refinements can add statements at the beginning or end of a method. However, adding statements in the middle of a method or adding parameters to a method signature is impossible or requires cumbersome workarounds [156, 157]. The CPP, by contrast, allows individual statements and (parts of) expressions to be annotated, down to the level of single characters [158]. As a consequence, it handles fine-grained feature interactions on the implementation level, such as the *optional feature problem*, more gracefully than FOP [159, 155].

While beneficial in certain situations, the fine-grained variability offered by the CPP can lead to *undisciplined annotations* [206]. An annotation is called undisciplined if it does not align with a syntactical unit of the host programming language. For example, annotating an entire statement is disciplined but annotating a single opening brace is not. Not only do such annotations pose problems for refactoring and other automated analyses and transformations [2, 23, 35, 112, 115, 113, 114, 154, 157, 296, 345, 344, 369, 370], but several studies also suggest that they hinder program comprehension [238, 221, 240]. However, there is also contradicting evidence, suggesting that the lack of annotation discipline by itself is harmless [327]. Even though the discussion about harmfulness is still ongoing, refactorings have already been proposed that transform undisciplined into disciplined annotations [206, 240].

Undisciplined annotations can be seen as one of the first variability-aware code smells because they posses all of the important properties: They are a distinctive pattern in the source code that is directly tied to the (mis)use of a variability mechanism. Moreover, there is evidence that this pattern occurs in real-world software and that it may be harmful. One of the goals of this thesis is to research whether further patterns with similar properties exist. As such, the work on undisciplined annotations has served as an inspiration and motivation for this thesis.

To summarize, composition- and annotation-based variability mechanisms differ with regards to the separation of concerns, code obfuscation, and granularity. As I explain in more detail in Chapter 4, these differences profoundly affect how the code of the underlying host language is shaped and, consequently, how code smells appear.

3.5 Clone & Own Variant Development

Not all software product families are developed with the help of structured SPLE approaches such as FOSD. Reports from the industry show that companies frequently choose the *clone-and-own* strategy [91, 245, 79, 317, 349]. In a nutshell, clone-and-own means that a new development artifact is created by copying an existing, similar artifact and adapting the copy until the new requirements are met. The size of the cloned artifacts varies, ranging from a handful of functions and individual components to entire products [79]. In my thesis, I focus on the upper end of the range, that is, on cloning of entire products as way to create a software product family. Although cost-effective in the short term, clone-and-own causes maintenance problems in the long term, as I will outline in this section. As part of my thesis, I propose an approach to *migrate* clone-and-own product families into an SPL to alleviate those problems (see Chapter 6). In this section, I provide the background to my approach. In particular, I explain why and how companies apply clone-and-own, which problems this may induce, and how clone-and-own relates to code clone detection and refactoring.

3.5.1 Reasons for Clone & Own

The benefits of SPLE as a way to develop software product families have been known for several decades. This raises the question why companies continue to employ unstructured reuse via clone-and-own. The interview study by Dubinsky et al. reveals four main reasons [79]:

1. Efficiency. Clone-and-own is perceived as a simple but efficient reuse mechanism because development starts from already implemented and verified artifacts. Sometimes unawareness of other reuse strategies is the reason for this perception. In other cases, it is rooted in previous failed attempts at adopting systematic reuse.

2. Independence. In contrast to other reuse strategies, developers can freely change cloned artifacts. Neither do they have to worry about side effects on other products, nor do they have to synchronize release cycles with the remaining product family as it is the case with SPLE. These reasons make clone-and-own popular in the industry, but also in the open-source world, where it often takes the form of *forking* [270, 349, 43]. To fork is to start a new project by copying the version control repository of an existing project. Open-source developers fork for a variety of reasons, for example to reactivate an abandoned project, add missing functionality, remove unneeded functionality, or to customize the project for their own use case. Industrial developers also fork, but their reasons are more related to management aspects [80]. For example, companies fork to accommodate different release schedules and to reduce coordination overhead.

3. Short-term thinking. Companies using clone-and-own tend to focus on the success of individual products while, at the same time, postponing reuse issues until later. Therefore, they are unable or unwilling to invest in systematic reuse.

Sometimes the success of a new product is uncertain or the size of the future product family is unknown [183]. A comparison of the costs of different approaches to develop a program family reveals that clone-and-own is a sensible choice in such a situation. In Figure 3.3, I plotted the costs for single product development and for SPLE based on estimates reported in the literature (see Clements and Northrop [53], p. 226f. and Pohl et al. [295], p. 9f.). Similar estimates are unavailable for clone-and-own, which makes it difficult to determine exactly where its cost curve lies. In Figure 3.3, it was placed between single product development and SPLE based on the following reasoning: On the one hand, reuse through cloning is more efficient than single product development, where everything is developed from scratch. On the other hand, reuse through cloning is less efficient than SPLE, where reuse is structured and planned. Comparing all cost curves in Figure 3.3, we see that SPLE clearly pays off for a large number of variants. At the same time, its initial planning phase makes SPLE more expensive than the other options when only a few variants are developed. This planning phase requires an up-front investment that will never be recovered if some of the variants fail. Considering single system development and clone-and-own, we notice that neither one requires such an up-front investment. Additionally, we notice that the costs of clone-and-own rise more slowly than the costs of single system development as the number of variants increases. Hence, if the success or the number of future variants are uncertain, clone-and-own is the most attractive and cost-efficient option of all three.

4. Lack of governance. Systematic reuse requires a certain organizational structure and specific roles that are responsible for managing and measuring reuse. Conversely, the absence of such roles favors clone-and-own as the reuse approach. It is therefore not surprising that clone-and-own (forking, in particular) is common in the loosely organized open-source world [270, 349, 43]. However, companies, too,

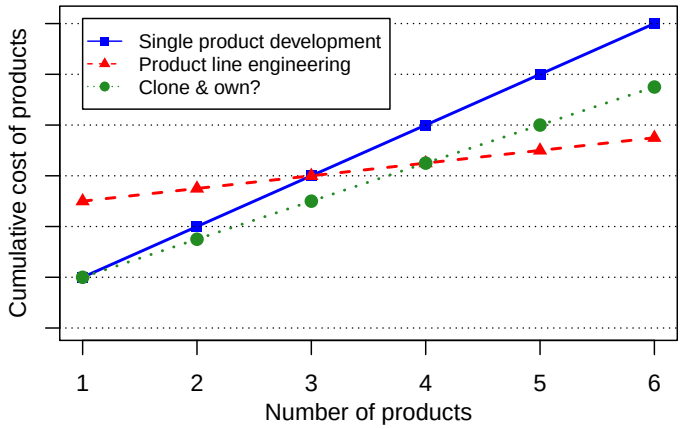


Figure 3.3: Comparison of product line engineering with single-product development and clone & own *(Adapted from Clements and Northrop [53], page 227)*

frequently lack the organizational structure for managing reuse and, thus, resort to clone-and-own [79]. Since such companies typically do not measure reuse, they cannot even quantify whether clone-and-own is economically sensible or whether a systematic reuse approach would suit them better.

3.5.2 Technical Realization

Clone-and-own is usually implemented with the help of the *branching* or *forking* capabilities of a *Version Control Systems (VCSs)* such as SUBVERSION, or GIT (cf. [350, 79, 317] as well as Apel et al. [13], Chapter 5). Branching and forking are conceptually similar with regards to clone-and-own but the technical details differ. For simplicity, I focus on branching in this section but I briefly return to forking at the end.

Feature branches. Before explaining why branching is problematic for variant development, it is helpful to understand how branching is used in single-product development. Branching in single-product development is used extensively to isolate concurrent changes from each other. I call this style of using branches the *feature-branch style,* and it is illustrated in Figure 3.4. All activity in this example is centered around the *main* branch (shown in light gray), which is where the stable version of the product is developed. Whenever a new feature is implemented or a bug is fixed, the developer branches off of the *main* branch and creates a so-called *feature branch*, such as the branch *bugfix* (shown in orange). A branch is essentially a copy. Hence, the developer working in the *bugfix* branch can experiment freely without accidentally breaking the stable product in *main* and without disturbing the development of feature *A* and *B,* which takes place in separate branches (shown in blue and red). Once the bugfix has stabilized, it is integrated into the stable product by *merging* the corresponding changes (commits `C8` and `C11`) back into the *main* branch. Afterwards, the *bugfix* branch is closed.

The merge step is critical because concurrent changes can cause *merge conflicts*. A merge conflict occurs if commits in concurrent branches affect the same code. For example, if the commits `C8` in *bugfix* and `C9` in *main* change the same line in the same

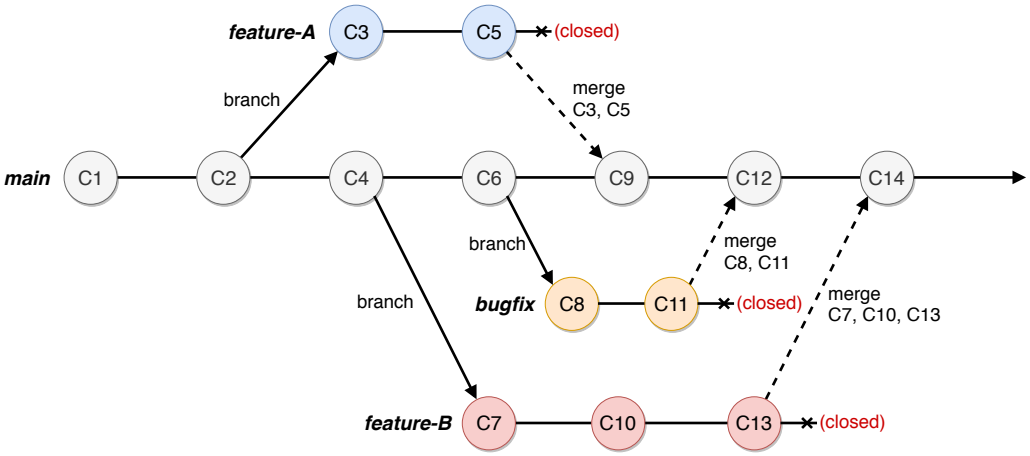


Figure 3.4: Branching and merging in single-product development

file, then the attempt to merge the bugfix back into *main* will cause a merge conflict [247]. Simple merge conflicts can be resolved automatically, but more complex conflicts must be resolved manually by a developer, which is time-consuming and error-prone. How to avoid such conflicts is still not well understood [201]. Therefore, the only safe advice is to organize work in such a way that changes in each branch are focused on distinct parts of the code base.

Advantages of the feature-branch style. Using branches in the style described above does not eliminate merge conflicts or other problems related to concurrent development but it helps avoid them. The following properties in particular are helpful.

1. Feature branches are created for a specific, narrow purpose. Keeping the purpose narrow ensures that a feature branch only contains commits related to its purpose. It is not necessary to manually *cherry-pick* which commits to merge into the target branch and which to leave out. As a consequence, both effort and the potential for errors remain low.

2. Feature branches are short-lived, which limits the amount of concurrent changes that can occur in the parent branch. Moreover, the developers in different branches work on a single product, and, assuming proper coordination, are aware of each other's activities. Due to these factors, changes are rarely merged into a code base that contains unknown modifications, that is, modifications that can cause merge conflicts or, worse, semantic conflicts from which bugs arise.

3. Finally, the rule to determine the merge target is simple: A feature branch is merged back into the branch from which it was created. Not only does this rule avoid the problems connected to merging changes into a unknown environment (see previous point), but it also minimizes the risk of missing a merge target.

These helpful properties are lost when branches are used for variant development. I explain this style of using branches next.

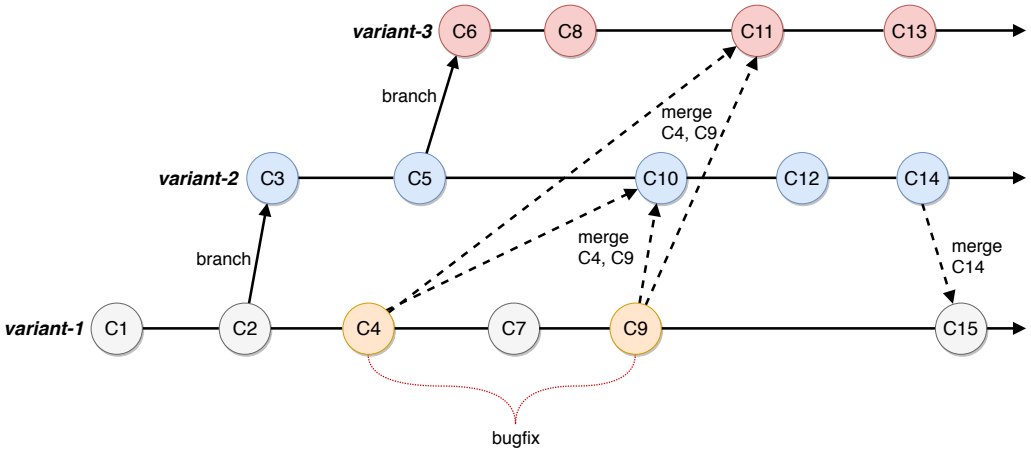


Figure 3.5: Branching and merging for variant development *(Adapted from Apel et al. [13] and Pfofe [292])*

Variant branches. In Figure 3.5, I illustrate the *variant-branch style,* which is how clone-and-own uses branches. The example encompasses three variants, the original product, *Variant 1*, and the derived products, *Variant 2* and *3*. Every variant is developed in its own branch, *variant-1*, *variant-2* (branched off at commit `C2`), and *variant-3* (branched off at commit `C5`). Each of those branches contains commits that are unique to that branch, such as `C7`, which only occurs in *variant-1*. These are the commits that implement variant-specific changes. For example, commit `C7` may implement a feature that is exclusive to *Variant 1*. In contrast to these variant-specific commits, other commits are merged to multiple variants. For example, the commits `C4` and `C9` fix a bug in an artifact that was cloned to all three variants. Thus, `C4` and `C9` are merged from *variant-1* to *variant-2* and *variant-3*. Commit `C14`, in turn, comprises an enhancement that was first implemented for *Variant 2* but is also needed in *Variant 1*. For this reason, `C14` was merged from *variant-2* to *variant-1*.

Disadvantages of the variant-branch style. The style depicted in Figure 3.5 works well for keeping variant-specific changes local but it causes problems for changes to reused (cloned) artifacts. The reason is that the helpful properties of the feature-branch style no longer hold. In particular:

1. Variant branches lack the narrow purpose of feature branches. Variant-specific adaptations, such as commit `C7`, are mixed with changes to cloned artifacts, such as the bugfix in commits `C4` and `C9`. Thus, the commits that implement a change to a cloned artifact must be cherry-picked before they can be propagated to other variants. This is tedious and error-prone.

2. Variant branches are long-lived. Over time, variant-specific adaptations accumulate and the implementations of cloned artifacts start to deviate [386]. The lack of coordination among the developers of different variants, which is typical for clone-and-own [79], contributes to these deviations. As a result, changes to cloned artifacts are increasingly difficult to merge because the source and the target of the merge are increasingly dissimilar. In extreme cases, changes

must be reimplemented manually, resulting in repetitive work and additional costs [79].

3. Finally, there is no simple rule to decide where to merge which changes and reuse is rarely documented when clone-and-own is used. Consequently, the developer performing a merge may fail to remember all the relevant merge targets [147, 79]. For instance, the developer implementing the bugfix in the example in Figure 3.5 might propagate the fix to *Variant 2* but forget *Variant 3*, leaving the variant vulnerable.

A final drawback of the variant-branch style is the difficulty to create variants that encompass new feature combinations (Apel et al. [13], p. 104). As an example, assume that branch *variant-1* implements features A and B, and branch *variant-2* implements features C and D. Based on these branches, a new variant should be created that comprises only features A and C, but not B or D. Creating this new variant through a simple merge is impossible because the implementations of A and B are tangled with each other and likewise, the implementations of C and D are tangled with each other. As a result, the corresponding feature code must be extracted first, which is a time-consuming, costly process [315].

Forking. Apart from relying on the branching capabilities of the version control systems, the variant-branch style can also be implemented with the help of *forking:* To create a new variant, the version control repository of an existing product is copied and all variant-specific adaptations are applied to the copy, called the *fork*. Hosting services, such as GitHub, allow developers to maintain a traceability link between the original repository and the fork [349]. Through this link, it is possible to propagate changes between repositories by way of a *pull request*. A pull request essentially comprises a list of commits that implement a particular change, along with a description of that change. Sending a pull request to the maintainers of another, linked repository is an invitation for them to integrate the change into their own repository. If they accept, all the respective commits are merged ("pulled") into their repository.

Forking is used both in the industry and in open-source to develop variants [79, 270]. Even though the technical details differ, using forks or (variant-style) branches for variant development is essentially the same approach: Variants are created by copying and freely adapting a code base, change propagation involves cherry-picking and merging commits, and whether the variants stay synchronized depends on the memory and good will of the maintainers. As a consequence, the problems for variant development are also the same.

3.5.3 Negative Effects of Clone & Own

The previous section already mentioned some technical problems, such as merge conflicts, that arise from clone-and-own but further problems have been documented in the literature [386, 147, 300, 79, 39, 80, 349]. I summarize these problems in this section.

Missed reuse opportunities. Studies of clone-and-own in open-source and the industry report several reasons why reuse opportunities are missed. First, if reuse is not valued, developers tend to implement new solutions right away without even trying to identify existing, reusable solutions [386, 79]. Second, even if the developers are willing to reuse, decentralization of information and the lack of reuse management cause problems [349, 386, 79, 80, 39]. In particular, developers using clone-and-own report having difficulties maintaining an overview of existing features and bugfixes, especially in program families comprising a large number of variants. Hence, reuse only happens if they happen to remember a reusable solution or are willing to actively search for one.

Unpropagated changes. When clone-and-own is used, changes to reused artifacts originate in one variant and then must be actively propagated if other variants should also benefit. Change propagation sometimes does not always take place, and the resons for that differ depending on who is responsible for initiating the propagation process: If the implementors of the original change are responsible, they face the problem that reuse is not documented. As a result, they sometimes forget to propagate a change to all the necessary target variants [147]. If, on the other hand, the maintainers of the target variants are responsible, they may not be aware of new changes from other variants, or are unable or unwilling to spend time and effort on integrating them [349]. The consequences are the same in both cases: Some variants do not receive new features or worse, are left vulnerable to bugs, even though fixes already exist.

Change propagation is costly. Developers from the open-source world report that integrating changes from other forks requires a large amount of developer effort [300, 349]. The sheer number of changes is one factor, but the need to review and possibly revise changes from other forks also plays a role. Companies relying on clone-and-own face similar problems but since they typically do not measure reuse, they cannot quantify the costs that arise from these problems [79].

Proliferation of incompatible solutions. Solutions that have not been designed with reuse in mind can be too specialized and inflexible to be reused for a similar problem [79]. Thus, highly specialized, incompatible solutions are implemented multiple times, wasting development effort and causing the code in different variants to diverge from each other. A case study report from Yoshimura et al. indicates that the extent of such divergencies quickly reaches a point where a switch from clone-and-own to SPLE becomes infeasible [386].

Unexpected costs. Even though clone-and-own is perceived as a simple reuse approach, it can be difficult in surprising ways. Identifying the ideal source artifact for cloning is sometimes hard, especially when there are many similar artifacts to choose from [79]. The following step, integrating the clone, is also sometimes more complex than anticipated. In severe cases, the attempt to clone-and-own is abandoned and the required artifact is rewritten instead.

Repetetive tasks. Finally, developers working with clone-and-own complain about an excessive number of repetitive tasks [79]. For example, if a bug is discovered, multiple clones have to be checked whether they also contain the bug. Once the

bug is fixed, the bug must be propagated or, if that is infeasible, reimplemented. Repetitive tasks are not limited to code but also involve other areas such as running tests or updating the documentation. Not only are these tasks tiresome, they also cost time and money, and the cost increases with each additional variant.

3.5.4 Summary

Clone-and-own as a reuse approach requires minimal planning and makes the creation of new variants quick and cheap. However, clone-and-own becomes expensive during maintenance and evolution when change propagation takes up a substantial share of the development effort. The root cause of these problems is that clone-and-own focuses on the differences between variants but neglects the fact that variants share a lot of commonalities (cf. Apel et al. [13], Chapter 5). Maintaining and evolving these commonalities efficiently is the strength of SPLE but the weakness of clone-and-own.

The larger a software product family grows, the more apparent the weaknesses of clone-and-own will become. At some point a *migration* to an SPL is desirable. In Chapter 6, I propose an approach to migrate a clone-and-own product family to an SPL. My approach was inspired by the insight that many of the problems that plague clone-and-own are well known from research on *code clones*. I outline the fundamentals on code clones next, thus providing the necessary background for Chapter 6.

3.6 Code Clones

Code clones are pieces of source code that occur in the same or similar form in multiple locations [312]. As mentioned in Section 2.1, the code smell community knows code clones under the name Duplicated Code (Fowler et al. [107], p. 76). Usually, code clones result from copying a piece of code from one location to another, optionally customizing the copy afterwards [152].

If there are just two pieces of code that are similar, they are called a *clone pair* [312]. If three or more pieces of code are involved, the more general terms *clone class* or *clone group* are used.

With these definitions in mind, I will devote the rest of this section to describing types of code clones and techniques to detect them. Afterwards, I summarize empirical studies of the negative effects of code clones on software development and describe approaches to deal with code clones.

More comprehensive overviews of code clone research are found elsewhere [181, 313, 312]. Specifically, Koschke explains why code is cloned and which consequences this has. Moreover, he covers the evolution of clones, ways to detect, manage and remove them, as well as approaches to present clone detection results (Koschke [181], Chapter 2). Roy et al., in turn, focus on code clone detection. They present a classification of different types of code clones, give an overview of the available detection techniques and tools, and evaluate the suitability of those tools in different scenarios [313, 312]. For a thorough treatment of code clones specifically in SPLs, see Schulze's thesis [323].

3.6.1 Types of Code Clones

Code clones differ in their degree of textual similarity, which impacts the refactorings required to consolidate them. Roy et al. categorize clones into four types, namely, *Type-1* to *Type-4* clones [312]. The category number aligns with the degree of similarity between the original and the cloned fragments, with Type-1 clones having the highest similarity and Type-4 clones the lowest.

In Listing 3.6, I show examples of Type-1 to Type-3 clones, which are the types that are most relevant for my migration approach. The original code fragment is depicted in Listing 3.6 (a) and the cloned fragment is shown in (b). Type-1 clones, highlighted in green, are exact copies of each other, except for changes in whitespace or comments. Type-2 clones subsume all Type-1 clones, but also allow for renaming (for instance, function or type names may differ). In Listing 3.6, the green and yellow areas together form a Type-2 clone. Type-3 clones allow further modifications beyond simple renaming. Specifically, they allow statements to be added, deleted, or changed with respect to the original code. There is a special name for a Type-3 clone that *only* encompasses added, deleted, or changed statements but no renames: It is called a *gapped clone.* Together, the green, yellow, and red areas in Listing 3.6 form such a clone. Finally, there are also Type-4 clones, but they are not depicted in Listing 3.6. Type-4 clones, also called *functional clones,* implement similar functionality but share little or no textual similarity [312]. Instead of resulting from copy and paste programming, they often constitute independently implemented solutions for the same or similar problems.

```
1 void calc(int a) {                    Type-2
2   if (a >= n) {                       Type-1
3     c = a + 1; // Comment 1
4   } else
5     c = a + 1; // Comment 2
6 }
7
```

(a) Original code

```
1 void recalc(int a) {                  Type-2
2   if (a >= n) {                       Type-1
3     c=a+1;
4     d=a+1;                            Type-3
5   } else
6     c=a+1; // Comment 2
7 }
```

(b) Code clone

Examples of code clones of Type-1 to Type-3. The red areas form Type-1 clones. Red and yellow together form a Type-2 clone. Red, yellow and green together form a Type-3 clone.

Listing 3.6: Examples of different types of code clones

In clone-and-own product families, all of these clone types are possible. Assuming there are two products in the family, *product A* and *product B,* then all program elements that were only copied but never modified will be Type-1 clones. If identifiers (e. g., variables or type names) or constants (e. g., numbers or string literals) have been changed, there will also be Type-2 clones. Small additions, deletions, and modifications, such as some added statements inside a function, will lead to Type-3 clones. Type-4, in turn, will be signs of functionality that was implemented independently in both products. According to the literature, such duplicate implementations can also arise from clone-and-own, especially if the product family is large or if there is little coordination between the teams working on different variants [386, 79, 349]. Unfortunately, Type-4 clones are almost impossible to detect automatically [312]

(more on clone detection in the next section). Finally, there may be some code is *not* cloned. Such code may encompass a customization or a feature that only exists in a single variant. It could also be a bugfix that was never propagated.

3.6.2 Code Clone Detection

Manual clone detection is tedious and error-prone, especially in cloned product families that contain large amounts of code clones. Therefore, my migration approach relies on automatic code clone detection.

Classification. Many clone detection techniques have been proposed in the literature. Roy et al. classify them into *text-based, token-based, tree-based,* and *metrics-based* techniques [312]. The differences lie in efficiency, programming-language independence, and the types of clones each technique can detect. For example, text-based clone detectors are highly efficient and handle *gapped clones* well but they tend to miss clones with minor variations, such as formatting or names [312]. Tree-based detectors, by contrast, are more robust to such to differences but since they depend on a parser, they are highly language-dependent and computationally expensive [312].

The migration approach I propose in this thesis (see Chapter 6) relies on a token-based detector, which falls somewhere in the middle of the design spectrum: Compared to text-based clone detectors, it handles minor variations better. At the same time, it is more efficient and less language-dependent than tree- or metrics-based detectors [312].

Token-based. Tokens are the smallest syntactical units of a programming language, such as keywords, identifiers, and braces. Token-based clone detectors (e. g., [24, 25, 149, 203]) convert the input text into a token sequence and detect clones by searching for runs of identical tokens [312]. Some token-based detectors can deal with renaming and are thus able to find Type-2 clones.

The quality of the results of token-based clone detectors critically depends on the *clone length,* which is the number of tokens that must be identical so that a clone pair is reported. If this number is too low, too many irrelevant clones are reported and precision suffers. If, on the other hand, it is too high, too many relevant clones are *not* reported and recall declines.

In a migration context, the biggest disadvantage of token-based clone detectors is that they do not respect the boundaries of syntactical units of the programming language. For example, they report clones that start at the end of one function and end at the beginning of the following function [245]. Such results are not meaningful as they actually contain two shorter clones that just happen to appear next to each other in the source code.

Summary. Clone detectors generally support Type-1 and 2 clones well, but few tools can handle Type-3 or 4 clones [312]. The focus on low-level similarities and the failure to respect syntactical boundaries (in the case of text- and token-based detectors) is a challenge in a migration context because features are high-level and typically comprise program elements in multiple locations. Moreover, the focus

on similarities means that systematic variations between similar but not identical features may be missed. Thus, clone detection can support migration but it needs to be complemented with other techniques and must be guided by human expertise.

3.6.3 Effects of Cloning

Fowler et al. list DUPLICATED CODE as "number one in [their] stink parade" of code smells (Fowler et al. [107], p. 76). A lot of research on the effects of cloning agrees with that assessment. In particular, code clones increase maintenance effort because cloned code is changed more frequently than other code [217, 216]. Moreover, the failure to propagate changes to all members of a clone class code causes problems, such as inconsistent bugfixes [203, 147]. Increased fault-proneness is further negative consequence of cloning, especially for long clones comprising several hundred lines of code [259].

Other findings, however, indicate that cloning is sometimes harmless and can even be a reasonable design choice [59, 177, 152, 197, 299, 371]. For example, Cordy, as well as, Kapser and Godfrey point out that extending an existing solution for reuse bears the risk of introducing faults [59, 152]. In finance, among other domains, the cost of such faults can be so high that developers rather resort to clone-and-own.

The aforementioned work has studied the effects of cloning in single software systems but we must keep in mind that clone-and-own as a way to develop program families produces clones on a massive scale. As discussed in Section 3.5, managing clones at such a scale is a major challenge in terms of coordination and maintenance effort. Since the prerequisite organizational structures are usually missing when clone-and-own is used, the challenge is often too great.

3.6.4 Dealing with Code Clones

Once a clone detector has analyzed a software system, the question is what to do with the results. A first possible step is to assess the cloning situation with the help of visualizations. Such visualizations provide an abstract overview that helps development teams gauge the extent of cloning and identify hotspots that require particular attention. Treemaps, scatter plots, and heat maps, among others, have proven useful in this regard [304, 212, 134].

After assessing the cloning problem, developers basically have two options of dealing with the detected clones. The first option is to *consolidate code clones* through refactoring. This option is frequently advocated and it is also the option that I pursue in this theses. However, consolidation is not always feasible, for example because clones do not align with syntactic units [24] or because limitations of the programming language render certain refactorings impossible [263]. In such cases, developers have a second option, *clone management*. The core idea is not to remove the clones but to reduce their negative effects. To this end, editor-based solutions have been proposed, which help developers change clones consistently [253, 361, 77, 138, 78]. Some clone management approaches also identify the creation of new clones by actively monitoring copy and paste activities [138] or by taking evolutionary information from the version control repository into account [268].

In my thesis, I have not explored the applicability of clone management techniques to migrate cloned product families to an SPL. Instead, I focus on code clone consolidation, which I discuss in more detail next. More comprehensive treatments on approaches to deal with code clones can be found in the overviews by Koschke [181] and Roy et al. [312] as well as the dissertation by Schulze (Schulze [323], Chapter 2).

Code clone consolidation. Early work by Baxter et al. relies on function-like macros to consolidate clones [36]. Specifically, their clone detector generates a CPP macro for each clone class and replaces the original locations of the cloned code with an invocation of the macro.

In languages such as C, function-like macros are sometimes the only available solution, but other languages provide a similar, but safer solution in the form of generic programming (e. g., *templates* in C++ or *generics* in JAVA). By using placeholders instead of concrete types, generic programming is effective at unifying two algorithms or data structures that only differ in the types involved [29].

Design patterns [108, 165] are another, more heavy-weight solution for clone consolidation. Balazinska et al. present a redesign process based on the STRATEGY and the TEMPLATE design patterns that is especially suitable for consolidating Type-2 and 3 clones [27, 26]. The core idea is to refactor the commonalities of the clones into one class and to refactor the variations into one ore more helper classes.

A large body of work proposes object-oriented refactorings as a means to consolidate clones [81, 135, 132, 136, 136, 133, 148, 199, 263, 326, 385, 392]. For my migration approach (see Chapter 6), the PULL UP family of refactorings (Fowler et al. [107], p. 320ff), comprising PULL UP METHOD, PULL UP CONSTRUCTOR BODY and PULL UP FIELD, is the most relevant. Many authors propose PULL UP refactorings to consolidate clones in a single software system [81, 135, 132, 133, 385, 136, 326, 199, 392]. These refactorings apply when several subclasses with a common superclass define a method, constructor or field in the same way (i. e., as a Type-1 clone). To consolidate such definitions, one definition is copied to the superclass and the redundant definitions are deleted from the subclasses.

As Ducasse et al. already pointed out, the context of clones determines which refactorings can be used [81]. For instance, PULL UP METHOD is only applicable if the cloned methods reside in classes with a common superclass. Sometimes, such a superclass can be introduced, in which case EXTRACT SUPERCLASS or EXTRACT INTERMEDIATE SUPERCLASS are possible solutions [385, 136, 199, 392]. If not, other refactorings may be applicable, such as MOVE METHOD [136, 326, 199], EXTRACT CLASS [136, 199], and EXTRACT UTILITY CLASS [263, 392]. If none of these refactorings work, for example, because the clones only cover parts of a method, *preliminary refactorings* may be of help [81, 135, 148]. Preliminary refactorings prepare the code base for code clone consolidation. To this end, several refactorings have been proposed, including FORM TEMPLATE METHOD, PARAMETERIZE METHOD, and, most commonly, EXTRACT METHOD [81, 135, 132, 133, 136, 148, 199, 326, 392].

In addition to the above mentioned OOP refactorings, AOP refactorings have also been considered for clone consolidation. In particular, EXTRACT FEATURE INTO

Aspect, Extract Fragment Into Advice, and Move Method From Class to Inter-Type have been discussed [263, 326].

```
1  class Loan {
2    double interestRate;
3    abstract double getMonthlyInterest();
4  }
```

```
5  class EuroLoan extends Loan {
6    double principal;
7    @Override
8    double getMonthlyInterest() {
9      return principal
10             * interestRate / 12;
11   }
12 }
```

```
13 class ForeignCurrencyLoan extends Loan {
14   double foreignPrincipal;
15   double exchangeRate;
16     @Override
17     double getMonthlyInterest() {
18       double principal = foreignPrincipal
19                          * exchangeRate;
20       return principal
21              * interestRate / 12;
22   }
23 }
```

(a) Before refactoring

The subclasses of `Loan` implement `getMonthlyInterest()` similarly (highlighted in green) but not identically (highlighted in red).

```
1  class Loan {
2    double interestRate;
3    double getMonthlyInterest() {
4      return getEuroPrincipal()
5             * interestRate / 12;
6    }
7    abstract double getEuroPrincipal();
8  }
```

```
9  class EuroLoan extends Loan {
10   double principal;
11   @Override
12   double getEuroPrincipal() {
13     return principal;
14   }
15 }
```

```
15 class ForeignCurrencyLoan extends Loan {
16   double foreignPrincipal;
17   double exchangeRate;
18   @Override
19   double getEuroPrincipal() {
20     return foreignPrincipal * exchangeRate;
21   }
22 }
```

(b) After refactoring

Method `getMonthlyInterest()` is now defined only once (see green highlights), and the variations are encapsulated in `getEuroPrincipal()` (see red highlights).

Listing 3.7: A Form Template Method refactoring to consolidate code clones

Clone consolidation example. In Listing 3.7, I illustrate how a Form Template Method refactoring is used to consolidate Type-3 clones. The original code, depicted in Listing 3.7 (a), involves the superclass `Loan` and its subclasses `EuroLoan` and `ForeignCurrencyLoan`. The first subclass implements loans in the European currency; the second one is for loans in a foreign currency. Both subclasses implement the method `getMonthlyInterest()`, whose task is to calculate the monthly interest payment in Euros. The implementations are mostly identical (see green highlights), except that `ForeignCurrencyLoan` encompasses some additional code. This additional code converts the loan's principal (i. e., the amount of money that the borrower still owes the lender) from the foreign currency to Euros (see red high-

lights). As such, the methods are Type-3 clones, which means that a simple Pull Up Method refactoring is infeasible. However, a related refactoring, Form Template Method, is applicable.

The code after applying Form Template Method is shown in Listing 3.7 (b). Method `getMonthlyInterest()` has been turned into a single template method in the superclass (see Lines 3–6 in Listing 3.7 (b)). This template comprises on the one hand, the fundamental interest calculation (see green highlights) but on the other hand, also accommodates for the optional currency conversion (see red highlights). Specifically, the principal is no longer accessed directly but through an abstract helper method (see Lines 4 and 7 in Listing 3.7 (b)), which both subclasses implement differently (see Lines 11–14 and Lines 18–21). In summary, Form Template Method replaces cloned methods in two or more subclasses with a single template method in a superclass. This template encompasses all the code that was identical in the original methods and provides variations points to accommodate the differences.

3.7 Summary

In this chapter, I have explained the fundamentals of *software product families* and two approaches how such product families are developed. The first approach is build a *highly configurable software system,* a system that can be configured so that multiple, similar software products can be derived. Depending on which development process is used, a highly configurable software systems is also sometimes referred to as a *Software Product Line (SPL).* The products encompassed by a highly configurable system exhibit variations but also share commonalities. On the domain level, these variations and commonalities are referred to in terms of *features.* By organizing the features of a highly configurable software system in a *feature model (FM)*, it is possible to express the variability of the whole product family. On the implementation level, there are two kinds of mechanisms to implement variable features, namely annotation-based and composition-based *variability mechanisms.* Both kinds of mechanisms have their own advantages and disadvantages, and affect the structure of the source code in fundamentally different ways. As I will show in Chapter 4 of my thesis, these differences have an effect on the appearance of code smells in highly configurable software systems.

The second part of this background chapter was devoted to the other approach to create software product families, the *clone-and-own* approach. Creating software product variants with clone-and-own is easy, but maintaining and evolving them is not. As explained in this chapter, these problems are rooted in the massive amount of *code clones* that clone-and-own creates. To lay the groundwork for Chapter 6, in which I present a concept to *migrate* a family of cloned product variants to an SPL, this chapter also provided the background on techniques to detect and remove code clones.

4. Variability-Aware Code Smells

This chapter is based on and shares material with the VaMoS'15 paper "Code Smells Revisited: A Variability Perspective" [96] and the SCAM'15 paper "When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells" [97].

In the previous chapters, I described code smells, which are patterns in the source code that result from flawed design decisions. Moreover, I explained how variability in highly configurable systems is modeled and implemented. This chapter brings both topics together by introducing code smells specifically for highly configurable systems. I call these smells *variability-aware code smells.*

In Section 3.4.3, I discussed how annotation- and composition-based variability mechanisms differ with respect to many properties, such as separation of concerns, the granularity of variability, and the tendency to obfuscate the source code. In particular, I showed how each mechanism shapes the source code of the underlying host language in a different way. One thing that all variability mechanisms have in common, though, is that variability is implemented explicitly and thus is part of the code base. This adds another dimension of complexity to the design of a highly configurable software system: Apart from designing the structure of program elements (e. g., classes, and methods), developers also have to design the structure of variability. Variability-aware code smells are based on the observation that some variability-related design decisions are better, and some are worse. Unfortunately, established code smells do not cover the additional dimension of variability and are therefore useless for distinguishing the good from the bad decisions. The reason is that established code smells address issues in the structure of the host language, but are oblivious to the effect of variability. The main proposition I make in this chapter is that it is necessary to take variability into account as a first-class concept for variability-related code smells and their detection. Only then can we extend the established foundations of code smells to the domain of highly configurable software systems. This is of special importance because highly configurable software systems are typically large, long-lived systems. Any problems regarding program comprehension, maintainability or evolvability will only worsen over time.

In the first sections of this chapter, I revisit code smells in the light of variability. To this end, I extend existing, object-oriented code smells with the notion of variability, thereby creating an initial catalog of variability-aware code smells. This catalog covers both annotation- and composition-based variability mechanisms, using the C preprocessor and *Feature-Oriented Programming (FOP)* as representatives.

In the later sections of this chapter, I propose a metrics-based technique to automatically detect a selection of these smells. The technique has been implemented in a tool called Skunk and evaluated on five mature, widely used open-source systems. This evaluation serves two purposes. First, it demonstrates that the detection of variability-aware code smells is feasible. Second, it helps us understand how frequently such smells occur in practice and how instances of these smells look. In particular, I make the following contributions:

- A catalog of six code smells that take variability into account. For each smell, I discuss possible negative effects on program comprehension, maintainability, and evolvability. Moreover, I present examples how annotation- and composition-based variability mechanisms, respectively, affect the shape of these smells.
- An initial validation of my code smell catalog based on a survey with 15 researchers in the *Software Product Line (SPL)* field. The results reveal that most participants (a) observed these smells in real-world systems and (b) acknowledge that these smells may hinder maintenance and evolution.
- A set of metrics that capture different aspects of implementing variability with the help of C preprocessor directives and a concept for combining these metrics to detect variability-aware code smells.
- An implementation of this concept in the variability-aware code smell detection tool Skunk.[1] Among other capabilities, Skunk allows for the parametrization of thresholds and weighting factors for individual metrics.
- An empirical study of the variability-aware code smell Annotation Bundle. To this end, I analyzed five software systems with up to 285 KLOC. Besides a quantitative evaluation, 100 instances of this code smell have been reviewed manually to (a) assess the effectiveness of the detection method and (b) provide qualitative insights into how the Annotation Bundle manifests itself in source code.

4.1 Derivation Methodology

Before presenting my catalog of variability-aware code smells, I first explain how I derived them from established, object-oriented code smells. The basis for my proposed smells are the code smells described by Fowler et al. [107]. Specifically, I have considered how variability constructs, such as `#ifdefs`, can affect the language elements of the smell description, and how this will affect the code shape. For instance,

[1] https://github.com/wfenske/Skunk

the object-oriented code smell Long Method describes a method with too many statements, indicating that the method is too complex to be understood easily. Applying my methodology to this smell, with a focus on annotation-based variability, my question was: "What will a Long Method look like if some (or many) of the statements are guarded by `#ifdefs`?" This methodology works straightforward for many object-oriented smells besides Long Method (e. g., Duplicated Code, Switch Statements). However, it does not work for all smells. One counterexample is Primitive Obsession, which criticizes the use of primitive data types (e. g., `char`, `int`) when a class would be more appropriate. I found no way in which variability could affect this smell or a variety of other smells.

In the following catalog, I focus on six object-oriented smells, most of which have been shown to occur frequently in practice. For all six smells, I apply my methodology and distinguish between the two variability mechanisms cpp and FOP in case that their interactions with language elements matters. As I will show, there can be considerable differences. I discuss those differences in more detail when I compare my variability-aware smells Annotation Bundle and Long Refinement Chain.

4.2 A Catalog of Variability-Aware Code Smells

Next, I present six variability-aware code smells that I have derived using the method just described. For each smell, I start with a summary of the original object-oriented code smell, followed by a description of the derived smell. I further state which variability mechanisms (annotation-based, composition-based or both) the smell applies to. I then present an illustrative example and finally discuss potential problems for program comprehension, maintenance, and evolution that are caused by the respective smell. For my discussion, *maintenance* comprises bug fixes, quality improvements and other minor changes, whereas *evolution* means adding new functionality or making major modifications.

4.2.1 Inter-Feature Code Clones

Derived from: Duplicated Code (Fowler et al. [107], p. 76)

Copying an existing piece of code and pasting it somewhere else leads to *code clones*. For reasons I explained in Section 3.6, code clones (a. k. a. the smell Duplicated Code) are considered a code smell in non-configurable software systems.

Variability-aware description. There are two forms of code duplication in highly configurable software systems. First, code may be duplicated within a feature. In this case, the resulting clones are unaffected by variability and hence are associated with the same problems as code clones in non-configurable software systems (see Section 3.6.3). The second case, when there are two or more features that contain similar code, is more interesting. I call this smell Inter-Feature Code Clones. The smell can originate from intentional cloning but can also arise unintentionally, for instance because developers working on different variants or features fail to coordinate their work sufficiently. As discussed in Section 3.5, unintentional cloning can quickly reach alarming levels and thus cause severe problems for maintenance and evolution.

Applies to: Annotation-based and composition-based mechanisms

Example. In Listing 4.1, I show a composition-based example of Inter-Feature Code Clones, taken from the Graph Product Line (GPL).[2] The GPL is a product line of standard graph algorithms [214]. In the GPL code example, features *BFS* (breadth-first search) and *DFS* (depth-first search) contain an exact clone of the method `GraphSearch()`. This is surprising at first, as the two search algorithms are very different. A closer look at the rest of the implementation (not shown here) revealed that most of the work is performed by a helper method, which the features *BFS* and *DFS* implement differently. Thus, the implementation is, in fact, correct.

```
1  public class Graph {                Feature BFS
2    public void GraphSearch(Workspace w) {
3      VertexIter itr = getVertices();
4      /* more source code... */
5      for (itr=getVertices(); itr.hasNext();) {
6        Vertex v = itr.next();
7        if (!v.visited) {
8          w.nextRegionAction(v);
9          v.nodeSearch(w);
10       }
11     }
12   }
13 }
```

```
1  public class Graph {                Feature DFS
2    public void GraphSearch(Workspace w) {
3      VertexIter itr = getVertices();
4      /* more duplication... */
5      for (itr=getVertices(); itr.hasNext();) {
6        Vertex v = itr.next();
7        if (!v.visited) {
8          w.nextRegionAction(v);
9          v.nodeSearch(w);
10       }
11     }
12   }
13 }
```

Listing 4.1: Inter-Feature Code Clones in the GPL

Problems. I argue that Inter-Feature Code Clones are an even bigger obstacle than Duplicated Code in non-configurable software. First, the aforementioned unawareness of clone instances in other features increases the likelihood of inconsistent changes. Secondly, the variability in an SPL increases the complexity as the features containing the clones may be combined with a multitude of other features. Consequently, when a bug is fixed or other modifications are performed, the consistent propagation of the changes to other clones may be unnecessary or even lead to semantic errors. Hence, the developer has to verify for each feature in every valid configuration whether the change is both syntactically *and* semantically correct.

4.2.2 Annotation Bundle

Derived from: Long Method (Fowler et al. [107], p. 76f.)

The longer a method, the more difficult it is to understand. Even though this is true for many methods comprising hundreds upon hundreds of statements, the "long" in Long Method does not necessarily refer to the number of statements. As Fowler et al. emphasize, the criterion for a Long Method is "not [...] length but the semantic distance between what the method does and how it does it." (Fowler et al. [107], p. 77). Thus, methods with only a few densely coded statements can be "long" whereas others, which encompass many very simple statements, are still "short".

[2] http://spl2go.cs.ovgu.de/projects/49

Variability-aware description. Like a Long Method, an Annotation Bundle does not necessarily consist of many statements. Instead, it consists of many variable parts. A large number of features controls which of these parts are included or excluded. On the code level, this results in many (groups of) statements that are annotated. Several different annotations are involved, maybe even nested.

Applies to: Annotation-based mechanisms

Example. In Listing 4.2, I show a function implemented in C, which is heavily annotated with cpp directives. It is taken from the open source database management system MySQL, version 5.6.17.[3] Although the function is not long in terms of C statements, there are five different cpp macros that control which of these statements make up a concrete implementation. Moreover, these macros are nested and sometimes negated. For instance, the statement on Line 23 is controlled by three different macros, two of which must be undefined for the statement to be included. Altogether, only a few statements of the whole function are base code that is compiled into every possible program (e. g., the `if` condition on Line 5 and the

[3] http://dev.mysql.com/downloads/file.php?id=451519

```
1  sig_handler process_alarm(int sig __attribute__((unused)))
2  {
3    sigset_t old_mask;
4
5    if (thd_lib_detected == THD_LIB_LT && !pthread_equal(pthread_self(),
         alarm_thread)) {
6  #if defined(MAIN) && !defined(__bsdi__)
7      printf("thread_alarm in process_alarm\n");
8      fflush(stdout);
9  #endif
10 #ifdef SIGNAL_HANDLER_RESET_ON_DELIVERY
11     my_sigset(thr_client_alarm, process_alarm);
12 #endif
13     return;
14   }
15
16 #ifndef USE_ALARM_THREAD
17   pthread_sigmask(SIG_SETMASK, &full_signal_set, &old_mask);
18   mysql_mutex_lock(&LOCK_alarm);
19 #endif
20   process_alarm_part2(sig);
21 #ifndef USE_ALARM_THREAD
22 #if !defined(USE_ONE_SIGNAL_HAND) && defined(SIGNAL_HANDLER_RESET_ON_DELIVERY)
23   my_sigset(THR_SERVER_ALARM, process_alarm);
24 #endif
25   mysql_mutex_unlock(&LOCK_alarm);
26   pthread_sigmask(SIG_SETMASK, &old_mask, NULL);
27 #endif
28   return;
29 }
```

Listing 4.2: Annotation Bundle; taken from MySQL, version 5.6.17, file `mysys/thr_alarm.c`.

function call on Line 20). However, the majority of the lines are either feature code or preprocessor directives (e. g., the block on Lines 21–27).

Problems. I argue that an Annotation Bundle is difficult to understand for a certain configuration or in its entirety. Having many variable parts in the method body obscures the view on the core functionality. Moreover, each annotation requires additional knowledge of the macros that are involved. Hence, to comprehend an Annotation Bundle, a developer has to work with many different abstractions on both the programming language level and the variability level.

Maintenance and evolution tasks are also hampered by heavily annotated methods. For instance, locating a bug is difficult if the exact configuration that exhibits the defect is not known. Moreover, developers have to take special care when changing heavily annotated code. Otherwise they might break the presence conditions of existing statements or introduce dangling references due to particular configurations they failed to consider.

4.2.3 Long Refinement Chain

Derived from: Long Method (Fowler et al. [107], p. 76f.)

Variability-aware description. The smell Long Refinement Chain is the composition-based counterpart of the Annotation Bundle smell. As such, it denotes a method with many variable parts due to feature refinement.

Applies to: Composition-based mechanisms

Example. In Listing 4.1, I show a Long Refinement Chain from GUIDSL [31]. GUIDSL is a product line configuration tool implemented in Java with FOP. In my code example, I show the `process()` method of class `Main` and all of its refinements. The method is introduced as an empty stub by feature *dmain* and subsequently refined by five other features *(fillgs, propgs, formgs, clauselist, modelopts).* In contrast to this heavily refined method, the average refinement depth in GUIDSL is lower than one. In other words, most methods are never refined at all. Each refinement of `process()` contains between three to nine additional lines of code. As such, each refinement contributes considerably to the overall method. Moreover, most of these refinements can occur in different combinations, depending on the feature selection.

Problems. In contrast to an Annotation Bundle, with a Long Refinement Chain, feature-specific parts of a method are encapsulated in refinements. Even though encapsulation is a favorable property, I argue that excessive refinement is problematic. My argument follows Chidamber's and Kemerer's reasoning that high values of the Depth of Inheritance Tree (DIT) [51] metric indicate complex programs. First, as with Annotation Bundles, it is hard to understand the effect of particular combinations of refinements for a concrete configuration. Secondly, when modifying an often-refined method or adding a new refinement, the developer must be aware of all existing refinements (and their combinations) and possible side effects of changing or adding code. Hence, I contend that methods with a Long Refinement Chain are harder to understand, maintain and evolve than methods with few refinements.

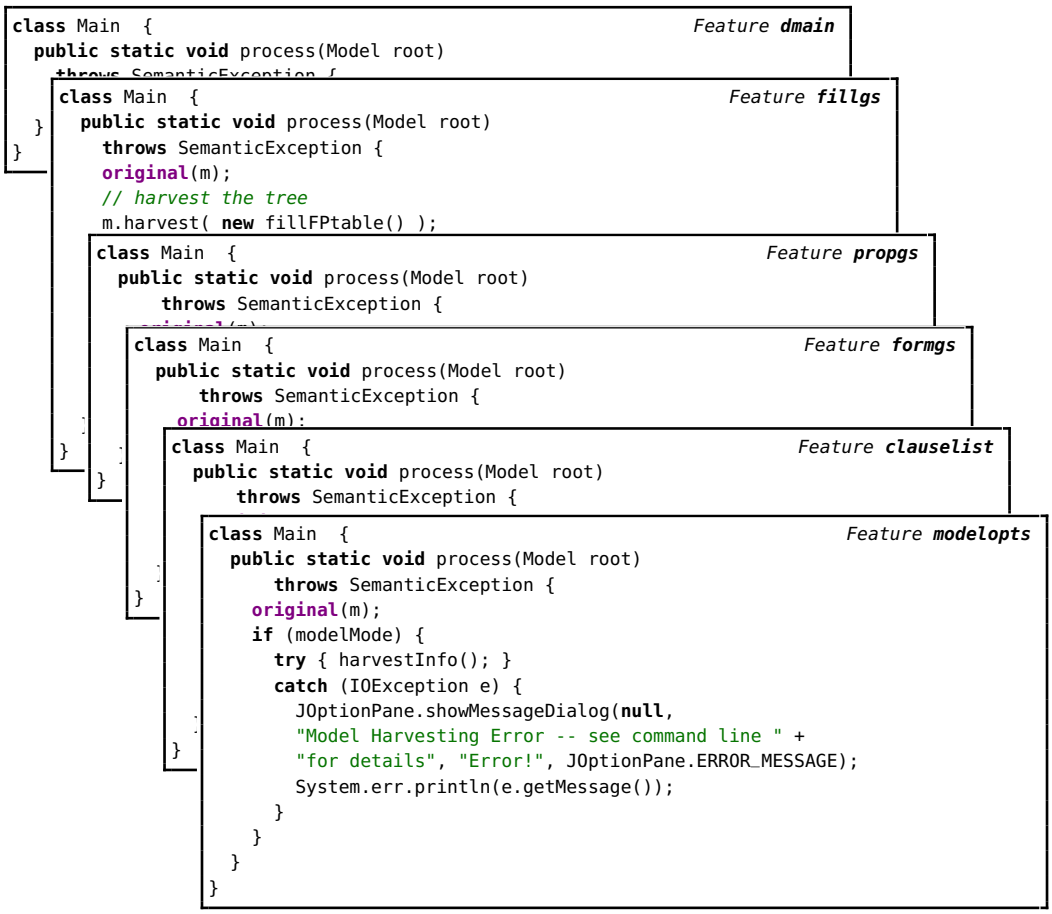


Figure 4.1: Long Refinement Chain in GUIDSL

Interestingly, even though I have derived both, Annotation Bundle and Long Refinement Chain, from the same object-oriented smell (Long Method), they manifest themselves very differently on the code level. This illustrates how much the chosen variability mechanism affects the shape of an object-oriented code smell when it is transferred to an SPL context.

4.2.4 Latently Unused Parameter

Derived from: Long Parameter List & Speculative Generality (Fowler et al. [107], p. 78, 83f.)

A method with many parameters hampers program comprehension as each parameter increases the cognitive burden of the caller. Furthermore, a Long Parameter List impedes evolution as it is frequently changed when additional data is needed (Fowler et al. [107], p. 78).

Speculative Generality, in turn, describes functionality that was added in anticipation of future evolution but is actually never used. Such functionality increases the complexity of the code without any immediate benefit (Fowler et al. [107], p. 83.). Speculative Generality can take several forms; *unused parameters* are one of them.

Variability-aware description. In an SPL context, parameter lists are also subject to variability. Sometimes, a parameter of a method is optional, that is, it is only needed by a certain feature, but is unnecessary for others. One solution to deal with an optional parameter would be a variable method signature, that is, a method signature that only includes the parameter if the corresponding feature is selected [311]. However, if the method signature is variable, the call sites also have to be variable. In particular, every call site has to take one form (with the parameter) or another (without the parameter). Effectively, variable method signatures cause variability to spread throughout the code base, which is undesirable. Rosenmüller et al. explain that another solution is to forward-declare the optional parameter upon introduction of the method [311], even if it is only used further down in the refinement chain. The disadvantage of forward-declaration, however, is that for all features higher up in the refinement chain, the forward-declared parameter will be unused. This, in turn, is a form of Speculative Generality.

Applies to: Annotation-based and composition-based mechanisms

Example. In Listing 4.3, I show a Latently Unused Parameter in code using annotation-based variability. The second parameter of the `push()` method, a `Transaction` object named `txn`, is only used if the feature *SYNC* is active (see Lines 4, 5, and 11). If, by contrast, *SYNC* is inactive, only Lines 7, 9, and 13 are compiled, which make no use of the `txn` parameter.

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

Listing 4.3: Latently Unused Parameter using annotations *(Listing adapted from Schulze [323])*

In FOP, where feature code is separated into distinct modules, the smell Latently Unused Parameter looks different. The parameter appears perfectly normal in some modules, but completely unused in others. In Listing 4.4, I show an example from the code base of the GPL. Here, feature *WeightedOnlyVertices* extends class `Graph` with method `addAnEdge()` (see Lines 2–4). The third parameter of this method is an integer, `weight`. This is reasonable, as the feature provides support for weighted graphs. Indeed, `weight` is used by helper method `addEdge()` (see Lines 3 and 6–10). Feature *DirectedOnlyVertices* also provides a method `addAnEdge()`. In order to be compatible with feature *WeightedOnlyVertices*, it has the same signature.

However, in this second method definition, the parameter `weight` is confusing and, in fact, unused.

```
1  public class Graph {                                  Feature WeightedOnlyVertices
2    public void addAnEdge(Vertex start, Vertex end, int weight) {
3      addEdge(start, end, weight);
4    }
5
6    public void addEdge(Vertex start,  Vertex end, int weight) {
7      addEdge(start, end);
8      start.addWeight(weight);
9      /* More source code ... */
10   }
11   /* More source code ... */
12 }
```

```
13 public class Graph {                                  Feature DirectedOnlyVertices
14   public void addAnEdge(Vertex start, Vertex end, int weight) {
15     addEdge(start, end);
16   }
17   /* More source code ... */
18 }
```

Parameter `weight` is used in *WeightedOnlyVertices* (see Lines 2 and 3), but not in *DirectedOnlyVertices* (see Line 14).

Listing 4.4: Latently Unused Parameter in FOP; taken from the GPL

Problems. The natural assumption is that a parameter has some effect on the method's outcome. Unused parameters make a method harder to understand because they foil this assumption (Fowler et al. [107], p. 277). Latently Unused Parameters only foil that assumption in particular cases, which I argue is at least as bad. Moreover, a Latently Unused Parameter introduces coupling between callers of the method and the feature that requires the parameter. For instance, a client application of the GPL that solely uses feature *DirectedOnlyVertices* but never *WeightedOnlyVertices*, will nonetheless have to supply a `weight` when calling `addAnEdge()`. This is not only confusing to developers of the client application. In addition, in order to understand the reason behind the extra parameter, the developers are forced to inspect *WeightedOnlyVertices* – a feature that is otherwise entirely irrelevant to them.

4.2.5 Large Feature

Derived from: Large Class (Fowler et al. [107], p. 78)

Fowler describes a Large Class as a class with a large number of instance variables, an abundance of methods, or both. Put simply, a Large Class is "trying to do too much" (Fowler et al. [107], p. 78). Likewise, Martin advises that a class should have a single responsibility – a single reason to change [233]. As an example, imagine a class aggregating news articles that is changed every time a new type of input is added (e. g., HTML, RSS) and every time a different output format (e. g., RTF, PDF) is requested. Handling different input and output formats are unrelated aspects, and the news aggregation class should not be responsible for both of them. Consequently, it is a Large Class.

Variability-aware description. I hypothesize that similarly to classes in an object-oriented system, features in an SPL can also be "trying to do too much," that is, have too many responsibilities. To describe them, I propose the variability-aware code smell Large Feature. First signs of this smell can be large numbers of lines of code and the introduction of many new elements, such as functions and classes. But counting lines of code and program elements is not enough. The only reliable way to detect a Large Feature is to analyze its responsibilities, that is, its reasons to change. For example, an air-conditioning feature whose implementation needs to change for every new temperature sensor and every new humidity sensor is likely too large. A solution would be to extract temperature and humidity measurement into child features.

Applies to: Annotation-based and composition-based mechanisms

Example. The *kernel* feature of the GUIDSL is a candidate for a Large Feature. According to the tool cloc,[4] the average size of a feature in GUIDSL is 412 non-blank, non-comment lines of code (SLOC). With 2 150 SLOC, feature *kernel* is much larger than that, and indeed, the feature exhibits an unfortunate aggregation of responsibilities. Most classes in *kernel* serve to implement abstract syntax trees. However, the command line interface, which is an entirely different responsibility, is also implemented here. Thus, both changes to the representation of abstract syntax trees as well as changes to the command line interface require modifications of the same feature. Moreover, one cannot simply create variants of GUIDSL without a command line interface because many other features depend on the abstract syntax tree implementation.

Problems. I argue that Large Features are problematic for the same reasons that Large Classes are problematic. First, mixing responsibilities makes the feature harder to understand. Secondly, changing one responsibility may have unwanted side-effects on other responsibilities. Thus, maintenance and evolution are hampered. Finally, even if only a portion of the Large Feature's functionality is needed, unwanted parts cannot be excluded. This hinders customization and increases footprint.

4.2.6 Switch Statements with Optional Cases

Derived from: Switch Statements (Fowler et al. [107], p. 82)

The smell Switch Statements is present when the same `switch` appears in different locations in a program (Fowler et al. [107], p. 82). It arises if type-specific behavior is implemented with the help of a `switch` statement or nested `if-else` statements. This is a smell in object-oriented languages because these languages provide subtype polymorphism (i. e., inheritance and overriding) to accomodate type-specific behavior. The problem with Switch Statements is code duplication. But in contrast to Inter-Feature Code Clones, where a group of statements is duplicated, Switch Statements mean that the control flow structure is duplicated. Hence, when adding a `case` clause to a duplicated `switch`, the developer has to find all the other occurrences of the `switch` and modify them accordingly. This is a tedious and error-prone task and increases maintenance costs.

[4] http://cloc.sourceforge.net/

Variability-aware description. Taking variability into account, I derive the new smell Switch Statements with Optional Cases. The smell is present when there is an optional `case` in a duplicated `switch` statement. The optional case is only active if a certain feature is selected, and otherwise inactive.

Applies to: Annotation-based mechanisms and (with workarounds) composition-based mechanisms

Example. In Listing 4.5, I show an annotation-based example for Switch Statements with Optional Cases. It has been taken from the file `pkcs11.c`, which provides cryptographic support for Firefox.[5] Both instances of the `switch` include the `case CKK_EC`, with the `case` guarded by the same preprocessor directive. The effect is that `case CKK_EC` is only present iff the macro `NSS_ENABLE_ECC` is defined.

In contrast to annotation-based mechanisms, composition-based mechanisms, such as FOP, lack direct support for optional `case` clauses in `switch` statements. Optional `case` clauses can be emulated, though, by combining multiple `switch` statements (one per feature) with `original` calls. I show an instance of this workaround in

[5] http://sourceforge.net/projects/portableapps/files/Source/Firefox/firefox-28.0.source.tar.bz2

```
1  static CK_RV
2  pk11_handlePublicKeyObject(
3    PK11Session *session,
4    PK11Object *object,
5    CK_KEY_TYPE key_type)
6  {
7    /* More code... */
8    switch (key_type) {
9    /* Handle other cases... */
10   case CKK_DH:
11     /* Handle CKK_DH... */
12     break;
13 #ifdef NSS_ENABLE_ECC
14   case CKK_EC:
15     /* Start handling CKK_EC... */
16     if ( !pk11_hasAttribute(object,
17               CKA_EC_POINT)) {
18       return CKR_TEMPLATE_INCOMPLETE;
19     }
20     pubKeyAttr = CKA_EC_POINT;
21     derive = CK_TRUE; /* for ECDH */
22     verify = CK_TRUE; /* for ECDSA */
23     encrypt = CK_FALSE;
24     recover = CK_FALSE;
25     wrap = CK_FALSE;
26     break;
27 #endif /* NSS_ENABLE_ECC */
28   default:
29     /* Handle default case... */
30 }
```

```
1  NSSLOWKEYPublicKey *
2  pk11_GetPubKey(
3    PK11Object *object,
4    CK_KEY_TYPE key_type,
5    CK_RV *crvp)
6  {
7    /* More code... */
8    switch (key_type) {
9    /* Other code for other cases... */
10   case CKK_DH:
11     /* Other code for CKK_DH... */
12     break;
13 #ifdef NSS_ENABLE_ECC
14   case CKK_EC:
15     /* Start handling CKK_EC... */
16     if (EC_FillParams(arena,
17           &pubKey->u.ec.ecParams.
18             DEREncoding,
19           &pubKey->u.ec.ecParams)
20         != SECSuccess)
21       break;
22     crv = pk11_Attribute2SSecItem(
23             arena,
24             &pubKey->u.ec.publicValue,
25             object, CKA_EC_POINT);
26     break;
27 #endif /* NSS_ENABLE_ECC */
28   default:
29     /* Other default code... */
30 }
```

Listing 4.5: Switch Statements with Optional Cases using annotations; taken from Firefox, file `security/nss/lib/softoken/pkcs11.c`

Listing 4.6. The code is based on TankWar,[6] an SPL of a Java game. TankWar was originally written in Ahead but the code excerpt has been translated to FeatureHouse, the FOP dialect used in this chapter. The example contains a method `init()` that is refined by two features, *Freeze* (see Lines 2–9) and *Firepower* (see Lines 12–19). Both refinements contain a switch statement (see Lines 4–8 and Lines 14–18, respectively). These `switch` statements refine each other as follows: Imagine a product that is the composition of *Firepower* after *Freeze* (and possibly some features before these two). Then, when `init()` method is called with `toolType=372`, the `original` call on Line 13 will invoke the *Freeze*'s `init()` on Line 2. After invoking another `original` method on Line 3, the `switch` statement is evaluated. Since `toolType` does not match the specified constant, the `case` clause on Lines 5–7 is skipped. Afterwards, control resumes at Line 14 where the refinement of the `switch` statement is evaluated. This time, `toolType` matches the `case` clause on Lines 15–17, and the corresponding statements are executed.

```
1  public class Tool {                                          Feature Freeze
2    protected void init(TankManager manager, int xPos, int yPos, int toolType) {
3      original(manager, xPos, yPos, toolType);
4      switch (toolType) {
5      case 371:
6        /* code specific to tool type 371 */
7        break;
8      }
9    }
10 }
```

```
11 public class Tool {                                        Feature Firepower
12   protected void init(TankManager manager, int xPos, int yPos, int toolType) {
13     original(manager, xPos, yPos, toolType);
14     switch (toolType) {
15     case 372:
16       /* code specific to tool type 372 */
17       break;
18     }
19   }
20 }
```

Listing 4.6: Switch Statements with Optional Cases in FOP; adapted from TankWar

Problems. Switch Statements with Optional Cases contain even more duplication than Switch Statements in non-configurable software. The reason is that apart from the replicated `case`s, the annotations (such as `#if`, `#endif`) that make some `case`s optional, are also replicated. Thus, not only does adding or removing a `case` require additional effort, but changing the (replicated) annotations of an optional `case` requires additional effort as well.

As the lengthy explanation of Listing 4.6 demonstrates, the FOP workaround for Switch Statements with Optional Cases makes it hard to follow the control flow. Moreover, the workaround may not always behave as expected. As an example, suppose that feature *Freeze* also contained a `case` clause for `toolType=372`.

[6] http://spl2go.cs.ovgu.de/projects/7

The expected behavior of composing the `switch` statements in features *Firepower* and *Freeze* would be for the `case` in feature *Firepower* to override the one in feature *Freeze*. But this does not happen because the `switch` statements are not truly merged. Instead, the FOP workaround simply executes them sequentially. Thus, a call to `init()` with `toolType=372` would first execute *Freeze*'s `case` for `toolType=372` and afterwards *Firepower*'s `case`. In summary, wile SWITCH STATEMENTS are already a smell in *Object-Oriented Programming (OOP)*, I argue that the smell becomes even stronger in FOP because refinements further obscure the control flow, and the FOP workaround may cause unexpected behavior.

4.3 Validation of the Catalog

While the description of the variability smells in the previous section have been elaborated with care, based on established code smells and on observations of variable code *in the wild,* they may be biased to the author's point of view. Hence, before developing tools to detect these smells it is imperative to verify whether the proposed smells "are really smells". This means finding out whether the described code patterns have been witnessed by others in practice and whether others agree they might pose problems regarding program comprehension, maintenance, and evolution. To this end, I conducted a survey among experts in the field of software product lines. In the following, I provide information about the setup, the particular questions and the result of this survey.

4.3.1 Objectives

The objective of my survey was to receive feedback on my proposed variability-aware code smells from product-line experts. Particularly, the survey was designed with the following two questions in mind:

RQ 1: *Do my proposed smells exist in design and implementation of SPLs?* Participants of the survey have different backgrounds in product lines such as teaching, analysis, implementation, or developing tool support. Nevertheless, my assumption was that most of them have to deal with SPLs on implementation level. Hence, with this question, I aim to elicit which of the proposed smells the participants encountered in the described (or similar) form.

RQ 2: *Are my smells problematic with respect to different aspects of SPL development?* Code smell are fundamentally just patterns in the source code, but that does not say anything about whether they are good or bad. A pattern could be an idiom or a design, both of which are beneficial. Code smells, on the other hand, are patterns that have negative effects. Hence, I am are also interested in how the participants would estimate the severity of my proposed smells with respect to program comprehension, maintainability, and evolvability, based on their experience with product-line implementation.

From the answers to the survey questions, I obtain valuable information about whether my proposed smells constitute real problems or not. This, in turn, lays the foundation for my work on detecting and qualitatively assessing variability-aware code smells, which I present later in this chapter.

4.3.2 Setup

In the following, I provide information about the concrete questions of the survey and the participants who answered them. The complete survey and all responses are available at the web page that supplements the paper in which my smell catalog was first published [96].[7]

Participants. The target audience of the survey were the participants of the 2014 international meeting on feature oriented software development (FOSD meeting, www.fosd.de/meeting2014), held at Schloß Dagstuhl in May 2014. The audience of this meeting consists of PhD students, postdoctoral researchers as well as professors, all of them working in the field of software product lines. The 2014 meeting had 32 participants, out of which two (Sandro Schulze and myself) were involved in the design of the survey. The other 30 participants received the survey two weeks before the meeting, and 17 of them turned in their responses. Two response sets were incomplete and thus have been excluded from further consideration.

One of the survey assumptions was that participants are familiar with general purpose programming and with SPL programming in particular. To validate that assumption, the survey included questions to measure each participants programming experience, based on the work of Feigenspan et al. [94]. Particularly, the survey asked questions about programming experience, different programming paradigms, and programming activity. Moreover, participants had to answer questions related to their experiences with different variability mechanisms and projects dealing with SPLs. The responses to these questions confirm that all participants are advanced programmers with a solid foundation in SPL implementation. Particularly, most of the participants are (very) experienced with the CPP and with feature-oriented programming. In summary, all participants are eligible for inclusion in the survey. Moreover, given their level of experience, I can expect reliable, informative results.

Survey – Structure & Questions. The survey consists of four sections, each of them covering a particular category of questions. The first section comprises questions about *personal data* such as gender, age, or the current affiliation of the participant. Next, there is a larger section with questions regarding the *background & experience* of the participant, which mainly focuses on the above mentioned programming experience. Moreover, participants were asked about their experience with SPL programming. This second section of the survey mainly ensures that the obtained answers are reliable.

The third section asks questions about the *knowledge of code smells.* Particularly, I was interested in how well participants know the concept of code smells, both, in general but also in terms of concrete smells. Given such knowledge, it is probably easier to identify and reason about variability-aware code smells.

Finally, the last section contains questions about the proposed code smells and thus constitutes the main part of the survey. Basically, it asks for each of the variability-aware code smells from Section 4.2 whether participants observed these smells, for which variability mechanism, and in which kind of project (e. g., open

[7] https://www.isf.cs.tu-bs.de/cms/team/schulze/material/vacs/

source, industrial). Moreover, participants had to estimate the severity of each code smell for the following aspects: program comprehension, maintainability, and evolvability (for instance, adding/changing code).[8]

The survey was accompanied by a short description of the six code smells, similar to the one in Section 4.2. This description is available on the supplementary page as well.

4.3.3 Results

Next, I present the main results of my validation survey. For the sake of brevity, I only present results about variability-aware code smells, because this has been the main objective of the survey. Detailed results (e. g., related to background and experience, knowledge of code smells, as well as additional qualitative comments) are available on the supplementary page.

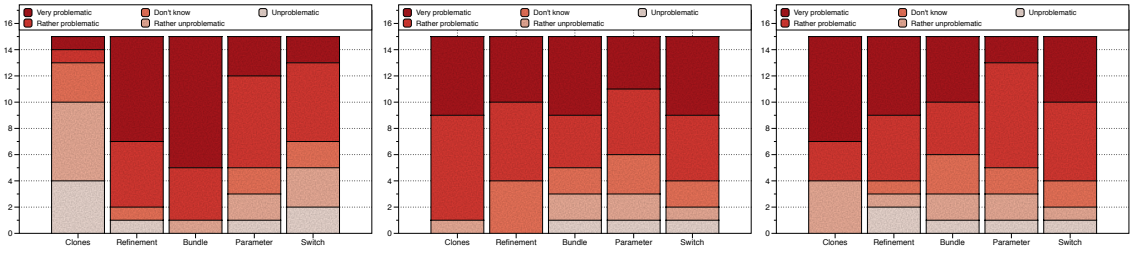
In Figure 4.2, I present the results related to the existence of my proposed smells. In a nutshell, my results reveal that most of the smells have been observed *in the wild* by the participants. Particularly, 63 % of the participants (on average) have observed at least one of the proposed smells. Even for the Switch Statements With Optional Cases smell, almost half of the participants confirmed that they observed this kind of smell. Moreover, participants were asked for which variability mechanism they observed the respective code smell. I did not ask this question for Annotation Bundle and for Long Refinement Chain because these smells are only defined for CPP and for FOP, respectively. For the other smells, the responses reveal that the smells occur roughly as often in SPLs implemented with the CPP as in SPLs using FOP. Beyond that, individual participants also observed certain smells for proprietary mechanisms.

[8]Due to a mistake in the survey setup, questions about severity are missing for Large Feature.

Lines show how often each smell has/has not been observed, bars indicate for which variability mechanism they have been observed.

Figure 4.2: Survey results on the occurrence of variability-aware code smells

Apart from information on the frequency of smell occurrences, the survey also asked participants to rate the negative the impact of each smell on SPL development. The ratings use a five-point Likert scale, ranging from "unproblematic" to "very problematic". I show the results in Figure 4.3. This figure is composed of three bar charts, which relate to three different aspects of software development. Chart (a) depicts the ratings related to program comprehension, chart (b) those related to maintainability, and chart (c) shows ratings related to evolvability. Overall, these charts reveal that the participants consider most of the code smells as problematic. First, regarding program comprehension (see Figure 4.3 (a)), especially Refinement Chain, Annotation Bundle, and Latently Unused Parameter are considered problematic, while Inter-Feature Code Clones are not seen as an issue. Secondly, the participants estimate that all code smells are mostly problematic for maintainability of SPLs (see Figure 4.3 (b)). Particularly, 61% on average consider the severity of the smells "rather problematic" or even "very problematic", with Inter-Feature Code Clones and Long Refinement Chain being the most severe smells. Similarly, the results reveal that participants confirm that these smells make SPLs difficult to evolve (see Figure 4.3 (c)). Although some participants consider particular smells to be rather unproblematic, the majority (i. e., 57 %) is convinced that they have a negative impact on evolution of SPLs.



(a) Program Comprehension **(b) Maintainability** **(c) Evolvability**

Results on how the survey participants estimate the impact of variability-aware code smells on three aspects of software development

Figure 4.3: Survey results on the negative impact of variability-aware code smells

4.3.4 Discussion

I now interpret and discuss the aforementioned results by relating them to the questions in Section 4.3.1. Furthermore, I enrich the previous quantitative results with selected qualitative comments.

RQ 1: *Do my proposed smells exist in SPLs?* The survey results confirm that my proposed variability-aware code smells occur regularly in SPLs. For all smells, a considerable amount of participants (more than 50 %) acknowledged that they faced the respective smells in SPL implementations. Moreover, participants observed these smells not only in toy examples or in programs they developed themselves; rather, they confirmed having witnessed them also in open source or industrial projects. This is also reflected by the following two comments, which I took from the survey:

> [Inter-Feature Code Clones] *"Our industry partner is struggling with inter-feature code clones due to a lack of awareness. . . ."*

> [Annotation Bundle] *"... in Linux, I have observed that in some cases a lot of #ifdefs are used in a method and some of them are nested making the method longer and more complicated."*

Additionally, I want to emphasize the fact that half of the affected projects referenced by the survey participants used the CPP as the variability mechanism. Since CPP usage is rare in academic projects, but common in industrial and open source development, this fact leads me to conclude that the proposed smells occur frequently *in the wild*.

RQ 2: *Are my smells problematic with respect to different aspects of SPL development?* Although my results reveal differences between particular code smells and the different aspects, the overall opinion is that my proposed smells impede program comprehension, maintenance, and evolution of SPLs. Particularly, up to 80% participants acknowledged possible problems. Of course, this result only reflects the personal opinion and experience of each participant and thus may be somewhat subjective (e. g., no independent measurements exist). However, each participant has long-time experience with SPLs, and worked on different projects from different domains. Moreover, these projects are not only proprietary SPLs but also encompass open source and industrial project. Hence, these results justify the conclusion that my proposed smells negatively affect SPL implementation and thus warrant further investigation. This conclusion is backed up by the previous qualitative comments of the survey participants, and also by the following ones:

> [Latently Unused Parameter] *"... at least identified this as a potential problem during own implementation."*

> [Switch Statement with Optional Cases] *"Have found myself in this issue a couple of times."*

Overall, the results of my survey confirm that (a) it is beneficial to include the notion of variability into our reasoning about code smells and (b) that variability-aware code smells, as proposed here, occur regularly and may impede SPL development. These are encouraging results, which prompted me to investigate variability-aware code smells in a more structured way. To this end, I performed an empirical study with the aim of gaining deeper insights into the frequency which these smells occur in practice and how they may affect SPL development. The study was performed with the help of an automatic code smell detection approach that I developed. I report on this detection approach and the corresponding empirical study in the remainder of this chapter.

4.4 Detection Concept

As discussed in the earlier sections of this chapter, variability has an impact on how source code is structured and can therefore lead to variability-aware code smells. In this section, I propose an approach for detecting such smells.

Whereas the catalog relates to both annotation-based and composition-based variability mechanisms, my smell detection approach targets only annotation-based mechanisms, specifically, programs written in C with CPP annotations. The rationale is that the smell detector mainly serves as a tool to support an empirical study of variability-aware code smells *in the wild*. Such a study necessitates realistic subject systems. Since the combination of C with CPP annotations is very popular in practice, realistic subject systems using this combination are easily found. By contrast, realistic subject systems using feature-oriented programming are virtually non-existent. Consequently, more relevant insights can be gained from detecting variability-aware code smells in programs written in C with CPP annotations than in programs using feature-oriented programming.

I chose a metrics-based smell detection approach and implemented it in a tool called SKUNK. Specifically, SKUNK's detection approach relies on static properties extracted from the source code. There is a large number of smell detection tools that also rely on (static) source code metrics (e. g., [229, 196, 257]). However, these tools could not be used as a basis for SKUNK because they lack metrics to capture variability in source code. Moreover, they focus detecting smells in JAVA and other object-oriented languages, whereas my goal is to detect smells in the procedural language C. Consequently, SKUNK was built on top of CPPSTATS[9], a framework to measure CPP usage in SPLs, and SRCML,[10] a flexible parsing infrastructure with good support for C and CPP annotations [204, 142, 57].

In the remainder of this section, I describe the metrics I propose to detect the variability-aware code smells ANNOTATION BUNDLE and LARGE FEATURE. SKUNK already computes metrics for other variability-aware code smells but I focus on these two because the corresponding metrics are the ones most well tested. Specifically, the metrics for ANNOTATION BUNDLE have been validated in a case study whose results have been analyzed qualitatively. I report the details about this case study in (see Section 4.6).

4.4.1 Metrics

My smell catalog in Section 4.2 comprises human-readable descriptions of variability-aware code smells, but these descriptions cannot be used as a specification for a smell detection tool right away. To develop such a tool, I first had to capture the essential properties of my smells with a set of metrics. I describe these metrics and how I combined them into a detection formula in the following paragraphs.

Metrics for Annotation Bundles

The atomic metrics that I use to detect ANNOTATION BUNDLEs are given in Table 4.1. Apart from the LOC metric, which simply counts lines of code, these metrics are specifically designed to capture the complexity added by preprocessor annotations. For instance, LOAC counts the lines of feature code within a function, whereas FL measures the number of feature locations.

[9] https://github.com/clhunsen/cppstats
[10] http://srcml.org/

The Annotation Bundle smell describes a heavily annotated function. Since "heavily annotated" is a property with many facets, I propose the aggregated metric AB_{SMELL} to measure how much a particular function suffers from the Annotation Bundle smell. I show the formula to compute the AB_{SMELL} metric in Equation 4.1.

$$w_1 \cdot \frac{LOAC}{LOC} \cdot FL + w_2 \cdot \frac{FC_{DUP}}{FL} + w_3 \cdot \frac{CND}{FL} \tag{4.1}$$

This formula consists of three terms, intended to capture three aspects of complexity introduced by `#ifdef` annotations: (1) Term one computes the ratio of annotated code (feature code) to all code. (In the remainder of this chapter, I will refer to this ratio as the $LOAC_{/LOC}$ ratio.) The reasoning behind this ratio is the assumption that, for instance, 10 lines of feature code in a function with a total length of 20 lines have more of a negative effect than 10 lines of feature code in a function of 100 lines. Furthermore, based on the intuition that a function with several small feature locations is more problematic than a function with a single, large feature location, the $LOAC_{/LOC}$ ratio is multiplied by the number of feature locations. As a result, with increasing scattering of feature code, the value of term one also increases. (2) The second term is designed to capture the average complexity of feature expressions within a function. If all feature expressions consist only of a single feature constant, term two evaluates to one. However, if complex feature expressions are used (i. e., feature expressions that refer to two or more feature constants), term two will evaluate to a value greater than one. (3) Finally, the third term accounts for nesting. Without nesting, this term evaluates to zero. By contrast, for functions with two feature locations, the second location being nested within the first one, the value will be 0.5.

Abbrev.	Full Name	Description
LOC	Lines of code	Source lines of code of the function, ignoring blank lines and comments.
LOAC	Lines of annotated code	Source lines of code in all feature locations within the function. Lines that occur in a nested feature location are counted only once.
CND	Cumulative nesting depth	Nesting depth of annotations, accumulated over all feature locations within the function. An `#ifdef` that is not enclosed by another `#ifdef` (a *top-level* `#ifdef`) has a nesting depth of 0; an `#ifdef` within a top-level `#ifdef` has a nesting depth of 1, and so on. CND is accumulated over all feature locations within the function. Thus, for example, if there are two feature locations in a function, each with a nesting depth of 1, CND of the function will be 2.
FC_{DUP}	Number of feature constants	Number of feature constants referenced by the feature locations in the function. Constants referenced multiple time are also counted multiple times.
FL	Number of feature locations	Number of blocks annotated with `#ifdef`s. An `#ifdef` containing a complex expression, such as `#if defined(A) && defined(B)`, counts as a single feature location. An `#ifdef` with an `#else` or `#elif` branch counts as two feature locations.

Table 4.1: Atomic metrics to capture the Annotation Bundle code smell

For the example Annotation Bundle in Figure 4.2 (see page 59), the values of the atomic metrics are as follows:

- LOC = 27 (27 lines of code excluding blank lines and comments),
- LOAC = 8 (eight lines of annotated code),
- FL = 5 (five feature locations),
- $FC_{DUP} = 7$ (there are five distinct feature constant, of which `USE_ALARM_THREAD` and `SIGNAL_HANDLER_RESET_ON_DELIVERY` occur twice), and
- CND = 1 (nesting only occurs once, for the feature location on Line 22).

Next, I use these metrics to compute the three terms of AB_{SMELL}. This computation yields 1.48, 1.4, and 0.2 for the first, second, and third term, respectively. Assuming that all weights (w_1, w_2, and w_3) are 1, the final AB_{SMELL} value for the function in Figure 4.2 is 3.08. Now that the value is computed, the following question arises: Is 3.08 a high value that developers should pay attention to, or is it a low value that can be ignored? To answer this question, the 3.08 must be compared to other AB_{SMELL} values. For example, it is possible to compare this value to the AB_{SMELL} values of other functions in the same project. If 3.08 is above the 95th percentile of all AB_{SMELL} values, the function should be inspected. If, on the other hand, 3.08 is close to the median, it is nothing out of the ordinary and can be ignored. I come back to the topic of interpreting smell metric values at the end of Section 4.4.2.

Metrics for Detecting Large Features

The metrics to detect Large Features are given in Table 4.2. In contrast to Annotation Bundle, which focuses on a single function definition, Large Feature focuses on a feature, which can be implemented in many places throughout the system. Consequently, the metrics are collected at a different scope: Whereas the metrics for Annotation Bundles are collected at the scope of an individual function, the metrics for Large Features are collected at the global scope, that is, taking the whole software system into account.

Abbrev.	Full Name	Description
CU_{FEAT}	Number of compilation units	Number of compilation units (i. e., `*.c` files) in which feature locations referencing the given feature constant occur.
FL_{FEAT}	Number of feature locations	Number of feature locations that reference the given feature constant.
$LOAC_{FEAT}$	Lines of code of the feature	Source lines of code annotated by the given feature constant, ignoring blank lines and comments.
CU_{TOTAL}	Number of all compilation units	Total number of all compilation units (i. e., `*.c` files) within the system.
FL_{TOTAL}	Number of all feature locations	Total number of feature locations throughout the system.
LOC_{TOTAL}	Total lines of code	Source lines of code within the entire system, ignoring blank lines and comments.

Table 4.2: Atomic metrics to capture the Large Feature code smell

The metrics for the feature of interest are listed in the upper half of Table 4.2. These metrics are collected by analyzing all compilation units (that is, all `*.c` files) in the system at hand. During the analysis, all feature locations in a compilation

unit are checked for references to the feature constant that belongs to the feature of interest. The $\mathrm{CU}_{\mathrm{FEAT}}$ metric is the number of compilation units that contain at least one feature location that references the feature constant. $\mathrm{FL}_{\mathrm{FEAT}}$, in turn, is the number of all such feature locations. Finally, $\mathrm{LOAC}_{\mathrm{FEAT}}$ is the number of non-blank, non-comment lines of code that occur inside these feature locations. These feature-specific metrics are combined with metrics about the entire system, which are listed in the lower half of Table 4.2. Together, these metrics are aggregated into the $\mathrm{LF}_{\mathrm{SMELL}}$ metric to detect Large Features. I show the formula to compute the $\mathrm{LF}_{\mathrm{SMELL}}$ metric in Equation 4.2.

$$w_1 \cdot \frac{LOAC_{FEAT}}{LOC_{TOTAL}} + w_2 \cdot \frac{FL_{FEAT}}{FL_{TOTAL}} + w_3 \cdot \frac{CU_{FEAT}}{CU_{TOTAL}} \tag{4.2}$$

The three terms of this formula relate to the size and to the degree of scattering of a feature: (1) The first term computes the ratio of code that belongs to the feature of interest to all code within the system. Basically, this metric measures the same property as the $\mathrm{LOAC}_{/\mathrm{LOC}}$ ratio in the detection formula for Annotation Bundles, but on a different scope. For features with very little code, the ratio of $\mathrm{LOAC}_{\mathrm{FEAT}}$ to $\mathrm{LOC}_{\mathrm{TOTAL}}$ will be close to zero, whereas for larger features, it assumes values closer to one. (2) The second term accounts for fine-grained scattering of the feature. This is expressed as the ratio of feature locations that reference the feature in question to all feature locations. The more scattered a feature is, the higher this ratio will become because highly scattered features are implemented in many different feature locations and, hence, have a high $\mathrm{FL}_{\mathrm{FEAT}}$ value. Note that this term ignores whether or not different feature locations reside in the same compilation unit or in different ones. This is what the third term is for. (3) The third and final term captures a more coarse-grained notion of scattering. It is the ratio of compilation units that contain parts of the feature's implementation to all compilation units. In the extreme case that feature code is present in every compilation unit, this ratio will be one. Otherwise it will be a value lower than one but greater than zero.

4.4.2 Parameterization and Thresholds

How much a particular atomic metric contributes to the complexity may vary depending on the individual perception of the developer working with the code. Hence, my detection method can be parameterized using *weights* and *thresholds*. If intimate knowledge of the source code or guidelines for CPP usage are available, users can encode this knowledge in the provided weights and thresholds to obtain more precise results.

For both the $\mathrm{AB}_{\mathrm{SMELL}}$ metric in Equation 4.1, as well as for the $\mathrm{LF}_{\mathrm{SMELL}}$ formula in Equation 4.2, the influence of each of the three terms is controlled by a weight (w_1, w_2, and w_3, respectively). For example, a user wishing to detect Large Features may want to put more emphasis on size than on scattering. Using the $\mathrm{LF}_{\mathrm{SMELL}}$ formula, this can be achieved by assigning a large value to w_1 and smaller values to w_2 and w_3.

In addition to weights, users can specify a lower boundary for a particular term with the help of thresholds. Such thresholds are intended to reduce false positives

in the result set. Following the ideas of Marinescu, Lanza, and Moha (for details, see Section 2.4.1, as well as [227, 196, 257]), thresholds in Skunk are either absolute or relative. As an example for an absolute threshold, a user can set the threshold for the $\text{LOAC}_{/\text{LOC}}$ ratio to 0.5 when detecting Annotation Bundles. That way, only functions with at least 50 % of annotated code will be considered Annotation Bundles. Relative thresholds, in turn, enable users to detect smells based on outliers, that is, program elements whose metric value is unusually high or low compared to other metric values in the analyzed system. Skunk uses relative thresholds to detect Large Features. For example, a user can specify that a feature is a Large Feature if its size is in the top 5 % range. To do so, the user would set the $\text{LOAC}_{\text{FEAT}}$ parameter to 0.95.

After parameterizing the detection formulas with weights and reducing the result set with thresholds, the last question is how to interpret the reported AB_{SMELL} and LF_{SMELL} values. According to previous work on code smell detection (e. g., [227, 196, 257]), the common way to interpret a given metric value is to compare it to other metric values in a reference corpus. This corpus may comprise all the function in the software system being analyzed but it could also be a curated corpus compiled by a group of experts. Given such a reference corpus, several strategies, such as ranking or normalization, can be used. Ranking means that all functions are sorted in descending order by their smelliness metric value. Guided by this ranking, developers can focus their attention on the highest-ranked functions, for example, on the functions whose smelliness value is above the 95th percentile. The other strategy, normalization, means that a given metric value is put in relation to some fixed reference value. For example, every AB_{SMELL} value could be divided by the maximum AB_{SMELL} value from the corpus, and if the result is close to 0.0, smelliness is low; if it is close to 1.0, smelliness is high. Beyond these simple approaches, further approaches to interpret metric values and determine proper weights and thresholds have been proposed in the literature. For example, if training data is available, machine learning techniques can produce good results. See Section 2.4.1 in Chapter 2 for more details.

4.5 Implementation

In Figure 4.4, I show the basic detection process of Skunk, the tool that implements my variability-aware code smell detection concept. Detection starts by preprocessing the source code of the analyzed system. Preprocessing is performed by two external tools, cppstats and src2srcml. cppstats extracts most variability-related information that Skunk requires, such as the locations of `#ifdef` directives. However, Skunk needs additional syntactic information (e. g., the location of function definitions), which cppstats does not provide. To extract this information, I rely on the output of src2srcml, a tool that transforms C code into an XML representation containing information similar to an *Abstract Syntax Tree (AST)*. One important property of this XML representation is that it preserves all preprocessor directives (such as `#ifdef`s) that appear in the original source code files. This is in contrast to most C parsers, which create the AST only after preprocessing, rendering them useless as a basis for Skunk.

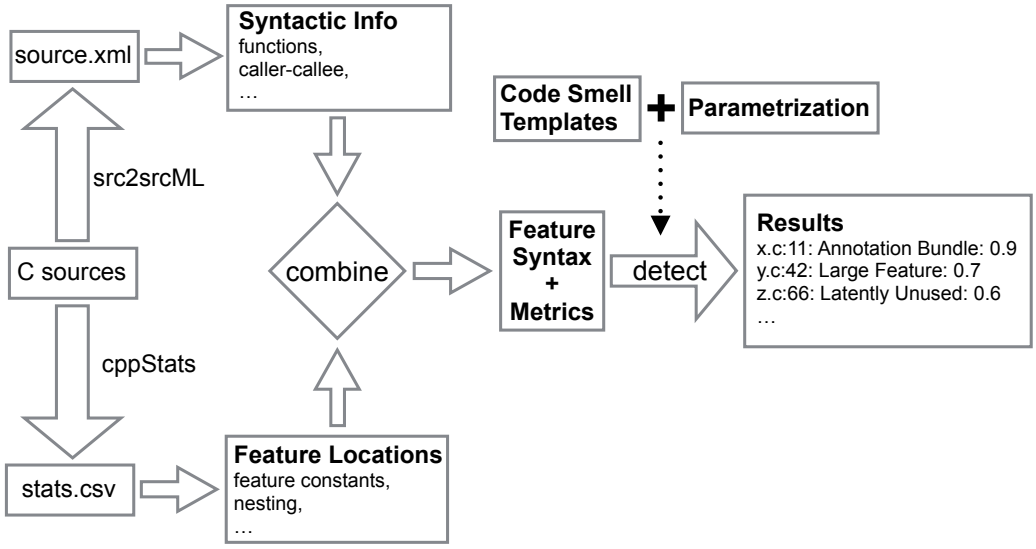


Figure 4.4: Variability-aware code smell detection architecture

After preprocessing, SKUNK extracts feature locations and other variability-related information from the results of CPPSTATS. Function definitions and further metrics are extracted from the output of SRC2SRCML. Next, SKUNK combines both sources of information in order to calculate the atomic metrics shown in Table 4.1 and Table 4.2. The results of this data extraction step are stored in an internal format for further processing. The extracted data is represented by the box labeled **Feature Syntax + Metrics** in Figure 4.4.

Based on the extracted data, the actual detection of variability-aware code smells takes place. Detection is controlled by a *Code Smell Template*, in which the user specifies his/her smell detection criteria, namely, the smell to detect as well as the weights and thresholds to apply. For instance, the user may want to limit reported ANNOTATION BUNDLES to functions that contain at least 50 % of annotated code. This can be achieved by setting the parameter `Method_LoacToLocRatio` to `0.5;mandatory`, where `0.5` is the threshold for the $\mathrm{LOAC}_{/\mathrm{LOC}}$ ratio and the keyword `mandatory` instructs SKUNK to ignore functions with a ratio below that threshold. For program elements that pass this filtering step, SKUNK computes the smell metric. In particular, when detecting ANNOTATION BUNDLES, the $\mathrm{AB}_{\mathrm{SMELL}}$ (see Equation 4.1) metric is computed, and when detecting LARGE FEATURES, the $\mathrm{LF}_{\mathrm{SMELL}}$ metric is computed (see Equation 4.2). Afterwards, the metric value, along with the function's name (in the case of ANNOTATION BUNDLES) or feature name (in the case of LARGE FEATURE) and location, are written to a results file. This results file is depicted as the box named **Results** in Figure 4.4. Finally, the results can be inspected by the user, who can identify functions or features of interest and take appropriate corrective action.

Besides setting the $\mathrm{LOAC}_{/\mathrm{LOC}}$ threshold, SKUNK's *Code Smell Templates* offer additional parameters to control the other thresholds and weights discussed in the previous section. Beyond that, SKUNK computes further metrics to be used in the

future to detect additional variability-aware code smells. The source code of Skunk, as well as a short how-to can be obtained from the tool's repository at GitHub.[11]

4.6 Case Study of Detecting Annotation Bundles

The operationalization of variability-aware code smells by way of metrics makes it possible to answer the interesting question to what extent these smells occur *in the wild.* To this end, I applied my smell detection tool in an exploratory case study using five highly configurable open-source software systems.

The main goal of this study was to establish a ground truth for the detection of Annotation Bundles but also to develop an infrastructure for detecting further variability-aware smells listed in my catalog (see Section 4.2). I focus specifically on the Annotation Bundle smell to allow for an in-depth, qualitative analysis of the detection results. In the remainder of this section, I present the setup of this case study, including research questions, subject systems, and methodology. Furthermore, I present the results and analyze them qualitatively. This qualitative analysis reveals insights into why some of the functions that Skunk reported as Annotation Bundles were actually smelly. Even more interesting, the analysis also explains why other alleged Annotation Bundles were, in fact, not smelly.

4.6.1 Research Questions

Basically, I want to answer two research questions aimed at assessing the usefulness of both, the concept of variability-aware code smells as well as my detection concept.

RQ 3: *Does my algorithm detect meaningful instances of the* Annotation Bundle *smell?*

With this question, I evaluate the precision of my detection algorithm. By means of manual inspection, I determine to what extent automatically detected instances of the Annotation Bundle smell align with human perception regarding understandability and changeability.

RQ 4: *Do* Annotation Bundles *reveal recurring, higher-level patterns of annotation usage?*

With this question, I explore why developers introduced the smell. In particular, I want to know whether implementation- or domain-specific characteristics foster the occurrence of the smell. Moreover, I investigate which of such patterns are more likely to negatively impact the source code.

4.6.2 Subject Systems

In Table 4.3, I show an overview of the subject systems of my exploratory case study. I used five highly configurable open-source systems of medium size. For each system, I list its version, domain (e. g., text editor), size (in LOC; measured using the cloc tool[12]), the ratio of annotated code to total system size (column **% LOAC**, given

[11] https://github.com/wfenske/Skunk/
[12] http://cloc.sourceforge.net

in percent), and the number of potential instances of the Annotation Bundle detected by Skunk. I report the ratio of annotated code to provide a rough impression of how much variable code each system contains. The given values are based on the measurments published by Liebig et al. for slightly older versions of my subject systems [204]. Note that values in LOC and % LOAC column only include non-blank and non-comment lines of C code. Moreover, header files have been excluded from both measurements, since Skunk does not analyze them.

Name	Version	Domain	LOC	% LOAC	AB_{pot}	Median AB_{smell}
Emacs	24.5	Text editor	247,403	29.0	20	6.9
Libxml2	2.9.2	XML processing library	215,751	69.5	76	4.0
Lynx	2.8.8	Text-based browser	115,102	43.8	76	10.9
PHP	5.6.9	Scripting language	117,813	18.2	26	5.1
Vim	7.4	Text editor	285,817	69.8	259	7.1

Version: version number; **Domain:** application domain; **LOC:** number of non-blank, non-comment lines of C code, ignoring header files; **% LOAC:** ratio of lines of annotated code (feature code) to total lines of code, in percent; **AB_{pot}:** number of potential Annotation Bundles reported by Skunk; **Median AB_{smell}:** median of metric values, computed over all candidates of AB_{pot}

Table 4.3: Overview of subject systems used in evaluation

My subject systems constitute a subset of the forty systems used by Liebig et al., who analyzed preprocessor usage [204]. I based my selection on three criteria: 1. I chose systems of medium size in order to get a sufficient number of potential instances of the Annotation Bundle smell but also to show the applicability to systems of reasonable size. 2. I targeted systems with a considerable fraction of annotated code, the rationale being that systems with a high proportion of feature code are more likely to contain Annotation Bundles than systems with little feature code. 3. In order to remove bias regarding specific properties of a particular application domain, I chose systems of four different domains. Nevertheless, I included two text editors (*Emacs* and *Vim*) to account for different coding styles within a single domain.

Threats to validity. The selection of subject systems poses a potential threat to the external validity of the findings. I concede that my findings may be not representative for domains that none of my subject systems belongs to. For instance, my findings may not be representative for embedded systems. However, as my systems represent different domains, my selection is not biased to only one specific domain. Consequently, my results can at least be generalized to similar domains.

4.6.3 Methodology

In the following, I describe my methodology for detecting and evaluating the Annotation Bundle smell. This methodology consists of three steps: (1) parametrization of the detection algorithm, (2) creating the sample set, (3) and validating these samples.

Parametrizing the Detection Algorithm

As mentioned in Section 4.4, the user can parametrize the detection process by providing thresholds and weighting factors for particular measures. For my case study, I set thresholds for the following four parameters:

- $\text{LOAC}_{/\text{LOC}} = 0.5$: As the considered smell focuses on problematic usage of CPP directives, I required each function to contain at least 50% annotated code. Otherwise, I assumed to obtain too many false positives, that is, functions that smell only a little or not at all.
- $\text{FC}_{\text{DUP}} = 2$: Since tangling and scattering are part of my AB_{SMELL} metric, I required the occurrence of at least two different feature constants.
- $\text{FL} = 2$: I argue that one large chunk of annotated code is something that happens frequently in C programs and hardly indicates problems for changeability or understandability. Thus, I chose a higher threshold.
- $\text{CND} = 1$: My intuition was that nesting significantly contributes to the complexity of an ANNOTATION BUNDLE. Hence, I set this threshold such that only functions with at least one nested `#ifdef` directive are reported.

All of the aforementioned parameters are mandatory for the detection process, that is, a function will only be reported by SKUNK if it adheres to *all* of these thresholds. Based on test runs of SKUNK on other system, I chose these thresholds to reduce the number of false positives. While I could have chosen higher thresholds, this might have resulted in decreasing recall, that is, the likelihood of missing potentially smelly functions would have increased.

Sample Selection

After running SKUNK on my subject systems, I determined the precision of my detection approach. Since there is no baseline for variability-aware code smells so far (e. g., a human oracle), I decided to manually inspect the reported smell instances and classify them into true positives and false positives. This inspection was restricted to a subset of the reported smell instances because inspecting all of them is infeasible in a reasonable time. To create this subset, I took a sample of 20 smell instances from each subject system by the following method: Half the samples (i. e., 10 per system) constitute those smell instances with the highest AB_{SMELL} value. This top-10 list also allows me to evaluate to what extent the AB_{SMELL} indicates the strength of the smell, i. e., whether functions with higher AB_{SMELL} values are more smelly than functions with lower values. For the second part of the sample set, I split the remaining result set into 10 equally distributed segments. For each of those segments, I randomly selected one entry and added it to my sample set. This way, I obtained a cross section of all smell instances for my evaluation. Moreover, I gain insights on a possible cutoff regarding my detection metric, that is, whether smells occur only above a certain metric value.

To illustrate the sample selection process, I use *Lynx* as an example. The detection result of *Lynx* consists of 76 potential smell instances. In step one, I included the top-10 results in my sample set. Next, I split the remaining 66 results into 10 segments, encompassing approximately seven smell instances each, and randomly selected one instance from each segment. This sample set was then manually validated, which I explain next.

Validating the Smells

I use the previously created sample set for assessing the precision of my detection algorithm. To this end, a *cross validation* was performed by reviewing each sample independently and assigning a rating to it, thus, simulating a human oracle. The cross validation was executed by Sandro Schulze, who is a coauthor of the paper in which this case study was first described [97], and by myself. We used a 3-point scale for assessing the sample smells with the ratings -1 (no negative impact), 0 (partial negative impact), and 1 (high negative impact). While the first indicates a false positive sample (i. e., no smell), the other two ratings indicate at least partial smells. The difference between a partial and a high impact is that the former means that only certain code fragments impede changeability and understandability; for the latter, the whole (or most of) the function is negatively affected.

For the actual inspection of the smells (and their impact), my coauthor and I mainly focused on how much preprocessor annotations interfere with data and control flow. Among other things, this includes annotated variable initializations and assignments, annotated conditional branches (both, whole blocks as well as just parts of the condition) and loops. Moreover, we considered nested annotations and compound feature expressions (i. e., `#ifdef` directives that reference multiple feature constants). For all of these occurrences, we rated how difficult annotations make it to understand the given function (e. g., its data and control flow) or to identify the location for a possible change. Note that in our review, we disregarded shortcomings that result *only* from inappropriate use of the programming language, such as sequences of nested conditionals or just overly long functions. Based on this review process, both of us individually assigned a rating to a respective smell sample. Moreover, we recorded qualitative data regarding different properties of the function in the sample. For example, we noted down the purpose of the function and why or why not we believed it was smelly. Moreover, we listed possible reasons why the original developers may have introduced variability.

After both of us had completed their review tasks, we compared our ratings, sample by sample. In case of different ratings, we discussed our ratings (including the qualitative observations) until we achieved a rating we both could agree on.

Threats to validity. While the study methodology was devised with care, it inherently comprises certain threats. First, parametrization may lead to missing smell instances in the results set. However, I have chosen conservative values for the respective parameters, thus mitigating the risk of missing "real" smells. In any case, I do not claim anything about the recall of my detection concept, only about its precision. Second, sampling bears the risk not being representative or too small regarding the whole data set. I mitigate this threat in two ways: (1) The selection method incorporates not only peak values (i. e., the top-10 result for the AB_{SMELL} metric) but also covers the rest of the value domain. This coverage is achieved using the randomized, equally distributed sampling strategy described above. (2) My sample sets encompass between 8% and 100% of the initial result set. This is a reasonable size for being representative. Finally, my validation process may introduce a bias, because the ratings are based on personal opinions, which renders the results subjective. Hence, I set up a strategy with concrete criteria for rating the inspected

smells *in advance*, thus preventing an entirely arbitrary rating procedure. Moreover, we first rated the samples independently and subsequently discussed our ratings in a review meeting.

4.6.4 Results

I provide the initial result set, containing all potential instances of Annotation Bundle after running my detection algorithm, in Table 4.3. The final results, after manual inspection and validation, are given in Table 4.4.

Initial Result Set

Basically, my data reveal that each of the subject system exhibits potential smells. Nevertheless, the amount of detected smells differs considerably, ranging from only 20 (for *Emacs*) to 259 (for *Vim*). These differences indicate that there is considerable heterogeneity in *how* different systems use CPP annotations. Moreover, the average AB_{SMELL} metric value (computed as the median over all values per system; see Table 4.3) exhibits differences as well. Hence, for each system, a different cutoff value may apply at which the result set turns from rather reliable results into an excessive number of false positives.

Inspection of Sample Set

In Table 4.4, I show the results for the manual inspection of the sample set. In particular, I show two different views on my results: ratings for the whole sample set (stated in the *all* subcolumns) and for a reduced sample set encompassing only the smells with the top-10 AB_{SMELL} metric values (stated in the *top10* columns). For the whole sample set, my data reveals that approximately 50 % of the detected smells have been discarded (rating of -1). On the negative side, this indicates that the precision of my detection algorithm is only average. On the positive side, the data also reveals that each system exhibits smells that have a medium or high impact on changeability and understandability of the source code. Moreover, precision differs considerably between systems, ranging from 25 % for *PHP* to 70 % for *Vim*. For the top-10 sample set, by contrast, I obtain more precise results. In particular, the average precision is 72 %, with four systems having a precision between 70 % and 90 %. Additionally, note that the number of smells with a high impact (rating $+1$) clearly exceeds the rating for medium impact. Thus, the higher the AB_{SMELL} value of a function, the more confident we can be that the use of preprocessor directives in that function is actually problematic. In summary, especially the smells with high metric values are good indicators for real shortcomings and thus are useful to guide developers to functions that require special attention.

The raw results, including code artefacts, metric values, and ratings, are available on the website that complements the original publication about this case study.[13] Next, I present the insights of my qualitative analysis.

[13] https://www.isf.cs.tu-bs.de/cms/team/schulze/material/scam2015skunk/

Name	AB_{-1}		AB_0		AB_{+1}		AB_{0+1}		% Prec.	
	top10	all	top10	all	top10	all	top10	all	top10	all
Emacs	2	8	4	8	4	4	8	12	80	60
Libxml2	3	12	3	4	4	4	7	8	70	40
Lynx	2	8	2	6	6	6	8	12	80	60
PHP	6	15	4	5	0	0	4	5	40	25
Vim	1	6	1	4	8	10	9	14	90	70
Total	14	49	14	27	22	24	36	51	72	51

AB_{-1}: samples with a manual rating of −1 (no impact); **AB_0:** samples with a manual rating of 0 (medium impact); **AB_{+1}:** samples with a manual rating of +1 (high impact); **AB_{0+1}:** samples with a manual rating of 0 or +1 (medium or high impact); **% Prec.:** ratio of manually confirmed smell instances rated 0 or +1 in relation to sample size, in percent. The *top10* columns refer to the top ten detection results. The *all* columns refer to all twenty samples.

Table 4.4: Detection results for Annotation Bundle smell

4.6.5 Qualitative Analysis

In this section, I report on qualitative observations that Sandro Schulze, who was my partner in the manual validation process, and myself made during our manual inspection of the samples. In particular, I relate these observations to the research questions posed in Section 4.6.1. I provide details why smells occur, whether they follow certain patterns, and why there are sometimes good reasons that developers write heavily annotated functions.

RQ 3: When is a Smell a Smell?

The goal of RQ 3 is to find out whether my algorithm detects meaningful instances of the Annotation Bundle smell. Overall, we made four important observations with respect to this question, which I detail in the following. *Basically, these observations show that my metric-based approach enables the detection of smells that have a negative impact.* However, they also shed a light on possible weaknesses of my detection algorithm. For instance, my algorithm disregards recurring patterns of annotations, which actually improve code understandability and thus cannot be considered smelly.

Observation 1: *There is no single reason for why functions smell.* During manual inspection of potential smells, we repeatedly noted that no single property (e. g., LOAC ratio, nesting depth) reliably predicts whether a sample was actually negatively affected by variability (according to my review partner's and my subjective perception). Rather, a combination of different properties is necessary to make a potential smell a real one. For instance, samples 10 and 14–20 of Libxml2 constitute unit tests for configurable (i. e., optional) functionality. I show an example of such a function in Listing 4.7. The implementation of this function depends on functionality provided by the feature `LIBXML_TREE_ENABLED`. Since the function would not compile without this functionality, most of the body is enclosed in two

```
1  static int
2  test_xmlValidateNCName(void) {
3    int test_ret = 0;
4
5  #if defined(LIBXML_TREE_ENABLED) || defined(LIBXML_XPATH_ENABLED) || defined(
       LIBXML_SCHEMAS_ENABLED) || defined(LIBXML_DEBUG_ENABLED) || defined (
       LIBXML_HTML_ENABLED) || defined(LIBXML_SAX1_ENABLED) || defined(
       LIBXML_HTML_ENABLED) || defined(LIBXML_WRITER_ENABLED) || defined(
       LIBXML_DOCB_ENABLED) || defined(LIBXML_LEGACY_ENABLED)
6  #ifdef LIBXML_TREE_ENABLED
7    int mem_base;
8    int ret_val;
9    xmlChar * value; /* the value to check */
10   int n_value;
11   int space; /* allow spaces in front and end of the string */
12   int n_space;
13
14   for (n_value = 0;n_value < gen_nb_const_xmlChar_ptr;n_value++) {
15     for (n_space = 0;n_space < gen_nb_int;n_space++) {
16       mem_base = xmlMemBlocks();
17       value = gen_const_xmlChar_ptr(n_value, 0);
18       space = gen_int(n_space, 1);
19
20       ret_val = xmlValidateNCName((const xmlChar *)value, space);
21       desret_int(ret_val);
22       call_tests++;
23       des_const_xmlChar_ptr(n_value, (const xmlChar *)value, 0);
24       des_int(n_space, space, 1);
25       xmlResetLastError();
26       if (mem_base != xmlMemBlocks()) {
27         printf("Leak of %d blocks found in xmlValidateNCName",
28                xmlMemBlocks() - mem_base);
29         test_ret++;
30         printf(" %d", n_value);
31         printf(" %d", n_space);
32         printf("\n");
33       }
34     }
35   }
36   function_tests++;
37 #endif
38 #endif
39
40   return(test_ret);
41 }
```

Listing 4.7: Test code for optional functionality in LIBXML2, file `testapi.c`.

large annotated blocks (see Lines 5 and 6). Especially the annotation on Line 5 looks daunting because it references ten feature constants. However, if we assume that `LIBXML_TREE_ENABLED` (referenced on Line 6) is disabled, the function becomes very simple. Only two statements remain, the variable definition on Line 3, and the return statement on Line 40. Consequently, the function will simply return 0 in configurations without `LIBXML_TREE_ENABLED`, i. e., the test will pass successfully. Like the other test functions in LIBXML2, the example in Listing 4.7 exhibits a high per-

centage of annotated code (approximately 95 %), which normally indicates a highly variable function. However, with only two feature locations per test, one immediately wrapped around the other, the annotations are very coarse-grained and easy to follow. Consequently, these samples were rated −1 by both authors, as they were easy to understand.

I learned from this observation that my detection formula, which combines several metrics, is a good baseline for detecting Annotation Bundles in many real-world scenarios. Nevertheless, as the Libxml2 samples show, there is room for improvement since particular patterns of `#ifdef` usage can lead to false positives.

Observation 2: *Interactions of preprocessor variability with runtime variability leads to smelly code.* Another observation we made is that annotated code amplifies the negative impact of an already complex control flow. For instance, function `ins_redraw` from Vim (sample 13, 83 LOC) is rather short. Nevertheless, it contains 11 (runtime) `if` statements, and most `if`s are nested over one or two levels within another `if`. While this control flow is already hard to understand, these (runtime) `if` statements are additionally annotated with 12 (compile-time) `#ifdef`s. Even more, some of these `#ifdef`s are *undisciplined* [206] because they annotate parts of an `if`-condition. As such complex interactions impede understandability and changeability, this example and similar ones have been rated +1 (high impact).

I learned from this observation that it matters *how* cpp directives interact with the structure of the host language. Taking this into account will make the detection algorithm more precise.

Observation 3: *Keep it short.* Many short functions (i. e., $LOC \leq 100$), even some with a top-10 AB_{smell} value, were rated −1 (no impact). Longer functions, by contrast, were more likely to be rated 0 or +1 (medium/high impact). The reasons may be related to the interaction between complex code in the host language and preprocessor variability, as discussed in Observation 2. Short functions are less likely to contain complex control flow. Hence, they inherently exhibit fewer interactions between preprocessor and runtime variability. An exemplary comparison of the LOC metrics for Lynx supports this intuition: samples rated −1, 0, and +1 have an average length of 83, 294, and 319 LOC, respectively. A similar trend can be observed in the Emacs samples. Both examples indicate that LOC may be positively correlated with preprocessor annotations that have a negative impact. Hence, I learned that the precision of the AB_{SMELL} metric may also benefit from taking LOC into account. Note, however, that LOC alone is not sufficient to identify Annotation Bundles because we also found a number of long, heavily annotated functions that were not smelly.

Observation 4: *Repetitive feature code aids comprehension.* Some smells were discarded although they exhibited high AB_{SMELL} values (e. g., due to a vast amount of feature locations and constants). The reason is that we observed a repetitive structure of the variable code, which aided comprehension when reading the code. For instance, function `f_has` in Vim (file `src/eval.c`) contains 95 % of annotated code and 177 feature locations. This function received the highest AB_{SMELL} value for Vim. However, most feature locations in `f_has` are related to a single statement

that initializes an array of strings. These strings represent the configuration options of a particular Vim installation so that Vim can report its configuration at runtime. In Listing 4.8, I show an excerpt of the respective code. As the excerpt illustrates, the feature locations in `f_has` merely control whether a particular option string is included in the array or not. For instance, if and only if Vim is built for the operating system OS2, this array will contain the string `"os2"`. Consequently, function `f_has` was rated −1, because the repetitive structure aids understanding and changing the code. Finding the place to add a new feature, for example, and inserting the necessary `#ifdef`s and string constant, will be straightforward. Note that our observation is in line with the work of Kapser and Godfrey regarding code cloning practices, which are not necessarily harmful and sometimes even beneficial [152]. In this sense, we argue that repetitive feature code is another argument in favor of their claim.

```
1  static char *(has_list[]) =
2  {
3  /* ...*/
4  #ifdef OS2
5    "os2",
6  #endif
7  #ifdef __QNX__
8    "qnx",
9  #endif
10 #ifdef UNIX
11   "unix",
12 #endif
13 /* ... more features ... */
14 };
```

Listing 4.8: Repetitive feature code in Vim

In summary, I learned that recurring structural similarities of annotated code should be considered by my detection algorithm. By taking such similarities into account, I may be able to obtain more precise results.

RQ 4: High-Level Patterns of Annotated Code

With RQ 4, I investigate whether more abstract patterns exist that lead to the introduction of the Annotation Bundle smell, or, conversely, patterns that prevent heavily annotated functions from being smelly. Based on qualitative analysis of my review partner and myself, we identified three such patterns and report another observation that may introduce further smells.

Observation 5: *Adapter pattern.* Some platforms (e. g., operating systems or compilers) provide implementations of functions that do fully conform to the expectations of a particular subject system. Hence, systems that rely on these functions provide adapters for platforms that do not provide a suitable implementation. For instance, Emacs provides an adapter for `gettimeofday`, whose purpose is to return the current time. On some platforms, this adapter will delegate to the platform-provided implementation and perform clean-up actions to work around a bug. On

other platforms, this adapter will delegate to another function, `ftime`, which provides similar functionality but a different interface. In the particular case of Emacs' `gettimeofday` adapter, the code was rated 0, indicating only a partial negative impact on some parts of the function. The alternative to an adapter would be to introduce variability at every call site of the respective function. This would lead to more annotated code throughout the system compared to the adapter, which encapsulates the variability in a single location. In general, I learned that adapters benefit overall code quality as they protect the rest of the system's source code from the encapsulated variability.

Observation 6: *Optional Feature Stub.* We found a number of functions doing nothing if a particular optional feature is not present. I call this pattern the Optional Feature Stub. The unit tests for Libxml2, described in Observation 1, are one example. As a further example, Emacs contains the function `apply_xft_settings`, whose purpose is to initialize the font drawing library libXft. I show an excerpt of this function in Listing 4.9. Lines 6 and 113 in this listing illustrate that the whole function body is enclosed in an annotation. Similar to the Adapter pattern, this annotation relieves the rest of the system from having to concern itself with the variability of libXft support. If libXft support enabled, a call to this function will perform the necessary work. Otherwise, it simply does nothing. Although Optional Feature Stubs have a LOAC ratio of almost 100 %, most samples that followed this pattern were rated −1. In particular, I argue that Optional Feature Stubs are beneficial, because callers of the stub remain oblivious of a particular feature being present or not. Hence, I learned that the Optional Feature Stub actually reduces variability-related complexity and consider them as beneficial patterns rather than being smells.

Observation 7: *Featurized God Function.* Especially in Vim, we found many functions with more than 1000 LOC (e. g., samples 2, 3, 4). Besides their sheer length, these functions contained a lot of variable code, usually annotated by a vast amount of different feature constants. For instance, function `win_line` (Vim, sample 2, 2153 LOC) contains 23 feature locations related to multibyte strings, 22 feature locations for right-to-left writing systems, and 16 feature locations related to syntax highlighting, among others. In reference to the anti-pattern God Class [305], I call these functions Featurized God Functions, because they encompass a huge degree of variability. Although samples from other systems constitute long functions as well, the samples indicate that Vim is especially prone to this kind of function. As a result, I learned that too much variability related to too many features is a common pattern for the Annotation Bundle smell as it impedes understanding of data and control flow.

Observation 8: *Other platform and library variability.* Generally, the samples revealed platform- and library-related differences as a frequent source of variability. For instance, differences in representing path names on Windows and Unix-like operating systems are a reason for variable code in Lynx (samples 8, 9, and 11), PHP (samples 4 and 8), and Vim (sample 15). Moreover, differences between alternative libraries or different versions of the same library foster the introduction of variability. For instance, Libxml2 can either use the iconv or the uconv library to convert text between different character encodings. Although the interfaces of

```
1  static void
2  apply_xft_settings (struct x_display_info *dpyinfo,
3                      int send_event_p,
4                      struct xsettings *settings)
5  {
6  #ifdef HAVE_XFT
7    FcPattern *pat;
8    struct xsettings oldsettings;
9    int changed = 0;
10
11   memset(&oldsettings, 0, sizeof (oldsettings));
12   pat = FcPatternCreate();
13   XftDefaultSubstitute(dpyinfo->display,
14                        XScreenNumberOfScreen (dpyinfo->screen),
15                        pat);
16   ... /* more code */
```

```
103  ... /* more code */
104      Vxft_settings
105        = make_formatted_string(buf, format,
106                                oldsettings.aa, oldsettings.hinting,
107                                oldsettings.rgba, oldsettings.lcdfilter,
108                                oldsettings.hintstyle, oldsettings.dpi);
109
110    }
111  else
112    FcPatternDestroy(pat);
113 #endif /* HAVE_XFT */
114 }
```

Listing 4.9: Optional Feature Stub in Emacs, file `src/xsettings.c`

iconv and uconv are similar, they are not identical. Hence, the Libxml2 function `xmlFindCharEncodingHandler` (sample 11) contains alternative feature code for each library. Portability has long been pointed as a major source of variability in C programs [343]. I learned that this still holds more than 25 years later.

4.7 Related Work

The work presented in this chapter is related to a large body of existing work, which can be divided into four topics. First, my catalog of variability-aware code smells is preceded by several other catalogs of code smells. I explain how the smells in my catalog differ from the existing ones. Second, others have studied the use of cpp directives, sometimes also identifying specific usage patterns. Third, my detection concept and its implementation, Skunk, builds on existing research into code smell detection. Finally, in the same way that code smells are related to refactoring, my variability-aware code smells complement the variant-preserving refactorings that have already been discussed in the literature.

Other code smell catalogs. Independently from my work, Apel et al. coined the term "variability smell" for code-smell-like problems in SPLs and present a catalog of fourteen such smells [13]. These smells which are based on the authors' own experience, as well as on reports from fellow researchers and practitioners. Compared to

the smells I present in Section 4.2, their smells are more general in that they include different kinds of problems in an SPL, such as overly complex *feature models (FMs)* or run-time overhead resulting from inadequately chosen variability mechanisms. By contrast, my smell catalog is restricted to the implementation level, and is based on established code smells known from object-oriented programming.

Schulze, with varying coauthors, provides details on the smell Duplicated Code in both, composition-based and annotation-based SPLs [324, 325, 323]. He found that disciplined annotations lead to more replication than undisciplined annotations [325]. Moreover, he observed more replication in composition-based SPLs than in annotation-based ones [323]. Schulze's work gives empirical evidence for the existence of one of the smells in my catalog, Inter-Feature Code Clones. Beyond this particular smell, my catalog also encompasses other smells.

Others proposed code smell catalogs for *Aspect-Oriented Programming (AOP)* [102, 260, 294, 346] and for *Delta-Oriented Programming (DOP)* [329]. For example, Schulze et al. discuss smells that arise from improper use of DOP constructs, such as overly complex application conditions of delta modules [329]. These catalogs are related to mine because both AOP and DOP can be used as variability mechanisms for SPLs. In contrast to AOP and DOP smells, variability-aware code smells are not limited to a single variability mechanism. Moreover, variability-aware code smells explicitly consider variability as the cause for smells. By constrast, AOP smells ignore this perspective because they focus on cross-cutting concerns and on the structure of pointcuts and advice.

Architectural smells (see Section 2.1) were originally proposed with object-oriented software systems in mind [110, 111] but they have also been explored in the context of SPLs [10]. In comparison to architectural smells, variability-aware code smells characterize issues that are more low-level. Thus, both kinds of smells complement each other.

Like my own smell catalog, all of the aforementioned code smell catalogs build on the foundational work on code smells and anti-patterns for object-oriented software by Fowler, Beck, Brown, and others [42, 107, 372, 165, 233]. These smell catalogs also apply in a product-line context because much of an SPL's code base is, in fact, not variable. My variability-aware code smells complement them in the cases where the improper use of a variability mechanism causes quality issues.

Patterns of preprocessor use. Many researchers investigated the use of the CPP and its potential negative impact on source code [343, 89, 204, 206, 327, 221, 243, 239, 239, 238, 241, 142, 298]. Specifically, Spencer and Collyer argue against the use of `#ifdef`s for building portable software [343]. Their argumentation is based on experiences with a single system but has not been validated on other systems or by other researchers or practitioners, as I have done. Ernst et al. empirically analyzed CPP usage patterns, but focus on potential problems of macro expansion and techniques to replace CPP usage [89]. By contrast, my code smells for annotation-based variability mechanisms center around conditional compilation, a different feature of the CPP. Moreover, neither Spencer and Collyer, nor Ernst et al. address the automatic detection of code anomalies. Next, Medeiros et al. investigated bugs related to preprocessor variability [239, 238, 241]. In particular, they found that developers

perceive variability-related bugs easier to introduce, harder to fix, and more critical than other bugs [238]. Other researchers studied the negative effects preprocessor discipline and nesting of `#ifdef` directives on program comprehension [327, 221, 243]. These studies underline the importance of my work, which investigates sources of complexity introduced by preprocessor variability. I evaluate the negative effects of annotation-based variability in more detail in Chapter 5. Other researchers have analyzed scattering, tangling, and nesting (as well as other properties) of `#ifdef` directives in highly configurable systems [204, 206, 142, 298]. Skunk is built on top of the tooling developed by Liebig and Hunsen and relies on some of the metrics they proposed [204, 206, 142]. But in contrast to my work, their analyses are statistical in nature, and do not discuss methods to detect concrete patterns of misuse. Furthermore, although the Skunk tool is limited annotation-based mechanisms, the concept variability-aware code smells also covers composition-base mechanisms.

Code smell detection. A large body of research has addressed the detection of code smells and anti-patterns in software using OOP, AOP and FOP, and I have summarized this work in Section 2.4.1 of in Chapter 2 of this thesis. The detection concept that was implemented in Skunk relies on metrics that are combined into detection strategies and compared to absolute and relative threshold values. As such, it builds on the work of Moha, Marinescu, and Lanza, who pioneered metrics-based detection for object-oriented code smells (e. g., [229, 228, 230, 226, 196, 257]).

Other work on code smell detection takes additional sources of information into accounts, such as historical changes and textual information extracted from identifiers and comments. Furthermore, machine learning has been used to improve detection results, for example, by learning the weights and thresholds of a detection strategy. This work, which I discuss in Section 2.4.3 and Section 2.4.2 of Chapter 2, is complementary to mine and could be used to improve the effectiveness of my detection approach in the future.

Skunk currently outputs its results in a text-based format. In future work, visualizations may help users explore these results more efficiently. I summarized corresponding work in Section 2.4.4 of Chapter 2.

In the present chapter, I propose the smell Inter-Feature Code Clones, which is an SPL-specific variant of *code clones.* An entire field of research is devoted to detecting *code clones*, and I have summarized this work in Section 3.6.2 of Chapter 3 of this thesis. It seems feasible that this work could be adapted to detect Inter-Feature Code Clones.

In summary, my code smell detection approach in Skunk builds on the foundations of metrics-based code smell detectors. The novelty is that Skunk detects variability-aware code smells instead of known object-oriented or aspect-oriented smells. Consequently, Skunk employs other metrics than existing code smell detectors and combines them into detection strategies in different ways. Skunk's precision might benefit from taking additional sources of information into account, such as code changes or traditional metrics related to the underlying host language. Moreover, machine learning techniques appear as a promising means to find better threshold values and to improve Skunk's detection formulas.

Variability-aware refactoring. Fowler et al. describe object-oriented code smells as indicators for source code that should be refactored [107]. My code smells can be used in the same way, but for the variable source code of an SPL. Following Fowler's and Beck's example, other smell catalogs also offer advice on how to refactor (and thus remove) each smell (e. g., [13, 260, 329]). In Chapter 6, I present variability-aware refactorings to remove Inter-Feature Code Clones in FOP code. Beyond that, I offer no refactoring advice in this thesis. Hence, my work is complementary to previous work on variability-preserving refactoring [330, 328, 174, 175], as well as recent advances in automating refactoring in the presence of cpp directives [205, 240].

4.8 Conclusion

Highly configurable software systems owe much of their configurability to variable source code structures. On the one hand, variability at the source code level opens up opportunities for innovative implementation patterns but on the other hand, it may lead to new kinds of design flaws. In this chapter, I have argued that to characterize these design flaws, we need a new kind of code smell, one that explicitly takes variability into account. As a solution, I have presented a catalog of six *variability-aware code smells*, which have been validated by means of a survey. Moreover, I developed a concept and tool support to detect variability-aware code smells automatically. I applied this concept to five highly configurable software systems, and manually inspected the detection results. The insights gained from this inspection indicate (a) that metrics-based detection of variability-aware code smells is feasible, (b) that variability-aware code smells occur *in the wild,* and (c) that variability-aware code smells may constitute problems for understanding, maintenance, and evolution.

Based on the findings in this chapter, I answer $\mathbf{RQ_T\,1}$ of this thesis – *How do variability mechanisms affect the code smell concept?* – as follows: Regarding subquestion $\mathbf{RQ_T\,1.1}$, I showed that certain themes from established code smells, such as replication or overly strong centralization, also occur at the level of variability. By transferring these themes to annotation-based and composition-based variability mechanisms, it is possible to characterize a new class of design flaws. As indicators of these flaws, I propose the concept of *variability-aware code smells.* Regarding $\mathbf{RQ_T\,1.2}$, my evaluation results indicate that variability-aware code smells actually occur in real-world software and can negatively affect the quality of the source code. Additionally, I found that variability-code smells, like traditional code smells, are only indicators of potential problems, not certain proof. In particular, I found evidence that certain instances of variability-aware code smells are either harmless, for example, because they are easy to understand, or even beneficial, because they encapsulate variability in such a way that the complexity in other parts of the code base is reduced. This evidence is especially interesting because it furthers our understanding of how developers use variability mechanisms in practice. Moreover, it can guide future research into best practices of implementing variability – practices that could form a catalog of *variability-aware design patterns.* Regarding $\mathbf{RQ_T\,1.3}$, I demonstrated that a metrics-based approach to detect variability-aware code smells is feasible, even though detection precision should be improved in the future.

In the next chapter, I study one of my variability-aware code smells, the Annotation Bundle, in greater detail. In particular, I focus on the individual metrics that make up the current detection formula for this smell. Moreover, I investigate an aspect of maintainability which the evaluations in the present chapter have not considered: change-proneness.

5. How Preprocessor Annotations (Do Not) Affect Maintainability

This chapter is based on and shares material with the GPCE'17 paper "How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness" [98]. The study described in this paper was redesigned and significantly extended under the guidance of Sven Apel and Sandro Schulze, leading to the revised study presented in this chapter.

In the case study described in the previous chapter, I focused on one variability-aware code smell in particular, the Annotation Bundle. By applying my detection approach to five software systems, I demonstrated that Annotation Bundles occur frequently in real-world source code and that at least some Annotation Bundles negatively affect program comprehension. These properties – frequent occurrence "in the wild" and having negative effects – are the two most important characteristics of a code smell. Thus, the case study lends support to the hypothesis that the Annotation Bundle is indeed a code smell.

Despite the encouraging findings, the previous chapter leaves two points open. First, it showed that my detection approach for Annotation Bundles lacks precision. While the detection formula may be constructed from the right metrics, the combination of those metrics or the metrics are likely not optimal. Second, program comprehension is only one aspect of software development that may be affected by smells but other aspects should also be considered. For example, does the Annotation Bundle also affect fault proneness or maintenance effort? To address these points, I conducted a follow-up study, which I describe in this chapter.

In this chapter, I analyze how the use of C preprocessor directives in functions relates to *code change frequency* as well as *code churn,* which are evolutionary metrics that have been shown to be associated with increased fault proneness and maintenance effort [84, 123, 86, 255, 338]. This analysis addresses the points left open by the previous chapter as follows: First, it avoids the problems of the imprecise detection

formula for Annotation Bundles because each metric (e. g., the number of feature locations) is analyzed individually, without making assumptions about potentially harmful combinations of metrics. Thus, preprocessor use is considered at its most basic level. The results of this analysis should indicate how a better detection formula can be constructed. Second, as the focus shifts from program comprehension to maintainability, I can explore whether certain patterns of cpp usage have any further negative effects.

I have mined the data for my study from the version control repositories of twenty open-source systems written in C. Statistical analyses of the data indicate that functions with cpp directives are changed more frequently and more extensively than other functions. However, after accounting for confounding factors (size, age, time since recent changes, and number of previous changes), the differences are small and inconsistent across different subject systems. I qualitatively inspected the change histories of a selection of change-prone and stable functions to determine the role that cpp use may have played in their respective change proneness or stability. I conclude that the presence of preprocessor directives sometimes (but not always) predicts change-prone functions but other indicators, such as the use of comments to structure code in a long function, must be considered as well. In particular, this chapter contributes the following:

- Quantitative and qualitative insights into the relationship between fine-grained use of cpp directives and change proneness.
- A methodology and publicly available tool support to mine fine-grained information on cpp use and changes from version control repositories.

5.1 Research Questions

My study is driven by two objectives that give rise to three research questions. The first objective is to find out whether the presence of cpp directives affects maintenance effort. This objective leads to the following research question:

> ***RQ 1:*** *Is code containing* cpp *directives harder to maintain than code without* cpp *directives?*

As stated by previous studies, cpp directives can be used in various ways, which lead to many different shapes of feature code [204, 298]. To reflect these shapes in my study, I differentiate between the following aspects of cpp use: (1) the number of preprocessor directives, (2) the number of macros controlling those directives, (3) their nesting depth, (4) the use of negation (e. g., the use of `#ifndef` or `#else` directives), and (5) the proportion of code enclosed in preprocessor directives. I reduce these aspects to binary properties when answering RQ 1 and look at each one in isolation. For example, regarding aspect (3), I test whether code with nested preprocessor directives is more change-prone than code without nested directives.

Several studies found that many metrics (e. g., the object-oriented coupling metric LCOM1) correlate with the size of a piece of code [87, 390, 124]. Size, in turn,

correlates with important maintenance aspects, such as change and fault proneness [208, 273, 123, 170, 338]. Together, these findings suggest that size may also be a confounding factor in my study. To determine whether this is the case, I ask the following research question:

> ***RQ 2:*** *Does the presence of* CPP *directives relate to the size of a piece of code?*

My second objective is to understand the impact of each aspect of CPP use on maintainability when all aspects are studied in combination. For example, does the number of preprocessor directives affect maintainability more strongly than the amount of nesting? To control for possible confounding factors, I consider not only the size of piece of code, but also its age, the time since the last change, and the number of previous changes. This way, I can also determine whether the size of a piece of code, for example, has a stronger effect on maintainability than the number of preprocessor directives. To address my second objective, I formulate the following research question:

> ***RQ 3:*** *Considering all aspects of* CPP *use in combination and in the context of possible confounding factors, what is the independent effect of each aspect on maintainability?*

5.2 Methodology

In this section, I describe the design of my study. In particular, I present my operationalization of maintainability and CPP use, state my research hypotheses, introduce the analyzed software systems, and explain my data collection process.

5.2.1 Measuring Maintainability

To answer my research questions and achieve the objectives stated in Section 5.1, I need a measure for maintainability. Unfortunately, a direct measurement of maintainability requires (controlled) experiments, similar to the ones that Sjøberg et al. performed to study the effect of object-oriented code smells on maintenance effort [337, 338]. Although such experiments are highly valuable, they limit the number of systems and the amount of code that can be investigated. Hence, I use *change proneness* as a proxy for maintenance effort and mine my data from version control repositories. Change proneness comes in two flavors: the number of changes (e. g., used by Khomh et al. [169, 170]) and the extent of changes (a. k. a. *code churn*; e. g., used by Romano and Pinzger [308]). While the former is a good predictor of later defects (i. e., fault proneness) [84, 123], the latter correlates with the effort of developers (i. e., hours spent) when performing maintenance tasks [86, 255, 338]. In my study, I consider both measures because both fault proneness and the number of hours spent are important performance figures that describe the quality and cost of a software project.

The level of granularity of my analysis is crucial. As CPP annotations can be used at a fine grain, a similarly fine-grained analysis is required to investigate possible effects. Hence, my analysis works at the function level: I measure annotation use and changes for each function individually.

5.2.2 Variables

Next, I describe how I operationalized the use of preprocessor directives and change proneness in my study. To this end, I defined the variables listed in Table 5.1. Both preprocessor use and change proneness are modeled as binary (dichotomous) variables (see upper part of Table 5.1) and metric variables (see lower part of Table 5.1). In both parts of the table, the independent variables are listed above the horizontal rule and the dependent variables (*changed*, as well as *commits* and *lines*) are listed below. While the binary variables only indicate whether a property is present or not, the metric variables also account for different extents to which a property is present. For example, $cnd_{>0}$ indicates whether there are nested `#ifdef` directives, but *cnd* can additionally indicate that nesting occurs twice, three times, or even more often. Defining both binary and metric variables enables me to study the relationship between preprocessor use and change proneness from different perspectives and using different statistical analyses (see Section 5.3). The analyses based on binary variables are robust (e. g., regarding different probability distributions) and their results are easy to interpret. However, possible insights are rather coarse-grained. Hence, I perform complementary analyses based on metric variables, whose results are harder to interpret but also offer richer insights.

Binary Variables	
$fl_{>0}$	*true* if the function contains at least one `#ifdef` directive; *false* otherwise
$fc_{>1}$	*true* if the `#ifdef` directives in the function reference two or more feature constants; *false* otherwise
$cnd_{>0}$	*true* if at least one `#ifdef` in the function is nested; *false* otherwise
$neg_{>0}$	*true* if negation occurs at least once in the `#ifdef` directives in the function; *false* otherwise
changed	*true* if the function was changed at least once in a period of time; *false* otherwise
Metric Variables	
fl	Number of feature locations
fc	Number of distinct feature constants referenced by the `#ifdef` directives in the function
cnd	Number of times that the `#ifdef` directives in the function are nested
neg	Number of negations in the `#ifdef` directives in the function
$loac_{/loc}$	Proportion of lines of annotated code (lines of code that are enclosed in `#ifdef`s) to all lines of code, ignoring blank lines and comments
commits	Number of times the function was changed over a period of time
lines	Number of lines of code that were changed in the function over a period of time

Independent variables are listed above the horizontal rule in both parts of the table; dependent variables (*changed*, *commits*, and *lines*) are listed below.

Table 5.1: Independent and dependent variables

To capture different aspects of preprocessor use, I consider four properties: the number of feature constants, the number of feature locations, the use of nesting, and the use of negation. The corresponding binary variables are $fc_{>1}$, $\mathit{fl}_{>0}$, $cnd_{>0}$, and $neg_{>0}$, and their metric counterparts are fc, fl, cnd, and neg. Note that each preprocessor directive references at least one feature constant. Therefore, I deliberately chose *not one* feature constant as the threshold for $fc_{>1}$, but *two* feature constants. Otherwise, $fc_{>1}$ would have been identical to $\mathit{fl}_{>0}$.

As an additional aspect of preprocessor use, I model the proportion of annotated code in a function with the metric variable $loac_{/loc}$. In contrast to fc, fl, cnd, and neg, which are counts, $loac_{/loc}$ is a fraction. This made it difficult for me to define a threshold that would make the binary variant of $loac_{/loc}$ sufficiently different from the other four binary variables. Hence, $loac_{/loc}$ lacks a corresponding binary variable.

Change proneness as a binary property is modeled by a single variable, *changed*. It expresses whether a function was changed over a period of time or not. As metric variables, I define *commits*, which models the frequency with which a function is changed, and *lines*, which models the extent of those changes. All three variables are measured in the context of a certain period of time, which I call a *commit window*. In Section 5.2.5, I explain commit windows in detail, and also describe the metrics that correspond to the variables in Table 5.1 (see Table 5.5).

In addition to preprocessor use, I consider four factors that may also influence change proneness and, thus, may confound my analyses. In particular, I consider (1) the size of a function, (2) its age, (3) the number of previous changes, and (4) the time since the last change. The corresponding control variables are listed in Table 5.2; the respective metrics are explained in Section 5.2.5 (see Tables 5.5 and 5.6). Previous studies have shown that change proneness is partially explained by the size of a piece of code (e. g., [390, 338]). As I assume that larger pieces of code also tend to contain more preprocessor annotations than smaller ones, it is possible that differences in function size, and not differences in preprocessor use, are the true drivers of change proneness. In addition to size, I also suspect that newly written code, code that was frequently changed in the past, as well as recently changed code is more likely to change in the future than older code, infrequently changed code, or code that has not changed for a long time, respectively. By controlling for all four factors, I can differentiate between the unique effect of preprocessor use on change proneness, on the one hand, and the effects of these others factors, on the other hand.

Metric Control Variables	
loc	Source *lines of code* of the function, ignoring blank lines and comments
age	*Age* of the function
mrc	Time since the *most recent change* to the function
pc	Number of *previous changes* to the function

Table 5.2: Control variables

5.2.3 Null Hypotheses

In this section, I formulate null hypotheses arising from my research questions. For RQ 1 and RQ 3, which are my main research questions, the independent variables

model preprocessor use and the dependent variables model change proneness. In RQ 1, preprocessor use is modeled by binary variables; in RQ 3, it is modeled by metric variables and possible confounders are taken into account. For both research questions, I formulate three groups of null hypotheses that correspond to the three possible operationalizations of change proneness (*changed*, *commits*, and *lines*). This way, I explore all possible combinations of binary/metric independent variables with binary/metric dependent variables. In the auxiliary research question RQ 2, the independent variables model preprocessor use in metric form, and the dependent variable, *loc*, models size in metric form.

RQ 1. In the first question, I consider preprocessor use with the help of the binary variables $fc_{>1}$, $fl_{>0}$, $cnd_{>0}$, and $neg_{>0}$. For the first group of null hypotheses, I model change proneness with the binary variable *changed*. Considering preprocessor use and change proneness in this fashion allows me to investigate how the presence (or absence) of the corresponding aspect of preprocessor use in a function affects the *likelihood* that this function undergoes a change. To this end, I formulate four null hypotheses, one for each independent variable:

> ***H*$_0$ 1.1 ($fl_{>0}$):** *Functions containing* at least one preprocessor directive *are* just as likely to be changed *as functions* without any preprocessor directives.

> ***H*$_0$ 1.1 ($fc_{>1}$):** *Functions containing preprocessor directives that reference* two or more feature constants *are* just as likely to be changed *as functions in which* fewer feature constants *are referenced.*

> ***H*$_0$ 1.1 ($cnd_{>0}$):** *Functions containing* at least one nested preprocessor directive *are* just as likely to be changed *as functions* without any nested preprocessor directives.

> ***H*$_0$ 1.1 ($neg_{>0}$):** *Functions containing preprocessor directives that use* negation at least once *are* just as likely to be changed *as functions* without preprocessor directives that use negation.

For the two following groups of null hypotheses, change proneness is modeled by the metric variables *commits* and *lines*, respectively. Thus, I investigate whether the presence or absence of a certain aspect of preprocessor use in a function is related to the *frequency* or *profundity* with which the function is changed. For brevity, I only explicitly state the two null hypotheses for $fl_{>0}$; the others are formulated accordingly:

> ***H*$_0$ 1.2 ($fl_{>0}$):** *Functions containing* at least one preprocessor directive *are* changed just as frequently *as functions* without any preprocessor directives.

> ***H*$_0$ 1.3 ($fl_{>0}$):** *Functions containing* at least one preprocessor directive *are* changed just as profoundly *as functions* without any preprocessor directives.

RQ 2. Analyzing correlations between preprocessor use and change proneness in isolation may produce misleading answers because possible confounding factors are ignored. The goal of RQ 2 is to test whether function size may be such a confounding factor. My independent variables in this question are the metric variables *fc*, *fl*, *cnd*, *neg*, and $loac_{/loc}$, and the dependent variable is always the metric variable *loc*. I formulate the following null hypothesis regarding the relationship between the number of `#ifdef` directives in a function (variable *fl*) to its size; as for RQ 1, there are corresponding hypotheses for the remaining independent variables (*fc*, *cnd*, *neg*, and $loac_{/loc}$), but I omit them for brevity.

> ***H_0 2 (fl):*** *The number of preprocessor directives in a function is unrelated to the size of the function.*

RQ 3. In case that H_0 2 is rejected, I turn to RQ 3. In this question, I investigate how different extents of different aspects of preprocessor use (e. g., number of `#ifdef`s, depth of nesting) relate to change proneness. Instead of looking at each aspect in isolation, as I do for RQ 1, I now consider all aspects of preprocessor use in combination. I model preprocessor use with the help of metric variables, namely *fc*, *cnd*, *neg*, and $loac_{/loc}$. As a further modification compared to RQ 1, I control for potentially confounding factors by including the control variables *loc*, *age*, *mrc*, and *pc* (see Table 5.2).

Note that *fl* is not included in the list of independent variables. The reason is that I found that *fl* strongly correlates with *fc*, which would have violated the assumptions of my statistical tests. See Section 5.3.3.1 for a detailed explanation.

Change proneness in RQ 3 is modeled by the binary variable *changed*, and by the metric variables *commits* and *lines*. In a nutshell, the goal of RQ 3 is to examine how much different extents of preprocessor use affect the *likelihood,* the *frequency,* and the *profundity* of changes to a function. To this end, formulate the following null hypotheses:

> ***H_0 3.1:*** *After accounting for differences in function size, age, number of previous changes, and time since the last change, the likelihood that a function is changed is unrelated to the extent of preprocessor use in that function.*

> ***H_0 3.2:*** *After accounting for differences in function size, age, number of previous changes, and time since the last change, the frequency of changes to a function is unrelated to the extent of preprocessor use in that function.*

> ***H_0 3.3:*** *After accounting for differences in function size, age, number of previous changes, and time since the last change, the profundity of changes to a function is unrelated to the extent of preprocessor use in that function.*

5.2.4 Subject Systems

I chose my subjects from a set of well-known systems used in previous studies of preprocessor usage [204, 142, 298]. This section describes my subjects and my selection criteria.

I used the following seven criteria to guide my selection.

1. well-known system, preferably used in previous studies of CPP usage
2. implemented in C (as CPP directives are frequent in C code)
3. systems from different application domains (to avoid bias)
4. open-source software (as I need access to the repository to extract change information)
5. GIT version control system (to be accessible by my tool infrastructure)
6. considerable development history (to minimize the effect of outliers in small datasets).

In addition to these criteria, I faced a number of practical and technical limitations that prevented me from including further systems that previous studies analyzed. One limitation was the scalability of my analysis infrastructure. Although my analysis scales to systems of almost 1 000 000 *SLOC (non-blank, non-comment source lines of code)* and development histories of well over 100 000 commits, LINUX and other popular open-source operating systems (e. g., FREE-, OPEN-, and NETBSD) proved to be too large. OPENSOLARIS would have been an interesting substitute in the operating systems domain, especially since it used to be a closed-source system. However, I found its GIT repository to be incomplete, preventing my tooling from reconstructing the evolution of OPENSOLARIS' code.

Other systems lacked a publicly accessible version control repository (e. g., MDNSRESPONDER, SENDMAIL), or they relied on a system other than GIT for version control (e. g., MPLAYER continues to use SUBVERSION). In a few of these cases, I found suitable replacements. For example, I replaced MPLAYER with MPV, an MPLAYER fork whose developers switched to GIT.

Finally, a few systems (e. g., GLIBC, VIRTUALBOX) contained highly uncommon C language constructs that triggered bugs in my analysis infrastructure. In the case of GLIBC, for example, I was able to fix those bugs, allowing me to include it in my study after all. However, for others, such as VIRTUALBOX, I was unable to do so, forcing me to exclude the system.

Based on my criteria and accounting for the aforementioned limitations, I selected twenty subject systems. The information about these systems is split across two tables, Table 5.3 and Table 5.4: In Table 5.3, I state the time period analyzed, the number of commits in that period, and the domain of each subject system. In Table 5.4, I report information about the size of each subject system, taking the latest version as the reference point. Specifically, I state the number of `.c` files, the number of function definitions, and, in parentheses, the percentage of functions containing at least one preprocessor directive. Moreover, I report the SLOC comprised by function definitions and, in parentheses, the percentage of code that is feature code.

System	Period	Commits	Domain
Apache	1996–2019	31 104	Web server
Blender	2002–2019	84 456	3D graphics creation
BusyBox	1999–2019	16 128	Unix command line tool suite
Cherokee	2006–2018	5 585	Web server
Gimp	1997–2019	42 885	Image editor
glibc	1972–2019	34 420	GNU C standard library
gnuplot	1987–2019	10 681	Plotting command line tool
Gnumeric	1998–2019	23 384	Spreadsheet
libxml2	1998–2019	4 501	XML parser and toolkit
mpv	2001–2019	46 531	Media player
MySQL	2000–2019	139 726	Database management system
OpenLDAP	1998–2018	22 256	LDAP directory service
OpenVPN	2005–2019	2 162	Secure network communication
php	1999–2019	110 684	Programming language interpreter
Pidgin	2000–2017	38 328	Instant messaging client
qemu	2003–2019	66 752	Hypervisor
PostgreSQL	1996–2019	46 343	Database management system
SQLite	2000–2019	20 176	Database management system
Subversion	2000–2019	59 020	Revision control
Vim	2004–2019	9 338	Text editor

Table 5.3: Subject systems: Development periods and domains

Note that the mean percentage of feature code in my subject systems is 6.7 %, much lower than the 24 % reported by Hunsen et al. [142]. The difference is rooted in the way I measure feature code. Hunsen et al. work on the file level and thus count every line of code belonging to a function, data type, and variable definition as feature code if the line is enclosed in an `#ifdef` directive. By contrast, I measure feature code on the function level, and I ignore data types and global variables. More importantly, I only consider a line as feature code if the opening `#ifdef` occurs *inside* the function body, not if it is placed before the start of the function definition. As a result, the percentages of feature code I report are lower than those reported in related work.

As can be seen in Table 5.3 and Table 5.4, my subjects cover a great variety of domains, sizes, and extent of preprocessor use. For example, the domains range from Web servers (Apache) to spreadsheet applications (Gnumeric), and the sizes range from just above 50 000 SLOC (in Cherokee) to almost 1 million SLOC (in qemu). Even though most of my subjects are purely open-source, I was able to include one formerly closed-source system, the 3D graphics software Blender. In summary, my subject selection allows me to gain robust insights that are representative for C software systems of many different domains, sizes, and extents of configurability.

5.2.5 Data Collection

In this section, I describe which data I collected from my subject systems and how. My data collection process comprises four tasks:

1. Identify all relevant commits from the subject's repository.

2. Visit all commits in level order and periodically take *snapshots* to extract metrics of all functions (e. g., the number of `#ifdef`s), yielding the preprocessor metrics listed in Table 5.5.

3. Record changes to functions (e. g., number of changes, number of lines added or deleted), yielding the change metrics listed in Table 5.6.

4. Combine the preprocessor and the change metrics and aggregate them into *commit windows*.

Overview. I describe each data collection task in detail in the following paragraphs, but to give an initial overview, I illustrate the process with help of the example in Figure 5.1. The example focuses on three functions in a fictitious C system and follows their evolution over 200 commits. The functions are called `main`, `print_help`, and `usage_error`, and all are located in file `src.c`. The commits are named c_1–c_{200} and belong to two branches, *master* and *refactor*. My analysis starts at the root commit, c_1, where I take a snapshot to determine which functions exist and what their *preprocessor metrics* are.

System	Files	Functions	Functions w/ `#ifdefs`	SLOC	Feature Code
Apache	330	5 710	(7 %)	161 489	(4 %)
Blender	1 446	35 729	(3 %)	804 927	(4 %)
BusyBox	656	4 871	(14 %)	144 071	(11 %)
Cherokee	168	1 824	(8 %)	51 738	(9 %)
Gimp	1 648	21 632	(2 %)	613 383	(1 %)
glibc	6 403	12 766	(9 %)	350 113	(7 %)
gnuplot	88	2 207	(11 %)	81 741	(8 %)
Gnumeric	348	10 404	(2 %)	242 547	(1 %)
libxml2	109	5 627	(25 %)	178 470	(19 %)
mpv	305	4 876	(2 %)	94 373	(1 %)
MySQL	212	3 410	(9 %)	72 999	(7 %)
OpenLDAP	552	5 722	(12 %)	247 759	(6 %)
OpenVPN	105	2 356	(15 %)	64 649	(12 %)
php	777	10 944	(12 %)	365 639	(8 %)
Pidgin	496	10 998	(3 %)	227 439	(3 %)
qemu	2 324	50 821	(3 %)	936 743	(3 %)
PostgreSQL	1 220	20 446	(5 %)	681 039	(2 %)
SQLite	320	8 631	(10 %)	216 238	(8 %)
Subversion	587	11 613	(3 %)	367 710	(2 %)
Vim	141	7 789	(25 %)	289 471	(18 %)
Mean	912	11 919	(9.0 %)	309 627	(6.7 %)

Table 5.4: Subject systems: Sizes and extents of configurability

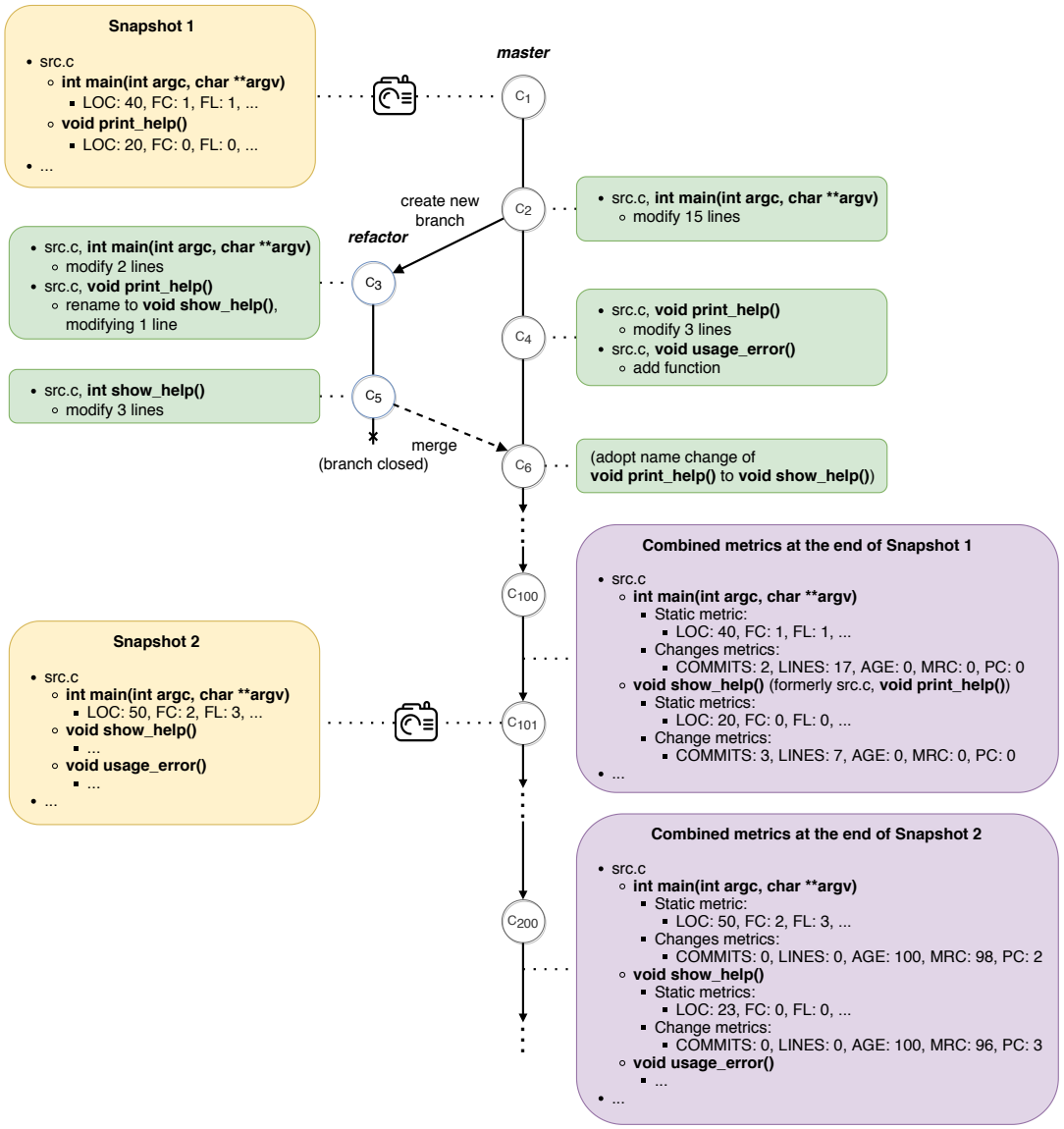


Figure 5.1: Illustration of the data collection process

The preprocessor metrics (see Table 5.5) capture the length of a function and five aspects of its use of CPP annotations, namely, the number of feature locations in its body (FL), the number of referenced feature constants (FC), the extent of nesting (CND), the number of negation (NEG), and the lines of code that are subject to compile-time configurability (LOAC). The last metric, $\text{LOAC}_{/\text{LOC}}$, is a derived metric, computed from LOAC and LOC. Except for NEG, all of these metrics already appeared in the same or similar form in the previous chapter as the atomic metrics of the ANNOTATION BUNDLE detection formula (see Table 4.1 on Page 73 in Section 4.4.1). They are also included in the present chapter because they operationalize the various aspects of CPP use on the function level well. In addition to these known metrics, I take NEG into account, which is an operationalization of a notable aspect of CPP use that was previously not considered: negation. In Table 5.5, I explain how the metrics computed and note how they are similar to or different from the metrics introduced in the previous chapter.

Metric	Description
LOC	Source lines of code of the function (measured in the same way as in Table 4.1 in the previous chapter, i. e., ignoring blank lines and comments).
FL	Number of feature locations (same as in Table 4.1).
FC	Number of feature constants referenced by the feature locations in the function. Constants referenced multiple times are counted only once. (FC is similar to the FC_{DUP} metric in Table 4.1, but in contrast to FC_{DUP}, FC *ignores* multiple references to the same constant.)
CND	Cumulative nesting depth of annotations (same as in Table 4.1).
NEG	The number of negations in the `#ifdef` directives in a function. Both `#ifndef X` and `#if !defined(X)` increase NEG by 1. An `#else` branch also increases NEG because `#if <expr> ... #else ... #endif` is expanded to `#if <expr> ... #endif` followed by `#if !<expr> ... #endif`. (This metric was *not* used in the previous chapter.)
LOAC	Source lines of code in all feature locations within the function (same as in Table 4.1).
$LOAC_{/LOC}$	Proportion of LOAC to all code in the function, i. e., LOAC÷LOC. (This metric already appeared in the previous chapter and was part of the detection formula for Annotation Bundles shown in Equation 4.1.)

Table 5.5: Preprocessor metrics

In Figure 5.1, the preprocessor metrics collected for *Snapshot 1* are shown in the yellow box in the upper left corner. They reveal that `src.c` contains two function definitions at revision c_1 (`main` and `print_help`), and that function `main` is forty lines long (LOC: 40), references one feature constant (FC: 1), and contains one `#ifdef` (FL: 1).

After taking this *Snapshot 1*, I visit the following commits to analyze how the identified functions change. Commits are visited in level order to ensure that every time a commit is analyzed, all changes from its ancestor commits are already available. In Figure 5.1, the changes of a commit are denoted by the green boxes that are placed next to a commit. For example, c_2 in the *master* branch changes fifteen lines in function `main`, and a second change occurs in the *refactor* branch, in c_3. In addition to changing `main`, c_3 also renames `print_help` to `show_help`. Concurrently to c_3, commit c_4 from the *master* branch also changes `print_help`, but without renaming it. Since my analysis is aware of branches, it keeps track of the different names under which this function is known in different branches. At c_6, when the *refactor* branch is merged back into *master*, my analysis combines the data from both branches and resolves the name change of `print_help`. Thus, from c_6 onward,

Metric	Description
AGE	*Age* of the function, i. e., number of commits since the function was created, relative to the start of the snapsnot.
MRC	Time since the *most recent change*, i. e., number of commits since the function was most recently changed, relative to the start of the snapsnot.
PC	Number of *previous changes*, i. e., number of commits that changed the function before the start of the snapshot.
COMMITS	The number of commits within the snapshot that have modified the function definition.
LINES	The number of lines added plus the number of lines removed, accumulated over the course of the snapshot.

Table 5.6: Change metrics extracted for each function

the function is known as `show_help` in the *master* branch. At the same time, my analysis remembers that the preprocessor metrics of this functions must be retrieved using the function's old name, `print_help`.

My change analysis continues in the described fashion until a certain number of commits has been visited. At this point, the change information is aggregated and combined with the preprocessor metrics from the snapshot, yielding a set of combined metrics (see purple boxes at the bottom right of Figure 5.1). For example, the combined metrics reveal that function `main` in `src.c` was forty lines long (LOC: 40) and contained one `#ifdef` (FL: 1) at the start of *Snapshot 1*. Moreover, the combined metrics reveal that `main` was modified twice between c_1 and c_{100} (COMMITS: 2) and that seventeen lines of code were changed (LINES: 17). Note that function `usage_error` is included only in the combined metrics of *Snapshot 2* but not of *Snapshot 1*. The reason is that `usage_error` was added in commit c_4 and thus was not present when *Snapshot 1* was taken. However, listing `usage_error` in *Snapshot 2* for the first time does not mean that `usage_error`'s earliest changes are entirely ignored. Instead, these changes are included in a different form, with the help of metrics that model a function's past evolution. I explain these metrics next.

In addition to COMMITS and LINES (see lower part of Table 5.6), my change analysis extracts three more change-related metrics: AGE, MRC, and PC (see upper part of Table 5.6). These metrics are computed based on information from previous snapshots, and I include them as additional control variables in my statistical analyses (see Section 5.3.3) to gain more robust results. As an example, consider the combined metrics of function `main` in *Snapshot 2* (see purple box at the bottom right of Figure 5.1). *Snapshot 2* starts at commit c_{101} and at this point, `main` is 100 commits old (AGE: 100), was previously changed twice (PC: 2), and the most recent change, which was at commit c_3, happened 98 commits ago (MRC: 98).

After this illustration of the overall data collection process, I now explain the four data collection tasks listed at the beginning of this section in detail.

Identifying relevant commits. As the first step in my data collection process, I classify all commits in the respective GIT repository as either *relevant* or irrelevant. A commit is relevant if it modifies at least one `.c` file, as this is the kind of file where functions in C are defined. Commits that only modify header files (which contain function declarations, not definitions) or other kinds of files, such as change logs or shell scripts, are marked as irrelevant. During change analysis, the irrelevant commits are only used to retrace branching and merging activities, but otherwise ignored.

Taking snapshots. To gather the preprocessor source metrics, I visit the commits in level order and take a *snapshot* of the entire system every time a fixed number of *relevant* commits has been visited. Taking a snapshot involves four steps, (1) checkout, (2) preprocessing, (3) gathering preprocessor metrics, and finally, (4) storing the metrics.

The first step is to check out the respective revision and copy all `.c` files to a dedicated directory. In the second step, preprocessing, the files are converted to SrcML [57], an XML representation that is easier to parse than raw, unprocessed C code. Using the SrcML representation, comments and empty lines are removed. Moreover, `#ifdef`s are normalized to ease subsequent analysis. For instance, `#ifndef FEATURE` is transformed to the equivalent `#if !defined(FEATURE)`. Third, I parse all SrcML files to identify the function definitions and collect their preprocessor metrics (see Table 5.5).

Finally, when all the metrics are collected, I store them in a map that is associated with the revision at which the checkout was performed. As keys in this map, I use so-called *function ids*, which I compose of a function's name, return type, parameter list, and the name of the file in which the function is defined. In later phases of my analysis, function ids allow me to unambiguously relate the preprocessor metrics of a function with the changes to that function, which I describe next.

Recording changes. In a nutshell, I record changes by inspecting the diffs of every commit to the previous revision and determining for every added and deleted line which C function, if any, is affected. Based on this data, I compute the change metrics listed in Table 5.6.

In Listing 5.1, I show an excerpt of the diff of commit 984cf003 in OPENVPN to illustrate how I identify changes. Unnecessary portions of the diff (e. g., on Lines 4 and 9) have been omitted. The diff consists of a block of metadata (see Lines 1–4), followed by a sequence of diffs for individual files. In the example, the block on Lines 5–9 contains the diff for the file `Makefile.am`, and the block on Lines 10–41 contains the diff for the file `manage.c`. Each file's diff consists of a four line header followed by a sequence of edits, which, in turn, convey the information about added and deleted lines (see, e. g., Lines 14–17 and Lines 18–21 in the example). In particular, the first line of an edit reveals the location of the edit, and the body encompasses one or more added and deleted lines, which are optionally surrounded by context lines. Deleted lines are prefixed with a "`-`", added lines are prefixed with a "`+`", and context lines are prefixed with a single space character (see Lines 39, 40, and 41, respectively). The context lines are included for technical reasons but since they do not represent changes they are irrelevant for my analysis.

```
commit 984cf0036c882c4fada83448aaa37bbd5ebb8130
Author: james <james@e7ae566f-a301-0410-adde-c780ea21d3b5>
Date:   Thu Oct 20 05:58:08 2005 +0000
...
diff --git a/Makefile.am b/Makefile.am
index 6519f2aa..424b167d 100644
--- a/Makefile.am
+++ b/Makefile.am
...
diff --git a/manage.c b/manage.c
index 7be67fd0..d17d9ce7 100644
--- a/manage.c
+++ b/manage.c
@@ -41,2 +41,3 @@
 #include "integer.h"
+#include "misc.h"
 #include "manage.h"
@@ -76,2 +77,3 @@
   msg (M_CLIENT, "net : (Windows only) Show info and routing table.");
+  msg (M_CLIENT, "ok type : Enter confirmation for NEED-OK request.");
   msg (M_CLIENT, "password type : Enter password for a queried password.");
@@ -522,0 +529,9 @@
+static void
+man_query_need_ok(struct management *man, const char *type)
+{
+  const bool needed = ((man->connection.up_query_mode == UP_QUERY_NEED_OK)
+                       && man->connection.up_query_type);
+  man_query_user_pass (man, type, "ok", needed, "ok-confirmation",
+                       man->connection.up_query.password, USER_PASS_LEN);
+}
+
@@ -1727,9 +1745,9 @@
 bool
 management_query_user_pass(struct management *man,
                            struct user_pass *up,
                            const char *type,
-                           const bool password_only)
+                           const unsigned int flags)
 {
   struct gc_arena gc = gc_new ();
   bool ret = false;
```

Listing 5.1: Excerpt of the diff of commit 984cf003 in OpenVPN

When extracting function changes from the diff in Listing 5.1, I ignore the first file-level diff because it does not affect a `.c` file; only the second diff is important. To analyze this diff further, I first check out both the previous and the current revision of the modified file, `manage.c`, and extract the start and end points of every C function. An excerpt of the extracted information is shown in Table 5.7, with the upper half of the table listing the function locations before the commit and the lower half listing the function locations after the commit. Next, I determine for each edit whether it overlaps these function locations. Specifically, I check whether deleted lines overlap with function locations before the commit (upper half of Table 5.7) and whether added lines overlap with function locations after the commit (lower

Revision	Start	End	Function Signature
f7868716	60	87	`static void man_help()`
	...	...	...
	1727	1797	`bool management_query_user_pass(struct management *man, struct user_pass *up, const char *type, const bool password_only)`
	...	...	...
984cf003	61	89	`static void man_help()`
	529	536	`static void man_query_need_ok(struct management *man, const char *type)`
	...	...	...
	1745	1837	`bool management_query_user_pass(struct management *man, struct user_pass *up, const char *type, const unsigned int flags)`
	...	...	...

Table 5.7: Function locations in `manage.c` extracted for the diff in Listing 5.1

half of Table 5.7). Depending on the overlap, I decide whether an edit (1) modifies the body of a function, (2) adds a function, (3) deletes a function, or (4) changes a function's id (by changing its name, return type, its parameter list, or by moving the function to another file).

Using the edits on Lines 14–41 in Listing 5.1 and aligning them with the function locations in Table 5.7, I next illustrate how I determine the type of change for an edit. The first edit (see Lines 14–17) adds one line of code to the file `manage.c` at position 42, but according to Table 5.7, no C function is defined at this position. Consequently, this edit is skipped. The second edit (see Lines 18–21) adds one line of code at position 78, which is within the body of the function `man_help()` (see Table 5.7). Hence, this edit is classified as a modification of function `man_help` adding a single line. The third edit (see Lines 18–21) adds nine lines of code. According to Table 5.7, this edit completely covers the location of the function `man_query_need_ok(...)` in the new revision of `manage.c`. Moreover, the upper half of Table 5.7 reveals that this function did not exist in the previous revision. Hence, this edit is classified as a function addition. Much like I detected this function addition, I also detect function deletions (not shown in the example). In particular, a function deletion is an edit in which the deleted lines fully cover a function definition, relative to the function's location before the commit. Coming back to the example, the fourth and final edit (see Lines 32–42 in Listing 5.1) illustrates how I handle changes to a function's id. The edit deletes one line and adds one line. According to Table 5.7, the deletion overlaps with the old location of the function `management_query_user_pass`, and the addition overlaps with the new location of a function of the same name, but with a different last parameter. Since the parameters are part of the function id, I treat this change separately. In particular, I classify this edit as a function id change of `management_query_user_pass` that modified two lines. After analyzing all four edits to `manage.c`, my analysis has recorded three changes. The corresponding data is listed in Table 5.8.

Hash	Old & New Function Id	Mod. Type	COM-MITS	LINES
984cf003	main.c: `static void man_help()` *(no id change)*	mod	+1	+1
984cf003	main.c: `static void man_query_need_ok(struct management *man, const char *type)` *(no id change)*	add	±0	±0
984cf003	main.c: `bool management_query_user_pass( struct management *man, struct user_pass *up, const char *type, const bool password_only)` main.c: `bool management_query_user_pass( struct management *man, struct user_pass *up, const char *type, const unsigned int flags)`	id change	+1	+2

Table 5.8: Function changes extracted from the diff in Listing 5.1

Apart from changing a function's parameter list, an edit can also change a function's id by changing the return type or the function name. Moreover, functions are sometimes moved to other files, which is difficult to track because GIT reports moved functions as two unrelated edits, one that deletes a function and one that adds a function. I handle all of these possible function id changes in a post-processing step, which takes place after I have seen all edits that belong to a particular commit. The basic idea is to inspect each edit that deleted a function and to try to match it to an edit that added the same or a similar function. If a match is found, I fold the edits into a function id change. The matching process has two phases. In the first phase, I match deletions to additions if the function names are the same. This resolves deletions and additions that result from moving a function definition to another file, possibly also changing the return type or parameter list. In the second phase, I match the remaining deletions and additions based on the similarity of the full function definitions (i. e., taking both function signature and body into account). Specifically, I calculate the Levenshtein distance (the number of single-character edits) between each deleted and added function, and based on this distance, compute the percentage of characters that the functions have in common. At 60 % or more, I treat two functions as similar; otherwise, as dissimilar. (I decided on the 60 % threshold after manually inspecting numerous changes involving potential renames in several subject systems, including APACHE and OPENLDAP. However, this process was no rigorous scientific experiment, and so the choice of 60 % as the threshold remains a threat to the internal validity of my study.) Then, starting with the pair with the highest similarity, I proceed to fold pairs of deleting and adding edits into function id changes. Finally, I go through the remaining deletions and additions, for which no match was found, and record them as actual deletions or additions. As a result of tracking function id changes so precisely, I am able to relate the preprocessor metrics of each function, which are taken at the start of a snapshot, to all subsequent modifications, even if the function's name, return type, or parameter list change, or if the function is moved to another file.

Formation of commit windows. My initial experiments revealed that the small number of commits between my snapshots prevents me from observing functions

that undergo a large number of commits and heavy changes. This made it difficult to distinguish between functions that are truly change-prone and those that are not. As a solution, I use a sliding window technique in which I aggregate the changes of several consecutive snapshots into a *commit window*. Based on my initial experiments, I set the window size to be five snapshots (i. e., 500 commits). I found this size to be sufficiently large to make change-prone and stable functions easily distinguishable, while, at the same time, keeping analysis runtimes within feasible limits.

Commit windows are formed as follows: For the first commit window, I use the combined preprocessor metrics and change information of *Snapshot 1* as the basis and extend this data with the change information (but not the preprocessor metrics) from *Snapshot 2, 3, 4,* and *5*. When adding the change information, I include only changes related to functions that are present in *Snapshot 1* but omit changes to functions that were added later on. However, the omitted changes are not lost: they will be included in future commit windows.

The second commit window is formed in the same way as the first, but starting at *Snapshot 2*, and the aggregation process continues until no more snapshots are left. Compared to an individual snapshot, the frequency and amount of changes captured in a commit window increase substantially. In summary, my sliding window technique produces data that reveal truly change-prone functions more clearly and allows me to obtain more robust results.

5.3 Statistical Analyses

In this section, I explain the statistical analyses I employed to test my null hypotheses.

I compute several statistics to investigate if and how much preprocessor use affects change proneness. In these statistics, the independent variables basically capture different aspects of preprocessor use. The dependent variable in turn, a. k. a. the outcome, captures change proneness. When testing a null hypothesis, I always follow a two-step process: First, I test whether an independent variable has a statistically significant effect on the dependent variable. Statistical significance refers to the degree of confidence that an observed effect is *not* merely the result of chance. It is indicated by the probability value or p value for short. The smaller the p value, the higher the significance. In my experiments, I will regard an independent variable as having a significant effect if $p < .01$ in the majority of subject systems. Thus, the likelihood that my observations result from random effects is less than 1 %.

Proving a statistically significant effect is enough to reject a null hypothesis, but it does not reveal whether the effect is large or small, nor whether it is positive or negative. Therefore, in case I find a significant effect, I take a second step and compute the effect size. If the effect size is positive, more preprocessor use corresponds to a higher change proneness; if it is negative, more preprocessor use corresponds to a lower change proneness. The absolute value of the effect size, in turn, indicates how strong the correspondence is.

In what follows, I describe in detail which statistical methods I apply to answer my research questions and how I define the corresponding independent and dependent variables. Moreover, I explain how to interpret the results of each method.

5.3.1 Answering RQ 1: Binary Classification with Binary and Metric Outcomes

I use a binary classification scheme to test H_0 1.1, H_0 1.2, and H_0 1.3. Basically, I combine the data of all commit windows for a system into one large dataset, and then split this dataset into an experimental group and a control group depending on the presence or absence of a certain aspect of preprocessor use. Functions that exhibit the aspect (e. g., contain at least one `#ifdef` directive), belong to the experimental group, the others to the control group. I then test whether the proportion of changed functions to unchanged functions differs between experimental and control group.

I create four pairs of experimental and control group, one for each of my four binary independent variables $fc_{>1}$, $fl_{>0}$, $cnd_{>0}$, and $neg_{>0}$ (see Table 5.1). To test H_0 1.1, which has a binary outcome, the dependent variable is *changed* (see Table 5.1). I use Fisher's exact test to check for statistically significant differences in the likelihood of changes between experimental and control group [104]. If there is a statistically significant effect, I report its size as an *odds ratio (OR)*. An OR is the factor by which the odds of the outcome being 1 are bigger in the experimental group than the respective odds in the control group. Thus, an OR of 1 corresponds to a neutral effect, values greater than 1 indicate positive effects, and values lower than 1 indicate negative effects. As an example, assume that the odds are 1:1 that a function from the control group is changed and that the OR is 2. Given this data, the odds that a function in the experimental group is changed can be calculated as $1{:}1 \times 2.0 = 2{:}1$. In terms of likelihoods, this would mean that the likelihood of a change is 50 % in the control group and about 67 % in the experimental group.

I perform Fisher's test on all twenty subjects, for all four independent variables. Additionally, I aggregate the subject-specific results into an overall result by computing a pooled p value and a pooled OR using the Mantel-Haenszel method [63].

I keep the same binary classification scheme to test H_0 1.2 and H_0 1.3 that I also use to test H_0 1.1, but I switch the dependent variables. Specifically, I consider the metric variable *commits* as the dependent variable and apply the *Mann-Whitney-U* test. The test will reveal if there is a statistically significant difference in changes frequencies in the experimental group compared to the control group. If the difference is significant, I compute the effect size in terms of *Cliff's delta* [54]. I proceed in the same way to test H_0 1.3, with the exception that *lines* is the dependent variable. To summarize the effect sizes over all subjects, I report the mean value and standard deviation of the Cliff's delta values of the significant correlations.

5.3.2 Answering RQ 2: Relation between Preprocessor Use and Function Size

In answering RQ 1, I only analyze preprocessor use as a binary property. Moreover, I study each aspect of preprocessor use in isolation. While this analysis is easy

to understand, it has two important limitations. First, the possible insights are rather coarse-grained because I learn nothing about the effects of other extents of preprocessor use. For example, are functions with two or three feature locations more change-prone than functions with just one feature location? Secondly, this analysis neglects possible confounding effects, in particular, the possible confounding effect of function size. For these reasons, I analyze the relationship between preprocessor use and function size in RQ 2 (see results in Section 5.4.4). To this end, I test whether the variables *fl*, *fc*, *cnd*, *neg*, and $loac_{/loc}$ correlate with *loc*. The statistical technique I use is the Spearman rank correlation coefficient [56], which indicates both statistical significance as well as effect size. I test H_0 2 based on the test outcomes. Based on the results, I can put the answer to RQ 1 into context and proceed to the more complex analyses in RQ 3.

5.3.3 Answering RQ 3: Different Extents of Preprocessor Use in Context

To answer RQ 3, I use regression analyses, which will reveal how different extents of preprocessor use affect change proneness, while, at the same time, allowing me to control for potential confounding effects. The basic idea of regression is to iteratively fit a formula to a dataset. The result is a regression model that describes the correlations between several independent variables and a dependent variable, called the outcome. Regression models are sometimes created to predict the outcome on another dataset, but I use them to deduce which correlations are statistically significant and what the effect sizes are.

5.3.3.1 Effect on the Likelihood of Changes

To test H_0 3.1, I use logistic regression, where the dependent variable (a. k. a. the outcome) has a binary type: the dependent variable is *true* if the outcome takes place and *false* if it does not. Given a set of independent variables, $X_1, X_2, \ldots, X_n$, logistic regression finds the parameters $\beta_1, \beta_2, \ldots, \beta_n$ that best fit the following formula:

$$\ln(\frac{p}{1-p}) = intercept + \beta_1 X_1 + \beta_2 X_2, \ldots, \beta_n X_n \tag{5.1}$$

In this formula, p is the probability that the dependent variable becomes *true*.[1] In the logistic regression models that I build to test H_0 3.1, the dependent variable is *changed*, and p is the probability that *changed* becomes *true*. I use R's implementation of logistic regression,[2] which reveals for each independent variable whether its effect on the outcome is statistically significant. If so, the effect size of an independent variable X_i is indicated by the corresponding coefficient β_i. The sign of the coefficient indicates whether the effect is positive or negative. A positive effect means that the outcome will become more likely if X_i increases. Conversely, a negative effect means that the outcome becomes less likely if X_i increases. The absolute value of a coefficient shows the strength of the effect. For ease of interpreting the effects strengths, I convert the coefficients to ORs, which is done by exponentiation, that is, by computing e^{β_i}. The resulting OR for an independent variable X_i is the

[1] Note that the p in (5.1) is unrelated to the significance level.

[2] https://www.rdocumentation.org/packages/stats/versions/3.6.1/topics/glm

factor by which the odds that the dependent variable becomes *true* increase if X_i increases by one unit, assuming all other independent variables $X_j, j \neq i$, are kept constant. As an example, consider two functions, f_1 and f_2, and assume that the FC metric for f_1 is 1, and for f_2, it is 2, and assume that all other metrics (e. g., CND, LOC) have identical values. Furthermore, assume that the odds of being changed are 2:5 for f_1 and that the coefficient for *fc* is 0.405. Given these values, I can compute the expected odds that f_2 changes in two steps: In the first step, I compute the OR of *fc*, which is $e^{0.405} \approx 1.5$. In the second step, I multiply the odds of f_1 with this factor. Thus, the expected odds that f_2 changes are $2{:}5 \times 1.5 = 3{:}5$. In short, the OR of *fc* in this example is the factor by which the odds to change increase if the number of feature constants increases by one.

Apart from estimating the coefficients $\beta_1, \ldots, \beta_n$, logistic regression also estimates a constant called the *intercept* (see Equation 5.1). In a null model (i. e., a model without predictor variables), the *intercept* is directly related to the proportion of successes in the dataset, that is, the proportion of data points where the dependent variable is *true*. More formally, the *intercept* in a null model is the log-odds of success in that dataset. In models that comprise one or several predictors, the *intercept* basically summarizes the effects that are *not* part of the model. Beyond that, it has no meaningful interpretation.

I next explain which dependent and independent variables I use in my logistic regression models. Since H_0 3.1 focuses on the likelihood of changes, my dependent variable is the binary variable *changed* (see Table 5.1). As independent variables, I include *fc*, *cnd*, *neg*, and $loac_{/loc}$. The variable $loac_{/loc}$ is included to investigate the influence of the amount of annotated code in a function. I chose $loac_{/loc}$ over *loac* for this purpose because *loac* correlates more strongly with *loc* than $loac_{/loc}$, thus potentially causing other multicollinearity problems. Calculation of the *variation inflation factor (VIF)* [76] of different model variants proved $loac_{/loc}$ to be less problematic than *loac*. As mentioned in Section 5.2.3, I was forced to exclude *fl* due to its strong correlation with *fc* (Pearson's $r = 0.86$, Spearman's $r_S > 0.99, p \ll .001$). In other words, there is a very strong tendency in my data that more feature constants are referenced in functions that contain more preprocessor directives. In addition to the preprocessor-related variables, I additionally include *loc*, *age*, *mrc*, and *pc* (see Table 5.1) to control potential confounding factors.

The exclusion of *fl* in favor of *fc* raises the question of what my regression models reveal about the effect of the number of feature locations on change proneness. The strong, positive correlation between *fc* and *fl* suggests that the answers I obtain for *fc* can generally be transferred to *fl*, and I found this was indeed the case by conducting a separate experiment: I computed an alternative set of regression models that comprised *fl* instead of *fc*, and found that the results were almost identical. In particular, *fl* and *fc* had almost the same number of significant effects, the effects had the same directions, and the magnitude of the effects was very similar, with *fc* having slightly stronger effects than *fl*. Thus, my findings regarding the effect of the number of feature constants in a function apply in equal measure to the number of feature locations in a function.

All independent and control variables except $loac_{/loc}$ are included in $\log_2$-transformed form. For example, *fc* and *loc* are included as $\log_2(fc)$ and $\log_2(loc)$. The transfor-

mation was applied because preliminary experiments showed that it leads to better-fitting models compared to including the variables without any transformation. The $\log_2$-transformation changes the interpretation of the ORs as follows: A one unit increase of $\log_2(loc)$ means that the underlying LOC metric doubles in value. Likewise, a one unit decrease of $\log_2(loc)$ means that LOC halves. Thus, the OR of $\log_2(loc)$ is the factor by which the odds increase (or decrease) if the value of the LOC metric is doubled (or halved). The majority of my independent and control variables (all except *loc*) can take on the value 0, but since the logarithm is undefined for 0, I added 1 to the original value before computing the logarithm. For example, the $\log_2$-transformed variable for the FC metric, $log_2(fc)$, is actually computed as $log_2(FC+1)$. Therefore, if $\log_2(fc)$ takes on the values $0, 1, 2, 3, \ldots, n$, then the underlying FC values will be $0, 1, 3, 7, \ldots, 2^n-1$, respectively. To summarize, the OR of $\log_2(fc)$ is the factor by which the odds increase if FC increases from 0 to 1 or (for greater values) if FC approximately doubles. The ORs of the other $\log_2$-transformed variables ($\log_2(cnd)$, $\log_2(neg)$, $\log_2(age)$, $\log_2(mrc)$, and $\log_2(pc)$) must be interpreted accordingly.

Apart from the full models, I also build simple models that comprise just the control variables. I compare these simple models to the full models using the McFadden statistic [237]. Based on this comparison, I can conclude how much better change proneness is explained when preprocessor-related variables are taken into account.

I build logistic regression models for each of my subject systems, using the combined data of all commit windows as the input. Differently from RQ 1, I cannot use the Mantel-Haenszel method to aggregate the resulting models because the independent variables for RQ 3 are metric, not binary. Hence, I use the following procedure: First, I identify in each model which independent variables have a statistically significant effect at $p < .01$. Afterwards, I compute the mean value of the regression coefficients of these variables and convert the mean coefficient to an OR by exponentiation. I reject H_0 3.1 for an independent variable if the variable has a significant effect in the majority of subject systems. Otherwise the evidence is insufficient to justify a rejection.

5.3.3.2 Effect on the Frequency and Profundity of Changes

The logistic regression models to test H_0 3.1 treat change proneness as a binary variable. Thus, they only reveal how preprocessor use affects the likelihood of changes, but not how it affects the frequency and extent of these changes. Therefore, I create another kind of regression model in which the frequency and the extent of changes is considered in metric form.

My dependent variables (*commits* and *lines*) constitute count data, which I found not to be normally distributed (cf. Figure 5.3 and Figure 5.4 in Section 5.4.1 for histograms). This rules out linear regression, which assumes a normal distribution. Also, a Poisson distribution cannot be used to describe my counts, as I found strong evidence of *overdispersion* [76], i. e., the variance of the statistical variables exceeds the mean. Hence, I chose *negative binomial regression* [137], a technique that other software engineering researchers have used to predict software faults (e. g., [280, 353, 109, 124]). The distribution characteristics of faults closely resemble those of my

dependent variables. Thus, the same regression technique is appropriate. Negative binomial regression estimates the coefficients $\beta_1, \beta_2, \ldots, \beta_n$ of the formula

$$\ln(Y) = intercept + \beta_1 X_1 + \beta_2 X_2, \ldots, \beta_n X_n \tag{5.2}$$

In this formula, $X_1, X_2, \ldots, X_n$ are the independent variables (such as, $\log_2(fc)$, $\log_2(loc)$), Y is the dependent variable (either *commits* or *lines*), and the *intercept* is a constant. As for logistic regression, the effect size of an independent variable is indicated by the coefficient. Additionally, R's implementation of negative binomial regression[3] estimates the statistical significance of each variable. The effect sizes are interpreted as follows: Given that β_i is the coefficient of X_i and assuming that all other variables $X_j, j \neq i$, are kept constant, then if X_i increases by one unit, I can expect Y to take on $1+\beta_i$ times its previous value. As an example, consider again two functions f_1 and f_2 that are identical in all metrics except for FC, which is 0 for f_1 and 1 for f_2. Furthermore, assume that the coefficient for $\log_2(fc)$ is 0.2 and that the dependent variable is *commits*. Then I expect f_2 to undergo 1.2 times as many commits as f_1.

I create two regression models for each of my subject systems, one where *commits* is the dependent variable and one where *lines* is the dependent variable. In both models, I use the same input data and include the same independent variables and control variables that I also use for the logistic regression models (see Section 5.3.3.1). Again, all independent and control variables except $loac_{/loc}$ are included in $\log_2$-transformed form.

As for testing H_0 3.1, I build simple models comprising just the control variables and compare them to the full models to measure how much the annotation metrics help explain change proneness. As a way to derive summary effect sizes over all subjects, I compute the mean and standard deviation of the regression coefficients for each independent variable if the variable's effect is statistically significant at $p < .01$. I follow the same procedure to decide whether to reject (or fail to reject) the null hypotheses that I also use for H_0 3.1. I use the models with *commits* as the outcome to test H_0 3.2 and the models with *lines* as the outcome to test H_0 3.3.

5.4 Quantitative Results

Next, I present the quantitative results of my analysis. After discussing descriptive statistics of the data extracted from my subject systems, I address each research question and reject or fail to reject the corresponding null hypotheses.

5.4.1 Descriptive Statistics

I now present the descriptive statistics to give an indication of the average extent of preprocessor use in my subject systems, the average frequency and extent of changes, and also of the average values of the controlled factors, such as function size. Moreover, I discuss the distributions of the corresponding statistical variables. This information will make it easier to put the results of correlation analyses in the following sections into perspective.

[3] https://cran.r-project.org/web/packages/MASS/index.html

I start with the overview in Figure 5.2. For this overview, I first computed the arithmetic means of all variables in each subject system and then created violin plots that depict the distributions of these mean values. The first six violin plots (in yellow) are related to the variables of preprocessor use, the following four (in gray) are related to the control variables, and the last three (in light blue) are related to

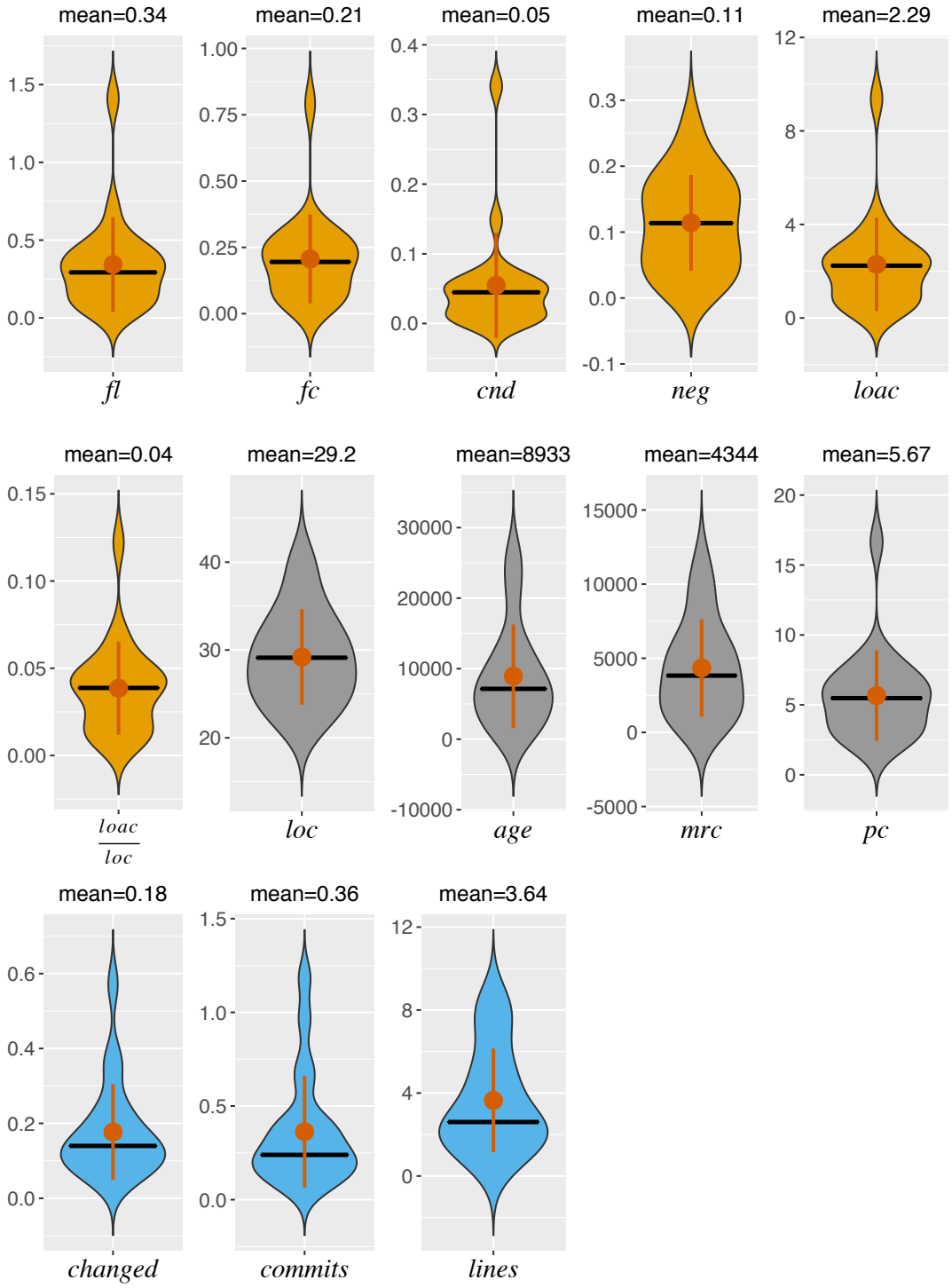


Figure 5.2: Distribution of the mean values of all variables in all subjects

the dependent variables. The black horizontal bars highlight the median value, the red dots highlight the mean value, and the red vertical lines extend to one standard deviation above and below the mean. Additionally, I state the overall mean value in numerical form above each plot. Except for *changed*, all variables are metric variables, and for those variables, the interpretation is straightforward. I take *fl*, the number of preprocessor directives per function, as an example. The mean value of *fl* over all subjects is 0.34, and thus, we can expect to find one `#ifdef` directive in three randomly chosen functions. The vertical extent of the violin plots indicate that this expectation varies between systems. For *fl*, it ranges from 0.05 in GIMP (one `#ifdef` directive in twenty functions) to 1.41 in VIM (about seven `#ifdef` directives in five functions).

Besides the metric variables, I have one dichotomous (binary) variable, *changed*, and for this variable, the violin plot visualizes the distribution of the average probability of the variable becoming *true*. Hence, the mean value 0.18 states that, on average, about one in five functions is expected to change over the course of a commit window (500 commits). Again, there is some variation between the subjects. On the low end, there is QEMU, where only one in twenty functions is changed ($M = 0.05$), and on the high end, there is OPENVPN, where eleven out of twenty functions are changed ($M = 0.57$).

Next, I take a more detailed look at the distributions of my variables in Figures 5.3 and 5.4. I use the data from BUSYBOX as an example; the distributions in other subjects look similar. The histograms in both figures follow the same color coding as Figure 5.2: The yellow ones are related to the variables of preprocessor use, the gray ones to the control variables, and the light blue ones to the dependent variables. The histograms in Figure 5.3 illustrate the skew of the distributions of the variables: Low values are very common whereas high values are very uncommon. For example, the histogram for *fl* shows that the vast majority of functions in BUSYBOX contains between 0 to 6 `#ifdef` directives, but there is a tiny minority that contains 47 `#ifdef` directives or more. Other researchers made similar observations regarding the distribution of software engineering data, both in relation to metrics of preprocessor use [298] as well as in relation to other metrics, such as lines of

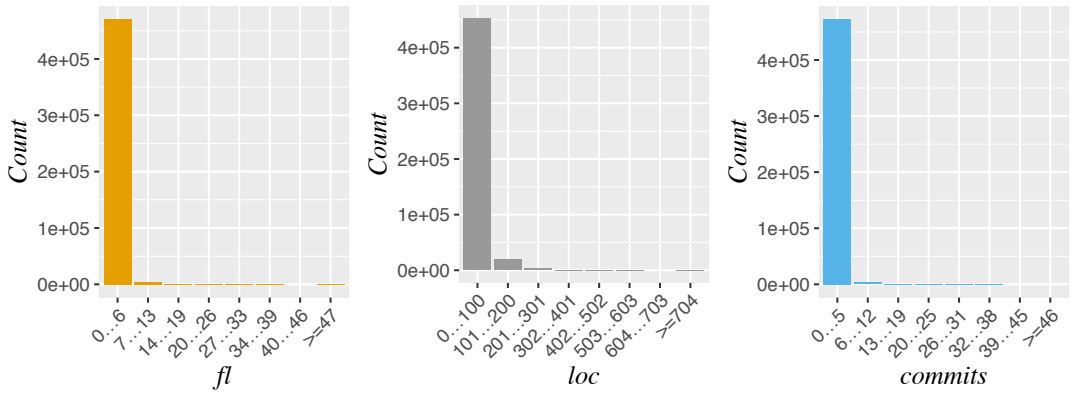


Figure 5.3: Histograms of selected variables without applying transformations (data from BUSYBOX)

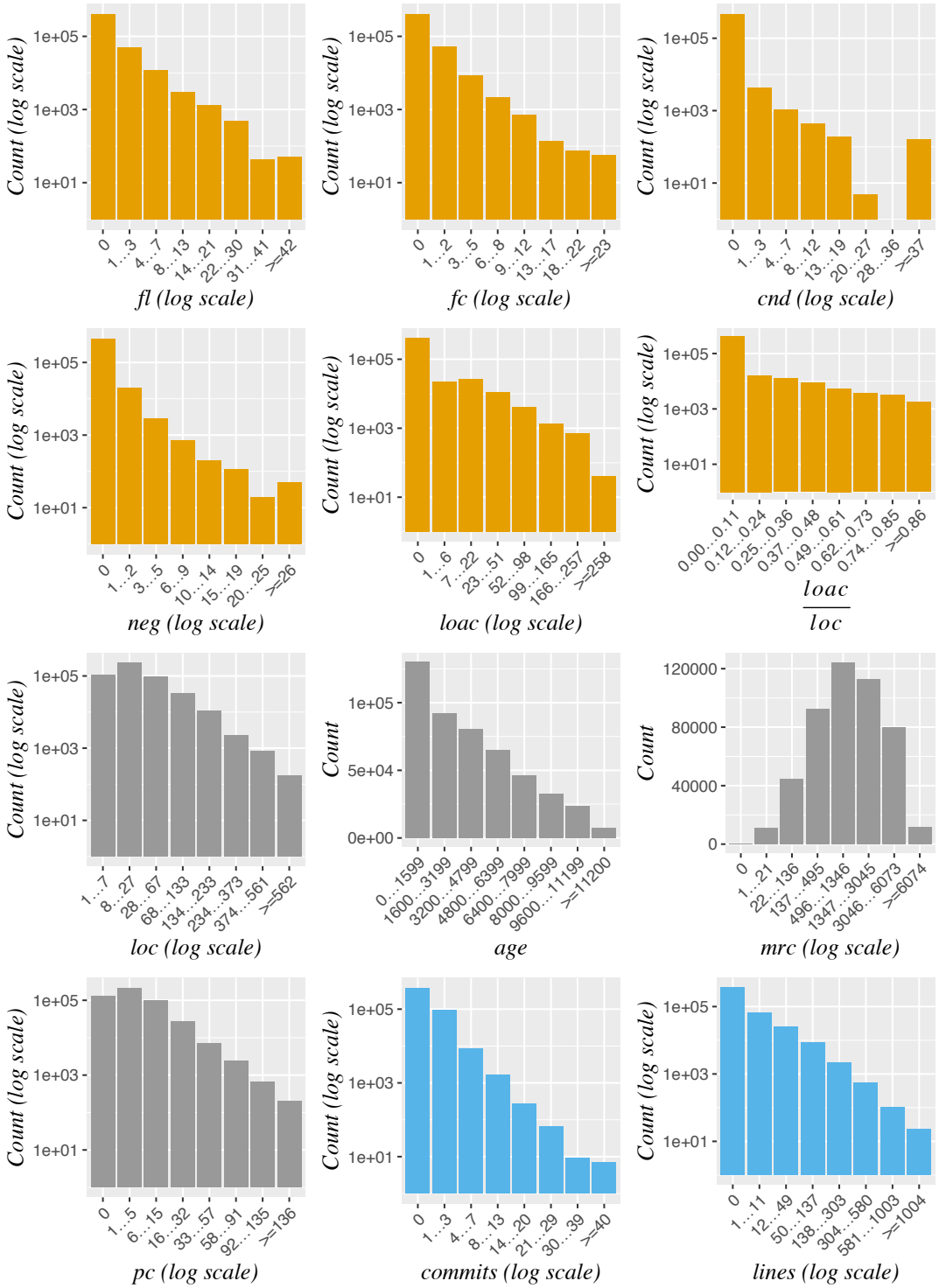


Figure 5.4: Histograms of all variables after applying transformations (data from BusyBox)

code and faults [338, 124]. To provide deeper insights into the distributions despite these skews, I created a second set of histograms in which the x- or y-axes (and sometimes both) use a logarithmic scale. A logarithmic scale on the x-axis increases the resolution for small values of the variable, and for some variables, such as *mrc*,

this was enough to make the shape of the distribution become visible. However, for most variables, such as *fl* and *commits*, the skew was so strong that I additionally used a logarithmic scale on the y-axis to prevent the high frequency of low values from completely overshadowing the low frequency of higher values. The resulting histograms are depicted in Figure 5.4. These histograms reveal two things: First, the extent of preprocessor use is low in most functions, with the vast majority containing no `#ifdef` directives at all. For example, 86 % of all functions in BusyBox contain no `#ifdef` directives, 10 % contain between 1 and 3, and only 2.5 % contain between 4 and 7. Second, most functions are not changed, and if they are, chances are high that they are changed only once and only by a few lines. For example, only 2.2 % of all functions in BusyBox are changed more than three times over the course of a commit window, and changes of more than eleven lines of code affect only 7.7 % of all functions.

5.4.2 RQ 1, H_0 1.1: Relationship between Binary Properties of Preprocessor Use and the Likelihood of Changes

By testing H_0 1.1, I answer the question whether the presence or absence of different aspects of preprocessor use affects the likelihood that a function is changed. The results of the statistical analyses are given in Table 5.9. For each independent variable, I report the number of subjects in which the variable had a statistically significant effect at three levels of significance. Moreover, I report the pooled p values and ORs computed over all subjects.

Variables		Significance per Subject				All Subjects	
Dep.	**Indep.**	**N**	**$<.001$**	**$<.01$**	**$<.05$**	***p***	**OR**
changed	$fl_{>0}$	20	20	20	20	$\ll .001$	2.19
changed	$fc_{>1}$	20	20	20	20	$\ll .001$	2.91
changed	$cnd_{>0}$	20	19	19	20	$\ll .001$	3.08
changed	$neg_{>0}$	20	20	20	20	$\ll .001$	2.20

N: total number of subjects; **$<.001$**, **$<.01$**, **$<.05$:** number of subjects where the effect was significant at the given p value; ***p*:** pooled p value over all subjects; **OR:** pooled odds ratio over all subjects

Table 5.9: Between-group differences regarding the likelihood of changes

In addition to the numeric results in Table 5.9, I also show the effect sizes as a set of box plots in Figure 5.5. The box plots were created as follows: Every box covers 50 % of the data, i. e., its height corresponds to one interquartile range, and the thick line marks the median value. The location of the whiskers depends on the minimum and maximum values in the data. The standard location of the whiskers is at 1.5 times the interquartile range above and below the box, respectively. However, if a whisker would lie beyond the actual minimum/maximum value, it is retracted to mark the actual minimum/maximum value. Finally, circles above and below the whiskers mark outliers, i. e., values whose distance from the box exceeds 1.5 times the interquartile range. Below each box plot, I indicate in parentheses in how many subjects the respective independent variable had a statistically significant effect at

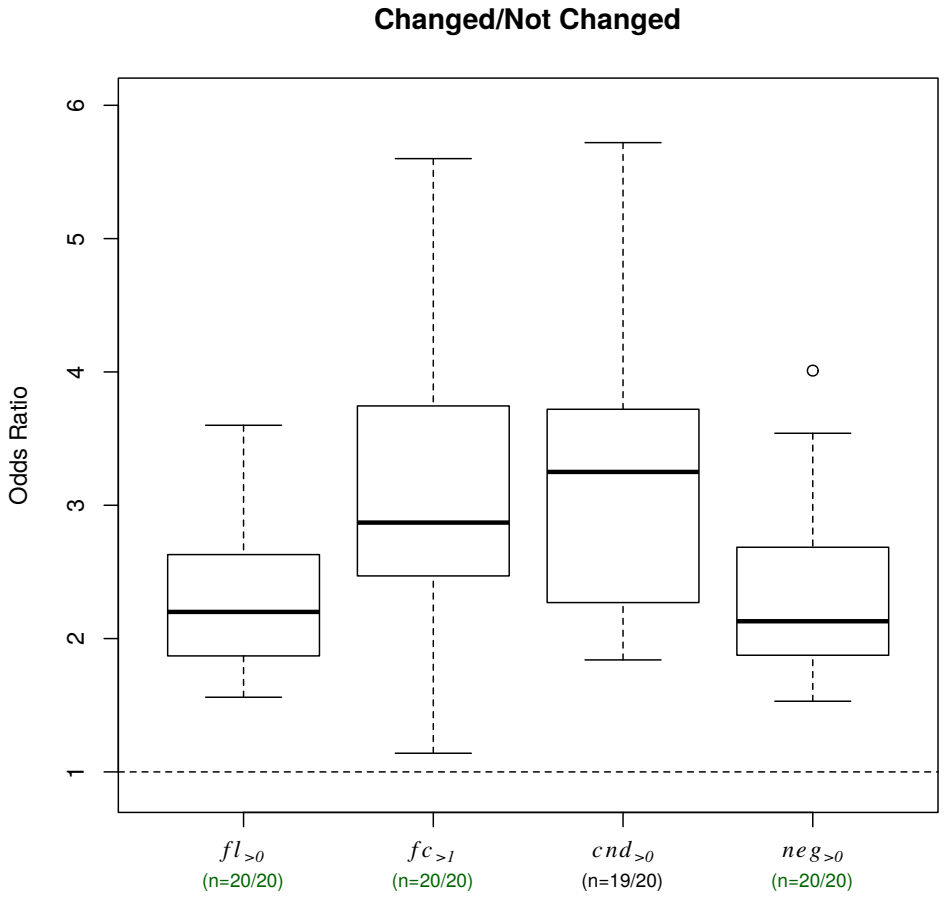


Figure 5.5: Between-group differences regarding the likelihood of changes

$p < .01$. Green indicates that a variable had a statistically significant effect in all subjects, and black that it had a significant effect in more than half of the subjects. As a visual aid to mark the value of a neutral effect, I inserted a dashed horizontal line at the OR 1.

The summarized results (see last two columns of Table 5.9) reveal that all four variables are associated with highly significant differences. The test results for the individual subjects are also very consistent. Specifically, the variables $fc_{>1}$, $fl_{>0}$, and $neg_{>0}$ have statistically significant effects at $p < .01$ in all twenty subjects, and $cnd_{>0}$ has statistically significant effects in all subjects but one, GNUMERIC ($p = .04$).

The pooled ORs of all variables are positive. Likewise, inspecting the individual results of every subject, I found that all ORs in all subjects are positive. Even in GNUMERIC, where $cnd_{>0}$ failed the significance test, the effect tends to be positive (OR = 1.18). I conclude that the direction of the effects of all four variables is consistent across subjects, and the direction is always positive.

The pooled ORs range from 2.19 (for $fl_{>0}$) to 3.08 (for $cnd_{>0}$). I use PIDGIN and $cnd_{>0}$ to illustrate the strength of these effects. In PIDGIN, the odds are 12:88 that functions without any nested annotations ($cnd_{>0} = 0$) are changed, and the OR of $cnd_{>0}$ is 2.7. Thus, the odds that functions with nested annotations ($cnd_{>0} = 1$) in PIDGIN are changed are $12{:}88 \times 2.7 \approx 32{:}88$, which, when normalized to 100, approximately amounts to the odds 27:73. Expressed in terms of likelihoods, a change

to a function in PIDGIN is 12 % likely if there are no nested annotations and 27 % likely if there are nested annotations. This difference is considerable. Since the average ORs of all four variables fall in the same range as the OR of $cnd_{>0}$ in PIDGIN, I conclude that all four variables are associated with considerable differences in the likelihood of changes.

The long whiskers of the box plots in Figure 5.5 indicate that the ORs of all four variables vary strongly between subjects. As an example, I inspected the OR of $cnd_{>0}$ in all subjects, finding it ranges from 1.84 at the low end (in MYSQL) to 5.72 at the high end (in SQLITE). As the box plots in Figure 5.5 show, the ORs of the other variables vary to a similarly high degree. I therefore note that the effects of all four variables vary greatly in strength from subject to subject.

In summary, I observe that preprocessor use, when viewed as a binary property, correlates with statistically significant differences in the likelihood that functions are changed. The direction of the statistical effect is always positive but it varies from subject to subject. Based on these observations, I *reject* H_0 1.1 and instead accept the following alternative hypotheses:

> ***H_a 1.1 ($fl_{>0}$):*** *Functions containing* at least one preprocessor directive *are* more likely to be changed *than functions* without any preprocessor directives.

> ***H_a 1.1 ($fc_{>1}$):*** *Functions containing preprocessor directives that reference* two or more feature constants *are* more likely to be changed *than functions in which* fewer feature constants *are referenced.*

> ***H_a 1.1 ($cnd_{>0}$):*** *Functions containing* at least one nested preprocessor directive *are* more likely to be changed *than functions* without any nested preprocessor directives.

> ***H_a 1.1 ($neg_{>0}$):*** *Functions containing preprocessor directives that use* negation at least once *are* more likely to be changed *than functions* without preprocessor directives that use negation.

5.4.3 RQ 1, H_0 1.2 and H_0 1.3: Relationship between Binary Properties of Preprocessor Use and the Frequency and Extent of Changes

By testing H_0 1.2 and H_0 1.3, I answer the question how much the presence or absence of different aspects of preprocessor use affects the frequency and the extent of changes to a function. I report the results of the statistical analyses in Table 5.10. The upper part of the table contains the results of the tests for H_0 1.2, where the dependent variable is *commits* (the number of changes). The lower part contains the results for H_0 1.3, where the dependent variable is *lines* (the number of lines changed). Similarly to Table 5.9, I report the number of subjects in which an independent variable had a statistically significant effect at three levels. Additionally,

I report the number of subjects in which the p value of the correlation exceeded .05, indicating a clearly insignificant effect. Moreover, I report effect sizes in terms of the mean and standard deviations of Cliff's delta, d. These averages encompass only effect sizes of the significant correlations ($p < .01$). The final column contains a qualitative assessment of the mean d following the thresholds proposed by Grissom and Kim [118]. According to these thresholds, d is *negligible* if $|d|<0.147$, *small* if $|d|<0.33$, *medium* if $|d|<0.474$, and *large* otherwise. In Table 5.10, the effects are symbolized by ○ for *negligible*, and ◐ for *small* (no stronger mean effects were observed).

Variables		Significance					Cliff's Delta	
Dep.	**Indep.**	**N**	**$<.001$**	**$<.01$**	**$<.05$**	**$\geq.05$**	$M\pm SD$	
commits	$fl_{>0}$	20	20	20	20	0	0.14±0.08	○
commits	$fc_{>1}$	20	19	19	19	1	0.20±0.11	◐
commits	$cnd_{>0}$	20	16	17	18	2	0.22±0.11	◐
commits	$neg_{>0}$	20	20	20	20	0	0.15±0.09	◐
lines	$fl_{>0}$	20	20	20	20	0	0.14±0.08	○
lines	$fc_{>1}$	20	19	19	19	1	0.21±0.11	◐
lines	$cnd_{>0}$	20	17	18	18	2	0.22±0.12	◐
lines	$neg_{>0}$	20	20	20	20	0	0.15±0.09	◐

N: total number of subjects; **$<.001$**, **$<.01$**, **$<.05$:** number of subjects where the effect was significant at the given p value; **$\geq.05$:** number of subjects where the effect was not significant at $p < .05$; **Cliff's Delta:** mean value and standard deviation of Cliff's delta (if significant); the symbols ○ and ◐ indicate *negligible* and *small* mean effects, respectively

Table 5.10: Between-group differences regarding the frequency and profundity of changes

As for RQ 1, H_0 1.1, I also depict the effect sizes as box plots, with the results related to H_0 1.2 in Figure 5.6 and those related to H_0 1.3 in Figure 5.7. The box plots were drawn using the same settings as in Figure 5.5. The difference is that the dashed horizontal line is now located at 0 as this is the neutral value for Cliff's delta.

In the data in Table 5.10, Figure 5.6, and Figure 5.7, I observe a high proportion of significant correlations. For both dependent variables, the variables $fl_{>0}$ and $neg_{>0}$ have significant effects ($p < .01$) in all twenty subjects, and $fc_{>1}$ has significant effects in all subjects except GNUMERIC. The variable $cnd_{>0}$ has significant effects on *commits* in seventeen subjects and on *lines* in eighteen subjects. The insignificant results were obtained in BLENDER, GIMP, and GNUMERIC.

Manual inspection of the Cliff's d values of all subjects revealed that all significant relationships are positive, both regarding *commits* and *lines*. The average effect sizes range from *negligible* to *small* but in a few subjects, I found stronger effects. In particular, I observed *medium* and *large* effects in four subjects (BUSYBOX, GNUPLOT, LIBXML2, and OPENVPN). For example, the effects of $fc_{>1}$ on *commits* are *medium* in BUSYBOX ($d = 0.33$) and OPENVPN ($d = 0.42$). The only *large*

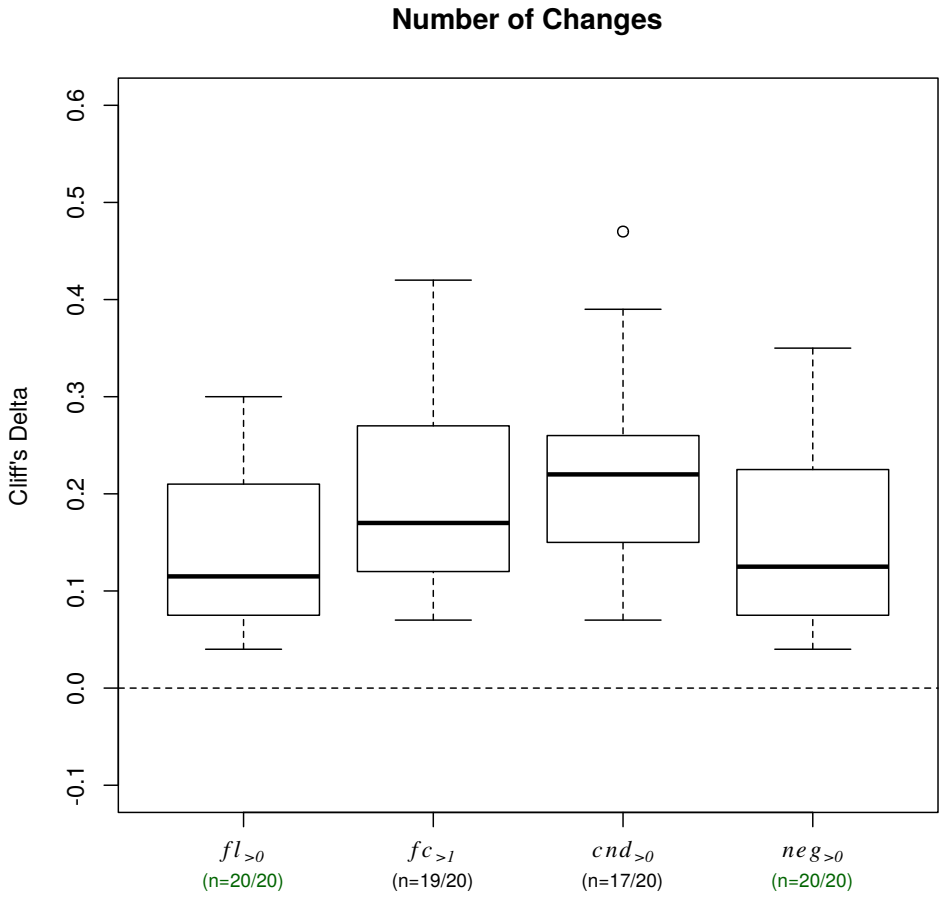


Figure 5.6: Between-group differences regarding the frequency of changes

effect was observed for $cnd_{>0}$ on *lines* in OpenVPN ($d = 0.52$). Compared to the results for RQ 1, H_0 1.1, where all four variables were associated with considerable average effects, I note that the average effects are now smaller. In other words, the differences in the *likelihood* that functions are changed are considerable, but at the same time, the differences in the *frequency* or *extent* of changes are rather small.

Similarly to RQ 1, H_0 1.1, I observe that the effects vary greatly between subjects. Taking $cnd_{>0}$ as the example again, I found that its effect on *lines* ranges from a *negligible* Cliff's d of 0.07 (in MySQL) to a *large* 0.52 (in OpenVPN). The effect sizes of the other variables vary to a similar degree, regarding both *commits* and *lines* as the outcome. This aligns with my previous observation that the effect strengths of all four variables vary greatly between subjects.

In summary, I observe that preprocessor use, when considered as a binary property, has a statistically significant effect in the majority of the subject systems on both the frequency with which functions are changed, as well as the extent of those changes. The direction of the effects is always positive but it varies from subject to subject. Based on these observations, I *reject* both H_0 1.2 and H_0 1.3 for all four variables and instead accept the alternative hypotheses stated below. For brevity, I only explicitly state the alternative hypotheses regarding the presence/absence of `#ifdef`s. The alternative hypotheses regarding the number of feature constants, nesting, and negation hold accordingly.

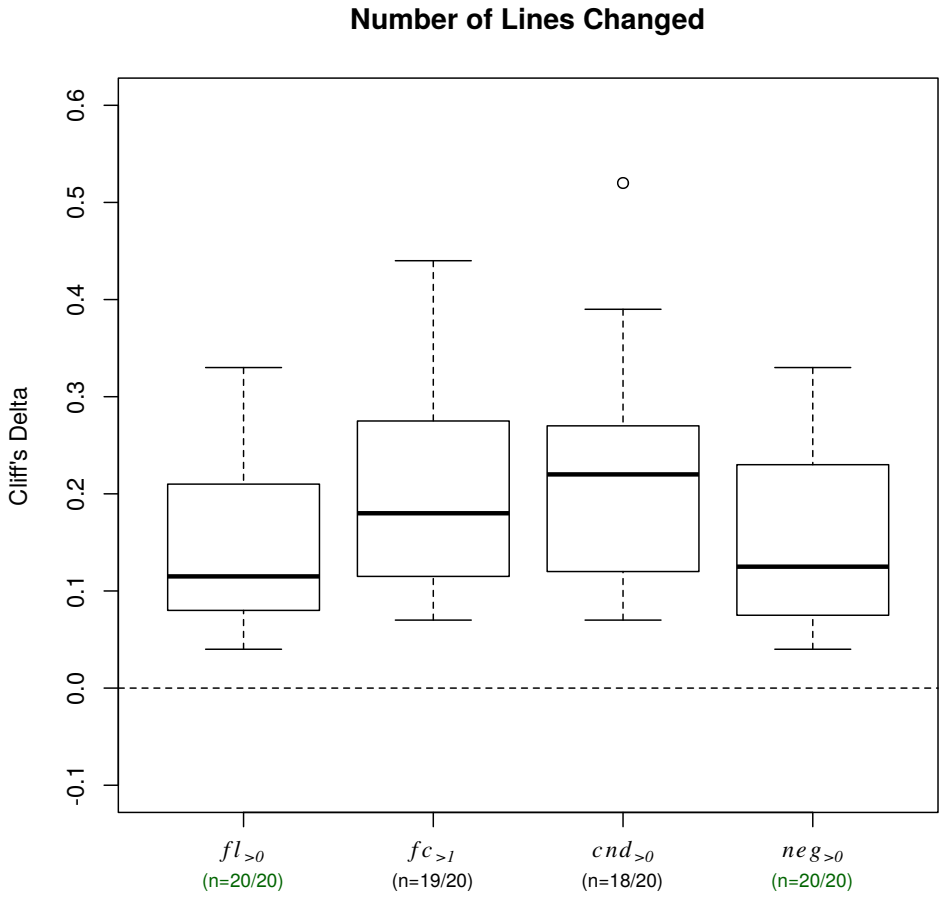


Figure 5.7: Between-group differences regarding the profundity of changes

H_a 1.2 ($fl_{>0}$): *Functions containing* at least one preprocessor directive *are* changed more frequently *than functions* without any preprocessor directives.

H_a 1.3 ($fl_{>0}$): *Functions containing* at least one preprocessor directive *are* changed more profoundly *than functions* without any preprocessor directives.

5.4.4 RQ 2: Relationship between Different Extents of Preprocessor Use and Function Size

In RQ 2, I ask whether different extents of CPP use affect a function's change proneness. Before I present the results, I investigate the relation of preprocessor use to function size, as size is a possible confounder. I initially illustrate this relation by means of Figure 5.8. In particular, I consider the variables *fl*, *fc*, *cnd*, *neg*, and $loac_{/loc}$ as independent variables and relate them to function size (variable *loc*) as the dependent variable. I differentiate between three metric values, resulting in three boxplots per metric. For *fl*, *fc*, *cnd*, and *neg*, which are integer-valued, I differentiate between 0, 1, and 2 or greater. For $loac_{/loc}$, which is a ratio, I consider the ranges 0–10 %, 10–20 %, and 20 % or greater. The width of each box depends on the number of functions that fall into the respective group. Due to limited space, I show only the boxplots for BusyBox; the other subjects exhibit the same trends.

My plots indicate that *fl*, *fc*, *cnd*, and *neg* correlate positively with function size, indicating that functions that use preprocessor directives more heavily are, on average, longer. Moreover, as the values of the metrics increase, function size varies more (indicated by longer whiskers), and the number of functions decreases considerably (indicated by leaner boxes). By comparison, the trend for $loac_{/loc}$ is less clear. It is also noteworthy that the number of functions decreases as $loac_{/loc}$ increases but in contrast to the results for *fl*, *fc*, *cnd*, and *neg*, the highest average sizes and the highest variations in size occur in the second group ($loac_{/loc}$ between 10 and 20 %), not in the third group.

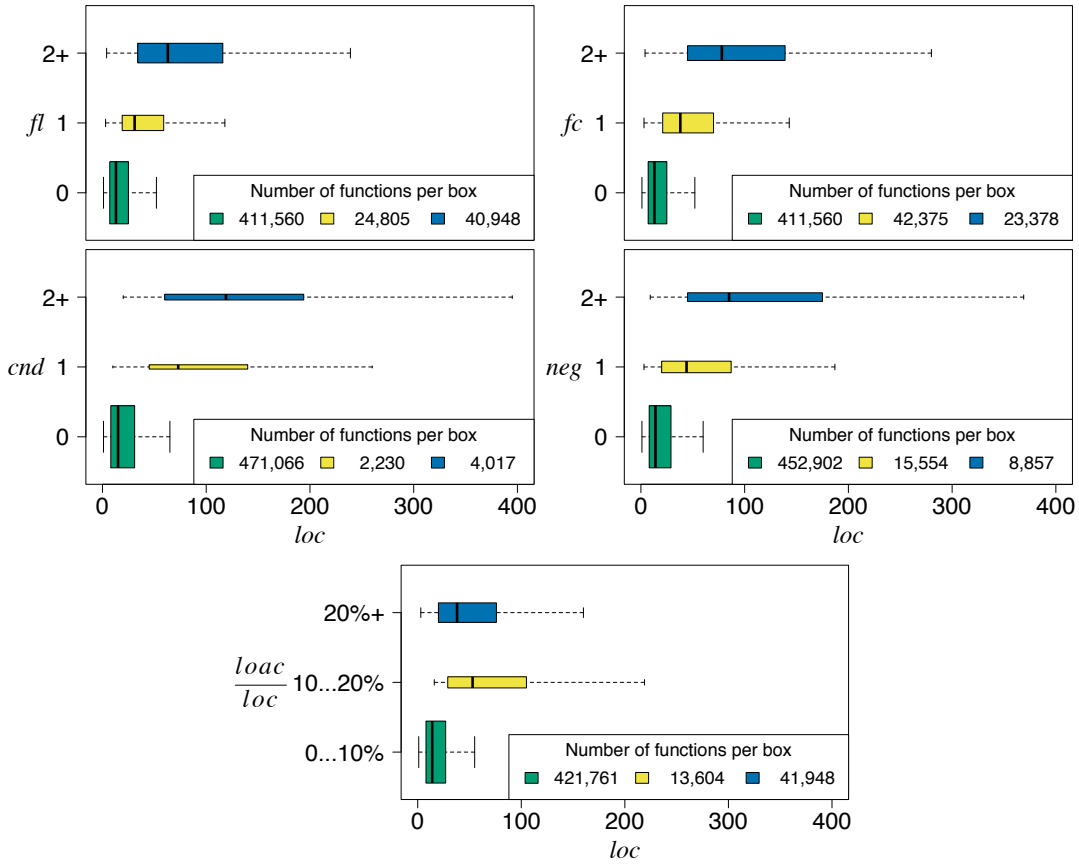


Figure 5.8: Distribution of function sizes for different extents of preprocessor use in BusyBox

In summary, the plots suggest that functions making heavy use of (complex) CPP directives tend to be larger. At the same time, such functions occur less frequently compared to functions that make less or no use of CPP directives.

I calculated Spearman's rank correlation coefficient, r_S, to measure the relationship between different values of my annotation-related metrics and function size. In all of my subjects, the number of functions without any annotations ($fl = 0$) is much greater than the number of functions with annotations ($fl \geq 1$) and this imbalance may mask possible correlations between my preprocessor metrics and function size. For this reason, I calculated the correlation coefficients on two datasets of each subject, a full dataset and a balanced dataset. The full dataset comprises the data of all functions in a subject. In the balanced dataset, in turn, the number of functions

without any annotations was downsampled (using random sampling) to match the number of functions with annotations.

I show the correlation coefficients, summarized over all subjects, in Table 5.11. The results for the full datasets are shown in the upper half of the table, and the results for the balanced datasets are shown in the lower half. In the column "Significant," I report the number of subjects in which correlations were significant, and the total number of subjects. In the last columns, I report the mean value and the standard deviations of the correlations. Spearman's rank correlation coefficient, r_S, can range from -1 to $+1$, indicating strong negative or strong positive correlations, respectively. The absolute value of r_S indicates how well the relationship between independent and dependent variable can be described by a monotonic function. However, the particular kind of relationship (e. g., logarithmic, linear, or quadratic) is unimportant. The correlation is *very weak* for $|r_S|<0.2$, *weak* for $0.2\leq|r_S|<0.4$, *moderate* for $0.4\leq|r_S|<0.6$, *strong* for $0.6\leq|r_S|<0.8$, and *very strong* for $|r_S|\geq0.8$. In Table 5.11, these different strengths are symbolized by ○ for *very weak*, ◐ for *weak*, and ● for *moderate* correlations. (*Strong* or *very strong* mean correlation strengths were not observed.) For both the full and the balanced datasets, the weakest correlation is highlighted in light gray, and the strongest one in dark gray.

	Variables		Significant	Spearman's r_S	
	Dep.	Indep.		M± SD	
Full	*loc*	*fl*	20/20	0.29 ±0.10	◐
	loc	*fc*	20/20	0.28 ±0.10	◐
	loc	*cnd*	20/20	0.12 ±0.06	○
	loc	*neg*	20/20	0.19 ±0.06	○
	loc	$loac_{/loc}$	20/20	0.27 ±0.09	◐
Balanced	*loc*	*fl*	20/20	0.53 ±0.06	●
	loc	*fc*	20/20	0.50 ±0.06	●
	loc	*cnd*	20/20	0.20 ±0.06	◐
	loc	*neg*	20/20	0.30 ±0.07	◐
	loc	$loac_{/loc}$	20/20	0.35 ±0.06	◐

Significant: number of subjects where effect was statistically significant and total number of subjects; **Spearman's rank r_S:** mean value and standard deviation of Spearman's rank correlation coefficient; mean correlation strength: ○ *very weak*, ◐ *weak*, ● *moderate*

Table 5.11: Correlations between extent of CPP use and function size

The data in Table 5.11 reveal that all correlations are significant in all subjects, both in the full and in the balanced datasets. In fact, all p values I obtained satisfy the .0001 level, indicating highly significant correlations. All independent variables correlate positively with function size, confirming the trend already suggested by Figure 5.8. Among all variables, *cnd* has the lowest and *fl* the highest correlation strength in both datasets. Comparing the average effects on both datasets, I observe only *very weak* or *weak* effects in the full datasets but stronger effects in the balanced datasets, where they range from *weak* to *moderate*. I believe that the weaker effects

in the unbalanced datasets are indeed caused by a masking effect, that is, the vast majority of functions without CPP directives overshadows the correlation between different extents of CPP use and function size. Based on these results, I *reject* $H_0\,2$ for all annotation metrics and accept the following alternative hypothesis:

> ***H_a 2 ($fl, fc, cnd, neg, loac_{/loc}$):*** *Functions containing more preprocessor directives, functions in which the preprocessor directives reference more feature constants, use more negation or are more deeply nested, as well as functions with a higher ratio of annotated code are also longer than functions in which these metrics have lower values.*

Given $H_a\,2$ and combining it with the existing evidence that larger functions are also more change-prone, I conclude that function size must be taken into account as a possible confounder when analyzing the relationship between preprocessor use and change proneness. I present the results next.

5.4.5 RQ 3, H_0 3.1: Relationship between Different Extents of Preprocessor Use and the Likelihood of Changes

By testing $H_0\,3.1$, I answer the question if and how much different aspects of preprocessor use affect the likelihood of changes to a function when all aspects are considered in combination and in the context of possible confounding factors, such as function size. To answer this question, I computed logistic regression models for all twenty subjects. I summarize the results in Figure 5.9, Table 5.12, and Table 5.13: If an independent variable had a statistically significant effect in a given subject at $p < .01$, I included the corresponding OR in the summary; otherwise it was excluded. In Figure 5.9, I provide a graphical overview of this summary. In Table 5.12, I report the number of subjects in which the effects where significant, the number of times the effect direction was positive or negative, as well as the mean effect sizes. In Table 5.13, in turn, I report the minimum and maximum effect strengths.

The box plots in Figure 5.9 were drawn using the same settings that I used in Figure 5.5, and the number of subjects in which an independent variable had a statistically significant effect is denoted below the x-axis in the same way. The dashed horizontal line at the OR 1 highlights the neutral effect.

Preprocessor use. The data in Figure 5.9 and Table 5.12 reveal that all four variables of preprocessor use significantly affect the likelihood of changes in a majority of the subjects. However, I also note that only one variable, $\log_2(fc)$, has a significant effect in all subjects. The average ORs deviate only by a small amount from neutral (see Table 5.12). To illustrate what these ORs mean in practice, I take fc (the number of distinct feature constants) as the example. The average OR of $\log_2(fc)$ is 1.14, meaning that the odds for a function to change multiply by 1.14 when the number of feature constants in a function approximately doubles. More precisely, the odds multiply by 1.14 if the number of feature constants increases one step in the sequence $0, 1, 3, 7, \ldots, 2^n-1$ (see Section 5.3.2 for details). According to the data in Figure 5.2, the mean probability of a function being changed is 0.18, which

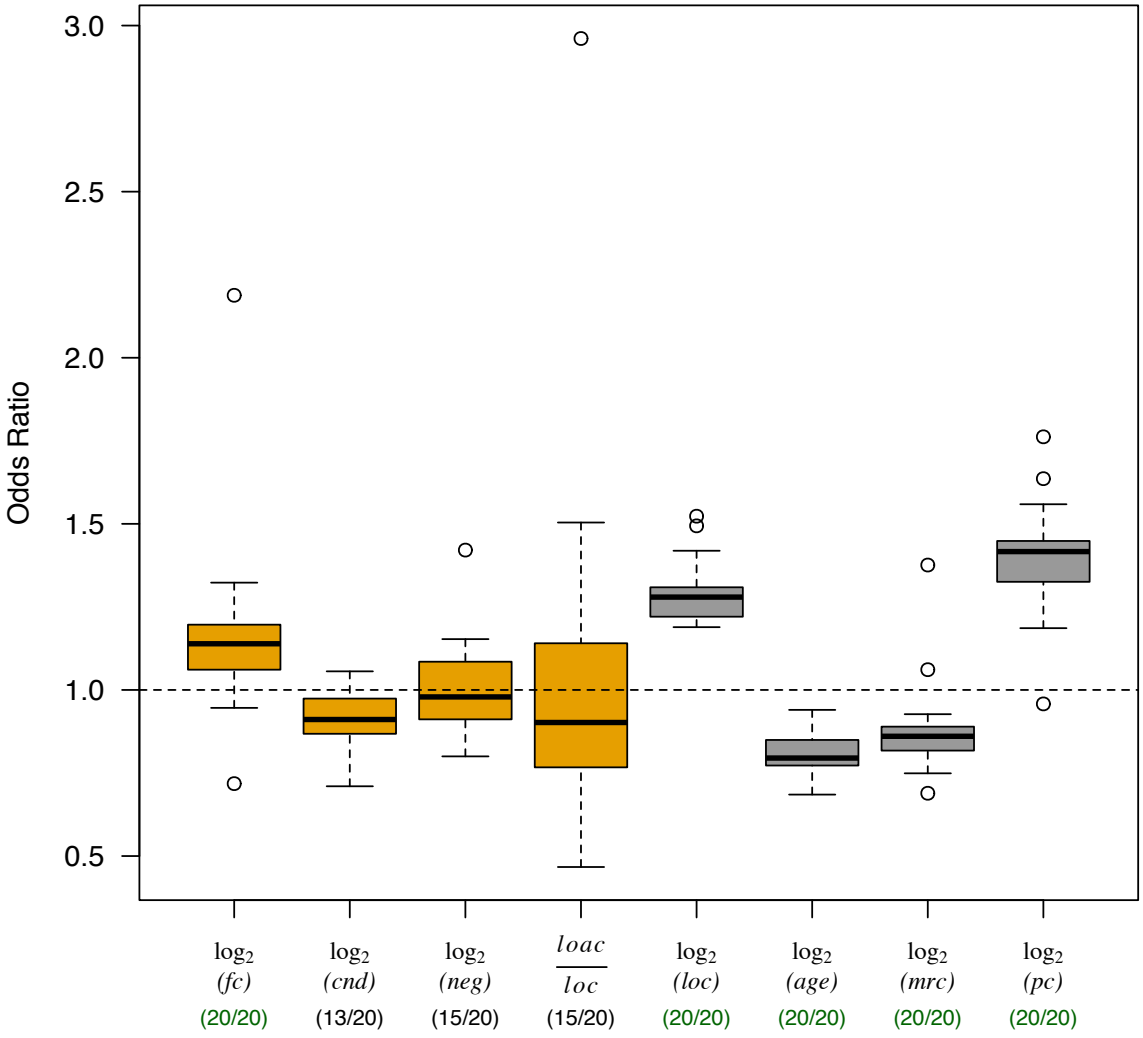


Figure 5.9: Summary of the regression results for change likelihood

corresponds to the odds 18:82 (18 functions changed vs. 82 not changed). Thus, the expected odds to change for a function with twice as many feature constant as the average function are $18{:}82 \times 1.14 \approx 20{:}82$. Expressed in terms of likelihoods, this corresponds to an increase of only 0.02 (from 0.18 to 0.20). Even a three-unit increase of $\log_2(fc)$, which corresponds to *fc* increasing from 0 to 7, only raises the likelihood of a change by 0.07 (from 0.18 to 0.25). Through manual inspection, I found that only 0.3 % of all functions in my subjects reference seven feature constants or more. Hence, such an increase in the number of feature constants must be considered extreme. As the data in Table 5.12 reveal, the ORs of the other preprocessor-related variables ($\log_2(fl)$, $\log_2(cnd)$, $\log_2(neg)$, $loac_{/loc}$) are even closer to a neutral effect than the OR of $\log_2(fc)$. Consequently, changes in these variables will affect the odds that a function is changed even less strongly than a change of $\log_2(fc)$. I conclude that realistic increases in my metrics of preprocessor use have very small effects on the likelihood that functions change.

In Table 5.12, I report how many of the significant effects were positive and how many were negative, and in Table 5.13, I complement this data with details about the minimum and maximum effects. The data reveal that every variable of preprocessor

	$\log_2$ *(fc)*	$\log_2$ *(cnd)*	$\log_2$ *(neg)*	$\frac{loac}{loc}$	$\log_2$ *(loc)*	$\log_2$ *(age)*	$\log_2$ *(mrc)*	$\log_2$ *(pc)*
Significant	20/20	13/20	15/20	15/20	20/20	20/20	20/20	20/20
+/−	**16**:4	3:**10**	7:**8**	4:**11**	**20**:0	0:**20**	2:**18**	**19**:1
Mean	1.14	0.90	1.01	0.96	1.29	0.80	0.87	1.39

Significant: number of subjects where effect was statistically significant; **+/−:** number of subjects where effect was positive/negative (only significant effects); bold font highlights effect direction that is in the majority; **Mean:** mean odds ratio (only significant effects)

Table 5.12: Average odds ratios in the regression models for change likelihood

use has positive effects in at least one subject and negative effects in at least one other subject. I thus conclude that there is no general pattern: It depends on the subject whether a given variable of preprocessor use is associated with an increase or a decrease in the likelihood of changes or whether there is no effect at all.

Variable	**Minimum**		**Maximum**	
	OR	**System**	**OR**	**System**
$\log_2(fc)$	0.72	Gnumeric	2.19	OpenVPN
$\log_2(cnd)$	0.71	Gnumeric	1.06	mpv
$\log_2(neg)$	0.80	Cherokee	1.42	libxml2
$loac_{/loc}$	0.47	OpenVPN	2.96	Gnumeric
$\log_2(loc)$	1.19	glibc	1.52	libxml2
$\log_2(age)$	0.69	SQLite	0.94	Subversion
$\log_2(mrc)$	0.69	Subversion	1.38	OpenVPN
$\log_2(pc)$	0.96	OpenVPN	1.76	Cherokee

Table 5.13: Minimum and maximum odds ratios in the regression models for change likelihood

Contrary to my expectations, the data in Table 5.12 reveal that the mean ORs of both $\log_2(cnd)$ and $loac_{/loc}$ are below 1, indicating a negative effect. In other words, higher nesting degrees of nesting or larger proportions of annotated code are associated with small decreases in the likelihood of changes. Regarding, $\log_2(neg)$ the data show an anomaly: Even though the majority of effects (eight out of fifteen) are negative, the mean effect is positive (OR = 1.01). The cause for this anomaly is the positive effect in libxml2 (OR = 1.42, see Table 5.13, also visible as an outlier in Figure 5.9), which is so strong that the weaker negative effects in other subjects are outweighed.

Control variables. All control variables have statistically significant effects in all subjects. I observe that the direction of the effects of the control variables conforms to my expectations. In particular, the effect of $\log_2(loc)$ is positive, meaning that given two functions, the longer function is more likely to be changed than the shorter one. The effects of $\log_2(age)$ and $\log_2(mrc)$ are negative, meaning that older functions

as well as functions whose most recent change is further in the past are less likely to be changed than younger functions or functions that were recently changed. Finally, the effect of $\log_2(pc)$ is positive, meaning that functions that were changed more often in the past are also more likely to be changed in the future.

The data about the number of positive and negative effects (see Table 5.12) indicate that the direction of the effects of the control variables is more consistent than it is for the variables of preprocessor use. Specifically, $\log_2(loc)$ always has positive effects, and $\log_2(age)$ always has negative effects. The results for the other two control variables are only slightly less consistent. In particular, $\log_2(mrc)$ has negative effects in eighteen subjects out of all twenty, and $\log_2(pc)$ has positive effects in nineteen. Through manual inspection of the individual results, I found the exceptions for $\log_2(mrc)$ to be OpenVPN (OR = 1.38) and Vim (OR = 1.06); for $\log_2(pc)$, it was OpenVPN (OR = 0.96).

A comparison of the ORs in Table 5.12 indicates that the preprocessor-related variable have smaller effects than the control variables. Taking $\log_2(fc)$ (OR = 1.14) and $\log_2(loc)$ (OR = 1.29) as the example, I see that the OR resulting from a two-unit increase of $\log_2(fc)$ is $1.14^2 \approx 1.30$, about the same effect as increasing $\log_2(loc)$ by one unit. A two-unit change in $\log_2(fc)$ means that a function references at least three feature constants, and a one-unit increase in $\log_2(loc)$ means that a function doubles in size. The question is, which increase is more realistic? Inspecting the data of my subjects, I found that 11 % of all functions are at least twice as long as the mean lenght of a function but that only 1.7 % of all functions reference three feature constants or more. Hence, I conclude that it takes big increases in the number of feature constants to achieve effects that are comparable to moderate increases in the size of a function. This answers the question how the statistical effects of *fc* and *loc* relate to each other, but the same question arises for other combinations of variables. I answer it in the following paragraphs.

Comparison of effect sizes. Relating the effect sizes of the preprocessor-related variables and the control variables to each other is difficult because all variables are measured in different units of measurement. I therefore recomputed the logistic regression models after *standardizing* the independent variables. A standardized variable X_i^* is derived from an unstandardized variable X_i by subtracting the mean value of X_i and dividing the result by the standard deviation of X_i. The effect of standardization is that all variables are centered around 0 and have a variance of 1. As a result, I can determine which independent variable has the greatest effect on the outcome by simply comparing the standardized regression coefficients with each other.

The summaries of the standardized logistic regression models are given in graphical from in Figure 5.10 and in numerical form in Table 5.14. The data reveal that first, the standardized ORs of the control variables differ more strongly from neutral than the standardized ORs of the variables of preprocessor use. I conclude from this observation that variations in the size and age of a function, the time since its last change, and the number of previous changes explain future changes better than differences in the extents of preprocessor annotations. Second, among the variables related to preprocessor use, $\log_2(fc)^*$ has the greatest effect whereas the

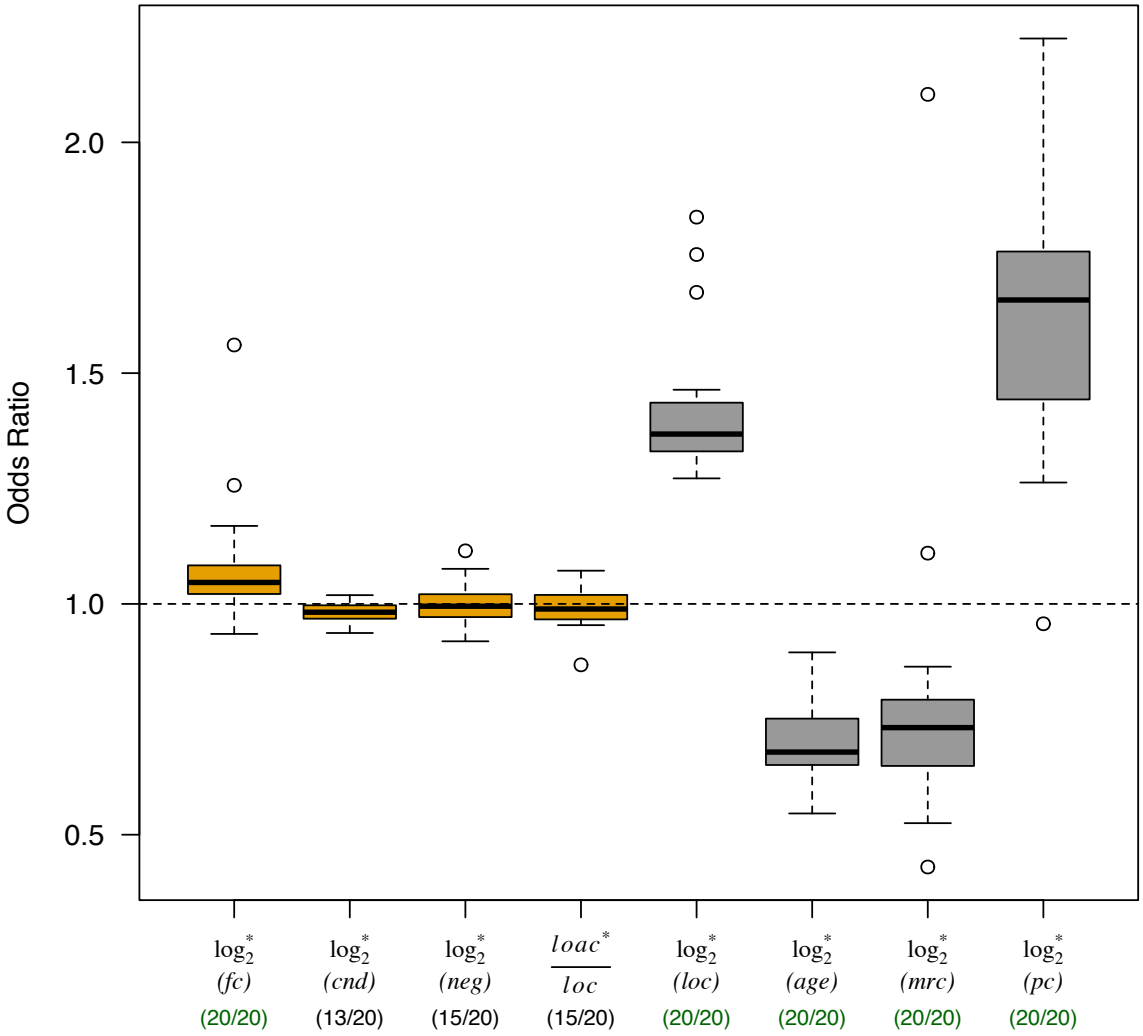


Figure 5.10: Summary of the regression results for change likelihood using standardized variables

others ($\log_2(fl)^*$, $\log_2(cnd)^*$, $\log_2(neg)^*$, and $loac_{/loc}{}^*$) have effects that are very close to neutral. I conclude from this second observation that variations in the number of feature constants explain future changes to a small extent but that variations in other preprocessor-related metrics have almost no explanatory power.

I analyzed the explanatory power of the preprocessor-related variables in more detail with the help of the McFadden statistics. To this end, I complemented the full

	$\log_2^*$ *(fc)*	$\log_2^*$ *(cnd)*	$\log_2^*$ *(neg)*	$\frac{loac*}{loc}$	$\log_2^*$ *(loc)*	$\log_2^*$ *(age)*	$\log_2^*$ *(mrc)*	$\log_2^*$ *(pc)*
Significant	20/20	13/20	15/20	15/20	20/20	20/20	20/20	20/20
+/−	**16**:4	3:**10**	7:**8**	4:**11**	**20**:0	0:**20**	2:**18**	**19**:1
Mean	1.07	0.98	1.00	0.99	1.41	0.69	0.74	1.59

Table 5.14: Average odds ratios in the regression models for change likelihood using standardized independent variables

regression models (i. e., the models discussed so far) with a set of reduced regression models, which only comprise the control variables. For the reduced models, the mean McFadden statistic, averaged over all subjects, is 0.136, and for the full models, it is 0.138. In other words, the control variables alone explain 13.6 % of the variance in the data, and the addition of the preprocessor-related variables improves this value by 0.2 %, yielding 13.8 %. I conclude that taking information about preprocessor use into account helps explain the likelihood of changes better, but only by a very small amount.

In summary, all four variables of preprocessor use, when considered in combination and in the context of my control variables, have a statistically significant effect on the likelihood of functions being changed in the majority of the subject systems. For *fc*, the majority of the significant effects is positive, and for *cnd*, *neg*, and $loac_{/loc}$, it is negative. Based on these results, I reject H_0 3.1 and instead accept the following alternative hypotheses:

> ***H_a 3.1 (fc):*** *After accounting for differences in function size, age, number of previous changes, and time since the last change, functions in which the preprocessor directives reference more feature constants are more likely to be changed in most but not all software systems than functions in which fewer feature constants are referenced.*
> *Depending on the system, the likelihood of being changed may also decrease as the number of referenced feature constants rises.*
> *For realistic changes in the number of feature constants, the effects are likely small.*

> ***H_a 3.1 (cnd, neg, $loac_{/loc}$):*** *After accounting for differences in function size, age, number of previous changes, and time since the last change, functions in which the preprocessor directives are more deeply nested or use more negation, as well as functions with a higher ratio of annotated code are less likely to be changed in most but not all software systems than functions in which these metrics have lower values.*
> *Depending on the system, the likelihood of being changed may also increase or not correlate with changes in these metrics.*
> *For realistic changes of these metrics, the effects are likely small.*

5.4.6 RQ 3, H_0 3.2: Relationship between Different Extents of Preprocessor Use and the Frequency of Changes

By testing H_0 3.2, I answer the question whether preprocessor use affects the frequency with which a function is changed. To answer this question, I computed negative binomial regression models for all subjects. Much like in Section 5.4.5, I summarize the models in Figure 5.11, Table 5.15, and Table 5.16. One important difference is the way in which I report effect sizes: In Section 5.4.5, I reported ORs, for which the value 1 corresponds to a neutral effect, but for the results related to H_0 3.1, I report the regression coefficients directly. For raw regression coefficients, the coefficient value 0 indicates a neutral effect, which is why the horizontal helper

line in Figure 5.11 appears at 0. The color coding below the x-axis in Figure 5.11 is the same as in Figure 5.9, with the addition that variables that do *not* have a significant effect in the majority of subjects are highlighted in red.

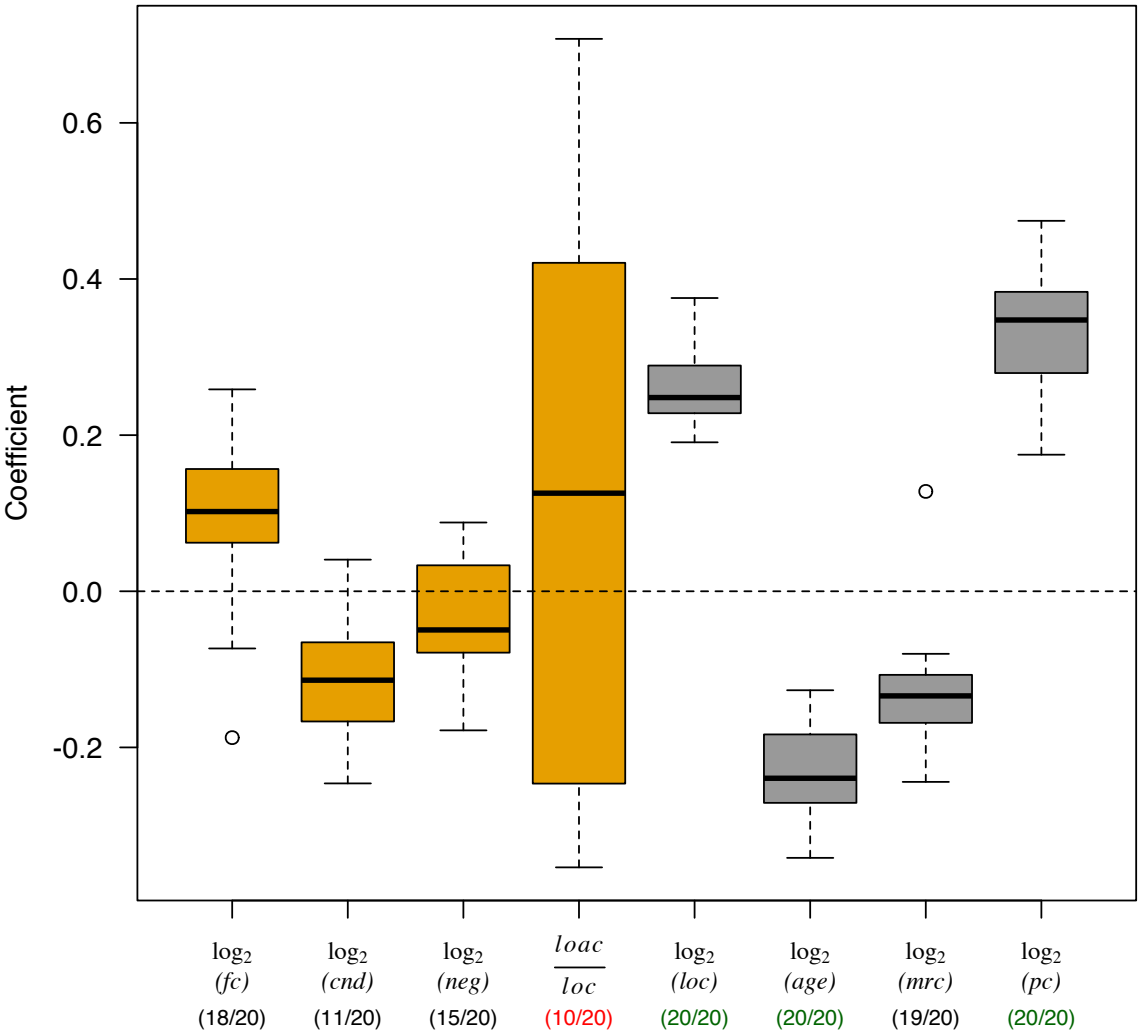


Figure 5.11: Summary of the regression results for change frequency

A visual comparison of Figure 5.11 with Figure 5.9 reveals that the results of negative binomial regression are similar to those of logistic regression. Specifically, most variables have statistically significant effects in the majority of systems, and once again, the control variables have significant effects more often than the preprocessor-related variables. Furthermore, for all variables except $loac_{/loc}$, I observe that the heights of the boxes relative to each other, as well as their position in relation to the line indicating the neutral effect are similar in Figure 5.9 and Figure 5.11. These observations allow me to draw two conclusions: first, whether the outcome is being changed or whether the outcome is the number of changes, the direction of the statistical effects of the independent variables is the same in both cases; second, the relative size of the effects is similar. For example, the absolute effect (i. e., the distance from a neutral effect) of $\log_2(fc)$ is again greater than the absolute effect of $\log_2(neg)$.

	$\log_2$ *(fc)*	$\log_2$ *(cnd)*	$\log_2$ *(neg)*	$\frac{loac}{loc}$	$\log_2$ *(loc)*	$\log_2$ *(age)*	$\log_2$ *(mrc)*	$\log_2$ *(pc)*
Significant	18/20	11/20	15/20	10/20	20/20	20/20	19/20	20/20
+/−	**15**:3	2:**9**	5:**10**	**6**:4	**20**:0	0:**20**	1:**18**	**20**:0
Mean	0.09	-0.10	-0.03	0.13	0.26	-0.24	-0.13	0.34

Table 5.15: Average effect sizes in the regression models for change frequency

Preprocessor use. The variables $\log_2(fc)$, $\log_2(cnd)$, and $\log_2(neg)$ have statistically significant effects in the majority of the subjects. The proportion of annotated lines of code, modeled by $loac_{/loc}$, has statistically significant effects in only half of the subjects. Contrary to its effect on the likelihood of changes, which was generally negative, the effect of $loac_{/loc}$ on the frequency of changes is generally positive (compare Table 5.12 and Table 5.15).

The interpretation of the coefficients of the variables of preprocessor use (see Table 5.15) reveals that the effect sizes are medium at best. As an example, consider the mean coefficient for $\log_2(fc)$, which is 0.09, and recall from Figure 5.2 that the mean number of changes to a function in one commit window (500 commits) is 0.36. Given these values, I can expect a function with twice as many feature constant as the average to change $0.36 \times (1 + 0.09) \approx 0.39$ times over a commit window. Put differently, an average function is changed once every 1389 commits, and with twice as many feature constants, it can be expected to change once every 1274 commits, a reduction by 8 %. Even at eight times as many feature constants (a three-unit increase of $\log_2(fc)$), the expected time until the next change is still 1072 commits, a reduction by 23 %. However, as discussed in Section 5.4.5, such an enormous increase in the number of feature constants is unrealistic. For these reasons, I argue that realistic differences in the extent of preprocessor use only have small effects on the expected frequency of changes to a function.

Variable	**Minimum**		**Maximum**	
	Coef.	**System**	**Coef.**	**System**
$\log_2(fc)$	-0.19	Gnumeric	0.26	OpenVPN
$\log_2(cnd)$	-0.25	Gnumeric	0.04	Pidgin
$\log_2(neg)$	-0.18	Cherokee	0.09	libxml2
$loac_{/loc}$	-0.35	Subversion	0.71	Gnumeric
$\log_2(loc)$	0.19	mpv	0.38	libxml2
$\log_2(age)$	-0.34	Gimp	-0.13	Cherokee
$\log_2(mrc)$	-0.24	Subversion	0.13	OpenVPN
$\log_2(pc)$	0.17	Pidgin	0.47	SQLite

Table 5.16: Minimum and maximum effect sizes in the regression models for change frequency

The summary of the effects directions in Table 5.15, as well as the minimum and maximum values of the regression coefficients in Table 5.16 indicate that the directions of the effects are inconsistent. For every variable, there are subjects in

which the effects are positive and subjects in which the effects are negative. Thus, the inconsistencies that I already witnessed in Section 5.4.5 (see Table 5.12) are repeated.

Control variables. Similar to the regression results regarding the likelihood of changes, the control variables $\log_2(loc)$ and $\log_2(pc)$ generally have positive effects, whereas $\log_2(age)$ and $\log_2(mrc)$ have negative effects. The direction of the effects is highly consistent across all subjects. The only exception is $\log_2(mrc)$, which has a negative effect in eighteen out of all twenty subjects, a positive effect in one, OpenVPN, and no significant effect in one, Vim. I note that OpenVPN and Vim are the same subjects for which the logistic regression results regarding $\log_2(mrc)$ also went against the general trend (see Section 5.4.5).

To illustrate the strength of the effects, I take $\log_2(loc)$ as the example. Assuming the mean number of commits to a function as the baseline, I can expect a function twice as long as the average to change $0.36 \times (1+0.26) \approx 0.45$ times per commit window. In other words, the time until the next expected change is reduced by 21 %, from 1389 to 1102 commits. Thus, doubling the size of a function has almost the same effect as increasing the number of feature constants by a factor of eight (23 % reduction). The difference is that overly long functions are relatively common whereas functions in which high numbers of feature constants are referenced are very rare.

I compared the McFadden statistics of the full and the reduced regression models to assess how much preprocessor-related information helps explain variations in change frequencies. I found that the reduced models explain 10 % of the variations, and the full models fare only slightly better, explaining 10.1 % of the variations.

Overall, the regression results for H_0 3.2 mirror the results for H_0 3.1 in many ways, such as which variables had statistically significant effects, which variables had positive or negative effects, and how big the absolute effects are in relation to each other. Furthermore, I found again that information about the extent of preprocessor use hardly adds any explanatory power if all the controlled factors are already accounted for. These similarities may be explained with the highly skewed distribution of *commits*: Averaged over all subjects, only 18 % of all functions are changed during the course of a commit window, and only 7.5 % (less than half of all changed functions) are changed twice or more. Therefore, considering the *number of changes* as the outcome is almost the same as considering the binary criterion *changed/not changed,* and for this reason, different regression techniques produce similar results.

In summary, three of my variables of preprocessor use (*fc*, *neg*, and *cnd*, but not $loac_{/loc}$), when considered in combination and in the context of my control variables, have a statistically significant effect on the number of times that functions are changed in the majority of the subject systems. Based on these results, I reject H_0 3.2 for *fc*, *neg*, and *cnd* and instead accept the following alternative hypotheses:

H_a 3.2 (fc): *After accounting for differences in function size, age, number of previous changes, and time since the last change, functions in which the preprocessor directives reference more feature constants are changed more frequently in most but not all software systems than functions in which these metrics have lower values.*
Depending on the system, the frequency of changes may also decrease or not correlate as the number of referenced feature constants rises.
For realistic changes in the number of feature constants, the effects are likely small.

H_a 3.2 (cnd, neg): *After accounting for differences in function size, age, number of previous changes, and time since the last change, functions in which the preprocessor directives are more deeply nested or use more negation are changed less frequently in most but not all software systems than functions in which these metrics have lower values.*
Depending on the system, the frequency of changes may also increase or not correlate with changes in these metrics.
For realistic changes of these metrics, the effects are likely small.

The evidence is insufficient to reject H_0 3.2 for $loac_{/loc}$. Instead, I continue to assume the following:

H_0 3.2 ($loac_{/loc}$): *After accounting for differences in function size, age, number of previous changes, and time since the last change, differences in the ratio of annotated code in a function have no consistent effect on the frequency with which a function is changed.*

5.4.7 RQ 3, H_0 3.3: Relationship between Different Extents of Preprocessor Use and the Extent of Changes

By testing H_0 3.3, I answer the question whether preprocessor use affects the extent of changes (measured in lines of code changed) that a function undergoes. I computed negative binomial regression models to answer this question and summarize the results in Figure 5.12, Table 5.17, and Table 5.18.

Preprocessor use. Compared to the results of testing H_0 3.2 (see Table 5.15), I observe that $\log_2(fc)$ and $\log_2(neg)$ continue to have statistically significant effects in the majority of the systems (see Table 5.17). For $\log_2(cnd)$ and $loac_{/loc}$, the situation has reversed: $\log_2(cnd)$ now has significant effects in a minority of the subjects (seven out of twenty), but $loac_{/loc}$ has significant effects in a majority (thirteen out of twenty).

Regarding the direction of the effects, the results also exhibit some similarities to the results for H_0 3.2, but also some differences. On the one hand, the average effect of $\log_2(cnd)$ remains negative and the average effect of $loac_{/loc}$ remains positive. On the other hand, the average effect of $\log_2(fc)$ has turned from positive to negative and the average effect of $\log_2(neg)$ has turned from negative to positive.

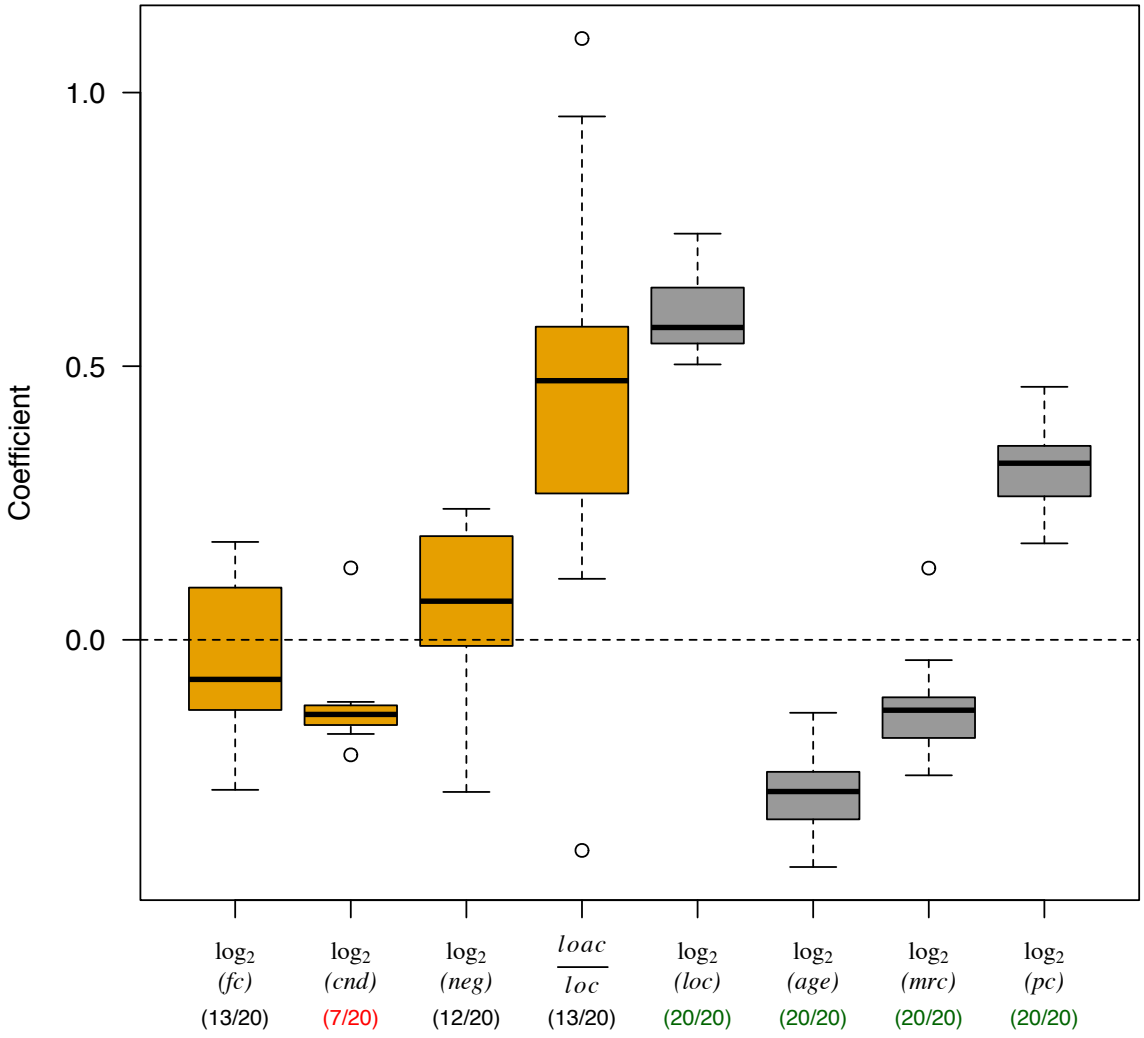


Figure 5.12: Summary of the regression results for change profundity

I use $\log_2(neg)$ as an example to illustrate the strength of the effects. The average coefficient of $\log_2(neg)$ is 0.06 (see Table 5.17), and the average number of changed lines in a function during one commit window is 3.64 (see Figure 5.2). Based on these values, the number of changed lines in a function that contains twice as much negation as the average can be expected to increase from 3.64 to $3.64 \times 1.06 \approx 3.86$. I argue that this difference is hardly noticeable in practice. Based on a comparison of the (unstandardized) regression coefficients, the absolute effect of $loac_{/loc}$ appears stronger than the effects of the other variables of preprocessor use. However, $loac_{/loc}$ stands for the proportion of annotated lines of within a function. Thus, its maximum

	$\log_2$ *(fc)*	$\log_2$ *(cnd)*	$\log_2$ *(neg)*	$\frac{loac}{loc}$	$\log_2$ *(loc)*	$\log_2$ *(age)*	$\log_2$ *(mrc)*	$\log_2$ *(pc)*
Significant	13/20	7/20	12/20	13/20	20/20	20/20	20/20	20/20
+/−	6:**7**	1:**6**	**9**:3	**12**:1	**20**:0	0:**20**	1:**19**	**20**:0
Mean	-0.02	-0.11	0.06	0.44	0.59	-0.28	-0.13	0.32

Table 5.17: Average effect sizes in the regression models for change profundity

value is 1.0 or 100 %. For a realistic increase of $loac_{/loc}$ by 10 %, the expected number of changed lines in a function only increases to $3.64 \times (1 + 0.44)^{0.1} \approx 3.78$.

The data about the direction of the effects and the minimum and maximum effect sizes (see Table 5.17 and Table 5.18) once again reveal that the average trends regarding the direction of the effects do not hold for all subjects. Every variable has positive effects in at least one subject and negative effects in at least one other subject. As for the results related to H_0 3.1 and H_0 3.2, there is no general pattern regarding the direction of the effects.

Variable	Minimum		Maximum	
	Coef.	System	Coef.	System
$\log_2(fc)$	-0.27	Pidgin	0.18	Gnumeric
$\log_2(cnd)$	-0.21	Gnumeric	0.13	mpv
$\log_2(neg)$	-0.28	Subversion	0.24	glibc
$loac_{/loc}$	-0.38	php	1.10	libxml2
$\log_2(loc)$	0.50	mpv	0.74	libxml2
$\log_2(age)$	-0.42	Gimp	-0.13	Vim
$\log_2(mrc)$	-0.25	OpenLDAP	0.13	OpenVPN
$\log_2(pc)$	0.18	libxml2	0.46	BusyBox

Table 5.18: Minimum and maximum effect sizes in the regression models for change profundity

Control variables. The results for the control variables show the same trends as for H_0 3.1 and H_0 3.2. All of them have significant effects on the extent of changes in all subjects, and the average directions of the effects are the same. Moreover, the data in Table 5.17 reveal that the directions of the effects are highly consistent from subject to subject. The only inconsistent result is related to $\log_2(mrc)$, which has a negative effect in nineteen subjects, but a positive effect in one subject, OpenVPN.

I use $\log_2(loc)$ as an example to illustrate the strength of the effects. According to Table 5.17, the average coefficient of $\log_2(loc)$ is 0.59. Thus, I can expect the number of lines changed in a function that is twice as long as the average function to be $3.64 \times (1 + 0.59) \approx 5.79$, which is an increase of 2.15 lines or 59 %. I conclude from this example that realistic changes in the control variables tend to produce noticeable changes in the number of lines changed. By comparison, the effects of realistic changes in the variables of preprocessor use are much more subtle. The McFadden statistics of the full and the reduced models confirm this conclusion: The average McFadden statistic is the same for both models, 0.04. In other words, taking only the controlled factors into account, I can explain 4 % of the variance. While this is an unsatisfactory result, additionally taking information about preprocessor use into account fails to yield any improvement.

In summary, three of my variables of preprocessor use (fc, neg, and $loac_{/loc}$, but not cnd), when considered in combination and in the context of my control variables, have a statistically significant effect on the extent to which functions are changed in the majority of the subject systems. Based on these results, I reject H_0 3.3 for fc, neg, and $loac_{/loc}$ and instead accept the following alternative hypotheses:

H_a 3.3 (neg, loac$_{/loc}$): *After accounting for differences in function size, age, number of previous changes, and time since the last change, functions in which the preprocessor directives use more negation, as well as functions with a higher ratio of annotated code are changed more profoundly in most but not all software systems than functions in which these metrics have lower values.*
Depending on the system, the profundity of changes may also decrease or not correlate with changes in these metrics.
For realistic changes in the values of these metrics, the effects are likely small.

H_a 3.3 (fc): *After accounting for differences in function size, age, number of previous changes, and time since the last change, functions in which the preprocessor directives use reference more feature constants are changed less profoundly than functions containing fewer or no preprocessor directives.*
Depending on the system, the profundity of changes may also increase or not correlate as the number of referenced feature constants rises.
For realistic changes in the number of feature constants, the effects are likely small.

The evidence is insufficient to reject H_0 3.3 for *cnd*. Instead, I continue to assume the following:

H_0 3.3 (cnd): *After accounting for differences in function size, age, number of previous changes, and time since the last change, functions in which the preprocessor directives are more deeply nested are changed just as profoundly as functions in which fewer nested preprocessor directives occur.*

5.5 Qualitative Analysis

The quantitive results presented in the previous sections and, in particular, the low explanatory power of my regression models suggest there are more reasons why particular functions are change-prone while others remain stable. To gain a better understanding of these reasons, I qualitatively analyzed the change histories of a number of functions in my dataset, putting the focus on the role that CPP use played in their change proneness or stability. Since such an analysis is time consuming, I had to restrict myself to a small sample of functions. At the same time, I wanted variation in the sample so it covers different points in the space spanned by the predictor and predicted variables considered in my study. To create a sample that satisfies both criteria, I selected eight functions that represent extreme points along the following three dimensions: (1) extent of preprocessor use (indicated by the number of CPP directives or feature constants), (2) function size (an important controlled factor according to my quantitative results), and (3) change proneness (as indicated by the number of changes). Thus, my sample comprises functions that were either short or long in combination with making light or heavy CPP use, and for every combination, it encompasses one function that changed rarely and one that changed frequently.

5.5.1 Short Functions with Heavy cpp Use

I found several short functions in my subjects that make heavy use of CPP directives. A change-prone example is function `usage` from OpenLDAP, which prints a text message explaining how to use OpenLDAP's `slapd` command line tool (see Listing 5.2). At revision d8cb38c2e from May 1999, this function comprised 19 SLOC and four preprocessor directives that referenced five feature constants. The reason for the unusually large extent of CPP use in this function is that the `slapd` tool possesses or lacks certain command line options depending on how OpenLDAP is configured and compiled. During the commit window starting on May 1999, `usage` was changed six times. Three of those changes were related to support for running `slapd` as a Windows NT service. This support was added in revision 4d13d40, retracted in revision 23c4b81 because it broke Unix builds, and then recommitted in b0aea66 with changes that resolved the build errors. During each of these changes, the command line interface of `slapd` was changed, and `usage` was modified to reflect the change. The subsequent revisions (49d9c99, 669b8f4, and 1708367) follow the same pattern: Each revision changed the command line interface of `slapd`, thus requiring a corresponding change of the function. The underlying reason for the frequent changes is that the implementation of `slapd`'s command line interface is scattered across different functions: `usage` is responsible for explaining the command line interface to the user, but the actual parsing of command line options happens in `main`. The presence of `#ifdef`s in both of these functions is an indicator of this scattering. Nevertheless, I see no causal relationship between the presence of `#ifdef`s in `usage` and the extent to which it is changed.

```
1  static void
2  usage( char *name )
3  {
4    fprintf( stderr, "usage: \%s [-d ?|debuglevel] [-f configfile]
       [-p portnumber] [-s sysloglevel]", name );
5      fprintf( stderr, "\n        [-a bind-address] [-i]" );
6  #if LDAP_CONNECTIONLESS
7    fprintf( stderr, " [-c]" );
8  #endif
9  #ifdef SLAPD_BDB2
10     fprintf( stderr, " [-t]" );
11 #endif
12 #ifdef LOG_LOCAL4
13     fprintf( stderr, " [-l sysloguser]" );
14 #endif
15 #if defined(HAVE_SETUID) && defined(HAVE_SETGID)
16     fprintf( stderr, " [-u user] [-g group]" );
17 #endif
18     fprintf( stderr, "\n" );
19 }
```

Listing 5.2: A short, change-prone function from OpenLDAP making heavy use of CPP directives for configurability

In contrast to the change-prone example from OpenLDAP, I found short functions that remained very stable despite comprising many CPP directives. One of them is `get_realtime` from MPV, shown in Listing 5.3, which comprises 11 SLOC and two

```
1  static void get_realtime(struct timespec *out_ts)
2  {
3  #if defined(_POSIX_TIMERS) && _POSIX_TIMERS > 0
4      clock_gettime(CLOCK_REALTIME, out_ts);
5  #else
6      // OSX
7      struct timeval tv;
8      gettimeofday(&tv, NULL);
9      out_ts->tv_sec = tv.tv_sec;
10     out_ts->tv_nsec = tv.tv_usec * 1000UL;
11 #endif
12 }
```

Listing 5.3: A short, stable function from MPV making heavy use of CPP directives for configurability

feature locations referencing one feature constant. Except for the function signature, every line of code is feature code. The function was created in 2013 by commit b78d11d328 under the name `get_pthread_time` in `osdep/threads.c`. Since then, the only change to this function happened in commit f47a4fc3d9 from 2014, when it was renamed to `get_realtime` and moved to a different file, `osdep/timer.c`. As of commit 8b563a0346 from 2019, the function still exists in that file. Timer-related functionality strongly depends on the operating system, which explains why this short function makes relatively heavy use of CPP directives. However, operating system developers try to keep their interfaces stable, which explains why the function did not change.

5.5.2 Long Functions with Heavy cpp Use

The majority of heavily annotated, change-prone functions I found were much longer than OpenLDAP's `usage`. Many of these long functions were `main` functions, which serve as the entry point to a C program. In MPV (formerly MPLAYER), I found an extreme example. In October 2005, at revision d28ad7d31d, the `main` function in `mplayer.c` was 2829 SLOC long and contained 131 feature locations that referenced 35 distinct feature constants. The function performed all kinds of tasks, such as reading configuration files, setting up signal handlers, loading media files, and handling the main event loop of the GUI. Blocks of code that were responsible for different tasks were separated by comments. MPV has always offered a large number of compile time configuration options, including support for various operating systems (e. g., Linux, Windows), CPU instruction set extensions (e. g., MMX, SSE2), and media file formats (e. g., Ogg, Matroska). These configuration options must be implemented somewhere in the code, and since MPV's `main` function implemented so much functionality, I am not surprised to find so many `#ifdef`s in its body. I also believe that the sheer wealth of functionality in `main` is the principal reason why this function was changed so frequently. During 2005, when it was at its peak size of almost 3000 SLOC, I frequently observed 30 changes or more per commit window. For example, when MPV's GUI was adapted to support different languages in November 2005, commit d291ddf016 changed `main` in 21 places. Later that month, in commit f99cc19968, much of the GUI-related code was extracted to separate functions, causing changes in 54 places. In 2006, `main` underwent major restructuring,

and once its size had stabilized at around 1200 SLOC, the number of changes per commit window dropped to around 16. Starting in 2012, `main` underwent a second round of restructuring, shrinking to 52 and finally to 4 SLOC. With all of its functionality extracted to other functions, `main` hardly changed anymore afterwards. In summary, I believe that the change proneness of MPV's old `main` function resulted from poor separation of concerns. The large number of `#ifdef`s and feature constants certainly was an indicator of this problem, but there were other tell-tale signs, such as the the extreme length, the use of comments to structure the code, and the high number of previous changes. For these reasons, I do not regard the presence of CPP directives in MPV's `main` as the main cause of its change proneness.

In QEMU, I found a stable counterexample to MPV's change-prone `main` function. Its name is `dsound_log_hresult`, and it is part of QEMU's DirectSound drivers for older Microsoft operating systems. The function is 138 SLOC long and most of the code belongs to a `switch` statement that translates numerical error codes into human-readable messages. Out of the 26 `case` labels, 25 are guarded by `#ifdef`s because the presence of most error codes is subject to compile-time variability. `dsound_log_hresult` was introduced with revision 1d14ffa97ea in 2005 and never changed afterwards. At this point in time, DirectSound had already existed for ten years, and I assume its *Application Programmer Interface (API)* was stable. Three years later, DirectSound was deprecated. I believe that this is the main reason why `dsound_log_hresult` never had to be changed, despite its length and extensive use of CPP directives.

5.5.3 Short Functions with Light cpp Use

In CHEROKEE, I found two examples of short functions that made no use of CPP directives, one of them change-prone and the other one highly stable. The change-prone function is `cherokee_handler_proxy_free`, which never encompassed more than 15 SLOC during its lifetime, but was changed fourteen times, adding up to a total of 83 changed lines. The function is a destructor that is responsible for freeing resources held by the data structure `cherokee_handler_proxy_t`. As expected, some of the changes to the destructor stem from changes to the data structure. For example, commit 4c96e5b2 renamed a field and commit 51a4ed0c added four new fields. Both commits performed corresponding changes to the destructor. However, I also found other kinds of commits. For example, commit af114041 only renamed a parameter, and commit 28401e17 introduced a feature enhancement that was removed again in commit 3a9c1d38. These changes show that the relationship to a data structure is not the only reason for a destructor to change.

The stable function is `cherokee_handler_init`, also related to a data structure, but in the role of a constructor. The function comprises 11 SLOC and has existed since revision ae9d9717 from April 2006 and continues to exist in the lastest revision that I analyzed, revision 9a75e65b from July 2018. During this time, only its whitespace was changed, which happend in revision 72f644413. In summary, these two functions demonstrate that tight coupling to a data structure sometimes may make a function change-prone, but not necessarily.

5.5.4 Long Functions with Light cpp Use

In Cherokee, I found function `process_active_connections`, a representative of a long change-prone function making little use of cpp directives. Specifically, this function encompassed up to 530 SLOC during its existence but never contained more than one `#ifdef`. On average, the function changed 29 times per commit window, resulting in 130 revisions overall. An HTTP connection in Cherokee goes through several phases, such as setup, header processing, and shutdown. All of these phases are handled by a long `switch` statement in `process_active_connections`, and many of the cases in this `switch` contain further, nested `switch` statements. Thus, the function has numerous reasons to change, which explains why it changed so frequently.

A counterexample from MySQL shows that long functions with a high cyclomatic complexity are not necessarily change-prone: At revision 4e7a3d8b955, I found function `ieee_read_cxx_class` in file `pstack/ieee.c`, which is part of a library that MySQL uses to handle segmentation faults. The function encompasses 541 SLOC, 63 `if` statements, 3 `switch` statements with 12 cases, but not a single `#ifdef`. Despite its complexity, the function never changed since its creation ten years earlier, in revision 8f62579c38e8. According to the header of `pstack/ieee.c`, the code was copied from the GNU Binutils project, which suggests that it was already mature when it appeared in MySQL. Thus, it is plausible that `ieee_read_cxx_class` never changed.

5.5.5 Summary of Qualitative Findings

My qualitative analysis lead to four main insights, which I summarize as follows.

Concern tangling. First, concern tangling may be a cause for change proneness because each concern that is present in a function is a potential reason why the function needs to be changed in the future. Concern tangling is sometimes indicated by the presence of cpp directives, but alternative indicators exist, such as a high number of SLOC, comments to structure the code, or a high number of previous changes.

Stability of external interfaces. Second, the stability of external interfaces fosters the stability of functions depending on those interfaces. This is true even if the presence of cpp directives suggests that multiple concerns are tangled in a single function.

Coupling to data structures. Third, coupling to a data structure can lead to change proneness, but it does not have to. For example, refactorings and experimental feature enhancements are also reasons why functions change.

High cyclomatic complexity. Finally, a high cyclomatic complexity may be a sign of concern tangling and of mixing different levels of abstraction in a single function. Therefore, a high cyclomatic complexity can be an indicator of change proneness. However, there are circumstances in which such code remains stable, for example, if mature code is cloned from an external source.

In summary, my qualitative analysis uncovered many reasons why functions might be stable or change-prone. However, the presence or absence of CPP directives was not among them.

5.6 Threats to Validity

In this section, I discuss possible threats to the validity of my findings and how I mitigated these threats.

5.6.1 Internal Validity

My statistical analyses can only reveal *correlations* between preprocessor use and change proneness. However, even where I found correlations, I cannot claim that preprocess use *cause* change proneness. On the other hand, the *lack of correlations,* which I frequently observed, strongly suggests the *lack of a causal relationship* between preprocessor use and change proneness, and my qualitative findings corroborate this assessment. My methodology is based in large part on studies of the effects of *code smells* on change or fault proneness (e. g., [273, 170, 124, 319]). Although I investigate different phenomena (preprocessor use, not code smells), the underlying questions are similar, and the same methods apply. Thus, I at least conform to the state of the art.

Most statistics are sensitive to the distribution of the data. Where possible, I used robust statistical tests so as not to violate any assumptions about the analyzed data. I chose logistic and negative binomial regression as my regression techniques and transformed the involved variables based on analyses of distribution characteristics and additional statistics on model fitness. Others used such models and transformations for similarly shaped data [353, 280, 109, 170, 124, 101].

In my study, the frequency and the extent of changes serve as proxies for maintenance effort. This is common in software engineering research (e. g., [62, 64, 216, 273]), but has known limitations. For instance, preprocessor directives may hinder program comprehension, and as a result, changing one line of feature code may require more effort than changing one line of other code. Since I cannot measure program comprehension with my methodology, I only claim that the use of preprocessor annotations is largely unrelated to the frequency and extent of changes *as per version control information.*

Developers may be reluctant to change feature code and choose workarounds instead. If such a workaround theory were true, preprocessor use would indirectly increase maintenance effort in a way that my methodology cannot detect. However, in their study of the influence of code smells on maintenance effort, [338] observed that rather than working around smells, developers remove them during maintenance. It is plausible that developers act in the same way when they encounter overly complex annotated code. The workaround theory is therefore unlikely.

Bugs in the tools I developed and in third-party tools could confound my analyses. I mitigate this threat by relying on mature tools where possible (SRC2SRCML [57], CPPSTATS [204, 142], REPODRILLER[4] (formerly known as METRICMINER [341]),

[4]https://github.com/mauricioaniche/repodriller

and EGit[5]). I checked for bugs in my own tools using regression tests and sample-based inspection of output data. A small number of files (<0.1 %) could not be parsed due to errors in my tool-chain. This is unlikely to skew my data to any relevant degree.

I collect data using a snapshot technique, which entails some imprecisions. For example, the preprocessor metrics of a function could change considerably in the course of a snapshot. Moreover, my heuristics to detect renamed and moved functions are not perfect: They produce a small proportion of false-negative results (renames that go undetected) and false-positive results (changes incorrectly classified as renames). As a consequence, the change histories of some functions end prematurely while in other cases, the change histories of unrelated functions are erroneously merged together. However, I carefully investigated these threats in preliminary experiments (see Section 5.2.5). To mitigate the first threat (missed renames), I chose the snapshot size accordingly and combined it with a sliding window technique. Other studies of change proneness use releases of a software system as snapshots (e. g., [69, 170, 124]). Since my snapshots are much shorter than typical release cycles, my analysis is at least as precise as the current state of the art. To mitigate the second threat (treating unrelated functions as renamed versions of each other), I implemented a comprehensive post-processing step (a major extension compared to my original study [98]) and tuned the thresholds of my rename detection heuristics to minimize the number of false-negative and false-positive results. In particular, I manually inspected a great number of potential renaming changes in several subject systems. Based on this inspection, I chose 60 % similarity (in terms of Levenshtein distance) as the threshold for determining whether a change is a rename or not. However, I did not formally measure precision and recall during this process and so a different threshold may have lead to more accurate results. Any remaining imprecisions will affect functions with and without CPP directives equally. Hence, my statistics and conclusions remain valid.

5.6.2 External Validity

I cover many well-known aspects of preprocessor use, e. g., the number of preprocessor directives, nesting and negation. Nevertheless, I miss some aspects, such as annotation discipline. I cannot generalize my findings to these aspects.

Software systems differ in how they use preprocessor directives and how changes are performed, depending on their domain. I mitigated these threats by choosing systems that differ in size and domains. All of my subject systems are open-source; no industrial systems were analyzed. The only exception is Blender, which evolved from a closed-source to an open-source system. Mitigating the bias toward open-source systems, Hunsen et al. showed that preprocessor use is the same for open- and closed-source systems [142]. Hence my results should be generalizable to at least other systems in the same domains, both open- and closed-source.

I only consider subjects written in C and using the CPP. I expect my findings to be generalizable to other procedural languages and other preprocessors that implement conditional compilation similarly to the CPP.

[5] www.eclipse.org/egit/

5.7 Discussion

The CPP has been criticized as the source of many maintenance problems, such as making code difficult to understand, change, and prone to subtle bugs [343, 92, 89, 206, 239, 238, 240, 221, 243]. In this chapter, I studied the change proneness of functions in C programs to judge whether CPP use is harmful or harmless. Change proneness can either mean that code is changed frequently, which has been shown to raise the likelihood of faults, or it can mean that code is changed to a greater extent, which has been shown to increase maintenance effort [84, 123, 86, 255, 338]. I empirically investigated both flavors of change proneness using repository mining techniques and now discuss my results.

For the first research question, I distinguished between four aspects of CPP use that I modeled as binary criteria: (1) whether a function contains at least one CPP directive, (2) whether the CPP directives in a function reference at least two feature constants, (3) whether a function contains at least on nested CPP directive, or (4) whether a function contains at least one negated CPP directive (e. g., an `#ifndef` or an `#else` directive). For each criterion, I separated functions into one group that fulfilled the criterion and another group that did not and examined which group of functions was more change-prone. My results show that functions fulfilling the criterion (i. e., exhibiting the corresponding aspect of CPP use) are more likely to change and they also change more often and more profoundly. Given the known, undesirable effects of change proneness, my findings suggest that functions exhibiting any of these four aspects of CPP use are more fault-prone and require more maintenance effort. Thus, I answer RQ 1 as follows:

> ***RQ 1:*** *Code containing* CPP *directives is more likely to change, is changed more often, and is changed more profoundly than other code.*

The answer to RQ 1 seems to confirm the existing critique of the CPP, but my results regarding research questions two and three call into question whether there really is a relationship between CPP use and maintenance problems. Size is a well-known confounding factor in studies on change and fault proneness [208, 87, 390, 273, 123, 338]. My quantitative results show that CPP use and function size correlate positively, meaning long functions are more likely to contain CPP directives, and conversely, functions containing CPP directives also tend be long. Thus, I answer RQ 2 as follows:

> ***RQ 2:*** *Function that use* CPP *directives to a greater extent tend to be longer than other functions. This is true for the number of* CPP *directives in the function body, the number of feature constants, the extent of nesting and negation in* CPP *directives, and the proportion of feature code.*

Hence, I controlled for size and for three other possible confounders: age, time since the last change, and number of previous changes to a function. After controlling for all four factors, I found that different extents of preprocessor use still have a statistically significant effect on change proneness, but this effect is limited in several

ways: Depending on the subject system, each aspect of preprocessor use may either increase change proneness, decrease change proneness, or have no statistically significant effect at all. The effects can be very strong in some subjects but negligible in others. In contrast to the inconclusive findings regarding CPP use, I observed that differences in the size, age and number of previous changes of a function, and in the time since the last change have much more consistent effects. Furthermore, realistic changes in these factors have much stronger effects than realistic changes in the extent of CPP use. Thus, I answer RQ 3 as follows:

> ***RQ 3:*** *The relationship between* CPP *use and maintainability lacks a clear, generalizable pattern. Whether or not code in which* CPP *directives are used to a greater extent is harder to maintain than other code strongly depends on the software system and the respective aspect of* CPP *use. The differences in maintainability caused by different extents of* CPP *use are likely small.*

I conclude that the extent of CPP use in a function is a very unreliable predictor of future bugs and maintenance effort. This conclusion is in line with other studies on the relationship between potentially harmful programming patterns and maintenance problems, such as increased effort and fault proneness. Notably, Hall et al. found that many object-oriented code smells have only small effects on fault proneness, and depending on the smell and the software system, the presence of the smell may decrease fault proneness [124]. Sjøberg et al. reported similarly inconsistent results in a study of the relationship of code smells on maintenance effort [338].

As the McFadden statistics of my regression models revealed, considering the amount of preprocessor use in a function yielded little if any improvement of the models' quality, which again underlines the unreliability of CPP use as a predictor for maintenance problems. Averaged over the data of all subjects, only 11 % of all functions contain CPP directives, and only 6 % contain more than one. In other words, functions in which the CPP is heavily used are rare, and my results suggest that if such functions are hard to maintain, then it is not because CPP directives make them change prone. Therefore, any negative effects of CPP use on maintainability are likely related to program comprehension, which several studies have shown to suffer in the presence of CPP directives [198, 243, 221, 238, 240].

The McFadden statistics of my regression models also suggest something else: For all three outcomes, the results were low, ranging from 0.043 to 0.138. In other words, my models generally predict the change proneness of functions very poorly. Other work has uncovered additional reasons of change proneness, such as fault fixes and API refactorings [255, 71, 176]. In the course of my own qualitative analysis, I encountered a great variety of such potential reasons. For example, I found that the presence of CPP directives may indicate an underlying problem, such as concern tangling, that may be a cause for change proneness. However, in the examples I inspected, concern tangling was not always accompanied by the presence of CPP directives and conversely, CPP directives were not always accompanied by concern tangling. Moreover, I found other possible indicators for tangled concerns, such as large code size or the use of comments as a structuring aid. Given that I only inspected the histories of eight functions, my qualitative results are no more than

anecdotal evidence, and thus, must be taken with a grain of salt. Nevertheless, they agree with my quantitative results, which suggest that that variations in the extent of preprocessor use are not a major driver for change proneness.

My findings have at least four implications that are relevant to the practitioner. First, adding a CPP directive to a function or increasing the complexity of the existing CPP annotations is likely harmless. Second, shortening a function, e. g., by performing an *extract method* refactoring, will increase its maintainability more than reducing the number of CPP directives in the function's body. Third, if the goal is to identify functions that may cause maintenance problems in the future, it is more reliable to look at its size and examine how it changed in the past than to measure the extent of CPP use. Finally, further studies into the true causes of change proneness as well as tools to identify these causes will likely benefit practitioners in reducing faulty changes and maintenance effort more than analyses focussed on CPP use. Nevertheless, practical tools for C programmers must take the CPP into account, and academia already developed corresponding prototypes, such as TypeChef [164] and Morpheus [205]. My tool infrastructure, too, may benefit practitioners, who could use it to reconstruct a C function's change history and thus predict future maintenance hotspots. And yet, many tools in industrial practice are very limited in their support of CPP directives, thus forcing practitioners to make maintenance decisions based on incomplete or unsound analyses. As a result, tasks that are simple in other programming languages require more effort and are at a higher risk to be completed with errors.

5.8 Related Work

Three avenues of research are related to mine. First, my work builds on and complements previous studies of problems arising from preprocessor usage and static code configurability. Second, my study has profited from and confirms exiting work on preprocessor usage. Third, there are many studies of the relationship between various static source code properties (e. g., object-oriented metrics and code smells) and maintenance problems. My work adds preprocessor usage metrics to that body of knowledge.

C preprocessor usage and variability-related problems. Several researchers studied empirically how CPP use relates to fault proneness and code comprehension. Syntax errors caused by an incorrect use of annotations were found to be rare, but once introduced, they are particularly long-lived [239, 238, 241]. Moreover, developers perceive CPP-related bugs as easier to introduce, harder to fix, and more critical than other bugs. Melo et al. showed that developers find bugs more slowly and less precisely when the amount of variability increases [243]. Other findings suggest that functions with security vulnerabilities use CPP directives to a greater than non-vulnerable functions [101].

Another line of work explored the use of colors to support or replace CPP-based variability [155, 93, 198]. Specifically, highlighting CPP-annotated code with background colors helps program comprehension in some (but not all) situations [93]. Others propose a combination of background colors and virtual separation of concerns [155]

as an alternative to CPP-based variability [198]. Experiments showed that this alternative improves the efficiency and correctness of program comprehension compared to using plain CPP directives.

It is an ongoing debate whether *undisciplined* annotations matter with regards to the speed and precision of bug-finding. Such annotations encompass only parts of a syntactical unit, for example, a parameter in a function declaration. A previous study by Schulze et al. suggested that discipline does nots matter [327], but newer studies suggest it does [221, 240]. It also matters to developers: They prefer disciplined annotations [238, 240, 221].

These studies relate preprocessor use to program comprehension and fault proneness. I complement this work with quantitative and qualitative empirical findings on a different maintenance aspect, namely change proneness.

C preprocessor use in general. Other work analyzed CPP use in highly configurable software, for instance, with respect to scattering and tangling [89, 204, 206, 142, 298, 289] or artifact co-evolution [74, 290, 267]. They do not relate CPP usage to maintenance, as I do. Nevertheless, I build on some of their tooling, and their insights into the statistical distributions of CPP usage metrics helped us choose appropriate statistical tests.

Object-oriented metrics and code smells. Many studies try to predict and explain maintenance problems with the help of object-oriented metrics, such as coupling and cohesion, or code smells (e. g., [308, 170, 273, 123, 124, 338, 272, 319]). The findings are mixed. For example, some studies report that (combinations of) code smells clearly have negative effects (e. g., [170, 272, 319]) whereas others observe no such relationships or only weak ones (e. g., [273, 338, 124]). Even though the investigated properties (e. g., code smells) are different from the ones I study (use of C preprocessor directives), the methodologies are similar. More interestingly, the secondary findings of these studies closely mirror my own. Specifically, the simple metric of code size has repeatedly been identified as a comparatively reliable predictor of future maintenance effort and faults [87, 273, 123, 124, 338]. Moreover, multiple studies acknowledge that metrics and code smells alone are insufficient to properly explain maintenance problems [123, 381, 384, 379, 271]. Instead, it is necessary to take many other factors into account.

5.9 Conclusion

One of the variability-aware code smells that I propose in my thesis is the Annotation Bundle. In the previous chapter, I presented a metrics-based detection approach for the Annotation Bundle and a case study that demonstrated that this smell sometimes negatively affects program comprehension. The study described in the present chapter was conducted to complement these findings in two ways. The first goal was to improve the detection formula for Annotation Bundles by investigating the corresponding metrics in more detail. The second goal was to gather evidence that Annotation Bundles – or more generally, C functions with unusually great extents of preprocessor directives – are not only hard to comprehend but also hard to maintain.

To reach these goals, I performed a quantitative and qualitative analysis of how preprocessor directives relate to an important maintenance aspect, change proneness. The quantitative part of my analyses revealed that preprocessor use correlates significantly and positively with change proneness when preprocessor use is considered in isolation. However, after controlling for potential confounding factors (i. e., differences in size, age, time since the last change, and number of previous changes), I observed much fewer significant correlations. Moreover, I observed that the remaining correlations were inconsistent, being positive in some subjects but negative in others.

The inconsistent relationship between preprocessor use and change proneness was confirmed in the qualitative part of my study. I found no evidence that the presence or absence of CPP directives *causes* a function to be change-prone or stable. In certain cases, CPP directives co-occurred with an underlying cause for change proneness, such as concern tangling. However, I encountered other indicators of such causes, including the use of comments as a structuring aid, large code size, and a great number of frequent changes in the past. According to my quantitative results, the latter two predict change proneness much more consistently than different extents of CPP use. Moreover, they also apply to functions without any CPP directives, which means they are not only more consistent than preprocessor-related metrics but also more versatile.

In summary, my findings call into question the criticism that preprocessor use makes code harder to maintain. More specifically in relation to $\mathbf{RQ_T\ 1}$ of this thesis, I must change my answer as follows: Regarding, $\mathbf{RQ_T\ 1.2}$, I found that measured by the number of commits and the number of lines changed, the *fine-grained use of preprocessor directives has no systematic effect on maintainability.* Thus, it seems unlikely that the ANNOTATION BUNDLE smell negatively affects fault proneness or maintenance effort. It is still possible that changes to code with preprocessor directives take longer than changes to other code. However, since my methodology did not capture program comprehension, future work is needed to prove or disprove this conjecture. Regarding $\mathbf{RQ_T\ 1.3}$, my study *failed* to provide new hints how the precision of my detection formula for the ANNOTATION BUNDLE smell could be improved because none of the underlying metrics had a systematic effect. More generally, I conclude that preprocessor-related metrics alone are very limited in their ability to detect code that is hard to maintain. Other metrics, such as code size and the extent of previous changes, appear to be more reliable and more versatile predictors.

In this chapter and the one preceding it, I have studied the effects of variability on the code smell concept, which answers $\mathbf{RQ_T\ 1}$ of this thesis. In the next chapter, I turn to $\mathbf{RQ_T\ 2}$ and explore novel refactorings to improve the internal structure of highly configurable software systems.

6. Variant-Preserving Refactoring to Migrate Cloned Product Variants

This chapter is based on and shares material with the VaMoS'14 paper "A Taxonomy of Software Product Line Reengineering" [99] and the SANER'17 paper "Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line" [95].

In Chapter 4, I presented my concept of variability-aware code smells, which are code smells that are specific to *Software Product Lines (SPLs)*. But not just code smells are affected by variability, refactoring is affected, too. I turn to this topic in this chapter.

The chapter starts with an in-depth look at the diverse meanings of "refactoring" in *Software Product Line Engineering (SPLE)*. In the world of non-configurable software systems, refactoring is commonly understood as a change to the structure of the source code to improve a certain quality aspect, such as readability or extensibility. Literature on SPLE, by contrast, uses the term "refactoring" in a more liberal fashion. For instance, *feature oriented refactoring* has been proposed as a means to bootstrap an SPL from a legacy application [211, 364, 162]. This type of refactoring is also sometimes called *migration.* Furthermore, *a model of refactoring physically and virtually separated features* has been presented, which translates one way of encoding variability into another [157]. For example, *Feature-Oriented Programming (FOP)* could be replaced with preprocessor annotations. More conservative extensions of refactoring to SPLs have also been discussed [330, 329]. For instance, extensions of several single-system refactorings (such as Pull Up Method) have been proposed as a means to reduce code replication in feature-oriented SPLs.

The common characteristic of these SPL reengineering activities is behavior preservation, but their connection to improving internal quality aspects is often unclear. I argue that overloading the term "refactoring" in this manner makes it difficult to

distinguish between SPL reengineering activities and to see how they are related. This is a problem for both researchers in the SPLE field as well as interested practitioners. The taxonomy I present in this chapter alleviates this problem by identifying three orthogonal *dimensions* along which SPL reengineering occurs. From these dimensions, I construct a detailed *taxonomy* of SPL reengineering, thereby giving distinct names to distinct activities and showing their relationship. Furthermore, I propose *definitions* for the three major classes of reengineering activities that I have identified. To show the utility of my taxonomy, I reclassify a part of the corpus previously surveyed by Laguna and Crespo [192] and classify selected additional work.

After this theoretical literature work, the chapter switches to a more practical perspective on refactoring. Refactorings for SPLs have to be *variant-preserving* [330, 13]. In other words, they must preserve the behavior of not just a single product, but of *all* products within the product line. As my literature classification will reveal, only few variant-preserving refactorings exist to date. This is especially true for refactorings that affect the source code, as opposed to refactorings that change configurations or the feature model. I contribute to closing this gap with two variant-preserving refactorings that target the source code level. These refactorings are embedded in a concept to migrate a clone-and-own-based software product family into an SPL. In this migration context, code clones, specifically Inter-Feature Code Clones (see Section 4.2.1), are a major challenge. My refactorings are a means to address this challenge. In summary, this chapter contributes the following:

- A taxonomy that distinguishes between different meanings of refactoring in SPLE. Moreover, a corpus of literature is classified in the framework of this taxonomy.
- Two variant-preserving refactorings for SPLs written in Java and using FOP as the variability mechanism.
- A concept and tool support to migrate families of cloned software products into an SPL based on those refactorings.
- A case study that (a) demonstrates the feasibility of my migration approach and (b) shows its potential to reduce Inter-Feature Code Clones.

6.1 Dimensions of Software Product Line Reengineering

Refactoring for single systems (as opposed to SPLs) aims at improving the structure of existing code. This notion of refactoring also exists in the SPL context. However, I have identified additional reengineering activities, which pursue other goals. To make these goals more tangible, this section introduces three orthogonal *dimensions* along which SPL reengineering occurs.

6.1.1 Quality

In the single-system context, refactoring has primarily been associated with changes to improve the design of existing code (Fowler et al. [107], p. 53). Goals include making code easier to understand and modify, and enabling future extensions. Improved quality is easy to argue for many refactorings. For instance, Extract Method is used to split up long methods or eliminate code clones. However, the inverse refactoring, Inline Method, usually leads to longer methods and code duplication – a *decrease* in code quality. Nevertheless, Inline Method may *improve* other properties, such as readability or performance. If, under a given set of circumstances, these are more important than code duplication, the result is an overall improvement. Therefore, I introduce *Quality* as the first reengineering dimension.

6.1.2 Variability Mechanism

Annotation-based and composition-based variability mechanisms occupy different regions in the design space with regard to separation of concerns, granularity of variation points, or language independence. My taxonomy reflects this by consistently distinguishing between these two classes of variability mechanisms.

Further differences exist between concrete mechanisms. For instance, although both *Virtual Separation of Concerns (VSoC)* and cpp-style preprocessors are annotation-based, VSoC enforces disciplined annotations, while the cpp alone does not [155]. Likewise, AspectJ and FeatureHouse are used for composition-based SPL implementation. Nevertheless, AspectJ employs pointcuts and advice, and is applicable only to Java, while FeatureHouse relies on superimposition and is language-independent.

Refactoring and other reengineering techniques must cope with these differences between implementation techniques. This leads to the second dimension along which my taxonomy distinguishes SPL reengineering approaches. I call it *Variability mechanism*.

6.1.3 Legacy→SPL

Many software systems that have not been developed as an SPL. These systems are certainly composed of disparate pieces of functionality, which could be called "features". But in contrast to an SPL, customization requires substantial effort because these features cannot easily be included or excluded. I refer to such systems as *legacy software products*. Various approaches to bootstrap an SPL from legacy software products have been proposed (e. g., [211, 215, 368, 6, 377]). In the following, I will refer to the process of bootstrapping an SPL from legacy products as *migration*. In order to distinguish migration from other reengineering activities, I introduce a third dimension, *Legacy→SPL*.

Some migration approaches consider only a single legacy product (e. g., [211, 215, 60, 368]). The result is an SPL from which new, tailored variants of the original product can be created. In the legacy context, however, customization is sometimes realized through *clone-and-own,* also referred to as *forking*: In order to create tailored variants of a successful software product, the product is copied and adapted

as needed [318, 314, 317]. The result is a *family of related legacy software products*. Clone-and-own is often implemented using the branching and merging capabilities of a *Version Control System (VCS)* [58, 350, 80, 373, 349]. However, VCSes lack the necessary support for mapping changes to features or tracking which variants implement which features – a shortcoming that research has only recently started to address (e. g., [12, 292, 293, 348]). Thus, practitioners still face the time-consuming and error-prone tasks of identifying changes and synchronizing to the proper variants. This makes long-term maintenance of cloned products expensive.

Compared to clone-and-own development, SPLE greatly reduces the effort for synchronizing changes between variants because features in an SPL are implemented only once, but shared among many variants. For this reason, migration is especially desirable when multiple legacy software products (i. e., legacy product families) are involved. Single-product migration approaches are ill-suited for this task, as they lack the means to consolidate similar or identical functionality that is present in more than one product. Instead, several researchers proposed specialized approaches (e. g., [6, 377]). For instance, Xue proposes to combine model comparison techniques and code clone detection in order to locate common and variable functionality in the legacy products being migrated [377]. Analyses of this kind are neither possible nor necessary when only a single legacy product is migrated. This is a marked difference between single-product and multi-product migration approaches. Consequently, I identify two important starting points on the *Legacy→SPL* dimension: *one* legacy software product and *many* legacy software products.

In Table 6.1, I give an overview of the reengineering dimensions from which my taxonomy is built. For each dimension, I list its name and summarize its meaning. The resulting taxonomy is discussed in the following section.

No.	Name	Description
(1)	*Quality*	Improve some property of the code, the feature model, or the feature-to-code mapping (e. g., readability, extensibility)
(2)	*Variability mechanism*	Differentiates between SPL implementation techniques (e. g., AOP, FOP, VSoC); Annotation-based and composition-based techniques are distinguished
(3)	*Legacy→SPL*	One or several legacy software product(s) are migrated to an SPL to enable mass-customization and systematic code sharing

Table 6.1: SPL reengineering dimensions

6.2 A Taxonomy of Software Product Line Reengineering

From the dimensions introduced in the previous section, I have constructed a taxonomy of SPL reengineering activities. In Figure 6.1, I show an overview. The

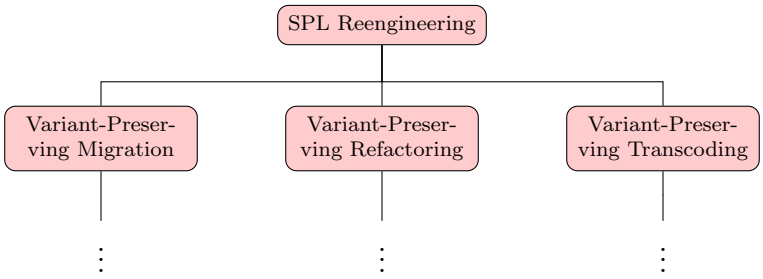


Figure 6.1: Main branches of SPL reengineering

complete taxonomy is depicted in the appendix, in Figure A.1. My taxonomy has three main branches, *variant-preserving migration, variant-preserving refactoring,* and *variant-preserving transcoding*. Each of these branches corresponds to one of the three dimensions, which is its *primary* dimension. For instance, *variant-preserving migration* transforms legacy software along the dimension *Legacy→SPL*. To each main branch, I have applied the *secondary* dimension *Variability mechanism*. This secondary dimension allows to discriminate, for instance, between *variant-preserving migration to VSoC* and *variant-preserving migration to FOP*.

In Figure 6.2, I visualize the relationship of the three SPL reengineering activities to legacy software and SPLs. Via variant-preserving migration, legacy software products (one or many) are transformed into an SPL. The internal structure of an SPL is improved via variant-preserving refactoring. Finally, variant-preserving transcoding transforms a product line SPL into an equivalent one, SPL', that uses a different variability mechanism. In Section 6.2.2–6.2.4, I further elaborate on these activities.

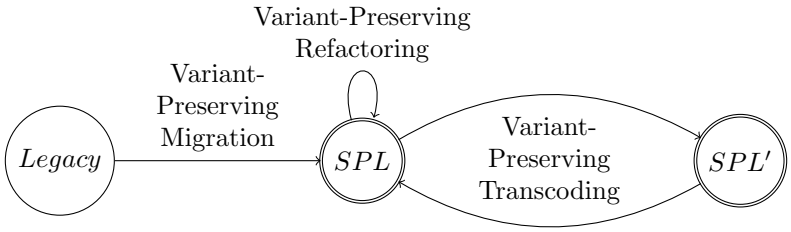


Figure 6.2: Relationship of variant-preserving SPL reengineering activities

I structure existing work according to our taxonomy. I describe the literature selection process in the next subsection. In the following subsections, I give definitions for the main taxonomy branches and describe them in detail.

6.2.1 Literature Selection

I reclassified a part of the corpus created by Laguna and Crespo [192]. The authors have surveyed 74 publications related to legacy system reengineering and product line refactoring. The scope of their survey is broader than ours, including literature that concentrates on guidelines, processes, or organizational issues. For instance, Bosch and Bosch-Sijtsema discuss the introduction of agile development methods in a project to reengineer an existing SPL [41]; techniques to change code or the

feature model (FM) are not discussed. In contrast to this and other, similar work, I focus exactly on those techniques. Hence, some of the material selected by Laguna and Crespo has been excluded. As I am interested in changes to a software system, analyses and metrics, such as those presented by Berger et al. [38], have not been reclassified. Moreover, I excluded work on *refactoring feature modules (RFMs)*, proposed by Kuhlemann et al. [190]. RFMs are essentially single-system refactorings that are applied automatically at product generation time. Therefore, they avoid the SPLE-specific complexity of synchronizing code changes with FM constraints.

These selection criteria resulted in a total of 19 reclassified publications of the original 74. I have extended this selection with two older and four newer publications (including one book) relevant to my topic. I believe that this classification gives a representative overview of existing work and shows the utility of my taxonomy. However, I do not claim completeness. The classification result is summarized in Table 6.2–Table 6.4. Approaches with a star (*) next to the author name were not part of the survey by Laguna and Crespo. They are positioned in the lower part of the tables.

6.2.2 Variant-Preserving Migration

Definition. The first main branch of my taxonomy is concerned with the transformation of one legacy software product or a family of related legacy software products into an SPL. Figure 6.3 shows a sketch of the corresponding section of our taxonomy. Before I can define variant-preserving migration, I first have to explain the meaning of *legacy software product* in the context of my taxonomy.

Definition 1. *A* legacy software product *is a piece of software that has been designed without planning for variability or strategic reuse of the artifacts from which the legacy software product is constructed.*

In short, for the purpose of my taxonomy, I regard any piece of software that has not been developed according to SPLE guidelines a legacy software product. Based on this definition, I propose to define *variant-preserving migration* in the following way:

Definition 2. Variant-preserving migration *is the process of transforming one legacy software product or a family of related legacy software products into a software product line such that for each migrated legacy software product there is a product line instance with the same external behavior.*

Dimensions. Variant-preserving migration is reengineering that occurs along the dimension *Legacy→SPL*. The difference between approaches to migrate one or many products is reflected by the nodes *1→SPL* and *Many→SPL*, respectively. In order to further differentiate approaches, I have applied the secondary dimension *Variability mechanism*. The leaf nodes are labeled according to the targeted variability mechanism.

The vertical dots in Figure 6.3 represent parts of the taxonomy that have been omitted for brevity. For example, the dots to the right of *1→VSoC* stand for reengineering activities that migrate one legacy software product to an SPL implemented

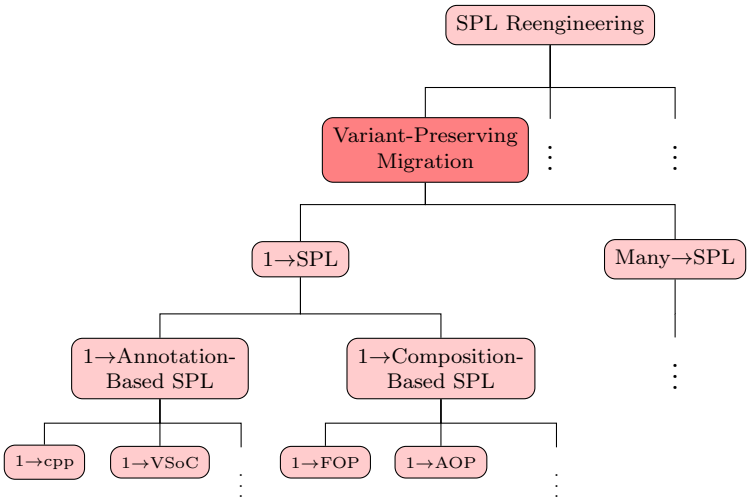


Figure 6.3: Variant-preserving migration

in some other annotation-based approach (e. g., XVCL or JAVAPP). The subtree rooted in *Many→SPL* (also represented by dots) is isomorphic in structure to the one below *1→SPL*.

Classification. Liu et al. propose *Feature-Oriented Refactoring (FOR)* as a process to decompose a single legacy program into its features [211]. A theory is developed that relates feature refactoring to algebraic factoring [211]. Furthermore, a five-step process is presented. The process relies on a domain expert to name features and label members of a feature (e. g., fields, methods). The code extraction step itself is automated. However, FOR also proposes *derivatives,* special feature modules for interacting features. If two or more features interact, a derivative is created that contains the implementation of this interaction. Reengineering these derivatives again requires manual intervention. FOR only considers a single legacy product. The result of FOR is an SPL implemented using AHEAD, a variability mechanism based on FOP. Consequently, I classify it *1→FOP*.

Alves et al. present eight patterns to extract variable functionality from legacy JAVA code to ASPECTJ aspects [8, 6, 7, 5]. These patterns are automated by the tool FLIPEX, part of the FLIP tool suite [5, 44]. Furthermore, Alves et al. discuss FM refactoring patterns [6]. However, the combination of code extraction patterns and FM refactoring is not explored in detail. Besides single product migration, a process to migrate a family of legacy products to an SPL is outlined. According to this process, each member of the legacy product family is migrated separately to a temporary SPL. These temporary SPLs are then refactored in such a way that all source code that is common to two or more products is located in features that have identical names in the temporary SPLs. All other code is moved to unique features. In order to form the final SPL, the temporary SPLs are superimposed. Alves et al. leave several question open regarding this migration process. How are developers expected to identify which code is common to multiple products and which code is unique? Moreover, what are good ways to accomodate product-specific customizations? Dealing with these issues manually is very time-consuming, even for small programs. On the other hand, automation is still subject to ongoing research.

Consequently, it is unclear whether the migration process of Alves et al. will be feasible on program families of realistic size.

Of the material from the survey by Laguna and Crespo, the largest part (13 publications out of 19) deals with variant-preserving migration. I show an overview of this work in Table 6.2. Most approaches migrate just a single legacy product, with the work of Alves et al., Xue et al., and Zhang et al. being notable exceptions. Moreover, there is emphasis on approaches that target composition-based variabil-

First Author	Classification	Note
Alves [5, 6, 7, 8]	*1→AOP*, *Many→AOP*	Eight transformations to extract variable parts of one legacy software product to AspectJ aspects [8, 7]; Description of process to migrate one or several legacy software product(s) to an *Aspect-Oriented Programming (AOP)* SPL [8, 6]; Approach is implemented in FLiP tool suite [5]
Calheiros [44]	*1→AOP*, *Many→AOP*	Tool demo of FLiPEx migration tool (part of FLiP tool suite); Implements approach of Alves et al. [8, 6]
Couto [60]	*1→javapp*	Case study migrating the open source *Unified Modeling Language (UML)* tool ArgoUML to ArgoUML-SPL
Kästner [156]	*1→AOP*	Case study migrating BerkeleyDB to an SPL using AspectJ; Argues that AOP is unsuitable for migration
Kästner [162]	*1→AOP*	Presents tool ColoredIDE (now called CIDE) for migration of a legacy software product to an AOP SPL; Uses *derivative* model by Liu et al. [211] to migrate interacting features
Liu [211]	*1→FOP*	Algebraic model of migration of one legacy product to Ahead; Introduces *derivative* model for interacting features
Lopez-Herrejon [215]	*1→FOP*	Migration of a single legacy Java product to FeatureHouse; Identifies eight migration patterns
Olszak [277]	*1→FOP*	Semi-automatic feature location and automatic restructuring of one legacy software product; features modules are represented as Java packages; Classes implementing more than one concern are not split, resulting in limited separation of concerns
Trujillo [364]	*1→FOP*	Case study migrating code, XML documentation, and tests of the Ahead tool suite to a composition-based SPL (implemented with Ahead)
Xue [377, 376]	*Many→XVCL*	Migration of family of cloned legacy software products to an annotation-based SPL; Combines FM comparison and code clone detection to locate common and variable features
Zhang [389]	*Many→XVCL*	Experience report migrating cloned legacy products to an SPL in order to create product variants for mobile devices
Apel * [13]	Various	Refactoring as a path toward a product line (pp. 203–210); Refactoring example catalog (pp. 201, 202) contains Extract Feature and Extract Shared Code suitable for migration of legacy software products
Valente * [368]	*1→VSoC*	Semi-automatic feature location and annotation in VSoC tool CIDE

Table 6.2: Classified work on variant-preserving migration

ity mechanisms. Approaches that target annotation-based mechanisms are rare by comparison.

6.2.3 Variant-Preserving Refactoring

Definition. Figure 6.4 details the part of my taxonomy that covers activities to change the FM or code of an SPL in a behavior-preserving way. The concept of *variant-preserving refactoring* was first introduced by Schulze et al. [330]. Apel et al. additionally differentiate between refactorings that increase or reduce the number of products in the SPL [13], but for simplicity, I repeat the original definition that Schulze et al. proposed:

Definition 3. *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.*

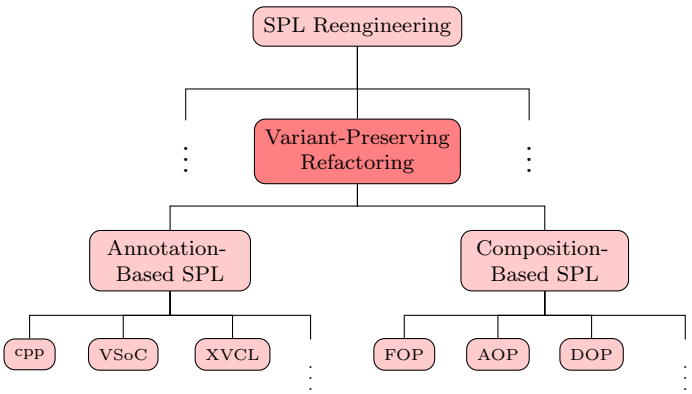


Figure 6.4: Variant-preserving refactoring

Dimensions. Variant-preserving refactoring primarily changes the code, FM, or the feature-to-code mapping along the *Quality* dimension. Approaches to variant-preserving refactoring can be distinguished by the variability mechanism of the SPL being refactored: Refactoring of CPP code must handle annotations, whereas refactoring of AOP programs deals with pointcut expressions and advice. Hence, my taxonomy takes the dimension *Variability mechanism* into account. Note that this results in a hierarchy that is structurally similar to the subtree rooted in *1→SPL* in Figure 6.3.

Classification. Variant-preserving refactoring pursues the same goal as refactoring defined by Fowler et al. (Fowler et al. [107], p. 53), that is, to improve the design of existing code. This is reflected in the literature on variant-preserving refactoring that I list in Table 6.3. For instance, Schulze et al. define variant-preserving refactoring and propose the variant-preserving refactorings Pull Up Field To

First Author	Classification	Note
Alves [6]	*AOP*	Focus on FM transformations that maintain or increase configurability; Synchronization with code changes remains vague
Ghanam [116]	*design patterns*	Introduce variation points in unit tests using the *factory* design pattern (composition-based); Reactive, agile process of SPL refinement
Şavga [321]	*FOP*	Challenges of refactoring in both *problem* and *solution spaces* (roughly: FM and code)
Apel * [13]	Various	Ch. 8 (pp. 193–212) defines refactoring in *feature-oriented software development*; Catalog of *variability smells* (roughly: code smells in an SPL)
Kim * [174, 175]	*Custom annotations*	Automation of 34 refactorings for SPLs in Java using a custom annotation scheme; Tool X15 [175]
Liebig * [205]	cpp	Automation of three refactorings for C with cpp directives; Tool Morpheus
Neves * [267]	*AOP*	Presents templates for the *safe evolution of SPLs* (considers code, FM, feature-to-code mapping); Validation of templates by analyzing evolution of two SPLs
Schulze * [329]	*DOP*	Code smell catalog and selected refactorings for *Delta-Oriented Programming (DOP)*; Tool DeltaJ
Schulze * [330]	*FOP*	Definition of consistent refactoring of code and FM; Catalog of four FOP refactorings that cross feature boundaries (e. g., Move Field Between Features)
Schulze * [328]	*FOP*	Discussion of challenges to automating refactorings for FOP; Tool VAmPiRE that automates Pull Up To Parent Feature

Table 6.3: Classified work on variant-preserving refactoring

Parent Feature, Pull Up Method To Parent Feature, Move Method Between Features, Move Field Between Features for FOP SPLs [330]. They discuss how to use these refactorings to remove code clones, which is a typical maintenance activity. Schulze et al. further report experiences in implementing variant-preserving refactorings for FOP using Pull Up To Parent Feature as an example [328]. Their report shows the technical and practical challenges that implementors of variant-preserving refactoring engines face.

Later work of Schulze et al. discusses refactoring delta-oriented SPLs [329]. A catalog of code smells and 23 DOP refactorings is presented. The refactorings are automated by the tool DeltaJ.

Alves et al. discuss FM refactoring and patterns for variant-preserving migration to an AOP SPL [6]. However, FM refactoring and migration are discussed separately.

The *factory* design pattern is employed by Ghanam et al. as a means to introduce variation points into an existing SPL [116].

Examples of *variability smells* and corresponding refactorings are given by Apel et al. ([13], Chapter 8). As discussed in Section 4.7, these variability smells are similar to the variability-aware code smells I propose in Chapter 4, but more broadly scoped. Furthermore, Apel et al. distinguish between three notions of behavior-preservation

for SPL refactorings, which differ in the number of products in the SPL that are preserved. They call refactorings *variability-enhancing* if they increase the number of potential products in the SPL. *Variability-preserving* refactorings, in turn, keep the number of potential products exactly the same. In other words, none of the existing (or potential) products are removed, and no new products are added. By comparison, the definition of Schulze et al. is less specific because it only demands that no products are removed. In the terminology of Apel et al., this encompasses both variability-preserving and variability-enhancing refactorings. Finally, Apel et al. categorize refactorings as *product-preserving* if they preserve the behavior of a given subset of products. For instance, a company may be satisfied if behavior is preserved only for the subset of products that are actually delivered to customers.

While the aforementioned work has addressed composition-based variability mechanisms, two refactoring engines for annotation-based variability mechanisms have been proposed recently [205, 174, 175]. Specifically, Morpheus can perform three refactorings (Rename, Extract Function, and Inline Function) on C code with preprocessor directives [205]. The analyses for the refactorings are performed on a variability-aware *Abstract Syntax Tree (AST)*, which is generated by TypeChef [161]. Kim et al., in turn, implemented their X15 engine without such an AST [174]. Instead they "lift" a number of precondition checks of an existing Java refactoring engine, R3, to make it variant-preserving. Thus, despite relatively few customizations to R3, the engine is capable of 34 refactorings, which makes it the most comprehensive variant-preserving refactoring engine to date. X15 can refactor SPLs in Java that use a custom annotation scheme as the variability mechanism. Details on the implementation are available in the accompanying tool paper [175].

Table 6.3 contains a summary of the research on variant-preserving refactoring. This summary indicates that much work is left for researchers and tool builders in the field of highly configurable software systems to reach the maturity of refactoring for single systems. The challenges of variant-preserving refactoring have been pointed out [321, 330], and some advances toward automation have been made [329, 328, 205, 174, 175]. Most of these contributions focus on composition-based variability mechanisms, mainly on AOP and FOP. By comparison, much fewer work has addressed refactoring for annotation-based variability mechanisms, despite the fact that annotations are much older and more widely used in practice [13]. Independently of the variability mechanism, automation remains a challenge.

6.2.4 Variant-Preserving Transcoding

Definition. Both annotation-based and composition-based variability mechanisms have their own benefits and drawbacks. It has been argued that one should be free to switch from one representation to the other, and even mix and match techniques, in order to profit from the respective benefits [14, 157]. For techniques that enable this switch, I propose the term *variant-preserving transcoding*.

Definition 4. *A substitution of the implementation technique of a software product line is called* variant-preserving transcoding *if for each instance of the original product line there is an instance of the new product line that has the same external behavior.*

Dimensions. Variant-preserving transcoding changes the variability mechanism of an SPL. Thus, the primary dimension is *Variability mechanism.* As there is a source mechanism and a target mechanism, I apply the dimension *Variability mechanism* a second time. Figure 6.5 shows a sketch of the resulting section of my taxonomy.

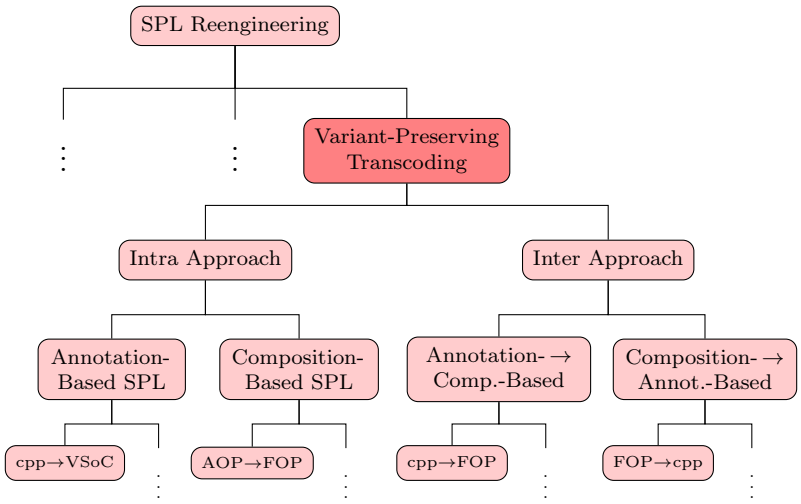


Figure 6.5: Variant-preserving transcoding

Approaches that stay in the same class of variability mechanism (either annotation-based or composition-based) are covered by the subtree rooted in *Intra Approach*. *Inter Approach*, by contrast, covers techniques that transcode the implementation from an annotation-based mechanism to a composition-based one or the other way around.

Classification. Kästner et al. present twelve transcoding "refactorings", named R.1–R.12 [157]. R.1–R.5 transcode feature modules to annotated code. They are complemented by R.6–R.12, which transcode annotated code back to feature modules. These transformations have been used to implement an import/export facility for the tool CIDE and proven correct for the Java subset Featherweight Java. CIDE itself uses the annotation-based technique VSoC. However, utilizing R.1–R.5 and R.6–R.12, it can import and export projects in the composition-based FOP languages Ahead, FeatureHouse and the AOP approach AspectJ. Hence, R.1–R.12 are *inter*-approach transcodings.

Prior to the work of Kästner et al., Kuhlemann et al. explored the equivalence of AOP and FOP [191]. They provide a set of rules to transcode aspect-oriented programs to equivalent feature-oriented programs. These rules are validated by expressing 23 aspect-oriented design patterns with the help of FOP language constructs.

Ribeiro and Borba present a tool to recommend inter-approach-transcodings from runtime, annotation-based variability (using `if-else` statements) to static, composition-based mechanisms [303, 302]. The tool does not perform these transcodings, though.

Apel et al. describe at least two smells related to the variability mechanism and informally discuss refactorings to rectify those smells ([13], Chapter 8). The first is Runtime Overhead, which can result from the use load-time binding of features

as opposed to compile-time binding. The solution is a Change Binding-Time refactoring to switch from runtime `if` statements to a compile-time mechanism, such as `#ifdef`s or FOP. The second smell is Language and Variability-Mechanism Overload, which can cause developers to struggle with a mixture of too many variability mechanisms, such as build-system variability, `#ifdef`s, and aspects. To remove the smell, Apel et al. recommend to use fewer, more uniform mechanisms.

Table 6.4 summarizes work on variant-preserving transcoding. Imports and exports à la CIDE are an interesting application of variant-preserving transcoding. Transcoding may also prove useful as a tool for variant-preserving migration of legacy preprocessor code to more structured variability mechanisms.

First Author	Classification	Note
Kästner [157]	*VSoC→FOP*, *FOP→VSoC*	Inter-approach transcoding between VSoC (annotation-based) and the FOP dialects Ahead and FeatureHouse (composition-based); Implementation as im-/exports in CIDE; Correctness proof for Featherweight Java
Ribeiro [303, 302]	`if-else`→*AOP*, `if-else`→*inheritance*, `if-else`→*mixins*, `if-else`→*patterns*	Tool to recommend inter-approach transcodings from `if-else`-statements (annotation-based) to various composition-based mechanisms (e. g., AOP, mixins, design patterns) and to "configuration files" (configuration files are not explained)
Apel * [13]	Various	Variability smells that indicate need for transcoding, such as Runtime Overhead (p. 199); Transcoding refactorings (e. g., Change Binding-Time maps runtime `if-else` to static `#ifdef` statements) (pp. 201, 202)
Kuhlemann * [191]	*AOP→FOP*	Discusses the feasibility of using FOP to implement 23 aspect-oriented design patterns; Rules to transform AOP into FOP

Table 6.4: Classified work on variant-preserving transcoding

6.2.5 Summary

In contrast to object-oriented programming, SPLE literature uses "refactoring" as an umbrella term to denote a variety of reengineering techniques. In the previous sections, I have proposed a taxonomy to clarify the connections and distinctions between these techniques. Moreover, I have classified a body of existing work according to this taxonomy.

My taxonomy reveals that SPL reengineering techniques belong to one of three categories. First, *variant-preserving migration* serves to create an SPL from one or several legacy software systems. The purpose of migration is to adopt SPLE for (a set of) software systems that were previously developed without a systematic reuse approach. Most "refactoring" approaches for SPLs that I classified actually fall in this category. Among those, the majority targets the migration of a single legacy software system into an SPL, for example, by making features optional. By comparison, much fewer approaches exist to migrate a set of related legacy software systems (e. g., systems created via clone-and-own) into an SPL. Second, *variant-preserving*

refactoring is a particular kind of change to the source code of an SPL, its feature model, or both. This change is performed in a way that preserves the behavior of all instances of the SPL. Similar to *Object-Oriented Programming (OOP)* refactoring, the goal is to improve internal quality attributes of the SPL, such as understandability or evolvability. The classified literature shows that some progress toward automating variant-preserving refactorings has been made. Nevertheless, tool support remains a major challenge. Finally, *variant-preserving transcoding* exchanges one variability mechanism for another. For example, transcoding makes it possible to import an FOP-based SPL into an annotation-based tool such as CIDE without risking a behavior change.

6.3 Open Challenges in Variant-Preserving Refactoring and Migration

My literature classification showed that there is little work on migrating families of related legacy software products to an SPL. More importantly, the published approaches focus on models, processes, and so on, but lack specifics at the implementation level. However, this is exactly what developers face during migration: the analysis and transformation of potentially huge amounts of legacy source code. Thus, an important aspect of migration is underrepresented in the current state of the art. Therefore, in the following sections, I will focus on the implementation-level aspects of a novel *Many→FOP* migration approach for Java. Specifically, I will discuss how to find commonalities and differences in the source code of cloned variants, and how to increase the amount of systematic source code reuse through *variant-preserving refactoring.* To this end, I propose two novel refactorings and explain how they have been automated in the tool FeatureIDE. Thus, I fill in another gap that my literature classification revealed, the lack of automation for variant-preserving refactorings.

My migration approach targets families of related legacy software products that have been created via clone-and-own. Due to the clone-and-own origin, I expect a large amount of code sharing, a. k. a. *code clones*. Thus, I consider migration primarily as a code-clone problem: If I can reduce the amount of code clones across product variants, I will also reduce the maintenance overhead caused by having to synchronize changes from one variant to another. Consequently, the main building blocks of my approach are code clone detection for identifying commonalities between product variants and variant-preserving refactoring [330] to consolidate them. As an important supporting concept, I propose *preparatory refactorings,* that is, refactorings that align variant-specific divergences that would otherwise prevent the consolidation of code clones.

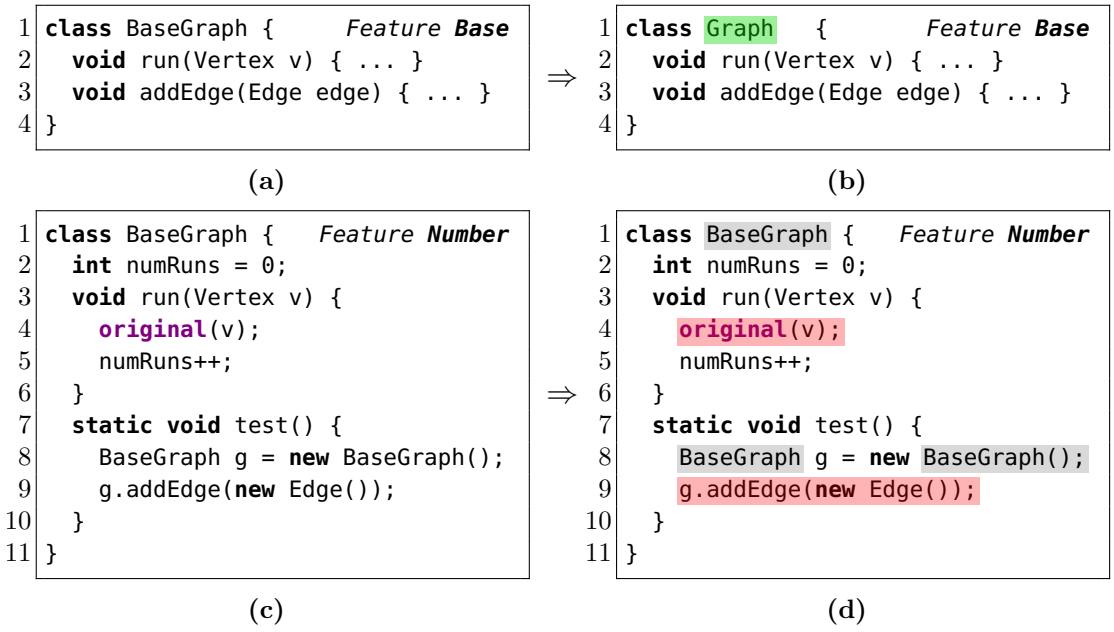
Apart from its focus on the implementation level, my migration approach is also novel in that it is flexible and incremental. Existing approaches attempt migration in one "big bang", without accommodating other development activities, such as enhancements or bug fixes. This is impractical because migration endeavors take considerable time and effort, and the chances of failure are high [52, 187]. Few companies will be willing to start a migration if it means that all other development

must be suspended for months or even years. In contrast to those "big-bang" approaches, my approach favors small, incremental, easy-to-understand steps, which can be interleaved with other activities. The goal is not full automation. Instead, the goal is to automate the tedious and error-prone tasks and leave the big design decisions to the developers.

Next, I explain the variant-preserving refactorings that I propose for code clone consolidation. Afterwards, I describe how to integrate these refactorings with code clone detection into a variant-preserving migration process.

6.4 Refactorings to Remove Inter-Feature Code Clones

Refactorings that fail to take variability into account are not generally behavior-preserving when applied to an SPL. I demonstrate this point with the help of the example in Listing 6.1. The example depicts an application of Eclipse's OOP Rename refactoring to a small graph product line with two features, *Base* and *Number*. The original code is shown on the left (see Listing 6.1 (a) and (c)). The code after the refactoring attempt is depicted on the right (see Listing 6.1 (b) and (d)). Although the goal was to rename class `BaseGraph` to `Graph`, the variability-unaware refactoring only renamed the code in the feature module of *Base* (see Listing 6.1, top row), but not in *Number* (bottom row). As a result, the feature module in the bottom row contains several errors, highlighted in gray and red in Listing 6.1 (d). In particular, the class in *Number* is still named `BaseGraph`, when it should be named of `Graph`. In effect, this has turned the class from a refinement into an introduction.



Example of an OOP Rename refactoring producing wrong results when applied to FOP code. Only the code in feature *Base* is renamed correctly (b), but the code in *Number* is not, leading to dangling references (red highlights in (d)).

Listing 6.1: An OOP Rename refactoring producing wrong results on FOP code

As a result, the calls to `original` and `addEdge` have become dangling references (highlighted in red), which will cause compilation errors in variants that contain the *Number* feature. A variant-preserving Rename refactoring would also have changed the class name in feature *Number* from `BaseGraph` to `Graph` (highlighted in gray in Listing 6.1 (d)). This would have kept the introduction-refinement relationship between the code in features *Base* and *Number* intact, and thus, would have avoided the dangling references (highlighted in red).

Next, I describe two variant-preserving refactorings for FOP, Pull Up To Common Feature and Rename. The Pull Up To Common Feature in particular generalizes and improves upon previous work by Schulze et al., who proposed a Pull Up To Parent Feature [330, 328]. Compared to their OOP counterparts, Pull Up and Rename, I contribute the following:

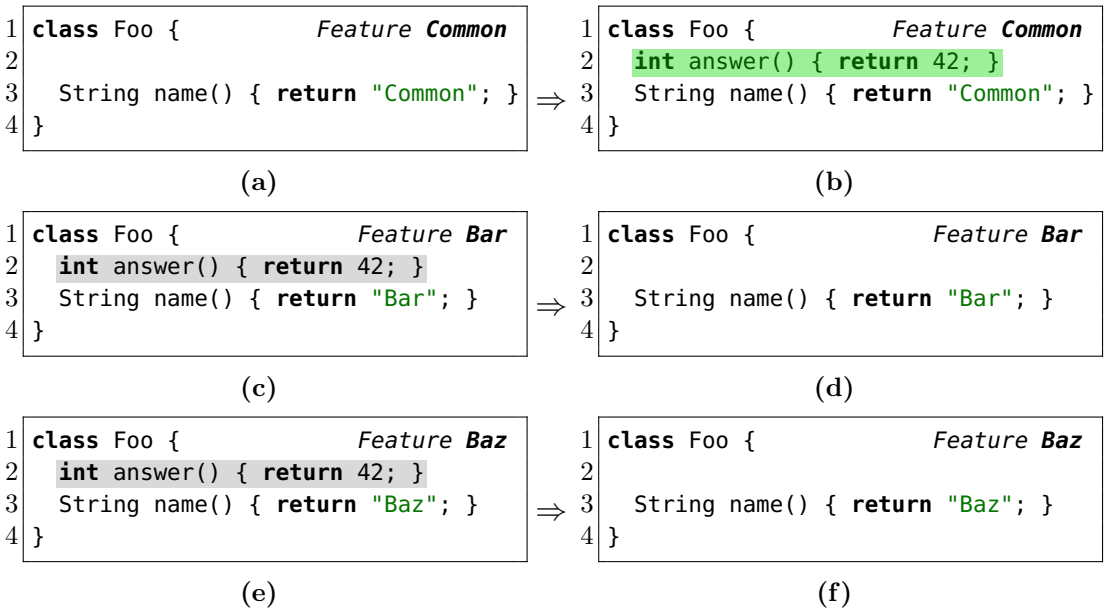
1. I extend the preconditions and mechanics of both refactorings so they become variant-preserving.
2. For the Pull Up To Common Feature refactoring, I provide an algorithm to identify the "common feature", that is, the feature into which the respective code fragment can be moved in a variant-preserving manner. If no suitable feature exists, the algorithm tries to create a new one.

Although I propose these refactorings to eliminate Inter-Feature Code Clones in a migration context, I anticipate they will also be useful in other settings, such as maintenance and evolution. In particular, Inter-Feature Code Clones could also arise in an SPL when a developer copies a piece of code from one feature to reuse it in another feature. My Pull Up To Common Feature refactoring is applicable in those situations, too. Regarding Rename, Murphy-Hill et al. found that it is by far the most frequently used refactorings in OOP [266]. There is no reason why things should be different for FOP. Thus, my variant-preserving Rename refactoring will have many uses besides enabling clone removal.

6.4.1 Pull Up To Common Feature

The Pull Up refactorings described by Fowler et al. (Fowler et al. [107], p. 320ff.) are used to move identical definitions of class members (i. e., fields, methods, and constructor bodies) from a set of subclasses into a common superclass. As Pull Up replaces multiple replicated definitions with a single definition, it is an effective means to remove code clones in a single software product. While removing clones is my goal as well, I want to remove them from *multiple software products.* To this end, I propose an extension of Pull Up for FOP, called Pull Up To Common Feature. The basic idea of my refactoring is to move identical definitions of class members from several *source features* into a single, common *target feature,* which is located higher up in the feature hierarchy than the source features. Hence, instead of moving code within the class hierarchy, as OOP Pull Up does, I move the code within the feature hierarchy.

I illustrate the application of Pull Up To Common Feature by means of the example in Listing 6.2. The example is a product line consisting of the features *Bar* and *Baz*, whose parent feature is *Common*. *Common* introduces class `Foo`, which



Application of the Pull Up To Common Feature refactoring to move the common definition of method `answer` in class `Foo` from features *Bar* and *Baz* (gray highlight in (c) and (e)) to the common feature *Common* (green highlight in (b)).

Listing 6.2: Application of Pull Up To Common Feature

is refined by both, *Bar* and *Baz*. More importantly, *Bar* and *Baz* contain a Type-1 Inter-Feature Code Clone, the method `answer` (highlighted in gray in (c) and (e)). By applying Pull Up To Common Feature, the definitions of `answer` in *Bar* and *Baz* are replaced with a single definition in feature *Common* (see (d), (f), and the green highlight in (b)) and thus, the code clone is removed. As a result, common functionality of product variants containing features *Bar* or *Baz* has been consolidated, while preserving the behavior of these variants.

I will now explain how Pull Up To Common Feature works in detail. Like Pull Up for OOP, my refactoring can be applied to fields, methods and constructors, meaning that there are in fact three refactorings. Since all of them are very similar, I focus on Pull Up Method To Common Feature as a representative, and explain this refactoring in terms of arguments, preconditions and mechanics. The arguments are supplied by the developer or some analysis tool and specify, for instance, which method definitions to pull up. Preconditions are properties that must hold in order for the program transformation to be behavior-preserving. Finally, the mechanics specify the program transformation itself.

Arguments:

1. n source features $F_s = \{f_{s_1}, f_{s_2}, \ldots, f_{s_n}\}$ with $n > 1$,
2. A target feature f_t,
3. A class name c,
4. A method signature m,
5. A selection $i, 1 \leq i \leq n$, denoting the definition m_i of method m in class c in feature f_{s_i}, where $f_{s_i} \in F_s$.

Preconditions:

1. Each $f_s \in F_s$ must contain a class c that defines a method with signature m.
   I suggest pulling up definitions only if they constitute Type-1 clones as these are currently the only ones for which the tool support can ensure fully automatically that the refactoring is behavior-preserving. However, developers may override this suggestion, e. g., to pull up method definitions that constitute higher-level clones.
2. All fields and methods referenced in m_i must be defined in f_t or one of the features implied by f_t.
   This precondition ensures that the selected method definition, m_i, will not reference any fields or methods that are undefined in the target feature. If such references exist, Pull Up To Common Feature *must be applied to the respective fields and methods first.*
3. Target feature f_t must be a concrete feature.
   Only concrete features can contain code, abstract ones cannot.
4. Any valid configuration that contains an $f_s \in F_s$ must also contain f_t and vice versa.
   This precondition enforces that all products containing one of the source features will have access to the new location of the code. Since this precondition additionally requires that at least one source feature is present whenever the target feature is present, it prevents the pulled up method definition from shadowing other definitions, for example, definitions in features composed before the target feature.
5. If a class c already exists in the target feature f_t, it must not define a method with signature m.
6. With F being the set of all features, no valid configuration containing an $f_s \in F_s$ may contain an $f_d \in F \setminus F_s$ that fulfills the following criteria:
   (a) f_d contains a class c that defines a method with signature m,
   (b) f_d is composed after f_t but before f_s.
   This precondition prevents definition m_i from being overwritten or refined by a definition from another feature before the source feature is composed.

Mechanics:

1. Create class c in f_t (unless it already exists) and create a new method with signature m in that class.
2. Copy the selected method definition m_i from s_i into the newly created method.
3. Delete the old definitions of m from all classes c in all source features $f_s \in F_s$.

Determining the Target Feature A critical point of Pull Up To Common Feature is how to determine a suitable target feature. In general, the target feature must be part of any valid configuration that contains one of the source features, and it must be composed before the source features. In case all source features have a common parent feature, the parent might serve as the target feature. Otherwise, a new target feature can usually be created.

In Algorithm 1, I show the function ensureTargetFeature that implements this task. The function takes as input the set of source features F_s, the name of

Algorithm 1 Find or Create a Target Feature for Pull Up To Common Feature

```
1: function ensureTargetFeature(F_s, c, m, fm, C)
2:     F_c ← findTargetFeatures(F_s, c, m, fm)
3:     if F_c ≠ ∅ then
4:         return choseClosest(F_c, F_s, fm)
5:     else
6:         return createTargetFeature(F_s, c, m, fm, C)
7:     end if
8: end function

9: function findTargetFeatures(F_s, c, m, fm)
10:    F_b ← {f ∈ features(fm) | implies(fm, f ⇔ ⋁_{f_s ∈ F_s} f_s)}
11:    o_min ← min({order(f_s) | f_s ∈ F_s})
12:    F_c ← {f ∈ F_b | order(f) < o_min ∧ concrete(f) ∧ ¬defines(f, c, m)}
13:    return removeShadowed(F_c, F_s, c, m, fm)
14: end function

15: function createTargetFeature(F_s, c, m, fm, C)
16:    F_i ← {f ∈ features(fm) | implies(fm, f ⇐ ⋁_{f_s ∈ F_s} f_s}
17:    o_min ← min({order(f_s) | f_s ∈ F_s})
18:    F_c ← {f ∈ F_i | order(f) < o_min}
19:    F_p ← removeShadowed(F_c, F_s, c, m, fm)
20:    if F_p = ∅ then
21:        error("Too many conflicting definitions.")
22:    end if
23:    f_p ← choseClosest(F_p, F_s, fm)
24:    f_t ← addChild(f_p, CONCRETE, OPTIONAL)
25:    addCTC(fm, f_t ⇔ ⋁_{f_s ∈ F_s} f_s)
26:    for C ∈ {C ∈ C | C ∩ F_s ≠ ∅} do
27:        C ← C ∪ {f_t}
28:    end for
29:    return f_t
30: end function

31: function removeShadowed(F_c, F_s, c, m, fm)
32:    F_d ← {f ∈ features(fm) \ F_s | defines(f, c, m)}
33:    F_r ← ∅
34:    for f_c ∈ F_c do
35:        if ∀f_s ∈ F_s, ∄f_d ∈ F_d : satisifable(fm ∧ f_c ∧ f_d ∧ f_s)
                                    ∧ order(f_c) < order(f_d) < order(f_s) then
36:            F_r ← F_r ∪ {f_c}
37:        end if
38:    end for
39:    return F_r
40: end function
```

the defining class c, the method signature m, the FM fm, given as a propositional formula, and the set of existing configurations as $\mathcal{C}$. In a migration context, these are the configurations of the products being migrated. First, ensureTargetFeature calls findTargetFeatures to identify suitable target features. If there are some, choseClosest (not shown) will select the one that is most appropriate. This helper function could be a metric or an interactive function that lets the developer make the decision. If no suitable target features exist, createTargetFeature

will create a new one. The bulk of Algorithm 1's work is, of course, performed by the two helper functions FINDTARGETFEATURES and CREATETARGETFEATURE, which I explain in the following.

The first helper function, FINDTARGETFEATURES, starts by identifying the features that are implied by all source features and which, in turn, also imply that at least one of the source features is selected (Line 10). This implements precondition 4 of PULL UP TO COMMON FEATURE. In Lines 11 and 12, this set of features is reduced to features that are concrete (precondition 3), do not already define method m (precondition 5), and are composed before the source features. (Assume that *order* returns the composition order of a feature.) Finally, REMOVESHADOWED is called on the set of remaining features (Line 13). This helper function implements precondition 6 by excluding potential target features that could be composed with another feature that contains a conflicting definition of method m (Line 35).

The second helper function, CREATETARGETFEATURE, first identifies parent features for the target feature f_t, which is about to be created (Lines 16–18). Note that these parents must also fulfill precondition 6 (Line 19) since conflicting method definitions would also affect f_t. If no suitable parent exists, the function aborts (Lines 20–22). Otherwise, a parent feature is chosen by CHOSECLOSEST. Next, the target feature f_t is created as a concrete, optional child of the chosen parent feature (Line 24). To fulfill precondition 4, a cross-tree constraint is added stating that f_t must be selected if and only if one of the source features F_s is selected (Line 25). Finally, after updating all existing configurations so that they conform to this new constraint (Lines 26–28), the new target feature is returned.

6.4.2 Rename

Code clone removal is sometimes impossible due to minor differences. For instance, if two methods are identical except for different names (as shown in Listing 6.3, gray highlights), PULL UP TO COMMON FEATURE is not applicable.

Feature s_1

```
1 class Circle {
2   int x, y, radius;
3   int getX() { return x; }
4   int getY() { return y; }
5   int getRadius() { return radius; }
6 }
```

(a)

Feature s_2

```
1 class Circle {
2   int centerX, centerY, diameter;
3   int getCenterX() { return centerX; }
4   int getCenterY() { return centerY; }
5   int getDiameter() { return diameter; }
6 }
```

(b)

The code that models the circle's center in s_1 and s_2 is a Type-2 clone, but naming differences (highlighted in gray) prevent its consolidation. By contrast, `getDiameter` and `getRadius` (highlighted in red), should not be consolidated, despite their similarity.

Listing 6.3: Naming differences that prevent code clone consolidation

Similar problems arise when trying to pull up cloned code from classes with different names. PULL UP TO COMMON FEATURE works by moving code up in the refinement hierarchy, but classes can only be part of the same refinement hierarchy if they have the same name. To enable clone removal in such situations, it is necessary to eliminate the differences by means of *preparatory refactorings.* To this end,

I propose a RENAME refactoring that takes variability into account. Essentially, I use RENAME to convert Type-2 clones into Type-1 clones and, thus, make them amenable to subsequent consolidation via PULL UP TO COMMON FEATURE.

Similarly to PULL UP TO COMMON FEATURE, my RENAME refactoring can be applied to different elements, such as classes and interfaces, methods (static and instance methods), fields (static and instance fields), method and constructor parameters, and to local variable. I describe RENAME INSTANCE METHOD as a representative.

Arguments:

1. Old method signature m_o,
2. The class, c, that defines m_o,
3. The feature, f, containing the definition of c,
4. New method name n.

Auxiliary Definitions: I introduce the following definitions to describe preconditions and mechanics more concisely.

1. Let m_n be the new method signature. It is constructed from m_o by replacing the old method name with the new one, n.
2. Let D_{m_o} be the set of classes containing c, as well as all classes that define methods that override or are overridden by c's definition of m_o. If m_o is private in c, D_{m_o} contains only c as private methods cannot be overridden in Java. Otherwise (m_o is non-private), D_{m_o} contains c, as well as all sub- and superclasses of c that contain a non-private definition of m_o.

Preconditions:

1. The introductions and refinements of the classes in D_{m_o} must not define a method with signature m_n.
   This precondition prevents RENAME *from producing duplicate definitions in the classes that define a method with signature m_o.*
2. If m_o is non-private in c, then for all classes d in D_{m_o} it must hold that there is no introduction or refinement of a subclass of d that defines a method m_n with a lower visibility than that of m_o in d.
   Overriding methods in JAVA *must not reduce the visibility of superclass methods. This precondition prevents* RENAME *from breaking that rule.*
3. For all classes d in D_{m_o} it must hold that there is no introduction or refinement of a superclass of d that defines a non-private method with signature m_n with a greater visibility than that of m_o in d.
   This precondition is also related to JAVA*'s visibility rules. It prevents renamed definitions of m_o from restricting the visibility of preexisting definitions of m_n.*

Mechanics:

1. Find all references to method m_o.
2. In all introductions and refinements of classes in D_{m_o} that contain a definition of m_o, create a new method with signature m_n and copy the contents of m_o into this new method.
3. Update the collected references to point to m_n.

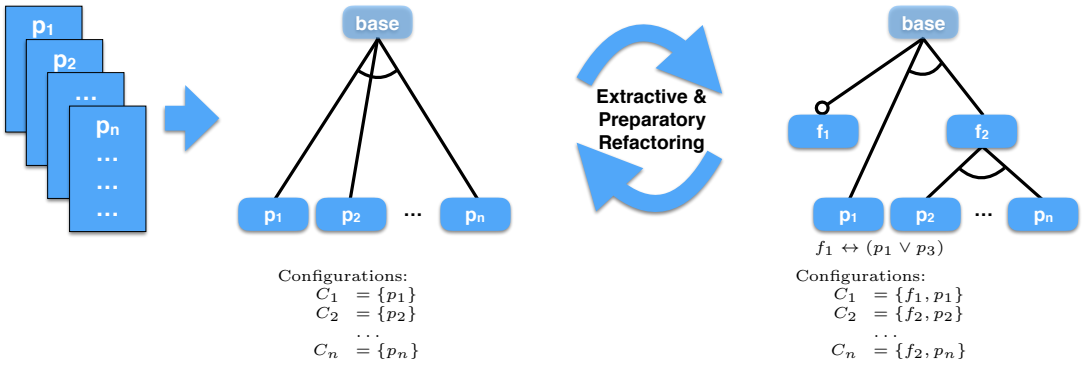
4. Remove the old definitions of m_o.

Note that these preconditions are rather liberal, thus potentially allowing renamings that are not variant-preserving. For instance, coming back to Listing 6.3, my preconditions would allow me to rename `getDiameter` to `getRadius` (or even to `getX`!), which is not a sensible change (see red highlights). I could prevent this with more restrictive preconditions, but only at the cost of precluding many useful applications, such as renaming `getCenterX` to `getX` and `getCenterY` to `getY` (gray highlights).

6.5 A Feature-Oriented Migration Process

The refactorings explained in the previous section can be combined with code clone detection to form a process for the variant-preserving migration of cloned product variants to an SPL. Since my refactorings target FOP, this is a *Many→FOP* approach in the terms of my taxonomy. I describe this process in the following paragraphs.

Initializing the SPL. I depict my migration process in Figure 6.6. Starting with n product variants, which are named $p_1, p_2, \ldots, p_n$, the first step is to create a trivial initial SPL whose FM contains only one alternative. This alternative consists of n features, $p_1, p_2, \ldots, p_n$, one for each of the original products. The source code of these products is moved without any changes into the corresponding feature modules. Together with the FM, n configurations, $C_1, C_2, \ldots, C_n$, are created, each with exactly one of the features $p_1, p_2, \ldots, p_n$ selected, while all others are deselected. Hence, it is possible to recreate the original product p_1 by choosing configuration C_1, whereas p_2 is recreated by choosing C_2, and so on.



Feature-model perspective of my migration process, showing the migration of n cloned product variants to a feature-oriented SPL via extractive and preparatory refactoring.

Figure 6.6: Feature-model perspective of my variant-preserving migration process

While this first step does not yet improve reuse, it forms the basis for the subsequent, iterative refinement process that constitutes the core of my migration approach.

Code clone extraction. The iterative refinement process encompasses two steps, *code clone extraction* and *preparatory refactoring*. These steps are executed repeatedly and will progressively reduce the amount of code clones in the features $p_1, p_2, \ldots, p_n$. To identify code clones, I use clone detection, as I will outline shortly. If possible, a code clone is removed in an extraction step by applying Pull Up To Common Feature. Recall that this refactoring removes code that is cloned across two or more features by moving this code to a single location, called a *common feature*. In the example in Figure 6.6, f_1 and f_2 are such common features.

Preparatory refactoring. If an extraction step is not directly possible, it is preceded by a preparatory refactoring, which aligns differing clones. As previously discussed for Listing 6.3, methods with identical bodies but different names cannot be pulled up. Similarly, extraction is impossible when otherwise identical classes have different names. In the terminology of code clone detection, these are examples of Type-2 clones. Thus, finding opportunities for preparatory refactoring hinges on the ability to identify such clones. Not all code clone detectors can do this, but for my process it is necessary to use a clone detector that does. This has also been taken into account for the design of my tool support. Thus, my tool helps the developer identify code that will benefit from preparatory refactoring. To align naming differences, the tool offers a Rename refactoring.

Besides differences in names, clones can differ in other ways. For instances, a developer could have added statements to a method after cloning it. Likewise, parameters may have been added or removed. These kinds of modifications lead to in Type-3 clones, which necessitate further preparatory refactorings, beyond Rename. For example, added statements could be extracted by means of an Extract Method refactoring. This shows that there are many ways in which additional preparatory refactoring would improve my tooling. However, due to the high implementation effort for a refactoring, the prototype tool currently only offers a single preparatory refactoring.

Step-wise refinement. Both steps, code clone extraction and preparatory refactoring, can be repeated as often as needed. Since each step only affects a small part of the code base, the correctness of each step can be easily verified. Moreover, as I propose to perform changes by means of variant-preserving refactoring, the external behavior of the affected variants is preserved. Hence, all variants remain in a working state, even if the migration is still ongoing. This is of special importance as it allows releasing new product versions during the migration period. From an industrial perspective, this is a major advantage over "big-bang" approaches (e. g., [6, 378]), which require migration to be completed before allowing other changes to the code.

Inter-system versus inter-feature code clones. Cloning frequently occurs within a single software product, e. g., if a method is copied from one class to another. Koschke calls these clones *intra-system* clones [182]. Although intra-system clones may constitute maintenance problems, they are not the focus of my approach because they do not originate from clone-and-own variant development. Instead, I am interested in *inter-system* clones, that is, functionality that has been copied from

one product variant to another. The challenge is to tell apart one kind of clone from the other.

In my migration process, each variant is initially converted to a distinct feature. At this point, it is still easy to distinguish between inter- and intra-system clones. But subsequently, many small refactoring steps happen, which move code from one feature to another. These code movements make it increasingly difficult to precisely track what was originally an inter- and what was an intra-system clone. In order to avoid these difficulties, I relax the concept of inter-system clones and approximate them with Inter-Feature Code Clones (i. e., clones between different features; cf. Section 4.2). Hence, my approach is to use a code clone detector and filter its results so that only clones between features are reported but not clones inside of features. As a result, developers can focus on the code clones that constitute the common functionality they need to extract, without being sidetracked by other, irrelevant clones.

6.6 Tool Support

The proposed migration process has been integrated into the Eclipse IDE. To this end, several existing tools were reused, namely FeatureIDE [358, 242], Fuji [17], the Eclipse refactoring framework, and a variability-aware extension of the *Copy/-Paste Detector (CPD)*[1] [360]. In this section, I discuss how each tool was reused in the implementation.

The basis of the tool support is FeatureIDE, an Eclipse framework for feature-oriented software development. FeatureIDE supports several phases of the development of SPLs, such as domain engineering (i. e., feature modeling), product configuration, and product generation. Notably, FeatureIDE already integrates the composer FeatureHouse, as well as the variability-aware compiler Fuji [17].

Guaranteeing that refactorings are variant-preserving requires a variability-aware AST over all product variants. Fuji's type checker is used to generate this AST.

The identification of code clones is performed by CPD, the token-based clone detector that is part of the PMD suite of static source code analysers. CPD was adapted to identify inter-feature code clones but ignore intra-feature code clones [360]. In Figure 6.7, a screenshot of the tool is shown. As the screenshot illustrates, code clones are highlighted with a warning. The warning appears as a yellow symbol next to the line number where a clone starts, and the cloned statements are underlined with a dotted yellow line. The corresponding tool tip shows how many lines are cloned and in which files the clones reside. The feature that each file belongs to is displayed in brackets before the filename. With the help of a so-called "quick fix", the developer can open all relevant files in an editor window.

Finally, the Rename and Pull Up To Common Feature refactorings were implemented based on Eclipse's refactoring framework for Java. Thus, existing machinery could be reused, such as the refactoring wizard. In Figure 6.8, an exemplary application of Pull Up To Common Feature on method `getVecY` is depicted.

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

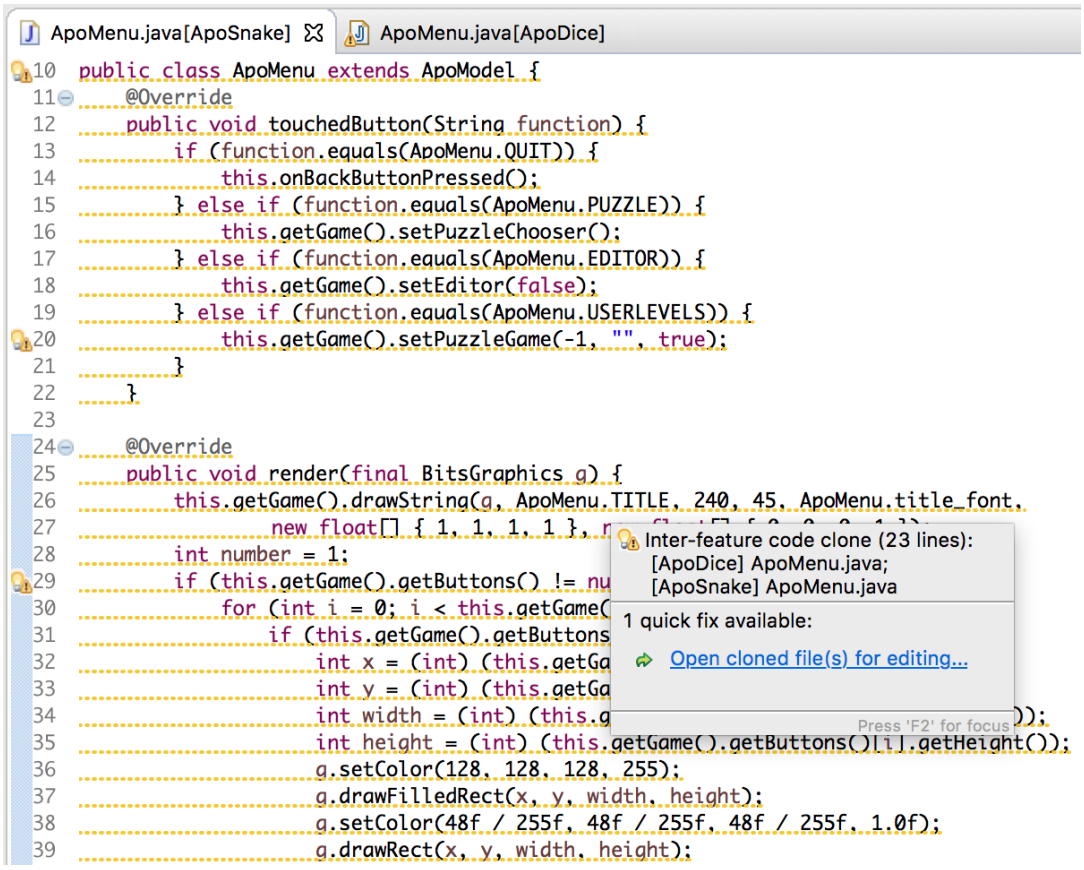


Figure 6.7: Inter-feature code clone detection with CPD

In the left dialog at (1.), the destination feature is selected. At (2.) all occurrences of the same method in other features are listed. Methods that are Type-1 clones are especially marked with the keyword "Clone" (see highlight (3.)) because these are the ones that can be pulled up safely. Other types of clones are not generally safe to pull up and, consequently, are not marked. The right part of Figure 6.8 contains the second dialog of the refactoring wizard. This dialog gives a preview of the changes that the refactoring will perform. The upper part of this preview at (4.) lists all the source files that will be modified. Depending on which file is selected in that list, the lower part of the preview at (5.) shows the planned modifications in detail.

6.7 Feasibility Study

In this section, I evaluate the effectiveness of my approach to migrate cloned product variants into an SPL. Specifically, I evaluate how effective my approach is in finding and consolidating Inter-Feature Code Clones. To this end, I answer the following research questions:

RQ 1: How much cloned code can be safely and automatically migrated using the* Pull Up To Common Feature *refactoring? An important aspect of my approach is to what extent I can automate the migration process. Hence, it is crucial to identify and migrate respective code fragments automatically.

RQ 2: How do preparatory* Rename *refactorings increase the amount of migratable code clones? As cloned products evolve, code clones diverge.

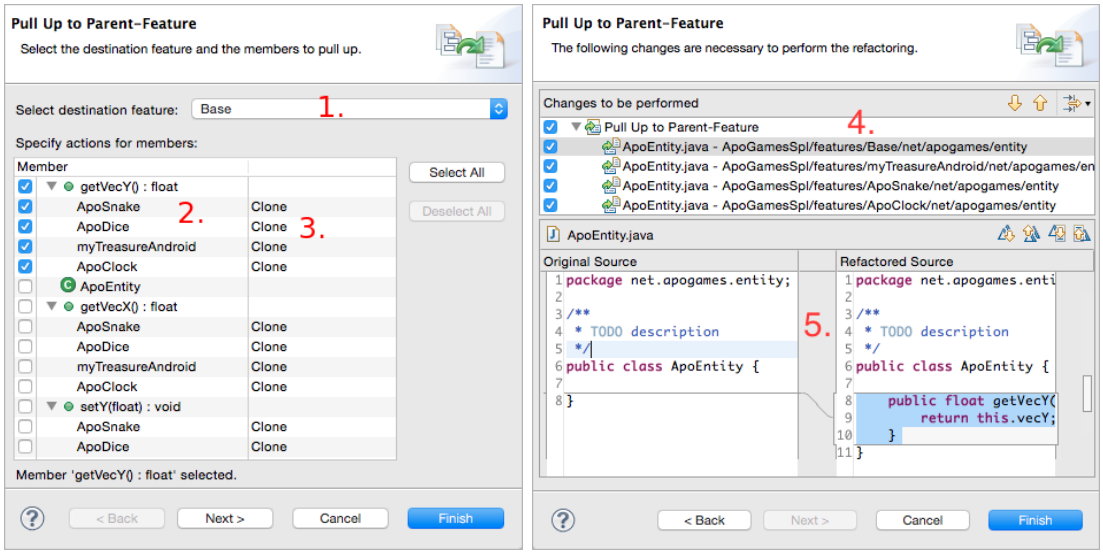


Figure 6.8: Wizard for the Pull Up To Common Feature refactoring *(Picture originally created by Steffen Schulze; reproduced from [331])*

Preparatory Rename refactorings can remove some of these divergences. I quantify their effectiveness by measuring the amount of migrated code again after renaming.

RQ 3: How much cloned code remains and why? My approach cannot migrate all Inter-Feature Code Clones into common features. With this question, I investigate which portion of the products still remains as cloned code, and why it was impossible to consolidate these clones.

6.7.1 Subject Systems

For answering my research questions, my subject systems must have the following properties. First, they must be Java programs that are cloned from each other. However, the specifics of the cloning process (e. g., forking in a VCS as opposed to manual copying) are irrelevant. Second, the variants must implement custom functionality, which will remain as variant-specific code. Third, the variants must have evolved over time. Evolution may lead to shared code that is modified without synchronizing the modifications to other variants. Such modifications allow me to study the nature of divergences and to gauge the effectiveness of my preparatory Rename refactoring.

In this evaluation, I use five programs of the ApoGames,[2] which are listed in Table 6.5. These programs are diverse games for Android written in Java. They originated from manual copying and subsequently evolved independently. In Table 6.5, I report the size of the programs in *non-blank, non-comment lines of code (LOC)* (as measured by cloc[3]) and how much of each program is identified as Inter-Feature Code Clones. The latter metric is given in the LOCC (line of cloned code) column. It was computed by CPD, which was configured to detect only clones with a minimum of ten cloned tokens. As the code clone rates (CCR)

[2] http://apo-games.de/index_android.php
[3] https://github.com/AlDanial/cloc

show, all programs have a high portion of inter-feature code clones, between 57.2 % and 80.0 %.

Product	LOC	LOCC	CCR
ApoClock	3,584	2,643	73.7 %
ApoDice	2,504	2,003	80.0 %
ApoMono	6,483	4,382	67.6 %
ApoSnake	2,946	2,350	79.8 %
myTreasure	5,322	3,042	57.2 %
total	20,839	14,420	69.2 %

LOC: Lines of code; **LOCC:** Lines of code clones; **CCR:** Code clone rate

Table 6.5: Statistics on the subject systems

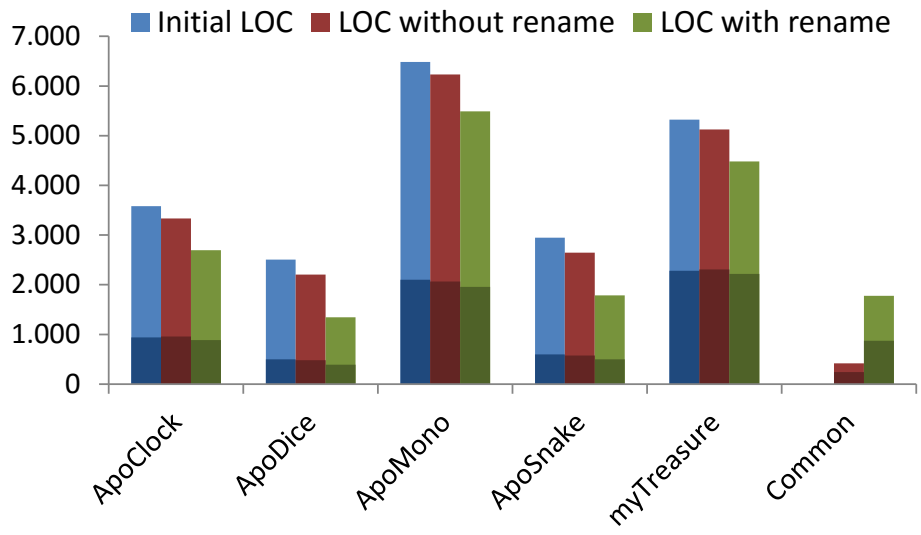
6.7.2 Methodology

I use the following methodology. First, the cloned products are transfered into a trivial product line as described in Section 6.5. Second, Pull Up To Common Feature is automatically applied to all program elements (methods, fields, etc.) that constitute Type-1 clones. After this step, no more methods or fields can be refactored using Pull Up To Common Feature. By measuring the remaining LOC and LOCC of the individual feature modules and of the extracted common modules, I answer ***RQ 1.*** Third, I manually apply preparatory Rename refactorings to all methods, fields and classes that only differ in name but not in content. Afterwards, I apply Pull Up To Common Feature again and measure LOC and LOCC once more. This is followed by an inspection of the remaining source code in the feature modules that correspond to the five original variants. The repeated measurements and the manual inspection provide the answers to ***RQ 2*** and ***RQ 3,*** respectively.

6.7.3 Results

I applied my methodology to the five ApoGames. In Figure 6.9, I show the LOC and LOCC measurements of each feature module of the five variants, as well as the LOC and LOCC of the modules for common code. For each module, I show from left to right (1) the initial LOC before migration, (2) the LOC after applying the Pull Up To Common Feature refactorings, and (3) the final LOC after preparatory Rename and Pull Up To Common Feature. Additionally, I report the lines that are identified as Inter-Feature Code Clones in the upper part (illustrated with a brighter color).

In the first step, 110 fields could be extracted into 32 common fields and 291 methods into 74 common methods. All extracted code belonged to three distinct classes. Overall, LOC were reduced by 4.2 % (879 LOC) and LOCC by 7.6 % (1,095 LOCC). In doing so, nine new features for common code were created. The common code size is 419 LOC, of which 187 LOC are shared among all variants.



Lines of code in each migration step of each program variant and amount of common code. Detected code clones are illustrated in brighter color.

Figure 6.9: Reduction in lines of code during feasibility study *(Chart originally created by Jens Meinicke; reproduced from [95])*

In the second step, 84 classes and eight methods were identified as candidates for preparatory refactoring. Their names were aligned using the Rename refactoring. After preparation, it was possible to pull up 473 additional fields into 150 common fields and 862 methods into 245 common methods. The pulled up code belonged to 26 distinct classes, whereas before preparation, there were only three such classes. Compared to the initial variants, the overall LOC was reduced by 15.6 % (3,259 LOC) and the LOCC by 25.4 % (3,664 LOCC). The final size of the common code is 1,779 LOC. The final overall size of the product line is 17,580 LOC, out of which 10,756 constitute clones (CCR is 61.2 %). The size of the variant-specific code (i. e., not considering the code in the common features) was reduced by 23.2 % (5,038 lines) to 15,801 LOC, out of which 9,850 are still cloned. The final feature model has 15 common features. The largest one of these common features contains 619 LOC that are shared among all variants. The second-largest one encompasses 466 LOC, which are shared between ApoDice and ApoSnake.

6.7.4 Discussion

Regarding ***RQ 1,*** Pull Up To Common Feature only reduced the code size by 879 LOC (4.2 %). Code was migrated from only three out of 114 distinct classes. The reason for the rather low reduction was a peculiar naming convention that required class names to have a variant-specific prefix. For instance, the menu class was called `ApoClockMenu` in ApoClock but `ApoDiceMenu` in ApoDice. It was necessary to revert these changes first, which leads to ***RQ 2.***

During preparatory refactoring, eight methods and 84 classes were renamed, reducing the number of distinct classes to 56. This made Pull Up To Common Feature applicable to a considerably larger amount of cloned code. After renaming, code could be pulled up from 26 distinct classes. This second round of Pull

Up To Common Feature reduced the variants' code by an additional 3,300 LOC or 15.8 %. I argue that by applying both extractive and preparatory refactorings, I substantially reduced code clones and fostered systematic reuse. Also, the number of additional features (15 in this case study), is still manageable, which indicates that my ensureTargetFeature algorithm avoids creating unnecessarily complex feature models.

Regarding ***RQ 3,*** there remain 9,850 LOCC. I reviewed these code clones and found that most of them come from large, similar methods with minimal customizations. Much of this code is graphics-related. These customizations caused a great number of missed refactoring opportunities. The reason is that Pull Up To Common Feature can only be applied safely and in a fully automated fashion if methods are completely identical. To consolidate further code clones, local customizations must be extracted using other preparatory refactorings, such as Extract Method and Extract Constant. Corresponding variant-preserving refactorings are part of future work. I also observed that different variants use different versions of third-party libraries. The conflicting *Application Programmer Interfaces (APIs)* of these library versions were another source of divergences that prevented the consolidation of clones. Possible solutions would be to either change all variants to use the same library version or to introduce a façade pattern to abstract from the API differences. However, neither solution lends itself well to automation.

6.7.5 Threats to Validity

My tool support integrates several external tools and a new implementation of variant-preserving refactoring. To ensure the validity of the implementation, unit tests for both refactorings were developed. Furthermore, refactoring results were manually inspected. Finally, all five variants were generated after the migration. No compile errors occurred.

CPD was used for code clone detection, which was configured so that clones must share at least ten tokens to be reported. Configuring this threshold is a balancing act. On the one hand, if the threshold is too high, some short clones are not reported, recall suffers, and clone consolidation opportunities are missed. On the other hand, if the threshold is too low, many meaningless clones are reported, precision degrades, and clones that are interesting candidates for consolidation become hard to find. How does this affect the quality criterion of my evaluation, which is the reduction in code clones? It has two opposite effects. On the one hand, had I used a higher threshold, I would have missed more opportunities to consolidate clones. On the other hand, the overall number of clones reported would have decreased. Thus, the *reduction in code clones* – the number of consolidated clones in relation to the number of reported clones – may very well have stayed the same. I argue that the ten tokens used in my evaluation are a good compromise. Recall that each identifier, each opening or closing parenthesis, and each semicolon counts as a single token. Hence, even the simple Java `get` method depicted to the right, which consists of exactly ten tokens, would have been reported. Thus, I only failed to consolidate some cloned fields or methods without a visibility qualifier. I argue that not con-

```
public int getFoo() {
  return foo;
}
```

solidating them is an acceptable error because such short clones encode very little logic.

In my evaluation, the preparatory Rename refactoring highly increased the opportunities to consolidate cloned methods and fields. This effect is caused by the previously mentioned naming convention in the ApoGames case study. For other systems, I do not expect Rename to have such a high impact, but I expect the total reduction in code clones to be comparable, if not higher.

The goal of clone-and-own is to reuse existing code and to integrate customizations. The ApoGames are arguably an extreme case because the cloning and customization was highly unstructured. Essentially, a dice-rolling game was copied and modified arbitrarily until it became a snake game, which is entirely something else. This is not the same as copying an online shopping system, for example, and customizing its design to fit one company or another. For this reason, I expect other program families to contain much less custom code than the ApoGames and much more reused code. As the effectiveness of my migration approach depends on how much code is reused, I should be able to extract larger shared components in other program families.

My implementation and refactorings currently only work for Java programs. Nevertheless, the process and underlying concept can be applied to other programming languages, as they all come with elements (e. g., functions and variables) that can be refactored. This will require specialized tools, such as for the generation of the variability-aware AST.

6.8 Related Work

At the beginning of this chapter, I presented a taxonomy of reengineering techniques for SPLs. In the following paragraphs, I discuss how my taxonomy relates to other taxonomies and surveys in the SPLE field. After the taxonomy, I proposed an approach to migrate legacy software products to an SPL. A migration approach requires both, reverse engineering techniques to uncover commonalities and variations, as well as reengineering techniques to actually transform the legacy code. Therefore, a considerable amount of existing work is related to mine. I chose code clone detection for the reverse engineering part and variant-preserving refactoring for the reengineering part, but other researchers made different choices. Moreover, reverse engineering and refactoring for SPLs, as well as clone detection and removal have also been explored separately, without the migration context. The next paragraphs will clarify how my work relates to this body of knowledge.

Other taxonomies and surveys. Laguna and Crespo have performed a systematic mapping study on software product line evolution [192]. The scope of their survey is broader than mine. Whereas I concentrate on techniques that change a software system, they also discuss work on processes, organizational issues, and metrics. Laguna and Crespo acknowledge the diversity of reengineering terms (e. g., refactoring, migration, restructuring), yet they do not provide a taxonomy. I extend their work by deriving three dimensions of SPL reengineering and constructing a detailed taxonomy from these dimensions. Furthermore, I reclassify a part of their corpus, but also include additional literature.

Krueger has offered two taxonomies for the field of SPLE [185, 186]. His first taxonomy consists of three approaches to SPLE adoption, the *extractive, reactive,* and the *proactive* approach [185]. This taxonomy complements mine as Krueger dicusses the overall processes, whereas I focus on concrete techniques. In the *extractive approach,* an SPL is built from one or several legacy application(s). In terms of my taxonomy, variant-preserving migration techniques can be employed for this task. In the *reactive approach,* an existing SPL is extended in order to satisfy new requirements. Any restructuring this might require can be achieved through what I call variant-preserving refactoring. However, if features have to be separated from the common code, migration techniques could also be employed. Finally, the *proactive* approach is presented, which equates to constructing an SPL from scratch. This is forward engineering, not reengineering and hence unrelated to my work.

Krueger's second taxonomy covers differences in binding times of variability mechanisms (e. g., clone-and-own at development-time, preprocessors at compile-time, or configuration files at runtime) as well as the evolutionary changes of product line artifacts and how to propagate them [186]. I only distinguish between annotation- and composition-based variability mechanisms, and mostly cover compile-time mechanism. Adding binding time as a subordinate dimension to the *Variability mechanism* dimension of my taxonomy would allow for a more fine-grained classification. The other factor that Krueger considers, how to manage evolutionary changes, is related to forward engineering and therefore beyond the scope of my taxonomy. Krueger furthermore references two topics he covered in his first taxonomy, namely scoping strategies for an SPL (planning far into the future versus adjusting to new requirements reactively) and SPLE adoption approaches (starting an SPL from scratch as opposed to the extractive approach). As already discussed for Krueger's first taxonomy, he focuses on processes, whereas my taxonomy encompasses techniques to implement these processes.

Svahnberg et al. present a comprehensive taxonomy of variability mechanisms [354]. Like Krueger, they consider binding time as a factor to distinguish between mechanisms, but they include other factors, too. The goal of their taxonomy is to offer advice on how to choose the most appropriate mechanism. Companies could use such advice to decide between the variant-preserving migration techniques that I cover in my taxonomy. Thus, their taxonomy and mine complement each other.

Migration to an SPL. As mentioned in Section 6.2.2 and Section 6.3, there are several approaches to migrate legacy software products to an SPL. Most only migrate a *single legacy product* (e. g., [211, 60, 368]) and therefore fail to address the challenge of identifying and consolidating commonalities (and differences) between multiple variants. Those approaches that migrate *multiple legacy products* (e. g., [6, 316, 376, 377]) focus on models but lack details on how to actually transform the source code. The description of my refactorings fills in those details. Moreover, my migration process is step-wise and incremental, whereas others happen in one "big bang". This makes my process more flexible and less risky.

Others have proposed frameworks for migrating cloned product variants [317, 235]. These frameworks abstract migration activities such as similarity analysis or merging of commonalities. My approach specifically uses clone detection for similarity

analysis and refactorings for merging commonalities. Hence, my approach can be seen as an instantiation of these frameworks.

Antkiewicz et al. propose the *virtual platform* as another alternative to existing risky and disruptive big-bang migration approaches [12]. The core idea of the virtual platform is that that clone-and-own and SPLE are not irreconcilable opposites. Instead, there is a spectrum of approaches in between, ranging from ad-hoc clone-and-own on one end to SPLE with a fully integrated platform on the other end. Antkiewicz et al. propose six governance levels to distinguish between these approaches and explain how to companies can transition from a low level to a higher level by changing their development process. Hence, they also outline an iterative process to go from clone-and-own to SPLE. But whereas I focus on iterative changes on the implementation level, they focus on iterative changes to the development process.

Beyond concrete approaches and abstract frameworks, the literature contains several case studies on migrating multiple legacy systems into an SPL (e. g., [48, 178, 151]). Zhang et al., in turn, derive an agile, incremental process from their own experiences in an industrial migration project [387]. Such case study reports concentrate on processes and lessons learned but not on techniques at the modelling or implementation level. Hence, these reports are complementary to my work.

Reverse-engineering of variability. There are multiple approaches to reverse-engineer commonalities and variations from the source code of cloned variants [244, 183, 234, 391, 179]. Specifically, Mende et al. used clone detection to identify (but not consolidate) core functionality in SPLs that were evolved by unmanaged, large-scale clone-and-own [244]. Koschke et al. recover individual product architectures and combine them into a common product line architecture [183]. Others propose program-dependency graphs [179] or architecture reengineering techniques [234] to identify product-specific variations.

Similarly to my work, some of these approaches incorporate code clone detection [244, 183, 234]. But code clone detection is just one technique to support the analysis phase of a migration project. Another popular technique is feature location, which uncovers the features that are present in a given legacy system and establishes feature-to-code mappings. To this end, *Concern Graphs, Program Dependence Graphs (PDGs), Formal Concept Analysis (FCA)*, dynamic and iterative procedures, among others, have been proposed [85, 160, 291, 307, 368, 375]. Detailed information can be found in the surveys by Rubin and Chechik [315], Dit et al. [75], as well as Assunção and Vergilio [21]. Many feature location techniques only apply to a single product, so they cannot be integrated into my approach right away, but some also apply to multiple variants (e. g., [391, 209]). Feature location techniques can identify features in their entirety, which clone detection alone cannot. Thus, incorporating feature location into my approach may lead to more cohesive, better structured results.

Weber et al. report initial insights from a case study in the health care domain [373]. The starting point is an open-source software system whose VCS repository was forked multiple times to customize it for different users (e. g., hospitals and private practices) and for different health care legislations. They propose to migrate this multi-repository structure into an SPL by analyzing the VCS histories so that a variability model can be created. This model, in turn, should serve as a guide for

a subsequent merging and refactoring phase. Although a feasibility study remains as future work, the proposal is interesting because it aims to exploit the evolutionary information that only VCS histories can offer. Other approaches (mine included) ignore this information since they only take the current state of each product variant into account.

While the aforementioned work focuses on analyses, I focus on the source-code transformations to execute the actual migration. Due to the higher level of abstraction, their analyses are complementary to my approach.

Refactoring of variable source code. Refactorings for different variability mechanisms have been explored, among them aspect-, delta- and feature-oriented programming, as well as C code with preprocessor annotations [7, 329, 328, 205]. My Pull Up To Common Feature refactoring is a generalization of the Pull Up Method To Parent Feature refactoring that Schulze et al. proposed earlier [328]. Contrary to my work, these refactorings either lack the migration context [329, 328, 205, 174, 175] or they neglect the challenges of identifying and consolidating commonalities in multiple variants [7] because they are geared toward single-system migration.

More recently, Kim et al. presented the X15 refactoring engine for Java SPLs that use a custom annotation scheme to encode variability [174, 175]. X15 currently supports the impressive amount of 34 refactorings. This was accomplished by "lifting" the precondition checks of an existing OOP refactoring engine to make it variability-aware. According to Kim et al., the lifting approach saves a lot of implementation effort compared to the traditional way of building variability-aware refactoring engines. From a maintenance point of view, this is an important advantage.

Since X15 is based on an OOP refactoring engine, an interesting question arises, which Kim et al. do not answer: Can X15 also perform refactorings that have no direct counterpart in OOP? For example, can it move code across feature boundaries, as my Pull Up To Common Feature refactoring does, or modify feature models and configurations? If not, how much effort would be necessary to extend X15?

Clone detection and removal. Much work has discussed how to detect and remove code clones from single (non-configurable) software systems (see in Section 3.6.2 and Section 3.6.4 in Chapter 3 of this thesis for summaries). While most of this work treats either detection or removal, some also combines both topics (e. g., [136, 326]), like I do in my migration process. The only clone consolidation refactoring in my process is Pull Up, but further refactorings, such as Form Template Method, have been proposed in the literature (e. g., [136, 199]). These refactorings would make my approach more flexible and effective, but to do so, they must be extended in a variant-preserving way.

Whereas the above work is restricted to single software systems, Schulze et al. focus on code clones in SPLs, also touching the topic of clone removal [330, 323, 328]. I built my variant-preserving refactorings on these conceptual foundations and apply them in a new context, the migration of cloned product variants.

Managing clone & own without migration. Whereas my approach is to migrate a product family to an SPL, there are alternatives that do not require a migration

but still facilitate maintenance [350, 357, 121, 293, 348]. For example, Pfofe et al. propose an IDE extension to facilitate propagating changes from one cloned variant to another [293]. In addition, their extension records feature-to-code mapping, which may be used for a migration later on. Stănciulescu et al., in turn, showed that projectional editing and VCSs can be integrated with each other to yield a variation control system [348].

6.9 Conclusion

"Refactoring" for single software systems has a well-established meaning: It is a behavior-preserving change to the structure of the source code to achieve a quality improvement. By contrast, SPLE literature uses the term for rather diverse (behavior-preserving) reengineering activities. To help distinguish these activities from each other, I have presented a taxonomy in this chapter that separates these activities into three categories: variant-preserving migration, variant-preserving refactoring, and variant-preserving transcoding. I have proposed definitions for these categories and shown how they relate to SPLs and (non-configurable) legacy software. Moreover, I have classified a corpus of existing work based on the conceptual framework of my taxonomy.

Furthermore, I contributed a semi-automatic, incremental migration process that focuses on the source code aspects of migrating families of cloned software products to an SPL. The core idea is to provide tool support for the tedious and error-prone migration tasks while leaving the design decisions in the hands of the developer. My process relies on code clone detection to identify commonalities of cloned product variants and on variant-preserving refactorings to extract these commonalities into shared artifacts. The process was evaluated in a case study involving five software products.

Based on the insights discussed in this chapter, I answer $\mathbf{RQ_T\ 2}$ of this thesis – *How can clone-and-own product families be migrated in a variant-preserving manner?* – as follows: Regarding, $\mathbf{RQ_T\ 2.1}$, my classification of existing literature according to my taxonomy indicates that little work on *variant-preserving refactoring* exists and so far is limited to a few variability mechanisms. Automation and tool support remain a particular challenge. Consequently, refactoring in the context of highly configurable software systems considerably lags behind the state of the art in the (non-configurable) OOP world. The variant-preserving Pull Up To Common Feature and Rename refactorings that I presented in this chapter are a small step toward rectifying this imbalance. Beyond their use in migration, I argue that these refactorings will also be useful for general maintenance and evolution activities in (FOP-based) SPLs.

Regarding $\mathbf{RQ_T\ 2.2}$, my literature classification shows that most work on *variant-preserving migration* only takes a single legacy system into account, whereas the migration of multiple legacy systems (e. g., product families created through clone-and-own) has received much less attention. Moreover, most existing approaches for multiple systems attempt migration in one "big bang" and ignore the source-code level. By contrast, the process I propose in this chapter is incremental and especially takes the source-code aspects of migration into account. During evaluation

on a case study, the amount of cloned code could be reduced by 25 %. Although this is a modest reduction, it is enough to demonstrate that my approach is at least feasible.

Regarding **RQ$_\mathbf{T}$ 2.3**, the experiences from the case study highlighted two important limitations. First, many clones could not be consolidated because they exhibited minor variations, such as added or modified statements. Additional preliminary refactorings, for example a variant-preserving Extract Method refactoring, would help prepare such clones for consolidation. Second, the purely source-code-centric view of my approach may not lead to optimal results in terms of the target architecture or the feature model. I envision that incorporating analyses on more abstract levels, such as architectural views and feature location techniques, will guide refactoring decisions toward an improved product line architecture and better code quality.

7. Conclusion and Future Work

In this chapter, I summarize the main conclusions of this thesis and discuss avenues of future work regarding code smells and refactoring of highly configurable software systems.

7.1 Conclusion

Code smells are indicators of deficient software design choices which manifest themselves in poorly structured source code. Refactoring, in turn, is a technique to improve these deficient designs by changing the structure of the source code while preserving the behavior. Research has shown that "smelly" code is associated with various problems and that refactoring can alleviate these problems. However, the vast majority of this research has focused on OOP and thus ignored the unique challenges that developers of highly configurable software systems face.

Highly configurable software systems are software systems in which the code base is no longer fixed but instead is subject to variability. For example, different pieces of code can complement each other or they may mutually exclude each other. In this thesis, I have investigated how this variability affects the code smell concept and how it affects refactoring.

In Chapter 4 of this thesis, I have argued that variability at the source code level poses new design challenges, which, in turn, create new opportunities to make bad design decisions. As a novel concept to identify these bad design decisions, I propose *variability-aware code smells* – code smells that explicitly take variability into account. Moreover, I have discussed how different variability mechanisms shape the source code differently and how these different shapes change the appearance of a given variability-aware code smell. Based on this reasoning, I have presented an initial catalog of six variability-aware code smells. Furthermore, I have developed a concept and tool support to detect variability-aware code smells in highly configurable software systems written in C and using CPP directives (a. k. a. `#ifdef`s) as the variability mechanism. Based on a survey and by applying my tool support in a case study, I demonstrated that variability-aware code smells exist in real-world

software and that they can pose problems regarding program comprehension. Interestingly, I also encountered a number of counterexamples in my case study, which only appeared problematic at first, but at closer inspection, revealed patterns of implementing variability that were harmless or even beneficial.

One of the variability-aware code smells proposed in this thesis is the Annotation Bundle, which is a function in which the use of cpp directives is excessive. In Chapter 5, I investigated the potential negative effects of this smell in greater depth. In particular, I investigated how different extents of using the various facilities of the cpp affects maintainability in terms of the frequency and extent of changes. The results show that after controlling for other confounding factors, the fine-grained use of preprocessor directives has litte or no systematic effect on maintainability. Given these results, it appears partly unjustified that the C preprocessor is criticized so frequently in the literature. Furthermore, I observed that truly great extents of cpp use are very rare in real-world source code, which implies that most maintainability problems in a highly configurable software system are unrelated to variability.

Combining the findings of Chapters 4 and 5, my conclusion about variability-aware code smells is ambiguous: On the one hand, I identified certain patterns of implementing variability that hinder code comprehension. On the other hand, I found no evidence that these patterns also negatively affect maintainability, and even if they do, most maintainability problems likely have other causes. Thus, if the general goal is to identify problematic source code in a highly configurable software system, then focusing *only* on variability implementation patterns is too restrictive.

The last chapter of this thesis, Chapter 6, explored refactoring of highly configurable software systems, both on the theoretical and on the practical level. On the theoretical level, I have shown that SPLE literature uses the term "refactoring" for many (behavior-preserving) reengineering activities, which often have little to do with improving the internal structure of a software system. To differentiate between these activities, I have proposed a taxonomy of SPL reengineering activities and classified a corpus of existing work. This taxonomy and literature classification will bring clarity to future discussions about SPL-related reengineering activities and help researchers and practitioners alike match available solutions to the problems at hand. On the practical level, I have investigated the usefulness of refactoring for migrating a family of software products created through clone-and-own to an SPL. Combining previous work on variant-preserving refactoring with code clone detection, I developed an incremental, semi-automatic migration process. The core idea is to identify common functionality in the code bases of the products being migrated and to refactor this common functionality into shared artifacts. I demonstrated that this approach is feasible and argue that it constitutes an important contribution to existing work on migration, which often ignores the source-code level aspects. Moreover, my refactorings will also be beneficial in other contexts since their applicability is not limited to migration.

7.2 Future Work

While I was working on this thesis, several ideas formed how this work could be extended and how its various limitations might be overcome. There is only so much

that can be done in a single thesis and so I pursued only a few of these ideas myself. The others I list here as potential avenues of future work.

Extending the catalog of variability-aware smells. The catalog of variability-aware code smells I proposed in this thesis is still small and should be extended in the future. In addition to deriving more variability-aware smells from existing smell catalogs, it would be interesting to solicit feedback from practitioners in order to learn from their experiences with implementing variability in real-world software. Not only may such feedback lead to more smells, it may also uncover best practices of implementing variability and thus lead to a catalog of *variability-aware design patterns.*

The variability of highly configurable software systems is encoded not only in the source code, but also on other levels, and in particular on the level of the build system. Therefore, another possible direction of future work is to investigate smells and design patterns for build-system-based variability mechanisms.

Variability-aware smell detection. The metrics-based detection approach proposed in Chapter 4 lacks precision as it reports many false-positives. On the one hand, these false-positives provided unexpected and highly interesting insights as I believe that some of them encompass beneficial patterns of implementing variability. On the other hand, false-positives are undesirable from a practitioner's point of view and should be excluded. Promising directions for improving detection precision include choosing better thresholds (e. g., through machine learning) and taking other metrics into account (e. g., information about the use of runtime `if` statements). Furthermore, it will be necessary to gain a deeper understanding of the properties that make the difference between harmful and beneficial patterns. Such an understanding will be needed to guide the search for better detection algorithms.

Negative effects of variability-aware code smells. In Chapter 5, where I investigated the effect of preprocessor use on change-proneness, I presented a comprehensive methodology and tool infrastructure. This methodology and infrastructure could also be applied to analyze the relationship of preprocessor use to other properties of maintenance and evolution. One possible objective is to incorporate data from issue tracking systems to study the effect on fault proneness from a different angle. Other aspects, for instance, co-changes, are of interest, too.

Another line of future work is to use my data and tools to gain a better understanding of change proneness in general, outside of the context of preprocessor use. The qualitative analysis in Chapter 5 already hinted at the wealth of underlying causes for the change proneness or stability of functions. A more comprehensive qualitative analysis will deepen the understanding of these causes and possibly lead to more accurate predictions of maintenance effort and thus benefit practitioners in cost estimations.

Code smells can affect many aspects of software development, such as code comprehension, maintainability, evolvability, and fault-proneness. Many methods have been used in the literature to investigate these aspects, including questionnaires, controlled experiments, and interviews. In this thesis, I focused on two of these

aspects (code comprehension and maintainability) and employed surveys, code inspection, and repository mining as my methods of investigation. To reach a more comprehensive understanding, future work should analyse the relationship between variability-aware code smells and additional aspects, such as fault-proneness. Furthermore, different methods, such as controlled experiments, should be incorporated to gather complementary insights.

Variant-preserving refactoring and migration. Future work to improve my migration approach includes the design and implementation of additional refactorings to enable the extraction of more code clones than currently possible. It will also be interesting to integrate the X15 refactoring engine because X15 is already capable of many variant-preserving refactorings. However, such an integration would entail major changes to my current tooling as X15 uses its own variability mechanism. Besides additional refactorings, my approach could be extended to other variability mechanisms. Of particular interest are annotation-based mechanisms, such as the popular combination of C with preprocessor directives.

In my migration case study, I noticed that few of the extracted shared features were actually cohesive and meaningful. I believe that this limitation results from an overly strong focus on the source-code level. It should be possible to overcome this limitation by combining my work with analyses on more abstract levels, such as architectural views and feature location techniques.

A. Appendix

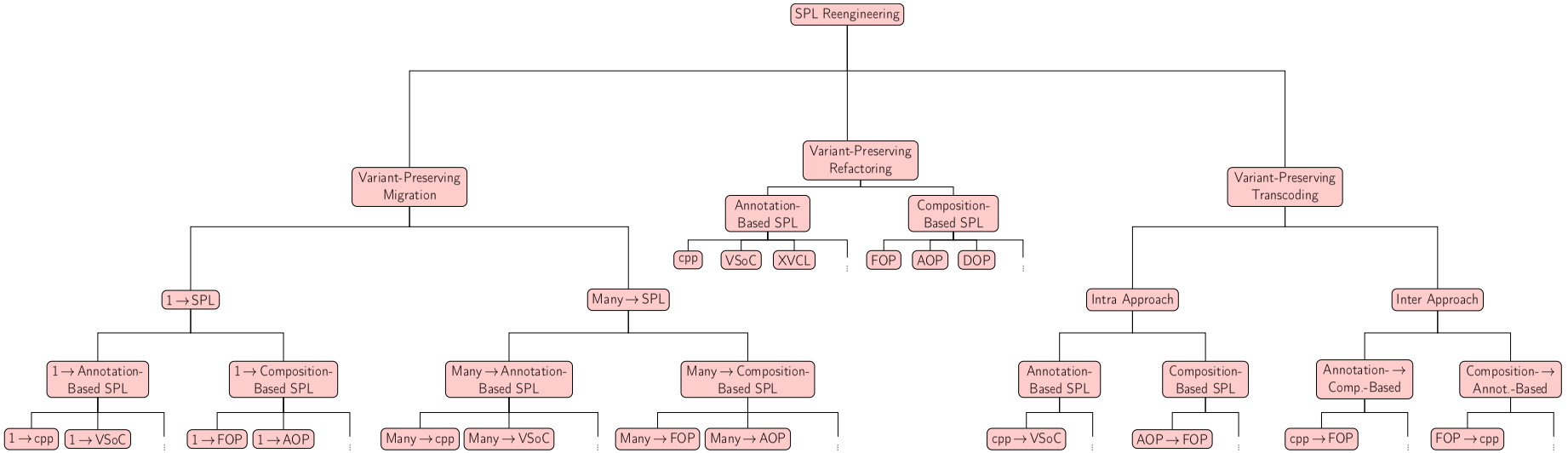


Figure A.1: Full taxonomy of SPL reengineering techniques

Bibliography

[1] M. Abbes, F. Khomh, Y.-G. Guéhéneuc, and G. Antoniol. "An Empirical Study of the Impact of two Antipatterns, Blob and Spaghetti Code, on Program Comprehension". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '11)*. IEEE, 2011, pp. 181–190.

[2] B. Adams, W. De Meuter, H. Tromp, and A. E. Hassan. "Can We Refactor Conditional Compilation Into Aspects?" In: *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD '09)*. ACM, 2009, pp. 243–254.

[3] *Airbnb JavaScript Style Guide*. 2014. URL: https://github.com/airbnb/javascript (visited on 05/31/2017).

[4] R. L. Akers, I. D. Baxter, and M. Mehlich. "Re-Engineering C++ Components Via Automatic Program Transformation". In: *Proc. ACM Symp. on Partial Evaluation and Program Manipulation (PEPM '04)*. ACM, 2004, pp. 51–55.

[5] V. Alves, F. Calheiros, V. Nepomuceno, A. Menezes, S. Soares, and P. Borba. "FLiP: Managing Software Product Line Extraction and Reaction with Aspects". In: *Proc. Int'l Software Product Line Conf. (SPLC '08)*. IEEE, 2008, p. 354.

[6] V. Alves, R. Gheyi, T. Massoni, U. Kulesza, P. Borba, and C. Lucena. "Refactoring Product Lines". In: *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE '06)*. ACM, 2006, pp. 201–210.

[7] V. Alves, P. Matos Jr., L. Cole, A. Vasconcelos, P. Borba, and G. Ramalho. "Extracting and Evolving Code in Product Lines with Aspect-Oriented Programming". In: *Trans. Aspect-Oriented Softw. Development IV*. Springer, 2007, pp. 117–142.

[8] V. Alves, P. Matos, L. Cole, P. Borba, and G. Ramalho. "Extracting and Evolving Mobile Games Product Lines". In: *Proc. Int'l Software Product Line Conf. (SPLC '05)*. Rennes, France: Springer, 2005, pp. 70–81.

[9] S. W. Ambler and P. J. Sadalage. *Refactoring Databases: Evolutionary Database Design*. Boston, MA, USA: Addison-Wesley, 2006.

[10] H. S. de Andrade, E. Almeida, and I. Crnkovic. "Architectural Bad Smells in Software Product Lines: An Exploratory Study". In: *Companion to the Proc. Working Conf. on Software Architecture (WICSA '14)*. ACM, 2014, 12:1–12:6.

[11] *Antenna: An Ant-to-End Solution for Wireless Java*. 2010. URL: http://antenna.sourceforge.net/ (visited on 10/15/2018).

[12] W. Antkiewicz Michałand Ji, T. Berger, K. Czarnecki, T. Schmorleiz, R. Lämmel, Ş. Stănciulescu, A. Wąsowski, and I. Schaefer. "Flexible Product Line Engineering with a Virtual Platform". In: *Companion to the Proc. Int'l Conf. on Software Engineering (ICSE '14)*. ACM, 2014, pp. 532–535.

[13] S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines – Concepts and Implementation*. Berlin Heidelberg, Germany: Springer, 2013.

[14] S. Apel, C. Kästner, M. Kuhlemann, and T. Leich. "Pointcuts, Advice, Refinements, and Collaborations: Similarities, Differences, and Synergies". In: *Innov. Syst. Softw. Eng.* 3.4 (2007). ISSN: 1614-5046.

[15] S. Apel and C. Kästner. "An Overview of Feature-Oriented Software Development". In: *J. Object Technol.* 8.5 (2009), pp. 49–84.

[16] S. Apel, C. Kästner, and C. Lengauer. "Language-Independent and Automated Software Composition: The FeatureHouse Experience". In: *IEEE Trans. Softw. Eng.* 39.1 (2013), pp. 63–79.

[17] S. Apel, S. Kolesnikov, J. Liebig, C. Kästner, M. Kuhlemann, and T. Leich. "Access Control in Feature-oriented Programming". In: *Sci. Comput. Prog.* 77.3 (2012), pp. 174–187.

[18] S. Apel, C. Lengauer, B. Möller, and C. Kästner. "An Algebraic Foundation for Automatic Feature-Based Program Synthesis". In: *Sci. Comput. Prog.* 75.11 (2010), pp. 1022–1047.

[19] F. Arcelli Fontana, M. V. Mäntylä, M. Zanoni, and A. Marino. "Comparing and Experimenting Machine Learning Techniques for Code Smell Detection". In: *Empir. Softw. Eng.* 21.3 (2016), pp. 1143–1191.

[20] F. Arcelli Fontana and M. Zanoni. "Code Smell Severity Classification Using Machine Learning Techniques". In: *Knowledge-Based Systems* 128 (2017), pp. 43–58.

[21] W. K. G. Assunção and S. R. Vergilio. "Feature Location for Software Product Line Migration: A Mapping Study". In: *Proc. Int'l Software Product Line Conf. (SPLC '14) Companion*. ACM, 2014, pp. 52–59.

[22] C. Atkinson, J. Bayer, and D. Muthig. "Component-Based Product Line Development: The KobrA Approach". In: *Proc. Int'l Software Product Line Conf. (SPLC 1)*. Kluwer Academic Publishers, 2000, pp. 289–309.

[23] L. Aversano, M. Di Penta, and I. D. Baxter. "Handling Preprocessor-Conditioned Declarations". In: *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM '02)*. IEEE, 2002, pp. 83–92.

[24] B. S. Baker. "A Program for Identifying Duplicated Code". In: *Computing Science and Statistics: Proc. Symp. on the Interface (CSS '92)*. Interface Foundation of North America, 1992, pp. 49–57.

[25] B. S. Baker. "On Finding Duplication and Near-Duplication in Large Software Systems". In: *Proc. Working Conf. on Reverse Engineering (WCRE '95)*. IEEE, 1995, pp. 86–95.

[26] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. "Advanced Clone-Analysis to Support Object-Oriented System Refactoring". In: *Proc. Working Conf. on Reverse Engineering (WCRE '00)*. IEEE, 2000, pp. 98–107.

[27] M. Balazinska, E. Merlo, M. Dagenais, B. Lagüe, and K. Kontogiannis. "Partial Redesign of Java Software Systems Based on Clone Analysis". In: *Proc. Working Conf. on Reverse Engineering (WCRE '99)*. IEEE, 1999, pp. 326–336.

[28] V. R. Basili, L. C. Briand, and W. L. Melo. "A Validation of Object-Oriented Design Metrics as Quality Indicators". In: *IEEE Trans. Softw. Eng.* 22.10 (1996), pp. 751–761.

[29] H. A. Basit, D. C. Rajapakse, and S. Jarzabek. "An Empirical Study on Limits of Clone Unification Using Generics". In: *Proc. Int'l Conf. on Software Engineering and Knowledge Engineering (SEKE '05)*. 2005, pp. 109–114.

[30] P. G. Bassett. *Framing Software Reuse: Lessons from the Real World*. Upper Saddle River, NJ, USA: Prentice-Hall, 1997.

[31] D. Batory. "Feature Models, Grammars, and Propositional Formulas". In: *Proc. Int'l Software Product Line Conf. (SPLC '05)*. Springer, 2005, pp. 7–20.

[32] D. Batory, J. N. Sarvela, and A. Rauschmayer. "Scaling Step-Wise Refinement". In: *IEEE Trans. Softw. Eng.* 30.6 (2004), pp. 355–371.

[33] G. Bavota, R. Oliveto, M. Gethers, D. Poshyvanyk, and A. De Lucia. "Methodbook: Recommending Move Method Refactorings via Relational Topic Models". In: *IEEE Trans. Softw. Eng.* 40.7 (2014), pp. 671–694.

[34] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. "An Empirical Analysis of the Distribution of Unit Test Smells and Their Impact on Software Maintenance". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '12)*. IEEE, 2012, pp. 56–65.

[35] I. D. Baxter and M. Mehlich. "Preprocessor Conditional Removal by Simple Partial Evaluation". In: *Proc. Working Conf. on Reverse Engineering (WCRE '01)*. IEEE, 2001, pp. 281–290.

[36] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. "Clone Detection Using Abstract Syntax Trees". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '98)*. IEEE, 1998, pp. 368–377.

[37] K. Beck and W. Cunningham. *Using Pattern Languages for Object-Oriented Programs*. Tech. rep. CR-87-43. Presented at the OOPSLA'87 Workshop on Specification and Design for Object-Oriented Programming. Tektronix, Inc., Sept. 1987.

[38] C. Berger, H. Rendel, and B. Rumpe. "Measuring the Ability to Form a Product Line from Existing Products". In: *Proc. Int'l Work. on Variability Modeling of Software-Intensive Systems (VaMoS '10)*. University of Duisburg-Essen, Germany, 2010, pp. 151–154.

[39] T. Berger, D. Nair, R. Rublack, J. M. Atlee, K. Czarnecki, and A. Wąsowski. "Three Cases of Feature-Based Variability Modeling in Industry". In: *Proc. Int'l Conf. on Model Driven Engineering Languages and Systems (MODELS '14)*. Springer, 2014, pp. 302–319.

[40] T. Berger, S. She, R. Lotufo, A. Wąsowski, and K. Czarnecki. "Variability Modeling in the Real: A Perspective From the Operating Systems Domain". In: *Proc. IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '10)*. ACM, 2010, pp. 73–82.

[41] J. Bosch and P. Bosch-Sijtsema. "Introducing Agile Customer-Centered Development in a Legacy Software Product Line". In: *Softw.: Pract. Exper.* 41.8 (2011), pp. 871–882.

[42] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. New York, NY, USA: John Wiley & Sons, Ltd., 1998.

[43] J. Businge, M. Openja, S. Nadi, E. Bainomugisha, and T. Berger. "Clone-Based Variability Management in the Android Ecosystem". In: *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME '18)*. IEEE, 2018, pp. 625–634.

[44] F. Calheiros, V. Nepomuceno, P. Borba, S. Soares, and V. Alves. "Product Line Variability Refactoring Tool". In: *Proc. ECOOP Work. on Refactoring Tools (WRT '07)*. Technical University of Berlin, Germany, 2007, pp. 32–33.

[45] A. Carette, M. A. A. Younes, G. Hecht, N. Moha, and R. Rouvoy. "Investigating the Energy Impact of Android Smells". In: *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER '17)*. IEEE, 2017, pp. 115–126.

[46] M. Cartwright and M. Shepperd. "An Empirical Investigation of an Object-Oriented Software System". In: *IEEE Trans. Softw. Eng.* 26.8 (2000), pp. 786–796.

[47] E. Casais. "The Automatic Reorganization of Object Oriented Hierarchies – A Case Study". In: *Object Oriented Systems* 2.1 (1994), pp. 95–115.

[48] W. Chae and M. Blume. "Building a Family of Compilers". In: *Proc. Int'l Software Product Line Conf. (SPLC '08)*. IEEE, 2008, pp. 307–316.

[49] A. Chatzigeorgiou, S. Xanthos, and G. Stephanides. "Evaluating Object-Oriented Designs with Link Analysis". In: *Proc. Int'l Conf. on Software Engineering (ICSE '04)*. IEEE, 2004, pp. 656–665.

[50] *CheckStyle*. 2004. URL: http://checkstyle.sourceforge.net/ (visited on 05/02/2017).

[51] S. R. Chidamber and C. F. Kemerer. "A Metrics Suite for Object Oriented Design". In: *IEEE Trans. Softw. Eng.* 20.6 (1994), pp. 476–493.

[52] P. Clements and C. Krueger. "Point/Counterpoint: Being Proactive Pays Off /Eliminating the Adoption Barrier". In: *IEEE Softw.* 19.4 (2002), pp. 28–31.

[53] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Boston, MA, USA: Addison-Wesley, 2001.

[54] N. Cliff. *Ordinal Methods for Behavioral Data Analysis*. Erlbaum, 1996.

[55] *Code Smell*. Dec. 2014. URL: http://wiki.c2.com/?CodeSmell (visited on 01/06/2020).

[56] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. 2nd ed. Erlbaum, 1988.

[57] M. L. Collard, H. H. Kagdi, and J. I. Maletic. "An XML-Based Lightweight C++ Fact Extractor". In: *Proc. Int'l Work. on Program Comprehension (IWPC '03)*. IEEE, 2003, pp. 134–143.

[58] R. Conradi and B. Westfechtel. "Version Models for Software Configuration Management". In: *ACM Comput. Surv.* 30.2 (June 1998), pp. 232–282.

[59] J. R. Cordy. "Comprehending Reality – Practical Barriers to Industrial Adoption of Software Maintenance Automation". In: *Proc. Int'l Work. on Program Comprehension (IWPC '03)*. IEEE, 2003, pp. 196–205.

[60] M. V. Couto, M. T. Valente, and E. Figueiredo. "Extracting Software Product Lines: A Case Study Using Conditional Compilation". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '11)*. IEEE, 2011, pp. 191–200.

[61] K. Czarnecki and U. W. Eisenecker. *Generative Programming*. Boston, MA, USA: Addison-Wesley, 2000.

[62] M. D'Ambros, A. Bacchelli, and M. Lanza. "On the Impact of Design Flaws on Software Defects". In: *Proc. Int'l Conf. on Quality Software (QSIC '10)*. IEEE, 2010, pp. 23–31.

[63] J. J. Deeks, D. G. Altman, and M. J. Bradburn. "Statistical Methods for Examining Heterogeneity and Combining Results from Several Studies in Meta-Analysis". In: *Systematic Reviews in Health Care: Meta-Analysis in Context*. John Wiley & Sons, Ltd., 2001, pp. 291–299.

[64] I. Deligiannis, M. Shepperd, M. Roumeliotis, and I. Stamelos. "An Empirical Investigation of an Object-Oriented Design Heuristic for Maintainability". In: *J. Syst. Softw.* 65.2 (2003), pp. 127–139.

[65] I. Deligiannis, I. Stamelos, L. Angelis, M. Roumeliotis, and M. Shepperd. "A Controlled Experiment Investigation of an Object-Oriented Design Heuristic for Maintainability". In: *J. Syst. Softw.* 72.2 (2004), pp. 129–143.

[66] A. van Deursen and L. Moonen. "The Video Store Revisited–Thoughts on Refactoring and Testing". In: *Proc. Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering (XP '02)*. 2002, pp. 71–76.

[67] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. "Refactoring Test Code". In: *Proc. Int'l Conf. on eXtreme Programming and Flexible Processes in Software Engineering (XP '01)*. 2001, pp. 92–95.

[68] K. Dhambri, H. Sahraoui, and P. Poulin. "Visual Detection of Design Anomalies". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '08)*. IEEE, 2008, pp. 279–283.

[69] M. Di Penta, L. Cerulo, Y.-G. Guéhéneuc, and G. Antoniol. "An Empirical Study of the Relationships Between Design Pattern Roles and Class Change Proneness". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '08)*. IEEE, 2008, pp. 217–226.

[70] C. Dietrich, R. Tartler, W. Schröder-Preikschat, and D. Lohmann. "A Robust Approach for Variability Extraction From the Linux Build System". In: *Proc. Int'l Software Product Line Conf. (SPLC '12)*. ACM, 2012, pp. 21–30.

[71] D. Dig and R. Johnson. "How Do APIs Evolve? A Story of Refactoring". In: *Softw. Maint. Evol.: Res. Pract.* 18.2 (2006), pp. 83–107.

[72] E. W. Dijkstra. *A Discipline of Programming*. Upper Saddle River, NJ, USA: Prentice-Hall, 1976.

[73] E. W. Dijkstra. "On the Role of Scientific Thought". In: *Selected Writings on Computing: A Personal Perspective*. Springer, 1982, pp. 60–66.

[74] N. Dintzner, A. van Deursen, and M. Pinzger. "FEVER: Feature-Oriented Changes and Artefact Co-Evolution in Highly Configurable Systems". In: *Empir. Softw. Eng.* 23.2 (2018), pp. 90–952.

[75] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk. "Feature Location in Source Code: A Taxonomy and Survey". In: *Softw.: Evol. Proc.* 25.1 (Nov. 2013), pp. 53–95.

[76] A. J. Dobson and A. Barnett. *An Introduction to Generalized Linear Models*. 3rd ed. CRC Press, 2008.

[77] E. Duala-Ekoko and M. P. Robillard. "Clonetracker: Tool Support for Code Clone Management". In: *Proc. Int'l Conf. on Software Engineering (ICSE '08)*. ACM, 2008, pp. 843–846.

[78] E. Duala-Ekoko and M. P. Robillard. "Tracking Code Clones in Evolving Software". In: *Proc. Int'l Conf. on Software Engineering (ICSE '07)*. IEEE, 2007, pp. 158–167.

[79] Y. Dubinsky, J. Rubin, T. Berger, S. Duszynski, M. Becker, and K. Czarnecki. "An Exploratory Study of Cloning in Industrial Software Product Lines". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '13)*. IEEE, 2013, pp. 25–34.

[80] A. N. Duc, A. Mockus, R. Hackbarth, and J. Palframan. "Forking and Coordination in Multi-Platform Development: A Case Study". In: *Proc. 2014 Int'l Symp. on Empirical Software Engineering (ESEM '14)*. ACM, 2014, 59:1–59:10.

[81] S. Ducasse, M. Rieger, and G. Golomingi. "Tool Support for Refactoring Duplicated OO Code". In: *Proc. ECOOP Work. on Experiences in Object-Oriented Re-Engineering*. Forschungszentrum Informatik, Karlsruhe, 1999.

[82] *Eclipse C/C++ Development User Guide > Refactor Menu Actions*. 2007. URL: https://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.cdt.doc.user%2Freference%2Fcdt_u_m_refactor.htm (visited on 11/26/2017).

[83] *Eclipse Java Development User Guide > Refactor Actions*. 2013. URL: http://help.eclipse.org/oxygen/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fref-menu-refactor.htm (visited on 11/26/2017).

[84] S. G. Eick, T. L. Graves, A. F. Karr, J. S. Marron, and A. Mockus. "Does Code Decay? Assessing the Evidence from Change Management Data". In: *IEEE Trans. Softw. Eng.* 27.1 (Jan. 2001), pp. 1–12.

[85] T. Eisenbarth, R. Koschke, and D. Simon. "Locating Features in Source Code". In: *IEEE Trans. Softw. Eng.* 29.3 (2003), pp. 210–224.

[86] K. El Emam. *A Methodology for Validating Software Product Metrics*. Tech. rep. NCR 44142. Ottawa, Ontario, Canada: National Research Council, June 2000.

[87] K. El Emam, S. Benlarbi, N. Goel, and S. N. Rai. "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics". In: *IEEE Trans. Softw. Eng.* 27.7 (2001), pp. 630–650.

[88] E. van Emden and L. Moonen. "Java Quality Assurance by Detecting Code Smells". In: *Proc. Working Conf. on Reverse Engineering (WCRE '02)*. IEEE, 2002, pp. 97–106.

[89] M. D. Ernst, G. J. Badros, and D. Notkin. "An Empirical Analysis of C Preprocessor Use". In: *IEEE Trans. Softw. Eng.* 28.12 (2002), pp. 1146–1170.

[90] *ESLint: The Pluggable Linting Utility for JavaScript and JSX*. 2013. URL: http://eslint.org/ (visited on 05/05/2017).

[91] D. Faust and C. Verhoef. "Software product line migration and deployment". In: *Softw.: Pract. Exper.* 33.10 (2003), pp. 933–955.

[92] J.-M. Favre. "Understanding-in-the-Large". In: *Proc. Int'l Work. on Program Comprehension (IWPC '97)*. IEEE, 1997, pp. 29–38.

[93] J. Feigenspan, C. Kästner, S. Apel, J. Liebig, M. Schulze, R. Dachselt, M. Papendieck, T. Leich, and G. Saake. "Do Background Colors Improve Program Comprehension in the #ifdef Hell?" In: *Empir. Softw. Eng.* 18.4 (Aug. 2013), pp. 699–745. URL: http://link.springer.com/article/10.1007%2Fs10664-012-9208-x.

[94] J. Feigenspan, C. Kästner, J. Liebig, S. Apel, and S. Hanenberg. "Measuring Programming Experience". In: *Proc. Int'l Conf. on Program Comprehension (ICPC '12)*. 2012, pp. 73–82.

[95] W. Fenske, J. Meinicke, S. Schulze, S. Schulze, and G. Saake. "Variant-Preserving Refactorings for Migrating Cloned Products to a Product Line". In: *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER '17)*. IEEE, 2017, pp. 316–326.

[96] W. Fenske and S. Schulze. "Code Smells Revisited: A Variability Perspective". In: *Proc. Int'l Work. on Variability Modeling of Software-Intensive Systems (VaMoS '15)*. ACM, 2015, pp. 3–10.

[97] W. Fenske, S. Schulze, D. Meyer, and G. Saake. "When Code Smells Twice as Much: Metric-Based Detection of Variability-Aware Code Smells". In: *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM '15)*. IEEE, 2015, pp. 171–180.

[98] W. Fenske, S. Schulze, and G. Saake. "How Preprocessor Annotations (Do Not) Affect Maintainability: A Case Study on Change-Proneness". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE '17)*. ACM, 2017, pp. 77–90.

[99] W. Fenske, T. Thüm, and G. Saake. "A Taxonomy of Software Product Line Reengineering". In: *Proc. Int'l Work. on Variability Modeling of Software-Intensive Systems (VaMoS '14)*. ACM, 2014, 4:1–4:8.

[100] J. Ferrante, K. J. Ottenstein, and J. D. Warren. "The Program Dependence Graph and Its Use in Optimization". In: *ACM Trans. Program. Lang. Syst.* 9.3 (1987), pp. 319–349.

[101] G. Ferreira, M. Malik, C. Kästner, J. Pfeffer, and S. Apel. "Do #ifdefs Influence the Occurrence of Vulnerabilities? An Empirical Study of the Linux Kernel". In: *Proc. Int'l Software Product Line Conf. (SPLC '16)*. ACM, 2016, pp. 65–73.

[102] E. Figueiredo, C. Sant'Anna, A. Garcia, and C. Lucena. "Applying and Evaluating Concern-Sensitive Design Heuristics". In: *J. Syst. Softw.* 85.2 (2012), pp. 227–243.

[103] M. Fischer, M. Pinzger, and H. Gall. "Populating a Release History Database from Version Control and Bug Tracking Systems". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '03)*. IEEE, 2003, pp. 23–32.

[104] R. A. Fisher. "On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P". In: *J. Royal Statistical Society* 85.1 (1922), pp. 87–94.

[105] M. Fokaefs, N. Tsantalis, and A. Chatzigeorgiou. "JDeodorant: Identification and Removal of Feature Envy Bad Smells". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '07)*. IEEE, 2007, pp. 519–520.

[106] M. Fowler. *CodeSmell*. Feb. 2006. URL: https://martinfowler.com/bliki/CodeSmell.html (visited on 05/22/2017).

[107] M. Fowler, K. Beck, J. Brant, and W. Opdyke. *Refactoring: Improving the Design of Existing Code*. Boston, MA, USA: Addison-Wesley, 1999.

[108] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley, 1995.

[109] K. Gao and T. M. Khoshgoftaar. "A Comprehensive Empirical Study of Count Models for Software Fault Prediction". In: *IEEE Trans. Reliability* 56.2 (2007), pp. 223–236.

[110] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. "Identifying Architectural Bad Smells". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '09)*. IEEE, 2009, pp. 255–258.

[111] J. Garcia, D. Popescu, G. Edwards, and N. Medvidovic. "Toward a Catalogue of Architectural Bad Smells". In: *Proc. Int'l ACM SIGSOFT Conf. on Quality of Software Architectures (QoSA '09)*. Springer, 2009, pp. 146–162.

[112] A. Garrido. "Program Refactoring in the Presence of Preprocessor Directives". PhD thesis. Champaign, IL, USA: University of Illinois, 2005.

[113] A. Garrido and R. Johnson. "Analyzing Multiple Configurations of a C Program". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '05)*. IEEE, 2005, pp. 379–388.

[114] A. Garrido and R. Johnson. "Embracing the C Preprocessor During Refactoring". In: *Softw.: Evol. Proc.* 25.12 (2013), pp. 1285–1304.

[115] A. Garrido and R. Johnson. "Refactoring C with Conditional Compilation". In: *Proc. IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '03)*. IEEE, 2003, pp. 323–326.

[116] Y. Ghanam and F. Maurer. "Extreme Product Line Engineering – Refactoring for Variability: A Test-Driven Approach". In: *Proc. Int'l Conf. on Agile Processes in Software Engineering and Extreme Programming (XP '10)*. Springer, 2010, pp. 43–57.

[117] M. L. Griss. "Implementing Product-Line Features by Composing Aspects". In: *Proc. Int'l Software Product Line Conf. (SPLC 1)*. Kluwer Academic Publishers, 2000, pp. 271–288.

[118] R. J. Grissom and J. J. Kim. *Effect Sizes for Research: A Broad Practical Approach*. Erlbaum, 2005.

[119] W. G. Griswold. "Program Restructuring as an Aid to Software Maintenance". PhD thesis. Seattle, WA, USA: University of Washington, 1991.

[120] D. Gupta, P. Jalote, and G. Barua. "A Formal Framework for On-Line Software Version Change". In: *IEEE Trans. Softw. Eng.* 22.2 (1996), pp. 120–131.

[121] J. van Gurp and C. Prehofer. "Version Management Tools as a Basis for Integrating Product Derivation and Software Product Families". In: *Proc. Int'l Work. on Variability Management – Working with Variability Mechanisms*. Fraunhofer IESE, 2006, pp. 48–58.

[122] S. Habchi, G. Hecht, R. Rouvoy, and N. Moha. "Code Smells in iOS Apps: How do They Compare to Android?" In: *Proc. Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE, 2017, pp. 110–121.

[123] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering". In: *IEEE Trans. Softw. Eng.* 38.6 (2012), pp. 1276–1304.

[124] T. Hall, M. Zhang, D. Bowes, and Y. Sun. "Some Code Smells Have a Significant but Small Effect on Faults". In: *ACM Trans. Softw. Eng. Methodol.* 23.4 (Sept. 2014), 33:1–33:39.

[125] G. Hecht, N. Moha, and R. Rouvoy. "An Empirical Study of the Performance Impacts of Android Code Smells". In: *Proc. Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft '16)*. ACM, 2016, pp. 59–69.

[126] G. Hecht, R. Rouvoy, N. Moha, and L. Duchien. "Detecting Antipatterns in Android Apps". In: *Proc. Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft '15)*. IEEE, 2015, pp. 148–149.

[127] J. Henkel and A. Diwan. "CatchUp! Capturing and Replaying Refactorings to Support API Evolution". In: *Proc. Int'l Conf. on Software Engineering (ICSE '05)*. ACM, 2005, pp. 274–283.

[128] F. Hermans and D. Dig. "Bumblebee: A Refactoring Environment for Spreadsheet Formulas". In: *Proc. Int'l Symp. on the Foundations of Software Engineering (FSE '14)*. ACM, 2014, pp. 747–750.

[129] F. Hermans, M. Pinzger, and A. van Deursen. "Detecting and Refactoring Code Smells in Spreadsheet Formulas". In: *Empir. Softw. Eng.* 20.2 (2015), pp. 549–575.

[130] F. Hermans, M. Pinzger, and A. van Deursen. "Detecting and Visualizing Inter-Worksheet Smells in Spreadsheets". In: *Proc. Int'l Conf. on Software Engineering (ICSE '12)*. IEEE, 2012, pp. 441–451.

[131] F. Hermans, M. Pinzger, and A. van Deursen. "Detecting Code Smells in Spreadsheet Formulas". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '12)*. IEEE, 2012, pp. 409–418.

[132] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. "ARIES: Refactoring Support Environment Based on Code Clone Analysis". In: *Proc. IASTED Conf. on Software Engineering and Applications (SEA '04)*. ACTA Press, 2004, pp. 222–229.

[133] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. "ARIES: Refactoring Support Tool for Code Clone". In: *Proc. Work. on Software Quality (WoSQ '05)*. ACM, 2005, pp. 1–4.

[134] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. "Method and Implementation for Investigating Code Clones in a Software System". In: *Inform. Software Tech.* 49.9-10 (2007), pp. 985–998.

[135] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. "Refactoring Support Based on Code Clone Analysis". In: *Proc. Int'l Conf. on Product Focused Software Process Improvement (PROFES '04)*. Vol. 3009. Springer, 2004, pp. 220–233.

[136] Y. Higo, S. Kusumoto, and K. Inoue. "A Metric-Based Approach to Identifying Refactoring Opportunities for Merging Code Clones in a Java Software System". In: *Softw. Maint. Evol.: Res. Pract.* 20.6 (Nov. 2008), pp. 435–461.

[137] J. M. Hilbe. *Negative Binomial Regression*. 2nd ed. Cambridge University Press, 2011.

[138] D. Hou, P. Jablonski, and F. Jacob. "CnP: Towards an Environment for the Proactive Management of Copy-and-Paste Programming". In: *Proc. Int'l Conf. on Program Comprehension (ICPC '09)*. IEEE, 2009, pp. 238–242.

[139] D. Hovemeyer and W. Pugh. "Finding Bugs is Easy". In: *ACM SIGPLAN Not.* 39.12 (Dec. 2004), pp. 92–106.

[140] M. Hozano, A. Garcia, B. Fonseca, and E. Costa. "Are You Smelling It? Investigating how Similar Developers Detect Code Smells". In: *Information and Software Technology* (2017).

[141] E. Hull, K. Jackson, and J. Dick. *Requirements Engineering*. 4th. Springer, 2017.

[142] C. Hunsen, B. Zhang, J. Siegmund, C. Kästner, O. Leßenich, M. Becker, and S. Apel. "Preprocessor-Based Variability in Open-Source and Industrial Software Systems: An Empirical Study". In: *Empir. Softw. Eng.* 21.2 (2016), pp. 449–482.

[143] F. Jaafar, Y.-G. Guéhéneuc, S. Hamel, and F. Khomh. "Mining the Relationship Between Anti-Patterns Dependencies and Fault-Proneness". In: *Proc. Working Conf. on Reverse Engineering (WCRE '13)*. IEEE, 2013, pp. 351–360.

[144] S. Jarzabek, P. Bassett, H. Zhang, and W. Zhang. "XVCL: XML-Based Variant Configuration Language". In: *Proc. Int'l Conf. on Software Engineering (ICSE '03)*. Portland, Oregon: IEEE, 2003, pp. 810–811.

[145] J. H. Johnson. "Identifying Redundancy in Source Code Using Fingerprints". In: *Proc. Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON '93)*. IBM Press, 1993, pp. 171–183.

[146] S. C. Johnson. *Lint, a C Program Checker*. Tech. rep. Computer Science Technical Report 65. Murray Hill, USA: Bell Telephone Laboratories, 1977.

[147] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. "Do Code Clones Matter?" In: *Proc. Int'l Conf. on Software Engineering (ICSE '09)*. IEEE, 2009, pp. 485–495.

[148] N. Juillerat and B. Hirsbrunner. "An Algorithm for Detecting and Removing Clones in Java Code". In: *Proc. Work. on Software Evolution through Transformations: Embracing the Change (SeTra '06)*. EASST, 2006, pp. 63–74.

[149] T. Kamiya, S. Kusumoto, and K. Inoue. "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code". In: *IEEE Trans. Softw. Eng.* 28.7 (2002), pp. 654–670.

[150] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson. *Feature-Oriented Domain Analysis (FODA) Feasibility Study*. Tech. rep. CMU/SEI-90-TR-21. Pittsburgh, PA, USA: SEI, 1990.

[151] K. C. Kang, M. Kim, J. Lee, and B. Kim. "Feature-Oriented Re-Engineering of Legacy Systems into Product Line Assets – A Case Study". In: *Proc. Int'l Software Product Line Conf. (SPLC '05)*. Springer, 2005, pp. 45–56.

[152] C. Kapser and M. W. Godfrey. ""Cloning Considered Harmful" Considered Harmful: Patterns of Cloning in Software". In: *Empir. Softw. Eng.* 13.6 (2008), pp. 645–692.

[153] C. Kästner. "CIDE: Decomposing Legacy Applications into Features". In: *Proc. Int'l Software Product Line Conf. (SPLC '07) Second Volume*. Kyoto, Japan: IEEE, 2007, pp. 149–150. ISBN: 978-4-7649-0342-5. URL: http://wwwiti.cs.uni-magdeburg.de/~ckaestne/splc07demo.pdf.

[154] C. Kästner and S. Apel. "Type-Checking Software Product Lines – A Formal Approach". In: *Proc. IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '08)*. IEEE, 2008, pp. 258–267.

[155] C. Kästner and S. Apel. "Virtual Separation of Concerns – A Second Chance for Preprocessors". In: *J. Object Technol.* 8.6 (2009), pp. 59–78.

[156] C. Kästner, S. Apel, and D. Batory. "A Case Study Implementing Features Using AspectJ". In: *Proc. Int'l Software Product Line Conf. (SPLC '07)*. IEEE, 2007, pp. 223–232.

[157] C. Kästner, S. Apel, and M. Kuhlemann. "A Model of Refactoring Physically and Virtually Separated Features". In: *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE '09)*. ACM, 2009, pp. 157–166.

[158] C. Kästner, S. Apel, and M. Kuhlemann. "Granularity in Software Product Lines". In: *Proc. Int'l Conf. on Software Engineering (ICSE '08)*. ACM, 2008, pp. 311–320.

[159] 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: *Proc. Int'l Software Product Line Conf. (SPLC '09)*. SEI, 2009, pp. 181–190.

[160] C. Kästner, A. Dreiling, and K. Ostermann. "Variability Mining: Consistent Semi-Automatic Detection of Product-Line Features". In: *IEEE Trans. Softw. Eng.* 40.1 (Jan. 2014), pp. 67–82.

[161] C. Kästner, P. G. Giarrusso, T. Rendel, S. Erdweg, K. Ostermann, and T. Berger. "Variability-Aware Parsing in the Presence of Lexical Macros and Conditional Compilation". In: *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '11)*. ACM, 2011, pp. 805–824.

[162] C. Kästner, M. Kuhlemann, and D. Batory. "Automating Feature-Oriented Refactoring of Legacy Applications". In: *Proc. ECOOP Work. on Refactoring Tools (WRT '07)*. Technical University of Berlin, Germany, 2007, pp. 62–63.

[163] Y. Kataoka, D. Notkin, M. D. Ernst, and W. G. Griswold. "Automated Support for Program Refactoring Using Invariants". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '01)*. IEEE, 2001, p. 736.

[164] A. Kenner, C. Kästner, S. Haase, and T. Leich. "TypeChef: Toward Type Checking #ifdef Variability in C". In: *Proc. Int'l Work. on Feature-Oriented Software Development (FOSD '10)*. ACM, 2010, pp. 25–32.

[165] J. Kerievsky. *Refactoring to Patterns*. Boston, MA, USA: Addison-Wesley, 2004.

[166] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Upper Saddle River, NJ, USA: Prentice-Hall, 1978.

[167] M. Kessentini and A. Ouni. "Detecting Android Smells Using Multi-Objective Genetic Programming". In: *Proc. Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft '17)*. IEEE, 2017, pp. 122–132.

[168] M. Kessentini, S. Vaucher, and H. Sahraoui. "Deviance from Perfection Is a Better Criterion than Closeness to Evil when Identifying Risky Code". In: *Proc. IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '10)*. ACM, 2010, pp. 113–122.

[169] F. Khomh, M. Di Penta, and Y.-G. Guéhéneuc. "An Exploratory Study of the Impact of Code Smells on Software Change-Proneness". In: *Proc. Working Conf. on Reverse Engineering (WCRE '09)*. IEEE, 2009, pp. 75–84.

[170] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol. "An Exploratory Study of the Impact of Antipatterns on Class Change- and Fault-Proneness". In: *Empir. Softw. Eng.* 17.3 (2012), pp. 243–275.

[171] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. "BDTEX: A GQM-Based Bayesian Approach for the Detection of Antipatterns". In: *J. Syst. Softw.* 84.4 (2011), pp. 559–572.

[172] F. Khomh, S. Vaucher, Y.-G. Guéhéneuc, and H. Sahraoui. "A Bayesian Approach for the Detection of Code and Design Smells". In: *Proc. Int'l Conf. on Quality Software (QSIC '09)*. IEEE, 2009, pp. 305–314.

[173] C. Kiefer, A. Bernstein, and J. Tappolet. "Mining Software Repositories with iSPARQL and a Software Evolution Ontology". In: *Proc. Int'l Work. on Mining Software Repositories (MSR '07)*. IEEE, 2007, pp. 10–18.

[174] J. Kim, D. Batory, and D. Dig. "Refactoring Java Software Product Lines". In: *Proc. Int'l Software Product Line Conf. (SPLC '17)*. ACM, 2017, pp. 59–68.

[175] J. Kim, D. Batory, and D. Dig. "X15: A Tool for Refactoring Java Software Product Lines". In: *Proc. Int'l Software Product Line Conf. (SPLC '17)*. ACM, 2017, pp. 28–31.

[176] M. Kim, D. Cai, and S. Kim. "An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution". In: *Proc. Int'l Conf. on Software Engineering (ICSE '11)*. ACM, 2011, pp. 151–160.

[177] M. Kim, V. Sazawal, D. Notkin, and G. Murphy. "An Empirical Study of Code Clone Genealogies". In: *Proc. Int'l Symp. on the Foundations of Software Engineering (FSE '05)*. ACM, 2005, pp. 187–196.

[178] M. Kim, J. Lee, K. C. Kang, Y. Hong, and S. Bang. "Re-Engineering Software Architecture of Home Service Robots: A Case Study". In: *Proc. Int'l Conf. on Software Engineering (ICSE '05)*. St. Louis, MO, USA: ACM, 2005, pp. 505–513.

[179] B. Klatt, K. Krogmann, and C. Seidl. "Program Dependency Analysis for Consolidating Customized Product Copies". In: *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME '14)*. IEEE, 2014, pp. 496–500.

[180] K. Kontogiannis, R. DeMori, M. Bernstein, M. Galler, and E. Merlo. "Pattern Matching for Design Concept Localization". In: *Proc. Working Conf. on Reverse Engineering (WCRE '95)*. IEEE, 1995, pp. 96–103.

[181] R. Koschke. "Identifying and Removing Software Clones". In: *Software Evolution*. Springer, 2008, pp. 15–36.

[182] R. Koschke. "Large-Scale Inter-System Clone Detection Using Suffix Trees and Hashing". In: *Softw.: Evol. Proc.* 26.8 (2014), pp. 747–769.

[183] R. Koschke, P. Frenzel, A. P. J. Breu, and K. Angstmann. "Extending the Reflexion Method For Consolidating Software Variants Into Product Lines". In: *Softw. Qual. J.* 17.4 (2009), pp. 331–366.

[184] J. Kreimer. "Adaptive Detection of Design Flaws". In: *Electronic Notes in Theor. Comput. Sci.* 141.4 (Dec. 2005), pp. 117–136.

[185] C. W. Krueger. "Easing the Transition to Software Mass Customization". In: *Proc. Int'l Work. on Software Product-Family Engineering (PFE '01), Revised Papers*. Springer, 2002, pp. 282–293.

[186] C. W. Krueger. "Towards a Taxonomy for Software Product Lines". In: *Proc. Int'l Work. on Software Product-Family Engineering (PFE '03), Revised Papers*. Vol. 3014. Springer, 2004, pp. 323–331.

[187] J. Krüger, W. Fenske, J. Meinicke, T. Leich, and G. Saake. "Extracting Software Product Lines: A Cost Estimation Perspective". In: *Proc. Int'l Software Product Line Conf. (SPLC '16)*. ACM, 2016, pp. 354–361.

[188] M. Kuhlemann. "Refactoring Feature Modules: Disciplined Generation of Reusable Modules". Dissertation. Germany: University of Magdeburg, 2011.

[189] M. Kuhlemann, D. Batory, and S. Apel. *Refactoring Feature Modules*. Tech. rep. 15. School of Computer Science, University of Magdeburg, 2008.

[190] M. Kuhlemann, D. Batory, and S. Apel. "Refactoring Feature Modules". In: *Proc. Int'l Conf. on Software Reuse (ICSR '09)*. Springer, 2009, pp. 106–115.

[191] M. Kuhlemann, M. Rosenmüller, S. Apel, and T. Leich. "On the Duality of Aspect-Oriented and Feature-Oriented Design Patterns". In: *Proc. AOSD Work. on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS '07)*. ACM, 2007. URL: http://doi.acm.org/10.1145/1233901.1233906.

[192] M. A. Laguna and Y. Crespo. "A Systematic Mapping Study on Software Product Line Evolution: From Legacy System Reengineering to Product Line Refactoring". In: *Sci. Comput. Prog.* 78.8 (2013), pp. 1010–1034.

[193] A. Lakhotia and J.-/. Deprez. "Restructuring Programs by Tucking Statements into Functions". In: *Inform. Software Tech., Special Issue on Program Slicing* 40.11 (1998), pp. 677–689.

[194] F. Lanubile and G. Visaggio. "Extracting Reusable Functions by Flow Graph Based Program Slicing". In: *IEEE Trans. Softw. Eng.* 23.4 (1997), pp. 246–259.

[195] M. Lanza and S. Ducasse. "Polymetric Views—A Lightweight Visual Approach to Reverse Engineering". In: *IEEE Trans. Softw. Eng.* 29.9 (2003), pp. 782–795.

[196] M. Lanza and R. Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Berlin Heidelberg, Germany: Springer, 2006.

[197] T. D. LaToza, G. Venolia, and R. DeLine. "Maintaining Mental Models: A Study of Developer Work Habits". In: *Proc. Int'l Conf. on Software Engineering (ICSE '06)*. ACM, 2006, pp. 492–501.

[198] D. Le, E. Walkingshaw, and M. Erwig. "#ifdef Confirmed Harmful: Promoting Understandable Software Variation". In: *Proc. IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC '11)*. IEEE, 2011, pp. 143–150.

[199] S. Lee, G. Bae, H. S. Chae, D.-H. Bae, and Y. R. Kwon. "Automated Scheduling for Clone-Based Refactoring Using a Competent GA". In: *Softw.: Pract. Exper.* 41.5 (2011), pp. 521–550.

[200] M. Lejter, S. Meyers, and S. P. Reiss. "Support for Maintaining Object-Oriented Programs". In: *IEEE Trans. Softw. Eng.* 18.12 (1992), pp. 1045–1052.

[201] O. Leßenich, J. Siegmund, S. Apel, C. Kästner, and C. Hunsen. "Indicators for Merge Conflicts in the Wild: Survey and Empirical Study". In: *Autom. Softw. Eng.* 25.2 (2018), pp. 279–313.

[202] W. Li and R. Shatnawi. "An Empirical Study of the Bad Smells and Class Error Probability in the Post-Release Object-Oriented System Evolution". In: *J. Syst. Softw.* 80.7 (2007), pp. 1120–1128.

[203] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. "CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code". In: *IEEE Trans. Softw. Eng.* 32.3 (2006), pp. 176–192.

[204] J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines". In: *Proc. Int'l Conf. on Software Engineering (ICSE '10)*. Cape Town, South Africa: ACM, 2010, pp. 105–114.

[205] J. Liebig, A. Janker, F. Garbe, S. Apel, and C. Lengauer. "Morpheus: Variability-Aware Refactoring in the Wild". In: *Proc. Int'l Conf. on Software Engineering (ICSE '15)*. ACM, 2015, pp. 380–391.

[206] J. Liebig, C. Kästner, and S. Apel. "Analyzing the Discipline of Preprocessor Annotations in 30 Million Lines of C Code". In: *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD '11)*. ACM, 2011, pp. 191–202.

[207] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis. "Identification of Refused Bequest Code Smells". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '13)*. IEEE, 2013, pp. 392–395.

[208] M. Lindvall. "Are Large C++ Classes Change-Prone? An Empirical Investigation". In: *Softw.: Pract. Exper.* 28.15 (1998), pp. 1551–1558.

[209] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed. "Variability Extraction and Modeling for Product Variants". In: *Softw. & Syst. Modeling* 16.4 (2017), pp. 1179–1199.

[210] *Linux Kernel Coding Style*. 2016. URL: https://www.kernel.org/doc/html/v4.12/process/coding-style.html (visited on 07/07/2017).

[211] J. Liu, D. Batory, and C. Lengauer. "Feature Oriented Refactoring of Legacy Applications". In: *Proc. Int'l Conf. on Software Engineering (ICSE '06)*. ACM, 2006, pp. 112–121.

[212] S. Livieri, Y. Higo, M. Matushita, and K. Inoue. "Very-Large Scale Code Clone Analysis and Visualization of Open Source Programs Using Distributed CCFinder: D-CCFinder". In: *Proc. Int'l Conf. on Software Engineering (ICSE '07)*. IEEE, 2007, pp. 106–115.

[213] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. "A Quantitative Analysis of Aspects in the eCos Kernel". In: *Proc. ACM SIGOPS/EuroSys European Conf. on Computer Systems (EUROSYS '11)*. ACM, 2006, pp. 191–204.

[214] R. E. Lopez-Herrejon and D. Batory. "A Standard Problem for Evaluating Product-Line Methodologies". In: *Proc. Int'l Conf. on Generative and Component-Based Software Engineering (GCSE '01)*. Ed. by J. Bosch. Vol. 2186. Lecture Notes in Computer Science. Springer, 2001, pp. 10–24.

[215] R. E. Lopez-Herrejon, L. Montalvillo-Mendizabal, and A. Egyed. "From Requirements to Features: An Exploratory Study of Feature-Oriented Refactoring". In: *Proc. Int'l Software Product Line Conf. (SPLC '11)*. IEEE, 2011, pp. 181–190.

[216] A. Lozano and M. Wermelinger. "Assessing the Effect of Clones on Changeability". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '08)*. IEEE, 2008, pp. 227–236.

[217] A. Lozano, M. Wermelinger, and B. Nuseibeh. "Evaluating the Harmfulness of Cloning: A Change Based Experiment". In: *Proc. Int'l Work. on Mining Software Repositories (MSR '07)*. IEEE, 2007, p. 18.

[218] W. Ma, L. Chen, Y. Zhou, and B. Xu. "Do We Have a Chance to Fix Bugs When Refactoring Code Smells?" In: *Proc. Int'l Conf. on Software Analysis, Testing and Evolution (SATE '16)*. IEEE, 2016, pp. 24–29.

[219] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, and E. Aïmeur. "SMURF: A SVM-Based Incremental Anti-Pattern Detection Approach". In: *Proc. Working Conf. on Reverse Engineering (WCRE '12)*. IEEE, 2012, pp. 466–475.

[220] A. Maiga, N. Ali, N. Bhattacharya, A. Sabané, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur. "Support Vector Machines for Anti-Pattern Detection". In: *Proc. IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '12)*. IEEE, 2012, pp. 278–281.

[221] R. Malaquias, M. Ribeiro, R. Bonifácio, E. Monteiro, F. Medeiros, A. Garcia, and R. Gheyi. "The Discipline of Preprocessor-Based Annotations Does #ifdef TAG n't #endif Matter". In: *Proc. Int'l Conf. on Program Comprehension (ICPC '17)*. IEEE, 2017, pp. 297–307.

[222] U. A. Mannan, I. Ahmed, R. A. M. Almurshed, D. Dig, and C. Jensen. "Understanding Code Smells in Android Applications". In: *Proc. Int'l Conf. on Mobile Software Engineering and Systems (MOBILESoft '16)*. IEEE, 2016, pp. 225–236.

[223] M. V. Mäntylä and C. Lassenius. "Subjective Evaluation of Software Evolvability Using Code Smells: An Empirical Study". In: *Empir. Softw. Eng.* 11.3 (2006), pp. 395–431.

[224] M. V. Mäntylä, J. Vanhanen, and C. Lassenius. "Bad Smells—Humans as Code Critics". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '04)*. IEEE, 2004, pp. 399–408.

[225] C. Marinescu. "Identification of Design Roles for the Assessment of Design Quality in Enterprise Applications". In: *Proc. Int'l Conf. on Program Comprehension (ICPC '06)*. IEEE, 2006, pp. 169–180.

[226] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. "iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '05) (Industrial and Tool Volume)*. IEEE, 2005.

[227] R. Marinescu. "Detecting Design Flaws via Metrics in Object-Oriented Systems". In: *Proc. Int'l Conf. and Exibition on Technology of Object-Oriented Languages and Systems (TOOLS '01)*. IEEE, 2001, pp. 173–182.

[228] R. Marinescu. "Detection Strategies: Metrics-Based Rules for Detecting Design Flaws". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '04)*. IEEE, 2004, pp. 350–359.

[229] R. Marinescu. "Measurement and Quality in Object-Oriented Design". PhD thesis. Timişoara, Romania: Polytehnica University of Timişoara, 2002.

[230] R. Marinescu. "Measurement and Quality in Object-Oriented Design". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '05)*. IEEE, 2005, pp. 701–704.

[231] R. Marinescu and C. Marinescu. "Are the Clients of Flawed Classes (Also) Defect Prone?" In: *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM '11)*. IEEE, 2011, pp. 65–74.

[232] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ, USA: Prentice-Hall, 2009.

[233] R. C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice-Hall, 2003.

[234] J. Martinez and A. K. Thurimella. "Collaboration and Source Code Driven Bottom-up Product Line Engineering". In: *Proc. Int'l Software Product Line Conf. (SPLC '12) (Volume 2)*. ACM, 2012, pp. 196–200.

[235] J. Martinez, T. Ziadi, T. F. Bissyandé, J. Klein, and Y. Le Traon. "Bottom-up Adoption of Software Product Lines: A Generic and Extensible Approach". In: *Proc. Int'l Software Product Line Conf. (SPLC '15)*. ACM, 2015, pp. 101–110.

[236] J. Mayrand, C. Leblanc, and E. Merlo. "Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '96)*. Vol. 96. IEEE, 1996, pp. 244–253.

[237] D. McFadden. *Quantitative Methods for Analyzing Travel Behavior of Individuals: Some Recent Developments*. Institute of Transportation Studies, University of California, 1977.

[238] F. Medeiros, C. Kästner, M. Ribeiro, S. Nadi, and R. Gheyi. "The Love/Hate Relationship with the C Preprocessor: An Interview Study". In: *Proc. European Conf. on Object-Oriented Programming (ECOOP '15)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015, pp. 495–518.

[239] F. Medeiros, M. Ribeiro, and R. Gheyi. "Investigating Preprocessor-Based Syntax Errors". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE '13)*. ACM, 2013, pp. 75–84.

[240] F. Medeiros, M. Ribeiro, R. Gheyi, S. Apel, C. Kästner, B. Ferreira, L. Carvalho, and B. Fonseca. "Discipline Matters: Refactoring of Preprocessor Directives in the #ifdef Hell". In: *IEEE Trans. Softw. Eng.* 44.5 (2018), pp. 453–469.

[241] F. Medeiros, I. Rodrigues, M. Ribeiro, L. Teixeira, and R. Gheyi. "An Empirical Study on Configuration-Related Issues: Investigating Undeclared and Unused Identifiers". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE '15)*. ACM, 2015, pp. 35–44.

[242] J. Meinicke, T. Thüm, R. Schröter, S. Krieter, F. Benduhn, G. Saake, and T. Leich. "FeatureIDE: Taming the Preprocessor Wilderness". In: *Proc. Int'l Conf. on Software Engineering (ICSE '16)*. ACM, 2016, p. 4.

[243] J. Melo, C. Brabrand, and A. Wąsowski. "How Does the Degree of Variability Affect Bug Finding?" In: *Proc. Int'l Conf. on Software Engineering (ICSE '16)*. ACM, 2016, pp. 679–690.

[244] 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. European Conf. on Software Maintenance and Reengineering (CSMR '08)*. IEEE, 2008, pp. 163–172.

[245] T. Mende, R. Koschke, and F. Beckwermert. "An Evaluation of Code Similarity Identification for the Grow-and-Prune Model". In: *Softw. Maint. Evol.: Res. Pract.* 21.2 (2009), pp. 143–169.

[246] T. Mens. "A Formal Foundation for Object-Oriented Software Evolution". PhD thesis. Belgium: Department of Computer Science, Vrije Universiteit Brussel, Sept. 1999.

[247] T. Mens. "A State-of-the-Art Survey on Software Merging". In: *IEEE Trans. Softw. Eng.* 28.5 (2002), pp. 449–462.

[248] T. Mens, S. Demeyer, and D. Janssens. "Formalising Behaviour Preserving Program Transformations". In: *Proc. Int'l Conf. on Graph Transformation (ICGT '02)*. Springer, 2002, pp. 286–301.

[249] T. Mens and T. Tourwé. "A Survey of Software Refactoring". In: *IEEE Trans. Softw. Eng.* 30.2 (Feb. 2004), pp. 126–139.

[250] T. Mens, T. Tourwé, and F. Muñoz. "Beyond the Refactoring Browser: Advanced Tool Support for Software Refactoring". In: *Proc. Int'l Work. on Principles of Software Evolution (IWPSE '03)*. IEEE, 2003, pp. 39–44.

[251] T. Mens, N. Van Eetvelde, S. Demeyer, and D. Janssens. "Formalizing Refactorings with Graph Transformations". In: *Softw.: Evol. Proc.* 17.4 (2005), pp. 247–276.

[252] P. F. Mihancea and R. Marinescu. "Towards the Optimization of Automatic Detection of Design Flaws in Object-Oriented Software Systems". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '05)*. IEEE, 2005, pp. 92–101.

[253] R. C. Miller and B. A. Myers. "Interactive Simultaneous Editing of Multiple Text Regions". In: *Proc. Proc. USENIX Conf.* USENIX Association, 2001, pp. 161–174.

[254] R. Mo, Y. Cai, R. Kazman, and L. Xiao. "Hotspot Patterns: The Formal Definition and Automatic Detection of Architecture Smells". In: *Proc. Working Conf. on Software Architecture (WICSA '15)*. IEEE, 2015, pp. 51–60.

[255] A. Mockus and L. G. Votta. "Identifying Reasons for Software Changes Using Historic Databases". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '00)*. IEEE, 2000, pp. 120–130.

[256] N. Moha. "Detection and Correction of Design Defects in Object-Oriented Designs". In: *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '07)*. ACM, 2007, pp. 949–950.

[257] N. Moha, Y.-G. Guéhéneuc, L. Duchien, and A.-F. Le Meur. "DECOR: A Method for the Specification and Detection of Code and Design Smells". In: *IEEE Trans. Softw. Eng.* 36.1 (2010), pp. 20–36.

[258] N. Moha, Y.-G. Guéhéneuc, A.-F. Le Meur, and L. Duchien. "A Domain Analysis to Specify Design Defects and Generate Detection Algorithms". In: *Proc. Int'l Conf. on Fundamental Approaches to Software Engineering (FASE '08)*. Springer, 2008, pp. 276–291.

[259] A. Monden, D. Nakae, T. Kamiya, S.-i. Sato, and K.-i. Matsumoto. "Software Quality Analysis by Code Clones in Industrial Legacy Software". In: *Proc. IEEE Int'l Software Metrics Symposium (METRICS '02)*. IEEE, 2002, pp. 87–94.

[260] M. P. Monteiro and J. M. Fernandes. "Towards a Catalogue of Refactorings and Code Smells for AspectJ". In: *Trans. Aspect-Oriented Softw. Development I*. Springer, 2006, pp. 212–258.

[261] I. Moore. "Automatic Inheritance Hierarchy Restructuring and Method Refactoring". In: *Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*. ACM, 1996, pp. 235–250.

[262] R. Morales, R. Saborido, F. Khomh, F. Chicano, and G. Antoniol. "EARMO: An Energy-Aware Refactoring Approach for Mobile Apps". In: *IEEE Trans. Softw. Eng.* X.X (2017). to appear, pp. 1–31.

[263] E. R. Murphy-Hill, P. J. Quitslund, and A. P. Black. "Removing Duplication From java.io: A Case Study Using Traits". In: *Companion to the Proc. ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, 2005, pp. 282–291.

[264] E. Murphy-Hill and A. P. Black. "An Interactive Ambient Visualization for Code Smells". In: *Proc. ACM Symp. on Software Visualization (SoftVis '10)*. ACM, 2010, pp. 5–14.

[265] E. Murphy-Hill and A. P. Black. "Seven Habits of a Highly Effective Smell Detector". In: *Proc. Int'l Work. on Recommendation Systems for Software Engineering (RSSE '08)*. ACM, 2008, pp. 36–40.

[266] E. Murphy-Hill, C. Parnin, and A. P. Black. "How We Refactor, and How We Know It". In: *IEEE Trans. Softw. Eng.* 38.1 (2012), pp. 5–18.

[267] L. Neves, P. Borba, V. Alves, L. Turnes, L. Teixeira, D. Sena, and U. Kulesza. "Safe Evolution Templates for Software Product Lines". In: *J. Syst. Softw.* 106 (2015), pp. 42–58.

[268] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. "Clone Management for Evolving Software". In: *IEEE Trans. Softw. Eng.* 38.5 (2012), pp. 1008–1026.

[269] *npm-coding-style*. 2016. URL: https://docs.npmjs.com/misc/coding-style (visited on 05/31/2017).

[270] L. Nyman and T. Mikkonen. "To Fork or not to Fork: Fork Motivations in SourceForge Projects". In: *Int. J. Open Source Softw. Proc.* 3.3 (2011), pp. 1–9.

[271] M. Ó Cinnéide, A. Yamashita, and S. Counsell. "Measuring Refactoring Benefits: A Survey of the Evidence". In: *Proc. Int'l Work. on Software Refactoring (IWoR '16)*. ACM, 2016, pp. 9–12.

[272] W. Oizumi, A. Garcia, L. da Silva Sousa, B. Cafeo, and Y. Zhao. "Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems". In: *Proc. Int'l Conf. on Software Engineering (ICSE '16)*. IEEE, 2016, pp. 440–451.

[273] S. M. Olbrich, D. S. Cruzes, and D. I. Sjøberg. "Are All Code Smells Harmful? A Study of God Classes and Brain Classes in the Evolution of Three Open Source Systems". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '10)*. IEEE, 2010, pp. 1–10.

[274] S. Olbrich, D. S. Cruzes, V. Basili, and N. Zazworka. "The Evolution and Impact of Code Smells: A Case Study of Two Open Source Systems". In: *Proc. 2009 Int'l Symp. on Empirical Software Engineering (ESEM '09)*. IEEE, 2009, pp. 390–400.

[275] R. Oliveira, B. Estácio, A. Garcia, S. Marczak, R. Prikladnicki, M. Kalinowski, and C. Lucena. "Identifying Code Smells with Collaborative Practices: A Controlled Experiment". In: *Proc. 2016 Tenth Brazilian Symp. on Software Components, Architectures and Reuse (SBCARS '16)*. IEEE, 2016, pp. 61–70.

[276] R. Oliveira, L. Sousa, R. de Mello, N. Valentim, A. Lopes, T. Conte, A. Garcia, E. Oliveira, and C. Lucena. "Collaborative Identification of Code Smells: A Multi-Case Study". In: *Proc. Int'l Conf. on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP '17)*. IEEE, 2017, pp. 33–42.

[277] A. Olszak and B. N. Jørgensen. "Remodularizing Java Programs for Comprehension of Features". In: *Proc. Int'l Work. on Feature-Oriented Software Development (FOSD '09)*. ACM, 2009, pp. 19–26.

[278] W. F. Opdyke. "Refactoring Object-Oriented Frameworks". PhD thesis. Champaign, IL, USA: University of Illinois, 1992.

[279] *org.jsesoft.jpp: A Java Macro Preprocessor*. 2002. URL: http://jsesoft.sourceforge.net/ (visited on 10/15/2018).

[280] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. "Predicting the Location and Number of Faults in Large Software Systems". In: *IEEE Trans. Softw. Eng.* 31.4 (2005), pp. 340–355.

[281] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. "Detecting Bad Smells in Source Code Using Change History Information". In: *Proc. IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '13)*. IEEE, 2013, pp. 268–278.

[282] F. Palomba, G. Bavota, R. Oliveto, and A. De Lucia. "Do They Really Smell Bad? A Study on Developers' Perception of Code Bad Smells". In: *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME '14)*. IEEE, 2014, pp. 101–110.

[283] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia. "Mining Version Histories for Detecting Code Smells". In: *IEEE Trans. Softw. Eng.* 41.5 (2015), pp. 462–489.

[284] F. Palomba, D. Di Nucci, A. Panichella, A. Zaidman, and A. De Lucia. "Lightweight Detection of Android-Specific Code Smells: The aDoctor Project". In: *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER '17)*. IEEE, 2017, pp. 487–491.

[285] F. Palomba, R. Oliveto, and A. De Lucia. "Investigating Code Smell Co-Occurrences Using Association Rule Learning: A Replicated Study". In: *Proc. IEEE Work. on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE '17)*. IEEE, 2017, pp. 8–13.

[286] F. Palomba, A. Panichella, A. De Lucia, R. Oliveto, and A. Zaidman. "A Textual-Based Technique for Smell Detection". In: *Proc. Int'l Conf. on Program Comprehension (ICPC '16)*. IEEE, 2016, pp. 1–10.

[287] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Comm. ACM* 15.12 (1972), pp. 1053–1058.

[288] D. L. Parnas. "On the Design and Development of Program Families". In: *IEEE Trans. Softw. Eng.* 1 (1976), pp. 1–9.

[289] L. Passos, R. Queiroz, M. Mukelabai, T. Berger, S. Apel, K. Czarnecki, and J. Padilla. "A Study of Feature Scattering in the Linux Kernel". In: *IEEE Trans. Softw. Eng.* (2018), p. 16.

[290] L. Passos, L. Teixeira, N. Dintzner, S. Apel, A. Wąsowski, K. Czarnecki, P. Borba, and J. Guo. "Coevolution of Variability Models and Related Software Artifacts". In: *Empir. Softw. Eng.* 21.4 (2016), pp. 1744–1793.

[291] X. Peng, Z. Xing, X. Tan, Y. Yu, and W. Zhao. "Iterative Context-Aware Feature Location (NIER Track)". In: *Proc. Int'l Conf. on Software Engineering (ICSE '11)*. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 900–903.

[292] T. Pfofe. "Automating the Synchronization of Software Variants". Master's Thesis. Germany: University of Magdeburg, Jan. 2016.

[293] T. Pfofe, T. Thüm, S. Schulze, W. Fenske, and I. Schaefer. "Synchronizing Software Variants with VariantSync". In: *Proc. Int'l Software Product Line Conf. (SPLC '16)*. ACM, 2016, pp. 329–332.

[294] E. K. Piveta, M. Hecht, M. S. Pimenta, and R. T. Price. "Detecting Bad Smells in AspectJ". In: *J. Univ. Comput. Science* 12.7 (2006), pp. 811–827.

[295] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Berlin Heidelberg, Germany: Springer, 2005.

[296] H. Post and C. Sinz. "Configuration Lifting: Verification Meets Software Configuration". In: *Proc. IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '08)*. IEEE, 2008, pp. 347–350.

[297] C. Prehofer. "Feature-Oriented Programming: A Fresh Look at Objects". In: *Proc. European Conf. on Object-Oriented Programming (ECOOP '97)*. Springer, 1997, pp. 419–443.

[298] R. Queiroz, L. Passos, M. T. Valente, C. Hunsen, S. Apel, and K. Czarnecki. "The Shape of Feature Code: An Analysis of Twenty C-Preprocessor-Based Systems". In: *Softw. & Syst. Modeling* 16.1 (2017), pp. 77–96.

[299] F. Rahman, C. Bird, and P. Devanbu. "Clones: What is That Smell?" In: *Empir. Softw. Eng.* 17.4-5 (2012), pp. 503–530.

[300] B. Ray and M. Kim. "A Case Study of Cross-System Porting in Forked Projects". In: *Proc. Int'l Symp. on the Foundations of Software Engineering (FSE '12)*. ACM, 2012, 53:1–53:11.

[301] *ReSharper by Language: C++*. 2017. URL: https://www.jetbrains.com/help/resharper/ReSharper_by_Language_CPP.html (visited on 11/26/2017).

[302] M. Ribeiro and P. Borba. "Improving Guidance when Restructuring Variabilities in Software Product Lines". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '09)*. IEEE, 2009, pp. 79–88.

[303] M. Ribeiro and P. Borba. "Recommending Refactorings when Restructuring Variabilities in Software Product Lines". In: *Proc. OOPSLA Work. on Refactoring Tools (WRT '08)*. ACM, 2008, 8:1–8:4.

[304] M. Rieger, S. Ducasse, and M. Lanza. "Insights Into System-Wide Code Duplication". In: *Proc. Working Conf. on Reverse Engineering (WCRE '04)*. IEEE, 2004, pp. 100–109.

[305] A. J. Riel. *Object-Oriented Design Heuristics*. Boston, MA, USA: Addison-Wesley, 1996.

[306] D. B. Roberts. "Practical Analysis for Refactoring". PhD thesis. Champaign, IL, USA: University of Illinois, 1999.

[307] M. P. Robillard and G. C. Murphy. "Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies". In: *Proc. Int'l Conf. on Software Engineering (ICSE '02)*. Orlando, Florida: ACM, 2002, pp. 406–416.

[308] D. Romano and M. Pinzger. "Using Source Code Metrics to Predict Change-Prone Java Interfaces". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '02)*. IEEE, 2007, pp. 303–312.

[309] D. Romano, P. Raila, M. Pinzger, and F. Khomh. "Analyzing the Impact of Antipatterns on Change-Proneness Using Fine-Grained Source Code Changes". In: *Proc. Working Conf. on Reverse Engineering (WCRE '12)*. IEEE, 2012, pp. 437–446.

[310] B. van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger. "On the Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test". In: *IEEE Trans. Softw. Eng.* 33.12 (2007), pp. 800–817.

[311] M. Rosenmüller, M. Kuhlemann, N. Siegmund, and H. Schirmeier. "Avoiding Variability of Method Signatures in Software Product Lines: A Case Study". In: *Proc. GPCE Work. on Aspect-Oriented Product Line Engineering (AOPLE '07)*. Workshop Website http://www.softeng.ox.ac.uk/aople/, 2007, pp. 20–25.

[312] C. K. Roy, J. R. Cordy, and R. Koschke. "Comparison and Evaluation of Code Clone Detection Techniques and Tools: A Qualitative Approach". In: *Sci. Comput. Prog.* 74.7 (2009), pp. 470–495.

[313] C. K. Roy and J. R. Cordy. *A Survey on Software Clone Detection Research*. Tech. rep. 541. Canada: Queen's University at Kingston, 2007.

[314] J. Rubin and M. Chechik. "A Framework for Managing Cloned Product Variants". In: *Proc. Int'l Conf. on Software Engineering (ICSE '13)*. San Francisco, CA, USA: IEEE, 2013, pp. 1233–1236.

[315] J. Rubin and M. Chechik. "A Survey of Feature Location Techniques". In: *Domain Engineering*. Springer, 2013, pp. 29–58.

[316] J. Rubin and M. Chechik. "From Products to Product Lines Using Model Matching and Refactoring". In: *Proc. Int'l Software Product Line Conf. (SPLC '10) (Volume 2)*. Lancaster University, 2010, pp. 155–162.

[317] J. Rubin, K. Czarnecki, and M. Chechik. "Managing Cloned Variants: A Framework and Experience". In: *Proc. Int'l Software Product Line Conf. (SPLC '13)*. Tokyo, Japan: ACM, 2013, pp. 101–110.

[318] J. Rubin, A. Kirshin, G. Botterweck, and M. Chechik. "Managing Forked Product Variants". In: *Proc. Int'l Software Product Line Conf. (SPLC '12)*. Salvador, Brazil: ACM, 2012, pp. 156–160.

[319] A. Saboury, P. Musavi, F. Khomh, and G. Antoniol. "An Empirical Study of Code Smells in JavaScript Projects". In: *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER '17)*. IEEE, 2017, pp. 294–305.

[320] D. Sands. "Total Correctness by Local Improvement in the Transformation of Functional Programs". In: *ACM Trans. Program. Lang. Syst.* 18.2 (1996), pp. 175–234.

[321] I. Şavga and F. Heidenreich. "Refactoring in Feature-Oriented Programming: Open Issues". In: *Proc. GPCE Work. on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE '08)*. University of Passau, Germany, Oct. 2008, pp. 41–46.

[322] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. "Delta-Oriented Programming of Software Product Lines". In: *Proc. Int'l Software Product Line Conf. (SPLC '10)*. Springer, 2010, pp. 77–91.

[323] S. Schulze. "Analysis and Removal of Code Clones in Software Product Lines". Dissertation. Germany: University of Magdeburg, Jan. 2013.

[324] S. Schulze, S. Apel, and C. Kästner. "Code Clones in Feature-Oriented Software Product Lines". In: *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE '10)*. ACM, 2010, pp. 103–112.

[325] S. Schulze, E. Jürgens, and J. Feigenspan. "Analyzing the Effect of Preprocessor Annotations on Code Clones". In: *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM '11)*. IEEE, 2011, pp. 115–124.

[326] S. Schulze, M. Kuhlemann, and M. Rosenmüller. "Towards a Refactoring Guideline Using Code Clone Classification". In: *Proc. OOPSLA Work. on Refactoring Tools (WRT '08)*. ACM, 2008, 6:1–6:4.

[327] S. Schulze, J. Liebig, J. Siegmund, and S. Apel. "Does the Discipline of Preprocessor Annotations Matter? A Controlled Experiment". In: *Proc. Int'l Conf. on Generative Programming: Concepts & Experiences (GPCE '13)*. ACM, 2013, pp. 65–74.

[328] S. Schulze, M. Lochau, and S. Brunswig. "Implementing Refactorings for FOP: Lessons Learned and Challenges Ahead". In: *Proc. Int'l Work. on Feature-Oriented Software Development (FOSD '13)*. ACM, 2013, pp. 33–40.

[329] S. Schulze, O. Richers, and I. Schaefer. "Refactoring Delta-Oriented Software Product Lines". In: *Proc. Int'l Conf. on Aspect-Oriented Software Development (AOSD '13)*. ACM, 2013, pp. 73–84.

[330] S. Schulze, T. Thüm, M. Kuhlemann, and G. Saake. "Variant-Preserving Refactoring in Feature-Oriented Software Product Lines". In: *Proc. Int'l Work. on Variability Modeling of Software-Intensive Systems (VaMoS '12)*. ACM, 2012, pp. 73–81.

[331] S. Schulze. "Feature-orientiertes Refactoring zur Migration von Produktvarianten". In German. Master's Thesis. Germany: University of Magdeburg, Feb. 2016.

[332] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw. "Building Empirical Support for Automated Code Smell Detection". In: *Proc. 2010 Int'l Symp. on Empirical Software Engineering (ESEM '10)*. ACM, 2010, 8:1–8:10.

[333] D. C. Sharp. "Reducing Avionics Software Cost Through Component Based Product Line Development". In: *Proc. Digital Avionics Systems Conf. (DASC '98)*. Vol. 2. IEEE, 1998, G32/1–G32/8.

[334] I. Shoenberger, M. W. Mkaouer, and M. Kessentini. "On the Use of Smelly Examples to Detect Code Smells in JavaScript". In: *Proc. European Conf. on the Applications of Evolutionary Computation (EvoApplications '17)*. Springer, 2017, pp. 20–34.

[335] F. Simon, F. Steinbrückner, and C. Lewerentz. "Metrics Based Refactoring". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '01)*. IEEE, 2001, pp. 30–38.

[336] J. Sincero, H. Schirmeier, W. Schröder-Preikschat, and O. Spinczyk. "Is the Linux Kernel a Software Product Line?" In: *Proc. Work. on Open Source Software and Product Lines (SPLC-OSSPL '07)*. 2007.

[337] D. I. Sjøberg, B. Anda, and A. Mockus. "Questioning Software Maintenance Metrics: A Comparative Case Study". In: *Proc. 2012 Int'l Symp. on Empirical Software Engineering (ESEM '12)*. ACM, 2012, pp. 107–110.

[338] D. I. Sjøberg, A. Yamashita, B. C. D. Anda, A. Mockus, and T. Dyba. "Quantifying the Effect of Code Smells on Maintenance Effort". In: *IEEE Trans. Softw. Eng.* 39.8 (2013), pp. 1144–1156.

[339] J. Śliwerski, T. Zimmermann, and A. Zeller. "When Do Changes Induce Fixes?" In: *ACM SIGSOFT Softw. Eng. Notes* 30.4 (2005), pp. 1–5.

[340] P. Sochos, I. Philippow, and M. Riebisch. "Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture". In: *Lecture Notes in Computer Science 3263*. Springer, 2004, pp. 138–152.

[341] F. Z. Sokol, M. Finavaro Aniche, and M. A. Gerosa. "MetricMiner: Supporting Researchers in Mining Software Repositories". In: *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM '13)*. IEEE, 2013, pp. 142–146.

[342] B. L. Sousa, P. P. Souza, E. Fernandes, K. A. Ferreira, and M. A. Bigonha. "FindSmells: Flexible Composition of Bad Smell Detection Strategies". In: *Proc. Int'l Conf. on Program Comprehension (ICPC '17)*. IEEE, 2017, pp. 360–363.

[343] H. Spencer and G. Collyer. "#ifdef Considered Harmful, or Portability Experience With C News". In: *Proc. Proc. USENIX Conf.* USENIX Association, 1992, pp. 185–197.

[344] D. Spinellis. "CScout: A refactoring browser for C". In: *Sci. Comput. Prog.* 75.4 (2010), p. 216.

[345] D. Spinellis. "Global Analysis and Transformations in Preprocessed Languages". In: *IEEE Trans. Softw. Eng.* 29.11 (2003), pp. 1019–1030.

[346] K. Srivisut and P. Muenchaisri. "Bad-Smell Metrics for Aspect-Oriented Software". In: *Proc. IEEE/ACIS Int'l Conf. on Computer and Information Science (ICIS '07)*. IEEE, 2007, pp. 1060–1065.

[347] K. Srivisut and P. Muenchaisri. "Defining and Detecting Bad Smells of Aspect-Oriented Software". In: *Proc. Int'l Computer Software and Applications Conference (COMPSAC '07)*. IEEE, 2007, pp. 65–70.

[348] Ş. Stănciulescu, T. Berger, E. Walkingshaw, and A. Wąsowski. "Concepts, Operations, and Feasibility of a Projection-Based Variation Control System". In: *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME '16)*. IEEE, 2016, pp. 323–333.

[349] Ş. Stănciulescu, S. Schulze, and A. Wąsowski. "Forked and Integrated Variants in an Open-Source Firmware Project". In: *Proc. Int'l Conf. on Software Maintenance and Evolution (ICSME '15)*. IEEE, 2015, pp. 151–160.

[350] M. Staples and D. Hill. "Experiences Adopting Software Product Line Development Without a Product Line Architecture". In: *Proc. Asia Pacific Software Engineering Conf. (APSEC '04)*. IEEE, 2004, pp. 176–183.

[351] M. Steinbeck. "An Arc-Based Approach for Visualization of Code Smells". In: *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER '17)*. IEEE, 2017, pp. 397–401.

[352] R. Subramanyam and M. S. Krishnan. "Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects". In: *IEEE Trans. Softw. Eng.* 29.4 (2003), pp. 297–310.

[353] G. Succi, W. Pedrycz, M. Stefanovic, and J. Miller. "Practical Assessment of the Models for Identification of Defect-Prone Classes in Object-Oriented Commercial Systems Using Design Metrics". In: *J. Syst. Softw.* 65.1 (2003), pp. 1–12.

[354] M. Svahnberg, J. Van Gurp, and J. Bosch. "A Taxonomy of Variability Realization Techniques". In: *Softw.: Pract. Exper.* 35.8 (2005), pp. 705–754.

[355] *Synopsys Static Analysis (Coverity)*. 2017. URL: https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.htm (visited on 07/10/2017).

[356] M.-H. Tang, M.-H. Kao, and M.-H. Chen. "An Empirical Study on Object-Oriented Metrics". In: *Proc. Int'l Software Metrics Symposium (ISMS '99)*. IEEE, 1999, pp. 242–249.

[357] C. Thao, E. V. Munson, and T. N. Nguyen. "Software Configuration Management for Product Derivation in Software Product Families". In: *Proc. Int'l Conf. on Engineering of Computer Based Systems (ECBS '08)*. IEEE, 2008, pp. 265–274.

[358] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, and T. Leich. "FeatureIDE: An Extensible Framework for Feature-Oriented Software Development". In: *Sci. Comput. Prog.* 79 (2014), pp. 70–85.

[359] T. Thüm, C. Kästner, S. Erdweg, and N. Siegmund. "Abstract Features in Feature Modeling". In: *Proc. Int'l Software Product Line Conf. (SPLC '11)*. IEEE, 2011, pp. 191–200.

[360] K. Tonscheidt. "Leveraging Code Clone Detection for the Incremental Migration of Cloned Product Variants to a Software Product Line: An Explorative Study". Bachelor's Thesis. Germany: University of Magdeburg, June 2015.

[361] M. Toomim, A. Begel, and S. L. Graham. "Managing Duplicated Code with Linked Editing". In: *Proc. IEEE Symp. on Visual Languages and Human-Centric Computing (VL/HCC '04)*. IEEE, 2004, pp. 173–180.

[362] T. Tourwé and T. Mens. "Identifying Refactoring Opportunities Using Logic Meta Programming". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '03)*. IEEE, 2003, pp. 91–100.

[363] A. Trifu and R. Marinescu. "Diagnosing Design Problems in Object Oriented Systems". In: *Proc. Working Conf. on Reverse Engineering (WCRE '05)*. IEEE, 2005, pp. 155–164.

[364] S. Trujillo, D. Batory, and O. Diaz. "Feature Refactoring a Multi-Representation Program into a Product Line". In: *Proc. Int'l Conf. on Generative Programming and Component Engineering (GPCE '06)*. ACM, 2006, pp. 191–200.

[365] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou. "JDeodorant: Identification and Removal of Type-Checking Bad Smells". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '08)*. IEEE, 2008, pp. 329–331.

[366] N. Tsantalis and A. Chatzigeorgiou. "Identification of Move Method Refactoring Opportunities". In: *IEEE Trans. Softw. Eng.* 35.3 (2009), pp. 347–367.

[367] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk. "An Empirical Investigation into the Nature of Test Smells". In: *Proc. IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE '16)*. ACM, 2016, pp. 4–15.

[368] M. T. Valente, V. Borges, and L. Passos. "A Semi-Automatic Approach for Extracting Software Product Lines". In: *IEEE Trans. Softw. Eng.* 38.4 (2012), pp. 737–754.

[369] L. Vidács, Á. Beszédes, and T. Gyimóthy. "Combining Preprocessor Slicing with C/C++ Language Slicing". In: *Sci. Comput. Prog.* 74.7 (2009), pp. 399–413.

[370] M. Vittek. "Refactoring Browser With Preprocessor". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '03)*. IEEE, 2003, pp. 101–110.

[371] S. Wagner, A. Abdulkhaleq, K. Kaya, and A. Paar. "On the Relationship of Inconsistent Software Clones and Faults: An Empirical Study". In: *Proc. Int'l Conf. on Software Analysis, Evolution, and Reengineering (SANER '16)*. IEEE, 2016, pp. 79–89.

[372] W. C. Wake. *Refactoring Workbook*. Addison-Wesley, 2003.

[373] J. H. Weber, A. Katahoire, and M. Price. "Uncovering Variability Models for Software Ecosystems from Multi-Repository Structures". In: *Proc. Int'l Work. on Variability Modeling of Software-Intensive Systems (VaMoS '15)*. ACM, 2015, pp. 103–108.

[374] M. Weiser. "Program Slicing". In: *IEEE Trans. Softw. Eng.* 4.10 (1984), pp. 352–357.

[375] N. Wilde and M. C. Scully. "Software Reconnaissance: Mapping Program Features to Code". In: *Softw. Maint.: Res. Pract.* 7.1 (1995), pp. 49–62.

[376] Y. Xue. "Reengineering Legacy Software Products into Software Product Line". PhD thesis. Singapore: National University of Singapore, 2013.

[377] Y. Xue. "Reengineering Legacy Software Products into Software Product Line Based on Automatic Variability Analysis". In: *Proc. Int'l Conf. on Software Engineering (ICSE '11)*. ACM, 2011, pp. 1114–1117.

[378] Y. Xue, Z. Xing, and S. Jarzabek. "Understanding Feature Evolution in a Family of Product Variants". In: *Proc. Working Conf. on Reverse Engineering (WCRE '10)*. IEEE, 2010, pp. 109–118.

[379] A. Yamashita. "Assessing the Capability of Code Smells to Explain Maintenance Problems: An Empirical Study Combining Quantitative and Qualitative Data". In: *Empir. Softw. Eng.* 19.4 (2014), pp. 1111–1143.

[380] A. Yamashita. "How Good Are Code Smells for Evaluating Software Maintainability? Results From a Comparative Case Study". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '13)*. IEEE, 2013, pp. 566–571.

[381] A. Yamashita and L. Moonen. "Do Code Smells Reflect Important Maintainability Aspects?" In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '12)*. IEEE, 2012, pp. 306–315.

[382] A. Yamashita and L. Moonen. "Do Developers Care About Code Smells? An Exploratory Survey". In: *Proc. Working Conf. on Reverse Engineering (WCRE '13)*. IEEE, 2013, pp. 242–251.

[383] A. Yamashita and L. Moonen. "Exploring the Impact of Inter-Smell Relations on Software Maintainability: An Empirical Study". In: *Proc. Int'l Conf. on Software Engineering (ICSE '13)*. IEEE, 2013, pp. 682–691.

[384] A. Yamashita and L. Moonen. "To What Extent Can Maintenance Problems Be Predicted by Code Smell Detection? – An Empirical Study". In: *Inform. Software Tech.* 55.12 (Dec. 2013), pp. 2223–2242.

[385] N. Yoshida, Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. "On Refactoring Support Based on Code Clone Dependency Relation". In: *Proc. IEEE Int'l Software Metrics Symposium (METRICS '05)*. IEEE, 2005, 10–pp.

[386] K. Yoshimura, D. Ganesan, and D. Muthig. "Assessing Merge Potential of Existing Engine Control Systems into a Product Line". In: *Proc. Int'l Work. on Software Engineering for Automotive Systems (SEAS '06)*. ACM, 2006, pp. 61–67.

[387] G. Zhang, L. Shen, X. Peng, Z. Xing, and W. Zhao. "Incremental and Iterative Reengineering Towards Software Product Line: An Industrial Case Study". In: *Proc. IEEE Int'l Conf. on Software Maintenance (ICSM '11)*. IEEE, 2011, pp. 418–427.

[388] M. Zhang, T. Hall, and N. Baddoo. "Code Bad Smells: A Review of Current Knowledge". In: *Softw. Maint. Evol.: Res. Pract.* 23.3 (2011), pp. 179–202.

[389] W. Zhang, S. Jarzabek, N. Loughran, and A. Rashid. "Reengineering a PC-Based System into the Mobile Device Product Line". In: *Proc. Int'l Work. on Principles of Software Evolution (IWPSE '03)*. IEEE, 2003, pp. 149–160.

[390] Y. Zhou, H. Leung, and B. Xu. "Examining the Potentially Confounding Effect of Class Size on the Associations Between Object-Oriented Metrics and Change-Proneness". In: *IEEE Trans. Softw. Eng.* 35.5 (2009), pp. 607–623.

[391] T. Ziadi, L. Frias, M. da Silva, and M. Ziane. "Feature Identification From the Source Code of Product Variants". In: *Proc. European Conf. on Software Maintenance and Reengineering (CSMR '12)*. IEEE, 2012, pp. 417–422.

[392] M. F. Zibran and C. K. Roy. "A Constraint Programming Approach to Conflict-Aware Optimal Scheduling of Prioritized Code Clone Refactoring". In: *Proc. Int'l Working Conf. on Source Code Analysis and Manipulation (SCAM '11)*. IEEE, 2011, pp. 105–114.

Ehrenerklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Ferner versichere ich, dass die Verwendung eigener und fremder Quellen als solche kenntlich gemacht habe. Ich habe keine Hilfe von kommerziellen Promotionsberatern in Anspruch genommen. Dritte haben von mir weder unmittelbar noch mittelbar geldwerte Leistungen für Arbeiten erhalten, die im Zusammenhang mit dem Inhalt der vorgelegten Dissertation stehen.

Insbesondere habe ich nicht wissentlich:

- Ergebnisse erfunden oder widersprüchliche Ergebnisse verschwiegen,
- statistische Verfahren absichtlich missbraucht, um Daten in ungerechtfertigter Weise zu interpretieren,
- fremde Ergebnisse oder Veröffentlichungen plagiiert,
- fremde Forschungsergebnisse verzerrt wiedergegeben.

Mir ist bekannt, dass Verstöße gegen das Urheberrecht Unterlassungs- und Schadenersatzansprüche des Urhebers sowie eine strafrechtliche Ahndung durch die Strafverfolgungsbehörden begründen können. Die Arbeit wurde bisher weder im Inland noch im Ausland in gleicher oder ähnlicher Form als Dissertation eingereicht und ist als Ganzes auch noch nicht veröffentlicht.

Magdeburg, 16. März 2020

Wolfram Fenske