ANOMALY AGGLOMERATION AS SIGN OF PRODUCT LINE INSTABILITIES

EDUARDO MOREIRA FERNANDES

ANOMALY AGGLOMERATION AS SIGN OF PRODUCT LINE INSTABILITIES

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Eduardo Figueiredo

Belo Horizonte

Março de 2017

EDUARDO MOREIRA FERNANDES

ANOMALY AGGLOMERATION AS SIGN OF PRODUCT LINE INSTABILITIES

Dissertation presented to the Graduate Program in Ciência da Computação of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Ciência da Computação.

Advisor: Eduardo Figueiredo

Belo Horizonte
March 2017

© 2017, Eduardo Moreira Fernandes.
Todos os direitos reservados

.

Ficha catalográfica elaborada pela Biblioteca do ICEx - UFMG

Fernandes, Eduardo Moreira.

F363a Anomaly agglomeration as sign of product line instabilities. / Eduardo Moreira Fernandes. – Belo Horizonte, 2017.
xxii, 68 f.: il.; 29 cm.

Dissertação (mestrado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.

Orientador: Eduardo Magno Lages Figueiredo.

1. Computação – Teses. 2. Engenharia de software. 3. Linha de produtos de software. 4. Anomalias de código. I. Orientador. II. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Anomaly agglomeration as sign of product line instabilities

EDUARDO MOREIRA FERNANDES

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]
PROF. EDUARDO MAGNO LAGES FIGUEIREDO - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]
PROF. CLÁUDIO NOGUEIRA SANT'ANNA
Departamento de Ciência da Computação - UFBA

[illegible]
PROF. MARCO TÚLIO DE OLIVEIRA VALENTE
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 22 de fevereiro de 2017.

Acknowledgments

To the almighty Lord, for giving me life, a bunch of dreams, and people to teach me how to achieve them. To my family and parents, for the support in both sunny and stormy days. I appreciate that. Also to the true friends from Brazil and all over the world. Many thanks for the company and the best wishes.

To the advisor of this dissertation, as well as the qualified researchers and students that I had the chance to work with. In addition, to the members of this dissertation defense. To the Federal University of Minas Gerais (UFMG) family, for having me in its institution during two key years of my academic life.

Finally, to the Graduate Program in Computer Science (PPGCC) of the Department of Computer Science (DCC), as well as the Coordination for the Improvement of Higher Education Personnel (CAPES). I appreciate every opportunity that I had, including the financial support. Thank you very much.

“Viciously have they attacked me from my youth,
yet they have not prevailed against me.”
(Psalms 129:2)

Resumo

Uma Linha de Produtos de Software (LPS) é um conjunto de sistemas de software que compartilham características comuns e variáveis. Para prover reúso em larga escala, os componentes de uma LPS devem ser de fácil manutenção. Portanto, desenvolvedores devem identificar as estruturas de código anômalas – isto é, as anomalias de código – que prejudicam a manutenção de LPSs. Caso contrário, mudanças em uma LPS podem eventualmente propagar-se a características sem aparente inter-relação e afetar diversos produtos da LPS. Após revisarmos a literatura, encontramos algumas estratégias de detecção e várias ferramentas para detecção de anomalias de código. Em geral, tanto as estratégias quanto as ferramentas apresentam resultados de detecção similares, e algumas ferramentas são compatíveis com LPS. Assim, assumimos que a detecção de anomalias individuais de código é um problema suficientemente tratado pela literatura. Trabalhos anteriores frequentemente assumem que anomalias isoladas são suficientes para caracterizar problemas de manutenção em LPS, ainda que cada anomalia possa representar uma visão parcial, insignificante ou inexistente da extensão de um problema. Portanto, tais estudos possuem dificuldades em caracterizar estruturas anômalas que indiquem problemas de manutenção em LPS. Nesta dissertação, estudamos o contexto de cada anomalia e observamos que algumas anomalias podem estar interconectadas, formando as chamadas aglomerações de anomalias. Duas ou mais anomalias compõem uma aglomeração em LPS se afetam em conjunto uma característica, uma hierarquia de características ou um componente. Caracterizamos três tipos de aglomeração de anomalias de código em LPS e investigamos o potencial de anomalias aglomeradas, ou não-aglomeradas, representarem, no contexto de LPS, fontes de um problema de manutenção específico: instabilidade. Analisamos diversas versões de quatro LPSs orientadas por características. Nossos resultados sugerem que aglomeração em hierarquia de características pode indicar até 89% de fontes de instabilidade em LPS, provendo melhores resultados em comparação a anomalias não-aglomeradas.

Palavras-chave: Anomalia de Código, Linha de Produtos de Software, Instabilidade.

Abstract

Software Product Line (SPL) is a set of software systems that share common and varying features. To provide large-scale reuse, the components of a SPL should be easy to maintain. Therefore, developers have to identify anomalous code structures, i.e., code anomalies, that are detrimental to the SPL maintainability. Otherwise, SPL changes can propagate to seemly-unrelated features and affect various SPL products. After reviewing the literature, we have found some detection strategies and several tools for code anomaly detection. In general, both strategies and tools provide similar detection results, and some tools are compatible with SPL. We then assume that the problem of detecting single code anomalies is sufficiently covered by the literature. Previous work often assume that each code anomaly alone suffices to characterize SPL maintenance problems, though each single anomaly may represent only a partial, insignificant, or even non-existent view of the problem extent. Consequently, previous studies have difficulties in characterizing anomalous code structures that indicate SPL maintenance problems. In this dissertation, we study the surrounding context of each anomaly and observe that certain anomalies may be interconnected, thereby forming so-called anomaly agglomerations. Two or more anomalies form an agglomeration in SPL when they affect the same SPL structural element, i.e., a feature, a feature hierarchy, or a component. We then investigate to what extent non-agglomerated and agglomerated anomalies represent sources of a specific SPL maintenance problem: instability. We analyze various releases of four feature-oriented SPLs. Our findings suggest that feature hierarchy agglomerations indicate up to 89% of sources of instability, i.e., a better result when compared with non-agglomerated anomalies.

Keywords: Code Anomaly, Software Product Line, Software Instability.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Software Product Line (SPL) is a set of software systems that share common and varying features (Pohl et al., 2005). Each feature is an increment in functionality of the product-line systems (Apel et al., 2013). The combination of features generates different products (Batory et al., 2003). SPL aims to provide large-scale reuse with a decrease in the maintenance effort (Pohl et al., 2005). The implementation of a feature can be distributed into one or more source files, called *components*. To support large-scale reuse, the components and features of a SPL should be easy to maintain. Therefore, developers should identify anomalous code structures, i.e., *code anomalies*, that are detrimental to the SPL maintainability. Otherwise, changes can propagate to seemly-unrelated features and affect various SPL products.

Code anomalies are symptoms of problems in a software system (Fowler, 1999). They harm the software maintainability in several levels, e.g., by affecting classes and methods (Fowler, 1999; Lanza and Marinescu, 2006). Code anomalies affect any system, including SPL (Fenske and Schulze, 2015). A previous work states that SPL-specific anomalies can be easier to introduce, harder to fix, and more critical than others, due to the inherent SPL complexity (Medeiros et al., 2015). For instance, *Long Refinement Chain* (Fenske and Schulze, 2015) affects the feature hierarchy. It hinders developers to understand the code and perform proper changes. These changes might affect several SPL products. Thus, understanding the negative impact of anomalies in the SPL maintainability is even more important than in standalone systems, as their side effects may affect multiple products. Still, there is little understanding about such impact.

Two approaches may support the detection of code anomalies in software product lines (Moha et al., 2010). The manual detection relies on source code inspection conducted by developers, a generally slow and error-prone activity (Munro, 2005). In turn, the automated detection counts on the support of *detection strategies* or equivalent techniques (Fard and Mesbah, 2013; Moha et al., 2010; Vidal et al., 2016). A detection strategy is a composition of metric-based rules that defines when a specific software component – e.g., a class, a method, or a package of the software systems – is prone to contain a given type of code anomaly (Lanza and Marinescu, 2006).

1.1 Motivation

Some studies assume that each code anomaly alone suffices to characterize SPL maintenance problems (Fenske and Schulze, 2015; Schulze et al., 2010). However, each single anomaly may represent only a partial view of the problem. That is, a maintenance problem tends to scatter into different parts of the code (Moha et al., 2010). For instance, *Long Method* is a method with too many responsibilities that, if isolated, represents a punctual, simple problem (Fowler, 1999). In turn, *Long Refinement Chain* is a method with too many refinements in different features (Fenske and Schulze, 2015) that, in isolation, is not critical depending on the refined method. However, if we observe both anomalies in the same method, we may assume an increasing potential of the anomalies in hindering the SPL maintainability, since an anomalous long method is excessively refined and causes a wider problem. As a result, previous studies have limitations to characterize anomalous structures that indicate SPL maintenance problems.

On the other hand, a previous work (Oizumi et al., 2016) has observed that certain anomalies may be interconnected, forming so-called *anomaly agglomerations*. They investigated to what extent these anomaly agglomerations support the characterization of maintenance problems in single systems. The authors define a code anomaly agglomeration as a group of two or more anomalous code elements directly or indirectly inter-related in the source code of a system (Oizumi et al., 2016). However, they did not characterize and studied specific types of anomaly agglomerations in SPLs.

Although a SPL is mostly similar to a general software system, such as an object-oriented system, one should consider SPL-specific structural elements when agglomerating anomalies. For instance, in feature-oriented SPLs (Apel et al., 2013), components are *constants* that implement basic SPL resources, or *refinements* that add resources to a given constant. Also, successive refinements to a constant, located in different SPL features, form a refinement chain (Batory et al., 2003). A SPL product will contain the implementation of all components of the refinement chain whose features are included in the product. Consequently, the code anomalies that affect these components are also included in the derived product. Thus, refinement chain is a SPL-specific structural element that is relevant when proposing types of anomaly agglomerations in SPL.

1.2 Studies on Code Anomaly Detection

In the first part of this dissertation (Fernandes et al., 2016), we present a systematic literature review (Kitchenham and Charters, 2007), and a comparative study, of code anomaly detection tools. We aimed at understanding the current support to single anomaly detection, different anomalies, and target programming languages. Regarding the literature review, we have found 84 tools, of which 29 are downloadable. Most of the tools are compatible with Java, C, and C++, and a some are compatible with SPL. The tools detect 61 different code anomalies, of which 20 are listed by Fowler (1999). With respect to the comparative study, the tools performed with moderate recall and high precision, in general. However, we observed that different tools tend to present similar detection results for the same software system.

In the second part of this dissertation (Fernandes et al., 2017a), we investigate different detection strategies for well-known code anomalies that affect any system, including SPL, such as *Large Class* and *Long Method* (Fowler, 1999). For this purpose, we conducted an *ad hoc* selection of detection strategies from the literature for comparison with novel strategies for both anomalies, in the SPL context. As a result, we have found a few different strategies proposed in the literature for each anomaly. In addition, we observed that the strategies also tend to provide similar detection results, as in the case of detection tools. From the two first parts of the dissertation, we assume that the problem of detecting single code anomalies is sufficiently covered by the literature. Thus, we were able to investigate code anomaly agglomerations.

1.3 Proposed Anomaly Agglomerations in SPL

In the third part of this dissertation, we propose three novel types of anomaly agglomerations in SPL, namely *feature agglomeration*, *feature hierarchy agglomeration*, and *component agglomeration*. The propose agglomerations rely on three key SPL structural elements, i.e., features, refinement chains, and components (Batory et al., 2003). Figure 1.1 illustrates each agglomeration. The feature f_1 has a *feature agglomeration* composed of the components c_1, c_2, and c_3. That is, at least one anomaly affects each of these components, and all are located in the same feature. In turn, the components c_3 located in the features f_1 and f_2 form a *feature hierarchy agglomeration.* In other words, each of these components are affected by at least one anomaly and, in addition, they are hierarchically interconnected by a refinement relationship in the SPL. Finally, the components c_1 of the feature f_1 and c_4 of the feature f_2 are affected by two or more code anomalies each. Therefore, both components have a *component agglomeration.*

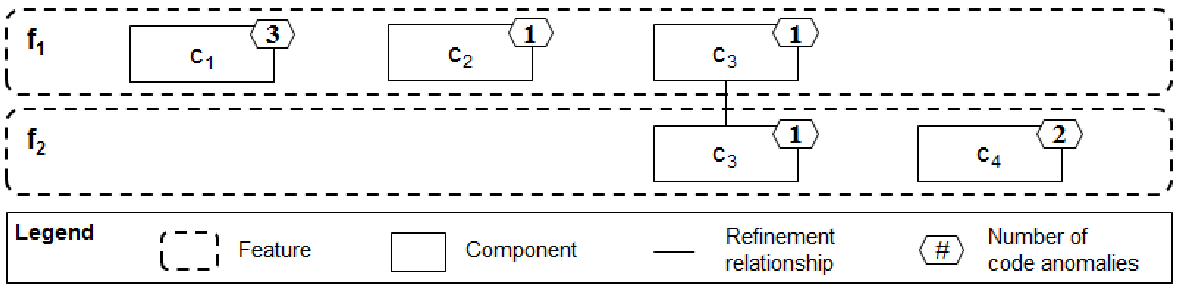


Figure 1.1. Abstract Examples of Anomaly Agglomerations in SPL

1.4 Evaluation of SPL Stability

Also in this dissertation, we investigate how often non-agglomerated *versus* agglomerated code anomalies occur in SPLs and whether they indicate sources of a specific SPL maintenance problem: instability (Ampatzoglou et al., 2015). We chose instability for studying because (i) it is one of the most harmful maintenance problems in software systems (Yau and Collofello, 1985), even in the SPL context, and (ii) previous work (Figueiredo et al., 2009; Khomh et al., 2009) has found evidence that code anomalies can induce to instability in software systems. We analyze different releases of four feature-oriented SPLs, namely MobileMedia (Figueiredo et al., 2008), Notepad (Kim et al., 2010), TankWar (Schulze et al., 2010), and WebStore (Gaia et al., 2014).

We perform our analysis in two steps described as follows. First, for each proposed type of anomaly agglomeration, we compute the strength of the relationship between agglomerations and instability in SPLs. For this purpose, we use Fisher's test (Fisher, 1922) to assess the statistical significance of the relationship, and Odds Ratio (Cornfield, 1951) to compute the possibility of anomaly agglomerations in indicating instability. Second, we compute the accuracy of anomaly agglomerations in indicating sources of instability in the target SPLs, per type of agglomeration. We compute recall and precision (Fawcett, 2006) to support the accuracy assessment.

As a result, our data suggest that *feature hierarchy agglomerations* and instability are strongly related and, therefore, this type of anomaly agglomeration is an effective indicator of instabilities. The high precision of 89% suggests that feature hierarchy can support developers in anticipating SPL maintenance problems. These findings are quite interesting because the implementation of feature-oriented SPLs is rooted strongly on the notion of feature hierarchies. It indicates that developers of feature-oriented SPLs should design carefully the feature hierarchies, since they might generate hierarchical structures that hamper the SPL maintainability.

1.5 Dissertation Outline

The remainder of this dissertation is organized as follows.

Chapter 2 provides background information to support the understanding of this dissertation. We discuss feature-oriented SPLs and problems that harm the maintainability of product lines, such as instability.

Chapter 3 discusses the two first parts of this dissertation. First, we present a systematic literature review and a comparative study of code anomaly detection tools. Second, we present a comparative study of code anomaly detection strategies.

Chapter 4 proposes three novel types of code anomaly agglomerations in SPL, namely *feature*, *feature hierarchy*, and *component agglomeration*. We rely on the main structural elements of SPL to propose each agglomeration.

Chapter 5 describes an empirical study to evaluate our novel types of anomaly agglomerations in the context of four feature-oriented SPLs. In this chapter, we present the study goal and research questions, target SPLs, and protocols. We also present and analyze the study results. Finally, we discuss threats to the study validity.

Chapter 6 discusses related work, including a study that proposes code anomaly agglomerations in the context of object-oriented software systems.

Finally, **Chapter 7** concludes this dissertation and presents future work.

Chapter 2

Background

Software Product Line (SPL) aims to provide reuse through a configurable set of systems that share features (Pohl et al., 2005). However, the SPL components should be easy to maintain for effective reuse. Therefore, developers should identify and fix code anomalies that harm the SPL maintainability. Code anomalies are symptoms of problems in a system (Fowler, 1999). They affect SPLs (Fenske and Schulze, 2015) and tend to be more critical in a SPL than in a single system (Medeiros et al., 2015). Previous work assume that each anomaly alone suffices to characterize SPL maintenance problems (Fenske and Schulze, 2015; Schulze et al., 2010), though a single anomaly may represent only a partial view of the problem extent. Thus, previous work have limitations to characterize anomalous structures that harm the SPL maintainability.

This dissertation addresses the aforementioned limitations by supporting the identification of major SPL maintenance problems. We propose three novel types of anomaly agglomerations in SPL, i.e., three techniques to inter-relate anomalies that affect different part of the SPL source code. Our study relies on a previous work that proposes techniques to agglomerate code anomalies in single software systems (Oizumi et al., 2016). We also evaluate each novel type of agglomeration as an indicator of a specific SPL maintenance problem: instability. To provide support for understanding the study presented in this dissertation, this chapter presents background information. Section 2.1 presents basic concepts of feature-oriented software product lines. Section 2.2 discusses SPL maintenance problems focused on instability.

2.1 Feature-Oriented Software Product Lines

Software Product Line (SPL) is a set of systems that share common and varying features (Pohl et al., 2005). Software products derived from a SPL differ themselves by specific sets of varying features (Pohl et al., 2005). Variability consists of all possible combinations of features that may compose a SPL product (Andrade et al., 2014). Through systematic, large-scale reuse, SPL aims to reduce time-to-market and improve software quality (Pohl et al., 2005). These advantages have attracted several organizations, such as Boing, Bosch, and Toshiba, to apply SPLs successfully in industry settings (Apel et al., 2013). Annotative (Liebig et al., 2010) and compositional (Andrade et al., 2014) techniques are used to develop a SPL. Examples of these techniques are preprocessors (Liebig et al., 2010) and aspect-oriented programming (Kiczales et al., 1997), respectively. Although annotative techniques are easier to apply, they can lead to several maintenance problems that compositional techniques aim to better address by supporting the modularization of features (Kästner and Apel, 2008).

In this dissertation, we analyze SPLs developed using Feature-Oriented Programming (FOP) (Batory et al., 2003). FOP is a compositional technique in which physically separated code units are composed, generally in compile-time or deploy-time, to generate different product-line systems (Kästner and Apel, 2008). FOP aims to guide the implementation of specific features in separated code units. As a broad concept, feature orientation has been largely applied in industry to support the development of SPLs (Lee and Muthig, 2006). The SPLs analyzed in this study are feature-oriented and implemented using the AHEAD (Batory et al., 2003) language-specific technology and the FeatureHouse (Apel et al., 2009) multi-language technology. Although a few alternative technologies support FOP, such as CaesarJ (Mezini and Ostermann, 2004) and FeatureC++ (Apel et al., 2005), the selected ones are relatively recent and rely on prior techniques for implementing feature-orientation.

We discuss the main structural elements of feature-oriented SPLs as follows.

Features. In feature-oriented SPLs, each feature is an increment in functionality of the product-line systems (Apel et al., 2013). The combination of features generates different SPL products (Batory et al., 2003). Let us illustrate features, and also the other main structural elements of feature-oriented SPLs, in the design level. Figure 2.1 presents the partial design view of MobileMedia (Figueiredo et al., 2008) in Release 7. MobileMedia is a SPL for creation of mobile applications that manage media resources, such as photos, music, and video. Figure 2.1 is a partial view of the SPL and represents only some of the components of MobileMedia. It depicts four features of the SPL, delimited by dashed lines: *MediaManagement*, *CreateMedia*, *Sorting*, and *SMSTransfer*. The first and second features implement management and creation of mobile media, such as photos. The third feature implements sorting of media. Finally, the fourth feature implements the SMS transfer between mobile devices.

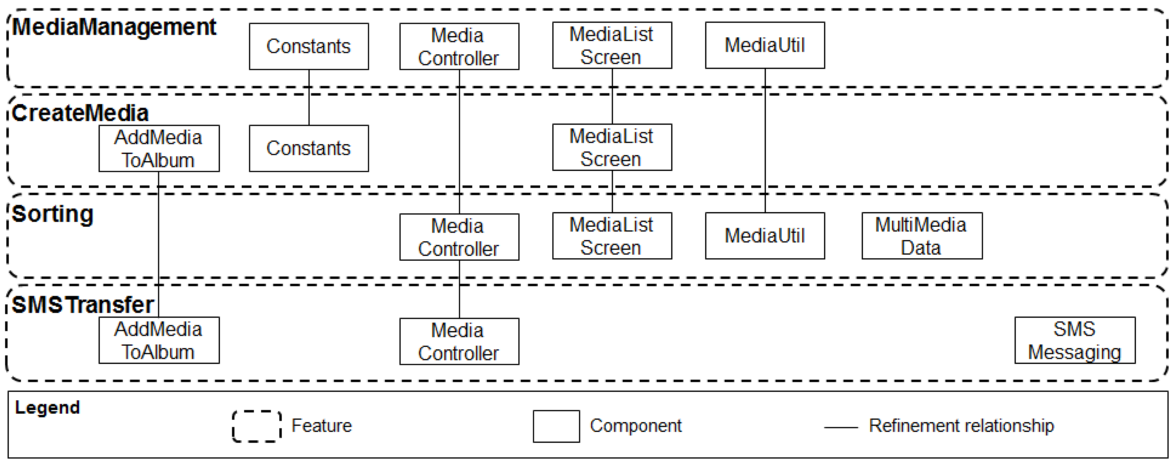


Figure 2.1. Partial Design View of the MobileMedia SPL

Components. In both AHEAD and FeatureHouse, a SPL is implemented through successive refinements, a technique in which complex systems are developed from an original system by incrementally adding functionalities by means of components (Batory et al., 2003). In the context of feature-oriented SPLs, a *component* is a code unit that partially implements the functionality of a feature. Components can be *constants* or *refinements*. A constant corresponds to the basic implementation of a functionality. In turn, a refinement is an implementation that refines a constant by adding or changing functionalities (Batory et al., 2003). A refinement can add attributes and methods to a constant, or override existing methods of the constant. In this study, we assume that a single source file implements exactly a constant or a refinement.

Listing 2.1 illustrates a constant called `ExampleClass` in AHEAD.The constant has m attributes and n methods. Listing 2.2 presents a code that refines the constant `ExampleClass` of Listing 2.1. Such refinement is explicitly declared in the source code via the *refines* keyword (line 1). Note that, in this example, the refinement adds one attribute and one method to the original implementation of `ExampleClass`.

Listing 2.1. Code Example of Component in AHEAD

```
public class ExampleClass {
  private Type attribute_1;
  ...
  private Type attribute_m;
  public Type method_1() { ... }
  ...
  public Type method_n() { ... }
}
```

Listing 2.2. Code Example of Refinement in AHEAD

```
public refines class ExampleClass {
  private Type attribute_m+1;
  public Type method_n+1() { ... }
}
```

Figure 2.1 depicts 14 different components, represented by boxes, distributed along the four features. Lines connecting boxes indicate a refinement relationship, in which there is a constant in the topmost feature and refinements in the features below. Since constants are the basis for refinements, a refinement relationship imposes a hierarchy between features. As an example, *SMSTransfer* has one constant, `SMSMessaging`, and two refinements, namely `AddMediaToAlbum` and `MediaController`. Therefore, the *SMSTransfer* feature is in the topmost hierarchical level for one component, i.e., `SMSMessaging`, but not for the other two components.

Refinement Chains and Feature Hierarchies. In feature-oriented SPLs, a constant can be refined in several features. Successive refinements of the same constant define a refinement chain. When generating a SPL product, only the bottom-most refinement of the chain is instantiated, because it implements all capabilities assigned to the respective chain (Batory et al., 2003). In this dissertation, we also refer to refinement chains as feature hierarchies, due to the order of components stablished by refinement relationships. Let us consider Figure 2.1 to illustrate this concept.

In Figure 2.1, there is a refinement chain of `MediaController` that cuts across three features: *MediaManagement*, *Sorting*, and *SMSTransfer*. Therefore, this refinement chain encapsulates code fragments of these features. As an example, by including the three features in a SPL product, the `MediaController` class resulting from this feature composition includes code of all three features. Nevertheless, if the *Sorting* or *SMSTransfer* features are not included in the SPL product, their respective refinements – i.e., the parts of code implemented into the not included features – will not compose the `MediaController` class. That is, only the resources implemented in the constant will be included in the code of `MediaController`.

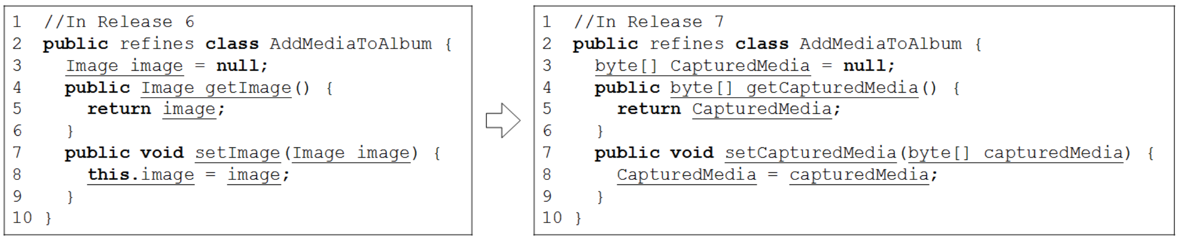
2.2 SPL Maintenance Problems and Instability

Software maintainability refers to how easy is to fix errors in a system when they occur, or to evolve the system due to requirements changes (Schneidewind, 1987). Previous work report that most of the development costs are related to maintenance tasks (Bennett and Rajlich, 2000; Yau and Collofello, 1985). There are different sources of problems that harm software maintainability, from human factors such as the expertise of developers code (Anquetil et al., 2007) to technical aspects of the system as code readability, modularization, and the instability of software components (Ampatzoglou et al., 2015; Baggen et al., 2012; Bennett and Rajlich, 2000).

To provide the assessment of maintainability in software systems, different techniques have been applied (Baggen et al., 2012; Yamashita and Moonen, 2013a). Some examples of techniques are the analysis of software metrics (Baggen et al., 2012) and the identification of anomalous code structures (Yamashita and Moonen, 2013a). We discuss below, in detail, a specific problem that harm maintainability, called instability.

Instability. Instability is the probability of a software system to change, due to changes performed in different parts of the source code (Ampatzoglou et al., 2015). A previous work (Yau and Collofello, 1985) states that most of the maintenance efforts are spent by developers in changing source code. Consequently, instability is significantly detrimental to the SPL maintainability. Stability is even more important for SPL than for single (i.e., non-configurable) software systems, since changes applied to one feature can propagate to other features and affect seemly-unrelated configurations of a product line (Conejero et al., 2009).

Figure 2.2 depicts instability in the context of this dissertation. In this figure, there are two parts of code extracted from the `AddMediaToAlbum` component of the *SMSTransfer* feature, in Releases 6 and 7 of MobileMedia. As the name suggests, this component is responsible for adding media to an album. Earlier, the whle SPL aimed at handling with photo only. Later, though, it has evolved to handle with additional types of media, such as music and video. Several component changed because of that evolution, including `AddMediaToAlbum`. Although this scenario reflects an evolution in the SPL requirements, the component change could have been avoided if MobileMedia supported a fully reusable structure for different media. Consequently, we assume that this change causes an instability that harms the SPL maintainability.



```
//In Release 6
public refines class AddMediaToAlbum {
  Image image = null;
  public Image getImage() {
    return image;
  }
  public void setImage(Image image) {
    this.image = image;
  }
}
```

```
//In Release 7
public refines class AddMediaToAlbum {
  byte[] CapturedMedia = null;
  public byte[] getCapturedMedia() {
    return CapturedMedia;
  }
  public void setCapturedMedia(byte[] capturedMedia) {
    CapturedMedia = capturedMedia;
  }
}
```

Figure 2.2. Example of Instability in MobileMedia

2.3 Final Remarks

This chapter presented the basic concepts of feature-oriented product lines and SPL maintenance problems with focus on instability. Our main goal with this chapter was to provide sufficient information to support the comprehension of the study proposed in this dissertation. First, we presented the main compositional elements of feature-oriented SPLs, namely *components*, *features*, and *feature hierarchies*. Second, we discusses the impact of instability on the SPL maintainability.

After understanding the implementation concepts of feature-oriented SPLs, we discuss in details another relevant topic for understanding this dissertation, i.e., the detection of code anomalies. In the next chapter, we present three studies that compose this dissertation aimed at investigating the state-of-the-art on anomaly detection. First, we discuss a systematic literature review on code anomaly detection tools. Second, we present a comparative study of detection tools. Third, we present a comparative study of detection strategies for code anomalies.

Chapter 3

Code Anomaly Detection

Code anomalies are symptoms of deeper problems in the source code of a software system (Fowler, 1999). These anomalies may occur in different elements of the system, such as components and methods. Moreover, the use of variability mechanisms in a SPL can introduce code anomalies, such as *Inter-Feature Code Clones* and *Long Refinement Chain* (Fenske and Schulze, 2015). As an example, a *Long Refinement Chain* occurs when a method has too many successive refinements in different features. It harms the SPL maintainability because it makes harder to understand the side effects caused by changing a feature with respect to the whole product line (Fenske and Schulze, 2015).

There are two main approaches aimed at detecting anomalies: manual and automated detection (Moha et al., 2010). In the manual detection, developers rely on abstract definitions of types of code anomalies to identify the anomalous code structures that harm the maintenance of a system (Munro, 2005). Fowler (1999) summarizes 22 different anomalies for object-oriented systems, such as *Large Class*, *Feature Envy*, and *Shotgun Surgery*. The author characterizes each anomaly focused on the code aspects that suggest the occurrence of the anomaly. For instance, *Large Class* is characterized by an overload of knowledge and responsibilities in a class, e.g., in terms of its number of attributes and methods. Alternatively to the manual detection, two techniques support the automated anomaly detection, i.e., detection strategies and tools (Fard and Mesbah, 2013; Moha et al., 2010; Vidal et al., 2016).

In this chapter, we present our rely on our preliminary studies to discuss both techniques: detection strategies and tools. Section 3.1 discusses a systematic literature review on code anomaly detection tools. Section 3.2 presents a comparative study of the tools identified in the literature review. Section 3.3 discusses the findings of a comparative study of detection strategies for two anomalies that harm the SPL maintainability: *Large Class* and *Long Method* (Fowler, 1999).

3.1 Literature Review on Detection Tools

Software maintenance and evolution are expensive activities and may represent up to 75% of the software development costs (Liu et al., 2012). One reason for this fact is that the development efforts focus on addition of new functionality or bug correction rather than on design maintainability improvement (Dig et al., 2007). Code anomalies are an important factor affecting the quality and maintainability of a software system (Hall et al., 2014). A code anomaly is any symptom of quality problem in a software system (Fowler, 1999). For instance, *Duplicated Code* occurs when two distinct parts of code implement the same functionality (Fowler, 1999). It may harm the maintainability of a system (Bellon et al., 2007).

Code anomalies can be detected in source code by either using manual or automated analyses (Moha et al., 2010). Tools support automated analysis usually relying on different detection strategies, such as metric-based strategies (Tsantalis et al., 2008). Since there are many code anomaly detection tools proposed in the literature, it is hard to enumerate them and say what code anomalies they are able to detect. Additionally, many tools are restricted to detect code anomalies in specific programming languages. Therefore, by providing a coverage study, we can catalogue which anomalies are detected in each programming language, for instance.

Previous work (Fontana et al., 2012; Moha et al., 2010) investigate the impact of code anomalies on the software quality and maintainability. For instance, Fontana et al. (2012) present a literature review covering seven code anomaly detection tools and evaluate four of these tools in terms of their detection results. Similarly, Moha et al. (2010) compares different detection tools. The authors compute *recall*, i.e., the rate of anomaly detections, and *precision*, i.e., the rate of correct anomaly detections (Fawcett, 2006) for these tools. However, none of these studies provides an extensive overview of the research topic with a comparative study of detection tools.

In the first part of this dissertation (Fernandes et al., 2016), we provide a systematic literature review (SLR) (Kitchenham and Charters, 2007) on code anomaly detection tools. For each tool, we present its main features, such as its developed programming language, compatible languages for anomaly detection, supported code anomalies, and other relevant information. We assume that some developers may find it difficult to choose the most appropriate tool according to their needs. Moreover, some available tools may be using redundant strategies for detection of code anomalies. To assess both assumptions, we provide an overview of the state of the art regarding code anomaly detection tools. With this study, we aim to support developers in finding tools, and researchers in identifying opportunities for future work.

Literature Review Protocol. There are several alternative terms for code anomaly in the literature, such as bad smell, code smell, and design anomaly. Aimed at identifying the largest set of studies that propose or use code anomaly detection tools, we designed the search string presented in Table 3.1. Note that we used the "*" symbol to represent multiple terms that derive from a given prefix. As an example, the terms *tools* and *tooling* are valid derivations of *tool**.

Table 3.1. Search String Designed for the Literature Review

(*tool** AND (*bad smell** OR *design smell** OR *code smell** OR *architecture smell** OR *design anomaly** OR *code anomaly**))

Table 3.2 summarizes the inclusion and exclusion criteria we applied in our literature review. Aiming to restrict the papers included, we defined four inclusion criteria and three exclusion criteria. For instance, as inclusion criteria papers published in the Computer Science area and, as an example of exclusion criteria, papers should be at least two pages long. We decided to include only papers published after 2000 in our study because of the publication of the *Refactoring* book by Fowler (1999) in 1999. This book defines the most well-known code anomalies.

Table 3.2. Inclusion and Exclusion Criteria of the Literature Review

Inclusion Criteria	Exclusion Criteria
Papers published in Computer Science	Papers published before 2000
Papers written in English	Papers shorter than two pages
Papers available in electronic format	Websites, leaflets, and grey literature
Propose or use detection tools	

We run the designed search string in July 2015 in six electronic data sources: ACM Digital Library[1], IEEE Xplore[2], Science Direct[3], Scopus[4], Web of Science[5], and Engineering Village[6]. BibTeX and text files (in this case, converted to BibTeX) were imported to a reference management tool. We manually download the BibTeX files of papers from ACM Digital Library, one by one, because it does not support automatic downloading. For the other electronic data sources, we were able to download the BibTeX or text file automatically.

We discuss some of our SLR findings as follows.

[1]http://dl.acm.org/
[2]http://ieeexplore.ieee.org/
[3]http://www.sciencedirect.com/
[4]http://www.scopus.com/
[5]http://webofknowledge.com/
[6]http://www.engineeringvillage.com/

Detection Tools. Through the SLR, we have found 84 code anomaly detection tools proposed or used in research papers. 29 out of the 84 tools are available online for download. Table 3.3 lists the 29 downloadable tools. The complete list of tools with references is available in Fernandes et al. (2016).

Table 3.3. Detection Tools Available Online for Download

Borland Together (Yamashita and Moonen, 2013b), CCFinder/CCFinderX (Kamiya et al., 2002), Checkstyle (Fontana et al., 2013), Clone Digger (Bulychev and Minea, 2008), Code Bad Smell Detector (Hall et al., 2014), Colligens (Medeiros, 2014), ConcernReCS (Alves et al., 2012), ConQAT (Deissenboeck et al., 2005), DECKARD (Jiang et al., 2007), DuDe (Wettel and Marinescu, 2005), Gendarme (Parnin et al., 2008), inCode (?), inFusion (Fontana et al., 2013), IntelliJ IDEA (Fontana et al., 2015), iPlasma (Marinescu et al., 2005), Java Clone Detector (JCD) (Juergens et al., 2009), jCosmo (Van Emden and Moonen, 2002), JDeodorant (Tsantalis et al., 2008), NiCad (Cordy and Roy, 2011), NosePrints (Parnin et al., 2008), PMD (Copeland, 2005), PoSDef (Chaudron et al., 2014), SDMetrics[7], SpIRIT/JSpIRIT (Vidal et al., 2015), Stench Blossom (Murphy-Hill and Black, 2010), SYMake (Tamrawi et al., 2012), TrueRefactor (Griffith et al., 2011), Understand (Singh et al., 2014), Wrangler (Li and Thompson, 2010)

Figure 3.1 presents the number of tools that aim to detect code anomalies in some of the most popular programming languages. We presented data for nine of the top-ten most popular programming languages based on the IEEE Spectrum[8] ranking. Only for the R language, the sixth most used programming language worldwide, we did not find a code anomaly detection tool. This ranking relies on data from different sources, such as GitHub, Google, and Stack Overflow. Although nine of the 10 most popular languages have at least one detection tool, there is a concentration of proposed tools for only three languages: Java, C, and C++. Moreover, languages with increasing popularity[9], such as PHP and JavaScript, have few compatible tools. These findings point out to a research opportunity in less explored languages.

Code Anomalies. We have found that the tools aim to detect 61 different anomalies. 20 out of the 61 anomalies are listed by Fowler (1999). Figure 3.1 presents the top-ten most frequent code anomalies detected by the tools. Overall, *Duplicated Code*, *Large Class*, and *Long Method* are the anomalies that most of the tools aim to detect.

[8]http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages
[9]http://spectrum.ieee.org/computing/software/the-2016-top-programming-languages

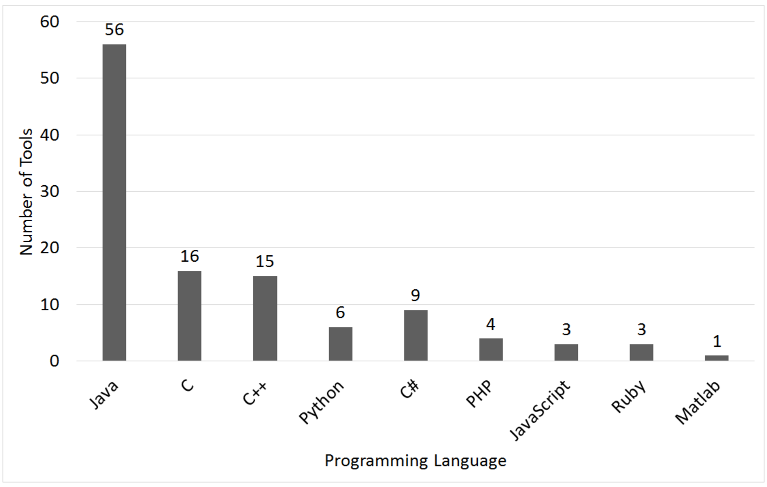


Figure 3.1. Number of Tools per Analyzed Programming Language

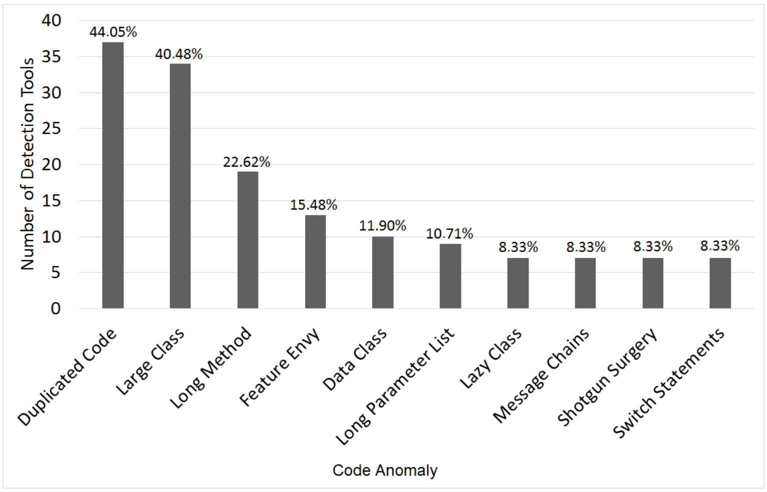


Figure 3.2. Percentage of Tools per Detected Code Anomaly

3.2 Comparative Study of Detection Tools

Also in the first part of this dissertation (Fernandes et al., 2016), we conducted a comparative study of detection tools for two well-known code anomalies: *Large Class* and *Long Method* (Fowler, 1999). The comparison of tools aimed at assessing agreement, recall, precision, and applicability of the tools. Thus, we aimed at understanding the actual support of tools to detect single code anomalies. For this purpose, we selected four detection tools, namely inFusion (Fontana et al., 2013), JDeodorant (Tsantalis et al., 2008), JSpIRIT (Vidal et al., 2015), and PMD (Copeland, 2005). These four tools are free for use, designed to detect the two mentioned code anomalies, and they we extracted from the 29 tools available online for download and identified in our SLR.

Tool Selection Protocol. The tool selection process was conducted as follows. First, we chose the Java programming language to study, since it is the most common language tools analyze, as shown in Figure 3.1. In addition, we restricted the set of tools to include only tools that are free for use, at least in a trial version. After applying these criteria to the tools listed in Table 3.3, we end up with eight tool, namely Checkstyle, inFusion, iPlasma, JDeodorant, PMD, JSpIRIT, Stench Blossom, and TrueRefactor. However, Checkstyle, iPlasma, TrueRefactor, and Stench Blossom have been later discarded by different reasons, as discussed below.

In our study, we discarded Checkstyle because it was not able to detect any instance of the studied code anomalies in the selected software systems. We also discard iPlasma because we could not run the tool properly. In addition, we are aware that the same research group developed both iPlasma and inFusion. Therefore, these tools probably follow similar detection strategies. We have not used TrueRefactor in this comparative study because it does not provide an executable file in the package we downloaded. Finally, we also discarded Stench Blossom because it lacks a code anomaly occurrence list, since it is a visualization tool with no listing feature. Thus, it is difficult to validate the results by computing recall, precision, and agreement.

Agreement, Recall, and Precision of the Tools. In this study, we analyzed two software systems: MobileMedia (Figueiredo et al., 2008), Release 9 (object-oriented version), and JUnit[10], Release 4. JUnit is an open source Java testing framework and MobileMedia is a SPL for applications that manage media on mobile devices. In the case of MobileMedia, we used reference lists with 7 *Large Classes* and 6 *Long Methods* provided by experts in the system. Thus, we were able to compute recall and precision. However, in the case of JUnit, we did not have reference lists of code anomalies. Consequently, we limited our analysis to the agreement computation.

To assess agreement of the detection tools, we compute the AC1 statistic coefficient (Gwet, 2014). It computes a value between 0 and 1, and reports the level of agreement using a scale that ranges from *Poor*, i.e., agreement smaller than 0.2, to *Very Good*, i.e., agreement higher than 0.8 (Altman, 1991). As a result, we identified a high agreement (in general, equals *Very Good*) among the tools with respect to the detection results for both MobileMedia and JUnit, although JDeodorant provided a larger list of code anomaly instances in comparison with the other tools. Therefore, we concluded that the tools provided redundant detection results.

We also computed precision and recall of the tools for MobileMedia, in terms of true positives (TP), false positives (FP), true negatives (TN), and false negatives

[10]https://github.com/junit-team/junit

(FN) (Fawcett, 2006). TP is the number of code elements identified as anomalous that are actually affected by an anomaly. FP is the number of code elements identified as anomalous that are not affected by an anomaly. TN is the number code elements not identified as anomalous that are actually not affected by an anomaly. Finally, FN is the number of code elements not identified as anomalous but affected by an anomaly. The formulas for precision (P) and recall (R) are illustrated in Equations 3.1 and 3.2.

$$P = \frac{TP}{TP + FP} \tag{3.1}$$

$$R = \frac{TP}{TP + FN} \tag{3.2}$$

As a result, we observed low-to-medium rates of recall for the tools, in addition to high rates of precision. For instance, the tools provided up to 67% of recall, in the case of JSpRIT for *Long Method*, but the four tools achieved very low precision rates (14%) in the case of *Large Class*. On the other hand, the tools generally provided high precision, with values higher than 50% (up to 100%) except in the case of JDeodorant, for both *Large Class* and *Long Method*.

Applicability of the Tools. We observed that some desired features are missing in the four tools of our comparative study. These features, if available, could improve the applicability, i.e., the user experience, of the tools. Table 3.4 list the five desired features per detection tool. We assigned an "X" to the tools that support each feature. For instance, only inFusion provides all five features, although two of the features are available only in the full and paid version of the tool. In addition, with respect to the *Highlight Anomaly Occurrences* feature, we expected that all tool highlight the location of each detected anomaly in the source code. All four detection tools somehow support this feature. Finally, for the *Result Export* feature, we expected that anomaly detection results were easy to export to an output file from all tools. However, most of the tools do not provide means to export results. In fact, only JDeodorant supports this feature in a free tool version, and inFusion supports the feature in a paid tool version.

Table 3.4. Features Provided per Code Anomaly Detection Tool

Feature	inFusion	JDeodorant	JSpIRIT	PMD
Allow Detection Settings	X	X	X	X
Detected Anomaly Filtering	X (in paid version)	X	-	X
Graph Visualization	X	-	-	-
Highlight Anomaly Occurrences	X	X	X	X
Result Export	X (in paid version)	X	-	-

3.3 Detection Strategies

A *detection strategy* is a composition of metric-based rules that defines when a specific software component has a code anomaly (Lanza and Marinescu, 2006). An anomalous component can be a class, a method, or a package in an object-oriented software system, for instance (Fowler, 1999). Detection strategies aim to support developers in assessing the maintainability of a system by combining software metrics in a more abstract level than the use of metrics isolately (Munro, 2005). To effectively support the code anomaly detection and, consequently, the maintainability of systems, detection strategies have to provide reasonable detection results (Almeida et al., 2004). Different strategies have been proposed for detecting several code anomalies (Abílio et al., 2015; Fenske and Schulze, 2015; Lanza and Marinescu, 2006).

In the second part of this dissertation (Fernandes et al., 2017a), we investigated detection strategies for two well-known code anomalies that may affect any system, including SPL: *Large Class* (Fowler, 1999) and *Long Method* (Lanza and Marinescu, 2006). With this study, we aimed to deeply investigate the literature support in the context of single anomaly detection. Due to the lack of different strategies for SPL-specific code anomalies, e.g., *Long Refinement Chain* (Fenske and Schulze, 2015), we limited our study to the comparison of strategies for general purpose anomalies. Our study analyzed three feature-oriented SPLs of a product-line repository (Vale et al., 2015): Berkeley DB (Rosenmüller et al., 2009), MobileMedia (Figueiredo et al., 2008), and TankWar (Schulze et al., 2010).

Study Protocol and Findings. OFirst, we compared detection strategies of the literature to assess if they are effective in the SPL context. Although we found different strategies for both *Large Class* and *Long Method*, they provided similar detection results. Second, we proposed novel strategies per anomaly and compared them with the existing ones. Table 3.5 presents the novel strategies. We derived the thresholds per metric via Vale's Method (Vale and Figueiredo, 2015). Given a benchmark of software systems, the method computes four thresholds that rely on percentile of the metric values extracted from the systems. As a result, we observed a slight increase of accuracy in our strategies in comparison with the existing ones.

In addition, we discussed the drawbacks of each strategy to provide insights that support new strategies for SPL. For instance, some metrics of length and complexity, such as Number of Parameters (NP) (Lorenz and Kidd, 1994) and McCabe's Cyclomatic Complexity (Cyclo) (McCabe, 1976), may not contribute to the identification of *Long Method* in feature-oriented SPLs. In our study, we considered these metrics as intuitive means to identify methods with excessive responsibilities. However, we observed that the analyzed SPLs generally present very low values for these metrics. Thus, we found difficult to identify *Long Method* instances in the product lines via NP and Cyclo, even though we derived low threshold for both metrics. This fact may related with the fact that feature-oriented SPLs take advantage of refinement chains to increment the functionalities of a method among SPL features. Thus, Each single component tend to be small and have low cyclomatic complexity.

Table 3.5. Novel Detection Strategies for *Large Class* and *Long Method*

Code Anomaly	Description and Impact on SPL
Large Class (Fowler, 1999)	A class with excessive knowledge and responsibilities. In SPL, a Large Class may be the source of an inflated feature that requires a more appropriate decomposition. Moreover, this anomaly may be the source of non-modularized features, suggesting the need of feature extraction
Large Class = (LOC > 77) AND (NOA > 4) AND (NOM > 10) AND (WMC > 17)	
Long Method (Fowler, 1999)	A method with too many responsibilities. Although this code anomaly is not exclusive to SPL, it can indicate problems if the different responsibilities relate to different features, for instance
Long Method = (MLOC > 13) AND (NP > 2) AND (Cyclo > 3)	

3.4 Final Remarks

This chapter presented the first and second parts of this dissertation, in which we studied single code anomaly detection. First, we presented a literature review on detection tools (Fernandes et al., 2016). We have found 84 tools for several programming languages, such as Java, C, and C++. We also presented a comparative study of tools. As a result, we found evidence that the available tools tend to provide similar detection results. Second, we discussed the findings of comparative study of detection strategies. Our comparison focused on two code anomalies that harm the SPL maintainability: *Large Class* and *Long Method* (Fowler, 1999). In general, the detection strategies also provided similar results for the same system, as observed in the case of detection tools.

After discussing the current techniques for code anomaly detection, i.e., detection tools and strategies, we characterize types of anomaly agglomerations that support the identification of SPL maintenance problems. In the next chapter, we propose three novel types of code anomaly agglomerations in the SPL context. For each type of anomaly agglomerations, we first present an informal definition with an abstract example. Second, we discuss the rationale and applicability of the agglomerations. Third, we formally define the type of anomaly agglomerations. Fourth, we discuss an example extracted from the source code of the MobileMedia SPL.

Chapter 4

Anomaly Agglomeration in SPL

Code anomalies are symptoms of deeper problems in a software system (Fowler, 1999). They make a source code element difficult to understand and, consequently, to maintain. Any software system is prone to have code anomalies (Macia et al., 2012). Even the SPL variability can introduce anomalies, e.g., because of feature interactions (Fenske and Schulze, 2015). As an example, *Long Refinement Chain* occurs when a method has too many successive refinements in different features of the product line. It harms the SPL maintainability because it makes harder to understand the side effects caused by changing the implementation of a feature or selecting a different set of features to compose a specific SPL product (Fenske and Schulze, 2015).

In general, code anomalies provide hints of maintenance problems in a software system (Fowler, 1999). However, each single anomaly may represent only a partial view of the problem. Due to the scattering of many maintenance problems throughout the source code, it is difficult to completely understand them (Moha et al., 2010). Thus, to provide a wider and more complete view of problems in a SPL, we propose three novel types of anomaly agglomerations. The proposed agglomerations rely on the main structural elements of feature-oriented SPLs, i.e., *features*, *feature hierarchies*, and *components* (Batory et al., 2003). We based our propositions on a previous work (Oizumi et al., 2016) that proposes anomaly agglomerations for object-oriented systems.

The remainder of this chapter is organized as follows. Section 4.1 proposes *feature agglomerations.* Section 4.2 proposes *feature hierarchy agglomerations.* Section 4.3 proposes *component agglomerations.* For each type of anomaly agglomerations, we present an informal definition with an abstract example, the rationale for proposing the agglomerations, and a formal definition with a real example extracted of Mobile-Media (Figueiredo et al., 2008), a SPL for media management in mobile devices.

4.1 Feature Agglomeration

This section proposes the *feature agglomeration.* A feature agglomeration is a group, with minimum length equals 2, formed by all anomalous components of a single feature. In other words, each component affected by at least one code anomaly, e.g., *Long Method* (Fowler, 1999), *Divergent Change* (Lanza and Marinescu, 2006), or *Long Refinement Chain* (Fenske and Schulze, 2015), contributes to form a feature agglomeration in the feature that it is located. Figure 4.1 illustrates an abstract example of feature agglomeration, composed by the boxes colored in gray and with bold font.

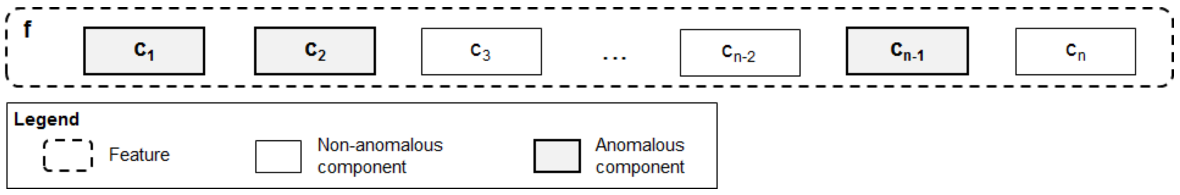


Figure 4.1. Abstract Example of Feature Agglomeration

There is a simple reason for considering a feature as a natural grouping of code anomalies, i.e., Feature-Oriented Programming (FOP) expects that developers implement all components related to a specific SPL functionality into the same feature (Apel et al., 2013; Batory et al., 2003). Although there might be no explicit, syntactic relationship among components of the same feature, they are typically located in the same folder at the SPL source code. Thus, grouping components by feature reflects the semantic relationship among components in feature-oriented SPLs.

With our definition of feature agglomeration, we hypothesize that the occurrence of different anomalies in components of the same feature are indicators of SPL maintenance problems. In other words, we consider anomalies from different components as a single anomalous structure at the feature-level. We expect that this wider view of anomalies may better indicate problems that harm the SPL maintainability. We formally define feature agglomeration as follows. In summary, if there are two or more anomalous components c in a feature f (i.e., $c \rightarrow f$), there is a feature agglomeration.

Definition 1.
Let f be a feature and c be an anomalous component.
Let $c \rightarrow f$ when an anomalous component c contributes to implement f.
A *feature agglomeration* of f is a set of anomalous components C in which there exists a relation $c \rightarrow f$, $\forall c \in C$ and $|C| \geq 2$.

We discuss an example of feature agglomeration that occurs in the *SMSTransfer* feature of MobileMedia (Figueiredo et al., 2008) as follows.

SMSTransfer. In Figure 4.2, the feature *SMSTransfer* has three components, namely `AddMediaToAlbum`, `MediaController`, and `SMSMessaging`. Only the two last components are anomalous, with one and three anomalies respectively. *Long Parameter List* affects `MediaController`. This anomaly in isolation provides only a limited view of SPL maintenance problems in the level of methods. In turn, `SMSMessaging` has *Large Class*, *Long Method*, and *Shotgun Surgery*. Such anomalies, by themselves, provide interesting hints of problems in the anomalous component. For instance, *Large Class* and *Long Method* suggest an overload of responsibilities in the component.

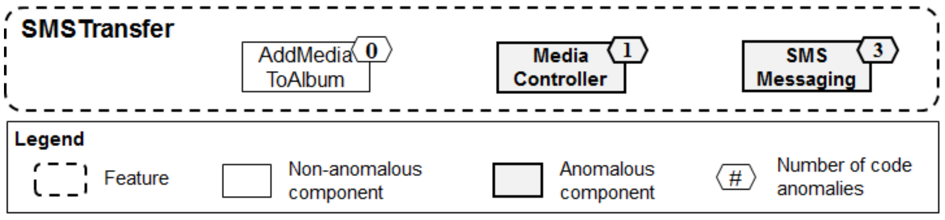


Figure 4.2. Feature Agglomeration of SMSTransfer

However, we expect to have a better comprehension of problems in the SPL by agglomerating these anomalies. If we consider the feature agglomeration formed in *SMSTransfer*, we may draw wider observations. While the instance of *Large Class* suggests a need of delegating the excessive responsibilities of `SMSMessaging` to other classes, we have *Long Parameter List* in `MediaController`. Let us assume that (i) components of a single feature interact to provide a SPL functionality and (ii) `MediaController` uses the overloaded component `SMSMessaging`. Thus, there is a possibility of changes in the `SMSMessaging` component that impact on the stability of the `MediaController` component. As a conclusion, we see a potential of feature agglomerations to improve our observation regarding the SPL maintenance problems in the level of features.

Summary of Feature Agglomeration. In general, we expect that *feature agglomeration* supports effectively the developers to understand the SPL maintenance problems that affect multiple source files, in the same feature, that implement together interrelated product-line functionalities.

4.2 Feature Hierarchy Agglomeration

This section proposes the *feature hierarchy agglomeration*. A feature hierarchy agglomeration is a set, with minimum length equals 2, composed by all anomalous components of a single feature hierarchy. In other words, each components with one or more code anomalies contributes to form a feature hierarchy agglomeration in the refinement chain it belongs to. Figure 4.3 provides an abstract example of feature hierarchy agglomeration, formed by the boxes in gray and with bold font.

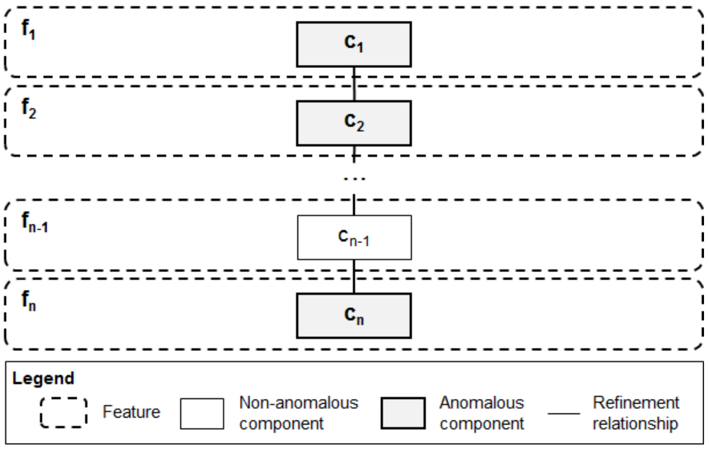


Figure 4.3. Abstract Example of Feature Hierarchy Agglomeration

In FOP, each component is a constant or a refinement (Apel et al., 2013; Fenske and Schulze, 2015). In order to refine the basic functionalities of a constant, i.e., to add or change the resources of the constant, a refinement defines an explicit inter-component, hierarchical relationship called refinement chain (Batory et al., 2003). Such relationship between the refined constant and the refinement is defined syntactically via the keyword `refines`. Thus, grouping components by refinement chain reflects the syntactic relationship among components in a FOP-based product line.

Extracted from MobileMedia, Listings 4.1 and 4.2 illustrate refinements in AHEAD. In Listing 4.1, the constant `PhotoViewController` of feature *MediaManagement* implements a controller with one method (`initCommandsMap`) and one attribute (`commands`). In Listing 4.2, the keyword `refines` into the refinement of feature *CopyMedia* indicates that `PhotoViewController` in feature *CopyMedia* refines the previously defined constant. In this example, the refinement extends the behavior of `initCommandsMap` by adding two new commands to the map. The original method is called in the refinement chain through the keyword `Super`. The base code and different feature modules are composed via the AHEAD tool suite (Apel et al., 2011) or the FeatureHouse composer (Apel et al., 2009), depending on the adopted technique.

Listing 4.1. Code Sample Extracted of Feature MediaManagement

```
public class PhotoViewController {
  protected Map commands;
  public void initCommandsMap() {
    commands = new HashMap();
  }
  ...
}
```

Listing 4.2. Code Sample Extracted of Feature CopyMedia

```
public refines class PhotoViewController {
  public void initCommandsMap() {
    Super().initCommandsMap();
    commands.put("Copy", new CopyPhoto());
    commands.put("Save␣Photo", new SaveCopiedPhoto());
  }
}
```

By defining feature hierarchy agglomeration, we assume that anomalies affecting different components of the same refinement chain are indicators of SPL maintenance problems. This assumption relies on the hierarchical dependency among components that inter-relate explicitly via refinements in the SPL feature model. We expect that the wide view provided by the analysis of an entire refinement chain provides better indicators of SPL maintenance problems, when compared with the analysis of individual components of the SPL. A formal definition of feature hierarchy agglomeration is presented as follows. In summary, if there are two or more anomalous components c in a refinement chain r (i.e., $c \to r$), there is a feature hierarchy agglomeration.

Definition 2.
Let r be a refinement chain and c be an anomalous component.
Let $c \to r$ when an anomalous component c belongs to r.
A *feature hierarchy agglomeration* of r is a set of anomalous components C in which there exists a relation $c \to r$, $\forall c \in C$ and $|C| \geq 2$.

We describe an example of feature hierarchy agglomerations that occurs in the `MediaUtil` feature hierarchy of MobileMedia as follows.

MediaUtil. In Figure 4.4, two components `MediaUtil` form a refinement chain that cuts through the features *MediaManagement* and *Sorting. Long Parameter List* affects both components. In addition, the component of the *MediaManagement* feature has

Large Class and *Long Method*. By analyzing each anomalous component, these anomalies indicate the occurrence of one or more maintenance problems that affect parts of the implementation of `MediaUtil`. However, the analysis of agglomerated components may improve this view of problems in the entire SPL.

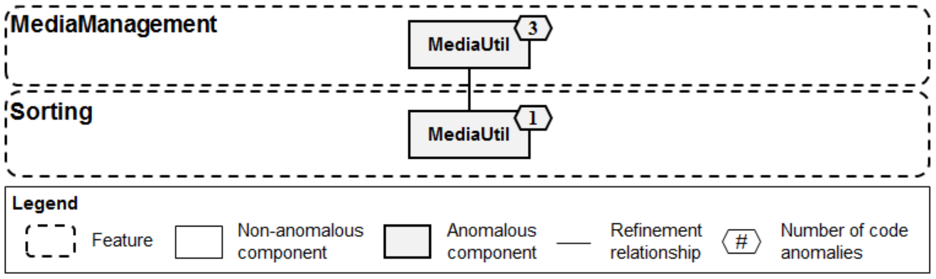


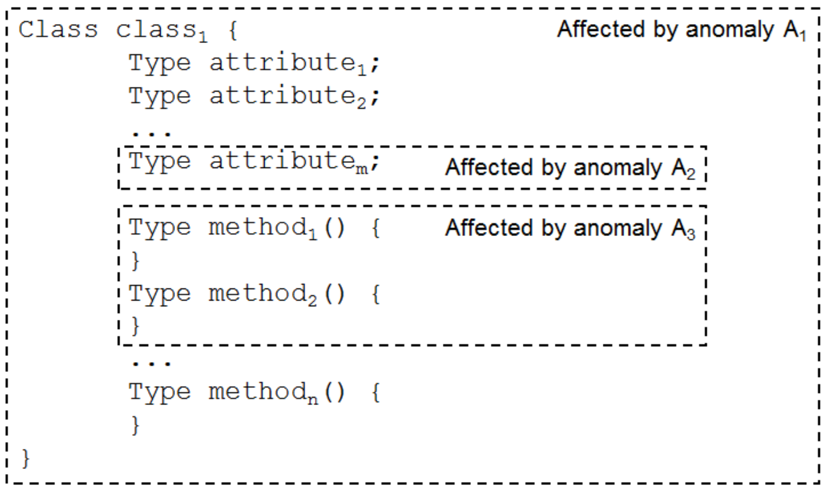
Figure 4.4. Feature Hierarchy Agglomeration of MediaUtil

In fact, the occurrence of *Large Class* and *Long Method* in a constant has impacts that we better understand when considering the entire refinement chain. In this case, the constant `MediaUtil` has one refinement below it in the feature hierarchy of the SPL. Thus, both anomalies may cause major maintenance problems in the SPL, since a product derived from the SPL that includes the features *MediaManagement* and *Sorting* will inherit all anomalies of the constant. In this context, the feature hierarchy agglomeration indicates that both *Large Class* and *Long Method* in `MediaUtil` are even more critical than we may assume based on the individual component analysis.

Summary of Feature Hierarchy Agglomeration. We assume that the impact of anomalies agglomerated by feature hierarchy is wider than we may observe via the analysis of individual SPL components. Thus, *feature hierarchy agglomeration* aims to indicate problems that affect a scattered concern associated with multiple features.

4.3 Component Agglomeration

In this section, we propose the *component agglomeration*. A component agglomeration is a set with two or more anomalous code elements of a component such that the set is affected by at least two types of code anomalies. For instance, a class with a *Large Class* (Fowler, 1999), in addition to methods affected by *Long Method* (Lanza and Marinescu, 2006), contributes to form a component agglomeration in the SPL component it is implemented. The same applies to the anomalous methods. Figure 4.5 presents an abstract example of component agglomeration.



```
Class class1 {
        Type attribute1;
        Type attribute2;
        ...
        Type attributem;
        Type method1() {
        }
        Type method2() {
        }
        ...
        Type methodn() {
        }
}
```

Affected by anomaly A_1
Affected by anomaly A_2
Affected by anomaly A_3

Figure 4.5. Abstract Example of Component Agglomeration

Components in FOP are code units that implement part of a SPL funcionality (Batory et al., 2003). Thus, we considered the code level of components as an intuitive way of grouping code elements that implement part of the product line. Such code elements, i.e., classes and methods, inter-relate explicitly by composing a single source file. Although this type of agglomeration is not specific to SPL, we defined component agglomeration as a mechanism to identify wider SPL maintenance problems that the analysis of single code anomalies are not able to support.

We provide a formal definition of component agglomeration as follows. In summary, if there are two or more anomalous code elements e in a component c (i.e., $e \rightarrow c$), with at least two different types of code anomaly affecting such components, there is a component agglomeration.

Definition 3.
Let c be a component and e be a code element.
Let $e \rightarrow c$ when a code element e belongs to c.
A *component agglomeration* of c is a set of anomalous code elements E when there exists a relation $e \rightarrow c \; \forall e \in E$, $|E| \geq 2$, and at least two types of code anomalies affect E.

We discuss an example of component agglomerations in the `MediaListController` component of MobileMedia as follows.

MediaListController. The code anomalies that affect the component `MediaListController` are, namely, *Long Method*, *Long Parameter List*, and *Long Refinement Chain*. The occurrence of the three anomalies in `MediaListController` represents, as a whole, a major threat to the component stability. This threat is mostly related to the several changes that the three anomalies may require to be fixed in the source code. Therefore, the component agglomeration has an important role in the identification of major maintenance problems.

Summary for Component Agglomeration. We expect that, by defining *component agglomeration*, we may support the identification of major SPL maintenance problems in a component caused by inter-related code anomalies.

4.4 Final Remarks

This chapter proposed three novel types of code anomaly agglomerations in the context of feature-oriented SPLs. For each type of agglomeration, we provided a formal definition, an abstract example, and a real example collected from the source code of MobileMedia (Figueiredo et al., 2008). Such examples aimed to motivate the use of agglomerated code anomalies in SPL for identifying major problems that harm the SPL maintainability. We expect that agglomerations provide a wider view of the problem extent in comparison with non-agglomerated code anomalies.

The examples provided in this chapter suggest the applicability of our types of code anomaly agglomeration in the identification of SPL maintenance problems. However, an evaluation of each type of agglomeration is required for us to draw further conclusion on the effectiveness of agglomerating anomalies in SPL. Thus, the next chapter presents an empirical study to evaluate the three proposed types of anomaly agglomerations. We describe the study settings, present and discuss the results.

Chapter 5

Evaluation of SPL Stability

In this dissertation, we propose three novel types of code anomaly agglomeration in SPL: *feature agglomeration*, *feature hierarchy agglomeration*, and *component agglomeration*. We rely on a previous work (Oizumi et al., 2016) that proposes agglomerations for object-oriented systems. By taking into account the main structural elements of feature-oriented SPLs, we aim to support a wider analysis of SPL maintenance problems than provided by the analysis of individual, non-agglomerated code anomalies. For each type of anomaly agglomerations, we provide illustrative examples extracted from the MobileMedia SPL (Figueiredo et al., 2008). However, we have to evaluate the effectiveness of anomaly agglomerations in identifying SPL maintenance problems.

For this purpose, we present in this chapter an empirical study to evaluate our novel types of anomaly agglomerations. The remainder of this chapter is organized as follows. Section 5.1 describes the evaluation settings, including research questions, target SPLs, and study protocols. Section 5.2 present the evaluation results for non-agglomerated code anomalies. Section 5.3 presents the results for each type of anomaly agglomeration. Section 5.4 discusses threats to the study validity.

5.1 Evaluation Settings

In this dissertation, we aim to investigate whether non-agglomerated and agglomerated anomalies indicate SPL maintenance problems. Thus, we evaluate our three novel types of anomaly agglomerations, namely *feature*, *feature hierarchy*, and *component agglomeration*. Given the extensive variety of problems that harm the SPL maintainability, we chose instability. We specifically investigate sources of instability in SPL. That is, we are concerned with the factors leading to frequent changes in SPL components. In particular, we investigate to what extent code anomaly agglomerations support the identification of sources of instability in SPLs. In other words, we are concerned about the relationship between agglomerated code anomalies in indicating parts of code that change frequently and, therefore, are instable.

To guide our study, we designed two research questions (RQs). We present and discuss each of them as follows.

RQ1. *Can non-agglomerated code anomalies indicate instability in SPL?*

We did not find previous studies that investigate non-agglomerated anomalies as indicators of instability in SPL. Therefore, with RQ1, we assess if non-agglomerated anomalies can provide instability hints in SPL. In this study, we consider a component as instable in a SPL if it has changed in at least two releases of the SPL. We made this decision based on two observations as follows. First, there are few available releases per SPL under analysis, seven at most. Second, after a manual analysis of the selected SPLs, we observed several components that change only two times but contain various code anomalies. Thus, we determined two as a minimum amount of changes. Note that we do not consider comment-related changes in the count of instability.

RQ2. *Can agglomerated code anomalies indicate instability in SPL?*

RQ2 focuses on the investigation of whether our three types of code anomaly agglomerations can be indicators of instability. We address this question according to two perspectives, i.e., strength (RQ2.1) and accuracy (RQ2.2), discussed as follows.

RQ2.1. *How strong is the relationship between agglomerations and instability?*

To answer RQ2.1, we compute the strength of the relationship between each type of agglomeration and instability. That is, we assess the potential of agglomerated anomalies in indicating instabilities. We say a relationship is strong if agglomerated anomalies identify at least 100% more instabilities than non-agglomerated anomalies. We chose this rounded percentage based on the guidelines of Lanza and Marinescu (2006). With a threshold of 100%, we aim to assure that the agglomerated anomalies actually performed better than non-agglomerated anomalies.

RQ2.2. *How accurate is the relationship between agglomerations and instability?*

To answer RQ2.2, we compute the accuracy of agglomerations in identifying instability, in terms of precision and recall. In other words, we assess if agglomerated anomalies are able to identify instability (i.e., recall) and if the agglomerated anomalies can correctly identify instabilities in feature-oriented SPLs (i.e., precision).

Target SPLs. For this study, we selected four SPLs implemented in AHEAD or FeatureHouse from different domains: MobileMedia (Figueiredo et al., 2008), Notepad (Kim et al., 2010), TankWar (Schulze et al., 2010), and WebStore (Gaia et al., 2014). MobileMedia provides products for media management in mobile devices (Ferreira et al., 2014; Vale et al., 2015). Notepad aims to generate text editors (Vale et al., 2015). TankWar is a war game for personal computers and mobile devices (Schulze et al., 2010). Finally, WebStore derives Web applications with product management (Ferreira et al., 2014; Vale et al., 2015).

Table 5.1 provides general information on the target feature-oriented SPLs. The first and second columns describe the SPL name and respective domain. The third column provides the number of releases available per SPL. The fourth and fifth columns provide the number of features and components for each SPL. We selected these SPLs for some reasons. First, these SPLs are part of a SPL repository proposed in a previous work (Vale et al., 2015). Second, they have been published and investigated in the literature (Ferreira et al., 2014; Schulze et al., 2010). Third, there are different releases for each SPL. This variety of releases allows us to compute instability for the SPLs throughout consecutive releases. Fourth, developers of these SPLs were available for consultation, except in the case of Notepad.

Table 5.1. General Information of the Target SPLs

SPL	Domain	Releases	Features	Components
MobileMedia	Media Management	7	25	141
Notepad	Text Editor	2	13	32
TankWar	Game	7	32	92
WebStore	E-Commerce	6	13	86

According to the developers of the four SPLs, each of them evolved to address different issues. The initial design of MobileMedia supported photo management only. However, MobileMedia evolved to implement management of other media types, such as video and music. This evolution required a revision of the SPL assets (Ferreira et al., 2014). Notepad was completely redesigned in the two available releases (Kim et al., 2010). Developers added new functions and created new features to ease the introduction of new functions and to improve the modularization of features. TankWar evolved only to refactor the SPL without changing any functions but to improve its maintainability. Finally, WebStore initially supported a few payment types and data management options, for instance. As WebStore evolved, it has changed to cover other new functionalities. Although WebStore and MobileMedia have similar evolution scenarios, the initial development of WebStore took into account future planned evolutions to make the SPL more stable (Ferreira et al., 2014).

Study Protocol 1: Identifying Sources of Instabilities. We first computed instability per SPL, based on the number of changes per component between releases. For this purpose, we count an instability index if the component changes between consecutive releases. In this study, we used the instability computed for MobileMedia and WebStore by a previous work Ferreira et al. (2014). To increase the data reliability, and to compute instability for TankWar and Notepad, we used the WinMerge tool[1]. The tools provides a visual comparison of source files from two different systems.

After, we identified the main sources of instability per SPL, based on the changed components. After, we computed the sources of instabilities, i.e., the reasons that lead to instability per component. We aimed at identifying groups of SPL components with similar sources of instability. As an example of source, we have the addition of a new feature the SPL that may affect the implementation of several components. Whenever was possible, we validated the detected instability with developers of the target SPLs by showing them the numbers obtained per component.

Table 5.2 presents the sources of instabilities identified in the four SPLs. The first column indicates the category and the sum of affected components per source. The second column presents the description of each source of instability. The last line (i.e., *Others*) represents the sources of instability that we were not able to categorize. As an example, we named *Add Crosscutting Feature* when a new feature is added to the SPL and it affects the implementation of existing features. This particular instability is interesting in SPL because, according to the Open/Closed Principle, software entities should be open for extension, but closed for modification (Meyer, 1988).

Table 5.2. Sources of Instabilities in SPL

Source	Description	# of Affected Components
Add Crosscutting Feature	When we add a new feature to the SPL and, consequently, the new functionalities are of interest of components from several existing features. Many components from different features change	122
Change from Mandatory to Optional	When we distribute the implementation of an existing feature to: (i) a new, basic mandatory feature, and (ii) a new optional feature, with specific functionalities	19
Distribute Code among Features	When we extract code parts of a component from an existing feature and, then, distributed these code parts to components from existing features	39
Pull Up Common Feature Code	When we extract code parts that are common into child features to a parent feature above in the feature hierarchy	63
Others	General sources unrelated explicitly to SPL maintenance, e.g., attribute renaming	195

[1]http://winmerge.org/

Study Protocol 2: Identifying Anomalies and Agglomerations. Our process of identifying code anomalies consists in three steps: (i) to define the anomalies for study, (ii) to define the metric-based detection strategies to identify each anomaly, and (iii) to apply the defined detection strategies to each SPL. A detection strategy is a composition of metric-based rules that defines when a specific software component is prone to contain a code anomaly (Lanza and Marinescu, 2006). Table 5.3 lists the 11 software metrics used to compose the strategies used in our study.

Table 5.3. Software Metrics Used to Compose the Detection Strategies

Metric Level	Abbreviation	Metric Name and Reference
Classes	CBO	Coupling between Objects (Chidamber and Kemerer, 1994)
	LOC	Lines of Code (Lorenz and Kidd, 1994)
	NCR	Number of Constant Refinements Abílio et al. (2015)
	NOA	Number of Attributes (Lorenz and Kidd, 1994)
	NOM	Number of Methods (Lorenz and Kidd, 1994)
	WMC	Weighted Methods per Class (Chidamber and Kemerer, 1994)
Methods	Cyclo	McCabe's Cyclomatic Complexity McCabe (1976)
	MLOC	Method Lines of Code (Lorenz and Kidd, 1994)
	NMR	Number of Method Refinements (Abilio et al., 2016)
	NP	Number of Parameters (Lorenz and Kidd, 1994)
	NOOr	Number of Operations Overrides (Miller et al., 1999)

Table 5.4 presents the list of code anomalies that we investigate with the respective detection strategies. Due to the limited set of software metrics available for computation per SPL, we adapted the detection strategies from the literature (Lanza and Marinescu, 2006) whenever possible, per code anomaly. We extracted the metric values per SPL via the VSD tool (Vale et al., 2015). We derived the thresholds per metric via the Vale's Method (Vale and Figueiredo, 2015).

Our analysis relies mostly on general purpose anomalies, except for *Long Refinement Chain* (Fenske and Schulze, 2015), but all of them relate somehow to the SPL composition. These anomalies affect the source code of SPLs in different levels, including feature hierarchies. For instance, *Divergent Change* is a class that changes due to divergent reasons (Lanza and Marinescu, 2006). If these reasons relate to different features, this anomaly may harm the SPL modularization. *Long Method* is a method with too many responsibilities (Fowler, 1999). This anomaly is harmful in SPLs if the responsibilities of the method relate to different features, for instance. Finally, *Long Refinement Chain* (Fenske and Schulze, 2015) is a method with excessive number of successive refinements. This SPL-specific anomaly is harmful since it hampers the understanding of side effects of changes in the generation of SPL products.

Table 5.4. Code Anomalies for Analysis

Code Anomaly	Description and Impact on SPL
Data Class (Fowler, 1999)	A class composed only of attributes, getters, and setters. Its lack of responsibilities suggests that other classes may manipulate the Data Class too much. Since SPL design requires modularization, we should eliminate Data Class *Data Class = (NOA > High) AND (NOM < High)*
Divergent Change (Fowler, 1999)	A class that changes due to multiple, divergent reasons. In SPL, these reasons can relate to different features, indicating problems in the SPL design, for instance *Divergent Change = (NOA > High) AND (CBO > High) AND (NOM > High)*
Large Class (Fowler, 1999)	A class with excessive knowledge and responsibilities. In SPL, a Large Class may be the source of an inflated feature that requires a more appropriate decomposition. Moreover, this anomaly may be the source of non-modularized features, suggesting the need of feature extraction *Large Class = (LOC > High) AND (CBO > Low) AND (WMC > High)*
Lazy Class (Fowler, 1999)	A class with little knowledge and few responsibilities. In SPL, a small portion of lines of code can eventually affect a refinement chain as a whole. Consequently, it may affect negatively many features and SPL functionalities or, sometimes, at least a group of products *Lazy Class = [(LOC < Low) AND (WMC < Low)] OR (CBO < Low)*
Long Method (Fowler, 1999)	A method with too many responsibilities. Although this code anomaly is not exclusive to SPL, it can indicate problems if the different responsibilities relate to different features, for instance *Long Method = (MLOC > High) AND (Cyclo> High)*
Long Parameter List (Fowler, 1999)	A method with an extensive list of parameters. Groups of parameters can be inter-related eventually. In SPL, this relation may suggest the need of new components to gather these parameters, for instance *Long Parameter List = (NP > High)*
Long Refinement Chain (Fenske and Schulze, 2015)	A method with excessive number of successive refinements. This SPL-specific code anomaly harms maintainability when adding a new refinement or changing a method, because it hinders the understanding of side effects of a particular refinement for the generation of a SPL product *Long Refinement Chain = (NMR > High)*
Shotgun Surgery (Fowler, 1999)	A class whose changes affect many other classes. Such classes can belong to different features, for instance. Therefore, the occurrence of Shotgun Surgery in SPL may indicate problems in the SPL design *Shotgun Surgery = (NOA > High) AND (CBO > Low) AND (NOM > High)*

Once detected the anomalies, we computed manually our three novel types of anomaly agglomerations per SPL. Other researchers double-checked the results in order to prevent errors. In case of divergence, the researches re-computed the agglomerations.

Study Protocol 3: Correlating Agglomerations and Instabilities. To answer our research questions, we defined a criterion for correlating agglomerations and instabilities. Consider a general agglomeration that can be either a feature, a feature hierarchy, or a component agglomeration. We say that such agglomeration indicates an instability when there exists an instable code element in the feature, feature hierarchy, or component that have the agglomeration. Even though agglomerations and instabilities may be located in more than two anomalous elements, we consider sufficient if the agglomeration is affected by at least one instability problem. Thus, an agglomeration fails to indicate instability when none of its components relates to an instability.

5.2 Results for Non-Aglomerated Code Anomamies

First, we investigate whether non-agglomerated code anomalies are sufficient indicators of instabilities in SPL. Therefore, we aim to answer RQ1.

RQ1. *Can non-agglomerated code anomalies indicate instability in SPL?*

In this study, we computed the statistical significance of the relation between non-agglomerated anomalies and instabilities via Fisher's exact test (Fisher, 1922). Such test returns a p-value that, given a confidence interval, indicates if the obtained results are statistically significant. In addition, we computed Odds Ratio (Cornfield, 1951) to compute the possibility of the presence or absence of a property (i.e., the anomaly non-agglomeration) to be associated with the presence or absence of other property (i.e., instability). Since Odds Ratio compares the possibilities of presence and absence for each property, we have to compute both the number of anomalies that agglomerated and do not agglomerate, as well as the number of instabilities and stabilities. We computed both statistics via the R tool[2].

Table 5.5 presents the results for non-agglomerated anomalies. The first column lists each feature-oriented SPL. The second column presents the number of non-agglomerated anomalies that indicate instabilities. The third column presents the number of agglomerated anomalies that do not indicate instabilities, i.e., they indicate stability. Finally, the fourth column presents the total number of anomalies per SPL.

Table 5.5. Analysis Results for Non-Agglomerated Anomalies

SPL	**Non-Agglomerated and Instability**	**Agglomerated and Stability**	**Total Number of Code Anomalies**
MobileMedia	1	11	87
Notepad	0	1	24
TankWar	0	2	106
WebStore	0	4	29

By comparing the second and third columns, we observe that for the 4 feature-oriented SPLs the number of non-agglomerated anomalies that indicate instability is very low. In general, this number is even lower than the number of agglomerated

[2]https://cran.r-project.org/

anomalies that indicate stability. Since each SPL has several code anomalies (fourth column), we may assume that anomaly agglomerations are potentially useful for identifying instabilities in SPL. In addition, considering all the four analyzed SPLs, we have a p-value of 0.1488 and Odds Ratio equals 0.0816. Thus, our results suggest that the possibility of a non-agglomerated anomaly to indicate instabilities is close to 0 when compared with an agglomerated anomaly.

Summary for RQ1. Our data suggest that non-agglomerated anomalies may not suffice to indicate instabilities in SPL. The low number of non-agglomerated anomalies that indicate instabilities supports this finding. On the other hand, there is a potential for agglomerations in indicating instabilities.

5.3 Results for Agglomerated Code Anomalies

In this section, we analyze the relationship between agglomerations and instabilities. We aim to answer RQ2 decomposed into RQ2.1 and RQ2.2 discussed as follows.
RQ2.1. *How strong is the relationship between agglomerations and instability?*

Table 5.6 presents the results per type of agglomeration. The first column lists each proposed type of agglomeration. The second column presents the number of anomaly agglomerations that indicate correctly an instability, considering the four SPLs we analyze. The third column presents the number of non-agglomerations that do not indicate instability. We omitted the number of non-agglomerations that indicate instability since we discuss these data previously in Section 5.2. The last two columns present the p-value computed via Fisher's test and the results obtained for Odds Ratio.

Table 5.6. Analysis Results for Agglomerated Anomalies

Type of Agglomeration	Agglomeration and Instability	Non-Agglomeration and Stability	p-value	Odds Ratio
Feature	31	6	1	1.1598
Feature Hierarchy	28	13	0.0478	3.8492
Component	28	124	0.8761	0.9290

Regarding the number of agglomerations that indicate instability in the target SPLs, we observed that an average of 94%, 78%, and 32% of the anomaly agglomerations indicate two or more instable components for feature, feature hierarchy, and component agglomeration, respectively. Note that, for all types of agglomerations, we obtained similar numbers of agglomerations that indicate instability, but the values of non-agglomerations that indicate stability vary according to the type of agglomeration.

Regarding p-value, we assume a confidence level higher than 95%. Only *feature hierarchy agglomerations* presented p-value lower than 0.05 and, therefore, it is the only type of agglomeration with statistical significance with respect to the correlation between anomaly agglomerations and instabilities in SPL.

Regarding Odds Ratio, we have a value significantly greater than 1 only for feature hierarchy agglomerations, around 3.8. It means that the possibility of a feature hierarchy agglomeration to relate with instabilities is almost 4 times higher than a non-agglomerated code anomaly. For the other two types of agglomerations, we have values close to 1 and, therefore, we may not affirm that such types of agglomerations have more possibilities to "host" instabilities when compared to non-agglomerated anomalies. Therefore, with respect to RQ2.1, we conclude that the relationship between agglomerations and instabilities is only strong for feature hierarchy agglomeration. This observation is quite interesting, since in FOP the features encapsulate the implementation of functionalities of the SPL. Besides that, our data suggest the refinement relationship may hinder this encapsulation by causing instability into multiple features. This problem is even more critical since the instabilities caused by a feature hierarchy agglomeration can eventually propagate to several seemly-unrelated SPL products.

We highlight that the relationship computed via Odds Ratio, i.e., the correlation between anomaly agglomerations and instability, does not imply in a cause-effect relationship. In other words, we may not affirm that the agglomerations are the cause of instability, but that there is a high proportion of agglomeration that indicate instability.

We also investigate the accuracy of anomaly agglomerations in indicating instability in SPLs. We then compute accuracy in terms of precision and recall (Fawcett, 2006) per agglomeration. In other words, we aim to answer RQ2.2 as follows.

RQ2.2. *How accurate is the relationship between agglomerations and instability?*

To assess accuracy of each type of agglomeration, we compute precision and recall in terms of true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) (Fawcett, 2006). TP is the number of agglomerations that indicate correctly instabilities. FP is the number of agglomerations that indicate incorrectly instabilities, i.e., indicate stability. TN is the number of non-agglomerations that does not indicate instability. Finally, FN is the number of non-agglomerations that indicate instability. The formula adopted to compute precision and recall (Fawcett, 2006) are presented in Equations 5.1 and 5.2, respectively.

$$P = \frac{TP}{TP + FP} \tag{5.1}$$

$$R = \frac{TP}{TP + FN} \quad (5.2)$$

Since even small-sized systems have several anomalies (Macia et al., 2012), developers should focus their maintenance effort on anomalies that represent the most critical maintenance problems. Thus, agglomerating anomalies can reduce the search space for finding those problems. We focus our analysis on accuracy computed in terms of precision and recall. In this dissertation, we compute precision and recall per type of agglomeration considering all instable components, regardless the sources of instability of each component. We made this decision because some instable components have multiple sources that relate to different types of agglomeration. For instance, the component `MediaController` of the *MediaManagement* feature has changed in Release 4 of MobileMedia SPL as a consequence of two sources of instability: *Add Crosscutting Feature* and *Distribute Code Among Features.*

Table 5.7 presents precision (P), recall (R), and the number of instable components indicated per type of anomaly agglomerations (#IC). This table also presents median, mean, and standard deviation for the results obtained for the four SPLs under analysis. We provide a discussion of our results per type of agglomeration as follows.

Table 5.7. Precision and Recall per Type of Agglomeration

Agglomeration	**Feature**			**Feature Hierarchy**			**Component**		
SPL	**P**	**R**	**#IC**	**P**	**R**	**#IC**	**P**	**R**	**#IC**
MobileMedia	76%	72%	65	100%	59%	30	50%	10%	8
Notepad	50%	20%	4	75%	50%	8	50%	25%	3
TankWar	92%	61%	37	82%	82%	66	65%	23%	17
WebStore	75%	60%	26	100%	24%	10	0%	0%	0
Median	**76%**	**61%**	**32**	**91%**	**54%**	**20**	**50%**	**16%**	**6**
Mean	**73%**	**53%**	**33**	**89%**	**54%**	**29**	**41%**	**14%**	**7**
Std. Dev.	**15%**	**20%**	**22**	**11%**	**21%**	**23**	**25%**	**10%**	**6**

Feature Agglomeration. The first three columns in Table 4 correspond to the results for feature agglomeration. We observed a precision with median of 76% and mean of 73%. We then observe that each 3 out of 4 feature agglomerations indicate instabilities. These results are expressive if we consider that agglomerations aim to provide a precise indication of instability, based on the high frequencies of code anomalies that may occur in any software system, including software product lines.

Regarding recall, we obtained a mean of 53%, with median of 61%, for the SPLs under analysis. We observed a percentage of recall equals or higher than 60% in 3 out of 4 SPLs. Indeed, low percentages of recall are expected in this study, since not all instabilities in SPL are related to anomalous code structures. Through a manual analysis of the four SPLs, we identified various sources of instability that do not relate with code anomalies. For instance, in MobileMedia some components have changed from one release to another because of the inclusion of new functionalities by means of features (e.g., in Releases 1 to 2). In TankWar, some components have changed due to the inclusion of FOP-specific mechanisms (e.g., in Releases 2 to 3).

Note that, for Notepad, the low rates of both precision and recall may be justified by the small percentage for both instable and anomalous components. As an example, Notepad has only 37.5% of instable components, against 58.9%, 79.3%, and 44.2% for MobileMedia, TankWar, and WebStore, respectively. Despite of that, in general, our results suggest that there is a high rate of feature agglomerations that, possibly, may indicate instabilities in the SPLs. However, since we did not observe statistical significance for this type of agglomeration (see Table 5.6), we may not affirm that feature agglomerations are indicators of instability in SPLs.

To illustrate the effectiveness of a feature agglomeration in indicating instability, let us consider the following example extracted from MobileMedia.

Example 1: The MediaManagement feature. Figure 5.1 presents the *MediaManagement* feature with four components: `Constants`, `MediaController`, `MediaListScreen`, and `MediaUtil`. For each component, we have the respective number of code anomalies represented by "#" on the upon-right side of the component. All these components are anomalous and, therefore, this set of components corresponds to a feature agglomeration. By analyzing in details each anomalous component separately, we observe that only `Constants` and `MediaListScreen` have less than two anomalies, with *Lazy Class* and *Long Refinement Chain* respectively. Although both anomalies are symptoms of maintenance problems in the constant, they provides a limited view of problems that affect the SPL as a whole.

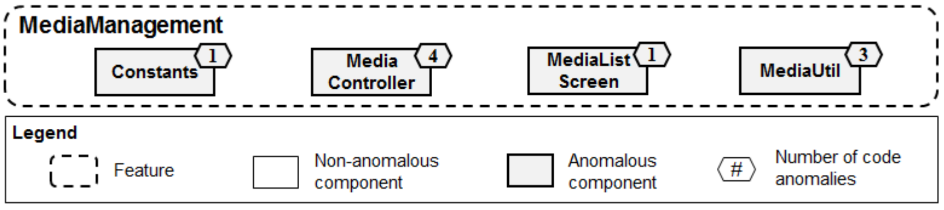


Figure 5.1. Feature Agglomeration of MediaManagement

In turn, by analyzing the entire feature agglomeration, we may observe wider issues. As an example, the components `MediaController` and `MediaUtil` have both *Long Method* and *Long Parameter List*. In general, these anomalies relate to high difficulty to maintain the affected code elements, the methods in this case. Since components of the same feature implement the same functionality, we expect that they access and use to one another. Thus, these anomaly occurrences in the same feature may lead to major maintenance problems in the entire feature. Moreover, attempts to treat these problems can lead to the overall feature maintainability issues.

In fact, the feature agglomeration formed by components from the *MediaManagement* feature (Figure 5.1) indicated relevant instabilities generated by a source of instability categorized as *Distribute Code Among Features* (see Table 5.2 for detailed description). In this case, the implementation of the component `BaseController` from feature *Base*, the most important controller of the SPL, was distributed to several features including *MediaManagement*. Consequently, this distribution of source code to other features made the components of the feature agglomeration instable.

Feature Hierarchy Agglomeration. The fourth, fifth, and sixth columns in Table 4 present precision, recall, and the number of instable components (#IC) for the analysis of feature hierarchy agglomeration. We obtained values similar to the first analysis, with respect to the feature analysis. First, regarding precision, we have a mean value of 89%, the highest value among types of agglomeration. These data suggest that the only a few feature hierarchy agglomerations, i.e., related to a refinement chain formed by components and its refinements, are not related to instabilities. We additionally obtained a mean recall of 54% for the target SPLs That is, the best value among agglomeration types. This result indicates that a significant number of feature hierarchy agglomerations are candidates to indicate instabilities. We conclude that the feature hierarchy agglomeration is an indicator of instabilities in SPL.

To illustrate feature hierarchy agglomerations that indicated instability, let us consider the following example extracted from MobileMedia.

Example 2: The MediaController feature hierarchy. In Figure 5.2, the features *MediaManagement*, *Sorting*, and *SMSTransfer* have a component `MediaController` each. Such components form a refinement chain. However, only the components of *MediaManagement* and *SMSTransfer* are anomalous and compose a feature hierarchy agglomeration. Both are instable and affected by *Long Parameter List*, an anomaly the provides a limited view of the extent of maintenance problems. In addition, the component located in the feature *MediaManagement* has *Large Class*, *Long Method*, and *Long Refinement Chain*. Although each anomaly provides hints of problems in the respective components, a view of the feature hierarchy may help us reasoning about major problems in the whole refinement chain.

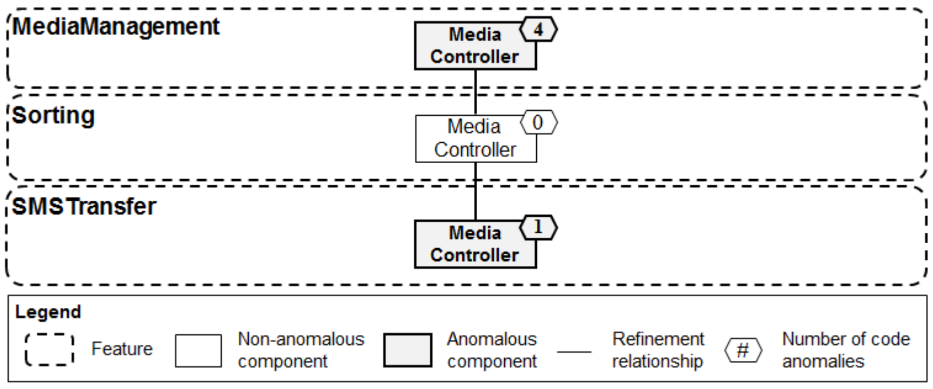


Figure 5.2. Feature Hierarchy Agglomeration of MediaController

Note that the component `MediaController` of feature *MediaManagement* is a constant and, therefore, the components below in the feature hierarchy are refinements. This constant has four code anomalies, as aforementioned. This number is high, since the highest number of anomalies in a component of MobileMedia was five, for `MediaAccessor` of feature *Base*. Thus, the concentration of anomalies affecting locally `MediaController` suggests the occurrence of one or more problems. Besides that, there are two other components refining the constant. Because of *Long Parameter List*, it may indicate an overload of responsibilities in the anomalous method. Consequently, such anomaly is even more critical than the individual analysis may suggest.

In fact, the feature hierarchy agglomeration formed by components of the refinement chain of `MediaController` indicated several relevant sources of instability. For instance, this agglomeration captured the instability caused by a source categorized as *Pull Up Common Feature Code* (Table 5.2). In this case, due to the addition of new types of media in MobileMedia, it was reorganized the implementation of feature *CopyPhoto* into two features: *CopyPhoto* and *CopyMedia*. This change affected all components from the agglomeration in terms of instability.

Component Agglomerations. The three last columns in Table 4 present precision, recall, and and the number of instable components (#IC) for the analysis of component agglomeration. In this case, we obtained values significantly different when compared to the feature agglomeration analysis. With respect to the four SPLs, we obtained a mean precision of 41%. This result points that less than a half of the observed component agglomerations relate, in fact, to instabilities. Based on these data, we may not affirm that this type of agglomerations is effective in indicating instabilities. Moreover, we obtained a mean recall of 14% for the SPLs. This result is very low when considering that systems tend to present several instable components and anomalies.

The low rates of precision and recall for component agglomeration may relate with the fact that this type of agglomeration analyses a single source file. Note that a single file tends to have less changes than a set of multiple files, as analyzed by both feature and feature hierarchy agglomerations. Consequently, the effectiveness of component agglomerations in identifying instability may have been negatively affected by the low number of possible instable files. Therefore, our data suggests that the component agglomeration is not an indicator of instabilities in SPL.

In spite of the low rates of precision and recall, we observed interesting cases of component agglomerations that indicate instabilities. Let us consider the following example, extracted from MobileMedia, to illustrate how component agglomerations support the identification of instability in SPL.

Example 3: The MediaController component. In Figure 5.3, the component with the highest amount of anomalies is `MediaController` of the feature *MediaManagement*. Four anomalies with potential to harm the SPL maintainability occur in this component, namely *Large Class*, *Long Method*, *Long Parameter List*, and *Long Refinement Chain*. By analyzing each anomaly separately, we limit our observations to the possible problems that the respective anomalies may cause.

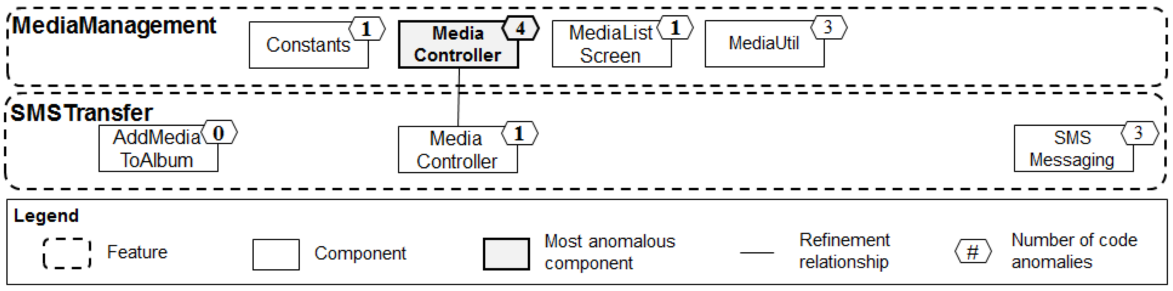


Figure 5.3. Component Agglomerations of MediaController

In turn, by agglomerating anomalies that affect the same component, we may draw observations that are more conclusive. For instance, if we consider *Large Class* and *Long Method* separately, we may overlook two important issues regarding `MediaController`. First, this component is a constant and many other components refine its implementation. Second, this component has a *Long Refinement Chain* that makes code harder to understand and evolve. This anomaly, summed to *Large Class* and *Long Method*, tend to harm the SPL maintainability even more.

In fact, code elements from the component `MediaController`, of the feature *MediaManagement*, indicated correctly different sources of instability. These sources include the following sources of instability in SPL (Table 5.2): *Distribute Code among*

Features regarding the implementation of component `BaseController` from feature *Base* and *Pull Up Common Feature Code* regarding the reorganization of feature *Copy-Photo*. We discuss both sources previously in this section, for feature agglomeration and feature hierarchy agglomeration.

Summary for RQ2. Our data suggest that *feature hierarchy* is the most effective type of agglomeration for identification of sources of instability in SPLs, due to the p-value lower than 0.05 (given a 95% confidence interval) and highest Odds Ratio close to 3.8. When compared to non-agglomerated anomalies, with Odds Ratio equals 0.08, we observe that feature hierarchy agglomeration is 3.8 times more effective in identifying instabilities. The high precision of 89% for this type reinforces our findings.

5.4 Threats to Validity

We rely on the guidelines of Wohlin et al. (2012) to discuss threats to the study validity with respective treatments as follows.

Construct and Internal Validity. We carefully designed our study for replication. However, a major threat to our study is the set of metrics used in the detection strategy composition. This set is restricted to the metrics provided by the SPL repository (Vale et al., 2015) adopted in our study. To minimize this issue, we selected some well-known and largely studied metrics, such as *McCabe's Cyclomatic Complexity (Cyclo)* (McCabe, 1976). The list of detection strategies used in this study is available in the research website (Fernandes et al., 2017c). Regarding the small length of the analyzed SPLs, we highlight the limited number of SPLs available for research, as the limited number of releases for the available SPLs. The low number of available releases has lead us to consider a component as instable if it has changed in two or more releases. To minimize this issue, we analyzed the SPLs in all available releases. Finally, we conducted the data collection carefully. To minimize errors, two authors checked all the collected data and re-collected the data in case of divergence.

Conclusion and External Validity. We designed a data analysis protocol carefully. To compute the statistical significance and strength of the relationship between agglomerations and instabilities, we computed the Fisher's test (Fisher, 1922) and Odds Ratio (Cornfield, 1951), two well-known and reliable techniques. We also computed precision and recall for the accuracy analysis of agglomerations, based on previous work (Oizumi et al., 2016). These procedures aim to minimize issues regarding the conclusions we draw. Two authors checked the analysis to avoid missing data and re-conducted the analysis to prevent biases. Regarding the generalization of findings, we expect that our results are extensible to other SPL development contexts than FOP. However, further investigation is required.

5.5 Final Remarks

This chapter presented an empirical study to evaluate our three novel types of code anomaly agglomerations, namely *feature*, *feature hierarchy*, and *component agglomeration*. First, we described the evaluation settings, including research questions, the target set of SPLs for analysis, and three study protocols for data collection and analysis. Our evaluation relied on four feature-oriented SPLs: MobileMedia (Figueiredo et al., 2008), Notepad (Kim et al., 2010), TankWar (Schulze et al., 2010), and WebStore (Gaia et al., 2014). As a result, we observed that feature hierarchy agglomeration was able to identify instaiblities in SPL with the highest rates of precision and recall in comparison with the other types of agglomeration.

In the next chapter, we discuss related work. Our discussion relies on studies that investigate (i) the impacts of anomalies on the SPL maintainability, (ii) techniques for identification of maintenance problems in SPL, and (iii) the use of code anomaly agglomerations as indicators of maintenance problems in object-oriented systems.

Chapter 6

Related Work

In this dissertation, we propose three novel types of anomaly agglomerations for feature-oriented SPL. An anomaly agglomeration occurs when two or more code anomalies inter-relate in the SPL source code. Each proposed agglomeration relies on the main SPL structural elements, i.e., *features*, *feature hierarchies*, and *components*. By proposing different types of anomaly agglomerations in SPL, we aim to support the identification of SPL maintenance problems that we may not identify with an analysis of a single code anomaly. Also in this dissertation, we evaluate the agglomerations as indicators of a specific problem that harms the SPL maintainability: instability.

This chapter discusses previous work that relate to ours as follows. Section 6.1 presents studies that characterize or investigates code anomalies in the SPL context. Section 6.2 discusses previous work that propose techniques for identifying sources of instability in SPL. Section 6.3 presents previous work that propose or apply code anomaly agglomerations as indicators of maintenance problems in object-oriented software systems. We also relate the previous work with this dissertation.

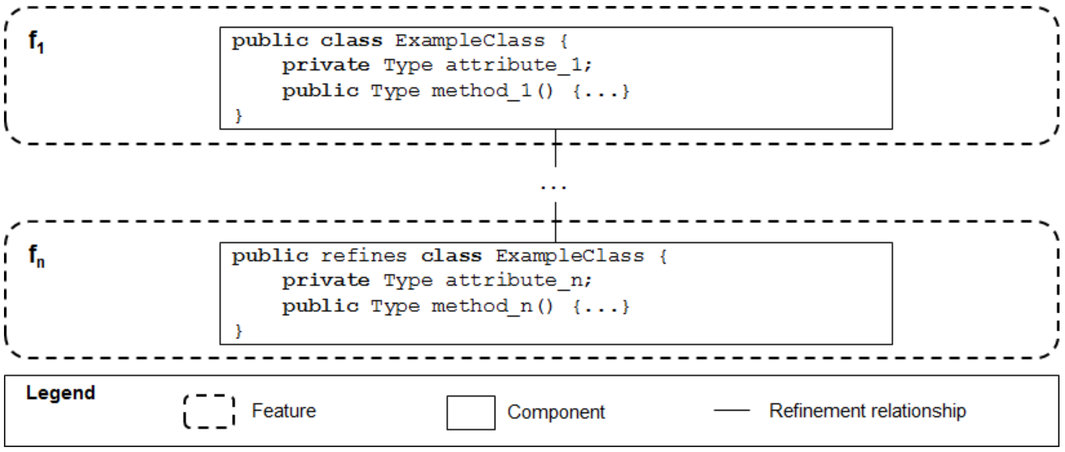
6.1 Code Anomalies in SPL

Code anomalies characterize symptoms of problems in the source code of a software system (Fowler, 1999). In general, code anomalies affect any system (Macia et al., 2012) and may occur in different code levels, such as components and methods (Fowler, 1999; Lanza and Marinescu, 2006). In the SPL context, variability mechanisms can introduce code anomalies in the product line (Fenske and Schulze, 2015). For instance, *Long Refinement Chain* is a method with several successive refinements in different features. It harms the SPL maintainability because it makes difficult to understand the side effects caused by changing a feature with respect to the whole product line (Fenske and

Schulze, 2015). Developers can identify code anomalies manually or automatically with the support of detection strategies, i.e., well-defined rules based on the characteristics of an anomaly (Marinescu, 2004), or detection tools (Moha et al., 2010).

Previous work (Andrade et al., 2014; Apel et al., 2013; Fenske and Schulze, 2015) has characterized and investigated code anomalies in SPL. Apel et al. (2013) discussed the use of code anomalies as indicators of potentially inadequate feature modeling or implementation. The authors provided a list of 14 code anomalies, such as *Unused Feature* and *Fat Products*, that affect different phases of the SPL engineering. For instance, *Unused Feature* is a feature that has never been included in SPL products and, therefore, should be discarded. *Fat Products* are products derived from a SPL that are too large and contain unnecessary functionalities, even after the configuration of a product that fits the client needs. It suggests that the feature modeling should be revised and optional code fragments may be extracted, for instance.

Similarly, Fenske and Schulze (2015) provided a catalog code anomalies, such as *Annotation Bundle*, *Inter-Feature Code Clones*, and *Long Refinement Chain*, that capture the notion of variability in SPL. In addition, the authors conduct an empirical study with SPL researchers to evaluate the relevance of the catalog of code anomalies. *Long Refinement Chain* is characterized as a method with too many refinements in different SPL features. This anomalous code structure may hinder developers in understanding the refineed method and performing proper changes. Figure 6.1 illustrates a *Long Refinement Chain* of `ExampleClass`. The feature f_1 implements the constant `ExampleClass` with one attribute and one method. The other features refine the constant by adding attributes and methods. For instance, f_n adds one attribute and one method. As far as a method is refined, it makes harder to understand the functionalities provided by a SPL product that includes the bottom-most refinements.



```
public class ExampleClass {
    private Type attribute_1;
    public Type method_1() {...}
}
```

```
public refines class ExampleClass {
    private Type attribute_n;
    public Type method_n() {...}
}
```

Figure 6.1. Abstract Example of Long Refinement Chain

Andrade et al. (2014) investigated code anomalies that affect the architecture of a SPL. For this purpose, they studied four code anomalies proposed for other contexts than SPL, in order to assess their occurrence on a sample SPL. As a result, they could not find occurrences of the four anomalies in the analyzed SPL. After, the authors proposed and discussed *Feature Concentration*, a SPL-specific code anomaly. *Feature Concentration* occurs when too many features are implemented in a single architectural unit. It suggests that the implementation of features in the SPL should be reviewed to prevent the overload of specific architectural units.

Finally, Vale et al. (2014) present a systematic literature review on code anomalies that affect SPLs implemented in different techniques, such as Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) and Delta-Oriented Programming (DOP) (Schaefer et al., 2010). The authors have found 70 code anomalies proposed in the literature. The anomalies were organized into three groups. The first group has the anomalies that affect a SPL in the lowest implementation level, i.e., the source code. The second group gathers anomalies that affect a SPL mostly in the architecture level. The third group contains the anomalies that affect in both code and architecture level.

Table 6.1 lists some of code anomalies identified by Vale et al. (2014). In addition, the table lists code anomalies proposed by (Fenske and Schulze, 2015) after the publication of the literature review. The first column presents the name of each code anomaly. The second column provides the technique for SPL implementation that may be affected by each anomaly. Finally, the third column describes the code anomalies.

Table 6.1. Examples of Code Anomalies in SPL, Adapted of Vale et al. (2014)

Code Anomaly	**Technology**	**Definition**
Annotation Bundle (Fenske and Schulze, 2015)	FOP	A method with too many variable parts. It may difficult the maintainability of the method
Duplicated Features (Schulze et al., 2013)	DOP	Two features have similar code implementations. It harms the SPL maintainability
Empty Features (Schulze et al., 2013)	DOP	A feature that has no effect on generated SPL products. It harms the SPL maintainability
God Aspect (Macia et al., 2011)	AOP	An aspect that implements two or more concern. It hamrs the SPL modularity
Inter-Feature Code Clones (Fenske and Schulze, 2015)	FOP	Duplicated parts of code that are located in different features. It harms the SPL maintainability
Large Aspect (Piveta et al., 2006)	AOP	A long aspect with too many code elements. It harms the aspect understandability
Lazy Aspect (Piveta et al., 2006)	AOP	An aspect that has only a few responsibilities. It harms the SPL maintainability
Long Refinement Chain (Fenske and Schulze, 2015)	FOP	A method with too many successive refinements in different features. It harms the SPL maintainability
Unused Features (Schulze et al., 2013)	DOP	A feature implemented in the SPL but not used by any product. It harms the SPL maintainability

Summary of Code Anomalies in SPL. We have found interesting previous work (Andrade et al., 2014; Apel et al., 2013; Fenske and Schulze, 2015) that investigates code anomalies in SPLs. All studies discuss the negative impacts of code anomalies in the SPL maintainability. In addition, they provide evidence on the importance of detecting and fixing code anomalies to prevent critical problems in the SPL design and the derivation of SPL products, for instance.

6.2 Instability in SPL

Instability is the probability of a software system to change, due to changes performed in different parts of the source code (Ampatzoglou et al., 2015). A previous study (Yau and Collofello, 1985) discusses that instability is significantly detrimental to the SPL maintainability. In general, instability is even more critical in SPL than in single software systems, because changes in one feature can propagate to other features and affect seemly-unrelated SPL products (Conejero et al., 2009).

Previous work (Cafeo et al., 2013; Figueiredo et al., 2008) has investigated instability in SPLs. For instance, Cafeo et al. (2013) conducted a study on software metrics as indicators of instabilities in evolving SPLs implemented using the aspect-oriented programming (Kiczales et al., 1997). For this purpose, the authors compared two sets of metrics. The first set is composed by general purpose metrics. The second set is composed by SPL-specific metrics related to feature dependency, i.e., that capture key properties of features in a SPL. The results indicated that the feature dependency metrics are better indicators of instability than the traditional metrics.

Figueiredo et al. (2008) presented another empirical study on instability in evolving SPLs. This study relies on design instability analysis and the evolution of two aspect-oriented SPLs through realistic development scenarios. The authors aimed to assess different aspects of design instability in the target product lines, such as feature modularization, feature dependency, and change propagation. As a result, they presented various development scenarios that affect the design stability of SPLs.

Summary of Instability in SPL. We have found relevant studies (Cafeo et al., 2013; Figueiredo et al., 2008) that investigate instability and its impact on SPL. In general, these studies emphasize that some implementation characteristics of the SPL implementation may induce to critical instability problems. They provide means to assess the instability in SPL, mainly focused on the aspect-oriented programming paradigm.

6.3 Code Anomaly Agglomerations

Code anomalies are symptoms of problems in the source code of a software system (Fowler, 1999). Such anomalies may support the identification of different maintenance problems in any system, including a SPL (Fenske and Schulze, 2015; Schulze et al., 2010). However, each single code anomaly may represent only a limited view of the problem extent. This limited view is a consequence of the scattered location of several maintenance problems into different parts of the code (Moha et al., 2010). Therefore, a solution for that limitation could be analyzing the inter-relations of code anomalies, i.e., the anomaly agglomerations (Oizumi et al., 2016).

Previous work (Oizumi et al., 2016; Yamashita and Moonen, 2013a) has investigated agglomerated anomalies as indicators of maintenance problems in systems. For instance, Yamashita and Moonen (2013a) conducted a study on the impact of inter-related anomalies on the maintenance of object-oriented systems. The authors assumed that, although individual anomalies harm the maintainability of systems, the interactions among anomalies may also be harmful. Their work evaluates the interactions among 12 code anomalies to understand how such interactions relate to maintenance problems. As a result, the authors concluded that anomalies located in the same software artifact tend to inter-relate and affect the system maintainability.

In particular, Oizumi et al. (2016) presented the most closely work to ours. The authors conducted an empirical study on the use of code anomaly agglomerations as indicators of design problems in object-oriented source code. For this purpose, they proposed types of code anomaly agglomerations that rely on source code elements. The authors showed that agglomerations are better indicators of design problems than code anomalies analyzed individually. The results suggested that some types of agglomeration could indicate problems with accuracy higher than 80%. However, the authors did not explore code anomaly agglomerations in the context of SPL.

Table 6.2 compares the types of anomaly agglomerations proposed by Oizumi et al. (2016) with the agglomerations proposed in this dissertation. The first column presents the study that proposes the agglomerations. The second column informs how each type of code agglomerations is located in the SPL. The *Implicit* location means that the inter-relation of code anomalies is not explicit in the source code. In turn, the *Explicit* location means that the inter-relation of anomalies is explicit in the source code. The third column provides the name of each type of agglomeration. Finally, the fourth column describes the types of anomaly agglomerations.

Table 6.2. Agglomerations by Oizumi et al. (2016) *versus* This Dissertation

Study	**Location**	**Type of Agglomeration**	**Description**
This dissertation	Implicit	Feature	Two or more anomalous components located in the same feature
	Explicit	Feature Hierarchy	Two or more anomalous components located in the same refinement chain
	Explicit	Component	Two or more anomalous code elements in the same components with two or more anomalies
Oizumi et al. (2016)	Implicit	Semantic	Two or more anomalies that affect code elements with the same concern
	Explicit	Inter-Component	Two or more anomalous design components with one or more anomalous elements each
	Explicit	Intra-Component	Two or more anomalous code elements located in the same design component

By analyzing Table 6.2, we observe that both studies propose three types of anomaly agglomerations, two of which take into account explicit inter-relations of code anomalies. In fact, this dissertation proposes novel types of anomaly agglomerations in the SPL context based on the agglomerations proposed by Oizumi et al. (2016). Moreover, both studies propose agglomerations that rely on *concerns*. A concern is an abstraction for a requirement of the software systems (Sommerville, 2010). Oizumi et al. (2016) propose *semantic agglomeration* as a group of anomalies that affect parts of code implementing the same concern. In turn, this dissertation proposes *feature agglomeration* that is a group of anomalous components of the same SPL feature and, therefore, they implement the same product-line concern.

Summary of Code Anomaly Agglomerations. After reviewing the literature, we have found studies (Oizumi et al., 2016; Yamashita and Moonen, 2013a) that evaluate the potential of code anomaly agglomerations in indicating software maintenance problems. Overall, both studies assume that the analysis of inter-related anomalies may indicate more critical, relevant problems than the analysis of individual anomalies. Moreover, these studies actually observed the expected benefits of grouping anomalies as agglomerations. In despite of that, none of them target on feature-oriented SPLs and instability, as we did in this dissertation.

6.4 Final Remarks

This chapter discussed studies closely related to the scope of this dissertation, i.e., code anomaly agglomerations as indicators of instability in SPL. Our discussion was divided into three research topics as follows. First, we presented previous work that introduce and investigate the occurrence of code anomalies in SPL. Second, we discussed studies aimed at investigating instability in software systems, including a study targeting on SPL. Finally, we discuss related work that focuses on agglomerating code anomalies to support the identification of maintenance problems in software systems. At this point, we build a link between these studies and this dissertation that focuses on a specific SPL maintenance problem: instability.

Although some of the work discussed in this chapter (Oizumi et al., 2016; Yamashita and Moonen, 2013a) investigated the inter-relation of code anomalies in object-oriented systems, none of them neither have used the investigated code anomaly agglomerations in the SPL context as indicators of instability. Nevertheless, they provided evidence that encourages the use of code anomalies in the context of SPL as well as the use of metrics that capture the properties of SPLs. Thus, their results justify the usage of code anomalies like *Large Class*, *Long Method*, and *Shotgun Surgery*. Similarly, the research conducted by Cafeo et al. (2013) motivated to use code anomalies that can capture properties of SPLs like *Long Refinement Chain*.

In the next chapter, we conclude this dissertation. First, we provide an overview of the main contributions of this dissertation. Second, we discuss opportunities for future work derived from the contributions and limitation of our study.

Chapter 7

Conclusion

Code anomalies are symptoms of problems in the source code of a software system (Fowler, 1999). Each code anomaly harms the system maintainability in a different level (Fowler, 1999; Lanza and Marinescu, 2006), by affecting packages, classes, or methods. As an example, *Large Class* is a class with too much knowledge of the system and too many responsibilities. It makes difficult to understand the source code and to perform maintenance tasks in the system. Code anomalies affect any software system, including software product lines (SPL) (Fenske and Schulze, 2015). Moreover, a previous work (Medeiros et al., 2015) states that SPL-specific code anomalies can be easier to introduce, harder to fix, and more critical than others, due to the inherent complexity of SPLs. Therefore, developers should identify and eliminate code anomalies that harm the SPL maintainability whenever possible.

Previous work (Fenske and Schulze, 2015; Schulze et al., 2010) assume that individual code anomalies are sufficient to characterize maintenance problems in a SPL. However, each anomaly in isolation may represent only a partial view of the problem extent. Thus, previous studies have limitations to characterize anomalous code structures that indicate SPL maintenance problems. To address these limitations, previous work (Oizumi et al., 2016; Yamashita and Moonen, 2013a) investigate to what extent agglomerating code anomalies may support the characterization of maintenance problems in object-oriented software systems. However, we still lack similar studies in the SPL context and focused on a specific SPL maintenance problem: instability.

To fill the aforementioned lack of studies, this dissertation presented our research on anomaly agglomerations as indicators of instability in SPL. We proposed novel types of anomaly agglomerations in SPL and evaluated them. The remainder of this chapter is organized as follows. Section 7.1 summarizes the main contributions of this dissertation. Finally, Section 7.2 discusses opportunities for future work.

7.1 Main Contributions

This dissertation aimed at contributing mainly to the SPL community and to the whole Software Engineering (SE) community. The dissertation is organized in three parts discusses as follows. First, we provided a systematic literature review on code anomaly detection tools (Fernandes et al., 2016). We also conducted a comparative study of the tools identified in the review, some of them compatible with SPL. Second, we conducted a comparative study of detection strategies for code anomalies that affect SPLs (Fernandes et al., 2017a). Through this study, we discussed aspects to take into account when proposing novel detection strategies for SPL. As a summary of our preliminary findings, we presented another work (Fernandes and Figueiredo, 2016). We then discussed our study goals and findings with a broad audience of SE. After, we refined our study in order to improve the quality of this dissertation.

Third, in this dissertation, we proposed three novel types of anomaly agglomerations in SPL, namely *feature*, *feature hierarchy*, and *component agglomeration*. Each type of anomaly agglomeration relies on a different structural element of feature-oriented SPLs. A *feature agglomeration* occurs if there are two or more anomalous components located in a single SPL feature. A *feature hierarchy agglomeration* occurs when two or more anomalous components belong to a single feature hierarchy, i.e., they are explicitly inter-related via a refinement relationship that cuts across the SPL features. A *component agglomeration* occurs when two or more anomalous code elements of a component are affected by at least two different code anomalies.

Also in this dissertation, we investigated whether non-agglomerated anomalies are indicators of instability in SPL. In fact, our findings suggest that non-agglomerated anomalies do not support the identification of anomalous code structures that cause instability and, consequently, harm the SPL maintainability. We then investigated to what extent the novel types of anomaly agglomerations indicate instability in SPL. Our study relied on the analysis of four feature-oriented SPLs from different domains: MobileMedia (Figueiredo et al., 2008), Notepad (Kim et al., 2010), TankWar (Schulze et al., 2010), and WebStore (Gaia et al., 2014). As a result, we observed that *feture hierarchy agglomeration* suffices to indicate instabilities in SPL. The observed statistical significance and high rates of recall and precision for this type of agglomeration reinforce our findings. Overall, we have found that anomaly agglomerations may effectively support the identification of instabilities in SPLs. We presented our main findings of third part of this dissertation in another work (Fernandes et al., 2017b).

7.2 Future Work

We present some opportunities for future work as follows.

- To conduct a large-scale evaluation of our three novel types of anomaly agglomerations for feature-oriented SPLs. This dissertation relies on the evaluation of only four feature-oriented SPLs. Thus, the generalization of our findings has an inherent threat. As an example of evaluation, we may analyse a larger set of FOP-based SPLs, with several releases, and from other domains.

- To conduct a empirical study in controlled environment with our three novel types of anomaly agglomerations for feature-oriented SPLs. For instance, we may conduct a case study in industry settings to assess if the proposed agglomerations actually support developers in identifying SPL maintenance problems. We also may assess if agglomerations support saving maintenance efforts.

- To develop and evaluate an anomaly agglomeration detection tool for feature-oriented SPLs. In this dissertation, we manually computed the agglomerations per SPL because of the small size and number of available releases per SPL. However, the automated detection of anomaly agglomerations could support both the developers that use the agglomerations to identify SPL maintenance problems and the researchers that conduct large-scale studies.

- To propose other types of anomaly agglomerations for feature-oriented SPLs. Our novel agglomerations rely on the main SPL structural elements, i.e., features, feature hierarchies, and components. However, we may explore other elements. For instance, in this dissertation we did not take into account the constraints among features that limit the inclusion of an optional or alternative feature. These constraints may support us in understanding the inter-relation of code anomalies that affect different SPL features.

- To investigate the cause-effect relationship among code anomalies that belong to the same agglomeration. In this study, we assume that code anomalies that form an agglomeration inter-relate. However, we do not characterize how such inter-relation occurs, except by the fact that they are located in a single SPL structural element. Therefore, we may explore the types of code anomaly that lead to the occurrence of other types of anomaly, and the implications of that relationship in the SPL maintainability.

- To investigate other techniques for SPL implementation. For instance, Aspect-Oriented Programming (Kiczales et al., 1997) is a technique that modularizes the SPL concerns of a system into code units called *aspects*. Another technique, Delta-Oriented Programming (Schaefer et al., 2010), aims to improve the flexibility of FOP-based SPLs via *delta modules*, i.e., code units that add, modify, and remove code of the SPL features. Both techniques have different SPL structural elements that we may explore with agglomerations. In addition, we may investigate other SPL maintenance problems than instability, such as error-proneness (Liebig et al., 2010) and understandability (Kästner et al., 2008).

Bibliography

Abílio, R., Padilha, J., Figueiredo, E., and Costa, H. (2015). Detecting Code Smells in Software Product Lines: An Exploratory Study. In *Proceedings of the 12th International Conference on Information Technology: New Generations (ITNG)*, pages 433–438.

Abilio, R., Vale, G., Figueiredo, E., and Costa, H. (2016). Metrics for Feature-Oriented Programming. In *Proceedings of the 7th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 36–42.

Almeida, E., Alvaro, A., Lucrédio, D., Garcia, V., and Meira, S. (2004). RiSE Project: Towards a Robust Framework for Software Reuse. In *Proceedings of the 5th International Conference on Information Reuse and Integration (IRI)*, pages 48–53.

Altman, D. (1991). *Practical Statistics for Medical Research.* Chapman & Hall.

Alves, P., Santana, D., and Figueiredo, E. (2012). ConcernReCS: Finding Code Smells in Software Aspectization. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 1463--1464.

Ampatzoglou, A., Chatzigeorgiou, A., Charalampidou, S., and Avgeriou, P. (2015). The Effect of GoF Design Patterns on Stability: A Case Study. *IEEE Transactions on Software Engineering (TSE)*, 41(8):781–802.

Andrade, H., Almeida, E., and Crnkovic, I. (2014). Architectural Bad Smells in Software Product Lines: An Exploratory Study. In *Proceedings of the 11th Working Conference on Software Architecture (WICSA)*, pages 1–12.

Anquetil, N., Oliveira, K., Sousa, K., and Dias, M. (2007). Software Maintenance Seen as a Knowledge Management Issue. *Information and Software Technology (IST)*, 49(5):515–529.

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines*. Springer.

Apel, S., Kästner, C., and Lengauer, C. (2009). FeatureHouse: Language-Independent, Automated Software Composition. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 221–231.

Apel, S., Leich, T., Rosenmüller, M., and Saake, G. (2005). FeatureC++: On the Symbiosis of Feature-Oriented and Aspect-Oriented Programming. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 125–140.

Apel, S., Speidel, H., Wendler, P., von Rhein, A., and Beyer, D. (2011). Detection of Feature Interactions using Feature-Aware Verification. In *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, pages 372–375.

Baggen, R., Correia, J. P., Schill, K., and Visser, J. (2012). Standardized Code Quality Benchmarking for Improving Software Maintainability. *Software Quality Journal (SQJ)*, 20(2):287–307.

Batory, D., Sarvela, J., and Rauschmayer, A. (2003). Scaling Step-Wise Refinement. In *Proceedings of the 25th International Conference on Software Engineering (ICSE)*, pages 187–197.

Bellon, S., Koschke, R., Antoniol, G., Krinke, J., and Merlo, E. (2007). Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering (TSE)*, 33(9):571–591.

Bennett, K. and Rajlich, V. (2000). Software Maintenance and Evolution: A Roadmap. In *Proceedings of the Conference on the Future of Software Engineering, co-located with the 22nd International Conference on Software Engineering (ICSE)*, pages 73–87.

Bulychev, P. and Minea, M. (2008). Duplicate Code Detection Using Anti-unification. In *Proceedings of the 2nd Spring/Summer Young Researchers' Colloquium on Software Engineering (SYRCoSE)*, pages 51–54.

Cafeo, B., Dantas, F., Cirilo, E., and Garcia, A. (2013). Towards Indicators of Instabilities in Software Product Lines: An Empirical Evaluation of Metrics. In *Proceedings of the 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 69–75.

Chaudron, M., Katumba, B., and Ran, X. (2014). Automated Prioritization of Metrics-Based Design Flaws in UML Class Diagrams. In *Proceedings of the 40th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 369–376.

Chidamber, S. and Kemerer, C. (1994). A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering (TSE)*, 20(6):476–493.

Conejero, J., Figueiredo, E., Garcia, A., Hernández, J., and Jurado, E. (2009). Early Crosscutting Metrics as Predictors of Software Instability. In *Proceedings of the 47th International Conference on Objects, Components, Models and Patterns (TOOLS EUROPE)*, pages 136–156.

Copeland, T. (2005). *PMD Applied: An Easy-to-Use Guide for Developers*. Centennial Books.

Cordy, J. and Roy, C. (2011). The NiCad Clone Detector. In *Proceedings of the 19th International Conference on Program Comprehension (ICPC)*, pages 219–220.

Cornfield, J. (1951). A Method of Estimating Comparative Rates from Clinical Data: Applications to Cancer of the Lung, Breast, and Cervix. *Journal of the National Cancer Institute*, 11(6):1269–1275.

Deissenboeck, F., Pizka, M., and Seifert, T. (2005). Tool Support for Continuous Quality Assessment. In *Proceedings of the 13th International Workshop on Software Technology and Engineering Practice (STEP)*, pages 127–136.

Dig, D., Manzoor, K., Johnson, R., and Nguyen, T. N. (2007). Refactoring-Aware Configuration Management for Object-Oriented Programs. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 427–436.

Fard, A. and Mesbah, A. (2013). JSNose: Detecting JavaScript Code Smells. In *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 116–125.

Fawcett, T. (2006). An Introduction to ROC Analysis. *Pattern Recognition Letters*, 27(8):861–874.

Fenske, W. and Schulze, S. (2015). Code Smells Revisited: A Variability Perspective. In *Proceedings of the 9th International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 3–10.

Fernandes, E. and Figueiredo, E. (2016). Detecting Code Anomalies in Software Product Lines. In *Proceedings of the 6th Theses and Dissertations of CBSoft (WTDSoft), co-located with the 7th Brazilian Conference on Software: Theory and Practice (CBSoft)*, pages 1–8.

Fernandes, E., Oliveira, J., Vale, G., Paiva, T., and Figueiredo, E. (2016). A Review-based Comparative Study of Bad Smell Detection Tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 1–12.

Fernandes, E., Souza, P., Ferreira, K., Bigonha, M., and Figueiredo, E. (2017a). Detection Strategies for Modularity Anomalies: An Evaluation with Software Product Lines. In *Proceedings of the 14th International Conference on Information Technology: New Generations (ITNG)*, pages 1–6.

Fernandes, E., Vale, G., Sousa, L., Figueiredo, E., Garcia, A., and Lee, J. (2017b). No Code Anomaly is an Island: Anomaly Agglomeration as Sign of Product Line Instabilities. In *Proceedings of the 16th International Conference on Software Reuse (ICSR)*, pages 1–16.

Fernandes, E., Vale, G., Sousa, L., Figueiredo, E., Garcia, A., and Lee, J. (2017c). No Code Anomaly is an Island: Anomaly Agglomeration as Sign of Product Line Instabilities – Data of the Study. `http://labsoft.dcc.ufmg.br/doku.php?id=about:no_code_anomaly_is_an_island`. [Online; accessed February 11, 2017].

Ferreira, G., Gaia, F., Figueiredo, E., and Maia, M. (2014). On the Use of Feature-Oriented Programming for Evolving Software Product Lines: A Comparative Study. *Science of Computer Programming (SCP)*, 93(A):65–85.

Figueiredo, E., Cacho, N., Sant'Anna, C., Monteiro, M., Kulesza, U., Garcia, A., Soares, S., Ferrari, F., Khan, S., Castor Filho, F., and Dantas, F. (2008). Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 261–270.

Figueiredo, E., Silva, B., Sant'Anna, C., Garcia, A., Whittle, J., and Nunes, D. (2009). Crosscutting Patterns and Design Stability: An Exploratory Analysis. In *Proceedings of the 17th International Conference on Program Comprehension (ICPC)*, pages 138–147.

Fisher, R. (1922). On the Interpretation of χ^2 from Contingency Tables, and the Calculation of P. *Journal of the Royal Statistical Society*, 85(1):87–94.

Fontana, F., Mangiacavalli, M., Pochiero, D., and Zanoni, M. (2015). On Experimenting Refactoring Tools to Remove Code Smells. In *Proceedings of the Scientific Workshops on the 16th International Conference on Agile Software Development Proceedings of the (XP)*, page 7.

Fontana, F., Zanoni, M., Marino, A., and Mantyla, M. (2013). Code Smell Detection: Towards a Machine Learning-based Approach. In *Proceedings of the 29th International Conference on Software Maintenance (ICSM)*, pages 396–399.

Fontana, F. A., Braione, P., and Zanoni, M. (2012). Automatic Detection of Bad Smells in Code: An Experimental Assessment. *Journal of Object Technology (JOT)*, 11(2):1–38.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code.* Object Technology Series. Addison-Wesley.

Gaia, F., Ferreira, G., Figueiredo, E., and Maia, M. (2014). A Quantitative and Qualitative Assessment of Aspectual Feature Modules for Evolving Software Product Lines. *Science of Computer Programming (SCP)*, 96(2):230–253.

Griffith, I., Wahl, S., and Izurieta, C. (2011). TrueRefactor: An Automated Refactoring Tool to Improve Legacy System and Application Comprehensibility. In *Proceedings of the 24th International Conference on Computer Applications in Industry and Engineering (CAINE).*

Gwet, K. (2014). *Handbook of Inter-Rater Reliability: The Definitive Guide to Measuring the Extent of Agreement Among Raters.* Advanced Analytics, LLC.

Hall, T., Zhang, M., Bowes, D., and Sun, Y. (2014). Some Code Smells have a Significant but Small Effect on Faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):1–39.

Jiang, L., Misherghi, G., Su, Z., and Glondu, S. (2007). DECKARD: Scalable and Accurate Tree-based Detection of Code Clones. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 96–105.

Juergens, E., Deissenboeck, F., Hummel, B., and Wagner, S. (2009). Do Code Clones Matter? In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 485–495.

Kamiya, T., Kusumoto, S., and Inoue, K. (2002). CCFinder: A Multilinguistic Token-based Code Clone Detection System for Large Scale Source Sode. *IEEE Transactions on Software Engineering (TSE)*, 28(7):654–670.

Kästner, C. and Apel, S. (2008). Integrating Compositional and Annotative Approaches for Product Line Engineering. In *Proceedings of the Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, co-located with the 7th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 35–40.

Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in Software Product Lines. In *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pages 311–320.

Khomh, F., Di Penta, M., and Gueheneuc, Y.-G. (2009). An Exploratory Study of the Impact of Code Smells on Software Change-Proneness. In *Proceedings of the 16th Working Conference on Reverse Engineering (WCRE)*, pages 75–84.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242.

Kim, C., Bodden, E., Batory, D., and Khurshid, S. (2010). Reducing Configurations to Monitor in a Sofware Product Line. In *Proceedings of the 1st International Conference on Runtime Verification (RV)*, pages 285–299.

Kitchenham, B. and Charters, S. (2007). Guidelines for Performing Systematic Literature Reviews in Software Engineering. Technical Report, Version 2.3, EBSE Technical Report.

Lanza, M. and Marinescu, R. (2006). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer Science & Business Media.

Lee, J. and Muthig, D. (2006). Feature-Oriented Variability Management in Product Line Engineering. *Communications of the ACM*, 49(12):55–59.

Li, H. and Thompson, S. (2010). Similar Code Detection and Elimination for Erlang Programs. In *Proceedings of the 12th International Symposium on Practical Aspects of Declarative Languages (PADL)*, pages 104–118.

Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, pages 105–114.

Liu, H., Ma, Z., Shao, W., and Niu, Z. (2012). Schedule of Bad Smell Detection and Resolution: A New Way to Save Effort. *IEEE Transactions on Software Engineering (TSE)*, 38(1):220–235.

Lorenz, M. and Kidd, J. (1994). *Object-Oriented Software Metrics: A Practical Guide*. Prentice-Hall.

Macia, I., Garcia, A., and von Staa, A. (2011). An Exploratory Study of Code Smells in Evolving Aspect-Oriented Systems. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 203–214.

Macia, I., Garcia, J., Popescu, D., Garcia, A., Medvidovic, N., and von Staa, A. (2012). Are Automatically-Detected Code Anomalies Relevant to Architectural Modularity? An Exploratory Analysis of Evolving Systems. In *Proceedings of the 11th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 167–178.

Marinescu, C., Marinescu, R., Mihancea, P., and Wettel, R. (2005). iPlasma: An Integrated Platform for Quality Assessment of Object-Oriented Design. In *Proceedings of the 21st International Conference on Software Maintenance (ICSM)*, pages 77–80.

Marinescu, R. (2004). Detection Strategies: Metrics-Based Rules for Detecting Design Flaws. In *Proceedings of the 20th International Conference on Software Maintenance (ICSE)*, pages 350–359.

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering (TSE)*, 2(4):308–320.

Medeiros, F. (2014). An Approach to Safely Evolve Program Families in C. In *Proceedings of the SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH)*, pages 25–27.

Medeiros, F., Kästner, C., Ribeiro, M., Nadi, S., and Gheyi, R. (2015). The Love/Hate Relationship with the C Preprocessor: An Interview Study. In *Proceedings of the 29th European Conference on Object-Oriented Programming (ECOOP)*, volume 12, pages 495–518.

Meyer, B. (1988). *Object-Oriented Software Construction.* Prentice Hall.

Mezini, M. and Ostermann, K. (2004). Variability Management with Feature-Oriented Programming and Aspects. *ACM SIGSOFT Software Engineering Notes (SEN)*, 29(6):127–136.

Miller, B., Hsia, P., and Kung, C. (1999). Object-Oriented Architecture Measures. In *Proceedings of the 32nd Hawaii International Conference on Systems Sciences (HICSS)*, pages 1–18.

Moha, N., Gueheneuc, Y.-G., Duchien, L., and Le Meur, A.-F. (2010). DECOR: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering (TSE)*, 36(1):20–36.

Munro, M. (2005). Product Metrics for Automatic Identification of "Bad Smell" Design Problems in Java Source-Code. In *Proceedings of the 11th International Symposium on Software Metrics (METRICS)*, pages 1–9.

Murphy-Hill, E. and Black, A. (2010). . An Interactive Ambient Visualization for Code Smells. In *Proceedings of the 5th Symposium on Software Visualization (SOFTVIS)*, pages 5–14.

Oizumi, W., Garcia, A., Sousa, L., Cafeo, B., and Zhao, Y. (2016). Code Anomalies Flock Together: Exploring Code Anomaly Agglomerations for Locating Design Problems. In *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pages 440–451.

Parnin, C., Görg, C., and Nnadi, O. (2008). A Catalogue of Lightweight Visualizations to Support Code Smell Inspection. In *Proceedings of the 4th Symposium on Software Visualization (SOFTVIS)*, pages 77–86.

Piveta, E., Hecht, M., Pimenta, M., and Price, R. (2006). Detecting Bad Smells in AspectJ. *Journal of Universal Computer Science (J.UCS)*, 12(7):811–827.

Pohl, K., Böckle, G., and van Der Linden, F. J. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer Science & Business Media.

Rosenmüller, M., Apel, S., Leich, T., and Saake, G. (2009). Tailor-Made Data Management for Embedded Systems: A Case Study on Berkeley DB. *Data & Knowledge Engineering (DKE)*, 68(12):1493–1512.

Schaefer, I., Bettini, L., Bono, V., Damiani, F., and Tanzarella, N. (2010). Delta-Oriented Programming of Software Product Lines. In *Proceedings of the 14th International Conference on Software Product Lines (SPLC)*, pages 77–91.

Schneidewind, N. (1987). The State of Software Maintenance. *IEEE Transactions on Software Engineering (TSE)*, 13(3):303–310.

Schulze, S., Apel, S., and Kästner, C. (2010). Code Clones in Feature-Oriented Software Product Lines. In *Proceedings of the 9th International Conference on Generative Programming and Component Engineering (GPCE)*, pages 103–112.

Schulze, S., Richers, O., and Schaefer, I. (2013). Refactoring Delta-Oriented Software Product Lines. In *Proceedings of the 12th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 73–84.

Singh, V., Snipes, W., and Kraft, N. (2014). A Framework for Estimating Interest on Technical Debt by Monitoring Developer Activity Related to Code Comprehension. In *Proceedings of the 6th International Workshop on Managing Technical Debt (MTD)*, pages 27–30.

Sommerville, I. (2010). Software Engineering. *Addison-Wesley.*

Tamrawi, A., Nguyen, H. A., Nguyen, H. V., and Nguyen, T. N. (2012). SYMake: A Build Code Analysis and Refactoring Tool for Makefiles. In *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*, pages 366–369.

Tsantalis, N., Chaikalis, T., and Chatzigeorgiou, A. (2008). JDeodorant: Identification and Removal of Type-Checking Bad Smells. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 329–331.

Vale, G., Albuquerque, D., Figueiredo, E., and Garcia, A. (2015). Defining Metric Thresholds for Software Product Lines: A Comparative Study. In *Proceedings of the 19th International Conference on Software Product Line (SPLC)*, pages 176–185.

Vale, G. and Figueiredo, E. (2015). A Method to Derive Metric Thresholds for Software Product Lines. In *Proceedings of the 29th Brazilian Symposium on Software Engineering (SBES)*, pages 110–119.

Vale, G., Figueiredo, E., Abílio, R., and Costa, H. (2014). Bad Smells in Software Product Lines: A Systematic Review. In *Proceedings of the 8th Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*, pages 84–94.

Van Emden, E. and Moonen, L. (2002). Java Quality Assurance by Detecting Code Smells. In *Proceedings of the 9th Working Conference on Reverse Engineering (WCRE)*, pages 97–106.

Vidal, S., Marcos, C., and Díaz-Pace, J. A. (2016). An Approach to Prioritize Code Smells for Refactoring. *Automated Software Engineering (ASE)*, 23(3):501–532.

Vidal, S., Vazquez, H., Diaz-Pace, J. A., Marcos, C., Garcia, A., and Oizumi, W. (2015). JSpIRIT: A Flexible Tool for the Analysis of Code Smells. In *Proceedings of the 34th International Conference of the Chilean Computer Science Society (SCCC)*, pages 1–6.

Wettel, R. and Marinescu, R. (2005). Archeology of Code Duplication: Recovering Duplication Chains from Small Duplication Fragments. In *Proceedings of the 7th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer Science & Business Media.

Yamashita, A. and Moonen, L. (2013a). Exploring the Impact of Inter-Smell Relations on Software Maintainability: An Empirical Study. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 682–691.

Yamashita, A. and Moonen, L. (2013b). To What Extent Can Maintenance Problems be Predicted by Code Smell Detection? – An Empirical Study. *Information and Software Technology (IST)*, 55(12):2223–2242.

Yau, S. and Collofello, J. (1985). Design Stability Measures for Software Maintenance. *IEEE Transactions on Software Engineering (TSE)*, 11(9):849–856.