Pontifícia Universidade Católica
do Rio de Janeiro

Bruno Barbieri de Pontes Cafeo

On the Relationship between Feature Dependencies and Change Propagation

TESE DE DOUTORADO

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática PUC-Rio as partial fulfillment of the requirements for the degree of Doutor em Ciências – Informática

Advisor: Prof. Alessandro Fabricio Garcia

Rio de Janeiro
September 2015

Bruno Barbieri de Pontes Cafeo

On the Relationship between Feature Dependencies and Change Propagation

Thesis presented to the Programa de Pós-Graduação em Informática of the Departamento de Informática do Centro Técnico Científico da PUC–Rio as partial fulfillment of the requirements for the degree of Doutor.

Rio de Janeiro, September 2nd, 2015

Bruno Barbieri de Pontes Cafeo

Bruno Cafeo joined PUC-Rio as a Ph.D. student on Software Engineering in the Informatics Department in 2011. He visited the University of Lancaster – UK in 2014 to work with Prof. Jaejoon Lee's team. In addition, he worked with the research group headed by Prof. Sven Apel at University of Passau – Germany in 2014. Bruno was also a collaborator in under-graduation courses at PUC-Rio. In addition, he has also been a collaborator and has participated in the writing of a number of research projects. Bruno began his history in Informatics in 2000, when he started his technical school in Informatics at the São Paulo State University (UNESP). After that, he received a B.Sc. degree in Computer Science in 2009 and a M.Sc. degree in Computer Science and Mathematical Computing in 2011 from the University of São Paulo (USP). His main research interests are in the area of software testing, aspect-oriented software development, feature-oriented programming, software product lines, software maintenance, and adaptive systems.

Bibliographic data

Cafeo, Bruno Barbieri de Pontes

On the Relationship between Feature Dependencies and Change Propagation / Bruno Barbieri de Pontes Cafeo; advisor: Alessandro Fabricio Garcia. — 2015.

167 f.; 30 cm

Tese (doutorado) - Pontifícia Universidade Católica do Rio de Janeiro, Departamento de Informática, 2015.

Inclui bibliografia.

1. Informática – Teses. 2. Dependência de características. 3. Propagação de mudanças. 4. Linha de produto de software. 5. Manutenção de software. 6. Interface de características. I. Garcia, Alessandro Fabricio. II. Pontifícia Universidade Católica do Rio de Janeiro. Departamento de Informática. III. Título.

CDD: 004

Acknowledgments

First and foremost I want to thank God for being my supervisor during the whole life.

My deep gratitude goes to my advisor Prof. Alessandro Garcia for the continuous support of my Ph.D. study, for his patience, motivation, and immense knowledge. His unwavering enthusiasm kept me constantly engaged with my research and make my time at PUC-Rio enjoyable.

My sincere thanks also goes to Prof. Jaejoon Lee, and Prof. Sven Apel, who provided me opportunities to join their team, and who gave access to their laboratories, research facilities, and to an unspeakable life experience abroad. Without their precious support it would not be possible to conduct this research.

I would like to thank my thesis committee for their insightful comments and encouragement, but also for the hard questions which incented me to widen my research from various perspectives.

I thank my fellow labmates in Brazil, UK, and Germany for the stimulating discussions, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last four years. In particular, I am grateful to Alessandro, Alexander, Benjamin, Bruno, Camila, Claus, Chico, Diegos, Eiji, Elder, Everton, Isela, Leonardo, Luciano, Manuele, Marcelo, Roberta, Roberto, and Willian. Also I thank my friends from LES and PUC-Rio for helping me sustain a positive atmosphere in which to do science. Particular thanks to Alexandre, Carol, Chico, Davy, Eduardo, Ingrid, Jose, Marx, Soeli, and Vera.

I am grateful for the time spent with flatmates, backpacking buddies, and new friends from Brazil, UK, and Germany. A special thanks to my new and old friends of a lifetime Alan, Alexandre, Ana Carolina, Ana Luiza, Andre, BabyRuth, Camila, Carol, Cristiano, David, Douglas, Diego, Fabiano, Gabriel, Geovane, Guilhermes, Jessica, Jonas, Juliana, Larissa, Leoni, Marcela, Marcus, Mihaela, Rachel, Simone, Sofia, Sven, Thiago, Tiago, Vanessa, and Vinicius to endure the ups and downs of my long journey together.

A special thanks to Bianca for her priceless support during the hardest moment of my journey, and for incented me to strive towards my goal. Thanks for spent sleepless nights with me and for always supporting me in the moments when there was no one to answer my queries.

Above ground, I am indebted to my family, whose value to me only grows with age. Words cannot express how grateful I am to my father, mother, and brother, who blessed me with a life of joy in the hours when the lab lights were off. Your prayer for me was what sustained me thus far. My deep gratitude to the sacrifices that you have made on my behalf during my whole life. None of this would have been possible without you.

Finally, I gratefully acknowledge CAPES and CNPq for the funding that made my PhD work possible.

Abstract

Cafeo, Bruno Barbieri de Pontes; Garcia, Alessandro Fabricio (advisor). **On the Relationship between Feature Dependencies and Change Propagation**. Rio de Janeiro, 2015. 167p. DSc Thesis — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Features are the key abstraction to develop and maintain software product lines. A challenge faced in the maintenance of product lines is the understanding of the dependencies that exist between features. In the source code, a feature dependency occurs whenever program elements within the boundaries of a feature's implementation depend on elements external to that feature. Examples are either attributes or methods defined in the realisation of a feature, but used in the code realising other features. As developers modify the source code associated with a feature, they must ensure that other features are consistently updated with the new changes – the so-called change propagation. However, appropriate change propagation is far from being trivial as features are often not modularised in the source code. In this way, given a change in a certain feature, it is challenging to reveal which (part of) other features should also change. Change propagation becomes, therefore, a central and non-trivial aspect of software product-line maintenance. Developers may overlook important parts of the code that should be revised or changed, thus not fully propagating changes. Conversely, they may also unnecessarily analyse parts that are not relevant to the feature-maintenance task at hand, thereby increasing the maintenance effort or even mis-propagating changes. The creation of a good mental model based on the structure of feature dependencies becomes essential for gaining insight into the intricate relationship between features in order to properly propagate changes. Unfortunately, there is no understanding in the state of the art about structural properties of feature dependencies that affect change propagation. This understanding is not yet possible as: (i) there is no conceptual characterisation and quantification means for structural properties of feature dependency, and (ii) there is no empirical investigation on the influence of these properties on change propagation. In this context, this thesis presents three contributions to overcome the aforementioned problems. First, we develop a study to understand change propagation in presence of feature dependencies in several industry-strength product lines. Second, we propose a measurement framework intended to quantify structural properties of feature dependencies. We also develop a study revealing that conventional metrics typically used in previous research, such as coupling metrics, are not effective indicators of change propagation in software product lines. Our proposed metrics consistently outperformed conventional metrics. Third, we also propose a method to support change propagation by facing the organisation of feature dependency information as a clustering problem. We evaluate if our proposed organisation has potential to help developers to propagate changes in software product lines.

Keywords

Feature Dependency; Change Propagation; Software Product Lines; Maintenance; Feature Interface.

Resumo

Cafeo, Bruno Barbieri de Pontes; Garcia, Alessandro Fabricio. **Investigando o Relacionamento entre Dependência de Características e Propagação de Mudanças**. Rio de Janeiro, 2015. 167p. Tese de Doutorado — Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Características são abstrações-chave para o desenvolvimento e manutenção de linhas de produto de software. Um desafio encarado na manutenção de linhas de produto de software é o entendimento das dependências que existem entre características. No código-fonte, uma dependência entre características ocorre sempre que um elemento de programa dentro dos limites de implementação de uma característica depende de elementos externos a esta característica. Exemplos são atributos ou métodos definidos na implementação de uma característica, mas utilizados no código responsável pela implementação de outra característica. A medida que desenvolvedores modificam o código-fonte associado com uma característica, eles devem garantir que outras características são consistentemente atualizadas com as novas mudanças – as chamadas propagações de mudanças. No entanto, a apropriada propagação de mudanças não é uma tarefa trivial, pois características geralmente não são modularizadas no código-fonte. Dessa forma, dado uma mudança em uma determinada característica, é desafiador revelar quais (partes de) outras características também devem ser alteradas. Propagação de mudanças se torna, portanto, um aspecto central e não-trivial da manutenção de linhas de produto de software. Desenvolvedores podem negligenciar partes importantes de código que deveriam ser revisadas ou alteradas, portanto não propagando mudanças de forma completa. Por outro lado, desenvolvedores também podem analisar de forma desnecessária partes de código que não são relevantes para a tarefa de manutenção de características, portanto aumentando o esforço de manutenção ou até propagando mudanças de forma indevida. A criação de um bom modelo mental da estrutura das dependências de características se torna essencial para ganhar compreensão sobre o complexo relacionamento de características com o objetivo de propagar mudanças de forma apropriada. Infelizmente, não existe entendimento no estado-da-arte sobre propriedades estruturais de dependências de características que afetam a propagação de mudanças. Este entendimento ainda não é possível, pois: (i) não existe meios de caracterização e quantificação para propriedades estruturais de dependências de características, e (ii) não existem investigações empíricas sobre a influência dessas propriedades na propagação de mudanças. Nesse contexto, esta tese apresenta três contribuições para superar os problemas acima mencionados. Primeiro, foi desenvolvido um estudo para entender a propagação de mudanças na presença de dependência de características em várias linhas de produto industriais. Segundo, é proposto um arcabouço de medidas com o propósito de quantificar propriedades estruturais de dependências de características. Nesse contexto, também foi desenvolvido um estudo revelando que métricas convencionais tipicamente aplicadas em trabalhos de linha de produto, tais como a métrica de acoplamento, não são indicadores efetivos da propagação de mudanças em linhas de produto de software. As métricas propostas nesta tese superaram consistentemente as métricas convencionais estudadas. Terceiro, esta tese propõe um método para auxiliar a propagação de mudanças encarando informações sobre a organização de dependência de características encarando-as como um problema de agrupamento. Foi avaliado se a organização de informações proposta nesta tese tem potencial para auxiliar desenvolvedores a propagar mudanças em linhas de produto de software.

Palavras-Chave

Dependência de Características; Propagação de Mudanças; Linha de Produto de Software; Manutenção de Software; Interface de Características.

Contents

List of Figures

List of Tables

1
Introduction

Software product lines have gained considerable attention in recent years both in industry and in academia. Product-line software engineering empowers high-level software reuse by exploiting commonality and variability among the member products of a product line [YE and LIU 2005]. To achieve these goals, product-line approaches decompose software into units of functionality called features. Features are cohesive units of behaviour in a software, and serve to express commonalities and variabilities in product lines [POHL *et al.* 2005, APEL *et al.* 2013]. For example, in a mobile operating system, individual products share a common set of features (e.g. phone call and text message), but differ in other features (e.g. screen resolution or media management).

Software maintenance is an inevitable activity in the lifecycle of all types of software systems, including software product lines. All successful software projects stimulate user-generated requests for change [DAM *et al.* 2006]. As in any type of software, a continual change effort is also needed in product-line's life cycle. In particular, an important concern in product-line maintenance is to preserve consistency between features by propagating changes in their implementation [THAO 2012]. As developers modify the source code associated with a feature, such as methods or attributes, they must ensure that other features are consistently updated with the new changes.

However, appropriate change propagation is far from being trivial as features are often not modularised in the source code [APEL *et al.* 2013]. Given a change in a certain feature, it is challenging to reveal which other features should also change. As a single feature is often not localised in a single module in the implementation [KÄSTNER *et al.* 2011], developers cannot simply resort to module dependencies to infer to which features should be changed. Change propagation becomes, therefore, a central and non-trivial aspect of software product-line maintenance.

Several studies [ARNOLD and BOHNER 1993, PARNAS 1994, BROOKS 1995, ARNOLD 1996, RAJLICH 1997, ZHIFENG and RAJLICH 2001, HASSAN and HOLT 2004] elaborate on the difficulties of propagating changes in the presence of module dependencies in stand-alone programs (i.e., non-product-line programs). To cite just a few, Arnold and Bohner present several models of change propagation that assist the propagation of changes based on module dependencies and slicing techniques [ARNOLD and BOHNER 1993]. Rajlich proposes a method to support the change process by using inconsistent module dependencies, i.e. simulation of mistakenly propagated changes [RAJLICH 1997]. Parnas tackled the issue of software aging and warned about modifications performed in the source code by developers, who are not sufficiently knowledgeable of the structure of the program which involves, among others, module dependencies [PARNAS 1994]. Although valuable, these and other studies about change propagation in stand-alone programs do not address the particularities of software product lines. The units of decomposition in software product lines are features rather than modules. As a single feature is often not localised in a single module, approaches for propagating changes in stand-alone programs are not applicable to software product lines.

In the context of software product lines, Ribeiro et al. identified that not fully propagating changes across features may cause an increase in the maintenance effort [RIBEIRO *et al.* 2011]. This study indicates a high complexity in propagating changes in software product lines due to feature dependencies. A *feature dependency* occurs in the source code of a product line whenever one or more program elements (e.g. code blocks, methods or attributes) within the boundaries of a feature depend on elements external to that feature [RIBEIRO *et al.* 2011]. A simple example is an attribute defined in one feature and used in another feature. In this way, a dependency between two features, differently from a dependency between two modules, is scattered through the source code of multiple modules of a program. Thus, exploring the complexity of feature dependencies is essential to bridge the existing gap between change propagation and software product lines.

In fact, feature dependency is not an unusual phenomenon in software product lines. In an exploratory study (Chapter 3), we observed that a widely-used product line (MobileRSS [MOBILE RSS 2015]) presented a total of 235 feature dependencies distributed over the source code of 31 features. The high number of feature dependencies (average of 7.5 dependencies per feature) is not an exception amongst product lines commonly used in other studies [KÄSTNER *et al.* 2008, KÄSTNER and APEL 2009, RIBEIRO *et al.* 2011]. In another study, Ribeiro et al. [RIBEIRO *et al.* 2011] showed that almost 66% of code blocks related to features present code realising feature dependency. So, it is reasonably common to find feature dependency along the product-line code.

As aforementioned, the boundaries of a feature often do not align with the modules of the program (i.e., the features crosscut the program's module boundaries) [KÄSTNER *et al.* 2011]. Program elements establishing a single feature dependency may be distributed all over the code of many modules. As a consequence, these program elements tend to vanish from developer's eyes during a maintenance [RIBEIRO *et al.* 2014]. So, developers are likely to ignore consciously (or not) feature dependencies while reasoning about feature maintenance. Moreover, the implementation of feature dependencies may involve several parts of code possibly affected by a change propagation. The more complex is the structure of a feature dependency, the more influential it might be in change propagation. So, the *structural properties* of feature dependency implementation are likely to impact on the way changes propagate. Structural properties of feature dependencies are characteristics related to the way a feature dependency is implemented in the source code, such as number of program elements involved in a feature dependency. In this way, exploring possible reasons for change propagation, such as structural properties of feature dependency may be primordial to support change propagation.

Recently, the research community began discussing the harmful impact of feature dependency on product-line maintenance [FERBER *et al.* 2002, LEE and KANG 2004, APEL and BEYER 2011, RIBEIRO *et al.* 2011, GARVIN and COHEN 2011, CATALDO and HERBSLEB 2011]. A lot of effort spent during code maintenance is devoted to comprehend dependencies and revisit code of potentially affected features [RIBEIRO *et al.* 2011]. However, there is a lack of studies exploring the relation between these two important phenomena – i.e., change propagation and feature dependencies. It is challenging to fully recognise the implementation of feature dependencies by simply analysing the source code. In this case, developers may overlook important parts of the code that should be revised or changed, thus not fully propagating changes. Conversely, they may also analyse unnecessarily parts that are not relevant to the feature-maintenance task at hand, thereby increasing the maintenance effort or even mis-propagating changes [RIBEIRO *et al.* 2011]. Given all the aforementioned reasons, there remains a need to explore (i) whether feature dependencies are frequent propagators of changes, (ii) whether all feature dependencies of a product-line are alike, (iii) which characteristics of feature dependencies are more important regarding their impact on change propagation, and (iv) what can be done to alleviate the effort of propagating changes in software product lines.

The remainder of this chapter is organised as follows. Section 1.1 defines the problem tackled in this thesis based on a running example. Section 1.2 describes the goal, subgoals and research questions. Section 1.3 presents the thesis contributions. Finally, Section 1.4 points out how this thesis is organised.

1.1 Problem Statement

In the following, we present a motivational example using an annotative approach called conditional compilation (Chapter 2). Conditional compilation is the most widely used mechanism to implement product-line features [KÄSTNER and APEL 2009]. A feature is a set of program elements surrounded by preprocessor directives. The preprocessor identifies the code that should be compiled or not based on preprocessor directives (i.e. `#ifdef` directives). The example presented here will serve to illustrate the problem governing the relation of feature dependency and change propagation. Even though in our example we use annotative approach, the problem statement presented in this section is also found in compositional approaches, such as aspect-oriented programming. However, these approaches are not widely used as annotative approaches to implement software product lines.

Figure 1.1 illustrates a code snippet of how a feature is represented in the program. In this source code (Figure 1.1), we show part of the feature code WEIGHTED of a product line to tailor graph data structures, including weighted edges and traversal algorithms. The selection of the feature WEIGHTED in a product allows to associate weight with graph edges. In this excerpt of code, we show the declaration of the attribute responsible for representing the weight of an edge in a graph (lines 3–5), and the methods responsible for getting and setting this weight (lines 7–18). It is worth to notice that the method `setWeight(int x)` (lines 11–18) only assigns positive values to weights. In addition, there is a method `foo()` (line 19) added only for illustrative purpose.

As aforementioned, features relate to each other in a product line. Whenever one or more program elements within the boundaries of a feature depend on elements external to that feature, we consider this as a feature dependency. So, in the source code, feature dependency implementation refers to the program elements in the source code responsible for establishing a relationship between features. Figure 1.2 illustrates an example of the implementation of part of a feature dependency in the product line. The code snippet in the figure illustrates part of the feature code DIJKSTRA. This

```
01. public class Edge{
02.  ...
03.  #ifdef WEIGHTED
04.   int weight;
05.  #endif
06.  ...
07.  #ifdef WEIGHTED
08.   public int getWeight(){
09.    return weight;
10.   }
11.   public void setWeight(int w){
12.    if (w==0)
13.     weight = Integer.MAX_VALUE;
14.    else if (w<0)
15.     weight = -1 * w;
16.    else
17.     weight = w;
18.   }
19.   public void foo(){...}
20.  #endif
21. }
```

Figure 1.1: Part of the feature code WEIGHTED of a product line of graph libraries.

feature addresses the shortest path problem for a non-negative edge path of a weighted directed graph [DIJKSTRA 1959]. Part of the feature dependency of this example is established by the method call to `getWeight()` (italic and underlined in line 05, Figure 1.2). The method `getWeight()` is a method that belongs to feature WEIGHTED (lines 8–10, Figure 1.1). So, a part of the DIJKSTRA feature code depends on elements of other feature, thus establishing a dependency of feature DIJKSTRA on feature WEIGHTED. The other method participating of the same dependency is the method call to `foo()` (italic and underlined in line 04, Figure 1.2). One should notice that only a part of the dependency was shown. The feature dependency of DIJKSTRA on WEIGHTED may be realised by other elements in several parts of the code comprising the DIJKSTRA feature. For instance, these parts could contain other method calls and attribute access to members of the WEIGHTED feature. All program elements contributing to the feature dependency of DIJKSTRA on WEIGHTED are the considered as the feature dependency implementation of DIJKSTRA on WEIGHTED[1].

Now, let us suppose the addition of a new feature BELLMAN-FORD (lines 9–13, Figure 1.3). This feature, similarly to feature DIJKSTRA, implements an algorithm that computes the shortest path in a weighted

[1] From hereafter, unless otherwise stated, feature dependency is used to refer to feature dependency implementation

```
01. public class Algorithm{
02.  ...
03.  #ifdef DIJKSTRA
04.   obj.foo();
05.   calculatePath(obj.getWeight());
06.   ...
07.  #endif
08.  ...
09. }
```

Figure 1.2: Part of the feature code DIJKSTRA with the realisation of a feature dependency.

directed graph. However, this feature is able to handle graphs with negative weight edges [BELLMAN 1958]. Figure 1.3 illustrates the inclusion of part of the BELLMAN-FORD feature code. It is important to notice that the feature responsible for implementing the Bellman-Ford algorithm also depends on feature WEIGHTED, similarly to feature DIJKSTRA. Part of the feature dependency of BELLMAN-FORD on WEIGHTED is realised by means of the method call to `getWeight()` (italic and underlined in line 11). It is important to mention that this method call is one of many other relationships between program elements in the source code that may comprise the feature dependency of BELLMAN-FORD on WEIGHTED.

```
01. public class Algorithm{
02.  ...
03.  #ifdef DIJKSTRA
04.   obj.foo();
05.   calculatePath(obj.getWeight());
06.   ...
07.  #endif
08.  ...
09.  #ifdef BELLMAN-FORD
10.   ...
11.   calculatePath(obj.getWeight());
12.   ...
13.  #endif
14.  ...
15. }
```

Figure 1.3: Partial code of the added feature BELLMAN-FORD emphasising the realisation of part of the feature dependency.

Due to all capabilities of the Bellman-Ford algorithm, the developer needs to allow the assignment of negative weight values to edges in the product line. So, it is necessary to change the method `setWeight(int x)` in order to assign negative weight values to edges. Figure 1.4 shows the method `setWeight(int x)` of feature WEIGHTED after the change. Moreover, since the Dijkstra algorithm is not able to handle negative weight values of edges, a change in the DIJKSTRA feature code must also be made to handle negative values. In our example, the developer decided to consider the weights as unsigned values.

Figure 1.6 illustrates a possible change in the code of the feature DIJKSTRA (lines 3–10) compared to the code of Figure 1.5.

```
01. ...
02. #ifdef WEIGHTED
03.  ...
04.  public void setWeight(int w){
05.   if (w==0)
06.    weight = Integer.MAX_VALUE;
07.   else
08.    weight = w;
09.  }
10. #endif
11. ...
```

Figure 1.4: Change in the method `setWeight(int x)`.

The change in feature DIJKSTRA happened because there was a change propagation through the dependency during the maintenance of feature WEIGHTED. In a general sense, change propagation is the set of changes required to other entities of a software system to ensure the consistency after a change in a particular entity [HASSAN and HOLT 2004]. In product lines, change propagation happens when a change in the code associated with a feature might imply changes on one or more dependent features. These changes are necessary to preserve the consistency between related features. In the context of our example, the change in feature WEIGHTED demanded a change in the dependent feature DIJKSTRA to ensure the consistency of the product-line. Moreover, as aforementioned, the feature dependencies could involve several program elements possibly affected by a change propagation.

In this example, only a part of the source code of the feature dependency

```
01. public class Algorithm{
02.  ...
03.  #ifdef DIJKSTRA
04.   obj.foo();
05.   calculatePath(obj.getWeight());
06.   ...
07.  #endif
08.  ...
09.  #ifdef BELLMAN-FORD
10.   ...
11.   calculatePath(obj.getWeight());
12.   ...
13.  #endif
14.  ...
15. }
```

Figure 1.5: Before the change in the feature DIJKSTRA.

```
01. public class Algorithms{
02.  ...
03.  #ifdef DIJKSTRA
04.   int w = obj.getWeight();
05.   ...
06.   if (w<0)
07.    w = -1*w;
08.   calculatePath(w);
09.   ...
10.  #endif
11.  ...
12.  #ifdef BELLMAN-FORD
13.   ...
14.   calculatePath(obj.getWeight());
15.   ...
16.  #endif
17.  ...
18. }
```

Figure 1.6: After the change in the feature DIJKSTRA.

was shown. The implementation of this feature dependency may be more complex. More program elements could be affected by the change propagation. Structural properties are crucial in this case because, for instance, a feature dependency involving many program elements may cause a severe change propagation through many parts of the source code. In addition, one can notice that certain structural properties of feature dependency may be related to the complexity of the change propagation. For instance, if the feature dependency of DIJKSTRA on WEIGHTED involves many elements, any change in WEIGHTED code is likely to impact on DIJKSTRA code. In other words, changes in feature WEIGHTED are more likely to be propagated to dependent features participating of a dependency involving many elements. The likelihood of the change propagation through the dependency of DIJKSTRA on WEIGHTED may be higher than in the dependency of BELLMAN-FORD on WEIGHTED. Put differently, it may happen a concentration of changes propagated through the dependency of DIJKSTRA on WEIGHTED. Thus, the change propagated through feature dependencies may be not all alike due to certain structural properties of feature dependency. This case evidence the importance of studying the relation between those structural properties of feature dependencies and the complexity of change propagation.

In this example, it is important to highlight and understand the steps followed during the maintenance. When a feature code is changed, the first step is to identify and comprehend dependencies. The next step is to revisit code of potentially affected features. Structural properties of feature dependency make these steps challenging. Developers may not revise or change important parts of the code. In addition, they may also unnecessarily analyse parts that are not relevant to the maintenance task. Finally, it is important to be aware about the differences of changes propagated by different feature dependencies. Structural properties of feature dependency may impact on change propagation by, for example, increasing the likelihood of propagation in certain dependencies. Therefore, it becomes primordial to explore the relation between feature dependencies and change propagation. This relation must be analysed in terms of structural properties of feature dependencies. As a consequence, measurement framework and solutions to support change propagation can be proposed based on those structural properties.

Section 1.1.1 explains the need for studying feature dependency as change propagators in software product line; Section 1.1.2 discusses the impact of structural properties of feature dependencies on change propagation. Finally, Section 1.1.3 addresses the need of a method for supporting change propagation based on those structural properties.

1.1.1 The Problem of Assessing the Relation between Change Propagation and Feature Dependencies

Propagation of changes is a central aspect of software development [HASSAN and HOLT 2004]. Since features usually relate to each other, once a feature is changed, other features may need to be changed as well in order to preserve consistency. The analysis of the motivating example (Section 1.1) demonstrates the need for propagating changes. In order to preserve consistency, the change of feature WEIGHTED demanded a change in feature DIJKSTRA. Moreover, one can also notice that this change was propagated due to the existence of a feature dependency. However, there is little understanding about changes propagated in the presence of feature dependencies.

To better understand the relation between feature dependencies and change propagation one should know first whether feature dependency often leads to change propagation – i.e. one should identify if feature dependency is a main driver of change propagation. Moreover, changes may propagate differently depending on the feature dependency. For instance, certain feature dependencies may be involved more often in change propagation than other feature dependencies along the evolution, the so-called concentration of changes. The characteristics of those feature dependencies must also be empirically analysed.

In the literature, feature dependencies are often assumed as one of the main drivers of change propagation [FERBER *et al.* 2002, LEE and KANG 2004, RIBEIRO *et al.* 2011]. However, there is a lack of studies about changes propagated through feature dependencies. In other words, there is no understanding about the frequency that changes are propagated through feature dependencies, or even if there is a difference in change propagation regarding certain feature dependencies. Therefore, empirically assessing the relation between feature dependencies and change propagation is the starting point to improve the task of propagating changes in product lines.

Thus, our first problem can be described as follows:

It is unclear the relation between feature dependencies and change propagation.

1.1.2 The Problem of Characterising Structural Properties of Feature Dependency

Propagation of changes in the presence of feature dependencies is a complex task [RIBEIRO *et al.* 2011]. Certain characteristics of feature dependencies may impact on change propagation. The more complex is the structure of a feature dependency, the more influential it might be on change propagation. In other words, certain structural properties of feature dependencies might imply they concentrate more changes along the product-line evolution. The example presented in Section 1.1 illustrates how structural properties of feature dependency can make change propagation quite challenging. In the example, only one element establishing the dependency of DIJKSTRA on WEIGHTED was shown (method `getWeight()`). The implementation of the dependency of DIJKSTRA on WEIGHTED may comprise, for instance, several program elements to establish a dependency. Even worse, these program elements may be scattered through many modules making the proper propagation of changes a difficult task to be accomplished.

In this context, it becomes primordial the need for defining and evaluating indicators to capture various complexity characteristics of the feature dependency structure. However, there is no characterisation of structural properties of feature dependency in literature. As a consequence, there is no knowledge about the relation between such structural properties and change propagation as well as indicators of change propagation in product lines.

Given the aforementioned issues, we consider that there is a need to explore structural properties of feature dependency, since they might be related to change propagation. To do so, firstly we need to characterise structural properties of feature dependencies. This involves an identification of recurrent structural properties of feature dependencies. After that, we need to identify which structural properties exert impact on change propagation. In other words, we need to identify which structural properties of feature dependencies are often related to change propagation. Finally, we need to assess the capability of structural properties indicate change propagation. Based on our findings developers will be able to distinguish feature dependencies by using structural properties. As a consequence, during product-line maintenance, it will be possible to drive efforts to certain dependencies based on their structural properties.

Therefore, our second problem can be stated as:

There is no characterisation of structural properties of feature dependency that may better indicate the occurrence of change propagation.

1.1.3 The Problem of Alleviating the Change Propagation Effort

Creating a good mental model of the structure of a complex system is one of many serious problems of software developers [MANCORIDIS *et al.* 1998]. Frequently, this problem is exacerbated in the context of product lines because a single feature dependency may be scattered and tangled in the implementation of various program elements. Only using structural properties of feature dependencies information (Section 1.1.2) do not support developers to reason about program elements of feature dependencies as well as the organisation of those dependencies.

In Section 1.1, we can note how complex can be to propagate a change in the presence of feature dependencies. The methods `getWeight()` and `foo()` are program elements involved in the dependency of DIJKSTRA on WEIGHTED. Nevertheless, the same method `getWeight()` is involved in the feature dependency of BELLMAN-FORD on WEIGHTED. So, we have two program elements of the feature WEIGHTED being used by two different features. One of the program elements (method `getWeight()`) is involved in two feature dependencies. The other program element is only involved in the feature dependency of DIJKSTRA on WEIGHTED. Therefore, reasoning about the impact of changing elements establishing feature dependencies is challenging. A change affecting the program element which is part of two different feature dependencies (method `getWeight()`) might demand revisiting the code of both dependent features. In this way, revisiting only one of the two dependent features may introduce an inconsistency. In addition, a change affecting the program elements used only by DIJSTRA (method `foo()`) is unlikely to affect both features (DIJKSTRA and BELLMAN-FORD). In this case, revisiting the code of feature BELLMAN-FORD is not needed to propagate a change.

Given the complexity of how features relate to each other [RIBEIRO *et al.* 2011], the number of program elements involved in feature dependencies is higher than in the given example. With no mechanism for gaining insight into the organisation of feature dependencies, developers are often forced to propagate changes to a dependent feature code without a thorough knowledge.

In this way, as product lines tend to change over time, it is inevitable that adopting an ad hoc approach to propagate changes will have a negative effect on the maintenance. A lot of effort may be devoted to comprehend dependencies and revisit code of potentially affected features.

In our attempt to alleviate the change propagation complexity in the presence of feature dependencies, it is necessary a solution that support developers to reason about the organisation of feature dependencies, mainly the ones presenting structural properties highly correlated with change propagation. To do so, we need to propose a technique to support developers creating a model of the structure of feature dependencies. This technique must consider characteristics of feature dependencies that are influential on change propagation. In addition, we need to provide an automated solution that proposes an organisation of feature dependencies information (i.e. program elements realising a feature dependency). After that, we need explore improvements in such representation in a way that our representation help developers to understand the complex organisation of feature dependencies and their possible impact on change propagation. Finally, we need to assess the capability of our solution supports change propagation. Based on our findings, less effort could be devoted to comprehend dependencies and revisit code of potentially affected features.

Thus, our third problem can be described as follows:

It is not known how to organise information related to feature dependency in order to support change propagation.

1.2 Goals and Research Questions

Given the problems described in the previous section, we are able to formulate our research questions. Therefore, the following overall research question (RQ) needs to be answered:

Research Question. *How to support product-line maintainers to propagate changes in the presence of feature dependencies?*

Each of the problems described in Sections 1.1.1, 1.1.2 and 1.1.3 can be translated to a specific research question. Therefore, our overall research question is decomposed in the following three research questions (RQ1, RQ2 and RQ3):

RQ1. *What is the relation between feature dependencies and change propagation?*

The first research question (RQ1) aims at evaluating whether feature dependencies are often related to change propagation. We also analyse how change propagation behaves in the presence of feature dependencies. In order to answer RQ1, a study of various product lines was conducted. Our results have shown that (i) there was often a strong relation between feature dependency and change propagation in those product lines, and (ii) changes propagate differently depending on certain characteristics of feature dependencies (e.g. concentration of changes in certain feature dependencies). The findings and discussion derived from this study are represented in Chapter 3. In particular, we observed that structural properties of feature dependencies could be good indicators of change propagation, which lead us to RQ2.

RQ2. *Are structural properties of feature dependency good indicators of change propagation?*

RQ2.1. *Which structural properties of feature dependency are omnipresent to different product-line implementation approaches?*

RQ2.2. *Which structural properties of feature dependency are good indicators of change propagation?*

The goal of RQ2 is to characterise structural properties of feature dependency that are related to change propagation. To do so, this research question is split in two sub research questions. The first one (RQ2.1) characterises recurrent structural properties of feature dependency. In addition, to completely answer RQ2, the second sub research question (RQ2.2) aims at assessing the effectiveness of structural properties of feature dependency as indicators of change propagation. However, only using structural properties of feature dependencies do not support developers to reason about program elements of feature dependencies as well as the organisation of those dependencies. Creating a good mental model of the structure of feature dependencies becomes primordial to propagate changes, thus leading us to RQ3.

RQ3. *How to organise information of feature dependency implementation to support change propagation?*

In RQ3, we aim at proposing a solution that support developers to reason about the organisation of feature dependencies, mainly the ones presenting structural properties highly correlated with change propagation. In order to answer RQ3, we proposed a way of organising the information of feature dependencies. In addition, we presented a study to evaluate our proposed organisation. Our results indicates that the proposed organisation has potential to (i) reduce the overlooking of important parts of the source code when changes must be propagated, (ii) alleviate problems related to mis-propagated changes (i.e. changes wrongly propagated). In addition, we observed that it may not demand a lot of effort from developers to become familiar with our organised feature interfaces.

1.3 Contributions

This section briefly describes the contributions of this thesis, including: (i) empirical findings revealing the relationship between feature dependencies and change propagation, (ii) identification and characterisation of a set of structural properties of feature dependency, rooted at empirical studies, which exert impact on change propagation, and (iii) a method for organising information of feature dependency implementation in order to support developers on performing change propagation. An overview of each contribution is presented in the following. We indicate in brackets the research question associated with each contribution.

– **Empirical findings on the relation between feature dependencies and change propagation (RQ1).** At first, by means of empirical studies, we aimed at evaluating to what extent feature dependencies are related to change propagation. We also explored whether changes propagate in different ways through feature dependencies. Our intent was to gather initial information about the characteristics of feature dependencies that might be related to change propagation. The study undertaken in this phase (Chapter 3) were carried out using information of feature dependencies and changes in twenty-one releases of five product lines. This study analyses both feature dependencies and change propagation through features during product-line evolution. This analysis enables us to identify the occurrence of change propagation through dependencies. The findings of this study are the following: (i) feature dependency is indeed one of the main drivers of change propagation, and (ii) it reveals that feature dependencies are not alike regarding change propagation. Based on these findings, developers could benefit, for instance, by differentiating feature dependencies and focusing on some with certain characteristics during product-line maintenance. The findings derived from this study are presented in detail in Chapter 3.

– **Feature dependency properties and metrics (RQ2).** Since feature dependencies are main drivers of change propagation in product lines, the need of indicators for feature dependencies in the context of change propagation becomes primordial. Moreover, since feature dependencies are not alike regarding the propagation of a change, characterise feature dependencies becomes essential for a proper change propagation. When feature dependencies are established, certain structural properties of the implementation of these dependencies may exert an impact on change propagation. However, there is no understanding in the literature about the relation between structural properties of feature dependency and change propagation. The studies undertaken in this phase (Chapter 4) of our research explore the role of structural properties of feature dependency on change propagation. Therefore, we identified recurring structural properties of feature dependency. We also explored the relation between such properties with change propagation. In addition, we defined metrics based on these properties. Finally, we identified the metrics that presented high correlation with change propagation. These findings are particularly important to developers as they can use the structural properties of feature dependencies as indicators of change propagation.

– **A method to organise information about feature dependency implementation (RQ3).** Understanding the intricate relationship that exist between features can be an arduous task. With no mechanism for gaining insight into feature dependencies organisation, developers are often forced to propagate changes to a dependent feature code without a thorough knowledge of the feature dependency organisation. In our attempt to alleviate the change propagation complexity, we propose an automated feature interface organisation in order to support developers creating a model of the structure of feature dependencies

(Chapter 5). The feature interface organisation is faced as a clustering problem. We propose an underlying model to represent the relationship between feature interface members. We also propose the use of a specific clustering algorithm for graphs based on simulation of stochastic flow in graphs (The Markov Clustering algorithm). The study undertaken in this phase were carried out using 10 releases of a software product line comprising almost 2,000,000 LOC in 6858 source code files. This study aims at analysing how close the clusters proposed by the algorithm are from the real simultaneous changes of feature interface members during product-line evolution. This step is particularly important because it provides empirical support to argue that feature interface should be organised in order to support change propagation. This study is related to RQ3. Based on this finding, developers may reduce the overall number of program elements revisited during change propagation. The findings derived from this study are presented in detail in Chapter 5.

1.4 Outline

The structure of the thesis is divided into seven different chapters. Chapter 1 has presented the introduction of the thesis, defines the problems tacked as well as the research questions, and the main contributions. In Chapter 2, we review the essential concepts explored in this thesis as well as some related work. Chapter 3 studies the relation between feature dependencies and change propagation. In Chapter 4 we explore strucutural properties of feature dependency on change propagation. Feature dependency properties are identified and related to change propagation for further use in algorithms for organising information about feature dependency implementation. Chapter 5 explore a method for organising information of feature dependencies. We define an underlying model considering results achieved in Chapters 3 and 4. After that, we present a solution to organise feature dependency information and also an evaluation of the proposed organisation. Finally, Chapter 6 shows the final considerations, research contributions, and directions for future research.

2 Background and Related Work

To lay a foundation for this work, this chapter introduces the essential concepts explored in our research. First, we review the concepts of software product lines in Section 2.1. We also present and discuss in detail the concept of features (Section 2.1.1), and feature dependency in product lines (Section 2.1.2) which are main concerns of interest when maintaining product-line source code. We also reviewed the different approaches for product-line implementation. In particular, we explain how features and their dependencies are implemented in these approaches (Section 2.2). Section 2.3 presents the concept of change propagation. Section 2.4 presents related work about: (i) the impact of feature dependency on program maintainability (Section 2.4.1), (ii) metrics for modular programming important for the understanding of this thesis (Section 2.4.2), and (iii) modularity of features (Section 2.4.3). Finally, Section 2.5 summarises this chapter.

2.1 Software Product Lines

Nowadays, stand-alone systems are the target of most of the software development approaches [RIBEIRO *et al.* 2011]. A stand-alone system consists of a single monolithic software, which addresses all the stakeholder's requirements. However, the use of stand-alone systems is not convenient for large scale software production [RIBEIRO *et al.* 2011]. Each group of stakeholders may need different requirements. Therefore, specific stakeholder's needs for software systems demand a streamlined process for generating customised products. To improve this process, instead of having only one monolithic system with all functionalities, it is possible to define altogether a set of systems as a customisable system. This set contains similar systems differentiated by features [CLEMENTS and NORTHROP 2001] (Section 2.1.1). This set of systems enables the treatment of specific customers' needs more easily as each customer is interested only in a specific subset of features.

The use of software product lines often achieves the aforementioned systematic construction of software systems with mass customisation. A software product line is *a set of software-intensive systems that share a common set of features satisfying the specific needs of a particular market segment or mission, and that are developed from a common set of core assets in a prescribed way* [CLEMENTS and NORTHROP 2001]. Thus, instead of having one stand-alone software, customers choose their particular product configuration. The reuse of the set of product-line's core assets is responsible for generating the products. This reuse brings significant improvements in the development process, such as reduction of development costs, enhancement of quality and reduction of time-to-market[CLEMENTS and NORTHROP 2001, POHL *et al.* 2005].

2.1.1 Feature

The software decomposition into features is essential to reap all of the product-line benefits. Features are units by which different products within a product line can be differentiated and defined [TRUJILLO *et al.* 2006], playing a key role for mass customisation. According to Kästner and colleagues, there are different views on the characterisation of what is a feature [KÄSTNER *et al.* 2011]. In the literature of software product lines, a feature is often characterised based on a syntactic perspective [APEL and KÄSTNER 2009]. In this perspective, features are characterised as cohesive units of behaviour in the implementation. Other authors characterise features using a semantic perspective, which is inspired by early work on telecommunication systems [JACKSON and ZAVE 1998, HAY and ATLEE 2000, FISLER and KRISHNAMURTHI 2008]. This perspective defines feature as semantic units of software systems, which interact and give rise to the observable behaviour a user is interested. In this perspective, the implementation of a feature is of less interest; instead, it focuses on the meaning of features and their interactions to its prospective users.

These two perspectives focus on different levels of abstraction, thereby different characteristics of features. In this thesis, we adopt the syntactical perspective since we are focusing on the implementation of features. In particular, we adopt the definition given by Apel et al. and Pohl et al., which state that a feature is *a cohesive unit of behaviour of software systems* [POHL *et al.* 2005, APEL *et al.* 2008].

2.1.2 Feature Dependency

A feature dependency defines *how the implementation of a feature is dependent on the implementation of other features.* A feature dependency occurs in the source code of a product line whenever one or more program elements (e.g. code blocks, methods or attributes) within the boundaries of a feature depend on elements external to that feature [RIBEIRO *et al.* 2011]. A simple example is an attribute defined in one feature and used in another feature.

A dependency between two features, differently from a dependency between two modules, may be scattered and tangled in the implementation of various modules of a program. This misalignment happens because features are often not modularised in the source code [APEL *et al.* 2013]. In other words, the boundaries of a feature often do not align with the modules of the program [KÄSTNER *et al.* 2011]. Program elements establishing a single feature dependency may be distributed all over the code of many modules. Section 2.2 illustrates how features and feature dependencies are implemented using different implementation approaches. This misalignment between features and modules also affects the implementation of feature dependencies. Structural properties of feature dependencies are *characteristics related to the way a feature dependency is implemented in the source code.* Structural properties of feature dependencies are related to, for instance, the number of program elements involved in a feature dependency, the number of modules in which the feature dependency is found, and the like. We provide a more systematic definition of feature dependency and its structural properties in Chapter 4. In this chapter, we will provide a terminology that is required in our measurement framework.

To the context of this thesis, it is also important to define the concept of transitive feature dependency (a.k.a. indirect dependency). A transitive feature dependency happens whenever a feature C depends on a feature B, and B is in turn dependent on a feature A; then C depends on A by transitivity. In addition, we can say there is a path of dependencies from C to A with two units of distance. We provide a detailed example of the concept of transitive feature dependency in Chapter 3. In this chapter, we will also consider transitive feature dependencies in the presented study.

2.2 Product-line Implementation Approaches

Existing approaches for product-line implementation can be categorised either as a compositional or as an annotative approach [KÄSTNER *et al.* 2008]. Sections 2.2.1 and 2.2.2 summarise each of them, respectively. Section 2.2.3 shows an example of implementation of an illustrative product line in each approach.

2.2.1 Compositional Approaches

The use of compositional approaches to implement product lines is justified by the possibility of having the maximum of feature implementation separated as distinct code units. To generate a product, code units are determined and composed, often at compile-time. There are many techniques that follow the compositional approaches, such as component technologies [SZYPERSKI 2002], mixin layers [SMARAGDAKIS and BATORY 1998], AHEAD [BATORY *et al.* 2003] and aspects [KICZALES *et al.* 1997].

Compositional approaches have the advantage of improving the modularity of features by trying to separate code implementing each feature. Therefore, this approach can achieve a high degree of alignment between each feature and a module. The code associated with one feature is often not tangled with code realising other features. However, one of the biggest problems of compositional approaches is that they are unable to implement product lines at a fine level of granularity [KÄSTNER *et al.* 2008]. In a fine level of granularity, developers are able to implement features by annotating the code at the level of attributes, operations, code blocks, and declarations. This limitation makes compositional approaches popular in academy, but hardly used in industrial projects so far [KÄSTNER *et al.* 2011].

2.2.2 Annotative Approaches

Annotative approaches implement features by annotating the source code. Preprocessor directives of conditional compilation is the most widely used mechanism in industry to implement software product lines using annotative approaches [KÄSTNER and APEL 2009]. The implementation of features are characterised by the use of preprocessor directives, such as `#ifdef` and `#endif`, to encompass the code associated with one or more features. Examples of tools for annotative approaches are pure::variants [BEUCHE *et al.* 2004], Gears [KRUEGER 2001] and Frames/XVLC [JARZABEK *et al.* 2003].

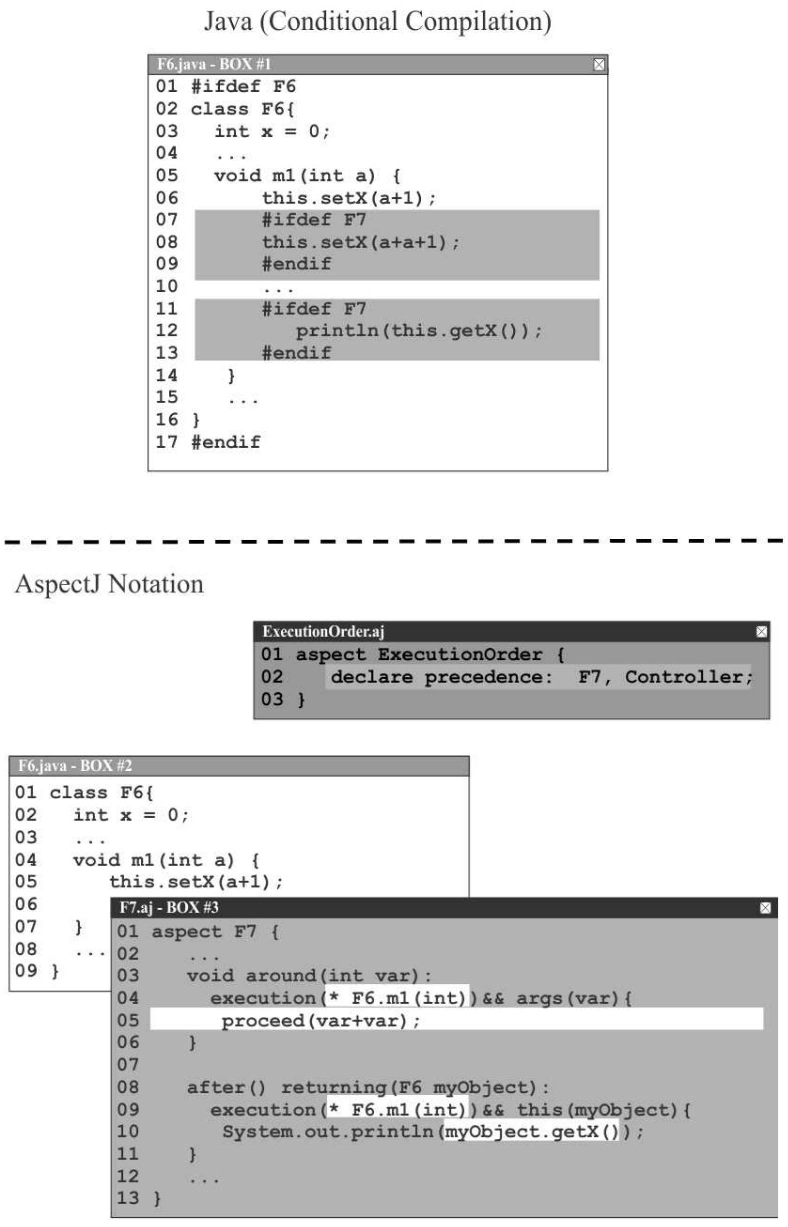
In contrast to compositional approaches, annotative approaches support the implementation of product-line features at fine level of granularity. In other words, the annotation of feature code scales from entire files to small code fragments. However, the code associated with a single feature is often scattered across the code base and tangled with other features' implementation. Therefore, annotative approaches lead to a lack of feature modularity [KÄSTNER *et al.* 2008, GACEK and ANASTASOPOULES 2001, ERNST *et al.* 2002]. Despite this well-known drawback, annotative approaches are commonly used because they are easy to learn and they are less intrusive to the development process [CLEMENTS and NORTHROP 2001]. In addition, developers can flexibly define on demand the location of each feature after they have already structured and implemented the modules of their programs.

2.2.3 Implementation of Product Lines

This section presents an example of the implementation of a software product line. We illustrate the implementation of features and feature dependencies. The example uses both implementation approaches presented in Sections 2.2.1 and 2.2.2 to implement the same illustrative software product line. We are considering Java with preprocessor directives as a representative of the annotative approach, which is a well-known and industry-strength technique for supporting feature implementation. Moreover, we are considering AspectJ as a representative of the compositional approach. AspectJ was chosen because it is a well-known language for aspect-oriented programming (AOP).

The implementation of product-line features using conditional compilation relies on the use of pre-processors directives like #ifdef and #endif. The annotated code fragments, enclosed by compilation directives, are addressed or removed from the source code depending on the selection of features. On the other hand, AspectJ supports the implementation of separate modules, the so-called aspects. Aspects have the ability to improve the modularity of crosscutting features that might spread their behaviours throughout the base code. The advice is the mechanism often used to implement features. Advices are programming units intended to implementing crosscutting behaviours. AspectJ supports three kinds of advice: before, around, and after. The kind of advice determines how it interacts with a well-defined point in the execution flow of a program, the so-called join points. Figure 2.1 exemplifies the implementation of features and feature dependencies.

In conditional compilation, the implementation of the features F6 and F7 are in the same file (f6.java, BOX #1). The implementation of feature F6 is realised by the white lines, and the code associated with feature F7 is coloured in grey. Blocks of #ifdef statements (BOX #1 – lines 07 and 11) associated with feature F7 are enclosing the code associated with feature F6. This fact indicates that there is a dependency between F6 and F7 due to the use of the attribute `a` (BOX #1 – line 08), the method `setX` (BOX #1 – line 08) and the method `getX` (BOX #1 – line 12).



```
01 #ifdef F6
02 class F6{
03   int x = 0;
04   ...
05   void m1(int a) {
06       this.setX(a+1);
07       #ifdef F7
08       this.setX(a+a+1);
09       #endif
10       ...
11       #ifdef F7
12          println(this.getX());
13       #endif
14    }
15    ...
16 }
17 #endif
```

```
01 aspect ExecutionOrder {
02    declare precedence:  F7, Controller;
03 }
```

```
01 class F6{
02   int x = 0;
03   ...
04   void m1(int a) {
05      this.setX(a+1);
06
07   }
08   ...
09 }
```

```
01 aspect F7 {
02    ...
03    void around(int var):
04      execution(* F6.m1(int))&& args(var){
05       proceed(var+var);
06    }
07
08    after() returning(F6 myObject):
09      execution(* F6.m1(int))&& this(myObject){
10       System.out.println(myObject.getX());
11    }
12    ...
13 }
```

Figure 2.1: Example of approaches for implementing product lines.

In AspectJ, the implementation of feature F6 is in one file (f6.java, BOX #2) whereas the implementation of feature F7 is in another file (f7.aj, BOX #3). An advice (BOX #3 – lines 03 and 08) is responsible for realising the dependency between F6 and F7. The definition of the value for the attribute `x` is performed by an advice around (BOX #3 – line 03), which changes the value of the argument of the method `m1`, and returns the control flow to `m1`. Moreover, the value of `x` is printed after the execution of `m1` through the advice after (BOX #3 – line 08), which calls the method `getX` (BOX #3 – line 10) from F6.

2.2.4 Comparison between Approaches

After introducing the two groups of approaches, we compare them based on several characteristics highlighted by Kästner et al. [KÄSTNER and APEL 2008]. However, it is important to mention that we focus on annotative approaches in all studies of this thesis. Annotative approaches are the most widely used approach in industry to implement product-line features [KÄSTNER and APEL 2009]. Nevertheless, we also consider compositional approaches in Chapter 4 in order to evaluate both approaches in terms of feature dependency implementation and its impact on change propagation.

Feature Traceability. Feature traceability is the ability to trace a feature from the domain space to the solution space [CZARNECKI and EISENECKER 2000]. Traceability is important, for instance, when developers want to maintain a specific feature, and thus need to find all the code related to this feature. The compositional approaches directly support feature traceability as the code implementing a feature can be often traced to a single code unit (e.g., aspect and class). In contrast, annotative approach does not provide direct support for feature traceability as feature annotations can be scattered over the entire source code.

Modularity. Compositional approaches provide a good modular reasoning of features. For example, modularity is well supported in components [SZYPERSKI 2002] and hypermodules [TARR *et al.* 1999]. However, compositional approaches like aspect-oriented languages or AHEAD [BATORY *et al.* 2003] are based on source code transformations and provide limited modularity. For instance, separate compilation is not supported in AspectJ [KICZALES *et al.* 1997] or AHEAD. On the other hand, modularity is not intended in annotative approaches. Some tools simulate the modularisation in annotative approaches to some degree. Tools like CIDE [KÄSTNER 2015] offers view and navigation support. However, similar to compositional approach, annotative approaches does not support separate compilation.

Granularity. Coarse-grained approaches only assemble files in a directory or modules, whereas fine-grained approaches scale from annotating entire files to even small code fragments [KÄSTNER *et al.* 2008c]. Compositional approaches only provide a coarse granularity. They usually compose components or introduce methods in existing classes. On the other hand, annotative approaches allow annotations at the level of attributes, operations and declarations. Therefore, it is not possible, for instance, to manipulate statements within a method in compositional approaches due to conceptual limitations [KÄSTNER *et al.* 2008c].

Adoption. According to Kästner et al. [KÄSTNER and APEL 2008], industry is very careful on adopting compositional approaches. These approaches heavily affect the existing development process. At most, after careful planning, frameworks or components are used [BASS *et al.* 2003]. In contrast, the adoption of annotative approaches is much faster. Annotative approaches often introduce only lightweight tools, which do not have a severe impact on the source code structure or on the development process [CLEMENTS 2002].

2.3 Change Propagation

We define change propagation as *changes required to other entities of the software system to ensure the consistency in a software system after a particular entity is changed* [LEHMAN and BELADY 1985, RAJLICH 1997, HASSAN and HOLT 2004, TSANTALIS *et al.* 2005]. As new features are added, others are enhanced, and bugs are fixed, developers are faced with the challenge of determining appropriate propagation of their changes through the evolving code of their product lines. The goal of change propagation in software product lines is to ensure the consistency between features.

For example, in Figure 2.1, let us suppose a change in the method signature of `getX()` to `getValueOfX()`. The method `getX()` is part of feature F6. Feature F7 depends on F6 and calls the method `getX()`. Therefore, a change in the signature of the method `getX()` will also affect the dependent feature F7. In conditional compilation, a change in the call from `this.getX()` to `this.getValueOfX()` (BOX #1 – line 12) must happen. In AspectJ, a change in the call from `myObject.getX()` to `my.ObjectgetValueOfX()` (BOX #3 – line 10) should happen to preserve consistency between related features.

In this example, it is important to highlight and understand the simultaneous change that happened in the features F6 and F7. Simultaneous change is *an event where different features have changed together*. The rationale is that changes made simultaneously in different features means that those features are related somehow. The change in the method signature implementing the feature F6 caused a change in the feature F7. All references made by the feature F7 to the method implementing the feature F6 (i.e. method `getX()`) had to be changed to preserve the consistency between those features.

2.4 Limitations of Related Work

In this section, we present the limitations of the related work to this thesis.

2.4.1 Impact of Feature Dependencies on Maintainability

The understanding of the impact of feature dependencies on product-line maintenance is essential to support change propagation. There has been considerable research effort to study the effects of feature dependencies in product-line development. The related studies try to understand and minimise negative effects of feature dependencies on product-line development.

Figueiredo and colleagues analysed the modularity and changeability of evolving product lines implemented using aspect-oriented programming [FIGUEIREDO *et al.* 2008]. They provided empirical evidence of how feature dependency changed over many releases. One of the main conclusions is how two different category of feature dependency (interlacing and overlapping) affect the stability of the source code by propagating changes in aspect-oriented systems. Similar to our work, Figueiredo and colleagues have as main concern to analyse the impact of feature dependencies on product-line maintenance. However, despite the similarities, their work associates structural properties of the programming language used with change propagation. In addition, the study of Figueiredo and colleagues focuses only on aspect-oriented product lines. Our aim is to provide a broad analysis about the impact of feature dependency on change propagation. The main concern is to identify structural properties of feature dependency agnostic of programming technique that might affect change propagation.

Cataldo and Herbsleb have empirically studied feature-oriented development in order to observe the impact that some attributes of this type of development have on integration failures [CATALDO and HERBSLEB 2011]. They concluded that dependencies and cross-feature interactions are drivers of integration failures. In another work related to failures, Garvin and Cohen examined what constitutes faults related to feature interactions [GARVIN and COHEN 2011]. They performed an exploratory study to conduct an investigation of faults on two feature-oriented open source systems and related them to feature interactions. The conclusion was that, using their testing criteria, there are faults that can be misunderstood and associated to feature interaction. In other words, the lack of information regarding the relationship between features may affect the understanding of faults. Both studies presented here concern the failures caused by the presence of feature dependencies. Differently from these studies, we focus on change propagation. Change propagation may be the root of several problems during the maintenance, including failures.

Ribeiro et al. performed an empirical evaluation on preprocessor-based software product lines focusing on the maintainability of feature dependency code [RIBEIRO *et al.* 2011, RIBEIRO *et al.* 2014]. They proposed an approach to reduce the effort of maintenance in product lines implemented using preprocessor by focusing on feature dependencies. The conclusion was that feature dependencies are reasonably common in preprocessor-based software product lines. Moreover, they highlight the impact of feature relationships on the maintenance by increasing the maintenance effort. Differently, our aim is to provide a broad analysis about the impact of feature dependency on change propagation. Our goal is to understand the relation between feature dependencies and change propagation in different implementation approaches. To do so, we need to empirically analyse such relation and identify structural properties of feature dependency agnostic of implementation approaches that might impact on change propagation.

These and other authors [FERBER *et al.* 2002, LEE and KANG 2004, APEL and BEYER 2011], have been also highlighted the impact of feature dependencies on several attributes of product lines. However, there is a lack of studies exploring in-depth the relation between these two important phenomena – i.e., change propagation and feature dependencies. Our aim is to identify feature dependency structural properties that may affect change propagation. In a first step towards such broad analysis, we conduct a study to examine the relation between feature dependency and change propagation on the evolution of one academic- and four industrial- medium-sized product lines (Chapter 3). In addition, we also identify and propose a measurement framework and metrics based on structural properties of feature dependencies. Two different implementation approaches are considered in the evaluation of our measurement framework. The goal is to analyse the effectiveness of our metrics suite in indicating change propagation in product lines implemented using different implementation approaches.

2.4.2 Metrics for Modular Programming

The rationale for proposing measurement framework based on structural properties has been identified to be related to the concept of cognitive complexity [CANT *et al.* 1994, BRIAND *et al.* 2001]. Simon argued about the limitations of human cognitive capabilities [SIMON 1978]. Developers can lose track of what they are doing and make irrational decisions in a complex context, the so-called cognitive complexity. Cognitive complexity is defined as the mental burden of developers when performing relevant operations in a specific context [BAGHERI and GASEVIC 2011]. So, the more complex is the environment to perform change propagation, the higher its degree of cognitive complexity will be. Therefore, high cognitive complexity of product lines can affect change propagation.

In the state-of-the-art, developers and researchers assume that classical modularity metrics can be used to analyse product-line attributes. However, the focus of those conventional metrics is on measuring properties of the structure of the language rather than properties of the product-line structure. Consequently, it is questionable whether conventional metrics are good indicators of change propagation. The metrics based on the structure of modules of implementation approaches do not capture the cognitive complexity of product lines. The vast majority of studies about metrics associate conventional metrics with stand-alone software attributes. In addition, there are studies proposing metrics for software product lines. However, there is a lack of studies relating product-line-based metrics with maintenance attributes. Therefore, we can divide the related work in (i) conventional metrics as indicators of software maintenance attributes, and (ii) metrics for software product line.

Implementation-approach-based Metrics and Software Maintenance

Several metrics have been proposed in the context of the implementation approaches studied in this thesis (Section 2.2). Software metrics have been widely used in the object-oriented software development in order to improve software maintainability [DANTAS *et al.* 2012]. Recently, some of these metrics were adapted to the context of aspect-oriented programming [CHIDAMBER and KEMERER 1994, HENDERSON-SELLERS 1995, ETZKORN and DELUGACH 2000, SANT'ANNA *et al.* 2003, CECCATO and TONELLA 2004, BURROWS *et al.* 2010, DANTAS *et al.* 2012] and feature-oriented programming [BARTOLOMEI *et al.* 2006, DANTAS *et al.* 2012].

To cite just a few in the context of compositional approaches, Burrows et al. proposed a novel metric to specific dependencies in aspect-oriented software systems that were not captured by conventional metrics [BURROWS *et al.* 2010]. In addition, Dantas and colleagues proposed some metrics and conducted an exploratory evaluation about the impact of specific programming techniques on change propagation [DANTAS *et al.* 2012]. They were particularly interested in analysing change propagation, which are undesirable for the software maintenance in programming techniques such as aspect-oriented programming and feature-oriented programming. The metrics proposed in these studies consider the composition code produced with different implementation approaches. However, these metrics focus on the composition of program modules rather than product-line features. These and other studies [SANT'ANNA *et al.* 2003, CECCATO and TONELLA 2004, BARTOLOMEI *et al.* 2006] propose metrics that do not consider structural properties of features and their dependencies. Thus, the use of such metrics might not be appropriate to product lines. In addition, there is a lack of studies proposing specific metrics for product lines implemented in compositional approaches. Our work aims at propose a suite of metrics applicable for product lines implemented using compositional approaches.

In the context of annotative approaches, Liebig et al. proposed a set of metrics to analyse the variability of forty preprocessor-based software product lines [LIEBIG *et al.* 2010]. These metrics intend to measure product-line properties in terms of comprehension and refactoring. Differently from their work, we focus on change propagation instead of comprehension and refactoring. In addition, they do not propose metrics related to feature dependencies in order to draw their conclusions. Apel and Beyer adapted conventional metrics that were originally proposed for procedural and object-oriented systems to a set of cohesion metrics based on clustering layouts.

Their goal is to provide a better understanding of cohesion in software product lines [APEL and BEYER 2011]. Similar to our work, they propose metrics agnostic to implementation approaches. However, the proposed metrics are based on feature cohesion. Our work proposes a measurement framework for structural properties of feature dependency. In addition, our metrics are based on characteristics of the source code instead of being based on visual observation of clusters. In addition, our main concern is with feature dependencies, which may be a driver of change propagation.

Metrics for Software Product Lines

There is a body of work aiming at assessing and predicting the quality of product lines, especially of product-line architectures. For instance, van der Hoek et al. propose a number of measures to assess the variability of a product-line architecture [VAN DER HOEK *et al.* 2003]. The suite of metrics is based on the concepts of service dependency and tries to predict volatility of the product line based on those service dependencies. However, inferring the volatility based only on service dependency might result in a distorted view of change prediction. According to Geipel and Schweitzer, considering only the number of dependencies might not be a good indicator of change propagation [GEIPEL and SCHWEITZER 2012]. In our work, in contrast, we followed the conclusions of Geipel and Schweitzer and we took advantage of change characteristics (e.g. concentration of changes) and dependency structure.

Another interesting point to observe is that the work of van der Hoek et al. [VAN DER HOEK *et al.* 2003] is based on metrics for software product line architectures. Actually, most of the work proposing metrics for evaluating attributes of maintenance in product lines focuses on the architecture and feature model artefacts. Rahman proposes a metric suite to measure product-line architectural quality attributes such as observability, configurability and modularity [RAHMAN 2004]. Cheng propose several measures based on architectural drivers [CHENG *et al.* 2006]. Her et al. developed a framework for evaluating the reusability of a product line's core assets [HER *et al.* 2007]. Bagheri et al. propose a suite of metrics for product-line feature models and validate them using valid measurement theoretic principles [BAGHERI and GASEVIC 2011]. Torkamani also proposes a suite of metrics for product-line architecture [TORKAMANI 2014]. His goal is to use the metrics suite to evaluate the reusability power of evolving software product lines. In spite of the large body of work, the focus is on architectural quality attributes instead of source-code attributes. In addition, they do not focus on change propagation. At the end, change propagation can only be reliably estimated based on source code analysis. Many of the links realising a feature dependency are only presented in the source code. The implementation approaches also influence the complexity and number of such links. Finally, architecture and feature models are often too abstract and incomplete in practice.

2.4.3 Modularity

[KÄSTNER *et al.* 2011] pinpoint two different notions of feature modularity: one based on locality and cohesion, and another based on information hiding and interfaces. In this work, we focus on the less explored notion of modularity: information hiding and interfaces. As aforementioned, in this thesis we focus on annotative approaches, which have a lack of modularity regarding information hiding and interfaces [KÄSTNER *et al.* 2011]. In the following, we present the most relevant studies regarding information hiding and interfaces in product-line implementation approaches.

Information hiding

Some approaches were developed to provide separation of concerns by hiding information in the source code. Conceptual Module [BANIASSAD and MURPHY 1998] is one approach that supports developers on maintenance tasks by setting lines of code to be part of a conceptual module. Moreover, it captures other lines that should be part of such module by means of queries. In addition, it computes dependencies between conceptual modules. This approach abstracts details from developers, so that they concentrate on the relationship among conceptual modules. Similar to our proposal, Conceptual Module is also concerned with maintenance tasks and dependencies. However, we aim to improve the comprehensibility of the source code by organising information about feature dependencies based on structural properties instead of abstract details from developers. Our goal with the organisation of feature dependencies information is to alleviate the cognitive complexity of software product line. Consequently, we may support developers to better understand the complex dependencies between features and help with the task of change propagation.

In another important work, Kästner and colleagues proposed the Colored IDE (CIDE). CIDE is an Eclipse-based tool for decomposing legacy applications into features [KÄSTNER *et al.* 2008]. It relies on VSoC approach [KÄSTNER and APEL 2009], which means that it is possible to hide code of features not interesting to a current maintenance task. Another approach that provides separation of concerns by hiding information is Mylyn [KERSTEN and MURPHY 2006]. Mylyn aims to reduce information overload by showing only relevant artefacts to a maintenance task. In this way, Mylyn can improve productivity in such a way that developers do not spend time searching for the artefacts needed to complete a task, and reducing the information overload. Our main concern is not related to information hiding like CIDE and Mylyn. Although, we also aim to provide information overload reduction by organisation feature dependencies information. The organisation of that information can guide developers to analyse only relevant parts of the source code during change propagation.

Interfaces

To the best of our knowledge, the only compositional approach that presents work on interfaces is aspect-oriented programming. Kiczales and Mezini proposed the so-called aspect-aware interfaces [KICZALES and MEZINI 2005]. This approach computes aspect's dependencies on a system's join points (see Section 2.2.3). As a result, it shows those dependences as annotations on explicit interfaces of advised code. By revealing such dependencies, a programmer can see how join points are being advised and avoid making changes that invalidate those uses. Open Modules propose the use of interfaces for exposing join points in classes, limiting the scope of advised code to the join points exposed by the developer [ALDRICH 2005]. Griswold et al. proposed Crosscutting Programming Interfaces (XPIs) [GRISWOLD *et al.* 2006]. XPIs consist of abstract interfaces aiming at decoupling the aspects from details of classes to provide better modularity during parallel evolution. Differently from these studies, our thesis will focus on organise feature dependencies information in feature interfaces. Specifically, we focus on product lines implemented with conditional compilation. Finally, our focus is to support change propagation by organising the feature dependencies information by means of feature interfaces. A *feature interface* comprises the program elements in the source code that are responsible for providing external access to other features. We provide a more systematic definition of feature interface in Chapter 5.

Regarding annotative approaches, there is a lack of studies proposing interfaces due to several challenges [KÄSTNER *et al.* 2011]. Recently, Ribeiro et al. proposed the pioneer work dealing with interface in product lines implemented with annotative approaches [RIBEIRO *et al.* 2011, RIBEIRO *et al.* 2014]. They propose the concept of emergent interfaces for product lines implemented with conditional compilation. This approach aims to establish interfaces between features on demand (emergent interfaces), to prevent developers from breaking other features when performing a maintenance task. Similar to our idea, Ribeiro et al. propose interfaces for product lines implemented with conditional compilation. Such interfaces are created on demand based on a maintenance task. Like our work, they aim to generate feature interfaces in order to better support maintenance tasks. However, in their approach, the maintainer need to know what parts of the source code must be changed in order to generate the emergent interfaces. In our approach, we aim to indicate, by means of an organised feature interface, which parts of the code might be affected due to a maintenance task.

There is further work that concentrates on interfaces for product lines or in variability-checking approaches supported by interfaces. For example, Kästner and colleagues propose a variability-aware module system for product lines [KÄSTNER *et al.* 2012]. This approach infers interfaces for modules focusing on type checking of product-line configurations. Li and colleagues propose a new methodology to verify cross-cutting features as open systems by using a model of semantic interfaces that supports automated, compositional, and feature-oriented model checking [LI *et al.* 2002, LI *et al.* 2002b]. Blundell et al. propose a parametrised interface for verifying product-lines [BLUNDELL *et al.* 2004]. Such interface lifts properties of individual features to composed features to verify temporal properties of such features. However, none of these studies deal with interfaces specifically for supporting maintenance tasks. Our approach aims at supporting developers to propagate changes during maintenance by providing organised feature dependencies information.

2.5 Summary

This chapter presented a review of the essential concepts explored in this thesis. Section 2.1 presented the definition of software product line as well as two important concepts for this thesis; the definition of feature (Section 2.1.1), and the definition of feature dependency (Section 2.1.2). Section 2.2 presented the implementation approaches addressed in this thesis as well as a brief comparison between them. We also presented in Section 2.3 the concept of change propagation. In Section 2.4, we presented the limitations of related work. Section 2.4.1 showed the studies focused on explore the impact of feature dependencies on maintainability. Section 2.4.2 presented some studies of software metrics important for the understanding of this thesis. Finally, Section 2.4.3 reported related work aiming at improving the modularity of features.

3 Feature Dependency as Change Propagators

Product-line features usually relate to each other in the source code by means of feature dependencies [YE and LIU 2005, RIBEIRO *et al.* 2011]. During the code maintenance of product lines, features must eventually be updated when changes occur in dependee features. Any change in the code associated with a feature might imply changes on one or more dependent features. Thus, a main concern in product-line maintenance is to preserve consistency between related features by propagating changes. Over the last fifteen years, the relevance of studying the impact of feature dependency on product-line maintenance complexity has been recognised [FERBER *et al.* 2002, LEE and KANG 2004, RIBEIRO *et al.* 2011, RIBEIRO *et al.* 2014]. However, there is a lack of studies investigating the relation between these two important phenomena – i.e., feature dependencies and change propagation in more depth.

Comprehending dependencies and revisiting code of potentially affected features make change propagation a time-consuming task [RIBEIRO *et al.* 2011]. We argue that feature dependencies may be an important driver to change propagation in software product lines, thereby impacting on the complexity of product-line maintenance. Understanding whether and how feature dependencies lead to change propagation is important to reduce the complexity of product-line maintenance. It is neither obvious nor well understood to what extent and how feature dependencies affect change propagation. This lack of knowledge may become a barrier for the longevity of software product lines [CATALDO and HERBSLEB 2011]. Software engineers need to understand the intricacies of maintaining inter-dependent features of a software product line.

For instance, developers should be aware if feature dependencies act frequently as change propagators. Certain feature dependencies may be involved more often in change propagation than other feature dependencies along the evolution. In this case, an identification of feature dependencies that cause more change propagation would support developers during the maintenance task. Developers can focus and analyse more carefully those feature dependencies in order to reason about changes that may propagate to other features. In addition, another important point to be understood by developers is whether and how changes propagate transitively through a path of dependencies. A transitive feature dependency happens whenever a feature C depends on a feature B, and B is in turn dependent on a feature A; then C depends on A by transitivity (a.k.a. indirect dependency). In this case, it may be a propagation of changes through the path of dependencies from C to A with two units of distance. In other words, a change in feature A may affect B and C – i.e. a change propagation extent of two features. Thus, it is also important to understand the extent of the change propagation through the path of feature dependencies, since programmers often inspect the code of immediately-neighbour features at best.

This chapter presents an in-depth study to examine the relation between feature dependency and change propagation. This study aims at answering the first research question of this thesis (RQ1 in Section 1.2), which states: *What is the relation between feature dependencies and change propagation?* In particular, we want to understand whether feature dependencies are related to change propagation. This is essential to show the importance of explore feature dependencies as change propagators. In addition, it is interesting to explore whether certain feature dependencies are involved more often in change propagation than other feature dependencies, and the extent of change propagation through paths of feature dependencies – i.e., transitive dependencies (see Chapter 2). These two last analyses are essential to reveal if there are certain characteristics of feature dependencies that may better indicate change propagation. It is important to mention that, in this chapter, feature dependency refers to either a direct dependency or transitive dependency.

To achieve our goals, we conducted an exploratory study on the evolution of one academic- and four industrial- medium-sized product lines implemented with conditional compilation. The analysed product lines comprise a total number of twenty-six releases[1] (i.e., twenty-one distinct evolution scenarios among them). Specifically, we analyse both feature dependencies and change propagation. Similar to several work [HASSAN and HOLT 2004, CATALDO *et al.* 2008, BIRD *et al.* 2011, JERMAKOVICS *et al.* 2011, GEIPEL and SCHWEITZER 2012, JOBLIN *et al.* 2015], in this study we use simultaneous changes (see Section 2.3) as indicators of change propagation. Simultaneous

[1] Public distribution of a new upgraded version with improvements in functionality and bug fixing.

changes are events where different features are changed in the same evolution. The rationale is that changes made simultaneously in different features during a single evolution means that those features are related somehow.

Thus, we aim at identifying and exploring change propagation through feature dependencies during additive changes (i.e., addition of a new feature) of evolving product lines. An additive change is classified as a perfective maintenance [GODFREY and GERMAN 2008]. Perfective maintenance is the most common type of maintenance comprising more than 60% of the maintenance tasks [SCHACH *et al.* 2003]. In summary, we make the following contributions:

- There is a strong relation between feature dependency and change propagation. In our study, this relation usually happens when there is a change in fragments of the feature code that are responsible for realising feature dependencies. So, a change in these fragments is more likely to demand a change in dependent features.
- Our findings revealed an inequality in the distribution of change propagation along feature dependencies. This counter-intuitive result basically indicates that a general minimisation of feature dependencies might not decrease the change propagation through paths of dependencies.
- Our analysis also evidenced a linear decay in the probability of change propagation in the path of feature dependencies. This means that the extent of change propagation in product-line features might be more severe than the extent in files of non-product-line systems (see [GEIPEL and SCHWEITZER 2012]). Moreover, our data revealed that it is common to find transitive feature dependencies (77% of the product line releases). Thus, several sampling-based analysis techniques for product line that disregard larger feature combinations might be tuned in order to consider more features in these combinations.

The analysis of the study was carried out taking into consideration the method presented in Section 3.1. The explanations for our findings are presented in the results and analysis sections (Section 3.2 and 3.3). Threats to validity are presented in Section 3.4. The results of our study are compared to results of previous studies in Section 3.5. Our final considerations are presented in Section 3.6. It should not be left unmentioned that the results of this chapter are reported in a submitted paper to an international journal.

3.1 Study Setting

This section presents the research aims and motivates the relevance of this study (Section 3.1.1), followed by an explanation of the study design to answer the research questions (Section 3.1.2). Section 3.1.3 presents the target systems. The section concludes with a description of the evaluation procedures (Section 3.1.4).

3.1.1 Research Aims

Maintenance of software product lines requires the effective propagation of changes. In this case, to preserve consistency, changes are propagated to dependent features during product-line evolution. Thus, it is important to find whether there is a relation between feature dependency and change propagation. In other words, we want to explore whether feature dependency is a driver for change propagation. If so, it is also important to explore this relation in terms of the extent of change propagation and the concentration of change propagation in some dependencies.

In this context, one can hypothesise: if there are simultaneous changes in dependent features during the evolution of one release to the next one, they may be caused by a change propagation through the existing dependencies between features. So, it should exist a relation between feature dependency and change propagation. Thus, the first research question of this study addresses whether this is the case: *[RQ1.1] Are changes more likely to propagate due to feature dependencies?*

The second research question was inspired by the assumption that all feature dependencies are assumed to equally propagate changes [FIGUEIREDO *et al.* 2008, RIBEIRO *et al.* 2011]. In other words, these studies assume that each dependency between features incurs the same impact on change propagation, whatever be the feature dependencies' characteristics. The equal impact of feature dependencies on change propagation has important ramifications. If the majority of feature dependencies equally impact the effort of propagating changes, they should be reduced. On the other hand, if only specific dependencies matter, ad hoc reduction of dependencies in product line might not reduce the effort of propagating changes. In this case, the identification of feature dependencies that cause more change propagation would ameliorate this problem. Thus, the second research question of this study is concerned about the equality of the impact of feature dependencies on change propagation: *[RQ1.2] Are feature dependencies equally involved on change propagation?*

Finally, if there is a relation between feature dependency and change propagation as hypothesised in research question RQ1.1, we can assume that changes might be propagated along the path of feature dependencies. In other words, a change in one feature may cause a change cascade along the path of feature dependencies even to distant features. In this context, the more features are affected by changes, the more costly it may be to maintain a software product line. So, it is important to understand the extent of the change propagation through the paths of feature dependencies, since programmers often inspect the code of direct neighbour features at best. Thus, the third research question is concerned about the relation between change propagation and distance between features in the path of dependencies: *[RQ1.3] Is there a relation between probability of simultaneous change and the distance between features in the path of dependencies?*

3.1.2 Study Design

This section explains in detail how we modeled and extracted the data to answer the research questions of the study.

Feature Dependency

Two mathematically equivalent notations are commonly used to represent dependencies between elements: the graph notation and the adjacency matrix notation [KAFURA and REDDY 1987, SULLIVAN *et al.* 2001, SANGAL *et al.* 2011, MACCORMACK *et al.* 2006, GEIPEL and SCHWEITZER 2012]. In this study, we chose the adjacency matrix notation because it is the most widely used notation to represent software dependencies [GEIPEL and SCHWEITZER 2012]. Moreover, it is very simple and easy to understand, enabling us to make several relevant computations.

We refer to the feature dependency matrix as FD, where a feature appears both in one row and in one column of the matrix. $FD_{a,b} \geq 1$ means that the feature a (row) depends on the feature b (column). Therefore, $FD_{a,b} = 0$ is interpreted as feature independence. There is a dependency of feature a on feature b if: (i) a references an attribute of b, or (ii) a calls a method of b. We set $FD_{a,b} = 1$ if at least one of these cases happen. To add more representative power to this approach, we also represent the distance d of transitive dependencies between features in the matrix entries, the so-called path of feature dependencies.

For instance, let us suppose that a feature C depends on feature B, and feature B depends on feature A. In this case, $FD_{B,A} = 1$, $FD_{C,B} = 1$ and $FD_{C,A} = 2$. Therefore, the value of FD represents the distance d between two features. The values of distance between features in paths of feature dependencies are calculated by using the Floyd-Warshal algorithm [FLOYD 1962]. The Floyd-Warshall algorithm aims at finding shortest paths between all pairs of vertices in a weighted graph (with no negative cycles) [CORMEN *et al.* 2009]. Moreover, it is worth to notice that FD is an asymmetric matrix since we consider dependency as a directional relationship. The example of the adjacent matrix of this example is presented in Figure 3.1.

	A	**B**	**C**
A	0	1	0
B	0	0	1
C	0	0	2

Figure 3.1: Example of a adjacent matrix of feature dependencies.

To perform the extraction of the adjacent matrix of feature dependencies automatically, we used the tool CIDE [KÄSTNER 2015] that supports mapping of features in the source code. CIDE was also used to relate program elements to features. The CIDE output was used by an extension of the tool GenArch+ [CIRILO *et al.* 2008] in order to generate the matrix of feature dependencies. GenArch+ uses an AST (Abstract Syntactic Tree), based on the source code, decorated with information about features in its nodes in order to extract feature dependencies. Each program element is represented by a node in the tree, and it is associated with one or more features. Thus, when one program element from one feature is used by a program element from another feature directly, we consider this as a feature dependency. The transitive dependencies of the feature dependency matrix were calculated by a program implemented in Java for the purpose of this study [CAFEO *et al.* 2015]. The algorithm receives as input a list of feature dependencies generated by the tool GenArch+, and generates an adjacent matrix with all feature dependencies (including transitive dependencies) of a product line.

Simultaneous Changes

Similarly to feature dependencies, we use a matrix structure notation to represent the simultaneous change of features along the evolution of the software product lines. Moreover, we model our matrix for simultaneous changes by adapting the simultaneous change model presented in the work of Geipel and Schweitzer [GEIPEL and SCHWEITZER 2012]. This matrix represents the view of the product-line evolution based on simultaneous changes. Its entries indicate the number of times the features have been changed simultaneously. Let us refer to this matrix as C, and the event of features being *"changed at the same time"* during an evolution as a simultaneous change.

To construct C, we need (i) the set of features, and (ii) the changes happened in the features along the evolution of a release to the next one. Let us use n to denote the number of the feature and m to refer to the number of evolution. An event in the product-line evolution can be expressed as a n-dimensional vector $\vec{e}$. Each entry shows in binary form whether a feature has been modified or not. For instance, in a product line with three features, we have changes in two of them along the first evolution. Therefore, the evolution scenario $\vec{e}_1 = (101)^T$ indicates that features one and three were modified simultaneously in evolution 1. Thus, each $\vec{e}$ corresponds to one product-line evolution.

The evolution matrix of a product line can be written by considering each $\vec{e}$ as a column of the evolution matrix E of size $n \times m$:

$$E = (\vec{e}_1 \vec{e}_2 \ldots \vec{e}_m) \tag{1}$$

To have a matrix representing the whole evolution scenario of a software product line, we need to multiply E with its transposed matrix E^T, and thus the simultaneous changes matrix C is derived:

$$C = EE^T \tag{2}$$

The matrix C has dimension $n \times n$ and indicates how many times each feature has been modified simultaneously with other feature. An entry $C_{B,A} = 4$, for instance, indicates that features A and B have been changed four times together considering the entire product-line evolution. Note that C, in contrast to FD, is a symmetric matrix. Figure 3.2 illustrates all simultaneous changes of the product line comprising features A, B and C. Along the evolutions features B and A changed simultaneously four times, while feature

C and B twice, and features C and A only once.

	A	B	C
A	0	4	1
B	4	0	2
C	1	2	0

Figure 3.2: Example of a adjacent matrix of simultaneous changes.

To perform this extraction automatically, we used an extension of the tool GenArch+ [CIRILO *et al.* 2008] to analyse the evolution and extract the changes in the features. The simultaneous changes matrix was calculated by a program implemented in Java for the purpose of this study [CAFEO *et al.* 2015]. The program receives as input a list of changes in features per evolution, and generates a matrix with the number of simultaneous changes for each pair of features along all evolution of a product line.

3.1.3 Target Systems

As target systems, we selected five product lines available in open repositories and managed by different developers. All of them have been developed using an annotative approach: Java with conditional compilation. We chose conditional compilation because, despite the controversial discussion in the literature about its benefits, it is a well-known and industry-strength technique [KÄSTNER and APEL 2009]. In addition, by large, it is the most commonly used in practice. All the product lines in this study are based on Java and conditional compilation. It is also worth noticing that, besides the Graph Product Line (see below), all of them are non-academic product lines and developed for different purposes. In the following we describe each target system:

– **Berkeley DB.** It is an open source database engine that can be embedded as a library into applications [BERKELEY DB 2015, THÜM and BENDUHN 2015]. The core features represent basic data management and transactional behaviour support whilst the variabilities include logging and statistics, among others. The evolution comprise the addition of optional features such as logging and file handle cache.

– **Mobile RSS.** It is a portable RSS reader for mobile phones on the Java ME platform [MOBILE RSS 2015, THÜM and BENDUHN 2015].

Mobile RSS basically provides features to parse, browse and read RSS feeds. Moreover, for instance, there are features to deal with compatibility issues of mobile phones and features for logging. The evolution comprise the addition of optional features.

- **Lampiro.** It is a messaging client for mobile devices based on a protocol called XMPP [LAMPIRO 2015, THÜM and BENDUHN 2015, XMPP 2015]. It is possible to connect with contacts of ICQ, Gtalk and Windows Messenger. There are many features including compression, encryption and languages supported. The evolution comprise the addition of mandatory and optional features.
- **Graph Product Line (GPL).** It is a product line of graph libraries that allows programmers to tailor graph data structures, including weighted and directed edges as well different traversal strategies and algorithms [LOPEZ-HERREJON and BATORY 2001]. The evolution scenarios comprise the addition of alternative and optional features.
- **Java Chat.** It is a simple chat application with features such as GUI, encryption and logging [THÜM and BENDUHN 2015]. The evolution scenarios comprise the inclusion of optional features.

Table 3.1 shows general data about the target product lines, such as lines of code (KLOC), number of preprocessor directives implementing features found in the source code (# of IFDEFS), number of features (# of Features), number of feature dependencies (# of Feature Dependencies), and number of product-line releases (# of Releases).

Table 3.1: General information about the target software product lines

Projects	**KLOC**	**# of IFDEFS**	**# of Features**	**# of Feature Dependencies**	**# of Releases**
Berkeley DB	39	4051	56	860	6
Mobile RSS	18	2990	31	235	6
Lampiro	31	164	18	112	6
GPL	1	582	26	119	4
Java Chat	0.6	105	9	60	4
mean	**28**	**18**	**1579**	**277**	**5**

In order to address our research question (Section 3.1.1), we focused on the evolution with additive changes (e.g. addition of a new feature). An additive change is classified as a perfective maintenance [GODFREY and GERMAN 2008]. A perfective maintenance comprises enhancements intended to make the system better – i.e. mainly the addition of new features in the context of software product lines [ALVES *et al.* 2005, FIGUEIREDO *et al.* 2008, SVAHNBERG and BOSCH 2000]. Evolution with perfective maintenance alter the intended outward semantics of the system, thus justifying the creation of a new product-line release. Moreover, more than 60% of the maintenance tasks are associated with a perfective maintenance [SCHACH *et al.* 2003]. In other words, it is the most common maintenance type in software evolution.

Based on the respective original systems from open repositories [KÄSTNER 2015, BERKELEY DB 2015, THÜM and BENDUHN 2015, MOBILE RSS 2015, LAMPIRO 2015], we created representations of evolutions scenarios comprising only addition of new features. The representation of the evolution scenarios were created based on information of the open repositories [KÄSTNER 2015, BERKELEY DB 2015, THÜM and BENDUHN 2015, MOBILE RSS 2015, LAMPIRO 2015], source code analysis and information about evolution scenarios extracted from studies about product-line evolution [FIGUEIREDO *et al.* 2008, ALVES *et al.* 2005]. In this way, each evolution of our target product lines comprises one or more type of change, such as inclusion of optional features, inclusion of alternative features, inclusion of mandatory features, and implementation of new product-line constraints due to the inclusion of new features. For instance, the evolution of BerkeleyDB involves the inclusion of optional features such as logging and file handle cache. The average number of included features is 9 in each evolution. One evolution scenario for this software product line was, for instance, the addition of features realising the registration of the actions of the database in a log file. These changes happened in the 4th and 5th releases. It is important to notice that other features not related to logging were also added in 4th and 5th releases.

These evolutions scenarios were retrospectively implemented by undergraduate and postgraduate students based on the latest release of each product line. In other words, based on the representation of the evolutions created by some authors of the papers, the students could isolate and simulate in the source code evolutions considering only the addition of new features. Good design principles and practices were used, enforced, and reviewed throughout the creation of all the software product line releases. These practices were applied to ensure that the purpose of each feature was achieved as expected. In all the cases, the evolution scenarios were also reviewed by experts in the field. The source code of all target product lines are available in the website of the study [CAFEO *et al.* 2015].

3.1.4 Procedures

This section describes the evaluation procedures to answer the research questions presented in Section 3.1.1.

The Relation Between Feature Dependency and Change Propagation (RQ1.1)

In order to understand the relation between feature dependency and change propagation, we make a twofold comparison based on two point of views. We are using probabilistic analysis because we want to analyse chances of the occurrence of change propagation in the presence of feature dependencies. Specifically, we are using conditional probability to analyse the data and answer RQ1.1. A conditional probability measures the probability of occurrence of an event given that (by assumption, presumption, assertion or evidence) another event has occurred [BERTSEKAS and TSITSIKLIS 2008].

Simultaneous changes view. We express the fact that two features (a and b) have been changed at least once simultaneously by $C_{a,b} \geq 1$. Moreover, the fact that there is a dependency between two features (a and b) is expressed by $FD_{a,b} > 0$. So, in our first comparison, to measure the impact of feature dependencies on simultaneous change we use the conditional probability $P_{FD} = P(C_{a,b} \geq 1 | FD_{a,b} > 0)$ given $a \neq b$. In other words, the probability that two different features have been changed at least once simultaneously, given that there is a dependency between them. P_{FD} is calculated as follows:

$$P_{FD} = \frac{|\{FD_{a,b} > 0 \land C_{a,b} \geq 1\}|}{|\{FD_{a,b} > 0\}|} \tag{3}$$

This means that we divide the number of simultaneous changes in features involved in dependencies by the total number of feature dependencies.

As a reference, we compute the conditional probability $P_{\neg FD} = P(C_{a,b} \geq 1 | FD_{a,b} = 0)$ given $a \neq b$. This means that we are interested in calculating the probability that two different features have been modified together at least once given that they are independent. $P_{\neg FD}$ is calculated as follows:

$$P_{\neg FD} = \frac{|\{FD_{a,b} = 0 \land C_{a,b} \geq 1\}|}{|\{FD_{a,b} = 0\}|} \tag{4}$$

This means that we divide the number of simultaneous changes in features that are not involved in dependencies by the total number of features with no dependencies.

The comparison of P_{FD} with $P_{\neg FD}$ reveals the possible degree of correlation between feature dependency and simultaneous change in terms of dependency.

Feature dependency view. In our second comparison, we are interested in analysing the other way: the probability of existing a feature dependency given that there is a simultaneous change or not. As aforementioned, we express the fact that there is a dependency between two features (a and b) by $FD_{a,b} > 0$ and that two features (a and b) have been changed at least once simultaneously by $C_{a,b} \geq 1$. A measure for the relation between simultaneous change and feature dependency is the conditional probability $P_C = P(FD_{a,b} > 0 | C_{a,b} \geq 1)$ given $a \neq b$. It means the probability that a feature dependency exists given that two different features have been changed at least once simultaneously. P_C is calculated as follows:

$$P_C = \frac{|\{C_{a,b} \geq 1 \wedge FD_{a,b} > 0\}|}{|\{C_{a,b} \geq 1\}|} \tag{5}$$

This means that we divide the number of simultaneous changes in features involved in dependencies by the total number of simultaneous change.

As a reference, we compute the conditional probability $P_{\neg C} = P(FD_{a,b} > 0 | C_{a,b} = 0)$ given $a \neq b$. This computation means that we are interested in calculating the probability that a feature dependency exists in the absence of a simultaneous change. $P_{\neg C}$ is calculated as follows:

$$P_{\neg C} = \frac{|\{C_{a,b} = 0 \wedge FD_{a,b} > 0\}|}{|\{C_{a,b} = 0\}|} \tag{6}$$

This means that we divide the absence of simultaneous changes in features involved in dependencies by the total number of simultaneous changes. The comparison of P_C with $P_{\neg C}$ reveals the possible degree of correlation between feature dependency and simultaneous change from the point of view of dependencies.

Equality on the Impact of Change Propagation (RQ1.2)

Given the number of feature dependencies per release of product line (e.g., 235 in the last release of MobileRSS), there are too many data points in order to undertake a comparison of all feature dependencies individually. Thus, to address our research question, we apply data aggregation [VASILESCU *et al.* 2011] using concentration statistics. The idea is to analyse the equality of the impact of feature dependencies on change propagation. This analysis allows us to make statements such as "10% of the feature dependencies are responsible for over 70% of change propagation".

As a statistic concentration, we adopt a method to analyse and visualise income inequalities in a population of a country called Lorenz inequality or Lorenz curve [LORENZ 1905, VASILESCU *et al.* 2011]. In this study, we use this technique to analyse the concentration of change propagation in feature dependencies. For a more in-depth description of Lorenz inequality, the reader may refer to the original work of Lorenz [LORENZ 1905]. First, we need the number of simultaneous changes for each feature dependency. Let us refer to this set as Π:

$$\Pi = \{C_{a,b} : FD_{a,b} \geq 1 \wedge C_{a,b} \geq 1\} \tag{7}$$

The next step is to normalise Π, so that the entries of the result π sum up to one:

$$\pi_n = \Pi_n / \sum_{l=1}^{|\Pi|} \Pi_l \tag{8}$$

After that, the entries of π must be rearranged in ascending order:

$$m < n \Rightarrow \pi_m < \pi_n \tag{9}$$

Finally, the Lorenz curve $L(x)$ with $0 \leq x \leq 1$ is calculated by cumulating the first x percent of the elements of π:

$$L(x) = \sum_{n=0}^{\lfloor x * |\pi| \rfloor} \pi_n \tag{10}$$

$L(x)$ is the percentage of simultaneous changes accumulated in the x percent less active feature dependencies. An equal distribution leads to $L(x) = x$. To compare the Lorenz concentration between the target product lines we compress it to one number named Gini coefficient [GINI 1921]. The Gini coefficient indicates the degree of distributional inequality of simultaneous changes amongst the feature dependencies of a software product line. Let us refer to the Gini coefficient as g:

$$g = 1 - 2 \int_0^1 L(x)\,\mathrm{d}x \tag{11}$$

The Gini coefficient takes a value between 0 and 1, with $g = 0$ denoting perfect equality meaning that x percent of the feature dependencies are responsible for x percent of the simultaneous changes. Conversely, $g = 1$ denotes perfect inequality with only one feature dependency responsible for 100 percent of the cumulative simultaneous changes. By using Gini coefficient we are able to compare different concentrations since such coefficient represents a concentration in a scalar value.

Feature Dependency Distance (RQ1.3)

Features might propagate changes along the path of feature dependencies. In this context, one can assume that there is a decreasing probability of simultaneous changes of features as the distance between features increases in the path of feature dependencies. In this context, we extend the concept of P_{FD} to $P_{FD}(d)$, giving the probability that two features connected by a dependency with distance d are changed together at least once. So, if changes propagate along a path of dependencies, which function P_{FD} in d would indicate the probability of change propagation?

It is known that, when only one dependent variable is being modelled, a scatterplot can suggest the form and strength of the relationship between variables. In our study, the scatterplots of all target systems suggest that there is a relationship between d and P_{FD}, and this relationship can be approximated as a linear function. Consequently, the most simple approximation for $P_{FD}(d)$ is:

$$P_{FD}(d) = -ad + b \tag{12}$$

We add a constant a with a minus sign to the equation because (i) we are supposing that there is a decreasing probability of change propagation related to distance, and (ii) as a rule, the constant term is always included in the set of regressors, which in our case is d. Moreover, since $P_{\neg FD}$ may be nonzero, we also add an intercept b to the equation.

Having the possible equation that delineates $P_{FD}(d)$, we need to test the goodness of fit of the statistical model proposed regarding our data. Measures of goodness of fit typically summarise the discrepancy between observed values and the values expected under the model in question. Since our statistical model proposed is a linear function, we fit the observed data $P_{FD}(d)$ to equation 12 with the ordinary least squares method. Ordinary least squares is a method for estimating the unknown parameters in a linear regression model. This method minimises the sum of squared vertical distances between the observed responses in the dataset and the responses predicted by the linear approximation. To check whether the model explains the patterns found in the data, the quality of the fit needs to be quantified. To measure the quality of fit, we consider the adjusted coefficient of determination ($adj.R^2$).

The coefficient of determination of a linear regression model is the quotient of the variances of the fitted values and observed values of the dependent variable. If we denote y_i as the observed values of the dependent variable, $\bar{y}$ as its mean, and $\hat{y}_i$ as the fitted value, then the coefficient of determination is:

$$R^2 = \frac{\sum (\hat{y}_i - \bar{y})^2}{\sum (y_i - \bar{y})^2} \tag{13}$$

The adjusted coefficient of determination ($adj.R^2$) of a linear regression model is defined in terms of the coefficient of determination (equation 13). R^2 is adjusted to account for the residual degrees of freedom (number of observations minus the number of fitted coefficients). If we denote n as the number of observations in the dataset, and p as the number of independent variables, the adjusted coefficient of determination is:

$$adj.R^2 = 1 - (1 - R^2)\frac{n-1}{n-p-1} \tag{14}$$

The adjusted coefficient of determination ($adj.R^2$) takes a value between -1 and 1, with values of $adj.R^2$ close to 1 denoting that the model fits the data well. In other words, high values of $adj.R^2$ provide evidence for the existence of change propagation, and thus a relation between probability of simultaneous change and distance along the paths of dependencies.

We are using the adjusted coefficient of determination to quantify the quality of the fit because (i) it is bound between -1 and 1 making the comparison between the target systems easier than unbound measures, and (ii) it overcomes specific problems of the coefficient of determination (R^2) in order to provide additional information by which we can evaluate our regression model's explanatory power.

3.2 Results

This section presents the results of the analysis described in Section 3.1.4. Section 3.2.1 describes the results of the probabilities extracted to answer research question RQ1.1. To address the research question RQ1.2, Section 3.2.2 shows the results obtained regarding the concentration of change propagation in certain feature dependencies. Finally, Section 3.2.3 checks whether the statistical model proposed represents the patterns found in our data in order to answer to research question RQ1.3.

3.2.1 Feature Dependency and Change Propagation

As pointed out in Section 3.1.4, the probabilities P_{FD}, $P_{\neg FD}$, P_C and $P_{\neg C}$ can provide evidence of the relation between feature dependency and change propagation. Table 3.2 shows the probability values for P_{FD} and $P_{\neg FD}$ for the five software product lines. These values indicate the probability that two features change simultaneously in the presence of feature dependency (P_{FD}) or in the absence of feature dependency ($P_{\neg FD}$). The comparison between these values indicates the influence of a feature dependency in the occurrence of a possible change propagation. It is important to notice that the amount of feature dependencies or simultaneous changes affect the probability results, but not the difference between such probabilities. A change in the amount of these variables should affect both the conditional probability and its reference probability. Since we are comparing a conditional probability with its reference, the difference between them is supposed to be the same (or similar) when the amount of one or both variables change.

Table 3.2: Summary of Probabilities.

Projects	$\mathbf{P_{FD}}$	$\mathbf{P_{\neg FD}}$	$\mathbf{P_C}$	$\mathbf{P_{\neg C}}$
BerkeleyDB	0.163	0.028	0.722	0.275
GPL	0.471	0.165	0.378	0.119
JavaChat	0.700	0.099	0.500	0.045
Lampiro	0.750	0.141	0.062	0.003
MobileRSS	0.366	0.117	0.524	0.202
mean	**0.504**	**0.114**	**0.425**	**0.123**

Looking at the values of P_{FD} and $P_{\neg FD}$, we can notice that P_{FD} ranges between 0.16 and 0.75, and $P_{\neg FD}$ is less than 0.16 for all software product lines. This basically means that it is likely that change propagation happens when a feature dependency exists. This superiority of P_{FD} against $P_{\neg FD}$ can be seen in the mean of these values ($P_{FD} = 0.504$ vs. $P_{\neg FD} = 0.114$). Moreover, this behaviour can be observed in all product lines analysed as shown in Figure 3.3.

In Figure 3.3, we can notice that the probability of simultaneous changes in dependent features (P_{FD}) is much higher than the probability in independent features ($P_{\neg FD}$). The smallest difference between P_{FD} and $P_{\neg FD}$ is observed in GPL. In this case, the chances of simultaneous changes in dependent features is almost three times higher than simultaneous changes in independent features ($P_{FD} = 0.47$ vs. $P_{\neg FD} = 0.17$). Additionally, the most significant difference is in JavaChat where the probability is seven times higher ($P_{FD} = 0.70$ vs. $P_{\neg FD} = 0.10$).

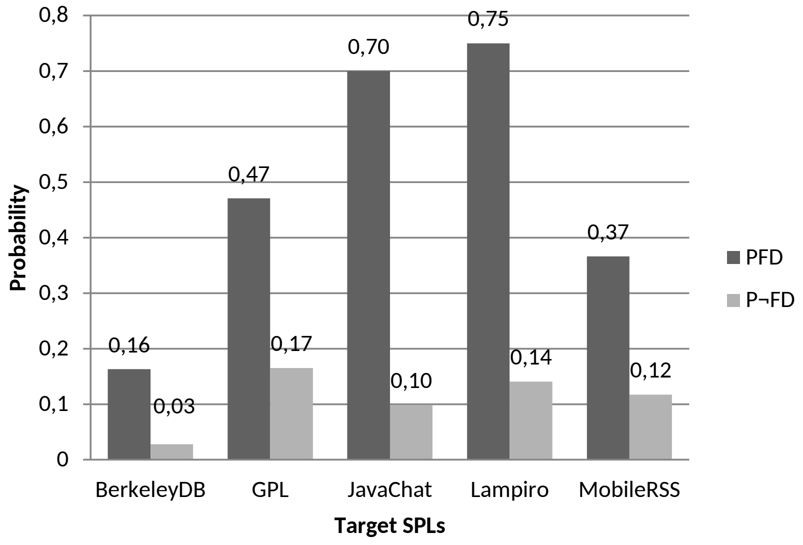


Figure 3.3: Comparison between P_{FD} and $P_{\neg FD}$.

Table 3.2 also shows the values of P_C and $P_{\neg C}$. These values show the probability that a feature dependency exists given that there is a simultaneous change (P_C) or not ($P_{\neg C}$) in features along product-line evolution. By comparing these values it is possible to conclude whether the occurrence of simultaneous changes is an indicative of the existence of a dependency between features. P_C ranges between 0.062 and 0.722 while $P_{\neg C}$ ranges between 0.003 and 0.275. Except for Lampiro, all values of P_C are greater than the highest value of $P_{\neg C}$ (0.275 for BerkeleyDB). The lowest value of P_C is 0.378 if Lampiro is discarded. Lampiro present odd results due to the way some dependencies between features are established. Our approach captures only feature dependencies based on the source as defined in Sections 2.1.2 and 3.1.2. However, there are some relationships between features in Lampiro that are not being considered by our definition. In this way, we are counting simultaneous changes between some of these features. Our approach does not consider these cases as a dependency. Nevertheless, the mean values presented in Table 3.2 also show that there is a considerable difference between P_C and $P_{\neg C}$. This difference can be observed in each product line as shown in Figure 3.4.

As we can see in Figure 3.4, there is a clear difference between P_C and $P_{\neg C}$ in terms of values in each product line. The lowest difference is presented in GPL where P_C is almost three times higher than $P_{\neg C}$ (0.38 and 0.12, respectively), while the most significant difference with almost thirteen times is

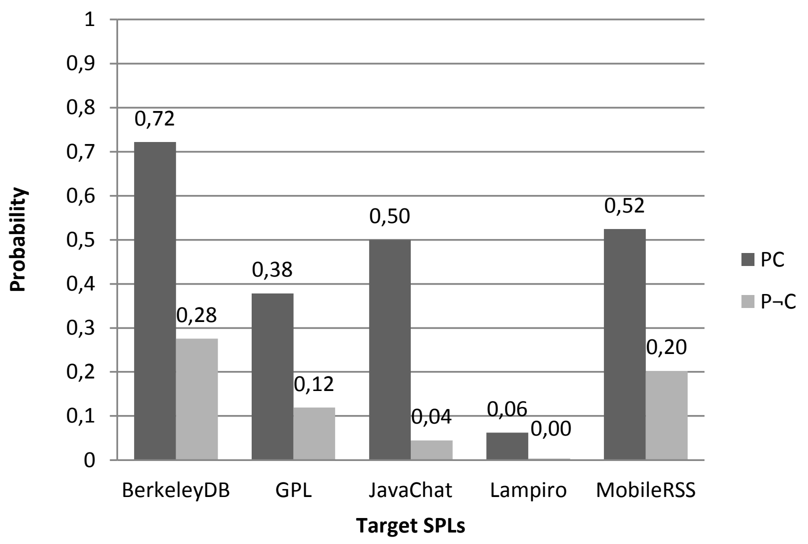


Figure 3.4: Comparison between P_C and $P_{\neg C}$.

encountered in JavaChat ($P_C = 0.50$ vs. $P_{\neg C} = 0.040$), excluding the unusual results of Lampiro.

In a nutshell, the results presented in this section shows a pronounced difference between P_{FD} and $P_{\neg FD}$. This difference provides evidence that the existence of a dependency raises the chance of a change propagation between features. In addition, the difference between P_C and $P_{\neg C}$ shows us that simultaneous changes are more likely to happen in dependent features than in independent features. A more in-depth analysis as well as the implications of the results are presented in Section 3.3.1

3.2.2 Inequality in Change Propagation Distribution

Figure 3.5 shows in dashed grey the Lorenz curve for BerkeleyDB. As a reference, the solid black line marks the line of equality. It can be seen that the Lorenz curve is strongly bent and that less than 20% of the dependencies concentrate 100% of the change propagation. This behaviour of inequality in the distribution of simultaneous changes can be confirmed by the high value of the Gini Coefficient g for BerkeleyDB ($g = 0.862$), which is not an exception amongst the target systems. Table 3.3 lists the Gini Coefficients g for all software product lines, and throughout the sample the concentration is very high; on average of 0.535. It is worth to notice that the coefficients for JavaChat and Lampiro are below of the mean. This happens because of the relative low number of feature dependencies.

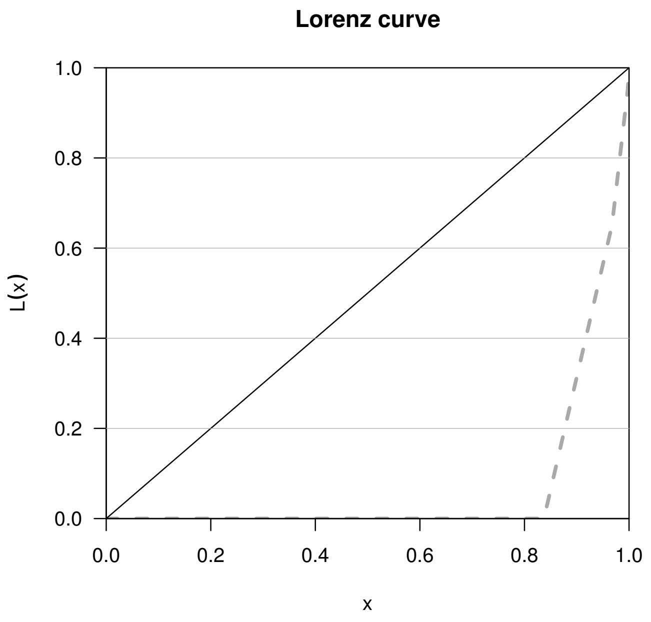


Figure 3.5: Concentration of change propagation through paths of feature dependencies in BerkeleyDB.

Table 3.3: Summary of Gini Coefficients.

Projects	**g**
BerkeleyDB	0.862
GPL	0.595
JavaChat	0.375
Lampiro	0.250
MobileRSS	0.657
mean	**0.535**

The data obtained in this section shows an inequality in the distribution of change propagation through the product-line dependencies. This result means that few feature dependencies are involved in many occurrences of change propagation. This is a counterintuitive finding that goes against the common assumptions on the relationship between module dependencies and change propagation. Some studies [TSANTALIS *et al.* 2005, SANGAL *et al.* 2011], which evaluate non-product lines, implicitly assume an equal distribution of change propagation amongst dependencies. In other words, dependencies are considered indicators of software changes in these studies. The analysis of the data and implications of the results are presented in Section 3.3.2.

3.2.3 Distance and Change Propagation

In Section 3.1.4, we observed that there is a decreasing probability of change propagation as the distance between features increases in the path of dependencies. In other words, if changes propagate along the paths of feature dependencies, there is a relationship between d and P_{FD}, and we assume that this relationship can be approximated as a linear function. Figure 3.6 presents a scatterplot of the relationship between d and P_{FD} in BerkeleyDB (solid black dots) and a reference graph with linear trend (grey dashed line). The linear trend of the relationship between d and P_{FD} in BerkeleyDB is not an exception amongst the software product lines analysed – i.e., all product lines presented the same linear trend between d and P_{FD}.

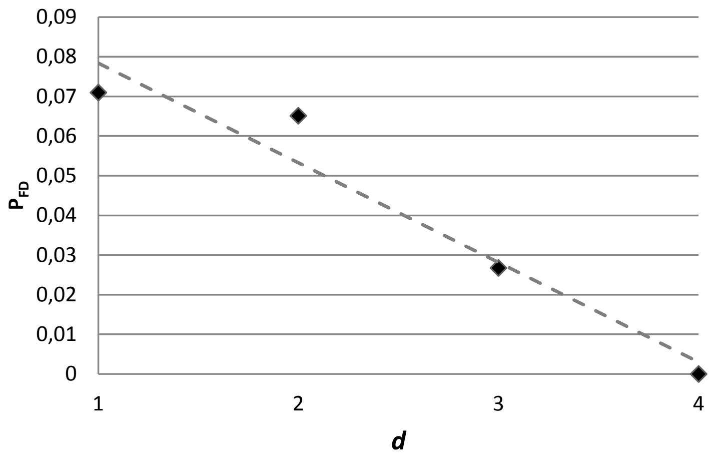


Figure 3.6: Relationship between d and P_{FD} in BerkeleyDB.

As we propose a statistical linear model to delineate $P_{FD}(d)$, we test the goodness of fit of the statistical model proposed regarding our data using the adjusted coefficient of determination ($adj.R^2$). As indicated by the high values of $adj.R^2$ in Table 3.4, the linear model proposed describes well $P_{FD}(d)$. BerkeleyDB has an $adj.R^2$ of 0.908. Moreover, all software product lines have an $adj.R^2$ larger than 0.7, evidencing the existence of change propagation, and thus a relation between probability of simultaneous change and distance along the paths of dependencies.

It is important to note that two product lines have values for $adj.R^2$ equal to 1 because the maximum values of d for these product lines are two. In other words, when $d = 2$ we obtain a linear relationship between d and P_{FD}.

Analysing the collected data, we can conclude that there is a transitive propagation of changes via a path of dependencies. Moreover, there is a decreasing probability of change propagation as the distance between

Table 3.4: Summary of the Adjusted Coefficient of Determination.

Projects	**Adj.R^2**
BerkeleyDB	0.908
GPL	0.799
JavaChat	1.000
Lampiro	1.000
MobileRSS	0.958
mean	**0.933**

dependent features increases in the path of dependencies. Finally, we can also observe that this relationship can be approximated as a linear function. This finding may have important implications. According to Geipel and Schweitzer, the extent of propagation in non-product-line systems has an exponential decay [GEIPEL and SCHWEITZER 2012]. Therefore, the extent of change propagation in product-line features might be more severe than the extent in files of non-product-line systems. The implications of this result is discussed in Section 3.3.3.

3.3 Discussion

This section reports an analysis of the results presented in Section 3.2 to explain the relation between feature dependencies and change propagation. Section 3.3.1 analyses the collected data for answering RQ1.1, i.e. the relation between feature dependency and simultaneous change. In order to address RQ1.2, Section 3.3.2 discusses the inequality of the distribution of change propagation through paths of feature dependencies. Finally, Section 3.3.3 presents a discussion about the extent of change propagated through feature dependencies and their implications in order to answer to RQ1.3.

3.3.1 The Relation between Feature Dependency and Change Propagation

The pronounced difference between P_{FD} and $P_{\neg FD}$ provides evidence that the existence of a dependency raises the chance of a change propagation between features. In addition, the difference between P_C and $P_{\neg C}$ shows us that simultaneous changes are more likely to happen in dependent features than in independent features. We can conclude that the results support our argument that dependencies are closely related to change propagation. In the following, we present two complimentary views that explain the close relation between feature dependencies and change propagation. In addition, we also explain the most recurring cases of simultaneous changes that are not related to change propagation.

Structural properties of feature dependency

Program elements establishing a single feature dependency may be distributed all over the code of many modules. Consequently, they tend to vanish from developer's eyes during a maintenance [RIBEIRO *et al.* 2014]. Thus, developers are likely to ignore consciously (or not) feature dependencies while reasoning about feature maintenance. Moreover, the implementation of feature dependencies may involve several parts of the code, possibly affected by a change propagation. To understand why feature dependencies are likely to propagate changes, we analysed the source code of the software product lines. In our analysis, we noticed that some developers often change parts of the code that affect program elements responsible for realising feature dependencies. Consequently, these actions demand changes on dependent features.

For instance, in BerkeleyDB the feature TRANSACTIONS depends on the feature ENVIRONMENT_LOCKING. The idea of the dependency is to lock the environment during a transaction. These two features changed simultaneously once along the evolution of BerkeleyDB. There was a change in the feature ENVIRONMENT_LOCKING in the second release of the BerkeleyDB. The change comprised the modification of a method responsible for locking several attributes of the database. The problem is that in the third release, both features changed simultaneously. The previous change did not preserve the consistency of between the features TRANSACTIONS and ENVIRONMENT_LOCKING. The developer overlooked important parts of the code of dependent features. The reason may be the number of program elements involved in this dependency. The feature dependency comprises 10 program elements (e.g. attributes and methods) scattered in 5 files. As a consequence, in order to preserve the consistency of the dependency, in the next release the same method of the feature ENVIRONMENT_LOCKING was changed again as well as two methods of the feature TRANSACTIONS.

Another interesting example in BerkeleyDB happened in the context of the feature dependency between TRANSACTIONS and CLEANER. The purpose of the dependency is to clean several configuration parameters after a transaction in the database. The implementation of the dependency between these features involves 23 program elements distributed along 3 files. Along the evolution of BerkeleyDB, these two features changed simultaneously four times. In the first time, a change in one method of the feature TRANSACTIONS was propagated to three methods of the feature CLEANER. However, in the second time, the change in one of the three methods of the feature CLEANER was undone due to a mis-propagated change.

As aforementioned in Chapter 1, the more complex is the structure of a feature dependency in the source code, the more influential it might be in change propagation. In the examples above, the widely-scattered implementation of a feature dependency was even related to two severe change propagation problems: (i) the first was an omission case, i.e. the change was not fully propagated, and (ii) the second was a mistake case, i.e. the change was propagated to the wrong program element. Therefore, the structural properties of feature dependency are likely to impact on change propagation. Structural properties of feature dependencies are characteristics related to the way a feature dependency is implemented in the source code, such as number of program elements involved in a feature dependency. In this way, exploring characteristics of the code structure related to change propagation, such as structural properties of feature dependency, may be primordial to support developers during software maintenance. Therefore, making structural properties of feature dependencies explicit to developers might support the change propagation in product lines. For instance, either computing or making the program elements involved in feature dependencies explicit would indicate the critical parts of the code that must be carefully changed due to a high probability of change propagation.

Feature dependencies are not alike

A cross-reading of the calculated probabilities shows us that the values of P_{FD} and $P_{\neg FD}$ are inversely proportional to P_C and $P_{\neg C}$, respectively. For instance, the lowest values of P_{FD} and $P_{\neg FD}$ from BerkeleyDB contrast with its highest values of P_C and $P_{\neg C}$. The low values of P_{FD} and $P_{\neg FD}$ in BerkeleyDB are due to the high number of feature dependencies (860 dependencies) and due to the number of change propagation happening in a few dependencies. These low values could lead us question the validity of the close relation between feature dependency and change propagation aforementioned. However, the high values of P_C and $P_{\neg C}$ confirm our statement by showing that when there is a simultaneous change between features, there is a high probability of having a dependency amongst these features. Thus, this behaviour in BerkeleyDB, which is not an exception amongst the the product lines analysed, can be interpreted as an indication of concentration of change propagation in certain feature dependencies. A few number of feature dependencies seems to concentrate a high number of recurrent changes, thus indicating that there are feature dependencies that are more likely to propagate changes. For instance, the dependency between features TRANSACTIONS and CLEANER were involved in 4 changes along the evolution of the BerkeleyDB. As aforementioned, the implementation of the dependency between these features involves 23 program elements distributed along 3 files. On the other hand, the dependency between features TRANSACTIONS and ENVIRONMENT_LOCKING were involved once in a change along the evolutions of the product line BerkeleyDB. This feature dependency is realised by 10 program elements scattered in 5 files.

The reason for this and other several inequalities found in all product lines analysed may be due to the complex structure of the feature dependency implementation – i.e. structural properties of feature dependencies. We can notice the differences of program elements and files involved in the implementation of the feature dependencies. In this way, exploring possible reasons that make change propagation more difficult – such as, structural properties of feature dependency may be primordial to understand and better support change propagation in product lines. In Section 3.3.2, we revisit in more detail the issue related to the inequality of the distribution of changes throughout dependencies (addressed by research question RQ1.2).

Non-dependent features

Another interesting point in our data is related to a specific case of simultaneous changes that happen in non-dependent features ($P_{\neg FD}$). There were many cases where non-dependent features changed simultaneously. The nonexistence of a feature dependency does not mean full absence of any form of relationship between features. Feature interactions [CALDER *et al.* 2003] and alternative features are other types of relationships between features not encompassed by our definition of feature dependency. To clarify the change propagation in non-dependent features, we explain the case of alternative features. Alternative features are features that are mutually exclusive in a product generated by the software product line. The way this type of constraint was implemented in our target product lines is not considered as a dependency in our study (see Section 3.1.2). These constraints are usually implemented by means of preprocessor directives. However, this mutually exclusive relationship between these features may demand change propagation. Alternative features often depend on the same program elements of a dependee feature. A change that affects these program elements will probably affect all alternative features using the same program elements, thus causing simultaneous changes in non-dependent features. Therefore, to alleviate situations where non-dependent features demand change propagation, we believe that making program elements realising feature dependencies explicit to developers may help reasoning about possible changes considering all types of relationships between features.

3.3.2 Consequences of the Inequality in the Distribution

The results presented in this study show a strong relation between feature dependency and change propagation. Apart from this point, the data obtained for answering RQ1.2 challenges the common assumptions on the relationship between dependencies and change propagation. Some studies [TSANTALIS *et al.* 2005, SANGAL *et al.* 2011], which evaluate non-product lines, implicitly assume an equal distribution of change propagation amongst dependencies. In other words, the dependency structure would be a measure for the software change behaviour in these studies. However, our data show an inequality in the distribution of change propagation through the product-line dependencies. So, it might be problematic to simply adapting stand-alone software approaches to analyse the change behaviour of a software product line. The inequality in the distribution of change propagation indicates that we cannot treat all dependencies alike. Few dependencies are related to most change propagation of a product-line evolution history. Based on this, we can pinpoint two important findings.

First, it is important to identify recurring characteristics in the code structure (e.g., structural properties of feature dependencies) that are related to the concentration of changes in certain feature dependencies. Certain feature dependencies may be involved more often in change propagation than other feature dependencies along the evolution, the so-called concentration of changes. The characteristics of those feature dependencies must also be explored. For instance, the realisation of some feature dependencies involve many program elements. We observed that those feature dependencies are frequently involved in change propagation. When a change affects one of the many program elements establishing such dependency, changes may be propagated to dependent features. Moreover, feature dependencies involving more program elements are more likely to propagate changes. Therefore, structural properties of feature dependencies (explored in Chapter 4) may be indicators that such dependencies are more likely to propagate changes – i.e., concentration of changes.

Second, as many dependencies are not involved in change propagation, only reducing all the dependencies will not necessarily reduce the change propagation effort in product lines. In fact, dependencies might indicate a high reuse [GEIPEL and SCHWEITZER 2012]. A dependency indicates that a functionality was not duplicated but reused. A highly connected feature is not necessarily an indicator of flawed design, but it might indicate it is a key feature in the product-line architecture. As a consequence, since these features are highly connected, it is natural they present a large number of program elements realising feature dependencies. In this case, it is very likely that not all program elements are relevant in a maintenance task. Therefore, we argue that, besides making these program elements explicit to developers (Section 3.3.1), organising the information about these program elements in an intuitive and helpful way would support developers to propagate changes in the presence of feature dependencies and drive efforts to specific parts of the source code. We will tame this problem in Chapter 5.

3.3.3 Extent of the Change Propagation

The results presented in the previous section (Section 3.3.1) indicate that transitive feature dependency is related to change propagation. In our research question RQ1.3, we were interested in discovering the extent of this propagation through paths of dependencies. The results presented in Section 3.2.3 show us the relation between probability of change propagation and distance between transitively dependent features. Analysing the collected data, we found that there is a decreasing probability of change propagation as the distance between transitively dependent features increases in the path of dependencies. Moreover, we observed that this relationship can be approximated as a linear function. So, these results provide evidence for the transitive propagation of changes via a path of dependencies.

This result about the relationship between distance and change propagation may have important implications. A structural property that characterises the distance of the dependency would indicate the possible paths of change in a product line. Features involved in these paths of feature dependencies are likely to suffer changes related to change propagation if the maintenance affects one of these features. Thus, developers should be also concerned with these distinct inter-related features during the product-line maintenance based on a property that indicates the distance between features. Reasoning about this property might help decreasing the maintenance effort of product lines along the evolutions.

In addition, the data show a linear decreasing relationship between distance and probability of change propagation indicating that the extent of change propagation in product-line features might be more severe than the extent in files of non-product-line systems. For instance, Geipel and Schweitzer found an exponential decay regarding the relation between probability of changes and distance amongst classes [GEIPEL and SCHWEITZER 2012]. In other words, it is more likely the extent of the change propagation in software product lines reach more features than the change propagation in modules of stand-alone programs. Therefore, change propagation in product lines can be a more complex task to be executed than in stand-alone programs. Therefore, the distance between features might help developers to concern about paths of feature dependencies during the maintenance of a feature in order to support the change propagation.

Another point in our data is related to the extent of change propagation in software product lines. Most of the product-line releases (77%) presented an distance higher than one. This information might be valuable to tune and/or complement existing approaches, such as combinatorial interaction testing. For instance, there are several sampling-based analysis techniques in combinatorial interaction testing that aim at covering all pairs of feature (i.e. distance of the dependency equals to 1), but disregard larger feature combinations [MARIJAN *et al.* 2013, PERROUIN *et al.* 2012, LOCHAU *et al.* 2012]. So, these approaches implicitly assume that a major fraction of feature dependencies are not transitive. According to our results, this assumption might have implications for the precision of those approaches. Since dependencies can propagate changes, it is possible that these dependencies can also propagate errors. In this case, approaches covering only a shallow distance of dependencies might be missing information to reveal errors, change propagation estimation, and the like.

3.4 Threats to Validity

This section discusses the study limitations based on the four categories of validity threats described by Wohlin et al. [WOHLIN *et al.* 2000]. Each category has a set of possible threats to the validity of an experiment. We identified these possible threats to our study within each category, which are discussed in the following with the measures we took to reduce each risk.

Conclusion Validity. The major risk here is related to the *random heterogeneity of subjects:* the chosen product lines came from different application domains (Section 3.1.3). In other words, there is a risk that the variation due to individual differences is larger than due to the treatment. Although this risk is considered a threat to the conclusion validity, it also helps to promote the external validity of the study by improving the ability to generalise the results of our experiment.

Internal Validity. The detected risk is that we are considering the data from multiple releases all together in our matrices of feature dependencies and simultaneous change. In other words, we are not considering the effect of time when analysing the concentration of change propagation on feature dependencies. Despite of the aggregated data, we argue that this risk is minimised due to the simulation of the evolution scenarios as well the number of evolution. Although simulating the evolution scenarios does not mean controlling the change propagation, we argue that we minimise the effect of time due to the diversity of the evolution scenarios. In other words, we minimise the chances of change propagations happen in feature dependencies that exist since early evolution by controlling the evolution scenarios.

Construct Validity. We detect two possible threats related to the *restricted generalizability across constructs*: (i) the use of conditional compilation as the variability mechanism for implementing features in the source code might increase the number of feature dependencies when compared to other variability mechanisms. With conditional compilation, features may be scattered in the source code. This means that a feature may present low cohesion, and, as consequence, a high coupling with other features (i.e. high number of feature dependencies), and (ii) we focus only in evolution with perfective maintenance, i.e. mainly the addition of new features. In this case, we might have other maintenance scenarios where the results are not similar to the ones presented in this study. Risk (i) cannot be completely avoided as all product lines analysed are implemented using the mechanism of conditional compilation. However, we argue that conditional compilation is the most widely used mechanism to implement product-line features [KÄSTNER and APEL 2009]. Moreover, except for the Graph Product Line, all target systems are non-academic projects. So, we believe these product lines were developed aiming for a good design, thus reducing the number of unnecessary feature dependencies. Nevertheless, the Graph Product Line was continuously improved over years by an academic community that uses it as a subject of studies. Risk (ii) also cannot be avoided due to the design of the study. However, we argue that more than 60% of the maintenance tasks are associated to a perfective maintenance [SCHACH *et al.* 2003]. In other words, it is the most common maintenance type in software evolution.

External Validity. We identified two risks in this category related to the interaction of the setting and treatment: (i) the target product lines might not be representative of the industrial practice, and (ii) the evolution scenarios might not represent relevant scenarios of evolution. We also identified one risk related to the interaction between selection and treatment: (iii) the subject responsible for implementing the evolution scenarios might not be representative of the population we want to generalise. In order to reduce risk (i), we evaluated product lines that come from heterogeneous application domains. In addition, all software product lines have been extensively used and evaluated in previous research [LIEBIG *et al.* 2010, APEL and BEYER 2011, RIBEIRO *et al.* 2011]. We believe the characteristics of the selected product lines, when contrasted with the state of practice in software product lines, represent a first step towards the generalisation of the findings observed in this study. Regarding risk (ii), we created product-line evolution scenarios based on previous studies of software product line evolution [ALVES *et al.* 2005, FIGUEIREDO *et al.* 2008, SVAHNBERG and BOSCH 2000]. We observed in these previous studies what are the recurring types of changes in product-line evolution. Moreover, we defined the procedures for the creation of each software product line release. Good design practices were used, enforced, and reviewed throughout the creation of evolution of all the product-line releases. An example of good design practice was to ensure the coupling of the dependent features was reduced to a minimum. In all the cases, the evolution scenarios were also reviewed. The intent of all these procedures was to guarantee that neither feature dependencies nor change propagation were artificially created, thus impacting on our conclusions. Finally, although this risk is considered a threat to the external validity, it also helps to promote the construct validity of the study. Regarding risk (iii), similarly to risk (ii), we argue that we defined clear procedures for the evolution, good design practices were used, and the evolution scenarios were systematically reviewed by experts in the field.

3.5 Related Work

Software change and change propagation. Previous research work have explored the understanding of software change and the impact of those changes on specific software properties [HASSAN and HOLT 2004, SANGAL *et al.* 2011, TSANTALIS *et al.* 2005, GEIPEL and SCHWEITZER 2012]. Most of these investigations, however, only concentrate on the analysis of module dependencies in stand-alone systems. For instance, Geipel and Schweitzer investigate the relationship between class dependency and change propagation [GEIPEL and SCHWEITZER 2012]. The study concludes a strong relationship between dependency and change propagation. Moreover, they revealed that half of all dependencies are never involved in change propagation. In a study about the impact of changes, Sangal et al. propose an approach that uses dependency models in order to manage complex software architecture [SANGAL *et al.* 2011]. Our work deals with two different main aspects when compared to those related work. First, we focus our research on feature change. Software product lines use features as the unit of abstraction. Thus, we explore changes on product-line features instead of the unit of abstraction of programming languages used to implement product lines (e.g.: classes or aspects). Second, we consider only product lines in our study. A software product line allows the generation of several products by combining different product-line features. So, the extent of a change propagation may affect from dozens to thousands different products depending on the characteristics of the product line.

Feature relationships vs. product-line maintenance. There are also investigations trying to understand and minimise negative effects of feature dependencies on product-line development. Cataldo and Herbsleb have empirically studied feature-oriented development in order to observe the impact that some attributes of this type of development have on integration failures [CATALDO and HERBSLEB 2011]. They concluded that dependencies and cross-feature interactions are drivers of integration failures. In another work related to feature dependencies, Ribeiro et al. performed an empirical evaluation on forty preprocessor-based software product lines focusing on the maintainability of feature dependency code [RIBEIRO *et al.* 2011]. They proposed an approach to reduce the effort of maintenance in product lines implemented using preprocessor by focusing on feature dependencies. The conclusion was that feature dependencies are reasonably common in preprocessor-based software product lines. Moreover, they highlight the impact of feature relationships on the maintenance by increasing the maintenance effort. These and other authors [FERBER *et al.* 2002, LEE and KANG 2004, GARVIN and COHEN 2011, APEL and BEYER 2011, CAFEO *et al.* 2012, CAFEO *et al.* 2013], have been highlighted the impact of feature dependencies on several attributes of product lines. In our work, we try to understand the link between feature dependency and change propagation because (i) our novel contribution is to empirically analyse the relation between change propagation and feature dependency, and (ii) change propagation may impact on several other software product lines attributes such as failures and maintenance effort.

3.6 Summary

Understanding whether and how feature dependencies propagate changes is important to reduce the maintenance complexity. First, maintainers need to be better informed about possible drivers of change propagation complexity. Feature dependency is a common type of relationship between features and it may be a change propagator to dependent features. If so, exploring how changes propagate through feature dependencies is essential. In this context, this chapter reported a study assessing the relation between change propagation and feature dependency as well as how propagated changes behave in presence of feature dependencies. Our study confirms the close relation between feature dependency and change propagation. In other words, feature dependencies are important drivers of change propagation.

We have also observed a number of new interesting outcomes as discussed along Section 3.3. For instance, according to our results, there is a high probability of a feature dependency propagating changes (average of $\approx$51%). Moreover, we identified the way feature dependencies are implemented (structural properties of feature dependency) may impact on change propagation. In addition, we also identified parts of the source code that are possible causes of this strong relation – i.e., code associated with feature dependency implementation. The results also revealed an inequality in the distribution of change propagation through the feature dependencies of a product line. This counterintuitive result indicates that (i) a general feature dependency minimisation might not reduce change propagation effort; and (ii) characterising structural properties of feature dependency is essential to distinguish feature dependencies, and thus help propagating changes. Finally, our analysis provided evidence about the existence of change propagation through paths of feature dependencies. Such analysis pointed to a linear relation between depth of dependency and change propagation, which indicates a more powerful change propagation in features than the exponential relation between distance and classes [GEIPEL and SCHWEITZER 2012].

As a consequence of the results, it is evident the need for characterising structural properties of feature dependency. The idea behind this investigation is to identify structural properties and investigate their correlation with change propagation (Chapter 4). As these structural properties may define the behaviour of change propagation, there is also a demand for strategies to explore such properties in order to support change propagation (Chapter 5).

4
The Role of Feature Dependency Properties on Change Propagation

Change propagation is a central aspect of product-line maintenance. There is growing evidence that certain characteristics of feature dependency impact on change propagation (Chapter 3). The more complex is the structure of a feature dependency, the more influential it might be to change propagation. Therefore, it becomes primordial defining and evaluating indicators to capture various complexity characteristics of feature dependency structure. The indicators should help developers to overcome the cognitive difficulties caused by change propagation in the presence of feature dependencies.

In a seminal study, Simon has extensively argued about the limitations of human cognitive capabilities [SIMON 1978]. Developers can lose track of what they are doing and make irrational decisions in complex contexts, the so-called cognitive complexity. Cognitive complexity is defined in this thesis as the mental burden of developers when performing relevant operations in a specific context [BAGHERI and GASEVIC 2011]. Hence, the more complex are feature dependencies, the higher is the cognitive complexity of a product line. Consequently, high cognitive complexity of maintaining software product lines may be related to complex feature dependencies, which might affect change propagation.

According to the causal chain model proposed by Briand et al., which provides the basis for empirical research on software indicators, structural properties are often the main reasons for the cognitive complexity [BRIAND *et al.* 1999]. As illustrated in the causal chain model (Figure 4.1), structural properties of the source code have impact on the cognitive complexity of software [BRIAND *et al.* 1999, BRIAND *et al.* 1999b]. The cognitive complexity of a program affect the difficulty of performing some tasks in the source code, such as change propagation. By adapting the causal chain model (Figure 4.1), we illustrate the findings of several authors stating that structural properties of non-product lines (e.g. coupling) can be used as indicators of change propagation [MANCORIDIS *et al.* 1998, SANGAL *et al.* 2011, GEIPEL and SCHWEITZER 2012].

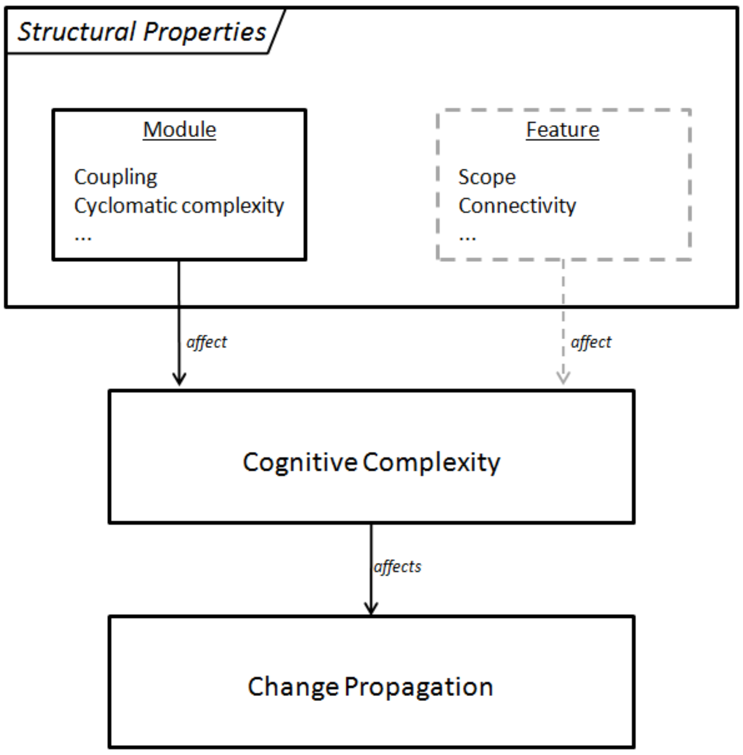


Figure 4.1: Relationship between structural properties, cognitive complexity and change propagation.

Some authors state that measurement frameworks are required to characterise attributes affecting the cognitive complexity of a program [CANT *et al.* 1994, BRIAND *et al.* 2001, FIGUEIREDO *et al.* 2008b]. Measurement frameworks have been mainly proposed to instantiate measures for software systems [FIGUEIREDO *et al.* 2008b]. A measurement framework can externalise relevant software structural properties (e.g., module coupling [CHIDAMBER and KEMERER 1994]) by means of a metrics suite. This externalisation helps developers to overcome the cognitive complexity by supporting quantitative analyses of structural properties. In addition, Briand and colleagues state that cognitive complexity of systems lies in the way modules collaborate [BRIAND *et al.* 1999]. In this case, it is reasonable to analogously hypothesise that structural properties of feature dependencies are related to cognitive complexity in software product lines. This hypothesis is represented by the dashed box and dotted arrow in Figure 4.1. Those structural properties of feature dependencies may affect change propagation in software product lines. Thus, it is important to identify relevant structural properties that are able to externalise the complexity of feature dependencies.

In the context of product lines, several studies make use of conventional module-oriented metrics [VAN DER HOEK *et al.* 2003, FIGUEIREDO *et al.* 2008, LIEBIG *et al.* 2010, DANTAS and GARCIA 2010, TIZZEI *et* *al.* 2011, APEL and BEYER 2011]. On the other hand, some authors argue that structural properties cannot be straightforwardly detected with conventional module-oriented metrics [VAN DER HOEK *et al.* 2003, APEL and BEYER 2011], such as Chidamber and Kemerer metrics [CHIDAMBER and KEMERER 1994]. The extension of existing measurement frameworks to cope with software product line is not straightforward since they need to be adapted in a number of ways. Consequently, they lack of standard terminology and formalism, leading to definitions of measures, which are ambiguous and/or difficult to understand. A growing number of studies have identified structural properties of product lines [SOBERNIG *et al.* 2014, PASSOS *et al.* 2015, QUEIROZ *et al.* 2015] and metrics for product-line [VAN DER HOEK *et al.* 2003, LIEBIG *et al.* 2010, APEL and BEYER 2011] to overcome the problems aforementioned. Their common goal is the association between: (i) the quantification of characteristics governing either product lines or programming techniques, and (ii) their impact on software maintainability. However, none of these studies focus on structural properties of feature dependency.

The area of product line measurement is still in its infancy. Particularly, feature dependency measurement suffers from not having a unified measurement framework. The terminology used is diverse and ambiguous. Their definitions make it not clear the target level of abstraction. They rely on terms of specific research groups and on specific implementation approaches, thereby hampering: (i) the process of instantiating measures, (ii) their adoption in academic and industry settings, and (iii) independent interpretation of the measurement results. In summary, there is a need to characterise and explore structural properties of feature dependency. Those structural properties can be used to compound a measurement framework in order to support developers on characterising and overcoming the cognitive complexity of software product lines.

Therefore, this chapter aims at answering the second research question (RQ2 in Section 1.2), which states: *Are structural properties of feature dependency good indicators of change propagation?* To do so, we first proposed a measurement framework to quantify structural properties of feature dependencies. Section 4.1 presents the requirements of the measurement framework. Section 4.2 reports the characterisation of structural properties of feature dependency composing our measurement framework. Section 4.3 describes the metrics suite proposed for quantifying the structural properties. Section 4.4 reports the evaluation of the framework through an empirical study. Section 4.5 presents data analysis and results of our study. Sections 4.6 and 4.7 present the threats to validity and related work, respectively. Finally, Section 4.8 describes the summary of the chapter. The results of this chapter were formally published in previous papers [CAFEO *et al.* 2012, CAFEO *et al.* 2013].

4.1 Measurement Framework Requirements

The purpose of our measurement framework is to instantiate metrics that are able to quantify structural properties of feature dependency. The characterisation of those structural properties is a starting point for promoting the quantification of metrics in order to indicate change propagation (Section 4.3). In this way, developers might be able to better cope with change propagation in the presence of feature dependencies. To reach these goals, five important requirements must be guaranteed:

Generality. The proposed framework should be generic enough to be used in different product-line implementation approaches. Software product line can be implemented using different implementation approaches (Section 2.2). Defining concepts based on programming techniques of a specific implementation approach could limit the use of our measurement framework. Therefore, our goal is to propose a measurement framework able to be employed in both annotative and compositional implementation approaches. To satisfy this requirement, the definition of structural properties of feature dependencies as well as the definition of metrics should be independent of implementation approaches.

Usage simplicity. Another important requirement of our measurement framework is to be simple to use. Existing measures for software product lines lack of standard terminology and formalism, leading to definitions of measures, which are ambiguous and/or difficult to understand. We deployed effort in representing all the framework elements through a few concepts, which are relevant to understand structural properties of feature dependencies. These concepts are associated with a basic terminology. Our aim is to avoid concepts to be expressed in an ambiguous and difficult manner.

Empiricism. Our measurement framework should be inspired in findings of empirical studies. However, there is no characterisation of structural properties of feature dependency in the literature. Consequently, there is no understanding about the role of structural properties of feature dependency on change propagation. To bridge this gap, we use previous studies to characterise structural properties of feature dependencies in our framework. The main goal is to ensure that our framework considers important structural properties associated with change propagation in previous studies (e.g., [APEL and BEYER 2011, RIBEIRO *et al.* 2011, CAFEO *et al.* 2012, CAFEO *et al.* 2013, RIBEIRO *et al.* 2014, PASSOS *et al.* 2015]). In addition, we also rely on the findings of a study previously presented in this thesis (e.g., Chapter 3). The findings of those studies must give hints that certain structural properties of feature dependencies eventually may have impact on change propagation.

Direct Properties of Feature Dependencies. Another main requirement of our measurement framework is that the characterisation of properties must be of direct properties of product-line feature dependencies. A direct property of feature dependency aims at defining attributes derived from the code structure of each feature dependency. Therefore, we do not resort to dependencies of enclosing modules to infer structural properties of inner feature dependencies.

Extensibility. Our last requirement concerns the extensibility of our measurement framework. The framework should expect extensions. In the context of our framework, an extension comprises, for instance, the definition of indirect structural properties of feature dependencies. For instance, composite properties of feature dependencies may also exert an impact on change propagation. Examples of composite properties comprise combinations of structural properties or properties considering multiple feature dependencies. Therefore, the idea is to allow developers to define such additional properties to enhance the power of our framework. We take advantage of the ease of use of our framework to satisfy this requirement. To provide evidence about the extensibility of our framework, we define a composite property in Section 4.2.3 and a metric for quantifying this property (Section 4.3).

4.2 Characterising Structural Properties of Feature Dependency

Our proposed measurement framework consists of basic concepts, structural properties of feature dependencies and a metrics suite for feature dependencies. Figure 4.2 illustrates a basic set of concepts related our measurement framework. This metamodel defines the relationship between different concepts of a program (i.e., a software product line), such as module, program elements, feature, feature dependencies, and the structural properties of feature dependencies.

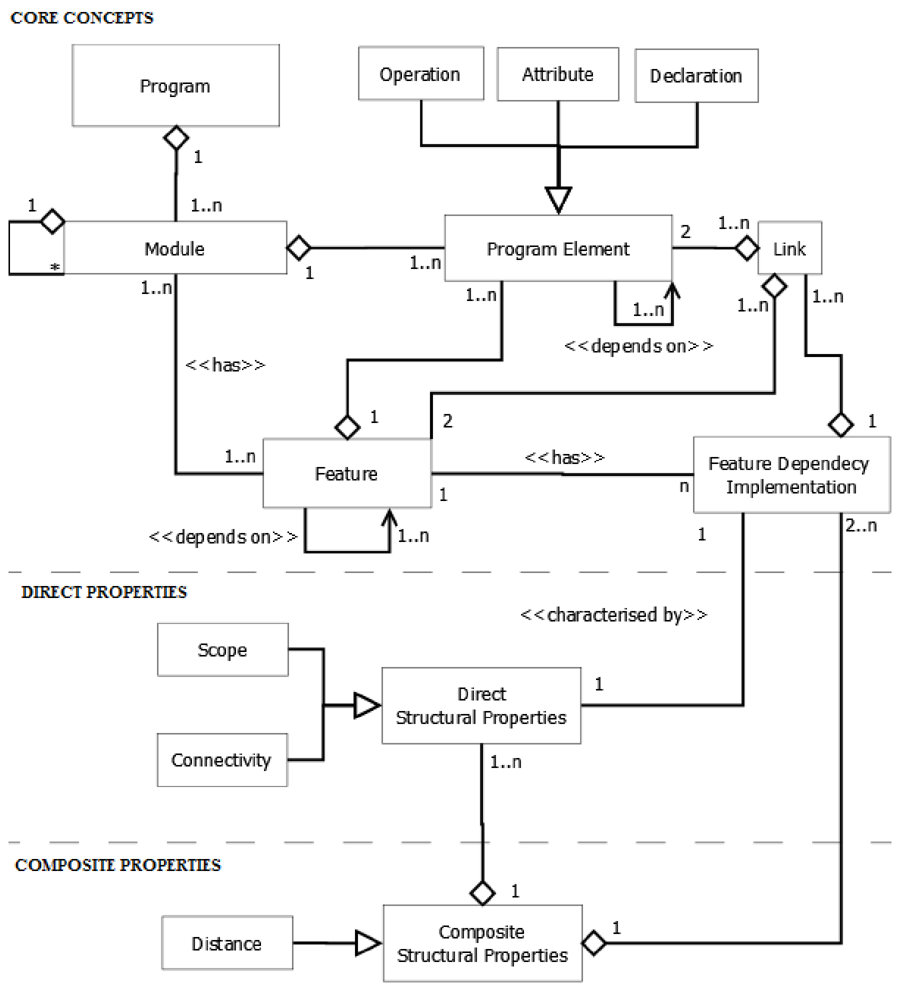


Figure 4.2: Meta model with basic concepts of the measurement framework.

Section 4.2.1 presents the basic terminology with a running example. This example will help the understanding of all concepts of the framework. Section 4.2.2 describes the direct properties of feature dependency, which may have some impact on change propagation. Section 4.2.3 describes a composite property created to show how our framework can be extended. Section 4.3 presents the metrics suite based on the structural properties of feature dependencies defined in the measurement framework.

4.2.1 Basic Terminology

To formalise our framework concepts, we have chosen set theory as it has been widely applied to define measurement frameworks in the literature [BRIAND *et al.* 1999, BARTOLOMEI *et al.* 2006, FIGUEIREDO *et al.* 2009, DANTAS *et al.* 2012]. In addition, set theory has an expressive power that allows us to capture the essence of each structural property of feature dependencies. The basic concepts of our framework are firstly introduced with a running example.

Running Example. Figure 4.3 shows a class diagram representing a single feature dependency in a product line. This feature dependency involves F1 and F2, which are enclosed by a subset of program modules. The set of modules realising this dependency is formed by `Class_A`, `Class_B`, `Class_C`, `Class_D` and `Class_E`. Each module contains a set of program elements. A program element is a sequence of statements. There exist three types of program elements: attributes, operations and declarations. These three program elements are generic enough to be mapped to specific elements in different programming techniques. For instance, methods are classified as operations in conditional compilation, whilst pointcut expressions and intertype declarations are classified as declarations in programming techniques like AspectJ [KICZALES *et al.* 1997].

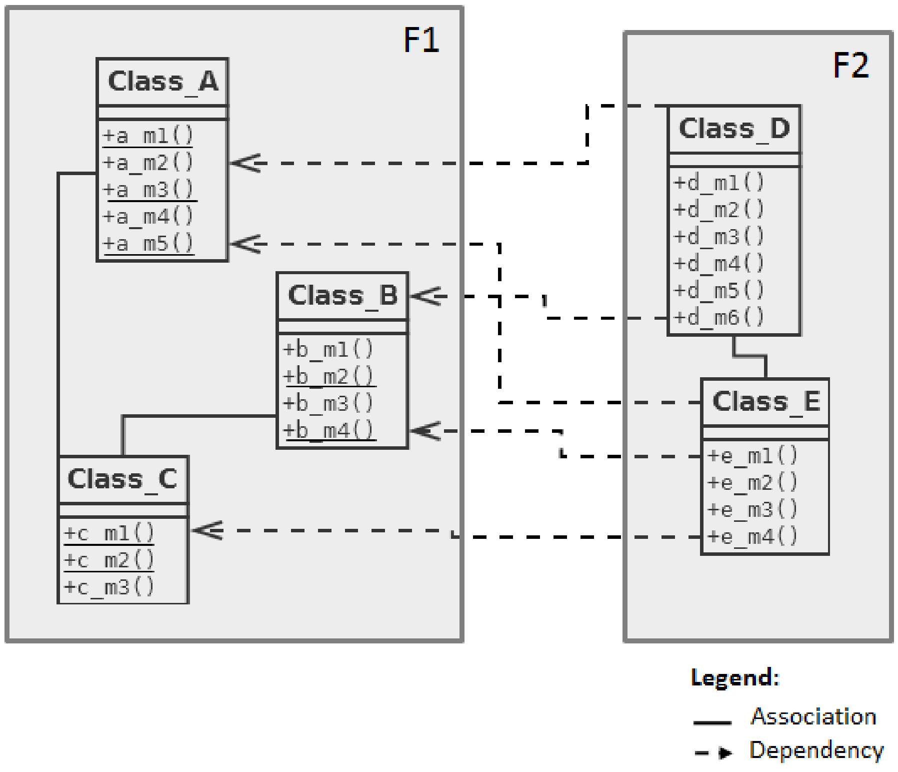
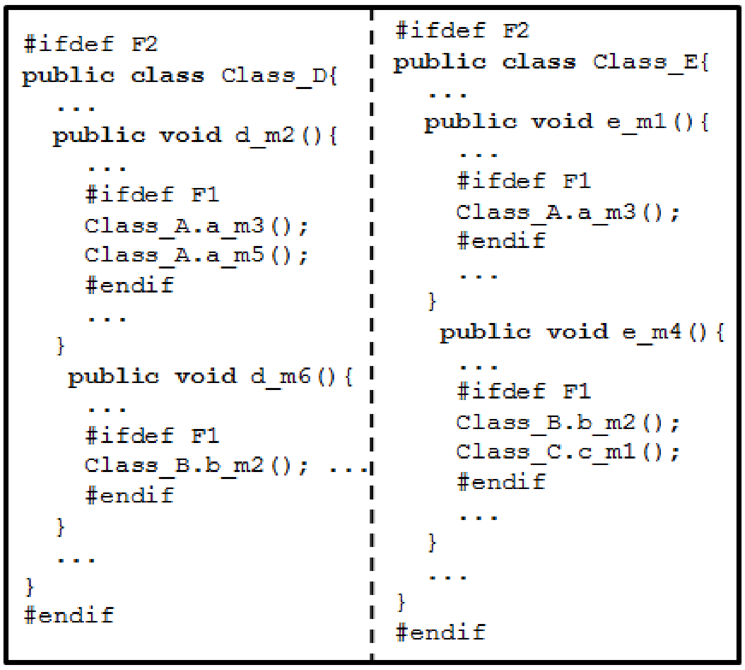


Figure 4.3: Feature dependency in conditional compilation (Java).

In this example, we can notice that feature F1 consists of three classes (`Class_A`, `Class_B` and `Class_C`), whereas F2 is implemented by two classes (`Class_D` and `Class_E`). We can also notice that `Class_D` and `Class_E` present a dependency relationship with `Class_A`, `Class_B` and `Class_C`. In other words, these references establish the feature dependency of F2 on F1. Figure 4.4 illustrates the implementation of the feature dependency F2 on F1 using the annotative approach. The implementation of the feature dependency is based on conditional compilation, the chosen representative technique for annotative approaches in our example. The reader can refer to Section 2.2 for more details about conditional compilation. It is also important to mention that, for clarity and simplicity's sake, we considered that each class is entirely dedicated to implement a feature. However, the same reasoning applies to classes that partially implement features.



```
#ifdef F2
public class Class_D{
  ...
  public void d_m2(){
    ...
    #ifdef F1
    Class_A.a_m3();
    Class_A.a_m5();
    #endif
    ...
  }
   public void d_m6(){
    ...
    #ifdef F1
    Class_B.b_m2(); ...
    #endif
  }
  ...
}
#endif
```

```
#ifdef F2
public class Class_E{
  ...
  public void e_m1(){
    ...
    #ifdef F1
    Class_A.a_m3();
    #endif
    ...
  }
   public void e_m4(){
    ...
    #ifdef F1
    Class_B.b_m2();
    Class_C.c_m1();
    #endif
    ...
  }
  ...
}
#endif
```

Figure 4.4: Implementation of a feature dependency.

Figure 4.4 shows the partial implementation of classes `Class_D` and `Class_E`. Both classes implement the feature F2. Therefore, theses classes are surrounded by a preprocessor directive identifying them as part of feature F2. In class `Class_D`, there are two methods (`d_m2()` and `d_m6()`) calling three methods from feature F1 (`a_m3()`, `a_m5()` and `b_m2()`). In class `Class_E`, there are also two methods (`e_m1()` and `e_m4()`) calling three methods from feature F1 (`a_m3()`, `b_m2()` and `c_m1()`). The relationships formed by the method calls are responsible for realising the feature dependency between F2 and F1.

The key concepts of our framework are formalised through the definitions 1 to 3 and illustrated using the running example presented above.

Definition 1 (Program, Module and Program Element). *A Program P consists of a set of modules M. A module M is composed by a set E_M of* *program elements e located in the same physical file. A program element e can be an attribute, an operation or any other form of declaration. Let Att_M be the set of attributes of M, Op_M be the set of operations of M and Dec_M be the set of declarations of M. The set E_M of program elements of M is defined as $E_M = Att_M \cup Op_M \cup Dec_M$.*

We define three useful components for further definitions: program, module and program element. A program is a set of modules, which consist of program elements. A program element can be an attribute, an operation or any other form of declaration. For instance, in languages supporting conditional compilation, methods are classified as operations. In languages such as AspectJ, pointcut expressions and intertype declarations are classified as declarations. In Figure 4.3, the modules of the program P are $M = \{\texttt{Class\_A}, \texttt{Class\_B}, \texttt{Class\_C}, \texttt{Class\_D}, \texttt{Class\_E}\}$. The program elements of these modules are their methods: $E_{Class\_A} = \{\texttt{a\_m1()}, \texttt{a\_m2()}, \texttt{a\_m3()}, \texttt{a\_m4()}, \texttt{a\_m5()}\}$, $E_{Class\_B} = \{\texttt{b\_m1()}, \texttt{b\_m2()}, \texttt{b\_m3()}, \texttt{b\_m4()}\}$, $E_{Class\_C} = \{\texttt{c\_m1()}, \texttt{c\_m2()}, \texttt{c\_m3()}\}$, $E_{Class\_D} = \{\texttt{d\_m1()}, \texttt{d\_m2()}, \texttt{d\_m3()}, \texttt{d\_m4()}, \texttt{d\_m5()}, \texttt{d\_m6()}\}$, and $E_{Class\_E} = \{\texttt{e\_m1()}, \texttt{e\_m2()}, \texttt{e\_m3()}, \texttt{e\_m4()}\}$. Therefore, $E_M = E_{Class\_A} \cup E_{Class\_B} \cup E_{Class\_C} \cup E_{Class\_D} \cup E_{Class\_E}$ is the set of program elements of all modules.

Modular units and programming mechanisms of the most programming techniques used to implement product lines were not designed with the idea of feature as a modular abstraction. Therefore, it is common to find a misalignment between the boundaries of feature implementation and modular units of programming techniques. Therefore, program elements implementing a feature in the source code are often spread over several modules' boundaries. Furthermore, a single module may contain intertwined program elements from different features [RIBEIRO *et al.* 2011]. In this way, we define feature as follows.

Definition 2 (Feature). *A feature F consists of a set of program elements, E_F, so that $E_F \subset E_M$.*

In our example (Figure 4.3), feature F1 is implemented by modules `Class_A`, `Class_B` and `Class_C` and their program elements; whereas feature F2 is implemented by modules `Class_D` and `Class_E` and their program elements. In other words, a feature comprises all program elements that are related to its implementation. Therefore, $E_{F1} = \{E_{Class\_A} \cup E_{Class\_B} \cup E_{Class\_C}\}$ and $E_{F2} = \{E_{Class\_D} \cup E_{Class\_E}\}$.

It is important to mention that features are defined as a set of program elements. Due to the misalignment between features and modules, program elements of features might belong to different modules. In addition, different modules may have program elements belonging to different features. Therefore, despite our running example comprises entire modules implementing a single feature, the same reasoning applies to modules partially implementing features. Our definition of feature relies only on program elements instead of modules. In other words, the definition of feature can be applied to program elements of modules implemented more than a single feature.

Definition 3 (Feature Dependency). *A feature dependency D_{FyFx} of F_y on F_x (i.e., F_y depends on F_x) can be defined as a set of at least one 4-tuple (e_j, e_i, F_y, F_x), where $e_j, e_i \in E$ and $e_j \in E_{F_y}$ and $e_i \in E_{F_x}$ and $F_y \neq F_x$.*

In the source code, a feature dependency is materialised when at least one program element inside the boundaries of a feature depends on a program element outside the feature. We refer to each relationship between program elements from different features (i.e., a 4-tuple from Definition 3) as a *link* from the dependent feature to the dependee feature. In Figure 4.3, for instance, there is a feature dependency between F_2 and F_1. Analysing the implementation shown in Figure 4.4, we can notice the several links between program elements from different features. All these links are part of the feature dependency of F_2 on F_1. We can define the feature dependency of F2 on F1 as the set of all links from F2 to F1, such as $D_{F2F1} = \{(\texttt{Class\_D.d\_m2}, \texttt{Class\_A.a\_m3}, F_2, F_1), (\texttt{Class\_D.d\_m2}, \texttt{Class\_A.a\_m5}, F_2, F_1), (\texttt{Class\_D.d\_m6}, \texttt{Class\_B.b\_m2}, F_2, F_1), (\texttt{Class\_E.e\_m1}, \texttt{Class\_A.a\_m3}, F_2, F_1), (\texttt{Class\_E.e\_m4}, \texttt{Class\_B.b\_m2}, F_2, F_1), (\texttt{Class\_E.e\_m4}, \texttt{Class\_C.c\_m1}, F_2, F_1)\}$.

In our running example, entire modules implement only part of a single feature. It might be the case that a single module implements more than one feature. In this case, it is also possible to find feature dependencies being realised within the same module. Once again, we highlight that our definition of feature dependency relies only on links between program elements. Therefore, feature dependencies established within inner program elements of a single module are also considered in our definition.

4.2.2 Direct Properties of Feature Dependency

Feature dependency entails new dimensions of complexity in product line. The understanding of feature dependencies relies on the understanding of their structural properties. Consequently, structural properties may impact on cognitive complexity, which in turn may impact on change propagation [BRIAND *et al.* 1999b, BAGHERI and GASEVIC 2011]. In the following, we characterised two direct properties of feature dependencies. These properties encompass the most important characteristics of the code structure of feature dependency. These properties might be important to reason about when propagating a change. The realisation of these properties on the source code is explained using the example provided in Section 4.2.1.

Feature Dependency Scope

The property *scope* refers to the enclosing context of the feature dependency implementation. The unit of measurement is still the feature dependency, but we compute the set of program elements of both dependent and dependee features involved in a feature dependency. This property externalises the complexity of understanding the (several) program elements realising a feature dependency.

The rationale behind this property is that developers should know the program elements involved in a feature dependency. This knowledge becomes primordial to understand the feature dependency in the source code and to reason about change propagation. The program elements involved in a feature dependency are the program elements that are likely to be changed. Some empirical results in the literature (e.g. [RIBEIRO *et al.* 2011, DANTAS *et al.* 2012]) and in our previous findings in Chapter 3 (Section 3.3.1 and 3.3.2) point towards this direction. Thus, developers can understand the whole dependency context by knowing the program elements realising it. Moreover, they must be aware of program elements more likely causing or suffering a change propagation.

Taking into consideration the example illustrated in Section 4.2.1, the feature dependency scope is characterised by the program elements involved in the feature dependency implementation of F2 on F1. In other words, the scope of the feature dependency of F2 on F1 involves the method callers and method callees. A change in a callee may propagate to a caller. Therefore, the program elements in the scope of the feature dependency implementation between F2 and F1 are: `a_m3()`, `a_m5()`, `b_m2()`, `c_m1()`, `d_m2()`, `d_m6()`, `e_m1()` and `e_m4()`.

Feature Dependency Connectivity

The property *connectivity* refers to the strength on the connection of a dependent feature on a dependee feature. Thus, this property represents the degree to which program elements of a feature depends on program elements of other feature. A tight connectivity occurs whenever a feature dependency has several links (see Definition 3). In this case, the likelihood a change be propagated from the dependee feature to a dependent feature is higher than a dependency with loose connectivity. Moreover, the number of program elements to be inspected is higher when there is a tight connectivity in the feature dependency. Consequently, a tight connectivity may represent an increase in the change propagation effort, as shown in literature (e.g. [APEL and BEYER 2011, DANTAS *et al.* 2012]) and in Chapter 3 (Section 3.3.2).

Considering the running example (Section 4.2.1), it is important to note that there are six links (i.e., six method calls) from feature F2 to feature F1, which comprises the 4-tuples of the set D_{F2F1}: (`D.d_m2`,`A.a_m3`,F_2,F_1), (`D.d_m2`,`A.a_m5`,F_2,F_1), (`D.d_m6`,`B.b_m2`,F_2,F_1), (`E.e_m1`,`A.a_m3`,F_2,F_1), (`E.e_m4`,`B.b_m2`,F_2,F_1), (`E.e_m4`,`C.c_m1`,F_2,F_1). In other words, the dependency between F2 and F1 can be described in terms of these six links between them.

4.2.3 Composite Properties of Feature Dependency

Direct properties of feature dependencies are the basic properties needed to start understanding important characteristics of feature dependencies. However, analysing only direct properties of feature dependencies may not be enough to reason about change propagation in some specific situations. So, we give an example of a property created using our framework. This property explicitly shows how our framework satisfies two requirements: “Empiricism” and “Extensibility” (Section 4.1). The property defined here is inspired in one finding presented in Chapter 3. This finding pinpoints the importance of the path of feature dependencies on change propagation. In addition, this property extends our measurement framework regarding our direct properties of feature dependencies previously defined in Section 4.2.2.

Feature Dependency Distance

This property refers to the path of dependencies between two features. The property *distance* concerns the length of a shortest path from one feature to another one. Thus, this property refers to transitive dependencies by considering the distance of the path of dependency for each program element involved in a transitive feature dependency. A path of feature dependencies is characterised when the same program element of a feature participates of different feature dependencies, and such feature plays the role of dependent and dependee feature in those different feature dependencies. It is important to mention that a feature dependency is defined as a directed relationship. In other words, the distance from feature F1 to F2 does not necessarily coincide with the distance from feature F2 to F1.

Features might propagate changes along the path of feature dependencies. In other words, a change in one feature may cause a change cascade along the path of feature dependencies even to distant features. In this context, the more features are affected by changes, the more costly it may be to maintain a software product line. So, based on findings of Chapter 3 (more specifically in Sections 3.1.4 and 3.3.3), we argue that it is important to understand the extent of the change propagation through dependencies, since developers often inspect the code of direct neighbour features at best.

Considering the running example (Section 4.2.1), let us suppose there is a feature F3 depending on feature F2 (not represented in the example), and this dependency is defined by $D_{F3F2} = \{(\texttt{G.g\_m1}, \texttt{D.d\_m6}, F\_3, F\_2), (\texttt{G.g\_m2}, \texttt{E.a\_m3}, F\_3, F_2)\}$. We should notice that feature F3 depends on feature F2, which depends on feature F1. So, we can say that feature F3 depends on F1 by composing the information of two feature dependencies (D_{F2F1} and D_{F3F2}). In addition, the distance from feature F3 to feature F1 is 2 while the distance from feature F1 to feature F3 is zero. The dependency path is characterised by the link from feature F3 to F2 – ($\texttt{G.g\_m1}$,$\texttt{D.d\_m6}$,F_3,F_2) –, and by the link from feature F2 to F1 – ($\texttt{D.d\_m6}$,$\texttt{B.b\_m2}$,F_2,F_1).

4.3 The Measurement Suite

This section presents a metrics suite that relies on the terminology presented in Section 4.2.1. The metrics are intended to quantify the structural properties of feature dependencies (Sections 4.2.2 and 4.2.3). The goal is also to provide support for studying and assessing the impact of structural properties of feature dependency on change propagation (Figure 4.1).

We defined four metrics based on structural properties of feature dependencies, namely: Local Scope (LoS), Global Scope (GoS), Feature

Dependency Connectivity (FDC), and Feature Dependency Distance (FDD). An overview of these metrics is presented in Table 4.1. This table provides short definitions of the metrics. In addition, Table 4.1 also illustrates the association of the defined metrics with the structural properties they are intended to measure. Each metric is described in terms of: (i) an informal definition (Table 4.1), (ii) a formal definition based on the terminology presented in Section 4.2.1, and (iii) an illustrative example.

The relationship between the structural properties (Sections 4.2.2 and 4.2.3) and the proposed metrics are illustrated in Table 4.1, column 2. The metrics LoS and GoS are directly associated with the scope of the feature dependency implementation. Therefore, they are used to quantify the structural property of scope of a feature dependency. The density of the dependency between two features is quantified by the FDC metric. In other words, this metric quantifies how strong the connection between dependent feature and dependee feature is. Finally, the distance from one feature to another is quantified by the FDD metric associated with distance, a composite property of feature dependency.

Table 4.1: Feature dependency metrics.

Metric	**Property**	**Metric Definition**
GoS	Scope	Quantifies the feature dependency implementation scope by counting all the program elements participating of a feature dependency implementation over the number of program elements of two features involved in the dependency.
LoS	Scope	The ratio between the number of program elements participating of a feature dependency in a module over the total of program elements in a module.
FDC	Connectivity	The ratio of the the number of links between dependent and dependee feature over the number of program elements in the dependent feature participating of the dependency.
FDD	Distance	The length of the shortest path between two features considering each program element involved in a feature dependency chain

For the formal definition of the metrics, let E_{FDy} be a set of all e_j that take part of at least one 4-tuple of the set of the feature dependency D_{FyFx}, and E_{FDx} be a set of all e_i that take part of the same feature dependency D_{FyFx}. In addition, let SM be a set of modules of P, SF be a set of features of P, and $SD = D_{FyFx} \cup D_{FxFy} \cup \ldots, \forall Fy \in SF$ and $Fx \in SF$ be a set of all feature dependencies of P.

Global Scope (*GoS*) metric. Given a program P, the GoS of D_{FyFx} can be defined as $GoS_{D_{FyFx}} = |E_{FDy} \cup E_{FDx}|/|E_{Fy} \cup E_{Fx}|$. As a result, the GoS of a program P can be defined as $GoS_P = \sum GoS_{D_{FyFx}}/|SD|, \forall D_{FyFx}$.

The value of this metric considers the number of program elements involved in the implementation of a dependency between two features. For the example illustrated in Section 4.2.1, there are eight different program elements involved in the scope of the implementation of the feature dependency of F2 on F1: `a_m3()`, `a_m5()`, `b_m2()`, `c_m1()`, `d_m2()`, `d_m6()`, `e_m1()` and `e_m4()`. Moreover, both features are composed by twenty-two different program elements (See Figure 4.3). Thus, the $GoS_{D_{F2F1}}$ value to the feature dependency illustrated in Section 4.2.1 is the number of program elements involved in the implementation of the feature dependency, divided by the total number of program elements of both features. This relation is equal to 0.36 (36%). This means that 36% of the program elements implementing both features are involved in the implementation of a feature dependency. In other words, developers tend to inspect 36% of the code of both features to comprehend dependencies and revisit code of potentially affected program elements when propagating a change. It is worth to notice that feature code may be scattered through many different modules, thus making this inspection challenging.

Local Scope (*LoS*) metric. Given a program P, and a module $M \in SM$, the LoS of D_{FyFx} over M can be defined as $LoS_{D_{FyFx},M} = |E_{FDy} \cup E_{FDx}|/|E_M|$, such as $E_{FDy} \subseteq E_M$ and $E_{FDx} \subseteq E_M$. As a result, the LoS of a program P can be defined as $LoS_P = \sum LoS_{D_{FyFx},M}/|SM|, \forall D_{FyFx}$.

The value of this metric considers the number of program elements involved in a feature dependency implementation within a specific module. Considering the example illustrated in Section 4.2.1, `Class_E` has two elements out of four involved in the implementation of the feature dependency of F2 on F1. Thus, the $LoS_{D_{F2F1},Class\_E}$ value to the feature dependency illustrated in Section 4.2.1 for `Class_E` is the number of program elements involved in the implementation of the feature dependency within `Class_E` over the total number of program elements of `Class_E`. This relation is equal to 0.5. This means that 50% of the program elements of `Class_E` are involved in the feature dependency implementation of F2 on F1. With this information, developers can, for instance, be aware of modules that are can be heavily affected by a change propagation as well as focus on specific modules to comprehend most of the feature dependency implementation.

Feature Dependency Connectivity (*FDC*) metric. Given a program

P, the FDC of D_{F2F1} can be defined as $FDC_{D_{FyFx}} = |D_{FyFx}|/|E_{DFx}|$. As a result, the FDC of a program P can be defined as $FDC_P = \sum FDC_{D_{FyFx}}/|SD|, \forall D_{FyFx}$.

The value of this metric considers the number of links from one feature to another to implement a dependency. In other words, this metric considers the total number of 4-tuple in the set D_{FyFx} comprising the implementation of the whole feature dependency. Considering the example illustrated in Section 4.2.1, the set D_{F2F1} has six 4-tuples, which are the six references of a program element of the dependent feature to a program element of a dependee feature. Thus, the $FDC_{D_{F2F1}}$ value to the feature dependency illustrated in Section 4.2.1 is the number of links involved in the implementation of the feature dependency over the total number of program elements of the dependee feature involved in the feature dependency. This relation is equal to 1.5. This means that developers should inspect an average of 1.5 program elements of the dependent feature per program element changed in the dependee feature in order to comprehend the feature dependency implementation as well as revisiting program elements to propagate a change.

Feature Dependency Distance (FDD) metric. Given a program P, the FDD_{FyFx} from a feature Fy to a feature Fx can be defined as the length of the shortest path from feature F_y to feature F_x, considering all possible links in the feature dependency chain in P from feature F_y to feature F_x.

The value of this metric considers the lowest number of links from one feature to another considering a feature dependency chain. To be considered a feature dependency chain, the program element of a dependent feature in the 4-tuple of feature dependency must be a program element of a dependee feature in another feature dependency. Considering the example presented in Section 4.2.3, we should notice that feature F3 depends on feature F2, which depends on feature F1. So, we can say that feature F3 depends on F1 by composing the information of two feature dependencies (D_{F2F1} and D_{F3F2}). The dependency chain is characterised by the link from feature F3 to F2 – (`G.g_m1`,`D.d_m6`,F_3,F_2) –, and by the link from feature F2 to F1 – (`D.d_m6`,`B.b_m2`,F_2,F_1). Thus, the distance from feature F3 to feature F1 is $FDD_{F3F1} = 2$.

4.4 Settings of the Framework Evaluation

This section describes our study configuration in terms of its goal (Section 4.4.1), the target systems used to evaluate the proposed framework (Section 4.4.2), the conventional metrics used in our study (Section 4.4.3), and the procedures followed to run the study (Section 4.4.4).

4.4.1 Research Goal

Classic metrics have been broadly used for a long time in stand-alone programs and found to be effective indicators of changes in the source code [FIGUEIREDO *et al.* 2008, TIZZEI *et al.* 2011]. So, on the one hand, the use of such conventional metrics is expected to be effective in product lines as well. This expectation emerges as the implementation of product lines is often performed by using the same programming techniques already analysed with these metrics. On the other hand, programming techniques often used to implement software product lines and the modularisation mechanisms of these techniques cannot often fully modularise product-line features. In other words, conventional metrics might be missing important details of the structure of the software product line. This problem might occur, for instance, when the modular units of the programming technique and their dependencies do not match the boundaries of product-line features and feature dependencies.

Therefore, it is questionable if conventional metrics are effective indicators of change propagation in the source code. In fact, the use of conventional metrics for product-line evaluation has suffered criticism in recent studies [VAN DER HOEK *et al.* 2003, APEL and BEYER 2011]. Such metrics are criticised for not being sensitive to features, and thus not taking into account structural properties of feature implementation. This problem is amplified by the lack of empirical validation into the effectiveness of metrics in indicating quality attributes in the context of software product lines, which in turn creates uncertainty for developers when deciding on measurement strategies. More specifically, the ability of metrics to indicate change propagation in evolving product line still lacks evaluation.

Concerned with the aforementioned issues, the main goal of this study is to evaluate the effectiveness of two suites of metrics as change propagation indicators in evolving product lines. For the purposes of the evaluation, we compare the effectiveness of conventional metrics (CK metric suite [CHIDAMBER and KEMERER 1994] – see Table 4.3), commonly used in empirical studies of software product lines, against feature dependency metrics based on our measurement framework (Table 4.1).

It is important to notice that our proposed framework and metrics suite are supposed to be independent of implementation approach. Thus, this study uses Java with conditional compilation and AspectJ as representative programming techniques for annotative and compositional approaches, respectively. The reader can refer to Section 2.2 for more details about implementation approaches.

4.4.2 Target Systems

The three medium-sized product lines used in this study include two board games and one embedded mobile application. The first one is actually a family of two board games called GameUP and encompassing the games called Shogi and Checkers [DANTAS and GARCIA 2010, GURGEL *et al.* 2011]. Each game is also a product line itself. The second product line is an embedded mobile software called MobileMedia (MM) [FIGUEIREDO *et al.* 2008], which allows users to manipulate images, videos and music on different mobile devices. In the following, we describe the target software product lines.

GameUP. It is a program family of three board games, which are by themselves product lines. In this work we only analysed Shogi and Checkers game releases. Shogi is a chess games whereas Checkers is an American checker game. Both of them provide features to manage various functionalities for customising the board (e.g. indicating moveable pieces) and the matches between players (e.g. indicating player turns). The evolution scenarios comprise the inclusion of optional and alternative features providing us a variety of feature dependencies, which are fundamental to conduct the investigation of this work.

MobileMedia. It is a software product line that provides support to manage photo, music, and video on mobile devices. The core feature represents basic media management actions such as create/delete media, label media and view/play media. The alternative features are the types of media supported such as photo, music and/or video. The optional features are transfer photo via SMS, count and sort media, copy media and set favourites. The evolution scenarios comprise different types of changes involving the inclusion of mandatory, optional and alternative features, as well as changing of one mandatory feature into two alternatives. Table 4.2 shows some general characteristics of the three target software product lines. For more information about each of them, the reader may refer to the respective work [FIGUEIREDO *et al.* 2008, DANTAS and GARCIA 2010, GURGEL *et al.* 2011].

It is important to mention that we are not using the same product lines used in previous chapters on purpose. To achieve our research goal we needed medium-sized product lines in order to deeply understand the implementation of features and their dependencies. In addition, to satisfy the requirement "Generality" of our framework, we needed the same product lines implemented in different approaches. The purpose was to understand the recurring structural properties of feature dependencies in different implementation approaches. This would not be possible if we conduct a wider and more superficial study using several and larger product lines, which were already contemplated in the previous chapter. We also avoided the use of same software product lines presented in Chapter 3 to avoid bias in our results. For instance, the structural properties of feature dependencies were characterised based on evidence of empirical results of studies presented in Chapter 3. In other words, there were already evidence of the impact of the characterised structural properties on change propagation in the previously analysed product lines. Thus, using the same software product lines could be a significant threat to the results of our study.

4.4.3 Target Metrics

Besides our metrics suite (Section 4.3), we select a representative set of six metrics for analysis. These metrics were proposed by Chidamber and Kemerer for object-oriented programs (CK metrics) and Ceccato and Tonella adapted them in order to make them applicable to aspect-oriented programs [CHIDAMBER and KEMERER 1994, CECCATO and TONELLA 2004]. Therefore, these metrics are useful benchmark for our study, because for each conventional CK metric for object-oriented programming there is an equivalent extension for aspect-oriented programming. This mapping between

Table 4.2: Characteristics of the target systems.

	Shogi	**Checkers**	**MobileMedia**
Application type	Board game	Board game	Mobile data
Prog. technique	Java, AspectJ	Java, AspectJ	Java, AspectJ
Number of releases	4	4	7
Avg. # of features	9	8	25
Avg. KLOC	4	2	10

metrics allows us to broaden our analysis and have results less dependent of programming techniques.

Some of these metrics have already been used in studies focusing on code change analysis of object-oriented and aspect-oriented systems, including software product lines [FIGUEIREDO *et al.* 2008, TIZZEI *et al.* 2011]. To the best of our knowledge, the correlation of such metrics with product-line change propagation has neither been deeply investigated nor compared with feature dependency metrics, despite their use in studies of software product lines.

The conventional metrics used in our study are presented in Table 4.3. The first column shows the name of the conventional metric and the second column presents a brief description of the metric based on the CK metric suite [CHIDAMBER and KEMERER 1994]. It is important to mention that the description of each metric was adapt to use our terminology. As a consequence, the description of the CK metrics used in our study can be applied to program implemented both in Java and AspectJ.

Table 4.3: Conventional CK metrics.

Metric	Definition
WMC	It is the number of operations in a class or aspect. Methods, advice and intertype declarations are counted as an operation.
DIT	It is the length of the longest path from a given class or aspect to the root class or aspect in the hierarchy.
NOC	It is the number of immediate sub-classes and sub-aspects of a module.
CBO	It is the number of classes, aspects or interfaces declaring methods or attributes that are possibly called or accessed by other class or aspect.
RFC	It is the number of methods and advices potentially executed in response to a message received by a given class or aspect.
LCOM	It is the number of pairs of operations working on different class or aspect fields minus pairs of operations working on common attributes.

4.4.4 Evaluation Procedures

The study was divided in four major phases: (1) the implementation of product-line releases with all programming techniques analysed (Java and AspectJ) as well as the alignment of product lines, (2) the assignment of features to elements of the source code as well as the identification of feature dependencies, (3) the quantification of change propagation and the target metrics, and (4) the extraction of the correlation between the target metrics and change propagation.

All phases were conducted by an independent group of three postgraduate students using the implementation of the three target product lines in

Java and AspectJ. Nevertheless, design practices were used, enforced, and reviewed throughout the creation and evolution of all the product-line releases [BUSCHMANN *et al.* 1996]. In all the cases, two experts in the field of product lines also reviewed the implementation. Moreover, the Java and AspectJ implementation and evolution scenarios of the MobileMedia were based on a previous study conducted by Figueiredo et al. [FIGUEIREDO *et al.* 2008]. The scenarios of Checkers and Shogi and their AspectJ implementations were based on a previous work of Dantas and Garcia [DANTAS and GARCIA 2010]. In the following, we detail the four major phases of our study.

Implementation of Aligned Product-line Releases

In the first phase, we implemented the conditional compilation version for four releases of Checkers and Shogi. The AspectJ version for four releases were made available from other studies [DANTAS and GARCIA 2010, GURGEL *et al.* 2011]. All the other product-line implementations were reused from other studies. This reuse procedure was important in order to ensure there was no bias in terms of the research questions being addressed in this present study. The AspectJ and Java with conditional compilation implementation for the seven releases of MobileMedia were available in a study of Figueiredo and colleagues [FIGUEIREDO *et al.* 2008]. All product-line releases were verified according to a number of alignment rules to assure that the implemented functionalities in different programming techniques were the same throughout all versions. Furthermore, design practices to ensure high degree of modularity and reusability were used throughout the creation of all the product-line releases [KUHLEMANN *et al.* 2007]. Some minor refactoring and corrections had to be performed when misalignments were observed. For instance, methods of product lines implemented with conditional compilation had to be split to be aligned with the implementation of advices of the AspectJ version crosscutting the beginning or the end of a method.

Identification of Feature Assignments and Dependencies

In this phase, we mapped features in the source code, so that we could perform the identification of feature dependencies. Each feature was mapped following the principles suggested by Kästner et al. [KÄSTNER *et al.* 2008, KÄSTNER *et al.* 2008c]. Their principles propose a virtual view of features on the source code, depending on a selection of features. The basic idea is to show code fragments with a background colour that represent an associated feature. Each feature has its own colour. Thus, we used background colours to highlight pieces of code that were implementing features. We used one-to-one mapping to ensure that each feature has one colour and every block of code was mapped to one feature. Once all features were mapped, we analysed the source code to identify the dependencies between features. A feature dependency occurs when there is a dependency link between features marked by different colours. We count dependencies by following our definition of feature dependency presented in Chapter 3 (Section 3.1.2) and Section 4.2.1 of this chapter. The tool CIDE [KÄSTNER 2015] supported the identification and counting of feature dependencies based on Definition 3 (Section 4.2.1).

Quantifying Change Propagation and the Target Metrics

To quantify change propagation, we counted every different program element that was changed from one release to the next one. By doing this we identified the change propagation between dependent features. The procedure adopted to identify and quantify change propagation was the same as the one presented in Chapter 3 (Section 3.1.2).

We also collected the measures of the target metrics. The collection of the metrics was divided in two steps. The first step was the collection of conventional metrics. Such metrics were collected aplying the CKJM tool [CKJM 2015] to the product lines implemented with conditional compilation. The AOPMetrics tool [AOPMETRICS 2015] was used to collect the equivalent CK metrics in the product lines implemented in AspectJ. The second step involved the collection of the feature dependency metrics. These metrics were collected using a tool for extracting metrics from the feature composition code [DANTAS *et al.* 2012]. This tool uses information of the structure of the feature and also the dependencies between features to extract the measures.

Correlating Metrics and Change Propagation

To analyse the effectiveness of each conventional and feature dependency metric as indicator of change propagation, we conducted a Spearman's rank correlation test. This test is a non-parametric test that allows us to measure the correlation between our independent variables (each evaluated metric) and our dependent variable (change propagation). For this test, we used a tool named R [R TOOL 2015]. We assumed the commonly used confidence level of 95% (that is, p-value threshold = 0.05). For evaluating the results of the correlation tests, we adopted the Hopkins criteria to judge the goodness of a correlation coefficient [HOPKINS 2015]: less than 0.1 means trivial correlation, 0.1–0.3 means minor correlation, 0.3–0.5 means moderate correlation, 0.5–0.7 means high correlation, 0.7–0.9 means very high correlation, and 0.9–1 means almost perfect correlation.

4.5 Data Analysis and Results

Tables 4.4 and 4.5 show the Spearman's rank correlation coefficients for MobileMedia and GameUP product lines, respectively. It is worth to notice that the metrics are sorted by crescent p-value. For each release of both applications, the target metrics (Section 4.4.3) are applied in a way that we can analyse the correlation of these metrics with change propagation. The correlation analysis is carried out in terms of both programming techniques: conditional compilation and aspect-oriented programming.

Table 4.4: Spearman's rank for MobileMedia.

Metric	Coefficient	p-value
GoS	0.7327273	0.0067
LoS	0.6546208	0.0209
FDC	0.5600000	0.0582
FDD	0.4690416	0.1240
DIT	-0.1909091	0.5523
WMC	-0.0820122	0.8000
LCOM	-0.0535800	0.8686
RFC	0.0142630	0.9649
NOC	0.0109490	0.9731
CBO	-0.0017860	0.9956

Table 4.5: Spearman's rank for GameUP.

Metric	Coefficient	p-value
FDC	0.8445441	0.0005
FDD	0.8095987	0.0014
GoS	0.6536422	0.0211
LoS	0.5354644	0.0728
NOC	-0.4928913	0.1035
CBO	-0.3986046	0.1993
DIT	-0.3982357	0.1998
WMC	-0.3668573	0.2408
LCOM	-0.3633298	0.2457
RFC	-0.3139452	0.3203

4.5.1 Feature Dependency Metrics Outperformed Conventional Metrics

Looking at the data, we can draw an interesting observation: the suite of feature dependency metrics better indicates change propagation than the conventional metrics suite. Tables 4.4 and 4.5 reveal that the feature dependency metrics presented a high correlation in both MobileMedia and GameUP according to the Hopkins criteria [HOPKINS 2015]. The only exception refers to the FDD metric for MobileMedia which presents a moderate correlation with change propagation. Despite this observation, the FDD metric presented a much superior correlation than DIT, the first ranked conventional coupling metric in MobileMedia, which was considered as having a minor correlation to change propagation. The observations herein lead us to the following conclusion: metrics that consider feature dependency structural properties substantially improve the ability of indicating change propagation.

4.5.2 Adapting Conventional Metrics are not Enough

Based on the conclusion presented in Section 4.5.1, the reader may wonder whether only adapting conventional metrics to use feature as modular unit may be enough to have good indicators for change propagation. In other words, if one considers features as the modular unit in the measurements instead of programming techniques units (classes and aspects), is it enough to improve the effectiveness of adapted conventional metrics? To answer this question, it is interesting to analyse the product lines implemented in AspectJ. Taking MobileMedia as a representative product-line, we can observe that it evolves by means of the addition of features with crosscutting behaviour. Crosscutting features are features that may affect many modules of the product line [CONEJERO and HERNÁNDEZ 2008]. This means that the modularisation in aspect-oriented programming is very close to the modular abstraction of a feature in this case. Thus, it is expected that conventional metrics would indicate more precisely the change propagation in software product lines. Analysing the data for each programming techniques, it is possible to observe that: the differences between correlations of feature dependency metrics and conventional coupling metrics are smaller when compared to conditional compilation, as expected. However, even with a modularisation matching the feature boundaries in aspect-oriented programming, the feature dependency metrics remains as better indicators of change propagation than conventional metrics. A further discussion about the comparison between programming techniques is presented in Section 4.5.5

This observation suggests an important insight: only adapting conventional metrics to use feature as modular unit, such as in some studies where changes were not the focus [VAN DER HOEK *et al.* 2003, APEL and BEYER 2011], might not be enough to have good change propagation indicators. In fact, Geipel and Schweitzer argue that is recommended to take advantage of two points in the context of code change: change characteristics and dependency structure [GEIPEL and SCHWEITZER 2012]. The feature dependency metrics suite tries to use both properties during the measurement. For instance, the metric FDC measures the number of program elements likely to be changed. Therefore, taking into account both change characteristics and dependency structures might be an explanation for the superiority of our metrics suite even when the modular unit of the programming technique is similar to the abstraction of a feature.

4.5.3 Combining Feature Dependency Metrics

Focusing on the data presented in Tables 4.4 and 4.5, we can notice an interesting behaviour regarding the feature dependency properties (Section 4.2.2). The correlation of the scope measures was higher than the connectivity measures in MobileMedia. However, we observe an opposite behaviour in GameUP: the connectivity-based metric (FDC metric) was a better indicator of change propagation than the scope-based metrics (GoS and LoS metrics).

This situation can be mainly justified because of the nature of the different evolution scenarios observed in these software product lines. The evolution scenarios of the MobileMedia comprise additions and changes of features. These addition and modifications often encompassed broad modifications across several modules realising the features. As a consequence, broad changes happened in the source code of other features, which are within the scope of the dependencies of the "original" feature. By "original" feature, we mean the feature that was the original target of the evolution scenario. Therefore, the scope property of feature dependency was determinant to capture the change propagation in MobileMedia evolution scenarios. On the other hand, the evolution scenarios of the GameUP comprise integration of features from other product lines. These integration scenarios require additions of source code to realise a new feature dependency or a change of existing feature dependencies. Therefore, these scenarios increase the degree of connectivity of existing features.

Based on the aforementioned observations, we can conclude that our direct properties of feature dependency are complementary indicators. They serve to capture various situations of change propagation as stimulated by different evolution scenarios. Therefore, we confirm that scope and connectivity are useful, complementary properties to characterise changes across feature dependencies. However, the analysis of each property in isolation may not be enough to help developers to propagate changes. The scope property characterises the extension of the feature dependency across the program, whilst the connectivity property characterises the strength of this dependency.

Thus, a feature dependency may present a tight connectivity with a low scope, and another feature dependency may have a high scope and loose connectivity. This is an important observation because the scope-connectivity ratio may be considered as a good indicator of change propagation. Therefore, such behaviour is an indication that the combination of both properties would create a metric that could be high correlated with change propagation.

4.5.4 Conventional Metrics vs. Product-line Changes

Tables 4.4 and 4.5 presented the results of the metrics for all releases of the product lines analysed. However, it is important to take a more careful look at features individually instead of the whole product line in order to analyse if the our results still hold in a more fine-grained analysis. So, taking into consideration each feature, we checked the result of each metric and compare it with the number of propagated changes in the feature. By doing this, we could confirm, as expected, the superiority of the feature dependency metrics for indicating change propagation in software product lines. For instance, analysing the conditional compilation version of MobileMedia, the feature CONTROLLER presents the highest GoS, which according to our conclusion, presents a high correlation with change propagation. In fact, confirming the metric indication, the feature CONTROLLER was the most changed feature along the product-line evolution. Unlike feature dependency metrics, conventional metrics were not good indicators of change propagation in product lines. The feature with the highest DIT (best indicator of change propagation among conventional metrics) in MobileMedia does not have the highest number of changes among features of the product line. This observation reinforces that conventional metrics are not effective indicators of change propagation in product-line features.

This finding unexpectedly holds even if one considers the modifications of the modular units of programming techniques (i.e., classes and aspects). For instance, analysing the conditional compilation version of MobileMedia, the class with highest CBO (widely used indicator of change propagation) does not indicate the class with the highest number of changes. This problem happen because of the expected misalignment between features and modular units of programming techniques. Due to this misalignment, change propagation is often not localised within a modular unit. Therefore, we can state that, even if we want to use conventional metrics (i.e. metrics based on modular units) to indicate changes in modular units (classes or aspects) of a product line implementation, conventional metrics are not good indicators.

4.5.5 Feature Dependency Properties and Product-line Implementation Approaches

Feature dependencies are realised in different ways depending on the programming technique used to implement them. Therefore, it is important to compare the impact of structural properties of feature dependencies in different programming techniques by using a common ground – i.e. a measurement framework agnostic of implementation approaches. Figures 4.5 and 4.6 present a representative case derived from Shogi game, where we can observe the variation of both GoS and change propagation. We chose GoS as a representative case of a structural property since this metric presented the highest correlation with change propagation in our study.

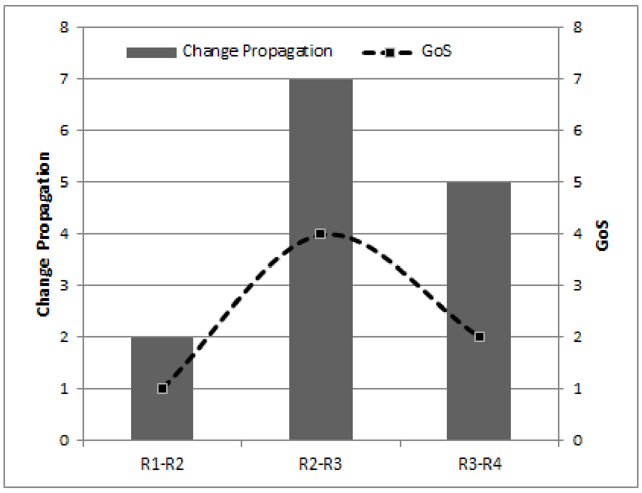


Figure 4.5: Trends of GoS and change propagation in annotative approaches.

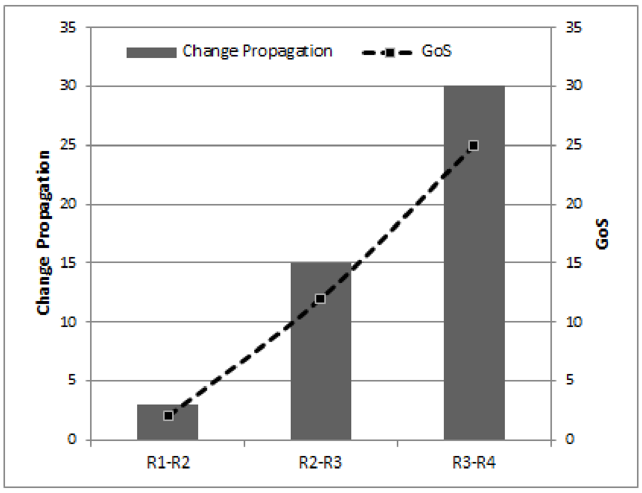


Figure 4.6: Trends of GoS and change propagation in compositional approaches.

As illustrated in the figure, the variation of GoS follows the variation of change propagation regardless of the programming technique used (Figures 4.5 and 4.6). By analysing the numbers, it is possible to state that GoS operates as an indicator of change propagation in evolving software product lines. It is important to mention that all metrics presented the same trend when compared with change propagation. This relationship between GoS and change propagation as well as other structural properties is particularly important to show that our measurement framework allows a fair comparison between implementation approaches. In this way, it is possible even to compare the implementation approaches to identify appropriate mechanisms to better implement feature dependencies based on structural properties. Moreover, our framework can be useful to early adopters who wish to use different implementation techniques when moving towards the evolution of product lines. As the existing relationship between structural properties of feature dependency and change propagation takes places regardless of programming technique, it is essential to understand how particular mechanisms of specific programming techniques deal with the implementation of feature dependency by analysing such structural properties of feature dependency.

It should not be left unmentioned that pointing out the best implementation approach for implementing feature dependencies is out of the scope of this thesis. One of the most important requirements of our measurement framework is related to the “Generality” regarding implementation approaches. However, the reader may refer to our published work [CAFEO *et al.* 2012] for a comparison between implementation approaches regarding feature dependency implementation and their impact on change propagation.

4.6 Threats to Validity

This section discusses the threats to validity that may affect the study presented in this chapter.

Conclusion validity. In this study, potential threats are related to the identification of features and their relationships. To reduce the influence of this threat, we identified features based on the feature models, which were validated in previous studies [FIGUEIREDO *et al.* 2008, DANTAS and GARCIA 2010, GURGEL *et al.* 2011]. In addition, the three postgraduate students that participated of the study investigation have good knowledge on feature-oriented domain analysis. Moreover, they performed careful revisions to check the alignment of each feature represented in the feature model with the product-line code.

Internal validity. The threats to internal validity reside on alignment rules used to identify feature dependencies in all programming techniques. To reduce this threat, we performed a detailed analysis of the product-line code in order to reduce the inconsistencies on the identification process. All product-line releases were verified according to a number of alignment rules to assure that the implemented functionalities in different programming techniques were the same throughout all versions. Furthermore, design practices to ensure high degree of modularity and reusability were used throughout the creation of all the product-line releases [KUHLEMANN *et al.* 2007].

Construct validity. Potential threats rely on the procedures for quantifying changes and feature dependencies in the software product lines (Section 4.4.4). They can be directly associated with the developer's style as this quantification are code-based. In order to ameliorate this issue, the quantification of feature dependency in each programming technique was widely discussed among experienced Java and AspectJ developers.

External validity. Threats associated with external validity concern the degree to which the findings can be generalised to the wider classes of subjects from which the experimental work has drawn a sample [PRADITWONG *et al.* 2011]. To better generalise the results, we selected medium-sized applications from different domains and developed by different programmers (Section 4.4.2). Moreover, they embrace several feature dependencies, which allowed us to investigate a wide range of feature dependency scenarios.

4.7 Related Work

Product-line structural properties. Passos et al. conducted a case study of cross-cutting features in the Linux kernel focusing on driver features [PASSOS *et al.* 2015]. They analysed the evolution of those features by exploring the structural property of scattering, linking findings to the kernel architectural decomposition, and studying how crosscutting features differ from non-cross-cutting ones. One of the main findings was that crosscutting features are harmful to the product-line evolution. However, differently from our work, they do not explore which properties of these features hinder evolution. In addition, they do not study the impact of feature dependencies on product-line evolution. Sobernig and colleagues quantified structural attributes of system decomposition in 28 feature-oriented software product lines [SOBERNIG *et al.* 2014]. They analyse how the alternative decompositions of feature orientation and object orientation compare to each other in terms of their association with observable properties of system structure (i.e., coupling and cohesion). In this study, it is important to notice they adapt conventional structural properties of the source code as well as conventional metrics to compare the feature-oriented and object-oriented decompositions of the same systems. In our thesis, we identify direct properties of feature dependencies to draw our conclusions. Finally, in a recent work, Queiroz et al. quantify three metrics in twenty preprocessor-based systems and define thresholds to the context of feature maintenance, specifically for ripple effects observed in the evolving source code [QUEIROZ *et al.* 2015]. However, once again, this study adapts metrics to the context of preprocessor-based systems and they neglect the impact of dependencies on code maintenance and change propagation.

Metrics for compositional approach languages. Burrows et al. proposed a novel metric to specific dependencies in aspect-oriented software systems that were not captured by conventional metrics [BURROWS *et al.* 2010]. Dantas and colleagues developed a metrics suite intended to quantify the composition code properties and support assessing the impact of composition measures on quality attributes of evolving applications [DANTAS *et al.* 2012]. The metrics proposed in these studies consider the composition code produced with different implementation approaches. However, these metrics focus on the composition of program modules rather than product-line features. Therefore, these metrics do not consider direct properties of features and their dependencies. Thus, the use of such metrics might not be appropriate to product lines. In addition, there is a lack of studies proposing specific metrics for product lines implemented in compositional approaches. Our work advances in the state of art showing (i) that product-line-specific metrics are better indicators of change propagation than the ones based on modular units of programming techniques, and (ii) a suite of metrics also applicable for product lines implemented using compositional approaches.

Metrics for annotative approaches. Liebig et al. proposed a set of metrics to analyse the variability of forty preprocessor-based software product lines [LIEBIG *et al.* 2010]. These metrics are intended to measure product line properties in terms of program comprehension and refactoring. Differently from their work, we focus on change propagation and identify structural properties as well as propose metrics. In addition, they do not propose metrics related to feature dependencies in order to draw their conclusions regarding the variability of the product lines analysed. Apel and Beyer adapted conventional metrics that were originally proposed for procedural and object-oriented systems to a set of cohesion metrics based on clustering layouts. Their goal is to provide a better understanding of the characteristics of cohesion in software product lines [APEL and BEYER 2011]. The study was conducted on forty proprocessor-based software product line. Similar to our work, they propose metrics agnostic to implementation approaches. In addition, the proposed metrics are mainly for drawing conclusions regarding feature cohesion in product lines. Our work proposes a measurement framework for structural properties of feature dependency. So, our metrics are based on characteristics of the source code instead of enabling visual observation of clusters. In addition, our main concern is with feature dependencies, which are one of the main drivers of change propagation (Chapter 3).

Product-line maintenance metrics. van der Hoek et al. developed a class of variability-aware coupling metrics to evaluate the structure defined by product-line architectures [VAN DER HOEK *et al.* 2003]. The suite of metrics is based on the concepts of service dependency and tries to predict volatility of the product line based on those service dependency. However, inferring the volatility based only on service dependency might result in a distorted view of change prediction. As shown in Chapter 3 (Section 3.3.3), considering only the number of dependencies might not be a good indicator of change propagation. In our work, in contrast, we followed the conclusions of Geipel and Schweitzer and we took advantage of change characteristics (e.g. concentration of changes) and dependency structure [GEIPEL and SCHWEITZER 2012]. Moreover, the work of van der Hoek et al. [VAN DER HOEK *et al.* 2003] is based on metrics for software product line architectures. Actually, most of the work proposing metrics for evaluating attributes of maintenance in product lines focuses on the architecture and feature model artefacts. They do not focus on change propagation. At the end, change propagation can only be reliably estimated based on source code analysis. Many of the links realising a feature dependency are only presented in the source code. The implementation approaches also influence the complexity and number of such links. In addition, architecture and feature models are often too abstract and incomplete in practice. Other examples are the work of Bagheri et al. and Torkamani [BAGHERI and GASEVIC 2011, TORKAMANI 2014]. Bagheri et al. propose a suite of metrics for product-line feature models and validate them using valid measurement theoretic principles [BAGHERI and GASEVIC 2011]. Torkamani also proposes a suite of metrics for product-line architecture [TORKAMANI 2014]. His goal is to use the metrics suite to evaluate the reusability power of evolving software product lines.

4.8 Summary

Structural properties of feature dependency are related to cognitive complexity, which, in turn, affect change propagation. Dealing with these properties is a key factor to comprehend feature dependencies and propagate changes when maintaining a product line. In this context, this chapter has presented a measurement framework for quantifying structural properties of feature dependency related to change propagation. The structural properties of our framework were defined using a few concepts to be simple to use (Section 4.2.1). The structural properties of our measurement framework were inspired in findings of empirical studies as shown in Sections 4.2.2 and 4.2.3. These properties are based on direct properties of feature dependencies. In other words, the properties are based on important attributes derived from the structure of a single feature dependency. Our framework was instantiated and evaluated in the context of two implementation approaches of software product line: compositional and annotative approaches (Sections 4.3 and 4.5). This instantiation satisfies the requirement to be generic regarding implementation approaches. The study involved three different product lines (15 releases) implemented in both approaches using Java with conditional compilation (annotative approach) and AspectJ (compositional approach). We proposed metrics based on direct and composite properties of feature dependencies (Sections 4.2.2 and 4.2.3) showing the power of extensibility of our framework. Our analysis revealed that structural properties of feature dependency, as supported by our metrics suite, were consistent indicators of change propagation.

Based on our results, we believe that the use of our measurement framework may help developers to better comprehend feature dependency implementation, thus supporting change propagation. Developers can concentrate their effort on feature dependencies that present higher values of feature dependency metrics based on structural properties (e.g. GoS and FDC). Using conventional metrics (e.g. CK metrics), developers do not have even a metric directly associated with feature dependency, which is one of the main drivers of change propagation (Chapter 3). In addition, our results have shown a lower correlation between conventional metrics and change propagation. Nonetheless, it is important to mention that only the use of the measurement framework does not allow developers to analyse the actual organisation of those dependencies. The measurement framework proposed in this chapter only provides indicators of which dependencies (i) should be carefully analysed during change propagation, and (ii) have structural properties of feature dependency that often exert impact on the cognitive complexity of product-line implementation. Thus, our measurement framework does not support developers to directly obtain and reflect upon each program element realising feature dependencies as well as the organisation of those dependencies. It is necessary a solution that supports developers to truly reason about the organisation of feature dependencies, mainly the ones presenting structural properties highly correlated with change propagation (Chapter 5).

5 Segregating Feature Interfaces to Support Change Propagation

Understanding the intricate relationship that exist between features can be an arduous task. Frequently, this problem is exacerbated because the implementation of a single feature dependency may be scattered over the source code of a product line. With no mechanism for gaining insight into the organisation of feature dependencies, developers are often forced to propagate changes to a dependent feature code without a thorough knowledge. In this way, as product lines tend to change over time, it is inevitable that adopting an ad hoc approach to propagate changes will have a negative effect on the maintenance. A lot of effort may be devoted to comprehend dependencies and revisit code of potentially affected features. Developers may overlook important parts of the code that should be revised or changed. In addition, they may also unnecessarily analyse parts that are not relevant to the feature-maintenance task at hand [RIBEIRO *et al.* 2011].

For example, suppose a product-line example with a feature SCREEN representing a device screen. Feature SCREEN has methods and attributes that are accessed by other features. Examples of program elements participating of feature dependencies are: `usePreSetConfig`, `setBrightness`, `setHPosition`, `setVPosition`, `showFrequency`, and `showResolution`. These program elements are responsible for giving external access to feature SCREEN by providing information (e.g. `showFrequency` and `showResolution`), and by setting configuration parameters for this feature (e.g. `usePreSetConfig`, `setBrightness`, `setHPosition` and `setVPosition`). Put differently, these elements appear in 4-tuples defining feature dependencies (see Section 4.2.1) as elements of a dependee feature – i.e. these elements are more likely to propagate changes to dependent features when modified (see Section 3.3.1). The problem is that those elements may be scattered through the source code. For instance, these elements may be distributed across different files responsible for implementing the functionalities of feature SCREEN. So, it is hard to fully recognise all these program elements in the source code of a product line. As a consequence, the most harmful elements to be modified vanish from developer's eyes during a maintenance. Figure 5.1 shows how the elements of SCREEN accessed by other features are distributed across multiple files. This example illustrates the difficulty of reasoning about the full realisation of each feature dependency.

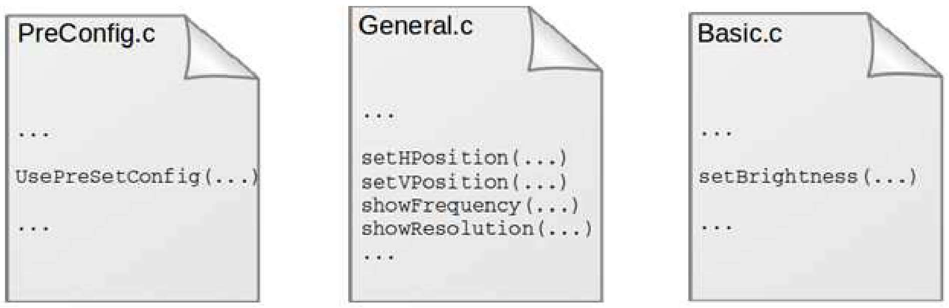


Figure 5.1: Methods of feature SCREEN involved in feature dependencies.

The difficulty in understanding the organisation of the elements that configure feature dependencies is related primarily to the lack of the intended modularity of features [KÄSTNER *et al.* 2011]. This is critical because, since a feature dependency can be faced as the communication between different feature "modules", we can say that understanding of feature dependencies relies on the understanding of the interface of a feature "module". A *feature interface* comprises the program elements in the source code that are responsible for providing external access to other features. In this thesis we focus on provided feature interfaces. A provided feature interface, similarly to a module interface, comprises the program elements belonging to a feature and used by another feature. So, from hereafter, unless otherwise stated, feature interface is used to refer to provided feature interface. Thus, using the basic terminology of our measurement framework presented in Section 4.2.1, the interface of a dependee feature comprises all program elements of this feature present in 4-tuples realising feature dependencies. In other words, in a 4-tuple (e_j, e_i, F_y, F_x), the element e_i would be one interface member of feature F_x. Therefore, the entire interface of F_x is composed by all e_i used in every dependent feature of F_x.

Only identifying and making feature interface members explicit to developers is insufficient for supporting the understanding of feature dependencies organisation. It is known that developers cope with cognitive complexity of large software systems by grouping (clustering) related elements into cohesive groups [MARTIN 1996]. In object-oriented design, for instance, interface members are segregated into more cohesive groups according to their clients. In the same vein, we argue that feature interfaces can be organised into identifiable clusters of program elements based on feature dependencies. These program elements within a cluster of a interface collaborate to achieve part of the overall purpose of feature dependencies. So, if a change must be made in a member of a cluster, we argue that the members within the same cluster are the most relevant to be revised when propagating changes.

In our example (Figure 5.1), let us suppose the feature CONTROLPANEL is responsible for controlling different devices, and INFOPANEL is responsible for showing device information. Figure 5.2 shows four program elements (`usePreSetConf`, `setBrightness`, `setHPosition`, and `setVPosition`) that set values to variables of the feature SCREEN, used by feature CONTROLPANEL – i.e. feature interface members of the feature SCREEN. The other two feature interface members (`showFrequency` and `showResolution`) just return information about SCREEN, and they are used by the feature INFOPANEL. So, for instance, a maintenance task to change the structure of SCREEN information is likely to involve only the members `showFrequency` and `showResolution`. Moreover, a change to one of these members will almost certainly affect the feature INFOPANEL, but it is unlikely to affect feature CONTROLPANEL. So, by understanding feature dependencies' organisation, developers are able to identify the relevant feature interface members to be revised during the maintenance task.

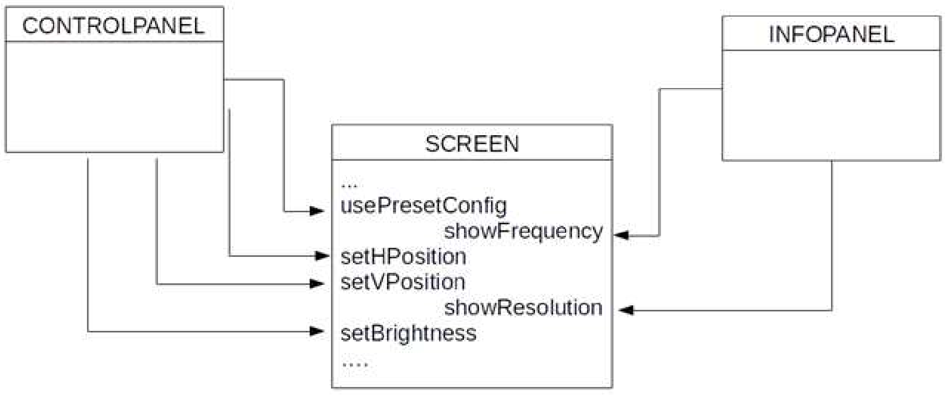


Figure 5.2: Features CONTROLPANEL and INFOPANEL accessing methods of feature SCREEN.

Given the complexity of how features relate to each other, the number of interface members involved in feature dependencies is higher than in the given example. Moreover, it becomes even more complex to understand how groups of interface members work together to participate of one or more dependencies. Finally, it is challenging to understand the effects of a change on other features when interface members involved in dependencies are changed. In this context, organising the usually large and not cohesive feature interface should help developers to understand the intricate relationship that exist between features. This organisation must take into account structural properties of feature dependencies (Chapter 4). For instance, if we consider the feature dependency scope (Section 4.2.2) to segregate members of a feature dependency, it may help developers to understand the implementation of those dependency. As a consequence, the overall number of program elements likely to be revisited during a change propagation will be reduced, thus better supporting developers on reasoning about change propagation.

This chapter, therefore, proposes a technique for automating the creation of an organised view of feature interfaces. This technique aims at answering the third research question of this thesis (RQ3 in Section 1.2), which states: *"How to organise information of feature dependency implementation to support change propagation?"*. Section 5.1 proposes a way of organising feature interface information as a clustering problem. Section 5.2 describes a study to evaluate our proposed organisation using a clustering algorithm. Related work is presented in Section 5.3. Finally, Section 5.4 presents the summary of the chapter.

5.1 Automated Interface Organisation

Creating a good mental model of the structure of a complex system is one of many serious problems of software developers [MANCORIDIS *et al.* 1998]. In our attempt to alleviate the change propagation complexity, we propose a technique for automating feature interface organisation in order to support developers creating a model of the structure of feature dependencies. The goal of our approach is to automatically partition[1] the members of a feature interface into clusters. By doing that, the resultant organisation maximises the relationship between the interface members that are grouped together in the same cluster by considering program elements affected in an eventual change propagation. In other words, we want to externalise structural properties of feature dependencies (Chapter 4) into our feature interfaces' organisation. For instance, we externalise the usually high and scattered scope of a feature dependency by partitioning a feature interface into groups of related members comprising part of the feature dependency scope. These members within the same group are likely to propagate changes the same parts of the source code. The clusters, once discovered, will represent a higher-level abstraction of a feature interface based on feature dependencies' structure. Each cluster contains a set of interface members that cooperate to perform some high-level

[1] We use the term *partition* in the traditional mathematical sense, that is, the decomposition of a set of elements (e.g., nodes of a graph) into mutually disjoint clusters.

function in part of one or more feature dependencies. The following sections detail our approach to organise feature interfaces using clustering.

5.1.1 Interface Organisation as a Clustering Problem

Clustering is the task of grouping a set of elements in such a way that elements in the same group (called a cluster) are more similar (in some sense or another) to each other than to those in other groups (clusters) [EVERITT *et al.* 2009]. Clustering objects into groups is a common task that arises in many applications such as data mining, web analysis, computational biology, machine learning, pattern recognition, and computer vision.

Cluster analysis itself is not a specific algorithm, but the general problem to be solved. It can be achieved by various algorithms that differ significantly in their notion of what constitutes a cluster and how to efficiently find them. In a theoretical setting, the objects are usually viewed as points in either a metric space (typically finite) or as vertices in a graph. Typical objectives include, for instance, minimizing the maximum diameter of a cluster (k-clustering) [HOCHBAUM and SHMOYS 1986], minimizing the average distance between pairs of clustered points (k-clustering sum) [SCHULMAN 2000], and minimizing the average squared distance to an arbitrary centroid point (k-means) [KANUGO *et al.* 2002]. These objectives interpret the distance between points as a measure of their dissimilarity: the larger the distance, the more dissimilar the objects. Another line of clustering algorithms interprets the distance or weights between pairs of points as a measure of their similarity: the larger the weight, the more similar the objects. In this case, the typical objective is to find a k-clustering that minimizes the sum of weights between pairs of objects in different clusters (minimum k-cut) [LEIGHTON and RAO 1999].

In the context of our problem, feature code often contain from dozens to thousand lines of code that are associated into a large number of cooperating features. Buried into these lines of codes, there are many feature interface members being accessed by other features. Fortunately, we often find that these feature interface members are organised into identifiable clusters that collaborate to achieve a higher-level purpose of feature dependencies. As a consequence, these members within the same cluster often propagate changes to the same program elements. However, the structure of these implicit relationships between interface members as well as affected program elements are not obvious from the source code structure. Our research therefore proposes an automatic technique that creates organised feature interfaces. We use a clustering algorithm to partition the members of a feature interface in a way that it derives groups of feature interface members propagating changes to the same parts of the source code. To do so, we propose a model that represent structural properties of feature dependencies by means of relationships between interface members. In the next section we define the underlying model used to represent those relationships between feature interface members and used as input for the clustering algorithm (Section 5.1.3).

5.1.2 Members Relationship Graph

In order to cluster the members of feature interfaces, the straightforward representation of the underlying model to be clustered that fits to our purpose is a graph. So, the first step of our organisation approach is to parse the source code and extract the feature dependencies. Based on these feature dependencies, we retrieve the feature interfaces. After that, we build the so-called *Members Relationship Graph* (MRG). Formally, a members relationship graph $MRG = (M, R)$ consists of two components M and R where: M is the set of members in a specific feature interface, and $R \subseteq M \times M$ is a set of pairs of the form $\langle u, v\rangle$ which represents the members relationships that exist within the set of members of a feature interface. A relationship between two members is characterised when the same program element of a dependent feature refers to both feature interface members. The rationale behind this graph structure is that members referred by the same dependent program element are likely to cooperate to perform part of the feature dependency. As a consequence, they give together a more complete insight into the feature dependency, thus supporting change propagation. In addition, each edge of the graph has a weight. The weight of an edge represents the number of distinct dependent program elements referring the pair of members.

The idea of the MRG is to model the relationship of feature interface members that may affect the same program elements in case of a change propagation. In this way, when feature interface members are used by the same program element of a dependent feature, edges between these members are created. In other words, these relationships between interface members are established based on the feature dependency scope. In addition, each edge has a weight. The weight represents how many times a pair of feature interface members is being used by different program elements of dependent features. In practical terms, the weights mean how strong is the collaboration between the feature interface members towards the implementation of (part of) the purpose of one or more feature dependencies. The weight of an edge is how we represent the feature dependency connectivity by means of the relationship between interface members. So, the idea of applying clustering algorithm to this underlying model is to identify close feature interface members as clusters in a way that the overall number of program elements likely to be revisited during change propagation will be reduced guided by the clusters of the organised feature interface. In this way, the next section presents the algorithm chosen to cluster the underlying model presented in this section.

5.1.3 The Markov Cluster Algorithm

The Markov Cluster (MCL) algorithm [DONGEN 2000] is a cluster algorithm for graphs based on simulation of stochastic flow in graphs. The MCL algorithm is based on the graph clustering paradigm, which postulates that natural groups in graphs have the following property: *"A random walk in a graph that visits a dense cluster will likely not leave the cluster until many of its vertices have been visited"*. Natural groups (clusters) in a graph are characterised by the presence of many edges (or more weighted edges) between the members of that cluster. In particular, this number should be high, relative to node pairs lying in different clusters. In other words, random walks on the graph will infrequently go from one cluster to another.

The MCL algorithm finds cluster structure in graphs by a mathematical bootstrapping procedure. The process deterministically computes (the probabilities of) random walks through the graph, and uses two operators transforming one set of probabilities into another. In this way, the algorithm uses stochastic matrices (also called Markov matrices) which capture the mathematical concept of random walks on a graph. The MCL algorithm simulates random walks within a graph by alternation of two operators called *expansion* and *inflation*. *Expansion* coincides with taking the power of a stochastic matrix using the normal matrix product (i.e. matrix squaring). *Inflation* corresponds with taking the Hadamard power of a matrix (a.k.a. the Schur product) [DAVIS 1962], followed by a scaling step, such that the resulting matrix is stochastic again, i.e. the matrix elements on the column stochastic matrix correspond to probability values. A column stochastic matrix is a non-negative matrix with the property that each of its columns sums to 1. Thus, given such a matrix M and a real number $r > 1$, the column stochastic matrix resulting from inflating each of the columns of M with power coefficient r is written $\Gamma_r(M)$, and Γ_r is called the inflation operator with power coefficient r. The summation of all the entries in column j of M

raised to the power r (sum after taking powers) is defined as $\sum_{r,j}(M)$. Then $\Gamma_r(M)$ is defined in an entrywise manner as:

$$\Gamma_r(M_{ij}) = \frac{M_{ij}^r}{\sum_{r,j}(M)} \tag{1}$$

Each column j of a stochastic matrix M corresponds with node j of the stochastic graph associated with M. Row entry i in column j (i.e. the matrix entry M_{ij}) corresponds with the probability of going from node j to node i. It is observed that for values of $r > 1$, inflation changes the probabilities associated with the collection of random walks departing from one particular node by favouring more probable walks over less probable walks.

Expansion corresponds to compute random walks of high length, which means random walks with many steps. An expansion associates new probabilities with all pairs of nodes, where one node is the point of departure and the other is the destination. Since high length paths are more common within clusters than between different clusters, the probabilities associated with node pairs lying in the same cluster will be relatively large. Inflation will have the effect of boosting the probabilities of intra-cluster walks and will relegate inter-cluster walks. Eventually, iterating expansion and inflation results in the separation of the graph into different segments. When there are no longer any paths between segments, the collection of resulting segments is simply interpreted as a clustering. With this, according to Dongen, the MCL algorithm can be written as [DONGEN 2000]:

```
1.   G is a graph
2.   set $\Gamma$ to some value    # affects cluster granularity
3.   set $M_1$ to be the matrix of random walks on G
4.
5.   while (change) {
6.       $M_2 = M_1 * M_1$            # expansion
7.       $M_1 = \Gamma(M_2)$           # inflation
8.       change = difference($M_1, M_2$)
9.   }
10.
11.  set CLUSTERING as the components of $M_1$
```

Lines 1 and 2 correspond to the inputs of the algorithm which are: an underlying model in a graph format (G), and a value for the parameter inflation (Γ). The input graph (G) is considered as the initial clustering result (M_1), i.e.

the graph where the algorithm will perform random walks and improve it until find the best clusterisation. Between lines 5 and 9 the algorithm alternates two operations: expansion and inflation. First, it expands the clustering matrix resultant of the iteration of both operations and saves it in another matrix (M_2). After that, it inflates matrix M_2 and set the value to the new resultant clustering matrix M_1. These two steps are repeated until the matrix resultant from a previous iteration is equal to the current iteration ($M_1 = M_2$). In this case, matrix M_1 is a stochastic matrix representing the clusters of nodes of the input graph.

The instantiation of the MCL algorithm in our context considers the input of the algorithm as the MRG (see Section 5.1.2) in adjacent matrix format. The other input for the algorithm is the parameter inflation(Γ). The value chosen for the parameter inflation is explained in Section 5.2.3. In a nutshell, the idea of applying the MCL algorithm in our MRG is to remove edges in order to find clusters of interface members that cooperate to each other. After defining the initial parameters, our MRG is assigned to M_1 as the initial result of the clustering algorithm. The algorithm will perform operations over M_1 to find the best clusterisation. Matrix M_1 stores a subgraph of MRG corresponding to the clusters found in each iteration by removing edges from pair of members that should not be in the same cluster. The iterative algorithm is repeated until the matrix resultant from a previous iteration (M_2) is equal to the current iteration (M_1). In other words, the refinement to find the best clustering continues until the result does not change in two iterations. This comparison is executed in line 8 by the function `difference`. In this case, matrix M_1 is the output representing the clusters of members based on the MRG.

The MCL cluster algorithm was chosen to organise feature interfaces because it is a simple algorithm that performs two simple algebraic operations on matrices. There are no high-level procedural instructions for assembling, joining, or splitting of groups. Cluster structure is bootstrapped via a flow process that is inherently affected by any cluster structure present. In addition, the differentials of MCL algorithm compared to other graph-based clustering algorithms are that (i) by varying a single parameter, clusterings on different scales of granularity can be found, and (ii) most importantly, the number of clusters can not and need not be specified in advance.

5.2 Exploratory Study

This section describes our study configuration in terms of its goal (Section 5.2.1), the target system used to evaluate the organisation of feature interfaces (Section 5.2.2), the evaluation procedures used to conduct the study (Section 5.2.3), the results of the study (Section 5.2.4), a discussion about the implications of our results (Section 5.2.5), and the threats to validity (Section 5.2.6).

5.2.1 Research Goal

Developers often do not reason about feature dependencies to make decisions regarding maintenance. The elements of feature interfaces, which configure the feature dependencies, may be spread all over the code, thus making feature interfaces large and difficult to understand. The grouping of these elements also play different roles in the context of feature dependencies, and these groups must be identified and understood in order to alleviate change propagation complexity. So, based on the importance of this problem, we claim that there is a need to organise feature dependency information. More specifically, we believe that externalising structural properties of feature dependencies into organising feature interfaces will help developers to decrease the number of program elements revisited to propagate a change.

Concerned with the aforementioned issues, the main goal of this study is to evaluate the effectiveness of organising feature interfaces. For the purposes of the evaluation, we compare the effectiveness of a feature interface organised using a clustering algorithm (see Section 5.1.3) against an unorganised feature interface. The oracle used to compare both organised and unorganised feature interfaces are the co-changes of program elements extracted from 10 evolutions of a product line (Section 5.2.2). Our claim is that the organisation of feature interfaces proposed in this chapter overcome limitations of unorganised feature interfaces regarding the understanding of the intricate relationship that exist between features. We expect that organised feature interfaces improves the understanding of feature dependencies organisation, thus reducing the overall number of program elements likely to be revisited during change propagation.

5.2.2 Target System

We selected Busybox [BUSYBOX 2015] as a paradigmatic case, representing many other product-line implementations based on conditional compilation [KÄSTNER *et al.* 2012]. Busybox is a real-world resource-efficient product line of UNIX utilities implemented in C language. It runs in a variety of POSIX environments such as Linux, Android, FreeBSD, and others [KÄSTNER *et al.* 2012]. Table 5.1 shows general data about the 10 releases analysed in this study, such as lines of code (KLOC), number of features (# of Features), and number of feature dependencies (# of Dependencies).

Table 5.1: General information about Busybox.

Release	**KLOC**	**# of Features**	**# of Dependencies**
1.13	183	646	630
1.14	188	668	620
1.15	185	696	671
1.16	191	722	702
1.17	196	738	681
1.18	209	759	718
1.18.5	199	759	719
1.19	192	776	761
1.20	194	781	762
1.21	195	766	749
mean	**193**	**731**	**701**

Variability in Busybox uses both variability at the composition level, automated by the build system, and variability at source-code level, encoded with #ifdef directives inside modules [KÄSTNER *et al.* 2012]. In this study, we focus on variability at the source-code level. For instance, taking the release 1.18.5 of Busybox as a representative release, all 522 source-code files contain more than 260k lines of code implementing variability at source-code level. Moreover, of 759 features, 471 (62%) control variability at source-code level with #ifdef directives [KÄSTNER *et al.* 2012].

It is important to mention that we are not using the same product lines used in previous chapters on purpose. To achieve our research goal we needed a product line with a large number of features and feature dependencies. In addition, fine-grained information about (a high number of) commits of the product line analysed herein had to be directly available. In other words, changes in methods associated with features should be available in commits' information. We also avoided the use of same software product lines presented in previous chapters to avoid bias in our results. For instance, the structural properties of feature dependencies were characterised based on previous empirical results in literature and on studies presented in this thesis (see Section 4.1). So, applying the approach presented in this chapter, which is based on structural properties, on previously used product lines could be a significant threat to the results of our study. Finally, we aimed at conducting a longitudinal study of one single product line. The idea was to carefully analyse the feature interfaces proposed by our solution in order to discuss in-depth the implications of our results and draw conclusions. To do so, we needed to understand the semantics of the interface members as well as the semantics of feature dependencies. This would not be possible if we conduct a wider and more superficial study using several product lines, which were already contemplated in previous chapters.

5.2.3 Evaluation Procedures

The study was divided in three major phases: (1) data extraction of feature dependencies, feature interfaces, member relationship graphs (MRG), and program elements co-changes, (2) clustering algorithm application, and (3) clustering evaluation. In the following we detail the three major phases of our study.

Data Extraction

The data used in this study have been retrieved from the Busybox repository [BUSYBOX 2015]. The Busybox repository uses a publicly available data set containing almost 2,000,000 LOC. From the repository, 10 releases of the Busybox containing 6858 source code files were analysed. Each C file has been scanned to mine feature dependencies (see Chapter 2) in the source code using an extension of the TypeChef tool [KENNER *et al.* 2010].

Feature dependency and feature interface extraction. To extract feature dependencies we analysed the control-flow graph of each entire release. Each node of the control-flows graph, representing a program element, was associated with a feature. The set of control flow going between nodes associated to different feature were classified as the realisation of a feature dependency. In other words, adjacent nodes belonging to different features were considered as one link belonging to the set of links of a feature dependency (see Section 4.2.1). Once we extracted all program nodes (i.e. program elements) participating of feature dependencies, we grouped together nodes belonging to the same feature. Nodes containing edges incident to it were classified as members of a feature interface. In other words, program elements from one feature that are being used by another feature are members of a provided feature interface.

Feature interface filtering. Once extracted all feature interfaces, we filter only interfaces with more than one member. After that, we construct the MRG (Section 5.1.2) using an extension of the TypeChef tool and R tool [R TOOL 2015]. The aim of the MRG is to related feature interface members. Weighted dges between nodes represents the members relationships that exist within the set of members of a feature interface. In this case, the topology of the graph becomes informative regarding the complex structure of feature dependencies and the possible extent of change propagation.

Co-change extraction. The last step of data extraction was to mine changes in program elements from the Busybox repository using a tool called Codeface [CODEFACE 2015]. The idea is to identify program elements co-change (i.e. simultaneous change in distinct program elements) within in a specific commit. Similar to several work [CATALDO *et al.* 2008, BIRD *et al.* 2011, JERMAKOVICS *et al.* 2011, JOBLIN *et al.* 2015], we assume that co-changes of distinct program elements in a single commit suggests the execution of a complete maintenance task. In other words, within a commit it is more likely to find program elements that cooperate to perform some high-level function in part of one or more feature dependencies. After that, we filtered out among several co-changes only co-changes involving members of feature interface.

In total we extracted 2286 feature interfaces out of 7311 features stored in the Busybox repository. After filtering feature interfaces with more than one member in the interface, the resultant number of feature interfaces was 650. The total number of feature interface members under analysis is 3154 while the biggest feature interface among 650 features comprises 46 members. The total number of feature dependencies analysed is 7013 while the maximum number in a release is 762. Regarding the co-changes extracted from the Busybox repository, we extracted a total of 3382 changes in program elements comprising 2592 commits for 10 evolution of Busybox. Table 5.2 shows general information of the releases regarding number of program elements changed (# of elements changed) and number of commits (# of commits).

Table 5.2: Change information about Busybox repository.

Release	**# of elements changed**	**# of commits**
1.13	626	384
1.14	535	272
1.15	775	438
1.16	646	422
1.17	485	356
1.18	30	5
1.18.5	48	49
1.19	283	229
1.20	197	166
1.21	257	271
mean	**388**	**259**

Clustering Algorithm Application

We chose to apply the MCL algorithm (Section 5.1.3) to our extracted MRG. However, to calibrate the MCL algorithm regarding its only parameter (i.e. inflation parameter), we used a tool called clm dist [MCL ALGORITHM 2015] implemented by the creator of the MCL algorithm. The tool computes the distance between clusterings by using different metrics such as split/join distance, variance of information measure, and Mirkin metric. This tool can suggest the value of the parameter based on the number of members to be clustered. For further details about the metrics used as well as the clm dist tool the reader may refer to [MCL ALGORITHM 2015]. We followed a procedure to test the MCL algorithm by generating clusters using the inflation values recommended by the author of the algorithm, which are: 1.2, 2, 3, 4, and 6. After generating the clusters for all Busybox releases, we compared the resultant clusterings with clm dist tool using all inflation values aforementioned. The value 2 presented the best results among the values tested in our cases. Hence, we decided to use this value to apply the MCL algorithm on the MRG. The MCL algorithm was applied to the MRG using a R tool library [R TOOL 2015]. The algorithm assigned to each feature interface member a number related to a cluster. In this way, members with the same number are considered members of the same cluster.

Clustering Evaluation

To validate the organisation of feature interfaces resultant of the application of the MCL algorithm, we used the *Jaccard distance.* The Jaccard distance is a statistic used for measuring the dissimilarity of sets. In this way, it measures dissimilarity between finite sample sets, and it is defined by subtracting the size of the intersection by the size of the union and divided it by the size of the union of the sample sets:

$$d_J(A, B) = \frac{|A \cup B| - |A \cap B|}{|A \cup B|} \tag{2}$$

Value of Jaccard distance are bound between 0 and 1. Values closer to 1 means a higher dissimilarity between sets. Values closer to 0 means lower dissimilarity between sets. By using Jaccard distance we are able to use the results and make statements such as "70% of the feature interface members (or interface members of a cluster) might have been unnecessarily analysed on average considering the commits from one release to the next one".

For each set of co-change of program elements we apply the Jaccard distance with every cluster generated by the algorithm. After that, we identify the cluster to be analysed by getting the lowest Jaccard distance. The Jaccard distance of that cluster is used to calculate the average dissimilarity of clusters within the commits from a release to the next one. After calculating the Jaccard distance using the clusters of feature interface members, we calculate the Jaccard distance using the co-change of the program elements and the whole interface (non-clustered interface). For each commit we apply the Jaccard distance and after analysing all commits we calculate the average Jaccard distance of a specific release.

We use a measure of dissimilarity of sets in our study to evaluate our approach. This is justified because our goal is to explore interface members that are likely to be unnecessarily evaluated when changes may be propagated. Those interface members increase the complexity of understanding the organisation of feature dependencies (i.e. cognitve complexity – see Chapter 4), thus increasing the complexity of change propagation. Therefore, the more dissimilar is the set with the commit changes, the worse is a feature interface organisation (clustered or non-clustered).

5.2.4 Results

To address our research goal we need to evaluate our clustering approach. The idea is to explore the solution proposed to organise feature interface members against a non-clustered feature interface. In this way, we apply the Jaccard distance in order to explore the dissimilarity (i.e. the Jaccard distance) between the sets of co-changes of program elements occurred during product-line evolution with both clustered and non-clustered feature interfaces. Table 5.3 shows the results of the Jaccard distance for both types of feature interfaces for each release analysed from Busybox. In addition, the box plot presented in Figure 5.3 shows a visual comparison of how the values of Jaccard distance are distributed for clustered and non-clustered interfaces.

Table 5.3: Comparison between Jaccard distance in Busybox releases.

Release	**Clustered Interface**	**Non-clustered Interface**
1.13	0.2263464	0.9015086
1.14	0.3745138	0.8869625
1.15	0.2767011	0.8712999
1.16	0.197385	0.8545383
1.17	0.2588888	0.9149021
1.18	0.2375843	0.9578755
1.18.5	0.3611111	0.900779
1.19	0.1918869	0.8614765
1.20	0.4325397	0.8878205
1.21	0.1071429	0.8289116
mean	**0.26641**	**0.88660745**

The rationale of the comparison between an organised and an unorganised feature interface is to simulate how developers deal with feature interfaces with no organisation. A high Jaccard distance value indicates the percentage number of interface members that might have been inspected unnecessarily based on commit changes.

Looking at the Jaccard distance (Table 5.3 and Figure 5.3) we can notice that the value for clustered interfaces ranges between 0.1 and 0.4. This behaviour means that, in the worst case, the number of feature members included in a cluster and not changed together with other members within the same cluster is at most 40% of the interface members. In other words, in the worst case of a clustered interface, 40% of the members within a cluster were not related to the commits retrieved from the Busybox repository. Moreover, one can notice the pronounced difference between clustered and non-clustered interface regarding the Jaccard distance by analysing the box plot in Figure 5.3. The concentration of Jaccard distance distribution for non-clustered interface close to 0.8 means that approximately 80% of the feature interface members were not related to the commits (and probably unnecessarily revisited). In addition, despite the higher variation in the Jaccard distance distribution for clustered interfaces, the highest Jaccard distance for clustered interfaces is much lower than the lower Jaccard distance of a non-clustered interface.

Table 5.3 and the box plot in Figure 5.3 already give insights into

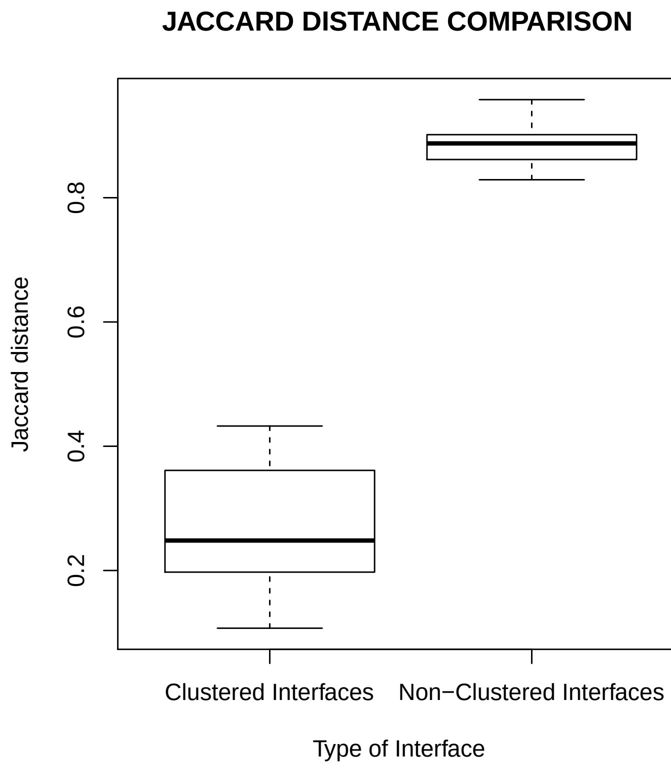


Figure 5.3: Box plots of Jaccard distance values for clustered and non-clustered interfaces.

the capability of the clustered interfaces of reducing the number of interface members likely to be unnecessarily analysed in commits – i.e. the difference between the Jaccard distance distributions. To provide statistical evidence of such capability, we use the same tests used by Romano and colleagues to analyse the significance of the difference between distribution, which in their cases were the difference between a metric before and after refactoring an interface [ROMANO *et al.* 2014]. Therefore, we compute the difference between the distribution of Jaccard distance for clustered and non-clustered interfaces using the paired Mann-Whitney test [MANN and WHITNEY 1947] and the paired Cliff's Delta effect size [GRISSOM and KIM 2005]. It should not be left unmentioned that the Mann-Whitney test and Cliff's Delta effect size were chosen because the Jaccard distances are not normally distributed.

First, we use the Mann-Whitney test to analyse whether there is a significant difference between Jaccard distance of clustered and non-clustered interfaces. The Mann-Whitney test is a nonparametric test that allows two groups, conditions or treatments to be compared without making the assumption that values are normally distributed. Significance differences are indicated by Mann-Whitney *p-values* lower than 0.01. Then, we use the Cliff's

Delta effect size (d) to measure the magnitude of the difference between the Jaccard distance between clustered and non-clustered interfaces. Cliff's Delta is bound between +1 and -1. Values close to +1 means that all selected values from one group are higher than the selected values in the other group, and values closes to -1 when the reverse is true. The value 0 expresses two overlapping distributions. The effect size is considered negligible for $d < 0.147$, small for $0.147 \leq d < 0.33$, medium for $0.33 \leq d < 0.47$, and large for $d \geq 0.47$ [GRISSOM and KIM 2005].

The distribution of Jaccard distance measured on clustered interfaces is statistically different from the non-clustered interfaces' values. The M-W *p-value* is <9E-5. In order words, the M-W *p-value*$\leq$ 0.01 allows us to say the difference between the distributions is statistically significant. The Cliff's Delta measured on both distributions is $d = -1$. This means that the effect size is statistically large and the difference between both distributions is relevant.

5.2.5 Discussion of the Results

This section presents a discussion of the results of the study presented in Section 5.2.4.

The benefits of using clustered interfaces

The results in the previous section (Table 5.3 and Figure 5.3) suggest there is a strong evidence that clustered feature interfaces can support developers to reason about change propagation in product lines. Both M-W p-value and Cliff's Delta effect size allow us to say the clustered interface overcomes non-clustered interfaces in terms of Jaccard distance. We can observe this fact by looking at the Jaccard distance for both types of feature interfaces in all releases (Table 5.3). Clustered interfaces always present a lower Jaccard distance when compared to non-clustered interfaces. We can also observe a concentration of high Jaccard distance for non-clustered interface. In addition, despite the higher standard deviation compared to non-clustered interfaces, there is a concentration of low Jaccard distances for clustered interfaces. So, since we are using the Jaccard distance in our study as a measure of the number of elements likely to be unnecessarily revisited during a change propagation, we can say the number of feature interface members revisited to understand a dependency and propagate changes (i.e. members within a cluster) is lower than in a non-clustered interface.

The pronounced difference when comparing the Jaccard distance between clustered and non-clustered interfaces indicates a significant reduction on members to be considered when developers need to reason about change propagation (≈ 62% of reduction on average) when we observe the average of Jaccard distance for both types of feature interfaces (Table 5.3). For instance, the commit number 2127 from release 1.17 of Busybox involved only one interface member (the method `expand_vars_to_list`) of the feature CONFIG_HUSH. It is worth to mention that the feature CONFIG_HUSH has a relative high scope and tight connectivity (Section 4.2.2) of feature dependencies when compared to other Busybox features. Feature CONFIG_HUSH has 23 interface members in total. So, in a non-clustered interface 22 out of 23 interface members could be revisited by the developer to reason about change propagation caused by a simple change in `expand_vars_to_list` (i.e. ≈ 96%). In the clustered interface of feature CONFIG_HUSH, the cluster containing `expand_vars_to_list` has only two members. Thus, a developer could unnecessarily revisit only one member out of 23 interface members of CONFIG_HUSH (i.e. ≈ 4%) in case of a change on `expand_vars_to_list`.

It is important to mention that some interface members that were not changed must be analysed in order to fully propagate a change. Even though those members were not changed in a commit, the developer would still need to reason about them to understand the impact of the change. Interface members "unnecessarily" analysed in one commit may be important in changes that might happen in other commits. The analysis of a set of cooperative interface members is important (i) to understand the organisation of feature dependencies, and (ii) to certify that there is no other changes to be made. In this context, we argue that our underlying model (MRG) makes explicit structural properties of feature dependencies. This is important because the MRG can capture the cooperative members amongst a set of interface members in part of a feature dependency (see Section 5.1.2). By capturing cooperative interface members and clustering them, we are able to (i) reduce the number of program elements revisited during a maintenance task that involves code change, and (ii) better indicate related interface members that should be inspected or changed; i.e. interface members that may propagate changes to the same parts of the source code.

Therefore, we argue that our approach can alleviate problems related to change propagation such as mis-propagated changes. For instance, in the previous example, a cluster with only two members minimises the change of a misunderstanding about feature dependencies organisation compared to a non-clustered interface with 23 members. The cognitive complexity of understanding the feature dependencies organisation as well as the impact of the changes is reduced. In this way, a misunderstanding of the context of the change propagation is unlikely to happen. As a consequence, we believe that mis-propageted changes are also unlikely to happen. In addition, our approach also reduce the chances of parts of the code being overlooked during a maintenance. By grouping only members that cooperate to each other to realise feature dependencies, the interface indicates which members are likely to be relevant to a be understood to propagate a change. Interface members that cooperate to each other must be understood together because they may propagate changes to the same parts of the source code. So, we also argue that clustered interfaces reduce the overlooking of important parts of the source code when changes must be propagated.

Stability of clustered interfaces

An important issue for the adoption of our approach is the stability of the clustered interfaces. A stable interface is the one that does not have significant changes in its members. A considerable change on our clustered interfaces in each release (i.e. a severe instability in our interfaces) could be impeditive for the adoption of our solution. Developers would need to become familiar with a new organisation of interfaces. However, we argue that, since there is no great changes in the structure of the product line, our clustered interfaces also does not suffer many changes in their structure. This fact can be observed in Table 5.4. We can notice that the number of feature members (# of feature members), number of feature members (# of feature members), and the average number of members in a cluster (Avg. # of members in a cluster) do not vary widely during the evolution of Busybox. For instance, there was a variation of 7 features from the first release (54 features) to the last release (61 features) analysed. In addition, we noticed that the number of feature members also remained stable (variation of 68 members). So, we can consider that the evolution of Busybox did not suffer great changes. In this case, we should expect that the average number of members in a cluster presents a relative stability. In fact, the average number of members in a cluster showed a very low variation along the evolution of Busybox. Based on this, we argue that our approach reflects the behaviour of the evolution of the product line. In other words, it is likely that features and their dependencies stabilise during the evolution. As a consequence, it is also likely that our clustered interfaces also stabilise with time. In this context, we argue that the adoption of our approach in evolving product lines does not demand a lot of effort from developers to become familiar with our clustered interface in each release.

Table 5.4: Information about the number of feature interface members of Busybox.

Release	# of features	# of features members	Expected avg. # of feature members	Avg. # of members in a cluster
1.13	54	263	4.87	3.81
1.14	57	309	5.42	3.96
1.15	59	315	5.34	3.89
1.16	60	314	5.23	3.83
1.17	55	304	5.53	4.11
1.18	58	325	5.60	4.11
1.18.5	59	326	5.53	4.07
1.19	62	334	5.39	4.07
1.20	61	333	5.46	4.06
1.21	61	331	5.43	4.04
mean	**58.6**	**315.4**	**5.38**	**3.9**

The role of structural properties of feature dependencies

Structural properties of feature dependency play a key role in our approach. By analysing the interfaces of the features we could notice that features involved in dependencies with high scope and tight connectivity (see Chapter 4) presented good Jaccard distance values regarding their interfaces. These cases were notable where the difference of the Jaccard distance between clustered and non-clustered interfaces were greater than the distance between the averages (e.g. releases 1.16, 1.19 and 1.21). Feature dependencies with high scope are critical because they are likely to affect many program elements in different parts of the source code when a change must be propagated. In addition, the nature of feature dependencies with high connectivity makes those changes are more likely to be propagated due to the tight connectivity between features. The combination of both properties makes change propagation in product line a complex task. Developers may be not able to fully understand the intricate and complex relationship that exist between features by looking at non-clustered interfaces.

Our approach captures these properties to group interface members that collaborate to realise the feature dependency. For instance, the commit number 2118 from release 1.17 of Busybox involved changes in two interface members of feature CONFIG_HUSH (`done_command` and `new_pipe`). It is worth to cite that the interface members `done_command` and `new_pipe` are two methods implemented in a class with more than 8 KLOC. These two members cooperate to implement part of many dependencies involving feature CONFIG_HUSH. Several dependencies involving CONFIG_HUSH are realised by many program elements scattered in the source code (i.e. evidence of a high scope of a feature dependency). In addition, these two members consistently appear together (i.e. evidence of tight connectivity in this part of the dependency). In our clustered interface, these two members (`done_command` and `new_pipe`) are grouped together in one cluster indicating this cooperation between both. In this way, we argue that our approach is able to capture key properties of feature dependencies that are able to alleviate the change propagation task in software product lines by clustering feature interfaces.

Complexity of feature dependencies

Not all feature dependencies are complex. Despite of the benefits of relying our underlying model on structural properties of feature dependencies, some feature dependencies present a low scope and/or loose connectivity. So, the targeted properties to be represented by the underlying model may not help clustering feature interface members. In these cases, our approach presented the worst results (e.g. releases 1.14, 1.18.5 and 1.20). Features involved in feature dependencies with low scope tend to present coarse-grained clusters of interface members (i.e. many members within a cluster) with no relation. In other words, not all members within a cluster collaborate to perform a high-level function in a feature dependency.

For instance, the commit number 2142 from release 1.17 of Busybox involved only one interface member of feature CONFIG_ASH changed (`setvar`). Feature CONFIG_ASH has 41 interface members. The member `setvar` is within a cluster with 28 more members. In addition, it is important to mention that only the member `setvar` of the entire cluster was changed considering all commits involved in this specific release of Busybox (release 1.17). As a consequence, we can expect an increase in the Jaccard distance value in situations as described above. On the other hand, features participating of feature dependencies with loose connectivity tend to have fine-grained clusters (i.e. few members within a cluster). An example is feature CONFIG_DC from release 1.17 with one member per cluster. The Jaccard distance value for feature interfaces similar to CONFIG_DC interface increase depending on the characteristics of the commits (e.g. number members involved in a commit).

So, to sum up, we can say that an improvement in our underlying model may be needed to improve our results in some specific cases. This can be achieved by representing other important structural properties and/or tuning (or even changing) the clustering algorithm used in our approach. Nevertheless, it is important to mention that cases of feature dependencies with these characteristics (i.e. low scope and/or loose connectivity) have not been often found in our study. This fact can be observed in Table 5.4. One can see that the average number of members in a cluster (Avg. # of members in a cluster) is close to 3.9 while the expected number of a coarse-grained cluster (i.e. a number close to the entire number of interface members) is more than 5 (Expected avg. # of feature members). The difference of more than 25% of members between the averages can be faced as an evidence that the cluster are not coarse-grained.

Nature of changes

Another important factor that may impact in our results are the commits used to validate our approach. Depending on some characteristics of the commit, the Jaccard distance value for some clustered interfaces tends to be high when compared to the majority of the results. For instance, the commit number 2121 involved changes in 20 interface members of 9 different features. One can notice that many interface members were changed in one single commit. These interface members changed in a commit may belong to one or more features. In this specific commit we highlight the unusual number of features being changed in a single commit. The reasons for a coarse-grained commit like that are many. For example, intrinsic characteristics of the maintenance task may demand changes in several interface members. The problem is that the more members are changed in one single commit, the lower is the chance that a cluster comprises all these elements. Developers might have to analyse several interface clusters within a feature to comprise all members that must be revisited. Even worse, when commits involve changes of members from different features, it is impeditive to have a good result since we are using commits as our ideal model of cluster. In the example presented here, we can also notice that the number of members changed in a single commit is higher than the average number of members within a cluster (see Table 5.4). As a consequence, the Jaccard distance value measured in this commit is expected to be high. Therefore, we can say that coarse-grained commits can compromise the effectiveness of our solution.

It should not be left unmentioned that the majority of the commits (including the ones analysed in our study) usually comprise changes in program elements that are somehow related (i.e. "atomic" maintenance task). Table 5.5 shows some information regarding the number of commits (# of commits), number of program elements changed from one release to the next one (# of program elements changed), and average of program elements changed per commit (Avg. # of program elements changed). One can observe that, in general, releases with an average number of program elements per commit greater than the total average number (i.e. 1.84 elements per commit) presented the worst Jaccard distance values. In this case, we can argue that only exceptional cases of commits may compromise our approach. Nevertheless, the results are better than the obtained results for non-clustered interfaces.

Table 5.5: Information about the number of program elements changed per commit.

Release	**# of commits**	**# of program elements changed**	**Avg. # of program elements changed**
1.13	384	626	1.63
1.14	272	535	1.97
1.15	438	775	1.77
1.16	422	646	1.53
1.17	356	485	1.36
1.18	5	30	6.00
1.18.5	49	48	0.98
1.19	229	283	1.24
1.20	166	197	1.19
1.21	271	257	0.95
mean	**259.2**	**388.2**	**1.86**

Independence of both history and amount of changes

An important point to be mentioned is the independence of our approach regarding the history of changes. Despite the influence of the nature of the changes in our results, our approach does not take into account information related to history of changes. Our approach relies only on the MRG to cluster the interface. This is important to mention because our organised feature interfaces can be adopted in early releases of software product lines. In this way, another concern was to select early releases of Busybox to show the effectiveness of our approach. For instance, our approach was applied in an early release of the Busybox (1.13 – 4th release) and the Jaccard distance indicates a low dissimilarity between proposed cluster interfaces and changes in commits (see Table 5.3). Actually, the Jaccard distance of release 1.13 is lower than further releases. In addition, our results for Jaccard distance also achieved good results in later releases of Busybox. This finding means that even though our approach does not take into consideration the history of change (e.g."learning" about frequent changes), our approach can be applied to the whole life cycle of a product line.

Finally, we also noticed an unexpected independence of our results regarding the amount of changes. Since we are considering the commits as our ideal model of clusters, it was expected that releases with few changes could impact on our results. Indeed, release 1.18.5 presented a high Jaccard distance when compared to other releases and with the average Jaccard distance of the analysed releases. The release 1.18.5 of Busybox presented a Jaccard distance of approximately 0.36 (Table 5.3). The number of commits for this release was 49 (see Table 5.5), which is lower than the average number of commits. However, we can notice that releases with several changes presented higher Jaccard distance (e.g. 1.14 and 1.20) than the result associated to release 1.18.5 of Busybox. In addition, we can also notice that release 1.18 had a good Jaccard distance (lower than the average Jaccard distance) even with only 5 commits with this release. These facts reinforce that our approach for organising feature interfaces is independent of the amount of changes to succeed. In fact, we could achieve a good precision (based on the Jaccard distance) in cases with a low number of commits depending on the nature of changes, as discussed before.

5.2.6 Threats to Validity

This section discusses the threats to validity that may affect the study presented in the previous section.

Conclusion validity. It concerns the relationship between the treatment and the outcome. In this study, potential threats come from the statistical tests used to support our conclusions. To mitigate this threat we argue that wherever possible, we used statistical tests observing the characteristics of our data. In particular, we used non-parametric tests which do not make any assumption on the underlying data distribution regarding variances and the types of distributions.

Internal validity. It is the degree to which conclusions can be drawn about the causal effect of independent variables on the dependent variables. In this experiment, potential threats come from the tuning of the MCL algorithm (Section 5.1.3) that may affect the results. Changing the only parameter of the algorithm will affect the granularity of the cluster, thus affecting the Jaccard distance (Section 5.2.3) used to compare the results. We mitigated this threat by calibrating the algorithm using five values suggested by the author of the algorithm (Section 5.2.3).

Construct validity. It concerns the relationship between theory and observation. In our study this threat can be due to the fact that we focus on conditional compilation as the variability mechanism for implementing features in the source code. With conditional compilation, features are often tangled and scattered in the source code. This choice means that a feature may have (i) many members in its interface, and (ii) many of them may not be cohesively related to each other and, therefore, may not be relevant for each change propagation. This situation could be different if we had decided to analyse product lines implemented using compositional approach (e.g. aspect-oriented programming). However, this situation cannot be completely avoided as all product lines analysed are implemented using the mechanism of conditional compilation. However, we argue that conditional compilation is the most widely used mechanism to implement product-line features [KÄSTNER and APEL 2009]. Moreover, Busybox (Section 5.2.2) is an industrial project. So, we believe the results extracted from this product line can be a first step towards the generalisation of the results. In fact, Busybox contains categories of features implemented in a wide range of different ways: from fully-modularized features to highly-scattered and highly-tangled features.

External validity. Threats associated with external validity concerns the degree to which the findings can be generalised to the wider classes of subjects from which the experimental work has drawn a sample [PRADITWONG *et al.* 2011]. In our work, this is a particularly important threat to validity because of the wide range of diverse product lines implemented using conditional compilation. In the experiment reported upon here, this threat to validity is somewhat mitigated by the fact that we selected Busybox (Section 5.2.2) to conduct our study. Busybox can be considered as a paradigmatic case which represents many other product-line implementations based on conditional compilation due to, for example, its size, number of features and number of valid configurations [KÄSTNER *et al.* 2012]. In addition, since we are analysing different releases of the same product-line, there is no risk that the variation due to individual differences of product lines is larger than due to the treatment.

5.3 Related Work

Feature modularity. Feature modularity has been a long-standing goal of feature-oriented software development [APEL *et al.* 2013]. While some researchers view features as modular unit of behavior and composition, others pointed out that, at the source-code level, most implementation mechanisms provide merely syntactic compositions, and thus lack proper interface abstractions and modular reasoning. In this context, Kästner et al. pinpoint two different notions of feature modularity: one based on locality and cohesion, and another based on information hiding and interfaces [KÄSTNER *et al.* 2011]. Modularity means locality and cohesion when a feature is viewed as a unit of composition that has the goal of making itself explicit in design and implementation [APEL *et al.* 2013]. Therefore, everything related to a feature is placed into a separate structure called feature module [APEL *et al.* 2008]. Another view of feature modularity is rooted in the concept of information hiding and interfaces. The idea is to distinguish between an internal and an external part of a feature module. The internal part is hidden. The external part is called interface and controls the communication between different feature modules [KÄSTNER *et al.* 2011]. Most of the work on feature modularity has focused on locality and cohesion of features as a criterion for system decomposition and assembly. Examples of approaches for improving feature modularity include architecture-based product lines (based on frameworks or components) [BASS *et al.* 2003], feature-oriented programming [PREHOFER 1997, BATORY *et al.* 2003], aspectual feature modules [APEL *et al.* 2008], feature cohesion [APEL and BEYER 2011], and superimposition [APEL *et al.* 2013c]. Despite the improvement of feature modularity in those cases, simple solutions like conditional compilation prevail in practice [GACEK and ANASTASOPOULES 2001, ERNST *et al.* 2002, KÄSTNER *et al.* 2008]. Nevertheless, there is a lack of studies driving efforts towards feature interfaces in solutions based on (i) locality and cohesion, and (ii) in widespread adopted solutions such as conditional compilation. Our work propose a way of organising feature interface. This organisation is based on structural properties of feature dependencies agnostic of implementation approach. In this way, our work enhance feature modularity by providing organised feature interfaces to support change propagation in both compositional and annotative approaches.

Finally, there is substantial progress in solving problems that threat modularity in features. The feature-interaction problem is considered a major threat to modularity in that the behavior of one feature may be affected by the presence of another feature [APEL *et al.* 2013d]. So, developers must analyse the consequences of all possible feature interactions to find the undesired ones. In other words, the feature interaction problem also hinders independent feature maintenance. Some studies deals with feature interaction and its problems [APEL and BEYER 2011, BATORY *et al.* 2011, APEL *et al.* 2013d]. Despite the similar focus of these studies to our approach, none of them aims at improving feature interfaces to support product-line maintenance. In addition, our work focuses on feature dependencies that can be inferred via syntactical relationships.

Feature interfaces. Ribeiro et al. proposed a solution for interfaces of product-line features [RIBEIRO *et al.* 2011, RIBEIRO *et al.* 2014]. They defined the concept of emergent interfaces for product lines implemented with conditional compilation. This approach aims at establishing – based on source code – interfaces between features on demand (emergent interfaces), with the goal of preventing developers from breaking other features when performing a maintenance task. Despite the generation of interfaces, this approach generates only interfaces related to specific parts of the source code that are of interest, and thus do not allow having a global view of the system. In other words, emergent interfaces do not address the problem of large/monolithic interface by segregating them. Our work deals with the complexity of the organisation of feature dependencies by segregating feature interfaces. In this way, developers are able to identify a complete part of the interface that is important to understand in order to propagate changes.

There is further work that concentrates on interfaces in approaches used to implement product lines or in variability-checking approaches supported by interfaces. For example, Kästner and colleagues propose a variability-aware module system for product lines [KÄSTNER *et al.* 2012]. This approach infers interfaces for modules focusing on type checking of product-line configurations. Kiczales and Mezini propose aspect-aware interfaces, computing an aspect's dependencies on a system's join points and displaying these dependencies as annotations on the explicit interfaces of advised code [KICZALES and MEZINI 2005]. Li and colleagues propose a new methodology to verify cross-cutting features as open systems by using a model of semantic interfaces that supports automated, compositional, and feature-oriented model checking [LI *et al.* 2002, LI *et al.* 2002b]. Blundell et al. propose a parametrised interface for verifying product-lines [BLUNDELL *et al.* 2004]. Such interface lifts properties of individual features to composed features to verify temporal properties of such features. However, none of these studies deal with interfaces specifically for supporting maintenance tasks. Our approach aims at supporting developers to propagate changes during maintenance by providing organised feature interfaces.

Automatic Modularisation. The problem of automatic modularisation (also referred to as automatic clustering) has been extensively researched lately. The idea is to help developers creating a good mental model of system's organisation. The field was established by the seminal work of Mancoridis et al. [MANCORIDIS *et al.* 1998]. In this work, the authors use hill climbing algorithm as the primary search technique for automated software module clustering. Several other meta-heuristic search techniques have been applied, including genetic algorithms [HARMAN *et al.* 2002, MITCHELL and MANCORIDIS 2002, MAHDAVI *et al.* 2003]. However, all these studies focus on the most common application of clustering in automatic modularisation: software module clustering. Differently, our work applies the idea of clustering in order to enhance the modularity of features by organising feature interfaces. Regarding the automatic modularisation of interfaces, after the introduction of the idea of interface segregation principle [MARTIN 1996] some studies have proposed ways of organising feature interfaces. For instance, in a recent work Romano et al. propose a way of refactoring fat interfaces (i.e. interfaces whose clients invoke different subsets of their members) of classes of object-oriented programs using genetic algorithms [ROMANO *et al.* 2014]. However, our work does not deal with refactoring ill-defined interfaces. In addition, identifying the interfaces of classes in an object-oriented program is much easier than in conditional-compilation-based programs since all the classes are already modularised. Moreover, we are focused on organising interfaces of product-line features instead of classes' interfaces. Finally, our study was the first to systematically investigate and compare the co-relation of clustered interfaces and non-clustered interfaces with respect to change propagation.

5.4 Summary

Creating a good mental model of the structure of a complex system is one of many serious problems of software developers [MANCORIDIS *et al.* 1998]. With no mechanism for gaining insight into feature dependencies organisation, developers are often forced to propagate changes to a dependent feature code without a thorough knowledge of the feature dependency structure. In our attempt to alleviate the change propagation complexity, this chapter has presented an approach to organise feature dependency information. In this chapter we focus specifically on feature interfaces. The goal of the proposed approach is to automatically partition the members of a feature interface into clusters so that the resultant organisation maximises the relationship between the interface members that are grouped together in the same cluster by considering program elements affected in an eventual change propagation. Each cluster, therefore, contains a set of interface members that cooperate to perform some high-level function in part of one or more feature dependencies.

The interface organisation was faced as a clustering problem. An underlying model was proposed and a clustering algorithm was applied to identify the cluster of members. We evaluate our approach in 10 releases of a product line comprising approximately 2,000,000 LOC. Our analysis revealed that organised feature interfaces may support developers when propagating changes. With an organised interface, there is a decreased effort to comprehend dependencies and revisit code of potentially affected features is decreased. The pronounced difference was of approximately 62% considering the average of Jaccard distances. In addition, we noticed that the good result of our approach relies on the underlying model (MRG) based on structural properties of feature dependencies. The MRG could capture the cooperation between interface members based on structural properties of feature dependencies. The analysis of a set of cooperative interface members is important (i) to understand the organisation of feature dependencies, and (ii) to certify that there is no other changes to be made. Therefore, we argue that our approach can alleviate problems related to change propagation such as mis-propagated changes. The cognitive complexity of understanding the feature dependencies organisation as well as the impact of the changes is reduced when few interface members are analysed. In addition, by grouping only members that cooperate to each other to realise feature dependencies, the interface indicates which members are likely to be relevant to a be understood to propagate a change. Interface members that cooperate to each other must be understood together because they may propagate changes to the same parts of the source code. So, we also argue that clustered interfaces reduce the overlooking of important parts of the source code when changes must be propagated. Finally, we observed that our clustered interfaces stabilise with time. In this context, we argue that the adoption of our approach in evolving product lines does not demand a lot of effort from developers to become familiar with our clustered interface after every change in a release.

6
Final Considerations

Feature dependencies play an important role in the maintenance of software product lines. As developers modify the source code associated with a feature, such as methods or attributes, they must ensure that other features are consistently updated with the new changes. However, appropriate change propagation is far from being trivial as features are often not modularised in the source code [APEL *et al.* 2013]. Given a change in a certain feature, it is challenging to reveal which other features should also change. As a single feature is often not localised in a single module in the implementation [KÄSTNER *et al.* 2011], developers cannot simply resort to module dependencies to infer to which features should be revisited or changed. Studies about change propagation in stand-alone programs, although valuable, do not address the particularities of software product lines. In other words, as a single feature is often not localised in a single module, approaches for propagating changes in stand-alone programs are not applicable to software product lines. So, understanding the relationship between feature dependencies and change propagation becomes, therefore, a central and non-trivial aspect of software product-line maintenance.

In this context, our intention in this thesis was to explore the relation between these two important phenomena – i.e., feature dependencies and change propagation (Chapter 3). First, we presented a study with an in-depth analysis about the relation of feature dependency and change propagation. We explored the probabilities of changes be propagated in the presence of feature dependency. We also analysed the extent of the propagation through paths of feature dependencies. A concentration of change propagation in few dependencies were also identified. This step (Chapter 3) is particularly important because (i) it provides empirical support to argue that feature dependency is one of the main drivers of change propagation in software product lines, and (ii) it reveals behaviours of propagated changes in the presence of feature dependencies, such as the extent of the propagation and concentration of changes in some dependencies. By using these findings, we could gather initial information about structural properties of feature dependencies that might be related to change propagation. For instance, feature dependencies concentrating more changes were the candidates to be analysed in order to identify those structural properties

The second step was to provide developers with support for characterising and quantifying these structural properties (Chapter 4). There is a gap in literature related to the characterisation of those structural properties of feature dependencies. A number of conventional metrics have been used in studies related to product-line maintenance. However, most of these metrics are defined based only on the properties of modules realising the features. This trend means that existing metrics might be missing important details about the structure of the software product line. In turn, the understanding of structural properties of feature dependencies is blurred because conventional metrics are not able to quantify those structural properties. As a consequence, developers and researchers cannot understand and study the impact of structural properties of feature dependency on change propagation. In order to support the understanding of this impact, a formalism on structural properties of feature dependencies was provided. In other words, we systematically defined a measurement framework based on structural properties of feature dependencies. Based on this formalism, metrics were created to quantify structural properties and we empirically compared them to conventional metrics regarding their ability to indicate change propagation.

Finally, it is recognised the importance of fully understanding the intricate relationship that exist between features in order to properly propagate changes. Feature dependency metrics only provide partial support to this understanding. Such full understanding of feature dependencies relies on the understanding of the interface of a feature. However, feature interface as they are supported in the state-of-the-art are insufficient. They do not support developers in reasoning about the organization of feature interfaces. So, the third step of our research was to propose an automated technique that creates an organised view of a feature interface. The goal of our approach was to automatically partition the members of a feature interface into clusters. We aim at deriving a resulting organisation that cohesively groups the members of an interface. The idea is to maximise the relationship between members into separate interfaces regarding their likelihood of being affected together by the same change. Therefore, we faced the interface organisation as a clustering problem. The idea of using clusters is to bring together similar elements in the same group. In our context, similarity means members that are likely to affect the same parts of the source code in case of change propagation. We evaluated the effectiveness of an organised feature interface against an unorganised feature interface regarding ripple changes across feature boundaries in the source code.

6.1 Review of the Contributions

In this thesis we discussed the need of exploring the relationship between feature dependencies and change propagation. Therefore, we claim that feature dependencies are one of the main drivers for change propagation. In this way, quantitative assessments of product-line change propagation should be rooted not only at quantifying module properties. In fact, change propagation must be guided by the understanding of the structural properties of feature dependencies that exert impact on change propagation itself. Based on a set of structural properties (Chapter 4), which were found to be harmful for change propagation, this thesis defines a measurement framework to support the quantification of structural properties of feature dependencies. Our framework satisfies five important requirements (Section 4.1) and it also allows to define both direct and composed properties of feature dependencies. In addition, we evaluate the metrics suite comparing it with a conventional metrics suite. Finally, we propose and evaluate a method for organising feature interfaces in order to support change propagation. Our method was inspired in the findings of our previous studies, reported in Chapters 4 and 5. Based on the studies presented in this thesis, we advanced the body of knowledge about the relation between feature dependencies and change propagation, which was a subject hitherto little explored in the literature. Hence, we reinforce here the contributions of this work, preliminary presented in Chapter 1:

- **Empirical findings on the relation between feature dependencies and change propagation (Chapter 3).** Comprehending dependencies and revisiting code of potentially affected features make change propagation a time-consuming task during product-line maintenance [RIBEIRO *et al.* 2011]. Therefore, understanding whether and how feature dependencies lead to change propagation is important to reduce the maintenance complexity. We argued that feature dependencies could be an important driver to change propagation in software product lines. To confirm our statement, we carried out an empirical study to evaluate to what extent feature dependencies are related to change propagation. The study undertaken in this phase (Chapter 3) was carried out using information of feature dependencies and changes of twenty-one evolution (or twenty-six releases) of five product lines. The results confirmed the close relation between feature dependency and change propagation. In other words, feature dependencies are important drivers of change propagation. We have also observed a number of new interesting outcomes. For instance, according to our results, there is a high probability of a feature dependency propagating changes (average of ≈51%). Moreover, we identified the way feature dependencies are implemented (structural properties of feature dependency) may impact on change propagation (Section 3.3.1). Developers could benefit from this finding by differentiating feature dependencies using their structural properties. This is an interesting finding since we also revealed an inequality in the distribution of change propagation through the feature dependencies of a product line. This counterintuitive result indicates that a general feature dependency minimisation might not reduce change propagation effort (Section 3.3.2). In addition, characterising structural properties of feature dependency is essential to distinguish the impact of feature dependencies on change propagation. The reasoning about these properties help developers on the task of propagating changes by driving efforts to "harmful" feature dependencies. Finally, our analysis also reflected upon change propagation through the path of dependencies. This analysis pointed to a linear relation between the depth of dependency and change propagation, which indicates a more extensive change propagation in features than the exponential relation between distance and classes in stand-alone programs [GEIPEL and SCHWEITZER 2012]. This information might be valuable to tune and/or complement existing approaches, such as combinatorial interaction testing (Section 3.3.3).

– **Feature dependency properties and metrics (Chapter 4).** The more complex is the structure of a feature dependency, the more influential it might be in change propagation. Therefore, it becomes primordial the need of indicators for the complex structure to those complex structure of feature dependencies. The indicators should externalise relevant characteristics of feature dependencies aiming at supporting developers to overcome the difficulties caused by the complex context of propagating changes in the presence of feature dependencies. Several authors [CANT *et al.* 1994, BRIAND *et al.* 2001] state that quantitative models are related to the concept of cognitive complexity. In this way, quantitative models can externalise relevant software structural properties (e.g. module coupling [CHIDAMBER and KEMERER 1994]), thus helping developers to overcome the cognitive complexity of code maintenance. In this context, we presented a measurement framework for quantifying structural properties of feature dependency. The purpose of the measurement framework is to make possible to quantify structural properties of feature dependency that impact on change propagation regardless of product-line implementation approach employed. Our framework was instantiated and evaluated in the context of two implementation approaches of software product line: compositional and annotative approaches. The study involved three different product lines (15 releases) implemented in both approaches using Java with conditional compilation (annotative approach) and AspectJ (compositional approach). Our goal was to evaluate the effectiveness of two suites of metrics as change propagation indicators in evolving product lines. For the purposes of the evaluation, we compare the effectiveness of conventional metrics commonly used in empirical studies of software product lines (CK metrics [CHIDAMBER and KEMERER 1994]) against feature dependency metrics based on our proposed framework. Our analysis revealed that structural properties of feature dependency, as supported by our metrics suite, were consistent indicators of change propagation. As main contributions, we identified recurring structural properties of feature dependency in different product-line implementation approaches, explored the relation between such properties with change propagation, defined metrics based on these properties, and identified the metrics that correlated the most with change propagation. Developers can concentrate efforts, for instance, on feature dependencies that present higher values of metrics based on structural properties (e.g. GoS and FDC). We also studied how such properties and metrics could be used to compare structural properties of feature dependencies across different programming techniques. Therefore, we believe that the use of our measurement framework may help developers to better comprehend feature dependency implementation, thus supporting change propagation in a wide range of contexts.

– **A method for segregating feature interfaces (Chapter 5).** Creating a good mental model of the structure of a complex system is one of the key challenges for software developers. Therefore, understanding the intricate relationship that exist between features can be an arduous task. With no mechanism for gaining insight into organisation of feature dependencies, developers are often forced to propagate changes to the code of dependent features without a thorough knowledge. A lot of effort may be devoted to comprehend dependencies and revisit code of potentially affected features. In this way, we proposed a technique for automating the creation of an organised view of feature interfaces. Our goal was to support the understanding of the relationship between feature dependencies by organising feature interface members. We face the problem of organising feature interface as a clustering problem. We believe that externalising structural properties of feature dependencies into organised feature interfaces will help developers to decrease the number of program elements revisited to propagate a change. By doing that, the resulting organisation maximised the relationship between the interface members that were grouped together in the same cluster within an interface by considering program elements affected in an eventual change propagation. The clusters represent a higher-level abstraction of a feature interface based on feature dependencies' structure. Each cluster contains a set of interface members that cooperate to perform some high-level function in part of one or more feature dependencies. To evaluate our claim, we conducted a study on our organised feature interfaces. The study undertaken in this phase were carried out using 10 releases of a software product line comprising almost 2,000,000 LOC in 6858 source code files. This study aimed at analysing how close the clusters proposed by the algorithm are from the real simultaneous changes of feature interface members during product-line evolution. Our analysis revealed that organised feature interfaces may support developers when propagating changes. With an organised interface, there is a decreased effort to comprehend dependencies and revisit code of potentially affected features is decreased. The pronounced difference was of approximately 62% considering the average of Jaccard distances. In addition, we noticed that the good result of our approach relies on the underlying model (MRG) based on structural properties of feature dependencies. The MRG could capture the cooperation between interface members based on structural properties of feature dependencies. By capturing cooperative interface members and clustering them within interfaces, developers are able to (i) reduce the number of program elements revisited during a maintenance task that involves code change, and (ii) better indicate related interface members that should be inspected or changed; i.e. interface members that may propagate changes to the same parts of the source code.

6.2 Future Work

In spite of various contributions of this thesis described in Section 6.1, there are many other directions for future work, some of which are described in the following paragraphs.

Additional Empirical Studies. The evaluation studies provided evidence of the close correlation between feature dependencies and change propagation. However, it is necessary to undertake additional studies. The behaviour of change propagation in presence of feature dependencies in other types of software maintenance tasks should be assessed. For instance, other changes beyond the scope of perfective maintenance should be evaluated in order to check whether our findings about the relationship between feature dependencies and change propagation holds in other contexts. In addition, it is important to assess the measurement framework in the context of other emerging programming techniques. For instance, it would be interesting to instantiate our framework in prominent programming techniques, such as Delta-oriented programming [SCHAEFER *et al.* 2010]. It is also of great importance to use and evaluate the usefulness of our framework in the context of other software attributes, such as error-proneness. For instance, we hypothesize that our feature dependency metrics may be also useful to indicate or predict bugs in software product lines. Due to the complexity of reasoning about feature dependencies, their structural properties might help to reveal potential sources of bugs. In fact, we have conducted a study towards this direction in partnership with the Federal University of Bahia. However, in this study we identified the need of characterising structural properties of feature dependencies in order to relate them to bugs and to the difficulty of localising and correcting those bugs. Finally, it is required a controlled experiment to assess the use of organised feature interfaces by developers when performing maintenance tasks in product lines. This experiment could allow us to evaluate, for instance, the effort of propagating changes with (and without) our organised feature interfaces.

Measurement Framework Refinement. These additional studies would enable us to reveal any extensions needed in our measurement framework. These further extensions are particularly important to validate the generality of our framework. In addition, these extensions can also contribute to the identification of additional structural properties that are harmful to change propagation as well as other important maintenance tasks.

Alternative Approaches for Organising Feature Interface. We faced the organisation of feature interfaces as a clustering problem. In this way, undertaking additional studies using different algorithms for clustering the interface members would enable us to reveal the best algorithm for clustering feature interfaces. In additional, regarding the MCL algorithm used in our proposed organisation, testing different parameters for inflation may be important. Since different values for this parameter impacts on clustering granularity, tuning the inflation may enable us to reach better organisation of feature interfaces depending on the number of changes per commit.

Tool Support. At least two improvements are needed to make the extraction of structural properties of feature dependency metrics usable in practice: (i) creating a graphical interface for such extraction, and (ii) incorporating strategies for identifying and/or mapping features and feature dependencies in the source code. In addition, another important tool support would be the representation of organised feature interfaces in a visual manner. Studies should be conducted to define the best way of visually representing the feature interfaces as well as the relationship between features. This tool could be integrated into a popular IDE, such as Eclipse, in a way that developers become aware of the structural properties of feature dependencies, feature interfaces and the intricate relationships between features while developing their software projects. Candidate tools for incorporating the aforementioned extensions are, for instance, the tools CIDE [KÄSTNER 2015] or TypeChef [KENNER *et al.* 2010].

Bibliography

[AOPMETRICS 2015] AOPMetrics. **Aopmetrics tool**. `http://aopmetrics.tigris.org/`, 2015. [Online; accessed 28-May-2015]. 4.4.4

[ALDRICH 2005] J. Aldrich. **Open modules: modular reasoning about advice**. In *Proceedings of the 19th European Conference on Object-Oriented Programming, ECOOP'05*, pages 144–168, Berlin, Heidelberg, 2005. Springer-Verlag. 2.4.3

[ALVES *et al.* 2005] V. Alves, P. Matos, L. Cole, P. Borba and G. Ramalho. **Extracting and evolving mobile games product lines**. In *9th International Conference on Software Product Lines, SPLC'05*, pages 70–81. Springer, 2005. 3.1.3, 3.4

[APEL *et al.* 2008] S. Apel, C. Lengauer, B. Möller and C. Kästner. **An algebra for features and feature composition**. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology, AMAST 2008*, pages 36–50, Berlin, Heidelberg, 2008. Springer-Verlag. 2.1.1, 5.3

[APEL and KÄSTNER 2009] S. Apel and C. Kästner. **An overview of feature-oriented software development**. *Journal of Object Technology*, 8:49–84, July/August 2009. Refereed Column. 2.1.1

[APEL and BEYER 2011] S. Apel and D. Beyer. **Feature cohesion in software product lines: an exploratory study**. In *33rd International Conference on Software Engineering, ICSE'11*, pages 421–430. ACM, 2011. 1, 2.4.1, 2.4.2, 3.4, 3.5, 4, 4.1, 4.2.2, 4.4.1, 4.5.2, 4.7, 5.3

[APEL *et al.* 2013] S. Apel, A. v. Rhein, P. Wendler, A. Grösslinger and D. Beyer. **Strategies for product-line verification: case studies and experiments**. In *International Conference on Software Engineering, ICSE'13*, pages 482–491, Piscataway, NJ, USA, 2013. IEEE Press. 1

[APEL *et al.* 2013] S. Apel, D. Batory, C. Kästner and G. Saake. **Feature-oriented software product lines: Concepts and implementation**. Springer Publishing Company, Incorporated, 2013. 1, 2.1.2, 5.3, 6

[APEL *et al.* 2013c] S. Apel, C. Kästner and C. Lengauer. **Language-independent and automated software composition: The featurehouse experience**. *IEEE Transactions on Software Engineering*, 39(1):63–79, 2013. 5.3

[APEL *et al.* 2013d] S. Apel, A. von Rhein, T. Thüm and C. Kästner. **Feature-interaction detection based on feature-based specifications**. *Computer Networks*, 57(12):2399 – 2409, 2013. 5.3

[ARNOLD and BOHNER 1993] R. Arnold and S. Bohner. **Impact analysis-towards a framework for comparison**. In *Conference on Software Maintenance*, pages 292–301, Sep 1993. 1

[ARNOLD 1996] R. S. Arnold. **Software change impact analysis**. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996. 1

[BAGHERI and GASEVIC 2011] E. Bagheri and D. Gasevic. **Assessing the maintainability of software product line feature models using structural metrics**. *Software Quality Journal*, 19(3):579–612, 2011. 2.4.2, 2.4.2, 4, 4.2.2, 4.7

[BANIASSAD and MURPHY 1998] E. L. A. Baniassad and G. C. Murphy. **Conceptual module querying for software reengineering**. In *Proceedings of the 20th International Conference on Software Engineering, ICSE'98*, pages 64–73, Washington, DC, USA, 1998. IEEE Computer Society. 2.4.3

[BARTOLOMEI *et al.* 2006] T. T. Bartolomei, A. Garcia, C. Sant'Anna and E. Figueiredo. **Towards a unified coupling framework for measuring aspect-oriented programs**. In *3rd International Workshop on Software Quality Assurance, SOQUA '06*, pages 46–53, New York, NY, USA, 2006. ACM. 2.4.2, 4.2.1

[BASS *et al.* 2003] L. Bass, P. Clements and R. Kazman. **Software architecture in practice**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 edition, 2003. 2.2.4, 5.3

[BATORY *et al.* 2003] D. Batory, J. N. Sarvela and A. Rauschmayer. **Scaling step-wise refinement**. In *Proceedings of the 25th International Conference on Software Engineering, ICSE '03*, pages 187–197, Washington, DC, USA, 2003. IEEE Computer Society. 2.2.1, 2.2.4, 5.3

[BATORY *et al.* 2011] D. Batory, P. Höfner and J. Kim. **Feature interactions, products, and composition**. In *Proceedings of the 10th*

ACM International Conference on Generative Programming and Component Engineering, *GPCE '11*, pages 13–22, New York, NY, USA, 2011. ACM. 5.3

[BELLMAN 1958] R. Bellman. **On a Routing Problem**. *Quarterly of Applied Mathematics*, 16:87–90, 1958. 1.1

[BERTSEKAS and TSITSIKLIS 2008] D. P. Bertsekas and J. N. Tsitsiklis. **Introduction to Probability, 2nd Edition**. Athena Scientific, 2nd edition, 2008. 3.1.4

[BEUCHE *et al.* 2004] D. Beuche, H. Papajewski and W. Schröder-Preikschat. **Variability management with feature models**. *Science of Computer Programming*, 53(3):333–352, December 2004. 2.2.2

[BIRD *et al.* 2011] C. Bird, N. Nagappan, B. Murphy, H. Gall and P. Devanbu. **Don't touch my code!: Examining the effects of ownership on software quality**. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, *ESEC/FSE '11*, pages 4–14, New York, NY, USA, 2011. ACM. 3, 5.2.3

[BLUNDELL *et al.* 2004] C. Blundell, K. Fisler, S. Krishnamurthi and P. V. Hentenryck. **Parameterized interfaces for open system verification of product lines**. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, *ASE '04*, pages 258–267. IEEE Computer Society, 2004. 2.4.3, 5.3

[BRIAND *et al.* 1999] L. Briand, J. Wust and H. Lounis. **Using coupling measurement for impact analysis in object-oriented systems**. In *IEEE International Conference on Software Maintenance*, pages 475–482, 1999. 4, 4, 4.2.1

[BRIAND *et al.* 1999b] L. C. Briand, J. Wüst, S. V. Ikonomovski and H. Lounis. **Investigating quality factors in object-oriented designs: An industrial case study**. In *21st International Conference on Software Engineering*, *ICSE '99*, pages 345–354, New York, NY, USA, 1999. ACM. 4, 4.2.2

[BRIAND *et al.* 2001] L. Briand, C. Bunse and J. Daly. **A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs**. *IEEE Transactions on Software Engineering*, 27(6):513–530, Jun 2001. 2.4.2, 4, 6.1

[BROOKS 1995] F. P. Brooks, Jr. **The mythical man-month (anniversary ed.)**. Addison-Wesley Longman Publishing Co., Inc., 1995. 1

[BURROWS *et al.* 2010] R. Burrows, F. C. Ferrari, A. Garcia and F. Taïani. **An empirical evaluation of coupling metrics on aspect-oriented programs**. In *Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics*, *WETSoM '10*, pages 53–58, New York, NY, USA, 2010. ACM. 2.4.2, 4.7

[BUSCHMANN *et al.* 1996] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad and M. Stal. **Pattern-oriented software architecture: A system of patterns**. John Wiley & Sons, Inc., New York, NY, USA, 1996. 4.4.4

[CKJM 2015] CKJM. **Ckjm tool**. `http://www.spinellis.gr/sw/ckjm/`, 2015. [Online; accessed 28-May-2015]. 4.4.4

[CAFEO *et al.* 2012] B. Cafeo, F. Dantas, A. Gurgel, E. Guimaraes, E. Cirilo, A. Garcia and C. J. P. Lucena. **Analysing the impact of feature dependency implementation on product line stability: An exploratory study**. In *26th Brazilian Symposium on Software Engineering*, *SBES'12*, pages 141–150, 2012. 3.5, 4, 4.1, 4.5.5

[CAFEO *et al.* 2013] B. Cafeo, F. Dantas, E. Cirilo and A. Garcia. **Towards indicators of instabilities in software product lines: An empirical evaluation of metrics**. In *2013 ICSE Workshop on Emerging Trends in Software Metrics*, *WETSoM'13*, pages 69–75. IEEE Press, 2013. 3.5, 4, 4.1

[CALDER *et al.* 2003] M. Calder, M. Kolberg, E. H. Magill and S. Reiff-Marganiec. **Feature interaction: A critical review and considered forecast**. *Comput. Netw.*, 41(1):115–141, Jan. 2003. 3.3.1

[CANT *et al.* 1994] S. Cant, B. Henderson-Sellers and D. Jeffery. **Application of cognitive complexity metrics to object-oriented programs**. *Journal of Object Oriented Programming*, 7:52–52, 1994. 2.4.2, 4, 6.1

[CATALDO *et al.* 2008] M. Cataldo, J. D. Herbsleb and K. M. Carley. **Socio-technical congruence: A framework for assessing the impact of technical and work dependencies on software development productivity**. In *Proceedings of the Second ACM-IEEE International*

Symposium on Empirical Software Engineering and Measurement, *ESEM '08*, pages 2–11, New York, NY, USA, 2008. ACM. 3, 5.2.3

[CATALDO and HERBSLEB 2011] M. Cataldo and J. Herbsleb. **Factors leading to integration failures in global feature-oriented development: an empirical analysis**. In *33rd International Conference on Software Engineering*, pages 161–170. ACM, 2011. 1, 2.4.1, 3, 3.5

[CECCATO and TONELLA 2004] M. Ceccato and P. Tonella. **Measuring the effects of software aspectization**. In *1st Workshop on Aspect Reverse Engineering*, 2004. 2.4.2, 4.4.3

[CHENG *et al.* 2006] S. Chang, H. La and S. Kim. **Key issues and metrics for evaluating product line architectures**. In *International Conference on Software Engineering and Knowledge Engineering*, *SEKE'06*, pages 212–219, 2006. 2.4.2

[CHIDAMBER and KEMERER 1994] S. Chidamber and C. Kemerer. **A metrics suite for object oriented design**. *IEEE Transactions on Software Engineering,*, 20(6):476–493, 1994. 2.4.2, 4, 4.4.1, 4.4.3, 6.1

[CIRILO *et al.* 2008] E. Cirilo, U. Kulesza and C. J. Lucena. **A product derivation tool based on model-driven techniques and annotations**. *Journal of Universal Computer Science*, 14(8):1344–1367, April 2008. 3.1.2, 3.1.2

[CLEMENTS and NORTHROP 2001] P. C. Clements and L. Northrop. **Software product lines: Practices and patterns**. *SEI Series in Software Engineering*. Addison-Wesley, August 2001. 2.1, 2.2.2

[CLEMENTS 2002] P. Clements. **Being proactive pays off/eliminating the adoption barrier**. *IEEE Software*, 19(4):28, 30–, 2002. 2.2.4

[CONEJERO and HERNÁNDEZ 2008] J. M. Conejero and J. Hernández. **Analysis of crosscutting features in software product lines**. In *13th International Workshop on Early Aspects*, *EA '08*, pages 3–10, New York, NY, USA, 2008. ACM. 4.5.2

[CORMEN *et al.* 2009] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. **Introduction to algorithms, third edition**. The MIT Press, 3rd edition, 2009. 3.1.2

[CZARNECKI and EISENECKER 2000] K. Czarnecki and U. W. Eisenecker. **Generative programming: Methods, tools, and applications**. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000. 2.2.4

[DAM *et al.* 2006] K. Dam, M. Winikoff and L. Padgham. **An agent-oriented approach to change propagation in software evolution**. In *Australian Software Engineering Conference*, pages 10 pp.–, 2006. 1

[DANTAS and GARCIA 2010] F. Dantas and A. Garcia. **Software reuse versus stability: Evaluating advanced programming techniques**. In *2010 Brazilian Symposium on Software Engineering*, pages 40–49, 2010. 4, 4.4.2, 4.4.4, 4.4.4, 4.6

[DANTAS *et al.* 2012] F. Dantas, A. Garcia and J. Whittle. **On the role of composition code properties on evolving programs**. In *ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, ESEM '12*, pages 291–300, New York, NY, USA, 2012. ACM. 2.4.2, 4.2.1, 4.2.2, 4.2.2, 4.4.4, 4.7

[DIJKSTRA 1959] E. Dijkstra. **A note on two problems in connexion with graphs**. *Numerische Mathematik*, 1:269–271, 1959. 1.1

[DONGEN 2000] S. Dongen. **A cluster algorithm for graphs**. Technical report, National Research Institute for Mathematics and Computer Science, Amsterdam, The Netherlands, The Netherlands, 2000. 5.1.3, 5.1.3

[ERNST *et al.* 2002] M. D. Ernst, G. J. Badros and D. Notkin. **An empirical analysis of c preprocessor use**. *IEEE Transactions on Software Engineering*, 28(12):1146–1170, 2002. 2.2.2, 5.3

[ETZKORN and DELUGACH 2000] L. Etzkorn and H. Delugach. **Towards a semantic metrics suite for object-oriented design**. In *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00), TOOLS '00*, pages 71–, Washington, DC, USA, 2000. IEEE Computer Society. 2.4.2

[EVERITT *et al.* 2009] B. S. Everitt, S. Landau and M. Leese. **Cluster analysis**. Wiley Publishing, 4th edition, 2009. 5.1.1

[FERBER *et al.* 2002] S. Ferber, J. Haag and J. Savolainen. **Feature interaction and dependencies: Modeling features for reengineering a legacy product line**. In *Software Product Lines*, volume 2379 of *LNCS*, pages 235–256. Springer Berlin Heidelberg, 2002. 1, 1.1.1, 2.4.1, 3, 3.5

[FIGUEIREDO *et al.* 2008] E. Figueiredo, N. Cacho, C. SantAnna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan,

F. Castor Filho and F. Dantas. **Evolving software product lines with aspects: an empirical study on design stability**. In *30th International Conference on Software Engineering*, *ICSE'08*, pages 261–270. ACM, 2008. 2.4.1, 3.1.1, 3.1.3, 3.4, 4, 4.4.1, 4.4.2, 4.4.3, 4.4.4, 4.4.4, 4.6

[FIGUEIREDO *et al.* 2008b] E. Figueiredo, C. Sant'Anna, A. Garcia, T. Bartolomei, W. Cazzola and A. Marchetto. **On the maintainability of aspect-oriented software: A concern-oriented measurement framework**. In *12th European Conference on Software Maintenance and Reengineering*, pages 183–192, 2008. 4

[FIGUEIREDO *et al.* 2009] E. Figueiredo, B. Silva, C. Sant'Anna, A. Garcia, J. Whittle and D. Nunes. **Crosscutting patterns and design stability: An exploratory analysis**. In *IEEE 17th International Conference on Program Comprehension*, pages 138–147, 2009. 4.2.1

[FISLER and KRISHNAMURTHI 2008] K. Fisler and S. Krishnamurthi. **Decomposing verification around end-user features**. In B. Meyer and J. Woodcock, editors, *Verified Software: Theories, Tools, Experiments*, volume 4171 of *Lecture Notes in Computer Science*, pages 74–81. Springer Berlin Heidelberg, 2008. 2.1.1

[FLOYD 1962] R. W. Floyd. **Algorithm 97: Shortest path**. *Communications of the ACM*, 5:345–, 1962. 3.1.2

[GACEK and ANASTASOPOULES 2001] C. Gacek and M. Anastasopoules. **Implementing product line variabilities**. In *Proceedings of the 2001 Symposium on Software Reusability: Putting Software Reuse in Context*, *SSR '01*, pages 109–117, New York, NY, USA, 2001. ACM. 2.2.2, 5.3

[GARVIN and COHEN 2011] B. Garvin and M. Cohen. **Feature interaction faults revisited: An exploratory study**. In *22nd International Symposium on Software Reliability Engineering*, *ISSRE'11*, pages 90–99. IEEE Computer Society, 2011. 1, 2.4.1, 3.5

[GEIPEL and SCHWEITZER 2012] M. M. Geipel and F. Schweitzer. **The link between dependency and cochange: Empirical evidence**. *IEEE Transactions on Software Engineering*, 38(6):1432–1444, August 2012. 2.4.2, 3, 3.1.2, 3.1.2, 3.2.3, 3.3.2, 3.3.3, 3.5, 3.6, 4, 4.5.2, 4.7, 6.1

[GINI 1921] C. Gini. **Measurement of inequality of incomes**. *The Economic Journal*, 31:124–126, 1921. 3.1.4

[GODFREY and GERMAN 2008] M. Godfrey and D. German. **The past, present, and future of software evolution**. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, *FoSM 2008*, pages 129–138, 2008. 3, 3.1.3

[GRISSOM and KIM 2005] R. J. Grissom and J. J. Kim. **Effect sizes for research: A broad practical approach**. Lawrence Erlbaum, 2 edition, 2005. 5.2.4

[GRISWOLD *et al.* 2006] W. G. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai and H. Rajan. **Modular software design with crosscutting interfaces**. *IEEE Software*, 23(1):51–60, January 2006. 2.4.3

[GURGEL *et al.* 2011] A. Gurgel, F. Dantas and A. Garcia. **On-demand integration of product lines: A study of reuse and stability**. In *2nd International Workshop on Product Line Approaches in Software Engineering*, *PLEASE '11*, pages 35–39, New York, NY, USA, 2011. ACM. 4.4.2, 4.4.4, 4.6

[HARMAN *et al.* 2002] M. Harman, R. M. Hierons and M. Proctor. **A new representation and crossover operator for search-based optimization of software modularization**. In *Proceedings of the Genetic and Evolutionary Computation Conference*, *GECCO '02*, pages 1351–1358, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. 5.3

[HASSAN and HOLT 2004] A. E. Hassan and R. C. Holt. **Predicting change propagation in software systems**. In *20th IEEE International Conference on Software Maintenance*, *ICSM'04*, pages 284–293. IEEE Computer Society, 2004. 1, 1.1, 1.1.1, 2.3, 3, 3.5

[HAY and ATLEE 2000] J. D. Hay and J. M. Atlee. **Composing features and resolving interactions**. *SIGSOFT Software Engineering Notes*, 25(6):110–119, November 2000. 2.1.1

[HENDERSON-SELLERS 1995] B. Henderson-Sellers. **Object-oriented metrics: Measures of complexity**. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. 2.4.2

[HER *et al.* 2007] J. S. Her, J. H. Kim, S. H. Oh, S. Y. Rhew and S. D. Kim. **A framework for evaluating reusability of core asset in product line engineering**. *Information and Software Technology*, 49(7):740–760, 2007. 2.4.2

[HOCHBAUM and SHMOYS 1986] D. S. Hochbaum and D. B. Shmoys. **A unified approach to approximation algorithms for bottleneck problems**. *Journal of the ACM*, 33(3):533–550, 1986. 5.1.1

[HOPKINS 2015] W. G. Hopkins. **A new view of statistics. sports science**. `http://www.sportsci.org/resource/stats`, 2015. [Online; accessed 28-May-2015]. 4.4.4, 4.5.1

[JACKSON and ZAVE 1998] M. Jackson and P. Zave. **Distributed feature composition: a virtual architecture for telecommunications services**. *IEEE Transactions on Software Engineering*, 24(10):831–847, 1998. 2.1.1

[JARZABEK *et al.* 2003] S. Jarzabek, P. Bassett, H. Zhang and W. Zhang. **Xvcl: Xml-based variant configuration language**. In *Proceedings of the 25th International Conference on Software Engineering, ICSE'03*, pages 810–811, 2003. 2.2.2

[JERMAKOVICS *et al.* 2011] A. Jermakovics, A. Sillitti and G. Succi. **Mining and visualizing developer networks from version control systems**. In *4th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE '11*, pages 24–31, New York, NY, USA, 2011. ACM. 3, 5.2.3

[JOBLIN *et al.* 2015] M. Joblin, W. Mauerer, S. Apel, J. Siegmund and D. Riehle. **From developer networks to verified communities: A fine-grained approach**. In *Proceedings of the IEEE/ACM International Conference on Software Engineering, ICSE 15*, pages 563–573. IEEE Computer Society, 2015. 3, 5.2.3

[KAFURA and REDDY 1987] D. Kafura and G. Reddy. **The use of software complexity metrics in software maintenance**. *IEEE Transactions on Software Engineering*, 13(3):335–343, 1987. 3.1.2

[KANUGO *et al.* 2002] T. Kanungo, D. Mount, N. Netanyahu, C. Piatko, R. Silverman and A. Wu. **An efficient k-means clustering algorithm: analysis and implementation**. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7):881–892, Jul 2002. 5.1.1

[KÄSTNER *et al.* 2008] C. Kästner, S. Trujillo and S. Apel. **Visualizing software product line variabilities in source code**. In *SPLC Workshop on Visualization in Software Product Line Engineering, ViSPLE'08*, pages 303–312, 2008. 1, 2.2, 2.2.1, 2.2.2, 2.4.3, 4.4.4, 5.3

[KÄSTNER and APEL 2008] C. Kästner and S. Apel. **Integrating compositional and annotative approaches for product line engineering**. In *Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering*, pages 35–40, 2008. 2.2.4

[KÄSTNER *et al.* 2008c] C. Kästner, S. Apel and M. Kuhlemann. **Granularity in software product lines**. In *Proceedings of the 30th International Conference on Software Engineering, ICSE '08*, pages 311–320, New York, NY, USA, 2008. ACM. 2.2.4, 4.4.4

[KÄSTNER and APEL 2009] C. Kästner and S. Apel. **Virtual separation of concerns – a second chance for preprocessors**. *Journal of Object Technology*, 8(6):59–78, September 2009. 1, 1.1, 2.2.2, 2.2.4, 2.4.3, 3.1.3, 3.4, 5.2.6

[KÄSTNER *et al.* 2011] C. Kästner, S. Apel and K. Ostermann. **The road to feature modularity?** In *15th International Software Product Line Conference, SPLC'11*, pages 51–58, New York, NY, USA, 2011. ACM. 1, 2.1.1, 2.1.2, 2.2.1, 2.4.3, 2.4.3, 5, 5.3, 6

[KÄSTNER *et al.* 2012] C. Kästner, K. Ostermann and S. Erdweg. **A variability-aware module system**. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '12*, pages 773–792, New York, NY, USA, 2012. ACM. 2.4.3, 5.2.2, 5.2.2, 5.2.6, 5.3

[KELLY 2006] D. Kelly. **A study of design characteristics in evolving software using stability as a criterion**. *IEEE Transactions on Software Engineering*, 32(5):315–329, 2006.

[KENNER *et al.* 2010] A. Kenner, C. Kästner, S. Haase and T. Leich. **Typechef: Toward type checking #ifdef variability in c**. In *Proceedings of the 2nd International Workshop on Feature-Oriented Software Development, FOSD '10*, pages 25–32, New York, NY, USA, 2010. ACM. 5.2.3, 6.2

[KERSTEN and MURPHY 2006] M. Kersten and G. C. Murphy. **Using task context to improve programmer productivity**. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering, FSE'06*, pages 1–11, New York, NY, USA, 2006. ACM. 2.4.3

[KICZALES *et al.* 1997] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier and J. Irwin. **Aspect-oriented programming**. In *European Conference on Object-oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997. 2.2.1, 2.2.4, 4.2.1

[KICZALES and MEZINI 2005] G. Kiczales and M. Mezini. **Aspect-oriented programming and modular reasoning**. In *Proceedings* *of the 27th International Conference on Software Engineering*, *ICSE '05*, pages 49–58. ACM, 2005. 2.4.3, 5.3

[KRUEGER 2001] C. W. Krueger. **Easing the transition to software mass customization**. In *Proceedings of the 4th International Workshop on Software Product-Family Engineering*, *PFE'01*, pages 282–293, London, UK, UK, 2002. Springer-Verlag. 2.2.2

[KUHLEMANN *et al.* 2007] M. Kuhlemann, M. Rosenmüller, S. Apel and T. Leich. **On the duality of aspect-oriented and feature-oriented design patterns**. In *6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, *ACP4IS '07*, New York, NY, USA, 2007. ACM. 4.4.4, 4.6

[LEE and KANG 2004] K. Lee and K. C. Kang. **Feature dependency analysis for product line component design**. In *Software Reuse: Methods, Techniques, and Tools*, volume 3107 of *LNCS*, pages 69–85. Springer Berlin Heidelberg, 2004. 1, 1.1.1, 2.4.1, 3, 3.5

[LEHMAN and BELADY 1985] M. M. Lehman and L. A. Belady, editors. **Program evolution: processes of software change**. Academic Press Professional, Inc., San Diego, CA, USA, 1985. 2.3

[LEIGHTON and RAO 1999] T. Leighton and S. Rao. **Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms**. *Journal of the ACM*, 46(6):787–832, 1999. 5.1.1

[LI *et al.* 2002] H. C. Li, S. Krishnamurthi and K. Fisler. **Interfaces for modular feature verification**. In *Proceedings of the 17th IEEE International Conference on Automated Software Engineering*, *ASE '02*, pages 195–, Washington, DC, USA, 2002. IEEE Computer Society. 2.4.3, 5.3

[LI *et al.* 2002b] H. Li, S. Krishnamurthi and K. Fisler. **Verifying cross-cutting features as open systems**. *SIGSOFT Softw. Eng. Notes*, 27(6):89–98, 2002. 2.4.3, 5.3

[LIEBIG *et al.* 2010] J. Liebig, S. Apel, C. Lengauer, C. Kästner and M. Schulze. **An analysis of the variability in forty preprocessor-based software product lines**. In *32nd International Conference on Software Engineering*, *ICSE'10*, pages 105–114. ACM, 2010. 2.4.2, 3.4, 4, 4.7

[LOCHAU *et al.* 2012] M. Lochau, S. Oster, U. Goltz and A. Schürr. **Model-based pairwise testing for feature interaction coverage**

in software product line engineering. *Software Quality Journal*, 20(3-4):567–604, September 2012. 3.3.3

[LOPEZ-HERREJON and BATORY 2001] R. Lopez-Herrejon and D. Batory. **A standard problem for evaluating product-line methodologies**. In *3rd International Conference on Generative and Component-Based Software Engineering*, *GCSE'01*, pages 10–24. Springer-Verlag, 2001. 3.1.3

[LORENZ 1905] M. O. Lorenz. **Methods of Measuring the Concentration of Wealth**. *Publications of the American Statistical Association*, 9(70):209–219, 1905. 3.1.4

[MCL ALGORITHM 2015] S. Dongen. **Mcl algorithm**. `http://micans.org/mcl/index.html`, 2015. [Online; accessed 09-July-2015]. 5.2.3

[MACCORMACK *et al.* 2006] A. MacCormack, J. Rusnak and C. Y. Baldwin. **Exploring the structure of complex software designs: An empirical study of open source and proprietary code**. *Manage. Sci.*, 52(7):1015–1030, July 2006. 3.1.2

[MAHDAVI *et al.* 2003] K. Mahdavi, M. Harman and R. M. Hierons. **A multiple hill climbing approach to software module clustering**. In *Proceedings of the International Conference on Software Maintenance*, *ICSM '03*, pages 315–, Washington, DC, USA, 2003. IEEE Computer Society. 5.3

[MANN and WHITNEY 1947] H. B. Mann and D. R. Whitney. **On a test of whether one of two random variables is stochastically larger than the other**. *The Annals of Mathematical Statistics*, 18(1):50–60, 03 1947. 5.2.4

[MANCORIDIS *et al.* 1998] S. Mancoridis, B. Mitchell, C. Rorres, Y. Chen and E. Gansner. **Using automatic clustering to produce high-level system organizations of source code**. In *6th International Workshop on Program Comprehension*, pages 45–52, 1998. 1.1.3, 4, 5.1, 5.3, 5.4

[MARTIN 1996] R. C. Martin. **The interface segregation principle: One of the many principles of ood**. *C PLUS PLUS REPORT*, 8:30–36, 1996. 5, 5.3

[MARIJAN *et al.* 2013] D. Marijan, A. Gotlieb, S. Sen and A. Hervieu. **Practical pairwise testing for software product lines**. In *17th International Software Product Line Conference*, *SPLC '13*, pages 227–235. ACM, 2013. 3.3.3

[MITCHELL and MANCORIDIS 2002] B. S. Mitchell and S. Mancoridis. **Using heuristic search techniques to extract design abstractions from source code**. In *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO '02*, pages 1375–1382, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc. 5.3

[PARNAS 1994] D. L. Parnas. **Software aging**. In *16th International Conference on Software Engineering, ICSE'94*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press. 1

[PASSOS *et al.* 2015] L. Passos, J. Padilla, T. Berger, S. Apel, K. Czarnecki and M. T. Valente. **Feature scattering in the large: A longitudinal study of linux kernel device drivers**. In *Proceedings of the 14th International Conference on Modularity, MODULARITY 2015*, pages 81–92, New York, NY, USA, 2015. ACM. 4, 4.1, 4.7

[PERROUIN *et al.* 2012] G. Perrouin, S. Oster, S. Sen, J. Klein, B. Baudry and Y. Traon. **Pairwise testing for software product lines: comparison of two approaches**. *Software Quality Journal*, 20(3-4):605–643, September 2012. 3.3.3

[POHL *et al.* 2005] K. Pohl, G. Böckle and F. J. v. d. Linden. **Software product line engineering: Foundations, principles and techniques**. Springer-Verlag New York, Inc., 2005. 1, 2.1, 2.1.1

[PRADITWONG *et al.* 2011] K. Praditwong, M. Harman and X. Yao. **Software module clustering as a multi-objective search problem**. *Software Engineering, IEEE Transactions on*, 37(2):264–282, 2011. 4.6, 5.2.6

[PREHOFER 1997] C. Prehofer. **Feature-oriented programming: A fresh look at objects**. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443. Springer Berlin Heidelberg, 1997. 5.3

[QUEIROZ *et al.* 2015] R. Queiroz, L. Passos, M. Valente, C. Hunsen, S. Apel and K. Czarnecki. **The shape of feature code: an analysis of twenty c-preprocessor-based systems**. *Software and Systems Modeling*, pages 1–20, 2015. 4, 4.7

[R TOOL 2015] The R Project. **R tool**. `http://www.r-project.org/`, 2015. [Online; accessed 28-May-2015]. 4.4.4, 5.2.3, 5.2.3

[RAHMAN 2004] A. Rahman. **Metrics for the structural assessment of product line architecture**. Master's thesis, *School of Engineering, Blekinge Institute of Technology*, Sweden, 2004. 2.4.2

[RAJLICH 1997] V. Rajlich. **A model for change propagation based on graph rewriting**. In *International Conference on Software Maintenance, ICSM '97*, pages 84–91. IEEE Computer Society, 1997. 1, 2.3

[RIBEIRO *et al.* 2011] M. Ribeiro, F. Queiroz, P. Borba, T. Tolêdo, C. Brabrand and S. Soares. **On the impact of feature dependencies when maintaining preprocessor-based software product lines**. In *10th International Conference on Generative Programming and Component Engineering*, *GPCE'11*, pages 23–32, New York, NY, USA, 2011. ACM. 1, 1.1.1, 1.1.2, 1.1.3, 2.1, 2.1.2, 2.4.1, 2.4.3, 3, 3.1.1, 3.4, 3.5, 4.1, 4.2.1, 4.2.2, 5, 5.3, 6.1

[RIBEIRO *et al.* 2014] M. Ribeiro, P. Borba and C. Kästner. **Feature maintenance with emergent interfaces**. In *36th International Conference on Software Engineering*, pages 989–1000, New York, NY, USA, 2014. ACM. 1, 2.4.1, 2.4.3, 3, 3.3.1, 4.1, 5.3

[ROMANO *et al.* 2014] D. Romano, S. Raemaekers and M. Pinzger. **Refactoring fat interfaces using a genetic algorithm**. In *IEEE International Conference on Software Maintenance and Evolution*, pages 351–360, 2014. 5.2.4, 5.3

[SANT'ANNA *et al.* 2003] C. Sant'anna, A. Garcia, C. Chavez, C. Lucena and A. Von Staa. **On the reuse and maintenance of aspect-oriented software: An assessment framework**. In *Brazilian Symposium on Software Engineering*, *SBES'03*, pages 19–34, 2003. 2.4.2

[SANGAL *et al.* 2011] N. Sangal, E. Jordan, V. Sinha and D. Jackson. **Using dependency models to manage complex software architecture**. In *20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, *OOPSLA '05*, pages 167–176. ACM, 2005. 3.1.2, 3.2.2, 3.3.2, 3.5, 4

[DAVIS 1962] C. Davis. **The norm of the schur product operation**. *Numerische Mathematik*, 4(1):343–344, 1962. 5.1.3

[SCHULMAN 2000] L. J. Schulman. **Clustering for edge-cost minimization**. In *Thirty-second Annual ACM Symposium on Theory* *of Computing*, *STOC '00*, pages 547–555, New York, NY, USA, 2000. ACM. 5.1.1

[SCHACH *et al.* 2003] S. R. Schach, B. Jin, L. Yu, G. Z. Heller and J. Offutt. **Determining the distribution of maintenance categories: Survey versus measurement**. *Empirical Software Engineering*, 8(4):351–365, 2003. 3, 3.1.3, 3.4

[SCHAEFER *et al.* 2010] I. Schaefer, L. Bettini, V. Bono, F. Damiani and N. Tanzarella. **Delta-oriented programming of software product lines**. In *Software Product Lines: Going Beyond*, volume 6287 of *Lecture Notes in Computer Science*, pages 77–91. Springer Berlin Heidelberg, 2010. 6.2

[SIMON 1978] H. A. Simon. **Rationality as process and as product of thought**. *The American Economic Review*, 68(2):1–16, 1978. 2.4.2, 4

[SMARAGDAKIS and BATORY 1998] Y. Smaragdakis and D. S. Batory. **Implementing layered designs with mixin layers**. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, *ECOOP'98*, pages 550–570, London, UK, UK, 1998. Springer-Verlag. 2.2.1

[SOBERNIG *et al.* 2014] S. Sobernig, S. Apel, S. Kolesnikov and N. Siegmund. **Quantifying structural attributes of system decompositions in 28 feature-oriented software product lines: An exploratory study**. Technical report, Institute for Information Systems and New Media, Vienna, Austria, 2014. 4, 4.7

[SULLIVAN *et al.* 2001] K. J. Sullivan, W. G. Griswold, Y. Cai and B. Hallen. **The structure and value of modularity in software design**. In *8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, *ESEC/FSE*, pages 99–108, New York, NY, USA, 2001. ACM. 3.1.2

[SVAHNBERG and BOSCH 2000] M. Svahnberg and J. Bosch. **Evolution in software product lines**. In *3rd International Workshop on Software Architectures for Products Families*, pages 391–422. Springer LNCS, 2000. 3.1.3, 3.4

[SZYPERSKI 2002] C. Szyperski. **Component software: Beyond object-oriented programming**. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002. 2.2.1, 2.2.4

[TARR *et al.* 1999] P. Tarr, H. Ossher, W. Harrison and S. M. Sutton, Jr. **N degrees of separation: Multi-dimensional separation of concerns**.

In *Proceedings of the 21st International Conference on Software Engineering, ICSE '99*, pages 107–119, New York, NY, USA, 1999. ACM. 2.2.4

[THAO 2012] C. Thao. **Managing evolution of software product line**. In *34th International Conference on Software Engineering (ICSE)*, pages 1619–1621, 2012. 1

[TIZZEI *et al.* 2011] L. P. Tizzei, M. Dias, C. M. F. Rubira, A. Garcia and J. Lee. **Components meet aspects: Assessing design stability of a software product line**. *Information and Software Technology*, 53(2):121–136, 2011. 4, 4.4.1, 4.4.3

[TORKAMANI 2014] M. A. Torkamani. **Metric suite to evaluate reusability of software product line**. *International Journal of Electrical and Computer Engineering*, 4(2):285–294, 04 2014. 2.4.2, 4.7

[TRUJILLO *et al.* 2006] S. Trujillo, D. Batory and O. Diaz. **Feature refactoring a multi-representation program into a product line**. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE'06*, pages 191–200, New York, NY, USA, 2006. ACM. 2.1.1

[TSANTALIS *et al.* 2005] N. Tsantalis, A. Chatzigeorgiou and G. Stephanides. **Predicting the probability of change in object-oriented systems**. *IEEE Transactions on Software Engineering*, 31(7):601–614, July 2005. 2.3, 3.2.2, 3.3.2, 3.5

[VASILESCU *et al.* 2011] B. Vasilescu, A. Serebrenik and M. Van Den Brand. **You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics**. In *27th International Conference on Software Maintenance, ICSM'2011*, pages 313–322. IEEE Computer Society, 2011. 3.1.4

[WOHLIN *et al.* 2000] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell and A. Wesslén. **Experimentation in software engineering: An introduction**. Kluwer Academic Publishers, 2000. 3.4

[XMPP 2015] XMPP Standards Foundation. **Xmpp**. `http://xmpp.org/xmpp-protocols/`, 2015. [Online; accessed 13-April-2015]. 3.1.3

[YAU and COLLOFELLO 1985] S. S. Yau and J. S. Collofello. **Design stability measures for software maintenance**. *IEEE Transansaction on Software Engineering*, 11(9):849–856, September 1985.

[YE and LIU 2005] H. Ye and H. Liu. **Approach to modelling feature variability and dependencies in software product lines**. *IEE Software*, 152(3):101–109, 2005. 1, 3

[ZHIFENG and RAJLICH 2001] Z. Yu and V. Rajlich. **Hidden dependencies in program comprehension and change propagation**. In *9th International Workshop on Program Comprehension*, pages 293–299, 2001. 1

[BERKELEY DB 2015] Oracle. **Berkeley DB**. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/downloads/index.html`, 2015. [Online; accessed 13-April-2015]. 3.1.3, 3.1.3

[BUSYBOX 2015] BusyBox. **Busybox**. `http://git.busybox.net/busybox`, 2015. [Online; accessed 09-June-2015]. 5.2.2, 5.2.3

[KÄSTNER 2015] C. Kästner. **CIDE tool**. `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/cide/`, 2015. [Online; accessed 13-April-2015]. 2.2.4, 3.1.2, 3.1.3, 4.4.4, 6.2

[CODEFACE 2015] Siemens. **Codeface**. `http://siemens.github.io/codeface/`, 2015. [Online; accessed 09-June-2015]. 5.2.3

[CAFEO *et al.* 2015] B. B. P. Cafeo, E. Cirilo, A. Garcia, F. Dantas and J. Lee. **Feature dependencies as change propagators: An exploratory study of software product lines**. `http://www.inf.puc-rio.br/~bcafeo/supsite_ist2015.html`, 2015. [Online; accessed 15-June-2015]. 3.1.2, 3.1.2, 3.1.3

[LAMPIRO 2015] Lampiro. **Lampiro**. `https://code.google.com/p/lampiro/`, 2015. [Online; accessed 13-April-2015]. 3.1.3, 3.1.3

[MOBILE RSS 2015] Mobile RSS. **Mobile rss**. `https://code.google.com/p/mobile-rss/`, 2015. [Online; accessed 13-April-2015]. 1, 3.1.3, 3.1.3

[THÜM and BENDUHN 2015] T. Thüm and F. Benduhn. **SPL2Go**. `http://spl2go.cs.ovgu.de/`, 2015. [Online; accessed 13-April-2015]. 3.1.3, 3.1.3

[VAN DER HOEK *et al.* 2003] A. van der Hoek, E. Dincel and N. Medvidovic. **Using service utilization metrics to assess the structure of product line architectures**. In *9th International Symposium on Software Metrics, METRICS '03*, pages 298–, Washington, DC, USA, 2003. IEEE Computer Society. 2.4.2, 4, 4.4.1, 4.5.2, 4.7