1
Introduction

A program family is a set of similar programs, named family members[1], that share common features and also realize variable features (Parnas 1976, Weiss and Lai 1999). Family members are developed from these common features, and variable features are included in some members. A feature is a prominent or distinctive user-visible aspect, quality or characteristic of a software system (Kang *et al.* 1990). Each feature is realized by code elements in the program implementation. Program families are often developed from either frameworks or reference architectures, which encompass the common features to be shared among the family members (Fayad *et al.* 1999, Weiss and Lai 1999). The variable features represent the particularities of each family member (Kang *et al.* 1990).

There are some advantages of developing program families, such as the possibility of evolving the implementation of common features and variable features independently. There are numerous examples of well-known program families in industry, such as Adobe Acrobat (Adobe 2012) and Mozilla (Mozilla 2012). In fact, software systems have been increasingly built as program families rather than stand-alone applications in a wide range of software domains (Alves *et al.* 2005, Figueiredo *et al.* 2008, Adobe 2012, Mozilla 2012, Android 2012). These domains range from popular operational systems or games for mobile phones, for instance, Android (Android 2012), GM (Alves *et al.* 2005) and BestLap (Figueiredo *et al.* 2008) to train control systems (Eixelsberger *et al.* 1998). Regardless of the domain, program families are often a key part of organizations' economic strategy as they become more profitable (Alves *et al.* 2005, Quilty and Cinnéide 2011). For instance, given the growing variety of mobile devices and their varying hardware constraints (e.g. memory, screen size), embedded mobile systems are often developed as a program family in order to accelerate their time-to-market and reach more users who own different devices (Alves *et al.* 2005).

The family members often evolve from frameworks or reference architec-

[1]For the remainder of the text we use the term family applications and family members interchangeably.

tures, while also accommodating specific requirements from customers. As new requirements emerge, a program family undergoes a wide range of changes during its evolution (Fluri *et al.* 2008, Alam *et al.* 2009). At the implementation level, code elements realizing each feature are often added, changed or even removed through the family evolution history. However, when such changes are performed in the program family's code without a careful planning, they tend to cause its degeneration over time (Parnas 1994, Weiss and Lai 1999).

According to Hochstein and Lindvall (Hochstein and Lindvall 2005), a degenerate software system is characterized by changes that lead to the continuous software quality decline. In particular, this decline makes the software code harder to maintain and evolve than it should be (Hochstein and Lindvall 2005). Changes made in the implementation of a program family can also lead to its degeneration. Similarly to stand-alone software systems, many symptoms (Hochstein and Lindvall 2005), such as code smells (Fowler 1999), could lead to the program family degeneration. However, we consider in this work degradation symptoms that are particularly critical to program families (Section 1.1.1). In the context of this work, a degenerate program family is characterized by changes that make it increasingly harder to distinguish the code elements realizing common and variable features in the family. When this distinction cannot be made in the implementation assets, it becomes difficult or prohibitive to maintain or evolve the program family (Parnas 1994).

Program family degeneration is often induced by feature code that is changed individually in a family member without considering other family members. There might be extreme cases where the code of the same common feature has been copied and changed inconsistently in each family member (Parnas 1994, Weiss and Lai 1999). Then, it becomes difficult to change and introduce new features in the family implementation. The underlying causes for the program family degeneration range from the lack of planning and understanding of the requirements to the time pressure and inadequate management (Eick *et al.* 2001, Hochstein and Lindvall 2005). We have observed that the program family degeneration is very often (Eick *et al.* 2001, Hochstein and Lindvall 2005). There are examples of program families in industry that degenerated over time, such as Mozilla and Netscape (Eick *et al.* 2001, Hochstein and Lindvall 2005).

Once degeneration symptoms manifest in a program family, it is particularly important, to foster restructuring actions in its implementation. The problem is that these restructuring actions cannot be performed before understanding how features are realized in the family implementation. This task is

challenging for two reasons. First, in degenerate program families, the identification of code elements realizing each feature is hard. Given inconsistent changes through the family evolution, it might be, for instance, that two or more family members are providing different implementations for the same common feature. Second, it is also hard, if not impossible, for developers to figure out which are the commonalities and variabilities as the family implementation evolves. Therefore, in order to regain control of the degenerate program family, developers need to be supported with means to identify the elements realizing common and variable features.

The remainder of this Chapter is organized as follows. Section 1.1 defines the problem tackled in this thesis. Section 1.2 points out some limitations of related work. Section 1.3 describes the aims and research questions. Section 1.4 presents the thesis contributions. Finally, Section 1.5 points out how this thesis is organized.

1.1 Problem Statement

A program family might degenerate due to unplanned changes in its implementation, thus hindering the maintenance and evolution of family members. Basically, the unplanned changes occur, for instance, when (i) one or more common features are updated individually in a member implementation without considering other family members, and (ii) the implementation of a common feature is included only in the code of a specific member. When these undesirable changes occur often, the program family degenerates over time. Section 1.1.1 discusses the degeneration of program families and Section 1.1.2 describes concrete examples of degeneration symptoms. Finally, Section 1.1.3 discusses how to recover feature code in degenerate families.

1.1.1 Degeneration of Program Families

The degeneration of program families can be caused for several reasons. In most cases, degeneration is provoked by features that started to be changed in inconsistent ways across the different members of the program family (Parnas 1994). Let's illustrate these inconsistent changes in the context of a program family with members sharing a software framework. Inconsistencies occur when the changes performed in both framework code and member code are not managed in an appropriate manner. In particular, when the program family code is partially or fully replicated and individually changed across several evolving members. Figure 1.1 illustrates this problem by showing the

version changes through a configuration management tool, e.g., Subversion (Pilato 2004). As pointed out in this figure, there are versions associated with both framework and family applications.

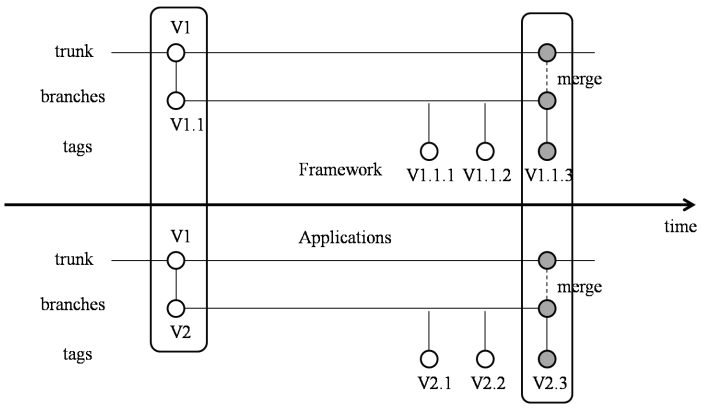

Figure 1.1: Evolution Strategy using SVN.

Figure 1.1 illustrates two cases of degradation symptoms (Parnas 1994). A degeneration symptom occurs when a common feature is updated in the framework code, but this modification is only performed in a specific framework version used in the implementation of one family member. Even though the change was intended to be reflected in all family members, it was only applied to a particular member. Figure 1.1 illustrates two cases of this degradation symptom. It is possible to observe that some framework versions (V1.1.1 and V1.1.2) are released only to support the evolution of a family application (V2.1 and V2.2). This means that there is a possibility that other family applications might not be compatible with the framework version V1.1.1 or the later ones.

In addition, a degeneration symptom can also be related to the undesirable inclusion of a variable feature into the common framework code. The inclusion of such a variable feature in the framework code means that it becomes now shared by all the family applications. This could be observed, for example, when a merge is performed (Framework V1.1.3 in Figure 1.1) and all the family applications are updated to work with this new version of the framework. However, given the varying nature of this feature, it turns out that this new feature is never used by several family members (Applications V2.3).

The aforementioned symptoms of family degeneration imply that maintenance and evolution of the family implementation become costly or even impeditive. The family implementation is no longer maintainable or evolvable without restructuring (Section 1). Then, developers will need to understand how feature code has evolved across the family history in order to restructure the family code. However, this is not a trivial goal to be achieved as it requires two main steps. First, it requires to identify the code elements realizing

each feature along each family member evolution. Second, it requires to understand the variability nature of the features' code elements throughout the program family history. These steps are essential to enable the restructuring of the program family code, and consequently the elimination of its degradation symptoms.

1.1.2 Examples of Degeneration Symptoms

This section describes concrete examples of the aforementioned degeneration symptoms. These examples have been observed and analysed in an evolving industrial program family, so-called OC. The OC program family belongs to a Brazilian company and the main family members of the company are for managing logistics of the oil industry. These family applications were derived from a single framework. We have omitted the name of the program family and its applications due to confidentiality agreement issues. The name "*Oil Control*" (OC) will be used to refer to the program family including the framework and three family members. The three family members are called *Application I*, *II* and *III*. Further information about the OC program family is presented in Section 3.1.1. Two features are used to illustrate the degeneration symptoms of this program family during its evolution: *Scenario* and *Export*. The feature *Scenario* is for managing the exportation and importation of products. The feature *Export* is for generating reports in different file formats.

1. **The implementation of common features becomes inconsistent across the evolving family members.** In this first case, the problem is that specific functionalities are updated in the framework code only for one family member. In Figure 1.2, both *Applications I* and *II* derived from the *OC framework* have the feature *Export*, which is a common feature. *Application I* inherited the basic code elements of the feature *Export* that belong to the *OC framework* and which are shared by all the family applications. According to the figure, the clear grey box represents one of these basic code elements in the *OC framework* and *Application I*, the `ExportDialog` class. On the other hand, this same code element (`ExportDialog` class) inherited by *Application I* was modified to work with *Application II*. This was carried out to implement a change related to the feature *Export* (lines 05-15) in framework code. For this reason, the `ExportDialog` class is represented by a dark grey box in *Application II*. As a result, a new version of the *OC framework* with this change was released in order to work exclusively with *Application II*.

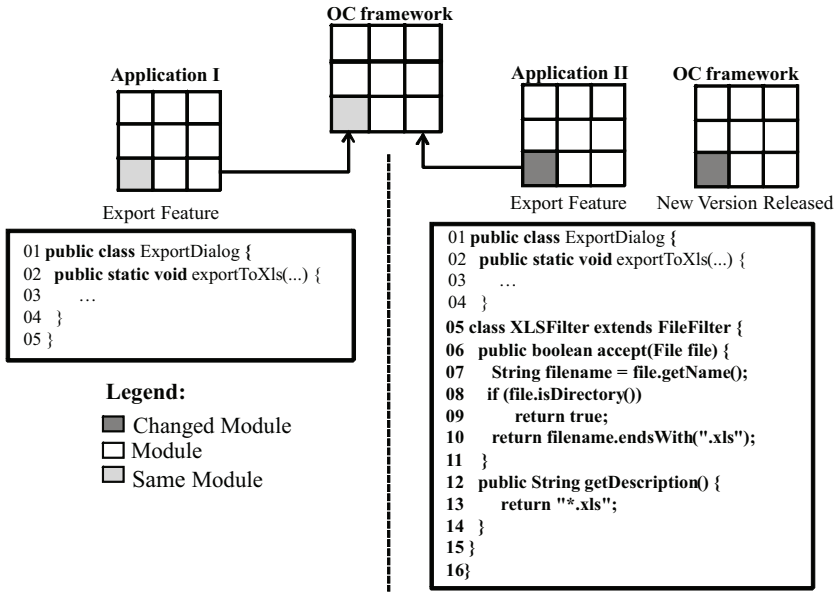

Figure 1.2: Modification of Framework Code Elements with Variable Code.

As a consequence, the *OC framework* has started to be evolved for single family applications. Therefore, the changes included within the `ExportDialog` class in the framework code for a specific application version (*Application II*) caused the elimination of the common characteristics of the program family. For example, we could notice the differences of the `ExportDialog` class in *Applications I* and *II*. In *Application II* code, the changes (lines 05-15) were performed within the `ExportDialog` base class, which realizes a common feature. The main problem associated with this degeneration symptom is that the modified framework code might not be compatible with the code of certain family members anymore. As a consequence, the code maintenance tasks for both framework and members become cumbersome. Inappropriate solutions might be further applied, such as the creation of an unnecessary branch of the framework version.

2. **The implementation of member-specific code is intermingled with core code.** In this case of degradation, code elements that should be in only one family member are placed into modules of the program family core at some point. This degeneration symptom is illustrated through the `ScenarioServiceProxy` class, which belongs to *Application I*, and it is represented by a grey box in Figure 1.3. This class realizes the feature *Scenario*, which is a variable feature. During the evolution of the framework and family applications, the `ScenarioServiceProxy` class was integrated into the program family core, also represented by a dark grey box in the *OC framework*. This implies that this new version

of the framework with this new feature was shared by all the versions of the family applications.

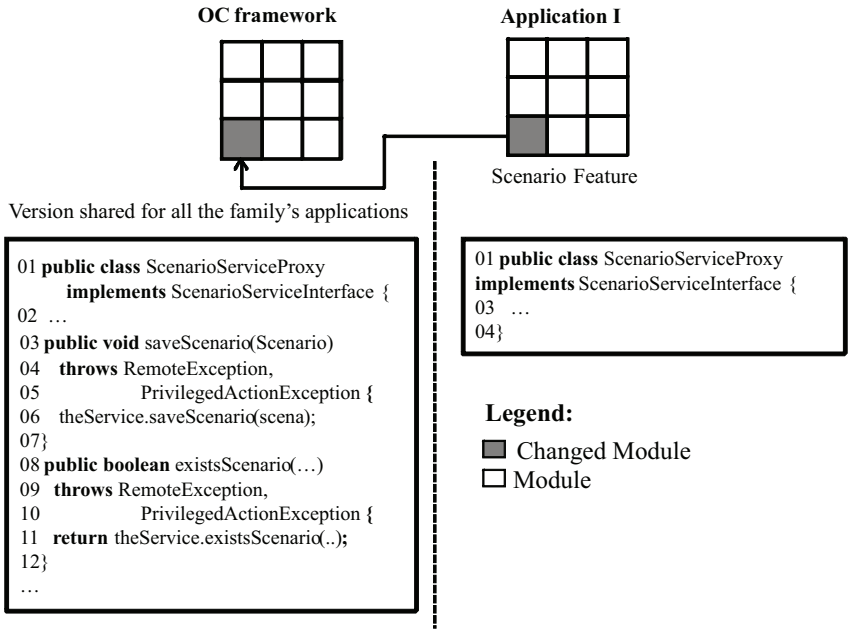

Figure 1.3: The Implementation of Member-specific Code is Intermingled with Core Code.

In addition, this class started being modified only in the *OC framework* versions (lines 03-12). For example, new methods were included within the `ScenarioServiceProxy` class in the *OC framework*, as illustrated in Figure 1.3 (lines 03-12). However, the changes included in the framework code should have only been performed in the application code (*Application I*) since they are application-specific code (variable feature code). All the applications of the program family have access to this specific code, but there is actually only one application that uses it. The side effect is that this code segment was replicated in the framework for all the versions of the applications but was used by only one application.

1.1.3 How to Recover Feature Code in Degenerate Families?

Given the presence of degeneration symptoms in program families (Section 1.1.2), it is important to observe that it is no longer possible to distinguish what code elements are realizing each family feature over time. It is not possible either to classify the code elements contributing to the implementation of common and variable features in the family implementation. Consequently, developers do not have full knowledge about the realization and classification of each feature' implementation element (e.g. method, attribute). The importance of this knowledge becomes apparent in a wide range of maintenance and

evolution tasks, such as: (i) understanding each feature implementation in a family member before realizing a given change, and (ii) performing a partial or complete reengineering of the family code.

In this context, a key step is to analyse which and how the code elements of the program family's features have been changed across the evolution history of each family member. This analysis is critical to understand how the variability nature of the features' code elements evolved over time. We call this process *forward recovery of family features*, which aims at: (i) analysing the evolution from the first version to the last one of each family member, and (ii) exploiting this forward analysis to identify and classify the implementation elements (e.g. methods and attributes) according to their variability degree. The forward recovery of the features' code elements is the representation of the program family obtained from the analysis of its source code.

However, as a consequence of the degradation symptoms (Section 1.1.1), this recovery is far from trivial because of the following issues: (i) the implementation of common features evolves in different ways across the members of the program family; (ii) the variability classification of features is no longer consistent across all the family members; and finally (iii) it is common that there is no proper and updated documentation about the program family, and only its source code remains as the reference artefact.

1.2 The State of the Art on Design Recovery and Feature Mapping

This section discusses related work which could be used to circumvent the problems related to the identification and classification of the features' implementation elements in degenerate program families. Previous contributions in two key areas of research are relevant in this context: design recovery and feature mapping. We discuss here related work in the broad spectrum of these areas that could somehow contribute to the resolution of each of these problems. As they cannot be used to address those problems, their limitations are made clear below.

According to Chikofsky and Cross (Chikofsky *et al.* 1990), design recovery is a reverse engineering activity in which domain main knowledge and external information are added to the observations of the subject system to identify meaningful high-level abstractions beyond those obtained directly by examining the system itself. In particular, design recovery aims to recreate the original design of a software system. Taking into consideration this definition, some research work has explored the software design recovery at different levels (Arango 1998, Kramer and Prechelt 1996,

Knodel 2005, Shi and Olsson 2006, Tsantalis *et al.* 2006, Krikhaar 2009, Abi-Antoun and Aldrich 2009, Alnusair and Zhao 2010). Research work has proposed different processes, methodologies and tools that extract information from the source code of individual applications and reason about it for recovering the original software architecture or specific design information (e.g. design patterns).

Abi-Antoun and Aldrich (Abi-Antoun and Aldrich 2009) defined a technique called SCHOLIA to statically extract hierarchical runtime architecture from object-oriented code using annotations. This technique follows the extract-abstract-check strategy and it is used to extract and compare an extracted architecture with the target architecture. Others such as (Kramer and Prechelt 1996, Kazman and Carrière 1999, Shi and Olsson 2006, Tsantalis *et al.* 2006, Alnusair and Zhao 2010) are interested in identifying design patterns by analysing the source code. Sartipi (Sartipi 2003) presented a model for recovering the high level design of legacy software systems based on user-defined architectural patterns and graph matching techniques.

Unfortunately, none of related work discussed above considers the concept of features during the design recovery process. However, there are recently-proposed approaches that try to cover this gap. There are also some studies that reported reengineering processes of legacy systems into product assets (Kang *et al.* 2005, Kang *et al.* 2006, Kästner *et al.* 2007, Tulio *et al.* 2012). Basically, these studies reported the challenges and adopted methodologies during the extraction process of product assets. They can be used to recover relevant information about each feature. For instance, Tulio *et al.* (Tulio *et al.* 2012) proposed a semi-automatic approach for extracting optional features based on a set of feature seeds. Other studies explored source code history to detect change dependencies related to crosscutting features (Adams *et al.* 2010, Nguyen *et al.* 2011, Marin *et al.* 2007). For instance, (Adams *et al.* 2010, Nguyen *et al.* 2011) proposed techniques for identifying code elements realizing non-functional crosscutting features in evolving software systems. However, these techniques do not support the evolution history analysis of program families. The advantages and shortcomings of these techniques are discussed in Section 2.4.

Moreover, there has been a growing range of techniques that assist developers in the feature mapping activity (Wilde and Scully 1995, Wong *et al.* 1999, Chen and Rajlich 2000, Eisenbarth *et al.* 2003, Eisenberg and Volder 2005, Robillard and Murphy 2007, Marin *et al.* 2007, Zhang *et al.* 2008). Feature mapping activity refers to the explicit identification of all code elements (e.g. methods) responsible for real-

izing each feature (Eisenbarth *et al.* 2003, Eisenberg and Volder 2005, Robillard and Murphy 2007). However, the feature mapping techniques were not explicitly conceived to explore evolving members of a program family. These techniques were designed to support feature mapping taking into consideration only a single version of a single application.

Therefore, although there are many studies that explore the design recovery at different artifact levels of a software system (e.g. architecture and source code), all of them do not tame the aforementioned degeneration symptoms in program families (Section 1.1). In particular, two main limitations were identified in the related work. First, they are unable to explore historical information of evolving program families. Second, they are unable to support the forward recovery (Section 1.1.3) of the code elements realizing their common and variable features even exploring individual application versions. To sum up, developers are not equipped with any kind of method or tool support for the forward recovery of features' implementation elements in degenerate program families.

1.3 Aims and Research Questions

The main goal of this thesis is to support forward recovery of program family's features. As mentioned in Section 1.1.3, the forward recovery is intended to identify the feature elements and their variability degree according to the full evolution history of the family members. The output of the forward recovery process is twofold: (i) the list of code elements realizing each feature within each family member version, the so-called feature mapping. There is a feature mapping for each version of the family member; and (ii) the classification of the code elements according to their variability degree. These outputs can be useful to a wide range of family restructuring actions. For instance, it can be used to support code refactoring activities across the program family in order to tame family degradation symptoms (Section 1.1.1).

As a first step in our research, we study how much challenging is to produce feature mappings made by developers. We perform a study to categorize mistakes (or mismatches) when developers are identifying feature elements in the source code. In the context of our work, this is particularly important for two reasons: (i) to reduce the occurrences of false negatives and false positives in the feature mappings generated for each version of the family member; and (ii) to classify the variability degree of the code elements realizing the family features.

Therefore, in order to achieve the aforementioned goal, we address the

following four research questions (RQ) in this thesis:

RQ1. Which are the typical mismatches made by developers when mapping features in the implementation?

RQ2. How to automatically generate feature mappings in evolving program families?

RQ3. How to automatically classify the variability degree of each family element?

RQ4. What is the accuracy of our forward recovery process?

The first research question is related to the mismatches that emerge when identifying the features' code elements. The aim is to analyse the typical mapping mismatches made by developers. We classify and document a set of recurring types of feature mapping mismatches. The second research question aims at defining a set of heuristics to help developers generate (or expand) feature mappings in evolving program families. The mapping heuristics generate feature mappings for all the versions of the family members. These heuristics are called mapping expansion heuristics as they involve the automatic identification of feature elements in the code departing from an initial mapping (seed). They also aim at reducing the number of mapping mismatches, detected in the previous research question (RQ1), through the feature evolution in the family code. The third research question aims at defining a set of recovery heuristics to classify the code elements realizing the family features. The recovery heuristics use as input information the expanded feature mappings generated by the mapping expansion heuristics (RQ2). The fourth research question evaluates the accuracy of our forward recovery process.

1.4 Thesis Contributions

This section briefly describes the contributions of this thesis, namely a classification of mapping mismatches, heuristics for expanding feature mappings, heuristics for recovering the features' implementation elements, the tool support that we developed and a set of studies to evaluate the accuracy of the two proposed suites of heuristics. These contributions are summarized in scientific publications presented in Table 1.1. There are also some indirect publications that arose during the definition and conception of this thesis, which are described in Table 1.2. All the contributions are briefly described as follows.

1.4.1 A Catalogue of Recurring Feature Mapping Mismatches (RQ1)

Identifying feature mapping mismatches is an important step to be performed before the maintenance and evolution tasks. This step is particularly important as the existence of mapping mismatches may lead developers to incorrectly understand and implement a change in a feature's module. Consequently, the lack of mismatches in feature mappings can help developers implement systematically and successfully maintenance and evolution tasks. In this sense, a set of recurring types of feature mapping mismatches were empirically identified and characterized in our research. More importantly, the identification and classification of mismatches was used as basic information for the definition and formalization of our suite of mapping expansion heuristics (Section 1.4.2). The mismatches are made either manually by developers or by applying existing feature mapping techniques (Chapter 2). They are related to our first research question (RQ1 in Section 1.3). The mismatches were identified and categorized through the empirical analysis of evolving feature mappings in program families. We also run two empirical evaluations in order to observe the occurrences of such mismatches using two different software systems. The catalogue of mapping mismatches is a concrete contribution of this work as it serves as documentation and guidance to tool developers and software engineers while correcting and extending their feature representation.

1.4.2 A Suite of Mapping Expansion Heuristics (RQ2)

Based on the identification and characterization of mapping mismatches (Section 1.4.1), a set of heuristics to automatically expand feature mappings in evolving program families was defined. The expansion refers to the action of automatically generating the feature mappings for each family member version by systematically considering its previous change history. These heuristics improve the accuracy of those mappings by reducing the occurrence of mismatches given a set of evolving members of the same family. They are related to our second research question (RQ2 in Section 1.3). First, we defined and formalized the mapping expansion heuristics. Second, we defined a heuristic method that takes into consideration the program family's change history. Finally, we design and implement a tool, named MapHist, which supports the use of the proposed mapping heuristics.

The MapHist tool was implemented as an Eclipse plug-in (Eclipse 2011). This suite of heuristics is a novelty of this thesis as it explores the knowledge and analysis of the program family's change history, which are not addressed

by existing research work. The mapping expansion heuristics were also useful to foster advances on the visualization of evolving program families. This was achieved through the integration of the mapping expansion heuristics with a visualization tool named SourceMiner evolution (SME) (Novais *et al.* 2012). This integration enables developers to visualize and analyse the feature evolution through multiple views provided by the SME tool.

1.4.3 Recovery Heuristics for the Classification of Feature Elements (RQ3)

A set of recovery heuristics for classifying the implementation elements of each family feature was defined and formalized. These heuristics rely on the analysis of the previously generated feature mappings by the mapping expansion heuristics (Section 1.4.2). The goal of these recovery heuristics is to reveal which implementation elements should be part of the common and varying parts of the new recovered program family. The heuristics analyse, for example, the amount of common implementation elements realizing a given feature in the mappings when considering all the family members. This type of information is relevant because if many family members, and consequently their features, reference the same set of classes throughout their versions, it can be a strong indicator that these classes should be present in the common features of the recovered program family. We implemented the heuristics to provide automatic support for the recovery of program family's features.

1.4.4 Empirical Evaluation (RQ4)

We evaluated the accuracy of the two proposed suites of heuristics through two industrial program families. The first suite that encompasses the expansion of feature mappings (Section 1.4.2) was evaluated by analysing the accuracy of the feature mappings generated by the MapHist tool with the participation of the original developers of the analysed program families. The second suite that encompasses the forward recovery heuristics (Section 1.4.3) was evaluated comparing the obtained results by the recovery heuristics and a manual approach. The manual approach consists of the original developers' participation of both case studies with the goal of evaluating and validating the results obtained by the recovery heuristics.

Table 1.1: Publications related to this Thesis.

Direct Publications	**Research Question(s)**
Nunes, C., Garcia, A., Lucena, C. History-Sensitive Recovery of Product Line Feature. In Proceedings of the International Conference on Software Maintenance (ICSM) - Doctoral Symposium, Romania, 2010, pp. 1-2.	all
Nunes, C., Garcia, A., Figueiredo, E., Lucena, C. Revealing Mistakes in Concern Mapping Tasks: An Experimental Evaluation. In Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), Germany, 2011, pp. 101-110.	RQ1
Nunes, C. On the Proactive Identification of Mistakes on Concern Mapping Tasks. In Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD) - Student Research Competition, 2011 (1st Place - http://src.acm.org/winners.html), pp. 85-86.	RQ1, RQ2
Nunes, C. Heuristic Expansion of Feature Mappings in Evolving Program Families. Candidate for the ACM Student Research Competition Grand Finals, 2012 (http://src.acm.org/2012/CamilaNunes.pdf)	RQ2, RQ4
Nunes, C., Garcia, A., Lucena, C., Lee, J. Heuristic Expansion of Feature Mappings in Evolving Program Families. Journal of Software Practice and Experience, 2012 (submitted).	RQ2, RQ4
Novais, R., **Nunes, C.**, Lima, C., Cirilo, E., Dantas, F., Garcia, A., Mendonça, M. On the Proactive and Interactive Visualization for Feature Evolution Comprehension: An Industrial Investigation. In Proceedings of the International Conference on Software Engineering (ICSE), Software Engineering in Practice, Zurich, 2012, pp. 1044-1053.	RQ2, RQ4
Nunes, C., Garcia, A., Lucena, C., Lee, J. History-Sensitive Heuristics for Recovery of Features in Code of Evolving Program Families. In Proceedings of the International Software Product Line Conference (SPLC), Brazil, 2012, pp. 136-145.	RQ3, RQ4

Table 1.2: Indirect Publications.

Indirect Publications
Figueiredo, E., Garcia, A, Maia, M., Ferreira, G., **Nunes, C.**, Whittle, J. On the Impact of Crosscutting Concern Projection on Code Measurement. In Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD), Porto de Galinhas, 2011, pp. 81-92.
Diniz, A., **Nunes, C.**, Silva, V. T., Fonseca, B., Lucena, C. JAAF+T: A Framework to Implement Self-Adaptive Agents that Apply Self-Test. In Proceedings of the ACM Symposium on Applied Computing, Sierre, 2010, pp. 928-935.
Nunes, I., Choren, R., **Nunes, C.**, Fabri, B. ; Carvalho, G., Lucena, C. Supporting Prenatal Care in the Public Healthcare System in a Newly Industrialized Country. In Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS) - Industry Track, 2010, Toronto, 2010, pp. 1723-1730.
Dantas, F., **Nunes, C.**, Garcia, A. ; Kulesza, U., Lucena, C. Stability of Software Product Lines with Class-Aspect Interfaces: A Comparative Study. In 4th Workshop on Assessment of Contemporary Modularization Techniques (ACOM), Jeju Island, 2010.

1.5 Outline of the Thesis Structure

In the remainder of this thesis document, we present the catalogue of the feature mapping mismatches. Based on these mismatches, we define a suite of heuristics for expanding feature mappings, a heuristic method for the recovery of features, a suite of recovery heuristics and the support tool. Finally, we also present the assessment of our suites of heuristics through two industrial program families. This document is structured as follows.

Chapter 2: *Background and Related Work*. This chapter presents the main concepts approached in this thesis (Section 2.1) and an overview of the current state of the art by contrasting the similarities and differences with respect to our work. It explains the motivation of feature mapping activity and its relevance to maintenance and evolution tasks (Section 2.2). It also describes current feature mapping techniques by explaining the main concepts, advantages and drawbacks. This chapter highlights the differences from our work and others that proposed methods and reengineering methodologies for the recovery of software system's artifacts (Section 2.3). Finally, it also presents some studies about code refactoring and history analysis (Section 2.4), and code clone tools (Section 2.5).

Chapter 3: *Mismatches in Feature Mappings*. This chapter presents the definition and classification of eight feature mapping mismatches. These mismatches were observed and categorized in members' versions of an evolving program family (Section 3.1). The mapping mismatches were described in two categories: feature characteristics and module characteristics (Section 3.2). It also discusses the relationships among the mapping mismatches (Section 3.3). Additionally, it empirically evaluates the occurrence of such mismatches in the context of two software systems (Section 3.4). Finally, it also distinguishes the contributions of the catalogue of mapping mismatches with respect to closest related work regarding feature mapping studies and techniques (Section 3.5). The results of this chapter have been reported in two papers (Nunes *et al.* 2010, Nunes *et al.* 2011).

Chapter 4: *Mapping Expansion Heuristics*. This chapter presents the heuristic method that takes into consideration the program family multi-dimensional history to expand the feature mappings (Section 4.3). It defines and formalizes a cohesive set of five mapping expansion heuristics (Section 4.4). It also presents our tool, MapHist, which provides support to the application of the proposed heuristics (Section 4.5). It evaluates the accuracy of the mapping heuristics through two program families (Section 4.6). Finally, it also discusses the integration of the mapping expansion heuristics with a visualization tool

in order to provide developers with different graphical representations to visualize the differences and evolution of the feature mappings (Section 4.8). The results of this chapter have been reported in three papers (Nunes 2011, Nunes *et al.* 2012a, Novais *et al.* 2012).

Chapter 5: *Recovery Heuristics of Feature Elements*. This chapter presents the suite of recovery heuristics. The first step was to extend the methodology of the mapping expansion heuristics by including a new step that supports the classification of features' implementation elements (Section 5.1). It defines and formalizes the recovery heuristics and their categories (Section 5.2). It also describes the algorithmic solution used to implement the recovery heuristics (Section 5.3). Finally, it evaluates the accuracy of the recovery heuristics through two industrial program families (Sections 5.4 and 5.5). The results of this chapter have been reported in two papers (Nunes *et al.* 2010, Nunes *et al.* 2012b).

Chapter 6: *Final Remarks and Future Work*. This chapter presents the final remarks, a summary of our contributions and future work.