2
Background and Related Work

A key activity to support the analysis of degenerate program families is the identification of how each feature is realized in the source code. This activity is called feature mapping. Once that feature mappings are produced for all family members, existing techniques can be used to recover fundamental information about each feature, such as its variability degree. Based on these analytical activities, refactoring strategies can be applied to address symptoms of program family degeneration.

Therefore, this chapter provides an overview of existing techniques to support feature mapping, design reengineering and refactoring activities. Along the sections, we also present an overview of the limitations of such techniques to be applied to the context of recovering degenerate program families. Given the empirical nature of certain contributions of this thesis, we also describe some existing studies on addressing different aspects of degenerate program families. Before discussing those techniques, we present an overview of the main concepts used throughout this thesis.

The remainder of this chapter is organized as follows. To begin with, the terminology addressed in this thesis is defined in Section 2.1. Subsequently, Section 2.2 discusses the feature mapping activity and its relevance to software maintenance and evolution tasks. It also discusses other similar terms commonly adopted in the literature, such as concept assignment problem, feature location and traceability. The definition of these terms is important to clearly understand their similarities and differences when comparing to the feature mapping activity. In addition, this section presents a set of existing techniques and tools for supporting feature mapping activity. Section 2.3 presents a critical review of existing studies that describe reengineering methodologies and techniques to improve the reusability of software infrastructures. These methodologies and techniques try to create design abstractions of a software system from different software assets, such as source code and design documentation. Their goal is to help developers understand the software assets and improve their reusability. Section 2.4 presents some studies that explore the code history as information source with the goal of identifying certain types

of features in individual evolving applications. Section 2.5 discusses clone detection techniques and presents the main aspects that differentiate them from our work. The discussion of these techniques is relevant in order to make it clear why they can not be used directly to classify the feature's implementation elements in evolving program families. Finally, Section 2.6 summarizes this chapter.

2.1 Terminology

Program families consist of two or more members that share common features (Weiss and Lai 1999). The different properties of each family member comprise the *variable features* (Weiss and Lai 1999, Xue *et al.* 2010). The program family increases as new family members are developed from the common features and realize different combinations of variable features (Weiss and Lai 1999). Given this scenario, it is worthwhile to study the common and variable features of each family member (Parnas 1976). A *feature* is defined as a prominent or distinctive user-visible aspect, quality, or characteristic of a software system (Kang *et al.* 1990). Features can be classified, for instance, as functional (e.g. sorting) and non-functional (e.g. exception handling) (Apel and Kästner 2009). *Variable features* can be classified into two categories: optional and alternative features (Kang *et al.* 1990). *Optional features* can be present or not in an software system. *Alternative features* should be present in a software system as they are mutually exclusive.

A program family may not be considered a software product line (SPL) (Weiss and Lai 1999, Parnas 2008). *SPL* is a set of software-intensive systems sharing a common, managed set of features that satisfy specific needs of a particular market or mission with a well-defined scope, and that are developed from a common set of core assets in a prescribed way (Clements and Northrop 2002). These assets are not only restricted to code elements, but any software element (e.g. architecture, requirements) used to generate the software systems of the SPL (Clements and Northrop 2002). SPL aims mainly at reducing time-to-market and increasing the reuse of software assets during the development of similar software systems (Clements and Northrop 2002, Apel and Kästner 2009). According to Parnas (Parnas 2008), several product lines could contain members of a family. The features of a SPL can be represented through a *feature model* that describe the common and variable features of software systems (Kang *et al.* 1990, Apel and Kästner 2009). The *feature model* also describes the relationships and dependencies among the features of a particular domain.

As our work is centered on analysing the program family evolution, it is also important to define and differentiate the terms software maintenance and evolution. The terms maintenance and evolution are very broad in the literature and have many definitions and interpretations. In this work, we have used the definitions proposed by IEEE. *Maintenance* is defined as the process of modifying a software system or component after delivering it with the goal of correcting faults, improving performance or other quality attributes, or adapting it to a changed environment (IEEE 1990). In a fine-grained view, Lientz and Swanson (Lientz and Swanson 1980) divided the aforementioned definition of maintenance into three categories: corrective, perfective and adaptive maintenance. On the other hand, *evolution* encompasses the process of managing all the changes made on software systems (Lehman 1980, Lehman 1984). According to the Lehman's laws of evolution (Lehman 1980, Lehman 1984), software systems are always condemned to change over time.

It is widely recognized that program families are constantly evolving over time (Parnas 1976, Parnas 1994). For this reason, a significant number of changes is carried out. At the implementation level, the implementation elements realizing the family features are often added, changed or even removed. The *implementation elements* are represented by methods, classes and attributes in the family member implementation when considering object-oriented programming. However, these changes can lead to the program family degeneration. A *degenerate program family* is characterized when its original source code has evolved in such a way that it is no longer possible to identify and classify the implementation elements realizing common and variable features in the family.

Therefore, it is essential to analyse the evolution of degenerate program families with the goal of forward recovering their features' implementation elements. The *forward recovery* analyses the program family evolution in order to identify and classify the features' implementation elements according to their variability nature. This way, the forward recovery requires feature mapping (Section 2.2.1) and feature classification.

2.2 Feature Identification Support

This section discusses the feature mapping activity (Section 2.2.1) and a set of techniques and tools that support such an activity (Sections 2.2.2 to 2.2.5).

2.2.1 Feature Mapping Activity

Understanding and restructuring how features are realized in the source code are often required in software maintenance and evolution tasks. For this reason, developers have to gather full knowledge about all the implementation elements that realize one or more features (Eisenbarth *et al.* 2003, Robillard and Murphy 2007). Maintenance and evolution tasks often require the explicit *identification* of all code elements responsible for realizing each feature of a software system, the so-called *feature mapping* (Wilde and Scully 1995, Chen and Rajlich 2000, Eisenbarth *et al.* 2003, Rohatgi *et al.* 2008). This is essentially important before performing a change in a software system as developers must identify the main code elements associated with these tasks. The feature mappings are also relevant when: (i) developers need to analyse and evolve software systems or program families over time (Parnas 1976, Eisenbarth *et al.* 2003, Robillard and Murphy 2007); and (ii) when the implementation elements of a feature are tangled and scattered through many modules of the software system (Kiczales *et al.* 1997, Robillard and Murphy 2007, Figueiredo 2009). In this thesis, the terms identification and location of code elements are used interchangeably.

Feature mappings are often required in a wide range of maintenance and evolution tasks performed on software systems and program families (Alves *et al.* 2005, Kolb *et al.* 2006). For instance, program families often evolve, and each feature implementation needs to be previously understood before realizing a change on it. To this end, feature mappings are fundamental to provide developers with a full knowledge about the realization of each feature and its relationships with other features in the code. Therefore, the lack of feature mappings in program families harms a wide range of maintenance tasks, such as (i) understanding how a feature is implemented in a family member (Chen and Rajlich 2000, Eisenbarth *et al.* 2003, Robillard and Murphy 2007); (ii) refactoring the implementation of specific features (Ratzinger *et al.* 2007, Kästner *et al.* 2007); (iii) the complete reengineering of the program family (Kang *et al.* 2005, Zhang *et al.* 2011); and (iv) the recovery of the family's software architecture (Eixelsberger *et al.* 1998). Some studies evaluated the accuracy of the feature mapping activity in different software systems. The limitations of these studies are important to motivate the catalogue of mapping mismatches related to specific features and/or code structure (see Section 3.5.1). The mapping mismatches are thoroughly discussed in Chapter 3.

There are also other terms used in the literature to the feature mapping activity. These terms also involve the action of assigning implementation ele-

ments to a feature, which are: concept assignment problem, feature location and feature traceability. Concept assignment problem is the process of discovering high-level concepts and assigning them to the respective artifacts within a specific program (Biggerstaff *et al.* 1994). It allows developers to understand the program through the relationship between the high-level concepts and parts of the program. Feature location techniques aim at identifying automatically at least one code element that contributes to the feature implementation based on the execution analysis of a program (Wong *et al.* 1999, Eisenbarth *et al.* 2003, Eisenberg and Volder 2005, Poshyvanyk *et al.* 2007). Therefore, this term has been widely used in the context of dynamic techniques for feature mapping (Section 2.2.3). The notion of concept assignment is more general than feature location because it is related to the understanding of any concept relevant to the program structure.

On the other hand, the term traceability is defined as a way of establishing relationships between the different software artifacts (Cleland-Huang *et al.* 2007, Grechanik *et al.* 2007, Rilling *et al.* 2007). These artifacts can be architectural descriptions and diagrams, requirements description or UML models, classes or methods, and test cases. The explicit relationships or links among the different artifacts in a software system allow developers to understand, for instance, the change impact analysis, the relations and the respective dependencies among these artifacts. In summary, traceability is different from the mapping activity as it is broader and deals with the semantic relationship among different software artifacts. The feature mapping activity is more specific and only focuses on explicitly identifying the code elements realizing the features. In the context of our research work, we have used the term feature mapping activity to the process of identifying code elements realizing a given feature.

There is a vast amount of techniques and tools addressing the feature mapping activity (Eisenbarth *et al.* 2003, Robillard and Murphy 2007, Antoniol and Gueheneuc 2005, Kellens *et al.* 2006, Savage *et al.* 2010, Adams *et al.* 2010). Feature mapping techniques are usually classified as static (Chen and Rajlich 2000, Robillard and Murphy 2007) (Section 2.2.2), dynamic (Wong *et al.* 1999, Eisenberg and Volder 2005, Koschke and Quante 2005, Cornelissen *et al.* 2009) (Section 2.2.3) or hybrid (Eisenbarth *et al.* 2003, Poshyvanyk *et al.* 2007, Rohatgi *et al.* 2008) (Section 2.2.4). These different techniques rely on different types of information, such as the static structure of software systems and the program execution traces. Unfortunately, none of them is able to identify the features' elements from an evolving program family. This limitation is mainly because these techniques

only take into consideration the history of individual application versions. The following subsections present an overview, advantages and limitations of each technique.

2.2.2 Static Techniques

Static techniques require the interaction between developers and the tool in order to assign semi-automatically each feature to the implementation elements (Chen and Rajlich 2000, Eisenbarth *et al.* 2003, Robillard and Murphy 2007, Marin *et al.* 2007, Zhang *et al.* 2008). For instance, some tools provide functionalities to search code elements of a chosen feature (e.g. lexical information). Consequently, developers can complement their feature mappings based on the outputs provided from a particular search.

The analysis of static tools is performed on a particular program code without the need of executing it. These tools are based on dependency graphs derived from the static code structure (Lyle and Weiser 1987, Chen and Rajlich 2000, Robillard and Murphy 2007). For this reason, they provide information with different types of dependencies in a system, such as data-flow dependencies. Some examples of static tools are ConcernMapper, ConcernTagger, FEAT, Concern Manipulation Environment (Chung *et al.* 2005, Robillard and Murphy 2007, Savage *et al.* 2010).

The static techniques that provide search mechanisms use the fan-in number to identify and search code elements of a given feature (Marin *et al.* 2007, Zhang *et al.* 2008). Fan-in is based on the number of distinct methods that invoke a given method. For instance, the concern graph proposed by Robillard and Murphy (Robillard and Murphy 2007) also uses the fan-in and fan-out queries. Fan-in queries return elements that know about the queried element. Fan-out queries return elements that the queried element knows about. Its goal is to help the developers to locate, analyse and document scattered features based on pre-defined queries. The static techniques only use the source code as the principal artifact, differently from the dynamic and hybrid techniques that also require suites of test cases (Sections 2.2.3 and 2.2.4). Moreover, the static techniques also provide the developer to visualize the classes, methods and attributes related to each feature from a tree-structure visualization. In particular, static techniques are more suitable for developers who have some knowledge of the software system's features and source code. This occurs because the developers have to manually map the features' implementation elements. For this reason, even though these static tools have facilitated the feature mapping activity, there are some drawbacks associated with these

tools. For instance, developers still need to check if their mappings are correct and complete. As a result, this activity tends to be cumbersome and performed in an ad-hoc fashion.

2.2.3 Dynamic Techniques

The dynamic techniques aim at identifying code elements that implement user-level features through program executions (Wilde and Scully 1995, Wong *et al.* 1999, Eisenberg and Volder 2005, Koschke and Quante 2005, Cornelissen *et al.* 2009, Savage *et al.* 2010). For example, some techniques are software reconnaissance (Wilde and Scully 1995) and execution slices (Wong *et al.* 1999). They rely on execution trace analyses to locate code implementing user-level features. User-level features only represent functional features which characterize a system's behavior that can be trigged by the user. The resulting execution traces are compared to the specific components that realize the user-level features. For instance, FLAT3 (Savage *et al.* 2010) combines dynamic traces with information retrieval to identify user-level features. FLAT3 is based on several existing tools, such as the lucene library, MUTT, ConcernTagger and ConcernMapper.

Most of these dynamic techniques rely on the use, extension and tuning of pre-existing test suites. These suites need to have a high degree of code coverage for a given system. This often implies that a test coverage analysis is required in order to estimate the accuracy of feature mappings. However, dynamic techniques are usually not accurate, and most of the tools only provide coverage at the level of methods and branches. Additionally, another limitation of the dynamic techniques is that they can not often distinguish overlapping features. This occurs because they only identify methods that are executed for specific scenarios of user-level features. Thus, the same methods may also contribute to several feature implementations. Other weaknesses are: limited scalability due to the huge amount of data may be derived from a program execution trace, the difficulty to decide which is the optimal set of test cases to be executed (Poshyvanyk *et al.* 2007, Cornelissen *et al.* 2009).

2.2.4 Hybrid Techniques

Other authors proposed hybrid techniques that blend characteristics of both static and dynamic techniques (Eisenbarth *et al.* 2003, Poshyvanyk *et al.* 2007, Rohatgi *et al.* 2008). They follow the same ideas of the dynamic techniques because they use execution traces of a program to

identify the code elements realizing each feature. Additionally, they aim at improving the accuracy of feature mappings by also integrating static techniques. Hybrid techniques aim at reducing the disadvantages and imprecision of both static and dynamic techniques. The most used techniques are based on latent semantic indexing (LSI) and scenario-based probabilistic ranking (SPR) (Poshyvanyk *et al.* 2007). LSI is an information retrieval method that analyses the relation between words and documents in bodies of text. On the other hand, SPR is a dynamic technique that analyses the link of the features in the source code and identifies entities of the program that implement specific features. However, the aforementioned weaknesses of both static and dynamic techniques (Sections 2.2.2 and 2.2.3) are still present in the hybrid tools that support the feature mapping activity.

2.2.5 The Differences of the Feature Mapping Techniques

As aforementioned, there are three types of techniques for feature mapping activity in the literature namely static, dynamic and hybrid techniques. Table 2.1 summarizes the main differences of these techniques according to seven criteria that we have just explained in previous sections, which are: support, types of features, target artifact, feature evolution, overlapping features, forward recovery of features' implementation elements and basis information. For instance, the static techniques offer a manual support for the mapping activity of any type of feature. Given the manual mapping activity, the detection and mapping of overlapping features have to be performed at the same way. For this reason, the mapping activity using static techniques tends to be error-prone and time-consuming. The static techniques rely on the source code and call-graph.

The dynamic techniques offer an automatic support and are only able to detect user-level features. However, the user-level features not always correspond to those ones that the developers need to investigate or analyse their evolution. In addition to the application's source code, the dynamic techniques also depend on a suite of quality test cases to activate the code elements of a given feature. In particular, the dynamic techniques are often unable to detect overlapping features. As can be observed in Table 2.1, none of the techniques provides support to the feature evolution mapping and the forward recovery of features' implementation elements.

Table 2.1: Differences among the Feature Mapping Techniques.

Criteria	**Static Techniques**	**Dynamic Techniques**	**Hybrid Techniques**
Support	Manual	Automatic	Manual and Automatic
Types of Features	Any	User-Level Features	Any
Target Artifact	Source code	Source code and Suite of Test Cases	Source code and Suite of Test Cases
Feature Evolution	No	No	No
Overlapping Features	Detected Manually	No	Detected Manually
Forward Recovery of Features' Implementation Elements	No	No	No
Basis Information	Call-graph	Program execution trace	Call-graph and Program execution trace

2.3 Reengineering Methodologies and Techniques

Knodel *et al.* (Knodel 2005) proposed a generic process for integrating and incorporating existing assets into an asset base of a Software Product Line (SPL) infrastructure. The goal of this work is to allow organizations to reuse existing assets in their own asset base rather than creating assets with similar functionalities. In addition, the authors highlight the need of maintaining the asset's internal quality. However, this process is very general and the proposed recovery process is based on a set of manual activities. Moreover, a tool is not presented to automate this process and even a detailed evaluation is not carried out. This work only focuses on asset reuse and is not interested in features that can impact on a set of elements or modules of a given asset. The authors mention that feature location techniques (Section 2.2) can be integrated into their asset recovery and incorporation process, but they do not mention how this can be done. They also mention the use of historical analyses to capture change and bug dependencies from bug tracking systems as they are used as input of their asset incorporation process. The goal is to determine couplings among files in order to evaluate the feasibility and effort of incorporating an asset into the asset base. However, this approach is not presented in detail and no example is provided. This work is very different from ours because we are interested in recovering the implementation elements of evolving program families' features. In addition, we exploit the evolution history of a program family to improve the feature recovery process (Chapter 5). On the other hand, they are more interested in defining a high-level process to incorporate existing

assets into the SPL infrastructure. In our work, we are interesting in defining history-sensitive heuristics for recovering features of evolving program families in a semi-automatic way (Chapter 5).

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 one. SCHOLIA represents a runtime object call graph during the architecture extraction. In order to reach the extraction, it is possible to annotate the code to define the system's architecture. This technique is concerned about comparing how the code is in conformance with the intended architecture in order to avoid architectural violations in the code. There are other studies that extract information from existing system implementations and reason about this information with the intention of reconstructing the original software architecture (Kazman and Carrière 1999, Krikhaar 2009). However, this thesis is different as it is centered on identifying and classifying the code elements realizing each feature in evolving program families. Additionally, it focuses on analysing the evolution of the program family's source code rather than its high-level architecture.

Kang *et al.* (Kang *et al.* 2006) reported the experience of re-engineering the credit card authorization systems (CAS) components. The main goal of the re-reengineering process was to improve the reusability of components. They described a set of principles for developing the re-engineering process. Kang *et al.* (Kang *et al.* 2005) also presented a re-engineering process of a legacy system into product line assets. They reported how the information was extracted from a legacy system. It involves: (i) the reverse engineering process; (ii) the identification of operation units from the object relationships by analysing the method invocations and data flows. Basically, these operations are classified into three categories: sensor, controller and actuator; and finally, (iii) the recovery of conceptual architecture and process architecture. Our work is different because we forward recover the features' implementation elements. We are not interested in components' reusability but in identifying and classifying the implementation elements in the source code realizing each feature in evolving program families.

Arango (Arango 1998) proposed a domain-engineering framework for the generation of reuse infrastructures. Basically, this work focused on defining a set of principles at the domain analysis level with the goal of providing software reuse. His approach is based on a set of systematic methods that

try to maximize the system reuse through the evolution of a model of the problem domain. He characterized a conceptual model of reuse and formalized it using first order logic. Additionally, the author also adapted a high-level recovery model with the goal of identifying information missing from the domain model. However, this work is different from ours because (i) it does not provide heuristics to deal with the forward recovery of features at the level of source code; (ii) it does not deal with evolving program families; and finally (iii) it only focuses on software reuse.

Tulio *et al.* (Tulio *et al.* 2012) proposed a semi-automatic approach for extracting a software product line (SPL) from legacy systems. This work uses the CIDE tool to associate colours to the lines of code that implement a feature. The goal of this work is to extract optional features based on a set of feature seeds. The algorithm proposed in this paper is a fixed-point algorithm with two phases. The first phase is called color propagation where the program elements that reference the seeds are visually annotated with a color reported by the developer. The second phase is called color expansion where the algorithm checks whether a color can be expanded to its enclosing lexical context. Our goal is different because we analysed program families and their evolution histories in order to support the forward recovery of the code elements realizing any type of feature.

There are many studies that extract information from the application's source code and generate pattern descriptions (Kramer and Prechelt 1996, Shi and Olsson 2006, Tsantalis *et al.* 2006, Alnusair and Zhao 2010). For example, Shi and Olsson (Shi and Olsson 2006) use control-flow graphs (CFG) for representing patterns. To do this, this work relies on processing the Abstract Syntax Tree (AST) of methods in order to build a CFG for program elements. After this construction, the control-flow graphs are analysed in order to verify restrictions related to a given design pattern. However, all of these studies are different from ours because they are concerned about extracting information of design patterns. On the other hand, our research work is centered on forward recovering implementation elements of a feature when analysing evolving program families.

2.4 Source Code Refactoring and History Analysis

This subsection briefly describes how some previous research work explored code refactorings and history analysis. Their purpose was often to propose refactoring techniques to extract SPLs, detect change dependencies through the use of APIs and frameworks, crosscutting features, and character-

istics of the software evolution. However, all of them are different from ours. Our goal is to identify and classify the feature's code elements of a evolving program family based on its complete evolution history rather than a single program version or a single horizontal program history.

Several research work have proposed and evaluated refactoring methods for extracting SPLs from legacy systems (Alves *et al.* 2005, Liu *et al.* 2006). These methods rely on the refactoring of features and evaluation of specific languages for managing variability in SPL (Kästner *et al.* 2007, Liebig *et al.* 2010). Others have explored the software historical information for different purposes, such as: (i) detecting code anomalies, for example, related to improper design decisions (Fowler 1999, Kazman and Carrière 1999, Ratiu *et al.* 2004); (ii) capturing code changes in order to analyse their impact on software quality and maintenance (Herzig 2010); and finally (iii) analysing dependencies of the code changes for detecting types of change patterns (Fluri *et al.* 2008, Alam *et al.* 2009, Wu *et al.* 2010).

Adams *et al.* (Adams *et al.* 2010) presented a feature mining technique named COMMIT that analyses the source code history to statistically cluster functions, variables, types, and macros that have been changed together intentionally. The authors compared this technique to others, called HAM and CBFA. The COMMIT technique is different from the others due to the following reasons: (i) it deals with variations of a feature in terms of its implementation elements which are often implemented by partially cloned code; (ii) it identifies variables and types related to a feature; and (iii) finally, it is able to identify composite features which are composed of multiple sub-features. The COMMIT workflow is based on all the lines of code that were added, removed or modified by a developer. This technique has shown to be more effective only to identify non-functional crosscutting features. A crosscutting feature is a feature in which its implementation is scattered across multiple modules (Adams *et al.* 2010). This means that the implementation of a crosscutting feature is not modular and cuts across the boundaries of many classes and methods. However, many other features of a program family are often domain-specific and do not exert a widely-scoped crosscutting impact on the modular structure (Alves *et al.* 2005, Figueiredo *et al.* 2008).

Nguyen *et al.* (Nguyen *et al.* 2011) proposed a tool, called XScan, for identifying, ranking, and recommending concern containers. Concern containers are defined as code units sharing a crosscutting feature. This tool identifies and recommends top-ranked groups that share some crosscutting features in both evolving aspect-oriented (AO) and non-AO programs. These groups are detected based on the similarity of interaction contexts related to two or more

method calls. Nguyen *et al.* defined a set of formulations and algorithms for recommending these concern containers. Basically, these formulations are described in terms of interaction similarity at different levels, such as internal and external similarities of the callees and callers of a given program method. This work only analyses non-functional crosscutting features, which are also defined as not modular and scattered across multiple modules in the application.

Dagenais *et al.* (Dagenais *et al.* 2007) defined a technique to infer structural patterns to be checked as a software system's source code evolves. The goal of this work is to make descriptions of a feature implementation in order to check if its structural characteristics are valid in future versions of the application's source code. Seven categories of structural characteristics are used: `callersOf()`, `calledBy()`, `accessorsOf()`, `accessedBy()`, `overrides()`, `implements()`, `declaredBy()`. The usefulness of this technique is tied to the characteristics of the feature being analysed. This occurs because the authors assume that the implementation elements that realize a high-level feature present common characteristics to infer patterns to be checked in an evolving software system. The technique proposed by Dagenais *et al.* is different from our for several reasons: (i) it only tries to leverage structural patterns that may exist among the elements in a mapping; (ii) it does not consider the evolution history analysis of program families; and finally (iii) it does not take into consideration the program family's change history in order to generate the feature mappings according to the current version of the family member.

Xue *et al.* (Xue *et al.* 2010) described a method for assisting developers to detect changes in product feature models. This method is based on a set of feature models that refer to each family product. These feature models are compared to each other by using a tool named GenericDiff, which is based on both dependencies and relationships. Feature models of different products are analysed to determine, for instance: (i) if the same features contain different names, and (ii) if the same features belong to different composite features. Xue *et al.* analysed the evolution of the requirements in terms of feature models. However, this work is very different from ours because Xue *et al.* (Xue *et al.* 2010) are concerned about detecting differences in feature model from different products rather than analysing evolving program family's source code and recovering the implementation elements realizing the family's features.

2.5 Clone Detection Techniques

Cloning is known as a phenomenon found in many software systems and its existence tends to harm maintenance tasks (Mayrand *et al.* 1996). Some research work has explored the detection and analysis of similar code fragments, the so-called *simple clones* (Johnson 1994, Baxter *et al.* 1998, Kamiya *et al.* 2002, Basit and Jarzabek 2005, Kim *et al.* 2005, Nguyen *et al.* 2009, Basit and Jarzabek 2009). Basically, they use different techniques, such as abstract syntax tree, plain text, tokens, and others. For example, Basit and Jarzabek (Basit and Jarzabek 2005, Basit and Jarzabek 2009) described a strategy for detecting design-level similarity patterns that relies on a token based approach. In order to reach this goal, the authors formulated heuristics to identify clone patterns and used data mining techniques to infer some properties, such as detecting structural clones. Structural clones are part of bigger replicated program structures.

Clone detection techniques are also used as a way to determine similar code that could be factored out into aspects (Bruntink *et al.* 2004, Bruntink *et al.* 2005). Bruntink *et al.* (Bruntink *et al.* 2004, Bruntink *et al.* 2005) assessed the identification of crosscutting features through different clone detection techniques. This occurs because the crosscutting features' code is not well modularized and is typically duplicated throughout the software system. The clone tools and techniques are only focused on detecting similar pieces of source code when comparing several files of a software system. Indeed, they can be used in a complementary way in our work. However, it is not trivial to use directly these clone detection techniques when recovering feature's implementation elements in evolving program families for two main reasons. First, these techniques do not implement the feature concept; i.e. they do not provide information about which implementation elements realize a given feature. Second, these techniques only compare files of a software system without the intention of classifying the implementation elements. Therefore, we go one step further because we explore the evolution history analysis in order to forward recover the features' implementation elements of program families. In other words, we take into consideration a more fine-grained perspective and reason to recover implementation elements realizing a feature in evolving program families.

2.6 Summary

This chapter presented the main concepts addressed in this thesis. It also presented an overview of existing studies and a critical discussion of their limitations. Section 2.1 presented the definitions of the main terms discussed throughout this research work, such as program family, family member, features. Section 2.2 presented the definition of feature mapping activity and other similar terms commonly used in the literature, which are: concept assignment problem, feature location and traceability. It also presented the most known techniques and tools responsible for supporting the feature mapping activity. In fact, there is a variety of techniques and tools that help developers map the implementation elements to features (Sections 2.2.2 to 2.2.4). The feature mapping is seen as a relevant information source for improving the maintenance and evolution tasks of software systems and evolving program families. This occurs because these mappings provide developers with a full knowledge of a software system in terms of the implementation elements realizing its features. However, all these existing techniques that support the feature mapping activity do not consider the knowledge of the program family's change history in order to observe how the features' implementation elements and their relationships have evolved over time.

Section 2.3 reported some studies that proposed methodologies and tools for recovering the original design of a software system (e.g. architecture) and extracting software product lines from a legacy system. However, there is no work that explores program families and their evolution history with the intent of forward recovering implementation elements realizing the features. Existing research work are limited as only support the feature mapping activity and consider the history of a single application. Also, they do not address the forward recovery of elements realizing each feature. Section 2.4 detailed studies that analyse the source code history with different purposes, such as proposing refactoring methods, detecting crosscutting features and dependencies. Finally, Section 2.5 presented some research work that explores the detection and analysis of similar code fragments. Despite all the efforts, none of related work discussed in this chapter provides any type of support for the forward recovery of features' implementation elements in evolving program families.