4
Mapping Expansion Heuristics

As discussed in Chapters 2 and 3, the feature mapping activity is one of the critical factors to maintain and evolve software systems successfully. The feature mappings are especially important when developers have to evolve or maintain program families (Chapter 3). However, many factors make feature mapping in evolving program families much harder than in single applications. First, the feature mappings performed by developers or conventional mapping tools may contain undesirable mismatches (Chapter 3). Second, program families often depart from a framework and evolve to different systems to accommodate various customer-specific requirements (Weiss and Lai 1999). Therefore, the feature mapping of the original system may have been changed for each family member. This tends to occur when the realization of the same feature evolved differently for each family member (Chapter 1). Consequently, it is primordial that developers identify and contrast the different feature mappings in each family member; they need to fully understand when and where the sets of implementation elements for each family feature start to diverge across the change history of family members. Third, the feature mappings per family member should be produced with high accuracy i.e., the feature mappings should contain the minimum number of mismatches or in a better case not contain them.

Chapter 2 presented many techniques and tools to assist developers in feature mapping activity. However, most of them only take into consideration each individual version of a single program; the change histories of those single applications are simply discarded. They do not consider how the realizations of the features have evolved over time in a program family. These existing techniques are limited because they provide developers with the knowledge of the implementation elements without considering the differences between the family members.

This chapter describes a cohesive suite of five heuristics that aim at expanding feature mappings in evolving program families. This chapter answers the second and fourth research questions of this thesis (RQ2 and RQ4 in Section 1.3). The mapping heuristics rely on the previously defined list of mapping

mismatches (Chapter 3). The mapping expansion refers to the action of automatically generating the feature mappings for each family member version by systematically considering its previous change history. Section 4.1 motivates the feature mapping expansion problem in evolving program families through a concrete example. Section 4.2 compares the idea and goal of the mapping expansion heuristics with closest related work. Section 4.3 discusses a heuristic method that takes into consideration the program family's change history. The suite of the mapping heuristics is described and formalized in Section 4.4. Section 4.5 presents the MapHist tool, that supports the use of the proposed heuristics. Section 4.6 shows the evaluation results of the mapping heuristics by means of two evolving program families. Section 4.7 identifies the threats to validity of this evaluation. The mappings generated by the proposed heuristics play a key role in a variety of maintenance tasks involving the evolving program family's source code. For this reason, Section 4.8 describes how the mapping heuristics were integrated with a visualization tool in order to assist developers in these tasks through graphical perspectives. Finally, Section 4.9 summarizes the chapter.

4.1 Motivating Example

Figure 4.1 illustrates how the feature mapping is performed when using conventional techniques for evolving applications (Chapter 2). This example is based on the OC program family that is used in the evaluation process of the heuristics (Section 4.6.1). For the sake of simplification, we selected two family applications, *Application I* and *II* and two of their versions. Figure 4.1 shows a piece of code of the feature *Scenario* in *Applications I* and *II* (Section 3.1.1). This figure illustrates the evolution of the `MainDesktop` class in versions 1 and 2 of *Applications I* and *II*. In the first versions of *Applications I* and *II* the developer mapped the `checkUnsavedScenarium()` method as being responsible for realizing the feature *Scenario* (see Feature Mapping - Version 1 of *Applications I* and *II* in the right side of Figure 4.1). However, the class attribute named `scenarioInfo` has not been mapped to the feature *Scenario*, which characterizes a mismatch (false negative) in feature mapping. Additionally, this happens because developers tend to focus mainly on the behavior implemented by classes and methods (Chapter 3). In the second version of *Application I*, the `MainDesktop` class was modified to extend a class named `BasicDesktop`, which has not been mapped to the feature *Scenario*. In addition to this change, in the second version of *Application II*, the `propertiesAction` attribute and the `changeDesktop()` method are added

to the `MainDesktop` class.

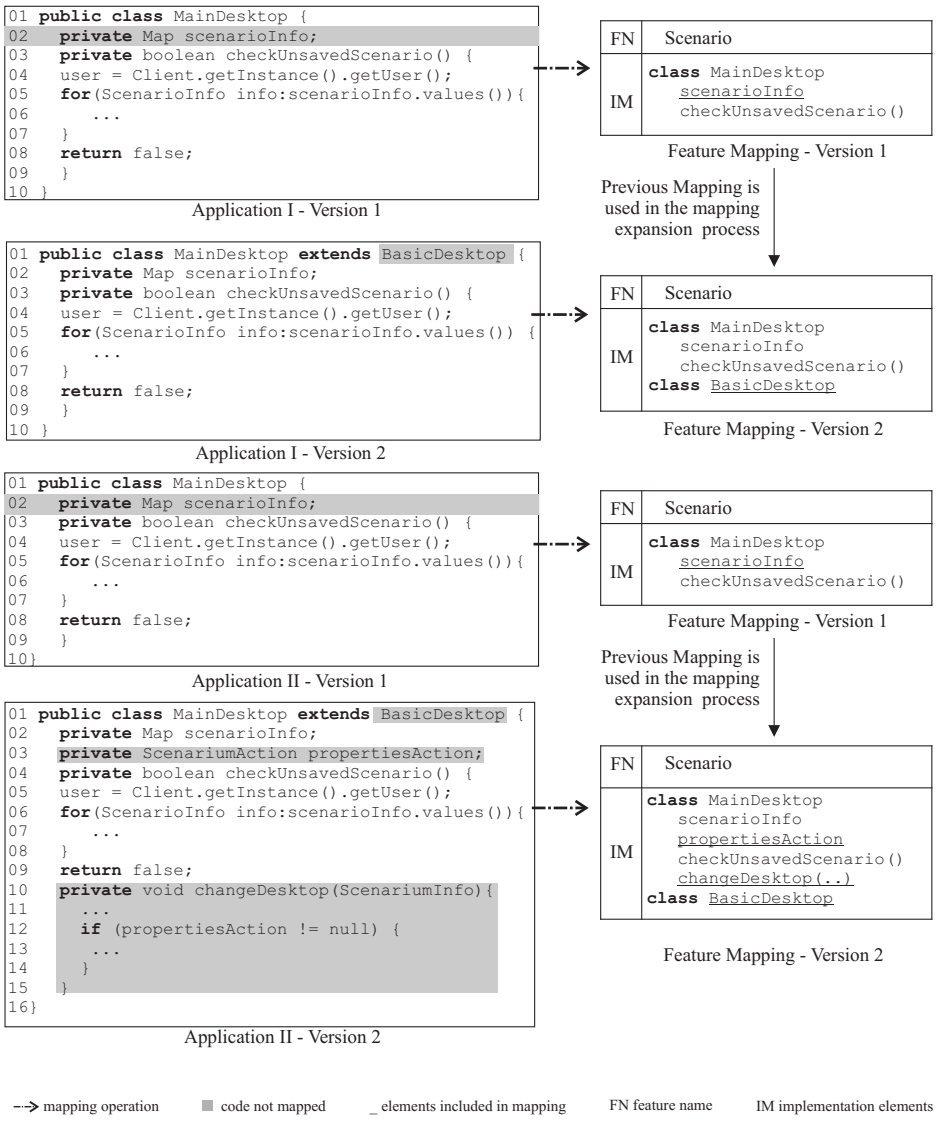

Figure 4.1: Piece of Code of the Feature Scenario in Applications I and II.

Figure 4.1 shows how the described evolution scenarios may lead to mapping mismatches in conventional feature mapping techniques. There are two issues to be taken into consideration: (i) the occurrence of mapping mismatches, such as the `scenarioInfo` and `propertiesAction` attributes and the `changeDesktop()` method, and (ii) the evolution of the `MainDesktop` class in different ways in *Application I and II*. The first issue is not addressed by most of the feature mapping techniques. This occurs because they rely on the interaction between developers and tools or the use, extension and tuning of pre-existing test suites (Chapter 2). They assume that developers should perform the mappings by themselves or should have available test suites to exercise and map the features' implementation elements (Section 2.2). However, these test suites are usually not accurate and do not provide

a good level of coverage. As aforementioned, this problem can become even more cumbersome when dealing with evolving program families. The second issue refers to the different changes that the feature's implementation elements have undergone over time in the family members. Existing techniques do not progressively consider the change history, the evolution of the `MainDesktop` class from version 1 to version 2 in *Applications I and II*.

Even running existing tools that support the feature mapping activity (Chapter 2) for every family application version, this leads to the production of each feature mapping from scratch. These tools do not consider the previous version of the feature mapping as the basis. Hence, this strategy is also likely to be ineffective: elements identified in the previous feature mapping might be missed in the new mapping. These mapping mismatches (i.e. false negatives) can occur, for instance, due to minor, albeit typical, structural changes in the current family application version. A few cases of recently-proposed techniques consider the change history of a particular program to support the feature mapping activity (Adams *et al.* 2010, Nguyen *et al.* 2011). However, they do not incrementally analyse the change history of the features' implementation elements.

Feature mapping expansion involves the automatic identification of feature elements in the code. It is performed by analysing and comparing each family application's history in order to automatically produce the feature mappings for all the versions of the program family. The comparison between the versions needs to be performed in order to find out what implementation elements were removed, added or modified when analysing the family applications' history (Doar 2007, Kawrykow and Robillard 2011).

As aforementioned, the expansion process should analyse the change history of the family applications in order to generate the feature mappings and reduce their mismatches. According to the example illustrated in Figure 4.1, the mapping expansion process includes the `scenarioInfo` attribute in feature mapping (see Feature Mapping - Version 1 of *Applications I* and *II*). The previous feature mapping (Version 1), the list of changes and the source code of the version are used to generate the feature mapping of the subsequent version (Version 2). The `MainDesktop` class, which has already been mapped, was modified to extend a class named `BasicDesktop` and new attributes and methods were added to it. The expansion process considers these changes in order to automatically generate the mappings of the different applications' versions. The expansion process includes the `BasicDesktop` class (see Feature Mapping - Version 2 of *Applications I* and *II*). The `BasicDesktop` class also realizes the feature *Scenario* as it contains methods that are used

in the `MainDesktop` class. Additionally, it includes the `propertiesAction` attribute and the `changeDesktop()` method (see Feature Mapping - Version 2 of *Applications II*). In this way, the mapping expansion process is able to deal with missing and incorrect elements mapped to a feature in an evolving program family when using conventional feature mapping techniques (Chapter 2).

4.2 Existing Limitations on Feature Mapping Expansion

This section aims at revisiting related work discussed in previous chapters with the goal of emphasizing their limitations on feature mapping expansion. We categorize related work into two groups: feature mapping techniques (Section 4.2.1) and approaches that explore the source code history (Section 4.2.2). None of related work discussed below explicitly support feature mappings in evolving program families. Nevertheless, we also highlight other relevant aspects that distinguish or complement our mapping expansion heuristics with respect to existing work.

4.2.1 Feature Mapping Techniques

As aforementioned in Section 2.2, there are many techniques and tools that support the feature mapping activity. However, all these existing techniques 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. Even though it is possible to run these existing tools for each application version individually, the influence of a feature mapping in a particular version of the family member can lead to other potential elements being missed or incorrectly detected in future versions of the same member. This means that feature mappings in evolving program families are more sensitive to mismatches (Chapter 3). Our mapping heuristics have the goal of expanding feature mappings in evolving program families by explicitly taking into consideration the program family's change history, and consequently reducing the number of potential missing and incorrect elements mapped to the features.

4.2.2 Source Code History

As presented in Section 2.4, some research work has explored the code history for identifying crosscutting features (Breu and Zimmermann 2006,

Adams *et al.* 2010, Nguyen *et al.* 2011). Adams *et al.* (Adams *et al.* 2010) presented a feature identification technique named COMMIT. This technique has shown to be more effective only to identify non-functional crosscutting features. A crosscutting feature is characterized in (Adams *et al.* 2010) as a feature in which its implementation is scattered across multiple modules. 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 (Figueiredo *et al.* 2008).

Nguyen *et al.* (Nguyen *et al.* 2011) proposed a tool, so-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 of code units that share crosscutting features in both evolving (non-)aspect-oriented programs. These groups are detected based on the similarity of interaction contexts related to two or more method calls. This work only analyses non-functional crosscutting features, which are also defined as not modular and scattered across multiple modules in the application. Breu and Zimmermann (Breu and Zimmermann 2006) introduced an aspect identification approach by analyzing how the fan-in number changes over time. This technique has the same goal of the approach proposed by Nguyen *et al.*

The similarities of these research work with ours are twofold: (i) they evaluated their feature identification techniques in evolving systems, and (ii) they identified implementation elements related to non-functional crosscutting features. These techniques are complementary to ours. Nevertheless, our goal is different as the proposed heuristics focus on the expansion of feature mappings in evolving program families. The proposed mapping heuristics consider the program families' change history and, therefore, are able to observe how the realizations of their features have evolved over time.

4.3 A Heuristic Method for Expanding Feature Mappings

This section provides an overview of the steps required for expanding feature mappings. First, we describe the multi-dimensional historical analysis adopted by all the mapping heuristics (Section 4.3.1). Second, the mapping expansion heuristics are based on the knowledge of the mapping mismatches described in Chapter 3. Finally, as the main goal of the heuristics is to expand feature mappings in evolving program families, a comparison strategy between the family versions is required. In particular, the process for expanding feature

mappings entails a list of elements that were changed, included and removed from one version to another. This list of changes is important to ensure that the heuristics are able to detect and update the features' implementation elements. The comparison strategy and mapping expansion process are described in Section 4.3.2.

4.3.1 Multi-dimensional History Analysis

The mapping expansion heuristics are based on a multi-dimensional historical analysis illustrated in Figure 4.2. The analysis takes into consideration both horizonal and vertical histories. The vertical history consists of a set of applications that belong to a program family. It increases when new members are developed from the program family core. The horizontal history refers to the evolution of each family member; it consists of the implementation elements that realize each feature (feature code) across the versions of a family member. These implementation elements have possibly been added, changed, or even removed in some member versions. They are represented by classes, methods and attributes in object-oriented systems. The vertical and horizontal histories represent the family members developed in different time points of the program family. The horizontal and vertical histories are similar to the terminology already proposed by Krueger (Krueger 2002), which deals with variation in both time and space.

For example, it considers a family application consisting of two versions (Version 1 and Version 2). Now, let's suppose that there are cases in Version 1 where a given feature is realized by one module. After that, this same module in Version 2 is changed to realize an evolution task in the same feature. This means that the feature was changed. This scenario is illustrated in Figure 4.2 where a feature was added in Version 1 (black box) and it was changed in Version 2 (grey box). More importantly, the horizontal histories of the same feature tend to be different across the family applications, as illustrated in Figure 4.2. A concrete example illustrating the horizontal differences of the feature *Export* across *Applications I* and *II* can be observed in Figure 1.2 (Section 1.1.2). This example shows the `ExportDialog` class in *Application I* and how it was modified in *Application II* to implement a change related to the feature *Export*. The mapping expansion heuristics rely on information generated from these two dimensions of the historical analysis. Figure 4.3 shows the methodology steps of the mapping expansion heuristics, and each of them are described as follows:

1. **Feature Selection**. A list of features to be analysed is required. This list

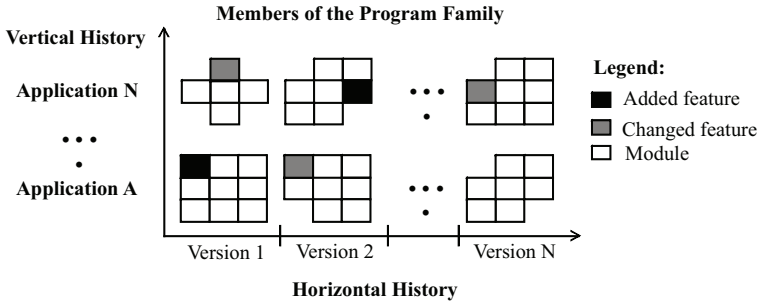

Figure 4.2: Multi-Dimensional History of a Program Family Evolution.

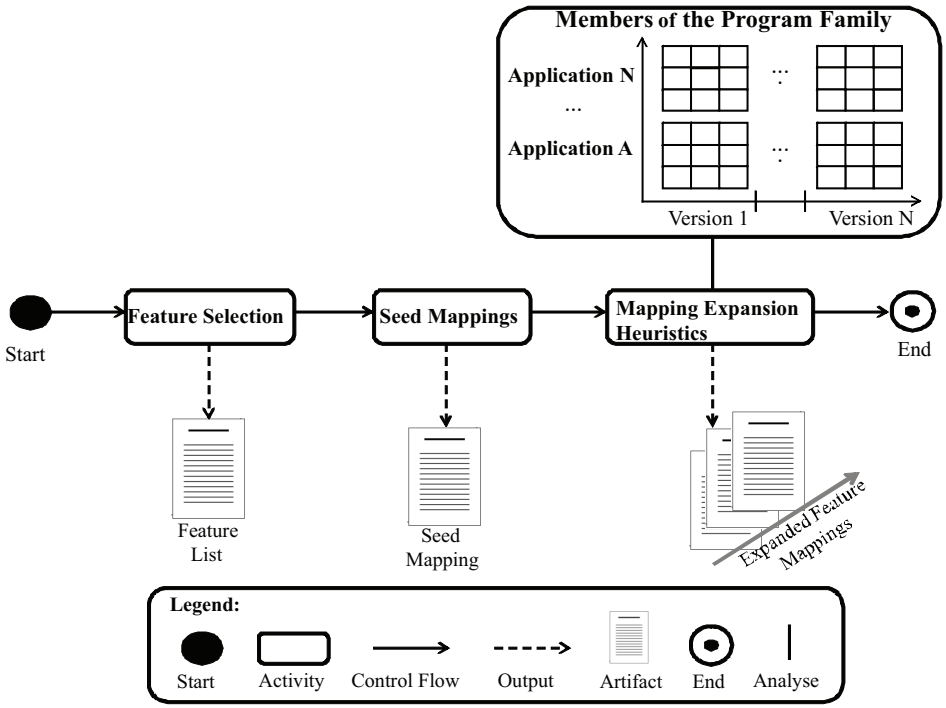

Figure 4.3: Methodology of the Mapping Expansion Heuristics.

of features must belong to one or more family members. The developer does not need to explicitly define if the chosen features are mandatory, optional or alternative (Kang *et al.* 1990). They do not need either to provide the feature relationships;

2. **Seed Mappings**. It is required to assign an initial set of elements responsible for realizing each selected feature (Seed Mappings in Step 1). The seed mapping is only required for the first version of each selected family member. This means that only one mapping file is required. Whether the developer selects features of many family members, it is required to provide all seed mappings of each first family member version as input;

3. **Mapping Expansion Heuristics**. The heuristics are responsible for

expanding feature mappings through the family applications. The expansion of feature mappings implies on the detection of mismatches. To this end, these heuristics use information about: (i) the seed mappings (Step 2), and (ii) the family application history analysis as illustrated in Figure 4.3.

The seed mappings are important in this expansion process because they provide starting points so that the heuristics can generate feature mappings with a high level of precision and recall. Nevertheless, the seed mappings can have varied sizes. This means that developers can provide seed mappings, for instance, with a set of three classes or more. This scenario is typically relevant as developers might not know all the implementation elements realizing a given feature in a family member. The reverse is also true: the presence of an initial feature mapping will also guide the heuristics to perform the feature mapping expansion. This is particularly important as different stakeholders have different perceptions about the elements that should pertain (or not) to the feature realization in the code. Additionally, the seed mappings might also contain mismatches made by developers and feature mapping tools (Chapter 3).

4.3.2 Comparison Strategy and Mapping Expansion Process

The comparison between the versions of the family applications is essential to obtain information about their change history. This change history helps to reduce the number of mismatches (Chapter 3) during the feature mapping expansion (Step 3 in Section 4.3.1). This information is related to the list of classes, methods and attributes that have been modified, added and removed from one version to another. The adopted comparison strategy is based on sequential analysis. If the number of versions provided for analysis is *V*, then the number of comparisons is *V - 1*. For example, if a family application has three versions being analysed (Versions 1, 2 and 3), then the number of comparisons is two (Versions 1-2 and Versions 2-3). This sequential analysis has also been used by other studies (Nguyen *et al.* 2009, Adams *et al.* 2010).

The mapping expansion process in evolving applications of the same program family widely depends on the precision of the version comparisons. The expansion process is the action of generating mappings regarding the features and their implementation elements of each family application version. As the mapping expansion process is performed for each version, it is required to observe the differences between the versions. The list of changes generated by the comparison strategy is used an input by the mapping expansion process.

This list of changes can be, for instance, refactoring of method or variable names, inclusion of lines of code in existing methods. The mapping expansion process analyses the list of changes in order to apply the following rules. First, implementation elements mapped to a feature in a previous application version and that were not removed or changed (e.g. new lines of code) in the next version, they are mapped to the same feature. Second, the removed elements are compared with the added ones. This comparison is required in order to know if these elements suffered, for example, a simple refactoring of their names. In this case, we compare the body of the removed method with the added one in order to verify if they are similar (e.g. using abstract syntax tree). If so, the added method is mapped to the same feature as they have the same body. Otherwise, it is not mapped. It is important to highlight that this added method can be mapped to other features.

4.4 Mapping Expansion Heuristics

This section describes the heuristics responsible for expanding feature mappings. Their intent is to improve the accuracy of those mappings by reducing the occurrence of mismatches given a set of evolving applications of the same family. The application of the heuristics follows a particular order (Section 4.4.1). The description of the heuristics is presented in terms of: (i) their strategy and rationale highlighting the novel ideas; (ii) the key part of their algorithmic solution; (iii) the differences and enhancements compared to existing work; (iv) their abstract representation through an illustrative example. The illustrative example comprises one family application and two versions in order to make easier the understanding of an evolving program family; and finally (v) their formalization that relies on set theory (Hrbacek and Jech 1944). The heuristics presented in Sections 4.4.2 and 4.4.4 are able to simultaneously detect more than one type of mismatch. For this reason, it was not needed to define one heuristic for each type of mismatch.

The following terms are defined and used in the formalization of the heuristics. Feature as *fe* in F(A), where F(A) represents the set of selected features of an application (A); M refers to all modules of an application; O refers to all methods of an application; attribute as a; method or operation as o; *M(fe)* refers to the set of modules (e.g. classes) that realize this feature *fe*; *O(fe)* is the set of operations (e.g. methods) of a given module M that realizes this feature *fe*; n is the number of family applications and v refers to their versions.

4.4.1 Execution Order of the Heuristics

The heuristics are executed following a certain order during the mapping expansion process. This execution order was aimed at generating mappings with a reduced number of mismatches. Moreover, when the feature mappings are updated and expanded by a given heuristic, the other heuristics use them naturally to keep on the expansion process. This execution order was established after deep analysis of how the number of mismatches could be reduced during the expansion mapping process. Additionally, we have observed how the heuristics could be better explored through the output provided by the previous heuristics. In the next sections it is explained in detail why this execution order is essential. Also, it is illustrated how this order is effective through a real example derived from the OC program family (Section 4.6.3). The execution order of the heuristics is described in Algorithm 1. Algorithm 1 manages the historical information (program family versions) to be executed by each heuristic. It is defined as follows: detecting omitted feature partitions - DFP (Section 4.4.2); detecting code clone mismatches - DCC (Section 4.4.3); detecting interfaces and super-classes - DIS (Section 4.4.4), detecting communicative feature mismatches - DCF (Section 4.4.5); and detecting omitted attributes - DOA (Section 4.4.6). This main algorithm is responsible for calling the respective heuristics taking into account the version of the program family to be analysed. In the next sections we will describe these respective heuristics.

Algorithm 1 processMappingExpansion(Versions) - Main Algorithm

```
1. for version in Versions do
2.     detectFeaturePartitions(version) //Heuristic DFP
3.     detectCodeClones(version) //Heuristic DCC
4.     detectInterfacesClasses(version) //Heuristic DIS
5.     detectCommunicativeFeatures(version) //Heuristic DCF
6.     detectOmittedAttributes(version) //Heuristic DOA
7. end for
```

4.4.2 Detecting Omitted Feature Partitions

Strategy and Rationale. The first heuristic is named DFP (Detecting Omitted Feature Partitions) and it is responsible for detecting methods comprising feature partitions that have not been mapped. Its goal is to circumvent occurrences of the mapping mismatch related to *multi-partition features* (Section 3.2.1). As these feature partitions correspond to entire methods, DFP can also detect occurrences of the mapping mismatches related to feature overlap-

ping and deficient module structure and documentation (Section 3.2.2). This occurs because these mismatches can also refer to entire methods realizing a feature. They are related as the occurrence of a mismatch can directly or indirectly imply another one, and vice-versa (Section 3.3). Given a specific feature, DFP analyses the already mapped methods by comparing them with the non-mapped ones in terms of *interaction similarity* (Nguyen *et al.* 2011). Interaction similarity is defined in terms of method interactions regarding its callees and callers in similar contexts (Nguyen *et al.* 2011). This means that two methods call or are called by similar ones. Figure 4.4 illustrates an example of two methods (`mC1` in the `Client1` class and `mC2` in the `Client2` class) with interaction similarity. They call the same methods (*mA* and *mB* in the `Server` class) in their body in a given interaction context. The interaction context is defined by the method calls before and after the calls to *mA* and *mB* in the `mC1` and `mC2` methods (Figure 4.4). Through the interaction similarity, DFP is able to capture feature partitions (methods) that do not contain explicit references to other ones.

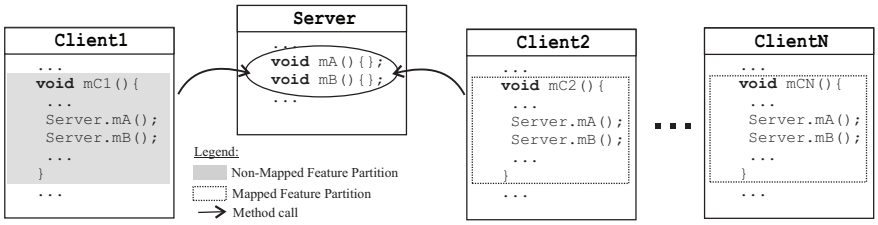

Figure 4.4: Example of Interaction Similarity.

We adapted the similarity strategy proposed by Nguyen *et al.* (Nguyen *et al.* 2011) by also considering information provided for both feature mapping and multi-dimensional history analysis (Section 4.3.1). The use of feature mappings was needed to consider all types of features. The original heuristic proposed by Nguyen *et al.* (Nguyen *et al.* 2011) is only able to capture implementation elements that realize features that are scattered across multiple modules. At large only non-functional features have this characteristic (Adams *et al.* 2010). As a consequence, a set of functional features, which are not scattered, is not captured. In other words, this means that functional features are implemented through only one or few modules (Figueiredo *et al.* 2009). Thus, DFP is a further development of the original heuristic because it is able to capture elements that implement any types of features regardless of whether they are widely scattered across multiple modules or concentrated on only one. This is possible because DFP relies on the seed mappings, which contain a list of features and the corresponding initial set of implementation elements that realize those features. For instance,

Figure 4.5 illustrates a widely scattered feature and a non-scattered feature. The `m1()` method in the `X` class realizes a widely scattered feature and it is called by many classes. These classes are called in a similar method with similar context, which is the method `mA()` in the `A` class. As a consequence, the `m1()` method is captured by the original heuristic. On the other hand, the `m2()` and `m4()` methods realize a non-scattered feature and call the same `m1()` method in the `X` class. As a consequence, these implementation elements are only captured by DFP.

The multi-dimensional history analysis is essential to observe how the interactions of the mapped and non-mapped methods evolved throughout the family applications' history. The historical analysis is essential for identifying methods with evolving interaction similarities and detecting such methods during the mapping expansion process. As part of this historical analysis, DFP also takes into consideration the methods that were modified and added with the goal of improving the accuracy of feature mappings (Section 4.3.2). During the family application evolution, for instance, new methods might call previously-mapped methods. As a result, the interaction similarity usually changes over time. Therefore, DFP checks if the new method and the mapped method have the same interaction similarity. If so, the feature mapping of the family application version is updated by the inclusion of this new method. The benefits of analysing the program family history are related to coverage and consistency of the feature mappings in each family version (Sections 4.3.1 and 4.3.2).

Algorithmic Solution. The essence of the algorithmic solution implemented by DFP is presented in Algorithm 2. We extended in our implementation the idea presented by Nguyen *et al.* (Nguyen *et al.* 2011) and considered the interaction similarity of the features' implementation elements during the program family evolution. The original algorithm was extended by DFP to take into consideration the seed mappings and the analysis of family application's methods under the perspective of different features. Now, the interaction similarity is realized by comparing the mapped methods, those that realize a given feature, with non-mapped ones. Thus, it is possible to contrast such similarities with other elements contained in the seed mappings in order to verify which other features can be realized by the non-mapped methods. As a result, DFP is able to update the whole mapping by adding methods that realize one or more features. Our assumption is that *methods having similar interactions in a family application tend to implement the same features*. The similarity strategy defined by Nguyen *et al.* (Nguyen *et al.* 2011) states that the crosscutting features have the same behavior and/or they are similarly processed. Also, it

contains thresholds for determining the similarity weights. These thresholds are defined to avoid comparing all the methods of the family application and harming the scalability of the algorithm. We also consider the same strategy for all the types of features because we believe that methods that realize the same feature tend to communicate with a set of similar classes. This means that they call similar methods or implement the same interfaces or abstract classes.

DFP compares the mapped methods (line 02) with the non-mapped methods of each family application version (line 03) with the goal of finding groups of similar methods that are not scattered and implement non-crosscutting features (line 04). These groups of methods are run for each version during the family applications' history and they are stored in a list named *similarMethodGroups* (line 04). After that, it is checked if the identified groups have been mapped as well. To do this, DFP analyses all the features (line 01) and their methods with the goal of adding the similar methods to the mapping considering the analysed feature (lines 05-06). Therefore, during the expansion process of mappings DFP is able to: (i) detect new interaction similarities, and (ii) update or remove existing ones because either mapped methods were removed or modified (line 07).

Algorithm 2 detectFeaturePartitions(version) - DFP Algorithm

1. $featureList \Leftarrow allFeatures(version)$
2. $mappedMethodList \Leftarrow mappedMethods(version)$
3. $nonMappedMethodList \Leftarrow nonMappedMethods(version)$
4. $similarMethodGroups \Leftarrow$
5. $findHistoryMethodGroups(mappedMethodList, nonMappedMethodList)$ *//it relies on the strategy for comparing interaction similarity*
6. **for** feature in featureList **do**
7. $addSimilarMethods(feature, similarMethodGroups)$
8. $updateSimilarMethods(feature)$
9. **end for**

Comparison to Existing Work. Dynamic and static analysis techniques do not address the idea proposed by DFP with respect to identify similar methods by considering the feature mappings and the program family history (Section 4.2.1). Regarding the similarity strategy proposed by Nguyen *et al.* (Nguyen *et al.* 2011), it was used in its essence to recommend or update methods in charge of implementing the same non-functional crosscutting features.

Illustrative Example. Figure 4.5 illustrates the abstract representation of how DFP works for non-scattered features. In this figure there is a method named `m2()` in the `X` class that has been mapped to the feature *fe* in the first version of the family application. It is observed in the second version of the

family application that the `m2()` method in the `X` class was modified. Now, it calls the `m1()` method. This means that its interaction similarity was changed. In addition, it is observed that the `m4()` method in the `Y` class also calls the `m1()` method in the second version. Thus, when DFP analyses the interaction similarities of the `m2()` and `m4()` methods in both family application versions and initial feature mapping, it observes that these methods are likely to realize the same feature. As a consequence, it adds them to the set of elements realizing the feature *fe* (Figure 4.5).

Formalization. The formal definition of DFP is defined in terms of number of family applications (n), the family application versions (v), the methods (o), the features (fe) and the relations (Formula 4-1). In the formalization of DFP it was defined the *partition(O,O(fe))* relation that indicates all mapped methods to (fe) are compared with all the methods of the family application. The goal is to discover if there are methods mapped to (fe) that are similar to others, which are likely to realize (fe) and have not been mapped, by using the similarity strategy. According to the illustrative example shown in Figure 4.5, we have: the number of applications (n = 1), the number of versions (v = 2), the methods to be analysed (`m1()`, `m2()` and `m4()`) and the feature *fe*. The DFP formalization is defined in the following way.

$$DFP(fe) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{v} \{o : o \notin O(fe) \wedge \exists o_j \in O(fe) \wedge (o, o_j) \in partition(O, O(fe))\} \quad (4\text{-}1)$$

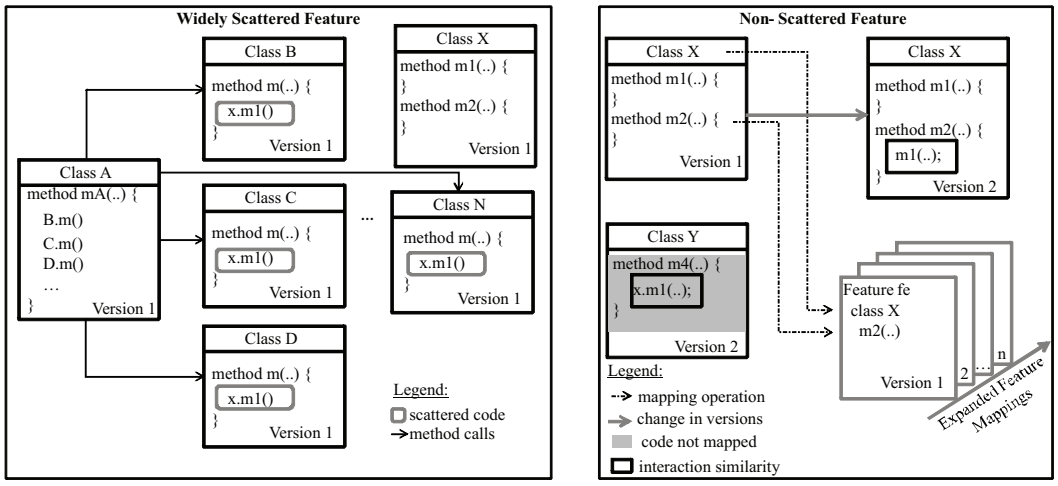

Figure 4.5: Abstract Representation of the Original Heuristic and DFP.

4.4.3

Detecting Code Clone Mismatches

Strategy and Rationale. This heuristic named DCC (Detecting Code Clone Mismatches) is responsible for detecting similar pieces of code that have not been mapped to a particular feature. Its goal is to detect the occurrences of the mapping mismatch related to *code clones* (Section 3.2.1). This heuristic analyses the entire family application version in order to detect all existing clones. After that, for each of the detected code clones, it checks if they have not been mapped but are similar to already mapped methods. If so, the code clones are mapped to all features (one or more) realized by the mapped methods. This occurs when a given mapped method is realizing more than one feature at the same time. In addition, DCC also analyses the history of the family applications with the goal of keeping track of the changes that the mapped methods have undergone and thus updating their clones. To this end, it verifies if the mapped methods and their respective clones, detected in previous versions, were modified in the next ones. The goal of DCC is to maintain the list of code clones realizing each feature. Thus, it can analyse all the features' implementation elements and know which clones are related to more than one feature. We have used the tree edit scripting algorithm proposed by Nguyen *et al.* (Nguyen *et al.* 2009) that uses the abstract syntax tree to represent the program source code and its clones. Basically, it calculates the structural similarity of two implementation elements through the distance of their structural characteristics. Thus, two implementation elements are considered clones if they have similar structure. Therefore, the novel idea provided by DCC is the use of clone detection when analysing the feature mappings and historical information of family applications.

Algorithmic Solution. Algorithm 3 presents the essence of the solution implemented by DCC. As there is a large number of clone detection tools in the literature, DCC can be jointly used with any existing tool (line 02). In the current version of the algorithm, we have used the tree edit scripting algorithm proposed by Nguyen *et al.* (Nguyen *et al.* 2009) to detect the code clones. After that, it verifies if the detected code clones are similar to already mapped methods (line 04-06). Finally, it adds all the clones to the features in which the mapped methods are related to (line 07-09). During the expansion process the family application versions are compared in order to observe if the mapped methods and their clones have been modified during each family application history.

Comparison to Existing Work. The heuristic DFP (Section 4.4.2), which adapted the similarity strategy proposed in (Nguyen *et al.* 2011), also uses clone detection to capture similar portions of code in a program. Apart

Algorithm 3 detectCodeClones(version) - DCC Algorithm

```
1. featureList ⇐ allFeatures(version)
2. clones ⇐ runCloneDetection(version)
3. elementList ⇐ mappedElements(version)
4. for element in elementList do
5.     cloneElements ⇐ clonesMappedElements(clones)
6. end for
7. for feature in featureList do
8.     mapAllCloneElements(feature, cloneElements)
9. end for
```

from (Nguyen *et al.* 2011), the dynamic, static and hybrid techniques for feature mapping activity (Section 4.2.1) have not considered detection of code clones in their essence. Although the heuristic DFP is able to deal with code clones, we decided to define a heuristic to maintain the list of clones associated with each feature. We made this choice because the detection of clones performed by DFP is run in the algorithm for calculating the similarity interaction. The heuristic DCC is able to capture all the clones of a family application and present them to the developers. Thus, if the developers need to observe only the clones of the features in an family application version, then it is possible when running the heuristic DCC. Another advantage is that DCC is decoupled of any existing clone tool. Hence, developers can use any tool and define their respective thresholds for detecting the clone codes. Consequently, it can also improve the individual performance of the heuristic DFP by using other clone tools (Section 4.4.2). The goal of DCC is to facilitate the maintenance tasks since the developer can observe the amount of similar methods which are used for implementing a given feature. In addition, our proposal of expanding the mappings by analysing the historical information and taking into consideration the change propagation of the family applications are the novelty when compared to (Nguyen *et al.* 2011) and existing work (Section 4.2.1).

Illustrative Example. Figure 4.6 shows an abstract representation of DCC. This figure illustrates a `m1()` method in the `X` class that has been mapped to feature *fe*. In the second version there is a `m2()` method in the `Y` class, which is similar to the `m1()` method, that has not been mapped. Consequently, DCC detects that these methods are clones. Hence, as the `m1()` method has already been mapped to *fe*, DCC also maps the `m2()` method to the same feature *fe*.

Formalization. The formal definition of this heuristic is defined in terms of number of family applications (n), the family application versions (v), the methods (o), the features (fe) and the relations (Formula 4-2). It was defined the *clone(o, o_j)* relation that establishes that a method o is similar to o_j by using the tree edit scripting algorithm (Nguyen *et al.* 2009). According to the

example showed in Figure 4.6, we have: the number of family applications (n = 1), the number of versions (v = 2), the methods to be analysed (`m1` and `m2`) and the feature *fe*.

$$DCC(fe) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{v} \{o : o \in O(fe) \wedge \exists o_j \notin O(fe) \wedge clone(o, o_j)\} \quad (4\text{-}2)$$

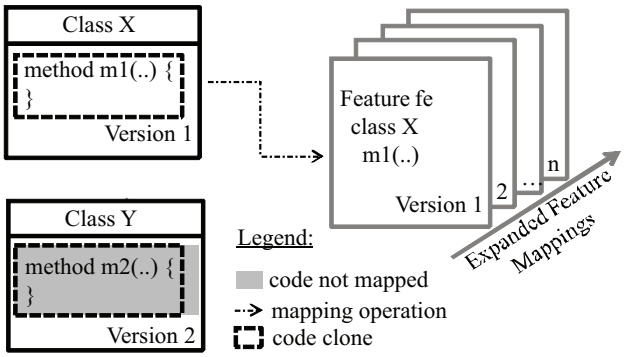

Figure 4.6: Abstract Representation of DCC.

4.4.4 Detecting Interfaces and Super-Classes

Strategy and Rationale. This heuristic named DIS (Detecting Interfaces and Super-Classes) detects two cases of super-classes and interfaces: either those that have not been mapped or those that have been incorrectly mapped. Its goal is to detect the occurrences of the mapping mismatch involving *interfaces and super-classes* (Section 3.2.2). As DIS detects entire super-classes and interfaces, the occurrences of the mapping mismatch related to the deficient module structure and documentation are also automatically detected (Section 3.3). For each feature, DIS analyses the super-classes and interfaces of all already mapped classes. First, it verifies if there are interfaces and/or super-classes that have not been mapped to the same feature and that are inherited by these mapped classes. This means that DIS captures any interfaces and/or super-classes inherited by mapped classes. Second, it also verifies if there are interfaces or super-classes that have been incorrectly mapped to a feature because, for instance, they belong to an API or framework.

The rationale behind DIS is that abstract classes provide pre-defined default behaviors that need to be comprehended and managed during the program family evolution. For example, in maintenance and evolution tasks it is required to know exactly how these default behaviors are defined in order to modify them and/or to implement specific behaviors (subclasses). Additionally, the mapping of interfaces is relevant because they also define default behaviors

and a common communication point among the modules. For this reason, DIS considers the mapping of all involved classes (subclasses, super-classes and interfaces) in the feature implementation. As part of the expansion process of mappings, DIS also takes into consideration the changes that a class has undergone during the family application history, such as starting to implement a new interface in a given family version. As a consequence, the expansion of the feature mappings is generated by taking into account such types of changes. The novel idea provided by this heuristic is the analysis of super-classes and interfaces regarding the historical information of the evolving program family (Section 4.3.1). Therefore, DIS is able to detect new super-classes and interfaces, update them and remove incorrect ones from the mappings generated during the program family's change history analysis.

Algorithmic Solution. Algorithm 4 presents the main part of the algorithmic solution implemented by DIS. Basically, this heuristic works seeking classes and interfaces that are inherited by implementation elements already mapped to a feature. DIS algorithm disregards all the classes that belong to API or standard libraries. The search is performed for each mapped element (line 03) by analysing its hierarchy of super-classes and interfaces of the abstract syntax tree (line 04) (Eclipse 2011). After that, DIS verifies if the identified super-classes and interfaces have been mapped. Otherwise, it maps these classes and interfaces to the same feature that their sub-classes were already mapped (line 06-08). Finally, the elements mapped incorrectly, for instance, which belong to an API are removed from feature mappings (line 09).

Algorithm 4 detectInterfacesClasses(version) - DIS Algorithm

```
1. featureList ⇐ allFeatures(version)
2. elementList ⇐ mappedElements(version)
3. for element in elementList do
4.    hierarchyElements ⇐ detectHistClassesInterfaces(element)
5. end for
6. for feature in featureList do
7.    addNewElements(feature, hierarchyElements)
8. end for
9. removeIncorrectElements(..)
```

Comparison to Existing Work. The dynamic techniques for feature mapping activity do not exhibit classes that belong to frameworks and APIs in their traces (Section 2.2.3). These techniques do not show also interfaces and super-classes which are involved in the feature implementation and that belong to the application domain. Developers when using static tools, in some cases, may know the starting point of the feature implementation but not all the involved implementation elements. However, in maintenance tasks it is primordial for the developers to have full knowledge about all the modules

and classes involved in the feature design and implementation. For instance, FEAT (Robillard and Murphy 2002) supports part of the concept presented by DIS. However, the developer needs to explicitly require the analysis from a selected class. Contrary to FEAT and other research work (Section 4.2), DIS provides support for the feature mapping expansion when analysing the multi-dimensional history of an evolving program family (Sections 4.3.1).

Illustrative Example. Figure 4.7 shows an abstract representation of DIS when detecting super-classes omitted from a feature mapping. For example, Figure 4.7 illustrates a `m2()` method in the `Y` class that has been mapped to the feature *(fe)* in the first version of the family application. However, in the second version of the family application the `Y` class was modified to inherit the `X` class, which has not been mapped. Hence, DIS verifies if the `X` class belongs to the application domain, then it adds this class automatically to the second version of the feature mapping.

Formalization. The formal definition of DIS is defined in terms of number of family applications (n), the family application versions (v), the modules (m), the features (fe) and the relations (Formula 4-3). A transitive relation was defined to formalize the heuristic DIS, so-called *descendant (m, m_j)*. This relation establishes that if a module (class) named m implements another one named m_j, $m \in M(fe)$ and m_j belongs to the application domain, then $m_j \in M(fe)$. Consequently, the module m_j is not part of a specific framework or API. According to the Figure 4.7, we have one application (n = 1) and two versions (v = 2), the classes to be analysed (`X` and `Y` classes) and the feature *fe*.

$$DIS(fe) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{v} \{m : m \in M(fe) \wedge \exists m_j \notin M(fe) \wedge descendant(m, m_j)\} \quad (4\text{-}3)$$

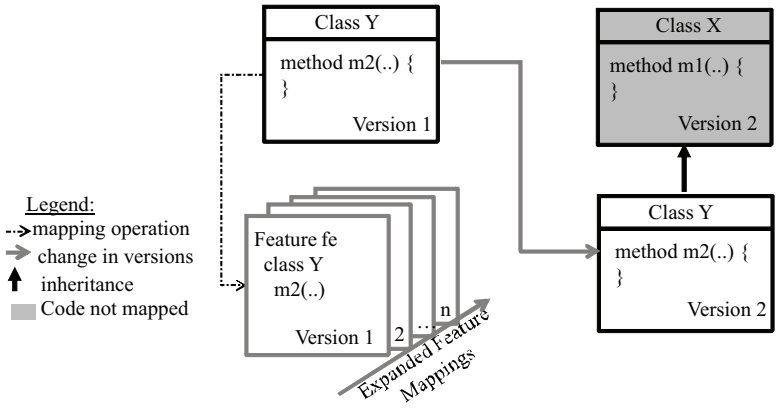

Figure 4.7: Abstract Representation of DIS.

4.4.5 Detecting Communicative Feature Mismatches

Strategy and Rationale. This heuristic named DCF (Detecting Communicative Feature Mismatches) detects implementation elements (i.e. classes and methods) incorrectly mapped to a feature. This occurs because there is a large interaction of this feature with other elements that realize different features. DCF has the goal of detecting occurrences of the mapping mismatch related to *overly communicative features* (Section 3.2.1). The heuristic DCF by means of fan-in number (Marin *et al.* 2007) is able to identify all classes and methods that are communicating with other ones already mapped. Fan-in number determines the number of distinct methods that invoke a given method (Marin *et al.* 2007). We use the fan-in number because it is able to capture all the dependencies from an implementation element. The dependencies refer to declarations and references (implicit and explicit) fostered by the use of object-oriented programming mechanisms.

The rationale behind DCF is to analyse the declarations and references from mapped elements. As a result, a set of implementation elements, that potentially are spread over many classes, is detected by DCF. These implementation elements are named referenced elements. As a second step, DCF compares the referenced elements with the already mapped elements. Those mapped elements which are not referenced ones are removed from the feature mapping. The explanation behind these removals is that such elements were incorrectly mapped because they contain strong dependencies with others that realize a different feature. The novel idea realized by DCF is related to the use of fan-in number from already mapped elements under a historical perspective of the family applications. It captures all the dependencies of a given implementation element when analysing the historical information with the goal of removing referenced elements incorrectly mapped. The strategy performed by DFP could not be used in the heuristic DCF for two reasons. First, DCF is not only focused on method calls but all kinds of references to other elements. Second, the use of thresholds required by the heuristic DFP to compare the interaction similarity of methods is restrictive (Section 4.4.2). Consequently, DFP does not consider all the dependencies from a given element.

Algorithmic Solution. Algorithm 5 presents the essence of the algorithmic solution implemented by DCF. The goal of DCF is to capture all the dependencies from an implementation element taking into account its declarations and references (implicit and explicit). DCF picks out each already mapped element (line 01) and uses a search pattern (line 03) that is provided by the Eclipse platform (Eclipse 2011). This pattern defines how the search

results are found and can be customized based on the references and declarations from an implementation element. A scope is also specified in order to define where the search pattern is applied. In this case, the scope is the family application version (line 04). After detecting the referenced elements (line 04), DCF analyses the already mapped elements in order to verify if they are not in the list of referenced elements, and consequently remove them from the mapping (lines 06-10).

Algorithm 5 detectCommunicativeFeatures(version) - DCF Algorithm

```
1.  elementList ⇐ mappedElements(version)
2.  for element in L do
3.     SearchPattern.createPattern(..)
4.     referencedElements ⇐ SearchEngine.createSearchScope(version)
5.  end for
6.  for element in elementList do
7.     if element is not in referencedElements then
8.        removedNonReferencedElement(element)
9.     end if
10. end for
```


Comparison to Existing Work. There are other research work and tools that use fan-in number with the goal of verifying certain types of dependencies among features (Robillard and Murphy 2002, Eisenbarth *et al.* 2003, Robillard and Weigand-Warr 2005, Marin *et al.* 2007, Zhang *et al.* 2008). Dynamic techniques are able to detect referenced implementation elements when exercising features as they are based on execution traces (Section 2.2.3). Additionally, some static analysis tools also implement the fan-in number in which the developers can obtain all the referenced elements when selecting a given implementation element (Section 2.2.2). In particular, the heuristic DCF provides an enhancement to these existing work by capturing all the dependencies from implementation elements and at the same time detecting inconsistencies in the mapping when analysing the multi-dimensional history of a program family. In addition, it also deals with the change analysis of the features' implementation elements which is not addressed by existing tools that rely on the call graph (Section 4.3.2).

Illustrative Example. Figure 4.8 shows an abstract representation of DCF. This figure illustrates the `m1()` and `m2()` methods in the `X` class that have been mapped to the feature *fe* in the first version of the family application. The `m4()` method in the `Y` class has also been mapped as both the `m1()` and `m2()` methods communicate with it. However, the `m3()` method in the `Y` class was incorrectly mapped. This mapping mismatch occurred because all the methods in the `X` class communicate with the `m4()` method. As a consequence, the entire `Y` class was incorrectly mapped. This way, DCF verifies each mapped element

in order to verify all its referenced elements. After that, it verifies if these referenced elements have been or not mapped to *fe*. According to this example, DCF observes that the `m1()` and `m2()` methods call the `m4()` method in the first version. However, DCF also verifies that the mapped elements do not have dependencies with the `m3()` method and thus removing it from the mapping. In the second version, it is observed that the `m1()` method was modified and does not communicate with the `m4()` method anymore. Consequently, DCF removes the `m1()` method and updates the feature mapping of the second version.

Formalization. The heuristic DCF is defined in terms of number of family applications (n), the family application versions (v), implementation element (ie), feature (fe) and relations (Formula 4-4). The implementation elements can be classes and methods. In order to formalize this heuristic it was defined the *connected(ie, ie_j)* relation that indicates that an implementation element `ie` is connected (or calls) another one ie_j. Based on the example in Figure 4.8, we have: n = 1, v = 2, the implementation elements to be analysed are the methods (`m1()`, `m2()`, `m3()` and `m4()`), and the feature (fe). The DCF formalization is defined as follows:

$$DCF(fe) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{v} \{ie : ie \notin O(fe) \wedge \exists ie_j \in O(fe) \wedge connected(ie, ie_j)\} \quad (4\text{-}4)$$

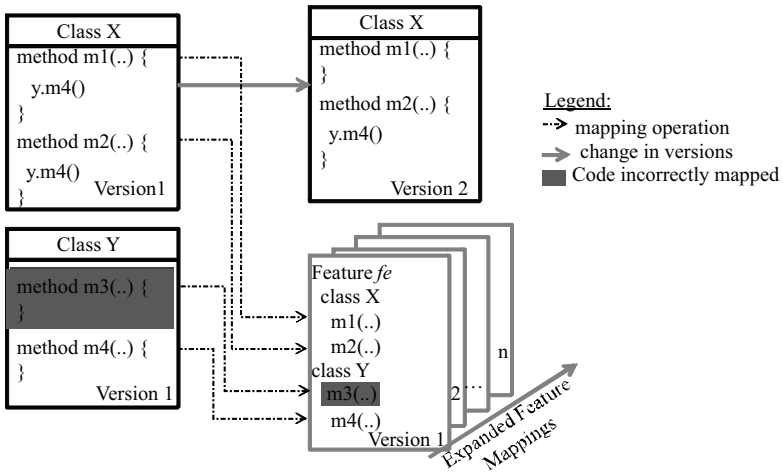

Figure 4.8: Abstract Representation of DCF.

4.4.6 Detecting Omitted Attributes

Strategy and Rationale. This heuristic named DOA (Detecting Omitted Attributes) is responsible for detecting class attributes that have not been mapped. Its goal is to detect occurrences of the mapping mismatch named

omitted attributes (Section 3.2.2). The idea of DOA is, first, to identify already mapped methods to each feature (*fe*). Then, based on attribute access graphs, it is possible to detect all class attributes which are accessed by the mapped methods within any classes of the family application version. Finally, it is checked if they have already been mapped or not to the same feature of the mapped method. As a consequence, DOA keeps track of all the class attributes that are used by the mapped methods. This heuristic also analyses the likelihood of omitted attributes being part of a feature by observing those attributes which are not already part of another feature. This analysis is performed by verifying which attributes have been incorrectly mapped to a feature, when actually the method that accesses them has been mapped to another feature. Additionally, DOA checks if those attributes were added and/or changed simultaneously with the mapped methods throughout the family application's change history. Therefore, the novel idea proposed by DOA is the multi-dimensional history analysis and the respective changes of the features' elements during the expansion process of the mappings (Sections 4.3.1 and 4.3.2).

Algorithmic Solution. Algorithm 6 presents the main part of the algorithmic solution implemented by DOA. First, DOA detects all the class attributes that are accessed by each mapped method of a family application version (line 03-05). This detection is performed by using the representation of the abstract syntax tree of source code. After that, DOA maps the attributes (line 07), if they were not mapped, to the feature that is being evaluated. DOA algorithm is run for each family application version in order to add or update the list of attributes in mappings during the expansion process (Section 4.4.1).

Algorithm 6 detectOmittedAttributes(version) - DOA Algorithm

```
1. featureList ⇐ allFeatures(version)
2. elementList ⇐ mappedElements(version)
3. for method in elementList do
4.     attributeList ⇐ accessMethodAttributes(method)
5. end for
6. for feature in featureList do
7.     addNewElement(feature, attributeList)
8. end for
```

Comparison to Existing Work. Dynamic techniques are useful for detecting attributes that implement features, and consequently they can be used for generating the seed mappings (Section 4.3.1). However, these techniques are only suitable for user-level features (Section 2.2.3). Other types of features, such as crosscutting features and/or fine-grained features that can not have their elements completely exercised and revealed with user/test inputs, are not detected. As a consequence, it is up to the developer performs either the entire

mapping of a feature or complement it through the use of other techniques. For instance, the developer could rely on static techniques (Section 2.2.2) for complementing their mappings and thus deciding which implementation elements are part of the feature. FEAT (Robillard and Murphy 2002) supports part of the idea realized by DOA as the developer can select a method and navigate on its attributes through queries available for classes and methods. However, the developer needs to explicitly carry out these actions of selecting the method and picking out the relevant attributes. In addition, the developer needs to perform these actions individually for each family application version as FEAT does not support the analysis of multiple versions. In summary, DOA provides the following enhancements when compared to existing tools: (i) it provides the expansion process of mappings in evolving program families (Section 4.3.1), and (ii) it maintains the consistency of the mappings during the expansion process for considering the program family's change history (Section 4.3.2).

Illustrative Example. Figure 4.9 illustrates an abstract representation of DOA. For example, in this figure there are a `X` class and a `m1()` method that have been mapped to the feature (*fe*) in the first version of the family application. However, the class attributes named `a1` and `a2` were not mapped and they also realize the same feature and are used by the `m1()` method. As the `m1()` method has already been mapped and it accesses these attributes, they are automatically added to (*fe*), as illustrated in Figure 4.9. DOA analyses the family application's history and the changes of the implementation elements with the goal of expanding the mappings. This occurs when, for example, the mapped attributes in the first version have their names modified in the second version of the `X` class (Figure 4.9). Consequently, DOA automatically generates the feature mapping of the second version of the family application taking into consideration such changes. This occurs because the `m1()` method and the `a1` and `a2` attributes have already been mapped and detected by DOA in a previous version (Version 1).

Formalization. The formal definition of DOA is defined in terms of number of family applications (n), the family application versions (v), the attributes (a), the features (*fe*) and the relations (Formula 4-5). It was defined *attri(fe)*, which is the set of attributes that implements this feature *fe*. Additionally, it was also defined the *use(o,a)* relation that indicates that a method (operation) `o` uses a class attribute `a`. According to the illustrative example shown in Figure 4.9, we have: the number of family applications (n = 1), the number of family versions (v = 2), the attributes to be analysed (`a1` and `a2`) and the feature *fe*. The DOA formalization is expressed as:

$$DOA(fe) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{v} \{a : a \notin attri(fe) \wedge \exists o \in O(fe) \wedge use(o, a)\} \quad (4\text{-}5)$$

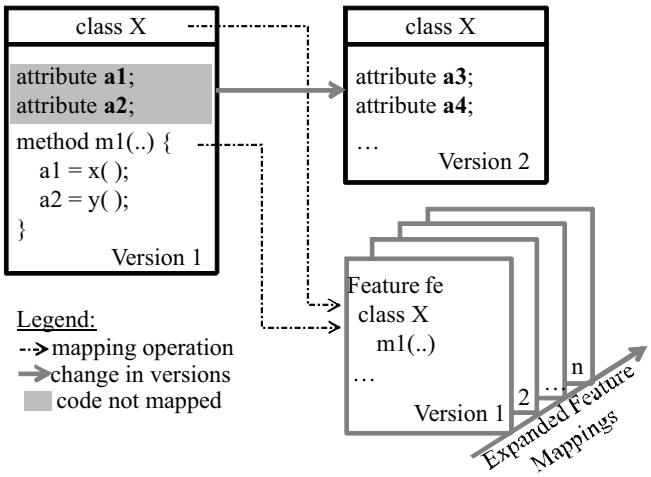

Figure 4.9: Abstract Representation of DOA.

4.5 MapHist: A Heuristic-based Tool for Expanding Feature Mappings

This section describes MapHist, a tool for expanding feature mappings when analysing evolving program families. It was implemented as an Eclipse plug-in (Eclipse 2011) and supports the use of the heuristics presented in Section 4.4. Section 4.5.1 presents the tool architecture and its main functionalities. Section 4.5.2 describes the representation of the implementation elements and complementary tools used by MapHist.

4.5.1 The MapHist Architecture

MapHist relies on the mapping from implementation elements to features. Figure 4.10 shows the MapHist architecture, which comprises four modules described as follows:

1. **Mapping Collector**. This module is responsible for collecting the features and the seed mappings with their initial set of implementation elements. According to our heuristic methodology (Section 4.3.1), the developer must select the family application versions that contain the seed mappings, which are used as input by the MapHist tool (Section 4.3.1). To be more specific, it loads the features and their implementation elements in a *collection* to be further processed by the mapping expansion

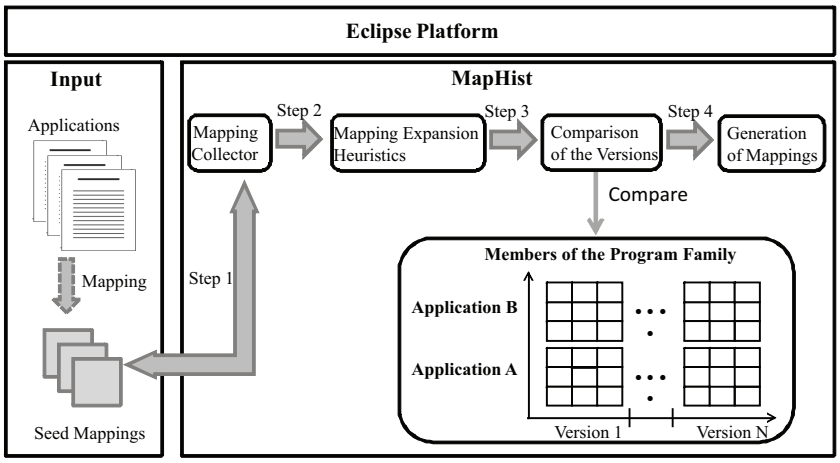

Figure 4.10: The MapHist Architecture.

heuristics (Step 2). In the current implementation, MapHist supports mapping files in XML format. Nevertheless, there is a MapHist interface that can be implemented to manage other mapping file extensions. For this reason, MapHist can be used together with any existing mapping tool (Section 4.2.1).

2. **Mapping Expansion Heuristics**. This module comprises the set of proposed mapping heuristics. The heuristics analyse the *collection* passed by the mapping collector (Step 1). After that, each heuristic processes this *collection* and executes its algorithm described in Section 4.4. As mentioned, during the expansion process of the mappings the heuristics also take into consideration the comparison among the versions (Step 3). The heuristics require the generation or update of the feature mappings when implementation elements that realize a given feature are detected. A pair containing the feature and its respective implementation elements is passed to the module of generation of the feature mappings (Step 4).

3. **Comparison of the Versions**. This module is responsible for comparing the program family's versions and reducing the occurrence of irrelevant information related to the change history of the family applications (Section 4.6.3). This irrelevant information was reduced through our comparison process among the versions. To this end, we have used JDiff (Doar 2007), which is a tool that compares the source code in Java language and generates a XML file in terms of changed, removed and added elements. As mentioned in Section 4.3.2, the comparison provides information that is used as input for the expansion process. To minimize irrelevant information to be processed by the heuristics we concentrated on defining a comparison strategy and a set of rules to be followed by

the expansion process. Even though we defined a comparison strategy and used a tool for analysing the changes in the source code structure, it was needed to deal with, for instance, small structural changes in a method, such as simple refactoring of methods and variable names. These kinds of structural changes do not modify the semantic behavior of the implementation element. For this reason, we complemented the use of the tool for analysis of source code changes, JDiff (Doar 2007), with a clone tool called Clever (Nguyen *et al.* 2009). Hence, it was possible to observe if a changed method in a given version had its body widely modified in the next ones (Section 4.3.2). We explored the multi-dimensional historical analysis when using different strategies: a comparison strategy, structural analysis of the program families' source code and code clone tools. This way, we do believe that it was possible to minimize irrelevant information concerning program family's change history to be processed by the heuristics.

4. **Generation of Feature Mappings**. This module is responsible for generating and manipulating the mapping files. It adds and removes implementation elements required by the mapping heuristics. At the end of the execution, a feature mapping file in XML format is generated for each family application version. Developers can navigate on these generated mappings in order to analyse and understand how the features have evolved over time.

4.5.2 Representation of the Implementation Elements and Use of Existing Tools

Considering that MapHist was implemented as an Eclipse plug-in, we have used the elements and relationships provided by JDT (Eclipse 2011). Basically, the heuristics manipulate elements that represent the program structure, such as compilation units, types, fields, methods, statements, parameters, signatures, return types. Also, a set of mechanisms supported by JDT was used. For instance, we implemented *Visitors* based on the AST (Abstract Syntax Tree) in order to implement the manipulation of an implementation element at a high level (i.e. methods) in the heuristic DOA (Section 4.4.6). Additionally, we also implemented a *Search Engine* to do search in the source code and have access to all references or declarations of an implementation element based on call graphs (i.e. declarations of fields, implicit, explicit and qualified references) for the heuristic DCF (Section 4.4.5). To implement the heuristic DFP (Section 4.4.2) we adapted the interaction similarity algorithm

proposed by Nguyen *et al.* (Nguyen *et al.* 2011) to consider the feature mapping and historical information. In the implementation of the DIS heuristic we also dealt with the elements provided by JDT.

In order to compare the bodies of the methods we have used the clone tool named Clever (Nguyen *et al.* 2009). The Clever tool has been used during the expansion process and in the implementation of the heuristic DCC because: (i) it already provides an Eclipse plug-in, and (ii) it uses a different approach to compute tree editing scripts that obtained good results in detecting code clones when compared to other related tools, such as CCFinderX (Kamiya *et al.* 2002). We have used the ConcernMapper mapping tool (Robillard and Weigand-Warr 2005). We have chosen the ConcernMapper tool for two main reasons. First, its plug-in has an easy integration with the Eclipse Platform. Second, its implementation is stable and it has an extensive documentation available. The mapping is performed when a given implementation element is related to a given feature. MapHist uses the representation of features provided by ConcernMapper, which has a list of features' names and the implementation elements that realize each feature.

We tried to maximize the use of existing tools because they have already been tested and evaluated in other software systems. Hence, we could concentrate on the conception of the original ideas proposed by the mapping heuristics. In addition, such tools have achieved good results when compared with their related work. It is important to highlight that the MapHist tool was designed to be decoupled of any kind of tool. To make this possible, interfaces are provided in order to allow the inclusion of the target tools.

4.6 Evaluation

This section describes the evaluation process of the mapping heuristics. Section 4.6.1 presents the target program families and the reasons why they were selected. Section 4.6.2 describes the study procedures followed during the evaluation process. Section 4.6.3 presents the analysis of the results achieved by our mapping heuristics. Section 4.6.4 presents some discussions concerning the results achieved by the mapping heuristics and other research work. Finally, Section 4.6.5 discusses the limitations of our work.

4.6.1 Target Program Families

Two evolving program families were selected to evaluate the accuracy of the mapping heuristics. One of them is the OC program family, which was

already described in Section 3.1.1. The second family comprises applications that follow a reference architecture used for generating Web information systems for different purposes, such as managing the control of pollutants, managing the drawing and manipulation of maps. This reference architecture prescribes a set of inter-dependent components that implement a wide range of features, such as storing, searching and updating entities, and automatic generation of HTML pages. There are six family applications derived from this reference architecture and they have been developed since 2008. For the purpose of our analysis and evaluation, we have selected three of them due to the number of available versions. For confidentiality agreement issues, the fictitious name of RAWeb is used to refer to the program family. We selected a total number of 13 versions, which refer to five versions of one family application and four versions of the second and third family applications. The reason for this particular set of versions is due to the fact that the RAWeb applications were developed through large increments. For this reason, there are few versions available of the family's applications. These applications have size that range from 100 KLOC to 110 KLOC.

Tables 4.1 and 4.2 describe the analysed features in OC and RAWeb program families respectively, and which family applications realize them. We selected common and variable features. This enabled us to observe how they have been implemented and modified for each version of the framework, reference architecture and family applications. Given the optional nature of the core implementation, the features *Route*, *Notification* and *Export* are not instantiated by all the family applications (Table 4.1). The same is true for the feature *Import Image*. It is important to highlight that we used *Applications I* and *II* of OC program family in the exploratory study for classifying the mapping mismatches (Chapter 3). For this reason, we included a new application (*Application III*) to evaluate the mapping expansion heuristics. Additionally, we selected different versions of *Applications I* and *II* of OC program family in order to avoid including bias in the assessment of the proposed mapping heuristics (Section 4.4). This bias is related to the fact that the heuristics are based on the catalogue of mapping mismatches.

Table 4.1: Analysed features in OC.

Family Applications	Features	Description
All	Logger	It saves information about program execution and/or errors.
All	Report	It represents the report exhibition, exportation and printing.
OC framework, Application I	Route	It represents a route of products between two points in the logistics context.
OC framework	Notification	It defines a system notification to users (i.e. email).
OC framework, Application I	Export	It represents the generation of reports for different files formats (i.e. Excel).
All	Transaction	It is responsible for storing and recovering data from the database and ensuring ACID properties.
Applications II and III	Stock	It manages the amount of different products in the logistics domain.
Application III	Importation	It manages and defines the rules of the importation of products.

Table 4.2: Analysed features in RAWeb.

Applications	Features	Description
All	Dynamic Forms	It creates the Web pages automatically by including all the fields and their types. Also, it validates the respective fields according to its type (e.g. String).
All	Dynamic Tables	It dynamically exhibits all the fields and their respective values of the object in a table. It also provides options to exclude and edit such objects.
All	Search Data	It allows the search for objects based on their fields. It also allows saving the search result.
Application I	Import Image	It imports large images in order to reduce them and consequently makes their manipulation easier.
All	Report	It represents the report exhibition, exportation and printing.

4.6.2 Study Design

A number of procedures were followed during the evaluation process of the mapping heuristics. The study procedures were organized in five steps:

1. The feature mappings were produced for each first version of the selected family applications of the respective program families following our methodology (Section 4.3.1). In the case of OC program family, the seed mappings of the features (Table 4.1) were provided by two developers who have used and reviewed the source code of the selected family applications for their documentation. These seed mappings were performed in cooperation by these two developers. The seed mappings of the OC program family were about 20% of coverage considering the functional

features and about 15% of the non-functional crosscutting features. In the case of RAWeb program family, the seed mappings were performed by one developer. The seed mappings of RAWeb family were about 10% for all the features. The developers involved in the production of the seed mappings have not developed the program families but they have knowledge of the selected members and features' code. This occurs because they have been involved in the documentation and evolution processes of such family applications. For this reason, they have significant knowledge of the features involved in the mapping expansion process. After the developers produced the seed mappings, we analysed such mappings in order to verify if they contained all the mismatches when considering each feature (Chapter 3). This procedure was performed with the goal of verifying if the mapping heuristics are able to expand the mappings in the presence of all types of mismatches;

2. We ran the mapping heuristics using the MapHist tool in order to provide the expanded mappings for all the versions of the family applications. Consequently, the mappings of each family application version were generated;

3. We measured the accuracy of the mapping heuristics by calculating precision and recall after generating all the feature mappings of each family application. The purpose of precision is to verify if the proposed heuristics are able to select only the implementation elements that realize a given feature; whereas recall measures verify if the heuristics are able to detect all the feature's implementation elements. The precision and recall measures were computed manually after validating the results with the developers who provided the seed mappings (Step 4). We have not evaluated each heuristic apart because the heuristics are dependent on the results of each other. As mentioned in Section 4.4.1, as feature mappings are updated by a given heuristic, they are used as basis by others during the expansion process. For instance, the heuristic DOA is only able to detect omitted attributes whether methods are detected and mapped by other heuristics, such as DFP, DCC. The heuristic DOA is an example of how it is hard to assess each heuristic separately due to the dependency among them. For this reason, we calculated the precision and recall measures of the entire mappings;

4. We manually validated the obtained results by the heuristics with the developers who provided the seed mappings of the program families' applications. This step was important in order to assess if the feature

mappings generated by the heuristics were correct according to the developers' knowledge. To perform this validation, we provided the feature mappings of the family applications and asked the developers who are knowledgeable about the systems' source code about about the mappings. The developers analysed the generated mappings in order to evaluate the correctness of the implementation elements realizing each feature. This way, we could observe which implementation elements were incorrectly mapped or were missing in feature mappings. It is important to highlight that the number of mapped elements increases or decreases depending on the number of added, removed or changed elements from one version to another one realizing the features (Section 4.6.3). This is valid to show that the number of elements is varying and that the heuristics are able to detect them and update the generated feature mappings;

5. Finally, we provided a general discussion involving the recall and precision measures achieved by the mapping heuristics and previously data reported by other works, such as XScan (Nguyen *et al.* 2011) and COMMIT (Adams *et al.* 2010). A direct comparison was not performed as it was not possible to run the COMMIT and XScan tools. The reasons were twofold. First, the COMMIT tool was only developed for supporting C code which is executed through Perl scripts. As a consequence, it could not be run because the selected program families have been developed in Java. Second, the XScan tool (Nguyen *et al.* 2011) only detects non-functional crosscutting features and we only selected two of them (*Logger* and *Transaction*), which have not been evaluated in (Nguyen *et al.* 2011). Both aforementioned tools were chosen because they are able to identify implementation elements that share non-functional crosscutting features in individual evolving applications (Section 4.2.2). We discussed if the mapping heuristics assessed on the top of evolving program families achieved as good results as using these existing tools. As the authors of these works calculated the precision and recall measures for the evaluated crosscutting features using their tools, we used the variation of reported values to guide our discussions. We have not selected the same applications used in the assessment of the aforementioned tools because some of these applications were developed in C language. Additionally, they are individual evolving applications and thus they are not part of evolving program families.

4.6.3 Data Analysis

This section shows the analysis of the mappings of each evolving application of the program families. We adopted this strategy for two reasons: (i) to explain the results associated with the common and variable features, and (ii) to analyse the precision and recall measures of the family application versions. The latter enabled us to systematically analyse the effectiveness of the proposed heuristics during the expansion process of the mappings. This way, we could verify if the change history analysis of the program family have influenced positively or negatively the accuracy of the mappings over time.

OC Framework. The results of the precision and recall measures for each version of the *OC framework* are shown in Table 4.3. The results are presented for each selected feature. As it can be observed, the mapping heuristics presented in 5 (five) out of 6 (six) features recall measures of 100%. The feature *Export* was the only one that presented recall measures that ranged from 93% to 97%. It is possible to observe that during the framework evolution, the recall has improved when analysing the historical information. In terms of precision, in 4 (four) out of 6 (six) features the values were higher than 90%. The lowest measures are related to the features *Logger* (from 66% to 71%) and *Route* (from 82% to 88%). The explanation behind the results associated with the features *Logger* and *Route* is that the OC family contains a set of classes that share the same code. These classes are responsible for creating the graphical user interface (GUI) through the Decorator pattern (Gamma *et al.* 1994). There are classes that do not directly realize those features but they contain interaction similarity with the others that realize them (Sections 4.4.2 and 4.4.3). Consequently, according to the strategies of interaction similarity and cloned code implemented by the heuristics DFP and DCC respectively, many classes were mapped to these features. As a result, they generated false positives. The factor that contributed, for instance, to the low precision of the feature *Logger* was because the main classes that realize it have not significantly changed throughout the selected versions of the *OC framework*. In general, this scenario is particularly real as non-functional crosscutting features do not have their requirements often changed. In order to circumvent this problem, a solution could be to increase the thresholds associated with both heuristics during the analyses of the historical changes. Consequently, it would be possible to rank and focus on methods with a higher degree of interaction similarity or code clones during the analysis of the evolving program families. The idea is to disregard pieces of code with small similarity. Despite the feature *Logger* has presented low values in terms of

precision, the feature *Transaction* achieved good results. Therefore, the lowest results are not associated with crosscutting features.

Table 4.3: Precision (P) and Recall (R) Results for each Version of the OC framework (OC family).

	Versions									
	V1		V2		V3		V4		V5	
Features	P	R	P	R	P	R	P	R	P	R
Export	100%	93%	100%	98%	100%	98%	100%	98%	100%	97%
Logger	66%	100%	70%	100%	71%	100%	71%	100%	71%	100%
Notification	92%	100%	93%	100%	93%	100%	93%	100%	93%	100%
Report	93%	100%	97%	100%	97%	100%	97%	100%	97%	100%
Route	82%	100%	84%	100%	87%	100%	88%	100%	88%	100%
Transaction	94%	100%	93%	100%	94%	100%	98%	100%	98%	100%

OC Applications. Tables 4.4, 4.5 and 4.6 show the results of the precision and recall measures for *Applications I, II and III*, respectively. For example, when comparing the values of the *Applications I, II and III*, it can be observed that there was a variation regarding the feature *Logger* in terms of recall measures. Basically, this difference occurred because the feature *Logger* in *Applications II and III* is more concentrated on a set of classes; whereas it is very scattered in the *Application I*. For instance, the heuristic DFP could not capture the methods that realize the feature *Logger* in *Application I* because they do not contain many similar method calls when compared to other ones. In particular, they implement a specific routine and thus they have a low interaction similarity. In general, the recall measures presented a good level of coverage for capturing all the relevant implementation elements of each feature. For this reason, the values were almost 100% for all the features (except *Logger* in *Application I*), which mean that the heuristics presented more false positives than false negatives. The precision measures for all the features improved when compared to the *OC framework* values. For example, the precision measures for the feature *Logger* improved for all the family applications. When analysing the variable features (i.e. Stock and Importation) the mapping heuristics also presented precision measures higher than 90% for each feature in all the versions; whereas the recall measures were 100%. Therefore, it is possible to observe from these values that the mapping heuristics have shown a good effectiveness with slight variations during the analysis of the family application's history.

Table 4.4: Precision (P) and Recall (R) Results for each Version of Application I (OC family).

	Versions									
	V1		V2		V3		V4		V5	
Features	P	R	P	R	P	R	P	R	P	R
Export	92%	100%	92%	100%	92%	100%	92%	100%	92%	100%
Logger	70%	90%	71%	92%	73%	93%	73%	93%	73%	93%
Report	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Route	80%	100%	83%	100%	84%	100%	84%	100%	84%	100%
Transaction	94%	100%	94%	100%	94%	100%	96%	100%	96%	100%

Table 4.5: Precision (P) and Recall (R) Results for each Version of Application II (OC family).

	Versions									
	V1		V2		V3		V4		V5	
Features	P	R	P	R	P	R	P	R	P	R
Logger	85%	100%	78%	100%	80%	100%	80%	100%	80%	100%
Report	92%	100%	92%	100%	92%	100%	92%	100%	92%	100%
Transaction	90%	100%	90%	100%	91%	100%	91%	100%	90%	100%
Stock	95%	100%	92%	100%	93%	100%	93%	100%	93%	100%

Table 4.6: Precision (P) and Recall (R) Results for each Version of Application III (OC family).

	Versions									
	V1		V2		V3		V4		V5	
Features	P	R	P	R	P	R	P	R	P	R
Importation	97%	100%	98%	100%	98%	100%	98%	100%	98%	100%
Logger	76%	100%	75%	100%	78%	100%	78%	100%	78%	100%
Report	97%	100%	98%	100%	98%	100%	98%	100%	98%	100%
Transaction	92%	100%	92%	100%	92%	100%	92%	100%	92%	100%
Stock	92%	100%	93%	100%	91%	100%	91%	100%	91%	100%

RAWeb Applications. Tables 4.7, 4.8 and 4.9 present the results of the precision and recall measures for the *Applications I, II and III* of RAWeb program family. As can be observed, the recall measures were 100% for all the features. This means that the heuristics were able to successfully detect all the code elements realizing the features. There are mainly two factors that contributed to these high results: (i) the evolution of the applications' versions is more related to the inclusion of entire classes, interfaces or methods instead of changing existing pieces of code and (ii) most of these classes are included to extend interfaces or abstract classes which were already previously mapped. This occurs because most of the included classes are totally responsible for implementing the analysed feature. In these cases, the heuristic DIS is able to capture these occurrences of classes and interfaces as it keeps track the use of already mapped classes during each family member history. For instance, the heuristics achieved maximum values for the feature *Report* in all RAWeb applications. This mainly occurred because most of the classes realizing the feature *Report* implement the `IListReport` interface, which had already been mapped in previous versions. The precision measures presented similar values when compared to the OC program family. In general, the precision measures

were higher than 82% for all the features. The lowest precision values are related to the feature *Dynamic tables* in *Application I*. This occurred because some classes that are used to realize the feature *Dynamic Tables* communicate with classes realizing the feature *Report*. We can conclude that the heuristics present similar behavior when analysing both program families.

Table 4.7: Precision (P) and Recall (R) Results for each Version of Application I (RAWeb family).

	Versions									
	V1		V2		V3		V4		V5	
Features	P	R	P	R	P	R	P	R	P	R
Dynamic Forms	93%	100%	94%	100%	94%	100%	94%	100%	94%	100%
Dynamic Tables	82%	100%	83%	100%	86%	100%	86%	100%	86%	100%
Search Data	95%	100%	95%	100%	97%	100%	97%	100%	97%	100%
Import Image	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%
Report	100%	100%	100%	100%	100%	100%	100%	100%	100%	100%

Table 4.8: Precision (P) and Recall (R) Results for each Version of Application II (RAWeb family).

	Versions							
	V1		V2		V3		V4	
Features	P	R	P	R	P	R	P	R
Dynamic Forms	98%	100%	98%	100%	99%	100%	99%	100%
Dynamic Tables	96%	100%	97%	100%	97%	100%	97%	100%
Search Data	92%	100%	93%	100%	93%	100%	93%	100%
Report	100%	100%	100%	100%	100%	100%	100%	100%

Table 4.9: Precision (P) and Recall (R) Results for each Version of Application III (RAWeb family).

	Versions							
	V1		V2		V3		V4	
Features	P	R	P	R	P	R	P	R
Dynamic Forms	97%	100%	97%	100%	98%	100%	98%	100%
Dynamic Tables	95%	100%	96%	100%	97%	100%	97%	100%
Search Data	93%	100%	93%	100%	93%	100%	93%	100%
Report	100%	100%	100%	100%	100%	100%	100%	100%

Figure 4.11 shows the number of added (add), changed (chg) and removed (rem) elements for a set of compared versions in the OC program family. These elements refer to the attributes, methods or classes. The goal is to show in absolute numbers the effectiveness of the mapping heuristics for detecting such elements during the historical analysis of the program family. According to this figure, we can observe that the number of changed and removed elements are fully detected by our heuristics. This good accuracy occurred due to the comparison strategy, tools used in our implementation and the set of rules defined for the expansion process (Sections 4.3.2 and 4.5.1). This enabled us to deal with any types of changes in an efficient way, such as refactorings of variable and method names, change of method parameters (Sections 4.5.1 and 4.5.2). In terms of added elements, the heuristics also presented relevant

effectiveness, even not detecting all the elements. This was mainly observed for the recall measures involving the features *Export* and *Logger* in the *OC framework* and *Application I*.

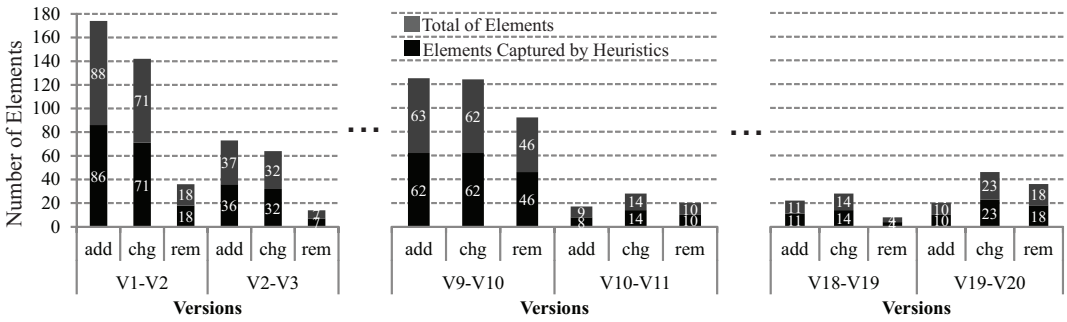

Figure 4.11: Number of Elements throughout the History in OC program family.

Figure 4.12 illustrates step-by-step how the heuristics expand the mappings and remove the mismatches by following the execution order defined in Section 4.4.1. The pieces of code realize the feature *Route* and were derived from the OC program family (Section 4.6.1). The provided seed mapping contains the following implementation elements already mapped to the feature *Route*: the `removeRoute(Route)` method of the `RouteServiceDB` class, the `makeModalConfig()` and `loadProperties()` methods of the `RouteConfigurator` class, and the `removeRoute(Route)` method of the `RouteProxy` class. First, the heuristic DFP is run and it detects the `updateRoute(route)` and `removeRoute(Route)` methods with interaction similarity. As the `removeRoute(Route)` method has already been mapped, DFP maps the `updateRoute(route)` method to the feature *Route* as well. Second, the heuristic DCC detects that the `removeRoute(route)` and `removeRoute(route)` methods are code clones, which are in the `RouteServiceProxy` and `RouteProxy` classes, respectively. As a consequence, the `removeRoute(route)` method in the `RouteServiceProxy` class is mapped to the feature *Route*. Third, the heuristic DIS detects that there are superclasses and interfaces, `RouteServiceInterface` and `Service`, which have not been mapped to the same feature and that are inherited by these mapped classes: `RouteServiceProxy`, `RouteProxy` and `RouteServiceDB`. As a result, `RouteServiceInterface` and `Service` are mapped to the feature *Route*. Fourth, the heuristic DCF detects that all the methods of the `RouteConfigurator` class were mapped. However, DCF observes that there are no methods that call the `makeModalConfig()` method. As a consequence, the `makeModalConfig()` method is removed from the mapping. Finally, the

heuristic DOA detects the class attribute named `instance` of the `RouteProxy` class and maps it to the feature *Route*.

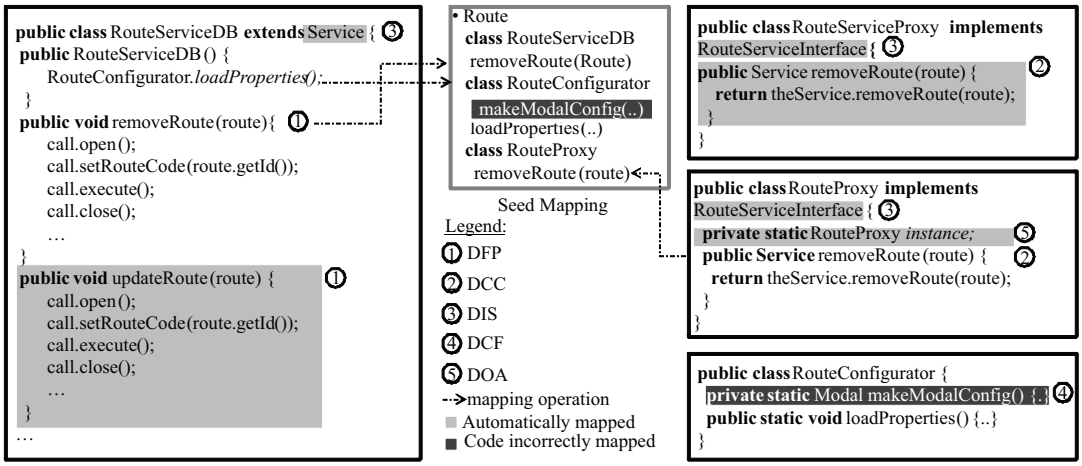

Figure 4.12: Workflow of the Mapping Expansion Heuristics.

4.6.4 Discussions

We also provided a general discussion concerning the results achieved by the proposed heuristics and the XScan (Nguyen *et al.* 2011) and COMMIT (Adams *et al.* 2010) tools. The discussion is based on the variations of results reported in the assessment of these tools (Nguyen *et al.* 2011, Adams *et al.* 2010). These variations were chosen because both studies did not present the values for each version of the evaluated application. As mentioned in Section 4.6.2, a direct comparison with such tools was not possible. We discussed if the mapping expansion heuristics applied to evolving program families presented acceptable results in the light of existing alternative tools. Both studies have evaluated a set of non-functional crosscutting features, whose their definition is presented in Section 4.2.2. For example, the similarity strategy supported by the XScan tool was evaluated for a set of non-functional crosscutting features in different applications, such as: persistence, iterator, undo. The precision measures ranged from 41% to 100% (Nguyen *et al.* 2011). Analysing these results against our precision measures, we can observe that our measures ranged from 66% to 100%, where the lowest values were 66% for the features *Logger* and 82% for *Route* in OC program family and *Dynamic Tables* in RAWeb family (Section 4.6.3). It is important to highlight that the *Logger* feature was not evaluated in (Nguyen *et al.* 2011). In terms of recall, the measures ranged from 89% to 100%, whereas our measures ranged from 90% to 100%.

On the other hand, the COMMIT technique presented results of precision and coverage measures that ranged from 45% to 75% (Adams *et al.* 2010). The authors only presented the average values considering all the detected crosscutting features. It was not reported precision and coverage measures associated with each feature individually as we showed in Section 4.6.3. However, based on the aforementioned measure ranges, it is possible to observe that they are inferior to the results obtained with our heuristics. When analysing the variations achieved by XScan (Nguyen *et al.* 2011) and COMMIT (Adams *et al.* 2010), it is possible to observe general gains of the mapping heuristics related to both techniques. On the other hand, it is important to highlight that COMMIT has also a complementary additional goal that is to detect code elements that have been changed together intentionally (Section 4.2.2). So, a direct comparison can not be drawn in this case.

In fact, concrete arguments and deep conclusions could not be described due to the restrictions of a direct comparison with such tools. When comparing quantitatively and qualitatively these tools a set of factors addressed by our mapping expansion heuristics has a direct influence in the results of precision and coverage measures. These factors are the following: analysis of evolving program families and their change history, the ability of dealing with any type of feature and not only non-functional crosscutting features, the input of seed mappings (Section 4.4).

4.6.5 Limitations

The limitations of the mapping expansion heuristics are related to two factors: seed mappings and new features that emerged in the change history of each family member. The seed mappings are required by our heuristics, which depend on inputs provided by the developers. They should contain the list of selected features and an initial set of implementation elements that realize each feature (Section 4.3.1). It is important to highlight that these seed mappings should be provided by developers who have reasonable knowledge of the program family and its features. The seed mapping is a pivotal characteristic that can influence on the good results generated by the heuristics. However, our heuristics do not depend on a complete and correct mapping as they have the goal of expanding feature mappings (Section 4.3.1).

In our evaluation, the seed mappings of the OC family were about 20% of coverage considering the functional features and 15% for the non-functional crosscutting features. On the other hand, the seed mappings of the RAWeb family were about 10% for all the assessed features. This occurs because

the functional features are often domain-specific and they are realized by a set of specific modules, which are not widely scattered in program families (Figueiredo *et al.* 2008, Figueiredo *et al.* 2011). Based on this coverage level, our heuristics have achieved good results in terms of precision and recall (Section 4.6.3). The second factor is related to new features that emerge in the family application versions. If new features arise but are not previously defined in seed mappings, they are not automatically captured by our heuristics. In these cases, developers need to provide more than one seed mapping and execute the heuristics from a given version that contains these features.

4.7 Threats to Validity

This section discusses the main threats to validity of this study according to the classification proposed by Wohlin *et al.* (Wohlin *et al.* 2000).

Conclusion Validity. Three threats were identified in this category. The first one is related to the set of selected features in our evaluation that can have favored the results achieved by the mapping expansion heuristics. To reduce this threat we selected functional features, which are related to the core domain of the program families (i.e. Notification), non-functional features which are scattered for the entire evolving program families (e.g. Logger) and variable features (e.g. Stock, Import Image). Additionally, these features were also chosen because they are representative in the program family domain as well as they have evolved over time. The second one is related to the validation of the mappings generated by the mapping heuristics. To reduce this threat we validated the mappings with the two developers responsible for providing the seed mappings of the selected program families. The third one is related to the use of thresholds to detect omitted feature partitions. This is a threat as the results can vary depending on the chosen values of the thresholds.

Construct Validity. Three threats were identified in this category. The first one is related to the procedures that should be followed to evaluate the mapping heuristics. Consequently, we defined clear procedures to be followed and applied during the analyses of all applications. They were responsible for making the evaluation more precise and thus reducing possible inconsistences. The second one is related to the seed mappings which were conducted by two developers in the OC family and one developer in the RAWeb family. These developers are not the original developers of the program families but have a good knowledge of the program families' source code and features. Additionally, it was also possible to validate the mappings generated by the heuristics with them. Finally, the third one is related to the scalability issues

of the algorithms. The scalability of the algorithms can be high due to the number of family members to be analysed.

Internal and External Validity. One threat was identified for the internal validity. It is related to the history of the evaluated program families. To reduce this threat we chose program families that are representative in the industry scenario and have been extensively maintained over time. Regarding the external validity only one threat was identified. It is related to the representativeness of the evolving program families. To reduce this threat we selected industrial program families that have a satisfactory complexity and significant size to be evaluated by the mapping expansion heuristics (Section 4.6.1).

4.8 Integration of the Mapping Heuristics with a Visualization Tool

The mapping heuristics also presented false positives and false negatives, as discussed in Section 4.6.3. For this reason, a graphical support has to be provided to help developers check and visually analyse the feature mappings. In this context, the mapping expansion heuristics also fostered advances on the visualization of evolving program families.

Existing diff and visualization tools only support the representation of modular structures in a program, such as packages, classes and methods (Cederqvist 1993, Pilato 2004, Voinea *et al.* 2005, D'Ambros *et al.* 2009). They do not support the visual representation of features scattered through the program; even worse, they do not enable to visualize the evolution properties of the feature code. The mapping expansion heuristics enable developers to visualize the differences among the feature mappings of the family member versions in an attractive and easier manner under different graphical visualization perspective. The visualization of the mapping evolution was achieved through the integration of the mapping expansion heuristics with a visualization tool named SourceMiner Evolution (SME) (Novais *et al.* 2011). This integration is relevant in software maintenance tasks as industrial software projects often involve the analysis of how one or more features evolved in the code history (Corbi 1989, Bennett and Rajlich 2000). The comprehension of feature evolution is far from trivial as it relies on: (i) selecting versions and features of interest of a given application, (ii) understanding the implementation elements (e.g. methods and attributes) that realize a feature across multiple versions, and (iii) identifying feature dependencies that emerged during the family application evolution.

To tackle these problems, we defined the so-called proactive and

interactive visualization strategy to enable feature evolution analysis (Novais *et al.* 2012). Figure 4.13 illustrates the integration of the mapping expansion heuristics, so called proactive phase, with the visualization tool, so-called interactive phase. This proactive and interactive visualization strategy helps developers in several tasks in the context of software evolution, such as: (i) understanding the evolution of features in terms of their implementation elements; (ii) comparing any versions of an evolving member of a program family; and (iii) interacting with the views.

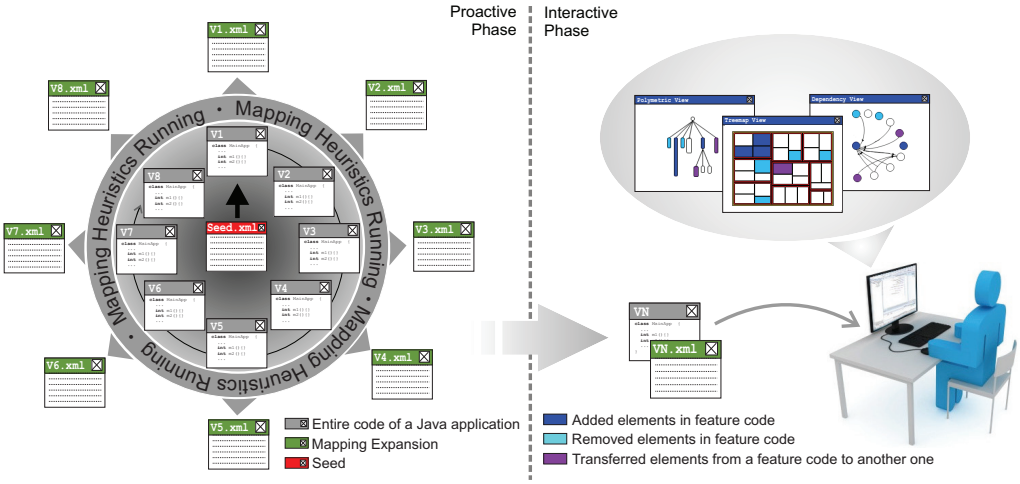

Figure 4.13: Proactive and Interactive Visualization Strategy.

The SME tool provides three views to support the feature evolution analysis. Each view provides means for analysing the feature evolution under different perspectives: structure, inheritance and dependency. The first view is based on a Treemap, which is a hierarchical 2D visualization that maps a tree-structure into a set of nested rectangles. This view addresses the structural perspective as it reveals how the software is organized into packages, classes and methods. The second one is the Polymetric view, which addresses the inheritance perspective. This view shows which classes extend others or implement certain interfaces. The third view aims to portray the dependency among the modules of an application. It uses interactive directed graphs to describe coupling between software's modules, in this case, software modules that depend on each other. These views are produced directly from source code and feature mappings produced by the mapping heuristics.

The SME tool produces interactive visualizations that allow developers to select and compare any two versions of a given family member. The developer can select any color and associate it to existing mapped feature. SME paints the implementation elements that realize a feature with the color selected by the developer. The developer uses a range bar slider to select any two sequential versions to observe the differences between them (Novais *et al.* 2011). The

three views will portray the elements of the most recent selected version and compare it with the other one. As an example, consider two sequential versions i and $i + 1$. The elements of version $i + 1$ are drawn in the views, and colors are used to portray their differences from the previous one. Three different colors are used to show developers the differences among the versions of the family's members. These colors are: light blue, dark blue and purple. SME paints in light blue color the implementation elements that realize features in version i but are removed in version $i + 1$. On the same token, if an implementation element realizes a new feature in version $i + 1$, it is painted in dark blue. A purple color is used to represent an element removed from a feature in version i and added to another feature in version $i + 1$. The last case, many times involve elements that have been removed from one feature and added to another one, or vice-versa. For this reason, these elements are named as transferred. Figure 4.13 shows the views and the colors used. SME also allows developers interacting with the views as developers can choose if they accept or not that a given implementation element, suggested by the heuristics during the expansion process of the mappings, is mapped to a feature.

Figure 4.14 illustrates an example of the Treemap view. It shows the evolution of the `TransactionService` class from version i to $i+1$. This example was extracted from the OC program family (Section 4.6.1). In `TransactionService` class there are some methods that realize the feature *Transaction*. We can observe that the method `shutdown()` was added to the version $i+1$ to realize the feature *Transaction*. On the other hand, the method `getProxy()` was removed from the feature *Transaction* realization in the code in version $i+1$. The heuristics detect both added and removed elements during the family member evolution and generate the feature mappings that are used by the views in SME. In this sense, the light blue color represents the method `getProxy()` and the dark blue color represents the method `shutdown()`.

In fact, we could observe how important the feature mappings are useful for a variety of software maintenance tasks by means of the integration of the mapping expansion heuristics with a visualization tool (Novais *et al.* 2012). This integration can also be used to eliminate the occurrences of false positives and false negatives in the feature mappings. We run two experiments and observed that the existence of feature mappings is useful to the feature evolution comprehension in evolving program families. As a consequence of the feature evolution comprehension, developers can implement code refactorings or even the complete reengineering of the program family. When developing the program family reengineering, it is also possible to recover its architecture. Additionally, developers are able to analyse the impact of a given refactoring

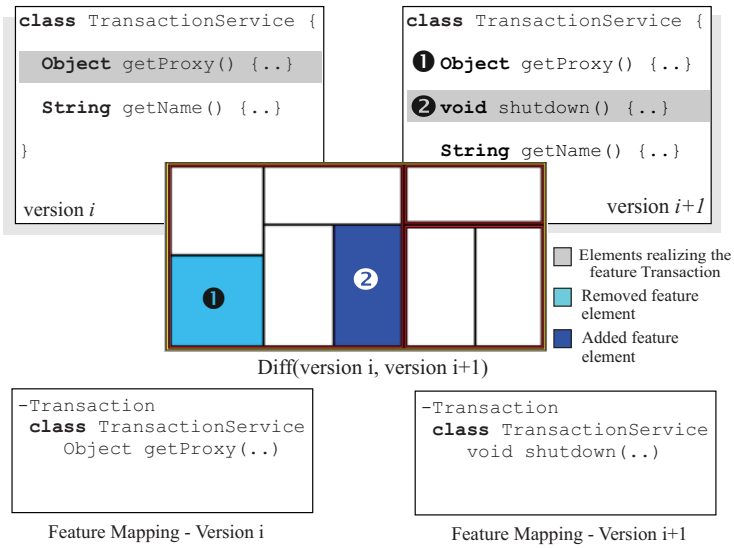

Figure 4.14: Evolution of the feature Transaction in the Treemap view.

over other modules through of the feature mapping visualization (Section 2.2.1). It is not the scope of this thesis to describe the aforementioned experiments. We intend to analyse in the future the use of this integration in the refactoring tasks of degenerate program families.

4.9 Summary

The software maintenance and evolution tasks require developers to explicitly identify all the implementation elements that realize each feature before implementing changes on it. The identification of the features' implementation elements is even more essential when developers need to analyse and maintain evolving program families. For instance, developers may need to understand how the features have evolved over time in a program family. Producing feature mappings of evolving applications of a same program family tends to be a cumbersome and time-consuming task for developers. Therefore, the process of expanding automatically feature mappings when considering evolving program families is valuable.

By analysing the state-of-the art of feature mapping techniques, we verified that there is no support to analyse the feature code and expand feature mappings during the program family evolution. In this chapter, we defined and formalized a suite of five mapping heuristics for expanding feature mappings in evolving program families (Section 4.4). They rely on a multi-dimensional historical analysis which encompasses both horizontal (versions) and vertical histories (family members). Additionally, these heuristics are based on the catalogue of mismatches (Section 4.3). We also design and implement a

prototype tool, the so-called MapHist, that supports the use of the proposed heuristics (Section 4.5).

The mapping heuristics are able to help developers understand how the feature code has been evolved in evolving program families. We also successfully evaluated the accuracy of the proposed mapping expansion heuristics in two industrial program families named OC and RAWeb program families (Section 4.6). In summary, our results demonstrated that our heuristics achieved good results in terms of precision (from 93% to 100%) and recall measures (from 66% to 100%). The variation of such results showed the accuracy of the mapping expansion heuristics when comparing with previously variations reported by related work (Section 4.6.4). Finally, the integration of the mapping expansion heuristics with the SME tool has been successfully used for feature evolution comprehension in program families (Section 4.8). This integration allows developers visualizing and analysing the feature evolution through multiple views.