5
Recovery Heuristics of Feature Elements

As discussed in Chapter 1, when a program family evolves over time, its source code might be degenerated and thus makes it difficult to maintain the members of the family. This degeneration is often induced by features that are changed individually without considering other family members. For this reason, it is important to identify and classify the implementation elements realizing the family features. There is a growing body of relevant work that have proposed techniques and tools to help developers explicitly identify code related to a feature as presented in Chapters 1 and 2. However, existing techniques that support the feature mapping activity are limited as they only consider the history of a single member product (Chapter 4). More specifically, there is no work that explicitly classifies features' implementation elements by taking into consideration the multi-dimensional history of a given program family.

To tackle this limitation, this chapter presents recovery heuristics to automatically classify the variability degree of each family element in degenerate program families. This chapter answers the third and fourth research questions of this thesis (RQ3 and RQ4 in Section 1.3). The proposed recovery heuristics use as input the feature mappings generated by the mapping expansion heuristics, i.e. the expanded feature mappings (Chapter 4). These heuristics rely on the analysis of the program family's history and the family's features are recovered as Java project packages, which separate the implementation of common features from variable ones (Section 5.2). This chapter also reports a systematic evaluation of the accuracy of the recovery heuristics in the context of two industrial evolving program families (Sections 5.4 and 5.5). Finally, Section 5.8 summarizes this chapter.

5.1 Recovery Methodology

The recovery heuristics also rely on the multi-dimensional historical analysis as well as the mapping expansion heuristics (Chapter 4). The recovery heuristics add a new step (Figure 5.1) to the methodology of the mapping expansion heuristics (Section 4.3.1), which is the *forward recovery*. Considering that extension, the recovery heuristics use as input the expanded feature mappings generated by the mapping expansion heuristics (previous step). The expanded feature mappings refer to all the versions of each family member after analysing both horizontal and vertical histories of a program family. More specifically, both horizontal and vertical histories are explored by the recovery heuristics through a set of feature mappings provided by the mapping heuristics (Chapter 4). There is a feature mapping for each version of the family member; it contains a list of features and the respective code elements for each feature realization within each family member version. For each family member, the "horizontal" set of feature mappings captures the evolution of the family's features in this specific member. The recovery heuristics use as inputs of their analyses all the horizontal feature mappings of all the family members. These feature mappings are essential to explore the multi-dimensional history of program family (Section 4.3.1).

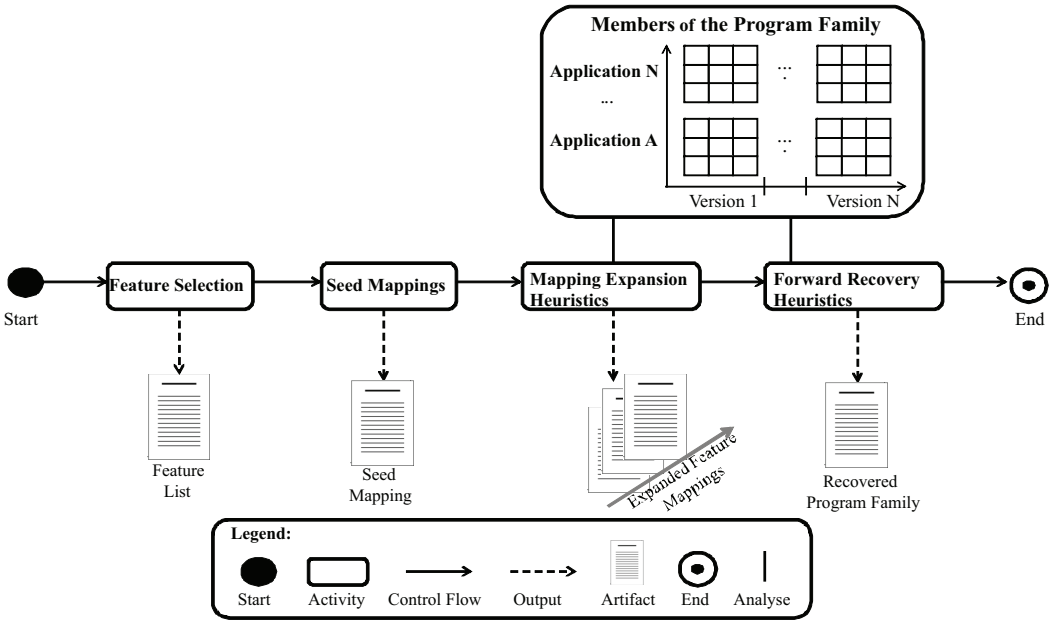

Figure 5.1: Methodology of the Recovery Heuristics.

5.2 Forward Recovery Heuristics

This section describes and formalizes the heuristics required to classify the implementation elements realizing the features of a program family (Sections 5.2.1 and 5.2.2). It also overviews the algorithmic solution and implementation of the classification process (Section 5.3). The aims of the recovery heuristics are to forward classify the code elements as part of common or variable features of the program family. This classification can be useful to help developers in the future, for instance, to: (i) determine how the existing family members departed from the original intended design, and (ii) implement code or design refactorings of the program family. Therefore, the output generated by the heuristics can be useful to circumvent the degeneration symptoms (Section 1.1). More specifically, the heuristics generate a Java project that is structured in terms of program family's features. The family's features in this project are firstly structured into two packages: common and variable. These packages are further decomposed into sub-packages that represent the categories defined by each heuristic (Sections 5.2.1 and 5.2.2). For each feature, its corresponding package contains recommended code elements forwardly detected by the historical analysis of all the family members.

The description of the heuristics is presented in the next sections in terms of: (i) their strategy and rationale, and (ii) an illustrative example and formalization. The following terms are defined and used in the formalization of the heuristics in the next sections. A feature as *fe* in F(P), where F(P) represents the set of selected features of the program family (P); *ie* as a code element, which can be an attribute, a method or a class; FM(*fe*) refers to the feature mapping, which contains the code elements realizing each feature; the number of family members (n) and their versions (v) involved in the recovery process.

5.2.1 Recovering Elements in Common

Strategy and Rationale. The first heuristic, named REC (Recovering Elements in Common), is responsible for classifying common code elements of a family; they refer to the code in common shared by the family members. For each family member, the common elements refer to code elements included in a specific version (not necessarily the first one), which remain untouched in the next versions. In this context, REC suggests that such common elements should be recovered as a part of the program family's core. For each feature, REC analyses the features' code elements of all the members' versions of the

program family. As aforementioned (Section 5.1), there is one feature mapping for each version of the family member. REC keeps track of the elements and analyses which ones are originally common to all the members' history.

To classify a code element in common across the family members, we analyse the elements' signature; we also analyse the element body in case of methods. The goal is to reveal similarities or differences between the elements. For instance, let's suppose that a method body was not changed but either the method name, or a parameter or a return type was modified. This means that the method's signature was changed, and consequently it is not considered as a common element. The granularity levels of the analysed elements are attributes, methods and classes. Ideally, common elements in the original family design should be modularly separated in the family core code; this strict separation should prevail along the family code history (Section 1.1). Nevertheless, these common elements in degenerate program families might have been moved to code realizing variable features in specific versions of one or more family members (Section 1.1). REC analyses the feature mappings of each program family version to detect these problems and infer a new set of common elements that should be part of the recovered program family.

The heuristic REC relies on four categories in order to find and classify common elements under the multi-dimensional history perspective: (i) *full vertical and horizontal similarity*, (ii) *full vertical similarity*, (iii) *full horizontal similarity*, and (iv) *partial vertical and horizontal similarity*. All these categories of common elements share a basic characteristic: the code elements, once included in the history of each family member, remained untouched in the following versions. The aforementioned categories differ in a way that common elements emerged along the horizontal and vertical histories of the program family. The categories are defined below and they are ordered based on the similarity degree of common elements; those categories representing higher similarity degrees are presented first.

Full Vertical and Horizontal Similarity. This category captures the implementation elements of a feature that are present in all family members and remained untouched in the next versions of each member. This category refers to the implementation elements that have been included to realize a certain feature since the beginning of the program family development. This means that such elements are originally common to all the vertical and horizontal histories of the family members. Figure 5.2 illustrates this category where a given feature element was introduced in the first version of the program family and it takes part of the vertical and horizontal histories.

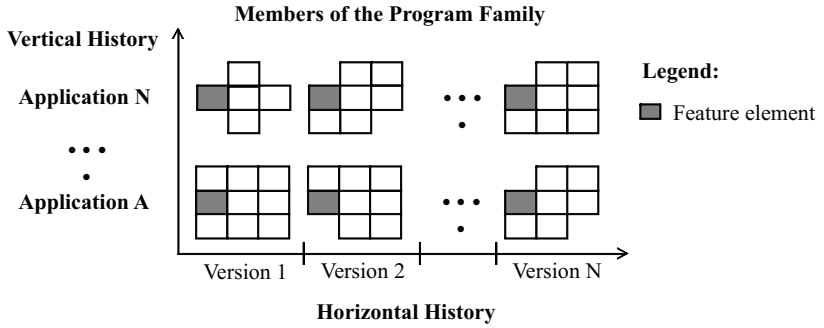

Figure 5.2: Illustrative Example of Full Vertical and Horizontal Similarity.

Full Vertical Similarity. This category captures the code elements of a feature that are fully present in each family member, i.e. they fully participate in the vertical history of the program family. They might have been introduced at any horizontal point of each family member history. In other words, this category comprises the feature's code elements that were included in each family member consistently; once they were included, they remained untouched for the rest of the horizontal history of each family member. This category provides developers with a list of code elements that are fully in common across all the family members. Figure 5.3 illustrates this category where the same feature element was included in version 1 of *Application A* and version 2 of *Application N* and it remained untouched for the entire horizontal history of both family members.

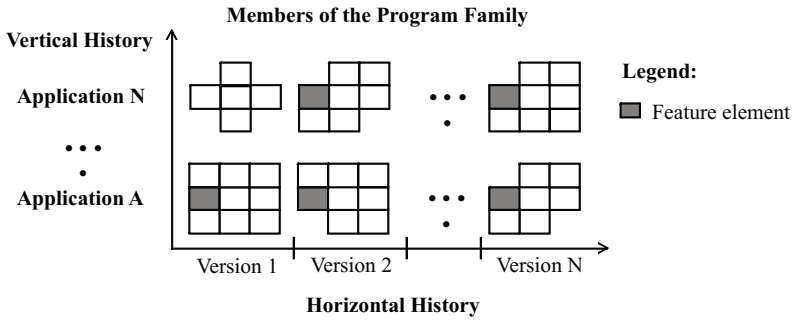

Figure 5.3: Illustrative Example of Full Vertical Similarity.

Full Horizontal Similarity. This category captures elements of a feature that are fully present in the horizontal history of a given family member. More specifically, these elements were included in the first horizontal point (first version) of a given family member and remained untouched during its entire horizontal history. This category is useful to the developers in order to reveal the common elements during the entire horizontal history of the family members. Figure 5.4 illustrates this category where the a given feature element was included in the first version of *Application A* and it remained untouched

for the entire horizontal history of this family application.

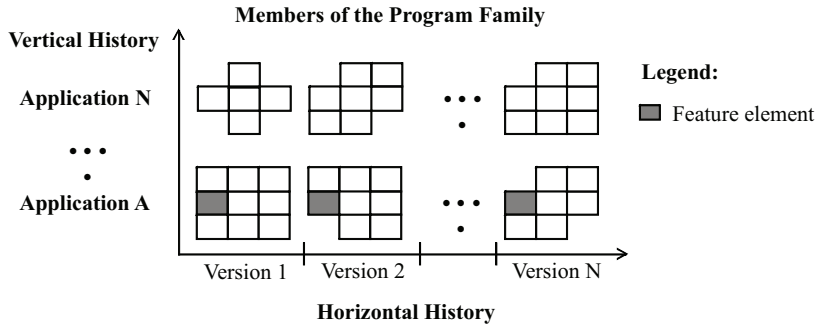

Figure 5.4: Illustrative Example of Full Horizontal Similarity.

Partial Vertical and Horizontal Similarity. This category captures the code elements of a feature that are present in a set of versions of the family members. This occurs because these elements are not fully present in all the horizontal and vertical histories of the program family. For this reason, this category is different from the aforementioned categories. Also, it is based on a threshold defined by the developers to find partial common code elements that is explained below. The assumption is that if code elements are present in many versions of the family members, this situation potentially indicates that they should be considered as common elements of the program family's core in the future. There are two possible cases of elements captured by this category. First, code elements that were included in any horizontal point of the family members and remained untouched in the next versions. These elements were originally included to realize an existing feature in a given member version; then, they started to be included in other versions of the same member and, also, in other versions of different members to realize the same feature. Second, it refers to the code elements of a feature that were removed from some versions of family members. However, these elements remain to realize the same feature in the rest of the family members. Figure 5.5 illustrates this category by showing a feature element that participates in the some versions of the program family history.

Formalization and Examples. The formal definition of REC is presented in Formulas 5-1 and 5-2. These formulas encompass all the categories defined above. It is defined in terms of number of family members (n), the member versions (v); a given code element (ie) mapped to a feature (fe); and a threshold (α) that defines the number of versions in which a given element should be present in when analysing all the family members. The threshold is used by the full vertical and horizontal similarity and partial vertical and horizontal similarity categories. This formalization, considering each category, is explained below.

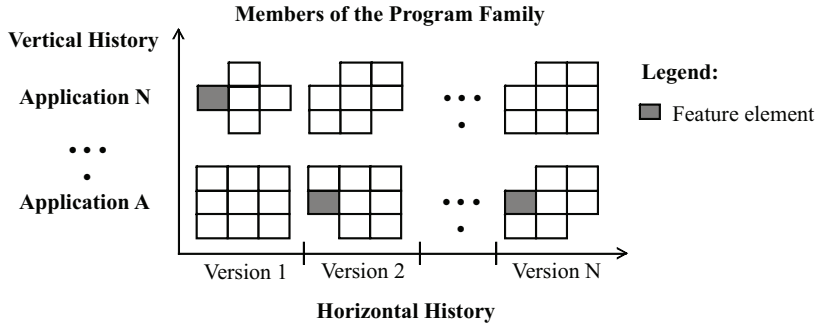

Figure 5.5: Illustrative Example of Partial Vertical and Horizontal Similarity.

$$REC(fe) = \bigcap_{i=1}^{n} \bigcap_{j=1}^{v} \{ie : ie \in FM(fe)\} \quad (5\text{-}1)$$

$$REC(fe) = \bigcap_{i=1}^{n} \bigcap_{j=1}^{v} \{ie : ie \in FM(fe)\} \geqslant \alpha \quad (5\text{-}2)$$

Code elements belong to the full vertical similarity category when they are present in all family members, which is represented by n in Formula 5-1. The full horizontal similarity category captures code elements that are present in all the versions (v) of one family member (n) in Formula 5-1. Elements are captured by the full vertical and horizontal similarity category when they are present in all family members and remained untouched in the next versions of each member. Finally, elements are captured by the *partial vertical and horizontal similarity* category if they are present in a particular number of program family versions based on a given threshold α, as described in Formula 5-2. This threshold comprises the minimum number of program family versions in which elements should take part to be classified as *partial*. The definition of the threshold for the *partial vertical and horizontal similarity* category is useful as developers may want to analyse specific members of the program family (i.e. a subset of program family versions). In addition, this threshold is also useful as depending on the analysed family, this value should vary to achieve a good accuracy regarding the captured elements.

Figure 5.6 illustrates an example of code elements that comprise the full vertical and horizontal similarity category. Considering the first three applications of the program family in Figure 5.6. For each application's version there are the feature mappings, which contain the features and their respective code elements. When analysing the evolution of the original code elements in feature mappings of each application version, we can observe that the `m1()` method, that realizes the feature *F1*, is present and untouched in all the N versions of the three family applications. Once the elements are detected as common elements of a particular feature, REC suggests them to be part of the new common elements of the recovered program family's core. Figure 5.6 also

illustrates an example of code elements detected by the partial vertical and horizontal similarity category. For instance, let's suppose that it is defined a $\alpha = 5$ (versions). This means that the elements must be present in at least five versions to be detected by the partial vertical and horizontal similarity category (REC(fe) $\geqslant 5$). Note that the methods `m2()` and `m3()` were included to realize the feature *F1* in versions 2 and 3 of Applications 1, 2 and 3, respectively. These methods (`m2()` and `m3()`) are suggested to be part of the common elements of the analysed feature in the recovered program family. This occurs because they exist in more than five versions of the program family.

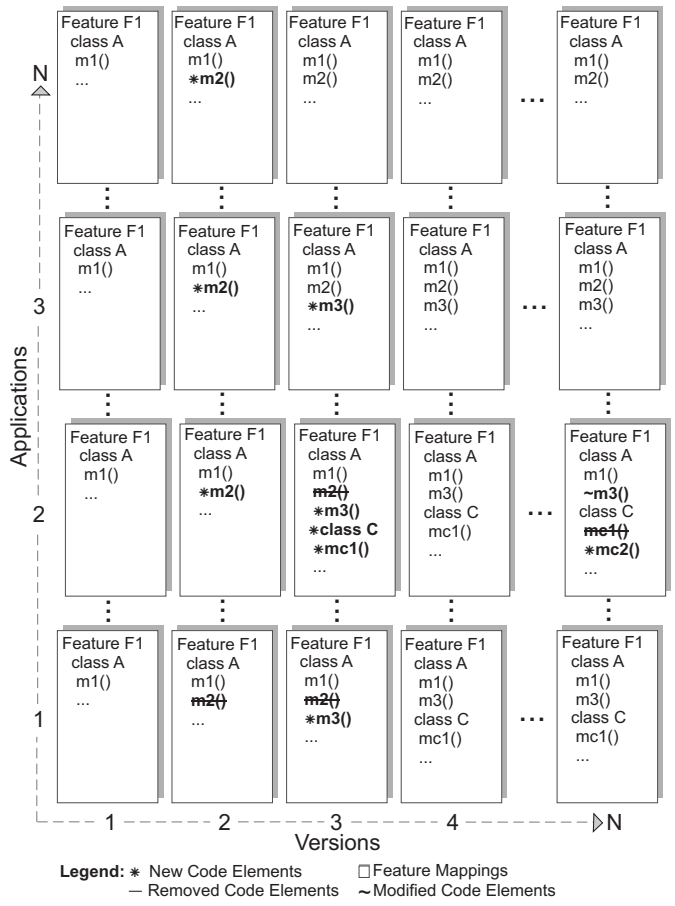

Figure 5.6: Feature Mappings of a Program Family.

Concrete Example. The heuristic REC is also described through concrete examples presented in Chapter 1 (Figures 1.2 and 1.3). The goal is to discuss how challenging the recovery tends to be to the developers when they need to classify precisely the code elements realizing the evolving common and variable features. Figure 1.2 illustrates the `ExportDialog` class in *Applications I* and *II*. We can observe that the `ExportDialog` class was modified in *Application II*. However, the `exportToXls()` method in the `ExportDialog` class has not been modified. This way, this method, that realizes the Report feature, is identified by the heuristic REC as being of the full vertical and

horizontal similarity category. Figure 1.2 illustrates a concrete example of the partial vertical and horizontal similarity category, where the feature *Scenario* was introduced in the framework code from a family application version. It is important to mention how cumbersome the identification and classification of the code elements of an evolving feature in program families. In particular, this difficult tends to be even more exacerbated due to degenerate nature of program families.

5.2.2 Recovering Variable Elements

Strategy and Rationale. The second heuristic, named RVE (Recovering Variable Elements), is responsible for classifying variable code elements that realize the features. The key characteristic that distinguishes variable elements from common ones is the following: they represent code elements that are modified or removed in the next versions of a family member after they were included in a given version of this member. Therefore, they are not present in the entire horizontal and vertical histories. Hence, RVE captures code elements of a feature that were modified or removed along the family history, exactly the opposite of REC. The idea of this heuristic is to observe the variable elements when analysing the horizontal history of each family member. It is important to mention that REC is first run, and consequently, it has priority over RVE. In this case, a given element is only captured by one heuristic. The same element is not captured by both heuristics. We did this choice in order to prioritize the code elements that can be shared by all the family members.

The rationale of this heuristic is contrary to the partial vertical and horizontal similarity category, which analyses the common elements taking part of the other family members. For instance, these variable elements might have been added in a family member version to implement new requirements of an existing feature (Figure 1.2 in Section 1.1.2). After that, they were changed or removed in the next versions of the same family member with the goal of customizing specific versions. The idea behind this heuristic is to analyse the elements of a feature and observe their changes throughout the horizontal histories of the family members. We defined two categories to find variable elements of a feature: horizontal variability and vertical variability.

Horizontal Variability. This category captures code elements that were included to realize an existing feature in any horizontal point of a given family member but they do not participate in its entire horizontal history. This occurs because these elements were possibly modified or removed in the next versions of the family member. For this reason, this category is different from the full

horizontal similarity category (Section 1.1.2), which captures elements that remained untouched during the entire horizontal history of a family member. Figure 5.7 illustrates this category by pointing out a feature element that was introduced in version version 2 of *Application A* and modified in version N.

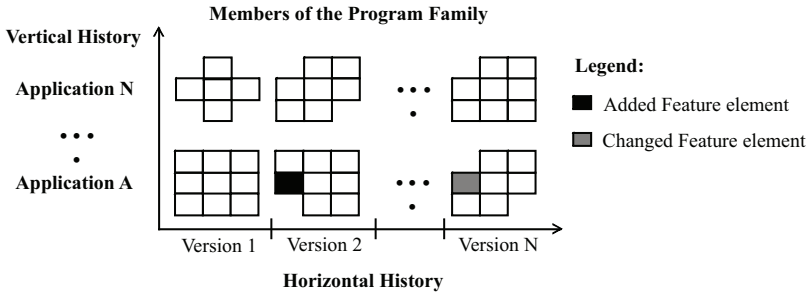

Figure 5.7: Illustrative Example of Horizontal Variability.

Vertical Variability. This category captures code elements of a feature that are present in a few family members. In other words, they do not participant in the entire vertical history of the program family. These elements were introduced at one specific horizontal point of a family member but were not consistently included in all the family members. This occurs because these code elements were removed or modified across all the family members. This category is fundamentally different from the full vertical similarity category that captures common elements that participate in the vertical history of the family (Section 1.1.2). Figure 5.8 illustrates this category by showing a feature element included in version 2 of *Application A* and that was modified in version 2 of *Application N*.

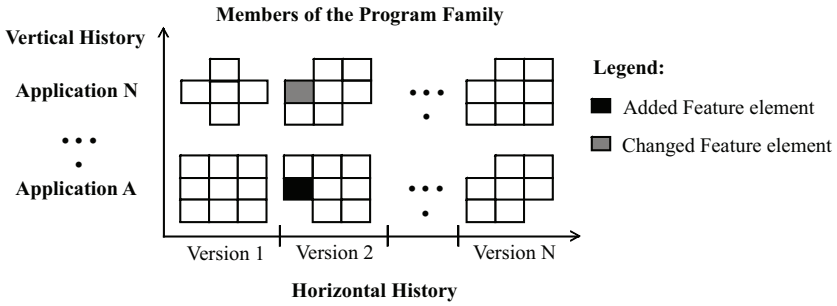

Figure 5.8: Illustrative Example of Vertical Variability.

Formalization and Example. The formal definition of RVE is presented in Formula 5-3. The formalization of RVE is defined if there is a given code element that belongs to the current version of the feature mapping (j) but it does not belong to the previous one (j-1) or to the next one (j+1). In other words, this element is not present in the entire horizontal history of the members and vertical history of the program family.

$$RVE(fe) = \bigcup_{i=1}^{n} \bigcup_{j=1}^{v} \{\exists ie \in FM_{ij}(fe) \cdot (ie \notin FM_{ij-1}(fe) \vee ie \notin FM_{ij+1}(fe))\} \quad (5\text{-}3)$$

Figure 5.6 illustrates an example of code elements classified as variable elements of a feature. For instance, let's consider Applications 1, 2 and 3 in Figure 5.6, respectively. Note that in version 3 of the Application 2 the `C` class and the `mc1()` method were included. In version N of Application 2, the `mc1()` method was removed and the `mc2()` method was included within the `C` class. In this example, the `mc1()` and `mc2()` methods are captured by the horizontal variability and vertical variability categories, respectively. It is possible to observe that variable elements of a feature take part of some versions of a family application but they are not present in its entire horizontal history. Therefore, RVE is able to detect the variable elements of the features by analysing the changes and removals of such elements during the evolution of family applications. After that, it suggests these elements to be part of the feature variable code of the recovered program family.

Concrete Example. Figure 1.2 (Chapter 1) illustrates how the feature Report was changed to include the `XLSFilter` internal class in the `ExportDialog` class in a specific version of *Application II* (Section 1.1.2). This internal class is identified by the heuristic RVE as it is present in a specific version of *Application II*, which characterizes a vertical variability. In this case, RVE captures this internal class and suggests it to be part of the variable elements of the feature *Report*. The more family applications and their many versions in a degenerated program family, the more difficult is for the developers to classify the variable elements that realize the features.

The result generated by both heuristics REC and RVE is a Java project with two macro packages labeled common and variable, as illustrated in Figure 5.9. These two macro packages represent the classification of the features. The feature is classified as *common* if it exists in all the mappings. Otherwise, it is classified as *variable*. The feature *F1*, illustrated in Figure 5.6, is classified as common. There are also sub-packages defined by the heuristics. Then, the heuristics analyse the code elements of each feature and include them in the respective packages: *common.fullvh.F1* (full vertical and horizontal similarity), *common.partialvh.F1* (partial vertical and horizontal similarity), *common.vvariability.F1* (vertical variability) and *common.hvariability.F1* (horizontal variability). We can observe that the code elements included in the Java project are classified and grouped according to the categories proposed by the recovery heuristics REC and RVE (Figure 5.9).

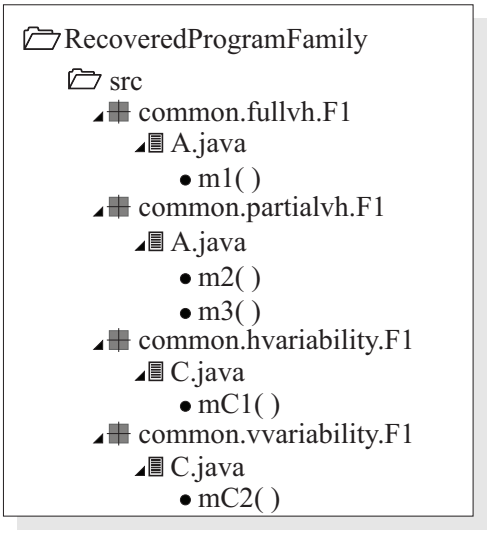

Figure 5.9: Java Project of the Recovered Program Family.

5.3 Algorithm Solution and Implementation

This section describes the algorithm solution used by the recovery heuristics and their implementation. Code 7 describes the main algorithm, which encompasses three key parts.

1. **Analyse the mappings**. First of all, the feature mappings of all the members' versions of the program family are used as input information in the recovery process (line 01). These feature mappings are produced for all the versions of the family members by using the mapping expansion heuristics (Chapter 4). The feature mappings could be produced by other feature mapping techniques (Section 2.2). For each feature of a given family member, it is verified in how many versions the feature's code elements appear. The implementation elements are analysed in detail, including their signature and their body; i.e., through a syntactic comparison. The goal is to verify if they are the same during the program family evolution. For each element, it is computed the number of versions that this element is included. This analysis is performed for each family member's history through the processing of the feature mappings. After that, all the elements are stored in a hash table (line 01). Each entry in the table contains a pair formed by the feature name and the corresponding list of code elements that realize it through the history of each family member;

2. **Calculate the occurrence rate**. The occurrence rate of each implementation element contained in `hashApps` (line 01) is verified by the

`verifyOccurrences(element)` method (line 04). Based on the occurrence rate of the elements, they are separated and included in a list maintained by each heuristic and its respective categories (line 05);

3. **Detect the elements and create the recovered project of the program family**. The last step is to compare the lists of the family members in order to classify the elements according to the categories defined by each heuristic (line 08). This comparison is performed by verifying the untouched elements and their occurrence rate in the lists. The features are also classified as variable or common. Finally, the Java project is created and the elements are added to it. Packages are labeled with the names of the categories (Figure 5.9) and the respective elements realizing each feature are added to each category (line 09).

The forward recovery was implemented as an extension of the *MapHist* architecture as illustrated in Step 5 in Figure 5.10 (Chapter 4). This was possible as the recovery heuristics rely on the expanded feature mappings of the program family generated by the mapping heuristics (Chapter 4).

Algorithm 7 Main Algorithm of the Forward Recovery

```
1. hashApps ⇐ analyseMappings()
2. for feature in hashApps do
3.     for element in hashApps do
4.         verifyOccurrences(element)
5.         setElements(element)
6.     end for
7. end for
8. compareLists()
9. generateRecoveredProject()
```

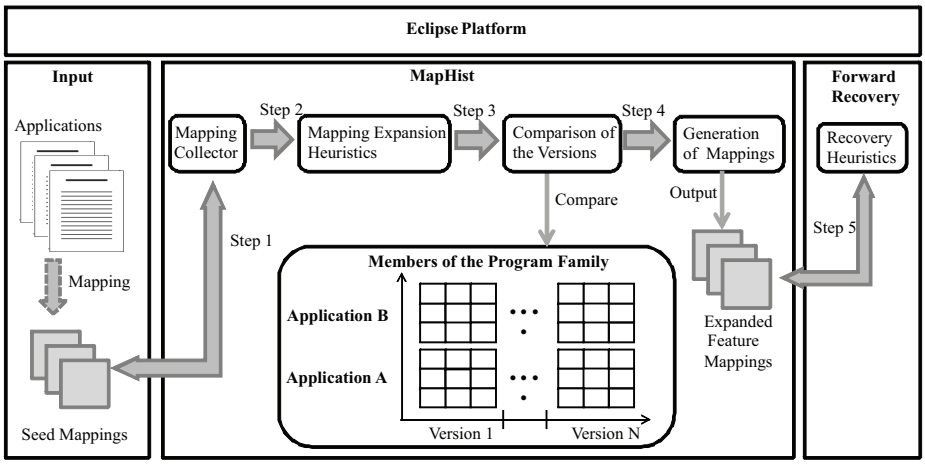

Figure 5.10: The Forward Recovery Implementation.

5.4 Assessment Methodology

This section describes the evaluation procedure of the recovery heuristics regarding both OC and RAWeb program families (Section 4.6.1). This section describes the industrial program families used in the evaluation. It also describes the procedures for evaluating the recovery heuristics. The evaluation focuses on both (i) the accuracy of the recovery heuristics in terms of precision and recall measures and (ii) examples of how the developers can use the classification of heuristics-based features' code elements to restructure the program family' code. Five main procedures were followed to assess the recovery heuristics:

1. **Seed Mappings**. The feature mappings are provided as input to the recovery of the program family. In order to expand the feature mappings developers need to provide seed mappings as input to the MapHist tool, which supports the mapping expansion heuristics (Chapter 4). As explained in Chapter 4, the seed mappings are used as basis to the mapping expansion (Step 2);

2. **Expansion of Feature Mappings**. The expansion of the feature mappings is realised in this step. Feature mapping expansion involves the automatic identification of feature elements in the code departing from the seed mappings (Step 1). As mentioned in Chapter 4, the mapping heuristics take into consideration the provided seed mappings and produce the feature mappings of each version of the family members. We discuss the implications of the feature mapping imperfections in the accuracy of the recovery heuristics (Section 5.5);

3. **Forward Recovery Heuristics**. We run the recovery heuristics (Section 5.2) in order to classify the features' code elements of the selected program families. At the end of the recovery process, a Java project is created with packages identifying: (i) the classification of each feature (common or variable), and (i) the classification of its code elements based on the categories defined by the recovery heuristics (Sections 5.2.1 and 5.2.2);

4. **Evaluation of the Results**. The Java project, which represents the recovered program family, was double-checked with the developer responsible for producing the seed mappings of both program families. Two developers analysed the recovered project of the OC program family and one developer analysed RAWeb. The goal was to evaluate the accuracy

of the identified code elements realizing the respective features (Section 4.6.1). To perform this evaluation, we provided developers with the list of recovered elements per feature and asked them to analyse their correctness. The developers analysed the recovered features' code elements based on their history throughout the program family. Also, they used their experience and knowledge of the program families' source code to identify the missed elements of a feature. For instance, a recovered element was observed for all the family members in order to check if it was present in all of them or was modified in any version of the family members. In this way, developers were able to verify the correct and missed elements comprising the features of each category;

5. **Accuracy of the Forward Recovery Heuristics**. After the developers assessed the recovered features' code, the accuracy of the recovery heuristics was measured by calculating recall and precision of each category per feature. The purpose of recall measures is to verify for each category if the heuristics are able to classify all the code elements of each feature. We focused on the presentation of results for the following categories: *full vertical and horizontal similarity (F)*, *partial vertical and horizontal similarity (Pt)*, and *horizontal and vertical variability (V)*. The purpose of precision is to verify if the heuristics are able to classify only the code elements that realize a feature based on the aforementioned categories.

5.5 Discussion

The goal of this section is to discuss the precision and recall measures of the recovery heuristics. This enabled us to analyse their accuracy during the recovery process. The results are presented individually for each feature. We also discuss how the classifications are useful for the family developers to understand the family evolution and make certain decisions. Before running the recovery heuristics, we set the threshold for the *partial vertical and horizontal similarity* category. In the case of the OC program family, we defined $\alpha = 10$, REC(fe) $\geqslant 10$ whereas for the RAWeb we defined $(\alpha) = 7$, REC(fe) $\geqslant 7$. These thresholds were chosen to capture half of selected versions of the program families. We made this choice as they are closely related to the explanation described in Section 5.2.1 to classify potential common elements present in the program family history. However, optimal values for the thresholds may obviously vary depending on the set of chosen members and versions (Section 5.7).

	Features	# Identified Elements			Recall (%)			P
		F	Pt	V	F	Pt	V	(%)
OC	Logger	769	13	676	92	100	100	71
	Route	595	0	598	100	N/A	85	87
	Notification	785	14	249	100	N/A	100	93
	Export	140	0	631	97	N/A	100	95
	Stock	0	0	2913	N/A	N/A	100	94
	Importation	0	0	2568	N/A	N/A	100	98
	Persistence	365	0	554	100	N/A	100	95
	Report	666	19	1340	100	100	100	96
RAWeb	Dynamic Forms	1917	806	894	97	100	92	91
	Dynamic Tables	1397	670	1169	97	100	100	86
	Search Data	383	192	551	95	100	100	98
	Import Image	0	0	150	N/A	N/A	100	100
	Report	341	410	429	100	100	100	99

Figure 5.11: Precision (P) and Recall (R) of the Full (F), Partial (Pt), and Variable (V) Categories.

The forward recovery classifies and includes all the code elements that contribute to the realization of a given feature. First of all, the features were classified into common or variable (Sections 5.2.1 and 5.2.2). Then, the elements are further classified into the aforementioned categories (F, Pt or V) being assessed. Figure 5.11 shows the recovery results of each feature regarding the number of identified elements, the recall measures based on each category (F, Pt and V), and the overall precision measure of the categories. The identified elements refer to attributes and methods of the program families' classes. The common features in OC are *Route*, *Logger*, *Notification*, *Persistence*, *Report*, *Export*. The variable features are *Stock* and *Importation*. In RAWeb, the common features are *Dynamic Forms*, *Dynamic Tables*, *Search Data*, *Persistence* and *Report*; and the variable one is *Import Image*.

Feature Recovery Accuracy. The heuristics have demonstrated to be accurate when recovering either non-functional (crosscutting) features or functional features. We can observe that the recall measures ranged from 92% to 100%. For instance, the lowest values of the *horizontal and vertical variability* category are related to the features *Route* and *Dynamic Forms*. On the other hand, for the other features, the recall measures for detecting the variable elements were 100%. The variable code is easier to detect because they have explicit dependencies to classes present in the base code. In addition, these classes generally had already been previously recovered in the context of categories F and Pt. This scenario can be observed through the example illustrated in Figure 1.2 (Chapter 1), where the method `exportToXls()` in the `ExportDialog` class was captured by the *full vertical and horizontal similarity*

category (F). Consequently, the heuristic RVE is able to observe that the `ExportDialog` class was modified and, hence, classify the modifications (e.g. the internal `XLSFilter` class in Figure 1.2) as variable elements. Even though the recall measures were not 100% for all the features, they have presented high accuracy in the recovery of feature code. Therefore, it seems that our proposal for forward recovery provides developers with a reasonably-reliable way to understand the program family evolution (Section 5.2).

Vertical and Horizontal Variability. The precision measures were higher than 90% in 10 out of 13 features. More specifically, for the variable features *Stock*, *Importation* and *Import Image* the heuristics successfully classified and identified all the code elements as variable elements. Additionally, the recall measures were also high for this category of variable features. In particular, these features are present in only two members of the family. Their elements were only evolved in these family members and thus they remained correctly classified as vertical and horizontal variability throughout the program family evolution. As a result, there were not elements that emerged during the program family evolution and modified the essence of these variable features.

Full and Partial Vertical and Horizontal Similarities. Regarding the precision measures of the common features, the results ranged from 71% to 99%. The lowest values are related to the features *Logger*, *Route*, *Dynamic Tables*. This means that a set of elements were detected, but they do not precisely realize these respective features. It is possible to notice that there were elements identified by the categories F, Pt and V that realize the features *Logger* and *Dynamic Tables*. We could observe that most of the cases of code elements identified and captured by the *full vertical and horizontal similarity* category (F) refer typically to the interfaces and abstract methods that are specialized by the family applications.

For instance, the entire `PersistenceManagement` interface is entirely dedicated to realize the feature *Persistence* in Code 5.1 and, therefore, it is classified as (F). This interface remained untouched since the beginning of the program family history. For some cases, interfaces like `PersistenceManagement` realizing the features are untouched throughout the vertical and horizontal histories of the program family. Therefore, the `PersistenceManagement` interface is included in the package named *common.fullvh.persistence*. This category (F) helps developers: (i) to understand which code elements have been originally used for all the family members and (ii) to implement separately the code elements like the `PersistenceManagement` interface as being part of the new family core code. On the other hand, we also observed cases where interfaces

and abstract classes are redefined typically to change the return type or the implementation of concrete methods. As a result, these elements turn into partial (Pt) or variable elements (V). Therefore, the categories (F, Pt and V) provide insights to the developers of which elements should be restructured as potential common or variable elements in the family's source code.

Code 5.1: Full Common Code (F) of the feature Persistence.

```
01 public interface PersistenceManagement {
02   void begin();
03   void open();
04   void commit();
05   void close();
06   ...
07 }
```

Feature Code of Specific Member Versions. It is related to the common code that was modified for specific versions of a family member. For instance, all the detected elements realizing the feature *Route* were classified as full (F) and variable (V). We noticed that most of the elements classified as (F) are implemented in the framework code and they are massively used for only one family member. The implementation of the feature *Route* reflects the same scenario illustrated in Figure 1.3 (Chapter 1). In this scenario, there are classes in the framework code that are used exclusively for one family member. Another example is related to the feature *Export*, where some common elements were changed by specific versions of the family applications and turned into variable elements (V). This scenario reflects the same problem illustrated in Figure 1.2 (Chapter 1), where the common element is modified for accommodating a new requirement in a family application.

5.6 Usefulness of the Proposed Categories

This section discusses how the categories of the recovery heuristics can play an important role in the developers' decision making during the restructuring process of the program family's source code.

The *full vertical and horizontal similarity* category is useful to help developers observe and understand the implementation elements that remained untouched during the program family's history. For instance, Code 5.2 illustrates a piece of code that realizes the feature *Dynamic Forms* in RAWeb program family. Developers can use such an information to make sure of which elements should be present in the new program family's core during the restructuring of its source code. Therefore, the *full vertical and horizontal similarity* category positively influences the quality and accuracy of the restructuring of the program family's source code. This occurs because the proposed recovery heuristics support a systematic analysis of the program family's evolution and add more value to the maintenance tasks.

Code 5.2: Full Common Code (F) of the feature Dynamic Forms.

```
01 public class AtributoHelper {
02   public Atributo getAtributoById(Session, id) {
03       ...
04   }
05
06   public List getAtributoByClasse(Session, String id) {
07       ...
08   }
09 }
```

The *full vertical similarity* category is useful to help developers know which elements fully participate in the program family's vertical history. This category provides developers with an information about the implementation elements shared at any horizontal point by all family members. Developers can also use this potential classification with the goal of restructuring the detected implementation elements as being part of the new program family's core. On the other hand, the *full horizontal similarity* category represents a complete and thorough investigation of the code elements involved in the horizontal history of a family member. This category helps the developers understand mainly the potential implementation elements that realize the variable features and remained untouched across the entire horizontal history of a family member. As a result, developers are able to better restructure the variable features of the program family as long as the value of detailed analysis of each member is systematically demonstrated the *full horizontal similarity* category.

The *partial vertical and horizontal similarity* category provides developers with a way to analyse a set of versions of the program family. Based on this category, developers can decide if they will consider those implementation elements as being part of the program family's core or variable. For instance, Code 5.3 shows a piece of the `PdfReport` class, which realizes the feature *Report*. This class was detected by the partial category according to the threshold $(\alpha) = 7$ for the RAWeb program family (Section 5.5). This category allows developers focus on parts of the program family in order to make decisions about the restructuring of the implementation elements.

Code 5.3: Partial Common Code (F) of the feature Report.

```
01 public abstract class PdfReport {
02     ...
03     public abstract boolean exportPdf();
04     protected abstract boolean initValues();
05     protected abstract Map<Integer, List<PdfPCell>> getHeaderMap();
06     ...
07
}
```

The *horizontal variability* and *vertical variability* categories detect the implementation elements that are not present in the entire horizontal and vertical histories of the program family. These elements are classified as horizontal or vertical variability because they were only realize a given feature in a specific family member version. For instance, Code 5.4 illustrates a piece of the `PersistenceHelper` class, which is present in one member of the RAWeb family. These categories show positive effects on how developers can restructure the variability of the family's features.

Code 5.4: Horizontal Variability Code (F) of the feature Persistence.

```
01 public class PersistenceHelper {
02     importaAtributo();
03     importaRelacionamento();
04     ...
05 }
```

5.7 Threats to Validity

This section discusses the threats to validity of our evaluation.

Conclusion Validity. We identified two threats in this category: (i) the set of selected members and their versions: our recovery process is largely dependent on the selected sample of members and versions. To mitigate this threat, we tried to select members and versions that underwent a large number of changes and defined thresholds to better characterize the classification of features' code elements; (ii) the selected features: it is related to the set of features selected in our assessment. If they were not representative of different types of features, they could bias the results of the heuristics evaluation. To mitigate this threat, we selected a wide range of features, such as functional features (e.g. Notification and Dynamic Forms) and non-functional features (e.g. Logger and Persistence). These features were also chosen because they are critical in the domain of the selected program families; (iii) validation of the feature's code elements: it refers to the validation of the code elements identified by the heuristics. To circumvent this threat, we validate the results with the developers who are responsible for maintaining and evolving the selected program families; and finally (iv) the use of threshold: it is related to the use of threshold to detect code elements associated with the partial vertical and horizontal similarity category. We set the thresholds to represent the half of the versions in both program families. This is a threat as the results can vary depending on the chosen values of the thresholds.

Construct Validity. We identified two main threats in this category: (i) seed mappings: it refers to the seed mappings, which were used in our recovery process. In order to make the seeds provision realistic, we ensured that the seed mappings had a low coverage degree, ranging from 15% to 20% per feature (Chapter 4). We also mitigated this threat as we involved developers who are knowledgeable about the features' code elements; and (ii) feature mappings: it is related to the feature mappings of each member version used by the recovery heuristics. The accuracy of these mappings can directly influence the results of the recovery heuristics. However, to the best of our knowledge, we have relied on mapping expansion heuristics that exploit feature mapping techniques (Chapter 2). In addition, they have presented high precision and coverage measures for the expanded feature mappings (Chapter 4). Even in the presence of imperfect feature mappings, our recovery heuristics have shown to be highly accurate for most features (Section 5.5).

Internal and External Validity. The threat identified for the internal validity is related to the history of the evaluated program families. To reduce this threat, we chose two industrial program families, OC and RAWeb (Section 4.6.1), that have been extensively maintained over time. One threat to external validity was also identified and it is related to the choice and representativeness of the program families. To reduce this threat we selected program families that consist of several evolving members and have a noticeable complexity and significant size. This allowed us to better evaluate the accuracy of the proposed heuristics. However, other program families have to be evaluated in the future in order to provide more evidences about the accuracy of the heuristics.

5.8 Summary

The program family evolution often implies on the inclusion of new features' code elements, changes in existing ones and removals. When such changes are carried out in an uncontrolled manner, the program family source code tends to degenerate and affects important quality attributes (e.g. maintainability). As a consequence, the family members become difficult to be maintained and evolved. In particular, this occurs mainly because it is not trivial to identify and classify what code elements realize each feature in each family member. In this context, this chapter presented history-sensitive heuristics for the forward recovery of features in code of evolving program families (Section 5.2). They explore the multi-dimensional historical analysis through a set of feature mappings (Section 5.1). As a result, these heuristics generate a Java project of the new recovered program family that can be used

as basis by the developers, for instance, to implement the code refactoring of the program family. In this Java project, features are classified as common or variable and grouped in two macro packages. Within these macro packages there are sub-packages, which contain the code elements that realize each feature. Developers can easily from this recovered project to understand and restructure the features' implementation elements according to the categories defined by the recovery heuristics (Sections 5.2.1 and 5.2.2).

To the best of our knowledge, the forward recovery described in this chapter is the first attempt of supporting a specific facet of family recovery: the classification of feature variability in degenerate program families when considering its multi-dimensional history analysis. It is also the first attempt of recovery in the context of degenerate program families when considering its multi-dimensional history analysis. We successfully assessed our recovery heuristics on the top of two industrial program families (Section 5.5). The good results demonstrated the accuracy of the heuristics to classify the features' elements. They presented recall measures that ranged 85% to 100%; whereas the precision measures ranged from 71% to 99%.