Heuristic Expansion of Feature Mappings in Evolving Program Families

Camila Nunes
Opus Research Group – Software Engineering Lab, Informatics Department, PUC-Rio, Brazil
cnunes@inf.puc-rio.br
Supervisors: Alessandro Garcia and Carlos Lucena

1. MOTIVATION AND PROBLEM

A feature can be defined as a prominent or distinctive user-visible aspect, quality or characteristic of a software system [5]. Feature mapping is one of the longest-standing problems in the broad spectrum of software engineering [1][2][3][4]. This activity is key to software maintenance as it consists of explicitly identifying all code elements responsible for realizing each feature [1][2][3][4]. Feature mapping is particularly relevant and challenging when developers need to analyse and maintain program families over time [6]. A program family is a set of similar programs, named family members, which share common features [6]. The variable features represent the particularities of each family member.

There are numerous examples of well-known program families in industry, such as Adobe Acrobat [7], Mozilla [8]. In fact, software systems have been increasingly built as program families rather than stand-alone applications in a wide range of software domains [7][8][9][11][13]. These domains range from popular operational systems or games for mobile phones (e.g. Android [9], GM [11] and BestLap [13]) to train control systems [14]. Independently from the domain, program families are often a key part of organizations' economic strategy as they become more profitable [11][12]. For instance, given the growing variety of mobile devices and their varying hardware constraints (e.g. memory, screen size), embedded mobile systems are often developed as a program family [11] in order to accelerate their time-to-market.

The Role of Feature Mappings in Evolving Program Families. However, program families often evolve, and each feature implementation needs to be understood in order to realize each change. To this end, a feature mapping is fundamental to provide developers with a full knowledge about the realization of each feature and its relationships with other features in the code. Therefore, the lack of feature mappings in program families harms a wide range of maintenance activities, such as (i) understanding how a feature is implemented in a family member [15]; (ii) refactoring implementations of specific family features [16]; (iii) the complete reengineering of the program family [17][18]; (iv) the recovery of the family's software architecture [14]; and (v) the risk and cost analysis of adopting such a program family [12].

Varying Feature Mappings in Program Families. However, many factors make feature mapping in program families much harder than in single applications. First, program families often depart from one single core system but evolve to similar but different systems to accommodate various customer-specific requirements [6]. Therefore, the feature mapping of the original system may also have changed for each family member. Second, there might be cases where even the realization of the same feature evolved differently for each family member. Third, the feature mappings per family member should be produced with high accuracy. Without accurate feature mappings developers might, for instance, implement a change in non-related modules or even miss relevant modifications for a given change request [19].

A Motivating Example. Figure 1 illustrates how the feature mapping is performed when using conventional techniques for stand-alone applications [1][2][3][4]. This example is based on an evolving member of the family, which is named *Member I* due to the copyright restrictions[1]. This member is part of an industry evolving program family used in our evaluation (Section 4) and belongs to a logistic domain. For the sake of simplification, we selected only one *Member I* of this family and two of its versions. Basically, the feature *Scenario* represents exportation and importation properties of products [23][24]. This figure illustrates some elements of the `MainDesktop` class in the versions 1 and 2 of *Member I*. The developer mapped the `checkUnsavedScenario()` method that realizes the feature *Scenario* in the first version of *Member I* (see Feature Mapping - Version 1 in the right side of Figure 1). However, the attribute named `scenarioInfo` has not been mapped, which characterizes a false negative in the feature mapping. This happens because developers tend to focus mainly on the behaviour implemented by classes and methods [23]. In the second version of *Member 1*, the `MainDesktop` class was modified to extend the `BasicDesktop` class, which has not been mapped to the feature *Scenario*.

Proposed Solution. To the best of our knowledge, developers are not equipped nowadays with any kind of automated support for producing feature mappings in evolving program families (Section 2). In order to address this problem, our proposed solution leverages the notion of feature mapping expansion. Feature mapping expansion involves the automatic identification of feature elements in the code departing from an initial mapping (seed). To this end, we define and formalize a set of nine heuristics for expanding feature mappings for all the versions of the family members (Section 3). Even if developers perform manual identification of feature elements or apply existing techniques for each member version, they will produce many false positives and false negatives (Section 2). Hence, the proposed heuristics also aim at reducing the number of missing and incorrectly mapped elements through the feature evolution in the family code. We refer to these false negatives and false positives as *feature mapping mismatches* (Section 3.1).

[1] The copyright restrictions do not allow us making available the names and source code of the family members.

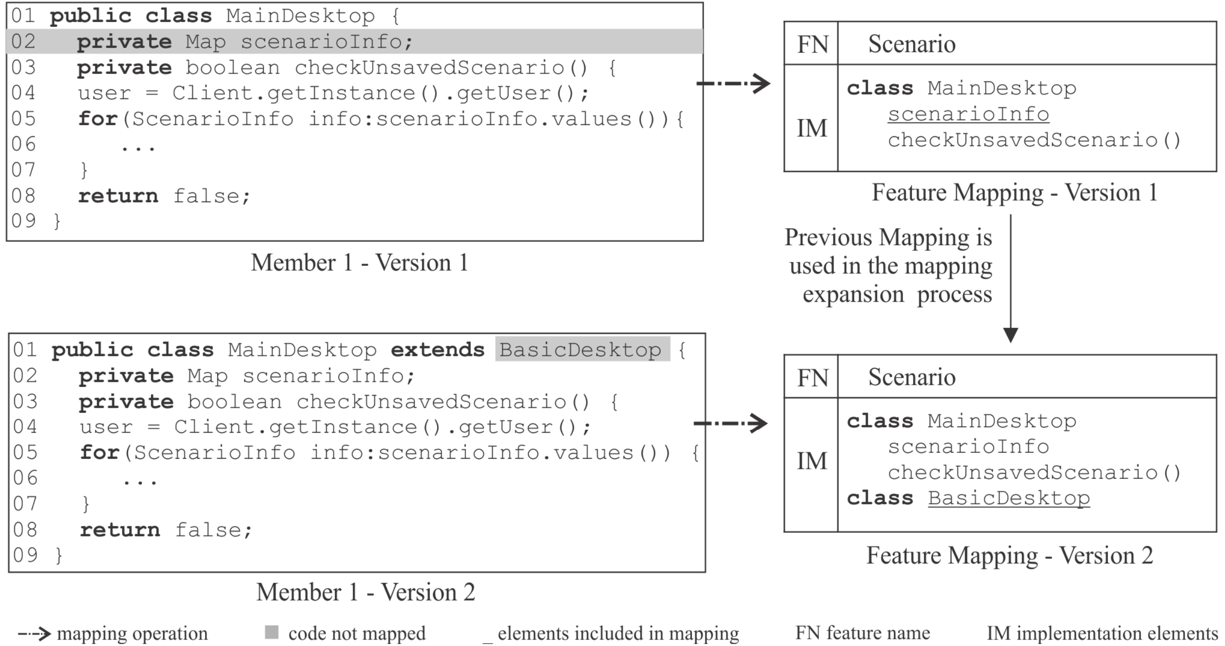

Figure 1. Feature Mappings in Evolution Scenarios.

Figure 1 shows how the described evolution scenarios may lead to mismatches in conventional techniques for feature mapping. The proposed mapping expansion is able to reduce such mismatches in evolving program families. It includes the `scenarioInfo` attribute in feature mapping (see Feature Mapping - Version 1 in Figure 1). The previous feature mapping and the list of changes from one version to another are used to generate the feature mapping of the subsequent version. The expansion process identifies that the `MainDesktop` class was modified in version 2. This process automatically generates the mapping of version 2 and includes the `BasicDesktop` class. The `BasicDesktop` class also realizes the feature *Scenario* as it contains methods that are used within the `MainDesktop` class. Therefore, the mapping expansion is intended to improve the accuracy of the feature mappings in evolving program families. Our solution can be applied to mappings produced by any conventional feature mapping technique (Section 2).

The mappings generated by our heuristics play a key role in a variety of maintenance tasks involving the program family's source code. In particular, we have evaluated their usefulness in two contexts, such as: (i) the recovery and classification of the implementation elements of each family feature (Section 4.1), and (ii) understanding how one or more features and their relationships evolved in the program family's history. To this end, we integrated the mapping heuristics with a visualization tool in order to assist developers in these tasks through graphical perspectives (Section 4.2).

2. RELATED WORK

Feature Mapping Techniques. Several techniques and tools have been proposed for assisting developers to find and map code related to a feature. These activities are called feature location and feature mapping, respectively. These techniques are based on different approaches and they are mainly classified into static [1][3], dynamic [4], and hybrid techniques [2]. All these techniques do not consider the knowledge of the program family change history in order to observe how the features' implementation elements and their relationships have evolved over time. Even when developers run these existing tools for each member version individually, the feature mapping of a particular version of the family member can influence future versions of this same member. This problem essentially occurs because these techniques only take into consideration the history of individual applications. Even worse, it is not known which types of mismatches can arise during the feature evolution mapping. Additionally, there is no evidence of which mismatches are more frequent.

Source Code History Analysis. There are two works that identify non-functional features in individual evolving applications. Nguyen et al. [21] proposed a tool, called XScan, for identifying, ranking, and recommending code elements sharing non-functional features. Adams et al. [22] presented a feature mining technique called COMMIT, which analyses the source code history to statistically cluster functions and types that have been changed together intentionally. Our goal is different as the proposed heuristics focus on the expansion of feature mappings in evolving program families. Unlike these aforementioned works, the proposed heuristics consider the change history of program families and, therefore, are able to observe how the realizations of their features have evolved over time.

3. OUR APPROACH AND UNIQUENESS

Our solution (Sections 3.2 and 3.3) is intended to support feature mapping expansion and address several types of mapping mismatches (Section 3.1).

3.1 Mismatches in Feature Mappings

A set of recurring types of feature mapping mismatches were empirically identified and characterized in our research [23]. These mismatches are made either manually by developers or by applying existing automated techniques (Section 2). We detected

and documented the mismatches through the empirical analysis of evolving feature mappings in program families. Developers were responsible for mapping several features in member versions of different program families using the ConcernMapper static tool [3]. The set of eight feature mapping mismatches and their causes in evolving program families were identified (Table I). We also observed the occurrence of these mismatches in existing techniques for feature mapping [25].

Table I. Feature Mapping Mismatches.

Mapping Mismatch	Description
Multi-Partition Feature	It occurs when the number of disconnected 'partitions' of scattered feature implementation grows as the family evolves.
Overly Communicative Features	It occurs when dependencies between the features' implementation are changed or removed as program family evolves.
Feature Code Clones	It occurs when the existence of similar pieces of code implementing the same feature in different modules increases over time.
Feature-Sensitive Anomalies	It occurs when the existence of anomalies associated with the feature implementation increases during the program family evolution.
Feature Overlapping	It occurs when the degree of overlapping between two features (i.e. sharing one or more code elements) tend to increase through the program family evolution.
Interfaces and Super-Classes	It occurs when developers map the main class realizing the feature but miss to include its super-classes and interfaces as the program family evolves over time.
Omitted Attribute	It occurs when an attribute is missing in a feature mapping as new methods realizing the features are added, changed or removed during the program family evolution.
Deficient Module Structure and Documentation.	It is related to the absence or incorrect mapping of entire classes or methods totally realizing a feature during its evolution.

3.2 Multi-Dimensional History Analysis

The proposed heuristics for mapping expansion are based on a multi-dimensional historical analysis [24][25]. This analysis is based on the comparison of the family members' histories and the analysis of the feature code evolution in the program family. The multi-dimensional analysis takes into consideration both horizontal and vertical histories. The horizontal history refers to the evolution of each member and consists of the implementation elements that realize a feature (feature code). These implementation elements have possibly been added, changed, or even removed in specific versions of a family member. The vertical history consists of a subset or all the members that belong to a program family.

The methodological steps for using our mapping heuristics are: (i) *Feature Selection*: it refers to the list of features that belong to one or more family members under analysis; (ii) *Seed Mappings*: it represents an initial set of elements that contribute to the implementation of each selected feature (Step 1), so-called *seed mappings*; and (iii) *Mapping Expansion Heuristics*: the heuristics expand the feature mappings (Section 3.4) through the family members by using the seed mappings and the members' history analysis. The mapping expansion process in evolving members of the same program family depends on the precision of the version comparisons. The comparison among the versions of the family's members is essential to obtain information about their change history. This information is related to the list of classes, methods and attributes 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.

3.3 Mapping Expansion Heuristics

The proposed heuristics are responsible for expanding feature mappings. Their intent is to improve the accuracy of those mappings by reducing the occurrence of mapping mismatches given a set of evolving members of the same family [23][25]. For this reason, these heuristics rely on a set of recurring mapping mismatches made by developers which were identified (Section 3.1). We briefly present five heuristics due to space limitations (Table II) [25][26]. Some heuristics are able to simultaneously detect more than one type of mismatch (DFP and DIS). For instance, given a specific feature, the heuristic DFP analyses the previously mapped methods by comparing them with the non-mapped ones in terms of interaction similarity [21]. Interaction similarity is defined in terms of method interactions regarding its callees and callers in similar contexts [21]. This means that two methods call or are called by similar ones. Existing techniques do not address the idea proposed by DFP with respect to identify similar methods by considering the feature evolution and the members' history analysis (Section 2).

Table II. Description of the Mapping Expansion Heuristics.

Heuristic	Description
DFP - Detecting Omitted Feature Partitions	It detects methods comprising multi-partition features that have not been mapped.
DCC - Detecting Code Clone Mismatches	It detects occurrences of code clones have not been mapped to a particular feature.
DIS – Detecting Interfaces and Super-Classes	It detects two cases of super-classes and interfaces: either those that have not been mapped or those that have been incorrectly mapped.
DCF – Detecting Communicative Feature Mismatches	It detects implementation elements incorrectly mapped to a feature that have a large interaction with other elements that realize different features.
DOA – Detecting Omitted Attributes	It detects occurrences of omitted attributes that have not been mapped.

4. APPLICABILITY

The mappings generated by the heuristics have been exploited with two different purposes: the recovery of features in the source code of program families (Section 4.1), and the visualization of feature evolution in a program family (Section 4.2).

4.1 Recovery of Features in Code

We defined history-sensitive heuristics for the forward recovery of features in the code of evolving program families [27]. The forward recovery of features encompasses their classification (i.e. common or variable features) and identification of the code elements (e.g. methods and attributes) that realize them. The

recovery heuristics use as input information provided by the mapping heuristics, i.e. the expanded feature mappings. The expanded feature mappings refer to all the versions of each family member after analysing both horizontal and vertical histories of a program family. 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. The recovery heuristics use as inputs of their analyses all the horizontal feature mappings of all the family members. The heuristics aim at identifying and classifying the code elements as part of common or variable features of the program family. The output generated by the heuristics is 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 composed of sub-packages that represent the several categories defined by each heuristic [27]. For each feature, the corresponding package contains the recommended code elements forwardly detected from the historical analysis.

4.2 Visualization of Program Family Features

Our approach also fosters advances on the visualization of evolving program families. Existing techniques only support the representation of modular structures in a program, such as packages, classes and methods [15]. 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. Our heuristics (Section 3) enable developers to visualize the differences among the generated feature mappings. It is possible to analyse the feature code evolution through two or more family member versions under graphical perspectives [15]. The feature evolution visualization was achieved through the integration of the mapping expansion heuristics with a visualization tool, named SourceMiner Evolution (SME) [15]. The SME tool provides three views to support the feature evolution analysis: structure, inheritance and dependency. The SME tool uses three different colors to show developers the differences among the versions of the family's members. For instance, the light blue color represents the implementation elements that realize features in version *i*, but are removed in version *i + 1*. The dark blue represents elements that now realize a feature in version *i + 1*.

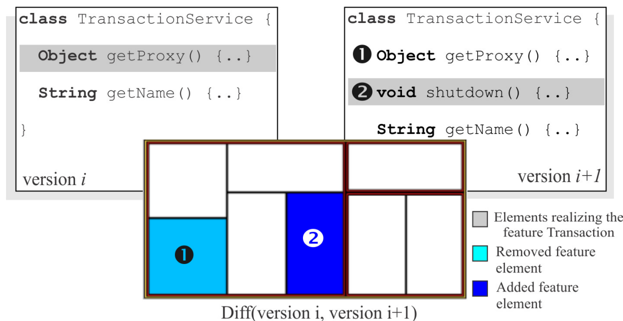

Figure 2. Feature Code Evolution in the Treemap view.

Figure 2 illustrates an example of the Treemap view, which is a hierarchical 2D visualization that maps a tree-structure into a set of nested rectangles [15]. It shows the evolution of the `TransactionService` class from version *i* to *i+1*. This example was extracted from the OC program family (Section 5). 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 light blue color represents the method `getProxy()` and the dark blue color represents the method `shutdown()`.

5. EVALUATION

We evaluated the accuracy of our heuristics by means of two industry program families, named OC and RAWeb; they consisted of seven and six evolving family members, respectively [25]. The implementation of the OC family departed from the same framework. The size of the programs varied from 40 KLOC to 100 KLOC (each family member) and 130 KLOC (the framework). All of them have been evolved since 2006. We analysed the evolution of 12 features which have evolved through 33 versions of these program families. These selected features have involved more than 200 modules as part of the feature mappings. We measured the accuracy of our heuristics by calculating precision and recall after generating all the feature mappings for each version. 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.

For instance, the results of the precision and recall measures for each version of the framework in OC program family are shown in Table III. As it can be observed, the mapping heuristics presented recall measures of 100% in 5 (five) out of 6 (six) features. 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) 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 this case study 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 [28]. There are classes that do not directly realize those features but they contain interaction similarity with the others that realize them.

6. CONTRIBUTIONS

A Catalogue of Recurring Mapping Mismatches. The catalogue of mapping mismatches guides software engineers in promoting the correctness and completeness of their feature mappings [23][26]. This catalogue can also be used in conjunction with existing mapping techniques to check the correctness of their generated mappings.

A Suite of Mapping Expansion Heuristics. The expansion heuristics rely on the multi-dimensional historical of program families [25][26]. The heuristics support the automatic elimination of mapping mismatches. We also designed and implemented a tool, named MapHist [25], which supports the use of the proposed heuristics.

Table III. Precision % (P) and Recall % (R) Results for each Version of the OC framework.

	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

A Set of Heuristics for Feature Recovery. We defined a set of history-sensitive heuristics for the forward recovery of features in code of evolving program families [24][27]. The outcomes of this forward recovery are useful to help developers, for instance, to: (i) determine how the family members departed from the original intended design, and (ii) implement code-level or design-level refactorings of the program family.

Feature Visualization in Evolving Program Families. The feature mappings have been successfully used for feature evolution comprehension in program families [15][27]. The integration of the mapping expansion heuristics with the SME tool allows developers visualizing and analysing the feature evolution through multiple views. Other analyses can also be derived from the expanded mappings, such as the detection of architectural violations [20] and analysis of feature dependencies during the program family evolution.

7. REFERENCES

[1] Eisenbarth, T. et al. 2003. Locating features in source code. *IEEE Trans. Softw. Eng*, vol. 29, 210–224.

[2] Poshyvanyk, D. et al. 2007. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *TSE*, vol. 33, 420–432.

[3] Robillard, M. and Murphy, G. 2007.Representing concerns in source code. *ACM Trans. Softw. Eng. Methodol* v.16, n.1.

[4] Savage, T. et al. 2010. FLAT3: feature location and textual tracing tool. *Proc. of the ICSE*, vol. 2, ACM, NY, 255-258.

[5] Kang, K. et al. 1990. Feature-oriented domain analysis (foda) feasibility study. Technical Report, Carnegie-Mellon.

[6] Parnas, D. L. 1976. On the design and development of program families. *IEEE Trans. Softw. Eng.* vol. 2, n. 1, 1-9.

[7] Adobe. http://www.adobe.com/. Accessed in April.

[8] Mozilla Project. http://www.mozilla.org/en-US/. Accessed in April.

[9] Android. http://www.android.com/. Accessed in April.

[10] Fayad, M. et al. 1999. *Building application frameworks: object-oriented foundations of framework design*. John Wiley & Sons, Inc., New York, NY, USA.

[11] Alves, V. et al. 2005. Extracting and evolving mobile games product lines. *Proc. of the SPLC*, 70–81.

[12] Quilty, G. and Cinnéide, M. 2011. Experiences with software product line development in risk management software product lines. *Proc. of the SPLC*, 251-260.

[13] Figueiredo, F. et al. 2008. Evolving software product lines with aspects: an empirical study on design stability. *Proc. of the* ICSE, ACM, NY, USA, 261-270.

[14] Eixelsberger, W. et al. 1998. Software architecture recovery of a program family. *Proc. of the ICSE*. IEEE Computer Society, DC, USA, 508-511.

[15] Novais, R., Nunes, C. et al. 2012. On the proactive and interactive visualization for feature evolution comprehension: an industrial investigation. *Proc. of the ICSE, Software Engineering in Practice* (to appear).

[16] Kastner, C, et al. 2007. Automating feature-oriented refactoring of legacy applications. *Proc. of the WRT*, 62–63.

[17] Kang, K. et al. 2005. Feature-oriented re-engineering of legacy systems into product line assets - a case study. *Proc. of the SPLC*, 45–56.

[18] Zhang, G. et al. 2011. Incremental and iterative reengineering towards software product line: An industrial case study. *Proc. of the ICSM*, 418–427.

[19] Revelle, M., et al. 2005. Understanding concerns in software: Insights gained from two case studies. *Proc. of the ICPC*, Los Alamitos, CA, USA, 23–32.

[20] Hochstein, L. and Lindvall, M. 2005. Combating architectural degeneration: a survey. *Inf. Softw. Technol.* vol. 47, n. 10, 643-656.

[21] Nguyen, T. et al. 2011. Aspect recommendation for evolving software. *Proc. of the ICSE*, ACM, NY, USA, 361–370.

[22] Adams, B. *et al*. 2010. Identifying crosscutting concerns using historical code changes. *Proc. of the ICSE*, 305–314.

[23] Nunes, C. et al. 2011. Revealing mistakes in concern mapping tasks: an experimental evaluation. *Proc. of the CSMR*, IEEE, Los Alamitos, USA, 101–110.

[24] Nunes, C. et al. 2010. History-sensitive recovery of product line features. *Proc. of the* ICSM – DS, IEEE, DC, USA, 1-2.

[25] Nunes, C. et al. 2012. Heuristic Expansion of Feature Mappings in Evolving Program Families. Submitted to JSME, http://www.inf.puc-rio.br/~cnunes/jsme2011/.

[26] Nunes, C. On the proactive identification of mistakes on concern mapping tasks. 2011. *Proc. of the AOSD, First Place at ACM Competition*, NY, USA 85–86.

[27] Nunes, C. et al. 2012. History-sensitive heuristics for recovery of features in code of evolving program families. *Submitted to SPLC*.

[28] Gamma, E., et al. 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.