3
Mismatches in Feature Mappings

As discussed in Chapter 2, the feature mapping activity is primordial to guide software developers in several maintenance and evolution tasks, such as understanding and restructuring the implementation of existing features. This activity is even more valuable when little documentation of the software systems is available or original developers are not involved in its development anymore. Without feature mappings developers might, for instance, implement a change in non-related modules or even miss relevant modifications for a given change request (Revelle *et al.* 2005). As a result, these changes tend to cause the program family degeneration and make difficult the identification and classification of code elements realizing the features (Chapter 1).

Many techniques and tools have been proposed to support the feature mapping activity (Section 2.2). Even though these techniques and tools have facilitated this activity, it is still manual and error-prone to a large extent (Revelle *et al.* 2005, Robillard and Murphy 2007, Robillard *et al.* 2007, Figueiredo 2009, Figueiredo *et al.* 2011). In particular, even when the feature mapping activity is partially supported by a tool, developers still need to verify if their initial assignments are correct and complete. The goal is to maintain correct and complete mappings according to the current version of the software system. However, this is not trivial to achieve as each feature is usually scattered and tangled across the modular decomposition of a software system (Eisenbarth *et al.* 2003, Robillard and Murphy 2007, Antoniol and Gueheneuc 2005).

Correct and complete mappings are still more difficult to achieve when there are cases where the realization of the same feature evolved differently for each family member over time. This mainly occurs when it is required to generate correct and complete mappings for each version of the family members departing from an initial mapping (Chapter 4). In this context, the mapping developers or reviewers need to check for each feature and each family member version if: (i) elements in the module implementations are missed (i.e. false negatives), and (ii) all the mapped elements are correct (i.e. false positives). The occurrences of false negatives and false positives in feature mappings

are so-called *mapping mismatches*. The checking activities are essential to guarantee the maximum mapping precision before the actual software change is carried out. Nevertheless, software engineers are not equipped with any kind of guidance to promote or review the correctness and completeness of their feature mappings. As a result, the feature mapping activity is often cumbersome and performed in an ad-hoc fashion.

In fact, the number of mapping mismatches tends to be high according to recent studies (Revelle *et al.* 2005, Robillard *et al.* 2007, Figueiredo 2009, Figueiredo *et al.* 2011). However, existing studies do not try to characterize and classify actual recurring mismatches made by different developers. For instance, they do not investigate which feature properties and module structures tend to lead to such mismatches. There is also no work in the literature that analyses which mismatch categories mostly influenced inaccurate feature mappings. In particular, it is not known in the literature which types of mismatches can arise during the feature evolution mapping. Additionally, there is no evidence of which mismatches are more frequent. This limitation has also been becoming increasingly relevant given the growing number of empirical studies that rely on the assumption of reliable feature mappings (Revelle *et al.* 2005, Robillard *et al.* 2007, Greenwood *et al.* 2007, Figueiredo *et al.* 2009, Ferrari *et al.* 2010, Figueiredo *et al.* 2011).

To tackle the aforementioned problems, this chapter presents the characterization and classification of eight feature mapping mismatches commonly made by developers. It answers the first research question of this thesis (RQ1 in Section 1.3). More specifically, we detect and classify the recurring mapping mismatches concerning an analysis of a program family (Section 3.1). These mismatches are associated with various properties of features and modules in the source code (Section 3.2). We present concrete examples and discuss potential reasons on why such mismatch categories might occur on feature mapping activity. We also discuss and identify potential relationships among the mapping mismatches (Section 3.3). In order to further evaluate to what extent these mismatches also occur in wider contexts, we run two experiments using two software systems (Section 3.4). The goal of such experiments is to verify the occurrence rate of the mismatch categories. We also compare our work with previous studies (Section 3.5) and summarize the chapter (Section 3.6).

3.1 Identifying Mapping Mismatches

This section describes the OC program family (Section 3.1.1) used for identifying the recurring types of mapping mismatches. Additionally, it also describes the study procedures (Section 3.1.2).

3.1.1 Target Program Family

As aforementioned in Section 1.1.2, the OC program family is from the logistics domain of the oil industry. It has been developed in Java and consists of seven evolving members. The family members were derived from a single framework and are considered part of a program family for several reasons: (i) they share functionalities from the original framework, and (ii) they underwent various forms of changes, which resulted in significantly different functionalities at different levels of granularity. The analysed framework and applications of OC program family were chosen because: (i) they represent real applications in the industrial scenario of the logistic domain; (ii) these applications have been evolved since 2006 and contain many versions (horizontal history); and (iii) the framework's size is approximately 130 KLOC and the applications' sizes range from 40 KLOC to 100 KLOC. Therefore, they have significant size and complex modules.

Table 3.1: Features analysed in OC Program Family.

Features	Description
Logger	It saves information about program execution and/or errors.
Product	It represents a product and its characteristics in the logistic domain.
Report	It represents the report exhibition, exportation and printing.
Notification	It defines a system notification to users (e.g. email).
Route	It represents a route of products between two points in the logistic context.
Export	It represents the generation of reports for different files formats (e.g. Excel).
Scenario	It represents exportation and importation properties of the products.
Blend	It identifies a blend of products, a composition of products.
Exception Handling	It is the policy to handle exceptional conditions (i.e., the strategy within try-catch blocks).
Persistence/Transaction	It is responsible for storing and recovering data from the database and ensuring ACID properties.

We selected the framework and two members of the OC program family in order to reveal the mapping mismatches. Four versions of each family member and framework were selected, which represent a total of 12 versions. Table 3.1 describes the analysed features. A set of (non-)crosscutting features

was selected. The following crosscutting features of the OC program family were chosen: *Persistence/Transaction*, *Exception Handling* and *Logger*. We also selected features of the logistic domain: *Product*, *Report*, *Notification*, *Route*, *Export*, *Scenario* and *Blend*. These features were chosen because: (i) they represent important functionalities in this domain, and (ii) they are also relevant under an architectural perspective of the program family as they have affected the original architecture during the evolution of each system element.

3.1.2 Study Procedures

This section describes the study preparatory steps followed to detect and classify the feature mapping mismatches made by developers, and explain the reasons behind them. Basically, the study procedures were divided into three stages. First, we selected the versions and features to be mapped in each family member. As aforementioned, a set of twelve versions of the OC program family was chosen. Multiple versions of the OC program family were used for the feature mapping activity in order to find out how feature realizations evolved over time. This was interesting for our analysis because we could observe the nature of the feature mapping mismatches throughout the family member versions. Second, two developers were responsible for mapping the features' implementation elements by using the ConcernMapper tool (Robillard and Weigand-Warr 2005), which is a static tool (Section 2.2.2). The developers had to produce mappings with precision and coverage as maximum as possible. They mapped the same set of features in twelve versions of the OC program family (Section 3.1.1). These developers are experienced and have several programming skills, including extensive knowledge of object-oriented programming and Java language.

The ConcernMapper tool (Robillard and Weigand-Warr 2005) was used as its plug-in is well documented and has an easy integration with Eclipse. It was used as the goal was to identify technique-agnostic mismatches made by developers. We did this choice as our goal was to detect the highest number of possible mapping mismatches when developers interact with the source code. In fact, interaction with source code is also required in feature mappings partially derived with hybrid and dynamic techniques (Sections 2.2.3 and 2.2.4). We also observed that the identified mismatches can also occur on the use of conventional dynamic techniques (Section 3.5.2).

Finally, all the feature mappings produced by each developer were analysed by the author of the thesis in order to verify the differences among them and thus observing and correlating the occurrences of the mapping

mismatches. All the other developers were involved in review meetings in order to evaluate the mapping accuracy. They were involved in the following activities: (i) the validation of detected false positives and false negatives (mapping mismatches), and (ii) the discussion about the characterization of detected common mapping mismatches (Section 3.2). The evaluation of the feature mappings is representative as it contains several types of (non-)crosscutting features with different sizes (number of code elements) and scopes.

3.2 Catalogue of Mismatches in Feature Mappings

The mismatches were revealed by observing the feature mappings of the selected family members (Section 3.1.1). The goal was to identify, characterize and categorize recurring mapping mismatches in order to help developers avoid them. A set of eight mismatches was characterized and classified into two broad categories: (i) Feature Characteristics: types of mismatches that were found to be related to particular properties of how features are realized in the source code (Section 3.2.1), and (ii) Module Characteristics: types of mismatches that are related to properties of the modularity units affected by the mappings, such as classes, super-classes and methods (Section 3.2.2). For each category, we identified a set of mismatches and described the reasons for their occurrences. Additionally, we also used illustrative examples to explain each mismatch subcategory. The categories of mismatches are related to missing or incorrect code elements in the mappings. This means that for the absence of elements, pieces of modularity units were omitted. That is, developers did not map either coarse-grained or fine-grained elements responsible for realizing the feature. Incorrect elements are those that, although mapped, do not contribute to realizing the feature.

3.2.1 Feature Characteristics

This category is mainly related to feature implementation. The causes for the occurrence of mapping mismatches refer to: (i) dependencies among the implementation elements that contribute to realizing the same feature; (ii) interaction among features which is based on how feature realizations share implementation elements (e.g. methods); and (iii) the existence of modularity anomalies, such as code smells (Fowler 1999, Figueiredo *et al.* 2009), associated with the feature implementation, such as the Feature Envy smell (Fowler 1999, Figueiredo *et al.* 2009).

Multi-Partition Feature. The multi-partition feature is a specific case of feature implementation scattered in several modules (classes or methods). A scattered feature has multiple partitions when: (i) a sub-set of modules (i.e. a given partition of the feature) that implement this feature contain explicit references among them, and (ii) one or more of the other modules (i.e. forming another partition of the same feature) do not contain explicit references to the sub-set in (i). The lack of explicit references between feature partitions is the reason for elements of a partition are not often included in feature mapping. Figure 3.1 illustrates a case of multi-partition feature, which is represented by a disconnected graph. According to this figure, classes in the A.1 sub-graph contain explicit references to each other. On the other hand, classes in the A.2 sub-graph which implements the same feature are not explicitly connected to classes in the A.1 sub-graph. This mismatch was made by two developers that did not map the modules without explicit references (A.2). The reason is that they started the mapping activity by browsing the code of classes that contain explicit references to each other in a specific feature partition (A.1). As a result, the other classes responsible for implementing the feature were not mapped. This mismatch happened, for instance, with the feature *Scenario*. This feature is very scattered and is realized by several classes that do not have explicit references among them. It is common to observe that feature partitions are often added, modified or removed through the program family evolution.

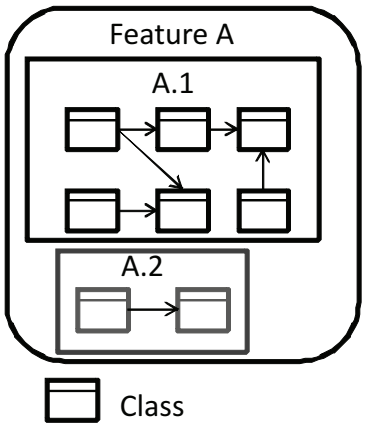

Figure 3.1: Multi-Partition Feature.

Overly Communicative Features. Overly communicative features are characterized when there are two sets of interconnected classes realizing two different features. Then, let's consider the features A and B. There are dependencies among the classes realizing the features A and B in order to allow the communication between these features. Figure 3.2 illustrates how this mismatch is characterized. We observed that this scenario occurred in the following sit-

uation. The developers start mapping the classes that implement the feature A. After that, they incorrectly map a set of classes that implement the feature B as being part of the feature A (or vice-versa). The reason for this mismatch is that most of the classes of the feature A communicate with other ones that implement the feature B, so called *overly communicative features*. For instance, this scenario occurred with the features *Scenario* and *Notification* in OC program family (Table 3.1). There are several classes related to the feature *Notification* that communicate with classes of the feature *Scenario*, such as `CommonScenariumServiceDB` class in Code 3.1. Therefore, developers that started to map the feature *Scenario* also mapped classes related to the feature *Notification* (lines 03-06) as being part of the feature *Scenario*. The strong dependencies between the two features in the source code tend to cause misunderstanding on feature mapping activity. It is naturally reasonable that dependencies among the features' implementation elements might be changed or removed as program families evolve.

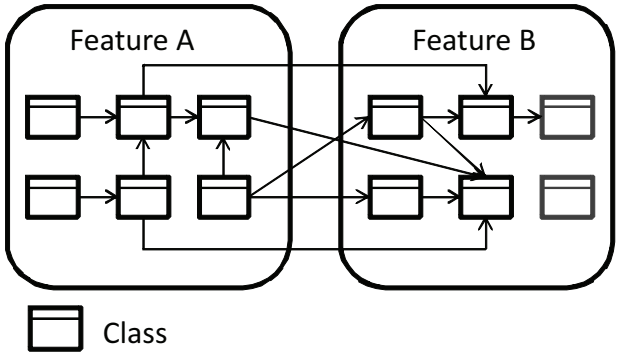

Figure 3.2: Overly Communicative Features.

Code 3.1: Example of Communicative Feature.

```
01 public class CommonScenariumServiceDB {
02  public void sendNotification(ScenariumInfo) {
03    N  notif = serviceNotificationData.buildApprovalNotification(..);
04    N  NotificationService.getInstance().notifyToAllUsers(..);
05  }
06 ...
07 }
  Label:
     N - Feature Notification
```

Code Clones. It encompasses the existence of cloned code in the source code. They are similar pieces of code implementing the same feature in different modules (Johnson 1994, Baxter *et al.* 1998, Kamiya *et al.* 2002, Basit and Jarzabek 2005,

Kim *et al.* 2005, Nguyen *et al.* 2009, Basit and Jarzabek 2009). The existence of multiple code clones is the reason for developers to miss or neglect the mapping of all clone occurrences of a particular feature. In addition, if the same clone instance is related to more than one feature, developers tend to neglect the mapping of the clones of a given method to different features. This scenario can become even more difficult when instances of code clones increase or decrease over time.

Feature-Sensitive Code Smells. It is related to the existence of code smells associated with the feature implementation in the source code (Fowler 1999, Carneiro *et al.* 2010). Code smells are symptoms in the source code that may be indicative of software quality problems (Fowler 1999). Although the traditional definitions of code smells are not directly based on the definition of separation of features, some research work have associated their occurrences with a poor modularization of features (Greenwood *et al.* 2007, Figueiredo *et al.* 2009, Figueiredo *et al.* 2011). There are several kinds of code smells (Fowler 1999) that usually make them hard to produce accurate feature mappings, such as Feature Envy, God Class, God Method. For example, a method that implements a feature that is different from the main feature realized by the class implementation is classified as *Feature Envy*. God Classes are classes that implement more than one feature at the same time, whereas God Method refers to methods that implement more than one feature. Two specific types of mismatches associated with the occurrence of code smells were observed in the OC program family and are discussed below. They are Features Interlacing and Features Overlapping.

Feature Interlacing. It occurs when two or more features partially affect one (or more) module(s) in common (Greenwood *et al.* 2007, Figueiredo *et al.* 2008). The interlacing can be classified into two categories: module and method-level interlaces. We have observed that the higher the tangling among features in a module, the more difficult the feature mapping activity is. For instance, there were cases where lines of code of specific methods were not mapped to a particular feature. Basically, the main factor associated with this mismatch is the existence of blocks of code inside the class that do not implement the main purpose of a method. This means that feature interlaces are often associated with the Feature Envy code smell (Fowler 1999). Code 3.2 illustrates a slice of code that implements the following features: *Exception Handling*, *Logger* and *Notification*. It is possible to observe the interlacing among these three features. In fact, this interlacing

hampered the developers' understanding and performance during the mapping activity. However, feature interlaces are not necessarily associated with bad OO programming practices; even when using well-known object-oriented techniques, it is not possible to completely separate all tangled features of interest (Figueiredo *et al.* 2008).

Code 3.2: Example of Interlacing among Features.

```
01 EH try {
02       if (!isLoggedUser) {
03 N       notifyNewUserLogged(user);
04       }
05 EH } catch (InvalidLoginException) {
06 L    theLogger.log(...);
07 EH } catch (RequestInternalException) {
08 L    theLogger.log(login, requestException);
09 N    mailLoginErrorToSupport(..);
10     }
  Labels:
       EH - Feature Exception Handling
       N  - Feature Notification
       L  - Feature Logger
```


Feature Overlapping. It occurs when two features entirely share one or more code elements (e.g. methods, attributes or classes). This mismatch is different from feature interlacing because the shared elements entirely contribute to both features rather than being disjoint. Feature overlapping can be classified as component overlapping, operation overlapping, or lines of code overlapping (Figueiredo *et al.* 2008). In these cases, developers tend to map the implementation elements to one of the features and not mapping the same elements to other ones that are realized by the same code fragments. We mainly noticed that the existence of this mismatch is often related to the occurrences of the following code smells: God Class, God Method, and Feature Envy (Fowler 1999). Code 3.3 illustrates this mapping mismatch, which is classified as lines of code overlapping. According to this figure, there is a God Method that is responsible for starting a set of services. Lines of code of the features overlap this God method, such as the *Logger* (line 04), Base Data Access (line 03), *Transaction* (lines 6-7) and Instrumentation code (lines 05-08). We have observed two different problems: (i) different features being implemented in the same method, and (ii) instrumentation code only used to test the system and activate the transaction. As a result, this method was not mapped for all the features that it realizes, such as *Logger* and *Transaction*. Therefore, this element was omitted in feature mapping. The occurrences of feature overlapping might also decrease or increase during the program family evolution.

Code 3.3: Example of God Method.

```
01 private void createLogisticServices() {
02  loginService = LoginService.getInstance();
03  userService = UserServiceDB.getInstance();
04  logService = LogService.getInstance();
05  if (isInTestMode()) {
06      transactionService = TransactionService.getInstance();
07  }
08 }
```

3.2.2 Module Characteristics

This category is associated with the modularity properties of the implementation elements. Basically, these properties are associated with interfaces and super-classes, attributes, and entire classes and methods.

Interfaces and Super-Classes. It is characterized when developers map the main class realizing the feature but miss to include its super-classes and interfaces in the mapping. Developers often neglect super-classes and interfaces in the application as they require the system navigation and analysis through the program hierarchical structure. However, the opposite problem might also occur: the incorrect inclusion of super-classes and interfaces in the mapping. These are typically the cases of super-classes and interfaces comprising a framework or API realizing a different feature. However, developers incorrectly map these super-classes and interfaces as being part of the feature implementation under analysis. Code 3.4 illustrates these two cases in the OC program family. In the first case, the `NotificationServiceInterface` interface was not mapped. It should be included in the mapping as it refers to the feature *Notification*, which is responsible by the notification services of the user. In the second case, the super-class was incorrectly mapped as the `AbstractAction` class is related to an API for manipulating users' request. As program families evolve over time, the inclusion, removal and change of interfaces and super-classes are often.

Code 3.4: Piece of Code with Super Classes and Interfaces.

```
// Omitted interface for the feature Notification
01 public class NotificationService implements NotificationServiceInterface {
02   ...
03}

// Incorrect mapping for the feature Scenario
01 public abstract class ScenariumAction extends AbstractAction {
02  ...
03}
```

Omitted Attribute. It occurs when an attribute is missing in a feature mapping (Adams *et al.* 2010). We found that in general developers tend to mainly focus their mappings on the behavior implemented by the methods. They often forget to observe the class data when producing the mapping of each feature. Attributes in feature mappings are especially important as they might indicate other types of implementation elements that take part of a feature implementation. This occurs because an attribute might be used and modified by more than one method. Code 3.5 illustrates an example where developers only mapped the pieces of source code inside the `checkUnsavedScenarium()` method. They did not map the `infos` and `selectedInfo` attributes that realize the feature *Scenario*. The occurrences of omitted attributes may directly increase or decrease when new methods realizing the features are added, changed or removed during the program family evolution.

Code 3.5: Piece of the Code of Feature Scenario.

```
01 public class MainDesktop {
02  private Map<Code<ScenariumInfo>, ScenariumInfo> infos;
03  private ScenariumInfo selectedInfo;
04  private boolean checkUnsavedScenarium() {
05    user = Client.getInstance().getUser();
06    context = ServerMonitor.getInstance().getLoginContext();
08    for (ScenariumInfo info:infos.values()) {
09       ...
10    }
11   return false;
12}
```

Deficient Module Structure and Documentation. It is related to the absence or incorrect mapping of an entire implementation element. Entire implementation elements are defined as classes and methods totally responsible for realizing a feature. Intuitively, it is expected that they are easier to be mapped to the feature because their entire structure contributes to the feature implementation. However, this mismatch was commonly found in OC program families, and the reasons can be associated with a series of module-specific properties. Some of these reasons found were: (i) names of classes, methods and attributes that do not directly reflect the feature functionality - i.e. there is no naming pattern for variables and methods during the system evolution; (ii) absence of detailed comments in the module code in order to help understand its purpose; and (iii) classes and methods that are misplaced in packages and classes, respectively. Examples of features omitted in the OC program family were *Scenario* and *Notification*. For instance, there is a feature *Permission* in the OC program family, which is responsible for defining permissions for

several types of features, such as *Scenario* and *Product*. This way, two possible steps followed by developers to map the implementation elements realizing the feature *Permission* are: (i) they identify the dedicated classes related to the feature *Permission*, and (ii) they select the exact occurrences of methods that realize the permission of the feature *Scenario*.

Code 3.6 shows the `FolderAdminFrame` class that is responsible for allowing access to the administrator tree and more specifically the folders of the feature *Scenario*, such as the `getFrame()` method. This method is part of the feature *Permission* but it was not mapped by two developers. The reason for this mismatch is that developers did not verify which methods implement the permission of the feature *Scenario*. For this reason, this entire method was incorrectly mapped to the feature *Permission*. On the other hand, for the case where implementation elements were incorrectly mapped we noticed that this mismatch is also related to the existence of well-known code smells: Feature Envy, God Method and God Class (Fowler 1999). In both cases, developers tend to incorrectly map parts of the class structure to the feature. As this mismatch refers to the existence of entire classes and methods realizing a feature, its occurrences may often arise over time.

Code 3.6: Example of Features Scenario and Permission

```
01 public class FolderAdminFrame extends GenericFrame {
02  private ScenariumInfoTree infoTree;
03  public FolderAdminFrame getFrame(GenericFrame, ScenariumInfoTree) {
04      frame = GenericFrame.getGenericWindow(..);
05      if (frame == null) {
06          frame = new FolderAdminFrame(..);
07      }
08      return new FolderAdminFrame(...);
09  }
10  ...
11}
```

3.3 Correlating the Mapping Mismatches

The previous sections presented eight mapping mismatches grouped into two categories. These mismatches are not fully independent, and the occurrence of a mismatch can directly or indirectly imply another one and vice-versa. This section discusses the potential relationships between the mismatches observed in the mappings of the OC program family (Section 3.2). Documenting such relationships help developers understand and identify alternative reasons for a particular mapping imperfection. Figure 3.3 provides an overview of the mapping mismatch relationships, which are represented

by arrows connecting two mismatches. The relationships are named "*can be related*" and "*can influence*". The term "*can be related*" represents the case when a mismatch can be seen, and consequently quantified over different perspectives depending on its granularity and specificity. The term "*can influence*" means the existence of a mismatch can affect the emergence of another one.

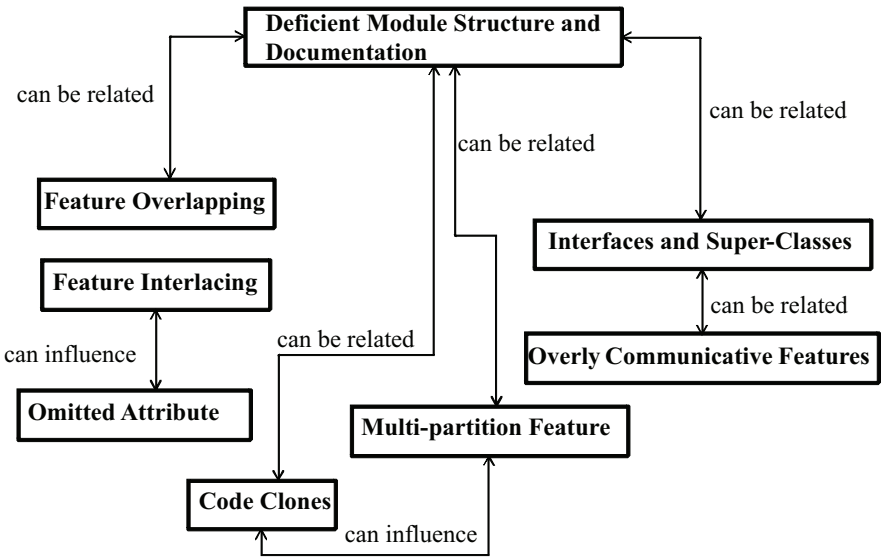

Figure 3.3: Mapping Mismatches Relationships.

The mismatch classified as *Deficient Module Structure and Documentation* is that one with more associations with other mismatch categories (Figure 3.3). In this case, it contains a relationship indicating that it "*can be related*" to the following mismatches: *Feature Overlapping*, *Interfaces and Super-classes*, *Multi-Partition Feature*, *Code Clones* and *Overly Communicative Features*. For instance, Code 3.4 shows an example of mapping mismatch where the `NotificationServiceInterface` interface is missed; i.e. it is a false negative in feature mapping. This mapping mismatch can be analysed and quantified under two different perspectives: (1) the entire `NotificationServiceInterface` interface or the deficient module structure and documentation was not mapped to the feature *Notification*, or (2) the interface and super-class were not mapped to the feature *Notification*. Hence, this mapping mismatch can be quantified as: *Deficient Module Structure and Documentation* and/or *Interfaces and Super-Classes*. The other relationships follow the same reasoning. However, they need to be carefully analysed depending on each case. This explanation also applies to the relationship between the following mismatches: *Interfaces and Super-classes* and *Overly Communicative Features*.

Regarding the relationship named "*can influence*", we observed in our analyses that it occurs, for instance, between the following mismatches: *Feature Interlacing* and *Omitted Attribute*. This is due the cascade effect that the former mismatch can generate. For instance, as described in Code 3.5 there is a interlace involving the feature *Scenario* and other ones related to the system core. This feature interlace might have affected the manifestation of the mapping mismatch, which in turn caused the attributes being omitted in the mapping. Analogous reasoning applies to the relationship between *Multi-Partition Feature* and *Code Clones*.

3.4 Experimental Evaluation

This section presents the evaluation of the mapping mismatches through two experiments. It shows and discusses the main results in terms of the frequency rate in which mapping mismatches occur in other contexts.

3.4.1 Selected Software Systems and Features

We used two software systems in the experiments in order to verify the occurrence of mapping mismatches. The chosen systems were: (i) a typical Web-based system called Health Watcher (Soares *et al.* 2002), and (ii) an evolving software system called MobileMedia (Figueiredo *et al.* 2008). We selected the object-oriented versions implemented in Java of both systems. A set of (non-)crosscutting features was selected for each software system. Five features from Health Watcher were selected: *Concurrency*, *Distribution*, *Exception Handling*, *Persistence* and *View*; and four features from MobileMedia: *Exception Handling*, *Security*, *Sorting* and *Favourites*. Descriptions of these features are provided in Table 3.2. Descriptions of the features *Exception Handling* and *Persistence* are presented in Table 3.1. These features were chosen because (i) they are representative in these software systems, and (ii) they are different in terms of functionality and granularity. Therefore, the selection and assessment of different kinds of features were important to enable us to observe the occurrence of mismatches performed by developers during the mapping activity.

It is important to highlight that the members of the OC family do not contain detailed and extensive documentation, but only the source code remains as basis artefact (Section 3.1.1). On the other hand, both Health Watcher and MobileMedia systems contain detailed documentation, such as application domain, description of the features, main classes involved

Table 3.2: Features analysed in Health Watcher and MobileMedia systems.

Systems	Features	Descriptions
HealthWatcher	Concurrency	It provides a control for avoiding inconsistent information stores in the system database.
	Distribution	It is responsible for externalizing the system services at the server side and supporting their distribution to the clients.
	View	It is responsible for processing the web requests submitted by the system users.
MobileMedia	Security	It improves the user's privacy and it requires authentication to access to albums (i.e. login and password).
	Sorting	It provides a service for sorting media by the number of accesses.
	Favourite	It provides services to set favourite media and visualize them.

in the system architecture (Soares *et al.* 2002, Figueiredo *et al.* 2008). These systems were chosen as: (i) they were developed by different designers, which complicated even further the feature mapping activity; (ii) we needed reliable and complete reference mappings, which were performed independently by the original developers; and finally (iii) the diversity of the systems was also important to analyse how the mismatches are representative under the perspective of different developers.

3.4.2 Experimental Procedures

We ran the experiments with 26 software developers from two institutions. In the first one, 13 undergraduate Computer Science students in their final year of study mapped features onto Health Watcher. In the second one, 13 graduate (Master and PhD) students mapped features onto MobileMedia. All these developers claimed to have knowledge of Java and object-oriented programming, Web technologies, database systems, UML and Software Engineering. Before starting the experiment, it was explained to the developers how the mapping activity should be done. This demonstration worked as a training session since not all developers were familiar with the feature mapping activity. In this training session, we demonstrated the mapping activity of one feature being realized by two classes of a software system. We used a different feature and software system to avoid biasing the experiment results. The training session also included some guidelines of feature mapping activity, for instance, it was indicated that two or more features could be mapped to the same code fragment.

In the experiment we focused on a manual mapping activity in order to maintain the goal of the study (Section 3.1.2). Developers involved

in the experiment received the source code of four classes of each system: Health Watcher and MobileMedia. The chosen classes in the Health Watcher system were `EmployeeRecord`, `Employee`, `ServletInsertEmployee` and `HealthWatcherFacade`; whereas in the MobileMedia system were `MediaData`, `AlbumListScreen`, `AlbumController` and `MediaListController`. We restricted our experiment to four classes and three features of each system because we wanted to be able to complete the experiment in one hour. These classes were selected because they are representative and important to realize the chosen features. Additionally, they are relevant classes because belong to different layers in the system architecture. We selected few classes in the experiment as the feature mappings were performed by hand by the developers. We did this choice in order to avoid manipulating many variables, such as the use of tools. In addition, as aforementioned our study is centered on observing the recurring mapping mismatches performed by the developers; i.e. technique-agnostic mismatches. The produced mappings were also used to analyse the impact of feature mappings on crosscutting feature measures (Figueiredo *et al.* 2011).

The detection and quantification of the mapping mismatches were performed in a manual way. First, we separated the mappings performed by each developer considering each feature. Second, we started to analyse the mappings and associate them with the types of mapping mismatches (Section 3.2). During our analyses, we only count one type of mismatch per wrongly mapped code fragment. We selected the mismatch category that, according to the developer, was the main cause for a mapping mismatch. It is also important to highlight that other reasons for the mismatches may exist, and this may be further explored in future work. However, we focused on the analysis of the mappings under the perspective of our mapping mismatch classification derived from the analysis of the OC program family (Section 3.2).

3.4.3 Quantifying the Mapping Mismatches

Figure 3.4 presents the occurrence rate of the mismatches in both Health Watcher and MobileMedia systems considering the 26 developers involved in the experiments. This figure is organized in terms of the categories of mismatches, the total number of mismatches for each type of category, and the total number of developers that made this mismatch in both systems considering all the features. For the set of selected classes in the Health Watcher system, the following mismatches were not detected: *Feature Overlapping*, *Code Clones* and *Multi-partition Feature*. On the other hand, it was not observed

the following mismatches in the MobileMedia system: *Code Clones*, *Overly Communicative Features* and *Interfaces and Super-Classes*. The occurrences of code clones were the only one mismatch not observed in both systems considering the set of selected classes in the experiments. The strategy that we followed to detect it was to search for code blocks with similarity degree equal or larger than 90% among the classes realizing the same feature. We realized that the occurrences of code clones could not be found in the software systems used in the experiment because they were implemented from existing frameworks and APIs; it was visible that the goal of their implementation was to minimize the occurrence of redundant code.

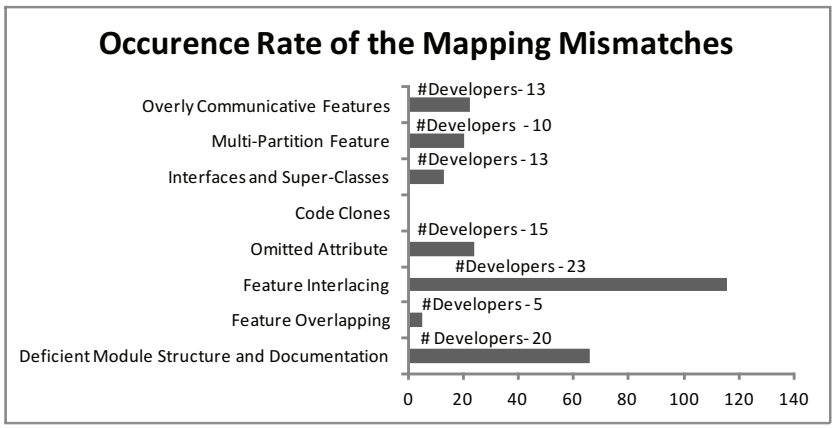

Figure 3.4: Mapping Mismatches in both Health Watcher and MobileMedia systems.

According to these results, we observed that the developers tend to make the same mapping mismatches. For example, *Deficient Module Structure and Documentation* mismatch was evident for almost all developers considering the following features: *Distribution*, *View* and *Sorting*. Generally, these mismatches tend to occur when there are few methods (e.g. 1-2 methods) inside a class. However, this mismatch occurred even in a class totally responsible for realizing the feature *View* (e.g. the `ServletInsertEmployee` class). We observed that the developers mapped only some lines of code of the `ServletInsertEmployee` class to the feature *View*. However, the developers should have mapped it completely instead.

The occurrence of *Feature Interlacing* is mainly related to specific crosscutting features, namely: *Exception Handling*, *Security* and *Concurrency*. The reason is that crosscutting features are tangled with other features in object-oriented systems (Figueiredo *et al.* 2011, Greenwood *et al.* 2007, Figueiredo *et al.* 2009, Figueiredo *et al.* 2008). The occurrence of *Multi-Partition Feature* was more evident in fine-grained features, such as *Favorites* and *Sorting*. Similarly to crosscutting features, the implementation of these

fine-grained features tends to be scattered over several methods. The mismatches *Omitted Attribute* and *Interfaces and Super-Classes* tend to uniformly occur with different kinds of features. We discuss below some occurrences of mapping mismatches grouped by feature.

We observed that the feature *Distribution* is related to the following mismatches: *Deficient Module Structure and Documentation*, *Overly Communicative Features* and *Interfaces and Super-Classes*. Code 3.7 illustrates an example of the *Interfaces and Super-Classes* mismatch where all the developers neglected the mapping of the `IFacade` interface (line 02) to the feature *Distribution* in Health Watcher system. In addition, they did not also map the entire `rmiFacadeExceptionHandling()` method (lines 04-06), which provoked the *Deficient Module Structure and Documentation* mismatch. Regarding the *Overly Communicative Feature* mismatch, 13 developers made it due to the fact that there is a large interaction between classes of the feature *Distribution* and classes that access the database repository. As a consequence, they mapped the classes that access the database as being part of the feature *Distribution*. The same reasoning behind this mismatch applies to the feature *Persistence*.

The mapping mismatches associated with the feature *Security* were: *Omitted Attribute* and *Feature Interlacing*. For instance, most of the developers did not map the `passwd` attribute, which contributes to realizing this feature. In addition, the mapping of this feature includes lines of code tangled with other features. As a consequence, many developers did not map all the lines of code. It is possible to perceive how these mismatches are related and in fact, one can influence another (Section 3.3). However, this is not always true and for this reason, they were documented separately.

Code 3.7: Piece of Code of the Feature Distribution.

```
01 public class HealthWatcherFacade extends java.rmi.server.
    UnicastRemoteObject
02          implements IFacade {
03  ...
04 private void rmiFacadeExceptionHandling(..){
05  ...
06 }
07}
```

The feature *Sorting* is mainly realized by the `MediaListController` and `MediaData` classes. This feature is scattered over four methods of the `MediaListController` class as illustrated in Code 3.8. Two of these methods, `exchange()` and `bubbleSort()`, are totally dedicated to realizing the feature *Sorting*. Two mapping mismatches were recurrently associated with this feature. First, most of the developers (D2, D3, D5, D7, D12, D13) did not map at

least one of these methods (e.g. *Deficient Module Structure and Documentation* mismatch) or they incorrectly mapped other methods that do not realize the feature *Sorting*. Second, developers only mapped a few implementation elements that realize the feature, and consequently characterized the *Multi-Partition Feature* mismatch. In fact, most of the developers did not map the method calls that contribute to realizing the feature *Sorting* (lines 03, 05, 07, 09-11). The occurrence of *Multi-Partition Feature* mismatch also occurred for the feature *Favourite*.

Code 3.8: Piece of code of the Feature Sorting.

```
01 public boolean handleCommand(Command) {
02   ...
03   showMediaList(...);
04   ...
05   showMediaList(...);
06 }
07 public void showMediaList(...) {
08   ...
09   if (sort) {
10     bubbleSort(medias);
11   }
12   ...
13 }
14 private void exchange(MediaData, int, int) {
15   ...
16 }
17 public void bubbleSort(MediaData) {
18   ...
19 }
```

The feature *Exception Handling* is mostly associated with the *Feature Interlacing* mismatch. Code 3.9 illustrates an example of the occurrence of this mismatch in the MobileMedia system. The main issue with the mapping of this feature is that most of the developers mapped the entire code inside *try-catch* statements. That is, developers tend to associate the entire try block with the feature *Exception Handling*. As illustrated in Code 3.9, the pieces of code mapped to the feature *Exception Handling* do not actually include the code fragments on lines 02 to 04. Basically, this mapping mismatch occurs due to the interlacing of the features *Exception Handling* and *Security*.

Code 3.9: Piece of Code of the Feature Exception Handling.

```
01 try {
02   password = getCurrentScreen();
03   getAlbumData().createNewAlbum(..)
04   getAlbumData().addPassword(..);
05 } catch () {
06   ...
07 }
```

3.4.4 Threats to Validity

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

Conclusion Validity. We identified two possible threats to this category: (i) *reliability of mappings*: subjective decisions were made during the feature mapping activity, and (ii) *heterogeneity of developers*: two developers were involved in the mapping activity of the OC program family (Section 3.1.1) and 26 developers were involved in the experiment regarding the Health Watcher and MobileMedia systems. To reduce the risks associated with the category (i) for all the software systems, the developers received instructions before starting the feature mapping activity. In order to classify and categorize the mapping mismatches, the developers studied the selected systems (Section 3.1.1). In addition, there were meetings with the development team in order to obtain the needed knowledge to accomplish this activity in a better way. Considering the two experiments, the developers received instructions and explanations about the software systems and features before starting the mapping activity. We tried to reduce the risk (ii) involving developers with similar knowledge. Two experienced developers were in charge of feature mapping activity of the OC program family. These developers are not the original developers of the OC family but have a good knowledge of is source code and features. On the other hand, a group of 13 undergraduate and post-graduate Computer Science students were selected to accomplish the experiment using the Health Watcher and MobileMedia systems, respectively (Section 3.4). All these students claimed to have experience in the following topics: object-oriented programming, Java language, Web technologies, database systems, UML and Software Engineering.

Construct Validity. We identified the following risks: (i) *mapping mismatches treated in an inadequate way*: specific mismatches that should be treated in a different way might have biased the results, (ii) *interaction of the developers with the system*: the developers were aware of the proposed study. To reduce the risk (i) we defined procedures to be followed during the feature mapping activity and quantification of the mapping mismatches. The quantification strategy of the mismatches was performed in a manual way in which we analysed them under a perspective of each feature. However, other deductions could be performed based on the mapping mismatches relationships (Section 3.3). Regarding the risk (ii) the developers were prepared to accomplish this task through the instructions they received before the feature mapping activity. These developers were selected because they have

knowledge of the relevant topics addressed in the experiments. In addition, they performed the experiment in a voluntary way.

Internal and External Validity. We only identified one possible risk for internal validity: *the complexity of the features*. This complexity might have made one developer make more mismatches than others. However, this threat was minimized because all the developers mapped the same set of features using the same versions of the system. Threats to external validity are conditions that allow results generalization. The first identified risk was the selected systems. In order to minimize this threat, we chose members of an industrial program family from the logistic domain named OC program family, a typical Web-based system named Health Watcher (Soares *et al.* 2002, Greenwood *et al.* 2007), and an evolving software system named MobileMedia (Figueiredo *et al.* 2008). All the target software systems are representative from different domains and they have a significant size. Health Watcher and MobileMedia were extensively used and evaluated in previous research work (Soares *et al.* 2002, Greenwood *et al.* 2007, Figueiredo *et al.* 2008, Ferrari *et al.* 2010). In addition, these systems contain many types of (non-)crosscutting features with different complexity degrees. This way, they enabled us to observe the differences among the feature mappings. It is important to mention that we only evaluated a part of these systems in the experiments (four classes of each one). As a consequence, we do not observe, for instance, occurrences of *Code Clones* in these classes.

3.5 Limitations of Related Work

This section discusses how our research addresses the limitations of existing work. The key related work are classified into two categories: feature mapping studies and techniques for supporting feature mapping activity.

3.5.1 Feature Mapping Studies

A few recent studies were conducted to better understand the feature mapping activity (Revelle *et al.* 2005, Robillard *et al.* 2007, Figueiredo 2009). For instance, Robillard *et al.* (Robillard *et al.* 2007) conducted an empirical study to assess the overall accuracy of interactive manual mapping activity in four systems. The subjects of this study used the ConcernMapper tool (Robillard and Weigand-Warr 2005) to perform the feature mappings. The authors selected 16 features in the target systems. For each feature they asked three subjects to produce feature mappings. However, in this work the authors

do not identify and classify the recurring types of mapping mismatches made by the subjects. In addition, they do not have the intention to ameliorate the accuracy of the mappings performed by the subjects.

Revelle *et al.* (Revelle *et al.* 2005) identified which factors influenced the consistent feature mapping activity in two case studies. Two developers were instructed to map features in the source code. They also provided some guidelines to help developers map features in the source code. However, the influential factors identified by the authors are very general, and they do not explicitly reveal the actual mapping mismatches made by developers. In addition, the authors do not catalog the mapping mismatches taking into consideration certain properties related to specific features and/or code structure. In addition, both Robillard's and Revelle's studies do not consider different types of (non-)crosscutting features.

Figueiredo (Figueiredo 2009) conducted an experimental study to investigate the accuracy of feature mappings performed in a manual way. He evaluated the impact of the mappings on the precision of metrics to quantify crosscutting features properties. The amount of hits, false positives and false negatives for each feature mapping was measured. His work is different from ours because he has not identified or categorized the mapping mismatches made by developers. As a consequence, developers are still left without any guidance on how to correct and extend their feature representation.

Revelle and Poshyvanyk (Revelle and Poshyvanyk 2009) evaluated 10 different techniques that encompass various combinations, such as textual, dynamic and static analyses. The authors selected two open source systems for assessing the effectiveness of these techniques at finding implementations of features. They did many combinations in order to identify which ones tend to improve the accuracy of the produced mappings. They also found that on average the use of marked traces is more effective to discover more methods that realize a feature than full traces. This work is different from ours because despite analysing several combinations in terms of feature mapping techniques, Revelle and Poshyvanyk do not characterize and classify the types of mapping mismatches performed by each combination of tools.

3.5.2 Mapping Mismatches and Existing Techniques

This section discusses aforementioned related work with the intent of describing their limitations when considering mapping mismatches. As mentioned in Chapter 2, some tools and techniques support developers in manually performing feature mappings, such as

ConcernMapper, ConcernTagger, FEAT, Feature Manipulation Environment, and intentional views (Robillard and Weigand-Warr 2005, Mens *et al.* 2006, Robillard and Murphy 2007). There are several semi-automatic techniques for supporting the feature mapping activity (Eisenbarth *et al.* 2003, Mens *et al.* 2006, Robillard and Murphy 2007, Kellens *et al.* 2006, Antoniol and Gueheneuc 2005, Savage *et al.* 2010, Adams *et al.* 2010). The evaluation of these techniques focuses mainly on hits, false positives and false negatives. In general, the rate of mapping mismatches is often very high (Revelle *et al.* 2005, Figueiredo 2009). However, all these studies do not reveal the nature of mismatches made manually by developers when either completing or correcting feature mappings. Our work is different because we are interested in characterizing and classifying the recurring mapping mismatches. This classification is also useful for enhancing feature mining techniques. The categories of mapping mismatches were identified in the exploratory study using a static technique, ConcernMapper tool (Robillard and Weigand-Warr 2005). However, these mismatches can also occur on the use of dynamic techniques. As aforementioned in Section 2.2.3, the dynamic techniques are not able to detect, for instance, overlapping features. This occurs because they rely on the analysis of execution traces that are executed for a specific scenario and considering only user-level features. Other mapping mismatches follow the same rationale, such as *Multi-Partition Features*, *Code Clones*.

3.6 Summary

Developers and maintainers need constantly to understand, restructure and extend features during the maintenance and evolution tasks. The feature mapping activity is important to allow developers to have the full knowledge about all the implementation elements realizing a given feature. Even though there are many tools and techniques that facilitate the feature mapping activity, the developers still need to verify if their mappings are correct and complete. However, there is no characterization and classification of which mapping mismatches are more frequent in the literature.

To fill this gap, this chapter presented and discussed a series of recurring mismatches made by developers when mapping a set of (non-)crosscutting features. Initially, we analysed the feature mappings of an evolving program family. Two experienced developers were in charge of mapping 10 features in the source code of OC family versions by using the ConcernMapper tool (Robillard and Weigand-Warr 2005) (Section 3.1). We classified the mapping

mismatches into two categories, depending whether a feature or module property was the most influential factor for the mismatch (Section 3.2). For each category, we described a set of mismatches and explained how they occur. For instance, high feature tangling (Greenwood *et al.* 2007, Figueiredo *et al.* 2009) is interleaved with other features at the level of methods or modules; and overly communicative features are characterized when there are many dependencies among classes that implement different features. As a consequence, these intricate feature realizations tend to harm the correct identification of elements in the source code. We also identified the relationships among the respective mapping mismatches (Section 3.3).

As a second step, we verified the frequency rate of the mismatch categories through two experiments (Section 3.4). These experiments were important in order to verify if in fact the categorized mapping mismatches are representative and relevant. These experiments involved 26 developers and two different systems with various features. The analysis of the feature mappings demonstrated the occurrence of many documented mismatches. The experimental results confirmed that the mapping mismatches often occur when developers need to interact with the source code.

The feature mapping mismatches tend to be more frequent in degenerate program families due to the different realizations of the same feature for each family member. In this sense, the catalogue of mapping mismatches is a key step to reduce the occurrences of false positives and false negatives during the generation of feature mappings in degenerate program families. These feature mappings are generated by a set of heuristics, so-called mapping expansion heuristics (Chapter 4). In particular, this catalogue is also useful as the mapping expansion heuristics identify features' implementation elements departing from an initial mapping, which can have even more mapping mismatches.