Diploma Thesis

Evolution in Feature-Oriented Model-Based Software Product Line Engineering

submitted by

Christoph Seidl

born December 5, 1982 in Freiburg im Br.

Technische Universität Dresden

Faculty of Computer Science
Institute for Software and Multimedia Technology
Chair for Software Technology

Supervisor: Dipl.-Inf. Florian Heidenreich
Professor: Prof. Dr. rer. nat. habil. Uwe Aßmann

Submitted: August 31, 2011
# Contents

1 Introduction
   1.1 Motivation for Evolving Software Product Lines .............................. 5
   1.2 Outline of the Thesis ......................................................... 6

2 Background and Scope
   2.1 Concepts and Terminology ................................................... 7
      2.1.1 Software Product Lines .................................................. 7
      2.1.2 Model-Driven Software Development .................................... 9
      2.1.3 FeatureMapper ............................................................. 11
   2.2 Scope ................................................................................. 14
   2.3 Related Work ....................................................................... 15

3 Evolution of Software Product Lines ............................................... 19
   3.1 Evolutions ................................................................. 20
      3.1.1 Evolutions in the Problem Space ...................................... 20
      3.1.2 Evolutions in the Solution Space ..................................... 31
   3.2 Classification Systems for Evolutions ....................................... 41
      3.2.1 Classification by Behavior Preservation ............................ 44
      3.2.2 Classification Systems in the Literature ........................... 45
      3.2.3 Classification by Semantical Extent of Model Changes .......... 46
   3.3 Remapping Operations ......................................................... 51
      3.3.1 Remapping in the Problem Space ..................................... 53
      3.3.2 Remapping in the Solution Space ..................................... 60
   3.4 Classification and Remapping of Evolutions ............................... 62
      3.4.1 Classification and Remapping of Problem Space Evolutions .... 63
      3.4.2 Classification and Remapping of Solution Space Evolutions ... 66

4 A Framework for Evolutions in FeatureMapper ..................................... 73
   4.1 Relevant Technology ......................................................... 73
      4.1.1 Refactory ................................................................. 74
      4.1.2 EMFText ................................................................. 78
   4.2 Implementation ................................................................. 79
      4.2.1 Implementation of the Evolutions System ......................... 79
      4.2.2 Implementation of the Remapping System ....................... 90
      4.2.3 Implementation of the User Interface System .................. 101
CONTENTS

4.2.4 Implementation of the Test Suite .................................. 103
4.3 Possibilities for Extension ............................................. 105
4.3.1 Adding New Evolutions ............................................. 105
4.3.2 Adapting Existing Evolutions ................................. 110

5 Example Project .......................................................... 113
5.1 Initial Situation in 2001 ................................................. 115
5.2 First Revision in 2006 .................................................... 118
5.2.1 Removing the Cassette Player ................................. 119
5.2.2 Adding an MP3 CD Player ......................................... 120
5.2.3 Adding a Personal Navigation Device ....................... 128
5.2.4 Changing the Implementation of the UI Builder ......... 133
5.2.5 Summary of the Changes of the First Revision in 2006 .... 135
5.3 Second Revision in 2011 ................................................ 137
5.3.1 Creating a Multi-Format CD Player .......................... 138
5.3.2 Enhancing Voice Recognition to Control the Audio Player . 140
5.3.3 Restructuring Personal Navigation Maps .................. 143
5.3.4 Changing the Implementation of the CD Player .......... 148
5.3.5 Summary of the Changes of the Second Revision in 2011 .... 149
5.4 Conclusion of the Example Project ............................... 151

6 Conclusion ................................................................. 155
6.1 Summarized Findings ............................................... 155
6.2 Limitations and Drawbacks ......................................... 156
6.3 Possibilities for Future Work ..................................... 158
6.4 Theoretical and Practical Contributions ...................... 159

A Object Remapping Specification (*.orspec) .................. 161
A.1 Object Remapping Specification Model .......................... 161
A.2 Object Remapping Specification Syntax ....................... 161

B DocBooklet (*.docbooklet) .............................................. 163
B.1 DocBooklet Model ..................................................... 163
B.2 DocBooklet Syntax .................................................... 163

C NavMap (*.navmap) .......................................................... 165
C.1 NavMap Model .......................................................... 165
C.2 NavMap Syntax .......................................................... 165

List of Figures ............................................................... 166
List of Tables ................................................................. 169
List of Listings ............................................................... 172
Bibliography ................................................................. 173
Chapter 1

Introduction

Software is almost everywhere—as application for smart phones, in the multimedia system of cars or as desktop program for home computers. With this wide variety of platforms, it becomes more and more important to provide programs supporting multiple systems. For instance, an application for a home computer might also be available in a smart phone version. However, different platforms have different characteristics, such as processor speed or display size. Thus, there are equally different requirements for the software of these systems. One successful approach of dealing with this variety is the concept of software product lines. In a software product line, functionality is described in terms of so called features that each realize a set of requirements. For example, one particular feature might provide a user interface for large displays for home computers whereas another feature creates a version for the small displays of smart phones. The appropriate feature may then be chosen to build one particular application for a specified platform.

However, the area of possible applications for software product lines extends beyond providing customized versions of one application for different platforms. In a more general setting, software product lines encompass an entire set of related applications. For example, an office suite might be implemented in terms of features that can be used to create the individual tools such as a text processor or a spread sheet calculation program. These characteristics make software product lines appealing to the industry and a successful approach to software reuse.

1.1 Motivation for Evolving Software Product Lines

As any other software system, product lines are subject to change over time. The reasons for modifications reach from adding new functionality due to altered requirements to performing corrective maintenance. The sum of all possible types of changes is referred to as evolution. Due to the fact that a software product line is presented in the form of individual features that can be combined to a potentially large number of unique applications, it is complicated to foresee the full extent of performing a particular modification and it is easy to unintentionally harm individual applications. Thus, evolving a software product line manually is both tedious and error prone so that tool support is required.

Unfortunately, there are very few practical approaches to the evolution of product lines. However, there are tools for the general work with software product lines. One program in
CHAPTER 1. INTRODUCTION

this area is FeatureMapper, which is able to define software product lines and allows to create individual applications from a specified combination of features. The goal of this thesis is to implement an approach to evolving software product lines in the context of FeatureMapper.

The envisioned system is intended to aid users in altering software product lines by performing the major part of the modification process. Furthermore, a number of possible modifications commonly used when altering software product lines will be provided in the form of specific evolutions. As a result of this mechanism, individual changes are reproducible and their effects on the product line are known in advance. Once the system is realized, it will be possible to evolve software product lines swiftly and without undesired side effects.

1.2 Outline of the Thesis

After this introduction, the rest of the thesis is structured into five chapters. In Chapter 2, background information necessary to understand the explanations in the thesis will be provided. In particular, the basic terms and concepts of software product lines will be introduced. Furthermore, the general idea of model-driven software development will be described and the specifics of FeatureMapper will be discussed. In addition, the scope for the work in the thesis will be set. To close the chapter, a survey of work related to the field of evolving software product lines will be provided.

In Chapter 3, the theoretical foundations for the evolution system of the thesis will be laid. For this purpose, the concrete modifications of software product lines of the thesis will be presented in detail. After that, popular classification systems will be discussed in order to group evolutions by their relevant distinguishing characteristics with respect to their effect on software product lines. Then, operations will be introduced that can be used to remedy the undesired side effects of evolutions when using them in the context of software product lines. Finally, the presented evolutions will be attributed to their respective groups in the classification system and steps necessary to adapt them for the use in software product lines will be introduced.

Chapter 4 will be dedicated to the work required to realize the described theoretical evolution system. For one, the relevant technology the system is built upon will be introduced. Furthermore, the chapter will encompass the concrete implementation of the system with its architecture and a detailed discussion of the realizing classes. As last section of the chapter, an explanation will be presented on how to extend the system with new evolutions in order to adapt it to future requirements.

In Chapter 5, the theoretical and practical work of the previous chapters will be put to test in an extensive example project. During the example, the majority of the presented evolutions will be applied to a number of different models within a variety of use cases. The project will show both applicability and usefulness of the implemented evolution system in a real world scenario.

Finally, Chapter 6 will present a conclusion of the thesis including a brief summary of the previous chapters. Furthermore, limitations and drawbacks of the current approach will be discussed and areas for future work will be inspired. The chapter will close with a list of theoretical and practical contributions of the evolution system, which will emphasize the relevance of the thesis.
Chapter 2

Background and Scope

In the following chapter, background information necessary for the explanations in the thesis will be provided. In the first part, essential concepts and terms will be introduced. In the second part, areas of work will be defined that are within the scope of the thesis. In the third part, a survey of scientific fields adjacent to the topic of the thesis will be presented to close the chapter.

2.1 Concepts and Terminology

In the introduction section, the need for evolutions for software product lines was motivated. However, before details on the theory and implementation of those evolutions can be explained, a number of elementary terms and concepts has to be established. First, the general field of software product lines with its inherent vocabulary will be explained. After that, the basic principles of model-driven software development will be presented. Finally, FeatureMapper, which aids the work with product lines, will be introduced. The information contained in these sections will provide the basis for all subsequent explanations of evolutions for software product lines.

2.1.1 Software Product Lines

A software product line is a systematic approach for software reuse in a series of related applications [PBvdL05]. It separates functionality into a core and multiple so called features. The core functionality is part of all related applications of a software product line. The individual characteristics of a single program of the product line are reflected by a unique combination of features, which are used in conjunction with the core functionality. The concept of features distinguishes a software product line from other concepts of reuse such as frameworks. In a framework, merely the common part of a group of applications is described. The concrete options for variability at extension points are not specified. In contrast, in a product line, the concrete characteristics of all extension points and thus all possible products are described as well. Due to this reason, a software product line is primarily used within a single company that controls its development. Furthermore, a software product line is a commercially successful approach to software reuse [CBS+07], which makes it attractive to the industry.

Formally speaking, a feature is an abstraction of a group of functional and/or non-functional
requirements for the resulting application [Bor11, CAK+05]. One can imagine a product line for
the multimedia system in a car. In this context, features might be CDPlayer, OnBoardComputer
or PersonalNavigation. From a single product line, a multitude of individual applications can
be derived, which are called “variants” [FNS09] or “products”. A variant is created by combining
the functional core with a subset of (or potentially all) the features of the product line. Thus,
each variant of a software product line is defined by a unique combination of features [TBK09].

However, not all constellations of features are permissible as products. Hence, a mechanism
is required that describes rules for which features may be combined to form valid variants. This
is done in what is called a feature model. One popular type of feature model is the feature
tree [Bor11]. It structures all features in a tree-shaped graph and annotates the edges with
additional information. An example of a feature tree can be seen in Figure 2.1, which picks up
the scenario of an automotive multimedia system.

Figure 2.1: Example of a feature tree with annotated edges.

In this example, the feature AutomotiveMultimedia is the root feature of the feature tree.
It is always selected and hence part of all valid configurations. The feature OnBoardComputer
is a mandatory feature, marked by a filled circle on the lower end of the incoming edge. This
means, that the feature has to be selected for all valid variants provided that its parent feature
is itself part of the configuration [AGM+06]. The feature PersonalNavigation is an optional
feature as signaled by the hollow circle at the lower end of the incoming edge. Optional features
do not necessarily have to be part of a valid variant of the software product line and thus can
be deselected at will. The feature CDPlayer describes two child elements AudioCDPlayer and
MP3CDPlayer out of which exactly one has to be selected. This is called an alternative group
and it is visualized by an empty arc between the upper ends of the respective edges in the feature
tree. In contrast, an or group allows selection of one to all features and is visualized by a filled
arc. For example, out of the child features of AudioPlayer, the CassettePlayer, the CDPlayer
or both may be selected.

In addition to the graphical syntax, it is also possible to use textual constraints [Bor11].
For example, a constraint can be used to formulate cross-tree dependencies. In the above
scenario, the use of feature VoiceRecognition only makes sense if at least one of AudioPlayer
or PersonalNavigation is present. Hence, a constraint VoiceRecognition \rightarrow AudioPlayer
OR PersonalNavigation may be introduced, which states that whenever the voice recognition
is selected, an audio player or a personal navigation device has to be present as well.
2.1. CONCEPTS AND TERMINOLOGY

All the features of a feature model describe a view on a software product line from the point of the problem domain. As such, the feature model and all its contained features reside in what is called the problem space [CE00]. The problem space consists of domain specific abstractions [AK08, HKW08] but does not tell anything about the realizing artifacts. Thus, a feature model in the problem space alone can not be used to create executable applications. Therefore, implementation artifacts are needed as a further part of a software product line. These artifacts are part of the so called solution space, which groups assets [BTG10, Bos01] such as source code, design models or documentation [Bor11], which are implementation oriented abstractions [AK08, HKW08].

Even though solution space assets might be logically connected (such as a UML class model and its implementing source code), they are not aware of the features they realize or even that they are part of a feature at all. This kind of knowledge is externalized in the feature mapping model [Bor11, HSS+10, AMC+05], which connects logical expressions of features with their realizing assets. In its most basic form, this means that a single feature is mapped to a number of solution space artifacts. For example, the previously mentioned feature CDPlayer might be mapped to classes in a UML diagram that are responsible for the player’s logic, the source code that implements these classes and a section in the user manual that describes the operation of the CD player. However, a simple one-to-many mapping is not always possible in practical scenarios [HSS+10]. For instance, part of the control logic for playing CDs might be used for the feature AudioCDPlayer as well as for the feature MP3CDPlayer. For this purpose, it is possible to map arbitrary logical expressions of features to solution space assets. In the aforementioned scenario, the respective part of the control logic might be the target of a mapping with a logical expression AudioCDPlayer OR MP3CDPlayer. This statement is satisfied if at least one of both features is part of a variant. Hence, the solution space asset is used by both the audio and the MP3 CD player. This logical term of a feature mapping is called a feature expression and it is also used for product derivation, where a particular feature expression describes a variant of the software product line [HSS+10, Hei09].

2.1.2 Model-Driven Software Development

Model-driven software development is a software development methodology that aims to abstract from the computational details of a program. For this purpose, so called models are employed to capture knowledge of a particular application domain. For example, a model could be used for design specification as with UML\(^1\). A code generator might then create (part of) the source code for the described application. Furthermore, models can also be used for domain specific languages or to describe certain file formats. In either case, a model should be formulated so that it makes sense to a person familiar with the target domain and that it can be used as basis for an implementation [VS\(+\)06]. The main benefit of the model-driven software development approach is its increase in productivity due to simplifying the design of domain specific artifacts.

In model-driven software development, each model conforms to a so called metamodel. Informally speaking, a metamodel is the blueprint for the creation of a model. More formally, a metamodel specifies the type and structure of elements for a particular type of model and how they may be interrelated. As such, a model is an instance of its particular metamodel. This

\(^1\)http://www.omg.org/spec/UML/
relation is captured by the Meta Object Facility\(^2\) (MOF) specified by the Object Management Group (OMG). It defines four different layers of abstraction where each lower layer is an instance of the layer immediately above it. In Figure 2.2, a graphical representation of the MOF layers with example content is presented.

Figure 2.2: The Meta Object Facility layers including example content.

The lowest level is called the M0 Layer. On it, concrete objects of the real world, e.g., objects in a computer program, are represented. The layer immediately above is the M1 layer, which contains models. Models are used to specify the structure of concrete objects of the M0 layer. A popular example for the M1 level is a concrete UML model. The models on the M1 level themselves conform to so called metamodels of the M2 layer. An example for content of the M2 layer would be the specification of UML, which describes that there are elements called classes and associations in the models of UML and that they can be connected. Finally, the M3 layer is used for metametamodels, which describe the structure of metamodels. For example, the OMG uses a language called MOF 2.0 to define the metamodel for UML on this layer.

Due to the complexity of the MOF 2.0 language, it is tedious to implement it to its full extent. As remedy, the OMG has introduced a variant of MOF that solely contains the core elements. It is called Essential MOF (EMOF) and represents a subset of the complete MOF 2.0. Its main intent is to simplify the creation of metamodels. A practical modeling implementation that is widely compatible with EMOF is Ecore of the Eclipse Modeling Framework (EMF) [SBPM08]. EMF is a popular basis for implementing model-driven software development approaches so that

\(^2\)http://www.omg.org/mof/
2.1. CONCEPTS AND TERMINOLOGY

A wide variety of applications uses Ecore. Most notably for this thesis, FeatureMapper employs Ecore-based models for software product lines. This means, that the artifacts in the solution space of a software product line have to be instances of Ecore classes in order to target them with the tool. Furthermore, the feature tree and the mapping in FeatureMapper are instances of Ecore metamodels as well. In consequence, FeatureMapper does not target artifacts such as source code or documentation directly but requires a suitable model representation of the elements in order to be applicable. In other words, this means that FeatureMapper exclusively focuses on model-based software product lines. Further specifics of the tool will be explained in the following sections.

2.1.3 FeatureMapper

FeatureMapper is an application for working with model-based software product lines, which is being developed at Technische Universität Dresden [HKW08, HSS+10]. The tool can be used to create feature models and a mapping from feature expressions to solution space artifacts. Furthermore, valid configurations of features can be used to create a concrete variant of a product line. For this purpose, the feature mapping is used to resolve the selected features to all mapped solution space assets. The individual characteristics of FeatureMapper when dealing with problem or solution space will be explained in the following two sections. Furthermore, the mapping will be discussed in the third section.

2.1.3.1 The Problem Space in FeatureMapper

In the problem space, FeatureMapper uses a cardinality-based feature tree [Hei09, CHE05], which varies slightly from the aforementioned regular feature tree. However, the cardinality-based feature tree is able to express all rules contained in the regular feature tree and merely uses a different representation. Most notably, the concepts of features and groups are separated into individual entities. Furthermore, each feature has a name assigned to it that is assumed to be unique throughout the product line. Generally, a feature may contain an arbitrary number of groups and groups may contain any number of features. Both groups and features have a minimum and maximum cardinality.

For groups, the cardinality specifies how many of the child features may be selected for a valid variant. Therefore, an alternative group is a group where exactly one of the child features may be selected. Hence, it has a minimum and a maximum cardinality of one. On the other hand, for an or group, between one and all child features may be selected, which results in the respective minimum and maximum cardinality. In order to avoid logical contradictions, the minimum cardinality of a group may never be less than zero and the maximum cardinality must not exceed the number of contained features. Naturally, the minimum cardinality must not be greater than the maximum cardinality.

For features, the cardinality specifies how often the feature itself may be selected for a valid product. This means that optional features have a value of zero for the minimum cardinality, whereas mandatory features have a value of one. Values larger than one are not permissible for either minimum or maximum cardinality of features as they would imply that a feature is used multiple times and FeatureMapper does not support cloned features [CHE05].
As all the constructs of the aforementioned feature tree can be translated to entities of the cardinality-based feature tree, it is obvious that all of its models can be converted to cardinality-based notation as well. Figure 2.3 shows the same model as Figure 2.1 but in the cardinality-based notation used by FeatureMapper. When compared to the regular feature tree, using a cardinality allows for even more precise descriptions. For example, a group containing four features may have a minimum cardinality of two and a maximum cardinality of three. This cannot be expressed in the notation of the regular feature tree. In addition, the effects of many of the evolutions described in this thesis can be expressed much more uniformly by using a cardinality than with the syntax of the regular feature tree.

2.1.3.2 The Solution Space in FeatureMapper

In the solution space, FeatureMapper imposes very few constraints on the form of the realization artifacts. It is only required that the targeted asset has a representation in the form of an EMF\textsuperscript{3} Ecore Model [HKW08]. There are predefined metamodels of popular formats such as UML\textsuperscript{4}. Furthermore, EMFText\textsuperscript{5} provides the possibility to use a great number of textual languages. Among others, the complete syntax for the Java programming language is provided and the syntax for other languages, such as documentation formats, can be specified as well. FeatureMapper provides great liberty in the use of models as they do not have to be adapted to being used in the context of a software product line at all. In particular, models do not have to be aware that they are part of a feature mapping which entails that common editors familiar to users may still be employed for the modification of solution space models.

For the content of the solution space models, it is worth noticing that FeatureMapper uses a negative variability approach [HSS\textsuperscript{+10}]. This means that all possible combinations are comprised.

---

\textsuperscript{3}The Eclipse Modeling Framework (EMF) provides tools to work with models. Among others, it specifies Ecore as a representation for metamodels. See \url{http://www.eclipse.org/emf} for further information.

\textsuperscript{4}\url{http://www.eclipse.org/uml2}

\textsuperscript{5}EMFText is a project of Technische Universität Dresden, which allows specification of textual syntax for an arbitrary Ecore metamodel and creates parsers and editors from this specification. EMFText currently provides the syntax for approximately 90 different languages. See \url{http://www.emftext.org} or Section 4.1.2 for details on EMFText.
2.1. CONCEPTS AND TERMINOLOGY

In one large model and during the product derivation process, unused parts are removed. For example, the original UML model in a product line would contain all classes of all features even if any given variant of the product line would only contain a subset of them. When a concrete variant of the product line is created, the parts not relevant to the selected features are removed from the model. All model elements that are not related to any feature are considered part of the core of the product line and thus are included in all variants [HSS+10]. The opposite of negative variability is positive variability where many smaller models are composed according to the selected features to create a concrete product. However, this approach tends to require many small artifacts, which are hard to create and maintain individually. Therefore, FeatureMapper employs a negative variability approach.

2.1.3.3 The Mapping in FeatureMapper

FeatureMapper uses an external annotation model for its feature mapping [HKW08]. This means that neither problem nor solution space models are aware of the fact that they are part of a mapping. Instead, the mapping information is kept in a separate Ecore-based model. Due to this fact, changes to the feature mapping can be performed largely with the same tools and approaches as modifications to the feature model or solution space models, which are Ecore-based models as well. This characteristic will be exploited by a part of the work in this thesis.

Conceptually, a feature mapping connects the problem and the solution space. Hence, it has an end in either one of these spaces. In the problem space, an expression of features connected by the logical operators AND, OR and NOT is provided. In FeatureMapper, this expression is referred to as “term”. The end in the solution space depends on the type of mapping that is used. FeatureMapper distinguishes three different mappings. In Figure 2.4, a UML diagram of the relevant classes is presented as overview. Furthermore, all three types of mappings will be explained in the following paragraphs.

Figure 2.4: UML diagram of the different types of mappings in FeatureMapper.

The first type of mapping is the element mapping. It relates a feature expression from the problem space to one particular solution space element, which has to be a subclass of an EMF EObject. Examples of solution space elements might be a class in a UML diagram or a
CHAPTER 2. BACKGROUND AND SCOPE

paragraph in a documentation format provided that there are Ecore-based models for them. If the logical expression is satisfied by a concrete selection of features, the element will be included in the variant.

The second type of mapping is the property value mapping. It maps a feature expression to a value that will be assigned to a certain solution space element. From a technical view, a property value mapping contains a reference to the targeted solution space element and an arbitrary number of so called property values. In turn, a property value contains a feature expression and a value that will be assigned to the solution space element of the property value mapping.

For example, a property value mapping might have the multiplicity of a UML association end as target element. Then its property values could specify numbers for the multiplicity that are used when their respective terms are satisfied by a product configuration. When a feature Small Size is used, this number might be 50, when feature Large Size is used it might be 100. The possible types of property values are an empty value, a primitive value (such as a number or a string) and using another EObject as target.

Finally, the last type of mapping is the color mapping. It relates a user chosen color to a particular feature expression. The color can then be used to highlight all solution space artifacts affected by the respective feature expression. Color mappings are special as they do not reference a solution space element. Due to this reason, color mappings have to be treated with care when evolving a software product line.

It is worth noticing that FeatureMapper allows at most one mapping to an individual solution space element at all times. This means, if two separate features would each map to a certain EObject, it is impossible to use two individual rules. Instead, one mapping has to be specified, that uses an OR connection of both features as its term.

2.2 Scope

With the background information of Section 2.1, it is possible to define the scope of work for the thesis. The overall goal is the implementation of evolutions for software product lines that keep the feature mapping and all possible products intact. As the results are to be implemented in FeatureMapper, there is a focus on model-driven development of product lines with Ecore-based metamodels. During the course of the thesis, theoretical and practical aspects of evolving product lines will be discussed.

In the theoretical part, it is expected to find relevant evolutions of feature-oriented model-based software product lines. It is to be investigated, how the modifications of model elements performed by these evolutions affect the rest of the product line. A special focus is on examining the effects on the mapping model. Furthermore, means have to be found that compensate for model changes that might destroy the mapping by adapting the feature mapping so that the validity of the product line as a whole is maintained. The conceived evolutions are to be categorized by their significant distinguishing characteristics with respect to their effect on the product line.

In the practical part, both the evolutions and the concepts for maintaining the feature mapping are to be implemented in FeatureMapper. The technical validity of the implementation
2.3 RELATED WORK

is to be shown by use of a test suite. Furthermore, the feasibility of the concepts is to be confirmed by means of an example scenario where the majority of the evolutions is applied to a software product line.

Besides defining requirements for the results of this work, a precise definition of essential terms is necessary to set the scope. In this thesis, the expression “evolution” is defined as altering an existing product line [CPGS07, MWD+05, BTG10, SB99, HP00, MR06]. An evolution might be a change that maintains functionality or behavior but could also be an arbitrary modification. Most notably, “evolution of software product lines” does not refer to changing an individual product or a product family towards creating a software product line as it does for many authors in a different context [KK07, AMC+05, Bos02, LBL06, Aoy01].

Based on this definition, it is assumed that there is a valid software product line present before any modification is applied. Within this project, evolutions are to be conceptualized and implemented that maintain this validity of the feature mapping and the software product line as a whole. However, it is not within scope to prove the relevance of these modifications to practical scenarios.

Furthermore, there will be no mechanism for the detection of changes to relevant models that were performed outside of FeatureMapper. Instead, all evolutions are assumed to be executed exclusively from within the tool.

To aid users during the evolution of product lines, it is intended for the processes of modifying models and adapting the feature mapping to be automated as far as possible. However, it is not feasible to aim for a fully automated approach as there are points during the process where a user’s choice is required. Furthermore, it is argued that a semi-automatic approach to evolution can greatly increase productivity [MT04, Bor11] whereas a fully automatic approach tends to over complicate matters because it might make changes for generalities sake that are not needed in the concrete case [MT04]. As one particular example, users should have the option to not perform modifications of the feature mapping at all.

Due to the flexibility of FeatureMapper in handling arbitrary models, very little a priori knowledge on the characteristics of the targeted artifacts exists. Thus, the implementation of evolutions has to be generic in the sense that it can easily be adapted to the structure of a specific type of model. Approaches that entail adding additional information to models, such as annotation, are not feasible as they would defy the general applicability of FeatureMapper. This entails that measures for adapting the feature mapping should be configurable via a declarative approach.

Lastly, all evolutions will target model-based software product lines. Other approaches, such as aspect oriented product lines, will not be covered in this thesis.

2.3 Related Work

Even though the topic of evolving software product lines is relatively young, it has many adjacent scientific areas that can be used as inspiration. In particular, there is substantial work in the field of modifying software systems and classifying particular modifications. The latter will be discussed in detail in Section 3.2 and will thus be omitted at this point for the sake of brevity. However, the literature in the area of modifying existing software systems will be surveyed in
this section. In particular, work in the area of code and model refactoring will be inspected.

The field of code refactoring is well established and catalogs of common modifications, such as [Fow99], exist. However, the work in this thesis focuses on modifying model-based artifacts rather than textual source code. Even though there are parallels between the fields of code and model refactoring, there also are significant differences. For example, source code refactoring is performed on a syntax tree, whereas model refactoring works on graph shaped structures. As a consequence, both areas face different challenges and the insights from the field of code refactoring can not necessarily be employed for the evolution of software product lines. Yet, some of the refactorings inspired evolutions in this thesis as, for example, a model representation of the Java programming language may be target of an evolution. In particular, modifications from the catalog of Fowler in [Fow99] inspired the evolutions in Section 3.1.2.2. Furthermore, some of the refactorings defined for source code can be adapted to be used on UML diagrams as both artifacts have largely similar structures such as classes and methods.

In addition to the established area of source code refactoring, the relatively young [CH06] field of model refactoring also influenced the work in this thesis. Most notably, the work of Reimann et al. in [Rei10] and [RSA10] provides a basis for the implementation of evolutions. The authors present a system for generic model refactoring called Refactory. It allows to specify arbitrary modifications for models on an abstract level and then map them for use with a concrete metamodel. This functionality builds the basis for the implementation of the evolutions presented in this thesis. Due to its pivotal role in the implementation, Refactory will be explained in more detail in Section 4.1.1. Zhang et al. present a different approach to model refactoring using a model transformation engine in [ZLG05]. Similar to Refactory, the tool of the authors allows to target generic or domain specific models. However, it builds upon an editor called GME (Generic Modeling Environment), which uses a proprietary language for the specification of metamodels. In contrast, Refactory employs EMF Ecore to specify metamodels, which integrates seamlessly with FeatureMapper as both tools build on the same modeling framework. Thus, Refactory is preferred for the implementation of evolutions.

Furthermore, there is additional work in the field of model refactoring that focuses on the process of transforming a model. For example, Porres [PCS03] presents a system of rule based transformations that can be used to modify a particular model. The described rules feature a precondition and an execution part. The precondition checks whether a given transformation is applicable for the current part of the model and the execution part performs the modifications. Specifying preconditions to check the applicability of a modification inspired the design of the implementation for evolutions in this thesis. The respective classes use a similar mechanism to ensure that a particular evolution can only be executed on the current selection if the specified criteria are met (see Section 4.2.1). In addition, Varró [Var06] proposes a system that can derive rules for transforming a source model to a target model from a set of examples. With this approach, it might be possible to create a system that extends its set of evolutions by monitoring the work of its users. This mechanism would make a useful extension of the evolution system but is out of the scope of the thesis.

The work in the field of source code and model refactoring mostly influenced the design of modifications for the solution space of a product line. However, there is also work on evolving the feature tree in the problem space. On a theoretical level, there is work of various authors on
the topic. For example, in [AGM+06], Alves et al. introduce a number of different modifications that can be applied to the feature model in the problem space of a software product line. The described steps of modification are very fine grained, which makes it hard to employ them as useful practical evolutions directly. However, their general idea can be used to build more complex evolutions. In the concrete case, the described steps inspired the creation of the Insert Feature (see Section 3.1.1.2) and Pull Up Feature (see Section 3.1.1.6) evolutions. Furthermore, Svaen-berg and Bosch [SB99] present the findings of a case study of the evolution of software product lines in two companies. As part of their work, they identified concrete steps of modification that are commonly used in software product line evolution. These findings inspired the Split Feature evolution of the thesis (see Section 3.1.1.4). Finally, Czarnecki et al. [CHE05] describe the so-called specialization of feature models by reducing their configuration options. As part of their work, the authors define the steps for a modification that has been implemented as Remove Feature in this thesis (see Section 3.1.1.7). In addition, Botterweck et al. have done significant work in the area of software product line evolution. For example, in [BPD+10], the authors introduce a system for planning the evolution of software product lines in terms of changes to features. Unfortunately, the general focus of the paper and other work by the authors such as [BPPK09, EBLSP10] is on planning evolution and not on performing it so that their work can not serve as basis for the thesis. However, a short catalogue of problem space evolutions is presented as part of the paper [BPD+10], which contains modifications similar to the evolutions of the variation type of features (see Section 3.1.1.1) and groups (see Section 3.1.1.9) presented in this thesis.

Besides the theoretical work, there is also a practical application of evolutions of a feature model. The program pure::variants6 is a tool for the work with software product lines much like FeatureMapper. Besides its core functionality to create variants of a product line, the tool also offers basic support for evolving a feature model. In particular, it offers the option to change the variation type of features and groups. This means, that a feature can be made “optional” or “mandatory” and groups can be altered to be “alternative” or “or” groups. These modifications inspired the evolutions Transform to Optional Feature and Transform to Mandatory Feature for features as well as Transform to Or Group, Transform to And Group and Transform to Alternative Group for groups.

Even though there is extensive work on the subjects of code and model refactoring that influenced the work on this thesis, none of the presented approaches can be employed directly to evolve software product lines while maintaining the feature mapping. This is mostly due to the fact that the presented modifications are merely inspected in the context of their containing model. However, in the course of a software product line, a model resides in either problem or solution space and may be connected to the respective other space via feature mapping. Thus, the effects of an evolution may be far more extensive than can be captured by inspecting a single model. Before these evolutions can be applied in the context of a product line, it has to be determined whether they need to be adapted.

The tool pure::variants is an exception to the above problem as it successfully applies evolutions to the feature model of a software product line. However, the few provided modifications are elementary in nature so that they do not even threaten the validity of the mapping model.

---

6http://www.pure-systems.com/pure_variants.49.0.html
CHAPTER 2. BACKGROUND AND SCOPE

(see Section 3.2.3.1). The tool possesses no dedicated means to counter the effects of more complex evolutions on the mapping model. Thus, the evolution system of pure::variants can not be used for the general case of evolving a software product line. Due to the lack of a suitable alternative, this thesis will present an original approach to evolving software product lines while maintaining the validity of the mapping model.
Chapter 3

Evolution of Software Product Lines

In the course of this thesis, the expression “evolution” is used as a synonym for an arbitrary modification of a software product line. When studying the literature on adjacent topics, one often finds the term “refactoring”. A refactoring is defined as a change to a software system that preserves semantics and/or the user visible behavior of the system [Fow99, Bor11]. Certainly, this quality is of relevance to scenarios such as restructuring an existing application while maintaining its original functionality. However, there are also cases where behavior preservation is of no relevance. For example, during corrective maintenance, preserving semantics most likely is not desirable at all because the misbehavior of a system is to be fixed [CPGS07]. In fact, in many scenarios it suffices to know how a modification affects behavior rather than relying on the change being semantics preserving [MTM07, Rei10, RSA10].

The modifications presented in this thesis are intended to be used in a multitude of evolution scenarios. On the one hand, they may be used for continuous evolution [Aoy01] where only relatively small changes are introduced to a product line. On the other hand, they also have to be applicable in discontinuous evolution scenarios [Aoy01], where the majority of the product line changes in one increment. At the same time, the introduced changes might be part of adding new functionality, fixing existing problems or restructuring the product line. Due to this diversity of possible application scenarios, the modifications presented in this thesis are not limited to semantic preserving changes. Consequently, all modifications are uniformly referred to as evolutions.

This chapter will discuss the theoretical part of evolving software product lines in four sections. First, the evolutions of this thesis and their functionality will be presented. Then, common approaches to classifying evolutions of a product line with their respective problems will be discussed and a new classification will be introduced. After that, remapping operations will be inspected, which are required to maintain the consistency of a product line when applying specific evolutions. Finally, the introduced evolutions will be attributed to their respective groups in the presented classification and the concrete steps required for remapping will be explained.
3.1 Evolutions

The evolutions presented in this thesis have a large number of possible application areas. For example, evolutions might add new functionality or increase maintainability of a particular realization artifact. Defining evolutions for product lines means covering a wide variety of different models. In the problem space, modifications for the feature model have to be provided. In the solution space, a large number of different models have to be targeted. In the course of this thesis, UML, Java and a fictitious documentation format called DocBooklet are used as representatives of solution space models that can be evolved. However, the presented approach can be extended to target arbitrary other Ecore-based models. According to the distinction of the problem and solution space, the realized evolutions are presented in two separate sections describing the modifications in each of the spaces.

Besides that, it was considered to introduce a separate section for modifications that only affect the mapping model, such as removing the mapping of a particular feature or solution space element. This type of modification is characterized by the fact that it does not alter the structure of problem or solution space models. Instead, it merely performs changes to the mapping model. Due to this reason, this type of change does not constitute a full-fledged evolution but rather is a pseudo-evolution for the mapping. Section 3.3 will show that these alterations of the mapping model actually are a remapping procedure that might also be necessary for other evolutions to keep the product line consistent. Thus, pseudo-evolutions for the mapping can be implemented using the functionality provided in this thesis if needed. As there are no real modifications performed on the models of either problem or solution space, the pseudo-evolutions are considered void alterations followed by a remapping procedure. Due to this fact, it was refrained from dedicating a separate section to this type of modification. Instead, actual evolutions for the problem and solution space are described in detail.

3.1.1 Evolutions in the Problem Space

Evolutions in the problem space modify the feature model of a product line. This means that either features or groups of a feature tree are affected by each of the evolutions. The following sections will present nine different evolutions for features and five for groups, which create, modify and delete elements of the feature tree. A special case is the evolution Remove Feature and Owned Assets, which does not only modify the feature tree but also alters solution space models as will be explained in Section 3.1.1.8. It is worth noting that changes to features may also have an effect on other elements of the feature tree. For example, whenever a feature ceases to exist, it has to be checked if constraints that applied to the removed feature are now obsolete. Among others, this may happen for the evolutions Split Feature and Remove Feature explained in the following.

Besides the implemented features of this thesis, there were a number of candidate evolutions that were rejected for various reasons. For one, the evolution Rename Feature was designed to assign a new name to an already existing feature. As all references to a particular feature use the same memory object in FeatureMapper, there is only a single place where the name is stored. As the name of the feature instance can be changed by setting the respective property, a dedicated evolution would have no additional benefit.
3.1. EVOLUTIONS

Besides Rename Feature, two further evolutions were conceptualized but not implemented. The evolution Move Feature was designed to relocate an existing feature from its current group to a new group in the feature tree adapting the group’s cardinality. For the evolution to be applicable, it is necessary to select a suitable target location. The evolution Push Down Feature was intended to move a feature downwards in the hierarchy to a more specialized group making the feature dependent on its new parent [AGM+06]. As there might be multiple potential groups for the push down, a selection of the appropriate target is required as well.

Implementing this functionality currently is not possible when using Refactory (see Section 4.1.1) as is done in this thesis. The tool uses generic refactorings based on roles that are resolved to concrete objects during runtime of the evolution. In order to offer suitable targets for an evolution that relocates features, knowledge of which objects play a certain role would have to be available before the user input phase of the evolution. Unfortunately, at the current stage of development, Refactory retrieves this information too late in the control flow so that this type of evolution can not be realized. Yet, in total, there are 14 evolutions that have been implemented, which constitutes a solid basis for modification of the problem space.

3.1.1.1 Evolutions for the Variation Type of Features

The variation type decides whether a feature is mandatory or optional. This distinction is made by setting the minimum cardinality of a feature either to one or to zero. The maximum cardinality has to be set to one in either case because cloned features are not supported. Two evolutions are provided to aid in the process of changing the variation type of a feature.

The evolution Transform to Mandatory Feature modifies the cardinality to create a mandatory feature. In detail, it changes both the minimum and maximum cardinality to one. The evolution Transform to Optional Feature changes the cardinality of the selected feature so that it becomes optional. Hence, its minimum cardinality is set to zero and, if necessary, its maximum cardinality is changed to one. The modifications of the variation type of features were inspired by [BPD+10] and pure::variants. In Figure 3.1, examples of both evolutions for the variation type of features are displayed.

Even though the evolutions for the variation type of features seem somewhat artless, opportunities for their use arise rather frequently. For example, they are a great aid in reworking the results of other evolutions, such as Split Feature (see Section 3.1.1.4), by transforming optional to mandatory features and vice versa. Furthermore, both evolutions can be used on a selection of multiple features at once, which saves time when compared to editing the cardinality manually.

3.1.1.2 Insert Feature

The Insert Feature evolution creates a new feature within a group. Its intended use is to create an entirely new feature in a product line that has little to no similarity to already existing features. The new feature is created as an optional feature and its name is determined by user input. When the evolution is executed on a group, the feature is added as last child. However, if it is performed with another feature being selected, the new feature is inserted directly behind the selected one in the same group. Executing insert feature on the root element of the feature tree produces no result as the root feature does not have a parent group and thus may not have any sibling features.
Figure 3.1: Example of the evolutions for the variation type of features.

(a) original feature tree

(b) after Transform to Mandatory Feature was applied to feature FeatureA

(c) after Transform to Optional Feature was applied to feature FeatureG

Figure 3.2: Example of a feature tree before and after the Insert Feature evolution.

(a) original feature tree

(b) after Insert Feature was applied to feature FeatureF
As part of the evolution, the cardinality of the parent group is modified in accordance with the performed changes. As the new feature is an optional feature and thus has a minimum cardinality of zero, the minimum cardinality of the parent group does not have to be altered. However, the maximum cardinality of the parent group is increased by one provided that it was equivalent to the number of features in the group before adding the new feature. An equivalent modification of the feature tree can be found in [AGM+06]. In Figure 3.2, the effects of the Insert Feature evolution are visualized.

It was refrained from using the name “Add Feature” for the evolution to disambiguate it from the operation of creating a new feature using the context menu of the mapping view in FeatureMapper. This operation would only add a new child element but not adapt the cardinality of the parent group in accordance with the change.

3.1.1.3 Duplicate Feature

The Duplicate Feature evolution copies an existing feature within its group. Its intended use is to create a new feature that is largely similar to an already existing one. For this purpose, the evolution creates an exact clone of the original feature by copying its minimum and maximum cardinality. Furthermore, all constraints that apply to the original feature are copied to the new feature as well. However, the name of the new feature is determined by user input. If necessary, the cardinality of the parent group is adapted to compensate for the new feature. This means, if the maximum cardinality of the group is equal to the number of contained features before the duplication, the value is increased by one. Furthermore, the minimum cardinality is increased as well if the minimum cardinality of the group was equal to the number of mandatory features and the new feature itself is mandatory. In this case, the minimum cardinality is equal to the sum of the minimum cardinality of each of its contained features. Additionally, it is possible for Duplicate Feature to copy all child elements of the selected feature as well. The decision on whether this operation is performed is left to the user. In Figure 3.3, the effects of the Duplicate Feature evolution are shown in an example.

Figure 3.3: Example of a feature tree before and after the Duplicate Feature evolution.
CHAPTER 3. EVOLUTION OF SOFTWARE PRODUCT LINES

3.1.1.4 Split Feature

The *Split Feature* evolution breaks up a selected feature into its constituent parts. It is used when a feature has to be modeled in a more fine-grained way than it currently is. It is up to the user’s decision in how many parts a feature is split as long as the number is greater than two (because otherwise no split would be performed). Furthermore, users are allowed to provide names for the split parts.

In the process of the evolution, the selected number of features is added as children to a new group below the original feature. All split parts have the same variation type as the original feature, which means that they have an equivalent cardinality. Furthermore, the containing group is created with the cardinality of an and group. In addition to that, constraints that originally applied to the selected feature are remodeled to now constrain the split parts. The *Split Feature* evolution was inspired by [SB99]. In Figure 3.4, an example of the evolution is presented, which demonstrates the process of modification.

Figure 3.4: Example of a feature tree before and after the *Split Feature* evolution.

3.1.1.5 Merge Features

The evolution *Merge Features* melds multiple source features to one target feature. The intended use is to make the product line more coarse grain by ridding it of too detailed features while still maintaining the original functionality. The merged feature should satisfy all requirements that applied to the source features. The *Merge Features* evolution is useful because too many features clutter the feature tree and make it hard to maintain [HSS+10].

The evolution encompasses two different kinds of merge. As first case, two or more sibling features can be merged onto a new target feature that resides on the same level. In this case, a name for the newly created feature may be provided by the user. In a second scenario, an arbitrary number of features can be merged with their parent feature. In the case that more than one feature is selected, the features have to be siblings as well. The second type of merge
3.1. EVOLUTIONS

is capable of inverting the changes made by the Split Feature evolution. In Figure 3.5, examples of both types of merge are displayed.

Figure 3.5: Example of a feature tree before and after the Merge Features evolution with both types of merge.

In order to perform the merge, a number of physical changes to the feature model is required. First, the selected features are deleted and then, the target feature of the merge is created if necessary. However, semantically the features are perceived as being melded into one.

3.1.1.6 Pull Up Feature

The Pull Up Feature evolution relocates a feature in the hierarchy in upwards direction. The evolution should be used when a particular feature has been given additional responsibilities that make it independent of its containing group. The procedure of pulling up a feature is implemented in accordance with the Pull Up Node operation described in [AGM+06]. In the first step, the feature’s parent group and its containing superordinate feature are determined. The superordinate feature itself is contained within another group. Then, the selected feature is relocated to be a child of the group containing the superordinate feature. Thus, the selected feature becomes a sibling of its old parent feature. The cardinality of the original and the new group of the selected feature are adapted to compensate for the changed number of contained features. Furthermore, the original group is removed if it is empty now that the selected feature
has been relocated. In Figure 3.6, an example is shown where the evolution is applied.

Figure 3.6: Example of a feature tree before and after the Pull Up Feature evolution.

(a) original feature tree

(b) after Pull Up Feature was applied to feature FeatureE

Naturally, Pull Up Feature cannot be performed for the root feature as it has no parent group. Neither is it possible to pull up features that are on the first level after the root feature. In this scenario, the selected feature would have to become a sibling of the root feature, which is impossible because the root feature has to be unique.

3.1.1.7 Remove Feature

The Remove Feature evolution deletes a given feature from the feature tree. It is used to alleviate the feature model of a feature that is no longer required—potentially because it exclusively fulfilled requirements that recently were rendered obsolete.

Figure 3.7: Example of a feature tree before and after the Remove Feature evolution.

(a) original feature tree

(b) after Remove Feature was applied to feature Feature A

Along with the originally selected feature, all child features are deleted from the feature tree. Furthermore, the evolution keeps track of all constraints that applied to the deleted features. Out of this set, all constraints that no longer reference a feature after the removal are deleted as well. Finally, the cardinality of the group that contained the originally selected feature is adapted. The procedure is analog to that of the majority of the other problem space evolutions.
However, the difference is that the cardinality of the group is reduced instead of increased. In detail, this means that for mandatory features the group’s minimum cardinality is reduced by one if it was equivalent to the number of mandatory features in the group before the removal. For optional features, which have a minimum cardinality of zero, no change of a group’s minimum cardinality is required. Similarly, the maximum cardinality of a group is decreased by one if it was equivalent to the number of all features in the group before the evolution. The creation of the Remove Feature evolution was inspired by [CHE05]. In Figure 3.7, an example of applying the evolution is displayed.

### 3.1.1.8 Remove Feature and Owned Assets

When a feature of a product line is rendered obsolete over the course of time, it may be desirable to delete it from the feature tree and further remove all realization artifacts that implemented it. For this purpose, the Remove Feature and Owned Assets evolution is provided. The first part of the evolution is to delete a feature from the feature tree analog to the procedure in the Remove Feature evolution. This means, that descendant features are removed along with the originally selected feature. Furthermore, for Remove Feature and Owned Assets, the implementation artifacts used exclusively to realize the deleted features are removed from the solution space. To ensure that the removal does not impact the realization of other features by mistake, only elements that are owned by the deleted features are removed.

Figure 3.8: Example of a feature tree and UML model before the Remove Feature and Owned Assets evolution.

A solution space element is owned by a particular feature if two conditions are met. First, only mappings to the element may exist whose feature expression has no references to other features than the ones deleted during the course of the evolution. It is worth noticing that both an element mapping and a property value mapping may point to the same solution space element. The second condition is that none of the descendant elements of a particular solution space asset contradict the prior findings. This would happen if a descendant element is part of a mapping whose feature expression contains references to other features than the deleted ones. This would suggest that descendants of the solution space element are in fact used by features that remain in the feature model and thus, the realization artifact must not be removed. With
these criteria, it is possible to locate all solution space assets that are owned by the deleted features. These elements are then deleted as well.

Figure 3.9: Example of a feature tree and UML model after the Remove Feature and Owned Assets evolution.

In addition to the owned solution space assets, elements depending on these artifacts may also be deleted. For example, in UML models, an association to a class that was deleted as owned element should also be removed. Otherwise, the remaining association would be invalid because it targets a no longer existing element. The deletion of dependent associations is part of the Remove Feature and Owned Assets evolution and helps in evolving product lines that contain UML models. In Figure 3.8 and Figure 3.9, an example of a feature tree and UML model before and after the evolution was applied is displayed.

3.1.1.9 Evolutions for the Variation Type of Groups

The variation type of a group is determined by its minimum and maximum cardinality. It decides whether a group is an “or”, “and” or “alternative” group. An or group requires at least one but allows all features to be selected. An and group requires all mandatory features to be selected but has no restrictions regarding optional features so that they may or may not be selected. Finally, an alternative group requires exactly one of its features to be selected.

There are three different evolutions that allow the modification of the variation type of a group and thus transform it to one of the aforementioned types. For Transform to Or Group, the minimum cardinality of a group is set to one. The maximum cardinality is equal to the number of features contained in the group. In contrast, Transform to And Group sets the minimum cardinality to the number of contained mandatory features. As with an or group, the maximum cardinality is set to the number of all features contained in the group. Lastly, Transform to Alternative Group sets both minimum and maximum cardinality to one. Modifications similar to the presented evolutions can be found in [BPD+10] and pure::variants. In Figure 3.10, Figure 3.11 and Figure 3.12, examples of each of the evolutions for the variation type of groups are shown.

Besides the evolutions for the variation type of features, the modifications of the variation type of groups are probably among the most heavily used evolutions in the problem space. This
3.1. EVOLUTIONS

Figure 3.10: Example of a feature tree before and after the Transform to Or Group evolution.

(a) original feature tree

(b) after Transform to Or Group was applied to the group containing FeatureC and FeatureD

Figure 3.11: Example of a feature tree before and after the Transform to And Group evolution.

(a) original feature tree

(b) after Transform to And Group was applied to the group containing FeatureC and FeatureD

Figure 3.12: Example of a feature tree before and after the Transform to Alternative Group evolution.

(a) original feature tree

(b) after Transform to Alternative Group was applied to the group containing FeatureE and FeatureF
3.1.1.10 Duplicate Group

The evolution *Duplicate Group* creates an exact clone of a selected group. It is intended to be used when an entire set of features has to be duplicated in order to create similar new features. The newly created group has the same cardinality as its archetype. During the process of duplicating a group, all contained features and their descendant groups and features are cloned as well. The functionality of *Duplicate Group* is demonstrated by use of an example in Figure 3.13.

Figure 3.13: Example of a feature tree before and after the *Duplicate Group* evolution.

![Feature Tree Example](image)

(a) original feature tree  
(b) after *Duplicate Group* was applied to the group containing FeatureE and FeatureF

3.1.1.11 Merge Groups

The *Merge Groups* evolution combines the child features of multiple groups within a single group. The evolution is intended to be used when the features of two or more separate groups are meant to be made part of the same group. On an operational level, *Merge Groups* detaches the children of all but the first selected group from their original parent group and then reattaches them as children of the first group. After that, the cardinality of the group is adapted to compensate for the newly added features while trying to preserve its original variation type. For or groups, the minimum cardinality is not altered whereas for and groups it is increased by the number of newly added mandatory features. For both types of groups, the maximum cardinality is
increased by the number of newly added (optional and mandatory) features. On the other hand, alternative groups maintain their minimum and maximum cardinality of one in either way. All other selected groups are empty after these modifications and thus are removed from the feature model.

Figure 3.14: Example of a feature tree before and after the Merge Groups evolution.

For the evolution to be performed, at least two groups have to be selected as input. Furthermore, all selected groups have to be direct children of the same feature and thus are siblings on the same level of the feature tree. In Figure 3.14, an example of the Merge Groups evolution is shown.

3.1.2 Evolutions in the Solution Space

FeatureMapper is capable of handling a wide variety of different models in the solution space. Likewise, the evolution mechanism should be capable of providing different modifications for this great number of models. The basis of the system can be used to specify evolutions for arbitrary metamodels. In this chapter, evolutions for three different target metamodels are introduced as an example. The first type of model is UML, which is used for design specification. The second type of model results from translating source code in the Java programming to a model instance. An explanation of this process is presented in Section 5, where the evolutions are applied in an example project. Finally, the third type of model is the so called DocBooklet format. It is a toy language to serve as representative of textual languages for documentation. A more complete explanation of the format can again be found in Section 5. Furthermore, the definition of the language can be obtained from Appendix B.

The evolutions presented in the following section merely provide a glimpse at model refactoring. Work dedicated solely to the topic, such as [Rei10, MMBJ09, Mar05], gives a more complete overview. However, this sample of evolutions shows the applicability of the presented approach to arbitrary solution space models.
3.1.2.1 Evolutions for UML

The Unified Modeling Language (UML) describes a graphical syntax for design specification of software applications. Through this, its structures are largely similar to those of object oriented programming languages. For example, in UML, there are classes, methods and attributes, which are essential parts of object oriented programming languages as well. Due to this close relation, many evolutions that can be applied to source code have a largely equivalent counterpart for UML models. The following sections will explain evolutions for UML diagrams that could equally be described for source code. Even though the list of evolutions for UML diagrams is by no means complete, the selected examples show the applicability of the evolution approach to UML models.

3.1.2.1.1 Duplicate Class

When extending a UML model, it might happen that a new class is required that has almost the same connections to other entities as an already existing class. In such a case, it would be easiest to copy the existing class and all its associations to create a basis for the new class. The evolution Duplicate Class aids users in this process. It allows to select a class from a UML diagram, which is then cloned. Along with the class, all the incoming and outgoing associations are copied as well. The newly created associations are modified to target the duplicated class where they originally used the selected class. In addition, a name for the duplicated class can be provided by the user. Through this procedure, creating similar class structures can be performed swiftly. In Figure 3.15, the Duplicate Class evolution is demonstrated in an example.

Figure 3.15: Example of a UML diagram before and after the Duplicate Class evolution.

(a) original UML diagram

(b) after Duplicate Class was applied to the class ClassB

3.1.2.1.2 Extract Super Class

The evolution Extract Super Class is used to move selected attributes up the inheritance hierarchy to a newly created class placing the elements on a more abstract level. It is an implementation of the equally named modification of [Fow99]. For the evolution to be triggered, at least one attribute has to be selected. Then, a new class with a provided name is created and a generalization from the original to the new class is added. Finally, the selected attributes are moved to the new class. The procedure can be used for equivalent attributes of multiple classes when their commonalities are to be extracted to a single base class. This procedure is
3.1. EVOLUTIONS

Figure 3.16: Example of a UML diagram before and after the *Extract Super Class* evolution.

![UML diagram example](image)

3.1.2.1.3 Replace Method with Method Object

The evolution *Replace Method with Method Object* is used when the functionality described by a method has to be relocated to a separate class. For example, a method `readCDText()` might have to be relocated from a class `CDPlayer` to its own class `CDTextReader` if the contained functionality is important enough to be defined in its own domain. To perform this process, a new class is created and the selected method is moved to it. To signal the reference from the original class to the newly created method object class, an association is added. It is possible to apply the evolution for multiple methods at once, which then are all moved to the new class. The evolution is an implementation of the equally named refactoring of [Fow99]. The procedure of extracting a method object using the aforementioned example is visualized in Figure 3.17.

Figure 3.17: Example of a UML diagram before and after the *Replace Method with Method Object* evolution.

![UML diagram example](image)

3.1.2.1.4 Inline Method Object

The evolution *Inline Method Object* is the inverse operation to the aforementioned *Replace Method with Method Object*. It incorporates a piece of functionality that is defined in a separate class into its referencing class. For example, a method `decodeMP3()` from a class `MP3Decoder` might be inlined in its referencing class `CDPlayer` if the described functionality does not justify defining a separate class. This situation might happen due to a restructuring that affects the method object and reduces its responsibilities. During the process of evolution, the methods in the method object class are relocated to the referencing class. Furthermore, the method object class and the reference to it are deleted from the diagram. The described evolution is a specialization of the modification *Inline Class* from [Fow99]. The results of performing the evolution in the previously mentioned scenario are shown in Figure 3.18.
3.1.2.1.5 Remove Class

The last evolution for UML is Remove Class. As the name suggests, the evolution deletes a selected class from the diagram. Furthermore, it also removes all incoming and outgoing associations of the class. This evolution is employed to remove no longer needed classes from the UML diagram while leaving the model in a valid state as all referenced entities are also deleted. The effects of the Remove Class evolution are shown in Figure 3.19.

Figure 3.19: Example of a UML diagram before and after the Remove Class evolution.

3.1.2.2 Evolutions for Java

The Java language is a widely used general purpose programming language. As such, it is employed in the development of many applications. Furthermore, it can be used to implement part of the realization artifacts of a software product line. As these artifacts are subject to change as all other solution space elements, it is essential to provide the option to evolve Java source code.

In its natural form, Java source code is represented by a textual syntax. Through the use of EMFText (see Section 4.1.2) it is possible to create a representation of the source code in the form of an Ecore-based model. This makes Java source code a viable option for a target of the evolution system presented in this thesis. A number of modifications tailored specifically to Java programs is introduced in the following paragraphs.

3.1.2.2.1 Insert Parameter

Program logic within a Java class is structured into methods that can be called with varying input parameters from multiple sites. Thus, the grouping of program code into methods is an essential part of reusing functionality. However, over the course of time, responsibilities of a particular method may change. In turn, it might be necessary to adapt the signature of the method to require additional input for the modified calculation procedure. The evolution Insert Parameter allows to add a new parameter to an existing method within a class. It offers the
possibility to assign a name to the newly created parameter and to choose its type from a list of possible options. The new parameter is then added to the list of input parameters as last element. The evolution is an adaptation of Add Parameter from [Fow99]. An excerpt from Java source code before and after applying the evolution is shown in Listing 3.1.

Listing 3.1: Example of Java source code before and after the Insert Parameter evolution.

a) original Java source code

```java
public void example(String param1, int param2) {
  //...
}
```

b) after Insert Parameter was applied to the method example()

```java
public void example(String param1, int param2, String param3) {
  //...
}
```

3.1.2.2 Rename Element

During the course of restructuring existing Java code, it may happen that the name of an element no longer seems appropriate. For example, naming conventions may have changed or a

Listing 3.2: Example of Java source code before and after the Rename Element evolution.

a) original Java source code

```java
public void method() {
  //...
  example("Test", 7);
}

public void example(String param1, int param2) {
  //...
}
```

b) after Rename Element was applied to the method example()

```java
public void method() {
  //...
  otherMethod("Test", 7);
}

public void otherMethod(String param1, int param2) {
  //...
}
```
more expressive phrase entered the mind of the developer. In this case, all existing occurrences of the element have to be supplied with the new name. The evolution Rename Element allows to change the name of existing attributes and methods in a Java source file. The declaration as well as all references are modified to use the altered name. For example, when a method is renamed, all calls to that method use the new name as well, which maintains consistency of the source code. Therefore, using the Rename Element evolution is a much swifter alternative to a manual renaming procedure. The described modification is a more general version of the Rename Method refactoring from [Fow99]. The effects of the evolution on a Java source code fragment are shown in an example in Listing 3.2.

3.1.2.2.3 Extract Method

Listing 3.3: Example of Java source code before and after the Extract Method evolution.

a) original Java source code

```java
public class Example {
    private double value1;
    private int value2;

    public void example() {
        value1 = 1.3;
        value2 = 7;
        double result = value1 * value2;
        System.out.println(result);
    }
}
```

b) after Extract Method was applied to the first two statements of the method example()

```java
public class Example {
    private double value1;
    private int value2;

    public void example() {
        initializeValues();
        double result = value1 * value2;
        System.out.println(result);
    }

    private void initializeValues() {
        value1 = 1.3;
        value2 = 7;
    }
}
```
3.1. EVOLUTIONS

As a negative side effect of adding new functionality to a class, it may happen that a method becomes convoluted by adding too many lines of code to it. A possible solution is to restructure the method’s contents into further methods performing part of the functionality of the original operation. The new methods are then called within the original method body so that the original calculation procedure is maintained. The evolution *Extract Method* offers an easy mechanism for performing this type of modification. First, a block of statements within a method has to be selected. Then, a name for the new method is chosen by the user. After that, a new method is created directly behind the original method and finally, the original lines of code are moved to the new method. With this evolution, it is easy to break up convoluted methods into smaller functional units. The evolution is an implementation of the equally named refactoring from [Fow99]. In Listing 3.3, an example of applying the *Extract Method* evolution is presented.

3.1.2.2.4 Remove Statement

Evolving source code may also mean removing obsolete elements. In particular, it is possible that only individual parts of a class or method are no longer needed while its containing element should be retained. In this case, removing an individual statement of the programming language is the appropriate modification. Usually, this type of change can be performed directly within a code editor. However, in the context of a software product line, simply deleting an element may render the mapping model invalid as will be shown in Section 3.4.2.2. Due to this fact, a procedure dedicated to the removal of a statement is required.

Listing 3.4: Example of Java source code before and after the *Remove Statement* evolution.

```
public double example(double value1, int value2) {
    double result = value1 * value2;
    System.out.println(result);
    return result;
}
```

b) after *Remove Statement* was applied to the second statement in the method `example()`

```
public double example(double value1, int value2) {
    double result = value1 * value2;
    return result;
}
```

The evolution *Remove Statement* allows to delete a logical line of source code from a Java file. Through this mechanism, it is possible to remove no longer needed functionality. However, removing statements that are referenced by other elements may cause problems. For example, deleting a variable declaration does not automatically remove all uses of the variable. Thus, it is the responsibility of the user to ensure that the rest of the source code remains valid after a statement was removed. In Listing 3.4, the modification of a Java source code fragment is displayed to demonstrate *Remove Statement*. 
3.1.2.3 Evolutions for DocBooklet

DocBooklet is a derivative of the DocBook format. It features an XML-like textual notation imitating the syntax of its archetype. DocBooklet serves as a representative of documentation formats and is used within the example project in Section 5 showing the practical usefulness of the evolution approach. A DocBooklet file is structured into chapters, sections and paragraphs. In Listing 3.5, an example of a DocBooklet file is presented. A full specification of the language and its syntax can be found in Appendix B. To show how the work of this thesis can be applied to documentation formats, evolutions for the DocBooklet format are defined in the following sections.

Listing 3.5: Example of a DocBooklet file.

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"Example Document"</title>
  <chapter>
    <title>"Example Chapter"</title>
    <para>"First paragraph."</para>
    <section>
      <title>"Example Section"</title>
      <para>"Second paragraph."</para>
      <para>"Third paragraph."</para>
    </section>
  </chapter>
</book>
```

3.1.2.3.1 Duplicate Chapter/Section/Paragraph

During the evolution of a document, it might happen that a largely similar section has to be written again. In order to aid this process, an existing section can be cloned. The **Duplicate Section** evolution copies all of the contents of a section including its paragraphs to a new section located directly behind the original. The duplicate can then be edited to create an altered version of the section. Equivalent evolutions also exist for chapters and paragraphs to clone these elements. In the case of duplicating a chapter, all contained sections and paragraphs are copied as well. For paragraphs, only the contained text is duplicated. The effects of the **Duplicate Section** evolution are demonstrated in Listing 3.6 by using an example.

3.1.2.3.2 Move Chapter/Section/Paragraph

When restructuring a document, one essential modification is to change the order of sections. In particular, a section may be moved before or after another section. These modifications are reflected by the **Move Up Section** and **Move Down Section** evolutions respectively. Likewise, chapters and paragraphs may be moved in a similar way. For this purpose, four additional evolutions exist. The evolutions **Move Up Chapter** and **Move Down Chapter** affect chapters
3.1. EVOLUTIONS

Listing 3.6: Example of a DocBooklet file before and after the Duplicate Section evolution.

a) original DocBooklet file

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"Example Document"</title>
  <chapter>
    <title>"Example Chapter"</title>
    <para>"First paragraph."</para>
    
    <section>
      <title>"Example Section"</title>
      <para>"Second paragraph."</para>
      <para>"Third paragraph."</para>
    </section>
  </chapter>
</book>
```

b) after Duplicate Section was applied to section “Example Section”

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"Example Document"</title>
  <chapter>
    <title>"Example Chapter"</title>
    <para>"First paragraph."</para>
    
    <section>
      <title>"Example Section"</title>
      <para>"Second paragraph."</para>
      <para>"Third paragraph."</para>
    </section>
    
    <section>
      <title>"Example Section"</title>
      <para>"Second paragraph."</para>
      <para>"Third paragraph."</para>
    </section>
  </chapter>
</book>
```
CHAPTER 3. EVOLUTION OF SOFTWARE PRODUCT LINES

Listing 3.7: Example of a DocBooklet file before and after the Move Up Section evolution.

a) original DocBooklet file

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"Example Document"</title>
  <chapter>
    <title>"Example Chapter"</title>
    <para>"First paragraph."</para>

    <section>
      <title>"Example Section A"</title>
      <para>"Second paragraph."</para>
      <para>"Third paragraph."</para>
    </section>

    <section>
      <title>"Example Section B"</title>
      <para>"Fourth paragraph."</para>
      <para>"Fifth paragraph."</para>
    </section>
  </chapter>
</book>
```

b) after Move Up Section was applied to section “Example Section B”

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"Example Document"</title>
  <chapter>
    <title>"Example Chapter"</title>
    <para>"First paragraph."</para>

    <section>
      <title>"Example Section B"</title>
      <para>"Fourth paragraph."</para>
      <para>"Fifth paragraph."</para>
    </section>

    <section>
      <title>"Example Section A"</title>
      <para>"Second paragraph."</para>
      <para>"Third paragraph."</para>
    </section>
  </chapter>
</book>
```
whereas Move Up Paragraph and Move Down Paragraph relocate a selected paragraph. A scenario where Move Up Section is applied serves as example of all the related evolutions. The effects of the evolution can be seen in Listing 3.7.

3.1.2.3.3 Extract Chapter/Section

When extending a document, it might happen that a particular series of paragraphs gains in relevance so that it ultimately should form its own section. To relocate the affected paragraphs, the evolution Extract Section may be used. It creates a new section and places all selected paragraphs in it. Thus, the increased relevance of the paragraphs is reflected in the document. A similar procedure can be executed on sections that should be placed in an individual chapter. For this case, the evolution Extract Chapter was defined. However, extracting a paragraph is not possible as paragraphs are the smallest individual element of the DocBooklet format and thus, there is no source to extract them from. The Extract Section evolution is demonstrated in Listing 3.8 using an example.

3.1.2.3.4 Remove Chapter/Section/Paragraph

Over the course of time, it might happen that a particular part of a document becomes outdated and thus has to be deleted. Usually, this procedure can be performed directly in a text editor. However, if the removed part of the document participated in a mapping of a product line, the consistency of the mapping model would be destroyed as will become apparent in Section 3.4.2.3. To allow the deletion of outdated elements from a DocBooklet file within the context of a software product line, specific evolutions are provided.

The Remove Section evolution is provided for DocBooklet files to delete content that was rendered obsolete. The evolution detaches a section from its containing chapter leaving the remainder of the file intact. Equivalent evolutions are provided for chapters and paragraphs. In Listing 3.9, the behavior of the Remove Section evolution is demonstrated in an example.

3.2 Classification Systems for Evolutions

Building a classification of evolutions means forming groups of modifications with common characteristics. In order to attribute evolutions to particular groups, the boundaries of the different classes of evolutions have to specified explicitly. In the context of this thesis, the interesting characteristic of evolutions is how they affect the software product line as a whole. Furthermore, it is of interest when and how the feature mapping has to be adapted in order to compensate for changes in a particular model. Naturally, the classification has to be exhaustive in the sense that all evolutions presented in the thesis can be attributed to a specific group. Through this, the classification will provide a logically structured view on evolutions for product lines and will help in addressing a particular type of evolution in writing and conversation.

The following three sections deal with classifying evolutions of software product lines. In the first section it will be explained why a grouping by whether evolutions preserve semantics or not is not suitable for the problem at hand. After that, the second section will explain the most prominent classification systems in the literature and discuss their individual shortcomings with respect to the focus of this thesis. Finally, the third section will introduce a new classification
Listing 3.8: Example of a DocBooklet file before and after the Extract Section evolution.

a) original DocBooklet file

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"Example Document"</title>
  <chapter>
    <title>"Example Chapter"</title>
    <para>"First paragraph."</para>
    <section>
      <title>"Example Section"</title>
      <para>"Second paragraph."</para>
      <para>"Third paragraph."</para>
    </section>
  </chapter>
</book>
```

b) after Extract Section was applied to paragraph “Second Paragraph”

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"Example Document"</title>
  <chapter>
    <title>"Example Chapter"</title>
    <para>"First paragraph."</para>
    <section>
      <title>"Extracted Section"</title>
      <para>"Second paragraph."</para>
    </section>
  </chapter>
</book>
```
Listing 3.9: Example of a DocBooklet file before and after the Remove Section evolution.

a) original DocBooklet file

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"Example Document"</title>
  <chapter>
    <title>"Example Chapter"</title>
    <para>"First paragraph."</para>
    <section>
      <title>"Example Section A"</title>
      <para>"Second paragraph."</para>
      <para>"Third paragraph."</para>
    </section>
    <section>
      <title>"Example Section B"</title>
      <para>"Fourth paragraph."</para>
      <para>"Fifth paragraph."</para>
    </section>
  </chapter>
</book>
```

b) after Remove Section was applied to section “Example Section A”

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"Example Document"</title>
  <chapter>
    <title>"Example Chapter"</title>
    <para>"First paragraph."</para>
    <section>
      <title>"Example Section B"</title>
      <para>"Fourth paragraph."</para>
      <para>"Fifth paragraph."</para>
    </section>
  </chapter>
</book>
```
3.2.1 Classification by Behavior Preservation

When changes to a software system have to be classified, the usual characteristic for grouping is whether the changes preserve semantics/behavior or not [TBK09, AGM+06, RSA10, Rei10, Fow99]. In particular, this means that changes are divided into the two groups of refactorings as semantic preserving modifications and arbitrary changes. However, when looking at the effect of evolutions on a software product line as a whole, this classification seems unsatisfactory for two reasons.

The first problem is that defining the term “semantic preserving” is not possible for the general case due to the diverse nature of the models used in a product line [MT07]. For the feature model in the problem space, according to Borba, “behavior” is defined as the set of all configurations that can be derived from that model [Bor11]. The author defines a refactoring as being a change that does not constrain the configurability of the feature model. In other words, after the modification it has to be possible to derive at least all the configurations as before. Furthermore, it is possible that a refactoring creates additional configuration options. Should a change reduce the number of configurations without adding new options, it is considered a specialization of the feature model. Thüm, Batory & Kästner build on this definition and refine it [TBK09]. In their work, the group of refactorings is split up further. Modifications that extend the configuration options and still contain all the original configurations are called generalizations. The term “refactoring” is hence restrained to changes that produce exactly the same configuration possibilities as in the original feature model.

Attributing practically relevant evolutions on feature models to those categories is problematic due to the complex nature of the modifications. The functionality of certain evolutions largely depends on the context they are used in, which also affects the change in configurability of the feature model. For example, if the evolution Duplicate Feature is used on an optional feature, it extends the configuration options of the feature model [AGM+06, Bor11], which constitutes a generalization. However, if that very same evolution is used on a mandatory feature, the configuration options are restrained while adding new ones at the same time [Bor11]. This would make the evolution an arbitrary modification in this context. Furthermore, the classification seems to be unable to adequately capture all evolutions in the appropriate groups. The authors of [TBK09] provide criteria for each group but admit that the evolution Split Feature would falsely be classified as arbitrary change because it both adds and removes configuration options. However, the evolution really is a refactoring as the sum of the split parts are considered to make up for the original feature. Therefore, a classification by preserving semantics seems both inadequate and problematic in the problem space.

Evolutions in the solution space face similar problems when trying to group them by whether they preserve semantics. As in the problem space, the definition of “semantics” is precarious. However, in this case, it is the multitude of different models that can be processed which causes problems. As FeatureMapper and its evolution mechanism are generic in the sense that they can be used on arbitrary Ecore-based solution space models, there is very little a priori knowledge about the respective metamodels. As a consequence, a precise assessment whether a specific
evolution changes behavior or not can not be made without knowing the concrete type of model it is applied to. For example, the principle steps of creating a new container and then moving existing elements to it could be used for the evolution Extract Super Class in UML diagrams as well as for the evolution Extract Chapter in a documentation format such as DocBooklet. For the UML metamodel, this change can be considered semantic preserving as the class the modification was applied to inherits all properties of the newly created super class. With the documentation format, the refactoring would take a selected block of text and extract it to a new chapter. Defining semantics for a document is most likely done in the way users perceive the information. When the structure of the information was changed, it seems reasonable to assume that the impression of the reader and hence the semantics of the model have changed as well. Therefore, equivalent modifications may be perceived as changing or preserving semantics depending on the context they are used in. Thus, it is not possible to give an adequate definition of semantics for solution space models for the general case.

The second problem with a classification by whether models preserve semantics is that it only considers a single model at a time. One characteristic of the evolutions in this thesis is that in many cases they extend beyond the bounds of a single model and affect the mapping model as well. As a consequence, evolutions might have an effect on the software product line as a whole. This effect is not captured when classifying evolutions by whether they preserve semantics or not. Therefore, it appears that this classification treats an inadequate characteristic of the evolutions to describe their effect on a product line.

3.2.2 Classification Systems in the Literature

In the literature, there are a number of further classification systems for modifications to models that each have one primary characteristic of grouping. Porres in [PCS03] uses the relation of source and target model of a transformation for classification. For this, the terms “mapping transformation” and “update transformation” are introduced. A mapping transformation relates the elements of a source model to the elements of a different target model. As such, it can be perceived as a form of translation that can, for example, occur in model to model or model to text transformations. An update transformation, on the other hand, modifies a given model in place. As all modifications presented in this thesis work directly on the provided models and do not translate them into a different representation, all evolutions are considered update transformations. Consequently, this classification does not yield any relevant insights.

Mens et al. present two further classification systems in [MCG05]. The first one differentiates “endogenous” and “exogenous” transformations. Endogenous transformations can be perceived as a form of rephrasing where elements of one language are changed to other elements of the very same language. In contrast, exogenous transformations are a form of translation where elements of a source language are related to elements of a target language. An example of an exogenous transformation is code generation from UML models where a modeling language is translated into an implementation language. Endogenous transformations can be found in operations such as code optimization or simplification. Much like the categorization of Porres presented before, this classification uses characteristics of the source and target models as key for grouping. As a consequence, it also has the same problems as the aforementioned approach. Additionally, all evolutions are classified as endogenous transformations so that the system is not feasible for the
classification of evolutions.

The second type of classification by Mens et al. uses the level of abstraction as primary key and divides modifications into “horizontal” and “vertical” transformations. In a horizontal transformation, both the source and target model are located on the same level of abstraction. In a vertical transformation, the source and target model reside on different levels so that either the source or the target model is more detailed than its counterpart. For example, this type of classification can be useful in the model driven architecture\(^1\), where the transformation of platform independent to platform specific model would constitute a vertical transformation. However, this type of classification seems less suitable for grouping evolutions in a software product line as they do not affect the level of abstraction of a model and would thus have to be classified as horizontal transformations in their entirety.

Alves et al. in [AGM+06] use the reversibility properties of changes to feature models as means for classification. The authors distinguish unidirectional and bidirectional modifications. Unidirectional modifications can only be performed in one direction because they extend the configuration options that can be derived from the feature model and thus are not reversible. Bidirectional modifications maintain the configuration options and thus can be reversed. For example, a mandatory feature \texttt{FeatureA} being a child of another feature \texttt{FeatureB} could be transformed to an optional feature without a change in configurability if a constraint \texttt{FeatureB} $\rightarrow$ \texttt{FeatureA} was added at the same time. This kind of transformation can then be reversed to create the original form of the feature model. As the configurability of a software product line is perceived as the semantics of the feature model, this classification is merely a coarse grain version of grouping modifications by whether they preserve semantics. Therefore, this approach suffers from equivalent problems as described in the previous section.

Finally, Borba suggests in [Bor11] that evolutions in a software product line should be grouped by the space they originate from. Hence, there are evolutions for the problem space and the solution space. This seems to be a reasonable criterion for grouping as it helps to address particular groups of evolutions in conversation. However, the classification still does not capture the essence of the problem because the effect of evolutions on the product line is not inspected.

Considering the systems of classification presented above, each evolution would have to be classified as a horizontal, exogenous update transformation without giving any consideration to its effects on the product line. Furthermore, the classification of Alves et al. would only describe the semantic preserving qualities of an evolution with regard to the feature model. Thus, the only viable classification appears to be by the space an evolution originates from. Obviously, this is not sufficient to capture the effect of evolutions on the product line as a whole. As remedy, a more suitable classification is presented in the next section.

### 3.2.3 Classification by Semantical Extent of Model Changes

In order to determine the effect of an evolution on the product line as a whole, it seems necessary to inspect the changes to all types of models involved in a product line. For this purpose, a new classification for evolutions on software product lines is introduced.

Before explaining the individual categories of this classification, a closer look at the constituent parts of a software product line is required. In the problem space, there is only the \(^1\)http://www.omg.org/mda/
3.2. CLASSIFICATION SYSTEMS FOR EVOLUTIONS

feature model. In the solution space there are various models of different types. However, for the sake of this classification, these can all be treated uniformly as “solution space models”. A special role comes to the mapping, which connects the feature model and the respective solution space models. Even though it is a model in its own domain in the implementation of FeatureMapper [HKW08], it also connects the problem and solution space logically.

For the classification presented in this section, the primary key of classification is the semantical extent of the changes made to models with regard to their respective spaces (i.e., problem or solution space). For this purpose, two major groups of evolutions are differentiated: “intraspatial” and “interspatial” evolutions. In order to make it easier to grasp the individual characteristics of the presented categories, concrete evolutions are named as examples of particular groups. The general functionality of these evolutions is briefly recapitulated in this section. A complete categorization of the evolutions presented in Section 3.1 will be performed in Section 3.4.

3.2.3.1 Intraspacial Evolutions

The defining characteristic of intraspacial evolutions is that the extent of their changes is contained within their originating space. This means that only either problem or solution space are affected by the changes performed during an evolution. These characteristics are summarized in Definition 3.1. A visualization of the different types of intraspacial evolutions that may occur in a software product line is presented in Figure 3.20.

**Definition 3.1 (Intraspacial Evolutions)** An intraspacial evolution is a modification that exclusively performs changes whose effects are contained within the originating space so that only problem or solution space are affected.

![Figure 3.20: Visualization of intraspacial evolution types.](image)

In Figure 3.20(a), an intraspacial evolution is shown that is confined to the problem space so that only the feature model is affected. An example of one such evolution would be *Insert Feature* where a completely new feature is added to the feature model. As newly added elements can not immediately have a mapping assigned to them, the boundary to the mapping is not crossed. Another example would be *Pull Up Feature*, which lifts a feature from its current group to the group located one level above in the feature tree. As merely the structure of the problem space is modified, the mapping is not touched. A special group of evolutions in the problem space are those that exclusively operate on groups. As groups cannot have a mapping to solution space elements in FeatureMapper, this type of evolution is always intraspacial by definition.

In Figure 3.20(b), an intraspacial evolution for the solution space is displayed. Again, all changes are contained within their originating space. Due to the diverse nature of the solution
space models, a generic example of an intraspatial solution space evolution can not be provided for all types of solution space models. Instead, an example is given for the case of evolving Java source code. In this case, Rename Element is a representative of intraspatial evolutions as the mapping does not depend on the name of an object. Thus, the border to the mapping is not crossed.

Assuming that all the elements affected by an evolution are part of a mapping, the following criteria can help to identify an intraspatial evolution. As long as existing elements are merely modified, the boundary to the mapping will not be crossed and evolutions are guaranteed to be intraspatial. When new elements are added during an evolution, determining whether the original space is left or not depends on the concrete case. If the new elements have a logical connection to already existing ones, it is likely that their mapping will be reused in some sort, which makes the evolution reach into the mapping. For example, a logical connection exists in Duplicate Feature but not in Insert Feature making only the latter an intraspatial evolution.

Thus, a mere structural analysis of changes does not suffice for an adequate classification. Instead, the intent of the changes has to be considered. However, when an object is removed, the evolution always leaves its original space provided that the deleted element participated in a mapping. Due to this fact, evolutions that remove an element are generally considered interspatial unless it is ensured that the removed element may never be part of a mapping (e.g., groups in the feature tree of the problem space).

3.2.3.2 Interspatial Evolutions

The group of interspatial evolutions contains all those evolutions that have an effect on the mapping in a product line. This means that the performed changes have to reach beyond their original space. The category of interspatial evolutions is further divided into two sub groups called interspatial evolutions of the first or second degree respectively. Both groups will be explained in the following paragraphs.

As stated in Definition 3.2, interspatial evolutions of the first degree affect exactly one of the two spaces of a product line and the mapping in between the spaces. Thus, there are two different scenarios in which this situation may occur as shown in Figure 3.21.

**Definition 3.2 (Interspatial Evolutions of the First Degree)** An interspatial evolution of the first degree is a modification that performs changes whose effects reach beyond the originating space to the feature mapping. Interspatial evolutions of the first degree affect either problem or solution space as well as the feature mapping so that remapping is required in all cases.

Figure 3.21: Visualization of interspatial evolution types of the first degree.
3.2. CLASSIFICATION SYSTEMS FOR EVOLUTIONS

In Figure 3.21(a), an interspatial evolution reaching from the problem space to the mapping is visualized. This case may appear in evolutions that are triggered from the feature model and make modifications that reference another feature that has a mapping. One such case is the evolution *Duplicate Feature* that clones an existing feature and adds the copy to the group of the original feature. If the original feature was mapped to solution space elements, the border to the mapping is crossed as the cloned feature should share the same mappings.

Similarly to the evolutions crossing the problem space boundary, evolutions may also cross the border between the mapping and the solution space. This scenario appears when an interspatial evolution of the first degree originates in the solution space as can be seen in Figure 3.21(b). For example, an element in a solution space model may be added that has a reference to another element with a mapping. This can happen in evolutions such as *Extract Super Class* in UML diagrams where attributes of an original class with a mapping are extracted to a newly created super class. As the super class is logically connected to the original class, it should share the same mapping. Thus, the border to the mapping is crossed.

As stated in Definition 3.3, interspatial evolutions of the second degree affect both the problem and solution space as well as the mapping, which connects them. With this definition, there are two types of interspatial evolutions of the second degree, which are depicted in Figure 3.22.

**Definition 3.3 (Interspatial Evolutions of the Second Degree)** An interspatial evolution of the second degree is a modification that performs changes whose effects reach beyond the originating space to the opposite space including the feature mapping. Interspatial evolutions of the second degree affect the problem and solution space as well as the feature mapping so that remapping is required in all cases.

Figure 3.22: Visualization of interspatial evolution types of the second degree.

In Figure 3.22(a), an evolution that emerges in the problem space, spans the mapping and reaches to the solution space is visualized. Due to the complex nature of this type of evolution, practical applications are rare. However, an example is the evolution *Remove Feature and Owned Assets*, which deletes a selected feature from the problem space and also removes all exclusively mapped model elements from the solution space. Naturally, the mapping is affected as elements at both ends of the mapping cease to exist.

An alternate type of interspatial evolution of the second degree is depicted in Figure 3.22(b), where an evolution originates in the solution space and reaches over the mapping to the problem space. Again, concrete evolutions are rare for this scenario. Yet, there is the example of *Remove Asset and Owning Features*, where a solution space element is deleted with all features that are mapped solely to this particular element. Obviously, this type of evolution affects the mapping as well.
CHAPTER 3. EVOLUTION OF SOFTWARE PRODUCT LINES

Under the assumption that all affected elements are part of a mapping, the following criteria can help to identify interspatial evolutions. Whenever new elements are added that have a logical connection to already existing elements, the boundary to the mapping is crossed. For example, this happens in Duplicate Feature where the newly added feature is a clone of an already existing feature making the evolution interspatial in nature. Removing elements from either space always constitutes an interspatial evolution as the mapping is necessarily affected.

Interspatial evolutions of both the first and second degree always have an effect on the mapping. If the mapping is left unchanged after the evolution, it might result in an invalid state of the software product line as for example no longer existing elements are part of a mapping. To counter this effect, interspatial evolutions require a subsequent remapping in order to keep the product line as a whole consistent.

Prior approaches to evolving software product lines seem to have neglected the need for remapping only looking at the effects on the space the evolution originated from. Due to this fact, evolutions that actually are interspatial in nature have wrongfully been treated as intraspatial evolutions. In consequence, no remapping has been performed effectively rendering the product line invalid after performing an evolution. This thesis presents remedy for the problem by remapping critical elements affected by an interspatial evolution.

3.2.3.3 Criteria for Classification by Semantical Extent of Model Changes

Section 3.2.3.1 and Section 3.2.3.2 have each presented criteria for attributing a particular evolution to the group of intraspatial or interspatial evolutions respectively. As the semantical extent of the changes is relevant for the classification, a mere inspection of the structural changes performed by an evolution does not suffice for an appropriate classification. Thus, an automated approach for identifying the group an evolution belongs to is not feasible. Therefore, evolutions have to be attributed to their groups manually. For this purpose, a decision diagram is presented in Figure 3.23 to steer the decision making process.

The diagram contains the criteria for attributing evolutions to their respective groups in the classification presented in the previous sections. On an abstract level, it suffices to distinguish changes into the three groups of adding, modifying or removing elements [BPD+10]. In order to identify structural changes of an evolution, this distinction is suitable as well. Therefore, it will be inspected which category the modifications of an evolution belong to before their semantical extent is analyzed.

If at least one element is deleted from a model, the evolution is necessarily interspatial as the original element might have had a mapping. For all interspatial evolutions, it further has to be determined if they modified models of both problem and solution space or merely one of the two. If models of both spaces were altered, the evolution is interspatial of the second degree, otherwise it is interspatial of the first degree.

If no elements were deleted from the model but new elements were added, it depends on whether the created elements have a logical relation to previously existing elements to decide whether the evolution is intraspatial or interspatial. The answer to this question cannot be found automatically but has to be decided by someone with an understanding of the intent of the evolution. If there is no connection to previously existing elements, the evolution is intraspatial. If the connection can be established, the evolution is interspatial and it requires a subsequent
remapping.

If no elements were added to the model, it remains to be determined if existing elements were modified. If this is the case, the evolution is intraspatial as no other modifications were performed. In the negative case, the result is that no evolution was performed because the target model was not modified at all.

The decision diagram can be refined with further knowledge of the model. For example, if an element is deleted and it is known that this particular type of element may never be part of a mapping, the respective evolution is not necessarily interspatial. This would be the case with groups in the feature tree of the problem space. However, for the general case of arbitrary solution space models, the decision diagram allows adequate classification of evolutions by semantical extent of model changes.

3.3 Remapping Operations

The sections leading up to this chapter have explained that there are evolutions that have effects extending beyond the bounds of their originating space. In consequence, this group of interspatial evolutions might break or damage the mapping and thus render the product line invalid. As remedy, it is necessary to adapt the mapping to counter these effects. In the course of the thesis, this procedure is referred to as remapping. As it is remapping that is performed, all operations modify or at least reference an existing mapping. In particular, creating entirely new mappings is out of the scope of the remapping operators.

The steps carried out during remapping are considered to be rather a complementary operation to evolutions than a part of them. As such, remapping is performed after the evolution has completed its modifications. In the following sections, a number of remapping operators
are presented that can be composed to a remapping plan in order to describe more complex modifications of the mapping model. The remapping in its entirety is perceived as an optional operation. Users may decide whether it is appropriate for the given scenario to perform remapping or not. For example, when using Duplicate Class for UML models (see Section 3.1.2.1.1), it may or may not be useful to copy the mapping of the original class. In either way, the validity of the mapping model will be maintained in this concrete case. However, in general, there is no guarantee that the product line will be valid if remapping is canceled. Deselecting certain remapping steps individually is not possible because it bears the risk of creating a remapping plan that results in an inconsistent mapping. For example, if not all mappings of a deleted solution space element are either moved or removed during remapping, the feature mapping is damaged.

Yet, creators of an evolution can grant some freedom of choice for remapping steps provided that all possible decisions are safe with regard to product line consistency. As a remapping operator may potentially affect multiple mappings at the same time, it is possible to allow users to deselect individual mappings from remapping. For example, a feature might be mapped to three solution space elements and a user decides to only copy the mapping of two of those artifacts during the course of copying a mapping. Furthermore, for remappings that potentially have multiple targets, it is possible to allow individual targets to be excluded from remapping. For example, users might then choose to copy a mapping only to a subset of all potential targets.

Naturally, this kind of freedom of choice in the remapping process bears the risk of creating an inconsistent mapping. It is thus the responsibility of the creators of an evolution to ensure that this liberty is granted only if all possible choices do not damage the consistency of the feature mapping. Furthermore, it has to be ensured that an element is never the source and target of a remapping step at the same time as the behavior of the operator is undefined in this case.

Some details of how a remapping operation is performed are largely influenced by design decisions made by FeatureMapper. Most notably, there is a distinction of three types of mappings: element mappings, property value mappings and color mappings. The characteristics of these types of mappings have already been introduced in Section 2.1.3.3 and thus will not be explained at this point. However, it is important to recapitulate that FeatureMapper allows realization assets to appear at most once as the solution space end of one type of mapping. Hence, no two mappings of identical type to the same solution space element may exist at any time. This entails that remapping in the problem space usually means changing the feature expression of a mapping. The specifics will become apparent in the following section where problem space remapping is explained in detail. On the other hand, FeatureMapper allows equivalent feature expressions to be the problem space end of multiple mappings. This simplifies remapping in the solution space as no special measures have to be taken to avoid duplicate values for the end in the problem space. Furthermore, color mappings do not possess a solution space end as they map a feature expression to a color. Hence, color mappings can only be remapped from the problem space but not from the solution space. Due to these distinct characteristics, different operators for remapping are provided for the problem and solution space, which will be explained in the following two sections.
3.3. REMAPPING OPERATIONS

3.3.1 Remapping in the Problem Space

Remapping operations in the problem space are usually performed by changing the feature expression of the affected mappings. The only exception appears when removing a mapping. In part, the need to modify the feature expression is due to a design decision of FeatureMapper that prohibits a solution space element to appear in more than one mapping of the same type. This imposes constraints on how particular remapping operations can be realized. For example, it is not possible to create a physical copy of a mapping and then modify the feature expression of the cloned mapping as this would create a second mapping to the same solution space element. Instead, a logical copy has to be created. When inspecting the physical copies during product derivation, either one or even both feature expressions of the mappings would have to evaluate to true for the targeted solution space element to be included in the resulting variant. Therefore, this constellation is equivalent to an OR connection of both feature expressions. Hence, instead of creating a physical copy with two different feature expressions to the same solution space element, the original mapping is maintained and its feature expression is substituted with an OR connection of the old and new terms.

In FeatureMapper, a mapping is created by first composing a logical expression of features and then assigning it to solution space elements or a particular color. Naturally, there are a multitude of semantically equivalent expressions that could be used for a mapping. Yet, the concrete phrasing of the chosen feature expression is likely to bear some significance to its creator. Thus, modifications of logical terms should maintain readability and structure as much as possible so that users are aided in recognizing the meaning of a feature expression even after it was altered. Consequently, using different representations for the terms such as the conjunctive normal form\textsuperscript{2} is not an option even though it would ease the process of logical transformation.

Due to this reason, it is not trivial to simplify logical expressions after they were transformed. In consequence this means that some operators might produce logical expressions that contain redundancies and could be represented as a more compact term. One such example can be seen in Table 3.4 where Remove Feature Mapping is demonstrated. Furthermore, it is possible that the resulting feature expression of a remapping operator is either a tautology, which is always true, or a contradiction, which can not be satisfied. An example of a tautology could be the expression FeatureA OR NOT FeatureA, which is satisfied in all possible scenarios. This kind of expression could be the result of the Move, Copy or Split Feature Mapping operators as they all use a logical OR to connect partial expressions. On the opposite side, a contradiction could have the form FeatureA AND NOT FeatureA, which can never be true. This particular example can be created by Move Feature Mapping when using FeatureB as source and the expression NOT FeatureA as target on the input of FeatureA AND FeatureB. Unfortunately, this kind of behavior is part of the regular functionality of the remapping operators and thus can not be avoided entirely. A solution that might alleviate the problem would be to check all modified feature expressions for satisfiability and present the results to the user. However, the implementation in this thesis does not currently use such a mechanisms. Therefore, users are encouraged to verify the feature expressions resulting from a remapping operation in the problem space.

\textsuperscript{2}http://mathworld.wolfram.com/ConjunctiveNormalForm.html
3.3.1.1 Move Feature Mapping

The **Move Feature Mapping** operator is intended to relocate a mapping from a source feature to one or potentially many target features. Conceptually, moving a feature mapping is equivalent to removing the original mapping and replacing it with equivalent rules for all of the targets. From a logical point of view, this means that for $n$ target features the remapping operator produces $n$ individual rules. However, Section 2.1.3.3 explained that FeatureMapper imposes a constraint stating that each solution space element may be part of at most one instance of a type of mapping. Hence, instead of $n$ separate rules, one complex rule has to be formulated. If satisfied, the feature expression of each individual rule would suffice for the solution space element to be included in a particular variant. Naturally, this same behavior has to be exposed by the complex term. Thus, the complex feature expression is formulated by concatenating the individual rules by a logical OR operator. Therefore, the physical procedure of moving a feature mapping produces merely a single rule instead of $n$ logical rules. An example of moving a feature mapping is shown in Table 3.1.

<table>
<thead>
<tr>
<th>Table 3.1: Example of Move Feature Mapping.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) original mapping</td>
</tr>
<tr>
<td>FeatureA AND FeatureD</td>
</tr>
<tr>
<td>b) logical result that can not be implemented</td>
</tr>
<tr>
<td>FeatureB AND FeatureD</td>
</tr>
<tr>
<td>FeatureC AND FeatureD</td>
</tr>
<tr>
<td>c) after Move Feature Mapping (Move: FeatureA → FeatureB, FeatureC)</td>
</tr>
<tr>
<td>(FeatureB AND FeatureD) OR (FeatureC AND FeatureD)</td>
</tr>
</tbody>
</table>

As described in the introductory paragraph to remapping, it is possible for remapping operators to permit user interaction in order to configure the behavior of the operator. In the case of **Move Feature Mapping**, it is possible to choose only a subset of all potential targets as effective targets provided that the creators of the evolution and its remapping have enabled it. This means, that the source mapping is only moved to some of the possible target features instead of all. However, at least one target has to be selected so that the move operation can be performed.

Furthermore, it is possible to allow that only individual mappings are moved by the operator while others remain at the source object, which gives users the liberty of selecting only the mappings they want to move. For example, a user might choose to only move three of the total of five mappings from a particular source feature to its targets. However, this bears the risk of leaving mappings at the source feature that should have been moved in order for the product line to remain consistent. For example, if a solution space element was deleted, no mappings to it may remain in the system and all mappings must be moved. However, if the original element is still present, it might be a viable option to not move all of its mappings. It is the responsibility of the creators of the evolution to foresee these situations and allow individual deselection of mappings only if it is safe for product line consistency in all possible scenarios.
3.3. REMAPPING OPERATIONS

3.3.1.2 Copy Feature Mapping

The *Copy Feature Mapping* operator duplicates an existing mapping from one source feature to one or multiple target features. In its implementation, *Copy Feature Mapping* is largely equivalent to the *Move Feature Mapping* operator. It faces the same constraint of FeatureMapper that prohibits multiple rules for the same solution space element and is thus forced to formulate a single complex feature expression of multiple separate terms instead. In this procedure, the logical term of the original feature expression is duplicated $n$ times replacing the reference to the source feature with a reference to each target feature. The constituent parts are then connected with the original feature expression via the OR operator. In the case of *Copy Feature Mapping*, executing the operator on $n$ target elements would thus result in $n + 1$ individual logical rules as the original mapping is maintained. Despite this difference to *Move Feature Mapping*, the rest of the procedure is similar as all partial expressions are connected by an OR operator. The process of copying a feature mapping is demonstrated in Table 3.2 through an example.

<table>
<thead>
<tr>
<th>Table 3.2: Example of <em>Copy Feature Mapping</em>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>a) original mapping</td>
</tr>
<tr>
<td>FeatureA AND FeatureD</td>
</tr>
<tr>
<td>b) logical result that can not be implemented</td>
</tr>
<tr>
<td>FeatureA AND FeatureD</td>
</tr>
<tr>
<td>FeatureB AND FeatureD</td>
</tr>
<tr>
<td>FeatureC AND FeatureD</td>
</tr>
<tr>
<td>c) after <em>Copy Feature Mapping</em> (Copy: FeatureA → FeatureB, FeatureC)</td>
</tr>
<tr>
<td>(FeatureA AND FeatureD) OR (FeatureB AND FeatureD) OR (FeatureC AND FeatureD)</td>
</tr>
</tbody>
</table>

Creators of an evolution and the subsequent remapping may permit configuration of the *Copy Feature Mapping* operator. First, it is possible to allow user selection of only a subset of all potential targets. In this case, the mapping would only be copied to the effective targets instead of all targets. Second, users can be allowed to exclude individual mappings from remapping. For the excluded mappings, the remapping operator will not be performed leaving their feature expression unchanged. In contrast to *Move Feature Mapping*, this procedure is relatively unproblematic for *Copy Feature Mapping*. The *Move Feature Mapping* operator bears the risk that the source feature no longer exists and that not remapping an entry would damage product line consistency. As *Copy Feature Mapping* maintains the original mapping, it is assumed that the source feature remains in the system even after the evolution and its remapping were performed. As a consequence, it is impossible that neglecting to copy a mapping will break product line consistency. However, the semantic impact of not copying individual mappings depends on the concrete scenario and the intent of the evolution that requires remapping. Hence, the decision of whether to allow exclusion of individual mappings is left to the designer of an evolution.
3.3.1.3 Remove Feature Mapping

The *Remove Feature Mapping* operator logically deletes an existing mapping of a given feature. The process distinguishes two different cases. The first one are mappings that have only the affected feature as expression. The second one are mappings that have more complex feature expressions. In the first case, the mapping is physically deleted from the mapping model. In the second case, the feature expression has to be reformulated by removing all references to the affected feature. If the feature is negated, the NOT operator is removed along with it. Should the feature be part of a statement with multiple operands (i.e., OR/AND) that would only have one operand after the removal, the complex statement is replaced by a reference to the remaining feature. Examples of the presented cases of removing a feature mapping are shown in Table 3.3.

Table 3.3: Example of Remove Feature Mapping.

<table>
<thead>
<tr>
<th>FeatureA</th>
<th>UML:Class[Element]</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureA OR FeatureB OR FeatureC</td>
<td>UML:Class[Element]</td>
</tr>
<tr>
<td>NOT FeatureA AND FeatureB</td>
<td>UML:Class[Element]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>- (mapping removed)</th>
<th>- (mapping removed)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureB OR FeatureC</td>
<td>UML:Class[Element]</td>
</tr>
<tr>
<td>FeatureB</td>
<td>UML:Class[Element]</td>
</tr>
</tbody>
</table>

It is worth noticing that removing a feature mapping is not a direct inverse operation to copying a feature mapping. This is due to the functionality of both operators. For the sake of clarity, an example is provided in Table 3.4. The example shows that the result of first duplicating and then removing a feature mapping is neither syntactically nor semantically equivalent to the original mapping. For the *Remove Feature Mapping* operator it is impossible to determine, which parts of the term were originally copied from another feature expression. Therefore, manual modification of the feature mapping is required if more than the mere feature is to be removed from the feature expression.

Table 3.4: Example of copying and then removing a feature mapping.

<table>
<thead>
<tr>
<th>FeatureA AND FeatureB</th>
<th>UML:Class[Element]</th>
</tr>
</thead>
</table>

| (FeatureA AND FeatureB) OR (FeatureC AND FeatureB) | UML:Class[Element] |

| (FeatureA AND FeatureB) OR FeatureC | UML:Class[Element] |

Analog to the other remapping operators, *Remove Feature Mapping* gives the creators of an evolution the choice to allow individual exclusion of particular mappings. This means that some
of the affected mappings may in fact not be removed during the process of remapping by choice of the user of an evolution. In the case of Remove Feature Mapping, this kind of freedom seems particularly dangerous as it is easy to damage the mapping of the product line. For example, a feature might have been removed from the feature tree and therefore all its mappings have to be moved or removed as well. If individual mappings are deselected from processing, this might create a mapping with a no longer existing feature in the term, which would render the mapping model invalid. However, there are cases when individual deselection of mappings for the Remove Feature Mapping operator is in fact useful. For example, an evolution that merely modifies the mapping might allow some of the existing mappings of the source feature to remain in the mapping model without any harm to the product line.

### 3.3.1.4 Split Feature Mapping

The Split Feature Mapping operator is employed to distribute a mapping from one source feature to multiple targets. It is intended to be used in conjunction with the Split Feature evolution, which breaks up a feature into multiple parts (see Section 3.1.1.4). On first glance, Split Feature Mapping might seem like the Move Feature Mapping operator with variable targets. However, Move Feature Mapping exclusively uses the logical OR operator to concatenate individual terms, whereas Split Feature Mapping gives a choice between using the OR and the AND operator. The desired outcome of the operator is largely determined by the semantics of the solution space model and thus can not be automated. Therefore, users are given the opportunity to customize the behavior of Split Feature Mapping for individual mappings. It is worth noting that Split Feature Mapping can be used with a single target feature. In this case, the behavior of the remapping operator is in fact similar to that of Move Feature Mapping.

In Figure 3.24, a UML diagram is displayed, which will be used to demonstrate all possible application scenarios of Split Feature Mapping. In this example, an evolution broke up a feature FeatureA into its constituent parts FeatureAPart1 and FeatureAPart2. The original feature was mapped to all UML elements of Figure 3.24 and now has to be reassigned via the Split Feature Mapping operator. The first partial feature of the split is intended to use ClassA and all required elements while the second part should use ClassB.

In this example, the case of replacing a reference to the source feature with an AND expression of all target features is only true for the association from ClassA to ClassB as this element should only be part of a variant if both feature parts are selected at the same time.

Figure 3.24: UML diagram as example of a solution space model for Split Feature Mapping.

In the most basic case, users can select one of all potential targets as effective target. In this case, the reference to the source feature in the feature expression of the mapping is replaced with a reference to the selected target feature. For example, in Figure 3.24, ClassA should be
used by FeatureAPart1 independent of whether any other features are selected. Therefore, a user may decide to only use FeatureAPart1 as target for remapping the mapping to ClassA. An equivalent scenario exists for ClassB and FeatureAPart2.

Furthermore, it is also possible to select more than one target feature at a time. In this case, the reference to the source feature in the feature expression is replaced by a complex expression of the effective target features. Whether this expression is built using the AND or the OR operator depends on the intended result and the choice of the user. If the mapped element should be part of a variant if at least one but possibly even all of the effective targets are selected, the OR operator has to be used. In the UML example, this case applies to the mapping to BaseClass, which is required in all cases where at least one of its subclasses should be part of a variant. However, the association between ClassA and ClassB is part of a different scenario. In this case, the UML element should only be used if all target features appear at the same time. Thus, an AND expression of the targets has to be used as a substitute for the reference to the source feature in the original feature expression.

All the different cases of using Split Feature Mapping described above are presented in Table 3.5. The original mapping before the remapping is performed is displayed in Table 3.5. The result of the Split Feature Mapping operator configured according to the scenario described in the previous paragraphs is shown in Table 3.5.

Table 3.5: Example of Split Feature Mapping.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Expression</th>
<th>Mapping to Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureA</td>
<td>FeatureAPart1</td>
<td>UML:Class[BaseClass]</td>
</tr>
<tr>
<td>FeatureA</td>
<td>FeatureAPart2</td>
<td>UML:Class[ClassA]</td>
</tr>
<tr>
<td>FeatureA</td>
<td></td>
<td>UML:Class[ClassB]</td>
</tr>
<tr>
<td>FeatureA</td>
<td></td>
<td>UML:Generalization[ClassA-ClassB]</td>
</tr>
<tr>
<td>FeatureA</td>
<td></td>
<td>UML:Generalization[ClassB-ClassB]</td>
</tr>
<tr>
<td>FeatureA</td>
<td></td>
<td>UML:Association[ClassA-ClassB]</td>
</tr>
</tbody>
</table>

b) after Split Feature Mapping (Split: FeatureA \rightarrow FeatureAPart1, FeatureAPart2)

<table>
<thead>
<tr>
<th>FeatureAPart1 OR FeatureAPart2</th>
<th>Expression</th>
<th>Mapping to Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureAPart1 AND FeatureAPart2</td>
<td>FeatureAPart1 OR FeatureAPart2</td>
<td>UML:Class[BaseClass]</td>
</tr>
<tr>
<td>FeatureAPart1</td>
<td>FeatureAPart1</td>
<td>UML:Class[ClassA]</td>
</tr>
<tr>
<td>FeatureAPart2</td>
<td>FeatureAPart2</td>
<td>UML:Class[ClassB]</td>
</tr>
<tr>
<td>FeatureAPart1</td>
<td>FeatureAPart1</td>
<td>UML:Generalization[ClassA-ClassB]</td>
</tr>
<tr>
<td>FeatureAPart2</td>
<td>FeatureAPart2</td>
<td>UML:Generalization[ClassB-ClassB]</td>
</tr>
<tr>
<td>FeatureAPart1 AND FeatureAPart2</td>
<td>FeatureAPart1 AND FeatureAPart2</td>
<td>UML:Association[ClassA-ClassB]</td>
</tr>
</tbody>
</table>

3.3.1.5 Merge Feature Mapping

The Merge Feature Mapping operator contracts the mappings of one or multiple features and combines them with the mapping of the target feature. It is the remapping operator complementary to the Merge Features evolution presented in Section 3.1.1.5, which melds an arbitrary number of connected features to a single feature. The Merge Feature Mapping operator is capable of nullifying the changes made to the feature model by Split Feature Mapping. Hence, Merge Feature Mapping is considered the inverse operator to Split Feature Mapping. Furthermore,
3.3. REMAPPING OPERATIONS

*Merge Feature Mapping* can be used with a single source feature. In this case, the behavior of the remapping operator is similar to that of *Move Feature Mapping*.

Logically, all occurrences of the source features either individually or as operands of an AND or OR operator have to be substituted with the target feature. Physically, the merge is performed by altering the feature expression of a mapping in the following way: First, all occurrences of individual source features are replaced by the target feature. Second, operator lists are contracted if they now contain the target feature multiple times. This procedure effectively merges the mappings of multiple source features on one target feature. The effects of *Merge Feature Mapping* are demonstrated in Table 3.6.

Table 3.6: Example of *Merge Feature Mapping*.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>a) original mapping</strong></td>
<td></td>
</tr>
<tr>
<td>FeatureAPart1 OR FeatureAPart2</td>
<td>UML:Class[BaseClass]</td>
</tr>
<tr>
<td>FeatureAPart1</td>
<td>UML:Class[ClassA]</td>
</tr>
<tr>
<td>FeatureAPart2</td>
<td>UML:Class[ClassB]</td>
</tr>
<tr>
<td>FeatureAPart1</td>
<td>UML:Generalization[ClassA-BaseClass]</td>
</tr>
<tr>
<td>FeatureAPart2</td>
<td>UML:Generalization[ClassB-BaseClass]</td>
</tr>
<tr>
<td>FeatureAPart1 AND FeatureAPart2</td>
<td>UML:Association[ClassA-ClassB]</td>
</tr>
<tr>
<td><strong>b) after Merge Feature Mapping</strong> (Merge: FeatureAPart1, FeatureAPart2 → FeatureA)</td>
<td></td>
</tr>
<tr>
<td>FeatureA</td>
<td>UML:Class[BaseClass]</td>
</tr>
<tr>
<td>FeatureA</td>
<td>UML:Class[ClassA]</td>
</tr>
<tr>
<td>FeatureA</td>
<td>UML:Class[ClassB]</td>
</tr>
<tr>
<td>FeatureA</td>
<td>UML:Generalization[ClassA-BaseClass]</td>
</tr>
<tr>
<td>FeatureA</td>
<td>UML:Generalization[ClassB-BaseClass]</td>
</tr>
<tr>
<td>FeatureA</td>
<td>UML:Association[ClassA-ClassB]</td>
</tr>
</tbody>
</table>

The example in Table 3.6 builds upon the one in Table 3.5, which was used to illustrate the behavior of *Split Feature Mapping*. Hence, the solution space entities likewise refer to the UML classes of Figure 3.24. Prior to the scenario presented here, the previous feature mapping from FeatureA to all entities in the UML diagram was split into the two parts FeatureAPart1 and FeatureAPart2. The result of this operation is presented in Table 3.6. It is the starting point for the example of *Merge Feature Mapping*. In this example, the operation of splitting a feature mapping is reversed by the use of *Merge Feature Mapping*. Therefore, a merge from source features FeatureAPart1 and FeatureAPart2 is performed for the target FeatureA. It can be seen that all individual occurrences of FeatureAPart1 and FeatureAPart2 are replaced by FeatureA. Furthermore, all complex expressions of both partial features using the OR or AND operator are processed in the same way. Thus, the result of the *Merge Feature Mapping* operator in this example is equivalent to the original mapping of Table 3.5, which shows that the individual source features were merged on the target feature successfully.

It is worth noticing that *Merge Feature Mapping* might produce logically problematic feature expressions if two source features appear as operands of the same logical operator but have different polarity (i.e., one of the occurrences is negated whereas the other is not). When the source features are replaced by the target feature in this scenario, one of two things happens:
In an OR expression, the result will be a tautology as an expression of the form $A \ OR \ NOT \ A$ is always true. In contrast, with an AND expression, the result will be a contradiction of the form $A \ AND \ NOT \ A$, which can never be satisfied. The first case would effectively render the solution space elements of the affected mappings part of the core of the product line as they are included in all variants. The second case would create dead solution space elements, which can never be included in any variant. As it is likely that neither of the two cases is the desired result of the remapping, users are advised to apply special caution when performing *Merge Feature Mapping*.

### 3.3.2 Remapping in the Solution Space

Besides the feature remapping operators for the problem space, it is also possible to perform a remapping from a solution space perspective. In this case, the respective modifications are called object remapping operators. There are two elementary differences between feature and object remapping operators. For one, a feature remapping operator mostly changes a mapping’s feature expression. In contrast, object remapping operators affect the solution space end of a mapping, which means that they modify the target of the reference to a realization artifact. The only exception is *Remove Object Mapping*, which merely deletes a mapping. The second difference of remapping operators for the problem and solution space is that individual mappings are copied differently. In the problem space, mappings have to be copied logically as no two mappings of the same type may exist that point to the same solution space asset. This same constraint still holds for the solution space. However, if a given target object has no previously existing mapping, a physical copy of a mapping for the new target may be created. The details of this procedure will be explained in the following sections.

In sum, three different remapping operators for the solution space are provided that move, copy and remove mappings of a given realization artifact. With these operators, it is possible to adequately remap the evolutions presented in Section 3.1.2. Furthermore, the operators have proven to be sufficient to compensate for the changes performed by the evolutions of Refactory (see Section 4.1.1), which is used as basis for the implementation of the evolution system. Thus, equivalents to the operators that split and merge mappings of the problem space have not been conceptualized. Instead, the following sections will focus on the operators to move, copy and remove an object mapping.

#### 3.3.2.1 Move Object Mapping

The *Move Object Mapping* operator relocates the mappings of a specified solution space artifact to an arbitrary number of targets. Logically, the mappings are moved from the source to all targets. Physically, the original mapping is first copied $n$ times for all target objects. Then, the solution space end of the copied mappings is replaced to point to the respective target object in the second step. Finally, the original mapping is deleted from the mapping model. Optionally, it is possible for users to select effective targets from the list of all potential targets for each individual affected mapping. For example, a mapping could be copied to only two of the three potential target objects if the creator of the remapping permitted it.

Special care is to be applied if a mapping is copied to a target that already participates in another mapping. In this case, the original mapping can not be duplicated physically because
3.3. REMAPPING OPERATIONS

then there would be two mappings of the same type pointing to the same solution space element but with potentially different feature expressions. This situation is explicitly prohibited by FeatureMapper. Thus, the mapping has to be copied logically much like in the problem space. This means, that the feature expressions of the original and the already existing mapping for the target have to be concatenated using the OR operator and then be used as term for the mapping. This way, satisfying at least one of the partial feature expressions leads to the solution space element being included in a variant, which is the same effect as creating a physical copy. In Table 3.7, an example of applying Move Object Mapping with two targets is shown where one of the elements participates in a mapping and the other one does not.

<table>
<thead>
<tr>
<th>FeatureA AND FeatureB</th>
<th>UML:Class[ElementA]</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureC</td>
<td>UML:Class[ElementC]</td>
</tr>
</tbody>
</table>

b) after Move Object Mapping (Move: ElementA → ElementB, ElementC)

<table>
<thead>
<tr>
<th>FeatureA AND FeatureB</th>
<th>UML:Class[ElementB]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(FeatureA AND FeatureB) OR FeatureC</td>
<td>UML:Class[ElementC]</td>
</tr>
</tbody>
</table>

3.3.2.2 Copy Object Mapping

The Copy Object Mapping operator duplicates the mapping of a specified solution space element for an arbitrary number of target objects. Its inner structure is largely equivalent to the Move Object Mapping operator presented in the previous section. However, copying a mapping does not delete the mapping of the original element from the mapping model. To perform the Copy Object Mapping operation, the mapping of the source object is first duplicated n times for all targets. After that, the value of the solution space end of the mapping is substituted with the respective target object. Unlike in the problem space, remapping in the solution space allows to create physical copies of mappings. This means, that the entire entry of a mapping can be cloned. However, Copy Object Mapping has to face the same situation as Move Object Mapping if the target of a copy operation already participates in another mapping. In this case, cloning the original mapping physically and redirecting it to the new target would violate the constraint that there may be only one mapping of each type for a given solution space element. This situation is resolved through creating a logical copy of a mapping by using the same means as in the problem space. In detail, this means that the feature expression of both the original and the already existing mapping for the target have to be connected using the OR operator. A physical copy is not created in this case but the existing mapping has its feature expression modified instead. This procedure constitutes the copy operation for mappings to different solution space assets. In Table 3.8, an example of Copy Object Mapping is presented, which covers the two cases where a target object participates in another mapping and where it does not.
**Table 3.8: Example of Copy Object Mapping.**

<table>
<thead>
<tr>
<th>a) original mapping</th>
<th>b) after Copy Object Mapping (Copy: ElementA → ElementB, ElementC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureA AND FeatureB</td>
<td>FeatureA AND FeatureB</td>
</tr>
<tr>
<td>FeatureC</td>
<td>FeatureA AND FeatureB</td>
</tr>
<tr>
<td>UML:Class[ElementA]</td>
<td>UML:Class[ElementB]</td>
</tr>
<tr>
<td>UML:Class[ElementC]</td>
<td>UML:Class[ElementC]</td>
</tr>
</tbody>
</table>

3.3.2.3 Remove Object Mapping

The *Remove Object Mapping* operator deletes the mapping of an arbitrary number of source elements of the solution space. When compared with the equivalent operation for the problem space, the *Remove Object Mapping* operation is rather straightforward. In the problem space, the feature expression of a mapping has to be modified by removing all references to the features that have their mapping deleted. In contrast, in the solution space, it is possible to simply delete a given entry of the mapping model to remove the mapping from its solution space element. This is due to the fact that each mapping has exactly one realization artifact as solution space end. Thus, if the element is removed, the mapping would point to a null value, which is prohibited by convention. Therefore, the mapping has to be deleted whenever its respective solution space element has its mapping removed. A demonstration of this procedure can be seen in Table 3.9.

**Table 3.9: Example of Remove Object Mapping.**

<table>
<thead>
<tr>
<th>a) original mapping</th>
<th>b) after Remove Object Mapping (Remove: ElementA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FeatureA</td>
<td>FeatureA OR FeatureB OR FeatureC</td>
</tr>
<tr>
<td>FeatureA OR FeatureB OR FeatureC</td>
<td>FeatureA AND FeatureB</td>
</tr>
<tr>
<td>NOT FeatureA AND FeatureB</td>
<td>UML:Class[ElementA]</td>
</tr>
<tr>
<td>UML:Class[ElementA]</td>
<td>UML:Class[ElementA]</td>
</tr>
</tbody>
</table>

| - (mapping removed) | - (mapping removed) |
| - (mapping removed) | - (mapping removed) |
| - (mapping removed) | - (mapping removed) |

3.4 Classification and Remapping of Evolutions

The following section will use the classification system of Section 3.2.3 to group evolutions by their common characteristics. The classification system distinguishes intraspatial and interspatial evolutions. Intraspatial evolutions exclusively perform modifications whose effects are contained within their originating space. In contrast, interspatial evolutions alter models in a way that also has an effect on the mapping model for example by removing a mapped element. In consequence, interspatial evolutions require a subsequent remapping to maintain consistency of a product line. Due to this tight interconnection of the classification and the need for remap-
ping, the concrete steps required for remapping after a particular evolution are explained along with the reasons for placing it in a particular group. The following explanations are structured into separate sections for the evolutions in the problem and solution space analog to Section 3.1. Each section will close with an overview of the categorized evolutions.

3.4.1 Classification and Remapping of Problem Space Evolutions

Section 3.1.1 presented a variety of different evolutions for the feature model in the problem space. The following section inspects these evolutions and gives reasons for their classification. For this purpose, a brief recapitulation of the functionality of each evolution is presented. Furthermore, the steps required for the remapping of the identified interspatial evolutions are explained.

The evolution *Transform to Optional Feature* modifies the cardinality of a feature to make it optional. Likewise, *Transform to Mandatory Feature* changes the cardinality of a feature to make it mandatory. Both evolutions alter the variation type of a feature and are thus considered intraspatial evolutions as they merely modify parameters of a feature, which has no effect on the mapping. In consequence, no remapping is required for *Transform to Optional Feature* and *Transform to Mandatory Feature*.

The *Insert Feature* evolution adds an entirely new feature to an existing group. It is classified as an intraspatial evolution. The reason is that the newly created feature has no relation to previously existing elements of the feature tree that might have a mapping. As a consequence, no remapping is required for the *Insert Feature* evolution.

On the other hand, the *Duplicate Feature* evolution is classified as an interspatial evolution of the first degree. The reason is the connection of the newly created feature and the originally selected feature. As the original feature is duplicated, there is clearly a relation to previously existing elements in the feature model. Should the original feature be part of at least one mapping, a remapping operation is required to update the feature mapping. In the case of duplicating a feature, it seems sound to also clone any existing mapping. Hence, the *Copy Feature Mapping* operator is used on the original feature for remapping. Furthermore, cloned child elements also have to be remapped. The ratio is the same as for the original feature so that *Copy Feature Mapping* is used in this case as well.

It is worth noting the elementary difference between *Insert Feature* and *Duplicate Feature*. Even though both evolutions add a new feature to a group, they are classified as intraspatial and interspatial evolution respectively. The reason is that *Insert Feature* creates a completely new feature while *Duplicate Feature* clones an existing one. This shows that inspecting mere structural changes does not suffice for an appropriate classification that captures the effects on a product line. Instead, the intent of the changes has to be included in the process of classifying an evolution.

The *Split Feature* evolution refines a selected feature by dividing it into several parts. Obviously, the split parts have a logical connection to the original feature so that the original mapping has to be distributed as well. This makes *Split Feature* an interspatial evolution of the first degree. The subsequent remapping is performed by means of the *Split Feature Mapping* operator. In fact, the remapping decides the major part of the semantics of the *Split Feature* evolution. If solution space elements should appear in a variant only when features are selected
Table 3.10: Overview of classification and remapping of problem space evolutions.

<table>
<thead>
<tr>
<th>Evolution</th>
<th>Category</th>
<th>Remapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transform to Optional Feature</td>
<td>intraspatial</td>
<td>-</td>
</tr>
<tr>
<td>Transform to Mandatory Feature</td>
<td>intraspatial</td>
<td>-</td>
</tr>
<tr>
<td>Insert Feature</td>
<td>intraspatial</td>
<td>-</td>
</tr>
<tr>
<td>Duplicate Feature</td>
<td>interspatial</td>
<td>Copy the mapping of the original feature to the new feature.</td>
</tr>
<tr>
<td>Split Feature</td>
<td>interspatial</td>
<td>Use split mapping to distribute the mapping of the original feature to its constituent parts.</td>
</tr>
<tr>
<td>Merge Features</td>
<td>interspatial</td>
<td>Use merge mapping to combine the mapping of the constituent parts on the target feature of the merge.</td>
</tr>
<tr>
<td>Pull Up Feature</td>
<td>intraspatial</td>
<td>-</td>
</tr>
<tr>
<td>Remove Feature</td>
<td>interspatial</td>
<td>Remove the mapping of the deleted feature.</td>
</tr>
<tr>
<td>Remove Feature and Owned Assets</td>
<td>interspatial</td>
<td>Remove the mapping of the deleted feature, all owned assets and dependent elements in the solution space.</td>
</tr>
<tr>
<td>Transform to Alternative Group</td>
<td>intraspatial</td>
<td>-</td>
</tr>
<tr>
<td>Transform to Or Group</td>
<td>intraspatial</td>
<td>-</td>
</tr>
<tr>
<td>Transform to And Group</td>
<td>intraspatial</td>
<td>-</td>
</tr>
<tr>
<td>Duplicate Group</td>
<td>interspatial</td>
<td>When features are duplicated with the groups, copy the mapping of the original features to the new features.</td>
</tr>
<tr>
<td>Merge Groups</td>
<td>intraspatial</td>
<td>-</td>
</tr>
</tbody>
</table>

In combination, the split mapping has to be connected by the AND operator. However, if the solution space elements should appear if either one (or even all) of a set of features is selected, the split mapping has to be connected by the OR operator. Furthermore, it is also possible that solution space elements are assigned to a single feature individually. In either case, these decisions depend on the desired outcome of the evolution and can not be automated. Instead, users are given the possibility of configuring the remapping. It is assumed that the sum of all individual parts satisfies the same requirements as the original feature did. Therefore, the default setting for the remapping operator is to reformulate the feature mappings by connecting all split parts with the AND operator. However, users may decide the details on how to redirect existing mappings.

During the course of melding two or more features as part of the Merge Features evolution, a new feature is created and the selected ones are removed. When merging with the parent feature, only the second step has to be performed. This type of modification has an effect on the mapping model as mappings to no longer existing elements must not remain in the model without being adapted. As a consequence, the Merge Features evolution is classified as an interspatial evolution of the first degree and thus requires remapping. The remapping operator that is used is Merge Feature Mapping, which can combine separate mappings. With the first
3.4. CLASSIFICATION AND REMAPPING OF EVOLUTIONS

Performing the Pull Up Feature evolution makes a feature independent of its parent by elevating it to the next highest level in the feature tree. To categorize the evolution, its modifications to the feature model have to be inspected. During the evolution, features are relocated within the feature tree by detaching them from their old group and adding them to a new group. Subsequently, the cardinality of the affected groups is altered. Neither of these two operations is problematic for the mapping as all features continue to exist in their original form. Furthermore, the evolution might remove groups from the feature model if they are empty after the feature has been moved upwards. As groups are not allowed to have a mapping, this procedure is unproblematic for the mapping model as well. In consequence, Pull Up Feature is classified as an intraspatial evolution and thus does not require any remapping.

The Remove Feature evolution deletes a number of selected features from the feature tree. As the deleted features might have a mapping assigned to them, the evolution potentially affects the mapping model. If a mapped feature is removed, the mapping remaining in the system would be invalid because it applied to a no longer existing feature. Due to this fact, Remove Feature is considered an interspatial evolution of the first degree. Consequently, remapping is required in order to keep the mapping model intact. Therefore, Remove Feature Mapping is used on the originally selected as well as all child features. This remapping procedure ensures that no invalid mappings remain in the mapping model so that the consistency of the product line is maintained.

Applying Remove Feature and Owned Assets first removes a feature and all its descendants from the feature tree and then deletes the solution space elements that were assigned exclusively to those features as well. In other words, the evolution modifies both the problem and the solution space. Furthermore, the removal of elements from either space requires adapting the mappings of the deleted objects. In consequence, Remove Feature and Owned Assets is classified as an interspatial evolution. Due to its effect on both spaces, it is further considered to be of the second degree. The subsequent remapping employs Remove Feature Mapping to rid the mapping model of the no longer relevant mappings. As the deletion in the solution space also removes realization elements depending on the owned assets, a further object remapping might be necessary. For example, associations depending on the deleted elements are removed as well. Therefore, the Remove Object Mapping step is employed for the additionally deleted solution space artifacts. Performing these remapping steps ensures the validity of the mapping model.

Besides the evolutions performed on features, there also is a number of evolutions for groups of a feature tree. For example, the evolutions Transform to Or Group, Transform to And Group and Transform to Alternative Group modify the variation type of a group. All of these evolutions are considered intraspatial for two reasons. First, they merely modify an existing element, which does not affect the mapping. Second, in the feature model of FeatureMapper, groups are not allowed to have a mapping assigned to them. As a consequence, changes that merely affect groups can not affect the mapping. Therefore, no remapping is required when using evolutions to change the variation type of groups.

The evolution Duplicate Group copies a selected group and its contents. In FeatureMapper, groups may not have a mapping, which means that modifications to them can never cause an
evolution to become interspatial. As a consequence, Duplicate Group might easily be perceived as an intraspatial evolution. However, a group may contain features that in turn are part of a mapping. When using the Duplicate Group evolution, contained features are copied as well. In this case, the evolution behaves similar to Duplicate Feature. This characteristic of Duplicate Group makes it an interspatial evolution of the first degree. In consequence, the evolution requires remapping. For this purpose, Copy Feature Mapping is used for all pairs of original and duplicate features in order to maintain the consistency of the product line.

Finally, the Merge Groups evolution relocates all features contained in the selected groups to one common group. Through this, it alters both groups and features. Thus, an individual inspection of both types of modifications is required for an appropriate classification. Changing groups is generally unproblematic as groups can not have a mapping in FeatureMapper. However, altering features might have an effect on the mapping model. In the particular case of Merge Groups, the modification to the features is unproblematic as merely their position in the feature tree is changed, which does not affect the mapping. Thus, Merge Groups is considered an intraspatial evolution and thus does not require remapping. An overview of classifying evolutions in the problem space can be found in Table 3.10.

3.4.2 Classification and Remapping of Solution Space Evolutions

A number of different evolutions for the solution space has been introduced in Section 3.1.2 for three diverse types of realization artifacts. In particular, modifications for UML models, Java source code and the documentation format DocBooklet have been presented. Among them are intraspatial as well as interspatial evolutions. However, the evolutions have not yet been classified. The following sections will analyze the changes made by each evolution and their resulting effect on the product line. As a result, the inspected evolutions will be attributed to their respective group of the classification system presented in Section 3.2.3. Furthermore, appropriate remapping steps will be discussed for the evolutions that are identified as being interspatial. With this knowledge, it is then possible to specify an appropriate remapping procedure to keep a product line consistent after an interspatial evolution was performed.

3.4.2.1 Classification and Remapping of UML Evolutions

In total, five different evolutions are presented for UML models. The modifications performed by the evolutions include adding, modifying and deleting elements. Thus, each evolution has to be inspected individually to perform an adequate classification.

The Duplicate Class evolution copies an existing class and all its dependent associations. Thus it adds new elements to a UML model. Section 3.2.3.1 and Section 3.2.3.2 explained, that the mere creation of new elements is no sufficient condition for a classification as either intraspatial or interspatial evolution. Therefore, it has to be inspected if the newly added elements have a logical connection to previously existing elements that might have a mapping. This is clearly the case as the newly created class is a clone of the selected class. Equivalent holds for the duplicated associations. Thus, the ratio for classifying Duplicate Class is equivalent to that of the problem space evolution Duplicate Feature so that the evolution can be considered interspatial. The subsequent remapping has to copy the mapping of the original elements to the
respective duplicates. Hence, the Copy Object Mapping operator is employed to duplicate the mapping of the original to the duplicate class as well as the mapping of the original associations to the newly created associations.

The Extract Super Class evolution creates a new super class and moves the selected attributes of one or many classes to it. Thus, the evolution modifies existing elements and adds a new class to a UML model. Modifying existing elements is an intraspatial operation as the border to the mapping of a product line is not crossed. However, adding a new class may constitute an interspatial operation that makes the entire evolution interspatial if a connection between the newly created class and a previously existing element exists. Due to the fact that the newly created class serves as super class for the existing class, a connection can clearly be perceived. Therefore, Extract Super Class is considered an interspatial evolution, which requires remapping. For this purpose, the mapping of the original class is extended to its super class by employing Copy Object Mapping. Furthermore, it has to be ensured that the new class is part of all variants that require at least one of the extracted attributes. Thus, the mapping of the moved attributes is also copied to the new super class. With this remapping, the super class will be available in all variants that require at least one of the extracted attributes or the original class.

Table 3.11: Overview of classification and remapping of UML evolutions.

<table>
<thead>
<tr>
<th>Evolution</th>
<th>Category</th>
<th>Remapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duplicate Class</td>
<td>interspatial</td>
<td>Copy the mapping of the original class to the new class. Furthermore, copy the mapping of all descendant elements and dependent associations to their respective clones.</td>
</tr>
<tr>
<td>Extract Super Class</td>
<td>interspatial</td>
<td>Copy the mapping of the original class and the extracted attribute(s) to the new super class.</td>
</tr>
<tr>
<td>Replace Method with Method Object</td>
<td>interspatial</td>
<td>Copy the mapping of the original class to the new class and its incoming association.</td>
</tr>
<tr>
<td>Inline Method Object</td>
<td>interspatial</td>
<td>Move the mapping of the method object class and its incoming association to the referring class.</td>
</tr>
<tr>
<td>Remove Class</td>
<td>interspatial</td>
<td>Remove the mapping of the deleted class.</td>
</tr>
</tbody>
</table>

When using the Replace Method with Method Object evolution, a new method object class is created, which serves as target for the relocation of the extracted methods. Furthermore, an association between the original class and the method object class is created. Again, the newly created class has a connection to previously existing elements. In this case, the new class represents part of the functionality described by the original class. Furthermore, the relation of both classes is clearly visible through the association connecting them. Thus, Replace Method with Method Object is classified as interspatial evolution. The subsequent remapping uses the newly created class and its incoming association as target of multiple Copy Object Mapping operations. The sources are the original class and the extracted methods. The created mapping includes the method object class and its incoming association in all variants that require the
original class or one of the extracted methods.

The *Inline Method Object* evolution is the inverse operation to *Replace Method with Method Object*. Thus, the evolution relocates the methods in the method object class to the class referring to the method object and then deletes the method object class and its incoming association. According to Section 3.2.3.2, deleting an element is a sufficient condition for classification as interspatial evolution as the mapping model is necessarily affected. Subsequently, remapping is required to keep the product line consistent. In the case of *Inline Method Object*, the mappings of the method object class and its incoming association are moved to the referring class. If the referring class already has a mapping of equivalent type, the result of this procedure is similar to removing the mapping of the method object class and its association. Otherwise, it is ensured that the referring class is included in variants that required the method object class because satisfying the same conditions now includes the referring class as well.

Finally, the *Remove Class* evolution deletes a selected class and all its dependent associations from a UML model. Again, deleting existing elements suffices for a classification as interspatial evolution. To maintain the validity of the product line, remapping is necessary. In this case, the existing mappings of the deleted class and its associations are deleted as well. For this purpose, the *Remove Object Mapping* operator is employed, which deletes the respective mappings from the mapping model.

The classification of UML evolutions revealed that all of the presented modifications are interspatial in nature. Therefore, it is essential to apply appropriate remapping steps after the respective changes to make the evolutions suitable for the use with software product lines. The results of the classification of UML evolutions can be found in Table 3.11.

### 3.4.2.2 Classification and Remapping of Java Evolutions

The evolutions for the Java programming language presented in this thesis aid in the modification of source code files. The performed changes encompass the creation and modification of source code fragments. Thus, each of the provided evolutions has to be inspected in detail to determine its category in the classification system.

When adding a parameter to a method using the *Insert Parameter* evolution, a new element is created in the model of the modified Java code. It has been argued before that creating new elements may be an indicator for both an intraspatial or an interspatial evolution depending on the semantic of the changes. Thus, it has to be determined whether there is a connection between the newly created element and an already existing one. In the case of the newly created parameter, this connection can not be established. Thus, *Insert Parameter* is classified as an intraspatial evolution. As intraspatial evolutions do not affect the mapping model, no remapping is required after performing the changes.

During the course of *Rename Element*, the selected source code element is assigned a new name and all references are modified to reflect this change. Thus, the modifications of the evolution potentially affect a large part of the model reflecting the source code. However, the mere modification of existing model elements is not sufficient for a classification as interspatial evolution. In particular, changing the names of elements has no effect on the mapping model as it uses memory objects for reference and not their names. Thus *Rename Element* is an intraspatial evolution, which does not require remapping.
Table 3.12: Overview of classification and remapping of Java evolutions.

<table>
<thead>
<tr>
<th>Evolution</th>
<th>Category</th>
<th>Remapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert Parameter</td>
<td>intraspatial</td>
<td>-</td>
</tr>
<tr>
<td>Rename Element</td>
<td>intraspatial</td>
<td>-</td>
</tr>
<tr>
<td>Extract Method</td>
<td>interspatial</td>
<td>Copy the mapping of the original method and the extracted statement(s) to the new method.</td>
</tr>
<tr>
<td>Remove Statement</td>
<td>interspatial</td>
<td>Remove the mapping of the deleted statement.</td>
</tr>
</tbody>
</table>

The modifications performed during the course of the *Extract Method* evolution are more complex than those of prior evolutions for Java. First, a new method is created. Then, the extracted statements are relocated and finally, a method call to the new method is added at the position of extraction. All in all, two new elements are created and an arbitrary number of statements is moved. Moving an element is unproblematic as it is realized by setting an attribute of the model element to a new value and thus is considered a change of existing elements. However, creating new elements may suffice for a classification as interspatial evolution if a connection between the new element and a previously existing element can be established. Logically, this connection exists due to the fact that the new method holds the statements of the extraction site. Physically, the connection is manifested in the form of the method call. Thus, *Extract Method* is classified as interspatial evolution. The subsequent remapping consists of two *Copy Object Mapping* operations that each have the new method and the method call as targets. The sources of the copy operations are the original method and the extracted statements respectively. Through this remapping, the extracted method is included in all variants that require the original method or one of the extracted statements.

Finally, the *Remove Statement* evolution can be used to delete obsolete statements from the Java source code. Section 3.2.3.2 explained that removing elements from a model is a sufficient condition for classification as interspatial evolution. Hence, *Remove Statement* is a member of this group of evolutions and requires remapping. In accordance with the performed modification, the respective mapping is deleted as well by using the *Remove Object Mapping* operator. Through this, the consistency of the product line and its mapping is maintained.

In summary, half of the presented evolutions constitutes intraspatial evolutions whereas the other half is interspatial. Evolutions of the latter category were provided with adequate remapping steps to ensure the validity of the mapping model after the modifications. A summary of the classification and remapping of Java evolutions can be found in Table 3.12.

### 3.4.2.3 Classification and Remapping of DocBooklet Evolutions

As documents are one part of the solution space of a product line, they may be subject to change similar to any other realization asset. For this purpose, a number of evolutions is provided for DocBooklet files, which serve as example of a documentation format. The modifications are used to alter the structure and content of a document. Thus, their functionality is diverse and has to be inspected in detail to perform an adequate classification of the evolutions.

The group of evolutions to duplicate chapters, sections and paragraphs copies an existing
element and adds the clone to the document. Thus, the evolutions add new elements to the file. Again, it has to be established if the new element has a relation to previously existing elements in order to determine its group in the classification. As with the other duplicate evolutions, a relation of the new element to a previously existing element can clearly be established because the element was duplicated. Thus, the respective evolutions are interspatial and require remapping. Analog to the previously introduced duplicate evolutions, the mapping of the original element is copied to the clone by use of Copy Object Mapping. An analog procedure is applied for all descendant elements, which were copied as well.

Table 3.13: Overview of classification and remapping of DocBooklet evolutions.

| Evolution         | Category               | Remapping                                                      |
|-------------------|------------------------|                                                               |
| Duplicate Chapter | interspatial (first degree) | Copy the mapping of the original chapter/section/paragraph to the new chapter/section/paragraph. Copy the mapping of all descendants to their clones. |
| Duplicate Section |                        |                                                                |
| Duplicate Paragraph |                        |                                                                |
| Extract Chapter   | interspatial (first degree) | Copy the mapping of the original chapter/section and its contents to the new chapter/section. |
| Extract Section   |                        |                                                                |
| Move Up Chapter   | intraspatial           |                                                                |
| Move Up Section   |                        |                                                                |
| Move Up Paragraph |                        |                                                                |
| Move Down Chapter |                        |                                                                |
| Move Down Section |                        |                                                                |
| Move Down Paragraph |                        |                                                                |
| Remove Chapter    | interspatial (first degree) | Remove the mapping of the deleted chapter/section/paragraph. |
| Remove Section    |                        |                                                                |
| Remove Paragraph  |                        |                                                                |

The Extract Section evolution is used to create a new section from a set of paragraphs. Likewise, Extract Chapter creates a new paragraph from a number of selected sections. In both cases, a new container is created and existing elements are relocated to it. Therefore, a new element is added and existing elements are modified. The mere modification of existing elements does not justify a classification as interspatial evolution. However, adding new elements has the potential of doing so if the created elements have a logical connection to previously existing elements. In the case of Extract Section and Extract Chapter, this relation exists as the new container holds contents of the original container. Thus, the evolutions are considered to be interspatial and require remapping. It is likely that the newly created sections or chapters should be included in the same variants as their archetypes. Thus, the mapping of the existing container is duplicated to point to the newly created container by means of Copy Object Mapping. Through the use of remapping, the restructuring of the document can be performed while still providing the same content to individual variants of the product line.

To further change the structure of a document, evolutions are provided that can reorder the content of a DocBooklet file. In particular, it is possible to move chapters, sections and paragraphs up or down in a document. Each of the evolutions relocates existing content but
3.4. CLASSIFICATION AND REMAPPING OF EVOLUTIONS

does not create or remove elements. According to Section 3.2.3.1, this type of modification is unproblematic for the mapping model so that all six move evolutions are regarded as intraspatial evolutions. In consequence, no remapping is required and the evolutions can be used within a product line without further adaptation.

Finally, it is possible to delete obsolete content from a document by using one of the evolutions *Remove Chapter*, *Remove Section* or *Remove Paragraph*. Each of these evolutions deletes the respective selected element from the document. This type of operation has an effect on the mapping model so that all three evolutions are considered to be interspatial. Consequently, it is necessary to perform a remapping to keep the product line consistent. In the case of *Remove Chapter*, *Remove Section* and *Remove Paragraph*, the *Remove Object Mapping* step is employed to delete the mapping of the no longer existent elements from the mapping model.

The analysis of the DocBooklet evolutions revealed that six of the provided 14 evolutions are intraspatial. The remaining eight evolutions are interspatial. With the presented remapping steps, it is possible to use these evolutions for software product lines without jeopardizing the validity of the mapping model. The results of the classification of DocBooklet evolutions are displayed in an overview in Table 3.13.
Chapter 4

A Framework for Evolutions in FeatureMapper

The following chapter will present the realization artifacts building upon the theoretical basis laid out by the previous chapters. In the first section, relevant technology for the implementation will be introduced. After that, an overview of the implementation of the evolution system will be provided in the second section. Finally, in the third section, detailed instructions on how to extend the system with further evolutions and remappings will be presented for the problem as well as the solution space.

4.1 Relevant Technology

In order to provide evolutions for software product lines, a system needs to be able to create, modify and delete mappings from features to solution space artifacts, handle a multitude of different model types and make modifications to models according to a specification. Due to the complexity of these tasks, it seems prudent to employ tools that provide part of the required functionality.

First of all, FeatureMapper provides the basic functionality for working with model-based software product lines. It lets users create a feature tree as well as the mapping from problem to solution space. As it uses Ecore-based models extensively, it builds on the functionality of the Eclipse Modeling Framework (EMF) and so does the work of this thesis.

For the part of evolving models, the tool Refactory is used as it allows to define generic model refactorings. Additionally, EMFText is employed to define a declarative remapping language and to extend the number of different supported solution space model types. The tool is capable of generating a parser and editor for a textual syntax to EMF models so that textual languages can be targeted by evolutions as well. Furthermore, EMFText is used to define a textual language to describe the steps necessary for remapping.

FeatureMapper and EMF have already been introduced in prior sections and thus will not be discussed further. However, the basic concepts of Refactory and EMFText will be explained in detail in the next two sections.
4.1.1 Refactory

Refactory\footnote{http://www.reuseware.org/index.php/Refactoring} [RSA10, Rei10] is a tool for generic model refactoring developed at Technische Universität Dresden. It provides genericity for modifications by first specifying the abstract steps required to perform a refactoring independent of a concrete type of model and then mapping this specification to individual elements of a particular metamodel.

For example, Extract Super Class for UML models is specified as a mapping to UML of a generic refactoring ExtractX. ExtractX specifies that there are a number of “extractees”, an original container, a new container and a so called “container container”, which is the parent of both the original and the new container. The abstract steps for ExtractX are to create a new element of the type “new container” in “container container”, remove the “extractees” from the original container, place them in the new container and finally create the link between the original and new container. In the concrete case of Extract Super Class, the “extractees” are attributes, the original container is the class that contains them, the new container is the newly created super class and the “container container” is the package both classes reside in. However, this general procedure might just as well be used in a documentation format to extract a particular paragraph from a chapter to a newly created superordinate chapter. All that is needed to make the refactoring available to the documentation format is a mapping to its metamodel elements.

In more detail, Refactory uses role modeling [RWL96] to create genericity. A role can be “played” by a particular object for a certain period of time before it is dismissed again. Through this, a role provides a particular view on that object for a limited time. This characteristic makes role modeling a very suitable approach for looking at objects and their connections for the duration of a refactoring.

Despite its name, Refactory is not limited to semantic preserving changes but in fact allows arbitrary modifications of Ecore-based models. The following explanations will use the term “refactoring” for an individual modification to stay consistent with Refactory’s terminology. However, the more general phrase “evolution” could be used as well. This makes Refactory a useful tool for implementing evolutions for model-based software product lines. However, the development branches of Refactory and FeatureMapper should remain independent so that no custom version of Refactory is required to execute the evolutions in this thesis. As a consequence, modifications to Refactory are only possible if they pursue the goal of generic model refactoring but not the specifics of evolving software product lines. On occasion, this results in minor inconveniences. For example, the menu entry for all currently applicable modifications is named “Refactor” even though it really contains the more general evolutions. However, these problems do not hinder the functionality of evolving software product lines, which makes them acceptable.

Refactory uses up to four different types of files to specify a new refactoring. Out of these four files, two specify the abstract refactoring and one creates a binding to a concrete metamodel. The optional fourth file allows to use regular Java source code to perform changes that can not sufficiently be expressed in the abstract refactoring. After they were specified, all files have to be registered with Refactory by use of individual extension points. The names of the respective extension points will be mentioned but an explanation of creating extensions for them is omitted at this point. Details on binding the extension points can be found in Section 4.3.1. All four types of files will be explained thoroughly in the following paragraphs.
4.1. RELEVANT TECHNOLOGY

4.1.1.1 Role Model (*.rolestext)

Refactory requires a role model to be specified in a *.rolestext file and registered to the extension point org.emftext.refactoring.rolemodel. The main intent of the role model is to define the roles participating in a particular refactoring by giving them a name. Additionally, roles may have a modifier applied to them. There are three possible options for role modifiers. First, “input” marks a role as being the input to the refactoring, which means that the user has to select an element of the type that is mapped to this role for the refactoring to be applicable. Second, “super” states that it suffices if a class of a selected element is derived from the type mapped to this role. Hence, the mapped class is a super class of the actual type of the selected element and the object is not required to be exactly of the specified type. Third, a role may be marked as “optional”, which means that it does not necessarily have to be mapped to a concrete metamodel class.

Furthermore, the role model specifies how roles reference each other through role collaborations. Among these are the role composition and the role association, which are conceptually equivalent to the UML class composition and association respectively. Both collaborations have a role as source and target element, which each have a minimum and maximum cardinality applied to it. With collaborations, it is possible to express relations of roles like the one mentioned in the above example where a role “container” possesses many elements of role “extractee”. In this case, a role composition from role “container” with minimum and maximum cardinality of one to the target role “extractee” with a cardinality from one to unlimited has to be specified. Additionally, the “transitive” modifier may be used on a role collaboration. It states that a relation between two roles does not necessarily have to be present directly on the mapped model elements but may be established through other intermediate elements as well. The entire role model of ExtractX is presented in Listing 4.1 in the form of a *.rolestext file to illustrate some of the concepts described in this section. A complete definition of the syntax and semantic of role models can be found in [Rei10].

Listing 4.1: Complete role model for ExtractX.

```
RoleModel ExtractX {
  input super ROLE Extractee;
  ROLE OrigContainer;
  ROLE NewContainer(input newName);
  ROLE ContainerContainer;

  transitive : OrigContainer [1..1] <-- Extractee extracts [0..*];
  transitive : NewContainer [1..1] <-- Extractee moved [0..*];
  transitive : ContainerContainer [1..1] <-- NewContainer target [1..1];
  transitive : ContainerContainer [1..1] <-- OrigContainer source [1..1];
  transitive : OrigContainer [1..*] --> NewContainer reference [1..*];
}
```
CHAPTER 4. A FRAMEWORK FOR EVOLUTIONS IN FEATUREMAPPER

4.1.1.2 Refactoring Specification (*.refspec)

While the role model defines which roles exist and how they relate to one another, it does not specify the steps needed to perform a particular refactoring. This is done by the second type of file required by Refactory, the so called refactoring specification. It is defined in a *.refspec file, which supports a domain specific language to formulate instructions on how to carry out a specific refactoring. It has to be registered using the extension point org.emftext.refactoring.refspec.

The first conceptual part of the refactoring specification is dedicated to retrieving concrete objects from the roles defined in the role model. For this purpose, Refactory offers object assignment commands (out of which three will be presented) that can resolve roles to the objects bound during runtime. The instruction “filter” selects all concrete objects matching the requested role, whereas “trace” selects parent elements of the specified input that fit. As third assignment command, “uptree” selects the most special element in the containment hierarchy that plays the requested role and is common to all input elements. This is used to determine common container elements.

The second conceptual part of the *.refspec file specifies the steps needed to perform the refactoring. For this purpose, Refactory has instructions to create, move and remove elements as well as to set values. For the sake of brevity, the details of these operations are omitted at this point. Details on the refactoring specification can be found in [Rei10]. Furthermore, an exemplary glance at the capabilities and syntax of a *.refspec file for the generic ExtractX refactoring is provided in Listing 4.2.

Listing 4.2: Complete refactoring specification for ExtractX.

```
REFACTORING FOR <ExtractX>

STEPS {
    object containerContainerObject := ContainerContainer from uptree(INPUT);
    object origContainerObjects := OrigContainer as trace(INPUT);

    create new nc:NewContainer in containerContainerObject;
    assign nc.newName;
    move OrigContainer.extracts to nc distinct;
    set use of nc in origContainerObjects;
}
```

4.1.1.3 Role Mapping (*.rolemapping)

Having the role model and the refactoring specification in place concludes the generic part of a refactoring. What remains to be done in order for the refactoring to be applicable, is to bind the generic part to a concrete type of model. This is done in the third type of file called a role mapping, which is specified in a *.rolemapping file and then has to be registered using the extension point org.emftext.refactoring.rolemapping. As first step of binding a generic refactoring, a particular type of target model, such as UML, has to be selected by specifying
its namespace URI. In the second step, entities of the selected target model type have to be assigned to the roles defined in the role model. It is possible to map concrete as well as abstract entities. In the latter case, all specializing sub-types are permissible for the bound role. To finalize the mapping, a name has to be assigned to the newly created concrete refactoring. In Listing 4.3, it is shown how the generic refactoring $ExtractX$ is mapped to the UML metamodel to create the concrete refactoring $Extract Super Class$. A detailed explanation of role mappings can be found in [Rei10].

Listing 4.3: Complete role mapping of $ExtractX$ as $Extract Super Class$ for the UML metamodel.

```
ROLEMODELMAPPING FOR <http://www.eclipse.org/uml2/3.0.0/UML>

ExtractSuperClass maps <ExtractX> {

Extractee := Property;
OrigContainer := Class {
    extracts := ownedAttribute;
    reference := superClass;
};
NewContainer := Class(newName -> name) {
    moved := ownedAttribute;
};
ContainerContainer := Package {
    source := packagedElement;
    target := packagedElement;
};
}
```

Naturally, a generic refactoring consisting of role model and refactoring specification can be mapped multiple times to create a number of different concrete refactorings. The mappings may use the same or different metamodels for each concrete refactoring. As an example of the first case, the generic refactoring $ExtractX$ can be mapped to the UML metamodel again to create another concrete refactoring $Extract Interface$, which creates a new interface and makes the selected class implement all of its methods. An example of the second case might be to reuse the refactoring in a different context such as in the aforementioned extraction of paragraphs to sections in a documentation format. Through this mechanism, a refactoring can easily be reused in an entirely different language with only minimal effort.

### 4.1.1.4 Post Processor

Refactory allows an optional fourth type of file to complement a concrete refactoring. It is called “post processor” and it is a regular Java source class. A post processor has to be registered using the extension point `org.emftext.refactoring.postprocessor`. The only requirement
is that the provided class implements the interface `IRefactoringPostProcessor`. For the sake of convenience, an abstract base class called `AbstractRefactoringPostProcessor` is provided, which implements all methods of the interface so that only individual pieces of functionality have to be customized. The purpose of a post processor is to perform modifications that can not sufficiently be expressed in the refactoring specification using the domain specific language of Refactory. For example, one shortcoming of the refactoring specification is that it can not copy elements. If this type of modification is required, a post processor has to be employed.

In the control flow of Refactory, the modifications of the refactoring specification are performed first and then the post processor is executed. This makes the post processor an ideal mechanism for performing a remapping after an interspatial evolution. Hence, the implementation of evolutions for software product lines uses post processors to perform the remapping in both problem and solution space. However, both implementations vary significantly due to the specific requirements of evolutions in each of the spaces. Details on this can be found in Section 4.2.

### 4.1.2 EMFText

EMFText\(^2\) is an Eclipse plug-in developed at Technische Universität Dresden, which allows to define textual syntax for arbitrary Ecore metamodels. For this purpose, a concrete syntax has to be specified in the form of a *.cs file, which describes rules for the textual representation of the individual metamodel elements. With a valid concrete syntax specification, EMFText can automatically create a parser for the resulting language. Additionally, a custom editor for the language is provided, which features syntax highlighting and code completion. The text written in the editor will automatically be converted to an instance of the underlying metamodel in the background so that it can be processed with the regular tools used for EMF models. Furthermore, EMFText enables FeatureMapper to map to textual representations of models. Through this, a feature can be mapped to a method in a programming language or a chapter in a documentation format provided that an EMFText concrete syntax exists. For example, the DocBooklet format used in this thesis was defined using EMFText. The concrete syntax serves as an example, which can be found in Appendix B.2.

Refactory, which was introduced in the previous section, makes heavy use of EMFText for its different configuration files. Additionally, the fact that EMFText files are automatically converted to Ecore models yields that Refactory can be used on the textual representation as well. This effectively allows to modify the textual and the regular representation of models making the combination of EMFText and Refactory a valuable asset for the evolution of model-based software product lines.

Furthermore, EMFText is used directly in this thesis as well. In the solution space, it is possible to declaratively specify the concrete remapping procedure in an *.orspec file as will be introduced in Section 4.2.2.2.2. This type of file builds on the configuration files used by Refactory and the roles declared in the rolemodel. Hence, it seems a natural choice to mimic the syntax used by Refactory for the *.orspec files. This eases the transition between the file types when support for remapping is added to existing evolutions in the solution space. Employing the same technology as Refactory to parse the configuration files seems a sound choice so that

\(^2\)http://www.emftext.org
4.2. IMPLEMENTATION

this thesis uses EMFText as well. For the sake of brevity, the concrete syntax for *.orspec files and the EMF model it builds on are omitted at this point. However, the full details can be viewed in Appendix A.2 and Appendix A.1 respectively.

4.2 Implementation

All classes for the implementation of the evolution for software product lines in FeatureMapper can be found in the package org.featuremapper.evolution. Within this package, four major systems of the implementation are distinguished in separate packages. First, “evolutions” contains the implementation of the evolutions presented in Section 3.1. Second, the remapping operators from Section 3.3 can be found in the package “remapping”. Third, the classes in the package “UI” create ties to the user interface to enable graphical input and output of relevant values for evolutions and remapping. Finally, the package “test” provides a test suite used to validate the implementation of the concepts presented in this thesis. In Figure 4.1, an overview of the systems and their relations is provided.

Figure 4.1: Architectural overview of the major systems of the implementation.

The majority of the systems is further divided into subpackages for the problem and solution space implementations respectively. Great care was taken to use equivalent concepts for both spaces as far as possible. However, the specific characteristics of either space require adaptations at some points. The detailed explanations of the systems in the following sections reflect this fact. Thus, they first explain the general concepts of a package before discussing the individual details of the implementations for either one of the two spaces.

4.2.1 Implementation of the Evolutions System

The evolutions presented in this thesis are implemented as members of the package org.featuremapper.evolution.evolutions. The base class for the evolutions in source code is AbstractEvolution. It provides basic functionality such as retrieving objects for roles, simplifying remapping plans or informing users about problems during an evolution. Its subclasses implement evolutions for problem as well as solution space. The most common classes for evolution are presented in Figure 4.2, which serves as a reference for the explanations in the following paragraphs.
Using these classes, evolutions are implemented in the Java programming language. However, evolutions in the solution space do not necessarily have to be declared in source code. Alternatively, the generic refactoring mechanism of Refactory can be used to describe the steps of modification declaratively in the refactoring specification. The concrete approach used when implementing a new evolution depends on the complexity of the performed operations. If the evolution is simple enough to be implemented generically using the refactoring specification, this option may be chosen as the abstract specification can then be reused within another context. However, if the performed changes are too complex or require knowledge specific to the target meta model, an evolution should be implemented as sub class of \texttt{AbstractSolutionSpaceEvolution}.

The last point is the reason for why there are no generic evolutions in the problem space. Each evolution on the feature tree uses knowledge specific to that type of model. For example, the cardinality and name of an element are modified. Furthermore, the structuring into groups and features is key to many modifications. As a result, the problem space uses evolutions in source code that are subclasses of \texttt{AbstractProblemSpaceEvolution}. The specifics of the implementation for evolutions in either space are left to the respective subsections.

It is worth noting that all evolutions can be executed independently of the control flow of Refactory. Each evolution takes the required input values as parameters to its constructor. This avoids misconfiguration due to neglecting to specify required values, which might happen if the parameters were to be set by individual methods. After this configuration, an evolution can be executed directly. This design decision makes evolutions more versatile and allows them to be tested without the need to configure Refactory.

However, evolutions can still benefit from the user interface integration and generic refactoring capabilities of Refactory. To integrate them with Refactory, a so called \texttt{AbstractEvolutionAdapter} is required. The adapter class inherits from \texttt{AbstractRefactoringPostProcessor}, which in turn implements the interface \texttt{IRefactoringPostProcessor}. This relation bridges the gap to Refactory so that evolution adapters can be registered to be executed during the control
flow of Refactory. The adapter then creates and configures an appropriate evolution, which will be executed.

Each adapter is registered as post processor to a concrete refactoring. As explained in the introductory section to Refactory, a concrete refactoring consists of a role model, a generic refactoring specification and a role mapping, which ties the abstract refactoring to a specific metamodel. In the case of problem space evolutions, the concrete refactoring is implemented as dummy consisting of a simple role model, a refactoring specification that merely loads objects of the specified roles but does not perform any modifications and a role mapping to the metamodel of the feature tree. Solution space evolutions may choose to use the same generic refactoring and map it to the targeted solution space metamodel if they want to implement the real evolution in source code. In both these scenarios, the evolution is executed by the evolution adapter registered as post processor.

In order to create an evolution, evolution adapters have to convert the parameters received from Refactory into valid parameters for the evolution. For example, it has to be decided, how objects of a role are to be used for the evolution. With these values, the evolution related to the adapter is created. Through this specific configuration process, each evolution adapter is tied to exactly one evolution. Thus, it is necessary to create a pair of evolution and evolution adapter for each modification that should be available through the menu provided by Refactory. A UML diagram displaying a subset of the evolutions and their evolution adapters is shown in Figure 4.3.

Figure 4.3: UML diagram showing a subset of the evolutions with their respective evolution adapters.

### 4.2.1.1 Implementation of Evolutions in the Problem Space

All evolutions in the problem space modify the feature tree. Thus, they use the mechanisms provided by Refactory to be available only if suitable elements of the feature tree are selected. For this purpose, a simple role model containing only a single role called “Element” is provided. The refactoring specification merely serves as dummy that loads the objects matching this role. In the role mapping, the “Element” role is then mapped to either a group or a feature. The concrete choice depends on whether an evolution should be triggered for a feature or a group. The only exception is the Insert Feature evolution, which can be executed on features as well.
The basic class for evolutions in the problem space is AbstractProblemSpaceEvolution. It provides the elementary sequence of method calls that perform an evolution in five steps. It is assumed that all specialized evolution classes take the required configuration values as parameters to their constructor so that all modifications can be performed. First, it is checked whether the preconditions for applying the evolution are met. Second, the modifications to the feature model are performed according to the concrete evolution. After this point, the last three steps will only be performed if it is not a preview run of the evolution used to display preliminary results of the modification in the user interface. In the third step, the remapping for the evolution will be performed by creating a remapping plan, letting users set parameters in the remapping dialog and then executing the plan. As fourth step, the mapping model is saved in order to keep it consistent with the performed modifications. Finally, the user interface is updated to reflect the changes made during the evolution and remapping. In Listing 4.4, an excerpt from AbstractProblemSpaceEvolution in Java source code is shown that reflects these five steps.

Listing 4.4: Java source code excerpt from AbstractProblemSpaceEvolution showing the five steps required for problem space evolutions.

```java
checkPreconditions();
evolve();

if (!getIsPreviewMode()) {
    remap();
    saveMappingModel();
    updateUI();
}
```

The exact operations performed in each of these methods depend on the concrete evolution and are thus implemented in the subclasses of AbstractProblemSpaceEvolution, which will be explained in the following paragraphs. An overview of all concrete evolutions for the problem space is provided in Figure 4.4.

The names of the sub classes are chosen in accordance with the names of the evolutions they implement. An explicit mapping between classes and evolutions is thus omitted at this point. Instead, the most noteworthy characteristics of the implementation are presented for all concrete classes.

All evolutions for the variation type of features and groups have the common base class AbstractVariationTypeEvolution. Its concrete subclasses implement the evolutions for the variation type presented in Section 3.1.1.1 and Section 3.1.1.9 respectively. Their implementations consist of mere modifications of the minimum and maximum cardinality of the target element. The particular nature of the changes has already been described in the aforementioned sections and the operational details are elementary so that no further explanation is necessary at this point.

The Insert Feature evolution is special as it can be triggered when either a feature or a group as groups. Its specifics will be explained later.
4.2. IMPLEMENTATION

Figure 4.4: UML diagram of the classes for evolutions in the problem space.

is selected. Making this possible requires adaptation of the role mapping. The role “Element” from the role model is thus mapped to the class FeatureTreeNode, which is the base class of features as well as groups in the feature tree. A distinction is then made in the source code of the evolution to determine the adequate steps of inserting a feature for the selected tree node.

The evolutions Duplicate Feature and Duplicate Group are both capable of copying an entire branch of the feature tree containing not only the selected element but also all of its children and further descendants. Thus, the subsequent remapping needs to be performed for all indirectly copied elements as well. During the course of evolution, both Duplicate Feature and Duplicate Group also copy all references to constraints originating from the original features. Furthermore, the user interface of FeatureMapper is updated in a way that all duplicated elements are selected provided that their original counterparts were selected prior to the evolution.

The Split Feature evolution allows users to select into how many parts a feature should be split. Furthermore, names can be provided for all constituent parts. After creating the split parts, constraints referring to the original feature are diverted to them. As last step of the evolution, the user interface is updated so that all split parts are selected if the original feature was previously selected.

Merging can be performed for features and groups in two individual evolutions. Each of the
evolutions uses a precondition to check for an appropriate number of selected features. For the
*Merge Features* evolution, it suffices to select a single feature if it was chosen to merge with the
parent feature. In all other scenarios, the *Merge Features* and *Merge Groups* evolutions require
at least two elements to be selected. In either case, it is checked that the selected elements are
siblings in the feature tree.

The *Pull Up Feature* evolution detaches a feature from its group and reattaches it as child of
the group one logical level above the current one. The preconditions of the evolution check that
the selected feature is at least two logical levels below the root feature to avoid the problems
described in Section 3.1.1.6.

The *Remove Feature* evolution deletes a selected feature from the feature tree. During the
course of the evolution, all no longer required constraints are also deleted. For this purpose, the
implementation keeps track of all referenced constraints of a feature. This ensures that only the
constraints affected by the evolution that are no longer used are removed while leaving all other
constraints unchanged even if no feature should reference them.

The *Remove Feature and Owned Assets* evolution builds upon *Remove Feature*. It reuses
the functionality to delete a selected feature from the problem space and adds support to delete
owned solution space assets. In Section 3.1.1.8, a definition was provided for which criteria
have to be met to consider a solution space element as being an owned asset. This definition is
applied to locate the respective realization artifacts. Furthermore, the owned assets are analyzed
to find UML elements that participate in associations. As the owned assets are to be deleted,
the dependent associations can be deleted as well as they would be rendered invalid if one
of their targets ceases to exist. The subsequent remapping is performed in accordance with
Section 3.4.1 as a mixture of feature and object remapping steps. This type of remapping is
exemplary for interspatial evolutions of the second degree, which affect problem and solution
space simultaneously.

Some of the more complex evolutions described in this section require a user interface to
allow interaction with users. In particular, the evolutions *Insert Feature*, *Duplicate Feature,*
*Split Feature* and *Merge Features* come with dialog pages dedicated to receiving configuration
parameters. In Figure 4.5, an overview of the respective classes is shown.

Figure 4.5: Parameter pages for the problem space evolutions requiring user input.
The common base class for the user input dialog pages for problem space evolutions is the class AbstractParameterPage. It inherits from the class ModelRefactoringWizardPage provided by Refactory, which allows adding custom pages to the refactoring wizard. The completed wizard pages are passed as parameters to post processors of a refactoring. In the concrete case of problem space evolutions, this means that the parameter pages are passed to the evolution adapters, which then convert the contained values to parameters for their evolution. For example, this mechanism is used to pass a new name for a duplicated feature entered by a user to the Duplicate Feature evolution.

It is worth noting that the parameter pages for problem space evolutions are provided in the package org.featuremapper.evolution.evolutions.problemspace.parameterpages. This package is part of the evolution subsystem rather than the UI system even though it clearly contains classes that might justify placing it in the latter. However, it was decided that the ties to the evolutions are much stronger than those to the rest of the user interface. In consequence, evolutions and their parameter pages are perceived as a functional unit that should not be broken up. Hence, evolutions and their parameter pages are both part of the evolution subsystem.

4.2.1.2 Implementation of Evolutions in the Solution Space

Evolutions in the solution space have to be able to target many different types of models. Within this thesis, UML diagrams, Java source code and DocBooklet documents are used as an example. However, the approach could be extended to target arbitrary Ecore-based models. To make evolutions widely applicable, it seems prudent to exploit the generic modification mechanism of Refactory. For one, this can be done by specifying the steps for the evolution directly in a refactoring specification and then adding remapping support where necessary. Furthermore, there is also the option to perform the changes of an evolution and the subsequent remapping directly in Java code. This option has the benefits that more complex modifications can be performed and that remapping is easier as all relevant objects are directly available. Therefore, evolutions of this thesis favor the second approach. Explanations on how to implement generic refactoring specifications can be found in [Rei10] and instructions on how to make them suitable for the use in software product lines are presented in Section 4.3.2. The classes required to implement evolutions for the solution space in source code are introduced in the following paragraphs.

The base class for all source code evolutions in the solution space is AbstractSolutionSpaceEvolution. It was designed to feature largely equivalent structures as its problem space counterpart AbstractProblemSpaceEvolution. In detail, the solution space evolution uses the same five steps for evolution as in the problem space (see Listing 4.4). The functionality and purpose of these methods has already been explained in the section on the implementation of problem space evolutions. However, the solution space employs one further method for evolution called updateSolutionSpace(). It is used to populate the changes made during the evolution to the data structures held by FeatureMapper. There is one elementary reason for why this method is only required in the solution space. FeatureMapper owns the feature model in the problem space but not the realization artifacts of the solution space. This means that the memory object held for the feature tree is also the very same instance that is being evolved. Thus, the modifications are populated automatically. However, solution space models are opened in a separate editor, which holds its own memory instance of the model. Hence, when changes are
Listing 4.5: Java source code excerpt from AbstractSolutionSpaceEvolution showing the six steps required for solution space evolutions.

```java
checkPreconditions();
evolve();

if (!getIsPreviewMode()) {
    remap();
    updateSolutionSpace();

    saveMappingModel();
    updateUI();
}
```

performed on the instance of the editor, they are not automatically reflected in the instance held by FeatureMapper. Due to this fact, the two instances of the same solution space model are in a different state. This causes problems for the consistency of the product line. As remedy, the affected solution space references of FeatureMapper are refreshed directly after the evolution by unloading and then reloading the respective models. After that, the instances of the solution space model held by FeatureMapper and the respective editor are synchronized again. The six steps for solution space evolutions are shown in Listing 4.5 as a Java source code fragment.

The majority of the concrete solution space evolutions builds upon this structure and uses AbstractSolutionSpaceEvolution as base class. However, there are also a number of specializations of the class that represent generic solution space evolutions in source code that can be applied to a particular model merely by providing an adequate role mapping. An overview of the generic solution space evolutions is provided in Figure 4.6.

Figure 4.6: UML diagram of the classes for generic evolutions in the solution space.
4.2. IMPLEMENTATION

The class AbstractMoveXTowardsEvolution is the base class of both MoveXTowardsBackEvolution and MoveXTowardsFrontEvolution. The concrete classes implement evolutions to move an element in a list of references one step to the front or to the back. An application of this evolution can be found for the DocBooklet format where it is used to move elements in the document up and down (see Section 4.2.1.2.3). The class DuplicateXEvolution provides generic functionality to clone an arbitrary element of a solution space model. The evolution allows extensions to respect individual characteristics of a particular target model. For instance, it is used as basis to copy a UML class and extended to also duplicate dependent associations in the implementation of the evolution Duplicate Class for UML (see Section 4.2.1.2.1). The evolution described by the class ExtractXEvolution performs the steps of creating a new container and then moving elements from an already existing container to the newly created one. This principle procedure may be reused in a number of different evolutions. In the concrete case, it is used as basis for the DocBooklet evolutions used to extract chapters and sections (see Section 4.2.1.2.3). Finally, the class RemoveXEvolution provides functionality to delete an arbitrary element from its containing model. Again, the evolution can be extended to respect the characteristics of a particular target model. For example, it is used to implement an evolution to delete a class from its UML model. In this case, an extension was added that also deletes the dependent associations to avoid creating an inconsistent model (see Section 4.2.1.2.1). Naturally, all of these generic evolutions create appropriate remapping steps where necessary and thus can directly be used for product line modification.

After this introduction of generic solution space evolutions in source code, the following sections will explain the specifics of the evolutions for UML, Java and DocBooklet.

4.2.1.2.1 Implementation of UML Evolutions

The implementation provides classes for four of the evolutions for UML models described in Section 3.1.2.1. With the exception of Extract Super Class, each of the evolutions for UML models is implemented in source code and, thus, as specialization of the class AbstractSolutionSpaceEvolution. An overview of the classes for UML evolutions is shown in Figure 4.7.

The Duplicate Class evolution copies an existing class in its respective UML model. For this purpose, it extends the generic evolution in class DuplicateXEvolution. Furthermore, functionality is added to the concrete evolution that also removes dependent associations, which either point to or originate from the deleted class as they would render the UML model invalid if left unchanged after the class was removed. The class ReplaceMethodWithMethodObjectEvolution implements the evolution with the equivalent name presented in Section 3.1.2.1.3. As part of the modifications required to perform the evolution, a new association has to be created. For the role names at the end of the association as well as the association name itself, the naming conventions of the TopCased UML editor are implemented, which is often used in conjunction with FeatureMapper. The evolution Inline Method Object reverses the procedure of the previous evolution by destroying a method object class after relocating the contained methods to its referring class. In order for the evolution to be applicable, a suitable class structure has to be present that contains a method object class with at least one method in it and a referring class that will hold the moved methods. The preconditions of the class InlineMethodObjectEvolution check

http://www.topcased.org/
for such a structure so that the modifications of the model are only performed if they can be completed successfully. As last evolution in source code, Remove Class is implemented in the class RemoveClassEvolution. It reuses the generic RemoveXEvolution to delete the class from its containing model. Furthermore, the concrete evolution provides an extension to also remove dependent associations along with the selected class. Through this procedure, UML models are left intact after deleting a particular class.

The evolution Extract Super Class has a special role within the UML evolutions. It is the only evolution that is implemented entirely using the generic mechanism of Refactory in its refactoring specification. However, this yields the problem that the interspatial extract procedure does not have support for remapping as would be required to maintain consistency of the product line. As remedy, the evolution is adapted for the use in product lines. For this purpose, the steps required for remapping are described in a declarative format operating on role, which is then assigned to the generic evolution in the refactoring specification. Details on this procedure can be found in Section 4.2.2.2.2. Using this approach, generic evolutions of Refactory can be upgraded to make them useable within software product lines.

4.2.1.2.2 Implementation of Java Evolutions

The majority of the evolutions for the Java programming language is implemented using Refactory’s generic refactoring specification or one of the presented generic evolutions. In fact, there is only one subclass of AbstractSolutionSpaceEvolution for Java. In Figure 4.8, this class is shown in a UML diagram.

The evolution Insert Parameter is implemented in source code. In its evolve() method, it first creates a new parameter of a specified type. Then, it assigns a given name to the parameter and adds it to the signature of the selected method. The values for the parameter type and name are gathered from a configuration page added to the evolution wizard. On the page, users can
4.2. IMPLEMENTATION

Figure 4.8: UML diagram of the classes for evolutions of Java source code in the solution space.

AbstractSolutionSpaceEvolution
InsertParameterEvolution

4.2. IMPLEMENTATION

Figure 4.8: UML diagram of the classes for evolutions of Java source code in the solution space.

type a name and choose the parameter type out of a list of options offering int, boolean and double. This list can easily be extended if the need arises. The chosen values are then reflected in the modified Java source code. The evolution Remove Statement is created by mapping the generic evolution RemoveXEvolution to the metamodel of Java. As no specific steps are required when deleting a statement, the evolution and its subsequent remapping can be used in their original form.

The remaining two evolutions for Java employ the constructs used in the refactoring specification of Refactory to perform their respective changes on the source code. First of all, Rename Element uses steps to change the name of a particular construct in Java such as a method or an attribute. Second, the evolution Extract Method maps the generic ExtractX of Refactory (not the evolution) to the metamodel used for Java. The latter of the two evolutions is interspatial in nature and thus requires remapping. However, Refactory has no native means for performing a remapping procedure. Thus, the evolution is once again adapted to the use in product lines by adding remapping support through the use of a declarative remapping language. The details of this procedure will be covered in Section 4.2.2.2.2. With the provided implementation, it is possible to perform the evolutions of Section 3.1.2.2 on their respective elements of Java source code.

4.2.1.2.3 Implementation of DocBooklet Evolutions

There is a total of 15 evolutions for the DocBooklet documentation format. However, these evolutions can be grouped into four categories similar to the headlines of Section 3.1.2.3. The majority of the evolutions are created by directly mapping one of the presented generic evolutions. However, in the case of Extract Chapter and Extract Section, specific extensions of the generic ExtractXEvolution are required. Therefore, concrete subclasses are created, which can be seen in Figure 4.9. For the Extract Chapter evolution, both sections and paragraphs may be selected. For the Extract Section evolution, only paragraphs are permissible. In both cases, the containing element of the selection has to be a chapter or a section respectively. The specialized classes ensure that the evolutions can only be triggered if these preconditions are met so that all modifications can be completed successfully.

The remaining evolutions each use one of the presented generic evolutions and their respective remapping operations where necessary. Thus, no further details on the specifics of an implementation have to be provided apart from stating which generic evolutions are employed. The evolutions to duplicate a chapter, section or paragraph all map the generic DuplicateXEvolution to the metamodel of DocBooklet. The evolutions to move chapters, sections or paragraphs up or down in a document use the generic MoveXTowardsFrontEvolution and MoveXTowardsBackEvolution respectively. Finally, the evolutions to delete a chapter, section or paragraph
each employ the generic RemoveXEvolution. By using the generic evolutions, the modifications for DocBooklet can be created swiftly and with automatic support for remapping to keep the mapping model of a product line consistent.

4.2.2 Implementation of the Remapping System

The classes central to the implementation of the remapping operations described in the theoretical section reside in the package org.featuremapper.evolution.remapping. Prior to performing a remapping in FeatureMapper, the respective mapping model has to be loaded. After that, a number of classes is involved in executing the remapping. An overview of the basic classes relevant to the implementation for remapping is provided in Figure 4.10, which serves as a reference for further explanations.

A single remapping operator from Section 3.3 is represented by what the implementation calls a remapping step. Common to both problem and solution space is the class AbstractRemappingStep, which serves as basis for the specializations for both spaces. The subclasses in either space are presented in their respective subsections. As already explained, the individual remapping steps can be composed to create more complex remappings. The implementation reflects this in the form of the so called remapping plan. A remapping plan consists of one to many remapping steps that all have to be performed during the course of a remapping. A
4.2. IMPLEMENTATION

A common implementation of the remapping plan is provided for both spaces in the form of the class `RemappingPlan`.

A remapping plan merely serves as container for multiple remapping steps and provides an interface to execute all steps sequentially. It is not possible to deselect particular remapping steps from execution, because this would bear the potential of breaking the consistency of the mapping model if an inappropriate combination of remapping steps is chosen. Thus, the responsibility for finding an adequate combination of remapping steps to maintain the consistency of the mapping model would effectively be delegated to the user of an evolution, which is in contrast to the very basic goals of this thesis. However, users may still decide to perform no remapping at all if they want to perform changes to the mapping model manually. Therefore, remapping plans can either be executed in total or not at all.

Remapping steps perform two tasks. First, they check whether all preconditions are met so that the remapping step can be applied. A precondition for all steps is that the source element for the remapping has to be provided. For remapping steps that also have a target element, an analog restriction exists. Furthermore, elements that are used as source of a remapping step must not be the target of the step at the same time. The behavior in this case would be undefined so that such a scenario is prohibited. After checking the preconditions, remapping steps perform the actual changes to the mapping model as second task. The particular nature of these changes depends on the implemented operator and is thus delegated to the concrete subclasses of `AbstractRemappingStep`.

The second part of a remapping step makes modifications to the mapping model. As there is always a connection to previously existing mappings, this can be perceived as a form of model transformation. In consequence, it was considered to employ a model to model (M2M) transformation engine to perform the changes on the feature model for remapping. However, the idea was dismissed because the changes required for remapping are rather elementary when compared to the capabilities of an M2M engine. Thus, an M2M engine would introduce a large overhead in functionality. Furthermore, using an external model transformation engine would create an additional dependency for FeatureMapper. This is another drawback of using an M2M engine for the remapping. Thus, it was decided to manually adapt the mapping model where necessary.

Unfortunately, creating a remapping plan can not be automated as its individual steps depend on the semantics of the changes performed during an evolution. For example, the evolutions `Insert Feature` and `Duplicate Feature` from Section 3.1.1.2 and Section 3.1.1.3 both add a new feature to the feature tree. In the first case, the feature is completely new and thus does not require remapping. However, in the second case, the feature is created as clone of an already existing feature, which means that a potentially existing mapping would have to be copied to the new feature during remapping. Thus, an understanding of the semantics of an evolution is required in order to create a remapping plan. Therefore, the implementation provides means of specifying a remapping plan manually. In either space, objects of the classes `AbstractFeatureRemappingStep` or `AbstractObjectRemappingStep` can be added to an element of type `RemappingPlan` directly via source code. In addition, the solution space allows composition of individual remapping operators by means of a declarative textual language. This concept will be explained in more detail in Section 4.2.2.2.
4.2.2.1 Implementation of Remapping in the Problem Space

The remapping steps in the problem space have to be able to remap all types of mappings presented in Figure 2.4. Element mappings and color mappings are treated directly and largely analog. Even though the first type maps to a solution space element whereas the second type maps to a color, both mappings have the same problem space end in the form of a single feature expression. Remapping element mappings and color mappings is thus identical for all operators except for Merge Feature Mapping, which was explained in Section 3.3.1.5 and requires special care when merging colors (see below). However, property value mappings as third type of mapping are a special case that has to be treated differently. A single property value mapping has multiple property values, which each have their own feature expression as problem space end of a mapping. Hence, remapping a property value mapping means changing a subset of all the feature expressions of its property values. This representation seems particularly intricate when tracking user choices on whether a specific property value has to be remapped or not. As a consequence, remapping operators in the problem space are provided with a view on property value mappings that is more suitable for remapping. For this purpose, the class SinglePropertyValueMapping is introduced, which encapsulates one individual property value and maintains the relation to its containing property value mapping. The interface of SinglePropertyValueMapping allows it to treat property values largely equivalent to element or color mappings. The common functionality is captured by the marker interface IFeatureRemappable, which is implemented by all classes that might be affected directly by a remapping operation in the problem space. In Figure 4.11, the relevant classes for remapping are shown in a UML diagram.

Figure 4.11: UML diagram of the classes affected by remapping in the problem space.

The remapping steps introduced in Section 3.3 are implemented as subclasses of the abstract class AbstractFeatureRemappingStep. It provides basic functionality for remapping steps in the problem space such as letting programmers decide whether users of a remapping may exclude individual IFeatureRemappable objects from the remapping. For example, this could be useful for the Copy Feature Mapping step when users are given the liberty to copy only a subset of all mappings.

Each of the remapping steps in the problem space has the potential to modify the feature
4.2. IMPLEMENTATION

expression of an IFeatureRemappable. In Section 3.3.1, it was explained that altering feature expressions might create redundancies. To alleviate this problem, the class TermSimplifier was introduced. It contains functionality to structurally simplify a given logical expression without changing its semantics. Furthermore, the concrete phrasing of the term is preserved as far as possible so that users are likely to recognize the term even after it was simplified. In detail, two basic operations are performed in the TermSimplifier. First, terms are contracted where possible. For one, this means that two double negations are replaced by the respective positive term. The second contraction appears with AND and OR operators that are nested within similar n-ary operators. For example, an AND statement that is the operand of another AND statement may just as well provide its operands directly. In the second step of simplification, duplicate operands to n-ary operators are removed. For example, an AND statement might have a reference to the same feature as operand twice in the same polarity. In this case, it suffices to only keep one of the two parameters and remove the other. The term simplifier is used in all remapping steps for the problem space after a feature expression was modified to prevent redundancies as far as possible.

In total, there are five concrete subclasses of AbstractFeatureRemappingStep equivalent to the five remapping operators. However, move, copy and split feature mapping have another abstract class as intermediate parent. It is called OneToManyFeatureRemappingStep and captures the commonalities of remapping operators that allow one source but multiple target features. In particular, programmers can choose whether the selection of subsets of all potential target features is allowed for a remapping step or not. For example, users could then choose to copy a feature mapping to only two of the three features that were provided as targets of the remapping step. In either case, at least one target has to be selected for each IFeatureRemappable element processed by the remapping step. The classes relevant for remapping steps in the problem space are summarized in Figure 4.12.

Figure 4.12: UML diagram of the feature remapping steps used for remapping in the problem space.
The class \texttt{MoveFeatureMappingStep} relocates a feature mapping from one source to potentially many target features. The procedure is performed by replacing all references to the source feature. In the case that only one target is provided, the reference to this feature serves as substitute. If multiple features are used as targets, the references to them are connected via an OR operator before employing them as replacement. The new feature expression then creates the logical move for a given \texttt{IFeatureRemappable} object.

In the class \texttt{CopyFeatureMapping}, the logical copy operation described in Section 3.3.1.2 is performed. This means, that the original feature expression of a processed \texttt{IFeatureRemapapble} object is duplicated for all target features while the original term is maintained. For each target, the references to the source feature are replaced with references to the current target feature in one of the copies. The partial feature expressions are then connected using the OR operator to create a complex term representing a logical equivalent to multiple individual expressions. A special case when copying feature mappings is the color mapping. As it is actually possible to have multiple color mappings referencing the same color, no synthetic logical copy operation is required. Instead, physical copies of a color mapping can be created for all target features.

The \texttt{Split Feature Mapping} operator is implemented in the class \texttt{SplitFeatureMappingStep}. An \texttt{IFeatureRemappable} is distributed to multiple targets by replacing the references to the source feature with a complex term of all target features. It depends on the choice of the user, which operator is employed to connect the references to the target features. Possible choices are the AND operator and the OR operator. In the case that only one target feature is provided, this choice is irrelevant as the reference to the source feature is merely substituted for a reference to the target feature without the need for concatenation. In this case, the result is equivalent to that of the \texttt{Move Feature Mapping} operator.

The class \texttt{MergeFeatureMappingStep} provides the means to meld the \texttt{IFeatureRemappable} objects of one to many source features with those of a target feature. In its most basic form, the step replaces all occurrences of the source feature with those of the target feature. Due to the use of the \texttt{TermSimplifier}, which contracts nested operators of the same type and removes duplicate operands, this procedure suffices to implement the behavior presented in Section 3.3.1.5 for the general case. However, there are a number of special cases that have to be treated. For one, merging different color mappings is problematic. Without special care, two color mappings with the same term but different color could be created, which poses a contradiction. One possibility of dealing with this situation would be to create a blend of all conflicting colors. However, the resulting tones would probably be hard to distinguish as they might get very dark. Furthermore, it is likely that a user chose the original color set with care. Mixing colors would probably destroy the mental model of the user and thus should not be performed. In fact, it suffices to use any one of the provided colors for the merged feature expression to maintain this model. Thus, the first color encountered in this situation is chosen automatically while all others are discarded. The second problematic case when merging feature mappings arises with instances of the class \texttt{SinglePropertyValueMapping}. When merging property values, it is possible that their feature expressions become equivalent through the merge. This causes problems if two property values belong to the same property value mapping. In this case, two property values with equivalent term but different value would exist. This scenario poses a contradiction that can not be resolved automatically because an understanding of the semantics
is required to determine appropriate measures for dealing with the situation. Therefore, the remapping operation (along with its triggering evolution) is canceled at this point to inform users and to prompt them to resolve the problem manually before attempting to merge again.

Finally, the class `RemoveFeatureMappingStep` contains the functionality to delete the mapping from its input features. Physically, the operation is performed by removing the references to each feature of the source list from the term of every `IFeatureRemappable` that is processed. Through this modification, it may happen that n-ary operators with too few operands are created. For example, an AND expression with only one reference to a feature could be created if all the other operands have been removed. In this case, the owning operator is deleted and its operands are moved to the enclosing operator. An equivalent procedure is used for NOT statements that had their operand removed. The term resulting from the removal is then used as new feature expression. However, the procedure might create an empty term, if the original feature expression only consisted of references to features from the source list of the remapping step. This means that there no longer is an end in the problem space for the given `IFeatureRemappable`. In this case, the `IFeatureRemappable` is removed from the mapping model altogether.

Summing up, the five different remapping steps presented in this section implement the behavior of all remapping operators for the problem space described in Section 3.3.1. With the help of the class `TermSimplifier`, redundancies in the feature expressions resulting from the remapping operators are reduced while maintaining the original phrasing of a term as far as possible. Through this, mappings in the problem space can be modified in accordance with the executed evolution and the resulting term can still be recognized by the user who originally created it.

### 4.2.2.2 Implementation of Remapping in the Solution Space

In the solution space, there are two distinct mechanisms for performing a remapping for a particular evolution. The first option is to assemble a remapping plan in source code similar to what is done in the problem space. The second option is to use a declarative specification of the required remapping operations and assign it to a particular generic refactoring by using the extension point `org.featuremapper.evolution.objectremapping` (see Section 4.3.2). The first method is preferred when an evolution was specified in source code, the second one if a generic refactoring was specified with the help of a *.refspec file. These two approaches ensure a consistent specification language as either only Java or only declarative languages are used to define an evolution and its remapping. However, it is also possible to mix both approaches. For instance, a declarative remapping specification can be attached to an evolution specified in source code. Internally, both these approaches use the same classes to realize the remapping. The following sections will first explain the procedure of specifying a remapping explicitly using source code before introducing the details of the declarative remapping specification.

#### 4.2.2.2.1 Explicit Remapping in the Solution Space

Remapping in the solution space potentially affects element mappings and property value mappings as they both target a particular `EObject` of a solution space model. As all property values of a property value mapping reference the same `EObject`, it is possible to remap the entire prop-
 CHAPTER 4. A FRAMEWORK FOR EVOLUTIONS IN FEATUREMAPPER

Property value mapping directly without using a specialized view as the one in the problem space. Color mappings do not have to be considered for remapping as they map to a color and not an assets of the solution space. Due to these characteristics, the marker interface IFeatureRemappable is of no use in the solution space. Instead, the class Mapping can be used as sole target of remapping as it is the base for the affected classes ElementMapping and PropertyValueMapping. Retargeting a mapping in the solution space generally means exchanging the referenced EObject of a mapping. However, there are still cases when the feature expression of a mapping has to be adapted similar to the procedure in the problem space.

The individual remapping operators for the solution space presented in Section 3.3.2 are implemented as so called remapping steps analog to the problem space side. However, they have different names to signal their usage in the opposite space. Even though the solution space end of a mapping is called an element in FeatureMapper, it was refrained from introducing the class name “ElementRemappingStep” as it might easily mislead users into thinking that only element mappings are affected. However, property value mappings are also modified during the remapping procedure in the solution space so that a different name was required. As the solution space element is of type EObject, the name ObjectRemappingStep was established to avoid misleading programmers and readers. Thus, the remapping operators for the solution space are implemented as subclasses of the abstract class ObjectRemappingStep. An overview of the class and its descendants is shown in Figure 4.13 as UML diagram.

Figure 4.13: UML diagram of the object remapping steps used for remapping in the solution space.

Each of the concrete subclasses of ObjectRemappingStep contains the functionality to implement one of the remapping operators introduced in Section 3.3.2. However, the move and copy operators have an intermediate base class called OneToManyObjectRemappingStep. Analog to the problem space, this class provides functionality for operators that potentially have many targets. For example, a mapping can be copied from one solution space element to many others. In such a case, OneToManyObjectRemappingStep provides the possibility to allow deselection of certain potential targets so that the copy operation is only performed for a subset of all targets.

The class MoveObjectMappingStep implements the move operation for mappings in the solution space. Physically, the move operation is performed by three separate steps.
the original mapping is copied n times according to the number of target elements. This step
uses the same mechanism as \textit{Copy Object Mapping} described in the following paragraph. In
the second step, the elements of the copied mappings are set to the respective target objects.
Finally, the original mapping is removed from the mapping model.

The copy operation in the class \texttt{CopyObjectMappingStep} works in a largely equivalent way
as the move operation. However, it does not delete the original mapping but merely copies it for
all target elements. In the solution space, it is possible to perform a physical copy unlike in the
problem space where creating a logical copy by manipulating the feature expression is required.
However, the constraint of FeatureMapper of only one mapping of a type for a solution space
element still holds. Thus, a physical copy is only possible if the target element is not already
part of a mapping. Should a mapping already exist, the feature expressions of both the existing
and the copied mapping have to be concatenated using the OR operator. For element mappings,
this procedure is trivial as there is only one feature expression. In property value mappings, each
contained property value has its own feature expression. However, not every property value in
the existing mapping has a counterpart in the copied mapping. Thus, equivalent property values
in the existing and copied mapping have to be determined. This is done by comparing the value
they map to. If two property values are found to be equivalent, their terms are concatenated
using the OR operator. Otherwise, no modifications of the feature expression are required to
perform the \textit{Copy Object Mapping} operation.

Finally, the last object remapping step is implemented in the class \texttt{RemoveObjectMapping-
Step}. It provides functionality to delete an object mapping from the mapping model. In the
solution space, it is possible to physically delete the mapping unlike in the problem space where
the feature expression has to be modified to logically remove a feature. The reason for this is
that the solution space end of a mapping can only reference exactly one realization element.
Thus, if the mapping to this element is to be removed, the entire entry in the mapping model
has to be deleted in all cases. If multiple mappings have to be removed from the mapping model,
they are deleted one-by-one to utilize the listener mechanism of FeatureMapper to appropriately
update the user interface.

It is worth noting that the remapping steps in the solution space depend on a data structure
called the URI map. It is a mapping from a URI to the respective object it identifies. The
described objects are the elements partaking in a remapping procedure. The URI map is created
as early in the evolution process as possible. For evolutions in source code, this means that it
is assembled before the first modification of a model was performed. For declaratively specified
evolutions, it is created directly after the refactoring specification was executed. In either case,
the URI map is filled with the objects playing the roles specified in the role model of Refactory
for that particular evolution. The URI map is then used to find all mappings that are affected by
the remapping procedure. The reason for using a saved version of the URIs instead of calculating
them as needed is that evolutions may delete particular objects from a solution space model.
In this case, the URI of the respective element can no longer be established and thus the mappings
the object participates in can not be located effectively making remapping impossible. Creating
the URI map before performing the actual evolution ensures that all URIs can be retrieved
correctly and remapping can commence. Creating the URI map and then propagating it to
the respective remapping steps contained in a remapping plan is handled internally. Therefore,
creators of a remapping do not need to concern themselves with these details as the procedure is performed transparently. With the three provided remapping steps, it is possible to build a remapping plan in source code that is used to maintain consistency of a product line after an interspatial evolution was performed.

4.2.2.2.2 Declarative Remapping in the Solution Space

With the explicit remapping in source code of the previous section, consistency of a product line can be maintained. However, there is an alternative to creating the remapping plan explicitly. When using the generic evolution mechanism of Refactory, a declarative specification of the modifications performed during an evolution is employed (see Section 4.1.1.2). In this case, it seems prudent to provide an equally declarative description of the subsequent remapping steps. For this purpose, the so called object remapping specification (ORSpec) is provided as part of the thesis.

All problem space evolutions are defined explicitly using source code. Thus, the declarative ORSpec is used exclusively for remapping in the solution space. Furthermore, the ORSpec should be employed for evolutions that specify their steps of modification in the refactoring specification of Refactory. Evolutions that use source code for the modifications should preferably use an explicit remapping mechanism. In Section 4.3.1, the benefits of creating new evolutions using source code will be explained. However, it is also possible to adapt existing solution space evolutions in the refactoring specification for the use in software product lines (see Section 4.3.2). In this scenario, the ORSpec is employed.

As the EObject at the solution space end of a mapping is referred to as “element” by FeatureMapper, the name “Element Remapping Specification” would have also been possible. However, the remapping specification affects both element mappings and property value mappings. The name “Element Remapping Specification” would have easily led to believe that only the prior type of mapping is affected. Thus, the name was abandoned in favor of “Object Remapping Specification” to prevent confusion.

The ORSpec consists of a textual syntax defined using EMFText that was designed to closely relate to the syntax of the refactoring specification used by Refactory. Thus, the transition between the two languages is almost seamless. Both refactoring specification and object remapping specification build on the concept of roles and use the entities defined in the role model of an evolution. The refactoring specification describes the steps that need to be carried out in order to perform a modification of a model and an ORSpec describes the remapping operations necessary to maintain the consistency of the mapping model. Due to this connection, refactoring specification and ORSpec are closely related.

Each of the three remapping steps for the solution space presented in Section 3.3.2 can be created using the object remapping specification. However, the ORSpec works on roles where the individual remapping steps require concrete EObjects as source and target. This difference creates new challenges. For example, it is possible that a single role is resolved to multiple objects during runtime. However, the move and copy object mapping steps only permit a single object as source. Therefore, trying to create a single move or copy step using a role as input would have the potential to fail. Thus, a clean separation between the concepts presented in an ORSpec operating on roles and the remapping steps working with EObjects has to be provided.
4.2. IMPLEMENTATION

For this purpose, a single entry of an ORSpec is referred to as “remapping instruction”, which is then translated to a remapping step to resolve the roles to objects during runtime. In Listing 4.6, an example of an ORSpec with the possible remapping instructions is presented.

Listing 4.6: Example of an ORSpec showing the different remapping instructions and their modifiers.

```
OBJECT REMAPPING FOR <GenericEvolutionName>

STEPS {
    COPY MAPPING: RoleA, RoleB -> RoleC, RoleD, RoleE {
        description = "Essential copy mapping";
        options = SELECTABLE_TARGETS;
    }

    MOVE MAPPING: RoleF, RoleG -> RoleH, RoleI {
        description = "Optional move mapping";
        options = OPTIONAL;
    }

    REMOVE MAPPING: RoleJ, RoleK, RoleL;
}
```

The example shows that remapping instructions are largely analog to remapping steps. As mentioned before, the main difference is that remapping steps work on objects whereas remapping instructions use roles. As a single role can potentially be translated to multiple objects during runtime, it may be necessary to create more than one remapping step for a single remapping instruction. Due to this fact, users are further provided with the option to define multiple source roles for the copy and move remapping instructions, which are then transformed into remapping steps.

Furthermore, a remapping instruction allows configuration. For one, it is possible to provide a description of the remapping instruction that will be displayed in the user interface. For example, it is possible to communicate that a particular remapping operation affects the selected object whereas another remaps its children. In addition, options may be passed to a remapping instruction. The option SELECTABLE_TARGETS enables users to select only a subset of all potential targets of a remapping instruction. For example, a mapping can be copied only to two of the potential three targets. This concept has already been explained for remapping steps. The remove mapping instruction does not specify targets and thus the option to select targets has no effects. The second configuration choice is the option OPTIONAL. It is used to compensate for a mechanism of Refactory that can be used to define a role as being optional. An optional role does not necessarily have to be mapped in the role mapping and thus may not be resolvable to an object during runtime. If this particular role would be the sole source or target parameter of a remapping instruction, the respective parameter list might remain empty. In general, this behavior is not permissible and thus the translation process of the remapping instruction to a remapping step will be canceled with an error. For remapping instructions marked as being
optional, it is permissible to have an empty parameter list. If either source or target list of a remapping instruction remain empty, the translation will be canceled because it is assumed that the respective optional role of Refactory was not mapped to a model element.

After an ORSpec has been fully specified to perform the remapping for a particular refactoring specification, it has to be registered with the evolution system using the extension point org.featuremapper.evolution.objectremapping. An extensive example of creating and registering an ORSpec is presented in Section 4.3.2. After the registration, the respective evolution gains support for remapping. When the evolution is executed, the ORSpec and its remapping instructions are translated to a remapping plan containing remapping steps. An overview of the classes involved in the translation process is presented in Figure 4.14.

Figure 4.14: Overview of the classes involved in translating and executing an ORSpec.

The most notable class in the translation process is ORSpecTranslator. It takes an object of type ObjectRemappingSpecification as input parameter, which in turn contains multiple elements of the class ObjectRemappingInstruction and its specializations. Furthermore, the translator is provided with the information of which roles resolve to which objects in the course of a particular evolution. The respective data structure is provided by Refactory. Using this knowledge, each remapping instruction is translated. First, it is checked if a single remapping step suffices for the remapping instruction. This is the case if a move or copy object mapping step with merely one source element or a remove object mapping operation is to be created. If multiple remapping steps have to be created, the translation procedure is repeated. In a second step, the description and options of a remapping instruction are translated to the respective settings of the remapping step. Finally, the remapping step is added to the remapping plan created by the translator. Hence, the result of translating a remapping specification during runtime is a remapping plan that can be executed to modify the mapping model in a way that the validity of the product line is maintained.

For a concrete evolution, the translation of an ORSpec and the execution of the resulting remapping plan is triggered by a class called SolutionSpaceRemapper. When registering an ORSpec for a particular refactoring specification and thus a generic evolution, all derived concrete evolutions have to be provided with remapping support. Internally, each of the affected evolutions has the class SolutionSpaceRemapperAdapter registered as post processor that will automatically be triggered once the evolution is executed. The adapter creates and configures...
4.2. IMPLEMENTATION

an object of the class SolutionSpaceRemapper, which is then executed. Thus, the remapping can be performed at the usual place in the control flow. However, it is possible that a concrete evolution already defines a post processor. As there may be at most one post processor for each concrete evolution, this seems problematic with respect to registering the SolutionSpaceRemapper. However, the remapper is designed to act as proxy to existing post processors. The previously existing post processor of a concrete evolution is automatically passed to the remapper and will be executed prior to performing the remapping. Thus, the regular control flow is maintained and the evolution virtually executes two post processors at once. It is further ensured that the object remapping specification and thus the remapper are registered after all evolutions with their respective post processors were loaded. Therefore, it can not happen that the SolutionSpaceRemapper is registered before the original post processor which would effectively override the remapper. Instead, the registration that holds all evolutions is fully built before the first remapper is registered so that the order of registration is maintained in all cases.

Naturally, an object remapping specification is only required if the described evolution is interspatial in nature as intraspatial evolutions do not require remapping. Thus, it depends on the intent of the performed modifications whether a remapping specification has to be assigned to a particular evolution. Due to registering an ORSpec by using an extension point of the evolution system, the underlying structure of Refactory did not have to be modified to incorporate the declarative remapping language. In consequence, the main branch of development of Refactory can be used for the evolution system.

4.2.3 Implementation of the User Interface System

The user interface subsystem is responsible for creating the dialogs used to configure remapping plans and the individual remapping steps contained in them. Due to the diverse nature and varying configuration options of remapping steps for the problem and solution space, a number of different user interfaces has to be provided. For example, a remapping step may allow to exclude particular remappable artifacts from the remapping procedure. This means, that users have to be given the choice for each remappable element if it should be processed or not. Hence, the user interface for such a remapping step requires additional controls and functionality as compared to the user interface for a remapping step that prohibits individual exclusion. Due to this diversity, the assembly of user interface elements is delegated to builders [GHJV95], which encapsulate the logic to create an appropriate user interface. All classes of the user interface subsystem are located beneath the main package org.featuremapper.evolution.ui. The essential classes of this package are displayed in Figure 4.15.

In Figure 4.15, the classes for the problem space are shown on the left side and the respective counterpart for the solution space on the right side. The central region of the figure displays the common base class for the builders. The class RemappingDialog provides the general structure and layout of all remapping dialogs. In particular, this means that each dialog features a check box that allows to disable processing of the entire remapping plan. Furthermore, the dialog contains a series of buttons to commit or cancel remapping located beneath the area to configure a remapping plan and its steps.

The control to configure a remapping plan is assembled by the class RemappingPlanUI-Builder. It creates a so called ExpandBar, which is capable of holding multiple objects of the
class **ExpandItem**. An **ExpandItem** consists of a title area with the item’s name and a configuration area containing arbitrary controls. Each of the objects of class **ExpandItem** reflects the configuration options of a single remapping step of the remapping plan. Building these controls is again delegated to a builder. There are specialized classes of **AbstractRemappingStepUIBuilder** for the problem and solution space that create a control displaying all valid configuration options for a concrete remapping step. The regions of a remapping dialog created by the class **RemappingPlanUIBuilder** and the specializations of **AbstractRemappingStepUIBuilder** are visualized in Figure 4.16 using a screenshot.

In the problem space, additional specializations of the class **FeatureRemappingStepUIBuilder** exist. The class **OneToManyFeatureRemappingStepUIBuilder** creates the controls for remapping steps inheriting from the abstract class **OneToManyFeatureRemappingStep**. In particular, these are the move, copy and split feature mapping steps. All of these operations have in common that they potentially have multiple targets. It is possible to configure these steps to allow selection of only a subset of all targets to be used as effective targets for individual remappable artifacts. For example, a feature mapping might only be copied to two of the possible three target features. To allow setting the parameters for which effective targets are to be used, the class **OneToManyFeatureRemappingStepUIBuilder** is employed to assemble a specialized user interface for these steps. Yet, the **Split Feature Mapping** step may still require an even more dedicated control. In order to split a feature mapping, it is possible to concatenate the effective targets of a remappable artifact by either the OR or the AND operator. To let users make this decision, a dialog element representing this choice has to be added to the user interface. The class **SplitFeatureMappingStepUIBuilder** is adapted to this requirement and thus is capable of creating an appropriate user interface for the **Split Feature Mapping** step.

With the structuring into builders, it is possible to provide specialized versions of the user interface while still reusing the majority of the existing source code to create dialog elements. Through this, maintenance of the respective classes is eased as the single point of truth principle is obeyed. Furthermore, the visual appearance of the remapping dialogs is consistent throughout the application for all remapping steps. Therefore, users of the system encounter familiar means of entering configuration parameters, which makes it easy to use the remapping system.
4.2. IMPLEMENTATION

Figure 4.16: Visualization of the regions of a remapping dialog created by the specializations of AbstractRemappingPlanUIBuilder and AbstractRemappingStepUIBuilder.

4.2.4 Implementation of the Test Suite

Along with the realization of the evolution system, an extensive test suite is provided to demonstrate the validity of the implementation. Evolutions and remapping operators are tested individually as well as in conjunction in complex scenarios. The test cases for evolutions are further divided for the problem and solution space respectively. The test cases for remapping operators are split into the groups of feature and object remapping steps. Evolutions as well as remapping steps are tested directly without invoking Refactory as it seems infeasible to properly parametrize the tool for testing.

The general strategy for testing evolutions and remapping operators is to provide a set of predefined input and expected output models. At the very least, the feature tree, the mapping and one solution space model have to provided. Optionally, multiple solution space models can be used for testing. During a test case, the input models are modified and the actual result is compared to the expected output. If all compared pairs of models match, the test case is considered successful.

The implementation of the test suite employs a number of classes to reflect this general procedure and to adapt it to the specific requirements of individual evolution and remapping test cases. The main package for the test suite is org.featuremapper.evolution.test, which
CHAPTER 4. A FRAMEWORK FOR EVOLUTIONS IN FEATUREMAPPER

contains the classes in Figure 4.17.

Figure 4.17: UML diagram of the classes used for test cases.

The class AbstractTestCase is the basis for all specialized test cases. It handles the setup of a test case consisting of loading the relevant models, modifying them and comparing the results. The alteration of a model is implemented as abstract method so that specializing classes can decide on which modifications have to be performed for their particular test case. For AbstractEvolutionTestCase, the modifications are performed by evolutions. In the case of AbstractRemappingTestCase, remapping operators are employed to alter the input models. Specialized test cases can also decide on which solution space models to employ. For example, test cases for the evolution of UML models might not require a DocBooklet model in the solution space for the test cases to be performed. Furthermore, it is possible for specialized classes to perform additional setup steps. For example, the class AbstractSolutionSpaceTestCase allows manual creation of the URI map in order to simulate the integration into Refactory. Using these classes, over 100 test cases for evolutions and remapping steps were realized, which are used to validate the implementation of the evolution system.
4.3 Possibilities for Extension

During the course of this thesis, a number of diverse evolutions for the problem as well as the solution space have been presented. With these evolutions, a basis for modifying a software product line while maintaining its mapping has been established. However, most likely the need for additional evolutions will arise eventually. In order to make it easy to extend the existing evolution system, the following sections will explain the steps required to add a new evolution to the system. Adding evolutions in the problem and solution space requires a slightly different procedure. Yet, the basic steps are very similar. Especially registering artifacts required by Refactory is equivalent for both spaces.

In Section 4.1.1, four different file formats used by Refactory were introduced. The role model specifies which roles exist in a generic evolution and how they relate to one another. A refactoring specification describes the steps required to perform a particular modification and the role mapping binds the defined roles to concrete entities of a metamodel. The fourth artifact employed by Refactory is the so-called post processor. It is a regular Java class used to perform further modifications. Evolutions for software product lines largely depend on the functionality of the post processor. The main benefit is that programming in Java allows to create more sophisticated evolutions than with the domain specific language used by Refactory. Furthermore, performing a remapping is easier because potential source and target elements of a remapping step can be accessed directly via source code. Therefore, it is the preferred to create new evolutions in Java source code. However, it is further possible to adapt existing interspatial evolutions to employ remapping. Both of these approaches will be explained in the following sections.

4.3.1 Adding New Evolutions

When using an evolution in Java source code, the generic refactoring mechanism of Refactory is merely used to execute the evolution on a suitable selection after triggering it from the context menu. For this purpose, it suffices to create a minimalistic abstract refactoring. First, a role model with merely one role is created. Then, a refactoring specification loading the objects for this role has to be composed. Even though the objects in the refactoring specification are not used themselves, loading them is required for Refactory to located objects for the respective roles which can then be used by the post processor. Otherwise, an evolution would be passed an empty input list. In Listing 4.7 and Listing 4.8 both created artifacts are shown.

Listing 4.7: Example of a minimalistic role model.

```java
RoleModel SimpleEvolution {
    input super ROLE Element;
}
```

The created role model and refactoring specification then have to be made available for use by Refactory by binding the extension points `org.emftext.refactoring.rolemodel` and `org.emftext.refactoring.refspec` respectively. This concludes creating the generic part of an evolution. The following steps depend on the concrete model that should be evolved. The
Listing 4.8: Example of a minimalistic refactoring specification.

```plaintext
REFACTORING FOR <SimpleEvolution>

STEPS {
    /** Dummy */
    object element := Element from filter(INPUT);
}
```

explanations will use the implementation of the Duplicate Feature evolution as example of how to create a problem space evolution. An analog procedure can be used for solution space evolutions. The last paragraphs of this section will explain how to modify the procedure in order to create a solution space evolution instead.

As the feature tree is the only type of model in the problem space, genericity is not required for problem space evolutions. Instead, the specific knowledge of the characteristics of a the feature tree can be used to create sophisticated evolutions. First, the generic basis consisting of role model and refactoring specification has to be bound to an element of a metamodel. In the case of Duplicate Feature, that is the class Feature of the feature model. For this purpose, a file called “DuplicateFeature.rolemapping” is created with the content presented in Listing 4.9.

Listing 4.9: Example of a role mapping for Duplicate Feature.

```plaintext
ROLEMODELMAPIING FOR <http://www.tudresden.de/feature>

DuplicateFeature maps <SimpleEvolution> {
    Element := Feature;
}
```

After creating the file, it has to be registered using the extension point org.emftext.refactoring.rolemapping. Finally, the real evolution has to be implemented. In Section 4.2.1, it was explained that a design decision has been made to cleanly separate the implementation of the evolution system of this thesis from Refactory. However, both systems cooperate in order to perform modifications to models and maintain the mapping of a product line. In consequence, the steps of an evolution are implemented in one class and the connection between Refactory and the evolution is established using another class.

Evolutions are implemented as subclasses of the abstract class AbstractEvolution. For the case of problem space evolutions, a skeleton implementation exists in the class AbstractProblemSpaceEvolution. These classes are self sufficient and can be executed without being integrated into the control flow of Refactory. Connecting the evolution with Refactory is done by deriving a class from AbstractEvolutionAdapter. The sole purpose of the adapter is to translate the objects playing particular roles in Refactory to suitable input parameters to create and configure an evolution. The principle structure of an EvolutionAdapter for the Duplicate Feature evolution is shown in Listing 4.10. The details on how to acquire values for the parameters of an evolution will be explained later.
4.3. POSSIBILITIES FOR EXTENSION

Listing 4.10: Principle structure of the implementation of the evolution adapter for *Duplicate Feature*.

```java
public class DuplicateFeatureEvolutionAdapter extends AbstractProblemSpaceEvolutionAdapter<
    DuplicateFeatureEvolution>
{

    @Override
    protected DuplicateFeatureEvolution createAndConfigureEvolution() throws
    AcquireParametersException {

        Feature featureToDuplicate = ...;
        String duplicateFeatureName = ...;
        boolean duplicateChildren = ...;

        return new DuplicateFeatureEvolution(getRefactoredModel(),
            getResourceSet(), getIsFakeRun(), getUriMap(), featureToDuplicate,
            duplicateFeatureName, duplicateChildren);
    }
}
```

An evolution adapter then has to be registered as post processor to the created concrete evolution in the role mapping via the extension point `org.emftext.refactoring.postprocessor`. To identify a particular evolution, its name and the URI of the target meta model have to be provided along with the post processor. Once the adapter is registered as post processor, it is executed when the appropriate menu entry for the evolution was selected. After that, the real evolution in source code is instantiated and executed.

A problem space evolution provides methods for the principle procedure of performing an evolution on the feature tree in the form of a template method [GHJV95]. The mandatory part, which has to be implemented, is the method `evolve()` that is intended to perform the actual modification of the input model. Before the `evolve()` method is called, preconditions that have to be met for the evolution to be applicable can optionally be checked. For this purpose, the method `checkPreconditions()` can be overwritten. These two methods suffice to implement intraspatial evolutions. However, interspatial evolutions require a subsequent remapping to maintain the consistency of the product line. For this purpose, an adequate remapping plan for the evolution has to be created. The class `AbstractEvolution` presents the method `createRemappingPlan()` that can be overwritten to add appropriate remapping steps to the remapping plan. The created plan is then executed after the modifications of the evolution as part of the control flow specified by the template method of the evolution. The principle structure of the *Duplicate Feature* evolution is shown in Listing 4.11.

With this setup, the evolution is operational. No further registrations via extension points are required, as the evolution adapter creates the evolution appropriate for the executed role mapping. However, at the current stage of development, the evolution adapter is not yet able to acquire the parameters required to configure the evolution as this part has been omitted in the explanations. There are two means for retrieving parameters for an evolution that both are
Listing 4.11: Principle structure of the implementation of the Duplicate Feature evolution.

```java
public class DuplicateFeatureEvolution extends AbstractDuplicateEvolution {
    public DuplicateFeatureEvolution(EMObject refactoredModel,
                                        ResourceSet resourceSet, boolean isPreviewMode, Map<EMObject, URI> uriMap,
                                        Feature featureToDuplicate, String duplicateFeatureName,
                                        boolean duplicateChildren) {
        super(refactoredModel, resourceSet, isPreviewMode, uriMap,
              featureToDuplicate);
        //...
    }

    @Override
    protected void checkPreconditions() throws PreconditionViolatedException {
        //...
    }

    @Override
    protected void evolve() throws EvolutionException {
        //...
    }

    @Override
    protected RemappingPlan createRemappingPlan() throws RemappingSpecificationTranslationException {
        RemappingPlan plan = super.createRemappingPlan();
        //...
        return plan;
    }
}
```

used as part of the Duplicate Feature evolution. First, objects playing a particular role of the role model can be retrieved from Refactory. The evolution adapter has dedicated methods for this purpose. For example, `getMandatoryFirstObjectForRole()` and `getMandatoryObjectsForRole()` can retrieve one object or a list of objects playing a role identified by name. For the Duplicate Feature evolution, the feature that is to be duplicated is retrieved in this fashion. As second option, it is possible to offer a configuration page for an evolution as part of the evolution wizard and then retrieve configuration parameters from it. A wizard page used for configuration should subclass `AbstractParameterPage`, which provides the basic structure of the user interface. The derived class then has to be registered with Refactory using the extension point `org.emftext.refactoring.customwizardpage`, which requires equivalent input as when registering a post processor such as the evolution adapter. In Listing 4.12, the outline of the
4.3. POSSIBILITIES FOR EXTENSION

parameter page for *Duplicate Feature* is displayed as an example.

Listing 4.12: Principle structure of the parameter page for *Duplicate Feature*.

```java
public class DuplicateFeatureParameterPage extends AbstractParameterPage {

@Override
protected Control doCreateControl(Composite parent) {

    Composite composite = new Composite(parent, SWT.NULL);

    //...

    return composite;
}
}
```

This parameter page is now displayed whenever the *Duplicate Feature* evolution is executed. The evolution adapter holds a list of the parameter pages registered for a particular evolution. Furthermore, it provides an auxiliary method called `getMandatoryWizardPage()` to retrieve a specific parameter page, which takes the class of the wizard page to fetch as parameter. Through this, it is now possible to complete the implementation of the principle version of the evolution adapter presented in Listing 4.10. The complete implementation of the evolution adapter for the *Duplicate Feature* evolution can be seen in Listing 4.13.

The parameter pages of the evolution wizard are created using the standard widget toolkit (SWT)\(^4\). It is worth noting that the evolution adapter accesses the values of these pages after the user interface components have been disposed by SWT. This means that a direct access to user interface components (such as text fields) is no longer possible at this point in the control flow. As a remedy, the values of parameters should be made attributes of the parameter page that are set whenever the value of their respective user interface control is modified. Through this, the current value of a parameter can be retrieved from a parameter page even after the user interface components have been disposed.

Even though this example presented the process of creating a problem space evolution, implementing a solution space evolution can easily be derived from the explanations. The essential difference is to not use `AbstractProblemSpaceEvolution` and `AbstractProblemSpaceEvolutionAdapter` as base classes but the respective counterparts for the solution space. Furthermore, feature remapping steps of the remapping plan are abandoned in favor of the respective object remapping steps. The rest of the procedure is analog for the solution space.

With the aid of this example, it is now possible to add new evolutions to the system. However, sometimes it might also be necessary to adapt existing declaratively specified interspatial evolutions of Refactory to perform a remapping in order to keep a product line consistent. This procedure is explained in the following section.

\(^4\)http://www.eclipse.org/swt/
CHAPTER 4. A FRAMEWORK FOR EVOLUTIONS IN FEATUREMAPPER

Listing 4.13: Complete implementation of the evolution adapter for Duplicate Feature.

```java
public class DuplicateFeatureEvolutionAdapter extends AbstractEvolutionAdapter<
    DuplicateFeatureEvolution> {

    @Override
    protected DuplicateFeatureEvolution createAndConfigureEvolution() throws 
    AcquireParametersException {

        Feature featureToDuplicate = getMandatoryFirstObjectForRole("Element", 
        Feature.class);

        DuplicateFeatureParameterPage parameterPage = getMandatoryWizardPage(
            DuplicateFeatureParameterPage.class);
        String duplicateFeatureName = parameterPage.getElementName();
        boolean duplicateChildren = parameterPage.getDuplicateChildren();

        return new DuplicateFeatureEvolution(getRefactoredModel(),
            getResourceSet(), getIsFakeRun(), getUriMap(), featureToDuplicate,
            duplicateFeatureName, duplicateChildren);
    }
}
```

4.3.2 Adapting Existing Evolutions

Refactory comes with a number of predefined modifications for different target models. Furthermore, its mechanism of specifying refactorings generically, makes it easy to create customized refactorings for a wide range of languages. Therefore, it is likely to encounter evolutions that perform a particular modification but have not been prepared to work with software product lines. As it was explained in Section 3.2.3.2, this is problematic if the evolution is interspatial in nature as it will damage the mapping of the product line if no appropriate counter measures are taken. Fortunately, it is possible to upgrade existing generic evolutions to perform a remapping if necessary. For this purpose, the object remapping specification (ORSpec) introduced in Section 4.2.2.2.2 is employed.

An ORSpec is used to add remapping support to a generic evolution defined in a refactoring specification. The remapping procedure is then applied for concrete evolutions derived from the generic evolution. In the following, ExtractXWithReferenceClass serves as an example of a generic interspatial evolution, which requires remapping. Assigning an adequate ORSpec to it causes its dependent evolutions, such as Extract Method for Java to gain remapping support as well.

As mentioned before, an ORSpec operates on roles to provide a seamless integration into Refactory and its generic refactoring mechanism. It is then possible to define remapping instructions on roles that are later transformed into remapping steps on objects. These steps are made part of a remapping plan that is executed once the modifications of an evolution have been performed. For the example of ExtractXWithReferenceClass there are a total of five roles.
First, the elements that are to be extracted are represented by the role Extractee. They are located within an element of the role OrigContainer. In the process of the modifications, an object of role NewContainer is created to hold the extracted elements. In order to make the new container a sibling of the original container, a role ContainerContainer is required, whose respective element is the parent of both the original and the new container object. Finally, the optional role MovedReference represents the link of the original container to the new container. For the concrete evolution Extract Method in Java, this would be the method call to the extracted method. With these roles, it is now possible to create an ORSpec that performs an adequate remapping.

Due to the relation of original and new container, it seems sound to extend the original mapping to the new sibling. Thus, Copy Mapping is used to duplicate the mapping of the role OrigContainer to the role NewContainer. Furthermore, the extracted elements are moved, which could easily lead one to believe that their mapping has to be moved as well. However, the objects are relocated physically and thus continue to exist as part of a different container. This is unproblematic as the mapping still applies to the same objects so that it does not have to be adapted. However, the new container should be part of a variant if at least one of its contained elements is required by the configuration. Therefore, the mapping of the extracted elements is extended to the new container as well. Copying the mapping from multiple sources to the new container results in an OR expression of the separate feature expressions so that either one of the source mappings suffices to include the new container in a variant of the product line. This same procedure is used for ExtractXWithReferenceClass as well so that it is included in all configurations where the new container is required. The described ORSpec used for ExtractXWithReferenceClass is shown in Listing 4.14.

Listing 4.14: ORSpec for the generic interspatial evolution ExtractXWithReferenceClass.

```plaintext
OBJECT REMAPPING FOR <ExtractXWithReferenceClass>

STEPS {
  COPY MAPPING: OrigContainer -> NewContainer;
  COPY MAPPING: Extractee -> NewContainer {
    description = "Extractee -> NewContainer";
  };
  COPY MAPPING: OrigContainer -> MovedReference {
    options = OPTIONAL;
  };
}
```

The created file then has to be registered with the evolution system by using the extension point org.featuremapper.evolution.objectremapping. By populating the ORSpec, the generic evolution ExtractXWithReferenceClass gains support for remapping and thus can be
used for the evolution of software product lines even though it is an interspatial evolution that originally has not been designed to respect the mapping model.

Naturally, it is also possible to implement entirely new evolutions by employing the modification capabilities of the refactoring specification and then attaching an ORSpec file to it. However, the preferred method is to use an evolution implemented in source code as presented in the previous section to overcome some of the shortcomings inherent to using the combination of refactoring specification and ORSpec. In particular, it is only possible to target objects that still exist after the modifications of the refactoring specification. Thus, objects that were deleted can not be remapped. When using the evolution in source code, this problem is solved by the URI map presented in Section 4.2.2.2.1. Therefore, new evolutions should be created in Java source code.

With these explanations on how to create new and adapt existing evolutions, it is possible to provide a wide variety of different evolutions that are capable of modifying the artifacts of a software product line while still maintaining the respective mapping model.
Chapter 5

Example Project

Up to this point of the thesis, technical as well as implementation aspects of evolutions for software product lines have been explained. What remains to be demonstrated is that the evolutions are both usable and useful in practical application. For this purpose, the following chapter will present an example of a software product line that is being modified over the course of several years. The scenario is a collection of interconnected examples that demonstrate the majority of the evolutions presented in this thesis.

The concrete use case builds upon a software product line for an automotive multimedia system that represents the configuration choices for the multimedia system of an automobile. For example, it is possible to choose which CD player should be built into a particular car or to specify whether it should have personal navigation or not. Naturally, this type of product line has extensive effects on the hardware and thus on which physical components have to be installed in the car. However, there is also an abundance of software involved. For example, it has to be determined what type of controller functionality is required and which menu options should appear on the on board display. As it is a software product line that is being demonstrated, the explanations will neglect the physical part of the problem and focus solely on the software aspects involved in configuration. In addition to the mere software part, the choice of features for a concrete variant affects further artifacts. For example, the manual of the multimedia system should be adapted in accordance with the features used in a configuration so that that only those parts of the system are described that have actually been built into the car.

The example project demonstrates a number of different models. In the problem space, the feature model of FeatureMapper is used. In the solution space, four different types of models are employed to demonstrate typical elements used in the software development process. First, UML is used as a representative of design models featuring a graphical syntax for models. Unfortunately, the integration of Refactory into the UML Editor TopCased\(^1\), which is often used in conjunction with FeatureMapper, has not been completed by the time of this writing so that evolutions on UML models have to be triggered from the standard EMF model editor instead of from their graphical representation. The second type of solution space element is Java source code, which serves as a representative of implementation artifacts. The textual source code is parsed with EMF Text creating a model representation that can be used by FeatureMapper and

\(^1\)http://www.topcased.org/
CHAPTER 5. EXAMPLE PROJECT

the evolution process. The JaMoPP\textsuperscript{2} project (Java Model Parser and Printer) allows to parse Java files into model instances using EMF Text so that full-fledged programs can be mapped and evolved. The third artifact in the solution space is the user manual for the multimedia system representing documentation formats. Its meta model has a textual representation called DocBooklet, which is a derivate of the DocBook\textsuperscript{3} format. It features an XML-like syntax that is closely related to its archetype but features only a small fraction of the original expressibility. Yet, the DocBooklet format allows to create content structured into chapters, sections and paragraphs, which suffices to demonstrate basic principles of evolving documentation formats. The full definition of the DocBooklet format can be found in Appendix B. Finally, the fourth type of solution space model represents arbitrary media files that might be used as input to the application created as a variant of the product line. In the concrete case, the media file contains geo information used as mapping material by the personal navigation device in the car. The meta model describes a fictitious format called NavMap, which can be found in Appendix C. In addition to the presented types of models, it was also considered to demonstrate the use of configuration files and build scripts such as ANT\textsuperscript{4}. However, provided that there is an adequate model representation of the formats, the evolution procedure is similar to that of the presented artifacts. Thus, it was decided that the benefits of describing another format would be minimal and that the four chosen types of solution space models represent an essential part of the artifacts used in the software development process.

To show the eligibility of the evolution approach, it is further necessary to perform evolutions in a use case that is exemplary for actual practical application. Thus, a connection to a real world scenario has to be established. For this purpose, the automotive multimedia system demonstrated in the following is based on the system used in the MINI Cooper\textsuperscript{5} built by the BMW\textsuperscript{6} group. The series of cars was first launched in 2001 and had a major revision in 2006. These dates were used as basis for the evolution cycles in the example project. Furthermore, the features used in the problem space of the example were derived from the original configuration options for the MINI Cooper presented in its online configurator\textsuperscript{7}. Additionally, excerpts from the original MINI Cooper user manual were used as content for the example manual. However, the project in its entirety is purely fictitious. Especially the design and implementation aspects of the system do not have a real world counterpart as those details are not publicly accessible. Furthermore, the evolution steps do not reflect the actual changes made to the series of cars during the described time frames. For example, a third step of evolution for the year 2011 has been added to the project, which was not performed in the development of the MINI Cooper. Yet, all fictitious artifacts and evolutions have been chosen with great care with respect to plausibility to create a use case exemplary for practical application.

To prevent any confusion with the real car series built by the BMW group, the fictitious brand of car is called “MAXI Cheetah”. Its development has three separate iterations. In 2001, the car was first launched featuring a basic product line for configuration of the multimedia system.

\textsuperscript{2}http://www.jamopp.org/index.php/JaMoPP
\textsuperscript{3}http://www.docbook.org/
\textsuperscript{4}http://ant.apache.org/
\textsuperscript{5}http://www.mini.com
\textsuperscript{6}http://www.bmw.com/
\textsuperscript{7}http://www.mini.com/configurator/index.html?cm=mcom_forward_direct&action=vco. selectConfiguration&model=SU31&vgCode=SU31&vgModelCode=SU31
In 2006, a major revision was performed that also had effects on the product line. In 2011, the configuration options of the product line are modified again to reflect recent developments. The following explanations are divided into three sections devoted to each of these iterations. The first section describes the initial situation in 2001 and the second and third section explain the revisions of the car in 2006 and 2011 respectively. In order to describe the use case as realistically as possible, the latter two sections first explain the events leading up to the modifications that have to be performed during a revision. The rest of these sections describes which evolutions have to be used and demonstrates their application in the form of text as well as screenshots. Overcoming the challenges raised by the example scenario demonstrates the applicability and usefulness of the evolutions presented in this thesis.

5.1 Initial Situation in 2001

Before the introduction of the MAXI Cheetah to the market in 2001, the technical press praises the prototype of the car for its technical reliability at a relatively low price. The management of MAXI decides that these characteristics make the car a first class choice for young people. In order to increase sales, the company wants to accommodate this target group by providing all cars with a multimedia system that can be configured to only include required devices and thus further reduce costs.

One elementary part of the system is the on board computer which collects and displays vehicle data. It receives data from multiple sensors in the car, for example from the engine. The information gathered is then used to calculated values such as gas consumption. To communicate the calculated values to the driver, the results are visualized on a display in the middle console of the car. The on board computer is an integral part of car functionality as it is also used for internal error diagnosis. Thus, it is included in every configuration of the MAXI Cheetah.

Figure 5.1: Feature tree of the automotive multimedia product line in 2001.

Optionally, customers may decide to include an audio player into their car. Even in its most basic variant, the audio system features an FM/AM band radio tuner used to receive radio broadcasts around the world. Furthermore, it is possible to buy upgrades to play various media on the audio system. In particular, a cassette or CD player may be installed. Both these upgrades are independent of one another and may be bought in combination or individually. Looking at these configuration options from a software product line perspective creates the feature tree displayed in Figure 5.1.

Besides the feature tree in the problem space, the automotive multimedia product line fea-
features multiple solution space artifacts. The design of the system is specified in the form of a
UML model. Its contents are displayed in Figure 5.2. An essential part of each configuration
is the display in the middle console of the car. The class with similar name provides high level
methods to visualize graphical information. For example, it allows to draw primitives and bring
text to the screen. The class UserInterface is responsible for managing the interaction with
users of the car. It contains dialog elements for all present multimedia devices. Due to the
complexity of the user interface, the builder pattern [GHJV95] is used to create it. The class
responsible for assembly is UIBuilder. It creates appropriate menus for each of the multimedia
devices. However, the user interface built for a particular car depends on the concrete selection
of features. For example, if no audio player is included in the multimedia system, the respec-
tive menu should not be created either. Thus, the building procedure for the user interface is
adapted in the process of creating a particular variant of the product line. At runtime, the
created user interface redirects all requests in a particular menu to their respective device. In
the case of the on board computer, this logic is represented by the class OnBoardComputer. It
collects and evaluates sensor data from the car before visualizing it on the display. The class
AudioPlayer uses the display to show information on the currently played music. The class
delegates decoding of individual audio sources to specialized classes. For example, the classes
CassettePlayer and RadioTuner playback music cassettes digitally and decode radio signals
respectively. Equivalent is true for the class CDPlayer and the playback of audio CDs. However,
this class has an additional method to decode the CD text information saved on audio CDs,
which describes the currently played track and its album. In the case of CDs, this information
is displayed on the screen during playback. Naturally, the audio player requires the speaker
system of the car to play music. Thus, the class AudioPlayer has an association to the class
Speakers, which contains the control logic for the speaker system. Independent of the current
audio source, it is possible to regulate the volume level using the volume control represented by
the class VolumeControl. These classes of the UML model serve as design blueprints for the
implementation in source code.

The multimedia system of the MAXI Cheetah features a Java virtual machine and thus is
capable of interpreting byte code. Hence, Java was chosen as the implementation language for
the classes described in the UML model. However, implementation details in the source code
are neglected for the majority of the classes realizing the specified design to not clutter the
explanations. Yet, the class UIBuilder will have its implementation modified as an example
of evolutions on Java source code. The source code of the method createUI() of the class
UIBuilder is shown in Listing 5.1 as provided with the initial version of the product line.

Besides the design and implementation models, the product line also contains a user manual
for the multimedia system of the MAXI Cheetah. In its general form, the manual features
chapters describing the operation of the on board computer and the audio player. The latter
is further divided into sections describing the cassette player, the radio and the CD player.
Customers of the MAXI Cheetah are to be provided with a customized user manual containing
only the descriptions of the multimedia devices that were actually built into their car. Thus,
the manual has to be adapted in accordance with the concrete selection of features. In the case
of the optional features CassettePlayer and CDPlayer, this means that the respective sections
have to be removed from the manual if their features were not selected. If customers decide
Figure 5.2: UML diagram for the automotive multimedia product line in 2001.

Listing 5.1: Java source code of the method `createUI()` of the class `UIBuilder` of the year 2001.

```java
public UserInterface createUI() {
    UserInterface ui = new UserInterface();

    //build OnBoardComputer menu (block 1)
    Menu onBoardComputerMenu = new Menu();
    onBoardComputerMenu.setName("On Board Computer");
    //...
    ui.addMenu(onBoardComputerMenu);

    //build AudioPlayer menu (block 2)
    Menu audioPlayerMenu = new Menu();
    audioPlayerMenu.setName("Audio Player");
    //...
    ui.addMenu(audioPlayerMenu);

    return ui;
}
```
not to include the AudioPlayer feature in the configuration of their car, the entire chapter explaining it has to be removed from the user manual. In Listing 5.2, a shortened version of the user manual is shown in the previously introduced DocBooklet format. The complete contents of the user manual of the year 2001 can be found in digital form along with the example project.

Listing 5.2: Shortened version of the MAXI Cheetah user manual of the year 2001.

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>"MAXI Cheetah Manual"</title>
  <chapter>
    <title>"Audio Player"</title>
    <para>"So that you can start..."</para>
    <section>
      <title>"Radio"</title>
      <para>"With your car radio ..."</para>
      <para>"Frequency Modulation ..."</para>
    </section>
    <section>
      <title>"Cassette Playback"</title>
      <para>"Cassette equipment ..."</para>
    </section>
    <section>
      <title>"CD Playback"</title>
      <para>"Press repeatedly ..."</para>
    </section>
  </chapter>
  <chapter>
    <title>"On Board Computer"</title>
    <para>"The system runs a ..."</para>
    <para>"If a malfunction ..."</para>
  </chapter>
</book>
```

In order to customize the solution space models according to the above specifications, a mapping from feature expressions of the feature tree to the respective parts of the solution space models has to be established. In Table 5.1, all mappings of the 2001 version of the automotive multimedia product line can be seen in an intuitive notation.

With the feature tree, the three solution space models and the mapping between problem and solution space, the 2001 version of the automotive multimedia product line is fully specified.

### 5.2 First Revision in 2006

Five years after the introduction of the MAXI Cheetah, the car has proven to be a tremendous success. The MAXI management intends to maintain the good sales figures and therefore wants to produce an up to date version of the car series. In consequence, a major revision
of the technical details of the MAXI Cheetah is planned for the year 2006 that also affects the multimedia system and thus its product line. All in all, three major changes have to be performed. First, the cassette player will be removed as configuration option. Second, a CD player capable of playing the emerging digital audio format MP3 will be included in the product line and third, a personal navigation device will be added as optional feature for the car. Furthermore, minor changes will be performed on the implementation of the class UIBuilder.

### 5.2.1 Removing the Cassette Player

Recent statistics of the audio market show that the CD is the predominant medium for music and that there are very few new releases published on music cassette. Furthermore, demand for the cassette player has been low ever since the introduction of the MAXI Cheetah to the market and continues to decline. To cut production and storage costs of the cassette player, management decides that future configurations of the car will no longer contain a cassette deck so that the cassette player and all its implementing elements have to be removed from the product line.

This change is performed using the evolution Remove Feature and Owned Assets, which first deletes the feature from the feature tree, adapting the cardinality of its parent group, and then removes all elements that are used exclusively by the cassette player from their respective solution space model. In the concrete case, this means that the explaining section of the user manual as well as the class CassettePlayer of the UML model along with its incoming association are removed.
removed. In Figure 5.3, the process of applying the *Remove Feature and Owned Assets* evolution is visualized using screenshots.

The subsequent remapping for *Remove Feature and Owned Assets* applies the remove feature mapping operator to the feature *CassettePlayer*. It removes all references to the feature *CassettePlayer* from the feature expressions in the mapping model. If a feature expression is empty after the removal, the entire mapping will be deleted from the mapping model. In the concrete case, this means that the three mappings having the feature *CassettePlayer* as feature expression are deleted from the mapping model. The remapping dialog for the evolution are presented in Figure 5.4 and an excerpt from the complete mapping model is shown in Table 5.2 where strike through formatting is used to mark the entries deleted by the remapping.

<table>
<thead>
<tr>
<th>CassettePlayer</th>
<th>UML:Class[CassettePlayer]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CassettePlayer</td>
<td>UML:Association[AudioPlayer-CassettePlayer]</td>
</tr>
<tr>
<td>CassettePlayer</td>
<td>Manual:Section[CassettePlayback]</td>
</tr>
</tbody>
</table>

Table 5.2: Changes to the mapping model due to applying *Remove Feature and Owned Assets* to the feature *CassettePlayer*.

Performing the *Remove Feature and Owned Assets* evolution for the feature *CassettePlayer* effectively deleted the cassette player and all its implementation elements from the product line. Therefore, future configurations of the MAXI Cheetah no longer allow to include a cassette player.

### 5.2.2 Adding an MP3 CD Player

A study of the recent developments of the music market has lead to the removal of the cassette player from the multimedia system. These same statistics also revealed that there is a new digital audio format emerging called MP3. Occasionally, sales representatives have reported customers requesting a CD player capable of playing this new format. Encouraged by this tendency, the research and development department of MAXI has recently presented the first version of an MP3 player for cars built in-house. The management of MAXI has expressed its approval to use the new device as configuration option for the multimedia system in the upcoming revision of the MAXI Cheetah. However, the regular CD player should still be available. To disambiguate both CD players, the player with the capabilities of the old CD player is now referred to as audio CD player. Performing these changes requires modifying the feature model as well as solution space models.

The essential software part of the new MP3 CD player is the MP3 decoder programmed by the research and development department. It features a simplistic interface consisting of merely one method to decode MP3 files to an audio stream that can be played back using the cars conventional speaker system. The decoder is included in the UML design model of the product line as class *MP3Decoder*. Furthermore, an association from the class *CDPlayer* to the newly created class is added. These changes are performed manually in the UML editor and the created elements will later be the solution space end of a mapping.

With the upcoming distinction of audio and MP3 CD players, the class *CDPlayer* no longer needs to be capable of performing playback of audio CDs on its own. Therefore, its method
Figure 5.3: Applying *Remove Feature and Owned Assets* to the feature *CassettePlayer*.

a) screenshot of selecting the menu entry for the evolution

b) excerpt from the UML diagram showing the effects of the evolution

c) excerpt from the user manual showing the effects of the evolution

```xml
<book version="1.0" encoding="utf-8">
  <title>"MAXI Cheetah Manual"</title>
  <chapter>
    <title>"Audio Player"</title>
    <para>"So that you can start..."</para>
    <section>
      <title>"Radio"</title>
      <para>"With your car radio ..."</para>
      <para>"Frequency Modulation ..."</para>
    </section>
    <!-- Section "Cassette Playback" was removed -->
    <section>
      <title>"CD Playback"</title>
      <para>"Press repeatedly ..."</para>
    </section>
  </chapter>
  <!-- ... -->
</book>
```
to decode CD text is moved to a separate entity that can be used exclusively by the audio CD player. For this purpose, the evolution Replace Method with Method Object is employed for the UML model. It extracts the method readCDText() of the class CDPlayer and places it in a newly created class called CDTextReader. In addition, an association from CDPlayer to CDTextReader is added signaling the relation of both classes. The process of applying Replace Method with Method Object to the method readCDText() of the class CDPlayer is visualized in Figure 5.5.

The subsequent remapping of the evolution copies the mapping of the class CDPlayer to the class CDTextReader and the newly created association. In Figure 5.6, a screenshot of the remapping dialog for the evolution is depicted. Furthermore, an excerpt of the complete mapping model displaying the newly created mappings is shown in Table 5.3.

In addition to the changes to the UML diagram, the user manual also has to be prepared for the upcoming separation of audio and MP3 CD player. Therefore, individual sections on each of the players should be included. For this purpose, the original section on the CD player is duplicated using the Duplicate Section evolution. Title and content of both sections are altered...
5.2. FIRST REVISION IN 2006

Figure 5.5: Applying Replace Method with Method Object to the method `readCDText()` of the class `CDPlayer` in the UML diagram.

a) screenshot of selecting the menu entry for the evolution

![Screenshot of menu entry](image)

b) excerpt from the UML diagram showing the effects of the evolution

![UML diagram excerpt](image)

Figure 5.6: Remapping dialog after applying Replace Method with Method Object to the method `readCDText()` of the class `CDPlayer` in the UML diagram.

![Remapping dialog](image)

Table 5.3: Changes to the mapping model due to applying Replace Method with Method Object to the method `readCDText()` of the class `CDPlayer` in the UML diagram.

<table>
<thead>
<tr>
<th>CDPlayer</th>
<th>UML:Class[CDTextReader]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDPlayer</td>
<td>UML:Association[CDPlayer-CDTextReader]</td>
</tr>
</tbody>
</table>
manually to describe their respective CD player.

Figure 5.7: Applying *Duplicate Section* to the section “CD Playback” in the user manual.

a) screenshot of selecting the menu entry for the evolution

![Screenshot of selecting the menu entry for the evolution](image)

b) excerpt from the user manual showing the effects of the evolution (including manual content adaptation)

```xml
<?xml version="1.0" encoding="utf-8"?>
<book>
  <title>Example Document</title>
  <chapter>
    <title>Example Chapter</title>
    <para>First paragraph.</para>
    <section>
      <title>Example Section B</title>
      <para>Fourth paragraph.</para>
      <para>Fifth paragraph.</para>
    </section>
    <section>
      <title>Example Section A</title>
      <para>Second paragraph.</para>
      <para>Third paragraph.</para>
    </section>
  </chapter>
</book>
```

When duplicating a section in the user manual, the mapping applying to the original section has to be copied to the created clone in order to keep the product line consistent. For this purpose, the copy object mapping operator is employed. In Figure 5.8, a screenshot of the remapping dialog associated with *Duplicate Section* is shown and in Table 5.4, the relevant excerpt of the mapping model that contains the copied mapping is displayed.

After preparing the solution space models, the features for audio and MP3 CD player can be separated in the feature model. For this purpose, the evolution *Split Feature* is used on the feature *CDPlayer*. In the configuration dialog of the evolution, appropriate values are entered to create two split parts called *AudioCDPlayer* and *MP3CDPlayer*. After performing the evolution,
5.2. FIRST REVISION IN 2006

Figure 5.8: Remapping dialog after applying Duplicate Section to the section “CD Playback” in the user manual.

Table 5.4: Changes to the mapping model due to applying Duplicate Section to the section “CD Playback” in the user manual including manual content adaptation.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target(s)</th>
<th>Mappings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Section CD Playback</td>
<td>Section CD Playback</td>
<td>CDPlayer Manual:Section[CDPlayback]</td>
</tr>
<tr>
<td>Section CD Playback</td>
<td>ExampleProject_2001/resources/Manual.docbooklet</td>
<td>CDPlayer Manual:Section[AudioCDPlayback]</td>
</tr>
<tr>
<td>Section CD Playback</td>
<td>ExampleProject_2001/resources/Manual.docbooklet</td>
<td>CDPlayer Manual:Section[MP3CDPlayback]</td>
</tr>
</tbody>
</table>

the feature CDPlayer has two new child features as specified. The process of applying the Split Feature evolution is shown in Figure 5.9 in a series of screenshots.

The remapping dialog following Split Feature allows to redistribute the mappings of the original feature CDPlayer to its split parts AudioCDPlayer and MP3CDPlayer. The class CDPlayer in the UML diagram and its incoming association from AudioPlayer are used by both features. Thus, the OR operator has to be employed for concatenation. However, the CDTexReader class and its incoming association from CDPlayer are exclusively used for audio cd playback. As such, the respective mappings are redirected solely to the AudioCDPlayer feature but not the MP3CDPlayer. Equivalent holds for the section of the user manual describing the operation of the audio CD player. Likewise, the section on the MP3 CD player is redirected only to the feature MP3CDPlayer. These settings are reflected in the configuration dialog for the remapping step, which is presented in Figure 5.10. The effects of executing this remapping can be seen in Table 5.5 as excerpt from the mapping model.

After the remapping dialog, the mappings from feature MP3CDPlayer to the UML class MP3Decoder and its incoming association are added manually. This step completes the mapping process for the new MP3 CD player.

However, the group containing both specialized CD players should only allow selection of
Figure 5.9: Applying *Split Feature* to the feature CDPlayer.

a) screenshot of selecting the menu entry for the evolution

![Split Feature menu](image)

b) screenshot of the configuration dialog of the evolution

![Configuration dialog](image)

c) excerpt from the feature model showing the effects of the evolution

```
[CDPlayer]

[AudioCDPlayer OR MP3CDPlayer]
```

Table 5.5: Changes to the mapping model due to applying *Split Feature* to the feature CDPlayer.

<table>
<thead>
<tr>
<th>CDPlayer</th>
<th>UML:Class[CDPlayer]</th>
</tr>
</thead>
<tbody>
<tr>
<td>CDPlayer</td>
<td>UML:Class[CDTextReader]</td>
</tr>
<tr>
<td>AudioCDPlayer OR MP3CDPlayer</td>
<td>UML:Class[CDPlayer]</td>
</tr>
<tr>
<td>AudioCDPlayer OR MP3CDPlayer</td>
<td>UML:Association[AudioPlayer-CDPlayer]</td>
</tr>
<tr>
<td>AudioCDPlayer</td>
<td>UML:Class[CDTextReader]</td>
</tr>
<tr>
<td>AudioCDPlayer</td>
<td>UML:Association[CDPlayer-CDTextReader]</td>
</tr>
<tr>
<td>AudioCDPlayer</td>
<td>Manual:Section[AudioCDPlayback]</td>
</tr>
<tr>
<td>MP3CDPlayer</td>
<td>Manual:Section[MP3CDPlayback]</td>
</tr>
</tbody>
</table>
5.2. FIRST REVISION IN 2006

Figure 5.10: Remapping dialog after applying Split Feature to the feature CDPlayer.

exactly one of its features. For this purpose, the evolution Transform to Alternative Group is employed, which sets both the minimum and maximum cardinality of a group to one. The process of applying the evolution is visualized in Figure 5.11 by a series of screenshots.

Figure 5.11: Applying Transform to Alternative Group to the group of the specialized CD players.

a) screenshot of selecting the menu entry for the evolution

b) excerpt from the feature model showing the effects of the evolution

With these modifications of the automotive multimedia software product line, it is now possible for customers of the MAXI Cheetah to configure their vehicle to either contain an audio CD player as in the previous version of the car or to include the new CD player, which is
5.2.3 Adding a Personal Navigation Device

Besides the MP3 CD Player, the major addition to the automotive multimedia product line in the revision of 2006 is a GPS personal navigation system. The device can use positioning information from satellites to calculate the car’s terrestrial position and then calculate routes using the included mapping material. Maps are provided in the NavMap format and there is navigational information for many countries from across the world. An exemplary excerpt of the NavMap file provided with the navigation system can be seen in Listing 5.3.

To receive user input, the digital controls of the navigation device will be integrated into the rest of the user interface system of the car. Furthermore, an optional voice recognition mechanism is offered, which decodes voice commands of the driver and uses them to configure the navigation device. To communicate navigational information as output, the device uses both the speaker system and the display of the car. In this regard, the navigation system is very similar to the audio player, which utilizes the same system components. Thus, the feature PersonalNavigation is best realized by applying the evolution Duplicate Feature to the feature AudioPlayer. However, only the direct feature should be cloned but none of its descendants. This option has to be reflected in the configuration dialog of the evolution. The process of applying Duplicate Feature to the feature AudioPlayer is shown in Figure 5.12.

The adequate remapping to duplicating a feature is to copy the feature mapping from the original to the newly created feature. In the concrete case, only the mappings to the speakers and volume control have to be copied. The display belongs to the core of the product line and mappings to the audio player class in the UML diagram and its outgoing associations are specific to the AudioPlayer feature. Thus, these mappings will not be reused in the PersonalNavigation feature so that they are excluded from the Copy Feature Mapping operation in the remapping dialog. In Figure 5.13, a screenshot of the remapping dialog for Copy Feature Mapping displaying the described configuration is shown. In Table 5.6, the resulting changes to the mapping model are displayed.

Table 5.6: Changes to the mapping model due to applying Duplicate Feature to the feature AudioPlayer.

<table>
<thead>
<tr>
<th>AudioPlayer</th>
<th>UML:Class[Speakers]</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioPlayer</td>
<td>UML:Class[VolumeControl]</td>
</tr>
<tr>
<td>AudioPlayer OR PersonalNavigation</td>
<td>UML:Association[VolumeControl-Speakers]</td>
</tr>
</tbody>
</table>

The voice recognition feature is to be added as logical child of the personal navigation feature. For this purpose, an empty childgroup for the feature PersonalNavigation is created manually. With this group selected, the evolution Insert Feature is executed to create a new optional feature with a specified name. The same evolution is used to create the Maps feature, which represents the mapping material provided along with the navigation device. However, it is
not possible to operate the navigation system without mapping material so that the Maps feature has to be made mandatory. This is done by using the evolution Transform to Mandatory Feature on the feature Maps. Furthermore, the group containing both newly created features requires its cardinality to be set to allow adequate selection of its features. This is done by applying the evolution Transform to And Group, which sets the minimum and maximum cardinality in a way that mandatory features have to be selected and optional features may be selected. An excerpt of the feature model reflecting the summary of these changes is displayed in Figure 5.14.

After adding new features and duplicating reusable parts of the existing realization, elements
specific to the personal navigation device have to be incorporated into the solution space of the product line. For one, this is done on the design level in the UML diagram. The classes PersonalNavigation and VoiceRecognition are added along with their various incoming and outgoing associations. An excerpt from the UML diagram displaying the added elements is shown in Figure 5.15.

In addition, code is added to the Java class for the UI builder to create the user interface for the personal navigation device. The modified version of the source code is displayed in Listing 5.4.

After these modifications, the feature VoiceRecognition is mapped manually to the UML class with the same name and its outgoing association. The feature PersonalNavigation is mapped to its respective class and the outgoing associations to the speakers and the display as well as the incoming association from the user interface. Furthermore, the PersonalNavigation feature is also mapped to the respective lines of source code in the Java file for the UI builder that create the user interface for the personal navigation device. In addition, the feature Maps is mapped to the contents of the NavMap file. The relevant parts of the resulting mapping model are displayed in Table 5.7.

With these changes to the automotive multimedia product line, a personal navigation system can now be chosen as configuration option when purchasing the MAXI Cheetah.
Figure 5.13: Remapping dialog after applying *Duplicate Feature* to the feature *AudioPlayer*.

Figure 5.14: Excerpt from the feature model showing the newly created features *Voice-Recognition* and *Maps*.
Figure 5.15: Excerpt from the UML diagram showing the elements relevant to the personal navigation system.

Listing 5.4: Manually modified version of the Java source code of the method createUI() of the class UIBuilder.

```java
public UserInterface createUI() {
    UserInterface ui = new UserInterface();
    //...

    //build PersonalNavigation menu (block 3)
    Menu personalNavigationMenu = new Menu();
    personalNavigationMenu.setName("Personal Navigation");
    //...
    ui.addMenu(personalNavigationMenu);

    return ui;
}
```
5.2. FIRST REVISION IN 2006

Table 5.7: Manual changes to the mapping model to incorporate elements specific to the personal navigation system.

<table>
<thead>
<tr>
<th>PersonalNavigation</th>
<th>UML:Class[PersonalNavigation]</th>
</tr>
</thead>
<tbody>
<tr>
<td>PersonalNavigation</td>
<td>UML:Association[PersonalNavigation-Speakers]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Association[PersonalNavigation-Display]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Association[UserInterface-PersonalNavigation]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>Java:Class[UIBuilder]:Method[createUI]:Block[3]</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>UML:Class[VoiceRecognition]</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>UML:Association[VoiceRecognition-PersonalNavigation]</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>UML:Class[PersonalNavigation]:Method[receiveVoiceCommand]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Germany]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Poland]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[USA]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Canada]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Mexico]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Costa Rica]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Argentina]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Brazil]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Japan]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[China]</td>
</tr>
</tbody>
</table>

5.2.4 Changing the Implementation of the UI Builder

Besides the far reaching changes involved in realizing new features, there are also tasks in software maintenance that require changes of implementation details. In this case, the implementation of the UIBuilder is to be restructured. Currently, the method createUI() builds the user interface for all devices installed in a particular car. After adding the personal navigation device, the method has increased in the number of lines of code. In order to maintain readability of the code fragment, its functionality is to be distributed to several methods that create menus for each of the devices. For example, the menu for the audio player should be created by a method named createAudioPlayerMenu(), which is called from within the createUI() method. To achieve this restructuring, Extract Method is used on each of the three code fragments that create the user interface for the on board computer, the audio player and the personal navigation system. The parameter describing the user interface is added manually. The process of applying Extract Method is shown in Figure 5.16.

The remapping complementary to Extract Method copies the mapping of the original code fragment to the new method and its call from within createUI(). Thus, all traces of the creation of a particular menu are removed if its respective feature is not part of a particular configuration. The remapping operation creates the changes of the mapping model displayed in Table 5.8.

With these modifications, the readability of the method createUI() is preserved even though additional functionality has been added. Furthermore, the subsequent remapping maintains the consistency of the product line so that all prior configurations can still be derived successfully after the modification.
Figure 5.16: Applying Extract Method to the code fragments in the `createUI()` method of the class `UIBuilder` in Java source code.

a) screenshot of selecting the menu entry for the evolution

b) excerpt from the source code showing the effects of the evolution including the newly created private methods and their manually added parameter for the user interface

```java
public UserInterface createUI() {
    UserInterface ui = new UserInterface();

    //Call 1
    createOnBoardComputerMenu(ui);
    //Call 2
    createAudioPlayerMenu(ui);
    //Call 3
    createPersonalNavigationMenu(ui);

    return ui;
}

private void createOnBoardComputerMenu(UserInterface ui) {
    //build OnBoardComputer menu
    Menu onBoardComputerMenu = new Menu();
    onBoardComputerMenu.setName("On Board Computer");
    //...
    ui.addMenu(onBoardComputerMenu);
}

private void createAudioPlayerMenu(UserInterface ui) {
    //build AudioPlayer menu
    Menu audioPlayerMenu = new Menu();
    audioPlayerMenu.setName("Audio Player");
    //...
    ui.addMenu(audioPlayerMenu);
}

private void createPersonalNavigationMenu(UserInterface ui) {
    //build PersonalNavigation menu
    Menu personalNavigationMenu = new Menu();
    personalNavigationMenu.setName("Personal Navigation");
    //...
    ui.addMenu(personalNavigationMenu);
}
```
5.2. FIRST REVISION IN 2006

Table 5.8: Changes of the mapping model as result of applying *Extract Method* to the code fragments in the createUI() method of the class UIBuilder in Java source code.

<table>
<thead>
<tr>
<th>Class</th>
<th>Method</th>
<th>Call</th>
<th>Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>OnBoardComputer</td>
<td>UIBuilder.createUI.Block1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>UIBuilder.createUI.Block2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UIBuilder.createUI.Block3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>UIBuilder.createOnBoardComputerMenu</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>UIBuilder.createUI.Call2</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UIBuilder.createAudioPlayerMenu</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UIBuilder.createPersonalNavigationMenu</td>
<td></td>
<td>3</td>
</tr>
</tbody>
</table>

5.2.5 Summary of the Changes of the First Revision in 2006

During the course of the first revision of the automotive multimedia product line of the MAXI Cheetah in the year 2006, a number of changes was performed. The cassette player was removed from the product line and a new MP3 CD player was added as configuration option. Furthermore, a personal navigation device was added as optional feature. These changes added and removed features. Therefore, the feature model at the end of the 2006 revision is displayed in Figure 5.17.

Figure 5.17: The feature model of the automotive multimedia product line at the end of the 2006 revision.

Furthermore, new functionality was added in the solution space and existing implementations were restructured. Thus, the realizing artifacts of the product line were modified as well. As an example, the UML diagram at the end of the 2006 revision is displayed for reference in Figure 5.18.

Due to the various remapping steps that were performed in the course of the revision, the original mapping model has changed tremendously. In Table 5.9, the complete mapping model at the end of the 2006 revision is shown to provide an overview of the performed modifications.
Table 5.9: Complete feature mapping for the automotive multimedia product line at the end of the 2006 revision.

<table>
<thead>
<tr>
<th>Component</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioPlayer</td>
<td>UML:Class[AudioPlayer]</td>
</tr>
<tr>
<td>AudioPlayer OR PersonalNavigation</td>
<td>UML:Class[Speakers]</td>
</tr>
<tr>
<td>AudioPlayer OR PersonalNavigation</td>
<td>UML:Association[AudioPlayer-Speakers]</td>
</tr>
<tr>
<td>AudioPlayer OR PersonalNavigation</td>
<td>UML:Class[VolumeControl]</td>
</tr>
<tr>
<td>AudioPlayer OR PersonalNavigation</td>
<td>UML:Association[VolumeControl-Speakers]</td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>Manual:Chapter[AudioPlayer]</td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>Java:Class[UIBuilder]:Method[createUI]:Call[2]</td>
</tr>
<tr>
<td>Radio</td>
<td>Java:Class[UIBuilder]:Method[createAudioPlayerMenu]</td>
</tr>
<tr>
<td>Radio</td>
<td>UML:Class[RadioTuner]</td>
</tr>
<tr>
<td>AudioCDPlayer OR MP3CDPlayer</td>
<td>UML:Class[CDPlayer]</td>
</tr>
<tr>
<td>AudioCDPlayer OR MP3CDPlayer</td>
<td>UML:Association[AudioPlayer-CDPlayer]</td>
</tr>
<tr>
<td>AudioCDPlayer</td>
<td>UML:Class[CDTextReader]</td>
</tr>
<tr>
<td>AudioCDPlayer</td>
<td>Manual:Section[AudioCDPlayback]</td>
</tr>
<tr>
<td>MP3CDPlayer</td>
<td>UML:Class[MP3Decoder]</td>
</tr>
<tr>
<td>MP3CDPlayer</td>
<td>UML:Association[CDPlayer-MP3Decoder]</td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>UML:Class[OnBoardComputer]</td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>UML:Association[OnBoardComputer-Display]</td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>UML:Association[UserInterface-OnBoardComputer]</td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>Java:Class[UIBuilder]:Method[createUI]:Call[1]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Class[PersonalNavigation]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Association[PersonalNavigation-Speakers]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Association[PersonalNavigation-Display]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Association[UserInterface-PersonalNavigation]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>Java:Class[UIBuilder]:Method[createUI]:Call[3]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>Java:Class[UIBuilder]:Method[createPersonalNavigationMenu]</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>UML:Class[VoiceRecognition]</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>UML:Association[VoiceRecognition-PersonalNavigation]</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>UML:Class[PersonalNavigation]:Method[receiveVoiceCommand]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Germany]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Poland]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[USA]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Canada]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Mexico]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Argentina]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Brazil]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[Japan]</td>
</tr>
<tr>
<td>Maps</td>
<td>NavMap:Country[China]</td>
</tr>
</tbody>
</table>
The rest of the modified models is provided in digital form as part of the example project. These models summarize the changes of the first revision of the MAXI Cheetah in 2006 and they are used as basis for the automotive multimedia product line for five years until the second revision in 2011.

5.3 Second Revision in 2011

Five years have passed since the first revision of the MAXI Cheetah in 2006. The changes performed to the car and its multimedia system have proven to be a success and sales of the MAXI Cheetah are at a constant high level. However, the passage of time has yielded new requirements. The MAXI management intends to maintain the positive sales figures for the car and thus plans another revision for the year 2010 to address the raised issues. The majority of the proposed changes concern technical details of the car but there also are modifications to the multimedia system that have to be performed.

First of all, many customers have voiced their concerns over the distinction of audio CD player and MP3 CD player. They feel that in the year 2011, a CD player should be capable of handling both formats. Thus, the distinction of the two CD players is no longer appropriate. As a result, the two individual devices for the CD players are to be replaced by a model capable of playing audio CDs as well as MP3 CDs. Naturally, the configuration options of the product line have to reflect these modifications.

On the positive side, the voice recognition feature of the personal navigation device was
received with great enthusiasm by the customers. Almost all navigation systems are sold with the optional feature, which proves its eligibility. As a consequence, it was decided to extend the capabilities of the voice recognition feature so that it can not only control the personal navigation device but also the audio player. For the audio player to receive voice commands to control audio playback, modifications of implementation artifacts are required. Furthermore, the new capabilities of the voice recognition module make it independent of the personal navigation device so that the modified feature has to be relocated within the feature tree.

There are also changes to the personal navigation device. Along with the navigation system, customers currently have to buy maps for countries from all over the world even though the majority of the maps will never be used. Naturally, customers are not pleased with having to pay for features they do not use. To increase customer satisfaction, management decided to offer individual modules for different geographical areas of the world. After the revision, there should be distinct features for the maps of Europe, North America, Central America, South America and Asia, which are the key markets for the MAXI Cheetah. Customers should have the option to buy arbitrary combinations of these map modules as long as at least one of them is selected along with the personal navigation system to ensure its operability.

Furthermore, the research and development department of MAXI has abandoned the development of their MP3 decoder because of recurring errors that make maintenance of the code very expensive. As an alternative, an MP3 decoder of an external vendor has been licensed that now has to be integrated into the product line.

These four major changes to the automotive multimedia system will complete the revision of 2011.

5.3.1 Creating a Multi-Format CD Player

With the revision of 2011, the distinction of audio and MP3 CD player is rendered obsolete. The research and development department of MAXI has created a new player that is capable of handling both formats. To save development costs, the new hardware uses largely the same control logic so that most of the implementation can be reused.

However, the new player is capable of decoding CD text information in hardware and no longer requires a software module to perform this task. As a consequence, the class CDTextReader and its incoming association in the UML diagram can be removed. This is done by use of the evolution Remove Class for UML models, which deletes a class and all associations it participates in. The process of applying the evolution is documented in Figure 5.19.

Subsequent to removing an element from the UML diagram, the respective mappings also have to be deleted. For this purpose, the remove object mapping step is used. The dialog for the remapping step is shown in Figure 5.20 and the resulting changes to the mapping model are presented in Table 5.10.

Table 5.10: Changes to the mapping model due to applying Remove Class to the class CDTextReader of the UML diagram.

<table>
<thead>
<tr>
<th>AudioCDPlayer</th>
<th>UML:Class[CDTextReader]</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioCDPlayer</td>
<td>UML:Association[CDPlayer-CDTextReader]</td>
</tr>
</tbody>
</table>
Figure 5.19: Applying Remove Class to the class CDTextReader of the UML diagram.

a) screenshot of selecting the menu entry for the evolution

b) excerpt from the UML model showing the effects of the evolution

Figure 5.20: Remapping dialog after applying Remove Class to the class CDTextReader of the UML diagram.

With these preparatory modifications to the realization of the CD player, it is now possible to combine the individual features of the distinct CD players. This type of change is performed using the evolution Merge Features. The features AudioCDPlayer and MP3CDPlayer are to be merged with each other but also with their parent feature CDPlayer. For this purpose, the configuration dialog of the evolution offers the possibility to selected the option to merge with
the parent feature. The process of applying the evolution **Merge Features** is shown in Figure 5.21.

Figure 5.21: Applying **Merge Features** to the features **AudioCDPlayer** and **MP3CDPlayer**.

a) screenshot of selecting the menu entry for the evolution

![Screenshot of selecting menu entry](image)

b) screenshot of the configuration dialog of the evolution

![Configuration dialog](image)

c) excerpt from the feature model showing the effects of the evolution

![Feature model](image)

In consequence to the evolution, a remapping has to be performed to keep the software product line consistent. When combining features, the **Merge Feature Mapping** step is applied, which combines mappings of the source features on the target. In the concrete case, this means that the mappings of the features **AudioCDPlayer** and **MP3CDPlayer** are merged with previously existing mappings of **CDPlayer**. The remapping dialog of merge feature mapping is shown in Figure 5.22 and the resulting changes to the mapping model are displayed in Table 5.11.

With these changes to the product line, the new CD player capable of playing audio as well as MP3 CDs can now be built into the MAXI Cheetah. In consequence, customers no longer have to choose between two CD players but merely have to decide whether they want a CD player in their car or not.

### 5.3.2 Enhancing Voice Recognition to Control the Audio Player

The popularity of the voice recognition feature has lead the MAXI management to the decision that the audio player could be controlled by voice commands as well. For this purpose, the problem and solution space of the product line have to be modified. First, the design specification in the UML diagram has to be prepared for the changes. In order to be targets for the voice
5.3. SECOND REVISION IN 2011

Figure 5.22: Remapping dialog after applying *Merge Features* to the features *AudioCDPlayer* and *MP3CDPlayer*.

reorganization, both classes *PersonalNavigation* and *AudioPlayer* need a common base, which will be called *VoiceControllable*. To create it, the evolution *Extract Super Class* for UML models is used on the method `receiveVoiceCommand()` of the class *PersonalNavigation*. The evolution creates a new super class for the class *PersonalNavigation* and places the extracted method in it. This process is visualized in Figure 5.23.

The remapping to the evolution copies the mapping of the extracted method and its containing class to the newly created class. The remapping dialog is shown in Figure 5.24 and the changes to the mapping model are displayed in Table 5.12.

To complete the procedure of adding voice recognition support to the audio player, manual changes have to be performed. First, the association of the class *VoiceRecognition* to the class *PersonalNavigation* is changed to point to *VoiceControllable*. Second, the multiplicity of the association’s target end is changed so that multiple devices can be controlled. Furthermore, a generalization from class *AudioPlayer* to the newly created class *VoiceControllable* is added. Finally, the mapping to class *VoiceControllable* is changed so that the class is only included if the feature *VoiceRecognition* is selected. This change is shown in Table 5.13.

After the modifications of the solution space, the problem space has to be altered as well. The feature *VoiceRecognition* has gained additional responsibilities and thus has become independent of its previous parent feature *PersonalNavigation*. Therefore, the feature *Voice-
Table 5.11: Changes to the mapping model due to applying Merge Features to the features AudioCDPlayer and MP3CDPlayer.

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioCDPlayer OR MP3CDPlayer</td>
<td>UML:Class[CDPlayer]</td>
</tr>
<tr>
<td>AudioCDPlayer OR MP3CDPlayer</td>
<td>UML:Association[AudioPlayer-CDPlayer]</td>
</tr>
<tr>
<td>AudioCDPlayer</td>
<td>Manual:Section[AudioCDPlayer]</td>
</tr>
<tr>
<td>MP3CDPlayer</td>
<td>UML:Class[MP3Decoder]</td>
</tr>
<tr>
<td>MP3CDPlayer</td>
<td>UML:Association[CDPlayer-MP3Decoder]</td>
</tr>
<tr>
<td>MP3CDPlayer</td>
<td>Manual:Section[MP3CDPlayer]</td>
</tr>
<tr>
<td>CDPlayer</td>
<td>UML:Class[CDPlayer]</td>
</tr>
<tr>
<td>CDPlayer</td>
<td>UML:Association[AudioPlayer-CDPlayer]</td>
</tr>
<tr>
<td>CDPlayer</td>
<td>Manual:Section[AudioCDPlayback]</td>
</tr>
<tr>
<td>CDPlayer</td>
<td>UML:Class[MP3Decoder]</td>
</tr>
<tr>
<td>CDPlayer</td>
<td>UML:Association[CDPlayer-MP3Decoder]</td>
</tr>
<tr>
<td>CDPlayer</td>
<td>Manual:Section[MP3CDPlayback]</td>
</tr>
</tbody>
</table>

Figure 5.23: Applying Extract Super Class to the method receiveVoiceCommand() of the class PersonalNavigation in the UML diagram.

a) screenshot of selecting the menu entry for the evolution

![Screenshot of selecting menu entry](image1.png)

b) screenshot of the configuration dialog of the evolution

![Configuration dialog](image2.png)

c) excerpt from the UML model showing the effects of the evolution

```mermaid
graph LR
  PersonalNavigation --> VoiceControllable
  VoiceControllable --> receiveVoiceCommand
```

77x727
CHAPTER 5. EXAMPLE PROJECT
5.3. SECOND REVISION IN 2011

Figure 5.24: Remapping dialog after applying Extract Super Class to the method receiveVoiceCommand() of the class PersonalNavigation in the UML diagram.

Recognition has to be moved up in the feature tree. This type of modification is performed using the evolution Pull Up Feature. The process is displayed in a series of screenshots in Figure 5.25.

The evolution Pull Up Feature does not require a remapping operation. However, a constraint is added to the feature tree after the evolution was performed. The feature VoiceRecognition should only be selectable if at least one of the features PersonalNavigation or AudioPlayer was selected as well. The constraint VoiceRecognition → PersonalNavigation OR AudioPlayer ensures that voice recognition is only added to the configuration of a car if all of its requirements are met.

5.3.3 Restructuring Personal Navigation Maps

In addition to the previous changes, the feature Maps will be divided into several distinct features for geographical areas of the world. In particular, features for the mapping material of Europe, North America, Central America, South America and Asia are to be created as those are the main target areas for sales of the MAXI Cheetah. For this purpose, the evolution Split Feature
Table 5.12: Changes to the mapping model due to applying *Extract Super Class* to the method `receiveVoiceCommand()` of the class *PersonalNavigation* in the UML diagram.

<table>
<thead>
<tr>
<th>VoiceRecognition</th>
<th>UML:Class[PersonalNavigation]:Method[receiveVoiceCommand]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VoiceRecognition</td>
<td>UML:Class[VoiceControllable]:Method[receiveVoiceCommand]</td>
</tr>
<tr>
<td>VoiceRecognition OR PersonalNavigation</td>
<td>UML:Class[VoiceControllable]</td>
</tr>
</tbody>
</table>

Table 5.13: Manually modified mapping for the class *VoiceControllable* in the UML diagram.

<table>
<thead>
<tr>
<th>VoiceRecognition OR PersonalNavigation</th>
<th>UML:Class[VoiceControllable]</th>
</tr>
</thead>
<tbody>
<tr>
<td>VoiceRecognition</td>
<td>UML:Class[VoiceControllable]</td>
</tr>
</tbody>
</table>

Figure 5.25: Applying *Pull Up Feature* to the feature *VoiceRecognition*.

a) screenshot of selecting the menu entry for the evolution

(b) excerpt from the feature model showing the effects of the evolution
employed. It divides a given feature into an arbitrary number of child features in order to refine the feature model. Each of the constituents has a name according to the described regions. The evolution process including the configuration dialog for *Split Feature* is shown in Figure 5.26 with all settings for dividing the feature *Maps*.

![Figure 5.26: Applying *Split Feature* to the feature *Maps*.](image)

a) screenshot of selecting the menu entry for the evolution

b) screenshot of the configuration dialog of the evolution

c) excerpt from the feature model showing the effects of the evolution

Naturally, the mapping to the navigational information has to be adapted in accordance with the performed changes. This is done by using the *Split Feature Mapping* step. In the remapping dialog, existing mappings can be redirected from the feature *Maps* to a logical term of the constituent parts. However, for countries it suffices to use only the feature of the owning region as new target of the remapping. The remapping dialog is shown in Figure 5.27 and the changes to the mapping model are displayed in Table 5.14.

Currently, the features describing the mapping material of particular geographical areas are mandatory as their archetype, the feature *Maps*, is mandatory itself. However, it should be
Figure 5.27: Remapping dialog after applying Split Feature to the feature Maps.

Table 5.14: Changes to the mapping model due to applying Split Feature to the feature Maps.

<table>
<thead>
<tr>
<th>Source</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maps</td>
<td>NavMap:Country[Germany]</td>
</tr>
<tr>
<td>Europe</td>
<td>NavMap:Country[Germany]</td>
</tr>
<tr>
<td>Europe</td>
<td>NavMap:Country[Poland]</td>
</tr>
<tr>
<td>NorthAmerica</td>
<td>NavMap:Country[USA]</td>
</tr>
<tr>
<td>NorthAmerica</td>
<td>NavMap:Country[Canada]</td>
</tr>
<tr>
<td>CentralAmerica</td>
<td>NavMap:Country[Mexico]</td>
</tr>
<tr>
<td>CentralAmerica</td>
<td>NavMap:Country[Costa Rica]</td>
</tr>
<tr>
<td>SouthAmerica</td>
<td>NavMap:Country[Argentina]</td>
</tr>
<tr>
<td>SouthAmerica</td>
<td>NavMap:Country[Brazil]</td>
</tr>
<tr>
<td>Asia</td>
<td>NavMap:Country[Japan]</td>
</tr>
<tr>
<td>Asia</td>
<td>NavMap:Country[China]</td>
</tr>
</tbody>
</table>
possible to deselect superfluous mapping modules for the configuration of the car so that the features have to be made optional. For this purpose, the evolution Transform to Optional Feature is applied to all five specialized features at once. This process is visualized in Figure 5.28 in a series of screenshots.

Figure 5.28: Applying Transform to Optional Feature to the features Europe, NorthAmerica, CentralAmerica, SouthAmerica and Asia.

a) screenshot of selecting the menu entry for the evolution

![Figure 5.28 screenshot](image1)

b) excerpt from the feature model showing the effects of the evolution

![Feature Model](image2)

In order to allow combinations of the refined mapping modules, their containing group has to be converted into an or group, which allows selecting an arbitrary number of features but requires at least one. These modifications are performed using the evolution Transform to Or Group on the parent group of the specialized mapping modules. This process is visualized in Figure 5.29.

Figure 5.29: Applying Transform to Or Group to the group containing the refined mapping modules.

a) screenshot of selecting the menu entry for the evolution

![Figure 5.29 screenshot](image3)

b) excerpt from the feature model showing the effects of the evolution

![Feature Model](image4)
With these changes to the product line, it is now possible to make fine-grained selections of the mapping material, which is required for navigation in a particular part of the world. Through these additional configuration options, customers can tailor the setup of the personal navigation device to their requirements and budget.

### 5.3.4 Changing the Implementation of the CD Player

As last modification of the 2011 revision, the implementation of the CD player is modified. Despite ongoing efforts of the research and development department, the in-house MP3 decoder continues to be a source of disappointment due to producing mediocre quality and crashing unexpectedly. Due to this reason, maintenance cost of the MP3 decoder has increased to an immense level. As part of the revision, it was decided to reduce costs and free resources in the development department by licensing the MP3 decoder of an external vendor. As a result, the CD player no longer depends on the original MP3 decoder so that it can be removed from the system. However, the CD player still needs to call functions of the purchased library so that the method object \texttt{MP3Decoder} can be merged into the class \texttt{CDPlayer}. This is done by use of the evolution \textit{Inline Method Object} on the class \texttt{MP3Decoder} in the UML model. The process of applying the evolution is visualized in Figure 5.30.

Figure 5.30: Applying \textit{Inline Method Object} to the class \texttt{MP3Decoder} in the UML model.

![Image](image.png)

The remapping of the evolution moves all mappings of the \texttt{MP3Decoder} to the \texttt{CDPlayer} by using \textit{Move Feature Mapping}. In this particular case, the results of the remapping are semantically equivalent to the original mapping as both classes \texttt{MP3Decoder} and \texttt{CDPlayer} are targets of only the feature \texttt{CDPlayer}. However, the original mapping is moved away from the class \texttt{MP3Decoder}, which was deleted. Thus, the mapping is effectively removed from the mapping model. The remapping dialog after the evolution is shown in Figure 5.31 and the changes to the mapping model are visualized in Table 5.15.

Table 5.15: Changes to the mapping model due to applying \textit{Inline Method Object} to the class \texttt{MP3Decoder} in the UML diagram.

<table>
<thead>
<tr>
<th>MP3CDPlayer</th>
<th>UML:Class[MP3Decoder]</th>
</tr>
</thead>
</table>

By use of this evolution, the realization artifacts have been updated to use the newly licensed
5.3. SECOND REVISION IN 2011

Figure 5.31: Remapping dialog after applying Inline Method Object to the class MP3Decoder in the UML diagram.

MP3 decoder. As a result, the new revision of the MAXI Cheetah offers high quality MP3 playback in the car’s CD player.

5.3.5 Summary of the Changes of the Second Revision in 2011

During the course of the 2011 revision, three major changes to the automotive multimedia product line were performed. For one, the audio and MP3 CD player were merged to a CD player capable of handling both formats. Furthermore, the voice recognition was enhanced to not only control the personal navigation system but also the audio player of the car. As third change, the feature Maps was refined to distinct modules for various geographical areas, which can be selected in arbitrary combinations. Besides these major changes, there also was a change of realization artifacts as the in-house MP3 decoder was replaced with a decoder manufactured by an external vendor. To sum up the effects of these changes on the problem space, the complete feature model of the 2011 revision is displayed in Figure 5.32.

The modifications of the revision also had an effect on various solution space models. As an example, the complete UML model of the years 2011 is displayed in Figure 5.33.
Figure 5.32: The feature model of the automotive multimedia product line at the end of the 2011 revision.

Figure 5.33: The complete UML diagram of the automotive multimedia product line at the end of the 2011 revision.
5.4 Conclusion of the Example Project

During the course of the example project, the applicability of the evolution system of this thesis was demonstrated for a practical scenario. Modifications were performed for the feature tree in the problem space as well as various models in the solution space. In particular, three different types of solution space models were used: UML design models, Java source code and DocBooklet documentation. These models serve as examples of three essential types of artifacts in the software development process. Each of the demonstrated models was subject to change over the course of time so that the possibility arose to apply the evolutions presented in the thesis. The evolutions were used in a number of different scenarios including the creation of new content as well as the modification and removal of existing elements. Where feasible, the evolutions were configured using parameter pages to match the concrete use case. In consequence, only few manual modifications of the models were required to perform the projected alterations. Furthermore, applying the evolutions for the variation type of features to multiple features at once avoided having to use the same evolution multiple times, which proved to increase productivity.

Besides the general usefulness of the evolution system, the applicability of particular evolutions was demonstrated as well. During the course of the thesis, a total of 37 evolutions was introduced. However, 14 evolutions for DocBooklet are largely similar in nature and thus can be grouped. With the 4 groups of DocBooklet evolutions presented in Section 3.1.2.3, this leaves 27 unique modifications. Their application areas reach from the feature tree in the problem space to UML models, Java source code and DocBooklet documentation in the solution space. Out of the 27 unique evolutions, 16 were demonstrated in the example project encompassing all four types of target model. The remaining 11 evolutions were not used because the need for these particular modifications did not arise during the regular course of the example. Thus, it would have been necessary to extend the scope of the project to require additional evolutions. Most likely, the construction of further scenarios would have made the project seem artificial, which is in contrast to the original goal of creating an example that is close to real world application. Furthermore, the evolutions applied during the project suffice to provide an insight into the usage and usefulness of the evolution system so that it was refrained from demonstrating every single introduced evolution. Nonetheless, the remaining 11 evolutions are fully operational and can be applied if the need arises.

During the course of the example project, the evolution system successfully coped with different use cases and multiple evolutions were applied. To overcome the challenges posed by
Table 5.16: Complete feature mapping for the automotive multimedia product line at the end
of the 2011 revision.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioPlayer</td>
<td>UML:Class[AudioPlayer]</td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>UML:Association[UserInterface-AudioPlayer]</td>
</tr>
<tr>
<td>AudioPlayer OR PersonalNavigation</td>
<td>UML:Class[Speakers]</td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>UML:Association[AudioPlayer-Speakers]</td>
</tr>
<tr>
<td>AudioPlayer OR PersonalNavigation</td>
<td>UML:Class[VolumeControl]</td>
</tr>
<tr>
<td>AudioPlayer OR PersonalNavigation</td>
<td>UML:Association[VolumeControl-Speakers]</td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>Manual:Chapter[AudioPlayer]</td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>Java:Class[UIBuilder]:Method[createUI]:Call[2]</td>
</tr>
<tr>
<td>AudioPlayer</td>
<td>Java:Class[UIBuilder]:Method[createAudioPlayerMenu]</td>
</tr>
<tr>
<td>Radio</td>
<td>UML:Class[RadioTuner]</td>
</tr>
<tr>
<td>CDPlayer</td>
<td>UML:Association[AudioPlayer-CDPlayer]</td>
</tr>
<tr>
<td>CDPlayer</td>
<td>Manual:Section[AudioCDPlayback]</td>
</tr>
<tr>
<td>CDPlayer</td>
<td>UML:Association[CDPlayer-MP3Decoder]</td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>UML:Class[OnBoardComputer]</td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>UML:Association[OnBoardComputer-Display]</td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>UML:Association[UserInterface-OnBoardComputer]</td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>Java:Class[UIBuilder]:Method[createUI]:Call[1]</td>
</tr>
<tr>
<td>OnBoardComputer</td>
<td>Java:Class[UIBuilder]:Method[createOnBoardComputerMenu]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Class[PersonalNavigation]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Association[PersonalNavigation-Speakers]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Association[PersonalNavigation-Display]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>UML:Association[UserInterface-PersonalNavigation]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>Java:Class[UIBuilder]:Method[createUI]:Call[3]</td>
</tr>
<tr>
<td>PersonalNavigation</td>
<td>Java:Class[UIBuilder]:Method[createPersonalNavigationMenu]</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>UML:Class[VoiceRecognition]</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>UML:Association[VoiceRecognition-VoiceControllable]</td>
</tr>
<tr>
<td>VoiceRecognition</td>
<td>UML:Class[VoiceControllable]:Method[receiveVoiceCommand]</td>
</tr>
<tr>
<td>Europe</td>
<td>NavMap:Country[Germany]</td>
</tr>
<tr>
<td>Europe</td>
<td>NavMap:Country[Poland]</td>
</tr>
<tr>
<td>NorthAmerica</td>
<td>NavMap:Country[USA]</td>
</tr>
<tr>
<td>NorthAmerica</td>
<td>NavMap:Country[Canada]</td>
</tr>
<tr>
<td>CentralAmerica</td>
<td>NavMap:Country[Mexico]</td>
</tr>
<tr>
<td>CentralAmerica</td>
<td>NavMap:Country[Costa Rica]</td>
</tr>
<tr>
<td>SouthAmerica</td>
<td>NavMap:Country[Argentina]</td>
</tr>
<tr>
<td>SouthAmerica</td>
<td>NavMap:Country[Brazil]</td>
</tr>
<tr>
<td>Asia</td>
<td>NavMap:Country[Japan]</td>
</tr>
<tr>
<td>Asia</td>
<td>NavMap:Country[China]</td>
</tr>
</tbody>
</table>
the project, it was not necessary to modify the implementation presented in Section 4.2. Due to these points and the extensive nature of the example, the evolution system is regarded as being applicable to real world scenarios.
Chapter 6

Conclusion

This final chapter concludes the work of the thesis. First, a summary of the theoretical and practical part as well as the example project will be presented. Then, limitations of the evolution system at its current stage of development will be discussed. After that, possibilities for future work will be introduced. Finally, the chapter will close with an explanation of the theoretical and practical contributions of the thesis.

6.1 Summarized Findings

The theoretical work featured extensive explanations of a wide variety of evolutions for various target models. For one, there are 14 evolutions defined in the problem space, which modify the feature tree. Furthermore, three different types of models were targeted in the solution space, which serve as examples of typical artifacts of the software development process. For UML models, 5 evolutions were provided. For the Java programming language, 4 modifications were defined and for the DocBooklet documentation format, 14 evolutions were specified, which can be put into 4 groups. This wide variety of different evolutions provides a basis for the modification of software product lines.

Besides introducing individual evolutions, a number of classification systems for evolutions was discussed. It was argued that grouping evolutions by whether they maintain behavior or not is unsatisfactory when trying to capture the effect on the product line. Furthermore, a variety of additional classification systems was found to be equally unsuitable for this purpose for various reasons. In consequence, a new classification was introduced to capture the effect of evolutions on a software product line, which features two major groups of evolutions. Intraspacial evolutions have an effect that is contained within the space the evolution was triggered from so that the mapping of the product line is not affected. On the other hand, interspatial evolutions perform modifications whose logical extent reaches beyond the originating space. The latter group is further divided into interspatial evolutions of the first and second degree. Evolutions of the first degree affect the originally targeted space whereas evolutions of the second degree modify both spaces. Additionally, each interspatial evolution affects the mapping, which connects the adjacent spaces. Thus, for interspatial evolutions, the mapping has to be adapted in order to maintain the consistency of the product line.

For this purpose, a total of 8 remapping operators was specified. As the remapping procedure
differs in the problem and solution space, individual operators for either side were provided. In the problem space, there are operators to move, copy and remove as well as to split and merge mappings. For the solution space, operators to move, copy and remove were defined analog to those in the problem space. Finally, all presented evolutions were inspected to identify them as either intraspatial or interspatial evolutions so that they could be attributed to their respective group in the classification system. For interspatial evolutions, the individual steps necessary for remapping were explained in detail. With the provided remapping steps, it is possible to apply evolutions to software product lines without jeopardizing the consistency of the mapping.

In addition to the theoretical part of the thesis, the practical realization of the evolution system was discussed. An architectural overview of the implementation provided information on the structure of the subsystems. For each of the subsystems, individual classes with their responsibilities were explained. Furthermore, the test suite used to validate the implementation was discussed. Within the remapping subsystem, the object remapping specification was introduced. It allows to adapt evolutions to the use within software product lines even if the original modification was not designed for this purpose. Finally, the practical part gave detailed instructions on how to extend the evolution system with further modifications for the problem as well as the solution space if the need should arise.

Both the theoretical and practical results were put to test in an extensive example project, which described the change of an automotive multimedia system over the course of time. The majority of the presented evolutions was applied to perform the required modifications. Each of the evolutions was executed successfully and a modification of the existing implementation was not required. Besides the evolutions, only few manual modifications were required. Overcoming the challenges posed by the example project showed the applicability and usefulness of the evolution system.

6.2 Limitations and Drawbacks

Even though the evolution system was shown to be useful for real world scenarios, there are a few drawbacks that limit the applicability of the system. For example, the presented evolutions were designed to target merely a single solution space model at a time. This suffices for most practical applications but may cause problems if there are models in the same space that reference one another. For example, a UML model may be accompanied by a UML diagram model, which contains the information to visualize the original model in the graphical UML representation. Evolutions such as \textit{Duplicate Class} currently only affect the UML model but not the diagram representation. In consequence, the duplicated class is not shown in the diagram representation even if the original class is present. In a concrete case, this problem can be solved by implementing the additional steps of modification for the dependent model in the respective evolutions.

Another limitation exists with the use of the object remapping specification. When adapting a modification specified by Refactory for the use in a software product line, the expressiveness of the ORSpec may not be sufficient to perform an adequate remapping. The problem stems from the fact that only objects that play a particular role in Refactory can be targeted but not their descendants. For instance, if an element is deleted, its mapping should be removed along
with it. Specifying that the mapping of its descendants should be deleted as well is not possible as Refactory only provides the objects playing at least one of the specified roles. An equivalent situation may appear if a particular element is duplicated and its mapping should be copied to the clone including the respective descendants. However, this type of problem currently is of no practical consequence as Refactory allows to only delete elements that do not have descendants and it does not provide a copy operator.

The most severe limitation of the evolution system arises with models that identify elements of the model by their position in the containment hierarchy. For example, in DocBooklet a particular section might be targeted as the third section of the second chapter. All EMFText languages use this mechanism as their textual syntax does not necessarily allow to specify explicit IDs. Furthermore, other types of models may be affected as well. When performing a modification that alters the order of elements on one such model, the IDs have to be adapted to the changes. This may happen when adding or removing but also when reordering elements as with the Move Up Chapter evolution for DocBooklet. As the position of an element in the containment hierarchy was changed, it has a different ID after the modification and all references using this ID have to be updated.

For a single memory instance of a model, EMF adapts references to IDs according to the performed modifications automatically. However, if there are multiple instances of the same model, IDs and references may become inconsistent. Unfortunately, the evolution system is required to deal with two separate instances of the same solution space model. The first instance of the solution space model is held by FeatureMapper, which uses it as target of the mapping. The second instance of the very same model belongs to the specific solution space editor (e.g., UML editor). When the evolution is triggered from the editor, the modifications are performed on the respective instance of the model held by the editor. The instance of FeatureMapper is updated after the model was saved by unloading and then reloading the resource of the model.

During this process, the mapping targets of the solution space model are converted to proxy objects holding the URI of their targeted elements, which (in part) consists of an element’s ID. After the solution space model was reloaded, the proxies are resolved to objects using the stored URIs, which have not been altered in accordance to the performed changes. In consequence, targets of the mapping model may be invalid as they point to the wrong or a no longer existing element. For example, the Remove Section evolution might delete the aforementioned third section in the second chapter of a DocBooklet model. As part of the evolution, the mapping of the deleted section will be removed as well. However, mappings to the rest of the document may cause problems. For example, if there were a fourth and fifth section in the second chapter, they would now be the third and fourth section respectively. Thus, mappings to the old fourth section would now target the old fifth section and mappings to the old fifth section would now be invalid as no new fifth section exists. The latter harms the mapping model and prevents FeatureMapper from saving, which informs the user of the problem. However, in the first case, an unintended mapping is introduced to the mapping model by mistake, which is particularly dangerous as it can easily go unnoticed.

A solution to the problem would be to refrain from using the position of an element for identification. Unfortunately, not all solution space models permit using unique IDs (e.g., the textual languages of EMF Text). One possibility to overcome this issue would be to obtain IDs
from an external provider that has knowledge of the targeted type of model and thus can use other characteristics for identification than the position of an element. For the described case of DocBooklet, it might be possible to use a combination of the title and content of a section as ID. Until a solution to the described problem has been conceived and implemented, special caution has to be applied when evolving models that use the position of elements for identification.

6.3 Possibilities for Future Work

In addition to the functionality provided by the evolution system, further features may be implemented in the future. For example, it would be useful to check the logical terms created by remapping operators for satisfiability. For this purpose, the OWL checker of FeatureMapper might be employed to assess under which circumstances a term evaluates to true. The results could be used to inform users about a term being a tautology or to provide instructions on how to correct an invalid expression in order to make it satisfiable.

In addition, there are further extensions to the evolution system that can be conceived. Currently, the system suffers from the fact that Refactory provides objects for certain roles only after the configuration wizard was shown. Once this limitation is overcome, new features can be implemented. With the knowledge of concrete objects for certain roles before the configuration step, it would be possible to improve existing evolutions. For instance, it would be possible to suggest meaningful names for features related to their original counterpart in evolutions such as Duplicate Feature or Split Feature. Furthermore, entirely new evolutions could be created that currently cannot be implemented due to the lack of knowledge of concrete objects for roles. One such case is the Push Down Feature evolution, which can move a selected feature downwards in the hierarchy. Currently, the evolution cannot be implemented because it requires to specify a target located below the selected feature, which requires knowledge of the concrete object playing the respective role. Thus, implementing the functionality for Refactory to provide objects for roles before the configuration wizard is displayed would allow various extensions of the evolution system.

Another possible extension is concerned with the user interface metaphor employed for the remapping procedure. Currently, the evolution system uses a dialog based user interface for remapping after an interspatial evolution. This type of user interaction integrates seamlessly with the configuration wizard provided by Refactory. However, FeatureMapper generally uses a more visual approach as user interface metaphor. For instance, the procedure to create a new mapping lets users select solution space elements directly from a custom editor even if it displays a graphical syntax such as the TopCased editor with UML. Furthermore, the dialog based user interface of the evolution system does not always seem sufficient for a convenient remapping procedure. Problems arise especially when elements participating in a remapping procedure do not produce an expressive textual representation that can be used to distinguish them from other objects of the same type. For example, it is hard to distinguish two associations in a UML model if no significant name was provided for them. A visual approach to remapping might remedy this problem. Yet, it remains to be determined how the possibilities for configuration provided by the remapping operators could be expressed more visually. For remapping operators to move, copy or remove a mapping, the solution seems elementary as affected elements merely
have to be selected or deselected. However, the Split Feature Mapping operator provides a wide variety of possibilities for user interaction such as choosing effective targets or the concatenation operator for the split procedure, which can not be translated to a visual interaction metaphor directly. Finding a solution to this problem is non trivial and thus requires future work.

Besides the suggested extensions for the evolution system, new requirements and use cases are likely to yield further areas for future work. The implementation of the system aims at being easily extensible so that the evolution system can be adapted to new requirements quickly.

6.4 Theoretical and Practical Contributions

The survey of work related to this thesis in Section 2.3 suggests that a system for the evolution of software product lines similar to the presented one has not yet been conceptualized or implemented. Thus, the thesis provides a number of contributions on the theoretical as well as the practical level.

In the theoretical part, a wide variety of different evolutions for the problem as well as the solution space has been presented. Besides a number of evolutions conceived originally for this thesis, many modifications were inspired by the work in various different sources. The contribution of the thesis is to assemble a comprehensive list and to provide a uniform explanation of the effects of each evolution. Besides the presentation of the evolutions, the introduced classification system is one of the most important contributions of the thesis. The classification distinguishes intraspatial and interspatial evolutions to capture the effects of an evolution on a software product line. Knowing the group an evolution belongs to directly states whether the modification requires a subsequent remapping in order to keep the mapping of a product line consistent. For the group of interspatial evolutions, which requires the mapping to be adapted, a number of remapping operators is provided. Seemingly, a similar concept to make evolutions applicable to software product lines has not yet been conceived. The final theoretical contribution of the thesis results from attributing the presented evolutions to their respective categories in the classification system. As a number of evolutions has been identified as being interspatial in nature, the concrete steps required for remapping have been determined and documented so that the evolutions can be used in the context of a software product line.

Besides the theoretical part, the implementation of the evolution system provides a number of practical contributions. Most notably, it is possible to apply evolutions to elements in a software product line without jeopardizing the consistency of the mapping due to a remapping procedure subsequent to interspatial evolutions. Furthermore, the evolution system is integrated into FeatureMapper, which is an established tool for the work with software product lines. The capabilities of the tool to handle arbitrary solution space models generically is directly reflected in the evolution system, which can handle the same type of models. In order to demonstrate the system, a variety of different evolutions has been presented for three types of models essential to the software development process. Further evolutions can easily be added to extend the system. With this mechanism, it is further possible to target entirely new types of models that have not yet been discussed. Thus, the evolution system is very versatile and can easily be adapted to changing requirements. To achieve genericity, the evolution system employs the tool Refactory, which has not been used for the evolution of software product lines before. Due to
this fact, the modifications provided by Refactory are not directly applicable to software product lines without risking to damage the mapping. The thesis presented the concept of an object remapping specification, which provides a textual language that can be used to add remapping support to modifications that have not been conceived for the use in the context of a software product line. Through the capabilities of the evolution system, it is possible to employ existing modifications as well as entirely new evolutions in order to alter the elements in a software product line without jeopardizing the consistency of the mapping.

Finally, the practical relevance of the system was shown by the example project in Section 5. Monitoring the automotive multimedia system of the example over the course of time demonstrated that model-based software product lines are subject to change similar to any other software system. Thus, altering the product line is necessary to adapt to ever changing requirements. However, manual modifications of a product line are both tedious and error prone as unintended damage to the mapping is hard to avoid. Therefore, using the evolution system presented in this thesis, which maintains the mapping, is not only a valuable aid but an essential part in the work with model-based software product lines.
Appendix A

Object Remapping Specification (*.orspec)

A.1 Object Remapping Specification Model

A.2 Object Remapping Specification Syntax

```
SYNTAXDEF orspec
FOR <http://www.tudresden.de/objectremappingspecification>
START ObjectRemappingSpecification

IMPORTS {
}

OPTIONS {
  reloadGeneratorModel = "true";
  generateCodeFromGeneratorModel = "true";
}

TOKENS {
```

APPENDIX A. OBJECT REMAPPING SPECIFICATION (*.ORSPEC)

```plaintext
DEFINE REMAPPING_OPTIONS $'OPTIONAL'|'SELECTABLE_TARGETS'$;
}

TOKENSTYLES {
    "REMAPPING_OPTIONS" COLOR #0193CF, BOLD, ITALIC;
    "OBJECT" COLOR #1C6AB3, BOLD;
    "REMOVING" COLOR #1C6AB3, BOLD;
    "COPY" COLOR #1C6AB3, BOLD;
    "MOVE" COLOR #1C6AB3, BOLD;
    "REMOVE" COLOR #1C6AB3, BOLD;
    "MAPPING" COLOR #1C6AB3, BOLD;
    "TEXT" COLOR #AA1D7D;
}

RULES {
    ObjectRemappingSpecification ::= "OBJECT" "REMOVING" "FOR"
        roleModelName['<','>'] !0!0 "STEPS" #1 "{" #1 instructions+ : !0 "}"
        
    CopyObjectMappingInstruction ::= "COPY" #1 "MAPPING" ":" #1 sources[TEXT]
        ("," #1 sources[TEXT])* #1 "=" #1 targets[TEXT] ("," #1 targets[TEXT])* 
        (#1 "{" #1 ("description" ":" #1 description['"','"] ;" !0)
        ("options" ":" #1 options[REMAPPING_OPTIONS]
        ("," #1 options[REMAPPING_OPTIONS])* ";" !0)
        
    MoveObjectMappingInstruction ::= "MOVE" #1 "MAPPING" ":" #1 sources[TEXT]
        ("," #1 sources[TEXT])* ":=" targets[TEXT] ("," #1 targets[TEXT])* 
        (#1 "{" #1 ("description" ":" #1 description['"','"] ;" !0)
        ("options" ":" #1 options[REMAPPING_OPTIONS]
        ("," #1 options[REMAPPING_OPTIONS])* ";" !0)
        
    RemoveObjectMappingInstruction ::= "REMOVE" #1 "MAPPING" ":" #1 sources[TEXT]
        ("," #1 sources[TEXT])* (#1 "{" #1 ("description" ":" #1 description['"','"] ;" !0)
        ("options" ":" #1 options[REMAPPING_OPTIONS]
        ("," #1 options[REMAPPING_OPTIONS])* ";" !0)
        
```
Appendix B

DocBooklet (*.docbooklet)

B.1 DocBooklet Model

B.2 DocBooklet Syntax

```
SYNTAXDEF docbooklet
FOR <http://www.emftext.org/language/docbooklet>
START Book

IMPORTS {
}

TOKENSTYLES {
  "<?" COLOR #9292c9, BOLD;
  "?>" COLOR #9292c9, BOLD;
  "=" COLOR #000000;
  "<book>" COLOR #257a25, BOLD;
}
```
"</book>" COLOR #257a25, BOLD;
"<chapter>" COLOR #257a25, BOLD;
"</chapter>" COLOR #257a25, BOLD;
"<section>" COLOR #257a25, BOLD;
"</section>" COLOR #257a25, BOLD;
"<para>" COLOR #257a25, BOLD;
"</para>" COLOR #257a25, BOLD;
"<title>" COLOR #257a25, BOLD;
"</title>" COLOR #257a25, BOLD;
}

RULES {
  Book ::= "<?" "xml" #1 "version" "=" """\"1.0\"" "?" #1
         "encoding" "=" "\"utf-8\"" "?>" !0 "<book>" !1
         "<title>" title["""","] "</title>" !1 chapters+ !0 "</book>" !0;
  Chapter ::= "<chapter>" !1 "<title>" title["""","] "</title>" !0
             contents+ !0 "</chapter>" !0;
  Section ::= "<section>" !1 "<title>" title["""","] "</title>" !0
             contents+ "</section>" !0;
  Paragraph ::= "<para>" !1 text["""","] !0 "</para>" !0;
}
Appendix C

NavMap (*.navmap)

C.1 NavMap Model

C.2 NavMap Syntax

```
SYNTAXDEF navmap
FOR <http://www.emftext.org/language/navmap>
START NavMap

IMPORTS {
}

TOKENSTYLES {
  "<" COLOR #9292c9, BOLD;
  ">" COLOR #9292c9, BOLD;
  "=" COLOR #000000;
  "<navmap" COLOR #257a25, BOLD;
  ">" COLOR #257a25, BOLD;
  "</navmap>" COLOR #257a25, BOLD;
  "<country" COLOR #257a25, BOLD;
  "</country>" COLOR #257a25, BOLD;
  "<street" COLOR #257a25, BOLD;
```
APPENDIX C. NAVMAP (*.NAVMAP)

"/" COLOR #257a25, BOLD;
"<highway" COLOR #257a25, BOLD;

"name" COLOR #ad0014, BOLD;
}

RULES {
  NavMap ::= "<?" #1 "xml" #1 "version" "=" """#1.0"" #1 "encoding" "=" ""utf-8"" #1 "?>" !0
  "<navmap" #1 "name" "=" name[""",""] "">" countries+ !0 "</navmap>";
  Country ::= !1 "<country" #1 "name" "=" name[""",""] "">" structures+ !0
    "</country>";
  Street ::= !1 "<street" #1 "name" "=" name[""",""] "/"";
  Highway ::= !1 "<highway" #1 "name" "=" name[""",""] "/"";
}
List of Figures

2.1 Example of a feature tree with annotated edges. ........................................ 8
2.2 The Meta Object Facility layers including example content. ....................... 10
2.3 Example of a cardinality-based feature tree. .............................................. 12
2.4 UML diagram of the different types of mappings in FeatureMapper. ............... 13

3.1 Example of the evolutions for the variation type of features. ..................... 22
3.2 Example of a feature tree before and after the Insert Feature evolution. ........ 22
3.3 Example of a feature tree before and after the Duplicate Feature evolution. ..... 23
3.4 Example of a feature tree before and after the Split Feature evolution. .......... 24
3.5 Example of a feature tree before and after the Merge Features evolution with both types of merge. ................................................................. 25
3.6 Example of a feature tree before and after the Pull Up Feature evolution. ........ 26
3.7 Example of a feature tree before and after the Remove Feature evolution. ........ 26
3.8 Example of a feature tree and UML model before the Remove Feature and Owned Assets evolution. ................................................................. 27
3.9 Example of a feature tree and UML model after the Remove Feature and Owned Assets evolution. ................................................................. 28
3.10 Example of a feature tree before and after the Transform to Or Group evolution. 29
3.11 Example of a feature tree before and after the Transform to And Group evolution. 29
3.12 Example of a feature tree before and after the Transform to Alternative Group evolution. ................................................................. 29
3.13 Example of a feature tree before and after the Duplicate Group evolution. .... 30
3.14 Example of a feature tree before and after the Merge Groups evolution. ......... 31
3.15 Example of a UML diagram before and after the Duplicate Class evolution. ..... 32
3.16 Example of a UML diagram before and after the Extract Super Class evolution. 33
3.17 Example of a UML diagram before and after the Replace Method with Method Object evolution. ................................................................. 33
3.18 Example of a UML diagram before and after the Inline Method Object evolution. 34
3.19 Example of a UML diagram before and after the Remove Class evolution. ....... 34
3.20 Visualization of intraspatial evolution types. ............................................. 47
3.21 Visualization of interspatial evolution types of the first degree. ................. 48
3.22 Visualization of interspatial evolution types of the second degree. ............... 49
3.23 Decision diagram to find the appropriate group for evolutions in the classification by semantical extent of model changes. ............................................. 51
3.24 UML diagram as example of a solution space model for Split Feature Mapping. . 57
4.1 Architectural overview of the major systems of the implementation. .......... 79
4.2 UML diagram of the most common classes for evolution. .................. 80
4.3 UML diagram showing a subset of the evolutions with their respective evolution adapters. ......................................................... 81
4.4 UML diagram of the classes for evolutions in the problem space. .......... 83
4.5 Parameter pages for the problem space evolutions requiring user input. .... 84
4.6 UML diagram of the classes for generic evolutions in the solution space. ... 86
4.7 UML diagram of the classes for evolutions of UML models in the solution space. 88
4.8 UML diagram of the classes for evolutions of Java source code in the solution space. 89
4.9 UML diagram of the classes for evolutions of DocBooklet documents in the solu-
... tion space. ............................................................................. 90
4.10 UML diagram of remapping plan and remapping step with its specializations for the separate spaces. ......................................................... 90
4.11 UML diagram of the classes affected by remapping in the problem space. .... 92
4.12 UML diagram of the feature remapping steps used for remapping in the problem space. ................................................................. 93
4.13 UML diagram of the object remapping steps used for remapping in the solution space. ................................................................. 96
4.14 Overview of the classes involved in translating and executing an ORSpec. .... 100
4.15 UML diagram of the essential classes of the user interface subsystem. ... 102
4.16 Visualization of the regions of a remapping dialog created by the specializations of AbstractRemappingPlanUIBuilder and AbstractRemappingStepUIBuilder. 103
4.17 UML diagram of the classes used for test cases. ............................... 104
5.1 Feature tree of the automotive multimedia product line in 2001. ................. 115
5.2 UML diagram for the automotive multimedia product line in 2001. .......... 117
5.3 Applying Remove Feature and Owned Assets to the feature CassettePlayer. 121
5.4 Remapping dialog after applying Remove Feature and Owned Assets to the feature CassettePlayer. ....................................................... 122
5.5 Applying Replace Method with Method Object to the method readCDText() of the class CDPlayer in the UML diagram. ......................... 123
5.6 Remapping dialog after applying Replace Method with Method Object to the method readCDText() of the class CDPlayer in the UML diagram. 123
5.7 Applying Duplicate Section to the section “CD Playback” in the user manual. 124
5.8 Remapping dialog after applying Duplicate Section to the section “CD Playback” in the user manual. ...................................................... 125
5.9 Applying Split Feature to the feature CDPlayer. .................................. 126
5.10 Remapping dialog after applying Split Feature to the feature CDPlayer. .... 127
5.11 Applying Transform to Alternative Group to the group of the specialized CD players. ................................................................. 127
5.12 Applying Duplicate Feature to the feature AudioPlayer. .................... 130
5.13 Remapping dialog after applying Duplicate Feature to the feature AudioPlayer. 131
5.14 Excerpt from the feature model showing the newly created features VoiceRecognition and Maps. ................................................................. 131
5.15 Excerpt from the UML diagram showing the elements relevant to the personal navigation system. ................................................................. 132
5.16 Applying Extract Method to the code fragments in the createUI() method of the class UIBuilder in Java source code. ................................. 134
5.17 The feature model of the automotive multimedia product line at the end of the 2006 revision. ................................................................. 135
5.18 The complete UML diagram of the automotive multimedia product line at the end of the 2006 revision. .................................................. 137
5.19 Applying Remove Class to the class CDTextReader of the UML diagram. .... 139
5.20 Remapping dialog after applying Remove Class to the class CDTextReader of the UML diagram. ................................................................. 139
5.21 Applying Merge Features to the features AudioCDPlayer and MP3CDPlayer. . 140
5.22 Remapping dialog after applying Merge Features to the features AudioCDPlayer and MP3CDPlayer. ................................................................. 141
5.23 Applying Extract Super Class to the method receiveVoiceCommand() of the class PersonalNavigation in the UML diagram. ......................... 142
5.24 Remapping dialog after applying Extract Super Class to the method receiveVoiceCommand() of the class PersonalNavigation in the UML diagram. .... 143
5.25 Applying Pull Up Feature to the feature VoiceRecognition. .................. 144
5.26 Applying Split Feature to the feature Maps. ...................................... 145
5.27 Remapping dialog after applying Split Feature to the feature Maps. ........ 146
5.28 Applying Transform to Optional Feature to the features Europe, NorthAmerica, CentralAmerica, SouthAmerica and Asia. ................................. 147
5.29 Applying Transform to Or Group to the group containing the refined mapping modules. ................................................................. 147
5.30 Applying Inline Method Object to the class MP3Decoder in the UML model. . 148
5.31 Remapping dialog after applying Inline Method Object to the class MP3Decoder in the UML diagram. ................................................................. 149
5.32 The feature model of the automotive multimedia product line at the end of the 2011 revision. ................................................................. 149
5.33 The complete UML diagram of the automotive multimedia product line at the end of the 2011 revision. .................................................. 150
List of Tables

3.1 Example of Move Feature Mapping. ........................................ 54
3.2 Example of Copy Feature Mapping. ........................................ 55
3.3 Example of Remove Feature Mapping. ................................. 56
3.4 Example of copying and then removing a feature mapping. .... 56
3.5 Example of Split Feature Mapping. ................................. 58
3.6 Example of Merge Feature Mapping. ........................................ 59
3.7 Example of Move Object Mapping. ........................................ 61
3.8 Example of Copy Object Mapping. ........................................ 62
3.9 Example of Remove Object Mapping. ........................................ 62
3.10 Overview of classification and remapping of problem space evolutions. ........ 64
3.11 Overview of classification and remapping of UML evolutions. ........ 67
3.12 Overview of classification and remapping of Java evolutions. .......... 69
3.13 Overview of classification and remapping of DocBooklet evolutions. .... 70

5.1 Complete feature mapping for the automotive multimedia product line of 2001. . 119
5.2 Changes to the mapping model due to applying Remove Feature and Owned Assets to the feature CassettePlayer. ......................... 120
5.3 Changes to the mapping model due to applying Replace Method with Method Object to the method readCDText() of the class CDPlayer in the UML diagram. 123
5.4 Changes to the mapping model due to applying Duplicate Section to the section “CD Playback” in the user manual including manual content adaptation. .... 125
5.5 Changes to the mapping model due to applying Split Feature to the feature CDPlayer. ..................................................... 126
5.6 Changes to the mapping model due to applying Duplicate Feature to the feature AudioPlayer. ..................................................... 128
5.7 Manual changes to the mapping model to incorporate elements specific to the personal navigation system. ................................. 133
5.8 Changes of the mapping model as result of applying Extract Method to the code fragments in the createUI() method of the class UIBuilder in Java source code. 135
5.9 Complete feature mapping for the automotive multimedia product line at the end of the 2006 revision. ..................................................... 136
5.10 Changes to the mapping model due to applying Remove Class to the class CDTextReader of the UML diagram. ..................................................... 138

171
LIST OF TABLES

5.11 Changes to the mapping model due to applying *Merge Features* to the features
AudioCDPlayer and MP3CDPlayer. .............................................. 142

5.12 Changes to the mapping model due to applying *Extract Super Class* to the method
receiveVoiceCommand() of the class PersonalNavigation in the UML diagram. 144

5.13 Manually modified mapping for the class VoiceControllable in the UML diagram. 144

5.14 Changes to the mapping model due to applying *Split Feature* to the feature Maps. 146

5.15 Changes to the mapping model due to applying *Inline Method Object* to the class
MP3Decoder in the UML diagram. .............................................. 148

5.16 Complete feature mapping for the automotive multimedia product line at the end
of the 2011 revision. ................................................................. 152
List of Listings

3.1 Example of Java source code before and after the Insert Parameter evolution. 35
3.2 Example of Java source code before and after the Rename Element evolution. 35
3.3 Example of Java source code before and after the Extract Method evolution. 36
3.4 Example of Java source code before and after the Remove Statement evolution. 37
3.5 Example of a DocBooklet file. 38
3.6 Example of a DocBooklet file before and after the Duplicate Section evolution. 39
3.7 Example of a DocBooklet file before and after the Move Up Section evolution. 40
3.8 Example of a DocBooklet file before and after the Extract Section evolution. 42
3.9 Example of a DocBooklet file before and after the Remove Section evolution. 43
4.1 Complete role model for ExtractX. 75
4.2 Complete refactoring specification for ExtractX. 76
4.3 Complete role mapping of ExtractX as Extract Super Class for the UML metamodel. 77
4.4 Java source code excerpt from AbstractProblemSpaceEvolution showing the five steps required for problem space evolutions. 82
4.5 Java source code excerpt from AbstractSolutionSpaceEvolution showing the six steps required for solution space evolutions. 86
4.6 Example of an ORSpec showing the different remapping instructions and their modifiers. 99
4.7 Example of a minimalistic role model. 105
4.8 Example of a minimalistic refactoring specification. 106
4.9 Example of a role mapping for Duplicate Feature. 106
4.10 Principle structure of the implementation of the evolution adapter for Duplicate Feature. 107
4.11 Principle structure of the implementation of the Duplicate Feature evolution. 108
4.12 Principle structure of the parameter page for Duplicate Feature. 109
4.13 Complete implementation of the evolution adapter for Duplicate Feature. 110
4.14 ORSpec for the generic interspatial evolution ExtractXWithReferenceClass. 111
5.1 Java source code of the method createUI() of the class UIBuilder of the year 2001. 117
5.2 Shortened version of the MAXI Cheetah user manual of the year 2001. 118
5.3 Exemplary excerpt from the NavMap file provided with the personal navigation device. 129
5.4 Manually modified version of the Java source code of the method createUI() of the class UIBuilder. 132
Bibliography


[GHJV95] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.


Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, August 31, 2011