Slovak University of Technology in Bratislava
Faculty of Informatics and Information Technologies

Valentino Vranić

Multi-Paradigm Design with Feature Modeling

Školiteľ: doc. Ing. Mária Bieliková, PhD.

Dizertačná práca na získanie
vedecko-akademickej hodnosti philosophiae doctor
v odbore doktorandského štúdia:
25-31-9 Programové a informačné systémy

April 2004

To Brana and Sandra

Abstract

Based on the analysis of multi-paradigm software development and the concept of paradigm, a new method of multi-paradigm design with feature modeling is proposed in this thesis. The method enables an explicit reasoning about paradigms, viewed as solution domain concepts, and their appropriateness for given application domain concepts. Both application and solution domain are modeled using a conceptual modeling technique known as feature modeling adapted to the needs of multi-paradigm design. The process of paradigm selection is defined also in terms of feature modeling as a bottom-up paradigm instantiation over application domain concepts. Its output is a set of paradigm instances annotated with the information about corresponding application domain concepts and features. According to these paradigm instances, the code skeleton is being designed. The method is demonstrated and evaluated on the solution domain of AspectJ programming language and the application domain of feature modeling.

Acknowledgments

I would like to thank Pavol Návrat for directing me to the exciting fields of multi-paradigm design and aspect-oriented programming. His advices helped me crisp my ideas into the early versions of this thesis. I am deeply thankful to Mária Bieliková for her valuable suggestions which helped me putting the thesis into its final form. I am also very thankful to Peter Dolog for the numerous discussions we had during the early stages of this work.

Contents

1 Introduction **1**
1.1 Thesis Objectives 1
1.2 Thesis Structure 2

2 The Concept of Paradigm in Software Development **5**
2.1 The Meaning of Paradigm 5
2.2 Large-Scale Paradigms 6
2.3 Small-Scale Paradigms 7

3 Towards Multi-Paradigm Software Development **11**
3.1 Beyond Object-Oriented Programming 11
3.2 Aspect-Oriented Approaches 12
3.2.1 Aspect-Oriented Programming 13
3.2.2 Adaptive Programming 13
3.2.3 Composition Filters 15
3.2.4 Subject-Oriented Programming 15
3.3 Generative Programming 17
3.4 Multi-Paradigm Programming in Leda 18
3.4.1 Leda Programming Language 19
3.4.2 A Multi-Paradigm Design Method for Leda 19
3.4.3 Leda Approach Evaluation 20
3.5 Multi-Paradigm Design 21
3.5.1 Application Domain SCVR Analysis 21
3.5.2 Solution Domain SCVR Analysis 22
3.5.3 Transformational Analysis and Code Design 22
3.5.4 Multi-Paradigm Design Evaluation 23
3.6 Intentional Programming 24
3.7 Summary 25

4 Feature Modeling for Multi-Paradigm Design **27**
4.1 Basic Notions 28
4.2 Feature Diagrams 29
4.3 Concept References 31

4.4 Information Associated with Concepts and Features 32
4.4.1 Binding Time/Mode 32
4.4.2 Associated Information Applicability 33
4.5 Constraints and Default Dependency Rules 34
4.5.1 Constraints 35
4.5.2 Default Dependency Rules 36
4.6 Parameterization in Feature Models 36
4.6.1 Parameterized Feature and Concept Names 37
4.6.2 Parameterized Concepts 37
4.6.3 Representing Cardinality in Feature Models 38
4.7 Concept Instantiation 39
4.8 Equivalent and Normalized Feature Diagrams 40
4.9 Applying Feature Modeling 41
4.10 Feature Modeling Tool Support 44
4.10.1 ASADAL 44
4.10.2 AmiEddi 45
4.10.3 Captain Feature 45

5 Multi-Paradigm Design with Feature Modeling 47
5.1 The Process of Multi-Paradigm Design with Feature Modeling 47
5.2 Solution Domain Feature Modeling 49
5.2.1 Identifying Paradigms 50
5.2.2 Identifying Binding Times 50
5.2.3 First-Level Paradigm Model 51
5.2.4 Modeling Individual Paradigms 52
5.3 Transformational Analysis 56
5.3.1 Paradigm Instantiation 57
5.3.2 The Process of Transformational Analysis 58
5.4 Code Skeleton Design 60
5.5 Method Evaluation 61
5.5.1 Application Domain Feature Modeling 62
5.5.2 Solution Domain Feature Modeling 62
5.5.3 Transformational Analysis and Code Skeleton Design . 63

6 Related Approaches 65
6.1 Feature Modeling Techniques 65
6.1.1 Concept Instantiation 66
6.1.2 Concept Instances Represented by Feature Diagrams . 66
6.1.3 Concept References 67
6.1.4 Parameterization in Feature Models 67
6.1.5 Constraints and Default Dependency Rules as Logical Expressions 67
6.1.6 Referring to Concepts and Features 68
6.1.7 Representing Cardinalities 68

6.1.8 Information Associated with Concepts and Features . 69
6.2 Multi-Paradigm Approaches . . . 70
6.2.1 Multi-Paradigm Design . . . 70
6.2.2 Multi-Paradigm Programming in Leda . . . 73
6.2.3 Generative Programming . . . 73

7 Conclusions **75**
7.1 Summary of Contributions . . . 76
7.2 Further Work . . . 77

Bibliography **79**

A Domain of Feature Modeling **A-1**
A.1 Domain Definition and Scope . . . A-1
A.2 Feature Model . . . A-2
A.2.1 Feature Model . . . A-3
A.2.2 Feature Diagram . . . A-4
A.2.3 Node . . . A-7
A.2.4 Feature . . . A-8
A.2.5 Partition . . . A-10
A.2.6 Associated Information . . . A-11
A.2.7 AI Item . . . A-12
A.2.8 AI Value . . . A-13
A.2.9 Constraint . . . A-14
A.2.10 Default Dependency Rule . . . A-14
A.2.11 Link . . . A-15
A.2.12 <Plural Form> . . . A-16

B MPD_{FM} for AspectJ **B-1**
B.1 AspectJ Paradigm Identification . . . B-1
B.2 AspectJ Binding Times . . . B-2
B.3 AspectJ Paradigms . . . B-3
B.3.1 AspectJ Program . . . B-3
B.3.2 Class . . . B-4
B.3.3 Interface . . . B-7
B.3.4 Aspect . . . B-9
B.3.5 Inheritance . . . B-12
B.3.6 Method . . . B-13
B.3.7 Overloading . . . B-16
B.3.8 Pointcut . . . B-17
B.3.9 Inter-Type Declaration . . . B-19
B.3.10 Advice . . . B-21
B.4 Auxiliary Concepts . . . B-23
B.4.1 Access . . . B-23

B.4.2 Inheritance Line B-24
B.4.3 Type B-26
B.4.4 <Plural Form> B-27

C Applying $\textbf{MPD}_{\textbf{FM}}$ for AspectJ **C-1**
C.1 Transformational Analysis C-1
C.1.1 Feature Model C-1
C.1.2 Feature Diagram C-2
C.1.3 Node C-3
C.1.4 Feature C-5
C.1.5 Partition C-6
C.1.6 Associated Information C-7
C.1.7 AI Item C-8
C.1.8 AI Value C-8
C.1.9 Constraint C-9
C.1.10 Default Dependency Rule C-10
C.1.11 Link C-10
C.1.12 Plural Forms C-11
C.1.13 Normalization C-11
C.1.14 Linking C-12
C.2 Code Skeleton Design C-12

Chapter 1

Introduction

A quarter of a century since the Robert W. Floyd's Turing Award Lecture on paradigms of programming [Flo79], there is no common agreement on the precise meaning of the term *paradigm* in the field of software development. In spite of that, it has been widely used to denote any distinctive enough approach to programming or software development in general. As such, it has spread to the whole software development process. However, as software has finally to be expressed in the form of a program written in one of the programming languages, it is not surprising that the term paradigm is related mostly to programming languages as such.

Programming languages are often categorized according to paradigms they support. This is being done especially according to some of the more widely accepted paradigms, namely procedural, functional, logical, and object-oriented programming.

Having several paradigms, each of which has some advantages over the other ones, has naturally lead to the idea of integrating or combining several programming languages, each of which supports some paradigm, into one, *multi-paradigm* programming language.

It is important to note that advantages of a paradigm are relative to the problem being solved. A multi-paradigm programming language itself does not help in multi-paradigm design, which involves deciding which paradigm is appropriate for the problem being solved. This issue has not been as popular research subject as creating multi-paradigm languages. The reasons may lay in already mentioned insufficient understanding of the paradigm in software development, in which, in most cases, the term paradigm is taken for given.

1.1 Thesis Objectives

This thesis is devoted to the improvement of multi-paradigm design by employing the technique of conceptual modeling known as feature modeling.

As already stated, to be able to deal with the issue of multi-paradigm design, the concept of paradigm would be analyzed as such, as well as in the context of employing multiple paradigms simultaneously in contemporary approaches to software development.

Based on this analysis, the appropriate paradigm representation would be proposed. For this, feature modeling would be adapted to the specific needs of paradigm modeling. Subsequently, the entire method of multi-paradigm design, which consists of modeling both the application domain (which is being solved) and the programming language, and the process of finding the correspondence between the application domain concepts and paradigms, would be defined in terms of feature modeling.

The newly established method of feature modeling based multi-paradigm design would be evaluated by applying it to the AspectJ programming language, resulting in a paradigm model of this language, and a subsequent application of this model in transformational analysis of the domain of feature modeling, which constitutes an application domain of a considerable size.

1.2 Thesis Structure

Besides this, introductory chapter, the thesis is structured as follows:[1]

Chapter 2 analyzes the concept of paradigm in software development.

Chapter 3 is a survey of selected post-object-oriented software development paradigms which exhibit multi-paradigm features.

Chapter 4 introduces a conceptual modeling technique of feature modeling adapted to the use in multi-paradigm design.

Chapter 5 introduces a new method of *multi-paradigm design with feature modeling*.

Chapter 6 discusses approaches related to multi-paradigm design with feature modeling.

Chapter 7 concludes the thesis and summarizes its contributions.

Also, there are three appendices to this thesis introduced in conjunction with the method evaluation:

Appendix A introduces a feature model of the domain of feature modeling.

Appendix B establishes multi-paradigm design with feature modeling for AspectJ by providing a paradigm model of this programming language.

[1] This thesis is based on the publications of the author which are available at `http://www.fiit.stuba.sk/~vranic`.

Appendix C describes an application of multi-paradigm design with feature modeling for AspectJ to the domain of feature modeling.

Chapter 2

The Concept of Paradigm in Software Development

Paradigm is a very often used—and even more often abused—word in computer science in the context of software development. Its importance arose significantly with appearance of *multi-paradigm* approaches. Before discussing these approaches in Chapter 3, the concept of paradigm in software development requires a deeper examination.[1]

First the meaning of the paradigm will be discussed—both its well-established meaning in science and the actual meaning of the word—in order to learn when its use in computer science is justified and to gain a better understanding of the concept of paradigm itself (Section 2.1). The discussion will reveal two related meanings of paradigm in software development (Sections 2.2 and 2.3).

2.1 The Meaning of Paradigm

The term paradigm in science is strongly related to Thomas Kuhn and his essay [Kuh70], where it has been used to denote a consistent collection of methods and techniques accepted by the relevant scientific community as a prevailing methodology of the specific field.

In computer science, the term paradigm denotes the essence of a software development process (often referred to as *programming*, see Section 2.2). Unfortunately, this is not the only purpose this term is used for. Probably no science has accepted this term with such an enthusiasm as computer science has; there are a lot of methods whose authors could not resist the temptation to raise them to the level of paradigm (just try a keyword "paradigm" in some citing index or digital library, e.g. [NEC]). The cornerstone of such a use of the term paradigm in computer science was probably set by the Robert

[1] This chapter is based on [Vra02b, Vra00b, Vra00a, Vra00c, Vra00d].

W. Floyd's Turing Award Lecture on paradigms of programming [Flo79] in which the meaning of paradigm ranges from methods through algorithms to programming language idioms. Although not contradictory to the original meaning of the word paradigm, such an overuse causes confusion.

The basic meaning of paradigm is *example*, especially a typical one, or *pattern*, which is in a direct connection to its etymology (Greek "to show side by side") [Mer]. The meaning and etymology pose no restriction to the extent of the example or pattern it refers to. This is reflected in the common use of the word paradigm today: on the one hand, it has the meaning of a whole philosophical and theoretical framework of a scientific school (akin to Kuhn's interpretation), while on the other hand, it is simply an example as in linguistics where it has the meaning of an example of conjugation or declension showing a word in all its inflectional forms [Mer].

Computer science, being a *science* whose great part is devoted to a special kind of *languages* intended for programming, hosts well both of these two interpretations of paradigm covered in a more detail in the following text.

2.2 Large-Scale Paradigms

The *large-scale* meaning of paradigm, as it has already been mentioned, denotes the essence of a software development process. Coplien used the term large-scale paradigm to denote programming paradigms in, as he said, a "popular" sense [Cop99a].

Besides software development paradigm and software engineering paradigm, at least two more terms are used to refer to large-scale paradigm of software development: *programming paradigm* or, simply, *programming.* Although in common use (for historical reasons), one must be careful with these terms because of possible misunderstanding: programming sometimes stands for implementation only, as other phases of a software development process can also be referred to explicitly (e.g., object-oriented *analysis*, *object-oriented* design, etc.).

The name of a paradigm reveals its most significant characteristic. Sometimes it is derived from the central abstraction the paradigm deals with, as it is a function to functional paradigm, an object to object-oriented paradigm (according to [Mey97] it is not *object* but *class* that is the central abstraction in object-oriented paradigm), etc.

Lack of a general agreement on which name denotes which paradigm is a potential source of confusion. For example, although the term *functional paradigm* is usually used to denote a kind of application paradigm, as opposed to procedural paradigm, in [Mey97] it is used to denote exactly the procedural paradigm. It is hard to blame the author for misuse of the term knowing that the procedure is often being denoted as *function.*

It must be distinguished between the software development paradigm itself and the means used to support its realization. Unfortunately, this is another source of confusion. For example, any paradigm can be visualized by means of a visual environment and thus it makes no sense to speak about the visual paradigm (as in [Bud95]). Making a complete classification and comparison of the software development paradigms is beyond the scope of this text; see [Náv96] for the comparison of selected programming paradigms regarding the concepts of abstraction and generalization.

A software development paradigm is constantly changing, improving, or better to say refining. The basic principles it lays on must be preserved; otherwise it would evolve into another paradigm. Consider, for example, the simplified view on the evolution of object-oriented paradigm. First, there were commands (imperative programming). Then named groups of commands appeared, known as procedures (procedural programming). Finally, procedures were incorporated into the data it operated on yielding objects/classes (object-oriented paradigm).

However, according to Kuhn, paradigms do not evolve, although it might seem so; it is the *scientific revolution* that ends the old and starts a new paradigm [Kuh70]. A paradigm is *dominant* by definition and thus there can be only one at a time in a given field of science unless the field is in an unstable state. According to this, simultaneous existence of several software development paradigms indicates that the field of software development is either in an unstable state, or all these paradigms are parts of the one not yet fully recognized nor explicitly named paradigm.[2] That paradigm is beyond the commonly recognized paradigms and it is about the (right) use and combination of those paradigms. Therefore it can be denoted as *metaparadigm*.

2.3 Small-Scale Paradigms

Another perception of paradigm, based on the programming language perspective, is apparent in James O. Coplien's multi-paradigm design [Cop99b] (covered in more detail in Section 3.5). According to Coplien et al. [CHW98], paradigms such as procedures, inheritance and class templates can be factored out by identifying the common and variable parts of paradigms. A paradigm is then a *configuration of commonality and variability* [Cop99b]. This is analogous to conjugation or declension in natural languages, where the common is the root of a word and variability is expressed through the suffixes or prefixes (or even infixes) added to the root in order to obtain different forms of the word.

[2] The existence of several software development paradigms has also been observed by [Flo79], but it was misinterpreted as normal state of a science according to [Kuh70].

Scope, commonality and variability (SCV) analysis [CHW98] can be used to describe these language level paradigms. Here are the definitions of the three cornerstone terms in SCV analysis (instead of *entities* the word *objects* was used in [CHW98], but this could lead to a confusion with objects in the sense of object-oriented paradigm):

Scope (S): a set of entities;

Commonality (C): an assumption held uniformly across a given set of entities S;

Variability (V): an assumption true for only some elements of S.

SCV analysis of *procedures* paradigm illustrates the definition (based on an example from [CHW98]):

S**:** a collection of similar code fragments, each to be replaced by a call to some new procedure P;

C**:** the code common to all fragments in S;

V**:** the "uncommon" code in S; variabilities are handled by parameters to P or custom code before or after each call to P.

In the context of the small-scale paradigms, it is hard to find a single-paradigm programming language. The relationship between the small- and large-scale paradigms is similar to that between the programming language features and programming languages; the large-scale paradigms consist of the small-scale ones. With respect to this, the source of the name of a large-scale paradigm can be revised here: the name of a large-scale paradigm sometimes comes from the most significant small-scale paradigm it contains. For example, object-oriented (large-scale) paradigm consists of several (small-scale) paradigms: object paradigm, procedure paradigm (methods), virtual functions, polymorphism, overloading, inheritance, etc. Lack of a common agreement what are the exact characteristics of object-oriented paradigm makes it impossible to introduce the exact list of the small-scale paradigms that object-oriented paradigm consists of.

Having an expressive programming language that supports multiple paradigms introduces another issue: a method is needed for selecting the right paradigms for the features that are to be implemented. Such a method is a *metaparadigm* with respect to the small-scale paradigms. The small-scale paradigms metaparadigm is therefore a less elusive concept than the large-scale paradigms metaparadigm. One such small-scale metaparadigm, the already mentioned multi-paradigm design, is described in Section 3.5.

One can understand small-scale paradigms as a programming language issue exclusively, while large-scale programming paradigms seem to have a broader meaning as they are affecting all the phases of software development.

Actually, small-scale paradigms have an impact to all the phases of software development as well; either with or without an explicit support in analysis and design.

Chapter 3

Towards Multi-Paradigm Software Development

This chapter is a survey of selected post-object-oriented software development paradigms in which a move towards the integration of paradigms is apparent.[1]

After a brief look at object-oriented programming in the multi-paradigm light, the chapter proceeds with the description of selected multi-paradigm approaches. Among implicitly multi-paradigm approaches, i.e. approaches based on multiple paradigms, but not explicitly concerned with paradigm selection, aspect-oriented programming and related approaches (Section 3.2) and generative programming (Section 3.3) are described. Next, explicitly multi-paradigm approaches are described: multi-paradigm programming in Leda (Section 3.4), multi-paradigm design (Section 3.5), and intentional programming (Section 3.6). The chapter is concluded by an evaluation of the presented multi-paradigm approaches (Section 3.7).

3.1 Beyond Object-Oriented Programming

Human perception of the world is to the great extent based on objects. Object-oriented programming, well-known under the acronym OOP, is based precisely on this perception of the world natural to humans. But what is OOP exactly? This question seems to be an answered one. Actually, there is a plenty of answers to this question, but the trouble is that they are all different. OOP has passed a very long way of changes to reach the form in which it is known today. Yet, there is no general agreement about what are its essential properties (to some even inheritance is not an essential property of OOP, or it is being denoted as a minor feature [Bud95]). Perhaps the Bertrand Meyer's viewpoint that "'object-oriented' is not a boolean

[1] This chapter is based on [Vra02b, Vra00b, Vra00c, Vra00d].

condition" [Mey97] is the best characterization of this issue.

OOP is not always the best choice among all the paradigms. This is recognized even in the OOP literature. Thus Booch points out that there is no single paradigm best for all kinds of applications. But, according to Booch, OOP has another important feature: it can serve as "the architectural framework in which other paradigms are employed" [Boo94]. Although this statement is probably overestimated in its applicability to all paradigms, the truth is that some multi-paradigm languages (like Leda, see Section 3.4) are designed in this fashion. This reveals that OOP is multi-paradigmatic in its very nature and leaves not much space for the object-oriented purism.

The object-oriented purism comes from the dogma that everything should be modeled by objects. But not everything is an object; neither in the real world, nor in programming. Consider synchronization as a well-known example of a non-object concept; in natural language, we would probably refer to it as *aspect*. The aspects crosscut the structure of objects (or functional components, in general) making the code tangled. The pieces of code are either repeated throughout different objects or unnatural inheritance must be involved. Among other inconveniences, this "code scattering" has a bad impact on reuse.

There are also other problems with OOP, including those it was supposed to solve, which are mainly in the areas of reuse (discussed in [SN97]), adaptability, management of complexity and performance [CE00]. In the sense of the means for solution that are at the developer's disposal—that can be denoted as a solution universe—OOP is not a universal paradigm. Actually, OOP is not a universal paradigm in C++, which is just a part of the solution universe of software development, because it is not capable of making a full use of all of its features. OOP encompasses only a few interesting kinds of commonality and variability [Cop99a]. Other kinds are needed as well, so the non-object-oriented features of programming languages are often used even though the analysis and design were object-oriented.

3.2 Aspect-Oriented Approaches

According to one of those who stood upon its birth, Gregor Kiczales, aspect-oriented programming (AOP) is a new programming paradigm that enables the modularization of crosscutting concerns [KLM+97]. The field of aspect-oriented software development is a subject of the intense research and dozens of aspect-oriented approaches exist today (see [AOS, Fil03, DVB01]). In this section, the four foundational aspect-oriented approaches will be described and compared.

3.2.1 Aspect-Oriented Programming

Most of the aspect-oriented terminology (as well as its name) later adopted by others was coined by the PARC AOP group. Their research effort is being concentrated mainly on AspectJ [Ecla] (recently, the project has been transfered to Eclipse [Eclb]), a general purpose aspect-oriented extension to Java [LK98].

AOP appeared as a reaction to the problem known from the *generalized procedure languages* [KLM+97], i.e. languages that use the concept of procedure to capture functionality (besides procedural languages, this includes functional and object-oriented languages as well). In such languages some program code fragments that implement a clearly separable *aspect* of a system (such as synchronization) are scattered and repeated throughout the program code that becomes *tangled*. AOP aims at factoring out such aspects into separate units. Aspects *crosscut* the *base* code in *join points*. These must be specified so aspects could be *woven* into the base code by a *weaver*.

A simple example written in AspectJ (version 1.1.1), similar to the example from [LK98], in Fig. 3.1 illustrates the idea. Two classes are presented there, `Point` and `Line`, whose methods are of three kinds: creating, writing and reading (the implementation of the methods is omitted). Suppose we want to be informed what kind of access to these classes has been performed. In ordinary Java we would have to modify each method of both `Point` and `Line`. Moreover, this would result in a tangled code. In AspectJ both problems can be avoided using aspects. In our example it is the aspect `ShowAccesses` that solves the problem. Note that the original code remains unchanged.

The solution with aspects is undoubtedly more elegant than the tangled one would be. However, the information where an aspect is to be woven (i.e., join points) is included in the aspect itself. This complicates the aspect reuse. AspectJ addresses this problem with abstract aspects and named sets of join points, so-called *pointcuts*.

3.2.2 Adaptive Programming

Adaptive programming (AP), proposed by Demeter group [Dem] at Northeastern University in Boston, deals mainly with the traversal strategies of class diagrams. Demeter group has used the ideas of AOP several years before the name aspect-oriented programming was coined. After the collaboration with the PARC AOP group had begun (which at that time was a part of Xerox), Demeter group redefined AP as "the special case of AOP where some of the building blocks are expressible in terms of graphs and where the other building blocks refer to the graphs using traversal strategies" (building block stands for aspect or component) [Lie]. The traversal

```
class Point {
  int x,y;
  Point(int x, int y){...}
  void set(int x, int y){...}
  void setX(int x){...}
  void setY(int y){...}
  int getX(){...}
  int getY(){...}
}

class Line {
  int x1,y1,x2,y2;
  Line(int x1, int y1, int x2, int y2){...}
  void set(int x1, int y1, int x2, int y2){...}
  int getX1(){...}
  int getY1(){...}
  int getX2(){...}
  int getY2(){...}
}

aspect ShowAccesses {
  before(): execution(* (Point || Line).set*(..)) {
    System.out.println("Write");}
  before(): execution(* Point.get*(..)) {
    System.out.println("Read");}
  before(): execution((Point || Line).new(..)) {
    System.out.println("Create");}
}
```

Figure 3.1: Tracking access in AspectJ.

strategies are partial graph specifications through mentioning a few isolated cornerstone nodes and edges, and thus they crosscut the graphs they are intended for.

An example of AP is presented in Fig. 3.2. The left part of the figure presents a UML class diagram of a system. Assume we would like to count the people waiting at the bus stations along the bus route. In ordinary OOP this would require either the implementation of small methods in all of the affected classes (depicted shaded) or rough breaking of the encapsulation by exposing some of the classes' private data.

If we use a traversal strategy, as it is proposed in AP, there is no need for a change in the existing classes. In this case, the traversal strategy:

```
from BusRoute through BusStop to Person
```

solves the problem of getting to objects of the class `Person` along the bus route, which is sufficient to count them. The right part of Fig. 3.2 demonstrates the robustness of this technique: the traversal strategy mentioned above applies in this case as well although the class diagram it was constructed for has changed.

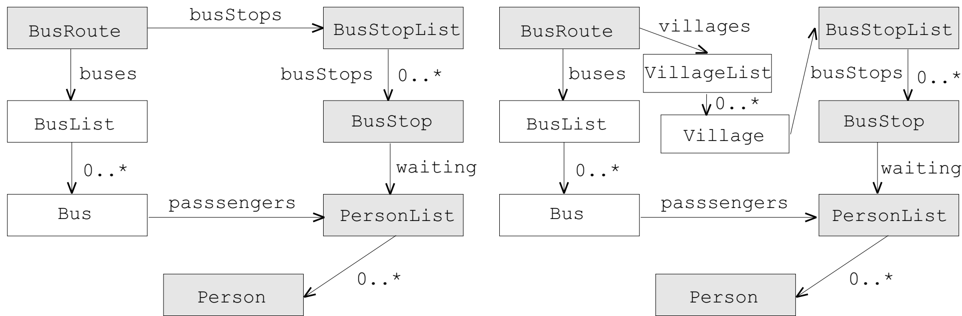


Figure 3.2: Traversal strategies (from [Lie97], ©1997 Northeastern University).

3.2.3 Composition Filters

Composition filters (CF) is an aspect-oriented programming approach in which different aspects are expressed as declarative and orthogonal message transformation specifications called *filters* [AT98].

A message sent to an object is evaluated and manipulated by the filters of that object, which are defined in an ordered set, until it is discarded or dispatched (i.e., activated or delegated to another object). A filter behavior is simple: each filter can either accept or reject the received message, but the semantics of the operations depends on the filter type. For example, if an `Error` filter accepts the received message, it is forwarded to the next filter, but if it was a `Dispatch` filter, the message would be executed. A detailed description of CF can be found in [AWB+93, Koo95].

In Fig. 3.3 two sets of filters (written in Sina language [Koo95], which directly adopt the CF model [AT98, AWB+93]) are shown. These filters are attached to the `Point` and `Line` classes from Fig. 3.1. The existence of the class `ShowAccess` is presumed. `ShowAccess` provides three methods—`WriteAccess`, `ReadAccess` and `CreateAccess`)—that simply write out the type of the access. They are called by the three corresponding `Dispatch` filters, in case the message was accepted. Afterwards, the method of the inner object, which has actually been called, is executed (`inner.*`).

From the perspective of AOP, the class `ShowAccess` implements the aspect, while the filters represent the join points. Thus, the join points in this case are separated from the aspect, which is better regarding the aspect reuse.

3.2.4 Subject-Oriented Programming

A concept can be defined by naming its properties. This is sufficient to precisely define and identify mathematical concepts, but the same does not

```
Point
  acc: ShowAccess;
  inputfilters
    WriteAccess: Dispatch = {set, acc.WriteAccess, inner.*};
    ReadAccess: Dispatch = {getX, getY, acc.ReadAccess, inner.*};
    CreateAccess: Dispatch = {Point, acc.CreateAccess, inner.*};
    Execute: Dispatch = {true => inner.*};

Line
  acc: ShowAccess;
  inputfilters
    WriteAccess: Dispatch = {set, acc.WriteAccess, inner.*};
    ReadAccess: Dispatch = {getX, getY, getX1, getY1, acc.ReadAccess, inner.*};
    CreateAccess: Dispatch = {Line, acc.CreateAccess, inner.*};
    Execute: Dispatch = {true => inner.*};
```

Figure 3.3: Tracking access example implemented using composition filters approach.

apply to natural concepts. Their definitions are *subjective* and thus never complete (more details about conceptual modeling can be found in [CE00]).

Subject-oriented programming (SOP), developed at IBM as an extension to OOP [IBM], is based on subjective views, so-called *subjects*. A subject is a collection of classes or class fragments whose hierarchy models its domain in its own, subjective way. A complete software system is then composed out of subjects by writing the *composition rules*, which specify the correspondence of the subjects (i.e., namespaces), classes and members to be composed and how to combine them.

As a result of the research effort in SOP, the Watson Subject Compiler was developed [KOHK96], which allows partial (subjective) definitions of C++ program elements and automates the composition required to produce a running program. There are also other platforms SOP support was built for, such as IBM VisualAge for C++ Version 4, HyperJ and Smalltalk.

The example from Fig. 3.1 reimplemented in Watson Subject Compiler-like syntax (the actual syntax could by slightly different) is presented in Fig. 3.4. Assume that the class `ShowAccess` is implemented in `Access` namespace and that the classes `Point` and `Line` are implemented in the `Graphics` namespace. The join-points, represented by composition rules, are separated from the aspect and represented by a separate class (as in CF approach). The composition rules for the methods `getY`, `getX1`, `getY1` and `getX2` are omitted in Fig. 3.4 (indicated by ellipsis) since they are analogous to the rules for `getX` or `getY2`.

This is not a characteristic case of the application of SOP (such can be found in [OHBS94, KOHK96, IBM]); it is presented here in order to show how a well-known AOP example can be easily transformed into its SOP version. Nevertheless, there is no general agreement whether SOP is AOP. In [CE00] SOP is viewed as a special case of AOP where the aspects accord-

```
namespace GraphicsWithAccess{
  class Point;
  class Line;}

GraphicsWithAccess.Point.Point :=
  Merge[Graphics.Point.Point, Access.ShowAccess.CreateAccess];
GraphicsWithAccess.Line.Line :=
  Merge[Graphics.Point.Line, Access.ShowAccess.CreateAccess];

GraphicsWithAccess.Point.set :=
  Merge[Graphics.Point.set, Access.ShowAccess.WriteAccess];
GraphicsWithAccess.Line.set :=
  Merge[Graphics.Line.set, Access.ShowAccess.WriteAccess];

GraphicsWithAccess.Point.getX :=
  Merge[Graphics.Point.getX, Access.ShowAccess.ReadAccess];
      . . .
GraphicsWithAccess.Line.getY2 :=
  Merge[Graphics.Line.getY2, Access.ShowAccess.ReadAccess];
```

Figure 3.4: Tracking access example implemented using subject-oriented approach.

ing to which the system is being decomposed are chosen in such a manner that they represent different, subjective views of the system. On the other hand, Kiczales et al. reject the very idea that SOP (which they call *subjective programming*) could be AOP, arguing that the methods from different subjects, which are being automatically composed in SOP, are components in the AOP sense since they can be well localized in a generalized procedure (routine) [KLM+97]. But this seem to be a more general issue, since it applies to AspectJ, too, where it has been identified as *aspectual paradox* by Liebrherr et al. [LLM99]:

> An aspect described in AspectJ, the PARC's AOP language, which has a construct for specifying aspects, is by definition no longer an aspect, as it has just been captured in a (new kind of) generalized routine.

As observed in [Cza98], SOP is close to GenVoca [BG97], a successful approach to software reuse. In GenVoca, systems are composed out of *layers* according to *design rules*: GenVoca layers can be easily simulated by subjects.

3.3 Generative Programming

Generative programming [CE00] is a software development paradigm based on modeling software system families using feature modeling, a conceptual

modeling technique which focuses on modeling common and variable features of the concepts in a domain.[2]

Generative programming aims at automations of software production by enabling a highly customized and optimized intermediate or end-product to be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.

Generative programming brings together object-oriented analysis and design methods, which provide effective system modeling techniques, with domain engineering methods, which actually enable development of the families of systems. It is also closely related to generic programming, which enables reuse through parameterization, domain-specific languages, which increase the abstraction level for a particular domain, and aspect-oriented programming, which is used to achieve the separation of concerns (see Figure 3.5).

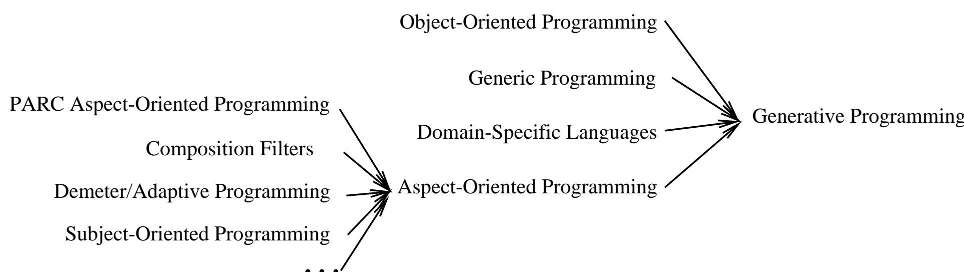


Figure 3.5: Generative programming and related approaches. The arrows represent "is incorporated into" relationship.

Generative programming first has to be tailored to a particular domain in order to be used. This process will yield a methodology for the families of systems to be developed, which can be viewed as a paradigm in its own right. This gives a certain metaparadigm flavor to generative programming.

In the solution domain, generative programming requires metaprogramming for weaving and automatic configuration. To support domain-specific notations, syntactic extensions are needed. Active libraries are proposed in [CE00], which can be viewed as knowledgeable agents interacting with each other to produce concrete components, as appropriate to cover this requirement.

3.4 Multi-Paradigm Programming in Leda

The question how to support multi-paradigm programming at the language level can be answered simply: create a multi-paradigm language.

[2] Chapter 4 describes feature modeling for multi-paradigm design. Section 6.1 describes how it differs from Czarnecki-Eisenecker feature modeling used in generative programming.

Timothy Budd took this route by creating a multi-paradigm programming language called Leda. Despite Leda is not widely used, it is worth consideration because it demonstrates the combination of paradigms. For example, the inference mechanism of logic programming can be used inside of a procedure.

The rest of this section is devoted to a brief overview of Leda programming language and a multi-paradigm design method for Leda (Sections 3.4.1 and 3.4.2). It ends with some remarks on large-scale paradigm based multi-paradigm programming languages in general (Section 3.4.3).

3.4.1 Leda Programming Language

According to Budd, Leda supports four programming paradigms [Bud95]: imperative (procedural, to be more precise), logic, functional, and object-oriented. The term paradigm, as used by Budd, denotes a large-scale paradigm (with respect to the classification of paradigms introduced in Chapter 2). This means that Leda actually supports more than four small-scale paradigms. This is clear having in mind that, for example, object-oriented paradigm breaks down into six or more small-scale paradigms, as has been shown in Section 2.3. Nevertheless, in order not to digress from the intent of this approach, just the mechanisms by which each of the four proclaimed paradigms is supported in the language will be discussed.

Leda has a Pascal-like (i.e., Algol-like) syntax and, moreover, the mechanism upon which all the four supported paradigms realization is based on in Leda are functions (procedures that return values). This makes a good background for *procedural* paradigm, denoted as *imperative* by Budd.

Logic paradigm is supported by a special type of function that returns *relation* data type and by a special assignment operator `<-`. These indicate when the inference mechanism, inherent to logic programming, is to be activated.

Functional paradigm requires no mechanism other than functions because Leda permits a function to be an argument of another function and to return a function. Thus, functional paradigm is achieved by using functions in the recursive fashion while refraining from assignments.

In addition to the basic mechanisms of *object-oriented paradigm*, such as classes, inheritance, encapsulation etc., Leda also supports *parameterized types* (considered by some authors a part of object-oriented paradigm [Mey97]).

3.4.2 A Multi-Paradigm Design Method for Leda

Leda *provides* its four paradigms; it does not force nor guid a developer to use any of them. To help with the selection among the four Leda's paradigms, an empirical design method has been proposed [KBV00]. Although proposed in connection with Leda, the method is not limited to it.

The method is based on a top-down decomposition of a system. Starting at the top level, the main paradigm for the overall system and then for each system component is being selected with respect to its advantages and disadvantages (called "pros" and "cons"), and its level of compatiblity with other paradigms, i.e. how it restricts the use of other paradigms in subcomponents. In addition, the merge points for mixing with other paradigms inside of a component are being analyzed.

3.4.3 Leda Approach Evaluation

Leda is an example of a programming language constructed to support multiple paradigms. As is usual with such languages, Leda is based on the large-scale paradigm perspective.

There are a lot of approaches that fall into this category. Yet another approach and an overview of similar approaches along with the discussion of the paradigms integration problems can be found in [VS95]. Such approaches are popular especially in the field of artificial intelligence because of the need to combine the two paradigms traditionally used in this field, logic and functional programming, both with each other and together with OOP.

Leda is a language created from scratch. Interconnection of existing languages that support different paradigms through an interface instead is an alternative to this. There is also a possibility of implementing one language on top of the other, but this leads to a certain degradation of performance. An example of interconnecting object-oriented and logic programming (Loops and Xerox Quintus Prolog) can be found in [KE88].

Different paradigms are expressed using different syntax. BETA language [Mad00] is supposed to overcome this inconvenience through a *unified syntax* achieved by introducing so-called *pattern* as an abstraction of all other programming language constructs appearing in the paradigms it supports. The approach is therefore denoted as *unified paradigm*, but it is not fundamentally different from the other large-scale paradigm based multi-paradigm programming languages.

Of course, creating a language that supports multiple paradigms and expecting it would be the best language to program in is futile as the search for the best programming paradigm. No matter how many paradigms are supported in a programming language, the number is finite and, obviously, it cannot embrace the future paradigms. One can argue that it is possible to extend the language with new programming constructs in order to support new paradigms. This is not only possible, but often practiced. Unfortunately, programming languages cannot be extended indefinitely due to limitations of parsing methods.

3.5 Multi-Paradigm Design

Multi-paradigm design (MPD), as proposed by Coplien [Cop99b, Cop00], is a metaparadigm intended for developing families of systems, therefore akin to domain engineering approaches. It deals with selcting the appropriate paradigm for a feature being designed and implemented.

MPD is based on scope, commonality, and variability (SCV) analysis (discussed in Section 2.3) or, to be more precise, SCVR analysis, where R stands for relationships between domains [Cop00], which are expressed by variability dependency graphs (explained further in this section).[3] Commonality analysis concentrates on common attributes while the aim of variability analysis is to parameterize the variation.

SCVR analysis is applied to both *application* and *solution domain.* A domain is an area of interest [Cop99b]. Two kinds of domains can be distinguished based on their role in software development: application and solution domains [Cop99b]. An application domain is a domain to which software development process is being applied. It is the body of knowledge that is of interest to users, sometimes denoted as a problem domain [Cop99b]. A solution domain is a domain in which a solution is to be expressed (usually a programming language). The major steps performed in MPD are:

- application domain SCVR analysis,
- solution domain SCVR analysis,
- transformational analysis, and
- code design.

These steps need not to be performed sequentially; they can be performed in parallel (to some extent) and revisited as needed. Before starting the actual MPD, it is recommended to evaluate the possibility to reuse an existing (similar) design. If the commonalities and variabilities of the application domain do not fit any existing solution domain structures, creation of a new application-oriented (i.e., domain-specific) language should be considered.

3.5.1 Application Domain SCVR Analysis

Commonality analysis of the application domain (usually denoted as problem domain) starts with finding commonality domains and creating domain dictionary. It then proceeds in parallel with variability analysis, whose results —the *parameters of variation* of a given commonality domain and

[3] In [Cop99b], neither SCV, nor SCVR analysis is mentioned; the term *commonality and variability analysis* is used instead.

their characteristics— are being summarized in *variability tables* (one per each commonality domain), as depicted in the upper part of Fig. 3.6.

As already has been mentioned, *variability dependency graphs* (denoted also as *domain dependency* graphs or diagrams) are used to capture the relationship between domains and their parameters of variation, which are also domains. Variability dependency graphs are directed graphs whose nodes represent domains and edges represent "depends on" relationship (in the direction indicated by an arrow) between domains and their parameters of variation. Despite the simple notation, variability dependency graphs are quite useful in identifying *overlapping* domains (such domains can be merged) and *codependent* domains, i.e. the domains with circular dependencies (which must be resolved).

3.5.2 Solution Domain SCVR Analysis

The same commonality and variability analysis as applied to the application domain is applied to the solution domain, i.e. the programming language. First, a description of the identified small-scale paradigms, manifested through the language features, structured according to commonality, variability, binding, and example is provided. The analysis proceeds with exploring the *negative variability*, a variability that violates the rule of variation by attacking the underlying commonality. A *positive variability*, as opposed to the negative one, can be parameterized. The negative variability has to be kept small. If it becomes larger than the commonality, the design should be refactored to reverse the commonality and variability.

The results of the solution domain commonality and variability analysis are summarized in the *family table*, as shown in the lower part of Fig. 3.6, and in the table used to express *features for negative variability*, where for each combination of the kind of commonality and kind of variability the language feature for positive variability and the one for the corresponding negative variability is introduced. In [Cop99b, Cop00] the C++ programming language has been analyzed as a solution domain.

3.5.3 Transformational Analysis and Code Design

The tables obtained in the preceding analyses are used in *transformational analysis*, which is, roughly speaking, a matching of application domain structures, described in variability tables, with solution domain structures, i.e. paradigms, described in family tables. Figure 3.6 shows how is this matching performed. Prior to the matching, the commonality domain has to be generalized (e.g., TEXT EDITING BUFFERS: *behavior, structure*), and the parameters of variation also (e.g., output medium: *structure, algorithm*).

Finally, the code corresponding to the transformational analysis results should be written. It is obvious that transformational analysis determines

Variability tables (from application domain SCVR analysis)

Text Editor Variability Analysis for Commonality domain:
TEXT EDITING BUFFERS (*Commonality: Behavior and Structure*)

Parameters of variation	Meaning	Domain	Binding	Default
Output medium *Structure, Algorithm*	...	Database, RCS file, TTY, UNIX file	Run time	UNIX file

Family table (from solution domain SCVR analysis)

Commonality	Variability	Binding	Instantiation	Language Mechanism
. . .				
Related operations and some structure (positive variability)	. . .			
	Algorithm (especially multiple), as well as (optional) data structure and state	Compile time	Optional	Inheritance
	Algorithm, as well as (optional) data structure and state	Run time	Optional	Virtual functions

Figure 3.6: Transformational analysis in MPD (according to an example from [Cop99b]).

only the code skeleton.

3.5.4 Multi-Paradigm Design Evaluation

MPD emphasizes solution domain analysis, underestimated in contemporary software development methodologies resulting into a gap between design and implementation.

To a certain extent, MPD enforces the *reuse of design*: both application and solution domain analysis can be reused independently; however, transformational analysis is not reusable. This brings MPD close to *design patterns*, as discussed in [Cop99b]. On the very cover of the design patterns cornerstone book [GHJV95] Steve Vinoski points out that a reusable design is “the real key to software reuse”. This claim is being justified in the ongoing research on reuse with design patterns [SN00].

Indeed, MPD and design patterns seem to be complementary; design patterns capture designers’ experience by documenting the recommended solutions for the common problems in software development, while MPD relies on this experience. However, to make a full use of design patterns in MPD, and in software development in general, a better way of their

representation is needed [SNB98].

Although the design patterns from [GHJV95] are inspired by Alexandrian patterns [Ale79], not all of them are the patterns in Alexandrian sense: some of them can be formalized as configurations of commonality and variability (unlike Alexandrian patterns). As such they can be incorporated directly into MPD (by adding them to the family table), as anything else that can be formalized as a configuration of commonality and variability (i.e., other paradigms and solution domain tools that are not supported by the main programming language, like databases or parser generators) [Cop99b].

One of the problems with MPD is the unsuitability of the notation used: only a few types of tables and variability diagrams with a lot of the relevant details expressed as informal text do not support transformational analysis sufficiently. A better notation could also ease the transition to the actual program code (the program skeleton).

3.6 Intentional Programming

Programming languages with fixed syntax are limiting otherwise unlimited number of programming abstractions. Intentional programming group at Microsoft Research offered a solution to this problem as a new software development paradigm called *intentional programming* (IP) [Sim99, Sim96, R□d99] (the project is on hold from Spring 2001 [Roe]). The idea behind IP is that programming abstractions, which are in IP denoted as *intentions*, could live better without their hosts, (fixed-syntax) programming languages, because of their limits in the accepted notations (due to underlying grammars).

A program in IP is represented by a so-called *intentional tree*, whose nodes represent intention instances. Each intention instance points to the corresponding intention declaration node that provides a method which specifies the process of transforming the subtree headed by the intention instance. The executable program is obtained in a process called *reduction* in which the intentional tree is traversed and transformed according to the rules indicated by intention declarations until it consists only of executable nodes. Such a *reduced* tree is represented in an intermediate language. The executable code is generated from this representation.

It would be inconvenient for a human to directly maintain the intentional tree. This is being performed in a programming environment with a special graphic editor instead of the usual text editor. This enables each intention to have its own graphic representation. Of course, entering a program in such an environment is quite different from entering it in a classic text editor. A program text, as we are used to it, is a complete and an unambiguous representation of the program. In IP environment this is not so. What is presented in IP editor is only a view of the actual program. To illustrate

this, consider one peculiarity: two distinct variables can have the same name (even if they reside the same scope). This is possible because the intentional tree does not rely on the names to identify intentions; the names are provided only for developers' convenience.

Although it can seem so, IP is not intended to push out the existing programming languages from the scene. It can import any program in any programming language if a parser for that language—in the form of a library—is added to IP environment.

3.7 Summary

There is a growing tendency towards multi-paradigm software development notable not only in the explicitly multi-paradigm approaches, but also in the implicit ones, such as aspect-oriented programming and generative programming. The three explicitly multi-paradigm approaches presented in this chapter are compared in Table 3.1 according to the selected criteria: the concept of *paradigm* the approach enforces, a programming *language* the approach is bound to, and whether the approach supports the *language extension*.

Approach	Paradigm	Language	Language extension
Multi-Paradigm Programming in Leda	large-scale	Leda	no
Multi-Paradigm Design	small-scale	any	not applicable
Intentional Programming	small-scale	none	yes

Table 3.1: The three multi-paradigm approaches compared.

It is important to note that these three approaches are not antagonistic; they are complementary. Multi-paradigm design arms us with techniques for dealing with multiple paradigms when a multi-paradigm environment is available. Intentional programming enables such an environment to be created and maintained easier than it is the case with the classical programming languages. Finally, multi-paradigm programming in Leda demonstrates how four specific programming paradigms can be combined.

As has been shown on the example of object-oriented programming in Section 2.3, there is no a common agreement what are the exact characteristics of the large-scale paradigms. This is even more notable in aspect-oriented programming, as has been shown by the comparison of the four foundational aspect-oriented approaches: despite they are all analogous and can be characterized as aspect-oriented, each one supports aspect-oriented programming by different programming language constructs. These programming language constructs can be modeled as small-scale paradigms, and this approach has been taken in multi-paradigm design by employing

SCVR analysis and applied to C++ [Cop99b].

However, as has been pointed out in Section 3.5.4, the table notation used in SCVR analysis is not suitable to represent paradigms, nor application domain structures. Moreover, the matching of the application domain structures to paradigms is hard to perform in this notation. Feature modeling, a modeling technique used in domain engineering to express commonality and variability in a domain, appears to be more appropriate for this task. The next chapter introduces feature modeling adapted to multi-paradigm design. Although there are many analogies between feature modeling and SCVR analysis (see Section 6.2.1), the change of the underlying modeling technique affects the whole approach significantly; the new, feature modeling based multi-paradigm design is proposed in Chapter 5.

Chapter 4

Feature Modeling for Multi-Paradigm Design

Feature modeling is a conceptual domain modeling technique in which concepts are expressed by their features taking into account feature interdependencies and variability in order to capture the concept configurability (a definition adapted from [CE00]).[1]

A *domain* is understood here as an area of interest [Cop99b].[2] Two kinds of domains can be distinguished based on their role in software development: application and solution domains [Cop99b]. An *application domain*, sometimes denoted as a problem domain [Cop99b], is a domain to which software development process is being applied. A *solution domain* is a domain in which a solution is to be expressed (usually a programming language).

Feature modeling for multi-paradigm design, described in this chapter, is based on Czarnecki-Eisenecker feature modeling [CE00, Cza98]. The origins of feature modeling are in FODA method [KCH+90]. Feature modeling has been used to represent models of application domains in many domain engineering approaches to software development beside FODA such as FORM [KKL+98], ODM [Sim95], or generative programming [CE00, Cza98]. Each such a method usually employs somewhat different feature modeling notation, adapted to its needs (compared in Section 6.1).

In *multi-paradigm design with feature modeling*, the method proposed in this thesis (Chapter 5), feature modeling is going to be used to express both application and solution domain concepts in order to simplify finding a correspondence and establishing the mapping between the application and solution domain concepts in transformational analysis. Therefore, Czarnecki-Eisenecker feature modeling has to be adapted, and this chapter will focus

[1] This chapter is based on [Vra].

[2] The notions domain, application domain, and solution domain have already ben defined in Section 3.5. Their definitions are repeated here for convenience.

on the adaptation.[3]

After introducing the basic feature modeling notions (Section 4.1) and the notation of feature diagrams (Section 4.2), including concept references as the means of structuring feature diagrams (Section 4.3), the information associated with concepts and features will be explained (Section 4.4). Subsequently, parameterization in feature models (Section 4.6), concept instantiation (Section 4.7), and equivalent and normalized feature diagrams (Section 4.8) will be discussed. Finally, some guidelines regarding the use of feature modeling (Section 4.9) and a brief overview of the available feature modeling tools will be given (Section 4.10).

To illustrate the technique of feature modeling, several examples will be introduced. To improve readability, the examples will be numbered and an end of each example marked by a filled square (■). Appendix A presents a detailed example of a feature model of the domain of feature modeling itself.

4.1 Basic Notions

A *concept* is an understanding of a class or category of elements in a domain. Individual elements that correspond to this understanding are *concept instances.*

A *feature* is an important property of a concept [CE00]. A feature may be *common*, in which case it is present in all concept instances, or *variable*, in which case it is present only in some concept instances. Any feature may be isolated and modeled further as a concept, therefore being a feature is actually a relationship between two concepts. However, the concepts identified only in the context of other concepts, i.e. as their features, will be referred to as features exclusively. This helps emphasize the main concepts in a domain.

The output of feature modeling is a feature model of a domain. A *feature model* consists of a set of feature diagrams, information associated with concepts and features, as well as constraints and default dependency rules associated with feature diagrams. A *feature diagram* is a directed tree whose root represents a concept and the rest of the nodes represent its features.

Each concept has a name unique in its domain. A concept is described by its features organized in a feature diagram. Each feature has a name unique among the first-level features of its parent. A feature may have its own features, which are sometimes denoted as *subfeatures* with respect to their parent [CE00].

The features connected directly to a concept or feature are being denoted as its *direct features*; all other features are its *indirect features* [CE00]. A feature or concept that has at least one direct variable feature is called a *variation point* [CE00].

[3]The differences are discussed in Section 6.1.

Concepts and features are referred to by their names. To avoid name clashes when combining features from several diagrams in one expression, their names should be qualified according to this convention:

Concept.Feature 1.Feature 2...Feature n

In case of combining concepts and features from several domains, the domain name followed by a colon should precede the expression:

Domain:Concept.Feature 1.Feature 2...Feature n

4.2 Feature Diagrams

Each concept is presented in a separate feature diagram. A feature diagram is drawn as a directed tree with edge decorations. The root represents a concept, and the rest of the nodes represent features. Edges connect a concept with its features, and a feature with its subfeatures.

Concept instances are represented by configurations of concept features achieved by a selection of these features according to their variability. A feature can be included in a concept instance only if its parent has been included. A concept instance *must* have all the mandatory features and *can* have the optional features.

There are two types of edges used to distinguish between *mandatory* features, ended by a filled circle, and *optional* features, ended by an empty circle. A concept instance *must* have all the mandatory features and *can* have the optional features.

The edge decorations are drawn as arcs connecting disjunct subsets of the edges originating in the same node. They are used to define a partitioning of the subnodes of the node the edges originate in into *alternative*, drawn as an empty arc, and *or-features*, drawn as a filled arc. The alternative features (whose origin is in FODA) have exclusive-or semantics. Or-features have inclusive-or semantics (added in Czarnecki-Eisenecker feature modeling).[4]

Exactly one feature can be selected from the set of alternative features. If the selected feature is optional, its inclusion in a concept instance is further decided based on that optionality.

Any subset or all of the features can be selected from the set of or-features. Among the selected features, each optional feature's inclusion in a concept instance is further decided based on that optionality.

Alternative and or-feature sets containing optional features are further discussed in Section 4.8.

A concept or feature is *open* if it is expected to have new direct variable subfeatures. This is indicated directly in feature diagrams by introducing the

[4] These two types of features are sometimes referred to as exclusive-or and inclusive-or [CEH03].

open concept or feature name in square brackets and optionally by ellipsis at its subfeatures. The openness may be explained further in the concept or feature description, which is a part of the information associated with concepts and features (explained in Section 4.4).

A feature diagram in Fig. 4.1 demonstrates the notation. Features f_1, f_2, f_3, and f_4 are direct features of the concept C_1, while other features are its indirect features. Features f_1 and f_2 are mandatory alternative features. Feature f_3 is an optional feature. Features f_5, f_6, and f_7 are mandatory or-features; they are also subfeatures of f_3. Feature f_3 is open. In addition, ellipsis expresses more precisely that new mandatory or-features are expected in the group of f_5, f_6, and f_7.

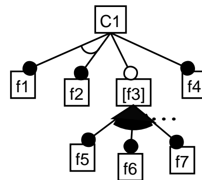


Figure 4.1: A feature diagram.

Example 4.1 Consider the text editing domain and its concept of a text editing buffer[5] whose feature diagram is presented in Fig. 4.2. Ignore the (R) marks that appear at some of the features for the moment; these will be explained in the next section. A text editing buffer represents the state of a file being edited in a text editor which is modeled as a mandatory feature *File*. Each text editing buffer employs some memory management scheme to be able to deal with big files that cannot be loaded in the working memory at once. This is modeled by a mandatory feature *Memory Management*.

A text editing buffer uses one of the available character sets. This is modeled by the mandatory feature *Character Set* and its mandatory alternative subfeatures. Feature *Character Set* is open; ellipsis indicates that new features (i.e., character sets) are expected in the existing group of alternative features. Debugging code might be useful during the development, but should not be present in the final product. This is modeled by the optional feature *Debugging Code*. All text editing buffers load and save their contents into a file, maintain a record of the number of lines and characters, the cursor position, etc., modeled by further mandatory features (bearing the corresponding names). ■

[5]This example is adapted from [Cop99b, Vra01a])

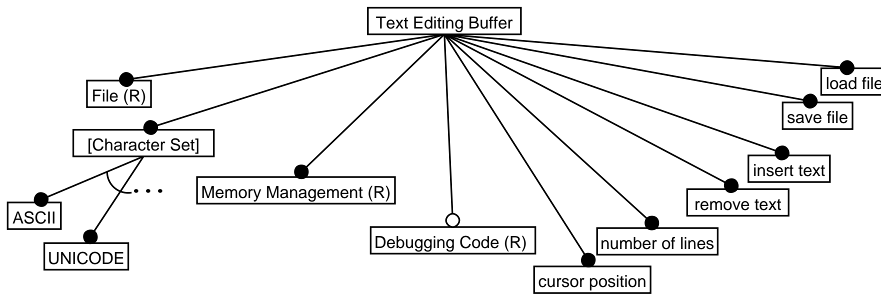


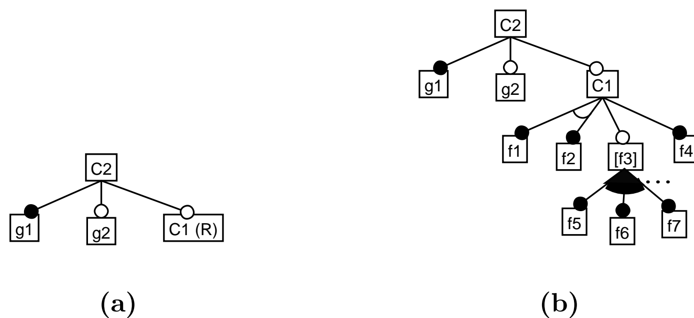
Figure 4.2: The feature diagram of the *Text Editing Buffer* concept.

4.3 Concept References

A concept can be referenced as a feature in another or even in its own feature diagram. A *concept reference* is a feature that stands for an already defined concept. The Ⓡ mark[6] follows the names of concept references in order to distinguish them from the rest of the features. The features *File*Ⓡ, *Memory Management*Ⓡ, and *Debugging Code*Ⓡ in Fig. 4.2 represent concept references.

The introduction of a concept reference is equivalent to the repetition of the whole feature diagram of the concept in the place of the reference. The referencing is used to avoid repeating in order to improve the readability of feature diagrams. The repeating can be used instead of referencing (although the referencing is more elegant) except in the case of a self-reference (i.e., when the concept is referenced in its own feature diagram) because that would lead to an infinite repetition of the referenced concept feature diagram.

Figure 4.3a presents the feature diagram of concept C_2 that refers to the concept C_1 from Fig. 4.1. Figure 4.3b presents the same feature diagram, but with the feature diagram of C_1 repeated instead of being referenced.



(a) **(b)**

Figure 4.3: Referring to concepts.

[6]For technical reasons, presented as (R) in diagrams.

If the concept being referenced belongs to another domain, and if the domain in which the concept is being referenced already contains a concept with the same name, then the reference name qualification should include the domain name (see Section 4.1). Otherwise, the concept name is sufficient.

It is not possible for a concept reference to have subfeatures. In case this appears to be needed, the referenced concept should be extended with the additional subfeatures. The applicability of the additional feature may be controlled by constraints in the information associated with the concept.

The subfeatures attached to the concept should be optional even if they ought to be mandatory for the concept reference. This may always be regulated by appropriate additional constraints on feature selection (explained in Section 4.5).

4.4 Information Associated with Concepts and Features

The following information (needed in multi-paradigm design) should be associated with concepts and features:

Description (D) Each feature and concept is provided with the description, the text explaining the meaning of the concept or feature. If the concept or feature is open, i.e. if new direct subfeatures are expected, this should be explained in its description also.

Presence rationale (Rp) Each feature may be provided with presence rationale, the text that specifies the reasons (beyond the obvious ones) for having it in the feature model of a concept.

Inclusion rationale (Ri) Each variable feature may be provided with inclusion rationale, the text that specifies the special reasons for its inclusion or non-inclusion in a concept instance, if any.

Binding time/mode (B) Each variable feature is provided with the information on binding time or mode.

Note (N) Any other information about a feature may be introduced as a note.

The above list is also the form the information associated with concepts and features should be introduced in. The abbreviation introduced in parentheses following the name of each part will be used in referring to it.

4.4.1 Binding Time/Mode

The information on binding time/mode requires further explanation. The binding time describes *when* a variable feature is to be bound, i.e. selected.

The binding time of a variable feature is determined in terms of the binding times available in the solution domain. The following list presents the usual binding times [Cop99b]:

Source time The time of program source code writing, when a programmer explicitly decides what is performed (e.g., that a class will provide some method).

Compile time The time of program source code compiling, when decisions are made by a compiler (e.g., which method to select among the overloaded ones).

Link time The time of program linking, when decisions are made by a linker.

Load time The time of program loading, when decisions are made by a loader.

Run time The time of program running, when decisions are made by the running program.

The binding mode describes *how* a variable feature is bound from the perspective of a running program. A variable feature may be bound *statically*, in which case it cannot be rebound at run time, or *dynamically*, in which case it can be rebound at run time [CE00]. Other, more specific binding modes may be defined as well, e.g. changeable binding as an optimized dynamic binding [CE00] (see also Section 6.1.8).

At the time of modeling application domain, the solution domain may be unknown or we may want to keep the application domain feature model free of the solution domain details. In that case, using the binding mode instead of the binding time is more appropriate.

Both binding time and binding mode further in this thesis will be denoted as binding whenever it is clear from the context whether the binding time, binding mode, or both are considered.

The binding may be indicated directly in feature diagrams. If at one variation point all its direct variable subfeatures have the same binding, it is sufficient to indicate the binding at one of them.

4.4.2 Associated Information Applicability

Table 4.1 summarizes the applicability of the individual parts of the information associated with concepts and different types of features.

A concept reference may be associated with its own information as any other feature, but the information associated with the concept it references applies to it as well. If no further information is to be provided with the concept reference, the information associated with it may be left out.

Table 4.1: The associated information applicability.

Information:	**D**	**Rp**	**Ri**	**B**	**N**
Concept	X				X
Common feature	X	X			X
Variable feature	X	X	X	X	X

4.5 Constraints and Default Dependency Rules

Feature diagrams define the main constraints on legal feature combinations in concept instances. Since feature diagrams are represented as trees, in all but simplest cases it is impossible to express all the constraints solely by a feature diagram. Additional constraints are expressed in a list of constraints associated with the feature diagram. Also, a list of default dependency rules is associated with each feature diagram in order to specify which features should or should not appear together by default.

Constraints and default dependency rules are predicate logic expressions formed out of specific and parameterized names of concepts and features (explained in Section 4.6.1), logical connectives, quantifiers, and parentheses. The intention of using predicate logic to express constraints and default dependency rules is to avoid ambiguities natural language is prone to. At this stage, the automated evaluation of the constraints and default dependency rules has not been considered, although that would certainly be useful.[7]

Commonly used quantifiers—$\forall$ (universal quantifier) and $\exists$ (existential quantifier)—and connectives— $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\underline{\vee}$ (xor), $\Rightarrow$ (implication), $\Leftrightarrow$ (equivalence)—should suffice to express constraints and default dependency rules, but other connectives and quantifiers may be used as well if they would improve the clarity of expressions. The precedence of connectives in descending order (from left to right) with the connectives with equal precedence placed together in parentheses is as follows:

$$\neg, \wedge, (\vee, \underline{\vee}), (\Rightarrow, \Leftrightarrow)$$

A feature name *f* in constraint or default dependency rule expressions stands for *is_in_instance(f)*, where *is_in_instance* is a predicate which is true if *f* is embraced in the concept instance, and false otherwise. Obviously, for each mandatory feature *f* *is_in_instance(f)* is true. Therefore, it makes no sense for a mandatory feature to appear in a constraint or default dependency rule. The same applies to concepts.

[7] Constraints as predicate logic expressions enable automated feature model consistency or concept instance validity checking. This issue is out of the scope of this thesis; see Section 6.1 for a brief information on approaches concerned with it.

Parameters in parameterized names of concepts and features (explained in Section 4.6.1) appearing in expressions must be quantified; otherwise, the expression containing parameterized names cannot be evaluated. However, if the name of a feature in a feature diagram of a parameterized concept is parameterized, no quantification is needed because the expression is then related to it as such.

Feature names in expressions should be fully or partially qualified to avoid name clashes (see Section 4.1). Note that since each expression is associated with exactly one feature diagram, the domain and the concept name are unnecessary. To avoid repeating long qualifications, as in:

$$A.B.C.x \lor A.B.C.y$$

the expression which a qualification is related to may be introduced in parentheses and preceded by the qualification as if it was a feature, transforming the above example into:

$$A.B.C.(x \lor y)$$

The feature that belongs to a concept referenced in another feature diagram (and which actually does not appear in the diagram directly) may also appear in the expressions. Such a feature should be qualified as if the referenced concept has been expanded. For example, in expressions regarding the feature diagram in Fig. 4.3a, the feature *C2.C1.f3.f7* may be referred to.

4.5.1 Constraints

A list of constraints associated with a feature diagram is a conjunction of the expressions it consists of. Thus, for a concept instance to be valid, all the constraints associated with the feature diagram must be fulfilled, i.e. they must evaluate to true.

Constraints express mutual exclusions and requirements among features, i.e. they determine which features cannot appear together and which must appear together, respectively [CE00]. A single constraint expression may express several mutual exclusions and requirements at once.

Constraints may have numerous equivalent forms (that may be derived one from another following the rules of predicate logic). However, constraints are intended to be read by humans; therefore they should be kept in the form which is as comprehensible as possible and accompanied by an explanatory note, if needed. Bearing this in mind, mutual exclusions may be expressed by connecting features with exclusive or, while requirements may be expressed as implications or equivalences, depending on whether the requirement is bidirectional or not.

Example 4.2 Consider again the text editing buffer feature diagram from Example 4.1 (page 30). If, for some reason, we would want to say that no text editing buffer employing ASCII character set may be provided with the debugging code, we would add the following constraint to it:

$$\mathit{Character\ Set.ASCII} \veebar \mathit{Debugging\ Code}$$

■

As has been said, the main constraints are expressed directly in feature diagrams and thus need not be repeated in the information associated with them. However, sometimes it may be needed to change a feature diagram constraint to associated one, or vice versa. In a feature diagram, mutual exclusion is expressed by alternative features. A requirement is expressed by a variable subfeature whose parent is also a variable feature: the subfeature *requires* its parent to be included. Also, a requirement may be expressed by or-features: at least one feature is *required* from the set of or-features.

In case of any contradiction among the constraints, it is impossible to instantiate the concept. While the notation of feature diagrams prevents the creation of contradictory feature diagrams, the constraint expressions contradictory either among themselves or to the feature diagram they are associated with are easily built.

4.5.2 Default Dependency Rules

A list of default dependency rules associated with a feature diagram is a disjunction of an implicit (and not displayed in the list) *true* and the expressions it consists of. The implicit *true* disjunct in a list of default dependency rules assures that it always evaluates to *true*.

Default dependency rules determine which features should appear together by default. They are expressed by the same means as constraints. The difference between default dependency rules and requirements is that requirements must hold for any concept instance, while default dependency rules are applied at the end of the process of concept instantiation if there are variable features left such that no explicit selection has been made among them. Which of these features will be included in the concept instance is decided according to the default dependency rules.

4.6 Parameterization in Feature Models

Two kinds of parameterization may appear in feature models: parameterized feature and concept names, explained in Section 4.6.1, and parameterized concepts, explained in Section 4.6.2. Section 4.6.3 explains how to represent cardinality in feature models using parameterized concepts.

4.6.1 Parameterized Feature and Concept Names

In case we need to reason about several concepts or features in constraints and default dependency rules (see Section 4.5), parameterized names of concepts and features may be used to skip the repetition of the expression with each concept or feature name.

A *parameterized name* of a concept or feature has the form:

$p_1 p_2 \dots p_n$

where for each $i \in [1, n]$ p_i is either a parameter or specific string and where exists $j \in [1, n]$ such that p_j is a parameter. For each parameter, a set of possible strings that may be substituted for it has to be defined in its description. Parameters are introduced in <> brackets to distinguish them from specific strings.

An expression in which one or more parameterized names appear is denoted as a parameterized expression. A parameterized expression is equivalent to the conjunction of the expressions created by substituting all the combinations of the appropriate specific strings for the parameters.

Parameterized names are the only way to express constraints and default dependency rules about subfeatures of an open feature because their number is unknown. Consider the feature diagram in Fig. 4.4. The feature *f* is open; further direct variable subfeatures of the form *f<i>*, where *<i>* is a natural number, are expected at it. All these features may be referred to by the parameterized feature name *f<i>*.

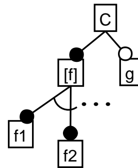


Figure 4.4: A feature diagram with an open feature.

4.6.2 Parameterized Concepts

The concept of a parameterized name allows for concepts to be parameterized, too. This is useful when there is a number of concepts with structurally equal feature diagrams.

A *parameterized concept* or *feature* is a concept or feature whose name is parameterized. Parameterized features may appear only in feature diagrams of parameterized concepts. A feature model containing a non-parameterized concept with parameterized features in its feature diagram would be inconsistent since it would define a set of different feature diagrams for a single concept. For the same reason, parameterized concepts may not be referenced in feature diagrams of specific (i.e., non-parameterized) concepts.

Example 4.3 Figure 4.5 shows an example of a parameterized concept. The name <*Plural Form*> is the plural form of <*Singular Form*>. ■

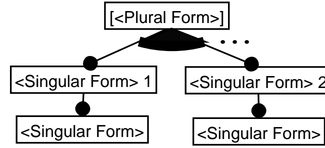


Figure 4.5: Dealing with plural forms using a parameterized concept.

4.6.3 Representing Cardinality in Feature Models

Parameterized concepts are capable of representing UML style cardinalities represented by a comma separated list of the *minimum..maximum* cardinality pairs [Obj03]. The exact cardinality is achieved by introducing it in both *minimum* and *maximum*. This may be achieved by a feature diagram in Fig. 4.6a with the following constraint which will assure the appropriate number of features according to the specified cardinality:

$$\bigvee_{<i>=1}^{<n>} \Big((\max<i> \neq * \Rightarrow \bigvee_{<j>=<min<i>>}^{<max<i>>-<min<i>>+1} \; \bigwedge_{k=1}^{i} <C><k>) \wedge \\ \wedge (\max<i> = * \Rightarrow \bigwedge_{k=1}^{<min<i>>} <C><k>) \Big)$$

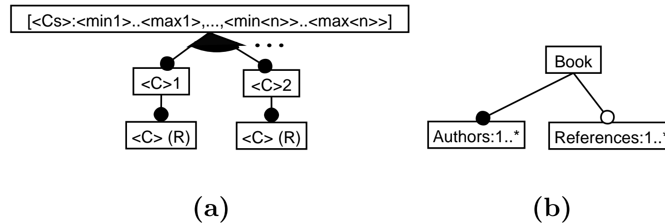


Figure 4.6: Parameterized concept for representing cardinality (a) and an example of its application (b).

The parameter <*Cs*> is the plural form of the parameter <*C*>. Note that parameters <*min*<*i*>> and <*max*<*i*>> are in fact doubly parameterized. This is to enable the parameterization of the number of *minimum..maximum* cardinality pairs.

The values allowed for both minimum and maximum cardinalities are natural numbers. Also, an additional value denoted by an asterisk is allowed for the maximum cardinality value meaning "many," as in [Obj03]. Zero cardinality is achieved by referencing the concept <*Cs*>:<*min1*>..<*max1*>,..., <*min*<*n*>>..<*max*<*n*>> as an optional feature.

This parameterized concept may be applied to any domain by including it in the feature model of the domain. Next, the set of the singular and plural forms of concept names corresponding to each other (representing possible values for <*C*> and <*Cs*>, respectively) has to be defined. Obviously, a feature model must include the concepts singular form concept names refer to. Finally, specific concept name and a set of *minimum..maximum* cardinality pairs should be substituted. An example is shown in Fig. 4.6b; a book has at least one author, and it may have zero (modeled by the optionality of *References:1..**) or more references.

4.7 Concept Instantiation

Concept instantiation mentioned in Section 4.1 abstracts from instantiation time. A full definition of a concept instance is given here.

An instance I of the concept C at time t is a configuration of C's features which includes the C's concept node and in which each feature whose parent is included in I obeys the following conditions:

1. All the mandatory features are included in I.
2. Each variable feature whose binding time is earlier than or equal to t is included or excluded in I according to the constraints of the feature diagram and those associated with it. If included, it becomes mandatory for I.
3. The rest of the features, i.e. the variable features whose binding time is later than t, may be included in I as variable features or excluded according to the constraints of the feature diagram and those associated with it. The constraints (both feature diagram and associated ones) on the included features may be changed as long as the set of concept instances available at later instantiation times is preserved or reduced.
4. The constraints associated with C's feature diagram become associated with the I's feature diagram.

If binding mode is used instead of binding time (see Section 4.4), and the concept instantiation is needed for some reason, the features bound statically are considered to be bound earlier than those bound dynamically.[8]

A concept may be instantiated in a top-down or a bottom-up fashion. The top-down instantiation starts by the inclusion of the concept node; then inclusion of each feature whose parent has been included is considered.

[8]In multi-paradigm design with feature modeling, only solution domain concepts are being instantiated, and the variable features of solution domain concepts have precise binding times (see Section 5.2).

The bottom-up instantiation starts at leaves and proceeds towards the root. Inclusion of the concept and each feature is stipulated by the inclusion of its children. This means that a feature may be considered for inclusion only if all of its mandatory subfeatures have been included and if each its variable subfeature has been included or left out respecting its variability constraints (including the associated ones) and binding time. This holds for the concept node itself; in case it cannot be included in the concept instance, the instantiation has been unsuccessful.

A concept instance is represented by a feature diagram derived from the feature diagram of the concept by showing only the features included in the concept instance.

A concept instance is regarded further as a concept and as such may be considered for further instantiation at later instantiation times. This is demonstrated in Fig. 4.7 on the concept repeated from Fig. 4.1 augmented with the binding time information (following the notation proposed in Section 4.4); it is assumed that there are no other constraints associated with the feature diagram.

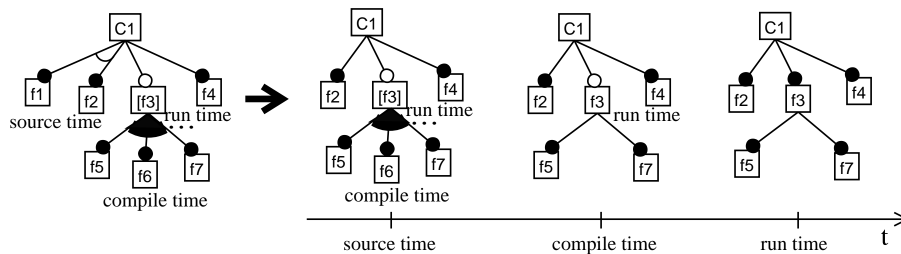


Figure 4.7: Concept instantiation with respect to instantiation time.

During instantiation, a concept reference may appear in a concept instance as any other feature if it is not replaced by the diagram of the concept it references prior to instantiation.

4.8 Equivalent and Normalized Feature Diagrams

Different feature diagrams may define the same set of concept instances; such feature diagrams are denoted as equivalent [CE00].

This is caused by sets of alternative and or-features that involve optional features. For each such a set of features, there are several different sets of features that define the same constraints on feature selectability; such sets will be denoted as equivalent.

Given a set of alternative features of which one or more features are optional, any feature optionality modification that preserves at least one of the features optional, will result in an equivalent set of features. The set

in which all alternative features are optional is considered to be the normal form in this case. An example is presented in the bottom of Fig. 4.8a.

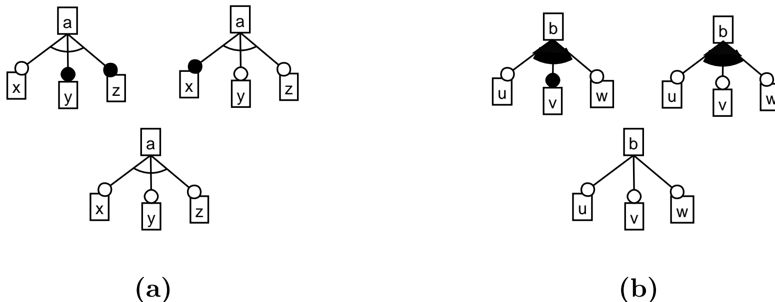


Figure 4.8: Equivalent feature diagrams and normalization: (a) optional alternative features, (b) optional or-features.

Similarly, given a set of or-features of which one or more features are optional, any feature optionality modification preserving at least one of the features optional will result in an equivalent set of features. The additional equivalent set is the set with all the features being optional; this set is considered to be the normal form in this case. An example is presented in the bottom of Fig. 4.8b.

A feature diagram whose each set of alternative or or-features with optional features is replaced by the equivalent normal form is said to be *normalized*. Examples of such transformations are shown in Fig. 4.8. The diagrams in the lower part of the figure are normalized.

Note that since the transformation to an equivalent feature diagram (including the normalization) does not affect the structure of a feature diagram, nor it changes the set of concept instances defined by it, all the constraints associated with it apply to the resulting feature diagram without change.

4.9 Applying Feature Modeling

This section gives some guidelines regarding the use of feature modeling as has been proposed in this chapter. These guidelines do not go into the details of extracting concepts and features from the domain related information. Please refer to [CE00] for more on strategies for identifying features.

Prior to modeling, the main concepts in a domain have to be identified. A key to identifying main concepts is the terminology found in the domain related information. The main concepts found in the domain related information should be listed. Further concepts may be discovered during the modeling of the main concepts.

Example 4.4 Consider the text editing domain. This domain has already been mentioned in Example 4.1 (page 30) in the context of the feature

diagram of the text editing buffer concept. This is an important concept in the text editing domain, which represents the state of a file being edited in a text editor. From this, it implies that a file is also a concept that should be considered. To deal with big files that cannot be loaded in the working memory at once a memory management scheme may be needed. Also, for development purposes, a debugging code may be useful. To summarize, four main concepts have been identified in the text editing domain: text editing buffer, file, memory management scheme, and debugging code. ■

After identifying the main concepts, we may proceed with their modeling. Features of concepts may be identified in the domain related information. Further features and concepts may emerge from the interaction of already identified features.

In modeling first-level features (including concept references), proceed as follows for each concept C from the set of identified concepts:

1. Describe C.
2. Identify C's direct common features and arrange them in the feature diagram with C as their parent.
3. Identify C's variable features and arrange them in the feature diagram with C as their parent.
4. Analyze C's feature interactions, i.e. determine the constraints of C's features.

In modeling subfeatures of the first-level features proceed as follows for each concept C:

1. From the set of identified first-level features of C select a not yet analyzed feature. Denote it F.
2. Describe F.
3. Determine F's presence rationale.
4. If F is a variable feature:
   (a) Determine F's inclusion rationale.
   (b) Determine F's binding time or mode.
5. Identify whether F has subfeatures (a concept reference cannot have subfeatures, see Section 4.3). If it does, then:
   (a) Identify F's direct common features and arrange them in the feature diagram with F as their parent.

(b) Identify F's direct variable features and arrange them in the feature diagram with F as their parent.

(c) Analyze F's feature interactions, i.e. determine the constraints of F's features.

(d) For each such an identified feature G of F perform this procedure from step 2 where F is set to G.

6. If F's subtree is repeated, F may represent a concept that has been overlooked. Consider introducing it as a concept in a separate feature diagram with F referring to it.

7. Analyze feature combinations and interactions:

   (a) Identify further constraints among the features and modify the corresponding features constraints accordingly.

   (b) Identify new features based on interactions of already identified features.

   (c) For each such an identified feature G perform this procedure from step 2 with F set to G.

Variable features binding times can be determined when the set of binding times defined in the solution domain becomes available. The set of binding times may be different for each solution domain. If the solution domain is unknown at the time of the application domain modeling, binding modes may be used instead of binding times (see Section 4.4).

The given order of steps of the procedures for obtaining feature models need not be followed strictly. The main purpose of introducing these procedures is to define precisely what has to be provided in a feature model.

Example 4.5 In Example 4.4 (page 41), four concepts have been identified in the text editing domain. The concept of a text editing buffer has already been presented in Example 4.1 (page 30). The only information that's missing is the information on binding: all the variable features that appear in the *Text Editing Buffer* feature diagram are statically bound. This example will consider the other three concepts identified in the text editing domain. Figure 4.9 shows their feature diagrams.

The concept of a file presented in Fig. 4.9a represents the state of a file being edited in a text editor. A file is identified by its name. A file may be read and written. The status presents an information on the last write operation performed on a file. A file may be of one of the several available types; there may be file types other than those presented in the diagram, so the concept is open. The alternative file type features of the *File* concept are bound dynamically because we need to be able to change the output file type at run time.

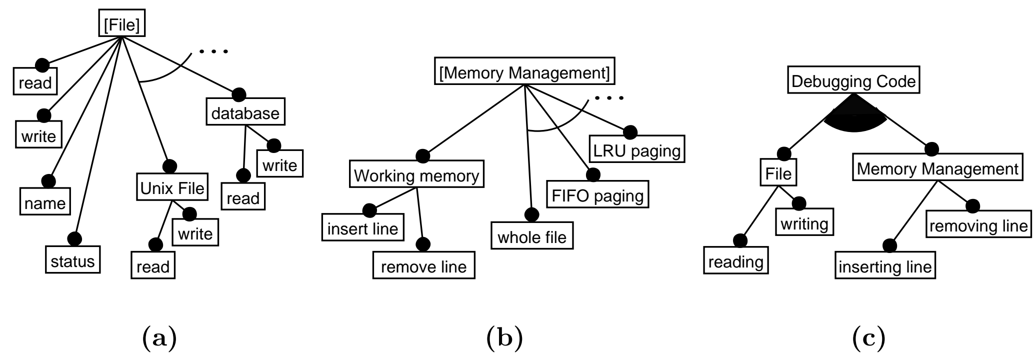


Figure 4.9: *File* (a), *Memory Management* (b), and *Debugging Code* concept (c) feature diagram.

The concept of memory management presented in Fig. 4.9b enables to deal with big files that cannot be loaded in the working memory at once on the line basis. Different memory management schemes are possible, which is modeled by alternative features; there may be memory management schemes other than those presented in the diagram, so *Memory Management* concept is open. This scheme is not supposed to change during the lifetime of a text editing buffer, so the alternative features of *Memory Management* concept are bound statically.

The concept of debugging code presented in Fig. 4.9c is intended to provide the debugging messages for development purposes. It is assumed here that the debugging code is provided either for file or memory management parts, or both, which is modeled by statically bound or-features. The file debugging is concerned with supplying the file type in case of reading, and the file status in case of writing. The memory management code displays the contents of the line being inserted into the working memory or removed from it. ■

4.10 Feature Modeling Tool Support

Feature modeling tool support is far from being satisfactory, although there are three tools that aim at such a support available.[9] This section will briefly present these tools.

4.10.1 ASADAL

ASADAL (version 1.0) [POS] is intended for use in conjunction with the FORM method [KKL+98] and besides feature modeling, it supports other models used in this method. However, ASADAL enforces FORM feature

[9]As of this writing, no other feature modeling support tools are available to the best knowledge of the author.

modeling notation that originates in FODA [KCH+90] and its layered feature model (briefly explained in Section 6.1), which makes it inappropriate for other methods.

4.10.2 AmiEddi

AmiEddi (version 1.3), available at [CE], and its successor Captain Feature [Cap] implement a more general Czarnecki-Eisenecker feature modeling notation [CE00]. Both tools store feature models in XML format. AmiEddi provides a repository model to organize large feature models. It supports different views of feature diagrams, in which some features may be hidden. Information associated with features is configurable through so-called metamodel editor [Bli01, CEH03].

AmiEddi lacks a mechanism for managing large feature diagrams as concept references. Also, it lacks a possibility to export associated information and a better export capabilities for both feature diagrams.

4.10.3 Captain Feature

Captain Feature (version 0.1) supports the extensions to the original Czarnecki-Eisenecker feature modeling notation proposed in [CBUE02] (briefly explained in Section 6.1.8).

In Captain Feature, the whole feature modeling notation should be configurable through a metamodel represented by a feature model [Bed02, CBUE02]. However, it seems[10] that it is not possible to edit this metamodel in Captain Feature. Also, it is not possible to create feature groups (to represent alternative and or-features), although these may be modified if they are already present in a diagram (as in the example file that comes with Captain Feature), but the problem that makes this tool completely unusable is its inability to save feature models.

[10] No user manual nor help system is available for Captain Feature.

Chapter 5

Multi-Paradigm Design with Feature Modeling

In this chapter a new method of multi-paradigm software development is proposed: *multi-paradigm design with feature modeling* (MPD_{FM}).[1] The method employs feature modeling, the modeling technique presented in the previous chapter.

The examples in this chapter introduced to illustrate solution domain feature modeling and transformational analysis in MPD_{FM} are related to AspectJ programming language [Ecla] (version 1.1.1). A whole AspectJ paradigm model, which defines MPD_{FM} for AspectJ, has been created and may be found in Appendix B.[2] Also, an application of MPD_{FM} for AspectJ to the domain of feature modeling is presented in Appendix C. Based on the results presented in these two appendices, the method has been evaluated.

Section 5.1 presents the process of MPD_{FM}. Solution domain feature modeling is presented in Section 5.2. Transformational analysis is presented in Section 5.3. Code skeleton design is described in Section 5.4. Section 5.5 evaluates the method.

5.1 The Process of Multi-Paradigm Design with Feature Modeling

MPD_{FM} follows the same process framework as multi-paradigm design (described in Section 3.5). However, all activities in MPD_{FM} are based on feature modeling. Figure 5.1 shows the four activities MPD_{FM} consists of:

Application domain feature modeling results in an application domain feature model.

---

[1] This chapter is based on [Vra].

[2] The AspectJ paradigm model presented in this thesis is based on the preliminary work presented in [Vra01a, Vra02a].

Solution domain feature modeling results in a solution domain feature model, i.e. paradigm model.

Transformational analysis establishes an application to solution domain mapping.

Code skeleton design translates an application to solution domain mapping into the code.

Application domain feature modeling follows the general process of feature modeling described in Chapter 4. Solution domain feature modeling has some peculiarities, so it will be described in detail (in Section 5.2) along with transformational analysis and code skeleton design (Sections 5.3 and 5.4).

The inputs to $\mathrm{MPD_{FM}}$ are:

Application domain related information contained in domain knowledge, the information about stakeholders, requirements, information about existing systems in the domain, etc.

Solution domain related information actually the information about the programming language that is going to be used, which is contained in the programming language textbooks and manuals, experience from existing systems, etc.

The output of $\mathrm{MPD_{FM}}$ is the code skeleton. However, the application domain feature model and solution domain feature model are also useful intermediate outputs of $\mathrm{MPD_{FM}}$. Once created, the solution domain feature model can be reused in transformational analysis of all the application domains that are to be implemented in that solution domain. The application domain feature model can be reused in transformational analysis if its implementation in another solution domain is required.

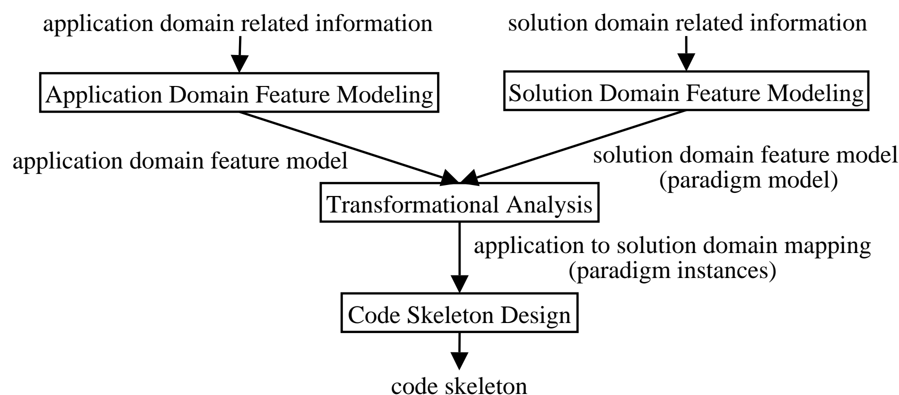


Figure 5.1: Multi-paradigm design with feature modeling.

The place of MPD_{FM} in the overall process of software development is shown in Fig. 5.2. Detailed design and implementation is based on the final result of MPD_{FM}, the code skeleton, as well as some of its intermediate results: the application domain feature model and application to solution domain mapping. At that point, the paradigm-specific methods pointed to by the small-scale paradigms selected in transformational analysis of MPD_{FM} may be used.[3]

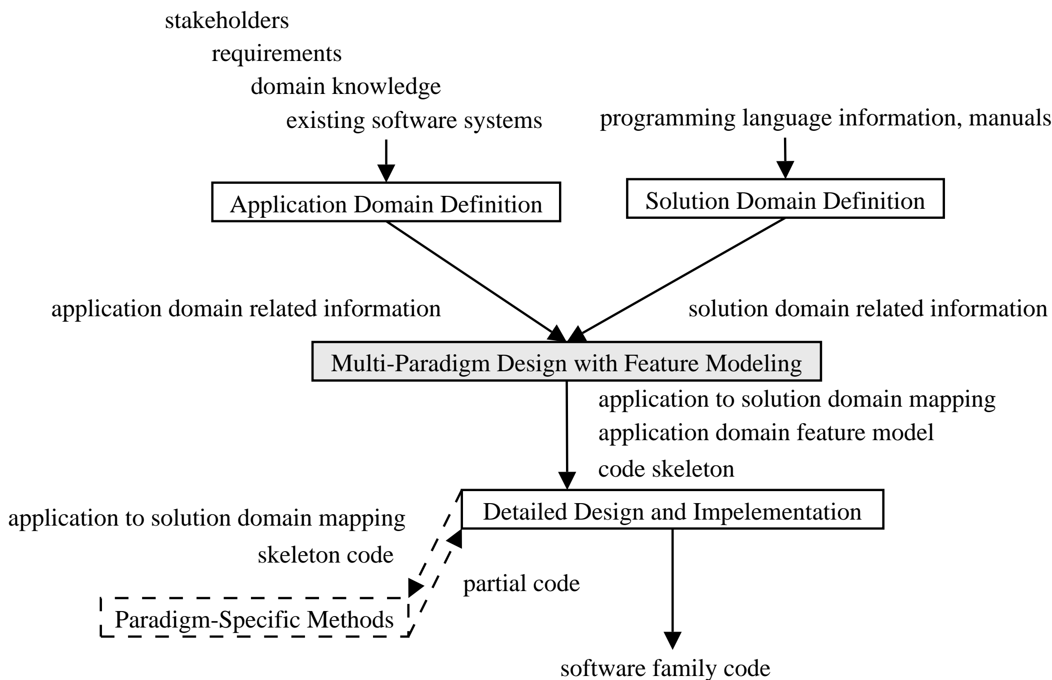


Figure 5.2: Multi-paradigm design with feature modeling and other activities during software development.

5.2 Solution Domain Feature Modeling

A solution domain is a domain in which a solution is to be expressed. A solution domain feature model, obtained by applying feature modeling to the solution domain related information, expresses the paradigms the solution domain supports. A paradigm is modeled as a solution domain concept.

The solution domain is here assumed to be a programming language, but other means of solution that are at disposal and that can be modeled as configurations of commonality and variability, as in case of multi-paradigm

[3]This is possible due to a fact that large-scale paradigms appear to be consisting of small-scale paradigms. This is discussed in Section 2.3.

design for C++ [Cop99b] (e.g., non-Alexandrian [Cop00, Vra02b] design patterns [GHJV95]), can be embraced in a model, too).

The solution domain concepts are the paradigms it provides. This is in compliance with the definition of a small-scale paradigm as a configuration of commonality and variability (see Section 2.3). Solution domain feature models will be referred to also as *paradigm models*.

Solution domain feature modeling starts by identifying the paradigms supported by it, which is described in Section 5.2.1. It proceeds with identifying binding times of the solution domain, described in Section 5.2.2, and with the actual modeling of the paradigms, described in Sections 5.2.3 and 5.2.4.

5.2.1 Identifying Paradigms

The identification of paradigms starts by the language concepts that can be used directly at the topmost level of programs. Such paradigms are *directly usable*. An example is the class paradigm in AspectJ programming language.

A directly usable paradigm may also be used by other paradigms; if it doesn't, then it is a *pure directly usable* paradigm. All other paradigms are *indirectly usable* paradigms. The important subconcepts of the directly usable paradigms constitute indirectly usable paradigms. An example is the method paradigm in AspectJ, which, unlike the class paradigm, can be used only inside of a class or aspect.

There may be several levels of indirectly usable paradigms. However, the first-level indirectly usable paradigms would probably be sufficient. This issue must be solved with respect to the purpose of the paradigm model, which is its use in transformational analysis. It is not feasible to model all the language constructs as paradigms. Much of such low-level paradigms would never be used during transformational analysis because the application domain feature model would be far less detailed. For example, a method in AspectJ may contain an assignment construct, so there could be an assignment paradigm. On the other hand, an application domain feature model would hardly mention assignments, so having the assignment paradigm in the paradigm model is futile.

During the identification of paradigms, it is useful to draw a hierarchy of paradigms. Fig. 5.3 presents a hierarchy of AspectJ paradigms. In the paradigm model, each parent paradigm would reference its daughter paradigms.

5.2.2 Identifying Binding Times

Based on the solution domain related information, the binding times the solution domain provides should be determined. For this, we must know

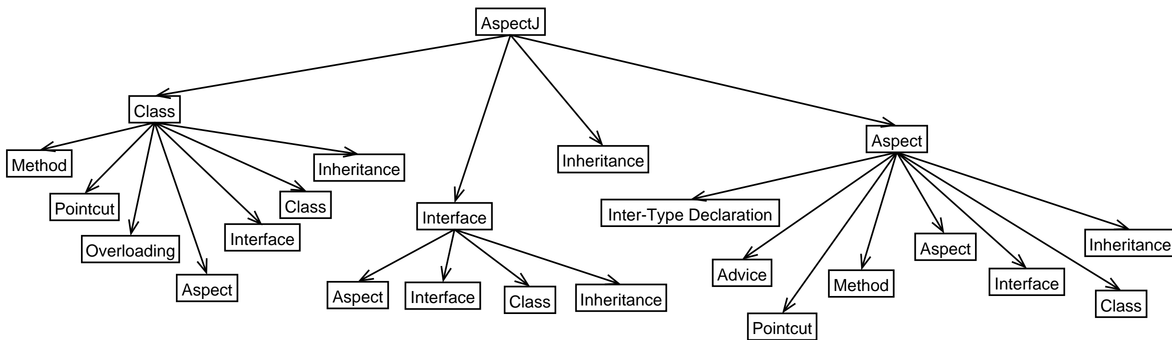


Figure 5.3: The hierarchy of AspectJ paradigms. The arrows mean "may use".

whether programs in a given programming language are compiled or interpreted, whether they are loaded at once or part-by-part, etc.

Usually, binding times include:

source time the time of program source code writing, when a programmer explicitly decides what is performed (e.g., that a class will provide some method)

compile time the time of program source code compiling, when decisions are made by a compiler (e.g., which method to select among the overloaded ones)

link time the time of compiled code units linking, when decisions are made by linker which precompiled code to link to a program

load time the time of program loading, when decisions are made by loader

run time the time of program running, when decisions are made by the running program

Binding times, as any other times, are comparable. There are later and earlier binding times. Thus, the binding times of a solution domain may be arranged into a sequence. The list of binding times introduced above forms such a sequence where the earliest binding time is source time, and the latest one is run time.

5.2.3 First-Level Paradigm Model

Having identified the paradigms, we may proceed with feature modeling. The directly usable paradigms represent the subconcepts of the solution concept, so their references should appear as features of the solution concept. As any other concept, the solution concept should be described.

To create a first-level paradigm model proceed as follows for each directly usable paradigm P:

1. If P may appear more than once in a program, introduce its reference in the solution domain feature diagram in plural, otherwise in singular. If needed, model the plural forms as individual concepts or as a parameterized concept.

2. Determine the variability of P's reference according to the restrictions posed by the programming language.

3. If P's reference is a variable feature, determine of P's reference binding time (usually source time).

4. Analyze solution concept feature interactions, i.e. determine the initial constraints among paradigms.

Example 5.1 The feature diagram of AspectJ programming language first-level paradigm model is presented in Fig. 5.4. All the directly usable paradigms of AspectJ are modeled as source time bound optional features of an AspectJ program as a solution concept.

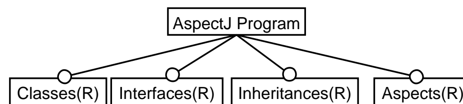


Figure 5.4: First-level AspectJ paradigms.

They are introduced in plural, since they may appear more than once in the same AspectJ program. The plural form is modeled by *<Plural Form>* parameterized concept presented in Fig. 5.5 (repeated from Fig. 4.5 for convenience) mentioned in Section 4.6.2. The concept *<Plural Form>* may be any one of AspectJ paradigms or *Type* concept (see Example 5.2 on page 53). The name *<Plural Form>* is the plural form of *<Singular Form>*. All the variable features have the source time binding. ■

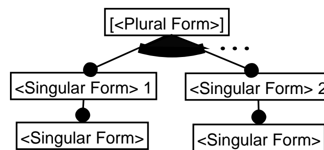


Figure 5.5: Dealing with the plural form of concepts using a parameterized concept.

5.2.4 Modeling Individual Paradigms

A paradigm is a concept and as such is presented in a separate feature diagram. Features of paradigms may be found in the solution domain related

information. Paradigms that may be used in the paradigm being modeled should be referenced by it. Further features may emerge from the interaction of already identified features.

The first-level features of a paradigm are identified similarly as the first-level features of any other concept (see Section 4.9). In addition, if the paradigm enables instantiation, it has to be modeled as a feature (or several features).

The subfeatures of the first-level paradigm features are modeled similarly as the subfeatures first-level features of any other concept (see Section 4.9). However, there are two differences. First, a binding time for each variable feature must be introduced, and it cannot be replaced by a binding mode. Second, there is no need for introducing the presence rationale for features, as their presence is practically defined by the solution domain.

If some feature's subtree is repeated, it may represent a concept. In a solution domain feature model, this concept may be a paradigm. Even if it is not a paradigm, it should be introduced in a separate feature diagram. Such concepts will be denoted as *auxiliary concepts* further (see Examples 5.2 and 5.3). In both cases, such a feature would be introduced in a separate feature diagram and referenced as needed.

Much of the paradigms correspond to the main constructs, i.e. structures, of the programming language. In transformational analysis, a node in the application domain feature model can match with the root of such a *structural paradigm*. The AspectJ structural paradigms are: class, interface, method, and aspect. The aspect paradigm will be considered as an example.

Example 5.2 The aspect paradigm is presented in Fig. 5.6. The aspect paradigm enables to articulate related structure and behavior that crosscuts otherwise possibly unrelated classes, interfaces, and other aspects (only static aspects are allowed) into a named unit.

An aspect is similar to a class in the sense that it also embodies related structure (fields) and behavior (methods). But this structure and behavior is used only to support the croscutting, which is achieved by two paradigms an aspect is a container of: the advice and inter-type declaration. In addition, the pointcut paradigm is used to specify the join points (where the aspect is to be attached).

As classses, aspects can also be instantiated, but the instantiation is automatic. By default, an aspect is a singleton, i.e. there is a single aspect per Java virtual machine. Further, it is possible to declare that an aspect instantiates per each of the specified objects (executing or target ones) at any of the join points specified by a pointcut or per each flow of control (as it is entered or below it) of the join points specified by a pointcut.

Aspects can be priviledged in order to override the access rules of the elements they crosscut. The aspect paradigm enables employing (inside of

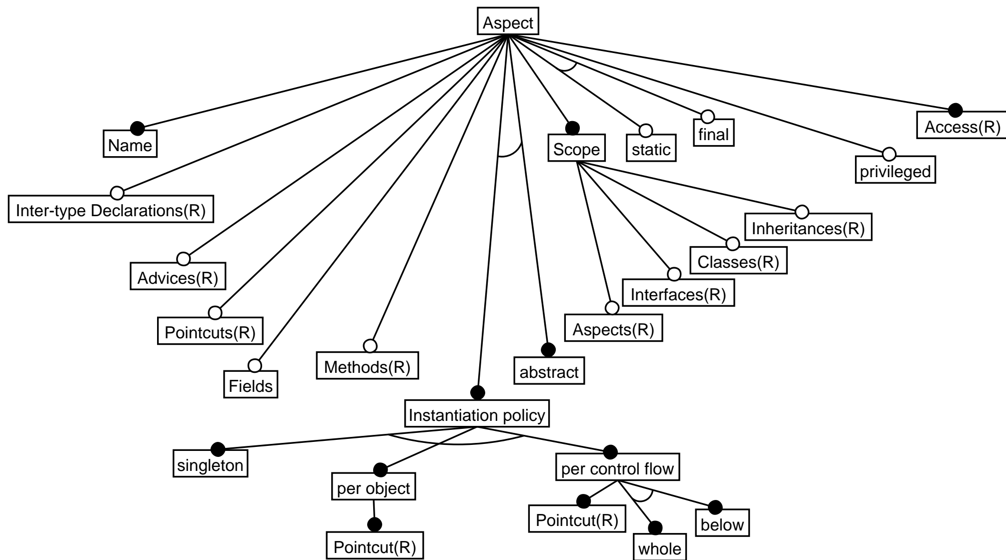


Figure 5.6: The aspect paradigm in AspectJ.

it) the same paradigms as the class paradigm beside inter-type declarations and pointcuts, which have a special position in it.

The parts of an aspect (without considering inheritance) are known at source time, which means that all the variable features presented in Fig. 5.6 have the source time binding.

The access to an aspect may be controled, which is modeled by an auxiliary concept *Access* (see Fig. 5.7). The default value is the package access.

The following constraint is associated with the aspect paradigm feature diagram:

$$abstract \veebar final$$

which means that the aspect is either final, or abstract.

A default dependency rule associated with the aspect paradigm feature diagram:

$$Instantiation\ policy \Rightarrow Instantiation\ policy.singleton$$

defines that the instantiation policy is by default single. ■

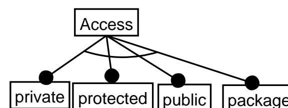


Figure 5.7: The access concept.

Besides structural paradigms, there are also paradigms that are about the relationship between some language structures. In transformational analysis, no single node in the application domain feature model will match with the root of such a *relationship paradigm*. The AspectJ relationship paradigms are: inheritance (a relationship between classes), overloading (a relationship between methods), inter-type declaration, advice, and pointcut. The last two will be considered as examples.

Example 5.3 Inside of an aspect, the advice paradigm (see Fig. 5.8) may be used to articulate the actions to be performed in the context of the join points specified by the pointcut. An advice provides a piece of code (in its body) to be run before, after, or in place (around) of a pointcut. The body of an advice is similar to the body of a method.

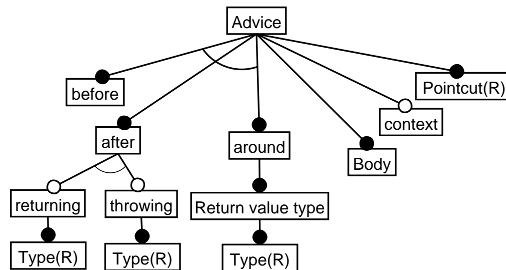


Figure 5.8: The advice paradigm in AspectJ.

An after advice can run after the execution of each join point specified by the *Pointcut*® completes normally, after it throws an exception, or after it does either one. In the last case, no matching based on the type, which is modeled by an auxiliary concept (see Fig. 5.9), being returned or exception being thrown can be made.

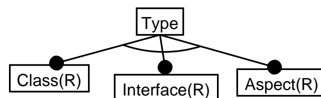


Figure 5.9: The type concept.

Around advice returns a value which will replace the original one at each join point specified by the *Pointcut*®. An advice can use a context exposed by its pointcut. The original join point return value may also be captured and returned, modified or not, by letting the original join point execute inside of the advice body. However, this AspectJ paradigm model does not go into such details as they could hardly be used in the transformational analysis. ■

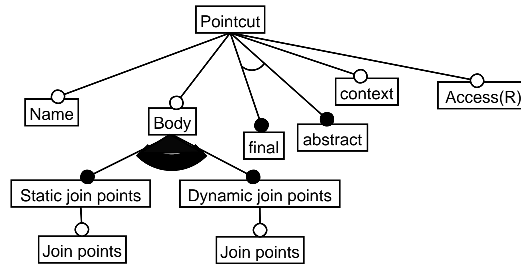


Figure 5.10: The pointcut paradigm in AspectJ.

Example 5.4 The pointcut paradigm (see Fig. 5.10) enables to specify the join points. Two kinds of join points exist: static and dynamic join points. Both are specified at source time, but are really determined later; static join points, such as method calls or executions, are determined at compile time, while dynamic join points, such as all method calls performed by an object of some type, may be determined only at run time. This means that the *Static join points.Join points* feature has the compile time binding, while *Dynamic join points.Join points* has the run time binding.

A pointcut is a logical expression formed out of the primitive pointcuts and the pointcuts already defined. It can be named or not (if it is specified directly in the place of its use). A pointcut can expose the context, i.e. an object or its fields, caught by some of the primitive pointcuts.

The access to a pointcut may be controled, which is modeled by an auxiliary concept *Access* (see Fig. 5.7). The default value is the package access.

The following two constraints are associated with the pointcut paradigm feature diagram:

$$abstract \veebar Body$$

$$Access \Rightarrow Name$$

which means that an abstract pointcut cannot have a body (or vice versa), and that only a named pointcut may have an access type specified. ■

5.3 Transformational Analysis

Transformational analysis in multi-paradigm design is a process of finding the correspondence and establish the mapping between the application and solution domain concepts. Transformational analysis in MPD_{FM} is performed as *bottom-up instantiation of paradigms over application domain concepts at source time.*

The input to transformational analysis are two feature models: the application domain one and the solution domain one. The output of transformational analysis is a set of paradigm instances annotated with application domain feature model concepts and features. Before presenting the process of transformational analysis and providing an example of it, the key issue of it—paradigm instantiation over application domain concepts—will be explained.

5.3.1 Paradigm Instantiation

In $\mathrm{MPD}_{\mathrm{FM}}$, paradigm instantiation is performed over application domain concepts. Paradigm instantiation over application domain concepts is a bottom-up instantiation of paradigms as solution domain concepts (see Section 4.7) in which the inclusion of paradigm nodes is stipulated by the mapping of the application domain concept nodes to them. By this, the correspondence of the paradigm instances to application domain concepts is enforced.

However, not all nodes of application domain concepts and paradigm instances need to be mapped. An inner[4] node of an application domain concept or paradigm feature diagram may act as an auxiliary node to ease the categorization of subfeatures. A feature represented by such a node may have no counterpart in the other domain.[5] Such nodes will be denoted as *mediatory*.

Further, there may (and usually will) be a mismatch in detailedness between the application and solution domain feature model. If solution domain feature model is more detailed, features of some paradigms or even some indirectly usable paradigms will not be mapped to in transformational analysis, but in spite of that they may be included in paradigm instances if determined so from the application domain concept semantics. In case of the application domain feature model is more detailed, there may be no corresponding nodes of the solution domain feature model for some of the non-mediatory nodes or even whole application domain concepts.

Any other non-mediatory feature diagram node of an application domain concept has to be mapped to the corresponding node of a paradigm instance. In general, only the correspondence between the nodes of the same category may be considered, i.e. between two concepts or between two features. Note that concept references are also features (as defined in Section 4.3). Further, semantics (according to the description) of the two nodes have to correspond to each other.

The binding times of the nodes being mapped must correspond. For the purposes of the binding time comparison, mandatory features are treated as if they have the earliest binding time the solution domain provides (which

[4]An inner node of a tree is a non-root and non-leaf node.

[5]However, there may be other mappings in which such a feature would be mapped.

is usually the source time, as discussed in Section 4.4.1). The binding time correspondence may mean equality, but it may be relaxed to mean that the binding time of the paradigm feature may not be earlier than required by the application domain concept feature (as that can only prolong the execution time).

If binding modes were used in the application domain analysis instead of binding times, then the correspondence between the application domain binding modes and the solution domain binding times has to be established. However, in most cases, run time binding corresponds to dynamic binding mode, and the rest of binding times correspond to static binding mode.

In addition, if features are bound later than at the instantiation time, their variability types must correspond, too. To a certain extent, a paradigm instance may accommodate to the variability type needed by an application domain concept (see step 3 of concept instantiation introduced in Section 4.7).

Each mapping between the nodes should be recorded in the form of an annotation, which is graphically presented by connecting the nodes with a dashed line. Annotations other than the feature diagram nodes of an application domain concept should be introduced in dashed boxes. For example, some paradigm features may have specific values intended for use in the code skeleton design (e.g., a name of the class).

5.3.2 The Process of Transformational Analysis

Transformational analysis is performed as follows. For each concept C from the application domain feature model, the following steps are performed:

1. Determine the structural paradigm corresponding to C:
    (a) Select a structural paradigm P of the solution domain feature model that has not been considered for C yet.
    (b) If there are no more paradigms to select, there may be a level mismatch: C may correspond to a paradigm feature, and not to a paradigm itself. Unless C has been factored out as a concept in step 1d, continue transformational analysis considering C only as a feature of the concepts where it is referenced, and not as a concept. Otherwise, the process has terminated unsuccessfully.
    (c) Try to instantiate P over C at source time. If this couldn't be performed or if P's root doesn't match with C's root, go to step 1a. Otherwise, record the paradigm instance created.
    (d) If there are unmapped non-mediatory feature nodes of C left, factor out them as concepts (introducing concept references in place of the subtrees they headed) and perform the transformational analysis of them. Subsequently, regard them as concept

references in C's feature diagram and reconsider the paradigm instance created in step 1c.

2. If there are relationships (direct or indirect ones) between the concept node of C and its non-mediatory features not yet mapped to relationships between the corresponding paradigm feature model nodes, determine the corresponding relationship paradigms for each such a relationship:

    (a) Select a relationship paradigm P of the solution domain feature model that has not been considered for a given relationship in C yet. If there are no more paradigms to select, the process has terminated unsuccessfully.

    (b) Try to instantiate P over the relationship in C at source time. If this couldn't be performed or if there are no P's nodes that match with the C's relationship nodes, go to step 2a. Otherwise, record the paradigm instance created.

Paradigm instances could be presented in the overall solution instance tree. However, this is not convenient since the solution instance tree would be too big to cope with it, and it would not provide any additional benefits compared to presenting paradigm instances individually.

A successful transformational analysis results in only one of the possible solutions. Carrying out transformational analysis differently can lead to another solution. Deciding which solution is the best is out of the scope of this method.

Example 5.5 Consider again the text editing buffers debugging code concept from Example 4.5 (page 43). Assume that the *File* feature matches with the class paradigm, and that its features *read* and *write* represent methods, while *name* and *status* are its attributes. Further, assume that the file types inherit from this base file class. In the following example, the transformational anaysis of the file debugging code will be performed (*Debugging Code.File*).

As has been presented in Example 4.5, the file debugging code consists of reading and writing part. *Debugging Code.File.reading* is concerned with reading files and supposed to provide an information on the type of the file before it has been read. *Debugging Code.File.writing* should provide an information on the status of the file after it has been written to.

One could choose the method paradigm for both these features because they represent functionality. However, a more careful examination of the description of the two features given in the previous paragraph reveals that this functionality is performed in connection with some other functionality. Recalling that the debugging code should be plugable, and thus separated

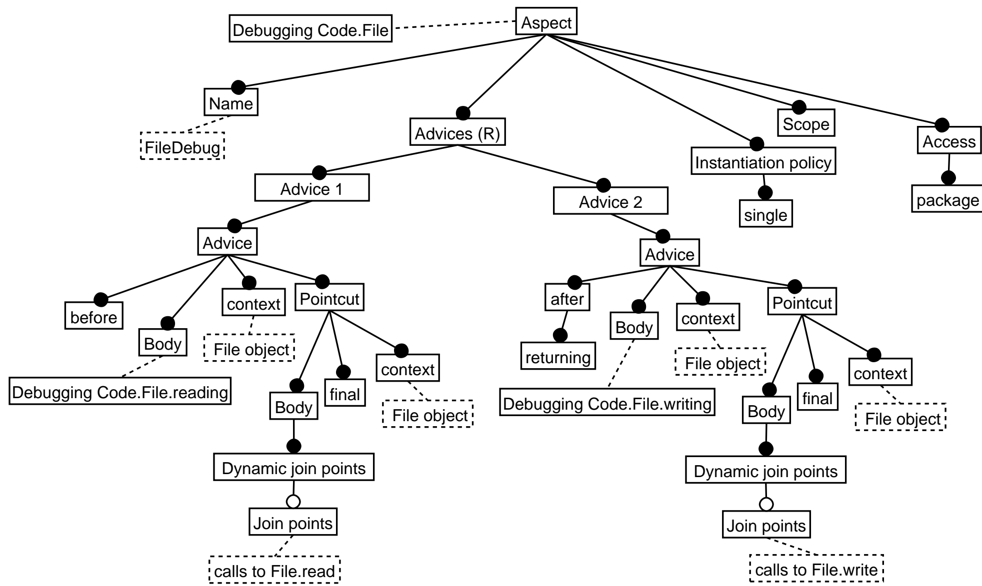


Figure 5.11: The file debugging code concept transformational analysis; an aspect with two advices.

from the rest of the code as much as possible, brings us to another form of expressing functionality in AspectJ: the advice paradigm.

As shown in Fig. 5.11, both *Debugging Code.File.reading* and *Debugging Code.File.writing* match with the body of a separate advice. An advice performs its actions with respect to the join points specified by a pointcut. In both cases, the pointcut would be unnamed, as it is needed only for this one application, and thus final (*Pointcut.final*). The context of the read method execution object would be needed to determine the file type in reading file advice and file status in writing file advice. Thus, the context should be exported by the pointcut (*Pointcut.context*) to be used by the advice (*Advice.context*). The reading file advice should be run before (*Advice.before*) the calls to *File.read* method, while the writing file advice should be run after (*Advice.after*) the calls to *File.write* method.

Note that Fig. 5.11 presents actually five paradigm instances: two pointcuts, two advices, and one aspect. Since paradigm instances are concept instances (see Section 4.7), and concept instances are specialized concepts, each paradigm instance could be presented in a separate diagram, as well, with enclosing paradigm referencing the enclosed paradigm instances. ■

5.4 Code Skeleton Design

Code skeleton design is performed by traversing paradigm instances and writing the source code manually. The paradigm instances obtained in trans-

formational analysis define the code skeleton, but the notes made during transformational analysis (as those accompanying the feature model element transformational analysis example) may also help mold the skeleton more accurately and make it more concrete.

Code skeleton design consists of the two major steps:

1. Transform the paradigm instances of structural paradigms into code.

2. Transform the paradigm instances of relationship paradigms into code.

The first step produces the basis the second one builds upon. This is because relationship paradigms are usually not represented by independent syntactical structures. Rather, they are attached to the syntactical structures representing structural paradigms.

Transforming a paradigm instance into the code means to analyze it and produce the corresponding piece of code according to the syntax of the programming language which serves as a solution domain.

Example 5.6 Following the transformational analysis of the file debugging code concept presented as a paradigm instance in Example 5.5 (page 59), the following code could be written:

```
aspect FileDC {
     before(File f): target(f) && call(* File.read(..)) { . . . }
     after(File f): target(f) && call(* File.write(..)) { . . . }
}
```

The code represents an aspect with two advices. The first one is being executed before reading any file, and the second one after writing each file. Both advices expose the current `File` object which is to be utilized in the advice bodies in order to output the file type in the first advice, and file status in the second advice. ■

5.5 Method Evaluation

In order to evaluate the proposed method of multi-paradigm design with feature modeling, it has been applied to an application and solution domain of a considerable size. The evaluation is concerned with the method scalability and reusability, and readability and comprehensibility of the artifacts created in its application. The results of this application will be evaluated in this section.[6]

[6] An additional evaluation of the method with respect to the related approaches is provided in Chapter 6.

5.5.1 Application Domain Feature Modeling

The application domain feature modeling of $\mathrm{MPD_{FM}}$ has been applied to the domain of feature modeling itself. Its model may be found in Appendix A. In the domain of feature modeling, twelve concepts have been identified. The maximum height of the feature diagrams in the model is two, and the largest feature diagram consists of 23 nodes (the one that represents the concept of feature diagram). According to these figures, the model should be quite readable and comprehensible.

Concept references enable to avoid producing bigger feature diagrams both in case the application domain scope is broad or a more detailed feature model of it is required.

Although not practically demonstrated in this thesis, feature models of application domains may be reused in transformational analysis. Such a reuse may arise in case of a transformation of the application domain into another solution domain, but a repeated transformation into the same solution domain may also be performed in order to explore alternative solutions. Actually, if binding times are used in an application domain feature model, it will depend on a solution domain which provides these binding times. In order to reuse such an application domain feature model in a transformational analysis with another solution domain, the binding times have to be adjusted to this solution domain or eliminated by using binding modes instead of binding times (see Section 4.4). In addition, an application domain feature model may be reused in other feature modeling based software development methods (such as FODA [KCH+90], FORM [KKL+98], or generative programming [CE00]), or vice versa. However, in both cases, an adaptation of an application domain feature model may be needed.

5.5.2 Solution Domain Feature Modeling

Solution domain feature modeling of $\mathrm{MPD_{FM}}$ has been applied to AspectJ programming language (version 1.1.1) resulting in an AspectJ paradigm model, which is provided in Appendix B. Creating a paradigm model of a solution domain can be viewed as a specialization of $\mathrm{MPD_{FM}}$ to that domain with respect to transformational analysis. Thus, the AspectJ paradigm model defines *MPD_{FM} for AspectJ*.

The AspectJ paradigm model consists of ten paradigms. The model includes directly usable and first-level indirectly usable paradigms. Also, four auxiliary concepts have been identified, making the total number of fourteen feature diagrams in the model. The maximum height of the feature diagrams in the model is three. The largest feature diagram consists of 25 nodes (the one that represents the aspect paradigm). These figures are similar to those of the application domain model, which suggests that feature diagrams of similar size, quite acceptable regarding the readability

and comprehensibility, may be expected in other domains.

Subsuming only directly and first-level indirectly usable paradigms are considered, paradigm models of other programming languages may be expected to be of similar size or smaller since AspectJ includes the whole Java. However, as in application domain modeling, concept references enable to avoid producing bigger feature diagrams even in more detailed paradigm models, which will encompass other than first-level indirectly usable paradigms, too.

As expected, paradigm models may be reused independently of the application domain feature models. AspectJ paradigm model has been reused in this thesis: it was used both in transformational analysis of the domain of text editing buffers, and in transformational analysis of the domain of feature modeling.

5.5.3 Transformational Analysis and Code Skeleton Design

$\mathrm{MPD_{FM}}$ for AspectJ has been applied to the domain of feature modeling. The application of $\mathrm{MPD_{FM}}$ for AspectJ took part in transformational analysis and code skeleton design (presented in Appendix C).

Transformational Analysis

Transformational analysis the domain of feature modeling with the proposed AspectJ paradigm model has been successfully performed. Its output is presented in Appendix C, Section C.1. Height of feature diagrams of some paradigm instances created in this transformational analysis reaches six with the number of nodes exceeding fifty. This is caused by expanding concept reference nodes in paradigms during their instantiation (see Example 4.5 on page 59). Introducing each instance of the referenced concept in a separate feature diagram and referring to it in the main diagram using a concept reference would solve this problem. However, the compound paradigm instances should be preferred whenever possible in order to provide a better overview of related paradigm instances.

Regardless of the application or solution domain size, paradigm instantiation, the main activity in transformational analysis, involves only a few application domain concepts and only one paradigm, possibly with the paradigms it refers to. Therefore, increasing the number of application domain concepts or paradigms would not affect the complexity of paradigm instantiation.

Code Skeleton Design

As has been demonstrated in Appendix C, Section C.2, code skeleton is easily derived from paradigm instances. This process is performed sequentially, so the number of paradigm instances doesn't affect its complexity.

The order in which paradigm instances are being transformed matters. In general, as proposed in Section 5.4, the structural paradigm instances have to be transformed before transforming relationship paradigm instances. However, as has been shown in Section C.2, it is sufficient to keep in mind that the relationship paradigm instances cannot be transformed before transforming the relationship paradigm instances related to them. This is especially useful if related paradigm instances are groupped during transformational analysis, as are the paradigm instances presented in Section C.1.

Chapter 6

Related Approaches

In this chapter, multi-paradigm design with feature modeling ($\mathrm{MPD_{FM}}$) and related approaches are compared. Since $\mathrm{MPD_{FM}}$ is a feature modeling based method, Section 6.1 compares the feature modeling as used in $\mathrm{MPD_{FM}}$ with other approaches to feature modeling. Following that, Section 6.2 compares $\mathrm{MPD_{FM}}$ with three related methods.

6.1 Feature Modeling Techniques

Feature modeling originates from Software Engineering Institute (SEI), where it was used in a domain analysis method FODA (feature-oriented domain analysis) [KCH+90] developed there, which became a part of MBSE (model-based software engineering) [Sofb]. Recently, MBSE has been replaced by PLP (product line practices) [CDKT01, Sofa], which also employ feature modeling. An adapted version of FODA feature modeling is also a part of FORM (feature-oriented reuse method) [KKL+98].

Since the publishing of FODA in 1990, several approaches have adopted FODA feature modeling, often in an adapted version [GFd98, CE00, Gey00]. Some work has been devoted primarily to extending feature modeling as such (with respect to UML) [RBSP02, Cla01], or even to formalize it [JG02].

Czarnecki-Eisenecker feature modeling [CE00] generalized FODA feature modeling notation and accepted a more general notion of a feature from ODM (Organization Domain Modeling) in which features are associated with particular domain practitioners and domain contexts [Sim95], i.e. a feature is any concept instance property important to some of the stakeholders [CE00]. Such understanding of a feature has been adopted also by FORM [KKL+98], a direct ancestor of FODA.

Czarnecki-Eisenecker feature modeling is also more abstract than FODA or FORM feature modeling. In Czarnecki-Eisenecker feature modeling, relationships between a feature and its subfeatures don't have any predefined semantics; the relationship is fully determined by the semantics of subfea-

tures. FORM feature modeling defines three types of relationships of a feature to its subfeature: composed-of, generalization/specialization, and implemented-by. Moreover, each feature is classified as a capability, operating environment, domain-technology, or implementation technique feature.[1] According to their type, features are placed into one of the four layers feature diagrams are divided into. On the other hand, Matthias Riebisch argues against the classification of features according to FORM and proposes to classify features into functional, interface, and parameter features [Rie03]. Therefore, it seems that it is better not to enforce such predefined feature categories in feature modeling.

Feature modeling used in MPD_{FM} is based on Czarnecki-Eisenecker feature modeling. However, it introduces the following new concepts:

- concept instantiation with respect to feature binding time,
- concept instances represented visually by feature diagrams,
- concept references,
- parameterization in feature models,
- constraints and default dependency rules represented by logical expressions,
- a dot convention for referring to concepts and features, and
- representing cardinalities without compromising the principles of feature modeling.

Sections 6.1.1 to 6.1.6) compare these new concepts with related ones in other approaches to feature modeling. Section 6.1.8 explains the adaptation of the information associated with feature diagrams proposed in [CE00] to the needs of MPD_{FM}.

6.1.1 Concept Instantiation

Concept instantiation with respect to feature binding time (see Section 4.7) is a generalization of concept instantiation as proposed in [CE00].

6.1.2 Concept Instances Represented by Feature Diagrams

Compared to the set representation proposed in [CE00], even if the features are qualified as proposed in Section 4.1, feature diagrams are a more appropriate way to represent concept instances (see Section 4.7). Moreover, they enable to represent concept instantiation in time.

[1] This classification has been proposed already in [KCH+90], but since FODA was concerned with user visible features, it dealt only with (application) capabilities.

6.1.3 Concept References

The problem of coping with the complex feature diagrams has been recognized already in [CE00], where complex diagram are divided into a number of smaller diagrams, which then may be referred to in the main diagram by introducing their roots.

Concept references, introduced by $\mathrm{MPD_{FM}}$ feature modeling (see Section 4.3), are a logical extension of this idea. $\mathrm{MPD_{FM}}$ feature modeling specifies how the information associated with the concept applies to its references and how it may be adapted to the needs of a particular reference.

Concept references enable a concept to reference itself (directly or indirectly). This enables feature diagrams to be viewed as trees while being in conformance with the fact that feature diagrams in general are directed graphs.

6.1.4 Parameterization in Feature Models

Parameterization in feature models (see Section 4.6) enables both to reason about feature models in a more general way and to create generic feature diagrams.

6.1.5 Constraints and Default Dependency Rules as Logical Expressions

In $\mathrm{MPD_{FM}}$, constraints and default dependency rules are expressed concisely as logical expressions (see Section 4.5). Logical expressions are capable of expressing both mutual exclusions and requirements among features. In fact, a single logical expression may encompass both types of the constraints. In FODA feature modeling, as well as in Czarnecki-Eisenecker feature modeling, constraints are expressed by explicitly stating which feature is mutually exclusive or requires which other feature.

In [SRP03],[2] constraints are written in an adapted version of Object Constraint Language (OCL) used in Unified Modeling Language [Obj03]. It is merely a matter of preference whether to use OCL syntax or traditional mathematical symbols for logical connectives (e.g., implies vs. $\Rightarrow$). However, in [SRP03], constraints are also accompanied by the information to be passed to the developer who instantiates the concepts that, for example, another feature has to be selected. For example, the constraint:

```
(if not(selected(Audio)
        implies selected('Add Music'))
then
  info("Select 'Add Music'")
endif)
```

[2] Constraints and default dependency rules representation proposed in this thesis has been developed independently of the approach proposed in [SRP03].

```
and
(if not(selected('Add Music')
        implies selected(Audio)
then
  info("Select Audio")
endif)
```

introduced in [SRP03] in $\mathrm{MPD_{FM}}$ feature modeling would be written simply as:

$$\mathit{Audio} \Leftrightarrow \mathit{Add\ Music}$$

Incorporating messages to developers significantly reduces the readability of such constraints. Moreover, such messages to the developer may be generated or, even better, a whole constraint may be passed instead.

The proposed form of expressing constraints and default dependency rules may be applied also to the constraints expressed directly by feature diagrams. This way, a whole feature diagram may be represented as a set of logical expressions. For the purpose of a graphical representation, a set of views of the feature diagram could be then defined. For each view, the relationships that should be shown would have to be specified with respect that the feature diagram should be a tree. The new constraints for the feature diagram could be then calculated to avoid duplicity (some of the constraints would be expressed in the feature diagram). In order to distinguish the primary relationships between the features expressed in a feature diagram from the constraints associated with it, one of the views could be denoted as primary.

The need to represent feature diagrams in a graphically independent form has been identified also in [LKL02]. The formalized feature modeling proposed in [JG02] actually relinquishes the feature diagrams completely, and with them the primary relationships between the features, too (though allowing for the visualization of desired feature relationships).

6.1.6 Referring to Concepts and Features

To refer to a concept or features unambiguously, a common dot convention is used in $\mathrm{MPD_{FM}}$ feature modeling (see Section 4.1). A similar convention is used in FeatuRSEB [GFd98], though without taking into account domain names, which may lead to ambiguities when talking about concepts and features from several domains.

6.1.7 Representing Cardinalities

In the original Czarnecki-Eisenecker feature modeling, introducing feature cardinalities was strictly avoided arguing that since the only semantics of an edge is whether to assert a feature or not, cardinality would only mean

to assert it several times, which is useless [CE00, p. 117]. Instead, to model the cardinality as a feature was recommended, as shown in Fig. 6.1. In spite of this, a later work these authors participated in proposes to use the UML-style cardinalities with features [CBUE02] (only at the child site of the edge, of course). Also, a generalized form of alternative and or-features is introduced in which the number of features which may be included is specified also as a cardinality (which does not contradict to the original Czarnecki-Eisenecker feature modeling).

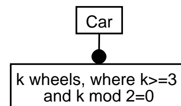


Figure 6.1: Representing cardinality in Czarnecki-Eisenecker feature modeling (from [CE00]).

As has been demonstrated in Section 4.6, plural forms of the concepts and cardinality in general can be specified by parameterized concepts without compromising the principles of feature modeling. If preferred, UML cardinalities can be used instead, provided they are defined as a notational extension with respect to the parameterized concept (see Section 4.6.3).

6.1.8 Information Associated with Concepts and Features

Information associated with feature diagrams subsumed by Czarnecki-Eisenecker feature modeling has been adapted to the needs of $\mathrm{MPD_{FM}}$ (see Sections 4.4).

The parts of the information associated with concepts and features not used in transformational analysis in $\mathrm{MPD_{FM}}$ have been left out: stakeholders and client programs, exemplar systems, availability and binding sites, and priorities. Since the openness of the features is expressed directly in feature diagrams, the open/closed attribute has been left out, too.

The information on rationale in Czarnecki-Eisenecker feature modeling explains why the feature is included (i.e., present) in the model and, if the feature is variable, it also introduces conditions and recommendations on its selection. In $\mathrm{MPD_{FM}}$ feature modeling, the two parts have been formally separated into the presence rationale and the inclusion rationale. In $\mathrm{MPD_{FM}}$ feature modeling rationales have clearly a recommendatory character, which is not that clear in Czarnecki-Eisenecker feature modeling, as rationale there introduces both *conditions* and recommendations.

A general approach to the issue of specifying feature binding (and availability, too) is proposed in [CE00]. Since this generality is not needed in $\mathrm{MPD_{FM}}$, a FODA approach to defining binding times [KCH+90] has been applied in $\mathrm{MPD_{FM}}$ feature modeling with binding times understood as in [Cop99b]. In addition, binding modes as defined in [CE00] have been used

with exception of the changeable binding mode, since it is only an optimized form of dynamic binding.

Constraints and default dependency rules are considered separately from the rest of the associated information (see Section 6.1.5), since in $\mathrm{MPD_{FM}}$ they apply to whole feature diagrams.

In [CE00], the description of the concepts and features is denoted as *semantic description*, which is somewhat superfluous, so only *description* has been used here (as in [KCH+90]).

6.2 Multi-Paradigm Approaches

Since $\mathrm{MPD_{FM}}$ is conceptually closest to *multi-paradigm design* (MPD), described in Section 3.5, the main part of this section is devoted to the discussion of differences and similarities between $\mathrm{MPD_{FM}}$ and MPD. However, $\mathrm{MPD_{FM}}$ is discussed in the context of two other related approaches: multi-paradigm programming in Leda and generative programming.

6.2.1 Multi-Paradigm Design

Both MPD and $\mathrm{MPD_{FM}}$ follow the same procedure consisting of the following main steps:

- application domain analysis,
- solution domain analysis,
- transformational analysis, and
- code skeleton design.

By employing feature modeling, $\mathrm{MPD_{FM}}$ introduces several improvements into this procedure. These will be discussed further in this section.

Feature Modeling and SCVR Analysis

As MPD, $\mathrm{MPD_{FM}}$ also applies the same means to both application and solution domain. While $\mathrm{MPD_{FM}}$ employs an adapted version of Czarnecki-Eisenecker feature modeling (presented in Chapter 4; differences discussed in Section 6.1), MPD employs scope, commonality, variability, and relationship (SCVR) analysis (see Sections 2.3 and 3.5).[3]

Feature modeling and SCVR analysis have much in common. SCVR analysis is based on the notions of commonality and variability (hence the name), and the notions of *common* and *variable* features are crucial in feature modeling.

[3] This section is based on [Vra01b, Vra01d].

A scope in SCVR analysis, defined as a set of entities [CHW98], is nothing but a concept in an *exemplar* representation.[4] Thus, SCVR analysis commonalities (assumptions held uniformly across the scope) and variabilities (assumptions true for only some elements in the scope) correspond to common and variable features of feature modeling, respectively. The scope is in [Cop99b] actually referred to as domain, so it can be concluded that a domain in MPD corresponds to a concept in feature modeling.

The main difference between feature modeling and SCVR analysis is that while commonalities and variabilities in feature modeling are primarily represented diagrammatically, which allows for representing hierarchical relationships between concepts, commonalities and variabilities in SCVR analysis are represented descriptively in family and variability tables.

In MPD, relationships between domains are represented using variability dependency graphs in which the nodes represent domains and the directed edges represent the "depends on (a parameter of variation)" relationship. Despite their simplicity, variability dependency graphs enable the identification of circular dependencies between domains (so-called codependent domains), and the identification of shared domains and their unification (i.e., reduction of variability dependency graphs). However, unlike feature modeling in $\mathrm{MPD}_{\mathrm{FM}}$, variability dependency graphs are used only in application domain analysis of MPD, and not in solution domain analysis, where representing hierarchical relationships between concepts, i.e. paradigms, is especially useful.

Parts of variability dependency diagrams can be derived from feature diagrams. Commonality domain depends on its parameters of variation, or—in the feature modeling terminology—a concept depends on its variation points. While the relationships between domains in variability dependency graphs have a particular semantics (dependence), the relationships in feature diagrams do not have any predefined semantics, which makes feature diagrams more abstract.

Table 6.1 aligns the terms of feature modeling with their variability and family table counterparts (the columns). The information provided in the variability and family tables is a subset of the information provided by a feature model.

While binding time in MPD is being denoted for a domain as a whole, feature modeling enables to specify binding time directly where it applies: at individual variable features.

In $\mathrm{MPD}_{\mathrm{FM}}$ feature modeling, instantiation is modeled by features. Its details are concept-dependent and modeled as subfeatures. An example is *Objects* feature of the class paradigm or *Instantiation policy* feature of the aspect paradigm (see Appendix B, Sections B.3.2 and B.3.4).

Negative variability, which is in SCVR analysis presented in separate ta-

[4]In exemplar representation, a concept is defined by the set of its instances [CE00].

Table 6.1: Feature modeling and MPD variability and family tables.

Feature modeling	**Multi-paradigm design**	
	Variability tables	**Family tables**
concept	commonality domain	language mechanism
common feature		commonality
variable feature		variability
variation point	parameter of variation	
alternative features	domain (of values)	
binding time	binding	binding
description (and rationales)	meaning	
default dependency rules	default (value)	
feature		instantiation

bles (negative variability tables), is in feature modeling expressed by features. The negative variability features of paradigms are actually their specializations. An example from MPD is the template specialization in C++ [Cop99b]. In MPD_{FM}, the template specialization would be a feature of the template paradigm.

Transformational Analysis

Performing transformational analysis as bottom-up instantiation of paradigms enables to overcome the MPD's problem of having to decide the conceptual correspondence between the paradigm and the parameter of variation at once (see Section 3.5).

MPD_{FM} does so by the means of decomposition. In MPD_{FM}, the conceptual correspondence is being decided bottom-up, so the correspondence of more complex concepts is decided according to the correspondence of their features, the correspondence of the features is decided according to the correspondence of the subfeatures (if any) and so on (see Section 5.3).

Transformational analysis in MPD_{FM} is performed between diagrams, i.e. in a visual fashion, which enables a better control over it than it is in MPD. Its output are annotated paradigm instances, which are more appropriate for this purpose than annotations of variability tables in MPD.

Code Skeleton Design

In MPD, transformational analysis results are only a guide in choosing a paradigm for an application domain structure. To construct a code skeleton, one must also follow the structure of the application domain and dependencies between the domains.

Annotated paradigm instances, the results of transformational analysis in MPD_{FM}, provide enough information about the mapping between the

application and solution domain concepts to obtain the main part of the skeleton code by traversing their trees. Augmenting the paradigm model with the translation rules would be a logical step towards the automation of transformational analysis in $\mathrm{MPD}_{\mathrm{FM}}$.

6.2.2 Multi-Paradigm Programming in Leda

A design method proposed in connection with multi-paradigm programming in Leda [KBV00] also aims at helping in the paradigm selection (see Section 3.4.2). However, it does so in a different way than $\mathrm{MPD}_{\mathrm{FM}}$.

While $\mathrm{MPD}_{\mathrm{FM}}$ is domain-oriented, Leda design method is concerned with the design of one system. In connection with that, $\mathrm{MPD}_{\mathrm{FM}}$ is based on feature modeling. Leda design method does not prescribe any special modeling technique.

The substantial difference is that $\mathrm{MPD}_{\mathrm{FM}}$ is performed in a bottom-up fashion, and Leda design method in a top-down fashion, which is related to the large-scale paradigm view it's being based on. Each small-scale paradigm in an $\mathrm{MPD}_{\mathrm{FM}}$ solution domain model is bound to a specific language feature. Large-scale paradigms may be understood as sets of small-scale paradigms. The elements of such a set often aren't precisely determined, which makes large-scale paradigms more abstract than small-scale paradigms (see Chapter 2). This naturally leads to its application to higher levels of abstraction first (starting at the top).

The selection of the main paradigm for the system or its part is a hard decision to make at once. Leda design method tries to help in arriving to this decision by comparing the impact of the selection of a paradigm to lower levels of the system. This is not needed in $\mathrm{MPD}_{\mathrm{FM}}$, since the paradigms are being selected in a bottom-up fashion.

6.2.3 Generative Programming

Although generative programming (see Section 3.3) is not primarily concerned with the paradigm selection, it is related to $\mathrm{MPD}_{\mathrm{FM}}$. Both approaches employ feature modeling, although with some differences (see Section 6.1).

Figure 6.2 shows the main phases of both generative programming and $\mathrm{MPD}_{\mathrm{FM}}$. The arrows between phases indicate the flow of results. The common phases are placed in the middle. As can be seen from the figure, one could do the domain scoping and application domain feature modeling without having to decide for either of the two approaches in these early phases. The difference is in the selection of paradigms: while in $\mathrm{MPD}_{\mathrm{FM}}$ it is performed directly as a matter of the primary concern, in generative programming it can be viewed as being delegated to the generator.

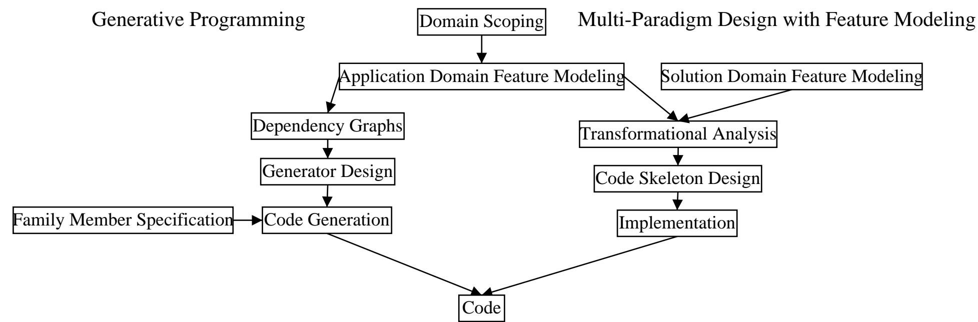


Figure 6.2: Generative programming and $\mathrm{MPD_{FM}}$.

Variability dependency graphs used in MPD to show the relationship between domains and their parameters of variation are similar to *component categories dependency graphs* used in generative programming to sort component categories into a GenVoca layered architecture. Although variability dependency graphs are actually not used in $\mathrm{MPD_{FM}}$,[5] they can be easily derived from $\mathrm{MPD_{FM}}$ feature diagrams (see Section 6.2.1).

Component categories dependency graphs are also being derived from feature models. A node in a component categories dependency graph represents either a component category or configuration repository (a composite node containing all the component categories that all other component categories depend on, but that do not depend on each other), and edges represent "uses" dependency (in the direction of an arrow). A configuration repository can be easily decomposed into a component categories dependency subgraph.

A component category is an abstraction of the component. When generating family members, a concrete component will take place of the component category. According to this, a component category represents a parameter of variation in the sense of MPD, or a singular variation point in the sense of feature modeling. The "uses" dependency has the same meaning as "depends on" relationship in MPD. According to [CE00], *A uses B* means that *B* is a *support* domain of *A* (e.g., "the domain of container packages is a support domain of the domain of matrix packages"), and this means that *A uses B* (i.e., the domain of matrix packages uses the domain of container packages).

[5] Variability dependency graphs were actually supposed to be integrated into $\mathrm{MPD_{FM}}$ [Vra01c], but were finally left out.

Chapter 7

Conclusions

Software development tends to be multi-paradigm. This can be seen in the approaches that integrate multiple paradigms, such as generative programming, multi-paradigm programming in Leda, or intentional programming. Even better example is aspect-oriented programming, which subsumes the existence of a base, i.e. *another*, paradigm, and thus is principally multi-paradigm, as opposed to intentionally created multi-paradigm approaches and programming languages.

Each multi-paradigm approach to software development has to deal with the issue of selecting an appropriate paradigm for a given application domain concept at least to some extent. As has been demonstrated by multi-paradigm design [Cop99b], where this process is seen as a mapping between the application (problem) and solution domain, it is possible to deal with this issue solely.

Multi-paradigm design with feature modeling ($\mathrm{MPD_{FM}}$), a new method of multi-paradigm software development proposed in this thesis, improves the multi-paradigm design by employing feature modeling to model both application and solution domain (Chapter 5). For this purpose, Czarnecki-Eisenecker feature modeling [CE00] has been extended and adapted (Chapter 4). The proposed extensions are applicable outside $\mathrm{MPD_{FM}}$, too.

By modeling paradigms as solution domain concepts, $\mathrm{MPD_{FM}}$ provides a better basis for the transformational analysis, the key activity of multi-paradigm design, in which paradigms (solution domain concepts) appropriate for given application domain concepts are being selected, has been proposed in terms of feature modeling as a bottom-up paradigm instantiation over application domain concepts (Section 5.3). Code skeleton, the final output of $\mathrm{MPD_{FM}}$, may then be obtained by traversing the trees of annotated paradigm instances, which represent the output of transformational analysis, and writing the source code manually, as explained in Section 5.4.

To obtain the whole code skeleton, transformational analysis should be performed for each application domain concept, as explained in Section 5.3.2.

It is also possible to do transformational analysis only of some application domain concepts (e.g., the critical ones) and do the rest of the design without $\mathrm{MPD_{FM}}$. The rest of the design would be restricted by such partial transformational analysis results.

$\mathrm{MPD_{FM}}$ offers several opportunities for reuse (see Section 5.5). Both application and solution domain feature models may be reused in different transformational analyses. Application domain feature models—possibly adapted—may also be reused in other feature modeling based software development methods (e.g., FODA [KCH+90], FORM [KKL+98], or generative programming [CE00]), or vice versa.

Creating a feature model of a solution domain can be viewed as a specialization of $\mathrm{MPD_{FM}}$ with respect to transformational analysis. Such a specialization of $\mathrm{MPD_{FM}}$ to AspectJ has been presented (see Appendix B) and applied in this thesis (in the examples in Chapter 5).

The following two sections summarize the contributions of this thesis and identify the possible directions of further work.

7.1 Summary of Contributions

The main contributions of this thesis are as follows:

1. Multi-paradigm design with feature modeling, a new method of multi-paradigm software development based on feature modeling which improves paradigm selection process (Chapter 5).
2. Multi-paradigm design with feature modeling for AspectJ, the method specialization to AspectJ defined by providing an AspectJ paradigm model (parts are introduced in Chapter 5; the complete model is presented in Appendix B).
3. Improvements of feature modeling (Chapter 4):
   (a) Concept instantiation with respect to instantiation time with concept instances represented by feature diagrams (Section 4.7).
   (b) Parameterization in feature models (Section 4.6).
   (c) Representing constraints and default dependency rules by logical expressions (Section 4.5).
   (d) Concept references to enable to deal with complex feature models (Section 4.3).
   (e) A dot convention to enable referring to concepts and features unambiguously (Section 4.1).
   (f) A parameterized concept for representing cardinality in feature modeling (Section 4.6.3).

4. A feature model of the domain of feature modeling itself, which provides a basis for further reasoning on this modeling technique (Appendix A).

5. An application of multi-paradigm design with feature modeling for AspectJ to the domain of feature modeling, which provides a basis for developing systems to support feature modeling (Appendix C).

In addition to contributions listed above, the thesis contributes to the understanding of the concept of paradigm in software development (Chapter 2), provides an overview of selected multi-paradigm approaches to software development (Chapter 3 and Section 6.2), and evaluates approaches to feature modeling (Section 6.1).

7.2 Further Work

$\mathrm{MPD}_{\mathrm{FM}}$ enables to reuse both application and solution domain feature models. These models are reused as a whole. However, some domains overlap, and this happens even if one of them is an application domain and the other one is a solution domain (e.g., <*Plural Form*> concept is in both AspectJ and feature modeling domain). Thus, the issue of overlapping domains is worth considering as a step towards reuse of individual concepts.

The reuse of individual concepts which are similar to each other would require their generalization. Subsequently, they would appear as specializations of a more general concept. This would be particularly useful for paradigm models of related programming languages.

Another interesting topic for further work would be experimenting with $\mathrm{MPD}_{\mathrm{FM}}$ specialization to solution domains other than programming languages, especially those that are used in conjunction with programming languages (e.g., design patterns).

Bibliography

[Ale79] Christopher Alexander. *The Timeless Way of Building.* Oxford University Press, 1979. Cited in [Cop99b].

[AOS] AOSD steering committee. Aspect-Oriented Software Development home page. `http://aosd.net`. Last accessed in March 2004.

[AT98] Mehmet Aksit and Bedir Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. In *Proc. of the Aspect-Oriented Programming Workshop at ECOOP'98*, Brussels, Belgium, 1998. Available at [Twe].

[AWB+93] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object-interactions using composition-filters. In *Proc. of 7thEuropean Conference on Object-Oriented Programming (ECOPP'93) Workshop*, LNCS 791, pages 152–184, Kaiserslautern, Germany, 1993. Springer. Available at [Twe].

[Bed02] Thomas Bednasch. Konzept und implementierung eines konfigurierbaren metamodells für die merkmalmodellierung. Master's thesis, Fachhochschule Kaiserslautern, Standort Zweibrücken, Fachbereich Informatik, 2002. In German. Available at [Eis].

[BG97] Don Batory and Bart J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering (special issue on Software Reuse)*, pages 67–82, February 1997. Available at [Pro].

[Bli01] Frank Blinn. Entwurf und implementierung eines generators für merkmalmetamodelle. Master's thesis, Fachhochschule Zweibrücken, Fachbereich Informatik, 2001. In German. Available at [Eis].

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, second edition, 1994.

[Bud95] Timothy A. Budd. *Multiparadigm Programming in Leda.* Addison-Wesley, 1995.

[Cap] Captain Feature. Project page. `https://sourceforge.net/projects/captainfeature`. Last accessed in March 2004.

[CBUE02] Krzysztof Czarnecki, Thomas Bednasch, Peter Unger, and Ulrich W. Eisenecker. Generative programming for embedded software: An industrial experience report. In D. Batory et al., editors, *Generative Programming and Component Engineering: ACM SIGPLAN/SIGSOFT*

Conference, GPCE 2002, LNCS 2487, pages 156—-172, Pittsburgh, PA, USA, October 2002.

[CDKT01] Gary Chastek, Patrick Donohoe, Kyo Chul Kang, and Steffen Thiel. Product line analysis: A practical introduction. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, June 2001. Available at [Sofb] (last accessed in February 2003).

[CE] Krzysztof Czarnecki and Ulrich W. Eisenecker. Generative programming - methods, tools, and applications. `http://www.generative-programming.org`. Last accessed in March 2004.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programing: Principles, Techniques, and Tools.* Addison-Wesley, 2000.

[CEH03] Krzysztof Czarnecki, Ulrich W. Eisenecker, and Simon Helsen. Generative programing: Methods, techniques, and applications. Slides and notes of the tutorial given at Net.ObjectDays 2003, September 2003. Available at `http://www.netobjectdays.org/pdf/03/slides/tutorial/gpce2003.pdf` (last accessed in March 2004).

[CHW98] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6), November 1998. Available at [Cop].

[Cla01] Matthias Clauβ. Modeling variability with UML. In *Proc. of Net.ObjectDays 2001, Young Researchers Workshop on Generative and Component-Based Software Engineering*, pages 226–230, Erfurt, Germany, September 2001. tranSIT.

[Cop] James O. Coplien. Home page. `http://www.bell-labs.com/people/cope`. Last accessed in March 2004.

[Cop99a] James O. Coplien. Multi-paradigm design and implementation in C++. Slides and notes of the tutorial given at *1st International Conference on Generative and Component-Based Software Engineering (GCSE'99)*, Erfurt, Germany, September 1999. Available at [Cop].

[Cop99b] James O. Coplien. *Multi-Paradigm Design for C++.* Addison-Wesley, 1999.

[Cop00] James O. Coplien. *Multi-Paradigm Design.* PhD thesis, Vrije Universiteit Brussel, Belgium, 2000. Available at [Cop].

[Cza] Krzysztof Czarnecki. Home page[1] `http://www.prakinf.tu-ilmenau.de/~czarn`. Last accessed in March 2004.

[Cza98] Krzysztof Czarnecki. *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Technical University of Ilmenau, Germany, 1998. Some chapters available at [Cza].

[1] This is Czarnecki's old home page, but it contains a lot of useful material. His new home page is `http://www.swen.uwaterloo.ca/~kczarnec`.

[Dem] Demeter group. Home page. `http://www.ccs.neu.edu/research/demeter`. Last accessed in March 2004.

[DVB01] Peter Dolog, Valentino Vranić, and Mária Bieliková. Representing change by aspect. *ACM SIGPLAN Notices*, 36(12):77–83, December 2001.

[Ecla] Eclipse.org. AspectJ project home page. `http://eclipse.org/aspectj`. Last accessed in March 2004.

[Eclb] Eclipse.org. Eclipse.org home page. `http://eclipse.org`. Last accessed in March 2004.

[Eis] Ulrich W. Eisenecker. Home page. `http://www.informatik.fh-kl.de/~eisenecker`. Last accessed in March 2004.

[Fil03] Robert E. Filman. A bibliography of aspect-oriented programming, version 1.22. Technical Report 03.01, Research Institute for Advanced Computer Science, NASA Ames Research Center, Moffett Field, California, June 2003.

[Flo79] Robert W. Floyd. The paradigms of programming. *Communications of the ACM*, 22(8):455–460, 1979.

[Gey00] Lars Geyer. Feature modelling using design spaces. In *Proc. of the 1st German Product Line Workshop (1. Deutscher Software-Produktlinien Workshop, DSPL-1)*, Kaiserslautern, Germany, November 2000. IESE.

[GFd98] Martin L. Griss, John Favaro, and Massimo d'Alessandro. Integrating feature modeling with the RSEB. In P. Devanbu and J. Poulin, editors, *Proc. of 5th International Conference on Software Reuse*, pages 76–85, Vicoria, B.C., Canada, 1998. IEEE Computer Society Press. Available at `http://favaro.net/publications` (last accessed in February 2003).

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

[IBM] IBM Research. Subject-Oriented Programming home page. `http://www.research.ibm.com/sop`. Last accessed in March 2004.

[JG02] Yu Jia and Yuqing Gu. The representation of component semantics: A feature-oriented approach. In Ivica Crnković, Stig Larsson, and Judith Stafford, editors, *Proc. of the Workshop on Component-based Software Engineering: Composing Systems From Components (a part of 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems)*, Lund, Sweden, April 2002. Available at `http://www.idt.mdh.se/~icc/cbse-ecbs2002` (last accessed in February 2003).

[KBV00] Charles D. Knutson, Timothy A. Budd, and Hugh Vidos. Multiparadigm design of a simple relational database. *ACM SIGPLAN Notices*, 35(12):51–61, December 2000. Available at `http://faculty.cs.byu.edu/~knutson/publications/Sigplan_Dec_2000_Knutson.pdf` (last accessed in February 2003).

[KCH+90] Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA): A feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA, November 1990. Available at [Sofb] (last accessed in March 2002).

[KE88] Timothy Koschmann and Martha Walton Evens. Bridging the gap between object-oriented and logic programming. *IEEE Software*, 60:36–42, July 1988.

[KKL+98] Kyo C. Kang, Sajoong Kim, Jaejoon Lee, Kijoo Kim, Euiseob Shin, and Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, January 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Vidiera Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proc. of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, Jyväskylä, Finland, June 1997. Springer. Available at [PAR].

[KOHK96] Matthew Kaplan, Harold Ossher, William Harrison, and Vincent Kruskal. Subject-oriented design and the watson subject compiler. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, 1996. Available at [IBM].

[Koo95] Piet S. Koopmans. On the definition and implementation of the Sina/st language. Master's thesis, Dept. of Computer Science, University of Twente, The Netherlands, August 1995. Available at [Twe].

[Kuh70] Thomas S. Kuhn. *The Structure of Scientific Revolutions.* University of Chicago Press, Chicago, 1970. Czech translation, OIKYMENH, 1997.

[Lie] Karl J. Lieberherr. Connections between Demeter/adaptive programming and aspect-oriented programming. Web document, College of Computer Science, Northeastern University, Boston, USA. Available at [Dem].

[Lie97] Karl J. Lieberherr. Demeter and aspect-oriented programming: Why are programs hard to evolve? Presentation slides, *3rd Conference Smalltalk und Java in Industrie und Ausbildung (STJA 97)*, Erfurt, Germany, 1997. Available at [Dem].

[LK98] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ. In *Proc. of 12th European Conference on Object-Oriented Programming (ECOPP'98) Workshops, Demos, and Posters*, LNCS 1543, Brussels, Belgium, July 1998. Springer. Available at [PAR].

[LKL02] Kwanwoo Lee, Kyo C. Kang, and Jaejoon Lee. Concepts and guidelines of feature modeling for product line software engineering. In Cristina Gacek, editor, *Proc. of 7th International Conference (ICSR-7)*, LNCS 2319, Austin, Texas, USA, April 2002. Springer.

[LLM99] Karl J. Lieberherr, David Lorenz, and Mira Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999. Available at [Dem].

[Mad00] Ole Lehrmann Madsen. Towards a unified programming language. In Jørgen Lindskov Knudsen, editor, *Proc. of 14th European Conference on Object-Oriented Programming(ECOOP 2000)*, LNCS 1850, Sophia Antipolis and Cannes, France, June 2000. Springer.

[Mer] Merriam-Webster OnLine. Merriam-Webster's Collegiate Dictionary. http://www.m-w.com. Last accessed in April 2004.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.

[NEC] NEC Research Institute. ResearchIndex: The NECI Scientific Digital Research Library. http://citeseer.nj.nec.com. Last accessed in March 2004.

[Náv96] Pavol Návrat. A closer look at programming expertise: Critical survey of some methodological issues. *Information and Software Technology*, 38(1):37–46, 1996.

[Obj] Object Management Group. UML resource page. http://www.omg.org/uml. Last accessed in February 2004.

[Obj03] Object Management Group. OMG unified modeling language specification, version 1.5. Technical report, March 2003. Available at [Obj].

[OHBS94] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proc. of 7th IBM Conference on Object-Oriented Technology*, July 1994. Available at [IBM].

[PAR] PARC. Software Design Area home page. http://www2.parc.com/sda. Last accessed in November 2001.

[POS] POSTECH SE Lab. ASADAL home page. http://selab.postech.ac.kr/realtime/public_html. Last accessed in February 2003.

[Pro] Product-Line Architecture Research group. Home page. http://www.cs.utexas.edu/users/schwartz. Last accessed in March 2004.

[RBSP02] Matthias Riebisch, Kai Böllert, Detlef Streitferdt, and Ilka Philippow. Extending feature diagrams with UML multiplicities. In *Proc. of the 6th Conference on Integrated Design and Process Technology (IDPT 2002)*, Pasadena, California, USA, June 2002. Society for Design and Process Science. Available at [Rie].

[Röd99] Lutz Röder. Transformation and visualization of abstractions using the intentional programming system. Presentation abstract, GCSE'99 Young Researchers Workshop, 1st International Conference on Generative and Component-Based Software Engineering, Erfurt, Germany, September 1999. Available at http://www.netobjectdays.org/mirrors/stja.cd/GCSE99_Young/Abstract_GCSE99YRW.htm (last accessed in March 2004).

[Rie] Matthias Riebisch. Home page. `http://www.theoinf.tu-ilmenau.de/~riebisch`. Last accessed in March 2004.

[Rie03] Matthias Riebisch. Towards a more precise definition of feature models. In M. Riebisch, J. O. Coplien, and D. Streitferdt, editors, *Modelling Variability for Object-Oriented Product Lines*, pages 64–76, Norderstedt, 2003. BookOnDemand Publ. Co. Available at [Rie].

[Roe] Lutz Roeder. Home page. `http://www.aisto.com/roeder`. Last accessed in April 2004.

[Sim95] Mark A. Simos. Organization domain modeling (ODM): Formalizing the core domain modeling life cycle. In *Proc. of the 1995 Symposium on Software reusability*, pages 196–205, Seattle, Washington, United States, 1995. ACM Press.

[Sim96] Charles Simonyi. Intentional programming—innovation in the legacy age, June 1996. Presented at IFIP WG 2.1 meeting.

[Sim99] Charles Simonyi. The future is intentional. *IEEE Computer*, 32(5):56–57, May 1999.

[SN97] Mária Smolárová and Pavol Návrat. Software reuse: Principles, patterns, prospects. *Journal of Computing and Information Technology*, 5(1):33–48, 1997.

[SN00] Mária Smolárová and Pavol Návrat. Reuse with design patterns: Towards pattern-based design. In Y. Feng, D. Notkin, and M.C. Gaudel, editors, *Proc. Software: Theory and Practice*, pages 232–235, Beijing, China, 2000. PHEI - Publishing House of Electronics Industry.

[SNB98] Mária Smolárová, Pavol Návrat, and Mária Bieliková. Abstracting and generalising with design patterns. In A. Gürsay U. Güdükbay, T Dayar and E. Gelenbe, editors, *Proc. of 13th International Symposium on Computer and Information Sciences (ISCIS'98)*, pages 551–558, Belek-Antalya, Turkey, 1998. IOS Press.

[Sofa] Software Engineering Institute, Carnegie Mellon University. A framework for software product line practice — version 3.0. `http://www.sei.cmu.edu/plp/framework.html`. Last accessed in February 2003.

[Sofb] Software Engineering Institute, Carnegie Mellon University. Home page. `http://www.sei.cmu.edu`. Last accessed in March 2004.

[SRP03] Detlef Streitferdt, Matthias Riebisch, and Ilka Philippow. Details of formalized relations in feature models using OCL. In *Proc. of the 10th IEEE Symposium and Workshops on Engineering of Computer-Based Systems (ECBS'03)*, pages 297–304, Pasadena, California, USA, April 2003. IEEE Computer Society. Available at [Rie].

[Twe] Twente Research and Education on Software Engineering (TRESE) group. Home page. `http://trese.cs.utwente.nl`. Last accessed in March 2004.

[Vra] Valentino Vranić. Feature modeling based transformational analysis in multi-paradigm design. Submitted to *Computers and Informatics (CAI)*, December 2003.

[Vra00a] Valentino Vranić. A concept of paradigm in the multi-pradigm software development. In *Proc. of 3rd Scientific Conference on Electrical Engineering and Information Technology for Ph.D. Students* ElITech 2000, Bratislava, Slovakia, 2000.

[Vra00b] Valentino Vranić. Multiple software development paradigms and multi-paradigm software development. In J. Zendulka, editor, *Proc. of 3rd International Conference on Information Systems Modelling (ISM 2000)*, pages 191–196, Rožnov pod Radhoštěm, Czech Republic, May 2000. MARQ.

[Vra00c] Valentino Vranić. Towards multi-pradigm software development. Presentation given at *4th Joint Conference on Knowledge-Based Software Engineering (JCKBSE 2000)*, Brno, Czech Republic., September 2000.

[Vra00d] Valentino Vranić. *Towards Multi-Pradigm Software Development.* Written part of the PhD examination, Slovak University of Technology in Bratislava, Slovakia, September 2000.

[Vra01a] Valentino Vranić. AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. In Jan Bosch, editor, *Proc. of 3rd International Conference on Generative and Component-Based Software Engineering (GCSE 2001)*, LNCS 2186, pages 48–57, Erfurt, Germany, September 2001. Springer.

[Vra01b] Valentino Vranić. AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. Technical report, Slovak University of Technology in Bratislava, Slovakia, May 2001.

[Vra01c] Valentino Vranić. Incorporating variability dependency graphs into multi-paradigm design with feature modeling. In *Proc. of 4th Scientific Conference on Electrical Engineering and Information Technology for Ph.D. Students* ElITech 2001, Bratislava, Slovakia, 2001.

[Vra01d] Valentino Vranić. A new basis for multi-paradigm design. Technical report, Slovak University of Technology in Bratislava, Slovakia, March 2001.

[Vra02a] Valentino Vranić. Multi-paradigm design with feature modeling. Lecture given at the Faculty of Organizational Sciences, University of Belgrade, Serbia., March 2002.

[Vra02b] Valentino Vranić. Towards multi-paradigm software development. *Journal of Computing and Information Technology (CIT)*, 10(2):133–147, 2002.

[VS95] Sanja Vraneš and Mladen Stanojević. Integrating multiple paradigms within the blackboard framework. *IEEE Transactions on Software Engineering*, 21(3):244–262, 1995.

Appendix A

Domain of Feature Modeling

This appendix presents a feature model of the application domain of feature modeling.

The domain of feature modeling has been selected for practical reasons. Since MPD_{FM} is a feature modeling based method, the MPD_{FM} CASE tool is among the software systems in this domain. Thus, this study is a step towards a tool support for MPD_{FM}. This feature model will be used transformational analysis presented in Appendix C.

This appendix is divided into two sections. Prior to the actual feature modeling, Section A.1 provides the domain definition and scope. Section A.2 presents the feature model.

A.1 Domain Definition and Scope

MPD_{FM} follows the process of domain engineering based approaches. Prior to the actual feature modeling, domain related information should be gathered and analyzed. This task is not an explicit part of MPD_{FM}. However, its results, the domain definition and scope, will be provided in this section, as they are needed further in feature modeling.

Sources of the information related to domain of feature modeling are the literature on feature modeling discussed in Section 6.1, including feature modeling for multi-paradigm design proposed in Chapter 4. Also, available feature modeling tools mentioned in Section 4.10 have been considered.

The domain of feature modeling is understood here as a domain of the tools that support feature modeling based software development methods. The feature modeling based methods (mentioned in the thesis), such as generative programming, FODA, FORM, FeatuRSEB, including the one proposed in this thesis, all have in common the central role of feature models from which traceability links to other models are provided. The variability lies in the notations of feature modeling employed by different methods. The systems built in the domain would represent feature modeling CASE tools

suitable for individual methods (possibly groups of methods), e.g. feature modeling CASE tool for generative programming or feature modeling CASE tool for MPD_{FM}.

The traceability links in feature models, in most cases, would lead to external models. MPD_{FM} employs feature modeling for application and solution domain modeling. Transformational analysis appears to be an external activity that should be linked to the feature models it is related to. However, the results of transformational analysis are concept instances represented again as feature models. A concept instance feature diagram should be linked to the feature diagram of the paradigm whose instance it is, and also with the concepts mapped to it. Transformational analysis is then viewed as a construction of yet another feature model.

Special support for code skeleton design, e.g. a tool automating this process, is not considered here. Code skeleton design may be performed by traversing paradigm instances and writing the source code manually. Traceability links from paradigm instances to the source code files may be provided.

A.2 Feature Model

The concepts identified in the domain of feature modeling are:

- feature model
- feature diagram
- node
- feature
- partition
- associated information
- AI item
- AI value
- constraint
- default dependency rule
- link

The purpose of this model is to provide a basis for further development by describing the core concepts of feature modeling. It does not cover all the concepts that might have been identified in the domain of feature modeling, e.g. those related to file management, visual side of feature diagrams, or

consistency checking. Also, some of the features of the concepts listed above have not been fully elaborated (this is indicated in the feature description).

Sections A.2.1–A.2.6 present feature diagrams of these concepts and the information associated with them. Section A.2.12 presents parameterized feature diagrams of plural forms needed for some of the concepts. Conventions introduced in Section B.3 apply to Sections A.2.1–A.2.12, too.

A.2.1 Feature Model

A feature model (Fig. A.1) represents the model of a domain obtained by the application of feature modeling.

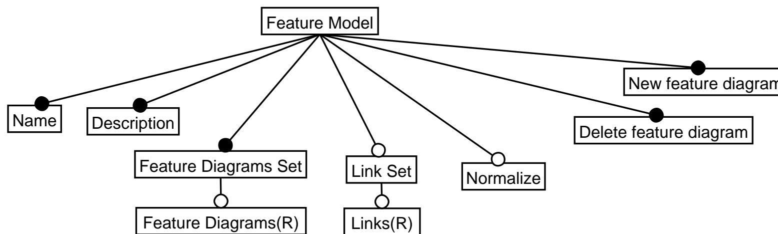


Figure A.1: Feature model.

Feature Model.Name

D: The feature model name.

Rp: The feature model name will serve as a domain name when referring to it or when referencing the concepts defined in it in other feature models.

Feature Model.Description

D: A feature model textual description.

Rp: The feature model description will enable a better orientation in the feature model.

Feature Model.Feature Diagram Set

D: A feature model consists of a set of feature diagrams.

Feature Model.Feature Diagram Set.Feature Diagrams®

D: A set of feature diagrams (Sections A.2.2 and A.2.12).

B: dynamic

Feature Model.Link Set

D: A feature model has a set of links.
Ri: There is a need to link feature models with other modeling artifacts.
B: static

Feature Model.Link Set.Links®

D: A set of links (Sections A.2.11 and A.2.12).
Ri: A feature model is linked with another modeling artifact.
B: dynamic

Feature Model.Normalize

D: Feature diagrams in a feature model may be normalized.

Feature Model.Create feature diagram

D: A feature diagram may be created.

Feature Model.Delete feature diagram

D: A feature diagram may be deleted from a feature model.

A.2.2 Feature Diagram

A feature diagram (Fig. A.2) presents a featural description of a concept graphically.

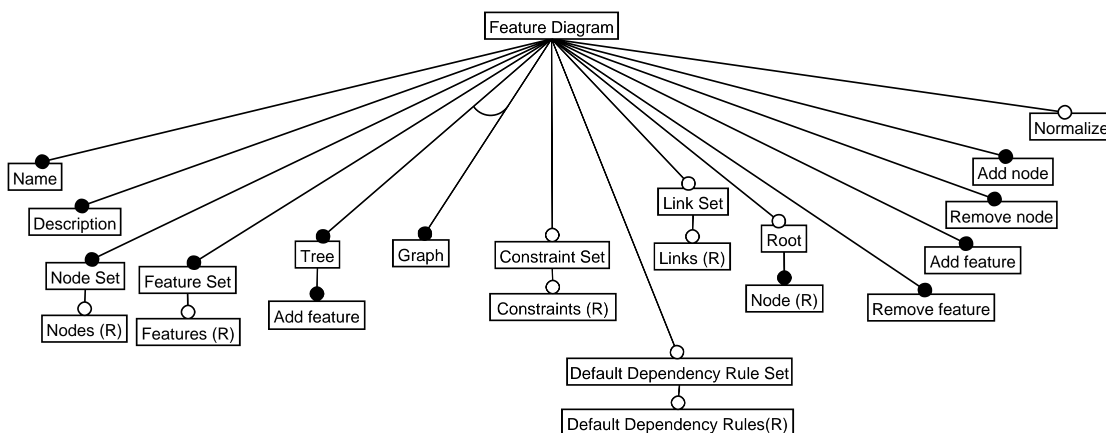


Figure A.2: Feature diagram.

Feature Diagram.Name

D: The name of a feature diagram.

Feature Diagram.Description

D: A feature diagram description.

Feature Diagram.Node Set

D: A feature diagram contains a set of nodes.

Feature Diagram.Node Set.Nodes®

D: The set of nodes a feature diagram consists of (Sections A.2.3 and A.2.12).
B: dynamic

Feature Diagram.Feature Set

D: A feature diagram contains a set of features.

Feature Diagram.Feature Set.Features®

D: The set of features in a feature diagram (Sections A.2.4 and A.2.12).
B: dynamic

Feature Diagram.Tree

D: A feature diagram is represented by a directed tree. It describes the features of a domain concept represented by its root node (*Root.Node*®).
B: static

Feature Diagram.Tree.Add feature

D: An operation of adding a feature to a feature diagram represented as a tree.
Rp: A feature should be added in a way that preserves the tree structure.

Feature Diagram.Graph

D: A feature diagram is considered to be a connected directed graph.
B: static

Feature Diagram.Root

D: The root of a feature diagram.

Feature Diagram.Root.Node®

D: The node (Section A.2.3) which represents a feature diagram root, i.e. the concept that feature diagram models.

Feature Diagram.Constraint Set

D: A feature diagram contains a set of constraints.
Ri: In some approaches to feature modeling, constraints are associated with feature diagrams.
B: static

Feature Diagram.Constraint Set.Constraints®

D: A set of constraints (Sections A.2.9 and A.2.12).
B: dynamic

Feature Diagram.Default Dependency Rule Set

D: A feature diagram contains a set of default dependency rules.
Ri: In some approaches to feature modeling, default dependency rules are associated with feature diagrams.
B: static

Feature Diagram.Default Dependency Rule Set.Default Dependency Rules®

D: A set of default dependency rules (Sections A.2.10 and A.2.12).
B: dynamic

Feature Diagram.Link Set

D: A feature diagram has a set of links.
Ri: There is a need to link feature diagrams with other modeling artifacts.
B: static

Feature Diagram.Link Set.Links®

D: A set of links (Sections A.2.11 and A.2.12).
Ri: A feature diagram is linked with another modeling artifact.
B: dynamic

Feature Diagram.Add node

D: An operation of adding a node to a feature diagram.

Feature Diagram.Remove node

D: An operation of removing a node from a feature diagram.

Feature Diagram.Add feature

D: An operation of adding a feature to a feature diagram.

Feature Diagram.Remove feature

D: An operation of removing a feature from a feature diagram.

Feature Diagram.Normalize

D: A feature diagram may be normalized.
B: static

Constraints:

1. ¬*Root.Node.Reference*
   A root may not be a concept reference.

A.2.3 Node

A feature diagram node. There may be several nodes with the same name in a single feature diagram.

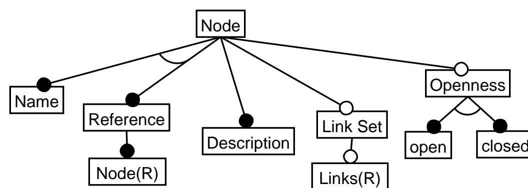


Figure A.3: Node.

Node.Name

D: A string representing the name of a node.
B: static

Node.Reference

D: A node is a reference to another node, i.e. a concept reference.
B: static

Node.Description

D: A textual description of the meaning of a node.

Node.Link Set

D: Feature model has a set of links.
Ri: There is a need to link nodes with other modeling artifacts.
B: static

Node.Link Set.Links®

D: A set of links (Sections A.2.11 and A.2.12).
Ri: A feature model is linked with another modeling artifact.
B: dynamic

Node.Openness

D: An attribute that describes whether new direct variable features of a node are expected.

Node.Openness.open

D: New direct variable features of a node are expected.

Node.Openness.closed

D: No new direct variable features of a node are expected.

A.2.4 Feature

A relationship between two nodes (Fig. A.4). Describes the variability of a subfeature with respect to its superfeature.

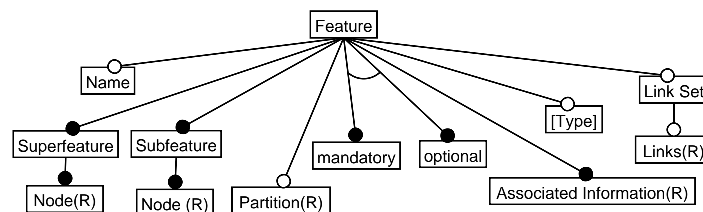


Figure A.4: Feature.

Feature.Name

D: The name of a feature (as a relationship).
Ri: In some approaches to feature modeling, relationships between nodes are named.
B: static

Feature.Superfeature

D: The node whose feature (*Subfeature*) is being defined.

Feature.Superfeature.Node®

D: A node (Section A.2.3).

Feature.Subfeature

D: The node that is being defined as a feature (of *Superfeature*).

Feature.Subfeature.Node®

D: A node (Section A.2.3).

Feature.Subfeature.Reference

D: A feature may be a concept reference.
B: dynamic

Feature.mandatory

D: The subfeature is mandatory, i.e. it must be included in a concept instance.
B: dynamic

Feature.optional

D: The subfeature is optional, i.e. it may be included in a concept instance.
B: dynamic

Feature.Partition

D: A feature may belong to a partition, which has an impact on the subfeature selectability.
B: dynamic

Feature.Associated Information

D: The information associated with the subfeature in the context of this feature.

Feature.Type

D: The type of a feature (e.g., "consists of"). May be selected from a list which should be extensible. (Not elaborated further.)

Rp: Needed in some methods (e.g., FORM).

B: static

Feature.Link Set

D: Feature model contains a set of links.

Ri: There is a need to link features with other modeling artifacts.

B: static

Feature.Link Set.Links®

D: A set of links (Sections A.2.11 and A.2.12).

Ri: A feature is linked with another modeling artifact.

B: dynamic

A.2.5 Partition

Features originating in one node may be divided into a set of disjunct partitions (Fig. A.5).

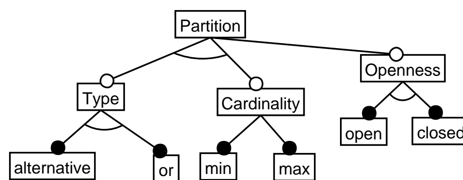


Figure A.5: Partition.

Partition.Type

D: The type of a partition.

Rp: To enable having both alternative and or-partitons.

Ri: Needed by some feature modeling based methods. If not included, all partitions are considered to be alternative as in FODA.

B: static

Partition.Type.alternative

D: The features in a partition are alternative, i.e. one of the features in the partition must be selected during concept instantiation.
B: dynamic

Partition.Type.or

D: The features in a partition are or-features, i.e. a non-empty subset of the features in the partition must be selected during concept instantiation.
B: dynamic

Partition.Cardinality

D: The cardinality of a partition defines the range of the number of features in the partition that may be included in a concept instance. Alternative features can be then expressed by cardinality 1..1, while or-features can be expressed by cardinality 1..∗.
Rp: To enable precise control over the number of selectable features in a partition.
Ri: A generalization of the partition type. If not included, all partitions are considered to be alternative.
B: static

Partition.Openness

D: An attribute that describes whether new direct variable features in a partition are expected.

Partition.Openness.open

D: There will be new direct variable features.
B: dynamic

Partition.Openness.closed

D: There will be no new direct variable features.
B: dynamic

A.2.6 Associated Information

Different approaches to feature modeling, and different applications of it, too, require different information to be associated with features. *Associated Information* concept (Fig. A.6) captures this demand by a fully configurable set of items associated information consists of.

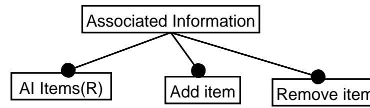


Figure A.6: Associated Information.

Associated Information.AI Items®

D: A set of items associated information consist of (Sections A.2.7 and A.2.12).

Associated Information.Add item

D: An operation of adding an item to the set of associated information items.

Associated Information.Remove item

D: An operation of removing an item from the set of associated information items.

A.2.7 AI Item

AI Item (Fig. A.7) represents a piece of information associated with features.

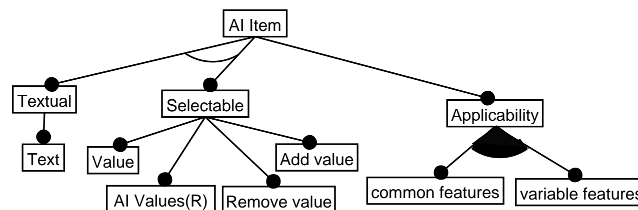


Figure A.7: AI Item.

AI Item.Textual

D: An AI item has a value represented by a free text.

B: static

AI Item.Textual.Text

D: A free text.

AI Item.Selectable

D: An AI item has a value selected from a set of predefined values associated with it.
B: static

AI Item.Selectable.Value

D: The value of an AI item.

AI Item.Selectable.AI Values®

D: The set of predefined values (Sections A.2.8 and A.2.12).

AI Item.Selectable.Add value

D: An operation of adding a value to the set of predefined values.

AI Item.Selectable.Remove value

D: An operation of removing an value from the set of predefined values.

AI Item.Applicability

D: The applicability of an AI item to features with respect to their optionality (to common features, variable ones, or both).

AI Item.Applicability.common features

D: An AI item is applicable to common features.
B: static

AI Item.Applicability.variable features

D: An AI item is applicable to variable features.
B: static

A.2.8 AI Value

AI Value (Fig. A.8) represents a value in a set of predefined values for selectable associated information items.

AI Value.Name

D: The name of an AI value.

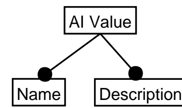


Figure A.8: AI Value.

AI Value.Description

D: The description of an AI value.

A.2.9 Constraint

A constraint on feature selection (Fig. A.9). Constraints express mutual exclusions and requirements among features beside those specified by the feature diagram.

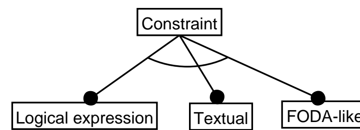


Figure A.9: Constraint.

Constraint.Logical expression

D: A constraint is represented by a logical expression.

B: static

Constraint.Textual

D: A constraint is represented by an informal text.

B: static

Constraint.FODA-like

D: A constraint is represented as in FODA.

B: static

A.2.10 Default Dependency Rule

A default dependency rule (Fig. A.10). Default dependency rules determine which features should appear together by default in concept instances.

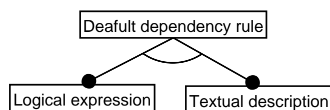


Figure A.10: Default dependency rule.

Default Dependency Rule.Logical expression

D: A default dependency rule is represented by a logical expression.
B: static

Default Dependency Rule.Textual description

D: A default dependency rule is represented by an informal text.
B: static

A.2.11 Link

A link (Fig. A.11) enables to connect a feature model or its parts to files other models are kept in.

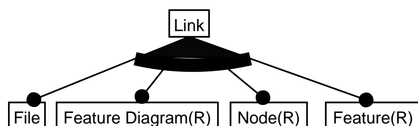


Figure A.11: Link.

Link.File

D: A link to a file.
Ri: Linking to a model other than current feature model.
B: dynamic

Link.Feature Diagram

D: A link to a feature diagram.
B: dynamic

Link.Node

D: A link to a node.
B: dynamic

Link.Feature

D: A link to a feature.
B: dynamic

A.2.12 <Plural Form>

Figure A.12 shows the feature diagram of the parameterized concept of the plural form with the meaning one or more occurrences of the concept in the singular form.[1] The name of *<Plural Form>* (Fig. A.12) is the plural form of the name of *<Singular Form><i>.<Singular Form>*.

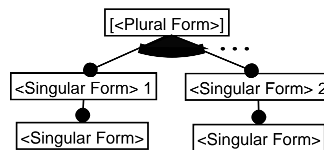


Figure A.12: Plural forms.

<Plural Form>.<Singular Form> <i>

D: One of the concepts the plural form refers to. The parameter <i> is a natural number.
B: dynamic

<Plural Form>.<Singular Form> <i>.<Singular Form>

D: A reference to one of the following concepts: *Feature Diagram*, *Node*, *Feature*, *Link*, *Constraint*, *Default Dependency Rule* or *AI Value*.

[1]This plural form concept is the same as the one in the AspectJ paradigm model (see Section B.4.4 in Appendix B) except for the binding time; the run time binding is needed here.

Appendix B

$\mathrm{MPD_{FM}}$ for AspectJ

The application of feature modeling as a part of the method proposed in this thesis to a solution domain, which is in most cases a programming language, results in a *paradigm model* of the solution domain, i.e. of the programming language. Since the purpose of such a model is to be used in transformational analysis of an application domain represented by a feature model, the solution model defines the domain specific rules of the transformation. Each such a paradigm model can be viewed as a specialization of multi-paradigm design with feature modeling.

This appendix provides a complete paradigm model of AspectJ programming language (AspectJ version 1.1.1). By this, it establishes $\mathrm{MPD_{FM}}$ for AspectJ. Section B.1 identifies AspectJ paradigms. Section B.2 binding times in AspectJ. Section B.3 presents the solution concept and each one of the AspectJ paradigms. Section B.4 presents the auxiliary concepts referred to by paradigms.

B.1 AspectJ Paradigm Identification

AspectJ is an aspect-oriented extension to Java.[1] As such, it provides all the paradigms Java does. In addition, it provides aspect-oriented features. However, the integration of the aspect-oriented features affects the base Java paradigms. Because of that, AspectJ will be analyzed here as a whole, without separating the aspect-oriented extension part from the Java base.

$\mathrm{MPD_{FM}}$ solution domain analysis starts by identifying the paradigms supported by it. AspectJ supports four directly usable paradigms: class, interface, inheritance, and aspect. These four paradigms are at the top-level of the AspectJ paradigm hierarchy, as is depicted in Fig. B.1. Each of them has a corresponding construct at the top level of the AspectJ syntax. This is obvious for the class, interface, and aspect paradigm.

[1] Section 3.2.1 provides some basic information on AspectJ and aspect-oriented programming developed at PARC.

The concept of inheritance is related to all the types in AspectJ, i.e. classes, interfaces, and aspects. The inheritance is usually thought of as a relationship among types (forming so-called inheritance hierarchy), not as being used in a type. Because of that, inheritance is also introduced as a directly usable paradigm, although it is not syntactically separated from the definition of subtypes. Each of the four directly usable paradigms of AspectJ be used in other paradigms, which is also indicated in Fig. B.1, i.e. there are no pure directly usable paradigms in AspectJ.

AspectJ paradigm hierarchy comprises indirectly usable paradigms as well, namely method, inter-type declaration, pointcut, advice, and overloading, which is indicated in Fig. B.1.

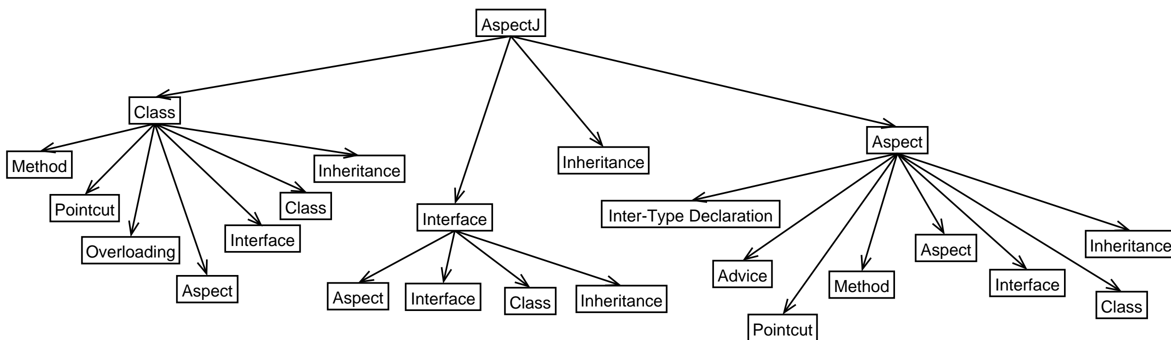


Figure B.1: The hierarchy of AspectJ paradigms. The arrows mean "may use."

B.2 AspectJ Binding Times

After writing the source code of a program in AspectJ, it is compiled into a number of `class` files, one per a class. Subsequently, the program may be executed. Classes are loaded individually as they are needed.

From the above, it follows that AspectJ provides the following binding times:

source time the time of program source code writing, when a programmer explicitly decides what is performed (e.g., that a class will provide some method)

compile time the time of program source code compiling, when decisions are made by an AspectJ compiler (e.g., which method to select among the overloaded ones)

load time the time of program loading, when decisions are made by a loader; programs in Java are loaded per class

run time the time of program running, when decisions are made by the running program

B.3 AspectJ Paradigms

In this section, the first-level AspectJ paradigm model is presented (Section B.3.1).Following that, each AspectJ paradigm is presented in a separate section (Sections B.3.2–B.3.8).

The following conventions apply to Sections B.3.1–B.3.8:

- The name of the section is the name of the concept being presented.
- The text that follows the section title is the concept description.
- Each section contains a figure that shows the feature diagram of the corresponding concept.
- In the qualification of the feature names, the concept name is omitted.
- Information associated with each feature is introduced in a structured way[2]
- At the end of each section, constraints and default dependency rules are introduced (if there are any).

B.3.1 AspectJ Program

The directly usable paradigms represent the subconcepts of the solution concept, an AspectJ program in this case, so their references should appear as features of the solution concept. Figure B.2 presents the feature diagram of an AspectJ program as a solution concept. The directly usable paradigms are modeled as optional features to indicate that they *can* be used in an AspectJ program.

Note that only runnable AspectJ programs are considered here. To be runnable, an AspectJ program must contain at least one class which, in turn, must provide the main method.

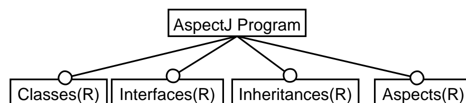


Figure B.2: The concept of an AspectJ program (the directly usable paradigms of AspectJ).

[2] As prescribed in Section 4.4).

AspectJ Program.Classes®

D: The classes in the AspectJ program (Sections B.4.4 and B.3.2).
B: source time

AspectJ Program.Interfaces®

D: The interfaces in the AspectJ program (Sections B.4.4 and B.3.3).
B: source time

AspectJ Program.Aspects®

D: The aspects in the AspectJ program (Sections B.4.4 and B.3.4).
B: source time

AspectJ Program.Inheritances®

D: The inheritances in the AspectJ program (Sections B.4.4 and B.3.5).
B: source time

Constraints:

1. ∀<*i*> ∈ *N Classes.Class* <*i*>*.Class.Access.(package* ∨ *public)*
2. ∀<*i*> ∈ *N Interfaces.Interface* <*i*>*.Interface.Access.(package* ∨ *public)*
3. ∀<*i*> ∈ *N Aspects.Aspect* <*i*>*.Aspect.Access.(package* ∨ *public)*
   The access to classes, interfaces, and aspects at top level of a program may be defined only as package.

B.3.2 Class

The class paradigm is inevitable in any runnable AspectJ program. A class embodies related structure (fields) and behavior (methods). This is indicated in Fig. B.3 by the corresponding features. An empty class is also a valid class, therefore these features are optional.

A class usually has a name, but anonymous classes are possible in methods, and this is indicated by optionality of *Class.Name* feature although this model doesn't go beyond the method level. A number of paradigms can be used directly inside of a class's scope: method, overloading, pointcut, inheritance, interface, aspect (only static aspects are allowed), and the class itself.

The class paradigm is usually used in conjunction with inheritance. However, inheritance is considered as a separate paradigm for the reasons explained in Section B.1. The presence of the feature *Inheritance* in Fig. B.3

indicates that inheritance can be used *inside* of a class by the types nested in it.

The parts of a class (without considering inheritance) are known at source time.

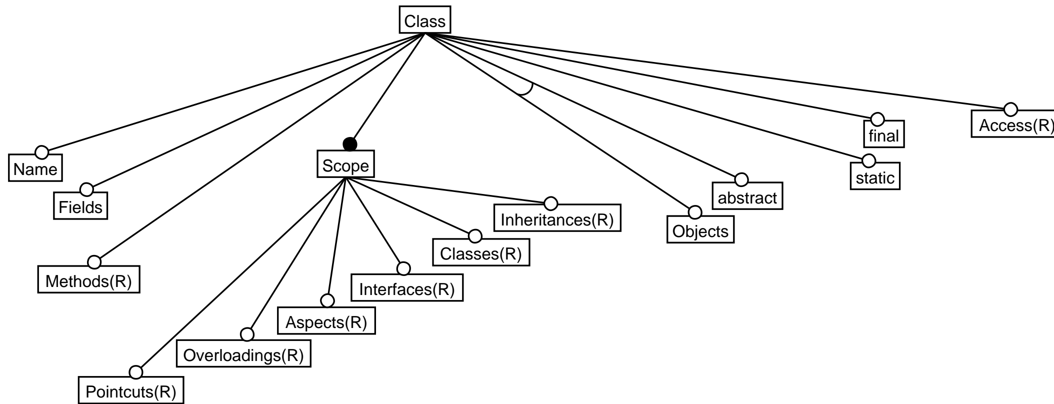


Figure B.3: The class paradigm.

Class.Name

D: Unless it is anonymous (in a method), a class has a name to enable its unique identification.

B: source time

N: Determine the value during transformational analysis.

Class.Fields

D: The data belonging to a class.

B: source time

Class.Methods®

D: The methods of a class (Sections B.4.4 and B.3.6).

B: source time

Class.Scope

D: A class may contain other paradigms in its scope.

Class.Scope.Pointcuts®

D: The pointcuts defined in a class (Sections B.4.4 and B.3.8).

B: source time

Class.Scope.Overloadings

D: The overloadings defined in a class (Sections B.4.4 and B.3.7).
B: source time

Class.Scope.Aspects

D: The aspects defined in a class (Sections B.4.4 and B.3.4).
B: source time

Class.Scope.Interfaces

D: The interfaces defined in a class (Sections B.4.4 and B.3.3).
B: source time

Class.Scope.Classes

D: The other classes defined in a class (Sections B.4.4).
B: source time

Class.Scope.Inheritances

D: The inheritances defined in a class (Sections B.4.4 and B.3.5).
B: source time

Class.Objects

D: A class may have instances.
B: source time

Class.abstract

D: A class may be abstract, i.e. prevented from being instantiated.
Ri: To prevent instantiation.
B: source time

Class.static

D: A class may be static.
B: source time

Class.final

D: A class may be final, i.e. prevented from having subclasses (Section B.3.5).
B: source time

Class.Access®

D: The control of the access to a class (Section B.4.1).

B: source time

Constraints:

1. $\forall$<*i*> $\in$ *N Scope.Aspects* $\Rightarrow$*Scope.Aspects.Aspect* <*i*>*.Aspect.static*
   The aspects in classes may only be static.

2. *abstract* $\Rightarrow$*Name*
   An abstract class must be named.

3. *abstract* $\veebar$ *static*
   A class may not be both abstract and static.

4. *abstract* $\veebar$ *final*
   A class may not be both abstract and final.

B.3.3 Interface

An interface represents a named common behavior in the form of declarations of methods that can be implemented by different classes and aspects. An interface can also contain definitions of constants. The interface paradigm enables employing (inside of it) the same paradigms as the class paradigm.

The methods in an interface are only declarations (constraint 1). Without having methods implementation, the overloading can be only declared.

Inheritance is here considered separately as in case of the class (Section B.3.3). As in the class paradigm (Section B.3.2), the presence of the feature *Scope.Inheritances®* in Fig. B.4 indicates that inheritance can be used *inside* of an interface by the types nested in it.

The parts of an interface (without considering inheritance) are known at source time.

Interface.Name

D: An interface has a name to enable its unique identification.

N: Determine the value during transformational analysis.

Interface.Constants

D: The constant data belonging to the interface.

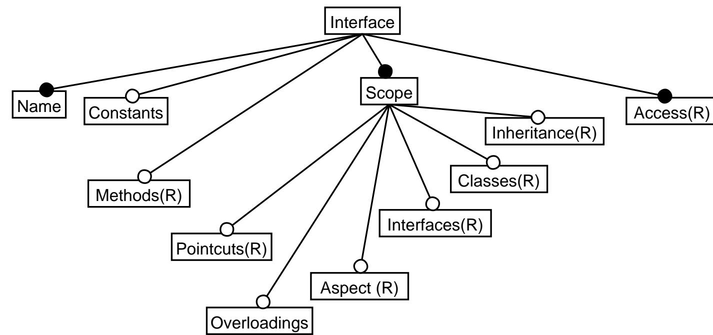


Figure B.4: The interface paradigm.

Interface.Methods®

D: The methods an interface provides (Sections B.4.4 and B.3.6).

Interface.Scope

D: An interface may contain other paradigms in its scope.

Interface.Scope.Pointcuts®

D: The pointcuts defined in an interface (Sections B.4.4 and B.3.8).
B: source time

Interface.Scope.Overloadings®

D: The overloadings defined in an interface (Sections B.4.4 and B.3.3).
B: source time

Interface.Scope.Aspects®

D: The aspects defined in an interface (Sections B.4.4 and B.3.4).
B: source time

Interface.Scope.Interfaces®

D: The other interfaces defined in an interface (Sections B.4.4).
B: source time

Interface.Scope.Classes®

D: The classes defined in an interface (Sections B.4.4 and B.3.2).
B: source time

Interface.Scope.Inheritances®

D: The inheritances defined in an interface (Sections B.4.4 and B.3.5).
B: source time

Interface.Access®

D: The control of the access to an interface (Section B.4.1).
B: source time

Constraints:

1. ∀*<i>* ∈ *N* ¬*Methods.Method <i>.Method.Body*
   In interfaces, methods are only declared.

2. ∀*<i>*, *<j>* ∈ *N* ¬*Scope.Overloadings.Overloading <i>.Overloading. Overloaded methods.Method <j>.Method.Body*)
   The overloading in interfaces may only be declared.

3. ∀*<i>* ∈ *N Scope.Aspects* ⇒*Scope.Aspects.Aspect <i>.Aspect.static*
   The aspects in interfaces may only be static.

B.3.4 Aspect

The aspect paradigm enables to articulate related structure and behavior that crosscuts otherwise possibly unrelated classes, interfaces, and other aspects (only static aspects are allowed) into a named unit.

As can be seen in Fig. B.5, an aspect is similar to a class in the sense that it also embodies related structure (fields) and behavior (methods). But this structure and behavior is used only to support the croscutting, which is achieved by two paradigms an aspect is a container of: advice and inter-type declaration. The pointcut paradigm is used in addition to specify the join points (where the aspect is to be attached).

As classes, aspects can also be instantiated, but the instantiation is automatic. By default, an aspect is a singleton, i.e. there is a single aspect per Java virtual machine. Further, it is possible to declare that an aspect instantiates per each of the specified objects (executing or target ones) at any of the join points specified by a pointcut or per each flow of control (as it is entered or below it) of the join points specified by a pointcut.

Aspects can be priviledged in order to override the access rules of the elements they crosscut. Using inter-type declarations (Section B.3.9), the precedence of the advices (Section B.3.10) can be set. The aspect paradigm enables employing (inside of it) the same paradigms as the class paradigm beside inter-type declarations and pointcuts, which have a special position in it.

The parts of an aspect (without considering inheritance) are known at source time.

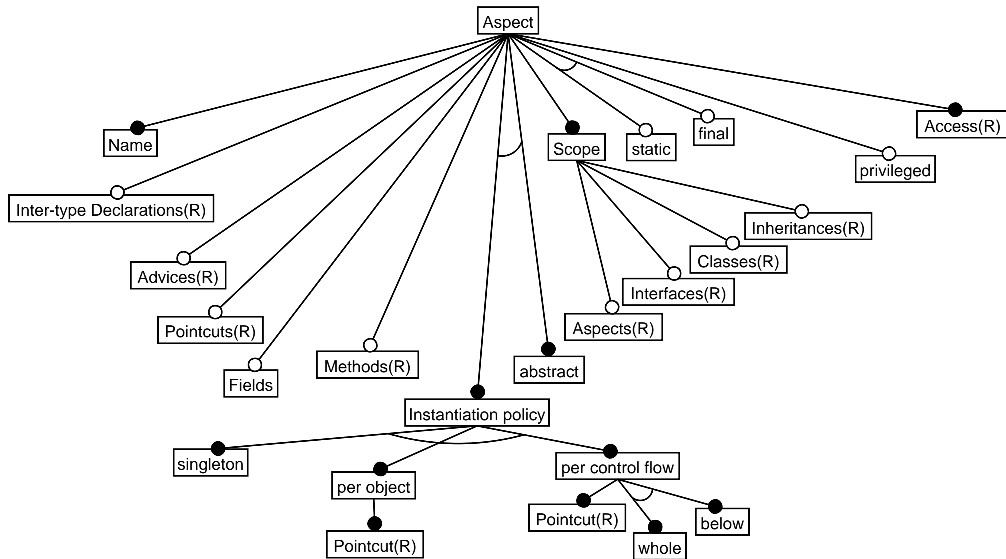


Figure B.5: The aspect paradigm.

Aspect.Name

D: An aspect has a name to enable its unique identification.

N: Determine the value during transformational analysis.

Aspect.Inter-Type Declarations

D: An aspect may contain introductions.

Aspect.Advices

D: An aspect may contain advices.

Aspect.Pointcuts

D: An aspect may contain pointcuts.

Aspect.Fields

D: The data that belongs to an aspect.

B: source time

Aspect.Methods®

D: The methods of an aspect (Sections B.4.4 and B.3.4).
B: source time

Aspect.privileged

D: An aspect may access all the members of all types in the program.
B: source time

Aspect.final

D: An aspect may have no subaspects.
B: source time

Aspect.abstract

D: An aspect cannot have instances.
B: source time

Aspect.Instantiation policy

D: The instantiation policy of an aspect.
B: source time

Aspect.Instantiation policy.single

D: An aspect has a single instance.
B: source time

Aspect.Instantiation policy.per object

D: There is an aspect instance for each object that is the executing object or the target object of the join points picked out by *Aspect.Instantiation policy.per object.Pointcut*.
B: source time

Aspect.Instantiation policy.per object.Pointcut®

D: The pointcut that defines join points (Section B.3.8).

Aspect.Instantiation policy.per control flow

D: There is an aspect instance for each flow of control of the join points picked out by *Aspect.Instantiation policy.per object.Pointcut*.
B: source time

Aspect.Instantiation policy.per control flow.whole

D: The control flow should be taken as a whole.
B: source time

Aspect.Instantiation policy.per control flow.below

D: The control flow should be taken below the entry point (i.e., without it).
B: source time

Aspect.Instantiation policy.per control flow.Pointcut®

D: The pointcut that defines join points (Section B.3.8).

Aspect.Access®

D: The control of the access to an aspect (Section B.4.1).
B: source time

Constraints:

1. *abstract* $\veebar$ *final*
   An aspect may not be both abstract and final.

Default dependency rules:

1. *Instantiation policy* $\Rightarrow$*Instantiation policy.single*
   An aspect has only a single instance by default.

B.3.5 Inheritance

The inheritance paradigm, presented in Fig. B.6, is used to express the commonality in structure and/or behavior that a type shares with its base type. The types that directly or indirectly inherit from one type form an inheritance hierarchy. In general, there may be many *inheritance lines* in an inheritance hierarchy. The inheritance line concept is presented in Section B.4.2.

Inheritance is a relationship between a type and its base type with which it shares the commonality in structure and/or behavior. Inheritance enables keeping the related code in one place to ease programming and maintenance, and polymorphism. Inheritance consists of one or more inheritance lines bound at compile time (when the subtype is built out of the base type and the members it brings itself).

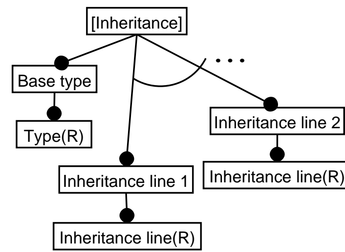


Figure B.6: The inheritance paradigm.

Inheritance.Base type

D: The type to be inherited from.

Inheritance.Inheritance line <i>

D: One inheritance line in the inheritance hierarchy.
B: compile time

Inheritance.Inheritance line <i>.Inheritance line®

D: The inheritance line (Section B.4.2).
N: The feature mapped to *Inheritance line <i>.Subtype.Inheritance line.Base type* must be the same as the one mapped to *Base type*.

Constraints:

1. $\forall$*<i>*$\in$ *N* $\forall$<T>$\in$ {*Class*, *Interface*, *Aspect*} *Base type.Type.<T>*
   $\wedge$ *Inheritance <i>.Inheritance line.Base type.Type.<T>*
   All the inheritance lines in one inheritance hierarchy have a common base type.

B.3.6 Method

The method paradigm is presented in Fig. B.7. Methods are used to express a named, possibly parameterized functionality (*arguments*) over some data and are (usually) placed in a class where that data (fields) belong. This functionality is expressed by the method body. A method may return a value.

A method body forms its own namespace where classes, interfaces, and aspects can be used. However, this paradigm model doesn't go beyond the method level, so this is not shown in Fig. B.7.

In AspectJ, methods may not be used directly. The method paradigm requires an embracing paradigm which may be the class or aspect paradigm.

A method may be the main method of a class, in which case it bears a special name: `main`. Each valid AspectJ program contains at least one such a method.

The parts of a method (without considering inheritance) are known at source time.

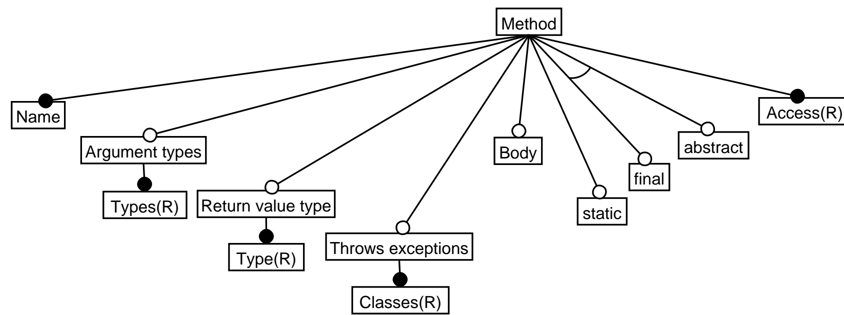


Figure B.7: The method paradigm.

Method.Name

D: A method has a name to enable its unique identification.

N: Determine the value during transformational analysis.

Method.Argument types

D: A method argument types.

B: source time

Method.Argument types.Types®

D: The types (Sections B.4.4 and B.4.3).

Method.Return value

D: A method return value type.

B: source time

Method.Argument types.Return value.Type

D: The type (Section B.4.3).

Method.Throws exceptions

D: The exceptions a method may throw.

B: source time

Method.Throws exceptions.Classes®

D: The classes that represent the exceptions (Sections B.4.4 and B.3.2).

Method.Body

D: The body of a method.

B: run time

Method.static

D: A method may be static.

B: source time

Method.final

D: A method may be final, i.e. prevented from being overridden.

B: source time

Method.abstract

D: A method may be abstract.

B: source time

Method.Access

D: Access control (Section B.4.1).

Constraints:

1. $\mathit{Body} \Rightarrow \neg \mathit{abstract}$
   An abstract method has no body.

2. $\mathit{abstract} \veebar \mathit{static}$
   A method may not be both abstract and static.

Default dependency rules:

1. $\mathit{Access.private} \Rightarrow \mathit{final}$
   A private method is final if not otherwise specified.

B.3.7 Overloading

The overloading paradigm (Fig. B.8) enables to have a set of methods that share the name, but differ in the argument list, according to which are they selected at compile time. The type of the return value is irrelevant. Each method has its own body, i.e. can implement a different algorithm. Therefore it is used to enable automatic algorithm selection for different argument types. Semantics of the methods is usually shared, too.

All the methods, including the inherited ones, may be involved in overloading. The overloading of inherited methods is here understood as an overloading in an inheritance line, which is expressed by the feature *Inheritance line* (Section B.4.2).

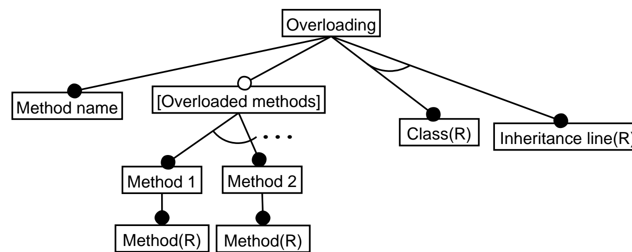


Figure B.8: The overloading paradigm.

Overloading.Method name

D: Method name.

N: Determine the value during transformational analysis.

Overloading.Overloaded methods

D: The overloaded methods. The methods are either from one class (*Class*) or from an inheritance line (*Inheritance line*).

B: source time

Overloading.Overloaded methods.Method <i>

D: An overloaded method.

B: compile time

Overloading.Overloaded methods.Method <i>.Method®

D: The method (Section B.3.6).

N: The value of *Overloaded methods.Method <i>.Method.Name* must be the same as *Method name*. Different *Overloaded methods.Method <i>.Method.Argument types.Types* are required for each *Overloaded methods.Method <i>.Method*.

Overloading.Class®

D: The overloaded methods belong to one class (Section B.3.2).

B: source time

Overloading.Inheritance line®

D: An inheritance line hierarchy whose classes the overloaded methods belong to (Section B.4.2).

B: source time

B.3.8 Pointcut

The pointcut paradigm is (Fig. B.9) enables to specify the join points. Two kinds of join points exist: static and dynamic join points. Both are specified at source time, but are really determined later; static join points, such as method calls or executions, are determined at compile time, while dynamic join points, such as all method calls performed by an object of some type, may be determined only at run time. This means that the *Static join points.Join points* feature has the compile time binding, while *Dynamic join points.Join points* has the run time binding.

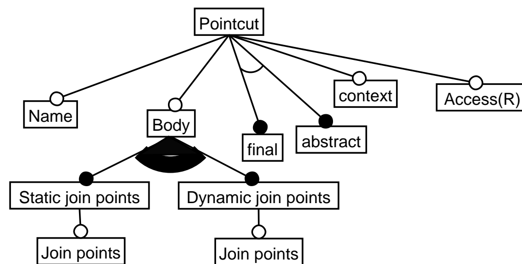


Figure B.9: The pointcut paradigm.

Pointcut.Name

D: The name of a pointcut.

B: source time

Pointcut.final

D: A pointcut may be final, i.e. prevented from being overridden by subclasses or subaspects. Each pointcut that has a body is by final by default.

B: source time

Pointcut.abstract

D: A pointcut may be abstract, i.e. intended to be overridden by subclasses or subaspects. An abstract pointcut has no body.

B: source time

Pointcut.Body

D: Body

B: source time

Pointcut.Body.Static join points

D: The join points that may be determined at compile time, such as method calls or executions.

B: source time

Pointcut.Body.Static join points.Join points

D: The set of static join points.

B: compile time

Pointcut.Body.Dynamic join points

D: The join points that may be determined only at run time, such as all method calls performed by an object of some type.

B: source time

Pointcut.Body.Dynamic join points.Join points

D: The set of dynamic join points.

B: run time

Pointcut.context

D: A pointcut can expose the context, i.e. an object or its fields, caught by some of the primitive pointcuts.

B: source time

Pointcut.Access®

D: The control of the access to a pointcut (Section ®AJ-Aux-Access).

B: source time

Constraints:

1. *abstract* $\veebar$ *Body*
   An abstract pointcut has no body.

2. *Access* ⇒*Name*
   Only a named pointcut may have an access type specified.

B.3.9 Inter-Type Declaration

The inter-type declaration paradigm (Fig. B.10) enables to introduce new fields and methods into types (classes, interfaces, aspects). Further, it enables to add inheritance dependencies, to declare illegal join points (prevented pointcut), and to wrap the specified exceptions at the join points from a given pointcut into an `org.aspectj.lang.SoftException`.

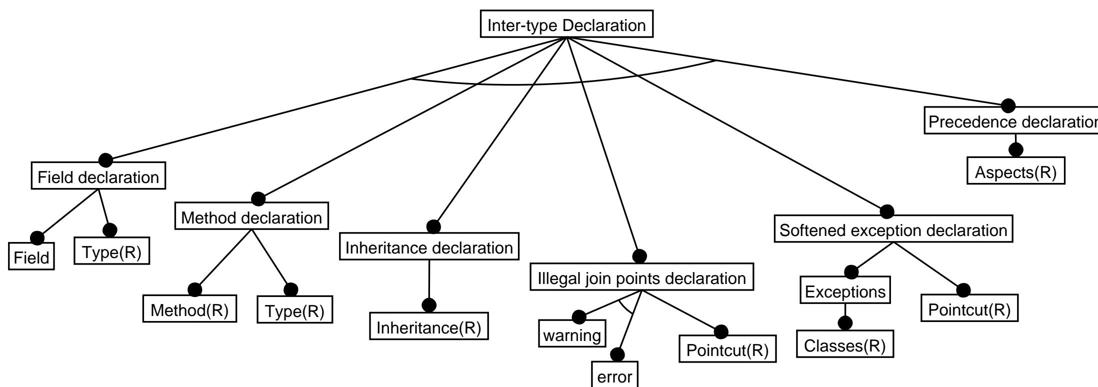


Figure B.10: The inter-type declaration paradigm.

Inter-Type Declaration.Field declaration

D: A field declaration.
B: source time

Inter-Type Declaration.Field declaration.Field

D: A field declared.

Inter-Type Declaration.Field declaration.Types®

D: The types in which *Field declaration.Field* is declared (Sections B.4.4 and B.4.3).

Inter-Type Declaration.Method declaration

D: A method declaration.

B: source time

Inter-Type Declaration.Method declaration.Method®

D: A method declared (Section B.3.6).

Inter-Type Declaration.Method declaration.Types®

D: The types in which *Field declaration.Method* is declared (Sections B.4.4 and B.4.3).

Inter-Type Declaration.Inheritance declaration

D: An inheritance declaration.

B: source time

Inter-Type Declaration.Inheritance declaration.Inheritance®

D: An inheritance declared (Section B.3.5).

Inter-Type Declaration.Illegal join points declaration

D: If any of the join points specified by *Illegal join points declaration.Pointcut* appears in the program, a compile time error or warning should be generated.

B: source time

Inter-Type Declaration.Illegal join points declaration.warning

D: Compile time warning should be generated.

B: source time

Inter-Type Declaration.Illegal join points declaration.error

D: Compile time error should be generated.

B: source time

Inter-Type Declaration.Illegal join points declaration.Pointcut®

D: The pointcut that defines join points (Section B.3.8).

Inter-Type Declaration.Softened exception declaration

D: Wrap *Inter-Type Declaration.Softened exception declaration.Exceptions* at the join points from *Inter-Type Declaration.Softened exception declaration.Pointcut* into a `org.aspectj.lang.SoftException`.

B: source time

Inter-Type Declaration.Softened exception declaration.Exceptions

D: The exceptions to be wrapped.

Inter-Type Declaration.Softened exception declaration.Exceptions. Classess®

D: The classes that represent exceptions (Sections B.4.4 and B.3.2).

Inter-Type Declaration.Softened exception declaration.Pointcut®

D: The pointcut that defines join points (Section B.3.8).

Inter-Type Declaration.Precedence declaration

D: The declaration of the aspect precedence which defines the order their advices are to be run on common join points.

B: source time

Constraints:

1. *Method declaration.Method.Body*
   A declared method must have a body.

2. ¬*Illegal join points declaration.Pointcut.Dynamic join points*
   Only static join points may be specified in an illegal join points declaration.

3. ¬*Softened exception declaration.Pointcut.Dynamic join points*
   Only static join points may be specified in a softened exception declaration.

B.3.10 Advice

Inside of an aspect, the advice paradigm (Fig. B.11) may be used to articulate the actions to be performed in the context of the join points specified by the pointcut. An advice provides a piece of code (in its body) to be run before, after, or in place (around) of a pointcut. The body of an advice is similar to the body of a method.

After advice can run after the execution of each join point specified by the *Pointcut*® completes normally, after it throws an exception, or after it does either one. In the last case, no matching based on the type being returned or exception being thrown can be made.

Around advice returns a value which will replace the original one at each join point specified by the *Pointcut*®. An advice can use a context exposed by its pointcut. The original join point return value may also be captured and returned, modified or not, by letting the original join point execute inside of the advice body. However, this AspectJ paradigm model does not go into such details as they could hardly be used in the transformational analysis.

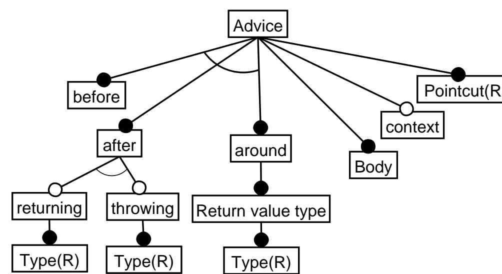


Figure B.11: The advice paradigm.

Advice.before

D: An advice is to be performed before the join points.

B: source time

Advice.after

D: An advice is to be performed after the join points.

B: source time

Advice.after.returning

D: An advice is able to modify the return value.

B: source time

Advice.after.throwing

D: An advice is able to modify the throwing of the exception.

B: source time

Advice.around

D: An advice is to be performed instead the join points.

B: source time

Advice.around.proceed

D: An around advice may allow the execution of the join points it affects to proceed.

B: source time

Advice.Body

D: The body of an advice. The body specifies the actions to be performed.

Advice.context

D: An advice can use a context exposed by its pointcut.

B: source time

Advice.Pointcut®

D: The pointcut that specifies the join points affected by an advice (Section B.3.8).

B.4 Auxiliary Concepts

In this section, auxiliary concepts refereed to by paradigms are introduced. Each such a concept is presented in a separate section (Sections B.4.1–B.4.4). The conventions listed in Section B.3 apply to these sections, too.

B.4.1 Access

The usual access control as known in ordinary Java is applicable to several paradigms of AspectJ.

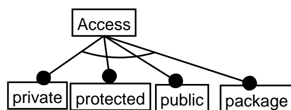


Figure B.12: The concept of access.

Access.private

D: A private access.
B: source time

Access.protected

D: A protected access.
B: source time

Access.public

D: A public access.
B: source time

Access.package

D: A package access.
B: source time

B.4.2 Inheritance Line

The inheritance paradigm is used to express the commonality in structure and/or behavior that the subtype shares with its base type (Section B.3.5). The types that directly or indirectly inherit from one type form an inheritance hierarchy. In general, there may be many *inheritance lines* in an inheritance hierarchy. Figure B.13 shows the feature diagram of the inheritance hierarchy concept.

Each one of the base type and subtype is either a class, interface, or aspect. Any type can inherit from any other type, except a class cannot inherit from an aspect. This is expressed in the constraints of the features.

There are two types of the inheritance: extending and implementing one. The extending inheritance combines inheritance of structure and behavior. It is used between the two elements of equal type and for an aspect to extend a class. The implementing inheritance is the inheritance of behavior.[3] It is used for classes and aspects to declare that they implement some behavior (specified in interfaces).

The extending inheritance allows for method overriding: the methods from the subtype override the methods with the same name and interface from the base type. A method to be executed among the overridden methods is being selected according to the exact object type.

The inheritance hierarchies are bound at compile time, but the base type methods overridden in the subtype are being selected at run time (the exact object type is known only at run time).

[3]With a bit of structure contained in the constants possibly defined in interfaces.

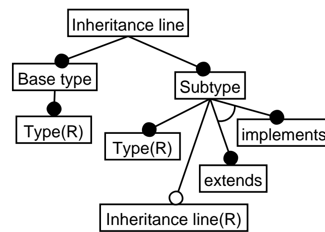


Figure B.13: The concept of inheritance line.

Inheritance Line.Base type

D: The type to be inherited from.

Inheritance Line.Base type.Type®

D: The type (Section B.4.3).

Inheritance Line.Subtype

D: The type that inherits. It is built out of the base type and the members it brings itself at compile time.

Inheritance Line.Subtype.Type®

D: The type (Section B.4.3).

Inheritance Line.Subtype.Inheritance line®

D: A subtype may be a base type to other types building further the subtype hierarchy.

B: source time

N: The feature mapped to *Subtype.Inheritance line.Base type* must be the same as the one mapped to *Subtype*.

Inheritance Line.Subtype.extends

D: The inheritance of structure and behavior.

Rp: The subtype can be an extension of the base type.

B: source time

Inheritance Line.Subtype.implements

D: The inheritance of behavior.

Rp: The subtype can be an implementation of the base type.

B: source time

Constraints:

1. *Base type.Type.Class* $\wedge$ *Subtype.Type.*((*Class* $\vee$ *Aspect*) $\wedge$ *extends*)
   A class may only be extended by another class or an aspect.

2. *Base type.Type.Interface* $\wedge$
   Subtype.Type.(((*Class* $\vee$ *Aspect*) $\wedge$ *implements*) $\vee$ (*Interface* $\wedge$ *extends*))
   An interface may only be implemented by a class or aspect, or extended by another interface.

3. *Base type.Type.Aspect* $\wedge$ *Subtype.Type.*(*Aspect* $\wedge$ *extends*)
   An aspect may only be extended by another aspect.

4. *Base type.Type.*(*Class* $\veebar$ *Class.final*)
   A final class may not be extended.

5. *Base type.Type.*(*Aspect* $\veebar$ *Aspect.final*)
   A final aspect may not be extended.

B.4.3 Type

Classes, interfaces, and aspects are in AspectJ denoted as types, as they encapsulate the common structure that may be used further. The feature diagram in Fig. B.14 describes this concept.

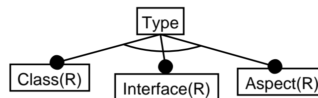


Figure B.14: The concept of a type.

Type.Class

D: The type is a class (Section B.3.2).
B: source time

Type.Interface

D: The type is an interface (Section B.3.3).
B: source time

Type.Aspect

D: The type is an *Aspect* (Section B.3.4).
B: source time

B.4.4 <Plural Form>

In AspectJ paradigm model, often there is a need to refer to concepts in plural. The feature diagram in Fig. B.15 presents these plural forms in a generic form. The concept <*Singular Form*> may be any of the AspectJ paradigms or *Type* concept (introduced in Section B.4.3). The name of <*Plural Form*> is the plural form of the name of <*Singular Form*> <*i*>.<*Singular Form*>.

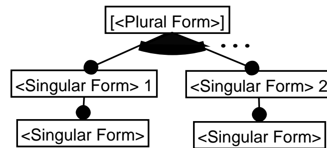


Figure B.15: The plural forms of concepts.

<Plural Form>.<Singular Form> <i>

D: One of the concepts the plural form refers to. The parameter <i> is a natural number.

B: source time

<Plural Form>.<Singular Form> <i>.<Singular Form>

D: A reference to one of the following concepts: *Class*, *Interface*, *Aspect*, *Inheritance*, *Method*, *Overloading*, *Pointcut*, *Inter-Type Declaration*, *Advice*, or *Type*.

Appendix C

Applying MPD_{FM} for AspectJ

This appendix presents an application of multi-paradigm design with feature modeling (MPD_{FM}) for AspectJ (introduced in Appendix B). The application domain under consideration here is the domain of feature modeling, whose feature model has been introduced in Appendix A.

This appendix is divided into two sections. The results of the transformational analysis performed using the AspectJ paradigm model are presented in Section C.1. The code skeleton corresponding to this transformational analysis is introduced in Section C.2.

C.1 Transformational Analysis

In this section the transformational analysis of the feature model of the domain of feature modeling using the AspectJ paradigm model is presented.

Individual paradigm instances are represented by feature diagrams (as proposed in Section 5.3) and their creation is explained.

C.1.1 Feature Model

Feature Model concept (Section A.2.1) incorporates data about a feature model and operations on this data. This roughly corresponds to the class paradigm. The class paradigm instantiated over the feature model concept is presented in Fig. C.1.

Based on their semantics, *Name*, *Description*, and *Feature Diagram Set* would be fields of the class corresponding to *Feature Model*. *Feature Diagrams*Ⓡ, a plural form (see Section C.1.12) of *Feature Diagram* concept, which has been found to correspond to the class paradigm (see Section C.1.2), describes further the structure of the field. This is beyond the

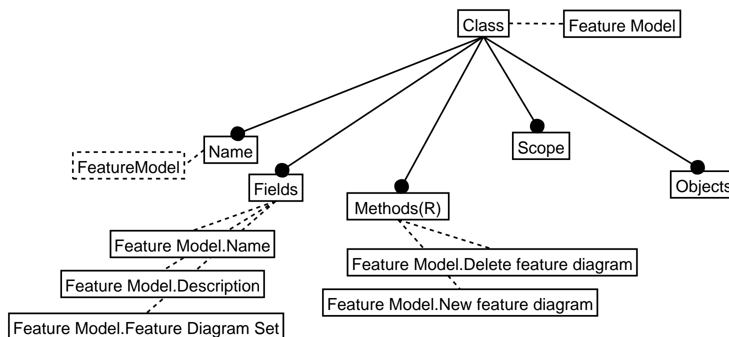


Figure C.1: The class paradigm instantiated on *Feature Model*.

detailedness of the AspectJ paradigm model used, so this feature stays unmapped.

Carrying further with the mapping to the class paradigm, *New feature diagram* and *Delete feature diagram* features would represent methods. Since these two features have no subfeatures, no sufficient details are available to create corresponding method paradigm instances, so their exact form remains unknown.

Feature Model concept may have instances (specific feature models), which has determined the inclusion of *Objects* feature in the paradigm instance.

At first glance, *Normalize* should be another method of the class corresponding to *Feature Model*. The fact it is optional with source time binding implies it could be simply removed from the source code if not needed. However, this issue concerns also *Feature Diagram* concept, since it also has *Normalize* feature (see Section A.2.2). A solution that takes this into account is presented in Section C.1.13. *Link Set* feature requires a similar solution described in Section C.1.14.

C.1.2 Feature Diagram

Transformational analysis of *Feature Diagram* concept (Section A.2.2) is similar to transformational analysis of *Feature Model* concept introduced in the previous section. The class paradigm instantiated over the feature diagram concept is presented in Fig. C.2. This paradigm instance determines the skeleton code for a class corresponding to *Feature Diagram* concept.

A feature diagram is either a tree or a graph, which is indicated by *Tree* and *Graph* statically bound alternative features. There is no corresponding feature in the class paradigm for these features, so we try to regard them as concepts, according to step 1d of the transformational analysis process (see Section 5.3.2). The two corresponding paradigm instances of the class paradigm to these concepts are shown in Fig. C.3 and C.4.

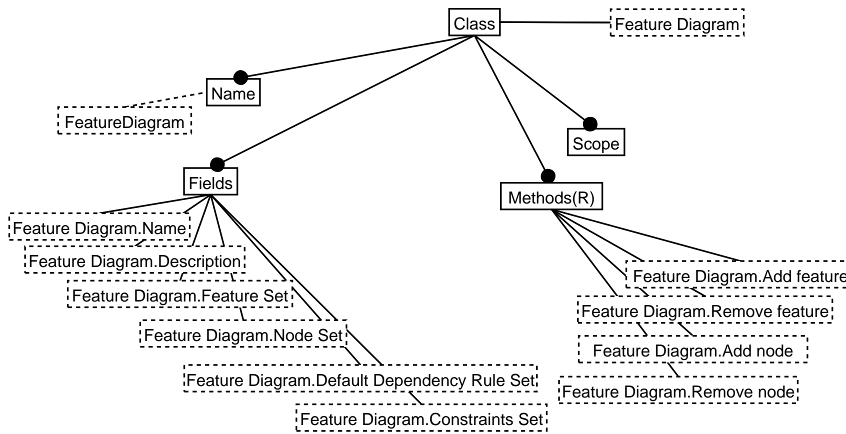


Figure C.2: The class paradigm instantiated on *Feature Diagram*.

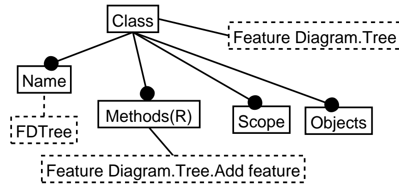


Figure C.3: The class paradigm instantiated on *Feature Diagram.Tree*.

The relationship between these two newly factored out concepts and *Feature Diagram* represents an inheritance. The corresponding inheritance paradigm instance is shown in Fig. C.5. By this, the method corresponding to *Feature Diagram.Add feature* will be overridden in the class corresponding to *Feature Diagram.Tree*.

As already mentioned in the previous section, *Feature Model.Normalize* and *Feature Diagram.Normalize* features are correlated; a solution is provided in Section C.1.13. Section C.1.14 presents a solution for *Link Set* feature.

C.1.3 Node

Node concept (Section A.2.3) represents a feature diagram node with related data. These data correspond to the fields of a class. The class paradigm

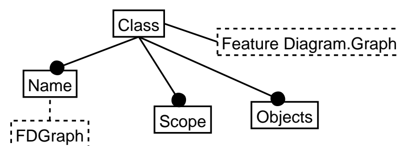


Figure C.4: The class paradigm instantiated on *Feature Diagram.Graph*.

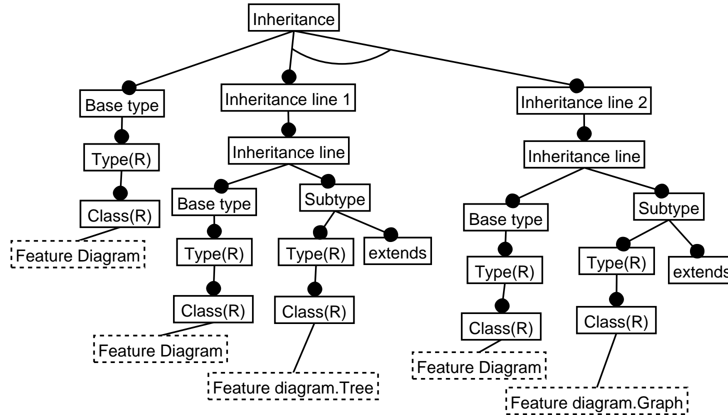


Figure C.5: The inheritance paradigm instantiated on *Feature Diagram*, *Feature Diagram.Tree*, and *Feature Diagram.Graph*.

instance instantiated on *Node* concept is presented in Fig. C.6.

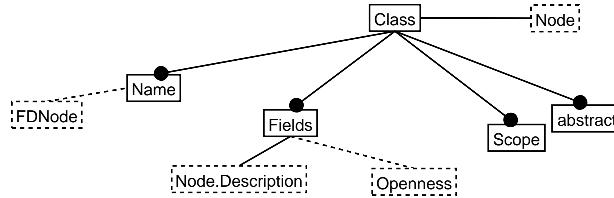


Figure C.6: The class paradigm instantiated on *Node*.

A feature diagram node represents either an independent concept or a feature, which is named, or a concept reference, which bears the name of the referenced concept. This is indicated by alternative features *Name* and *Reference*. Based on this, two types of the nodes may be distinguished. The corresponding class paradigms are presented in Figures C.7 and C.8. *Name* field has been introduced into the class paradigm instance corresponding to *Name* feature.

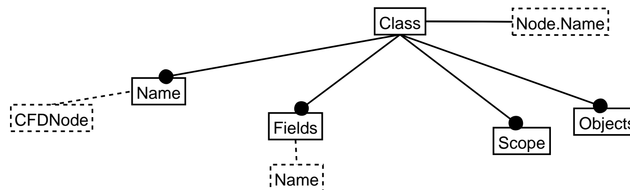


Figure C.7: The class paradigm instantiated on *Node.Name*.

The two types of the feature diagram nodes are related to *Node* concept by inheritance, as presented in Fig. C.9.

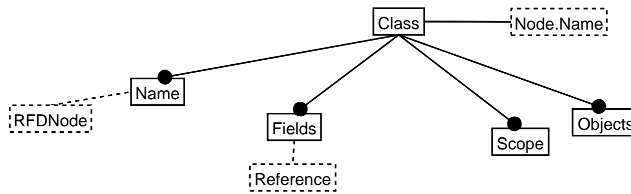


Figure C.8: The class paradigm instantiated on *Node.Reference*.

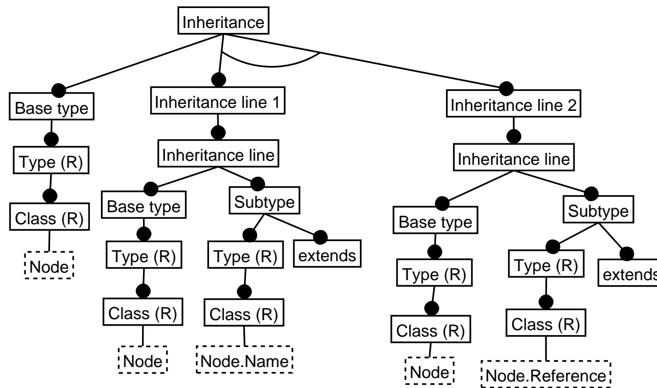


Figure C.9: The inheritance paradigm instantiated on *Node*, *Node.Name*, and *Node.Reference*.

The expectation of further features at some of the feature diagram nodes is being indicated in some approaches to feature modeling. This is modeled by *Openness* feature. This feature would appear as a class field if needed. Otherwise, it wouldn't be present. Another solution would be to employ an aspect as with *Link Set* feature.

Section C.1.14 presents a solution for *Link Set* feature. The subfeatures of *Openness* feature, which has been determined to be a class field, represent its possible values which is beyond the detailedness of the AspectJ paradigm model being used. The same situation is with *Reference.Node*® feature.

C.1.4 Feature

Feature concept (Section A.2.4) represents a relationship between two feature diagram nodes. This relationship bears with it some related data which includes an information about the two nodes in case (*Superfeature* and *Subfeature*). These two and *Associated Information* correspond to class fields. The class paradigm instance corresponding to *Feature* concept is presented in Fig. C.10.

Partition® feature denotes whether a feature belongs to a partition and is supposed to be dynamically changeable as a class field value. However, the AspectJ paradigm model used here doesn't go into such details, so this

feature stays unmapped. Recognizing the meaning of *Partition*® feature, *InPartition* field is introduced into the class paradigm instance as if it has been a feature of *Feature* and *Partition*® its subfeatures.

A similar situation with a similar solution is the one with the group of alternative features *mandatory* and *optional*. *Optionality* field is introduced into the class paradigm instance as if it has been a feature of *Feature* and *mandatory* and *optional* its subfeature.

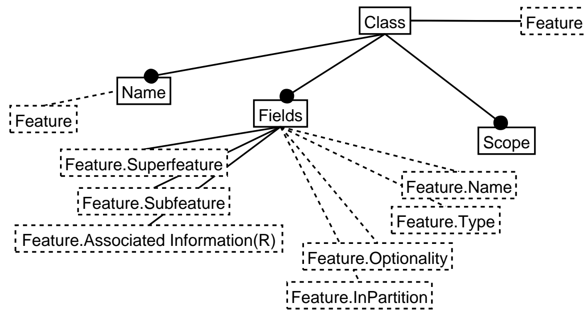


Figure C.10: The class paradigm instantiated on *Feature*.

In some approaches to feature modeling (e.g., FORM), features are being categorized, which is modeled by *Type* feature. Some approaches may require named (labeled) feature relationships, which is modeled by *Name* feature. These two features would appear as class fields if needed (independently of each other). Otherwise, they wouldn't be present. Features *Superfeature.Node*® and *Subfeature.Node*® represent values of the fields, which is beyond the detailedness of the AspectJ paradigm model being used. Section C.1.14 presents a solution for *Link Set* feature.

C.1.5 Partition

The class paradigm instance corresponding to *Partition* concept (Section A.2.5) is presented in Fig. C.11. It represents alternative feature partition and is instantiable (*Class.Objects*) as such.

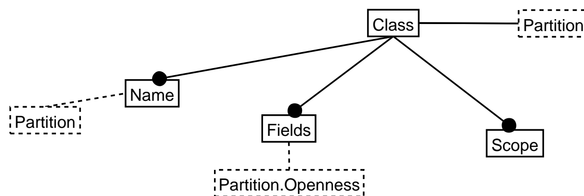


Figure C.11: The class paradigm instantiated on *Partition*.

There are two different notations for partitions in approaches to feature modeling represented by alternative features *Type* and *Cardinality*, which

leads to the use of inheritance. These two features are regarded further as concepts with the corresponding class paradigm instances presented in Figures C.12 and C.13. They are related to the class paradigm corresponding to *Partition* concept through inheritance, as shown in Fig. C.14.

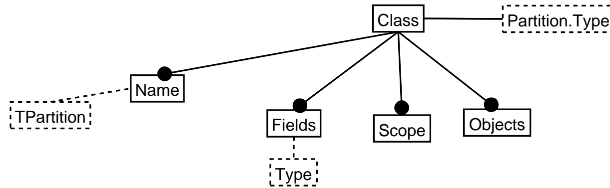


Figure C.12: The class paradigm instantiated on *Partition.Type*.

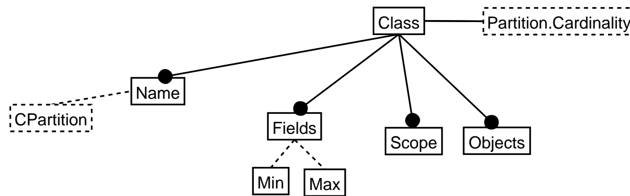


Figure C.13: The class paradigm instantiated on *Partition.Cardinality*.

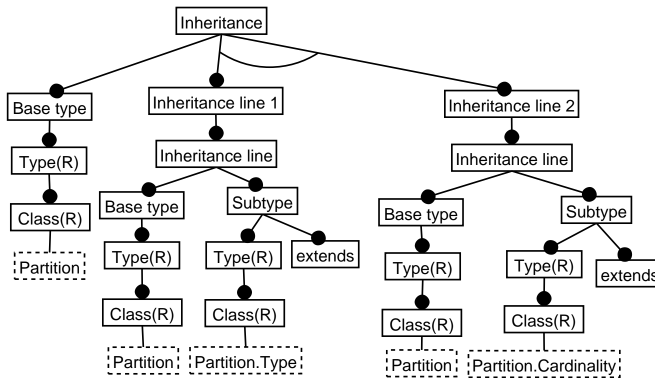


Figure C.14: The inheritance paradigm instantiated on *Partition*, *Partition.Type*, and *Partition.Cardinality*.

C.1.6 Associated Information

The class paradigm instance corresponding to *Associated Information* concept (Section A.2.6) is presented in Fig. C.15. Feature *Associated Information.AI Items*® represents its field, while features *Associated Information.Add item* and *Associated Information.Remove item* represent its methods.

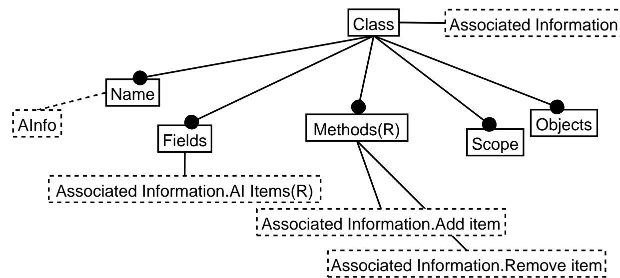


Figure C.15: The class paradigm instantiated on *Associated Information*.

C.1.7 AI Item

The class paradigm instance corresponding to *AI Item* concept (Section A.2.7) is presented in Fig. C.16. Feature *AI Item.Applicability* represents its only field.

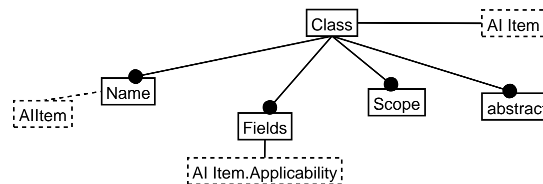


Figure C.16: The class paradigm instantiated on *AI Item*.

There are two different types of associated information items represented by alternative features *AI Item.Text* and *AI Item.Selectable*, which leads to the use of inheritance. These two features are regarded further as concepts with the corresponding class paradigm instances presented in Figures C.17 and C.18. They are related to the class paradigm corresponding to *Associated Information* concept through inheritance, as shown in Fig. C.19.

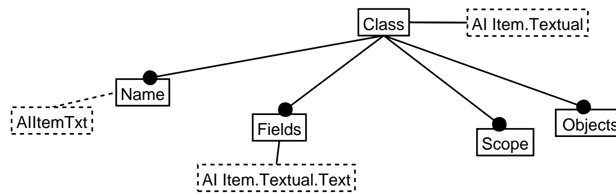


Figure C.17: The class paradigm instantiated on *AI Item.Textual*.

C.1.8 AI Value

The class paradigm instance corresponding to *AI Value*(Section A.2.8) concept is presented in Fig. C.20. Features *AI Value.Name* and *AI Value.Name* represent its fields.

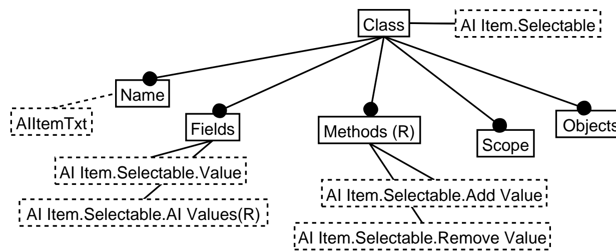


Figure C.18: The class paradigm instantiated on *AI Item.Selectable*.

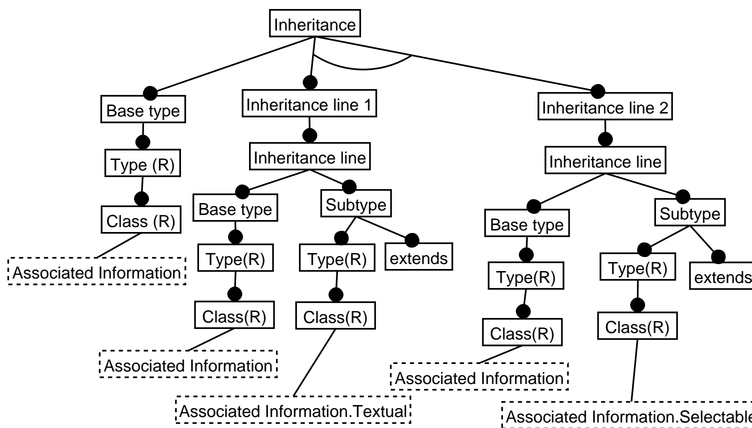


Figure C.19: The inheritance paradigm instantiated on *AI Item*, *AI Item.Textual*, and *AI Item.Selectable*.

C.1.9 Constraint

Constraint (Section A.2.9) can be represented by a class (Fig. C.21). Different approaches to feature modeling represent constraints in a different way. These different constraint representations correspond to subclasses of the class corresponding to *Constraint* concept (Fig. C.22).

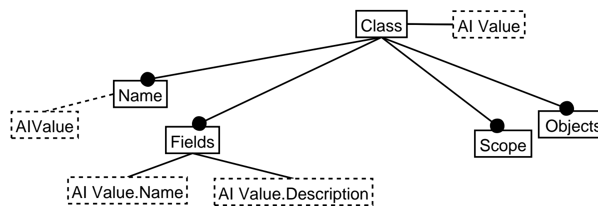


Figure C.20: The class paradigm instantiated on *AI Value*.

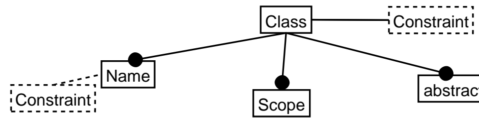


Figure C.21: The class paradigm instantiated on *Constraint*.

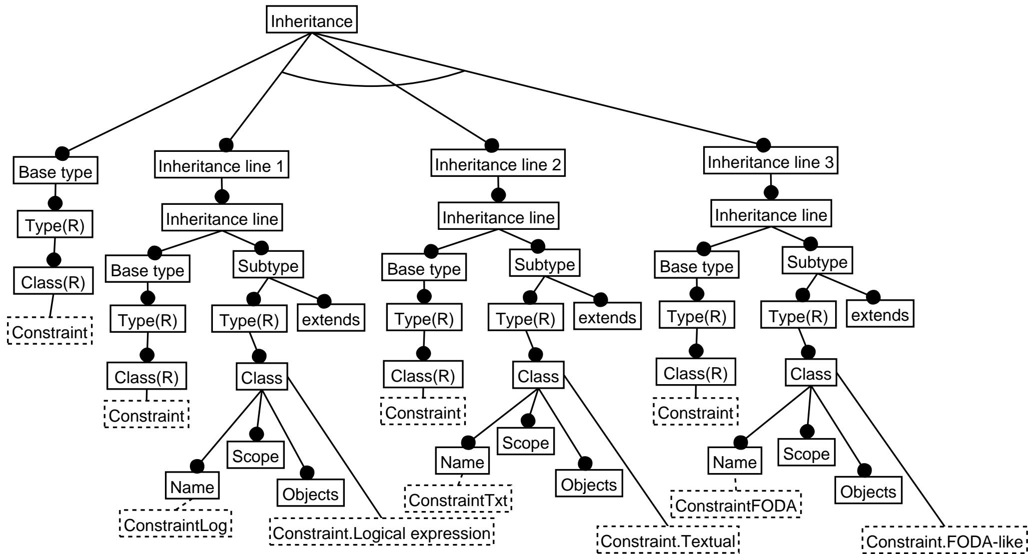


Figure C.22: The inheritance paradigm instantiated on *Constraint* and its subfeatures.

C.1.10 Default Dependency Rule

Default Dependency Rule concept (Section A.2.10) can be represented by a class (Fig. C.23). Different approaches to feature modeling represent constraints in a different way. These different constraint representations are subclasses of the class corresponding to *Constraint* concept (Fig. C.24).

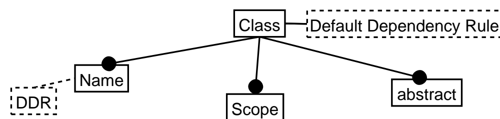


Figure C.23: The class paradigm instantiated on *Default Dependency Rule*.

C.1.11 Link

Link concept (Section A.2.11) can be represented by a class (Fig. C.25). Different types of links are subclasses of the class corresponding to *Link* concept (Fig. C.26).

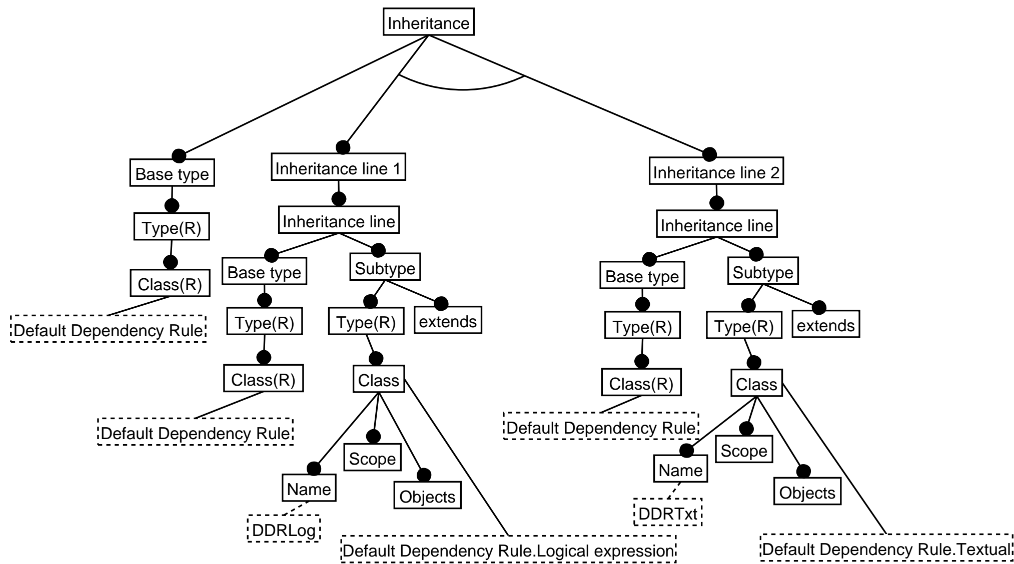


Figure C.24: The inheritance paradigm instantiated on *Default Dependency Rule* and its subfeatures.

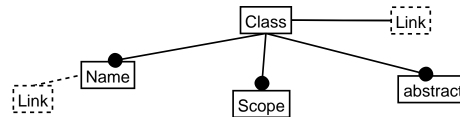


Figure C.25: The class paradigm instantiated on *Link*.

C.1.12 Plural Forms

Plural concept references *Feature Diagrams*®, *Concept*®, *Features*®, *Links*®, *AI Items*®, and *AI Values*® are special cases of the concept of plural introduced in Section A.2.12. This concept gets into the structure of data, which is beyond the detailedness of the AspectJ paradigm model used here. However, based on the specification provided by *Plural* concept, it may be concluded that a class field of a container type similar to *Array* should be used to implement such concept references.

C.1.13 Normalization

Both *Feature Model* and *Feature Diagram* concepts (Sections A.2.1 and A.2.2) have *Normalize* feature. From descriptions of these features we conclude that they match with the method paradigm. Further, we conclude that one without another makes no sense. For this reason, we instantiate an aspect paradigm with two inter-type method declarations as shown in Fig. C.27. *Normalize* feature is optional with statical binding, which is supported by a possibility to plug in or out the aspect at source time.

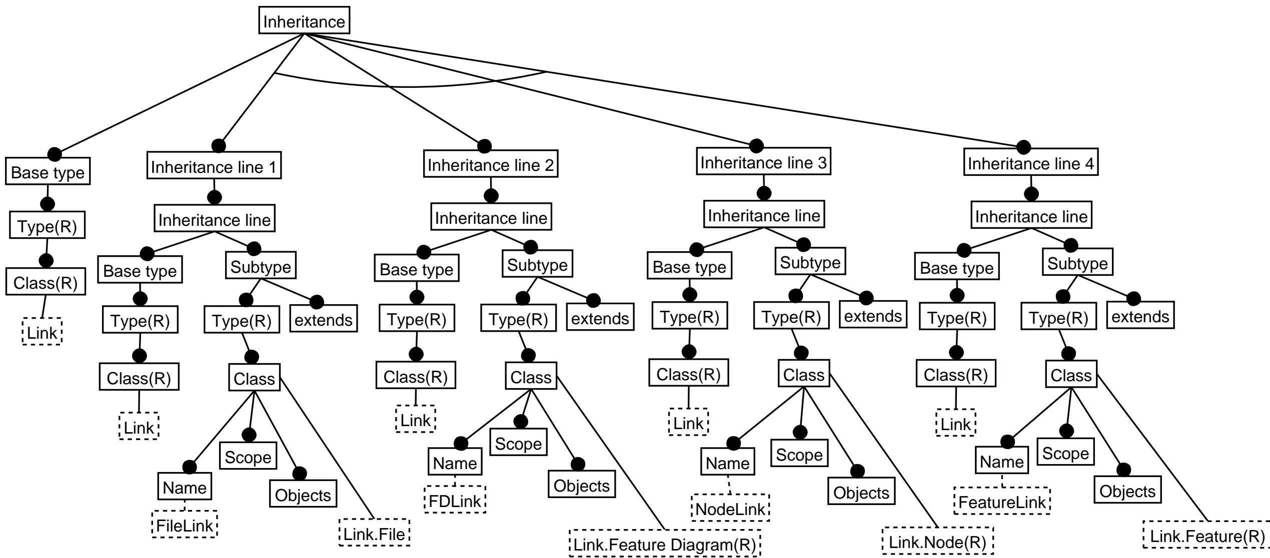


Figure C.26: The inheritance paradigm instantiated on *Link* and its subfeatures.

C.1.14 Linking

Concepts *Feature Model*, *Feature Diagram*, *Node*, and *Feature* (Sections A.2.1–A.2.4) have a common feature *Link Set*. From descriptions of these features we conclude that they correspond to a class field. These features should be included if interlinking of the feature model parts or their linking to other resources is needed. Therefore, we instantiate an aspect paradigm with four inter-type field declarations as shown in Fig. C.28. *Link Set*® feature is optional with statical binding, which is supported by a possibility to plug in or out the aspect at source time.

C.2 Code Skeleton Design

According to the paradigm instances created in transformational analysis of the domain of feature modeling (presented in Section C.1), code skeleton can be designed.

The structural paradigm instances should be transformed into code first (as proposed in Section 5.4). After that, relationship paradigm instances can be transformed into code, as they depend on the structural paradigm instances.

However, in the presented transformational analysis, the relationship paradigm instances are always introduced after the structural paradigm instances they depend on, so paradigm instances can be transformed into code in the order of appearance.

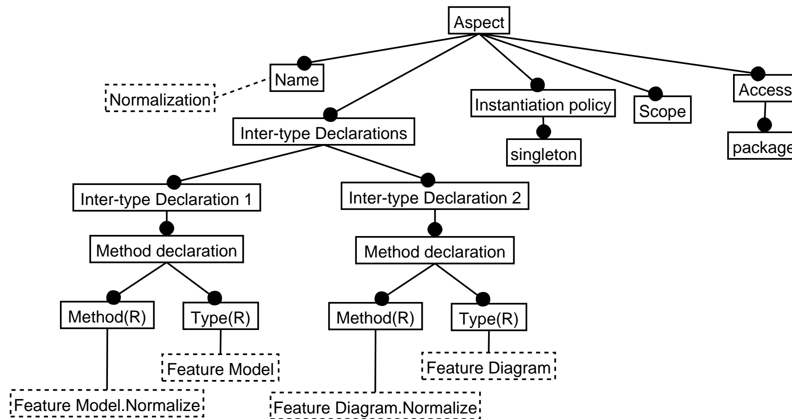


Figure C.27: The normalization aspect.

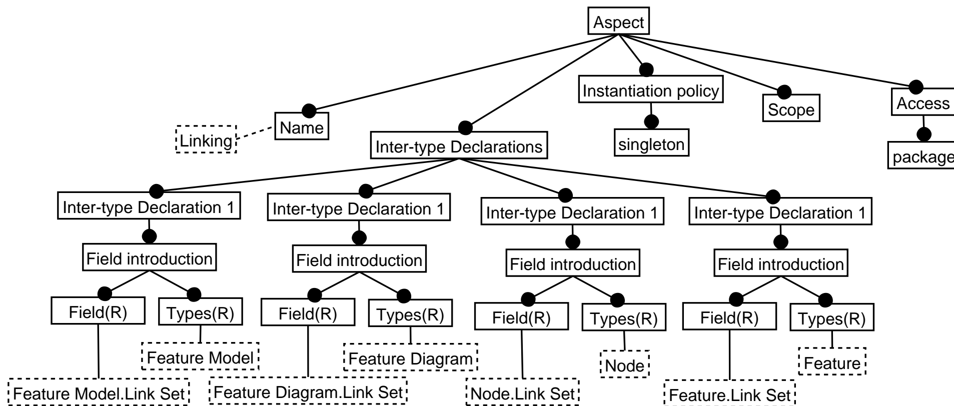


Figure C.28: The linking aspect.

The rest of this section is the code skeleton designed according to the paradigm instances from Section C.1. The statically optional parts are marked as such in the comments. When including such parts, the constraints specified in the application domain feature model have to be taken into account. The missing parts of the code skeleton are indicated by ellipses. Since the AspectJ paradigm model used for transformation doesn't go the beyond the method level, method bodies are left out completely.

```
class FeatureModel {
String Name;
String Description;
Array FeatureDiagrams;

... NewFD(...);
... DeleteFD(...);
}
```

```
abstract class FeatureDiagram {
String Name;
String Description;
Array Nodes;
Array Features;
Array Constraints;
Array DDRs;

... AddNode(...);
... RemoveNode(...);
... AddFeature(...);
... RemoveFeature(...);
}

class FDTree extends FeatureDiagram {
AddFeature();
}

class FDGraph extends FeatureDiagram {
}


abstract class FDNode {
String Description;
Boolean Openness; // optional
}

class CFDNode extends FDNode {
String Name;
}

class RFDNode extends FDNode {
CFDNode Reference;
}


class Feature {
CFDNode Superfeature;
CFDNode Subfeature;
AInfo AI;
Boolean Optionality;
Partition InPartition;
String Name; // optional
Type; // optional
}


class Partition {
```

```
Boolean Openness; // optional
}

class TPartition extends Partition {
Type;
}

class CPartition extends Partition {
int Min;
int Max;
}


class AInfo {
Array AIItems;

... AddItem(...);
... RemoveItem(...);
}

abstract class AIItem {
Applicability;
}

class AIItemTxt extends AIItem {
String Text;
}

class AIItemSel extends AIItem {
AIValue Value;
Array Values;

... AddValue(...);
... RemoveValue(...);
}

class AIValue {
String Name;
String Description;
}


abstract class Constraint {
}

class ConstraintLog extends Constraint {
}

class ConstraintTxt extends Constraint {
```

```
}

class ConstraintFODA extends Constraint {
}


abstract class DDR {
}

class DDRLog extends DDR {
}

class DDRTxt extends DDR {
}


abstract class Link {
}

class FileLink extends Link {
}

class FDLink extends Link {
}

class NodeLink extends Link {
}

class FeatureLink extends Link {
}


aspect Normalization {
... FeatureModel.Normalize(...) {
...};
... FeatureDiagram.Normalize(...) {
...};
}

aspect Linking {
Array FeatureModel.Links;
Array FeatureDiagram.Links;
Array FDNode.Links;
Array Feature.Links;
}
```