UDC 681.5.015

Multi-Paradigm Design with Feature Modeling

Valentino Vranić

Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technology
Slovak University of Technology, Ilkovičova 3, 84216 Bratislava 4, Slovakia
vranic@fiit.stuba.sk, http://www.fiit.stuba.sk/~vranic/

Abstract. In this article, a method for selecting paradigms, viewed as solution domain concepts, appropriate for given application domain concepts is proposed. In this method, denoted as *multi-paradigm design with feature modeling*, both application and solution domain are modeled using feature modeling. The selection of paradigms is performed in the process of feature modeling based transformational analysis as a paradigm instantiation over application domain concepts. The output of transformational analysis is a set of paradigm instances annotated with the information about the corresponding application domain concepts and features. According to these paradigm instances, the code skeleton is being designed. The approach is presented in conjunction with its specialization to AspectJ programming language. Transformational analysis performed according to the AspectJ paradigm model enables an early aspect identification.

1. Introduction

A quarter of a century since the Robert W. Floyd's Turing Award Lecture on paradigms of programming [1], 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. 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 is concerned with the issue of selecting a paradigm appropriate for the problem being solved. This issue is addressed by the method proposed in this article, *multiparadigm design with feature modeling* (MPD_{FM}). MPD_{FM} is based on the *small-scale paradigm* view, in which paradigms are understood as *solution domain concepts*. A solution domain is a domain in which a solution is to be expressed. Although some intermediate design notations may be considered as solution domains, too, the ultimate solution domain is a programming language. In a programming language understood as a solution domain, solution domain concepts correspond to programming language mechanisms.

By sticking to the small-scale paradigm view, MPD_{FM} avoids the problems
connected with the lack of precise definitions of the popular, *largescale* paradigms [2,3]. Small-scale paradigms can be represented as configurations of commonality and variability [3]. For this, MPD_{FM} employs *feature modeling*, which enables to explicitly deal with variability of concepts. Feature modeling is applied also to the *application domain*, the domain being solved. The two feature models, the application and solution domain one, enter *transformational analysis* in which application to solution domain mapping is being established. This mapping is expressed in the form of yet another feature model consisting of the paradigm instances annotated with the information about corresponding application domain concepts and features which determines the *code skeleton*. The whole process is captured in Fig. 1. In a detailed design and implementation that follows MPD_{FM}, methods specific to the large-scale paradigms pointed to by the small-scale paradigms selected in transformational analysis can be employed.

The rest of the article is structured as follows. Section 2 provides the necessary information on feature modeling in MPD_{FM}. Section 3 describes solution domain feature modeling and shows its use to capture aspect-oriented mechanisms of the AspectJ programming language. Section 4 describes transformational analysis based on feature modeling and demonstrates its application using the AspectJ paradigm model. Section 5 describes briefly code skeleton design. Section 6 discusses related approaches. Section 7 concludes the article.

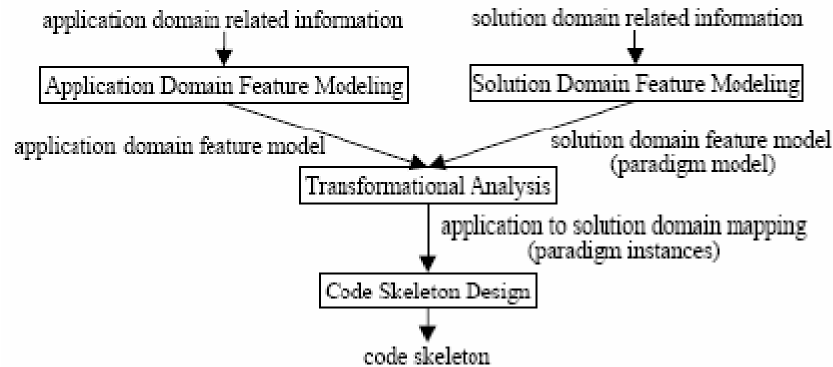

Fig.1. Multi-paradigm design with feature modeling

2. Feature Modeling for Multi-Paradigm Design

Feature modeling is a conceptual domain modeling technique in which concepts in a domaind, understood broadly as an area of interest [4,5], are being expressed by their features taking into account feature interdependencies and variability in order to capture concept configurability.

The origins of feature modeling can be traced back to FODA method [6]. Apart from the mentioned Czarnecki-Eisenecker generative programming, FODA feature modeling has been adopted and adapted by several other domain engineering approaches to software development [7,8,9,10,11,12]. Some work has been devoted primarily to extending feature modeling as such (with respect to UML) [13,14], or even to formalize it [15].

Feature modeling used in MPD_{FM} is based on the Czarnecki-Eisenecker feature modeling employed in generative programming [16,17]. It has been adapted and extended to fit the needs of MPD_{FM} by enabling concept instantiation with respect to instantiation time with concept instances represented by feature diagrams. Further, it brings in parameterization in feature models, enables to represent constraints among features by logical expressions, and introduces concept references to enable to deal with complex feature models (see [18] for details).

This section will provide the necessary information on feature modeling in MPD_{FM} invoking an example of an application domain concept on which further aspects of the method will be demonstrated. An exhaustive

description of the feature modeling for multi-paradigm design may be found in [18,19].

Feature modeling is based on the notions of concept and feature. A *concept* is an understanding of a class or category of elements in a domain. Individual elements that correspond to this understanding are called *concept instances*. A *feature* is an important property of a concept [17]. In general, a feature may be *common*, which means it is present in all concept instances, or *variable*, which means it is present only in some concept instances.

2.1. Feature Diagrams

Feature diagrams are the most important part of a feature model which also may contain information associated with concepts and features and constraints and default dependency rules associated with feature diagrams. An example of a feature diagram is presented in Fig. 2. This figure shows a feature diagram of the text editing buffer concept (adapted from [20], originally inspired by [4]). A text editing buffer represents the state of a file being edited in a text editor. This is modeled by a *mandatory* feature (*File*), which is denoted by a filled circle ended edge. Each text editing buffer employs some memory management scheme to deal with files larger than the working memory (*Memory Management*), which is also modeled by a mandatory feature. Also, each text editing buffer loads and saves its contents into a file, maintains a record of the number of lines and characters, the cursor position, etc., which is modeled by further mandatory features.

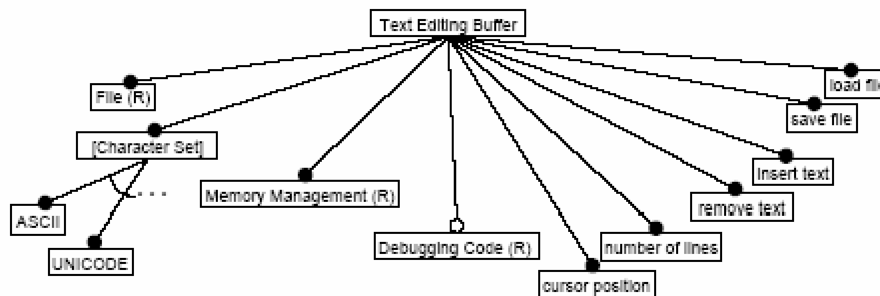

Fig.2. The feature diagram of the *Text Editing Buffer* concept

On the other hand, debugging code might be useful during the development of the text editing buffer, but would probably be undesirable in the final product. Thus, it is modeled by an *optional* feature (*Debugging Code*), which is denoted by an empty circle ended edge.

A text editing buffer will use *exactly one* of the available character sets (*Character Set*).This is specified by *alternative* features (*ASCII*, *UNICODE*...), which are denoted by an empty arc. Note the brackets around the *Character Set* feature's name. This means that it is an open feature; it is expected to have further variable subfeatures. In this case, they would represent other character sets in the group of alternative features, which is indicated by ellipsis placed at this group.

The alternative features just described are actually *mandatory alternative* features. There are also *optional alternative* features of which one or none must be selected. A mixed mandatory-optional alternative feature group is also possible, but its semantics are the same as if all the features were optional alternative.[1]

Feature diagrams may also contain *or-features*, which are denoted by a filled arc (see Fig. 3b). Any non-empty subset or all of the features can be selected from the set of or-features. Having an optional features in a group of or-features would change all its features into simple optional features.

A concept can be referenced as a feature in another or even in its own feature diagram, which is equivalent to the repetition of its feature diagram n the place of the reference. The Ⓡ mark[2] follows the names of concept references in order to distinguish them from the rest of the features. The features *Memory Management* Ⓡ, *File* Ⓡ, and *Debugging Code* Ⓡ in Fig. 2 represent concept references; Fig. 3 shows the feature diagrams of the corresponding concepts.

Note that, with exception of feature references, feature names have no absolute meaning and equally named features may represent different things.

However, no names should be repeated among sibling features, nor among concepts that belong to one feature model.

2.2. Feature Binding

For a variable feature either binding time or binding mode has to be specified. The binding time describes *when* a variable feature is to be bound, i.e. selected to become a mandatory part of a concept instance.

---

[1] This process is being denoted as feature diagram normalization [17].

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

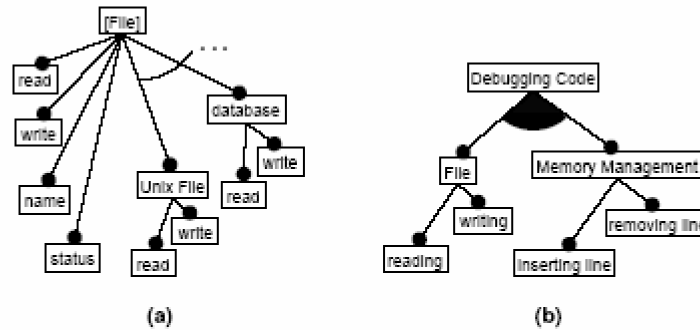

Fig.3. *File* (a) and *Debugging Code* concept (b) feature diagram

It is determined in terms of the binding times available in the solution domain. These usually include: source time, compile time, link time, and run time [4].

At the time of application domain modeling, the solution domain may be unknown or it may be undesirable to pollute the application domain feature model with solution domain details. In that case, using the binding mode instead of the binding time is more appropriate. 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 unbound and rebound, or *dynamically*, in which case its binding is fully controlled at run time. Other, more specific binding modes may be defined as well, e.g. changeable binding as an optimized dynamic binding [17].

Consider again the *Text Editing Buffer* concept (presented in Fig. 2); all its variable features are statically bound. The alternative file type features of the *File* concept in Fig. 3a are bound dynamically because we need to be able to change the output file type at run time. On the other hand, it is sufficient to determine the presence of the debugging code parts at source or compile time, so the corresponding or-features in Fig. 3b are bound statically.

2.3. Constraints Associated with Feature Diagrams

Feature diagrams define the main constraints on 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. Remaining constraints are introduced in a list of constraints associated with the feature diagram. Also, a list of default dependency rules may be associated with each feature diagram in order to specify which features should or should not appear together by default (details available in [18,19]).

To avoid ambiguities, constraints are specified by predicate logic expressions. In such an expression, a feature name *f* stands for *is in instance(f)*, a predicate which is true if *f* is embraced in the concept instance, and false otherwise. Feature names should be qualified to avoid name clashes, but since each expression is associated with a specific feature diagram, the domain and concept name are unnecessary. Some examples of constraints associated with feature diagrams will be introduced in Sect. 3.2.

2.4. Concept Instantiation

A general definition of a concept instance with respect to instantiation time is given here. An instance *I* of the concept *C* at time *t* is a *C*s specialization achieved by configuring its 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 associatedwith 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.

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. The bottom-up instantiation starts at leaves and proceeds towards the root; a feature may be considered for inclusion only if the set of its features selected for inclusion is correct according to the feature variability defined by the feature model.

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 as a concept and as such may be a subject of further instantiation.

During instantiation, concept references are treated as regular features. As such, they may appear in concept instances if they are not replaced by the diagrams of concepts they reference prior to instantiation.

In case of an open feature whose form of expected variable subfeatures is specified, the instance may contain any number of the subfeatures of the specified form. If this description is missing (as with the *Character Set* feature in Fig. 2), during instantiation, an open feature is considered as any other non-open feature.

3. Solution Domain Feature Modeling

This section describes how to apply feature modeling to a solution domain understood as a programming language in order to obtain its *paradigm model*, which is necessary for performing transformational analysis. Recall that the term *paradigm* in MPD_{FM} denotes a solution domain concept, which, in turn, corresponds to a programming language mechanism.

Solution domain feature modeling starts with paradigm identification. The paradigms that can be used directly at the topmost level of programs, i.e. *directly usable* paradigms, are identified first, e.g. the class paradigm in AspectJ programming language [21]. All other paradigms are *indirectly usable* paradigms. In AspectJ, an example would be the method paradigm, 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: its use in transformational analysis. It is not feasible to model all

[3] The AspectJ paradigm model is valid for the AspectJ language definition version 1.1.1 (which remains unchanged in the version 1.2 [21]).

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 the 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.

After identifying directly usable paradigms, binding times (see Sect. 2.2) of the solution domain should be identified. Following that, the first-level paradigm model may be created (Sect. 3.1) and the paradigms may finally be modeled (Sect. 3.2).

3.1. First-Level Paradigm Model

The directly usable paradigm references should appear as features of the solution concept. If a paradigm may appear more than once in a program, its reference should be introduced in the solution domain feature diagram in plural, otherwise in singular.[4] The variability of the paradigm references should be determined according to the restrictions posed by the programming language. If the paradigm reference is a variable feature, its binding time (usually source time) should be determined, too. Finally, initial constraints among paradigms may be determined.

As example, consider the feature diagram of the first-level AspectJ paradigms in Fig. 4. All the directly usable paradigms of AspectJ are modeled
as source time bound optional features of an AspectJ program as a solution concept. Modeling of these directly usable AspectJ paradigms leads to indirectly usable paradigms (which would appear as their features), namely method, overloading, pointcut, inter-type declaration, and advice.

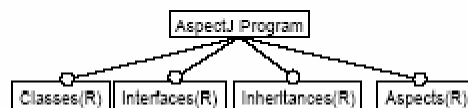

Fig.4. First-level AspectJ paradigms

[4] Plural forms should be defined with respect to singular forms (see [18,19] for details).

3.2. Modeling Paradigms

Each paradigm is considered to be a concept and thus it is presented in a separate feature diagram created according to the solution domain related information. Paradigms that may be used in the paradigm being modeled should be referenced by it. If a paradigm enables instantiation, it should be modeled as a feature (or features). If the feature is variable, its binding time has to be selected among the binding times identified in the solution domain. If none is appropriate, a new binding time should be established.

After creating an initial feature model of a paradigm, feature combinations and interactions should be analyzed to determine constraints and, possibly, identify new features (as proposed in [17] for feature modeling in general).

If some feature's subtree is repeated, it should be factored out as a concept into a separate feature diagram and referenced as needed. In a solution domain feature model, this concept may be a paradigm. If it doesn't appear to be a paradigm, it may be considered as an auxiliary concept.

Much of the paradigms correspond to the main constructs, i.e. structures, of the programming language (e.g., the class in AspectJ). In transformational analysis, there *may* be an application domain concept node that matches with the root of such a *structural paradigm*. Thus, it is possible that no application domain node will match with the root of a structural paradigm. This is especially inherent to the aspect paradigm in AspectJ, which will be introduced in Sect. 3.2.[5]

Besides structural paradigms, there are also paradigms that are about the relationship between some language structures. AspectJ examples include inheritance (a relationship between classes), overloading (a relationship between methods), and advice (a relationship between the advice code, i.e. its body, and the join points it affects). In transformational analysis, no application domain node will match with the root of such a *relationship paradigm*.

Three related paradigms from the AspectJ paradigm model—the aspect, advice, and pointcut paradigm—will be presented here to illustrate the process of paradigm modeling.

[5] Examples of aspect paradigm instances without application domain nodes matching their roots may be found in [18]

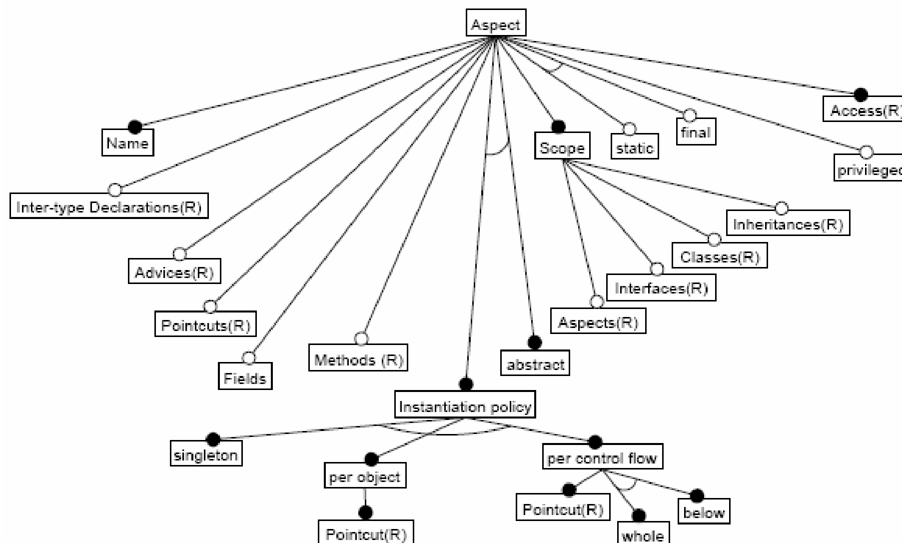

Fig.5. The aspect paradigm in AspectJ

Aspect. The aspect paradigm (see Fig. 5) 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 crosscutting, 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 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. Furthermore, 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 privileged in order to override the access rules of the elements they crosscut. 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, which means that all the variable features presented in Fig. 5 have source time binding.

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

final _abstract

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

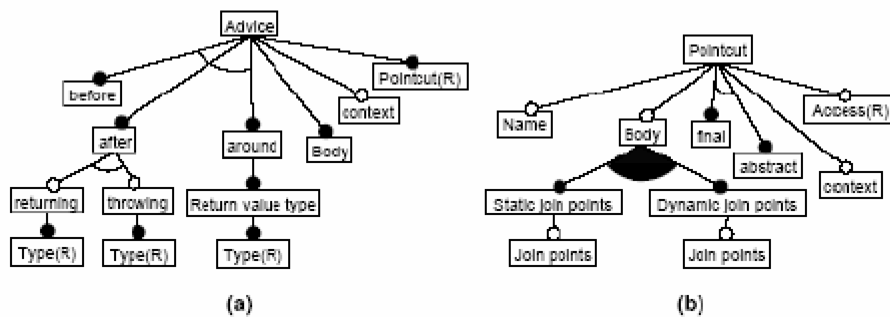

Fig.6. The advice (a) and pointcut (b) paradigm in AspectJ

Advice. Inside of an aspect, the advice paradigm (see Fig. 6a) 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. It can use the join points context exposed by its pointcut.

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 being returned or exception being thrown can be made.

An *around* advice returns a value which will replace the original one at each join point specified by the *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.

Pointcut. The pointcut paradigm (see Fig. 6b) 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 compile time binding, while *Dynamic join points.Join points* has run time binding.

A pointcut is a logical expression formed out of 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 following two constraints are associated with the pointcut paradigm feature diagram:

abstract _Body

Name,Access

which mean that an abstract pointcut cannot have a body (or vice versa), and that an access type can and must be specified in case a pointcut is named, respectively.

Fig.7. The type (a) and access (b) concept

The two auxiliary concepts referenced in the paradigms mentioned above are presented in Fig. 7. The variable features in Figures 5–7 whose binding time has not been explicitly introduced have source time binding.

4. Transformational Analysis

Transformational analysis in MPD_{FM} is a process of finding the correspondence and establishing the mapping between the application and solution domain concepts. It is performed as a *paradigm instantiation 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 the information about corresponding application domain 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.

4.1. Paradigm Instantiation Over Application Domain Concepts

In a paradigm instantiation over application domain concepts, a paradigm, i.e. a solution domain concept, is being instantiated in a bottom-up fashion (see Sect. 2.4) with inclusion of some of the paradigm nodes being stipulated by the mapping of the nodes of one or more application domain concepts to them in order to ensure the paradigm instances correspond to these application domain concepts.

Not all nodes of application domain concepts need to be mapped. An inner[6] application domain concept node 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 solution domain.[7] 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). Further, semantics 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 is usually the source time, as discussed in Sect. 2.2). The binding

[6] An inner node is a non-root and non-leaf node.
[7] However, there may be other mappings in which such a feature would be mapped.

tme 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 would "only" afect 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, constraints on their variability must correspond, too. To a certain extent, during the instantiation of a paradigm, its constraints may accommodate to the constraints of an application domain concept (as far as they obey the rules defined in step 3 of concept instantiation introduced in Sect. 2.4).

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).

4.2. The Process of Transformational Analysis

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.

The given order of steps of transformational analysis process need not be followed strictly; the main purpose of introducing it is to precisely define the output of transformational analysis. For example, one may choose to instantiate a relationship paradigm on an application domain concept prior to actually determining its structural paradigm.

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

4.3. A Transformational Analysis Example

Consider again the text editing buffers debugging code concept whose feature diagram is shown in Fig. 3c. 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 this example, transformational analysis of the text editing buffer's file debugging code part will performed. For this purpose, the feature corresponding to it, *Debugging Code.File*, will be factored out as a concept.

As may be seen from Fig. 3c, 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 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. 8, 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 we need it 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. 8 presents actually five paradigm instances: two pointcuts, two advices, and one aspect. Since paradigm instances are concept instances (see Sect. 2.4), 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.

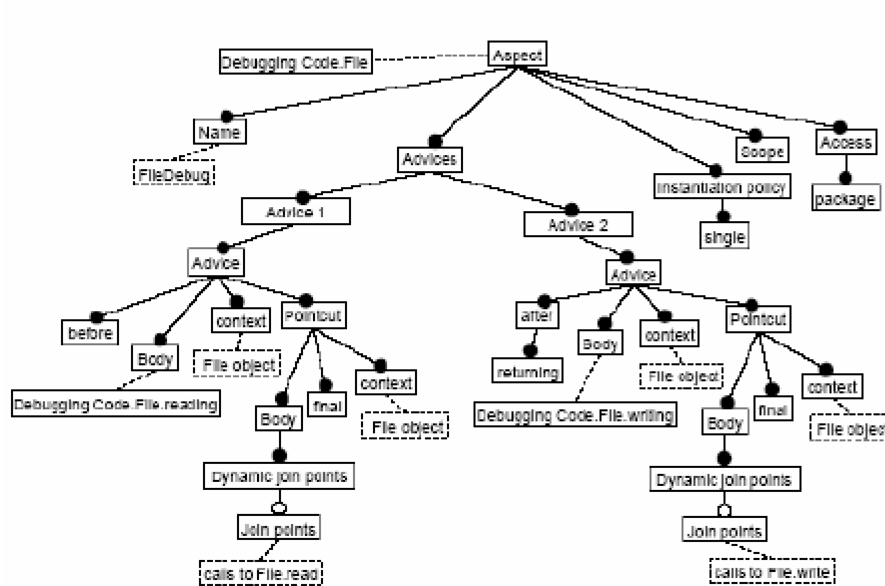

Fig.8. The file debugging code concept transformational analysis; an aspect with two advices

5. Code Skeleton Design

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

In code skeleton design, first the instances of structural paradigms are transformed into code. Subsequently, the instances of relationship paradigms are transformed, too.

The first step produces the basis for the second one because relationship paradigms are usually not represented by independent syntactical structures, but rather attached to the syntactical structures representing structural paradigms.

Following the transformational analysis of the file debugging code concept presented as a paradigm instance in Fig. 8, we could write the following code:

```
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.

6. Related Approaches

Conceptually, MPDFM is closest to *multi-paradigm design* (MPD) [4]. By employing feature modeling, MPD_{FM} introduces several improvements. One of the most important improvements is overcoming the MPD's problem of having to decide the conceptual correspondence between the paradigm and application domain concept at once.[8] By performing transformational analysis as a bottom-up paradigm instantiation over application domain concepts, the correspondence is decided part by part, at lower level features, which are more easily compared.

Feature modeling in MPD_{FM} also enables to visualize hierarchical relationships between the commonalities and variabilities in both application and solution domain models. In MPD, variability dependency graphs are used for this, but they are not capable of expressing variability constraints as feature diagrams are. Moreover, they are used only in application domain models, while representing hierarchical relationships between solution domain concepts, i.e. paradigms, is also needed.

While binding time in MPD is an attribute of a concept as a whole, in MPD_{FM} binding time is specified precisely where it applies: at individual variable features. Also, instantiation in MPD is just an attribute of a concept, while in MPD_{FM} it may be modeled in more details by features.

[8] In fact, MPD uses different terminology than MD_{FM}, e.g. a *domain* in MPD denotes a *concept* P in MPD_{FM}. See [20] for a detailed comparison.

Feature modeling enables to have a visual control over transformational analysis in MPD_{FM}. Its output, annotated paradigm instances, provide enough information about the mapping between the application and solution domain concepts to obtain the main part of the code skeleton from their trees, while in MPD, transformational analysis results are only a guide in choosing a paradigm for an application domain concept.

Negative variability, which is in MPD presented in separate tables (negative variability tables), is in feature modeling modeled by features. The negative variability features of paradigms are actually their specializations (e.g., consider the template specialization [4]).

A design method proposed in connection with *multi-paradigm programming in Leda* [22] is also related to MPD_{FM}. However, while MPD_{FM} is domain-oriented, Leda design method is concerned with the design of one system.

The substantial difference is that MPD_{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. The granularity of large-scale paradigms corresponds to the top level of a system or subsystem. However, the selection of the main paradigm for the system or a part of it is a hard decision to make at once. In Leda design method, a paradigm is selected based on the analysis of the application of each available paradigm impact to lower levels of the system.

Application domain feature modeling is a common activity of both *generative programming* [17] and MPD_{FM}, so it may be performed without having to decide which one of these approaches will be employed. Taking a closer look at generative programming reveals that it also aims at employing multiple paradigms. The difference is in the selection of paradigms: while in MPD_{FM} it is performed directly as a matter of the primary concern, in generative programming it can be viewed as being built into the generator.

7. Conclusions and Further Work

A new method of multi-paradigm software development called *multi-paradigm design with feature modeling* (MPDFM) has been proposed in this article. In this method, feature modeling is used to model both application and solution domain. For this purpose, Czarnecki-Eisenecker feature modeling [17] has been extended and adapted.

Consequently, 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. Subsequently, code skeleton, the final output of MPD_{FM}, is obtained by traversing the trees of annotated paradigm instances, which represent the output of transformational analysis, and writing the source code manually.

To obtain the whole code skeleton, transformational analysis should be performed for each application domain concept, as explained in Sect. 4.2. It is also possible to perform transformational analysis only of some application domain concepts (e.g., the critical ones) and do the rest of the design without MPD_{FM}. The rest of the design would be restricted by such partial transformational analysis results.

Creating a feature model of a solution domain can be viewed as a specialization of MPD_{FM} with respect to transformational analysis. Parts of such a specialization of MPD_{FM} to AspectJ regarding its aspect-oriented paradigms have been presented and applied in this article; its whole paradigm model is available in [18]. The AspectJ paradigm model has been successfully applied in transformational analysis of a feature model of the domain of feature modeling itself [18] (the feature model of feature modeling is available also in [19]).

From the viewpoint of aspect-oriented software development, transformational analysis according to the AspectJ paradigm model enables an early aspect identification. Of course, such aspects are valid in the context of AspectJ only, but this is also the case with language-specific design notations such as [23], which have to be used due to large differences in aspect-oriented mechanisms provided by individual aspect-oriented languages. An important difference is that an application domain model expressed in such a notation is heavily language-dependent, which is not the case with an application domain model in MPD_{FM}.

In MPD_{FM}, both application and solution domain feature models are reused as a whole: different application domains may be implemented in the same solution domain, and an application domain may be implemented in several solution domains. However, some domains overlap, and this happens even if one of them is an application domain and the other one is a solution 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 specialization of MPD_{FM} to design patterns or other intermediate solution domains and combinations of these in conjunction with programming languages as such.

Acknowledgements. The work was partially supported by Slovak Science Grant Agency VEGA, project No. 1/0162/03. I would like to thank Pavol N´avrat and M´ aria Bielikov´a for their valuable suggestions.

8. References

1. Floyd, R.W.: The paradigms of programming. Communications of the ACM **22** (1979) 455–460
2. Coplien, J.O.: 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 (1999) Available at http://www.old.netobjectdays.org/mirrors /stja.cd/Beitraege/ JimCoplien/Tutorial.ppt (accessed in June 2005).
3. Vranić, V.: Towards multi-paradigm software development. Journal of Computing and Information Technology (CIT) **10** (2002) 133–147
4. Coplien, J.O.: Multi-Paradigm Design for C++. Addison-Wesley (1999)
5. Coplien, J.O.: Multi-Paradigm Design. PhD thesis, Vrije Universiteit Brussel, Belgium (2000) Available at http://users.rcn.com/jcoplien/Mpd/Thesis/Thesis.pdf (accessed in June 2005).
6. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA): A feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA (1990) Available at [24] (accessed in June 2005).
7. Chastek, G., Donohoe, P., Kang, K.C., Thiel, S.: Product line analysis: A practical introduction. Technical Report CMU/SEI-2001-TR-001, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, USA (2001) Available at [24] (accessed in June 2005). 8. Geyer, L.: Feature modelling using design spaces. In: Proc. of the 1st German Product Line Workshop (1. Deutscher Software-Produktlinien Workshop, DSPL-1), Kaiserslautern, Germany, IESE (2000) Available at http://wwwagss.informatik.uni-kl.de/Veroeffentl/ FeatureModelingUsingDesignSpaces.pdf (accessed in June 2005).
9. Griss, M.L., Favaro, J., d'Alessandro, M.: Integrating feature modeling with

the RSEB. In Devanbu, P., Poulin, J., eds.: Proc. of 5th International Conference on Software Reuse, Victoria, B.C., Canada, IEEE Computer Society Press (1998) 76–85 Available at http://www.favaro.net/john/home/publications/rseb.pdf (accessed in June 2005).
10. Kang, K.C., Kim, S., Lee, J., Kim, K., Shin, E., Huh, M.: FORM: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering **5** (1998) 143–168
11. Simos, M.A.: Organization domain modeling (ODM): Formalizing the core domain modeling life cycle. In: Proc. of the 1995 Symposium on Software reusability, Seattle, Washington, United States, ACM Press (1995) 196–205
12. Software Engineering Institute, Carnegie Mellon University: A framework for software product line practice. (http://www.sei.cmu.edu/productlines/framework.html) Accessed in June 2005.
13. Claub, M.: Modeling variability with UML. In: Proc. of Net.ObjectDays 2001, Young Researchers Workshop on Generative and Component-Based Software Engineering, Erfurt, Germany, tranSIT (2001) 226–230
14. Riebisch, M., B¨ ollert, K., Streitferdt, D., Philippow, I.: Extending feature diagrams with UML multiplicities. In: Proc. of the 6th Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, California, USA, Society for Design and Process Science (2002) Available at http://www.theoinf.tu-ilmenau.de/»riebisch/publ/IDPT2002-paper.pdf, accessed in June 2005).
15. Jia, Y., Gu, Y.: The representation of component semantics: A feature-oriented approach. In Crnkovi´c, I., Larsson, S., Stafford, J., eds.: 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 (2002) Available at http://www.idt.mdh.se/»icc/cbse-ecbs2002/jiayu.pdf (accessed in June 2005).
16. Czarnecki, K.: 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) Available at http://www.prakinf.tu-ilmenau.de/»czarn/diss (accessed in June 2005).
17. Czarnecki, K., Eisenecker, U.W.: Generative Programing: Methods, Tools, and Applications. Addison-Wesley (2000)
18. Vranić, V.: Multi-Pradigm Design with Feature Modeling. PhD thesis, Slovak University of Technology in Bratislava, Slovakia (2004) Available at http://www.fiit.stuba.sk/»vranic.
19. Vranić, V.: Reconciling feature modeling: A feature modeling metamodel. In Weske, M., Liggsmeyer, P., eds.: Proc. of 5th Annual International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays 2004). LNCS 3263, Erfurt, Germany, Springer (2004) 122–137 20. Vrani´c, V.: AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. In Bosch, J., ed.: Proc. of 3rd International Conference on Generative and Component-Based Software

Engineering (GCSE 2001). LNCS 2186, Erfurt, Germany, Springer (2001) 48–57
21. Eclipse.org: AspectJ project home page. (http://eclipse.org/aspectj) Accessed in June 2005.
22. Knutson, C.D., Budd, T.A., Vidos, H.: Multiparadigm design of a simple relational database. ACM SIGPLAN Notices **35** (2000) 51–61
23. Stein, D., Hanenberg, S., Unland, R.: A uml-based aspect-oriented design notation for aspectj. In Kiczales, G., ed.: Proc. of 1st International Conference on Aspect-Oriented Software Development, ACM Press (2002) 106–112
24. Software Engineering Institute, Carnegie Mellon University: Home page. (http://www.sei.cmu.edu) Accessed in June 2005.

Valentino Vranić is a researcher at Institute of Informatics and Software Engineering, Faculty of Informatics and Information Technology of the Slovak University of Technology in Bratislava. He holds a Bc. (BSc.) and Ing. (MSc.) in information technology, and PhD. in program and information systems, all from the Slovak University of Technology in Bratislava. His main research interests are multi-paradigm software development, domain engineering, and aspect-oriented programming.