AspectJ Paradigm Model: A Basis for Multi-Paradigm Design for AspectJ*

Valentino Vranić

Department of Computer Science and Engineering
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology, Ilkovičova 3, 812 19 Bratislava, Slovakia
`vranic@elf.stuba.sk`
`http://www.dcs.elf.stuba.sk/~vranic`

Abstract Multi-paradigm design is a metaparadigm: it enables to select the appropriate paradigm among those supported by a programming language for a feature being modeled in a process called transformational analysis. A paradigm model is a basis for multi-paradigm design. Feature modeling appears to be appropriate to represent a paradigm model. Such a model is proposed here for AspectJ language upon the confrontation of multi-paradigm design and feature modeling. Subsequently, the new transformational analysis is discussed.

1 Introduction

In this paper the AspectJ paradigm model, a basis for multi-paradigm design for AspectJ programming language (version 0.8), is proposed. AspectJ is an aspect-oriented extension to Java [6]. Multi-paradigm design for AspectJ is based on Coplien's multi-paradigm design [3] (originally applied to C++ and therefore known as multi-paradigm design *for C++*) to a different solution domain. It employs feature modeling [5] for the task Coplien's multi-paradigm design used scope, commonality, variability, and relationship analysis [4].

Scope, commonality, variability, and relationship analysis, which is basically a scope, commonality, and variability analysis [1] enhanced with the analysis of relationships between domains [4], is used to describe the paradigms (mechanisms of a programming language) provided by the solution domain (i.e., programming language), as commonality-variability pairings [4, 3]. This way of describing paradigms is compact, but not expressive enough to meet the requirements of the transformational analysis, a process of aligning problem domain structures with available paradigms.

Moreover, the paradigms are often connected, but multi-paradigm design provides no means to express how. The application of feature modeling instead of scope, commonality, variability, and relationship analysis could help solve the problems mentioned here, as will be shown in this paper.

* This work was partially supported by Slovak Science Grant Agency, grant No. G1/7611/20.

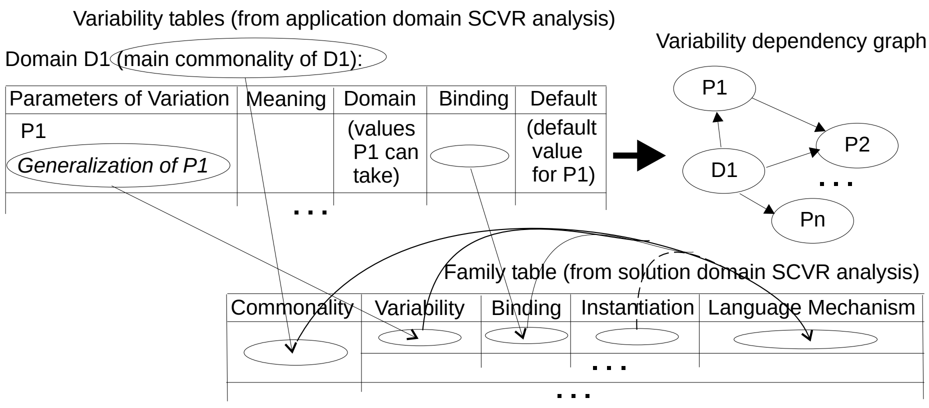

Figure 1. Transformational analysis in MPD.

Before presenting the actual AspectJ paradigm model, a critical survey of the issues regarding the multi-paradigm design for C++ (Sect. 2) and a basic information on feature modeling notation is provided (Sect. 3). Also, the relationship between feature modeling and techniques used in multi-paradigm design is analyzed (Sect. 4). AspectJ paradigm model is then presented (Sect. 5) and the impact of incorporating feature modeling into MPD on transformational analysis discussed (Sect. 6). Conclusions and further research directions close the paper (Sect. 7).

2 Multi-paradigm design for C++

Multi-paradigm design (MPD) for C++ [3] is based on the notion of small-scale paradigm [8], that can simplistically be perceived as a language mechanism (e.g., inheritance), as opposed to the (more common) notion of large-scale (programming) paradigm [2] (e.g., object-oriented programming; see [7] for a comparison of programming paradigms).

Figure 1 gives an overview of MPD. Scope, commonality, variability, and relationship (SCVR) analysis is performed on both domains, application and solution, with results summarized in variability (one for each domain) and family tables, respectively. The variability tables are incapable of capturing dependencies between the parameters of variation (that are also considered to be domains), so this is enabled by a simple graphical representation called *variability dependency graphs*. Each row of the family table represents a 4-tuple *(Commonality, Variability, Binding, Instantiation)* that determines the language mechanism.

The transformational analysis is actually a process of matching the application domain structures with the solution domain ones. First, the main commonality of the application domain, as identified by a developer, is matched with a commonality in the family table. This yields a set of rows in which the individual parameters of variation are resolved. Since parameters of variation (e.g., working set management) are too specific to be matched with general variabilities (e.g., algorithm) in the family table, each parameter of variation must be generalized before matching. This seem as a too big step to make at once.

The generalization problem and the fact that the matching is performed between variability table *3*-tuples and family table *4*-tuples (variability table has no instantiation column), are eclipsed by another problem: some C++ language mechanisms are missing from the paradigm model proposed. For example, classes and methods (procedures) are not even mentioned. On the other hand, inheritance is embraced in the model. Maybe Coplien considered classes and methods too trivial to mention, but this has not been stated explicitly.

Moreover, C++ mechanisms listed in the family table and negative variability table[1] are inconsistent with those described in the text [3]. Yet another problem with the paradigm model in MPD is that it does not capture the dependencies between paradigms. This is an important information, since there are paradigms that make no sense without other paradigms (e.g., inheritance without classes in C++).

3 Feature Modeling

Feature modeling is a conceptual modeling technique used in domain engineering. The version of the feature modeling whose notation is described here comes from [5].

Feature diagrams are a key part of a feature model. A feature diagram is basically a directed tree with the edge decorations. The root represents a concept, and the rest of the nodes represents features. Edges connect a node with its 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 the 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 from into *alternative* and *or-features.* A concept instance has exactly one feature from the set of alternative features. It can have any subset or all of the features from the set of or-features.

The nodes connected directly to the concept node are being denoted as its *direct features*; all other features are its *indirect features*, i.e. *subfeatures.* The indirect features can be included in the concept instance only if their parent node is included.

An example of a feature diagram with different types of features is presented in Fig. 2. Features f_1, f_2, f_3, and f_4 are direct features of the concept c, 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.

[1] A table that summarizes language mechanisms corresponding to exceptions to variability.

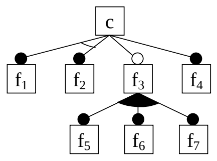

Figure 2. A feature diagram.

4 Applying Feature Modeling to Multi-Paradigm Design

Feature modeling is not unlike SCVR analysis. SCVR analysis, the heart of MPD, is based on the notions of commonality and variability (hence the name), and the notions of *common* and *variable* features is not unknown to feature modeling.

A common feature of a concept is a feature present in all concept instances, i.e. there must be a path of (pure) mandatory features leading from the concept to the feature. All other features are variable, i.e. any optional, alternative or or-feature is variable. The features to which variable features are attached are called *variation points*.

The scope in SCVR analysis, defined as a set of entities, is nothing but the concept in an *exemplar* representation.[2] The SCVR commonalities (assumptions held uniformly across the scope) and variabilities (assumptions true for only some elements in the scope) map straightforwardly to common and variable features of feature modeling, respectively.

The feature modeling enables to represent SCVR analysis commonalities and variabilities hierarchically and thus to express relationships among variabilities. For a solution domain SCVR analysis this means enabling to express how the paradigms it provides are related.

The most important results of SCVR analysis are provided in variability and family tables and variability dependency graphs.

4.1 Variability and Family Tables

Table 1 aligns the terms of feature modeling with its variability and family table counterparts (the columns). Only a fraction of the information provided usually by a feature model covers most of the needs of variability and family tables.

The parameters of variation are sometimes considered as subdomains (especially in variability dependency graphs). This is consistent with the feature modeling; the feature can be viewed as a concept.

Binding mode in feature modeling corresponds to binding time in MPD. The difference is that the set of binding times used in MPD is richer than the one used in feature modeling. This is due to a fact that the binding times in MPD are the actual binding times of a solution domain, like compile time, run

[2] The exemplar view of a concept is the one in which a concept is defined by a set of its instances [5].

Table 1. Feature modeling and MPD variability and family tables.

Feature modeling	**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 mode	binding	binding
semantic description, rationale	meaning	
default dependency rules	default (value)	
additional information		instantiation

time, etc. Feature modeling provides more abstract binding times, namely static, changeable, and dynamic binding. Each MPD binding time falls into one of these categories: source time and compile time bindings are static binding, link (load) time binding is a changeable binding, and run time binding is a dynamic binding.

The binding time applies only to variable features. It should be understood only as an auxiliary information to the transformational analysis. There is no notion of a unique binding time for a whole concept, as it is the case with a paradigm in MPD. Binding time should be indicated where it belongs—at variable features.

The feature modeling provides no counterpart for the family table column "instantiation", which indicates whether a language mechanism provides instantiation. This information should be provided as an attribute among the rest of the information associated with a feature model.

Possible values for instantiation in MPD are: *yes*, *no*, *not available* (*n/a*), and *optional*. It seems that *no* and *n/a* values are redundant: if a language mechanism does not provide instantiation, it can be only because the instantiation is not available for that mechanism. The *yes* value indicates that a mechanism is used only with instantiation, while *optional* means that it can be used both with instantiation and without it (a class doesn't have to be instantiated to make a use of the static fields and methods). Furthermore, the *optional* value does not make sense in the application domain—the instantiation is either needed or not.

4.2 Variability Dependency Graphs

In variability dependency graphs, the nodes represent domains and the directed edges represent the "depends on (a parameter of variation)" relationship; domain corresponds to a concept or feature (considered as a concept).

Parts of variability dependency diagrams can be derived from the feature diagrams. Commonality domain depends on its parameters of variation, or—in the feature modeling terminology—concept depends on its variation points. But, generally speaking, while the relationships between domains in variability dependency graphs have a particular semantics, this cannot be said for the relationships in feature diagrams. Moreover, the feature diagrams are trees, not general graphs. All this suggests that variability dependency graphs should be kept as a separate notation. For each domain from the variability dependency graphs there should be a corresponding concept or feature in the feature model.

5 AspectJ Paradigms

AspectJ is an interesting programming language to explore in the sense of MPD because it supports two large-scale paradigms: object-oriented programing and aspect-oriented programming. However, large-scale view is not sufficient to make a full use of the programming language in the design. We must turn to a finer granularity and find out what small-scale paradigms, i.e. language mechanisms, AspectJ provides (referred to as *paradigms* in the following text). As was discussed in the previous sections, feature modeling will be employed to describe these paradigms.

Figure 3 shows a feature diagram of AspectJ. The paradigms in the feature diagram are indicated by a capitalization of the initial letter (e.g., Class). Binding time is indicated at variable features; if not, source time binding is assumed. Sometimes binding time of a feature depends on other features, as indicated in the diagram. In the text, the names of paradigms are typeset in the **boldface style**. The root of the feature diagram is AspectJ as a solution domain. It provides the paradigms that *can* be used, which is indicated by modeling the topmost paradigms as optional features.

The paradigm model establishes a paradigm hierarchy. Each paradigm is presented in a separate diagram as an alternative to the one big overall diagram. Wherever a root node of a paradigm tree is present, it is as if a whole tree was included there.

6 Transformational Analysis

Transformational analysis—aligning application domain structures with the solution domain ones—is the key part of MPD. The basic idea of how the transformational analysis is to be performed when these structures are represented by feature models is presented by the means of an example. Afterward, some general observations about the process of transformational analysis are given.

6.1 An Example: Text Editing Buffers

Text editing buffers[3] represent a state of a file being edited in a text editor. Text editing buffer caches the changes until user saves the text editing buffer into the file. Different text editing buffers employ different working set management

[3] The example discussed here is an adapted version of the text editing buffers example from [3].

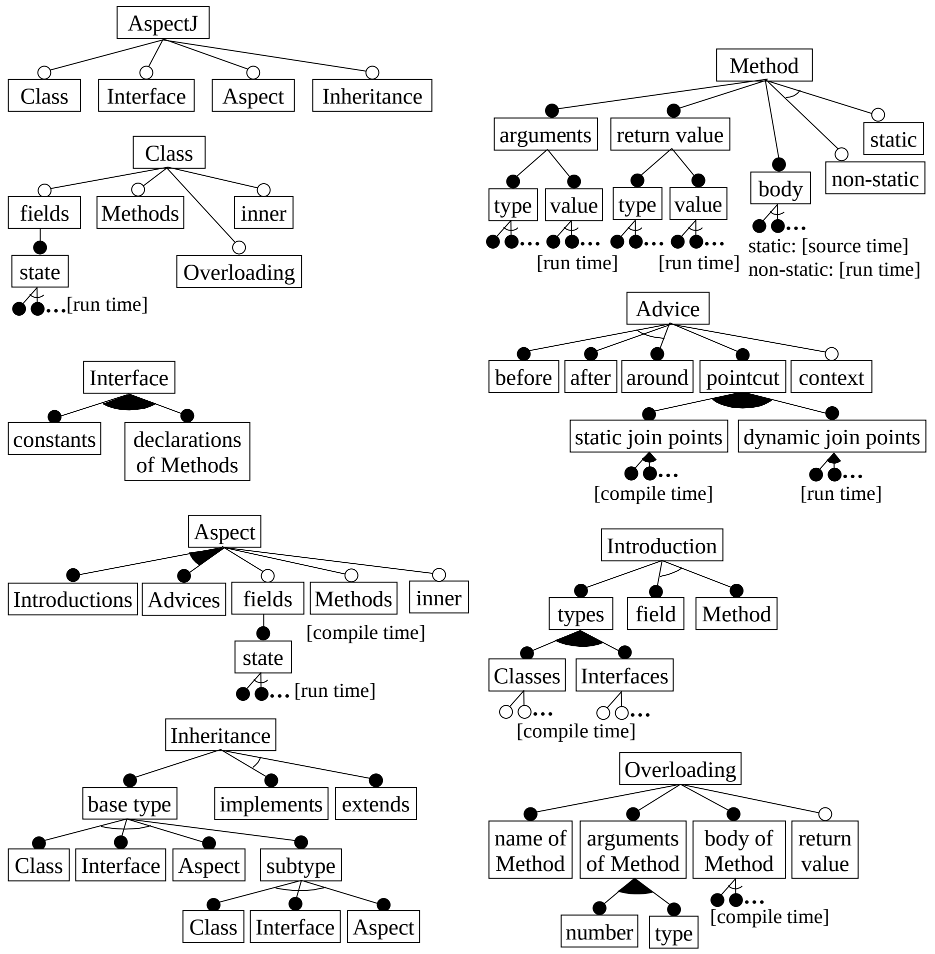

Figure 3. AspectJ paradigm model.

schemes and use different character sets. All text editing buffers load and save their contents into the file, maintain a record of the number of lines and characters, the cursor position, etc. The text editing buffer feature diagram is presented in Fig. 4. In the text, the feature names are distinguished by typesetting in the *italics style*. For simplicity, binding time and instantiation were not considered.

Now that feature models of both application and solution domains are available, we can proceed with the transformational analysis. We start with the unchangeable part of the application domain, i.e. the topmost common features. At this level a basic class or classes might be expected. The features *number of lines*, *number of characters*, and *cursor position* correspond to fields of the **class** paradigm. On the other hand, *yield data*, *replace data*, *load file*, and *save file*

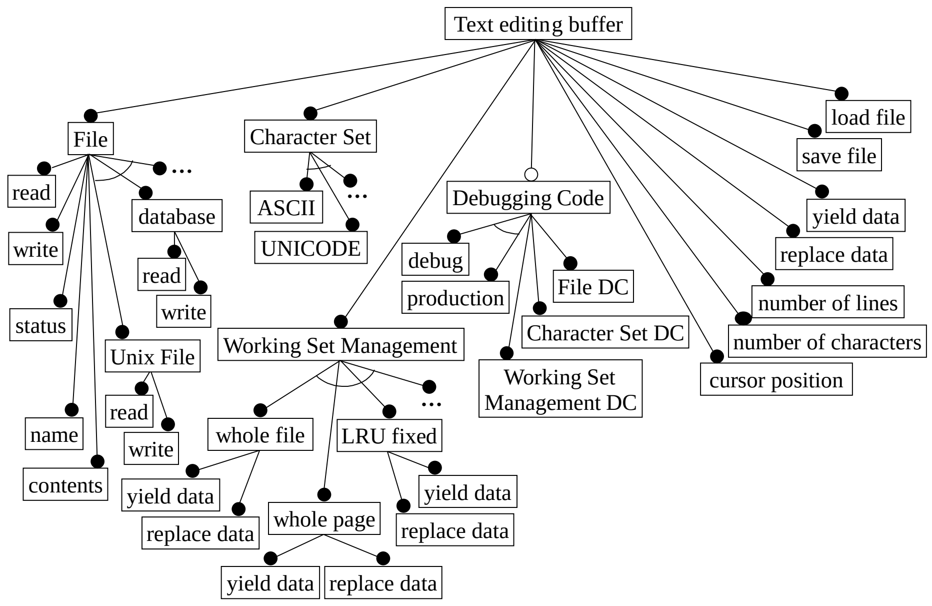

Figure 4. Text editing buffers feature diagram.

correspond to **method** paradigm. Accordingly, text editing buffer should be a class.

The rest of the topmost features are, apparently, variation points. The first one is *file*. All the files are read and written, but there are several file types and each one is read and written in a specific way. However, what is being read and written remains the same: *file name* and *contents*. We would probably expect to get the *status* of reading and writing. Thus we reached the leaves of the *file* subtree. If we compare these leaves to those of AspectJ feature diagram, they best match with arguments and return value. This brings us to **method** paradigm for *read* and *write* features.

We go up one level and discover that *database*, *RCS file*, *TTY*, and *Unix file* features match with the **class** paradigm. Accordingly, we expect that *file* would be a class too; so we match it with the **class** paradigm. The relationship between *file* and the file types matches with **inheritance**. Analogously, *character set* would be a class, and each type of it would be a subclass of that class.

The situation is similar with *working set management*: each type of *working set management* would be a class. But there is one difference: if we try to match it with **inheritance** further, we discover that we can match the whole *text editing buffer* with *base type* (because of *yield data*, *replace data*). So the *working set management* would be a primary differentiator.

Debugging code is somewhat special. It should be possible to turn it on and off easily (to obtain debug and production version, respectively). It is intended for *file*, *character set*, and *working set management* debugging; there is a special debugging code for each of those. For example, we would like to know when the file is being read from and written to. We already matched *file* with **class** and reading and writing with **method**, so it seems we must look for such a paradigm that can influence the execution of methods. There is only one such paradigm: **advice**. As **advice** is available only in **aspect** paradigm, the *file debugging code*, *character set debugging code*, and *working set management debugging code* will be aspects. *File debugging code* will provide two **advice**s, one for reading and the other for writing a file, and *character set debugging code* only one, as only a name of character set being used has to be announced.

Things are slightly more complicated with *working set management debugging code*, as we are interested in the general operations of working set management, as well as in the specific operations of each type of it (not displayed in the feature diagram). This points us to **inheritance**: *working set management debugging code* matches with a base aspect, while each of its or-subfeatures matches with a sub-aspect.

6.2 Transformational Analysis Outline

The text editing buffers example disclosed some regularities in the process of transformational analysis. The matching was performed starting at leaves towards the root. Rarely the leaves were considered alone. Mostly, a feature was considered together with its first-level subfeatures. Multiple nodes from the application domain can match with a single solution domain node if its name is in plural. Matching of nodes is done according to the type of the nodes, e.g. the overall match of mandatory or-nodes is successful if a match has been found for one or more leaves.

The matching is interdependent. If two features depend on each other, then it matters what paradigm the first feature was matched with. In other words, matching a feature with a paradigm constrains the further design.

Up to now, nothing has been said about how the actual matching of two nodes is performed. This can be compared to the matching between the domain commonality and parameters of variation from the variability table to the commonalities and variabilities from the family table. Two nodes match if they conceptually represent the same thing; do they—it is up to the developer to decide. However, a conceptual gap is significantly smaller than in the original MPD where developer was forced to make such decisions at a very high level of abstraction.

7 Conclusions and Further Research

The table representation of the application and solution domains used in multi-paradigm design for C++ performs unsatisfactorily during the transformational analysis. Moreover, the C++ paradigm model is incomplete. The application of feature modeling instead of scope, commonality, variability, and relationship analysis leads to a more appropriate representation—the feature model—which enables to represent relationships between paradigms.

In this paper, such a paradigm model of AspectJ is proposed. The development of AspectJ paradigm model was based on an extensive comparison of feature modeling and multi-paradigm design (for C++) presented in Sect. 4. The use of the AspectJ paradigm model—a new transformational analysis—was demonstrated on text editing buffers example (Sect. 6) and then the outline of the process was drawn. The process of transformational analysis is more visible and easier to perform with feature models than with tables.

The AspectJ paradigm model presented in this paper provides a basis for further research on multi-paradigm design for AspectJ and its subsequent improvements are expected especially regarding the transformational analysis. The relationship of negative variability tables used in multi-paradigm design and feature modeling has to be investigated. Variability dependency graphs have to be incorporated into the transformational analysis. The transformational analysis results should be noted in a more appropriate form than a textual representation is. A graphical notation would be suitable here, which points to the need for a CASE tool. Besides these immediate issues, the discussion of scope, commonality, variability, and relationship analysis and feature modeling has tackled a deeper question of the relation of multi-paradigm design and generative programming [5].

References

[1] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6), November 1998. Available at `http://www.bell-labs.com/people/cope` (accessed on May 14, 2001).

[2] James O. Coplien. Multi-paradigm design and implementation in C++. In *Proc. of GCSE'99*, Erfurt, Germany, September 1999. Presentation slides and notes. Published on CD. Available at `http://www.bell-labs.com/people/cope` (accessed on May 14, 2001).

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

[4] James O. Coplien. *Multi-Paradigm Design.* PhD thesis, Vrije Universiteit Brussel, Belgium, 2000. Available at `http://www.bell-labs.com/people/cope` (accessed on May 14, 2001).

[5] Krysztof Czarnecki and Ulrich Eisenecker. *Generative Programing: Principles, Techniques, and Tools.* Addison-Wesley, 2000.

[6] Gregor Kiczales et al. An overview of AspectJ. In *Proc. of ECOOP 2001—15th European Conf. on Object-Oriented Programming*, Budapest, Hungary, June 2001. Available at `http://aspectj.org` (accessed on May 14, 2001).

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

[8] Valentino Vranić. Towards multi-pradigm software development. Submitted to CIT, 2001.