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

Valentino Vranić

4th May 2001

Abstract

Multi-paradigm design for C++ is a metaparadigm: it enables to select the appropriate paradigm for a feature being modeled. Multi-paradigm design is based on scope, commonality, and variability analysis used to analyze and model both an application domain and solution domain. Subsequently, a technique called transformational analysis is used to match the application domain entities with the solution domain ones (called paradigms). The notation employed in multi-paradigm design consists mainly of tables. This doesn't seem to be enough to express all the important aspects of a domain, whether an application or solution one. Feature modeling seem to be a more suitable technique for this task. However, having application and solution domain represented as feature models has a great impact on the transformational analysis.

1 Introduction

This report is concerned with AspectJ paradigm model intended to be used in the multi-paradigm design for AspectJ. As I explained in the previous report [5], SCV (scope, commonality, and variability) analysis [1] is used to describe paradigms as commonality–variability pairings [3]. This way of describing paradigms is compact, but not clear.

Some paradigms build on other paradigms. Multi-paradigm design, based on scope, commonality, and variability analysis, provides no means to express these relationships between paradigms. Therefore, in [5] I suggested to apply feature modeling instead of scope, commonality, and variability analysis. Feature modeling also supports explicit reasoning about commonalities and variabilities. Actually it does so in a more general way.

To enable multi-paradigm design for AspectJ, the AspectJ paradigm model is needed. This is a tough task because that model is to be used in the transformational analysis to map problem domain structures to language mechanisms. Hence I decided first to build an initial (and incomplete) version of the AspectJ paradigm model and to try it on example to demonstrate and explore how the transformational analysis will look like when feature modeling is used.

Before discussing the main issue mentioned above, this report provides a basic information on feature modeling (Section 2). It proceeds with an analysis of the relationship between feature modeling and techniques used in MPD (Section 3). Then it provides a partial AspectJ paradigm model developed by applying feature modeling on AspectJ solution domain (Section 4) and discusses the impact of incorporating feature modeling into MPD on transformational analysis (Section 5). It ends with conclusions and further work proposals.

2 Feature Modeling

Feature modeling is a conceptual modeling technique used in domain engineering. Its origins are in FODA, i.e. feature-oriented domain analysis, a domain analysis method developed at the Software Engineering Institute. The version of the feature modeling referred to here is an adaptation of the original version. It is described in [4].

The feature modeling is based on the notions of concept and feature. The concepts are the way we perceive the world. It is not possible to define a concept precisely and cover all its instances, except for some special cases, e.g. mathematical concepts (see [4] for more details on conceptual modeling). The feature modeling makes it possible to explore the features concepts have in order to better understand them.

The basic steps of feature modeling (a micro-cycle of feature modeling) are:

1. Recording the similarities between instances, i.e. common features.
2. Recording the differences between instances, i.e. variable features.
3. Organizing the features into *feature diagrams*.
4. Analysis of the feature combinations and interactions.
5. Recording all of the additional information regarding features.

The result of feature modeling is, as expected, a feature model. It consists of feature diagram(s) and the information associated with the features like:

- semantic description
- rationale
- stakeholders and client programs
- exemplar systems
- constraints and default dependency rules
- availability sites, binding sites, and binding modes
- open/closed attribute
- priorities.

Feature diagrams (mentioned in step 3) are a key part of the feature model. They are used to represent concepts. A feature diagram is a directed tree with the edge decorations. The root represents a concept, and the rest of the nodes represent features.

Edges connect the 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. As the names indicate, a concept instance must have all the mandatory features and can have the optional features.

The edge decorations are drawn as arcs connecting 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. A concept instance 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, sometimes denoted as *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. 1. f_1, f_2, f_3, and f_4 are direct features of the concept c, while other features are its indirect features.

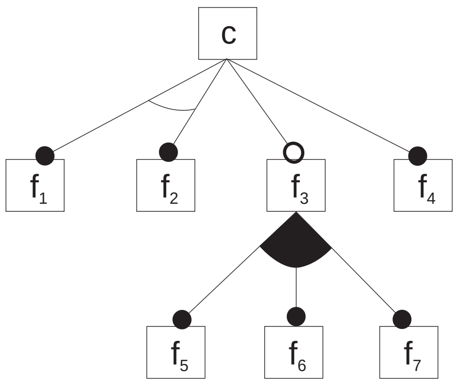

Figure 1: A feature diagram.

f_1 and f_2 are mandatory alternative features. f_4 is an optional feature. f_5, f_6 and f_7 are mandatory or-features; they are also subfeatures of f_4.

According to [4], a concept instance is described by a set of features from the feature diagram of the concept. However, if the same feature is present in the feature diagram at several distinct nodes, this is not sufficient to describe an instance. An element is either in the set or not, no matter how many times it was included into the set.

Redrawing the feature diagrams of this type not to include duplicated nodes results in graphs (more precisely, directed acyclic graphs). Such type of feature diagrams is actually mentioned in [4] (in a footnote) the feature sets cannot be used to describe an instance of a concept whose feature diagram is a graph. It is surprising when the authors do not recognize that some of the examples they introduce later in the book falls into this category (e.g., Fig. !4-3 on page 634).

It is obvious that there are several solutions to this problem. Although this is not the intent of this report, I would like to introduce a simple one. We can stay with sets, but the difference is that each subtree of a feature diagram should be described by a set:

$$\{root, \{f_1\}, \{f_2\}, \ldots, \{f_n\}\}$$

The process proceeds recursively to the feature diagram leaves: each feature unless it is at leaf, is the root of some subtree. For example, one of the instances of the concept from Fig. 1 in this notation could be written as: $\{c, \{f_1\}, \{f_3, \{f_5, f_6\}\}, \{f_4\}\}$.

3 Feature modeling and MPD

In [5] I suggested to use feature modeling instead of SCV analysis as a more general technique. In this section we will take a closer look at how these two techniques correspond to each other.

In MPD, SCV analysis is applied to both application and solution domain; the results are summarized in two kinds of tables, variability and family tables, respectively. Furthermore, another technique denoted as variability dependency graphs is used in MPD. We will consider both the variability and family tables, and variability dependency graphs in the context of feature modeling in order to how the entities they are based on correspond to concepts and features. Concepts and features are more general than the entities MPD is based on, as it is the case with object-orientation, too. In OO, we can say that classes correspond to concepts.[1]

[1]Not vice versa, i.e. that concepts correspond to classes, as stated in [4, p. 736], since a concept can be, for example, an aspect or something that cannot be modeled by a (single) class.

	SCV analysis	*Feature modeling*
scope	= set of entities	concept
commonality	= assumption held uniformly across the scope	common feature
variability	= assumption true for only some elements in the scope	variable feature

Table 1: SCV analysis and feature modeling equivalents.

3.1 SCV Analysis

Feature diagrams are capable of expressing commonalities and variabilities among concept instances. Let us see how they can be used in the context of SCV analysis, which is based around the notion of commonality and variability. As we saw, there is a notion of *common* and *variable* features in feature modeling. These can be distinguished by examining feature diagrams.

A common feature of a concept is a feature present in all concept instances, i.e. there must be a path of mandatory[2] features leading from the concept to the feature. All other features are variable, i.e. any optional, alternative or or-features is a variable feature. The features to which variable features are attached are called *variation point.*

If we consider common features in a relative manner, then any node, be it a feature or a concept, in a feature diagram can have *its* common features. Such features are denoted as common subfeatures. A common subfeature of a node in feature diagram is a feature present in all concept instances that have that node, i.e. there must be a path of mandatory features leading from the feature to the subfeature. Clearly, common subfeatures include common features.

As it is apparent from Tab. 1, SCV analysis and feature modeling are not far from each other. It is clear that the scope in SCV analysis is nothing but the *exemplar* representation of a concept.[3] The SCV commonalities and variabilities map straightforwardly to common and variable features of feature modeling, respectively. From the perspective of SCV analysis those common subfeatures that are not common features fall into the variability category.

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

3.2 Variability and Family Tables

Tab. 2 aligns feature modeling terms with its variability and family table counterparts. Clearly, only a fraction of the information usually provided by a feature model covers most of the needs of variability and family tables (compare Feature modeling column of Tab. 2 with the list of the information associated with the features at the beginning of this section).

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 itself if we decide to consider it as such (but until that it stays only a feature).

Two issues—binding time and instantiation—require a closer examination.

Binding time

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, like compile time,

[2]This means only pure mandatory features, not mandatory alternative or or-features.

[3]The exemplar view of a concept is the one in which a concept is defined by the set of its instances. See [4] for more on conceptual modeling.

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 (range)	
binding mode	binding	binding
semantic description, rationale	meaning	
default dependency rules	default (of range)	
additional information		instantiation

Table 2: Feature modeling and MPD variability and family tables.

run time, etc. Feature modeling provides us with more abstract binding times, namely static, changeable, and dynamic binding. Each MPD (MPD for C++) binding time fall into one of these categories, as follows:

- *source time* and *compile time* bindings are static binding;
- *link (load) time* binding is a changeable binding;
- *run time* binding is a dynamic binding.

The set of binding times depends on a solution domain. Different programming languages provide different mechanisms; each mechanism has its binding time. Since the solution domain is determining for transformational analysis, we will accept that binding mode attribute actually holds a value for binding time.

The binding time applies only to variable features. The binding times of the features in the application domain feature model is the *earliest* allowed binding time. This means that the binding time in the implementation may be changed to a later one, but not to an earlier one. The binding times of the features in the solution domain feature model are the exact binding times.

The binding times of the variable features are indicated in feature diagrams at binding sites (the parent features of those variable features) or directly at variable features in the form of node annotations.

There is no notion of a unique binding time for a concept, as it is the case with a paradigm in MPD. However, it is better to indicate binding time where it belongs—at variable features—than to have to provide an explanation about binding what is the binding time about. In case of such alternative or or-features whose binding time is the same, it is enough if we annotate only the first feature.

Instantiation

The feature modeling provides no counterpart for the family table column "instantiation". This column indicates whether a language mechanism provides instantiation. This does not seem to be a serious problem, since it is possible to provide this information among the rest of the additional information as an attribute.

To avoid misunderstanding, the instantiation considered here is not the instantiation of concepts. By definition it is possible to instantiate any concept. A concept is an idea and its instances are the materialization of that idea. The instantiation discussed here has to do with the *instantiation of concept instances* themselves. Consider, for example, the concept of class.

The instances of the concept of class are classes. But the classes themselves can be instantiated; their instances are objects.

Let's see what values are possible for the instantiation attribute. 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 (to make a use of the static fields and methods, a class doesn't have to be instantiated).

The instantiation attribute in solution feature modeling should take one of the three values for each concept: *yes*, *no*, and *optional.* In application domain feature modeling, the information whether the instantiation is needed should be provided with each feature. The mapping possibilities for application domain structures to solution domain mechanisms would be then:

instantiation in application domain	instantiation in solution domain
yes	yes, optional
no	no, optional

3.3 Variability Dependency Graphs

Variability dependency graphs are an auxiliary technique used in MPD to capture dependencies between domains. It is used as a part of application domain analysis and, consequently, in transformational analysis. It helps in exploring the circular dependencies among domains.

The notation of variability dependency graphs is trivial: the nodes represent domains and the arrows represent the "depends on (a parameter of variation)" relationship. If we are to translate this into the feature modeling terminology, then 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 feature modeling terminology—concept depends on its variation points. But, generally speaking, the relationships between domains in variability dependency graphs have a particular semantics while this is not so with 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. On the other hand, the variability dependency graphs have to be attached to the feature model. In a complete feature model no important concept may be overlooked. Undoubtedly, the domains depicted in the variability dependency graphs are important concepts. Therefore, for each domain from the variability dependency graphs there should be a corresponding concept or feature in the feature model.

3.4 Summary

To summarize, here is what is undoubtedly needed in MPD among the information associated with the feature model (with explanation):

- Semantic description: the meaning of a feature
- Rationale: why the feature is included in the model and when to select it (if the feature is variable)
- Constraints:
  - mutual-exclusion: with what other feature is illegal to combine the feature

    – requires: what other features are required by the feature

- Default dependency rules: default values
- Binding mode: as explained above
- Instantiation: as explained above

Open/closed attribute (the feature is open if new direct variable subfeatures are expected) is redundant since it is already being indicated (by ellipsis) directly in feature diagrams.

All this information is recognized in MPD, but it is not structured this way. Structuring the associated information this way eases the transition to full-fledged feature modeling.[4]

4 AspectJ Paradigms

It is not easy to provide a good feature model of an application domain, where this technique is mostly applied. It is even harder to provide a good feature model of a solution domain, where there is no previous experience in applying this technique (at least not published—to the best author's knowledge). Thus, it would be untrue to claim that a feature model of AspectJ language as a solution domain, i.e. its paradigm model, presented here is complete and perfect. The experience with its use will show how it could be improved.

AspectJ is an interesting programming language to explore in the sense of MPD because it provides at least two large-scale paradigms: object-oriented and aspect-oriented. 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 (we will call them simply *paradigms* in the following text). As we discussed in previous sections, we employ feature modeling to describe these paradigms.

A whole feature model of AspectJ is presented in Appendix A. It consists of a feature diagram and the information associated with it, as described in Section 3.2. Despite the fact that feature diagrams are presented textually, the original expressiveness is fully preserved. Here is the legend:

`*` mandatory feature

`o` optional feature

`)` alternative feature

`]` or-feature

`...` indication that feature is open

In the information associated with the feature diagrams the following abbreviation have been used:

SD semantic description

R rationale

C constraints

    m mutually exclusive features

[4]If there is a chance to get generative programming and MPD to work together, then this is a small step towards it.

r required features

D default

SD and **R** are described textually, while **m**, **r**, and **D** are described by features connected with logical adjuncts.

4.1 The Paradigms

The paradigms in the feature diagram are indicated by a capitalization of the initial letter (e.g., Class). In the text, the paradigm names 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 paradigms as optional features.

The paradigm model establishes a paradigm hierarchy. Each paradigm is presented in a separate diagram in order not to make a diagram too big and hard to maintain. It is the same as if there was a single big feature diagram of AspectJ. However, in that case some subtrees would have to be repeated.

4.2 Dependencies between Paradigms

Some paradigms depend on other paradigms. It is worth distinguishing between two types of dependencies between paradigms: building upon and requiring.

A paradigm can build upon other paradigms. This is indicated directly in feature diagram, since the paradigm that is built upon must be present as a feature in the feature diagram of the paradigm that builds upon it. For example, **class** builds upon **method**).

Some paradigms can be used only in the context of some other paradigms, i.e. they *require* the presence of other paradigms. Therefore, this is being indicated in the "Requires" part of constraints description among the information associated with the feature diagram. As example, consider **class** and method again: **method** cannot be used without a **class**. This situation can only arise with the paradigms that are built upon, but not all such paradigms fall into this category. For example, **inheritance** builds upon **class**, but **class** can be used without **inheritance**).

5 Transformational Analysis

Transformational analysis—aligning application domain structures with the problem domain ones—is a key part of MPD. While in Coplien's MPD it was performed as a table comparison, here it becomes a tree traverse.However, this is a very simplified view: the transformational analysis is a complicated process. It can even have a back impact on the application domain feature model.

As MPD for AspectJ (based on feature modeling) is in an early phase of development, I will explore only the basic idea of the transformational analysis on an example. Then I will try to make some general observations about the process of transformational analysis.

5.1 An Example: Text Editing Buffers

Text editing buffers[5] [3] maintain the logical copies of the file contents during editing in a text editor. The text editing buffer represents the state of the file being edited. It caches changes until user saves the text editing buffer into the file. A simple text editing buffers maintains a

[5]The example discussed here is an adapted version of text editing buffers example used throughout [3].

copy of the entire file, while more sophisticated text editing buffers employ some kind of working set management, e.g. a paging or swapping scheme.

All text editing buffers can load and save their contents into the file. They maintain a record of the number of lines and characters of their contents, the cursor position, etc. Text editing buffers differ in the file formats used and working set management performed. Moreover, they can differ in the character set.

The feature model is presented in Appendix B. Only the topmost features are provided with the associated information (structured as explained in Section 4). In the text, the feature names are distinguished by typesetting in the SMALL CAPITALS STYLE.

Now that we have feature models of both application and solution domains, we can proceed with transformational analysis. We start with the unchangeable part of the application domain, i.e. the topmost common features. This is the level where we can expect some basic class or classes, so let's compare these features to those of the **class** paradigm. Number of lines, number of characters, and cursor position correspond to fields. Yield data, replace data, load file, and save file correspond to **method** paradigm. Accordingly, text editing buffer should be a class.

We proceed with other features, that are, apparently, variation points. The first one is FILE. All the files are read and written. There are several types of files 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 map to 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 **class** paradigm. Accordingly, we expect that FILE would be a class too; so we match it with **class** paradigm. The relationship between FILE and its types matches with **inheritance**. Analogously, CHARACTER SET would be a class, and each its type would be a subclass of this class.

The situation is similar with WORKING SET MANAGEMENT. We can determine each type of WORKING SET MANAGEMENT as a class. But there is one difference: when we try to match it with **Inheritance** further, we discover that we can match a 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. We would like to be able to turn it on and off easily (debug and production versions, respectively). It is intended for FILE, CHARACTER SET, and WORKING SET MANAGEMENT; there is a special debug code for each one of them. 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 methods execution. 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 advices, for reading and 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 each its type specific operations (not listed in the feature diagram). We can try **inheritance** here: WORKING SET MANAGEMENT DEBUGGING CODE matches with a base aspect, while each of its or-subfeatures match with a sub-aspect.

5.2 Transformational Analysis Outline

The text editing buffers example disclosed some regularities in the process of transformational analysis. The mapping was performed from leaves to root. Rarely we considered the leaves alone. Mostly a feature with its subfeatures was considered together, and we searched the solution domain tree for such a structure. Multiple nodes from the application domain can match to a single solution domain node if its name is in plural. Matching of leaves is done according to the type of the leaves, e.g. among the leaves that are mandatory or-nodes, it is a match if one or more leaves finds their match.

The mapping is interdependent. If two features depend on each other, then it matters what paradigm the first feature was matched with. This means that the order in which the application domain features are mapped is significant. In other words, matching feature with paradigm constrains 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 in the variability table to the commonalities and variabilities in the family table. The first ones must be generalized prior to matching. Two nodes match if they conceptually represent the same thing; do they—it is up to developer to decide. However, the conceptual gap is significantly smaller because we are not forced to make such decisions at so high level of abstraction as in the original MPD.

6 Conclusions and Further Research

Some serious insufficiencies were identified in MPD [5] regarding the paradigm model, both the concrete one (i.e., C++) and conceptually (i.e. the techniques employed for developing a paradigm model), and the transformational analysis.

In this report, an AspectJ paradigm model was presented as a feature model of AspectJ. The model provides a basis for further research on multi-paradigm design for AspectJ and its subsequent improvements are expected.

The development of AspectJ paradigm model is based on an extensive comparison of feature modeling and multi-paradigm design (for C++) presented in Section 3.

The use of the AspectJ paradigm model— a new transformational analysis—was demonstrated on text editing buffers example (Section 5).

There is a lot of open issues suitable for further research. The main include (subtasks are expected in each):

- The relationship of MPD's negative variability table and feature modeling.
- Incorporating MPD's variability dependency graphs into transformational analysis
- Noting the results of transformational analysis.

Despite the difference in the solution domain, multi-paradigm design for AspectJ can and should be confronted with multi-paradigm design for C++. Since the multi-paradigm design of text editing buffers example for C++ is available in [3], and the multi-paradigm design of the same example for AspectJ was presented here, the two could be compared[6]

References

[1] James Coplien, Daniel Hoffman, and David Weiss. Commonality and variability in software engineering. *IEEE Software*, 15(6), November 1998. Available at [2].

[6] I plan to do so for the STJA paper.

[2] James O. Coplien. Home page. `http://www.bell-labs.com/people/cope`. Accessed on February 5, 2000.

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

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

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

A AspectJ Paradigm Model

```
AspectJ
     o Method
     o Class
     o Interface
     o Aspect
     o Inheritance


Method
     * arguments
           * type
                 ]* ...
           * value
                 ]* ...

     o return value
           * type
                 ]* ...
           * value
                 ]* ...

    )* static <compile time>
    )* non-static <run time>


Overloading
     * Method name
     * Method arguments
          ]* number <source time>
          ]     ]* ... <run time>
          ]* type
                ]* ... <run time>

     * Method body
          ]* ... <compile time>

     o return value <source time>


Class
    ]* fields <source time>
    ]      * state
    ]           ]* ... <run time>
    ]
    ]* Methods <source time>

     o inner
          )* in Class
          )* in Method
                )* named
                )* anonymous

     o Overloading


Interface
    ]* constants <source time>
    ]* declarations of Methods <source time>
```

```
Aspect
    ]* Introductions
    ]* Advices

     * fields
           * state
                ]* ... <run time>

     * Methods

     o inner
          )* in Class
          )* in Method


Introduction
     * Types
          ]* Classes  <compile time>
          ]* Interfaces  <compile time>

    )* field
    )* Method


Advice
    ]* before
    ]* after
    ]* around

     * body
     * pointcut
           ]* static join points
           ]      ]* Classes
           ]      ]* Method executions
           ]      ]* Method calls
           ]      ]* ...  <compile time>
           ]
           ]* dynamic join points
                  ]* objects
                  ]* Method receptions
                  ]* ...  <run time>
     o context



Inheritance
     * base type
          )* Class
          )* Interface
          )* Aspect

          * subtype
                 )* Class
                 )* Interface
                 )* Aspect

    )* implements
    )* extends
```

Method

SD:

R:

C:

  m:

  r: Class or Aspect

Overloading

SD:

R:

C:

  m:

  r: Method

Class

SD: Class groups related data (fields) and operations (methods). (Here we consider classes without inheritance.)

R: Encapsulation of structure and behavior.

C:

  m:

  r:

Interface

SD:

R:

C:

  m:

  r:

Aspect

SD:

R:

C:

  m:

  r: Class $\vee$ Interface

Introduction

SD:

R:

C:

  m:

  r: Aspect

Advice

SD:

R:

C:

m:

r: Aspect

Inheritance

SD:

R:

C:

m:

- $subtype = Class \Rightarrow basetype \neq Aspect$
- $subtype = Interface \Rightarrow basetype \neq Aspect \wedge basetype = Class$
- $subtype = Aspect \Rightarrow basetype \neq Interface$

r:

- $basetype = Class \Rightarrow subtype = Class \wedge extends(basetype)$
- $basetype = Interface \Rightarrow (subtype = Class \wedge implements(basetype)) \vee (subtype = Interface \wedge extends(basetype))$
- $basetype = Aspect \Rightarrow subtype = Aspect \wedge extends(basetype)$

B Text Editing Buffers Feature Model

```
Text Editing Buffer
     * File
           * read
                 * file name
                 * file contents
                 * status

           * write
                 * file name
                 * file contents
                 * status

          )* database  <run time>
          )      * read
          )              * file name
          )              * file contents
          )
          )      * write
          )              * file name
          )              * file contents
          )              * status
          )
          )* RCS File
          )      * read
          )      * write
          )
          )* TTY
          )      * read
          )      * write
          )
     )* Unix file
     )      * read
     )      * write
     )
     )* ...

* Character Set
     )* ASCII <source time>
     )      * character codes
     )
     )* EBCDIC
     )      * character codes
     )
     )* FIELDATA
     )      * character codes
     )
     )* UNICODE
     )      * character codes
     )
     )* ...

* Working Set Management
     )* whole file  <compile time>
     )      * yield data
     )      * replace data
     )
     )* whole page
     )      * yield data
     )      * replace data
     )
     )* LRU fixed
     )      * yield data
     )      * replace data
     )
     )* ...

o Debugging Code
     )* debug  <compile time>
     )* production

      * File DC
            * reading file
            * writing file

      * Character Set DC
            * name of character set being used

      * Working Set Management DC
            * whole file
            * whole page
            * LRU fixed
            * ...

* number of lines
* number of characters
* cursor position

* yield data
* replace data

* load file
* save file
```

File

- **SD:** The type of the file where the text is stored.
- **R:** The formatting of text lines is sensitive to the file type.
- **C:**
  - **m:**
  - **r:**
- **D:** Unix file

Character Set

- **SD:** The character set used by a text buffer. The character set is determined by a code table in which each character is given a code.
- **R:** Different buffer types use different character sets.
- **C:**
  - **m:**
  - **r:**
- **D:** ASCII

Working Set Management

- **SD:** Different text edditing buffers use different working set management schemes. Each such a scheme provides its own operations for that task.
- **R:** Optimization of memory use.
- **C:**
  - **m:**
  - **r:**
- **D:**

Debugging Code

- **SD:** The code for debugging purposes. Must not create an overhead in the final version, i.e. should not be executed in that case.
- **R:** Freeing code from bugs.
- **C:**
  - **m:**
  - **r:**
- **D:** production