A New Basis for Multi-Paradigm Design

Valentino Vranić

March 15, 2001

1 Introduction

This is a brief report on the progress in the work on establishing a method for multi-paradigm design for AspectJ, an aspect-oriented extension to Java [7]. The method builds on Coplien's multi-paradigm design for C++ [3].

The work on establishing the method is being performed in two dimensions: while multi-paradigm design for C++ is being transformed to fit AspectJ, it is also being improved.

Albeit Coplien's idea of multi-paradigm design seems to be a good one, there are significant inconsistences in the multi-paradigm design for C++; this is discussed in Section 2. Section 3 closes the article and presents an idea how to overcome the problems of multi-paradigm design.

Partial results of the SCV (scope, commonality, and variability) analysis [1] of AspectJ is presented in Appendix A. Appendix B proposes some definitions of the key concepts in the MPD's underlying paradigm model (please take everything stated in the appendices with reserve).

2 Inconsistencies in MPD for C++

The most significant inconsistences in MPD for C++ are regarding its paradigm model and so-called transformational analysis. The next two sections describe the two problems, respectively.

2.1 Paradigms

The paradigm model in multi-paradigm design (MPD) for C++ is based upon a concept of the small-scale paradigm [6], which is the closest to the concept of the language mechanism (see B).

If we perceive the paradigm as a language mechanism, than we must ask why are some C++ language mechanisms missing in this model. 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 in that case he should have stated it explicitly. Anyway, classes does not seem so trivial to me. In my opinion they should be embraced in the paradigm model (see Appendix B).

Another problem with the paradigm model in MPD for C++ is that it does not capture the dependencies between paradigms. Some paradigms, build upon other paradigms (consider inheritance and classes). The family table is not sufficient to capture everything important about paradigms (see Section 2.2).

2.2 Transformational Analysis

The transformational analysis in MPD for C++ is actually a mapping of the application domain structures to the solution domain ones, as depicted in Fig. 1.

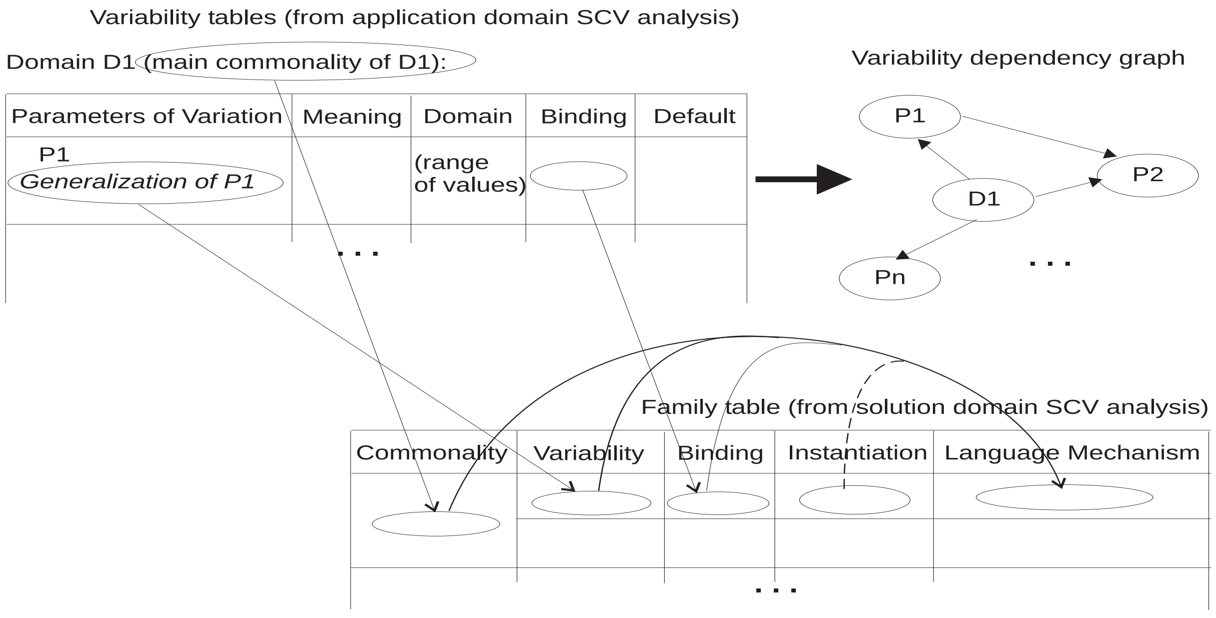
The application domain (SCV) analysis ends in variability tables, one per domain. The variability tables are capable of capturing dependencies between the parameters of variation. A simple graphical representation called variability dependency graphs can be used to do that, as shown in the right upper part of Fig. 1; the arrows mean "depends on". This notation is used in MPD for C++ to explore dependencies between domains, since parameters of variation can be domains in their own right.

The solution domain analysis[1] is being summarized in the family table. As can be seen from Fig. 1, the 4-tuple (Commonality, Variability, Binding, Instantiation) determines the language mechanism:

$$\text{(Commonality, Variability, Binding, Instantiation)} \rightarrow \text{Mechanism}$$

The mapping between these two types of tables is performed as follows. First, the main commonality of the application domain is mapped to a commonality in the family table. This yields a set of rows in which we proceed with resolving the individual parameters of variation. Since parameters of variation (e.g., working set management) are too specific to be mapped to general variabilities (e.g., algorithm) in the family table, each parameter must be first generalized. The generalized parameter of variation can be then mapped to a variability in the family table.

This should bring us to an appropriate language mechanism, but it is not sufficient to unambiguously determine the language mechanism because variability table has no column to map to the family table's instantiation column (i.e., we are trying to map 3-tuple to 4-tuple).



Variability tables (from application domain SCV analysis)
Domain D1 (main commonality of D1):
Parameters of Variation
Meaning
Domain
Binding
Default
P1
Generalization of P1
(range of values)
Variability dependency graph
P1
P2
D1
Pn
Family table (from solution domain SCV analysis)
Commonality
Variability
Binding
Instantiation
Language Mechanism

Figure 1: Transformational analysis in MPD.

3 Conclusions and Further Work

This article emphasized the inconsistences in the paradigm model of MPD for C++. The inconsistences have been found both in the solution and application domain parts of MPD. It seems that the root of the problems is common: the oversimplified model based on the SCV analysis.

[1]In the case of MPD for C++, the solution domain is C++, of course.

The problem is not in the SCV analysis itself, but in the assumption that it is sufficient to describe paradigms as a commonality–variability pairing (with some additional attributes) (i.e., a *single* commonality and a *single* variability).

I suggest to use the SCV analysis further, but to allow for *multiple* commonalities and variabilities. Since both commonalities and variabilities are actually *features*, the *feature modeling* [4] could be applied here.

The feature modeling should be applied to the application domain analysis as well. Actually, it is being applied already through the variability dependency graphs, but it is not used in the transformational analysis (at least not explicitly). The mapping between the application and solution domain is being performed using the tables.

Obviously, this new paradigm model can no longer rely on the table mapping. My further work is to provide an answer to the question how to do the mapping. Of course, before that, I must provide a new paradigm model of MPD for AspectJ by applying the feature modeling on AspectJ solution domain.

References

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

[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. *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. See [5].

[5] Krzysztof Czarnecki. Home page. `http:/www.prakinf.tu-ilmenau.de/~czarn`. Accessed on August 15, 2000.

[6] Valentino Vranić. Towards multi-pradigm software development. Written part of the PhD examination., September 2000.

[7] Xerox PARC. AspectJ home page. `http://aspectj.org`. Accessed on January 2, 2001.

A AspectJ solution domain analysis

This is an attempt to apply SCV analysis to AspectJ solution domain (without applying the feature modeling). The results are incomplete.

```
S: o        a collection of objects of kind o
C: c_1;     the list of commonalities that hold for objects in S
   c_2;
   ...
   c_m
V: v_1;     the list of variabilities among the objects in S
   v_2;
   ...
   v_m
B:          binding
I:          instantiation
```

```
Java:

Methods

     S: code fragments
     C: the common code (placed in the procedure)
     V: the "uncommon" code (regulated by parameters or placed before or after a specific method call)


Classes

     S: data structures, procedures
     C: common data structure;
      procedure signatures;
        procedures that operate on the same data structures (methods)
     V: the state of data structures


Interfaces

     S: data structure declarations, procedure signatures
     C: common data structure declarations;
        signatures of procedures that operate on the same data structures (methods)
     V: the implementation


Class Inheritance

     S: classes
     C: the common code (placed in the base class)
     V: the "uncommon" code (placed in the subclasses)


Interface Inheritance

     S: interfaces
     C: the common declarations and method signatures (placed in the base interface)
     V: the "uncommon" declarations and method signatures (placed in the subinterfaces)


Overloading

     S: methods of a class or classes in an inheritance hierarchy
     C: name and return type
     V: algorithm and signature



AspectJ:

Introductions

     S: code fragments repeated throughout a set of classes
     C: common code fragments (to be lexically introduced into classes)
     V: affected classes
     B: compile time


Static Advices

     S: similar actions performed with respect to some other actions
     C: the common code among the similar actions
     V: the "uncommon" code
     B: source time


Dynamic Advices

     S: similar actions performed with respect to some other actions
     C: the common code among the similar actions
     V: the "uncommon" code
     B: run time
```

B Paradigms

Definition 1 *Language construct (language mechanism). A language construct (language mechanism) is the smallest semantically indivisible syntactic element of the language.*

Definition 2 *Abstract small-scale paradigm. An abstract small-scale paradigm is a commonality-variability pairing.*

Definition 3 *Small-scale paradigm. A (concrete) small-scale paradigm is a pair of a language mechanism and the corresponding abstract small-scale paradigm.*

Definition 4 *Large-scale paradigm. A large-scale paradigm is a consistent set of the small-scale paradigms.*

Definition 5 *Constitutive small-scale paradigms. A large-scale paradigm is characterized by a set of the small-scale paradigms. These small-scale paradigms are called constitutive small-scale paradigms of a given large-scale paradigm and the set is called a constitutive set.*

The constitutive set of a given large-scale paradigm can be enriched with other small-scale paradigms while preserving its character. On the other hand, dropping out any of the constitutive small-scale paradigms leads to a principal change of the large-scale paradigm.

Some small-scale paradigms are considered as very simple (trivial). Therefore they can be omitted from the MPD.