Slovak University of Technology in Bratislava
Faculty of Electrical Engineering and Information Technology
Department of Computer Science and Engineering

Valentino Vranić

Towards Multi-Paradigm Software Development

Bratislava, September 2000

Abstract

Multi-paradigm software development is a spontaneous answer to attempts of finding the best paradigm. It was present in software development at the level of intuition and practiced as the "implementation detail" without even mentioning it in the design. Its breakthrough is twofold: several recent programming paradigms are encouraging it, while explicit multi-paradigm approaches aim at its full-scale support.

However, to reach this goal, multi-paradigm approach must be improved and refined.

Contents

Chapter 1

Introduction

The way of software development is changing. Enforced by the need for mass production of quality software and enabled by the grown-up experience of the field, it is moving towards the industrialization.

This report maps the state-of-the-art in the field of post-object-oriented software engineering; most notably, it is dedicated to the promising concepts of aspect-oriented programming, generative programming and, particularly, multi-paradigm software development.

This tendency can be felt not only in the new software development paradigms, i.e. aspect-oriented programming, which is bound to the existing paradigms; it is present already in the object-oriented programming. It is even more notable at language level. It's hard to find a language that is pure in the sense of prohibiting any other than its proclaimed (main) paradigm from being used in it. This is the implicit form of what is called multi-paradigm.

There are several approaches, which make this idea of multi-paradigm explicit by enabling the developer not only to combine multiple paradigms, but also to choose the most appropriate one for the given feature. This paradigm of paradigms is sometimes denoted as *metaparadigm*.

The structure of the rest of this report is as follows.

Chapter 2 explores the concept of *paradigm* in computer science and software engineering.

Chapter 3 is an overview of some recent post-object-oriented paradigms, namely aspect-oriented programming approaches and generative programming.

Chapter 4 proceeds with further recent post-object-oriented approaches. These are presented in a separate chapter because they exhibit an explicit multi-paradigm character.

Chapter 5 closes this report. It includes conclusions and proposals for further work.

Chapter 2

The Concept of Paradigm in Software Development

The paradigm is a very often used (but even more often abused) word in computer science and software engineering today. Its importance arose especially with appearance of so-called *multi-paradigm* approaches (which are discussed in Chapter 4). Before discussing them, the concept of paradigm in software development requires a deeper examination.

However, before we go into this specific analysis, it would be useful to consider a paradigm in a general sense; that is, the word *paradigm* itself. The meaning of the word *paradigm* is analyzed in Section 2.1. In Section 2.2 its common usage to denote a software development process as a whole is discussed. Section 2.3 explores further the concept of paradigm in software development revealing another level at which paradigms can be considered.

2.1 The Meaning of Paradigm

The term *paradigm* in science generally is strongly related to Kuhn and his work [Kuh97]. Although not explicitly defined in this essay, it leaves reader with understanding of paradigm similar to this quoted from [BN97]:

> A paradigm in general is a body of ways of formulating problems, methodological tools of their solution, standard methodologies of their elaboration. It is opinions, theories, methods, methodologies etc., which are accepted in a given field.

As Kuhn discloses in the supplementary material published as a part of the book in later editions, a certain reader after analyzing Kuhn's essay, concluded that the term *paradigm* is used there in at least twenty two different ways. Fortunately, most of the differences were stylistic and could be resolved. But, even so, the two incompatible meanings remained: paradigm as a constellation of groups' belief and paradigm as a shared model example. We will later see that this duality is not accidental and that it has its roots in the meaning of the word *paradigm.*

Probably no science has accepted this term with such enthusiasm as computer science did. In computer science (and software engineering) the term *paradigm* is used to denote the essence of the software development process (often reduced to just *programming*), which appears to be one of its key issues. Unfortunately, the term *paradigm* is used so often that hardly you can find a methodology or a method (or even just an improvement of the method) today that has resisted the temptation to "become" a paradigm. This abusing of the word *paradigm* introduces a confusion about its real meaning. This is why we'll take a brief look at the meaning of the word *paradigm*.

The Merriam-Webster Dictionary gives a following definition of the word *paradigm*:

1. example, pattern; especially: an outstandingly clear or typical example or archetype
2. an example of conjugation or declension showing a word in all its inflectional forms
3. a philosophical and theoretical framework of scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated.

Etymologically, *paradigm* comes from Late Latin *paradigma*, which comes form Greek *paradeigma*; this comes from *paradeiknynai* meaning "to show side by side". Closest to this original meaning is the first meaning, i.e. example or pattern. This is the most general meaning of the three meanings from the dictionary. It makes no restriction regarding the size of the example or the pattern denoted by the word *paradigm*.

The second meaning shows how the word *paradigm* found its realization in the micro-context—at language construct level. The third meaning obviously denotes something big and complex; this the realization of the word *paradigm* in the macro-context—as a framework of a discipline (this is actually a kind of Kuhn's definition of the term).

This duality of the term *paradigm* is present in the software development, too, as it will be shown in the next two sections.

2.2 Large-Scale Paradigms

The notion of paradigm in software development is used at two levels of granularity, and this comes as no surprise after the previous section. The first one, *large-scale*[1] level, is the one we usually mean when speaking of software development paradigms in a traditional sense.

This large-scale meaning of the term *software development paradigm* (or, more often, simply *paradigm*), denotes the essence of certain software development process. The name of a paradigm reveals the most significant characteristic of the paradigm. Sometimes, it is derived from a central abstraction the paradigm deals with, as it is a function to functional paradigm, an object to object-oriented paradigm[2] etc.

In spite of the fact that software development paradigm refers to all the phases of the software development process, not only to implementation, in place of a term *software development paradigm* often we can find a term *programming paradigm* or even just *programming* (e.g. object oriented programming, OOP). On the other hand, in order to be more explicit, expression *OO analysis and design* (OOA/D) can be used to refer to the analysis and design phases of OO software development process, and OOP to refer specifically to its implementation phase.

[1]Coplien used this term to denote programming paradigms in, as he said, a "popular" sense of the term [Cop99b].

[2]This is not so clear. For example, according to [Mey97] it is not *object* but *class* that is a central abstraction in the OOP.

Paradigm	*Main abstraction*
Imperative	command
Procedural	procedure
Object-oriented	object/class
Functional	function
Logic	expression

Table 2.1: Paradigms and their main abstractions

When speaking of software development paradigms, it must be distinguished between the concept of paradigm and the means that are used to support its realization. Any paradigm can be visualized by means of a visual environment and thus it makes no sense to speak of a visual paradigm as an independent paradigm. Otherwise we should consider syntax highlighting as a paradigm, too. Unfortunately, as it was already pointed out, this abuse of the word appears to be a source of confusion. So, for example, in [Bud95] the visual paradigm is mentioned with the observation that it is actually "a family of paradigms". This is correct only if we accept that all the paradigms are members of one (big) family. Well, yes, they are, but does this classification makes sense? Of course, this is not to say that it is not useful to group paradigms according to common features.

Making a complete classification and comparison of the software development paradigms is beyond the scope of this text; Návrat in [Náv96] compares selected programming paradigms regarding abstraction and generalization. Thus, Table 2.1 shows only five (well-known) paradigms and the main abstraction of each.

Programming language is often classified according to the paradigm it supports; so, among others, procedural, object-oriented and functional languages exist. However, this does not mean that language is incapable of supporting some other paradigm (e.g. C++). A programming language must not be confused with the paradigm it supports. Programming language can be seen as a vehicle for the application of a paradigm.

Software development paradigm is constantly changing, improving, or better to say refining. Basic principles it lays on must be preserved; otherwise it would change into another paradigm. So, it can be said that paradigms are at different levels of maturity.

As this report is concerned with post-object-oriented software development, let's consider the object-oriented paradigm and its predecessors depicted in Fig. 2.1. The arrows represent "evolved into" relationship. This is what makes these paradigms closer to each other than, say, object-oriented and logic paradigm. A simplified view of this paradigm evolution goes like this. First, there were commands (imperative programming). Then, named groups of commands appeared: procedures (procedural programming). Finally, procedures were put together with the data it operated on: classes/objects (object-oriented programming).

However, according to Kuhn, paradigms do not evolve, although it can seem so. He speaks of the *scientific revolution* which ends up the old and starts a new paradigm [Kuh97]. A paradigm is *dominant* by definition and thus there can be only one paradigm at a time in a given field. This is a contradiction with the existence of five or more software development paradigms indicating that the field is either in the unstable state, either all these paradigms are part of one big, but unrecognized paradigm standing above them: *metaparadigm*.

A software development paradigm, or a large-scale paradigm, as denoted in this chap-

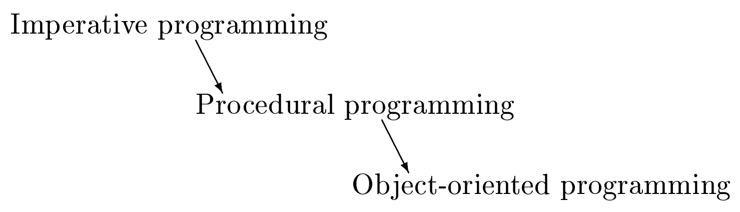

Figure 2.1: The evolution line of OOP.

ter, seem to be somehow an elusive concept, in the sense that it's hard to define it precisely. Another look at paradigms is offered in the next section.

2.3 Small-Scale Paradigms

There is another possibility to define programming paradigm. It comes out from its use to denote mechanisms of the programming languages. This is similar to paradigms in natural languages where (as we saw in Section 2.1) paradigm denotes an example of conjugation or declension showing a word in all its inflectional forms. These paradigms seem to be somehow "smaller", so we will refer to them as *small-scale* paradigms.

This perception of paradigm is apparent in Coplien's multi-paradigm design [Cop99c] (which is discussed in Section 4.2). According to Coplien et al. [CHW98], we can factor out paradigms such as procedures, inheritance and class templates. We can identify a common and a variable part, which together constitute a paradigm. This is analogous to conjugation or declension in natural languages, where the common is the root of the word and variability is expressed through the suffixes or prefixes (or even infixes), which must be added to obtain different forms of the word.

Scope, commonality and variability (SCV) analysis can be used to describe paradigms at language level, as it is presented in [CHW98]. Keywords of SCV analysis have the following meanings:

Scope (S): a set of entities[3]

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

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

For example, *procedures* paradigm according to SCV analysis looks like this (an example adapted from [CHW98]):

- S: a collection of similar code fragments, each to be replaced by a call to some new procedure P
- C: the code common to all fragments in S
- V: the "uncommon" code in S; variabilities are handled by parameters to P or custom code before or after each call to P.

[3]Instead of *entities* in [CHW98] the word *objects* was used. This could lead to misunderstanding because of OOP.

SCV analysis is not limited to description of the paradigms—it is of wider usability and importance, especially in the context of the multi-paradigm design (see Section 4.2).

If we take paradigms the way they are described in this section, then programming language that supports only one paradigm is more an exception than a rule. So, programming language can support, by the means of the language constructs, one or more programming paradigms. On the other hand, programming paradigm can, of course, be supported by several programming languages.

The relationship between small- and large-scale paradigms is similar to that between small-scale paradigms and programming languages; large-scale paradigms consist of small-scale ones. The name of the large-scale paradigm sometimes comes from the most significant small-scale paradigm it contains. For example, object-oriented (large-scale) paradigm consists of the several (small-scale) paradigms:[4] object paradigm, procedure paradigm,[5] virtual functions, polymorphism, overloading, inheritance etc.

Having a richly expressive programming language that supports multiple paradigms introduces another issue: a decision must be made which paradigm is appropriate for which feature to be implemented. That means we need a method for choosing paradigms that is above them, i.e. *metaparadigm* (a particular metaparadigm—multiparadigm design for C++—is described in Section 4.2).

2.4 Summary

In this chapter, starting from the general meaning of the word *paradigm*, we came to its specific use regarding software development. Two levels of its use were identified and briefly described: large-scale and small-scale.

Regarding the relationship between these two concepts, one more thing requires to be clarified. One could understand small-scale paradigms as a programming language issue only, while large-scale programming paradigms seem to be broader in scope as they are affecting all the phases of the software development. Actually, the small-scale paradigms have an impact on all the phases of the software development as well; either without formal support in the development process, or with it (as it is the case in Coplien's multi-paradigm design for C++, see 4.2).

Programming paradigm, as a concept, requires further investigation in order to gather a more precise understanding of both large- and small-scale paradigms. However, since this is beyond the scope of this report, and since a substantial level of understanding of the concept has already been achieved, we shall proceed with the analysis of some recent post-object-oriented paradigms.

[4]Lack of a common agreement what are the exact characteristics of the object-oriented paradigm makes impossible to introduce an exact list of the small-scale paradigms out of which object-oriented paradigm consists.

[5]Procedure paradigm is present through class methods.

Chapter 3

Recent Software Development Paradigms

Among the recent software development paradigms there is a significant group of those that appeared as a reaction to the issues tackled but not satisfactorily solved by the object-oriented programming.

Many of these paradigms actually build upon object-oriented paradigm. In spite of that some of them are claimed not to be bound to object-oriented paradigm (and in deed they are more generally applicable), they are still widely applied in connection with object-oriented programming (not accidentally, as we shall see).

In this chapter, several such post-object-oriented software development paradigms are discussed. However, the first section is a short excursion to the object-oriented programming because of its importance for the paradigms presented briefly in the following three sections.

3.1 Beyond Object-Oriented Programming

Human perception of the world is to a great extent based on objects. From our earliest days we encounter objects around ourselves, we find out their behavior, i.e. their properties and what we can do with them. Object-oriented paradigm is based precisely on this perception of the world natural to humans.

What exactly is the object-oriented programming? This question seems to be an answered one. Actually, there is a plenty of answers to this question, but the trouble is that they are all different (for possible reasons why is it so see [Cop96]).

The object-oriented programming (OOP) has passed a very long way of changes to reach the form in which it is known today. Yet, there is no general agreement on the definition of the essential properties of the object-oriented paradigm (to some, even inheritance is not an essential part of the object-orientation, or it is being denoted as a minor feature [Bud95]).

Booch, for example, makes difference between *major* and *minor* elements of the object model, which is "the conceptual framework for all things object-oriented" [Boo94]. The major elements are: abstraction, encapsulation, modularity and hierarchy. The minor elements, i.e. those not essential, are: typing, concurrency and persistence.

Meyer is more specific. He identifies a few dozens of criteria for object-orientation grouped in the three categories [Mey97]: method and language, implementation and environment, and libraries.

Booch's elements of object-orientation cover the first Meyer's category, method and language, and they are not in contradiction with any Meyer's criterion from this category, but they are not so restrictive. For example, according to Meyer's criterion *classes as types*, the type is modeled by a class, while Booch does not make such an explicit restriction. On the other hand, as Meyer says, "'object-oriented' is not a boolean condition"; something can be object-oriented only to some extent.

OOP is not always the best choice among all the paradigms. This is recognized even in the OOP literature. Thus Booch points out that there is no single paradigm best for all kinds of applications. But OOP has another important feature: it can serve well as "the architectural framework in which other paradigms are employed" [Boo94]. This reveals that OOP is multi-paradigmatic in its very nature and doesn't leave much space for the object-oriented purism.

This object-oriented purism comes from the dogma that everything should be modeled as an object. Thus, in the "pure" OOP we are taught to see everything as an object, but not everything is an object; neither in a real world, nor in programming. Synchronization is a well-known example of a non-object concept. In natural language, we would probably refer to it as an *aspect.* The aspects crosscut the structure of objects, or (i.e. functional components, in general), which makes the code tangled. The pieces of code are either repeated throughout different objects or unnatural inheritance (often multiple one)[1] must be involved. Among other, this "code scattering" has a bad impact on reuse.

There are other problems with OOP, concerning issues it was proposed to solve, mainly in the areas of reuse,[2] adaptability, management of complexity and performance [Cza98].

There is one more reason against the OOP as the best paradigm regarding the concept of paradigm (as discussed in Chapter 2). A paradigm must be universal in its field (or, at least, to be seen as such). Is OOP a universal paradigm in software engineering? To simplify the problem, let's consider just C++ as a part of the field of software engineering. So, is OOP a universal paradigm in C++? The answer that it's not because C++ provides non-object-oriented features and, moreover, enables to program in a completely non-object-oriented fashion. So, if OOP isn't the universal paradigm in C++, which is just a part of software engineering field, how can it be universal in the software engineering as a whole?

3.2 Aspect-Oriented Programming and Related Approaches

According to one of those who stood upon the birth of the aspect-oriented programming, Gregor Kiczales [KLM+97], *aspect-oriented programming (AOP) is a new programming paradigm*[3] *that enables the modularization of crosscutting concerns.*

Xerox PARC AOP group [Xer] is the integrating force in AOP. The name *aspect-oriented programming* was actually invented by them. Of course, the other groups doing AOP research are of no less importance. In fact, AOP ideas materialized in several places independently and, as soon as this was discovered, the collaboration among various groups and individuals working on AOP begun. Yet it is not clear will this process lead us towards unification of AOP approaches or will these AOP techniques grow up to loosely coupled, yet different, paradigms and thus preserving AOP's multi-paradigm character.

[1]This is not to claim that multiple inheritance is unnatural in general.

[2]Software reuse not only in the context of OOP is discussed in [SN97].

[3]Kiczales denotes it as *methodology.*

A list of groups and individuals doing AOP research is maintained by Xerox PARC AOP group and it is growing (a complete list is available at Xerox PARC AOP home page [Xer]). We'll take a closer look at four AOP techniques, which constitute the basis of AOP research until now and thus are of great importance for further AOP development:

- Kiczales et al. at Xerox PARC: AOP, AspectJ [Xer]
- Lieberherr et al. at Northeastern University: adaptive programming (AP) [Dem]
- Aksit et al. at the University of Twente: composition filters (CF) [TRE]
- Ossher et al. at IBM Research: subject-oriented programming (SOP) [IBM]

As it is usual with industrial methodologies (as opposed to formal ones), the focus in AOP research has been on the implementation phase. Thus all of the approaches mentioned are actually AOP implementation techniques. This means that there is an open field of establishing the AO analysis and design methodology in order to complete aspect-oriented development process.

AP and CF have been recently redefined by their inventors with respect to AOP as special cases of it (see Sections 3.2.2 and 3.2.3). This is not the case with SOP and there is no common agreement whether SOP is AOP or not (see Section 3.2.4).

Yet another questionable issue is whether AOP is OOP bound or not. A paradox is that although AOP techniques listed build upon OOP, the very idea of the AOP is not limited to it. This is because aspects tend to crosscut functional units of the system (referred to as *generalized procedures* in AOP papers [KLM$^+$97]), i.e. this problem arises in non-object-oriented systems as well.

3.2.1 Aspect-Oriented Programming

As mentioned before, Xerox PARC group gave name to AOP. Actually, most of the AOP terminology (like *aspect*, *crosscutting*, *tangling*, *weaving*) adopted later by others was invented by them. Most of research effort is being concentrated on AspectJ, a general purpose AOP extension to Java [LK98].

The idea of Xerox PARC AOP is best presented by an example. In Fig. 3.1 two classes are presented, `Point` and `Line`, with three kinds of methods: creating, writing and reading (implementations are not shown). Suppose we want to be warned by a text on the screen what kind of access to these classes has been performed. In ordinary Java we would have to modify each method of both `Point` and `Line`. This would result in what is known as *tangled* code. To avoid this, in AspectJ we can use aspects. In our example it is the aspect `ShowAccesses` that solves the problem. Note that the original code remains unchanged. Before running the ordinary Java compiler, so-called *weaver* must be used, which would weave the aspect into the code.

The solution incorporating aspects is undoubtedly more elegant than the tangled one, but consider again the Fig. 3.1. The information of where aspect is to be woven, known as *join-points*, is included in the aspect itself, which complicates the reuse of aspects.

3.2.2 Adaptive Programming

The adaptive programming (AP) proposed by Demeter group [Dem] at Northeastern University in Boston deals mainly with traversal strategies of class diagrams.

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

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

aspect ShowAccesses {
  static before void Point.set(*), void Line.set(*), void Line.set(){
    System.out.println("Write");
  }
  static before int Point.getX(), int Point.getY(),
    int Line.getX1(), int Line.getY1(), int Line.getX2(), int Line.getY2() {
    System.out.println("Read");
  }
  static before Point(*), Line(*) {
    System.out.println("Create");
  }
}
```

Figure 3.1: An AspectJ example (based on example from [LK98])

The Demeter group has used AOP ideas for several years before the name aspect-oriented programming was coined. The collaboration with the Xerox PARC AOP group then begun and Demeter group redefined the AP as the special case of AOP where one of the aspects is expressible in terms of graphs and where the other aspects or components refer to the graphs using *traversal strategies*. The traversal strategies are partial specifications of a graph pointing out a few cornerstone nodes and edges and thus crosscut the graphs they are intended for while only mentioning a few isolated nodes and edges.

For example, assume we have a UML class diagram of a system as presented in the left part of Fig. 3.2. Assume we would like to count on the people waiting at the bus stations all along the bus route. Clearly, in ordinary OOP, this would require either implementation of small methods in all of the affected classes (shaded ones) or rough breaking of the encapsulation principle by exposing some of the private data of the classes.

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

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

solves the problem of getting to `Person` objects along the bus route, which is sufficient to count them.

The right part of Fig. 3.2 demonstrates the robustness of this technique—the traversal strategy mentioned above applies in this case without any change although a class diagram it was constructed for changed.

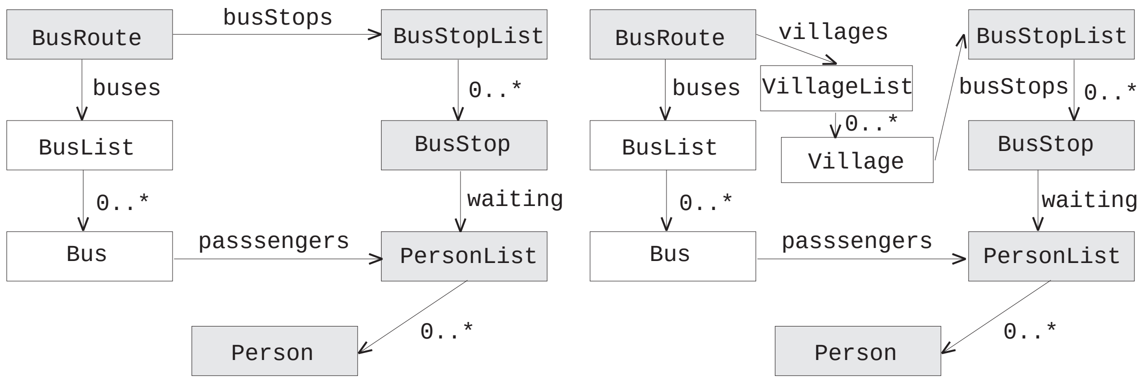

Figure 3.2: Traversal strategies (from [Lie97])

3.2.3 Composition Filters

Composition filters is an aspect-oriented programming technique where different aspects are expressed as declarative and orthogonal message transformation specifications called *filters* [AT98].

A message sent to an object is evaluated and manipulated by the filters of that object, which are defined in an ordered set, until it is discarded or dispatched (i.e. activated or delegated to another object).

The filter behavior is simple: each filter can either accept or reject the received message, but the semantics associated with these depend on the filter type; e.g. if an `Error` type filter accepts the received message, it is forwarded to the next filter, but if it was a `Dispatch` type filter, the message would be executed. Detailed description of the composition filters can be found in [AWB$^+$93, Koo95].

In Fig. 3.3 two sets of filters (written in Sina language [Koo95], which directly adopts the CF model [AT98, AWB$^+$93]) attached to the `Point` and `Line` classes from Fig. 3.1 respectively are shown. We assume the existence of the class `ShowAccess` with three methods: `WriteAccess`, `ReadAccess` and `CreateAccess` (the instance `acc` of this class is used in filters). These methods simply write out one of three possible messages about the type of the access. They are called by three corresponding `Dispatch` filters, in case the message was accepted. Afterwards, the method of the inner object, which has been called, is executed (`inner.*`).

If we consider this example in the AOP terminology, then the class `ShowAccess` actually implements the aspect, while filters represent the join points. Thus, the join points in this case are separated from the aspect, which is better regarding the aspect reuse.

3.2.4 Subject-Oriented Programming

We define a certain object, or more generally a concept, by its properties. This is sufficient to precisely define and identify mathematical concepts, but the same does not apply to natural concepts because their definitions are *subjective* and thus never complete (more details about conceptual modeling can be found in [Cza98]).

Subject-oriented programming is based on subjective views, so-called *subjects*. SOP is being developed at IBM (see [IBM]). It was proposed as an extension of the OOP and thus subject is a collection of classes or class fragments whose hierarchy models its domain in its own, subjective way. A complete software system is then composed out of subjects

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

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

Figure 3.3: A filter attaching example

by writing the *composition rules*, which specify the correspondence of the subjects (i.e. namespaces), classes and members to be composed and how to combine them.

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

The example from Fig. 3.1 reimplemented in Watson Subject Compiler-like syntax[4] is presented in Fig. 3.4. We assume that class `ShowAccess` is implemented in `Access` namespace and that classes `Point` and `Line` are implemented in `Graphics` namespace. In this case the join-points, represented by the composition rules, are separated from the aspect, which is represented by the separate class, as it was the case in CF approach, too. Composition rules for the classes `getY`, `getX1`, `getY1` and `getX2` are omitted in Fig. 3.3 (indicated by ellipsis) since they are analogous to the rule for `getX`.

This is not a characteristic case of the application of SOP (such as can be found in [OHBS94, KOHK96, IBM]); it is presented here in order to show how a well-known AOP example can be easily transformed into its SOP version. Nevertheless, there is no general agreement whether SOP is AOP. Czarnecki [Cza98] views SOP as a special case of AOP where the aspects according to which the system is being decomposed are chosen in such a manner that they represent different, subjective views of the system. On the other hand, Kiczales et al. [KLM$^+$97] reject the very idea that SOP (which they call *subjective programming*) could be AOP, arguing that the methods involved in automatic combination of methods for a given message from different subjects supported in SOP are components in the AOP sense since they can be well localized in a generalized procedure (routine). But this seem to be a more general issue, since it applies to AspectJ too, named *aspectual paradox* by Liebrherr et al. [LLM99]: “an aspect described in AspectJ, the Xerox PARC's AOP language, which has a construct for specifying aspects, is by definition no longer an aspect, as it has just been captured in a (new kind of) generalized routine”.

It is worth mentioning, as Czarnecki [Cza98] observed, that SOP is close to *GenVoca* approach [Bat99, BG97], where the systems are composed out of *layers* according to the *design rules* (for further information on this topic see [PLA]): GenVoca layers can be easily

[4]According to the information available in the papers regarding SOP, a composition presented is regular, although the actual syntax could by slightly different.

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

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

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

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

Figure 3.4: An example of the subject composition

simulated by subjects, which brings into connection AOP with GenVoca as a successful approach to reusability [Bat99]. Of course, this does not mean that we can assume that AOP is a practically proven technology; however, it speaks in favor of AOP.

3.3 Generative Programming

In his Ph.D. thesis, Czarnecki [Cza98] (and recently also in the book, which he wrote together with Eisenecker [CE00]) proposes a comprehensive software development paradigm, which brings together the object-oriented analysis and design methods with domain engineering methods that enable development of the families of systems: generative programming.

The definition introduced in [CE00] reads:

> Generative programming (GP) is a software engineering paradigm based on modeling software systems families such that, given a particular requirements specification, a highly customized and optimized intermediate or end-product can be automatically manufactured on demand from elementary, reusable implementation components by means of configuration knowledge.

GP is a unifying paradigm—it is closely related to object-oriented programming and three other paradigms (see Figure 3.5):

- object-oriented programming, providing effective system modeling techniques
- generic programming, which can be summarized as "reuse through parameterization"
- domain-specific languages, which increase the abstraction level for a particular domain and are highly intentional, and
- aspect-oriented programming, used to achieve separation of concerns.

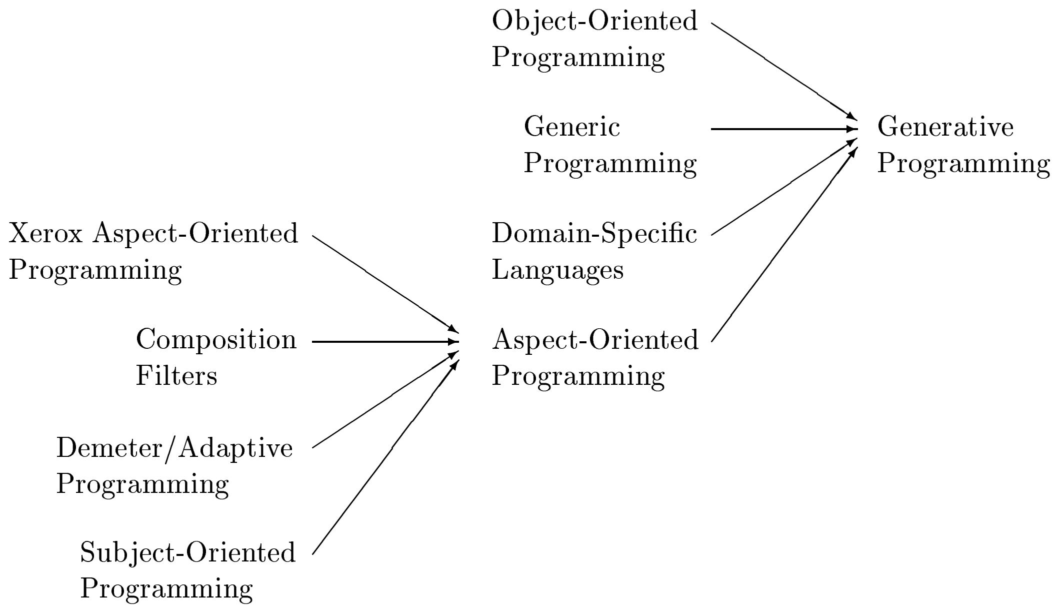
Object-Oriented Programming
Generic Programming
Generative Programming
Xerox Aspect-Oriented Programming
Domain-Specific Languages
Composition Filters
Aspect-Oriented Programming
Demeter/Adaptive Programming
Subject-Oriented Programming

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

Object-oriented programming is present in GP indirectly as well (not depicted)—through aspect-oriented programming approaches, which (although not object-oriented bound) actually build upon OOP (Section 3.2).

GP first has to be tailored to a particular domain in order to be used. This process will give us a methodology for the families of systems to be developed, which can be viewed as a paradigm itself. This gives a certain metaparadigm flavor to GP.

In the implementation field, GP requires metaprogramming for so-called *weaving* (i.e. joining the aspect part of the code with the functional one) and automatic configuration. To support domain-specific notations, it needs syntactic extensions. Czarnecki proposes *active libraries* as appropriate to cover this requirement. Active libraries, which can be viewed as *knowledgeable agents*[5] interacting with each other to produce concrete components, require appropriate programming environment.

3.4 Summary

Post-object-oriented paradigms presented here carry out a latent multi-paradigm idea with them. This is not strange since this idea can be identified already in their predecessor—object-oriented paradigm.

These paradigms are not object-oriented bound, but they fit well with object-oriented programming. Actually, the very fact that they arose in dominance of object-orientation in software development doesn't seem to be an accident.

Also, certain unifying tendencies have been identified among the paradigms described. This is especially apparent with generative programming, but the possibility of unifying the AOP approaches (probably adapted to suite the common shell), too. In the examples presented, we saw that some of these techniques can be applied interchangeably, namely

[5]It would be useful to consider some agent-oriented programming [Sho93] techniques here.

Xerox PARC AOP, CF and SOP, with no substantial difference. Traversal strategies in AP aim at different issues, but they are not in contradiction with other AOP techniques. However, the examples presented do not imply the interchangeability of the AOP techniques in a general case and a further investigation is required.

An important characteristic of the AOP approaches and GP is that they don't aim at pushing out any other approach from the scene but, on the contrary, seek for the best way to incorporate it. A further step from this partially hidden multi-paradigm nature of the described approaches is to reveal it completely and express it explicitly. Such approaches will be discussed in the next chapter.

Chapter 4

Multi-Paradigm Approaches

In the survey of the recent post-object-oriented software development paradigms given in the previous chapter a spontaneous move towards paradigms' integration became apparent. This chapter is a survey of several approaches that make this move towards multi-paradigm explicit.

One possible approach is to create a new language in such a manner that it would support multiple paradigms. This approach is demonstrated in section 4.1 on Budd's *multi-paradigm programming in Leda*. The other way is to determine the rules of selecting the paradigms for solving particular issues when richly expressive (i.e. supporting multiple paradigms) programming language is available. This is explored in section 4.2, which describes Coplien's *multi-paradigm design for C++*. However, each of the two has its shortcomes. This makes a place for the third one, Microsoft's *intentional programming*, briefly presented in section 4.3.

4.1 Multi-Paradigm Programming in Leda

The question how to support multi-paradigm programming at language level yields a simple answer: create a multi-paradigm language. Budd took this route towards multi-paradigm programming by creating a multi-paradigm language called Leda [Bud95].

According to Budd, Leda language supports four programming paradigms: imperative,[1] logic, functional and object-oriented. The term *paradigm* as used by Budd denotes a large-scale paradigm (with respect to classification of paradigms introduced in Chapter 2). This means that Leda actually supports more than four small-scale paradigms. This is clear if we remember that, for example, object-oriented paradigm breaks down into several small-scale paradigms (Section 4.2). Nevertheless, for simplicity, we will discuss just the mechanisms by which each of the four proclaimed paradigms is supported.

Leda has a Pascal-like (i.e. Algol-like) syntax and, moreover, the mechanism upon which all four supported paradigms realization is based in Leda are functions.[2] This makes a good background for the *imperative* (procedural) paradigm.

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

[1]Actually procedural, to be more precise.

[2]Meaning procedures returning values.

The *functional paradigm* requires no special mechanism than that provided by functions, i.e. procedures returning a value, since Leda permits a function to be an argument to the other function or to return a function as a result. Thus, when programming in Leda, a functional paradigm is achieved using the functions in a recursive fashion while refraining from assignments.

The *object-oriented* paradigm is supported similarly like in the C++ or Object Pascal. In addition to basic mechanisms of object-oriented paradigm, such as classes, inheritance, encapsulation etc., Leda supports *parameterized types* (by some authors also considered as a part of object-oriented paradigm, e.g. [Mey97]).

In spite of its limited use, Leda language is interesting because it demonstrates the combination of paradigms. For example, the inference mechanism of logic programming can be used inside of a procedure.

Of course, creating a language that supports multiple paradigms and expecting it would be the best language to program in is similar to a hunt on the best programming paradigm. Despite the number of supported paradigms in a programming language, that number is finite; the paradigms that would appear after the establishment of that language would not be included. One can argue that it is possible to extend the language with new programming mechanisms in order to support new paradigms. This is, indeed, possible and often practiced. Unfortunately, programming languages cannot be extended indefinitely due to limitations set by parsing methods.

Leda is an example of a language *created* (from scratch) in order to support multiple paradigms. However, we can consider interconnecting existing languages that support different paradigms through an interface instead of making a completely new language (a sort of language reuse). There is also a possibility of implementing one language on top of the other, but this leads to a certain degradation of performance. More on this topic and also an example of interconnecting object-oriented and logic programming (Loops and Xerox Quintus Prolog) can be found in [KE88].

4.2 Multi-Paradigm Design for C++

Multi-paradigm design for C++ (MPD), as proposed by Coplien [Cop99b, Cop99a, Cop99c], has its roots in multi-paradigm characteristics of C++. Despite these multi-paradigm characteristics, C++ is often considered to be only an object-oriented language. As such, C++ is used to implement the systems designed according to object-oriented paradigm. However, non-object-oriented features of C++ are widely used, but without their "legalization" in design.

Coplien proposes a particular metaparadigm intended for developing families of systems, which enables choosing the appropriate paradigm for the feature that has to be designed and implemented. It is based on the SCV (scope, commonality and variability) analysis mentioned in Section 2.3.

In his work regarding multi-paradigm design (cited above), Coplien abbreviates the name *scope, commonality and variability analysis* to just *commonality and variability analysis* and separates the two thus achieving two distinct analyses—*commonality analysis* and *variability analysis*. Despite this formal distinction, the two analyses are performed in parallel. Commonality analysis concentrates on common attributes while the aim of the variability analysis is to parameterize the variation. This process yields commonality/variability pairings. Any such commonality/variability pairing represents a programming paradigm—in the sense of the small-scale paradigms (discussed in Section 2.3).

The two analyses are performed on both application and solution domain independently and then the commonalities and variabilities of the application and solution domain analyses are lined up, leading to use of the "right" paradigms supported by the language for the corresponding analysis abstractions.

Although, in Coplien's own words, "MPD is a craft that is neither fully an art nor fully a rigorous discipline" [Cop99c, p. xv], and to great extent relies on designer's intuition and experience, it is a move towards greater regularization of the application of multiple paradigms in software development. Also, it has to be pointed out that despite it is called just *design*, MPD is a complete software development process resulting into a program implementation.

The major steps performed during the MPD are:

- commonality and variability analysis of the application domain
- commonality and variability analysis of the solution domain
- transformational analysis
- translation from the transformational analysis to the code.

These steps need not to be performed sequentially.[3] They can be performed in parallel and revisited as needed. Before starting the actual MPD, it is recommended to evaluate a possibility of the existing designs' reuse. Also, it is recommended to consider the use of the application-oriented languages. Coplien proposes this paradoxically as the last step. However, there is no point in doing an analysis of the solution domain that is not going to be used. Logically, the best time to choose the implementation language is after (possibly during) the commonality and variability analysis of the application domain (if we are not limited to a specific programming language according to requirements).

Commonality analysis of the application domain begins with finding *commonality domains* and creating *domain dictionary*. It then proceeds in parallel with the variability analysis. The results of the analysis —the *parameters of variation* for a given commonality domain and their characteristics— are summarized in the *variability tables* consisting of the following columns:

- parameters of variation
- meaning (the decision being made)
- domain
- binding (binding time)[4]
- default.

A parameter of variation can be a domain itself. To capture this relationship *variability dependency graphs*[5] are used. The notation of variability dependency graphs is quite simple: the domains are depicted as ellipses, and the arrows point from the domain

[3]However, it is not possible to perform the transformational analysis without having finished at least a part of the commonality and variability analyses of the application and solution domain.

[4]Describes how early the value of the parameter of variation is to be selected. The alternatives for C++ (in ascending order) are: source, compile, link (and load) and run.

[5]Coplien sometimes calls them *domain dependency graphs*.

to its parameters of variation. The variability dependency graphs are used to identify overlapping domains (which can be merged). Also, they help to identify *codependent domains*—domains with circular dependency (which must be resolved).

Commonality and variability analysis of the solution domain begins with the identification of supported paradigms, which is actually a kind of the SCV analysis (see Section 2.3). It results into an informal description of the identified paradigms structured as follows:

- commonality
- variability
- binding
- example.

It proceeds with exploring the *negative variability*—such a variability that violates the rule of variation by attacking the underlying commonality. A *positive variability*, as opposed to the negative one, is such a variability that can be parameterized. The negative variability has to be kept small. If it becomes larger than the commonality, the design should be refactored to reverse the commonality and variability.

The results of the commonality and variability analysis of the solution domain are being summarized in the two types of tables. One type is used to express *features for negative variability* and consists of the following columns:

- kind of commonality
- kind of variability
- language feature for positive variability
- language feature for corresponding negative variability.

The other type of the table, denoted as *family table*, expresses commonality and positive variability pairings in the domain of the programming language (that is being used) and contains the following columns:

- commonality
- variability
- binding
- instantiation
- language mechanism.

The tables obtained in the preceding analysis are used during the *transformational analysis*. The variability table is aligned with the family table to see which language feature is suitable for which part of design. This results into an annotated version of the variability table including the additional column representing the language paradigm to be used (denoted as *technique*).

Regarding the number of subdomains in the application domain and the number of paradigms used, several types of MPD can be distinguished:

- single domain—single paradigm
- multiple decoupled domains—single paradigm
- multiple decoupled domains—single paradigm for each subdomain
- multiple decoupled domains—multiple paradigms for each subdomain
- multiple subdomains in a directed acyclic graph—multiple paradigms.

With increasing number of domains and paradigms, the transformational analysis becomes more complicated. Two last categories require a combination of paradigms (see Fig. 4.1).

```
template <class T>
bool sort(T elements[], int nElements){
  . . .
}
```

Figure 4.1: A combination of the procedural and template paradigm (from [Cop99c]).

Multi-paradigm design as proposed by Coplien regularizes the use of multiple paradigms by first making the concept of paradigm more formal. To achieve this,

Coplien points out the need for solution domain (i.e. implementation environment) analysis, which is often underestimated. This results into a gap between design and implementation. Multi-paradigm design makes this gap smaller. It enforces *reusability of design*: both application and solution domain analyses can be reused independently (however, the transformational analysis is not reusable).

Aiming at reusability of design brings MPD close to *design patterns* [GHJV95].[6] The two approaches are not unrelated. Rather, they seem to be complementary; the design patterns capture the experience of designers by documenting the recommended solutions for the common problems in software development, while MPD relies on designer's experience.

The application of design patterns alone doesn't lead to a complete system implementation [GHJV95], and that is a case with MPD, too. Translation of the results of transformational analysis into code yields a code skeleton, but certainly not a full implementation.

MPD lacks a more sophisticated notation. The one proposed by Coplien encompasses only few types of tables and variability diagrams, which doesn't seem to be sufficient to capture all the relevant details of the MPD (mostly expressed just as informal text).

4.3 Intentional Programming

Programming languages with fixed syntax are limiting otherwise unlimited number of programming abstractions. Intentional programming group at Microsoft Research[7] offers a solution to this problem as a new software development paradigm called *intentional programming* [Sim96a, Sim96b, Sim99].

[6]One of the comments on the cover of this book (by Steve Vinoski) denotes reusable design as "the real key to software reuse"

[7]Led by Simonyi, the original developer of the MS Word and Excel.

The idea behind *intentional programming* (IP) is that programming abstractions—in IP terminology denoted as *intentions*—hosted by programming languages limited in the sense of accepted notations (due to underlying grammars), could live well without their hosts, (fixed-syntax) programming languages.[8]

However, one can argue that a programming language can be extended to support additional programming constructs, but this approach also has its limits because of the parsing methods. Eventually, such extensions lead to artificial constraints on the notation, as it is the case with a space that has to be inserted before a closing triangular bracket of the nested template in C++ [Cza98].

The solution proposed in IP is to have program represented by a so-called *intentional tree*, which is similar to abstract syntax tree.[9] This intentional tree consists of nodes representing intention instances. Each such an instance points to the corresponding intention declaration node. This node points to an intentional sub-tree, which represents the definition of the intention. The executable program is obtained in a process called *reduction*, in which the intentional tree is traversed and transformed according to the rules defined by intentions until it consists only of executable nodes. Such a *reduced* tree is represented in an intermediate language, which is to be translated into the executable code.

It is clear that IP needs (and has) a special and complex integrated programming environment, which is equipped with a special graphic editor instead of the usual text editor. This enables each intention to have its special graphic representation that best suits it.

Of course, entering a program in such an environment is completely different from entering it in a classic text editor, but one difference is especially interesting. A program text, as we are used to it, is a complete and an unambiguous representation of a program. In IP environment this is not so; it is not sufficient to examine the representation in the IP editor statically in order to obtain a full information about the program—intentions must be inspected individually. For example, two distinct variables—even if residing the same scope—can have the same name. However, there is a complete and unambiguous representation for a program written in IP: its intentional tree. But it is inconvenient to maintain intentional tree directly because of its complexity.

A program source representation in the IP programming environment seems strange at first sight, but it could be something perfectly normal in a near future. This change is comparable to textual program source representation replacing the punched cards one. On the other hand, IP counts on a binary format for the program files, which is a bit dangerous unless the format is made publicly available.

It should be pointed out that IP is not supposed to push out all the existing programming languages from the scene: it is meant to be capable of importing any program in any programming languages in order to reuse legacy code by a language-specific parser.[10]

[8]Other problems with classical programming languages are analyzed in [Cza98].

[9]According to Simonyi, it would be misleading to say that intentional tree is an abstract syntax tree "because there is no syntax an there are no productions" [Sim96a]. In fact, this is a little bit imprecise; since abstract syntax tree has nothing to do with productions, what he probably had in mind was a *concrete* syntax tree.

[10]IP environment can be extended with new parsers as libraries.

4.4 Summary

Although approaches discussed in the previous chapter carry a multi-paradigm flavor with them, they are not explicit about it. As we saw in this chapter, there are also approaches aiming at the explicit use of multiple paradigms.

Three such multi-paradigm approaches that were presented in this chapter are compared in Table 4.1 according to these criteria (of course, this comparison is not complete):

Paradigm: the concept of paradigm it enforces

Language: a programming language it is bound to

Language extension: a support for the language extension.

	Paradigm	Language	Language extension
MP in Leda	large-scale	Leda	no
MPD	small-scale	any	no
IP	small-scale	none	yes

Table 4.1: The three multi-paradigm approaches compared

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

Chapter 5

Conclusions and Further Work

This report started with the concept of *paradigm*, both in general sense and specifically in the sense of the software development. The analysis revealed two distinct meanings of the term *paradigm* according to the level of granularity: large-scale and small-scale.

After brief discussion of each of these two meanings, we focussed on selected post-object-oriented paradigms (aspect-oriented programming approaches and generative programming). Among these, a growing multi-paradigm tendency has been identified.

This multi-paradigm tendency materialized into approaches which articulate it explicitly. Three such approaches were discussed and compared: multi-paradigm programming in Leda, multi-paradigm design and intentional programming.

Multi-paradigm approach to software development makes the question which paradigm is the best to be a meaningless one—it has a potential of incorporating all the paradigms at disposal by the solution domain in developing a software system. It is a paradigm of paradigms—a metaparadigm.

In spite of this enthusiastic conclusion, multi-paradigm design (and multi-paradigm software development in general) must be further improved and refined if it is to be used in its full strength. Of multi-paradigm approaches considered, multi-paradigm design seem to be the most appropriate as the basis for the future form of the multi-paradigm software development.

There are many open issues regarding multi-paradigm design. MPD as proposed by Coplien is actually *MPD for C++*. Although we can speak of MPD in general as applicable to any programming language, before its actual application it has to be tailored to a given programming language yielding a method. Designing such a method for AspectJ would be especially interesting since it would not just enable the use of MPD for AspectJ in designing software systems, but also it could help to better understand the relationship between the aspect-oriented programming and multi-paradigm design.[1] Of course, it would be useful to tailor MPD to other programming languages as well—particularly Eiffel, for its proclaimed object-orientation, and Java, for its popularity (and similarity with C++).

Insufficiency of the notation used in MPD (few types of tables and variability diagrams) indicates the insufficiency in the method itself, which is too much based on the designer's intuition and experience. A method that would incorporate traditional analysis and design approaches into MPD could help to overcome this problem.

Since MPD relies on designer's experience, and since design patterns capture this

[1] Coplien denoted aspect-oriented programming as "the most fully general implementation of multi-paradigm design possible" [Cop00]

experience, the connection of the two should be considered in order to enable the use of the experience documented by design patterns in MPD.

Finally, translation of the results of transformational analysis into code yields just a code skeleton, but not a full implementation, which is left to an "experienced designer". Thus, a method is required that would make this transition to the actual program code more deterministic.

Bibliography

[AT98] Mehmet Aksit and Bedir Tekinerdogan. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. In *Proc. of AOP'98 workshop*, 1998. Available at [TRE].

[AWB$^+$93] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object-interactions using composition-filters. In *Object-Based Distributed Processing*, pages 152–184. Springer-Verlag, 1993. Available at [TRE].

[Bat99] Don Batory. Product-line architectures, generators and reuse. In *Proc. of GCSE'99 (co-hosted with the STJA 99)*, Erfurt, Germany, September 1999. Presentation slides and notes. Published on CD.

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

[BN97] Mária Bieliková and Pavol Návrat. *Funkcionálne a logické programovanie.* Slovak University of Technology in Bratislava, 1997. In Slovak.

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

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

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

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

[Cop] Home page of James O. Coplien. http://www.bell-labs.com/people/cope. Accessed on August 15, 2000.

[Cop96] James Coplien. Broadening beyond objects to patterns and to other paradigms. *ACM Computing Surveys*, 28A(4), December 1996. Available at http://www.acm.org/surveys (accessed on August 15, 2000).

[Cop99a] James O. Coplien. Multi-paradigm design. In *Proc. of GCSE'99 (co-hosted with the STJA 99)*, Erfurt, Germany, September 1999. Published on CD. Available at [Cop].

[Cop99b] James O. Coplien. Multi-paradigm design and implementation in C++. In *Proc. of GCSE'99 (co-hosted with the STJA 99)*, Erfurt, Germany, September 1999. Presentation slides and notes. Published on CD. Available at [Cop].

[Cop99c] James O. Coplien. *Multi-Paradigm Design for C++.* Addison-Wesley, 1999.

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

[Cza] Home page of Krzysztof Czarnecki. http:/www.prakinf.tu-ilmenau.de/∼czarn. Accessed on August 15, 2000.

[Cza98] 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 [Cza].

[Dem] Home page of Demeter group. http://www.ccs.neu.edu/research/demeter. Accessed on August 15, 2000.

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

[IBM] Home page of IBM Research, Subject-Oriented Programming. http://www.research.ibm.com/sop. Accessed on August 15, 2000.

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

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

[KOHK96] Matthew Kaplan, Harold Ossher, William Harrison, and Vincent Kruskal. Subject-oriented design and the watson subject compiler. In *Proc. of OOPSLA'96*, 1996. Available at [IBM].

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

[Kuh97] Thomas S. Kuhn. *Structure of Scientific Revolutions.* OIKYMENH, 1997. Czech translation.

[Lie97] Karl Lieberherr. Demeter and Aspect-Oriented Programming: Why are programs hard to evolve? Technical report, College of Computer Science, Northeastern University, Boston, 1997. Available at [Dem].

[LK98] Cristina Videira Lopes and Gregor Kiczales. Recent developments in AspectJ, 1998. Available at [Xer].

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

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

[Mic] Home page of Microsoft Research, Intentional Programming Group. http://www.research.microsoft.com/ip. Accessed on August 15, 2000.

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

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

[PLA] Home page of Product-Line Architecture Research Group. http://www.cs.utexas.edu/users/schwartz. Accessed on August 15, 2000.

[Sho93] Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.

[Sim96a] Charles Simonyi. Intentional programming—innovation in the legacy age, June 1996. Presented at IFIP WG 2.1 meeting, available at [Mic].

[Sim96b] Charles Simonyi. The intentional programming overview, July 1996. Available at [Mic].

[Sim99] Charles Simonyi. The future is intentional. *IEEE Computer*, May 1999. Available at [Mic].

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

[TRE] Home page of Twente Research and Education on Software Engineering (TRESE) Group. http://wwwtrese.cs.utwente.nl. Accessed on August 15, 2000.

[Xer] Home page of Aspect-Oriented Programming, hosted by the Xerox PARC Software Design Area. http://www.parc.xerox.com/aop. Accessed on August 15, 2000.