Towards Multi-Paradigm Software Development

Valentino Vranić

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

Multi-paradigm software development is a possible 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 a real support in design. Recently it is making a twofold breakthrough: several recent programming paradigms are encouraging it, while explicit multi-paradigm approaches aim at its full-scale support. In order to demonstrate this, a survey of selected recent software development (programming) paradigms (aspect-oriented approaches and generative programming) and multi-paradigm approaches (multi-paradigm programming in Leda, multi-paradigm design in C++, and intentional programming) is presented.

The survey is preceded and underpinned by the analysis of the concept of *paradigm* in computer science in the context of software development, since there is no common agreement about the meaning of this term, despite its wide use. The analysis has showed that there are two meanings of paradigm: large-scale and small-scale.

Keywords: software development, programming, large-scale, small-scale paradigm; commonality, variability analysis; multi-paradigm, aspect-oriented, generative, Leda, intentional programming; metaparadigm.

1. Introduction

The way software is developed is changing. Enforced by the need for mass production of quality software and enabled by the grown experience of the field, software development is moving towards industrialization. The question is no longer which single tool is the best one, but how to select the right tools for each task to be accomplished.

This article maps the state of the art in the field of post-object-oriented software development. Most notably, it is devoted to the promising concepts of aspect-oriented programming, generative programming and, particularly, to multi-paradigm software development.

The move towards multi-paradigm software development can be felt not only in new software development paradigms — e.g. aspect-oriented programming, which is bound to other paradigms from the first principles — it is present already in object-oriented programming. It is even more notable at the language level. It is hard to find a language that is pure in the sense of prohibiting any other than its proclaimed paradigm to be used in it.

What has been described is the implicit form of multi-paradigm software development. There are several approaches which make this idea explicit by enabling a developer not only to combine multiple paradigms, but also to choose the most appropriate one for the given feature of a system or family of systems.

The rest of this article is organized as follows. Section 2 explores the concept of paradigm in computer science in the context of software development. Section 3 is an overview of selected recent post-object-oriented paradigms, namely aspect-oriented programming approaches and generative programming. Section 4 proceeds with recent post-object-oriented approaches that exhibit explicitly the multi-paradigm character. Section 5 closes the article with conclusions and proposals for further work.

2. The Concept of Paradigm in Software Development

Paradigm is a very often used — and even more often abused — word in computer science in the context of software development. Its importance arose significantly with the appearance of so-called *multi-paradigm* approaches. Before discussing them, the concept of paradigm in software development requires a deeper examination. We must consider both the well-established meaning of paradigm in science and the actual meaning of the word in order to find out when its use in computer science is justified, and also to gain a better understanding of the concept of paradigm itself.

The term paradigm in science is strongly related to Thomas Kuhn and his essay [Kuh70], where it is used to denote a consistent collection of methods and techniques accepted by the relevant scientific community as a prevailing methodology of the specific field.

In computer science, the term paradigm denotes the essence of a software development process (often referred to as *programming*, see Section 2.1). Unfortunately, this is not the only purpose this term is used for. Probably no science has accepted this term with such an enthusiasm as computer science has; there are a lot of methods whose authors could not resist the temptation to raise them to the level of paradigm (just try a keyword "paradigm" in some citing index or digital library, e.g. [NEC]). Although not contradictory to the original meaning of the word paradigm, such an overuse causes confusion.

The basic meaning of paradigm is *example*, especially a typical one, or *pattern*, which is in a direct connection to its etymology (Greek "to show side by side") [Mer]. The meaning and etymology pose no restriction to the extent of the example or pattern it refers to. This is reflected in the common use of the word paradigm today: on the one hand, it has the meaning of a whole philosophical and theoretical framework of scientific school (akin to Kuhn's interpretation), while on the other hand, it is simply an example as in linguistics where it has the meaning of an example of conjugation or declension showing a word in all its inflectional forms [Mer].

Computer science, being a *science* whose great part is devoted to a special kind of *languages* intended for programming, hosts well both of these two interpretations of paradigm covered in more detail in the following text.

2.1. Large-Scale Paradigms

The notion of paradigm in the context of software development is used at two levels of granularity. Let us first discuss the *large-scale* meaning of paradigm, which, as it has already been mentioned, denotes the essence of a software development process. Coplien used the term large-scale paradigm to denote programming paradigms in, as he said, a "popular" sense [Cop99a].

Besides software development paradigm and software engineering paradigm, at least two more terms are used to refer to large-scale paradigm of software development: *programming paradigm* or, simply, *programming*. Although in common use (for historical reasons), one must be carful with these terms because of possible misunderstanding: programming sometimes stands for implementation only, as other phases of a software development process can also be referred to explicitly (e.g., object-oriented *analysis*, *object-oriented* design, etc.).

The name of a paradigm reveals its most significant characteristic [Vra00]. Sometimes it is derived from the central abstraction the paradigm deals with, as it is a function to functional paradigm, an object to object-oriented paradigm (according to [Mey97] it is not *object* but *class* that is the central abstraction in object-oriented paradigm), etc.

Lack of a general agreement on which name denotes which paradigm is a potential source of confusion. For example, although the term *functional paradigm* is usually used to denote a kind of application paradigm, as opposed to procedural paradigm, in [Mey97] it is used to denote exactly the procedural paradigm. It is hard to blame the author for misuse of the term knowing that the procedure is often being denoted as *function*.

It must be distinguished between the software development paradigm itself and the means used to support its realization. Unfortunately, this is another source of confusion. For example, any paradigm can be visualized by means of a visual environment and thus it makes no sense to

speak about the visual paradigm (as in [Bud95]). Making a complete classification and comparison of the software development paradigms is beyond the scope of this text; see [Náv96] for the comparison of selected programming paradigms regarding the concepts of abstraction and generalization.

A software development paradigm is constantly changing, improving, or better to say, refining. The basic principles it lays on must be preserved; otherwise it would evolve into another paradigm. Consider, for example, the simplified view on the evolution of object-oriented paradigm. First, there were commands (imperative programming). Then named groups of commands appeared, known as procedures (procedural programming). Finally, procedures were incorporated into the data it operated on yielding objects/classes (object-oriented paradigm).

However, according to Kuhn, paradigms do not evolve, although it could seem so; it is the *scientific revolution* that ends the old and starts a new paradigm [Kuh70]. A paradigm is *dominant* by definition and thus there can be only one at a time in a given field of science unless the field is in an unstable state. According to this, simultaneous existence of several software development paradigms indicates that the field of software development is either in an unstable state, or all these paradigms are parts of the one not yet fully recognized nor explicitly named paradigm. That paradigm is beyond the commonly recognized paradigms and it is about the (right) use and combination of those paradigms. Therefore it can be denoted as *metaparadigm*.

2.2. Small-Scale Paradigms

The notion of paradigm in computer science can also be considered at the small-scale level based on the programming language perspective. This perception of paradigm is apparent in James O. Coplien's multi-paradigm design [Cop99b] (covered in more detail 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 the common and variable part which together constitute a paradigm. A paradigm is then a *configuration of commonality and variability* [Cop99b]. This is analogous to conjugation or declension in natural languages, where the common is the root of a word and variability is expressed through the suffixes or prefixes (or even infixes) added to the root in order to obtain different forms of the word.

Scope, commonality and variability (SCV) analysis [CHW98] can be used to describe these language level paradigms. Here are the definitions of the three cornerstone terms in SCV analysis (instead of *entities* the word *objects* was used in [CHW98], but this could lead to a confusion with objects in the sense of object-oriented paradigm):

Scope (*S*): a set of entities;

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

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

SCV analysis of *procedures* paradigm illustrates the definition (based on an example 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*.

In the context of the small-scale paradigms, it is hard to find a single-paradigm programming language. The relationship between the small- and large-scale paradigms is similar to that between the programming language features and programming languages; the large-scale paradigms consist of the small-scale ones. We can revise here the source of the name of a large-scale paradigm: the name of a large-scale paradigm sometimes comes from the most significant small-scale paradigm it contains. For example, object-oriented (large-scale) paradigm consists of several (small-scale) paradigms: object paradigm, procedure paradigm (methods), virtual functions, polymorphism, overloading, inheritance, etc. Lack of a common agreement what are the exact characteristics of object-oriented paradigm makes it impossible to introduce the exact list of the small-scale paradigms that object-oriented paradigm consists of.

Having an expressive programming language that supports multiple paradigms introduces another issue: a method is needed for selecting the right paradigms for the features that are to be implemented. Such a method is a *metaparadigm* with respect to the small-scale paradigms. The small-scale paradigms metaparadigm is therefore a less elusive concept than the large-scale paradigms metaparadigm. One such small-scale metaparadigm, multi-paradigm design for C++, is described in Section 4.2.

One can understand small-scale paradigms as a programming language issue exclusively, while large-scale programming paradigms seem to have a broader meaning as they are affecting all the phases of software development. Actually, small-scale paradigms have an impact on all the phases of software development as well; either with or without an explicit support in analysis and design.

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 object-oriented paradigm. Many of these paradigms actually build upon object-oriented paradigm. Despite some of them are claimed not to be bound to object-oriented paradigm (and, indeed, they are more generally applicable), they are still widely applied in connection with it.

3.1. Beyond Object-Oriented Programming

Human perception of the world is to the great extent based on objects. Object-oriented programming, well-known under the acronym OOP, is based precisely on this perception of the world natural to humans. But what is OOP exactly? This question seems to be an answered one. Actually, there are plenty of answers to this question, but the trouble is that they are all different. 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 about what its essential properties are (to some, even inheritance is not an essential property of OOP, or it is being denoted as a minor feature [Bud95]). Perhaps Bertrand Meyer's viewpoint that "'object-oriented' is not a boolean condition" [Mey97] is the best characterization of this issue.

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, according to Booch, OOP has another important feature: it can serve as "the architectural framework in which other paradigms are employed" [Boo94]. Although this statement is probably overestimated in its applicability to all the paradigms, the truth is that some multi-paradigm languages (like Leda, see Section 4.1) are designed in this fashion. This reveals that OOP is multi-paradigmatic in its very nature and leaves not much space for the object-oriented purism.

The object-oriented purism comes from the dogma that everything should be modeled by objects. But not everything is an object; neither in the real world, nor in programming. Consider synchronization as a well-known example of a non-object concept; in natural language, we would probably refer to it as *aspect*. The aspects crosscut the structure of objects (or functional components, in general) making the code tangled. The pieces of code are either repeated throughout different objects or unnatural inheritance must be involved. Among other inconveniences, this "code scattering" has a bad impact on reuse.

There are also other problems with OOP, including those it was supposed to solve, which are mainly in the areas of reuse (discussed in [SN97]), adaptability, management of complexity and performance [CE00]. In the sense of the means for solution at the developer's disposal — that can be denoted as solution universe — OOP is not a universal paradigm. For example, OOP is not a universal paradigm either in C++ because it is not capable of making a full use of all of its features, or in C++ which is just a part of the solution universe of software development. OOP encompasses only a few interesting kinds of commonality and variability [Cop99a]. Other kinds are needed as well, so the non-object-oriented features of programming languages are often used even though the analysis and design were object-oriented.

```
class Point {
  int x,y;
  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 {
  int x1,y1,x2,y2;
  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 {
  before(): execution(* (Point || Line).set*(..)) {System.out.println("Write");}
  before(): execution(* Point.get*(..)) {System.out.println("Read");}
  before(): execution((Point || Line).new(..)) {System.out.println("Create");}
}
```

Fig. 1. Tracking access in AspectJ.

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, aspect-oriented programming (AOP) is a new programming paradigm that enables the modularization of crosscutting concerns [KLM+97]. We'll take a closer look at four main AOP approaches.

Xerox PARC Aspect-Oriented Programming

Most of the AOP terminology (as well as its name) later adopted by others was coined by Xerox PARC AOP group. Their research effort is concentration mainly on AspectJ [Xera], a general purpose AOP extension to Java [LK98].

AOP appeared as a reaction to the problem known from the *generalized procedure languages* [KLM+97], i.e. languages that use the concept of procedure to capture functionality (besides procedural languages, this includes functional and object-oriented languages as well). In such languages some program code fragments that implement a clearly separable *aspect* of a system (such as synchronization) are scattered and repeated throughout the program code that becomes *tangled*. AOP aims at factoring out such aspects into separate units. Aspects *crosscut* the *base* code in *join points*. These must be specified so aspects could be *woven* into the base code by a *weaver*.

A simple example written in AspectJ v1.0.0 (similar to the example in [LK98]) in Fig. 1 illustrates the idea. Two classes are presented there, `Point` and `Line`, whose methods are of three kinds: creating, writing and reading (implementation of the methods is omitted). Suppose we want to be informed 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`. Moreover, this would result in a tangled code. In AspectJ both problems can be avoided using aspects. In our example it is the aspect `ShowAccesses` that solves the problem. Note that the original code remains unchanged.

The solution with aspects is undoubtedly more elegant than the tangled one would be. However, the information where an aspect is to be woven (i.e., join points) is included in the aspect itself. This complicates the aspect reuse. AspectJ addresses this problem with abstract aspects and named sets of join points, so-called *pointcuts*.

Adaptive Programming

Adaptive programming (AP), proposed by Demeter group [Dem] at Northeastern University in Boston, deals mainly with the traversal strategies of class diagrams. Demeter group has used the ideas of AOP several years before the name aspect-oriented programming was coined. After the collaboration with the Xerox PARC AOP group had begun, Demeter group redefined AP as "the special case of AOP where some of the building blocks are expressible in terms of graphs and where the other building blocks refer to the graphs using traversal strategies" (building block stands for aspect or component) [Lie].

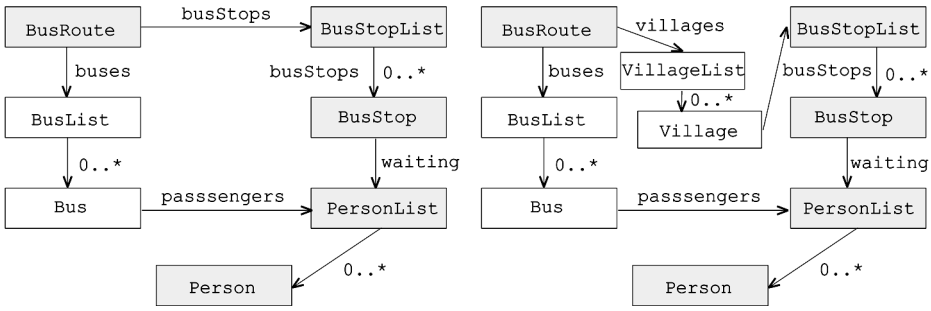

Fig. 2. Traversal strategies (from [Lie97], ©1997 Northeastern University).

The traversal strategies are partial graph specifications through mentioning a few isolated cornerstone nodes and edges, and thus they crosscut the graphs they are intended for.

An example of AP is presented in Fig. 2. The left part of the figure presents a UML class diagram of a system. Assume we would like to count the people waiting at the bus stations along the bus route. In ordinary OOP this would require either the implementation of small methods in all of the affected classes (depicted shaded) or rough breaking of the encapsulation by exposing some of the classes' private data.

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 objects of the class `Person` along the bus route, which is sufficient to count them. The right part of Fig. 2 demonstrates the robustness of this technique: the traversal strategy mentioned above applies in this case as well although the class diagram it was constructed for has changed.

Composition Filters

Composition filters (CF) is an aspect-oriented programming approach in which 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). A filter behavior is simple: each filter can either accept or reject the received message, but the semantics of the operations depends on the filter type. For example, if an `Error` filter accepts the received message, it is forwarded to the next filter, but if it was a `Dispatch` filter, the message would be executed. A detailed description of CF can be found in [AWB+93, Koo95].

In Fig. 3 two sets of filters (written in Sina language [Koo95], which directly adopts the CF model [AT98, AWB+93]), are shown. These

```
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.*};
```

Fig. 3. Tracking access example implemented using composition filters approach.

filters are attached to the `Point` and `Line` classes from Fig. 1. The existence of the class `ShowAccess` is presumed. `ShowAccess` provides three methods — `WriteAccess`, `ReadAccess` and `CreateAccess`) — that simply write out the type of the access. They are called by the three corresponding `Dispatch` filters, in case the message was accepted. Afterwards, the method of the inner object, which has actually been called, is executed (`inner.*`).

From the perspective of AOP, the class `ShowAccess` implements the aspect, while the filters represent the join points. Thus, the join points in this case are separated from the aspect, which is better regarding the aspect reuse.

Subject-Oriented Programming

A concept can be defined by naming its properties. This is sufficient to precisely define and identify mathematical concepts, but the same does not apply to natural concepts. Their definitions are *subjective* and thus never complete (more details about conceptual modeling can be found in [CE00]).

Subject-oriented programming (SOP), developed at IBM as an extension to OOP [IBM], is based on subjective views, so-called *subjects*. A 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 by writing the *composition rules*, which specify 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.

The example from Fig. 1 reimplemented in Watson Subject Compiler-like syntax (the actual syntax could by slightly different) is presented in Fig. 4. We assume that the class `ShowAccess` is implemented in `Access` namespace and that the classes `Point` and `Line` are implemented in the `Graphics` namespace. The join-points, represented by composition rules, are separated from the aspect and represented by a separate class (as in CF approach). The composition rules for the methods `getY`, `getX1`, `getY1` and `getX2` are omitted in Fig. 4 (indicated by ellipsis) since they are analogous to the rules for `getX` or `getY2`.

This is not a characteristic case of the application of SOP (such 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. In [CE00] SOP is viewed 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. reject the very idea that SOP (which they call *subjective programming*) could be AOP, arguing that the methods from different subjects, which are being automatically composed in SOP, are components in the AOP sense, since they can be well localized in a generalized procedure (routine) [KLM+97].

```
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];
```

Fig. 4. Tracking access example implemented using subject-oriented approach.

But this seems to be a more general issue, since it applies to AspectJ, too, where it has been identified as *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.

As observed in [Cza98], SOP is close to GenVoca [BG97], a successful approach to software reuse. In GenVoca, systems are composed out of *layers* according to *design rules*: GenVoca layers can be easily simulated by subjects.

3.3. Generative Programming

Krysztof Czarnecki and Ulrich Eisenecker propose 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 [CE00]:

> 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 four other paradigms (see Figure 5):

- object-oriented programming, providing effective system modeling techniques,
- generic programming, enabling reuse through parameterization,
- domain-specific languages, increasing the abstraction level for a particular domain, and
- aspect-oriented programming, used to achieve the separation of concerns.

In order to be used, GP first has to be tailored to a particular domain. This process will yield a methodology for the families of systems to be developed, which can be viewed as a paradigm in its own right. This gives a certain meta-paradigm flavor to GP.

In the solution domain, GP requires metaprogramming for weaving and automatic configuration. To support domain-specific notations, syntactic extensions are needed. *Active libraries*

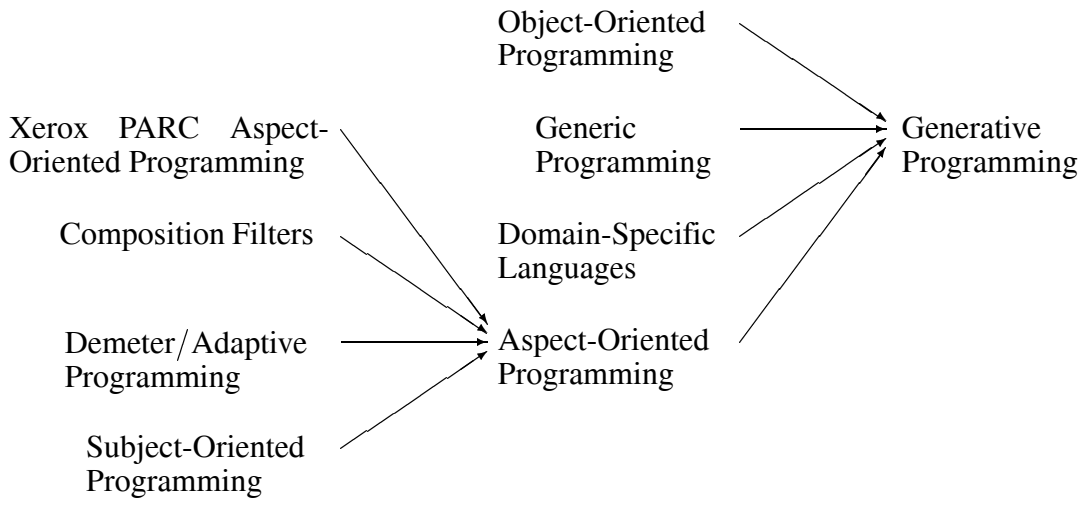

Fig. 5. Generative programming and related paradigms. The arrows represent "is incorporated into" relationship.

are proposed in [CE00], which can be viewed as *knowledgeable agents* interacting with each other to produce concrete components, as appropriate to cover this requirement.

4. Multi-Paradigm Approaches

In the survey of recent post-object-oriented software development paradigms presented in the previous section a spontaneous move towards the integration of paradigms became apparent. This section proceeds with explicit multi-paradigm approaches.

4.1. Multi-Paradigm Programming in Leda

The question how to support multi-paradigm programming at the language level can be answered simply: create a multi-paradigm language. Timothy Budd took this route by creating a multi-paradigm programming language called Leda.

According to Budd, Leda supports four programming paradigms [Bud95]: imperative (procedural, to be more precise) logic, functional, and object-oriented. The term paradigm, as used by Budd, denotes a large-scale paradigm (with respect to the classification of paradigms introduced in Section 2). This means that Leda actually supports more than four small-scale paradigms. This is clear having in mind that, for example, object-oriented paradigm breaks down into six or more small-scale paradigms, as has been shown in Section 4.2. Nevertheless, in order not to digress from the intent of this approach, just the mechanisms by which each of the four proclaimed paradigms is supported in the language will be discussed.

Leda has a Pascal-like (i.e., Algol-like) syntax and, moreover, in Leda the mechanism upon which all the four supported paradigms realization is based on are functions (procedures that return values). This makes a good background for *procedural* paradigm, denoted by Budd as *imperative*.

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

Functional paradigm requires no special mechanism other than those provided by functions, because Leda permits a function to be an argument of another function and to return a function. Thus, functional paradigm is achieved by using functions in the recursive fashion while refraining from assignments.

In addition to the basic mechanisms of *object-oriented paradigm*, such as classes, inheritance, encapsulation etc., Leda also supports *parameterized types* (considered by some authors a part of object-oriented paradigm [Mey97]).

Despite Leda is not widely used, it is worth consideration 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 to be the best language for programming is similar to a search for the best programming paradigm. No matter how many paradigms are supported in a programming language, the number is finite and, obviously, it does not embrace future paradigms. One can argue that it is possible to extend the language with new programming mechanisms in order to support new paradigms. This is not only possible, but often practiced. Unfortunately, due to limitations set by parsing methods, programming languages cannot be extended indefinitely.

Leda is an example of a language *created* (from scratch) in order to support multiple paradigms. We can study existing interconnecting 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. An example of interconnecting object-oriented and logic programming (Loops and Xerox Quintus Prolog) can be found in [KE88].

There are lots of approaches that fall into this category. Yet another approach and an overview of similar approaches, together with the discussion of the problems of paradigms integration, can be found in [VS95]. Such approaches are popular especially in the field of artificial intelligence because of the need to combine the

two paradigms traditionally used in this field, logic and functional programming, both with each other and together with OOP.

Different paradigms are expressed using different syntax. BETA language [Mad00] is supposed to overcome this inconvenience through a *unified syntax* achieved by introducing the so-called *pattern* as an abstraction of all other programming language constructs appearing in the paradigms it supports. The approach is therefore denoted as *unified paradigm*, but it is not fundamentally different from other "created to be multi-paradigm" languages.

4.2. Multi-Paradigm Design for C++

Multi-paradigm design for C++ (MPD), as proposed by Coplien [Cop99b, Cop00], has its roots in the multi-paradigm features of C++. Despite these multi-paradigm features, C++ is often considered to be just 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 a support in the (object-oriented) design.

MPD is a metaparadigm intended for developing families of systems, therefore akin to domain engineering approaches. It deals with choosing the appropriate paradigm for a feature being designed and implemented. MPD is based on SCV analysis (discussed in Section 2.2) or, to be more precise, SCVR analysis, where R stands for the relationship between the domains [Cop00], which are covered by variability dependency graphs (explained further in this section). On the other hand, neither SCV, nor SCVR analysis is mentioned in [Cop99b]; the term *commonality and variability analysis* is used instead to denote the same thing. Commonality analysis concentrates on common attributes while the aim of variability analysis is to parameterize the variation.

The major steps in MPD are: commonality and variability analysis of the application domain, commonality and variability analysis of the solution domain, transformational analysis and translation from the transformational analysis to the code. These steps need not to be performed sequentially. They can be performed in parallel (to some extent) and revisited as needed. Before starting the actual MPD, it is recommended to evaluate the possibility to reuse an existing (similar) design. If the commonalities and variabilities of the application domain do not fit any existing solution domain structures, creation of a new application-oriented (i.e., domain-specific) language should be considered.

Application domain analysis. Commonality analysis of the application domain (usually denoted as problem domain) starts with finding commonality domains and creating domain dictionary. It then proceeds in parallel with variability analysis, whose results — the *parameters of variation* of a given commonality domain and their characteristics — are being summarized in *variability tables* (one per each commonality domain), as depicted in the upper part of Fig. 6.

As already mentioned, *variability dependency graphs* (denoted also as *domain dependency* graphs or diagrams) are used to capture the relationship between domains and their parameters of variation, which are also domains. Variability dependency graphs are directed graphs whose nodes represent domains and the edges represent "depends on" relationship (in the direction indicated by an arrow) between the domains and their parameters of variation. Despite the simple notation, variability dependency graphs are quite useful in identifying *overlapping* domains (such domains can be merged) and *codependent* domains, i.e. the domains with circular dependencies (which must be resolved).

Solution domain analysis. The same commonality and variability analysis as applied to the application domain is applied to the solution domain, i.e. the programming language. First, a description with an example is provided of the identified small-scale paradigms, manifested through the language features, structured according to commonality, variability and binding. The analysis proceeds with exploring the *negative variability*, a variability that violates the rule of variation by attacking the underlying commonality. A *positive variability*, as opposed to the negative one, can be parameterized. The negative variability has to be kept small. If it becomes larger than the commonality, the design

Variability tables (from application domain SCVR analysis)

Text Editor Variability Analysis for Commonality domain:
TEXT EDITING BUFFERS (*Commonality: Behavior and Structure*)

Parameters of variation	Meaning	Domain	Binding	Default
Output medium *Structure, Algorithm*	...	Database, RCS file, TTY, UNIX file	Run time	UNIX file

Family table (from solution domain SCVR analysis)

Commonality	Variability	Binding	Instantiation	Language Mechanism
		. . .		
Related operations and some structure (positive variability)	. . .			
	Algorithm (especially multiple), as well as (optional) data structure and state	Compile time	Optional	Inheritance
	Algorithm, as well as (optional) data structure and state	Run time	Optional	Virtual functions

Fig. 6. Transformational analysis in MPD (according to an example from [Cop99b]).

should be refactored to reverse the commonality and variability.

The results of the solution domain commonality and variability analysis are summarized in the *family table*, as shown in the lower part of Fig. 6, and in the table used to express *features for negative variability*, where for each combination of the kind of commonality and the kind of variability the language feature for positive variability and the one for the corresponding negative variability are introduced.

Transformational analysis. The tables obtained in the preceding analyses are used in *transformational analysis*, which is, roughly speaking, a matching of application domain structures described in variability tables, with solution domain structures, i.e. paradigms, described in family tables. Figure 6 shows how this matching is performed. Prior to the matching, the commonality domain has to be generalized (e.g., TEXT EDITING BUFFERS: *behavior, structure*), as well as the parameters of variation (e.g., output medium: *structure, algorithm*).

MPD emphasizes the solution domain analysis whose underestimation in contemporary software development methodologies results in a gap between design and implementation.

To a certain extent, MPD enforces the *reuse of design*: both application and solution domain analysis can be reused independently; however, transformational analysis is not reusable. This brings MPD close to *design patterns*, as discussed in [Cop99b]. On the very cover of the design patterns cornerstone book [GHJV95] Steve Vinoski points out that a reusable design is "the real key to software reuse". This claim is being justified in the ongoing research on reuse with design patterns [SN00].

Indeed, MPD and design patterns seem to be complementary; design patterns capture designers' experience by documenting the recommended solutions for common problems in software development, while MPD relies on this experience. However, to make a full use of design patterns in MPD, and in software development in general, a better way of their representation is needed [SNB98].

Although the design patterns from [GHJV95] are inspired by Alexandrian patterns [Ale79], not all of them are the patterns in the Alexandrian sense: some of them can be formalized as configurations of commonality and variability (unlike Alexandrian patterns). As such they can be incorporated directly into MPD (by adding them to the family table), as anything else that can be formalized as a configuration of commonality and variability (i.e., other paradigms and solution domain tools that are not supported by the main programming language, like databases or parser generators) [Cop99b].

One of the problems with MPD is the unsuitability of the notation used: only a few types of tables and variability diagrams with a lot of relevant details expressed as informal text. With a better notation, like feature modeling [Vra01], transformational analysis could become more transparent. A better notation could also ease the transition to the actual program code (the program skeleton).

4.3. Intentional Programming

Programming languages with fixed syntax are limiting otherwise unlimited number of programming abstractions. Intentional programming group at Microsoft Research offered a solution to this problem as a new software development paradigm called *intentional programming* (IP) [Sim99, Sim96] (the project is on hold from Spring 2001 [Roe]). The idea behind IP is that programming abstractions, which are in IP denoted as *intentions*, could live better without their hosts, (fixed-syntax) programming languages, because of their limits in the accepted notations (due to underlying grammars).

A program in IP is represented by a so-called *intentional tree*, whose nodes represent intention instances. Each intention instance points to the corresponding intention declaration node providing a method which specifies the process of transforming the subtree headed by the intention instance. The executable program is obtained in a process called *reduction* in which the intentional tree is traversed and transformed according to the rules indicated by intention declarations until it consists only of executable nodes. Such a *reduced* tree is represented in an intermediate language. The executable code is generated from this representation.

It would be inconvenient for a human to directly maintain the intentional tree. This is being performed in a programming environment with a special graphic editor instead of the usual text editor. It enables each intention to have its own graphic representation. Of course, entering a program in such an environment is quite different from entering it in a classic text editor. A program text, as we are used to it, is a complete and an unambiguous representation of the program. In IP environment this is not so. What is presented in IP editor is only a view of the actual program. To illustrate this, consider one peculiarity: two distinct variables can have the same name (even if they reside the same scope). This is possible because the intentional tree does not rely on the names to identify intentions; the names are provided only for developers' convenience.

Although it can seem so, IP is not intended to push out the existing programming languages from the scene. It can import any program in any programming language if a parser for that language — in the form of a library — is added to IP environment.

4.4. Multi-Paradigm Approaches Compared

The three multi-paradigm approaches presented in this section are compared in Table 1 according to the selected criteria: the concept of *paradigm* the approach enforces, a programming *language* the approach is bound to, and whether the approach supports the *language extension*.

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

	Paradigm	Language	Language extension
MP in Leda	large-scale	Leda	no
MPD	small-scale	any	not applicable
IP	small-scale	none	yes

Table 1. The three multi-paradigm approaches compared.

5. Conclusions and Further Work

The concept of paradigm in computer science in the context of software development has been analyzed in this article. Two distinct meanings of paradigm in software development have been identified and discussed: large-scale and small-scale.

A survey of selected post-object-oriented paradigms, namely aspect-oriented approaches and generative programming, has been presented. A growing multi-paradigm tendency has been identified in these approaches. This tendency has materialized into explicit multi-paradigm approaches. Three such approaches have been discussed and compared: multi-paradigm programming in Leda, multi-paradigm design for C++, and intentional programming.

Multi-paradigm approach to software development makes the question which paradigm is the best (and therefore should replace all other paradigms) a meaningless one. It has a potential of incorporating all the paradigms at disposal of the solution domain. It is a paradigm of paradigms: a metaparadigm.

However, multi-paradigm software development must be further improved and refined if it is to be used in its full strength. Among the multi-paradigm approaches considered, multi-paradigm design (for C++), described in Section 4.2, seems to be the most appropriate as the basis for the future form of multi-paradigm software development.

Multi-paradigm design can be tailored to any programming language by applying commonality and variability analysis to it. It would be particularly interesting to establish multi-paradigm design for AspectJ (see Section 3.2) since it could help to understand better the relationship between multi-paradigm design and aspect-oriented programming (although, of course, AspectJ is not the same as aspect-oriented programming in general), which Coplien denoted as "the most fully general implementation of multi-paradigm design possible" [Cop00]. An initial work towards establishing multi-paradigm design for AspectJ has been reported in [Vra01].

The notation used in multi-paradigm design, which besides informal description embraces only two types of tables and a kind of simple graphs, is not appropriate. This is apparent especially during transformational analysis. Similarly to commonality and variability analysis of multi-paradigm design, *feature modeling* also expresses commonalities and variabilities explicitly, but using a more sophisticated notation (see [CE00] for more details on feature modeling and feature diagrams). Both solution and application domains can be represented as feature models, as has been demonstrated in [Vra01]. This eases transformational analysis and brings multi-paradigm design and generative programming closer to each other.

Acknowledgments

This work was partially supported by Slovak Science Grant Agency, grant No. G1/7611/20. I would like to thank Pavol Návrat for his valuable suggestions.

References

[Ale79] C. ALEXANDER. *The Timeless Way of Building*. Oxford University Press, 1979. Cited in [Cop99b].

[AT98] M. AKSIT AND B. TEKINERDOGAN. Solving the modeling problems of object-oriented languages by composing multiple aspects using composition filters. In *Proc. of the Aspect-Oriented Programming Workshop at ECOOP'98*, 1998. Available at [Twe].

[AWB+93] M. AKSIT, K. WAKITA, J. BOSCH, L. BERGMANS, AND A. YONEZAWA. Abstracting object-interactions using composition-filters. In

Proc. of 7thEuropean Conference on Object-Oriented Programming (ECOPP'93) Workshop, LNCS 791, pages 152–184, Kaiserslautern, Germany, 1993. Springer. Available at [Twe].

[BG97] D. BATORY AND B. 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 [Pro].

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

[Bud95] T. A. BUDD. *Multiparadigm Programming in Leda*. Addison-Wesley, 1995.

[CE00] K. CZARNECKI AND U. EISENECKER. *Generative Programing: Principles, Techniques, and Tools*. Addison-Wesley, 2000.

[CHW98] J. COPLIEN, D. HOFFMAN, AND D. WEISS. Commonality and variability in software engineering. *IEEE Software*, 15(6), November 1998. Available at [Cop].

[Cop] J. O. COPLIEN. Home page. `http://www.bell-labs.com/people/cope`. Accessed on November 15, 2001.

[Cop99a] J. O. COPLIEN. Multi-paradigm design and implementation in C++. Slides and notes of the tutorial given at *1st International Conference on Generative and Component-Based Software Engineering (GCSE'99)*, Erfurt, Germany, September 1999. Available at [Cop].

[Cop99b] J. O. COPLIEN. *Multi-Paradigm Design for C++*. Addison-Wesley, 1999.

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

[Cza] K. CZARNECKI. Home page. `http://www.prakinf.tu-ilmenau.de/~czarn`. Accessed on November 15, 2001.

[Cza98] K. 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. Partially available at [Cza].

[Dem] Demeter group. Home page. `http://www.ccs.neu.edu/research/demeter`. Accessed on October 30, 2001.

[GHJV95] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

[KE88] T. KOSCHMANN AND M. W. EVENS. Bridging the gap between object-oriented and logic programming. *IEEE Software*, 60:36–42, July 1988.

[KLM+97] G. KICZALES, J. LAMPING, A. MENDHEKAR, C. MAEDA, C. V. LOPES, J.-M. LOINGTIER, AND J. IRWIN. Aspect-oriented programming. In M. Aksit and S. Matsuoka, editors, *Proc. of 11th European Conference on Object-Oriented Programming (ECOOP'97)*, LNCS 1241, Jyväskylä, Finland, June 1997. Springer. Available at [Xerb].

[KOHK96] M. KAPLAN, H. OSSHER, W. HARRISON, AND V. KRUSKAL. Subject-oriented design and the watson subject compiler. In *11th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'96)*, 1996. Available at [IBM].

[Koo95] P. 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 [Twe].

[Kuh70] T. S. KUHN. *The Structure of Scientific Revolutions*. University of Chicago Press, Chicago, 1970. Czech translation, OIKYMENH, 1997.

[Lie] K. J. LIEBERHERR. Connections between Demeter/adaptive programming and aspect-oriented programming. Web document, College of Computer Science, Northeastern University, Boston, USA. Available at [Dem].

[Lie97] K. J. LIEBERHERR. Demeter and aspect-oriented programming: Why are programs hard to evolve? Presentation slides, *3rd Conference Smalltalk und Java in Industrie und Ausbildung (STJA 97)*, Erfurt, Germany, 1997. Available at [Dem].

[LK98] C. V. LOPES AND G. KICZALES. Recent developments in AspectJ. In *Proc. of 12th European Conference on Object-Oriented Programming (ECOPP'98) Workshops, Demos, and Posters*, LNCS 1543, Brussels, Belgium, July 1998. Springer. Available at [Xerb].

[LLM99] K. J. LIEBERHERR, D. LORENZ, AND M. MEZINI. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999. Available at [Dem].

[Mad00] O. L. MADSEN. Towards a unified programming language. In J. L. Knudsen, editor, *Proc. of 14th European Conference on Object-Oriented Programming(ECOOP 2000)*, Sophia Antipolis and Cannes, France, June 2000. Springer LNCS 1850.

[Mer] Merriam-Webster OnLine. Merriam-Webster's Collegiate Dictionary. `http://www.m-w.com`. Accessed on November 15, 2001.

[Mey97] B. MEYER. *Object-Oriented Analysis Software Construction*. Prentice Hall, second edition, 1997.

[NEC] NEC Research Institute. ResearchIndex: The NECI Scientific Digital Research Library. `http://citeseer.nj.nec.com`. Accessed on November 15, 2001.

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

[OHBS94] H. OSSHER, W. HARRISON, F. BUDINSKY, AND I. SIMMONDS. Subject-oriented programming: Supporting decentralized development of objects. In *Proc. of 7th IBM Conference on Object-Oriented Technology*, July 1994. Available at [IBM].

[Pro] Product-Line Architecture Research group. Home page. `http://www.cs.utexas.edu/users/schwartz`. Accessed on November 15, 2001.

[Roe] L. ROEDER. Home page. `http://www.aisto.com/roeder`. Accessed on November 21, 2001.

[Sim96] C. SIMONYI. Intentional programming — innovation in the legacy age, June 1996. Presented at IFIP WG 2.1 meeting, available at [Roe].

[Sim99] C. SIMONYI. The future is intentional. *IEEE Computer*, 32(5):56–57, May 1999.

[SN97] M. SMOLÁROVÁ AND P. NÁVRAT. Software reuse: Principles, patterns, prospects. *Journal of Computing and Information Technology*, 5(1):33–48, 1997.

[SN00] M. SMOLÁROVÁ AND P. NÁVRAT. Reuse with design patterns: Towards pattern-based design. In Y. Feng, D. Notkin, and M. Gaudel, editors, *Proc. Software: Theory and Practice*, pages 232–235, Beijing, China, 2000. PHEI - Publishing House of Electronics Industry.

[SNB98] M. SMOLÁROVÁ, P. NÁVRAT, AND M. BELIKOVÁ. Abstracting and generalising with design patterns. In A. G. U. Güdükbay, T Dayar and E. Gelenbe, editors, *Proc. of 13th International Symposium on Computer and Information Sciences (ISCIS'98)*, pages 551–558, Belek-Antalya, Turkey, 1998. IOS Press.

[Twe] Twente Research and Education on Software Engineering (TRESE) group. Home page. `http://trese.cs.utwente.nl`. Accessed on November 15, 2001.

[Vra00] V. VRANIĆ. Multiple software development paradigms and multi-paradigm software development. In J. Zendulka, editor, *Proc. of 3rd International Conference on Information Systems Modelling (ISM 2000)*, pages 191–196, Rožnov pod Radhoštěm, Czech Republic, May 2000. MARQ.

[Vra01] V. VRANIĆ. AspectJ paradigm model: A basis for multi-paradigm design for AspectJ. In J. Bosch, editor, *Proc. of 3rd International Conference on Generative and Component-Based Software Engineering (GCSE 2001)*, LNCS 2186, pages 48–57, Erfurt, Germany, September 2001. Springer.

[VS95] S. VRANEŠ AND M. STANOJEVIĆ. Integrating multiple paradigms within the blackboard framework. *IEEE Transactions on Software Engineering*, 21(3):244–262, 1995.

[Xera] Xerox PARC. AspectJ home page. `http://aspectj.org`. Accessed on November 15, 2001.

[Xerb] Xerox PARC. Software Design Area home page. `http://www.parc.xerox.com/sda`. Accessed on November 15, 2001.

Received: February, 2001
Revised: November, 2001
Accepted: December, 2001

Contact address:
Valentino Vranić
Department of Computer Science and Engineering,
Faculty of Electrical Engineering and Information Technology
Slovak University of Technology in Bratislava, Slovakia
Ilkovičova 3
812 19 Bratislava
Slovakia
Phone: +421 (2) 602 91 548
Fax: +421 (2) 654 20 587
e-mail: `vranic@elf.stuba.sk`
WWW: `http://www.dcs.elf.stuba.sk/~vranic`

VALENTINO VRANIĆ received his Bc. (BSc.) in 1997, and his Ing. (MSc.) in 1999, both in information technology, and both from the Slovak University of Technology in Bratislava. Since 1999 he is a PhD student at the Department of Computer Science and Engineering, Faculty of Electrical Engineering and Information Technology of the Slovak University of Technology in Bratislava. His main research interests are multi-paradigm software development and aspect-oriented programming. He is a member of the Slovak Society for Computer Science.