Multiple Software Development Paradigms and Multi-Paradigm Software Development

Valentino Vranić [*]
vranic@elf.stuba.sk

Abstract: While OOP (including OOA/D) is reaching the level of maturity of structured programming, new design paradigms are arising. To a big extent this is motivated by the faith in the existence of "the best paradigm", which could solve the difficulties software engineering deals with today. On the other hand, undoubtedly a great effort has been made in order to enable and manage the application of different paradigms to a single system (or a family of systems) development in a coexistent way. This is referred to as multi-paradigm design and implementation. What do new paradigms, like aspect-oriented programming, subject-oriented programming and generative programming, offer and what are the possibilities of a multi-paradigm design and implementation (as a metaparadigm)—these are the questions addressed in this article.

Keywords: software development paradigm, metaparadigm, multi-paradigm, aspect-oriented programming, subject-oriented programming, generative programming, multi-paradigm design, intentional programming.

1 Introduction

Why do new paradigms appear? The answer is simple: existing paradigms are not good enough. Consider the object-oriented paradigm. Many times before it has been pointed out that object-oriented paradigm (i.e. object-oriented programming, OOP) fails to solve some very important issues it was proposed it would. The problems appear to be mainly in the field of reuse [12], adaptability, management of complexity and performance [5]. These open issues create a space for new paradigms to rise and so we could expect that the upcoming paradigms would be better and better until one day the best one would appear. This is rather simplified view, but the point is that there is a certain evolution of paradigms. However, is there a best paradigm—that's a tough question.

Speaking of the paradigms' evolution yet another thing has to be considered: the integration of paradigms (which is known as *multi-paradigm* approach). The goal of such an integration can be a collaboration of paradigms (an example can be found in [7]), but the integration between paradigms can be loose, so paradigms would just coexist rather than build on each other or collaborate.

In this paper, several current software paradigms are briefly presented and then a multi-paradigm approach is considered together with two examples of it.

2 Multiple Software Development Paradigms...

In software engineering the term *software development paradigm* (or, more often, simply *paradigm*), is widely used to refer to the essence of certain software development process.

[*] Department of Computer Science and Engineering, Slovak University of Technology, Bratislava, Slovakia

A software development paradigm represents a consistent collection of methods and techniques accepted by the relevant scientific community as a prevailing methodology of the specific field. The name of a paradigm reveals what is a central abstraction it deals with, as it is an object to object oriented paradigm, a function to functional paradigm etc.

In spite of the fact that software development paradigm refers to all the phases of the software development process, not only to the implementation, in place of a term *software development paradigm* often we can found 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, and OOP to refer specifically to the implementation phase. Discussion on various programming paradigms can be found e.g. in [9].

Some recent software development paradigms include: agent-oriented programming, aspect-oriented programming and generative programming. In this short analysis, the principles of these paradigms and also of multi-paradigm approaches are presented.

2.1 Agent-Oriented Programming

There is a lot of confusion about the meaning of the term *agent*. Shoham [10] proposes *agent-oriented programming* paradigm based on the original sense of the word *agent*, which is "someone acting on behalf of someone else". Speaking more precisely, agent is an entity whose state is *viewed* as consisting of mental components such as beliefs, capabilities, choices and commitments. Thus, agents are not supposed to *be* intelligent entities in the sense of human intelligence, but they are supposed to have such a behavior, that we could *view* agents as intelligent in order to interact with them like we are used to interact with real persons.

As Shoham points out, agent-oriented programming can be viewed as a specialization of the OOP paradigm. Table 1 summarizes the relationship between AOP and OOP.

	OOP	Agent-Oriented Programming
Basic unit	object	agent
Parameters defining state of basic unit	unconstrained	beliefs, commitments, capabilities, choices...
Process of computation	message passing and response methods	message passing and response methods
Type of message	unconstrained	inform, request, offer, promise, decline...
Constraints on methods	none	honesty, consistency...

Tab. 1: OOP versus AOP (from [10])

Agent-oriented programming makes a move towards higher *intentionality*, which is a general trend in recently developed paradigms, such as intentional programming (see section 3.2).

2.2 Aspect-Oriented Programming and Related Paradigms

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 very natural view of the world.

What can be wrong then with OOP? In OOP, we are taught to see everything as an object, but not everything is an object neither in a real world, nor in programming itself. For example, synchronization is certainly not an object. It is usually perceived as something that can be marked as an *aspect*. The aspects crosscut the objects (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) must be involved.

Aspect-oriented programming (AOP), as proposed by Xerox PARC AOP group, is a new programming methodology that enables the modularization of crosscutting concerns [6].

Another three independently developed paradigms can be viewed as aspect-oriented decomposition approaches, which extend the OOP model to allow us the encapsulation of aspects [5]: subject-oriented programming (SOP), composition filters (CF) and Demeter/adaptive programming (AP). Although these approaches build upon OOP, the very idea of the AOP is not limited to it.

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 at all because their definitions are *subjective* and thus never complete (more details about conceptual modelling can be found in [5]).

Subject oriented programming (developed at IBM) reflects this subjectivity of concepts. It is based on subjective views, so-called *subjects*. 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 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 Czarnecki [5] observes, this is close to *GenVoca* approach [2], where the systems are composed out of *layers* according to *design rules*: GenVoca layers can be easily simulated as subjects.

SOP can be 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.

Composition filters is another paradigm closely related to the AOP. From the AOP point of view, CF is an aspect-oriented programming technique where different aspects are expressed as declarative and orthogonal message transformation specifications called *filters* [1].

A message sent to an object is evaluated and manipulated by the filters of that object, which are defined in an ordered set. Filters are fully separated from the class and thus can be reused separately.

The *adaptive programming* (proposed by the Demeter group) deals mainly with traversal strategies of class diagrams as 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 [8].

2.3 Generative Programming

In his Ph.D. thesis, Czarnecki [5] 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 (GP).

GP is a unifying paradigm—it is closely related to three other paradigms (see Figure 1): *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 AOP (which was discussed in section 2.2).

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 notions to cover these requirements. Active libraries, which can be viewed as *knowledgeable agents* (it would be useful to consider some agent-oriented programming techniques here, see section 2.1) interacting with each other to produce concrete components, require appropriate programming environment.

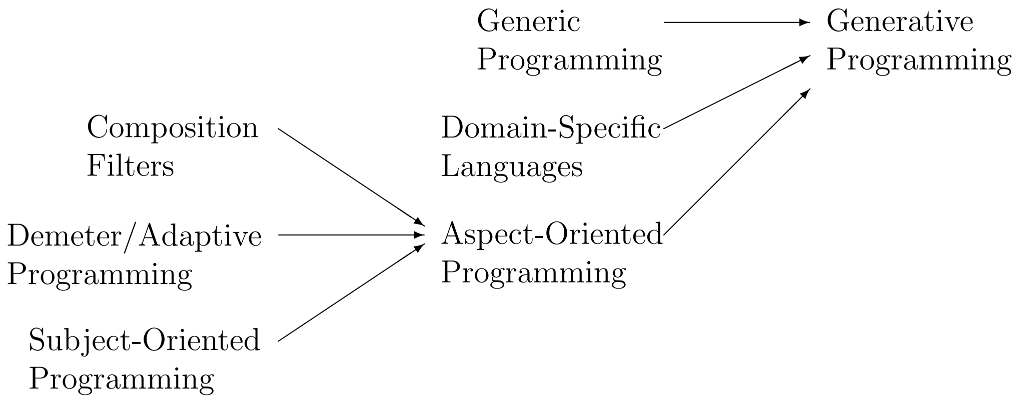
Generic Programming
Generative Programming
Composition Filters
Domain-Specific Languages
Demeter/Adaptive Programming
Aspect-Oriented Programming
Subject-Oriented Programming

Fig. 1: Generative programming and related paradigms. The arrows represent "is incorporated into" relationship.

3 ...and Multi-Paradigm Software Development

Up to this point it should be clear that there has been made a great effort in order to integrate the software development paradigms (see Figure 1). The integration of two or more paradigms can result into a new paradigm. However, if the paradigms, which are to be integrated, are not so close to each other, we gain something that is beyond a single paradigm concept. This is known as a *multi-paradigm* approach.

Closely related to the multi-paradigm approach is a *metaparadigm* as a method for choosing the appropriate features among a particular set of paradigms (ideally, it would be among all the paradigms).

3.1 Multi-Paradigm Design

Multi-paradigm design (and implementation) (MPD), proposed by Coplien, has its roots in multi-paradigm characteristics of C++. C++ is a kind of language, which could be specified as a multi-paradigm language, although it is often reduced to be only an OO language. As such, C++ is used to implement the systems designed according to OO methodology. In spite of that, non-object 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. This is achieved through two parallel analyses, commonality and variability, performed on both application and solution domain independently.

Commonality analysis concentrates on common attributes while the aim of the variability analysis is to parameterize the variation. Any commonality/variability pairing represents a paradigm in Coplien's MPD terminology.

The last step is to line up the commonalities and variabilities of the application and solution domain analysis, which leads us to use the "right" language features for the corresponding analysis abstractions. Coplien claims this mapping is often straightforward.

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, but introduces another issue. The implementation in a different programming language would probably require a new design; i.e. previous design would not make sense if the system was to be implemented in a different language.

Main idea of MPD is presented in [4]. More details with examples can be found in [3].

3.2 Intentional Programming

Programming languages with fixed syntax are limiting otherwise unlimited number of programming abstractions. Intentional programming group at Microsoft Research (led by Simonyi, the original developer of the MS Word and Excel) offers a solution to this problem as a new software development paradigm called *intentional programming*.

The idea behind *intentional programming* (IP) is that programming abstraction hosted by programming languages, which are limited in the sense of accepted notations (due to underlying grammars), could live well (and even better) without their hosts (i.e. programming languages). Such abstractions in IP are called *intentions*.

The solution proposed in IP is to have program represented by a so-called *intentional tree*. The intentional tree is similar to abstract syntax tree, but it would be misleading to call it by this name since there is no any syntax [11].

It is clear that IP needs (and has) a special and complex integrated programming environment. This sounds a bit like one of those clicking environments with various wizards, which are not at will to "real" programmers, but where we would be today if text editors never replaced punched cards. On the other hand, IP counts on a binary format for the program files, which is dangerous unless its exact 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 (IP environment can be extended with new parsers as libraries).

4 Conclusions

It doesn't seem that software development is at the edge of a revolution, but the evolution continues. We have tried to reveal some important directions towards which software development paradigms evolve.

Several new software development paradigms have been briefly presented and certain unifying tendencies have been identified among them. These unifying tendencies appear to culminate in multi-paradigm approaches. Two such approaches have been considered.

The need for more intentional programming abstractions (and their coexistence) is overcoming the capabilities of programming languages as traditional programming abstractions hosts. Intentional programming offers a possible solution to this problem.

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

Bibliography

1. 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 http://wwwtrese.cs.utwente.nl.
2. 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 http://www.cs.utexas.edu/users/schwartz.
3. James O. Coplien. Multi-paradigm design. In *Proc. of the GCSE '99 (co-hosted with the STJA 99)*, Erfurt, Germany, 1999. Published on CD, available at http://www.bell-labs.com/~cope.
4. James O. Coplien. Multi-paradigm design and implementation in C++. In *Proc. of the GCSE '99 (co-hosted with the STJA 99)*, Erfurt, Germany, 1999. Presentation slides and notes, published on CD, available at see http://www.bell-labs.com/~cope.
5. Krysztof Czarnecki. *Generative Programming: Principles and Tecniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models.* PhD thesis, Ilmenau Technical University, Germany, 1998. See http://www.prakinf.tu-ilmenau.de/~czarn.
6. 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 http://www.parc.xerox.com/aop.
7. Timothy Koschmann and Martha Walton Evens. Bridging the gap between object-oriented and logic programming. *IEEE Software*, 60:36–42, July 1988.
8. Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X, available at http://www.ccs.neu.edu/research/demeter.
9. Pavol Návrat. A closer look at programming expertise: Critical survey of some methodological issues. *Information and Software Technology*, 38(1):37–46, 1996.
10. Yoav Shoham. Agent-oriented programming. *Artificial Intelligence*, 60:51–92, 1993.
11. Charles Simonyi. Intentional programming—innovation in the legacy age, June 1996. Presented at IFIP WG 2.1 meeting, available at http://www.research.microsoft.com/ip.
12. Mária Smolárová and Pavol Návrat. Software reuse: Principles, patterns, prospects. *Journal of Computing and Information Technology*, 5(1):33–48, 1997.