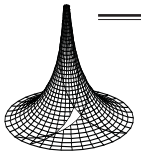

5/1 (2007), 171–182
tmcs@inf.unideb.hu
http://tmcs.math.klte.hu

Teaching
Mathematics and
Computer Science

Teaching multiparadigm programming based on object-oriented experiences

Zoltán Porkoláb and Viktória Zsók

Abstract. Multiparadigm programming is an emerging practice in computer technology. Co-existence of object-oriented, generic and functional techniques can better handle variability of projects. The present paper gives an overview of teaching multiparadigm programming approach through typical language concepts, tools in higher education. Students learning multiparadigm-oriented subjects would gain considerable expertise, which is highly needed by the industrial side in large-scale application development.

Key words and phrases: programming paradigms, teaching multiparadigm programming.

ZDM Subject Classification: B40, D40, P40, P50.

1. Introduction

In the software development process *abstractions* play a central role. An abstraction focuses on the essence of a problem and excludes the special details [6]. Abstractions depend on many factors: user requirements, technical environment, and the key design decisions. In software technology a *paradigm* represents the directives in creating abstractions. The notion of paradigm is widely discussed in [15]. The paradigm is the principle by which a problem can be comprehended and decomposed into manageable components [5].

There are several questions regarding abstractions and components. What types of categories can be established when the components are identified? What types of entities can be created and how? What should be created: procedures or

cooperating objects? Where are the boundaries between modules? What types of rules should be applied in composing abstractions? In practice a paradigm directs us in identifying the elements in which a problem will be decomposed and projected. The paradigm sets up the rules and properties, but also offers tools for developing applications.

Software development is fundamentally a human activity. Today it is largely supported by automated tools, but still considerably influenced by personal experiences, traditions, conventions and customs. These behavioural patterns have huge impact upon the software development process by determining the way programmers form abstractions. Many of these human factors are formed by higher education. The first programming language, abstraction methods, and practiced paradigms are imprinted in computer science students. Techniques and practices acquired in an early period tend to return later, even if the current problem would require a different approach. These impediments may hinder the programmers from producing outputs in the expected time frame and quality.

One possible solution is to prepare students to identify the nature of the problem to be solved and consciously choose the best-fitting tools and techniques. It is especially important to teach how different kinds of problems could be targeted using appropriate paradigms. Since complex problems usually have a multi-dimensional nature, in most cases they require multiparadigm approach researches.

2. Programming paradigms

The very first goal in computer science was the automation of computations. Computations which could not be executed by human resources were calculated automatically by a machine, therefore this effort was called *automatic programming*. Although at the beginning the hardware was unreliable and the software technology tools were rudimentary, the problems were still manageable at the expense of the paradigm subordinated to the hardware restrictions. The typical programming language was FORTRAN, which gained huge popularity despite its well known weaknesses, since the language provided high compilation efficiency: on average 10 machine code statements were generated from 9 FORTRAN commands, which was of major importance.

Since the hardware became cheaper and more reliable, the software architecture gained more importance in applicability, maintenance and innovation.

Earlier language constructs (like the infamous backward `GO TO`) were soon considered harmful [8]. *Structured programming* [7] prescribed sequences, branches, loops, procedures and emphasised the description of the adequate algorithm. In the second wave of imperative languages the attention was focused on data abstraction. Languages, like Pascal and their successors have sophisticated data abstraction constructs like enumeration, and features like strong typing, generics, tasks, exception handling, packages.

Object-orientation is evolved from block-structured languages. The Simula language introduced the first steps towards object-oriented programming and let a new paradigm appear. The object-oriented paradigm is based on identifying adequate data structures and methods upon them using the encapsulation principle. The languages supporting object-oriented programming contain special language constructs. Classes describe the objects with the same data structures and methods. Inheritance offers the possibility of building a hierarchy between similar classes, thus expressing the relationship between them.

The main feature of the object-oriented paradigm is the strong unity between data structures and methods inside a class, and the loose liaisons with other classes. The repeated features of similar classes (with similar data structures, behaviours and importance) can be grouped into a base class by generalisation. It is also possible to create new classes by specialisation. This can be achieved by extending the data structure and changing the behaviour of the base class by adding new methods. The obtained class hierarchy enables the definition of every important element of the program exactly one time facilitating further developments and code maintenance. Object-oriented programming is supported by strongly typed languages. These languages explore the polymorphic possibilities of dynamic linking during the run-time. The security of a construction is checked during compile-time. Class hierarchy is also a good design choice to express *subtype relationships*. According to the Liskov substitution principle [17] an object of type `T` could be substituted by an object of any subtype of `T`. However, subtyping and subclassing may diverge.

3. Multiparadigm programming

The object-oriented programming methodology is an extensively researched area, and at the same time it is the most widely used paradigm in the industry. In the early stage of the paradigm there were high expectations that the code written using the object-oriented paradigm would be shorter, clearer and

more maintainable. Although these great expectations where not baseless, and object-orientation proved to be effective in many cases, we experienced a number of its deficiencies. Problems with crosscutting concerns, and multidimensional separation of concerns [2] arise at design stage. Shortage of symmetric extensibility of class hierarchies – as described in the infamous expression problem [3, 21] blocks the development of effective extensible class libraries. Undeniably the object-oriented implementation techniques still cause efficiency problems.

As one can observe, in the past there was a paradigm turnout in about every 20 years. The turnout occurred every time when the size and the complexity of typical high-end applications exceeded the manageable extent of the older paradigm. Currently there are again many signs of a new paradigm turnout. The *aspect-oriented programming* [14] is already used in the industry to answer for the problems of crosscutting concerns and dangling code. The appearance of *generic programming* [1] to solve the expression problem is concretised in professional programs by the use of the C++ Standard Template Library.

Functional programming had a great history since LISP language, now we can experience a new renaissance. The ML language is an impure functional language, since it contains imperative elements (e.g. `let` statements). The ML dialects have introduced several language elements from the object-oriented programming paradigm. The Objective CaML dialect is based on the notion of objects, classes, class hierarchy, inheritance. JoCaML is an extension of the Object CaML for distributed, concurrent and mobile programming [9].

It is important to stress that a new paradigm does not completely replace the old one, but rather forms a new code organisation method above the old one. When object-orientation became popular, we did not disavow structured programming at all, we just introduced classes and inheritance hierarchy as a new way to organise the source code. Methods are still implemented according to the structural paradigm.

4. Teaching programming paradigms

Since the work of Dahl, Dijkstra, and Hoare, structured programming was the taught paradigm for undergraduate students. Naturally, the first programming language to be taught was a choice of Pascal, Modula (or Modula2), and C. Object-orientation introduced, sometimes optionally, for graduate students with the help of languages, like Simula67, and Smalltalk, was built on the experiences students received previously via structural languages. That approach let

the students learn structural programming and gradually acquire object-oriented principles.

Nowadays the focus is trended towards introducing object-oriented programming as the first paradigm. This is partly a result of the pressure coming from industry, wanting *production ready* programmers as soon as possible. The present first language is therefore either C# or Java (sometimes C++), which enables students to learn objects early, but having the danger to give the misleading suggestion that object-orientation is the only paradigm on stage.

Our approach is different. From the very beginning of the curricula we try to provide the students with the experiences of naturally coexisting programming paradigms. Our choice for the first programming language is C++, a multiparadigm language with well-separated procedural, object-oriented and generic features. Early codes are written in structured way. Writing a `hello world` program does not require the introduction of *output stream objects*, etc... Later, objects are introduced naturally, when more complex programs require their own data structures. Students can also accumulate generic programming knowledge by extensive use of the Standard Template Library.

Functional programming is another challenge to the first language to be taught. Since modern functional programming style is very close to the mathematical way of thinking, it is quite often proposed as the first paradigm for students. We do not follow this direction, however, some functional style solution is used in early C++ examples. Recursion is practiced from the beginning, and higher order functions are introduced in connection with Standard Template Library function objects. Our students are provided with larger mathematical foundations (about 30 percentages of all the credits in B.Sc.) than the average practice in computer science education. Therefore they are more receptive for the functional paradigm.

As mentioned earlier, the new paradigms can easily coexist with the older ones. We will illustrate this by the properties of the constructive interaction of several pairs of paradigms. Aspect oriented programming has the same basic structure as the object oriented one. The aspects decrease the code repetition inside classes and improve the modularity of a program. The paradigm is more efficient in following the positive and negative changes. Generic programming also gathers principles from the object-oriented style. It has several common basic constructs like abstract datatypes, classes, functors. Functional programming also presents similarities with object-oriented programming. It provides abstract

datatypes, classes, encapsulation, subtyping, basic structures. The generic paradigm also coexists with functional programming. The Standard Template Library provides functions, which are similar to the higher order functions of the functional programming style or to the functions parameterised by strategies. There are other common concepts in the two paradigms like generic data structures and functions, or parametrical polymorphism.

Higher education should accentuate the theoretical and practical teaching of the already widespread new paradigms and their supporting language tools and programming environments too. Students can deepen their knowledge-base by acquiring new programming paradigms and styles like generative programming, functional programming, aspect-oriented programming, logic programming. As a result they will be able to combine all these paradigms in a creative way and they will be able to use new tools and new technologies in their future work. It is very important to mention that a new paradigm never sets the previously gained programming experiences aside. Structural programming has overtaken the notions of the earlier imperative languages. The object-oriented paradigm does not cancel structural programming in the implementation of methods. The alternation of paradigms incorporates all the already existing tools, methods, experiences in a higher structural unity.

A typical example can be seen in generic programming. The generic algorithms are separated from the data structure contrary to the object-oriented principles. At the same time some data structures are implemented as template classes with well-defined public interfaces and private implementations. However even some algorithms appear as classes (functors). The aspect-oriented paradigm uses in the same way the notion of aspects, since aspects are very closely related to classes. Generative programming and object-oriented programming can safely work together using *mixin* [20] technique.

```
template <typename T1, typename T2>
class Mixin : public T1, public T2 { ... };
```

A mixin type inherits from their own template parameter, therefore it can export the public interface of the parameter type. This makes the construct extremely useful when components must be assembled automatically [2, 24].

There are a number of examples linking the generative and the functional paradigm. The `boost::bind` library [13] implements full functional programming environment in the compilation time of C++ programs. It supports arbitrary function objects, functions, function pointers, and member function pointers, and

is able to bind any argument to a specific value or route input arguments into arbitrary positions.

Functional programming is a paradigm that treats computations as evaluation of mathematical functions, so it is very appropriate to the mathematical ways of thinking. First-year students enter the university with a large mathematical knowledge, for this reason functional programming is appropriate for the first programming paradigm to be presented in the university curricula. The theoretical background of functional languages can form the thematic of more courses, which can be of lambda calculus, compiler construction, etc. Theoretical knowledge is indispensable for further researches or further compiler developments.

Special courses integrate the teaching of functional programming with commercial languages (e.g. Common LISP). Others combine the teaching of functional program design with modern object-oriented languages like C++, Java, Pizza. Some courses introduce the functional style in pedagogical programming environments, where this paradigm is used for implementation of programs with pedagogical relevance.

More advanced topics can form the subject of special courses, even a complete subject curricula can be formed out of these. Here we enumerate several advanced topics with their main issues. Special advanced compiler construction courses present the extensions and libraries of the modern functional languages like Haskell or Clean, where tracing, debugging, heap profiling are studied. Issues like combinator libraries, parsing libraries and grammar analysis are even more advanced concepts. Other types of special courses are formed at the boundaries of different informatics subjects (for example type-safe database handling in Haskell DB). Games and animations are developed as functional reactive animations in the framework of several computer science practical courses. There are several theoremproving tools developed in functional programming languages which form the subject of logic-oriented courses.

Patterns and skeletons are seen in monadic or in ObjectIO special programming courses, where arrows, meta programming (Template Haskell), advanced type classes (with multi-parameter, functional dependencies), abstract data structures are also studied. Higher order functional programming based on skeletons and evaluation strategies can solve high complexity problems and also distributed computations in GRID systems. Nevertheless, it is quite prevailing to use complementary paradigms in developing programs for multiparadigm based systems in distributed environments [11].

Since we introduced the multiparadigm-based education approach described above, we receive continously positive feed-back from the industry. Multinational software companies, like NOKIA, Siemens, Ericsson, as well as research institutes show growing interest for our graduated students. For instance, the number of our graduated students employed by Nokia Hungary Ltd. is increased by 45 percent in this period. These graduated students were hired mainly in those areas which require specific multiparadigm knowledge, i.e. generative programming skills (in NOKIA), and functional language skills (Ericsson) [16].

Students also recognised the importance of these fields. On M.Sc. level they likely choose subjects related to multiparadigm knowledge, like *Advanced C++* on generative programming and *Functional programming*.

5. When OO and other paradigms are contradictory

There are a few important situations where teaching generative programming requires extreme care since the rules contradict the usual object-oriented experiences.

One such situation is the handling of *subtype* relationships between parametric types. According to the Liskov substitution principle [17] an object of type `T` could be substituted by an object of any subtype of `T`. In pure object-oriented languages subtyping is often implemented using subclasses. This does not work with parametric types.

```
class Base { ... };
class Derived : public Base { ... };
template <typename T> class Gen { ... };

Gen<Base>     gb;
Gen<Derived>  gd;
```

In the above example `Derived` is a subtype of `Base`, but this does not hold for `Gen<Derived>` and `Gen<Base>`. Type parameters are *invariants* for subtype relationships. This, otherwise logical, rule often makes problem for students who are unexperienced in generative programming.

It is interesting, that the similar situation involving arrays are not handled in such a consistent way. Both Java and C++ allows the polymorphic usage of arrays, i.e. arrays are *covariant* constructions. The result in both languages are the loss of static type checking possibilities.

6. Related work

Research of multiparadigm programming has a long history. In 1986 IEEE published a special issue on multiparadigm programming [10].

The [23] paper discusses a compositional approach to multiparadigm programming. Since "most of the experienced programmers are confined to their favorite language's one paradigm", the author propagates multiparadigm approach using composition of a collection of single-paradigm programs.

An opposite opinion can be found in [22]. The paper describes that "most programming languages courses have students use several distinct languages to gain experience with different language paradigms and implementation issues". This practice gives some real experience in a number of languages, but the time spent on learning new languages and environments necessarily reduces the capabilities to learn the new paradigm itself. Therefore the author argue for using a single multiparadigm programming language, called GED. GED supports the imperative, functional, logic, and object-oriented paradigm. Experiences with GED are described in [19]. G language, the predecessor of GED is reviewed in [18].

One of the most referred multiparadigm programming language is the LEDA language [4]. LEDA is a general purpose language, which was designed both as a research tool and as a teaching environment. The language is based on imperative and object-oriented features, like assignment, class definition and single inheritance with polymorphism. For functional programming LEDA implements functions as first-class values including their dynamic creation capturing actual environment. Logic programming is supported via relations – data types allowing definitions of facts and inference rules. LEDA is a statically checked, strongly typed programming language which realized a larger set of paradigms, but it is still considerably smaller than C++.

Multiparadigm software design and its implementation in the C++ programming language are deeply investigated by James Coplien [6]. One of his most important conclusions is that different kind of domain problems should be targeted using different programming paradigms. The domain analysis, especially identifying positive and negative variability, helps to select the most appropriate paradigm.

7. Conclusion

The evolution of paradigms involves the evolution of teaching methodologies, tools, programming languages and environments. The academic and the industrial research provide the future challenges for the higher education of programmers. The new paradigms can coexist quite easily with the old ones producing many new valuable properties. The multiparadigm environments and tools give the opportunity to use more programming styles inside one application and to experience the multi-faceted property of one programming construct or feature.

The higher education of computer science has great responsibilities in preparing the programmer students when responds to the challenges provided by the industrial side. This involves the teaching of all those new technologies, paradigms, languages, tools which are not yet widely used in industry, but they have already proved their applicability. However, the new technologies, paradigms are not against the currently widely used ones. The new issues reflect the boundaries of the older ones and they enable a deeper understanding and applicability of them. Teachers continually adjust their lectures to the new technology requirements. The efforts for developing new topics according to the state-of-the-art will be exploited by the students in their future work.

References

[1] M. H. Austern, *Generic Programming and the STL*, Addison-Wesley, 1999.

[2] L. Bergmans, M. Aksit, Composing Crosscutting Concerns Using Composition Filters, *Communications of the ACM* **44**, no. 10 (2001), 51–57.

[3] K. B. Bruce, Some challenging typing issues in object-oriented languages, in: *Electronic Notes in Theoretical Comp. Sci.*, Vol. 82, (V. Bono and M. Bugliesi, eds.), Elsevier, 2003.

[4] T. A. Budd, T. P. Justice, R. K. Pandey, General-purpose multiparadigm programming languages: an enabling technology for constructing complex systems, *First IEEE International Conference on Engineering of Complex Computer Systems* (1995), 334–337.

[5] L. Cardelli, P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *ACM Computing Surveys* **17**, no. 4 (1985), 471–522.

[6] J. O. Coplien, *Multi-Paradigm Design for C++*, Addison-Wesley, 1998.

[7] O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, *Structured Programming*, Academic Press, 1972.

[8] E. W. Dijkstra, Go To Statement Considered Harmful, *ACM* **11**, no. 2 (1968), 147–148.

[9] C. Fournet, F. Le Fessant, L. Maranget, A. Schmitt, *The JoCaml language beta release*, Documentation and user's manual, INRIA, 2001.

[10] B. Hailpern (ed.), Special issue on Multiparadigm Languages and Environments, *IEEE Software* **3**, no. 1 (1986), 6–77.

[11] Z. Horváth, V. Zsók, P. Serrarens, R. Plasmeijer, Parallel Elementwise Processing in Concurrent Clean, *Mathematical and Computer Modelling* **38**, Elsevier (2003), 865–875.

[12] P. Hudak, S. Peyton Jones, Ph. Wadler, B. Boutel, J. Fairbairn, J. Fasel, K. Hammond, J. Hughes, Th. Johnsson, D. Kieburtz, R. Nikhil, W. Partain, J. Peterson, Report on the Programming Language Haskell, *ACM SigPlan Notices* **27**, no. 5 (1992), 1–164.

[13] B. Karlsson, *Beyond the C++ Standard Library: An Introduction to Boost*, Addison Wesley Professional, 2005.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, J. Irwin, Aspect-Oriented Programming, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland*, Springer-Verlag, LNCS 1241, (1997), 220–242.

[15] T. S. Kuhn, *The Structure of Scientific Revolutions*, 3nd ed., Univ. of Chicago Press, Chicago and London, 1996.

[16] J. Kurtz and Z. Porkoláb, Cooperative Work and Learning, *Proceedings of International Technology, Education, and Development Conference (INTED 2007)*, Valencia, Spain (2007).

[17] B. Liskov and J. M. Wing, A Behavioral Notion of Subtyping, *ACM Transactions on Programming Languages and Systems (TOPLAS)* **16**, no. 6 (1994), 1811–1841.

[18] J. Placer, The Multiparadigm Language G, *Computer Languages* **16**, no. 3–4 (1991), 235–258.

[19] J. Placer, The Promise of Multiparadigm Languages as Pedagogical Tools, *Proceedings of the ACM conference on Comp. Sci.* (1993), 81–86.

[20] Y. Smaragdakis, D. S. Batory, Mixin-Based Programming in C++, in: *Proceedings of Net. Object Days*, 2000, 464–478.

[21] P. Wadler, *The expression problem, Posted on the Java Genericity mailing list*, 1998.

[22] D. S. Westbrook, A Multiparadigm Language Approach to Teaching Principles of Programming Languages, 29^{th} *ASEE/IEEE Frontiers in Education Conference* (1999), 11b3–14.

[23] P. Zave, A Compositional Approach to Multiparadigm Programming, *IEEE Software* **VI**, no. 5 (1989), 15–25.

[24] I. Zólyomi, Z. Porkoláb, An extension to the subtype relationship in C++ implemented with template metaprogramming, *Proceedings of GPCE 2003*, Springer-Verlag, LNCS 2830 (2003), 209–227.

ZOLTÁN PORKOLÁB and VIKTÓRIA ZSÓK
EÖTVÖS LORÁND UNIVERSITY
FACULTY OF INFORMATICS
DEPARTMENT OF PROGRAMMING LANGUAGES AND COMPILERS
H–1117 BUDAPEST
PÁZMÁNY PÉTER SÉTÁNY 1/C
HUNGARY

E-mail: gsd@elte.hu

E-mail: zsv@inf.elte.hu

(Received February, 2007)