Beyond 2000, Beyond Object-Orientation

L. Kozma, Á. Frohner, T. Kozsik, Z. Porkoláb
Department of General Computer Science,
University Eötvös Loránd, Budapest
E-mail: kozma@ludens.elte.hu, szamcsi@elte.hu, kto@elte.hu, gsd@elte.hu

2001-05-04

1 Abstraction and Paradigm

In the very centre of software design is the idea of abstraction. Abstraction means to focus on the general and put the specific aside. We try to formulate what is common in problem domain and negligate differences. A paradigm is a set of conventions we use to understand the world around us [1]. In software, paradigms shape the way we formulate abstractions. What is the world made of? Do we divide the world into procedures? Data records? Modules? Or we divide it into classes? Co-operating data structures and generic algorithms? Classes and aspects? Components? In the practice, a paradigm defines us rules, conventions, sometimes tools make us be able to divide the problem into pieces that small enough to understand, code and test. Here is some examples of paradigms:

1.1 Procedural programming

A"Decide which procedures you want; use the best algorithms you can find." Here the focus is on the processing - the algorithm needed to perform the desired computation. The programming languages supporting procedural programming have adequate tools define and call procedures, passing parameters in a various way and handling return value.

1.2 Modular programming

A set of related procedures with the data they manipulate is often called a module. Modular programming has the main principle data hiding. Here the "Decide which modules you want, partition the program so that data is hidden within modules" .

2 Object-Oriented Programming

In the 90s the leading software paradigm was object-orientation. Classes and objects are the main abstractions. Cohesion and coupling are the main partitioning criteria: we form objects from similar responsibilities and/or cohesive

data structures. We use a special programming language construct: class to describe all objects with the same data structure, functionality and role. This is the most important feature of object-oriented construct: very strong binding between data and operations within a class and as week connections between different classes as possible. A class can act as a user-defined type, provides a full set of operations. Similarity between classes can be made explicit by using inheritance-hierarchy.

In the classical schoolbook example see the classes of vehicles. Classes Car and Truck are derived from a common base class, Vehicle. The common data members and algorithms are collected in Vehicle, additional data members can appear in derived classes. Some operations, like ... are used but not defined in Vehicle - the exact definitions appear in the derived classes. This is handled by the use of virtual functions - an appearance of (inclusion) polymorphism.

3 Generic Programming

Object-orientation has many advantages in a wide area of problem domains. In the same time there are certain domains where object- orientation does not effective or even fails. Let see an other school-book example: we have to implement a set of stacks, each have different types of elements: one stack has integer elements, other has doubles, etc... Easy to see: there is many common in the different type of stacks: how to allocate space for the elements, how handle the stack pointer, how to check the space has left at push operation and whether the stack is empty at pop.

Logical (and very object-oriented) way of thinking to create a base class - a general stack - where we are grouping all the data and algorithms, which is common for all stacks. Later we will create concrete stacks via inheritance - one for storing integers, other for doubles, etc... Here is the skeleton of the classes:

```
class stack {
public:
    virtual void push( ? );
    virtual ?    pop();
    // ...
private:
    int capacity;
    int stack_ptr;
    ?   *elems;
    // ...
};

class intStack {
public:
    void push(int);
    int  pop();
};
```

The problem lays in the points we marked with the question-mark. How we should store the elements without knowing their exact type? How to define

the signature of pop() and push() operations? If we dig down to the deep of the problem, we recognise the fundamental difference of the two examples. In the vehicle example the basic data-structure of the vehicles was common. The differences were in additional data members in the derived classes, and in certain algorithms. This was positive variance: doesn't hurt the "contract" which defined "vehicle" type. On the other hand differences between the stacks (and the general stack) show negative variability: it contradicts the assumptions that underlie the abstraction of (general) stack. These kinds of differences are handled only with serious difficulties in current object-oriented languages.

Experienced C++ programmers could choose different way to implement stacks. With the technique called templates - i.e. parametrising with types - one can easy create the template class stack:

```
template <typename T>
class stack
{
public:
    virtual void push( T );
    virtual T    pop();
    // ...
private:
    int capacity;
    int stack_ptr;
    T   *elems;
    // ...
};
```

Here we found the commonality in the algorithms. Creating stacks, push and pop operations share the same code - except the type parameter T. Therefore the idea is to implement these operations as generic algorithms which can be applied for all the types. Not only those types we already know, but they can work with classes we develop later. This is also a kind of polymorphism - parametric polymorphism.

Creating foundation libraries in this way has clear advantages. The implementor has to write O(n+k) size of code for k algorithms working on n classes. New algorithms or classes can be added with linear effort. This could be significantly better than binding classes and algorithms together in object-oriented way. Even better generic programming - as the method above is called - can most cases handle negative variability with a technique called template specialisation. Based on the work of Alexander Stepanov and David R Musser, Generic Programming became very popular in the last few years [2]. Standard Template Library in C++, JAVA Collection Framework and others are now part of the language standards [3]. Compile-type type checking, automatic instantiation, better performance (comparing to run-time solutions, like JAVA reflection or Smalltalk) and more simple code has made C++ STL a success and has led SUN Microsystems to propose inventing templates in JAVA languages [4].

4 Aspect-Oriented Programming

Traditionally, programs involving shared resources, multi-object protocols, error handling, complex performance optimisations and other systemic, or crosscutting concerns have tended to have poor modularity. The implementation of these concerns typically ends up being tangled through the code, resulting in systems that are difficult to develop, understand and maintain. Aspect-oriented programming [6] is a technique that has been proposed specifically to address this problem. One can separate the above mentioned concerns on source level into aspects and weave them into the original code using an automated tool before compilation. The granularity of the weaving points and the language of the aspect is determined by the actual implementation of the weaver.

To demonstrate the separation of concerns we have chosen a simple example of monitoring the state of a system. The programmer should deal only with the functional code of the system and separate the monitoring code. This separation allows the ”instrumentation” of the system without human modification of the original source code. For simplicity the system consists only the ”Variable” class (in Java language [7]):

```
public class Variable {
   private int v;
   public Variable() { v = 0; }
   public int getV() { return v; }
   public void setV(int v) { this.v = v; }
}
```

The monitoring code should report the value of the enclosed variable before and after every method call. How can we preserve the original class and still be able to get through the private visibility? In an object-oriented solution we would have to let others to directly access the - not any longer - private variable or place other kind of hooks inside the original source, which would decrease the readability and maintainability.

The monitoring concern will be implemented by using a general-purpose aspect-weaver called AspectJ [5]. This tool has been developed in the last couple of years at Xerox Palo Alto. In AspectJ, aspects are programming constructs that work by crosscutting the modularity of classes in carefully designed and principled ways. So, for example, a single aspect can affect the implementation of a number of methods in a number of classes. We modify the ”Variable” class by using the ”Trace” aspect:

```
aspect Trace {
    advise * Variable.*(..) {
        static before {
            System.out.println("Entering " +
            thisJoinPoint.methodName + " v=" + thisObject.v);
        }
        static after {
            System.out.println("Exiting " +
            thisJoinPoint.methodName + " v=" + thisObject.v);
        }
```

```
    }
}
```

Weaving together the original class and this aspect will produce a source, which is functionally equivalent to the following code:

```
public class Variable {
  private int v;
  public Variable() { v = 0; }

  public int getV() {
    int thisResult;
     System.out.println("Entering  "  +   "getV"  +  "  v="  +
this.v);
    thisResult = v;
    System.out.println("Exiting " + "getV" + " v=" + this.v);
    return thisResult;
  }

  public void setV(int v) {
     System.out.println("Entering  "  +   "setV"  +  "  v="  +
this.v);
    this.v = v;
    System.out.println("Exiting " + "setV" + " v=" + this.v);
  }
}
```

Placing crosscutting concerns into one aspect has the clear advantage of improving comprehension of the source and avoiding repetition of code segments in similar classes. The aspect- oriented paradigm allows different languages for the aspects, thus an implementation might choose the most appropriate one for a specific concern. The skeleton of the data structures and the basic functionality can be designed and implemented by the object-oriented paradigm, while specific concerns are described in a language, which is closer to the problem domain (e.g. path- expressions for concurrency issues).

5 Functional Programming

The previous section demonstrated that modern functional languages support generic programming and parametric polymorphism in a highly developed way. Moreover, there are still other lessons to learn from functional programming. In the following two questions will be discussed. First program correctness will be addressed, then the advantages of higher-order languages will be summarised. Finally we emphasise that the object-oriented and the functional programming paradigms do not exclude each other, e.g. they can be incorporated within the same programming language.

5.1 Program Correctness

Writing correct programs is very hard - every programmer is aware of this disappointing fact. Formal methods and testing can be applied to increase trust

in the correctness of a software product, but in case of a large piece of software it is often hard to make use of them. This can be the result of the heavy complexity of the code to be investigated and the huge gap between the problem specification and the implementation. Both the object-oriented and the functional programming paradigms try to overcome these difficulties.

According to the object-oriented programming paradigm, each implementation decision is encapsulated inside a small component, an object. The complexity of the program can be reduced significantly this way. (Section AOP explains what to do if it is not the case.) Furthermore, the (interfaces of and the relations between the) objects correspond to the entities of the "real world", hence the solution of a problem is obtained inside the problem domain, which reduces the gap between specification and implementation.

In the functional programming paradigm a program consists of a set of function definitions and an expression, the value of which is to be computed. This style of programming can be much more regarded as writing "executable specifications" rather then "programs". Specification and implementation are not far from each other any more in this paradigm. See for example this very natural implementation of the quicksort algorithm in the functional language Clean [Clean]. The program text is a direct representation of the ideas we have on quicksort (see a more complex example in [21]). E.g. when sorting a non-empty list, the beginning of the result is obtained by collecting and sorting the elements smaller than a dedicated element, and the end of the result is obtained by collecting and sorting the non-smaller elements.

```
qsort [] = []
qsort [x:xs] = qsort [y \\ y <- xs | y < x]
               ++ [x] ++
               qsort [y \\ y <- xs | y >= x]
```

Functional programming also increases program correctness via avoiding "side-effects". Pure functional programming forces the preservation of referential transparency: the value of an expression is independent of when it is evaluated, since the result of a function is uniquely determined by its arguments. This proves to be a great advantage when one wants to reason about program correctness. Such reasoning requires a very simple mathematical apparatus. The application of a function can always be replaced by the definition of the function with the actual arguments replacing the formal arguments. (This is called the principle of uniform substitution.) This, and the fact that the semantics of functional programs can be given with relatively simple computational models (lambda calculus, graph rewrite system) even enables the use of semi-automatised proof systems (e.g. [8]).

5.2 Higher-order languages

Functions are first-class citizens - this is a principle in functional programming. Manipulations with computations are possible in many programming languages: recall for example the function pointers of C or the subprogram types of Modula 2 or Ada 95. However, these languages do not consider these possibilities such a big goal as functional languages do. In functional languages the value of an expression can also be a function and act as a parameter or a

result in a function application. But partial application (also called "currying") is possible as well. Regard the following example expression, which increases the elements of a list by two. The well-known map function takes a unary function as its first argument and applies it elementwise to its second argument, a list. The function parameter here is the curried addition operator:

```
map ((+) 2) [1,2,3,4]
```

This flexibility in managing functions grants high expressive power to a programming language, increasing its capability for abstraction. As a completion let us remark on a fairly interesting phenomenon related to handling of functions as first- class entities. In functional languages the construct "lambda abstraction" is used to define functions "on the spot", right where they are applied. This concept reappears in Java: its "inner classes" are applicable for a very similar purpose.

5.3 About the co-existence of OOP and FP

Many words have been written so far on how functional languages can support the object-oriented paradigm. These ideas show up in most modern widely used functional programming languages, like ML, Haskell or Clean. The subsequent paragraphs will recall some of them with the intention of giving a foretaste.

Abstract data types can be implemented in many ways [9], [10]. One way is with the use of "type classes" [11], [12], which are similar to e.g. interfaces in Java. Implementation is assigned to a type class with instantiation allowing several alternative representations. Record types with functional components provide another facility: when used in combination with existential types, they can even give way to inhomogeneous data structures over the alternative representations. This latter solution enables powerful, object-level dynamic binding of operations.

Encapsulation is made possible by modules and the so-called "abstract datatypes" [12], [13]. Subtyping and inheritance can be supported both with type classes and (extensible) records. Dynamic typing is often added to statically typed languages as "dynamics" [14].

We can conclude that functional and object-oriented programming can be integrated within one framework. This marriage endows object-orientation with valuable properties and opens up a promising future now, when efficiency problems, the major argument against functional programming seems to disappear [15].

6 Component-Oriented Software Technology

The object-oriented technology has already been accepted in the software community as offering the most powerful tools for software development. "The Unified Modeling Language (UML) is becoming the standard palette used by software designers to paint their thoughts [16]. UML is

- an architecture-centric and use case driven program adopting and iterative development methodology.

- an excellent vehicle for communicating the system design among various stakeholders.
- a graphic language for specifying, solving a given problem and for documenting the given solution.
- a general-purpose modeling language that provides an extensive conceptual base for a broad spectrum of application domains. It is broad enough to cover real-time systems as well. Real- time modeling can be fully accommodated by specializing appropriate base concepts.
- the standard modeling language for software-intensive systems like Web applications.

Now beyond 2000 researchers are suggested to go beyond objects in search for better software technology [17]. The notion of component is more general than of an object, and in particular may be of either much finer or coarser granularity. From methodological aspect a component is a component because it has been designed to be used in a compositional way together with other components. It is designed as a part of a framework of collaborating components. Valid examples of components may be functions, macros, procedures, templates or modules. At a software technology level, the vision of component-oriented development is a very old idea, which was already present in the first developments of structured programming and modularity. An application can be viewed simultaneously as a computational entity that delivers results and as a construction of software components that fit together to achieve those results. The integration of these two aspects is not straightforward, since their goals may conflict. For example concurrency mechanism, which are computational, may conflict with inheritance, which is a compositional feature [18,19].

A component is a static abstraction with plugs. A software component static because it is a long-lived entity stored in a software base, independently of the applications in which it has been used. Abstraction means that a component is encapsulated. "With plugs" means that there are well-defined ways to interact and communicate with the component.

Component-oriented software development requires a change of mind-set, change of methodology and requires new technological support at the same time. A good example for these changes is a framework for component-based Client/Server computing with CORBA [20]. The most ambitious middleware undertaking ever, CORBA manages every detail of component interoperability, ensuring the possibility of interaction-based systems that incorporate components from multiple sources. A service specification of an object is completely separated from its implementations. CORBA is able to provide a self-specifying system that allows the discovery of other objects on the network. CORBA objects can exists anywhere on a network, their location is completely transparent. Details such as the language in which an object is written or the operating system on which it currently runs are also hidden to clients. The interface is the only consideration a client make when selecting a serving object.

7 Conclusion

Old and new paradigms could and should live together. Multi- paradigm programming promises to use all the advantages of different paradigms:

OO: mature large-scale design, good modularity

GP: parametrised type constructs, good handling of negative variability

AOP: crosscutting concerns

FP: formal-proof, data-flow optimisation

Components: easy deployment and configuration

We should also emphasize how are these paradigms are getting closer to the human way of thinking. 'Give me a complex language, where I can express my thoughts!'

References

[1] James O. Coplien: Multi-Paradigm Design for C++, Addison- Wesley, 1998.

[2] David Musser, Atul Saini: STL Tutorial and Reference Guide, Addison-Wesley, 1996.

[3] Bjarne Stroustrup: The C++ Language, Addison-Wesley, 1997.

[4] Philip Wadler: GJ, A Generic Java, Dr.Dobb's Journal, 2000.feb, pp. 23-28.

[5] Xerox, Palo Alto Research Center: AspectJ Home Page.
http://www.aspectj.org

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

[7] Sun Microsystems Inc.: The Source for Java(tm) Technology,
http://www.javasoft.com

[8] de Mol, M. and van Eekelen, M. (1999), A Proof Tool Dedicated to Clean. Selected Papers of Applications Of Graph Transformations With Industrial Relevance, AGTIVE '99, Kerkrade, The Netherlands, Springer-Verlag, LNCS. To Appear.

[9] Cardelli, L. and Wegner, P. (1985), On Understanding Types, Data Abstraction, and Polymorphism. ACM Computing Surveys, Vol 17 n. 4, pp 471-522, December 1985.

[10] Läufer, K. (1995), Type Classes with Existential Types. Journal of Functional Programming, Vol 6 n. 3, pp 485-517, May 1996.
http://www.math.luc.edu/~laufer/publications.html

[11] Hudak, P., Peyton Jones, S., Wadler, Ph, Boutel, B., Fairbairn, J., Fasel, J., Hammond, K., Hughes, J., Johnsson, Th., Kieburtz, D., Nikhil, R., Partain, W. and Peterson, J. (1992), Report on the Programming Language Haskell. ACM SigPlan Notices 27, (5), pp. 1-164.

[12] Software Technology Research Group (University of Nijmegen, the Netherlands), Home of Clean, http://www.cs.kun.nl/~clean

[13] Turner, D. A. (1985), Miranda: a non-strict functional language with polymorphic types. Proc. of the Conference on Functional Programming Languages and Computer Architecture, ed. J.P. Jouannaud, Nancy, France. Springer Verlag, LNCS 201, pp 1-16.

[14] Pil, M.R.C. (1999), Dynamic types and type dependent functions. Proc. of Implementation of Functional Languages (IFL '98), London, U.K., Hammond, Davie and Clack Eds., Springer- Verlag, LNCS 1595, pp 169-185.

[15] Plasmeijer, M.J. and M.C.J.D. van Eekelen (1993), Functional Programming and Parallel Graph Rewriting, Addison Wesley, Reading MA. ISBN 0-201-41663-8.

[16] UML in Action Comm. Of the ACM Vol. 42. No. 10 1999.

[17] O. Nierstrasz, L. Dami, Component-Oriented Software Technology In: Object-Oriented Software Composition Edited by O. Nierstrasz and D. Tsichritzis, Prentice Hall London, 1995

[18] S. Matsuoka, K. Taura A. Yonezawa: Highly Efficient and Encapsulated Re-use of Synchronisation Code in Concurrent Object- oriented Languages, In: Proc. of OOPSLA´93, ACM SIGPLAN Notices , vol. 28, no. 10, Oct. 1993, pp. 109-129.

[19] L. Blum, L. Kozma: Implementation Problems of a new Synchronization-Scheme, In: Proc. of the Fifth Symposium on programming Languages and Software Tools, Jyvaskyla, 1997, pp. 24-36.

[20] S. M. Lewandowski, Frameworks for Component-Based Client/Server Computing ACM Computing Surveys, Vol. 30, No. 1, March 1998, pp. 3-27.

[21] Horváth Z. - Zsók V. - Serrarens, P. - Plasmeijer, R.: Parallel Elementwise Processing in Concurrent Clean. To appear in: Proceedings of the 5th International Conference on Applied Informatics, Eger, Hungary. Jan. 2001.