JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 6, No. 2, Special Issue OOPS Track at SAC 2006, February 2007

Method overloading and overriding cause distribution transparency and encapsulation flaws

Antoine Beugnard, ENST Bretagne, Computer Science Department, CS83818, F-29238 Brest cedex 3
Salah Sadou, Valoria, Université de Bretagne Sud, F-56000 Vannes

Based on an experiment using three languages under .NET, this paper argues that the semantic differences between these languages regarding method overloading and overriding give rise to significant complexity and break encapsulation. We first recall the various interpretations of overriding and overloading in object oriented languages through what we call *language signatures*. Then, we perform an experiment with .NET components coded in different programming languages in order to observe the global behavior. From this, we show that overriding and overloading are not compatible with a key property of components: encapsulation. We conclude that, in the current state of the art, in order to build predictable assemblies, components must expose their internal structure! We propose a solution to this problem.

1 INTRODUCTION

For the past 25 years, object technologies have been spreading in programming languages, development method, and modelling technique. Almost all non-object languages have their "OO-extension": ADA, Caml, C and even COBOL. Analysis and design methods often rely on a unifying modelling language (UML) that relies itself on objects. It seems that the hope that object technology is the silver bullet solution has gone. Reusing is not so simple and assembling objects remains complex. The need for a better engineering process leads to the development of software by assembly of components. The idea to develop software systems like electronic ones is not new [12], but only recently have we seen component models being industrially used (CCM, DCOM, EJB, Fractal, .NET). All are implemented with objects.

We demonstrate that the various behaviors that object programming languages have relatively to method overriding and overloading semantics misfit with one of the main features of components: encapsulation.

This paper is organized as follows. We first recall in the next section some of these behaviors, and how component interfaces can be described with contracts. In sections 3 and 4 we present the experiments that demonstrate that the assembly of heterogeneous components cannot be predictable if their interface does not expose enough implementation details. Before concluding, we give some suggestions to solve this problem.

Cite this article as follows: Antoine Beugnard, Salah Sadou: *Method overloading and overriding cause distribution transparency and encapsulation flaws*, in Journal of Object Technology, vol. 6, no. 2, Special Issue OOPS Track at SAC 2006, February 2007, pp. 33–47
http://www.jot.fm/issues/issues_2007_2/article2

2 OVERLOADING AND OVERRIDING

The main contribution of object programming is, from our point of view, the easy access to dynamic dispatching that makes possible the development of frameworks. Frameworks can be easily extended and specialized thanks to dynamic dispatching. Being easy to use and safe, thanks to type systems (unlike in C where function pointers could be used to implement it), object-oriented languages allow the development of more flexible and reusable applications. However, the benefit of this "late-binding" is moderated by the time required to apply the lookup procedure that implements the dynamic dispatching and by the fact that any user needs to know the extension points (methods) of the framework s/he uses.

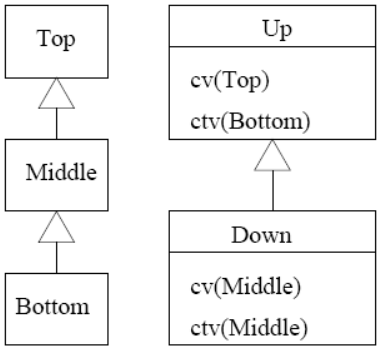


Figure 1: How will this model behave?

This powerful mechanism is, however, difficult to use since the *overriding intent* expressed in a diagram such as in figure 1 is interpreted differently by object programming languages. As it has already been published in [5] the interactions between overloading and overriding lead to very different behaviors. For instance, overriding can be invariant, covariant or contravariant, but nothing prevents a language from accepting these three possibilities at the same time. The late-binding resulting from the detection of this overriding can be simple, the most frequent case, or multiple, like in CLOS. Overloading can be allowed or not.

Interpretations

In order to show the different interpretations we have implemented the model of figure 1 in many languages. Then we called all possible cases and built what is called a language signature[1] [5]. On the basis of the following definition of receivers and parameters:

[1] The language signatures of many languages (including ADA, Java, Smalltalk, CLOS) and the source code of the experiment can be seen in [4].

```
//receivers
  Up u, ud;
  Down d;

//parameters
  Top t = new Top();
  Middle m = new Middle();
  Bottom b = new Bottom();
```

table 1 shows how we elaborated a language signature.

– First test u := new Up();	– Second test d := new Down();	– Third test ud := new Down();

u.cv(t);
u.cv(m);
u.cv(b);
	d.cv(t);
	d.cv(m);
	d.cv(b);
		ud.cv(t);
		ud.cv(m);
		ud.cv(b);

u.ctv(t);
u.ctv(m);
u.ctv(b);
	d.ctv(t);
	d.ctv(m);
	d.ctv(b);
		ud.ctv(t);
		ud.ctv(m);
		ud.ctv(b);

Table 1: Signature elaboration

Table 2 shows the signatures of C++ [17], Visual Basic [15], C# [11] and Java [3] since we use them in the experiment of section 3 and section 4. Each cell contains the name of the class where the applied method was found. The word "Error" denotes a compilation error.

	C++			VB	C#	Java
calls	u	d	ud	d	d	d
cv(t)	Up	**Error**	Up	**Up**	**Up**	**Up**
cv(m)	Up	Down	Up	Down	Down	Down
cv(b)	Up	Down	Up	Down	Down	Down
ctv(t)	Error	Error	Error	Error	Error	Error
ctv(m)	Error	Down	Error	Down	Down	Down
ctv(b)	Up	**Down**	Up	**Up**	**Down**	UP (**Error** before 1.4)

Table 2: C++, Visual Basic, C# and Java signatures

The first and the third columns are identical since the four languages adopt an invariant overriding policy. Hence, in our experiment, `u` and `ud` return the same results since both are statically declared as Up. All differences are in the second column, when a receiver `d` is declared `Down` and is actually `Down`. That is why we presented only this case for the other languages. Differences (in bold in the table)

are due to the overloading rules that are different in C++, C#, Visual Basic and Java.

C++ rejects the first line considering that the method `cv(Middle)` hides the previously defined method `cv(Top)`.

The last line is Down for C++ and C# following the intuitive semantics of OO languages where the most specific method, accordingly to the receiver, is selected.

The result of the last line for Visual Basic is due to the priority given to the parameter over the receiver in order to select the method; Bottom in `ctv(Bottom)` of Up is considered by Visual Basic as more specialized than Middle in `ctv(Middle)` of Down. This is a strict interpretation of overloading; the parameter is used to select the method.

Before the 1.4 version, the last line for Java gives an `Error`. This is due to the strict method overriding applied by Java. Note that between version 1.3 and 1.4 the interpretation of line 6, has changed! Java has now the same signature as VisualBasic.

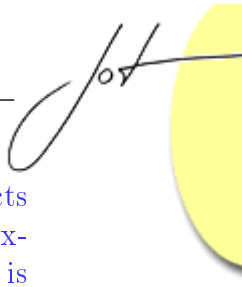
Implication for software component technology

A software component is defined in [18] as an independent software entity, that can be deployed and composed with others. Many models of components have been proposed. All rely on a feature that allows to describe components while hiding unnecessary details: encapsulation. Information needed to assemble a component is not the component itself - considered as a white-box - but the interface of the component. In that case, the component is seen as a black-box. Many authors have noticed that to be able to compose components a certain number of properties of components not typically considered part of the interface *should* be considered part of the interface. The component is then viewed as a grey-box [7].

We have proposed in [6] to attach a contract to a component in order to organize information of its interface. A contract may contain 4 levels: syntactic, semantic, behavioral, and quality of service. Only the first level is needed for the experiment we have done. This level ensures the "compilability" of interfaces; it is used to check that method names, return types, parameter numbers and types are compatible with the components it is assembled with.

In case of an assembly of heterogeneous components, what is the level of greyness of a component? As we will see, in order to predict the behavior of an assembly the syntactic contract of a component may need to expose its implementation language and deployment mode (distributed or not) but also make explicit assumptions on required properties of the implementation languages of its clients!

To show this situation we performed two experiments: the first experiment illustrates the consequences of the lack of consensual definition of overloading and overriding when distributing a one language program (example Java). We use two



mechanisms of distribution (CORBA and RMI) and show that distributing objects is not as transparent as it should be. The second experiment generalizes the experiment thanks to .NET and mixes 3 languages and shows that encapsulation is broken because of the different interpretations of overriding and overloading.

3 JAVA DISTRIBUTION EXPERIMENT

Using the testing protocol defined in section 2, we create a client program with three server objects:

- `Up u = getServerUp();`
- `Down d = getServerDown();`
- `Up ud = getServerDown();`

The methods `getServerDown()` and `getServerUp()` are used to get the remote reference to the server object according to the technology (RMI, CORBA or .NET). The type of the reference `u` is `Up`, of `d` is `Down` and of `ud` is declared as `Up` but the actual type of the instance is `Down`. In accordance with the testing procedure, each of these instances is invoked with the methods `cv` and `ctv` with the parameters of type `Top`, `Middle` and `Bottom`.

All used distribution technologies rely on the generation of proxies (also called stubs or skeletons) from the specification of interfaces. Hence we have *two* compilations phases, each of them leading possibly to errors. As done previously, the only removed statements are those triggering compilation errors.

Since there is an additional compilation stage, there are various sources of errors; they appear in the table as:

- **ICE**: Interface compilation Error (from IDL to skeleton source)
- **SCE**: Skeleton Compilation Error (from skeleton source to skeleton binary)
- **CE**: Compilation Error (as for previous sections)

We have tested Java with the two middlewares embedded with the Java Development Kit: RMI and CORBA. This leads to the comparison of the behavior of a language relatively to late-binding in three contexts: local, RMI-remote, CORBA-remote.

The interface specification in IDL-CORBA for our tests is:

```
module App{
```

```
  interface Top{};
  interface Middle : Top {};
  interface Bottom : Middle {};

  interface Up
    {
      string cv(in Top t);
      string ctv(in Bottom b);

      oneway void shutdown();
    };

  interface Down : Up
    {
    // Overloading is forbidden
    //      string cv(in Middle m);
    //      string ctv(in Middle m);
    };
};
```

The following code illustrates how a covariant redefinition can be simulated:

```
public class DownImpl extends DownPOA{
    ...
    public String cv(Top t){
    // If call needed
       // Middle m = (Middle)t; // POSSIBLE
                                //CAST ERROR
       // m.call();
    return("Down");
    }
... }
```

This code is error prone. A safer one, checking the dynamic parameter type may correspond to the following:

```
public class DownImpl extends DownPOA{
    ...
    public String cv(Top t){
    // If call needed
    if (t instanceof Middle){
         Middle m = (Middle)t;
```

	Java/CORBA			Java/RMI		
calls	u	d	ud	u	d	ud
cv(t)	Up	ICE	ICE	Up	Up	Up
cv(m)	Up	ICE	ICE	Up	Down	Up
cv(b)	Up	ICE	ICE	Up	Down	Up
ctv(t)	Error	ICE	ICE	Error	SCE	SCE
ctv(m)	Error	ICE	ICE	Error	SCE	SCE
ctv(b)	Up	ICE	ICE	Up	SCE	SCE

Table 3: Java/CORBA and Java/RMI results

```
        m.call();
        }
       else super.cv(t);
    return("Down");
    }
... }
```

Results and analysis

The second and the third columns of table 3 show systematic Interface Compilation Errors. This is due to the fact that CORBA forbids methods overloading and methods overriding. The only acceptable case is invariance. Contravariant redefinition cannot be implemented. Covariant redefinition could be implemented with a proper method body.

The most interesting behavior of Java is in line 6 of table 2. The `Error` result reappears when trying to compile the RMI skeleton, leading to many ambiguous calls with `ctv` method as visible in Java/RMI part of table 3.

The RMI skeleton code below shows that the RMI compiler prepares all possible calls for ctv. Alas, these are ambiguous, and then when the Java compiler tries to compile the skeleton, it fails. This shows that this problem is still not understood sufficiently. A high quality source to source compiler should never generate uncompilable code!

```
public final class DownImpl_Skel
  implements java.rmi.server.Skeleton
  {
  private static final
    java.rmi.server.Operation[]
        operations = {
```

```
        new java.rmi.server.Operation
         ("java.lang.String ctv(Bottom)"),
        new java.rmi.server.Operation
         ("java.lang.String ctv(Middle)"),
        new java.rmi.server.Operation
         ("java.lang.String cv(Middle)"),
        new java.rmi.server.Operation
         ("java.lang.String cv(Top)"),
};
....
public void dispatch(
      java.rmi.Remote obj,
      java.rmi.server.RemoteCall call,
      int opnum, long hash)
  throws java.lang.Exception{
....
switch (opnum) {
case 0: // ctv(Bottom)
Bottom $param_Bottom_1;
....
java.lang.String $result =
  server.ctv($param_Bottom_1);
....
}
....
}
.....
}
```

Our test shows that if a Java application uses a method overloading it cannot be distributed: CORBA forbids method overloading at IDL compilation stage, while RMI-Java enables the first stage of interface compilation, but detects and triggers errors only at implementation compilation stage. This late detection is problem-prone for application evolution.

This experiment highlights the fact that distribution techniques impact the distribution behavior because of the constraints they add to the normal behavior of a language like Java. In the next section we generalize the experiment using more languages, but removing the distribution.

4 .NET EXPERIMENT

The model of figure 1 may serve as an experiment of components assembly. Imagine, as in figure 2, that classes Up, Top, Middle and Bottom define a framework C1

written in a language *L1*. Later, an evolution of C1 is realized by the class Down written in a language *L2* that extends Up. This defines the component C2. A client C3 is written in a language *L3*.

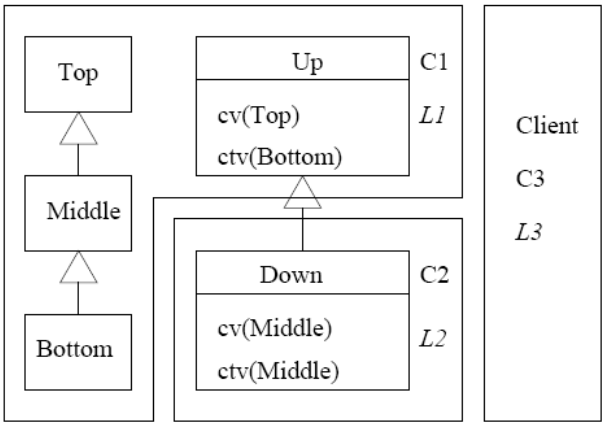


Figure 2: How will this assembly behave?

We have performed this experiment with the .NET framework [14] that claims language interoperability. We used 3 languages integrated into the framework: C#, Visual Basic and C++. Each of them was used to develop the 3 components, leading to 27 (3*3*3) different assemblies.

Results

Results are organized in 3 tables of 9 signatures. Each table is associated to a client language (C#, Visual Basic, C++ respectively). The programming language of the basic framework (L1) appears by column. The programming language of the extension (L2) appears by line. Each cell is the observed language signature.

L2/L1	C#	VB	C++
C#	C#	C#	**C++**
VB	C#	C#	**C++**
C++	C#	C#	**C++**

Table 4: C# client results (L3 = C#)

Table 4 shows the result for a C# client. The result is mainly a C# signature except for the C++ column where the C++ signature is observed.

Table 5 shows the result for a Visual Basic client. The result is mainly a Visual Basic signature except for the C++ column where the C++ signature is observed. Another difference is observed. When the framework is written in Visual Basic and the extension in C++, the observed signature is C# which is not used at all!

L2/L1	C#	VB	C++
C#	VB	VB	**C++**
VB	VB	VB	**C++**
C++	VB	**C#**	**C++**

Table 5: Visual Basic client results (L3 = VB)

L2/L1	C#	VB	C++
C#	**(C++/VB)**	**(C++/VB)**	**(C++/VB)**
VB	C++	C++	C++
C++	C++	C++	C++

Table 6: C++ client results (L3 = C++)

Table 6 shows the result for a C++ client. The result is mainly a C++ signature but for the C# line where the observed signature does not match any known signature. It looks like a mixing of C++ and Visual Basic signatures.

	C#/C#			VB/C#			C++/C#		
	u	d	ud	u	d	ud	u	d	ud
cv(T)	U	*E*	U	U	*E*	U	U	*E*	U
cv(M)	U	D	U	U	D	U	U	D	U
cv(B)	U	D	U	U	D	U	U	D	U
ctv(T)	E	E	E	E	E	E	E	E	E
ctv(M)	E	D	E	E	D	E	E	D	E
ctv(B)	*U*	*U*	*U*	*U*	*U*	*U*	*U*	*U*	*U*

Table 7: Detailed results for a C++ client and a C# extension

Table 7 shows the detailed results of the first line of the previous table (D denotes Down, U Up and E Error). It shows the mixing of C++ and Visual Basic signatures (see table 2 for comparison). Notice the last line where the Visual Basic behavior appears everywhere, even when Visual Basic is not used!

Interpretation

The signatures observed are mainly those expected by the client, i.e. the client's signature. C# and C++ client are the more stable with 6 out of 9 "good" cells while Visual Basic has only 5 out of 9 "good" results. The reasons for the "bad" behavior are mainly due to C++ when clients are C# or Visual Basic, probably because of the inheritance exception of C++ in cell line 1, column 2 of table 2 where `cv(Top)` of Up is hidden by the overloading `cv(Middle)` of Down.

If we try to write down the syntactic contract of the component C2, we would produce something like what is presented in table 8. In this table C3 corresponds to the client.

Services	Returns the result of method found in:
cv(Top)	Up Error if receiver is Down declared Down and C1 is written in C++; or if C3 is written in C++.
ctv(Bottom)	Up often, but Down when (1) C3 is written in C# and when receiver is Down declared Down, or (2) C3 is written in Visual Basic and when C1 is written in C++ or when (C1 is written in Visual Basic and C2 in C++), or (3) C3 is written in C++ and when C2 is written in Visual Basic or C++.
cv(Middle)	Up often, but Down when receiver is Down declared Down.
ctv(Middle)	Down Error when receiver is not Down declared Down.

Table 8: Component extension (C2) contract attempt

This table reveals that the contract makes references to implementation details of the component and, worse, to a client internal feature: the implementation language. The encapsulation is broken. The black-box whitens and borders fade away ...

Explicit references to the implementation language could be replaced by generic language features following the approach proposed in [8]. However, language signature diversity and the finer points of the interpretations convinced us that such generic language features would be more complex to describe than the simple reference to the language.

Beyond a deep understanding of these results, this experiment shows that in order to be able to predict the behavior of a component assembly, the client of a component must have information on how the overriding and overloading are interpreted. This information relies on the languages used to realize the component but also on the whole internal structure such as inheritance relationships, overloading and overriding that are actually done. So, what about encapsulation?

We are convinced, although experiments remain to be done, that the use of another language, like Eiffel, in the experiment would increase the diversity of behaviors.

The semantic distance between Eiffel and C++, C# or Visual Basic is too large as Eiffel signature of table 9 shows. Eiffel forbids overloading (Errors on line 4 and 5) and allows covariant overriding (column 3 lines 1,2 and 3). Column 2 is a mixing of C++ on the first line (Error) and Visual Basic on the last line (Up).

calls	u	d	ud
cv(t)	Up	Error	**Down**
cv(m)	Up	Down	**Down**
cv(b)	Up	Down	**Down**
ctv(t)	Error	Error	Error
ctv(m)	Error	**Error**	Error
ctv(b)	Up	Up	Up

Table 9: Eiffel signature

5 SUGGESTIONS

Table 8 illustrates that the current situation is impossible for programmers building assemblies; the table is far too complex. Simple rules must be chosen and used; components should have well-defined semantics and it should not depend on its clients. Components need to be reused in open and multi-language contexts. The behavior of the component assemblies must be predictable from the component's interface specification, probably with more information than the simple syntactic contract used during this experiment.

In order to reach this goal, here are some suggestions:

- We could study the interaction combinations among languages. But, as our experiment shows for only 3 rather semantically close languages, all combinations are to be considered. This is exponential, and as table 7 shows in its first line, the "combination operator" can lead to unexpected signatures. It is not a binary operation (combining two language signatures may result in a brand new one leading to other combinations to study, and so on). Because of the number of programming languages and the diversity of their signatures, this approach seems untractable. But could we do otherwise?
- We could enrich the level of information in the contract in order to express how the component interprets overloading and overriding. Overloading and overriding semantics should not remain implicit, but as our experiment shows, should be explicit. This solution also requires the previous approach and a good understanding of language combinations.
- We could compel the use of overriding to the invariant case and forbid overloading. With such a restriction, all object programming languages have the same behavior, hence the assembly is simple. This solution requires modification of compilers and languages which is practically impossible.
- At last, a new tool may be developed. An *assembly checker* that would detect "pathological cases" when components written in different languages are assembled.

The last solution seems to be the most tractable. Programming language history shows that programming language evolution follows a double movement; first a generation of ideas and concepts that are implemented and tested, then a selection among these concepts. This selection generates language restrictions. For instance, the "GOTO" has been replaced in order to improve the structure of programs. After developing "multi-language assembly checkers" it may be time for the next generation object programming languages to make a decision and select the right (and constraining) rules of overriding and overloading.

Overloading has already been criticized [13, 10] and this article may be considered as another argument against overloading; it interacts badly with overriding. The choice for a covariant, invariant or contravariant overriding is more open. In practice, invariance is largely used and its implementation is rather easy. Covariance is hard to implement, and not always safe, as Eiffel signature shows in column 3 line 1. Contravariant, if theoretically sound, is rarely implemented. Hence, a tradeoff is to choose invariant overriding.

6 CONCLUSION

The first experiment is restricted to Java techniques and shows that overriding and overloading choices have consequences on distribution. The .NET experiment uses very semantically close languages; overriding is invariant and overloading is allowed. In this context, the diversity of behaviors remains great.

Theoretical research on component assembly mainly focusses on static aspects of linking [9, 2, 1] even when it deal with dynamic linking; it is in the sense of "compiler linkers" not in the sense of "object-oriented late-binding". It looks for typing theories that ensure error-free compilations, ignoring the expected behavior. On the other hand, practical approaches propose component models implementations in homogeneous contexts (Java for instance) or leave the responsibility to programmers to manage the composition of components (CCM or CORBA for instance).

More experiments need to be done with more languages, and different scenarios. For instance, we could envisage experiments with two components without extension or extension within the client component. Anyhow, we feel that it is strange that research on overriding and overloading interactions is uncommon. In a context where model transformations (MDA [16]), language interoperability through middleware (CORBA, SOAP) or virtual machine (.NET, Java), and assembly of components are considered as key technologies, it is astonishing.

The problems of assembling components carried out in different languages or in a distributed environnement are real. We believe that the extension of components in different languages may become frequent in the future. Anyway, both situations can occur and should be considered in theories and in programming languages design.

7 ACKNOWLEDGMENTS

We want to thank the anonymous reviewers for their precise comments and suggestions.

REFERENCES

[1] D. Ancona, S. Fagorzi, and E. Zucca. A calculus for dynamic linking. In Springer, editor, *ICTCS'03 - Italian Conference on Theoretical Computer Science*, Lecture Notes in Computer Science 2841, 2003.

[2] D. Ancona and E. Zucca. Sound and complete inter-checking (the very essence of principal typings). Technical report, Universita di Genova, 2004.

[3] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, 2000.

[4] A. Beugnard. Comparison of various oo languages relatively to their late-binding semantics. http://perso-info.enst-bretagne.fr/~beugnard/papiers/lb-sem.shtml.

[5] A. Beugnard. OO languages late-binding signature. In *FOOL 9 (The Ninth International Workshop on Foundations of Object-Oriented Languages)*, Portland, Oregon, January 19 2002.

[6] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, pages 38–45, 1999.

[7] M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999. http://www.abo.fi/~mbuechi/publications/TR297.html.

[8] P. Crescenzo, C. Jalady and P. Lahire. Annotations of Classes and Inheritance Relationships : an Unified Mechanism in Order to Improve Skills of Library of Classes. *Proceedings of the Workshop on Managing Specialization/Generalization Hierarchies (MASPEGHI)*, at ASE'03. Montreal, Quebec, Canada, October 2003.

[9] S. Drossopoulou, S. Eisenbach, and D. Wragg. A fragment calculus towards a model of separate compilation, linking and binary compatibility. In *Logic in Computer Science*, pages 147–156, 1999.

[10] R. Ducournau. Spécialisation et sous-typage : thème et variations. *Technique et Science Informatique*, 21(10):1305–1342, 2002.

[11] ECMA. Standard-ecma334, C# language specification. http://www.ecma-international.org/publications/standards/Ecma-334.htm.

[12] M. D. McIlroy. Mass produced software components. In *NATO Conference on Software Engineering*, Garmisch, Germany, 1968.

[13] B. Meyer. Overloading vs. object technology. *Journal of Object-Oriented Programming*, pages 3 – 7, October/November 2001.

[14] Microsoft. Microsoft .NET. http://www.microsoft.com/net.

[15] Microsoft. Visual basic .net language specification. http://msdn.microsoft.com/library/en-us/vbls7/html/vbspecstart.asp.

[16] OMG. site mda. http://www.omg.org/mda.

[17] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1997.

[18] C. Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley and ACM Press, 1998.

ABOUT THE AUTHORS

Antoine Beugnard is an associate professor in Computer Science at ENST Bretagne, Brest, France. He could be reached at Antoine.Beugnard@enst-bretagne.fr. See also http://www-info.enst-bretagne.fr/~beugnard.

Salah Sadou is an associate professor in Computer Science at Vannes University Institute of technology, Vannes, France. He could be reached at Salah.Sadou@univ-ubs.fr. See also http://www-valoria.univ-ubs.fr/Salah.Sadou/.