Method overloading and overriding cause encapsulation flaw

Antoine Beugnard

To cite this version:

Antoine Beugnard. Method overloading and overriding cause encapsulation flaw. Object-Oriented Programming Languages ans Systems (OOPS), 21st ACM Symposium On Applied Computing (SAC), Apr 2006, Dijon, France. pp.1424 - 1428. hal-02166903

HAL Id: hal-02166903
https://hal.archives-ouvertes.fr/hal-02166903
Submitted on 27 Jun 2019

Method overloading and overriding cause encapsulation flaw

An experiment on assembly of heterogeneous components

Antoine Beugnard
ENST Bretagne
Computer Science Department
CS83818, F-29238 Brest cedex 3
Antoine.Beugnard@enst-bretagne.fr

ABSTRACT

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 recalls the various interpretations of overriding and overloading in object oriented languages through what we call *language signatures*. Then, we realize an experimentation 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 assembly, components must expose their internal structure! We propose a solution to this problem.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.12 [**Software Engineering**]: Interoperability

Keywords

objet oriented language interoperability, encapsulation, software component

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 rely more and more on a unifying modelling language (UML) that relies itself on objects. But it seems that the hope object is the silver bullet solution has gone. Reusing is not so simple and assembling object remains complex. The need for a better engineering process leads to the development of software by assembly of software components. The idea to develop software systems as electronic ones is not new [10], but only recently have we seen many components 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 methods overriding and overloading semantics misfit with one of the main feature of components: encapsulation.

The article is organized as follows. We first recall in the next section some of these behaviors, and in section 3 how components interfaces can be described with contracts. Then, in section 4 we present the experiment that demonstrates that the assembly of heterogeneous components cannot be predictable if their interface is not precise enough; that is exposes many implementation details. We conclude with some suggestions to solve this problem.

2. OVERLOADING AND OVERRIDING

The main contribution of object programming is, to our point of view, the easy access to dynamic dispatching that makes possible the development of frameworks. Frameworks can be easily extended, 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 softwares. However, the benefit of this "late-binding" is balanced out 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.

This powerful mechanism is however difficult to use since the *overriding intent* expressed in a diagram such as in figure 1 is understood differently by object programming languages. As it has already been published in [3] 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 to accept these three possibilities at the same time. The late-binding resulting of the detection of this overriding can be simple, the most frequent case, or multiple, like in CLOS. Overloading can be allowed or not.

In order to show the different interpretations we have implemented the model of figure 1 in many languages. Then we

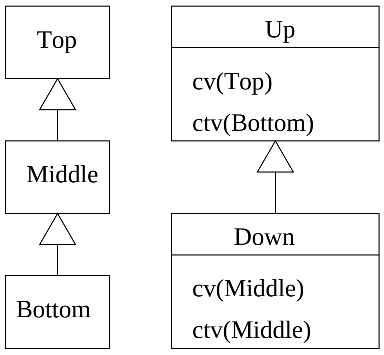

Figure 1: How will this model behave?

called all possible cases and built what is called a language signature [3]. Table 1 shows how the signature is elaborated. Each slot[1] contains the name of the class where the applied method was found. The word "Error" denotes a compilation error.

Table 1: Signature elaboration

```
procedure main
– receivers
        Up u, ud;
        Down d;
– parameters
        Top t = new Top();
        Middle m = new Middle();
        Bottom b = new Bottom();
```

– 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);

The next three tables show the signatures of C++ [15] (table 2), Visual Basic [13] (table 3) and C# [9] (table 4) since we use them in the experiment of section 4.

Table 2: C++ signature

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

First and third columns are identical since the three 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 actually Down. Differences (in bold in the tables) are due to the overloading rules that are different in C++, C# and Visual Basic.

[1]We use the word "slot" in this context and the word "cell" in the more complex tables of the following section.

Table 3: Visual Basic signature

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

C++ rejects the first line (of column 2) considering that the method `cv(Middle)` hides the previously defined method `cv(Top)`.

The last line of the second column 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.

Table 4: C# signature

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

3. COMPONENT AND CONTRACT

A software component is defined in [16] as independent software entity, that can be deployed and composed with others. Many models of component 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 compound components some implementation details must be exposed in order to ensure a correct assembly [5]. The component is then viewed as a grey-box.

Beugnard and al. have proposed in [4] 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 realized. This

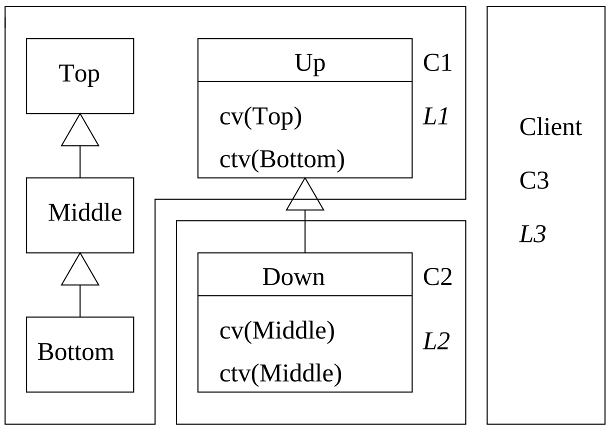

Figure 2: How will this assembly behave?

level ensures the "compilability" of interfaces; it is used to check that method names, return types, parameter numbers and types are compatible with components it is assembled with.

In case of an assembly of heterogeneous components what is level of greyness of a component? As we will see, in order to be able to predict the behavior of some assembly the syntactic contract of a component may need to expose its implementation language but also make explicit assumption on the implementation languages of its clients!

4. COMPONENTS ASSEMBLY

4.1 .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*.

We have realized this experiment with the .NET [12] framework 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.

The language signatures of many languages including (ADA, Java1.3, Java 1.4[2], Smalltalk, CLOS) and the source code of the experiment can be seen at (hidden)

4.2 Results

Results are organized in 3 tables of 9 signatures. A 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.

Table 5 shows result for a C# client. The result is mainly a C# signature but for the C++ column where the C++ signature is observed.

Table 6 shows result for a Visual Basic client. The result is mainly a Visual Basic signature but for the C++ column

[2]Yes, Java signature changed between 1.3 and 1.4!

Table 5: C# client results (L3 = C#)

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

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!

Table 6: Visual Basic client results (L3 = VB)

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

Table 7 shows 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. The new signature is a mixing of C++ and Visual Basic signatures.

Table 7: C++ client results (L3 = C++)

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

Table 8 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. Notice the last line where the Visual Basic behavior appears everywhere, even when Visual Basic is not used!

4.3 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 over 9 "good" cells while Visual Basic has only 5 over 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 slot 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 9. C3 is the client.

This table reveals that the contract makes references to implementation details of the component and, worst, 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 [6]. However, language signatures diversity and the finer points of the interpretations convinced us

Table 9: Component extension (C2) contract attempt

Services	Returns the result of method found in:
cv(Top)	Up Error when 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: Detailed results for a C++ client and a C# extension

	C#C#			VBC#			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**

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 the way 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 actually done. So, what about encapsulation?

4.4 More Experiments

We are convinced, but experiments remain to be done, that the use of the Eiffel language in the experiment would increase the diversity of behaviors. The semantic distance between Eiffel and C++, C# and Visual Basic being larger as Eiffel signature of table 10 shows. Eiffel forbids overloading (Errors on line 4 and 5) and allows covariant overriding (see 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).

Table 10: Eiffel signature

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

5. CONCLUSION

Our experiment is limited to the .NET environment, with very semantically close languages; overriding is invariant and overloading is allowed. In this context, the diversity of behaviors is great.

Theoretical researches on component assembly mainly focuss on static aspects of linking [7, 2, 1] even when they deal with dynamic linking; it is in the sense of "compiler linkers" not in the sense of "object-oriented late-binding". They look 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 let the responsibility to programmers to manage the composition of components (CCM or CORBA for instance).

In order to be fully usable, components need to be reused in open and multi-languages contexts. The behavior of the components assembly must be predictable from components interface specification[3].

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

- We could study the interactions combination among languages. But, as our experiment shows for only 3 languages rather semantically close, all combinations are to be considered. This is exponential, and as table 8 shows in its first line, the "combination operator" can lead to unexpected signatures. It is not a binary operation. 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 combination.

- We could eventually 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 to modify compilers and languages.

[3]Probably with more information than the simple syntactic contract used during this experiment.

An interesting study would be to count the number of "difficult cases" to be able to evaluate the impact on existing codes[4].

The latter solution seems to be the most tractable. But it may be considered as too strong a change. Limiting this restriction to the *public interface* would allow programmers to continue to use their well-known programming patterns involving overloading inside the component. 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. It may be time for object programming languages to make a decision and select the good (and constraining) rules of overriding and overloading.

Overloading have already been criticized [11, 8] and this article can 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 at column 3 line 1. Contravariant, if theoretically sound, is rarely implemented. Hence, a tradeoff could be to choose invariant overriding.

More experiments need to be made with more languages, and different scenarios. For instance, we could imagine experiments with 2 components without extension or extension within the client component. Whatever, we feel strange that researches on overriding and overloading interactions are uncommon. In a context where model transformations (MDA [14]), language interoperability through middleware (CORBA, SOAP) or virtual machine (.NET, Java), and assembly of components are considered as key technologies, it is astonishing!

The problem of assembling components realized in different languages is real. We believe that the extension of components in different languages may become frequent in the future. Whatever, both situations can occur and should be considered in theories and in programming languages design.

6. ACKNOWLEDGMENTS

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

7. 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] 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.

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

[5] 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.

[6] A. Capouillez, P. Crescenzo, and P. Lahire. Le modèle ofl au service du métaprogrammeur – application à java. *L'objet, LMO'2002*, pages 11 – 24, 8 2002.

[7] 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.

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

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

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

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

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

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

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

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

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

[4]An "object composability" metric that takes into account overriding and overloading problems is probably to be defined.