[This is the original version of an article published, in three parts, by *Software Development* in its “Beyond Objects” column: May, June and July 2002 issues.]

Multi-language programming: how .NET does it

Bertrand Meyer

ETH, Zürich / ISE, Santa Barbara

Part 1: Combining language models

What does it take to support several programming languages within one environment? The example of .NET, which has taken language interoperability to new heights, shows that it’s possible, but only with the right design, the right infrastructure, and appropriate effort on the basis of both compiler writers and programmers. In this three-part article, I’d like to go deeper than what I have seen published on the topic, to elucidate what it takes to provide true language openness. The experience that my colleagues have accumulated over the last three years of working to port Eiffel on .NET, as well as the countless discussions we have had with other .NET language implementers, informs this discussion.

Who needs more than one language?

Let’s start with the impolite question: Should one really care about multi-language support? When this feature of .NET was announced at the technology’s July, 2000 debut, Microsoft’s competitors sneered that it wasn’t anything anyone needed. I’ve heard the idea of multi-language development dismissed, or at least questioned, on the argument that most projects just choose one language and stay with it. But that doesn’t really address the issue. For one thing, the argument sounds too much like asserting, from personal observation, that people in Singapore don't like skiing. Lack of opportunity doesn't imply lack of desire or need. In pre-dot-NET environments, the effort required to interface modules from multiple languages was enough to make many people stick to just one; but with an easy way to combine languages seamlessly and effortlessly, they may — as early experience with .NET tends to suggest — start to appreciate their newfound freedom to mix and match languages.

Even more significant is the matter of *libraries* and reuse. Whether your project uses one language or more, it can take advantage of reusable components, which may have originated in different source languages. Here interoperability means that you can use whatever components best suit your needs, regardless of creed or language of origin.

This ability to mix languages offers great promise for the future of programming languages. The practical advance of new language designs has been hindered by library issues: you may have conceived the best language in the world, implemented an optimal compiler and provided brilliant tools, and still not get the users you deserve because you can't match the wealth of reusable components that other languages provide, merely because they've been around longer. Building bridges to these languages helps, but it's an enormous effort — with *n* languages, n^2 bridges! — if you have to do it separately for each one. In recent years, this library compatibility issue may have been the major impediment to the spread of new language ideas, regardless of their intrinsic value. Language interoperability can overturn this obstacle. Under .NET, as long as your language implementation satisfies the basic interoperability rules of the environment (as explained in this article), you can take advantage of components written in any other language whose implementers have adhered to the same rules. That still means some work for compiler writers, but it's work they must do once for their language — not once for each language with which they want to interface.

The language openness of .NET is a welcome relief after the years of incessant Java attempts at language hegemony. For far too long, the Sun camp has preached the One Language doctrine. The field of programming language design has a long, rich history and there is no credible argument that the alpha and omega of programming was uttered in Silicon Valley in 1995, closing off any future evolution. Microsoft's .NET breaks this unhealthy and unnatural lock.

Everyone will benefit — even the Java community, in fact: Now that there's competition again, new constructs are — surprise! — being considered for Java; one hears noises, for example, about introducing genericity sometime in the current millennium. Such are the virtues of openness and competition.

The more than 20 languages ported or in the process of being ported to .NET range from Cobol and Fortran to Smalltalk, Oberon, Eiffel, Java, Perl, Scheme and Python. How does this all work? Do languages have to sacrifice anything? Should we believe those who say that it's all smoke and mirrors, and that deep down all languages get reduced to a common denominator, whether we call it C#, Visual Basic.NET, managed C++ (or Java)? These are some of the questions I will examine in this article.

Language operability at work

Multi-language communication techniques aren't a new idea. For some time, Eiffel has included an "external" mechanism for calling out to C and other languages, and a call-in mechanism known as Cecil. The Java Native Methods Interface is similar. But all this addresses calls only. .NET goes much further:

- A routine written in a language L1 may call another written in a different language L2.
- A module in L1 may declare a variable whose type is a class declared in L2, and then call the corresponding L2 routines on that variable.
- If both languages are object-oriented, a class in L1 can inherit from a class in L2.
- Exceptions triggered by a routine written in L1 and not handled on the L1 side will be passed to the caller, which — if written in L2 — will process it using L2's own exception-handling mechanism.
- During a debugging session, you may move seamlessly across modules written in L1 and L2.

I don't know about you, but I've never seen anything coming even close to this level of interoperability.

Affirmative action

Let's examine how .NET's language interoperation works. Here's the beginning of an ASP.NET page (from an example at dotnet.eiffel.com). The associated system is mostly written in Eiffel, but you wouldn't guess this from the page text; as stated by the ASP.NET PAGE LANGUAGE directive, the program code on the page itself, introduced by <SCRIPT RUNAT="SERVER">, is in C#:

```
<%@ Assembly Name="conference" %>
<%@ Import Namespace="Conference_registration" %>
<%@ Page Language="C#" %>


<HTML>
        <HEAD>
        <TITLE>TOOLS CONFERENCE</TITLE>
        <SCRIPT RUNAT="SERVER">

        /* Start of C# code */   REGISTRAR conference_registrar;
        bool registered;
        String error_message;
        void Page_Init(Object Source, EventArgs E)
        void Page_Init( Object Source, EventArgs E )
                {
                conference_registrar = new REGISTRAR ();
                conference_registrar.start();
                … More C# code…
                 }
        …More HTML…
```

The first C# line is the declaration of a C# variable called conference_registrar, of type REGISTRAR. On the subsequent lines, we create an instance of that class, through a new expression, and assign it to conference_registrar; and we call the procedure start on the resulting object. Presumably, REGISTRAR is just some C# class in this system.

Presume not. Class REGISTRAR is an Eiffel class. The only C# code in this example application is on the ASP.NET page, whose start is shown above, and includes only a few more lines; its task is merely to read the text entered into the various fields of the page by a Web site visitor and to pass it on, through the conference_registrar object, to the rest of the system — the part written in Eiffel that does the actual processing.

Nothing in the above example (or the rest of the ASP.NET page) mentions Eiffel. REGISTRAR is not cited as an Eiffel class, or a class in any specific language: it's simply used as a class. The expression new REGISTRAR () that creates an instance of the class might look to the unsuspecting C# programmer as a C# creation, but in fact it calls the default creation procedure (constructor) of the Eiffel class. Not that this makes any difference at the level of the Common Language Runtime: at execution time, we don't have C# objects, Eiffel objects or Visual Basic objects; we have .NET citizens with no distinction of race, religion or language origin.

If we don't tell the runtime, in the above code, that REGISTRAR is an Eiffel class, how is it going to find that class? Simple: namespaces. Here's the beginning of the Eiffel class text of REGISTRAR:

```
indexing
        description: "[
                Registration services for a conference;
                include adding new registrants and new registrations.
                ]"
        dotnet_name: "Conference_registration.REGISTRAR"
class
        REGISTRAR
inherit
        WEB_SERVICE
create
        start
feature -- Initialization
        start is
                        -- Set empty error message.
                do
                        set_last_operation_successful (True)
                        set_last_error_message ("No Error")
                        set_last_registrant_identifier (-1)
                end

… Other features …
```

The line starting with dotnet_name says: "To the rest of the .NET world, this class shall be part of the namespace Conference_registration, where it shall be known under the name REGISTRAR". This enables the Eiffel compiler to make the result available in the proper place for the benefit of client .NET assemblies, whether they originated in the same language or in another one.

Now consider again the beginning of the ASP.NET page shown earlier:

```
<%@ Assembly Name="conference" %>
<%@ Import Namespace="Conference_registration" %>
<%@ Page Language="C#" %>

<HTML>
        <HEAD>
        <TITLE>TOOLS CONFERENCE</TITLE>
        <SCRIPT RUNAT="SERVER">
        … The rest as before …
```

The second line says to import the namespace Conference_registration; that does the trick. A namespace is an association between class names, a way of saying "The class name A denotes that code over there, the class name B denotes this other code here". In that association, the class name REGISTRAR will denote the Eiffel class shown above, since we took care of registering it under that name in the dotnet_name entry of its indexing clause.

The basic technique will always be the same:

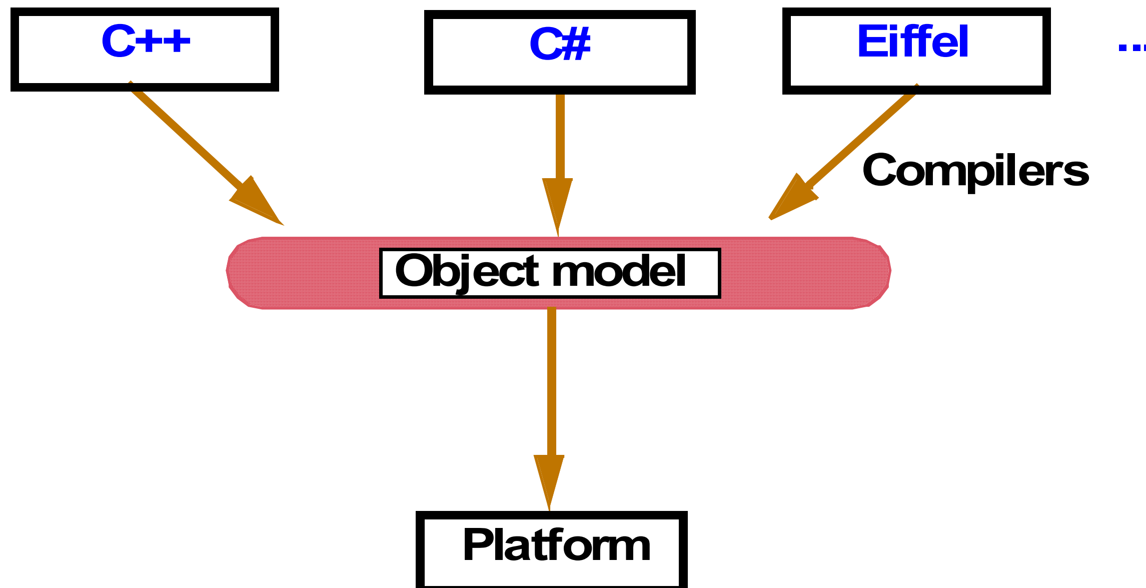
- 1 When you compile one or more classes written in language L1, you specify the namespaces into which they will be compiled, and the final names they must retain in that language.

- 2 When you write a system in a language L2 — the same as L1, or another one — you specify one or more namespaces to "import"; they will define how to understand any class name to which your system may refer.

The details may vary depending on the languages involved. On the producer side, L1, you may retain the original class names explicitly specify an external class name. On the consumer side, you may have mechanisms to adapt the names of external classes and their features to the conventions of L2. Some flexibility is essential here, since what's acceptable as an identifier in one language may not be in another: Visual Basic, for example, accepts a hyphen in a feature name, as in my-feature, but most other languages don't, so you'll need some convention to accept the feature under a different name. What's important is that you can have access to all the classes and features from any other .NET language.

Combining different language models

How does the interoperability work in practice? The first key idea is to map all software to the .NET Object Model. Once compiled, classes don't reveal their language of origin:



Starting from a source language, the compiler will map your programs into a common target. This by itself isn't big news, since we could use the same picture to explain how compilers map various languages to the common model of, say, the Intel architecture. What *is* new is that the object model retains high-level structures such as classes and inheritance that have direct equivalents in source programs written in modern programming languages, especially object-oriented ones. This is what allows modules from different languages to communicate at the proper level of abstraction, by exchanging objects — all of which, as .NET objects, are guaranteed to have well-understood, language-independent properties.

Object model discrepancies

Of course, the languages involved have their own properties, which may significantly differ from the .NET object model. That's to be expected: Otherwise, they wouldn't really be different languages, just a different syntax and minor variations on a single language theme. To a certain extent, this characterization could be applied to C# and Visual Basic.Net; one may claim that these two are, deep down, just one language, now that VB has become O-O . But it’s definitely incorrect if we consider the entire set of .NET language players. The case of non-O-O languages is the most obvious: right from the initial announcements, .NET has included languages like APL and Fortran, which no one would accuse of being object-oriented.

Even if we restrict our attention to O-O languages, we'll find discrepancies. Each has its own object model; while the key notions—class, object, inheritance, polymorphism, dynamic binding——are common, individual languages depart from the .NET model in some significant respects:

- Eiffel and C++ allow multiple inheritance; the .NET object model (as well as Java, C# and Visual Basic.NET) permits a class to inherit from only one class, although it may inherit from several *interfaces*.
- Eiffel and C++ each support a form of genericity (type parameterization): you can declare an Eiffel class as LIST [G] to describe lists of objects of an arbitrary type G without saying what G is; then you can use the class to define types LIST [INTEGER], LIST [EMPLOYEE], or even LIST [LIST [INTEGER]]. C++'s templates pursue a similar goal. This notion is unknown to the .NET object model, although planned as a future addition; currently you have to write a LIST class that will manipulate values of the most general type, Object, and then cast them back and forth to the types you really want.
- The .NET object model permits in-class overloading: within a class, a single feature name may denote several different features. Several languages disallow this possibility as incompatible with the aims of quality object-oriented development.

These object model discrepancies raise a serious potential problem: How do we fit different source languages into a common mold? There are two basic approaches: either change the source language to fit the model, or let programmers use the language as before, and provide a mapping through the compiler.

There is no absolute criterion as to which is better. You can find both approaches in current .NET language implementations. C++ and Eiffel for .NET provide contrasting examples.

The radical solution

C++ typifies the Procrustean solution: make the language fit the model. To be more precise, the name C++ on .NET denotes, on .NET, not one language but two: unmanaged and managed C++. Classes from both languages can coexist in an application: any class marked __gc is managed; any other is unmanaged. The unmanaged language is traditional C++, far from the object model of .NET; unmanaged classes will compile into ordinary target code (such as Intel machine code), but not to the object model. As a result they don't benefit from the Common Language Runtime and lack the seamless interoperability with other languages. Only managed classes are full .NET players..

But if you then look at the specifications for managed classes, you'll realize that you're not in Kansas any more (assuming, for the sake of the discussion, that Kansas uses plain C++). On the "no" side, there's no multiple inheritance except from (you guessed it) completely abstract classes, no support for templates, no C-style type casts. On the "yes" side, you will find new .NET mechanisms such as delegates (objects representing functions) and properties (fields with associated methods). If this sounds familiar, that's because it is: managed C++ is very close to C#, in spite of what the default Microsoft descriptions would have you believe.

Predictably, the restrictions also rule out any cross-inheritance between managed and unmanaged classes.

The signal to C++ developers is hard to miss: the designers of .NET are informing you that they don't think too highly of the C++ object model, and expect you to move to the modern world as they see it. The role of unmanaged C++ is simply to smooth the transition by allowing C++ developers to move an application to the managed side one class at a time. An existing C++ application will compile straight away as unmanaged. Then you'll try declaring specific classes as managed. The compiler will reject those that violate the rules of the managed world, for example if they use improper casts; the error messages will tell you what you need to correct to turn these classes into proper citizens of the managed world

For C++, this is indeed a defensible policy, as the language's object model — defined to a large extent by the constraint of backward compatibility with C, a language more than three decades old — is obsolete by today's standards.

Must languages adapt their semantics?

Only time will tell how successful the .NET strategy will be at convincing C++ programmers to move over to the managed world. But even if they wholeheartedly comply, it won't mean that other languages should follow the same approach. This is particularly true of object-oriented languages that have their own views of what O-O should be, with perhaps better arguments than C++. If you've chosen a language precisely because it supported such expressive mechanisms as multiple inheritance, Design by Contract and genericity, do you have to renounce them and step down to the lowest common denominator once you decide to use .NET?

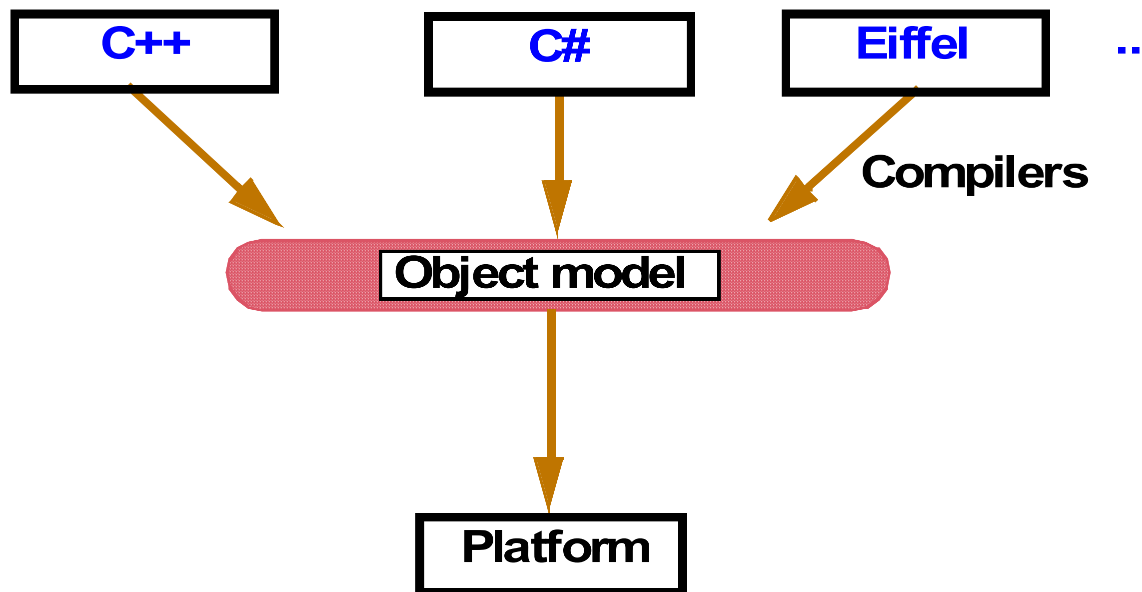
Fortunately the answer is no, at least not if "you" here means the application programmer. The scheme described in the first figure above doesn't require that all languages adhere to the .NET object model; rather that they *map* to that model. That mapping can be made the responsibility of compilers rather than programmers, enabling programming languages to retain their normal properties, and establishing a correspondence between the specific semantics of each language and the rules of the common object model.

Tune in next issue and discover how this all works out.

Part 2: Respecting other object models: the example of multiple inheritance

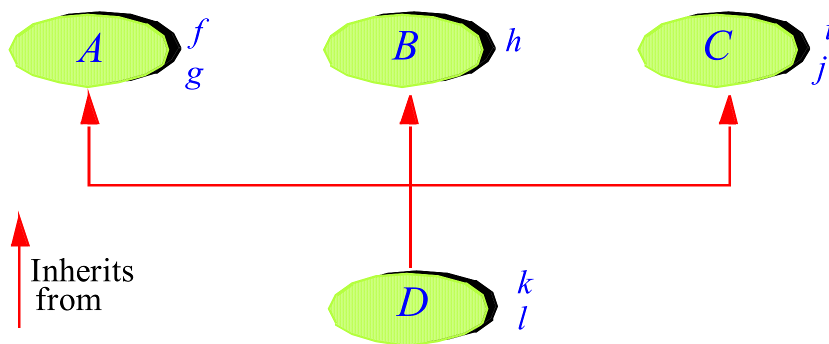
In the first article we discovered the power of multi-language combination in .NET but saw that C++ achieves it through a major language update: managed C++. Fortunately this is not the only possible approach. Let's see how languages can retain their own properties, all of them, while benefiting from the common object model and its promise of full interoperability with other .NET languages.

The initial picture, reproduced below, showed that languages have to be mapped into the object model. The trick will be to decide who takes care of the mapping: the language designer? The compiler writer? The application programmer?



The C++ answer put the omen on the language designer (to change the language) and the application programmer (to adapt programs). It's possible, however, to let the compiler writers do the job. This is good news for everyone — at least everyone who is not in charge of porting a compiler to .NET — since it means you can get the best of all worlds: continue using your language as before (presumably, if you chose that language it's that you like its properties); and yet rely on components coming from other languages.

The example of multiple inheritance will illustrate how language implementors can achieve this result.



Multiple inheritance from classes

Unlike the .NET object model, Eiffel supports full multiple inheritance: a class D may inherit from classes A, B and C. In .NET at most one of these three parents may be a class; the others, as we know, would have to be "*interfaces*", consisting only of methods devoid of any implementation. In Eiffel, there is no separate notion of interface; a class may be completely implemented, completely "*deferred*" (the equivalent of a .NET interface), or partially deferred, with some features implemented and others not. This full spectrum between the fully abstract and the fully concrete is essential to the seamless development approach of O-O software construction, making the approach applicable to analysis and design as well as implementation. You should be able to start system analysis and design with very abstract classes, close to the problem domain, then bring them progressively and seamlessly to a more concrete state with fewer and fewer deferred features, until nothing is deferred. In between, you will have many classes that have a mix of deferred and implemented features.

Central as it is to object technology, this possibility is not natively supported by .NET (or, for that matter, by Java). What if you want to apply O-O concepts properly in your language, and make D inherit from A, B and C that are not fully deferred?

The first reaction is that it won't work: the inheritance structure violates the rules of the .NET object model. The only solution then would be to follow the C++ route and change the programming language, creating a single-inheritance-only variant. (Such a subset, called Eiffel#, was indeed designed for Eiffel as a first iteration of the .NET version. Eiffel# was only a stepping stone and is now obsolete, as Eiffel for .NET now supports the full Eiffel language with multiple inheritance, through the techniques explained next.)

The second reaction notes that, taken from another angle, the problem is trivial. As long as it enables Eiffel code to use multiple inheritance, the compiler can hide any signs of inheritance from the rest of the world. Assuming for example that the classes have the features shown, then it suffices to show to the .NET world a class D that includes both its own immediate features (k and l) and those inherited from its parents (f, g, h, i, j), and has no inheritance relationship to A, B or C. This is known in the Eiffel environment as the **flat form** of a class: an inheritance-free equivalent with all the features, local and inherited.

With this technique, the inheritance structure will still apply within the Eiffel world, with its associated techniques of polymorphism and dynamic binding; for example with a1 of type A and d1 of type D you may have an assignment of the form a1 := d1. But to the rest of the world D is an orphan class.

The real challenge

The third natural reaction is sure to come soon: this solution is is not really acceptable. If you expose classes from an O-O language to other O-O languages, you'll want to expose their inheritance structure too. With the solution shown, C# or VB.NET code wouldn't be able to create an object of type B and, polymorphically, assign it to a variable of type A, even though this is legitimate in view of the inheritance structure, and permitted in Eiffel classes.

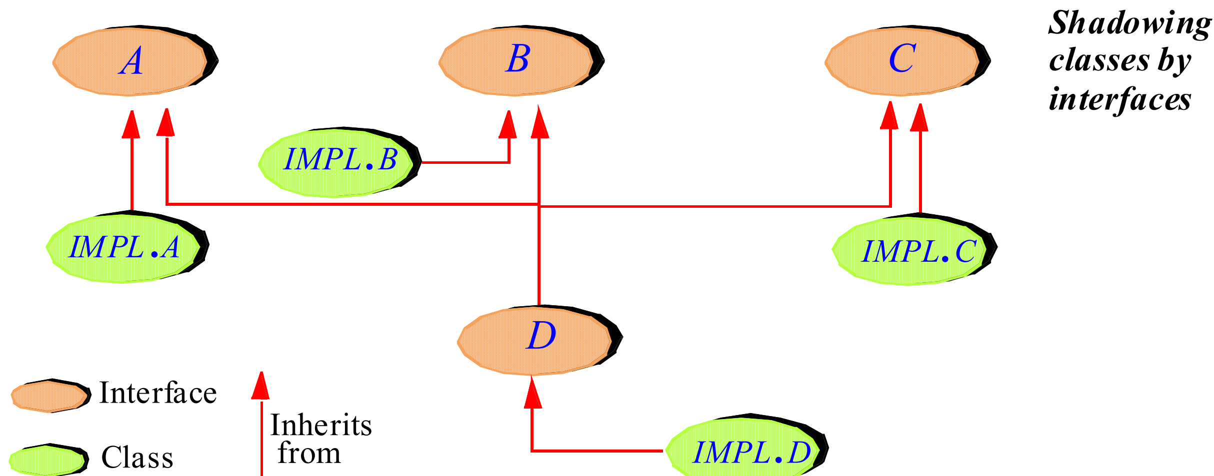
So we won't stop at this first solution, but it illustrates a key observation useful whenever someone asks whether it's possible to provide on .NET a language feature, not supported by the .NET object model, such as multiple inheritance, genericity or other. A gut reaction of the form ".NET can't support a multiple-inheritance language" is just silly. Supporting a particular language feature is a question for compiler writers. if you can compile multiple inheritance into C or into Intel machine code there is no reason you couldn't do it for the .NET virtual machine and its Intermediate Language (IL). But that is only part of the real question: in this multi-language environment, can we compile the construct *and* show other languages a reasonably accurate view of the original structure? In the multiple inheritance example, we may not be happy with a compilation technique that handles multiple inheritance but shows A, B, C and D to the rest of the world as if they were totally unrelated classes.

This is the true challenge of compiling on .NET a language that has multiple inheritance, or some other advanced feature not directly supported by the object model: how much of the original set of properties can we retain in the view that we present to components written in other languages?

Relying on interfaces

Let's see if we can show our partner languages a better view of our inheritance graph than just a flat structure with no inheritance links. A possible way of giving them more clues would be to designate as "favorite", for each class, at most one parent class, as in a troubled family where the children are ashamed of one of their parents. Based on some criterion, D would admit to our foreign-language friends that — for example — it inherits from A, hiding the others. The advertised inheritance structure would then conform to the .NET rules.

This solution, however, is unattractive. Under what criterion could we possibly choose which parent to retain? Multiple inheritance is precisely intended to let a class combine several equally important abstractions.



Shadowing classes by interfaces

Eiffel for .NET uses a better solution, taking advantage of the limited form of multiple inheritance permitted by .NET: multiple inheritance from interfaces For each Eiffel class, the Eiffel for .NET compiler produces both a class, called IMPL.A etc. on the figure, and an interface, which retains the original name of the class, for example A. The class, here , will inherit from these interfaces and nothing else. The original inheritance structure between Eiffel classes, single or multiple, is preserved among the .NET interfaces: you can see D inherit from A, B and C as the corresponding classes did in the original Eiffel.

All this is set up to allow writers of non-Eiffel modules to use the Eiffel classes through the resulting .NET interfaces, such as A, and not have to know about the implementation classes such as IMPL.A or anything else of the above scheme. a C# or VB.NET programmer may declare variables of the corresponding types like an Eiffel programmer would in :

```
A your_A_variable;   /* Declare variables with the appropriate types */
D your_D_variable;
```

You may then use polymorphism to assign a value of type D to a target whose type is any one of A, B and C; for example:

```
your_A_variable = your_D_variable
```

If a non-Eiffel class inherits from D, browsing tools — such as those of Visual Studio.NET — will show A, B and C among its ancestors.

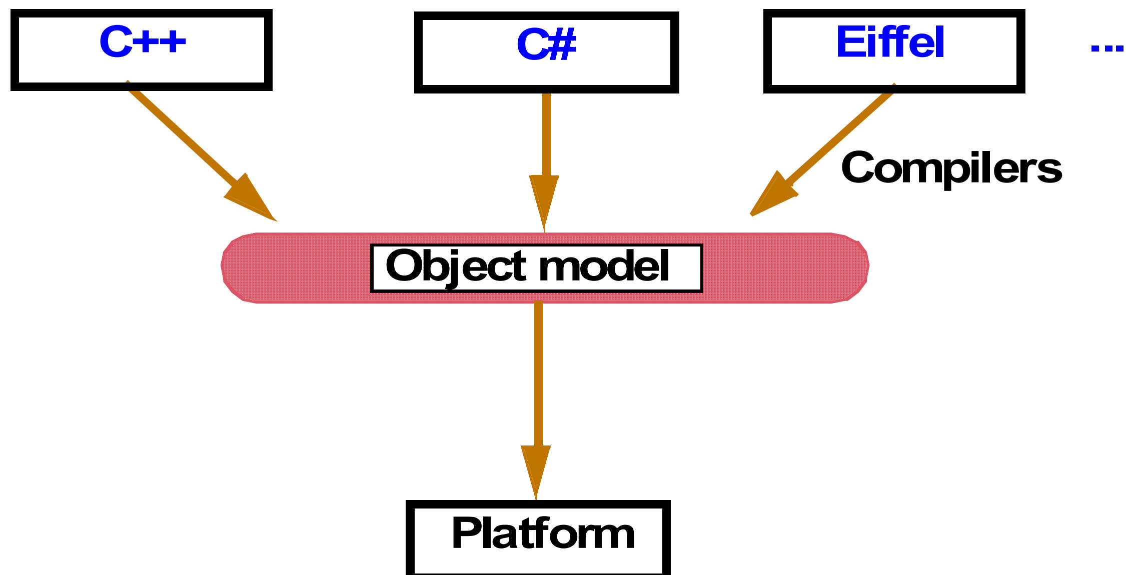
What about creating the corresponding *objects*? Interfaces don't have instances or constructors; the objects you'll want to create are instances of the IMPL … classes, created using the appropriate constructors. The technique finally retained (because another one, using static methods, doesn't satisfy the CLS rules to be seen in the last part of this article) is to generate yet another .NET class, called CREATE.A, for any non-deferred Eiffel class A. The generated class contains all the constructors (Eiffel's creation procedures) for A, which non-Eiffel clients will use to create instances of A. This is easy to explain in the class documentation.

Note how this architecture takes advantage of various .NET mechanisms. In particular, Impl and Create are .NET namespaces. The result is to export the full power of Eiffel's multiple inheritance to any .NET language, even though the .NET model doesn't by itself support multiple inheritance.

The language bus

The implementation of multiple inheritance in Eiffel helps us understand the relationship between .NET's multi-language nature and its object model.

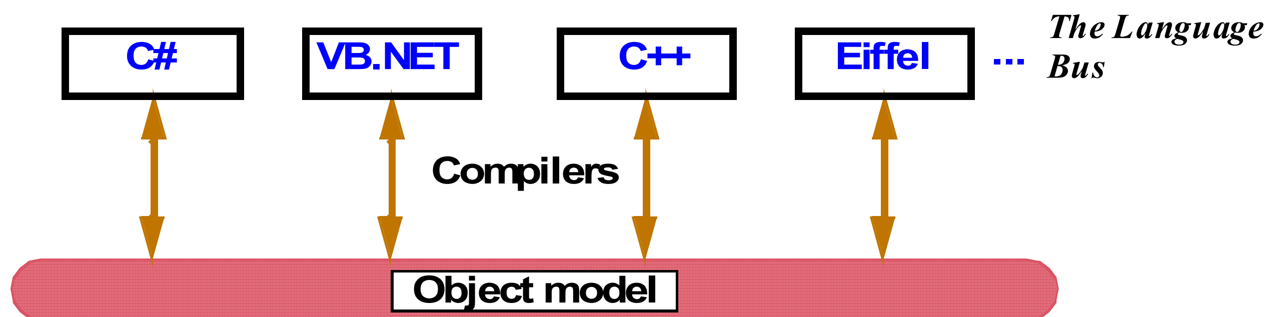
At first, the existence of a single object model seems to contradict the claims of language openness. This has led some people to dismiss those claims as hype and state that .NET really imposes a single language, or at least a single semantics. This view seems confirmed by the example of managed C++, But the example of multiple inheritance shows that it's wrong. .NET doesn't require languages to adopt the common object model; any language can keep its own view of the world, as long as the compiler is able to translate it to that common model. In our original picture



the responsibility of the mapping rests with the compiler, not the application programmers. It's also the compiler's task to let other languages retain as much as possible of the original view when they access the original software elements.

We have seen this work in the case of multiple inheritance: by cleverly using a common mechanism (multiple inheritance from interfaces), we can give other languages an essentially accurate view of the original inheritance structure, and let them pretend that they support multiple inheritance even when don't.

So what the common object model provides is not a stranglehold forcing all languages to support a single view; instead, it is a kind of language bus, enabling all languages to cooperate by agreeing on a basic set of common mechanisms, as suggested by this reinterpretation of the earlier figure:



Multiple inheritance was a felicitous case since we were able to present to the bus, and hence to other languages, a view that doesn't lose any essential property of the original model.

Projecting the right image to the rest of the world

We won't always be as fortunate, since some high-level constructs may be impossible to emulate in the object model. An example is genericity. As noted, Eiffel, and unmanaged C++ through its "templates", let you define type-parameterized classes. known as generic. The class LIST [G] denotes lists of objects an arbitrary type. G is known as its *formal generic parameter*. The type LIST [INTEGER] then denotes lists of integer objects; LIST [EMPLOYEE] denotes lists of employee objects. This is all type-checked: if n is of type INTEGER and e of type EMPLOYEE, the compiler for a generic language will accept

```
emplist: LIST [EMPLOYEE] ; intlist: LIST [INTEGER]

e: EMPLOYEE ; n: INTEGER

emplist.extend (e)    -- Procedure `extend' adds an element at end of list
intlist.extend (n)
```

but will reject emplist.extend (n) and intlist.extend (e).

Like Java, the .NET model, and as a consequence languages such as C# and VB.NET, do not support genericity: if you want the equivalent of a generic list class, you let it manipulate values of the most general .NET type, Object, and you cast back and forth between Object and the actual types needed, such as EMPLOYEE and INTEGER. This means that instead of the compile-time checks of a generic language you have to rely on run-time type checks, a penalty in both software reliability and performance.

Here too, as with multiple inheritance, a generic language can continue to benefit from its own mechanisms and enforce its own rules; but unlike with multiple inheritance, we don't find a ready-made mechanism (such as interface inheritance in the earlier cases) to emulate these mechanisms in the common object model. So when a generic class such as LIST is made available to the rest of the .NET world, it will appear exactly as if it were a non-generic .NET class, describing lists of Object values.

This summarizes the general rule for multi-language support in .NET:

MULTIPLE LANGUAGES UNDER .NET

- 1 No language needs to change for .NET.
- 2 The first task of a .NET compiler for any language L is to map the constructs of L to those of the .NET virtual machine, as it would do for any other target machine. As a result, different modules written in L can make full use of L's capabilities, and communicate with each other exactly as they would outside of .NET.
- 3 The compiler's second task, to let L play its full part in the .NET interoperability game, is to produce, for each of the constructs of L, an equivalent or emulation in the .NET object model, allowing other languages to interact with L modules in terms of the common concepts (the “language bus”).
- 4 In building that exported view of L constructs, the compiler may be able, for some constructs, to retain all of the original properties, as in the case of Eiffel's multiple inheritance; or it may have to forsake some of the more advanced properties, as in the case of genericity.

In the final article, we will see what remains to be done, with such a compiler, to benefit from the full extent of language interoperability promised by .NET. Come back for a description of a key property of .NET which, surprisingly, most current .NET books don’t cover or even mention: the CLS, or Common Language Specification.

Part 3: Interoperability: at what cost, and with whom?

In the first two articles we saw the unprecedented level of interoperability that .NET provides to language developers if compiler writers have done their job. What is the price to pay?

That price involves observing a number of constraints:

- First, you must, obviously, use a language for which a compiler is available for .NET. This means that the compiler is able to generate IL code that follows the rules of the .NET object model.

- Your code may also need to be *verifiable*, that is to say, pass the security rules imposed by the .NET security model. This is not an absolute requirement, and for some languages — especially untyped languages — it will never be met. For others, verifiability may be a compiler option: upon request, the compiler will generate verifiable code, perhaps by imposing some further requirements on your source texts, and possibly at some performance cost. You should strive for verifiability if you want your code to be used on sites that may not automatically trust you, for example if it is to be downloaded through the Internet.

- Finally, if your code must interface with software elements written in other .NET languages, you must make it **CLS-compliant**.

The role of the CLS

The CLS, or Common Language Specification, is a set of forty-one rules meant to ensure harmonious cooperation between .NET languages. It's part of the "*Common Language Infrastructure*", the specification of the basic .NET architecture standardized by the ECMA standards body (you can find a copy of the ECMA specification at http://www.dotnetexperts.com) . The rules play two complementary roles:

- They **restrict** what you can write; more precisely, what your language's compiler can do with your classes if they are visible to the outside world. Such rules come in addition to the requirements imposed by the .NET object model. For example, rule 40 specifies that any exception that is "thrown" (triggered) must be an object whose type is a descendant of the library class System.Exception, whereas IL would actually let you trigger exceptions of other types. If you want your language to be CLS-compliant, the compiler must ensure this rule in exported classes.

- The rules **require** that you accept foreign code that satisfies appropriate conditions. For example, the just cited rule 40 also requires of any language that it provide an exception mechanism, able to handle objects of type System.Exception.

In the first of these roles, the CLS specification is negative: it states what you may not offer to other languages. Its second role is positive: the rules state what you are required to accept from other languages. The contradiction is only apparent, since the specification's two roles are complementary: to enable various language models to communicate, the CLS must define the *maximum* of what a language may export to the rest of the world, and the *minimum* that it must be able to import from the rest of the world. Compliance is in fact defined at three levels, not just two, because you may "import" a class either by being its client or by inheriting from it:

THE THREE FACETS OF CLS COMPLIANCE

Framework CLS compliance:
Only use constructs permitted by the CLS rules.
Consumer CLS compliance:
Permit use of any framework-compliant class.
Extender CLS compliance:
Permit inheritance from any framework-compliant class.

A software product is framework-compliant if the compiler-generated code that it exposes to the rest of the world observes the CLS rules. For example it should never trigger any exceptions other than those of type System.Exception or a descendant. A language implementation is consumer-compliant if it enables its classes to "consume" (be a client of) framework-compliant classes. It is extender-compliant if it enables its classes to "extend" framework-compliant classes, that is to say, inherit from them with all the associated privileges: adding new features, redefining (overriding) inherited features. Extender compliance is a pretty taxing requirement, since it means that you must find a way to support or at least emulate all framework-compliant language mechanisms, including some that might be quite far from the core concepts of your language.

Some rules of the "negative" kind actually facilitate the task of consumers and extenders. For example, rule 10 says that when you are overriding (redefining) an inherited routine you may not change its export status, even though the basic object model would let you do that. This is a protection for extender languages, since it means they are not required to provide a mechanism for changing the export status of inherited routines.

It's all for show: CLS compliance in practice

Some of the CLS rules can appear scary at first if you rely on a language model that is significantly from the Java-C#-VB.Net family. To keep them in perspective, remember the following caveats:

- CLS compliance only matters for software elements that you wish to export to modules written in other languages, or import from them. So as long as you are talking to your own friends in your own team, you can indulge in whatever pleasures you have enjoyed in the past. It's only when other teams join the game that you must start thinking about maintaining proper appearance.

- Even then, it's only about appearance. What you really do is between you and your conscience; CLS compliance only matters for what you show to the rest of the world. As long as the view you present is CLS-compliant, it's no one's business that it might serve as a cover to non-CLSically-correct games. Don't ask, don't tell.

Let's explore these two points further, if only to reduce the risk of a heart attack when you encounter some of the actual rules. The first is important because of the practical nature of multi-language applications. It is not realistic, in an application containing C++, C#, Eiffel and Cobol elements, to expect that all of them will talk to elements written in one or more of the other languages A project is multi-language because it consists of a number of subprojects each written in a particular language; in practice, each subproject will usually include a few **bridge modules** that talk to other languages. CLS compliance only affects these bridge modules, typically a small subset of the software. Everywhere else, each subproject can behave as if it were single-language.

The other main source of multi-language combination, cited at the beginning of this article, is the use of libraries from another language. The visible classes of such libraries should be framework-compliant; classes that use them will have to be consumer-compliant and, for the usually small subset that needs to inherit from library classes, extender-compliant.

The complementary observation is about appearance. Even in a bridge module you can often pursue CLS-deviant practices as long as you present them, for outside consumption, in CLS-compliant clothing. Here is an example. Rule 16 specifies that CLS arrays start their indexing at zero. Counting from zero is part of the sad legacy of C. (How many fingers on your right hand? Count with me: zero, one, two, three, four!) What if your language indexes arrays from one, as in Fortran, or lets you specify arbitrary bounds, as in Eiffel? Well, all that really counts is to pretend to the rest of the world that your array is a good CLS citizen. So if you have a Fortran array indexed from 1 to 100, or an Eiffel array from 1901 to 2000 (perhaps to keep information associated with years of the twentieth century), your CLS-aware compiler will export it to the rest of the world as if it were a solid 0-to-99 citizen.

So whenever you feel like despairing at a seemingly unacceptable rule, remember that it's only for bridge classes, and only for show. It may look like you have to adopt the state religion, but in practice you may get away, much of the time, with a few genuflections in the right public places.

Enforcing the CLS

In presenting the CLS I have been freely relying on the term "you", as in "you must only use exceptions of a type conforming to System.Exception". Who really is "you": the application programmer? The language designer (in other words, could *you* mean me)? The compiler?

All these "you" are actually involved::

- 1 To ensure extender compliance, the language designer must ensure that the language has mechanisms to emulate all the constructs supported by the CLS.
- 2 To enable the production of framework-compliant code, the compiler writer must provide at least as an option the possibility of generating framework-compliant code.
- 3 To produce CLS-compliant subsystems or libraries, the application programmer must stay away — for the "bridge classes" intended to interact with other languages — from non-CLS constructs.

The last two points go together. A programming language will, almost inevitably, offer mechanisms that are not CLS-compliant. Even in C#, whose semantics is closest to the .NET object model and the CLS, you can easily write non-CLS-compliant code, for example by using non-CLS compliant types such as native unsigned int. If, as a programmer, you want to be sure that certain classes are CLS-compliant, you will ask the compiler to generate CLS-compliant code through a compiler option. For a class flagged with this option, the compiler might:

- Reject the class if it uses language constructs that the compiler can't map to CLS-compliant .NET constructs.
- Accept the class, but for certain constructs generate different code (for example, less efficient) from what it generates when CLS compliance is not required.

Instead of a compiler option, the current C# compiler produces warnings on request. The Eiffel compiler provides a compiler option, which doesn't reject any construct but may in some cases generate different code.

If you have requested the generation of CLS-compliant code, the code will be marked as such, at either the class or assembly level. This is achieved through a custom attribute, System.CLSCompliantAttribute, whose constructor takes a boolean argument, True signifying compliance. This is specified by rule 2: "Members of non-CLS compliant [classes] shall not be marked CLS-compliant". The attribute can be set for a class, or for an entire assembly, in which case individual classes may override the value set for the assembly.

If, in a consumer or extender language, you write a class that is a client of a foreign class, or inherits from it, the compiler for your language will check that the System.CLSCompliantAttribute is set to True for the foreign class.

Life with the CLS

The example of dealing with arrays through rule 16 illustrated how one can cope with the CLS, even in a language with a different view of the world. Here are a couple more examples.

Rule 5 states that all the names used in a class must be "distinct" except where the names are identical and "resolved via overloading". Overloading, also known as the most masochistic device introduced in the history of humankind, means that you can give the same name to several methods, as long as they differ by at least one argument types. This is a rare example of a facility that has no known advantage whatsoever, and many documented problems (it's confusing, and conflicts with O-O mechanisms such as polymorphism and redefinition). Nevertheless, under the influence of C++ and Java, it's in the .NET object model and, worse, in the CLS.

Should you care about overloading if your language doesn't have it and you want to be CLS-compliant? The answer would seem to be no, as you can just ignore this mechanism, like a computer's machine instruction that a compiler never uses for its generated code. But this only covers *framework* compliance. To be *consumer*-compliant, and especially *extender*-compliant, you do have to care. Extender compliance means that a class in your language must be able to inherit from a class that includes overloaded methods, for example a C# class with the methods

```
write (string s)
write (int n)
write (float r, string format)
```

In the original, they are all called write, but in your own class this would cause an error since you can't have overloading.

At first this case sounds like a show-stopper until you realize that if you must be able to inherit overloaded methods nothing forces you to inherit them under their original *names*. That would be an impossible requirement to enforce anyway since various languages have different naming conventions: Visual Basic, as we noted in the first article, allows hyphens in identifiers, which most other languages reject; C lets you start an identifier with an underscore, but this is not universal. So what overloading means in practice is that compilers for consumer and extender languages have to provide a mechanism for "*demangling*" (name ambiguity resolution), to make sure names that were mangled in the original look different to their users. For example the above methods could be renamed, using a simple demangling algorithm based on the types, into

```
write_string
write_int
write_float_string
```

The Eiffel team has proposed such an algorithm as an informal standard, to avoid every non-overloaded language reinventing its own conventions. The only negative remaining consequence of overloading is that consumers and extenders of overloaded languages will need some extra documentation, since the original documentation includes overloaded names that are not directly usable.

The support for overloading in the CLS is a design mistake. Even if it had a conceptual justification, overloading would still be a language concept, and one that concerns not the deeper semantic properties of a language but the external appearance of software texts, a mere facility for the program writer. It has no place in a general O-O model, even less in a scheme like the CLS whose very purpose is to enable many languages to collaborate. Overloading languages should never have been permitted to pollute the common conceptual setup with a marginal mechanism that complicates everyone's job. Instead, overloading should have been explicitly removed from the CLS, putting the onus on the overloading languages to provide a demangling algorithm to clean up any mangled names.

At least there is a fairly easy way out, as in the case of arrays. Some rules are harder to deal with for certain languages. Rule 19, which prohibits interfaces from including static methods, complicates the emulation of multiple inheritance in a CLS-compliant way; this is all the more regrettable that the rule limits the usefulness of interfaces and goes against the principles of object technology.

Other problems arise with rules 21 and 22 which impose on everyone creation policies coming from C++ and hard to justify through rational arguments: the need for constructors of a class to call constructors of parents; and the impossibility of using a constructor to reinitialize an object. (Since it's often necessary to reinitialize objects, what this rule means in practice is the need to duplicate methods and their code, always a major obstacle to good software practices.)

Here too, languages with different models can find ways to cheat, but these two rules cause more nuisance to their users. They go the furthest in trying to impose a single language model — like the Java model — onto everyone. This danger is ever-lurking in the .NET object model, and should be fought with the utmost energy since it threatens the whole purpose of the technology. Fortunately, most of the CLS rules are reasonable and do not cause any major trouble for other .NET languages.

Towards a programming language renaissance

The CLS completes the careful and innovative design of multi-language support under .NET. Only people who have not looked carefully enough can push the “common denominator” view (the assertion that it’s all a single language anyway). That’s plain wrong.

What .NET provides is the ability to map languages into a common model and hence obtain interoperability, while preserving the originality and independence of each language. This architecture holds the potential of a programming language renaissance, enabling languages to compete on merits, not political prejudice, and the field to blossom like never before.