Full Eiffel on .NET†

[+]Raphael Simon, [+]Emmanuel Stapf, [+]*Bertrand Meyer

[+]Interactive Software Engineering, Santa Barbara, California
*ETH (Swiss Federal Institute of Technology), Zürich, Switzerland

http://www.eiffel.com/, info@eiffel.com

Abstract

The full power of the Eiffel language and method, including Design by Contract™, multiple inheritance, genericity and many other advanced facilities, is now available on the Microsoft .NET framework.

Eiffel for .NET establishes a powerful basis for the construction of extendible, high-reliability applications, providing a unique platform for integrating components produced with many different languages and approaches, and bringing the benefits of Design by Contract to the .NET world.

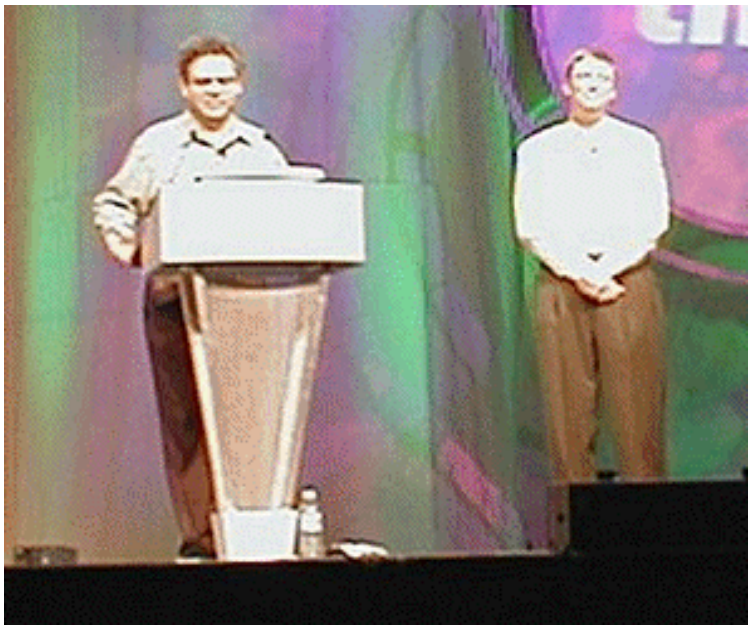
We describe the work done to integrate the two technologies, and the resulting tools for constructing ambitious enterprise and Web systems.

†An earlier version of this article appeared in July, 2000 under the title *Eiffel on the Web: Integrating Eiffel systems into .NET* (by the present authors and Christine Mingins). The present version describes the current state of the implementation, supporting the full Eiffel language.

1. INTRODUCTION

One of the most interesting aspects of the Microsoft .NET framework is the common basis it provides for implementing many different programming languages. One of the first language technologies to benefit from this openness was Eiffel, whose implementation by Interactive Software Engineering (ISE) was, in an early version, showcased at the very first public introduction of .NET in Orlando (see figure). That initial release featured a partial version of the Eiffel language, Eiffel#, described in the original version of this article and now obsolete. ISE has now completed the implementation of full Eiffel on .NET and a first integration into Visual Studio .NET.

Eiffel for .NET is a released product, available as part of the ISE Eiffel delivery starting with version 5.0 (the current release at the time of writing is 5.1).



Bill Gates (Microsoft) and Bertrand Meyer (ISE) at introduction of .NET technology, July 2000.

2. EIFFEL AND .NET: AN OVERVIEW

Eiffel for .NET is a full implementation of the Eiffel method and language running on the Microsoft .NET platform.

Eiffel is a comprehensive software development environment (ISE Eiffel) based on a method that covers the entire software lifecycle — not just implementation but also analysis, design and maintenance. The environment is based on the Eiffel language, thoroughly applying the principles of object technology and implementing the concepts of Design by Contract™ to produce highly reliable, extendible and reusable applications.

ISE Eiffel is particularly geared towards large and complex systems and is used by major organizations in the financial industry, defense, real-time and other industries for mission-critical developments. Universities worldwide also use Eiffel to teach programming and software engineering at all levels.

.NET is the next generation web technology developed at Microsoft, combining many technologies for building Internet applications. The specification of .NET is now an international standard, thanks to Microsoft's successful submission of the "*Common Language Interface*" to the ECMA standards organization, which adopted it in December of 2001. (One of the authors, Emmanuel Stapf from ISE, is a member of the corresponding ECMA Technical Committee.) Although a detailed presentation of .NET is beyond the scope of this article, we may note the following highlights, of special interest to application developers:

- The architecture relies on a virtual machine, so that compilers for any language always generate the same code, IL (Intermediate Language).

- The code that gets executed on any actual computer is native (binary) code for that computer, translated incrementally or not through a process known as JIT (best understood as meaning Judiciously Incremental Translation).

- The virtual machine's equivalent of an operating system is the Common Language Runtime (CLR), providing a number of crucial facilities — memory management, garbage collection, security, exception handling — to programs regardless of their language or origin (hence the word "common').

- The memory model used by the virtual machine and the CLR does not rely on addresses, bytes and words; instead, it is an *object-oriented* model based on the notions of type, class, object, inheritance, polymorphism, typing and dynamically bound calls.

- The language interoperability mechanisms of .NET, including the Common Language Runtime, IL, the object model and the Common Language Specification (CLS), enable the various parts of an application to use different programming languages — each chosen to be the best for the job at hand — and to achieve a degree of inter-language cooperation unprecedented in the software world. Not only may a module call a routine written in another module; a class in an object-oriented language may inherit from a class in another; exceptions cross language boundaries; so do debugging sessions; and all this is achieved without any special effort on the programmer's part, and without any need for languages to know about each other.

- A new development environment, Visual Studio .NET, provides advanced development facilities — compilation, browsing, debugging, user interface development — and is, like the rest of the technology, open to many languages.

- .NET provides thousands of reusable components extending across many application areas, from localization to networking and language analysis.

- Among the most important component libraries are ASP.NET, an innovative framework for building smart Web sites; ADO.NET, an object-relational interface library; and Windows Forms for graphical applications.

- ASP.NET and other Web-oriented mechanisms of .NET open the way to major advances in Web services and other advanced uses of the Internet.

- These mechanisms are potentially available to developers using any programming language — provided the implementors of that language offer a compiler that's compatible with .NET, not only by generating IL but also by observing the .NET rules of language interoperability.

.NET is attractive to Eiffel users since it follows many of the same ideas that they have accepted as essential to quality software development — use of an object model, automatic garbage collection, exception handling — and offers an integrated platform with direct access to thousands of reusable components, the prospect of full

interoperability with software elements written in both Eiffel and other languages, and the power of Web services and other Internet applications.

For .NET users, Eiffel provides the added value of an advanced object-oriented method and language that covers the entire lifecycle — not just programming, but the whole process starting with analysis and design and continuing with implementation, reuse, extension and maintenance —, unique reliability mechanisms such as Design by Contract™, advanced language features such as genericity and multiple inheritance, and the extensive body of reusable components developed by ISE and other parties, including the EiffelBase library of data structures and algorithms and the EiffelVision library for multi-platform graphics.

Eiffel on .NET provides an ideal combination for companies wishing to take advantage of best-of-breed technologies in operating systems, Internet and web infrastructure, software development methods, and development environments. In particular, the openness of Eiffel to other languages and environments combined with .NET's emphasis on language neutrality makes the resulting product an ideal vehicle for building applications containing components in many different languages, Eiffel serving as the "glue" between them. In the rest of this article we describe this combination and the challenges we faced when integrating ISE Eiffel into .NET.

3. ABOUT EIFFEL

Since the rest of this document defines Eiffel for .NET by describing how it is different from Eiffel, we first need to see the main characteristics of Eiffel. More details may be found in the book *Object-Oriented Software Construction, 2nd edition* [Meyer 1997] and *Eiffel: The Language* [Meyer 1992], as well as on the Eiffel Web site at http://www.eiffel.com/, from which some of the material has been extracted.

Eiffel is the combination of four elements: a method of system development, based on strong software engineering principles; a language supporting the method; a development environment, ISE EiffelStudio; and a rich set of reusable libraries.

Method

The Eiffel method, language and environment emphasize *seamless development*, the continuous production of a system through the successive phases of the lifecycle using a common set of notations and tools. The language, in particular, is as useful for analysis and design as for programming in the strict sense of the term. The tools provide graphical descriptions of system structures and enable developers both to produce software text from the graphics and to reverse-engineer the graphics from the text, switching at their leisure between the two modes. This means that Eiffel developers typically do not need a separate "CASE" tool (for example UML-based) for analysis and design but instead use a consistent framework throughout the process.

This seamless approach also supports *reversibility*: if a modification is made to the program, it will automatically be included in the analysis and design views. Since these views, like others graphical and textual views at various levels of detail, are

extracted from the software text by automatic tools, the various documents associated with a project are guaranteed to be consistent. This follows from the *Single Model* principle, one of the principles of the Eiffel approach.

Language

As a language Eiffel is a "pure" object-oriented language (arguably the most systematic application of object-oriented principles in existing languages) based on a small number of powerful concepts:

- Classes, serving as the sole basis for both the module structure and the type system.
- Inheritance for classification, subtyping and reuse.
- A careful and effective approach to multiple inheritance (renaming, selection, redefinition, undefinition, repeated inheritance).
- Contracts for writing correct and robust software, debugging it, and documenting it automatically.
- Disciplined exception handling to recover gracefully from abnormal cases.
- Static Typing for reliability and clarity.
- Dynamic binding for flexibility and safety.
- Genericity, constrained and unconstrained, for describing flexible container structures: you may declare a class VECTOR [G] to state that it will describe vectors of elements of any type, G denoting a "formal generic parameter"; to derive a usable type you provide an "actual generic parameter" as in VECTOR [INTEGER] (describing vectors of integers) or even VECTOR [VECTOR [INTEGER]] (vectors of vectors of integers).
- Covariance, enabling the flexible adaptation of routines when redefined in descendants of the class where they originally appeared.
- Agents: high-level functional objects describing partially bound routines, providing the power of functional languages in an object-oriented context and a type-safe way.
- Open architecture providing easy access to software written in other languages such as C, C++ and others.

For a flavor of the — clear and simple — language syntax, and the typical Eiffel style (not yet in this first version including contracts), here is the outline of a simple class *COUNTER* describing a counter:

```
indexing
        description: "[
                        Counters that you can increment by one,
                        decrement, and reset
                        ]"
class
        COUNTER

feature – Access

        item: INTEGER
                -- Counter's value.

feature -- Element change

        increment is
                -- Increase counter by one.
                do
                        item := item + 1
                end

        decrement is
                        -- Decrease counter by one.
                do
                        item := item - 1
                end

        reset is
                        -- Reset counter to zero.
                do
                        item := 0
                end

end -- class COUNTER
```

At run time this class will have instances: each instance is an object that represents a separate counter. To create a counter you declare the corresponding entity, say

```
my_counter: COUNTER
```

create the corresponding object

```
create my_counter
```

(where the keyword **create** introduces the the object creation operation), and can then apply to it the operations of the class (its features):

```
my_counter.increment
...
my_counter.decrement
...
print (my_counter.item)
```

Such operations will appear in features of other classes, called the **clients** of class *COUNTER*. A couple more comments about this example: all values are initialized by default, so every counter object will start its life with its value, *item*, initialized to zero (you don't need to call *reset* initially). Also, *item* is an attribute, which is exported in read-only mode: clients can say print (*my_counter*.*item*) but not, for example, *my_counter*.*item* := *657*, which would be a violation of the "information hiding" principle. Of course the class author may decide to provide such a capability by adding a feature

```
set (some_value: INTEGER) is
                -- Set value of counter to some_value.
        do
                item := some_value
        end
```

in which case the clients will simply use *my_counter*.*set* (*657*). But that's the decision of the authors of class *COUNTER*: how much functionality they provide to their clients. The indexing clause at the beginning of the class does not affect its semantics (i.e. the properties of the corresponding run-time objects), but attaches extra documentation to the class.

Design by Contract

Alone in design methodologies and languages, Eiffel directly enforces *Design by Contract™* through constructs such as class invariants, preconditions and postconditions. Assume for example that we want our counters to be always non-negative. The class will now have an invariant:

```
indexing ... class
        COUNTER
feature
        ...
invariant
        item >= 0
end
```

and feature *decrement* now needs a precondition, to make sure that clients do not attempt illegal operations. The keyword **require** introduces the precondition:

```
decrement is
                -- Decrease counter by one.
        require
                item > 0
        do
                item := item - 1
        ensure
                item = old item - 1
        end
```

The keyword **ensure** introduces the postcondition.

The precondition tells the client: *"Never even think of calling me unless you are sure the counter is strictly positive"*.

The postcondition says *"If you are good (i.e. observe the precondition) here is what I promise to do for you in return: I will decrease the counter by one."*

The invariant adds the promise that *"Also, all my operations will maintain the counter positive or zero"*. Preconditions, postconditions and invariants are called assertions.

Libraries

Eiffel emphasizes reuse at all steps and is supported by a rich set of libraries, carefully crafted with strict design and style guidelines. Two worth noting here are EiffelBase, covering the set of fundamental structures of computing science, and EiffelVision, an advanced graphical library providing portable graphic solutions across many platforms, which offers users the guarantee of both source-level compatibility and automatic adaptation to the look-and-feel of the target platform.

Challenges

This brief introduction to Eiffel has enough to suggest some of the issues that arose in the .NET integration. The .NET object model provides no native support for multiple inheritance (a class in the .NET model may inherit from at most one other class), for genericity, for covariance, for agents.

Several of these mechanisms, in particular multiple inheritance, proved difficult to implement under .NET. They have all now been successfully tackled, so that there is no difference in the language supported under Eiffel for .NET and other implementations.

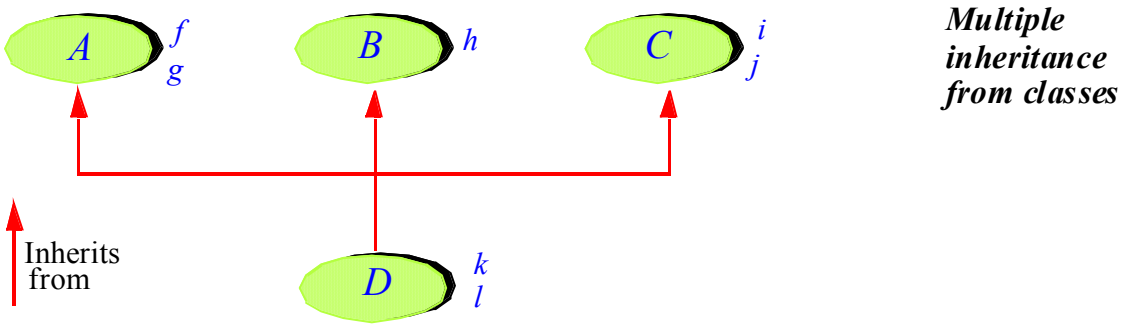
4. HOW EIFFEL RUNS UNDER .NET

Targeting .NET for a language compiler really means being able to produce IL and the associated metadata.

Goals

Generating IL would be enough if the aim was just to "compile Eiffel under .NET", but would fall short of our goal of providing a general-purpose framework for multi-language interoperability, since other languages would not be able to reuse Eiffel types without the metadata that describes them. Multiple inheritance provides a

typical example: producing an IL version of a multiple inheritance structure as shown below is, in itself, just a compiling issue, and not necessarily more difficult than implementing multiple inheritance for some other target code. The more ambitious issue is to make sure that code using these classes in another language can see the inheritance hierarchy and benefit from it, for example by declaring a variable of type A and assigning to it a value of type D (as Eiffel code can do through polymorphism).



One of the goals set by ISE regarding the integration of Eiffel was the ability to reuse existing types written in any language as well as the generation of types that can be understood by any other .NET development environment. Eiffel is a .NET *extender*, meaning that you can write Eiffel classes that inherit from classes written in other languages, extend them and then recompile them to IL, giving other environments the possibility of reusing the new type.

Another fundamental goal, in making Eiffel a full player in the .NET interoperability games, was to provide ISE Eiffel under Visual Studio .NET. As a result, Eiffel users have a choice between two modes of development:

- For an environment that is fully devoted to Eiffel, they can use the EiffelStudio environment.
- For multi-language development and close integration with other languages, for example multi-language debugging, they can use Eiffel under Visual Studio .NET.

An associated design goal was to avoid forcing users into a final choice between these two solutions: it must be possible to compile a given project alternatively in EiffelStudio or Visual Studio .NET.

Finally, it was deemed essential to enable the writing of ASP.NET applications and Web services in Eiffel, embedding Eiffel into ASP+ pages through the **@language="Eiffel"** directive.

Properties of the implementation

Giving Eiffel the status of "full player in the .NET interoperability games" has meant achieving the following properties of the implementation of Eiffel for .NET:

- Eiffel is, starting with version 5.2, fully integrated in Visual Studio .NET, taking advantage of the environment's mechanism for editing, compiling, cross-language browsing and (particularly important in practice) cross-language debugging. Visual Studio "solutions" have exactly the same status as those in other languages, and may integrate (or be integrated into) solutions in other languages.

- Eiffel for .NET generates **managed code**: the generated code runs under the control of the .NET Common Language Runtime (CLR), follows its constraints, and takes advantage of its mechanisms for memory management, garbage collection, exception handling, security, debugging and others. On .NET platforms, ISE Eiffel uses a runtime system that addresses similar issues; on .NET, its functions are taken over by the CLR.

- Eiffel for .NET generates **verifiable code**: you can produce code that will satisfy the .NET security requirements.

- Eiffel for .NET generates **CLS-compliant code**: the generated code satisfies the requirements of the Common Language Specification, a set of rules, now part of the international standard for .NET, that guarantees that modules produced from one language can be reused by others. This makes Eiffel an ideal tool for producing high-quality reusable .NET components, which any other .NET application, written for example in C# or Visual Basic.NET, can freely rely on.

- Eiffel for .NET is also **CLS-consumer** and **CLS-extender**: this means that Eiffel classes can use CLS-compliant code from other languages, and even inherit from a CLS-compliant class in any of these languages and add or redefine features.

- Eiffel for .NET is compatible with the **CodeDom** mechanisms, ensuring possible translation into other CodeDom languages, and usability as source language in **ASP.NET** for smart web pages and web services.

- Whether within Visual Studio .NET or independently from it, Eiffel for .NET is compatible with the **debugging** and **exception** mechanisms of .NET. A run-time error triggered and not processed in a non-Eiffel module will be handled by its Eiffel caller, and conversely.

- As a particularly significant consequence, contract violations detected on the Eiffel side (if contract monitoring is on) will be passed as exceptions to non-Eiffel callers. This equips applications with an invaluable technique to detect errors and improve their reliability by taking advantage of Eiffel's Design by Contract facilities.

Practical setup: EiffelStudio

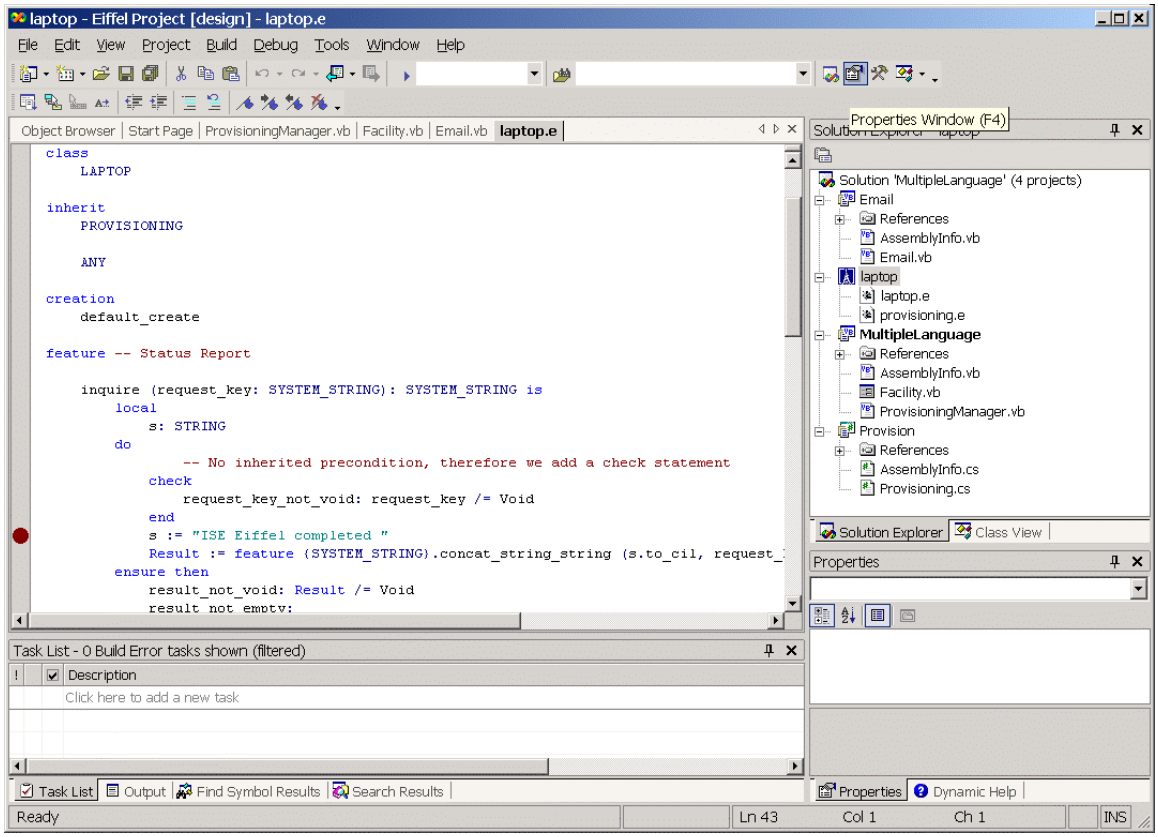
Under EiffelStudio, compiling for .NET simply means checking the appropriate option in the Project Settings. As a result, a few supplementary buttons will appear in the interface, including the button for the "assembly manager", discussed in the next section.

The result of this setup is that existing Eiffel programmers will be able to work the exact same way they did before the integration, while having access to all the mechanisms of .NET.

Practical setup: Visual Studio .NET

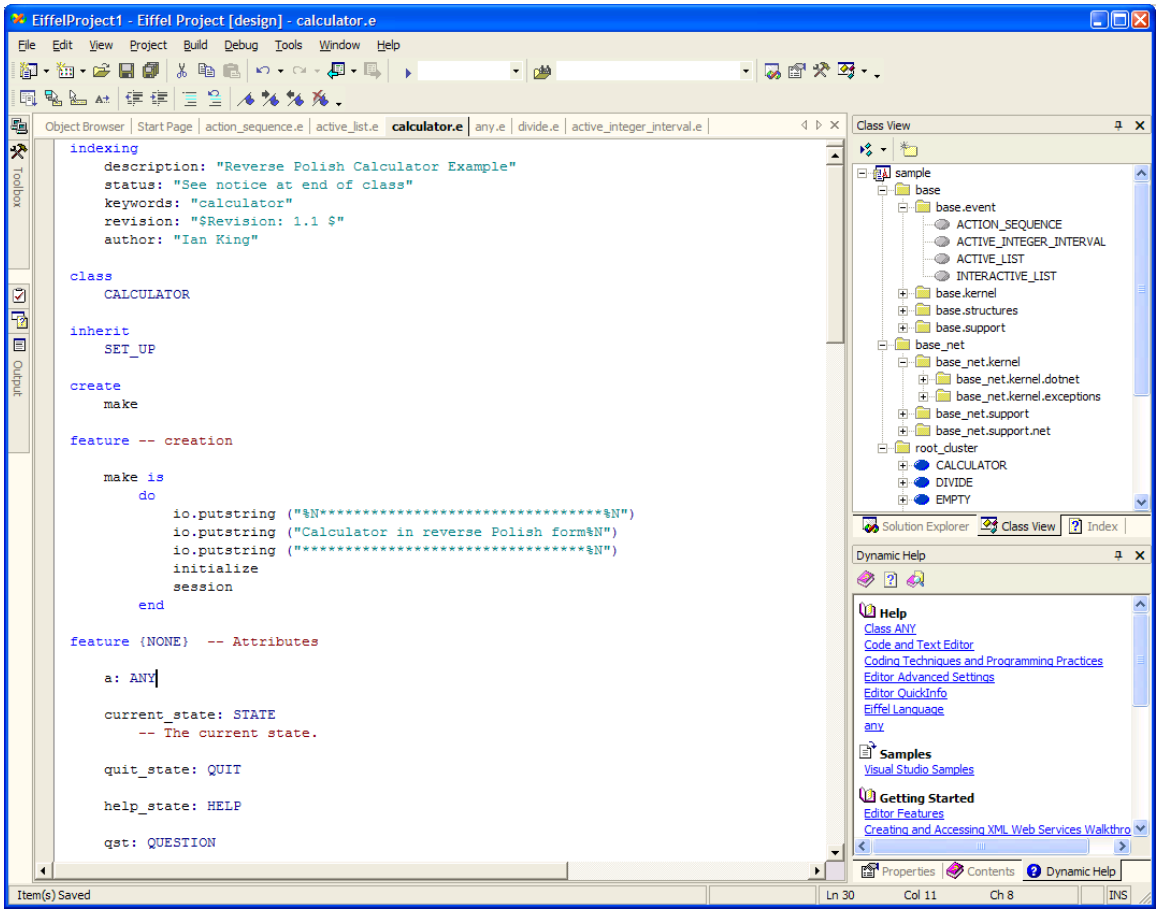
Under Visual Studio .NET, you may include an Eiffel solution as part of any project. The project may include elements in Eiffel and elements in other languages, as in this Microsoft demo involving C# and Visual Basic as well as Eiffel:

Eiffel and other languages under Visual Studio .NET



Here now is Visual Studio .NET opened on an Eiffel project:

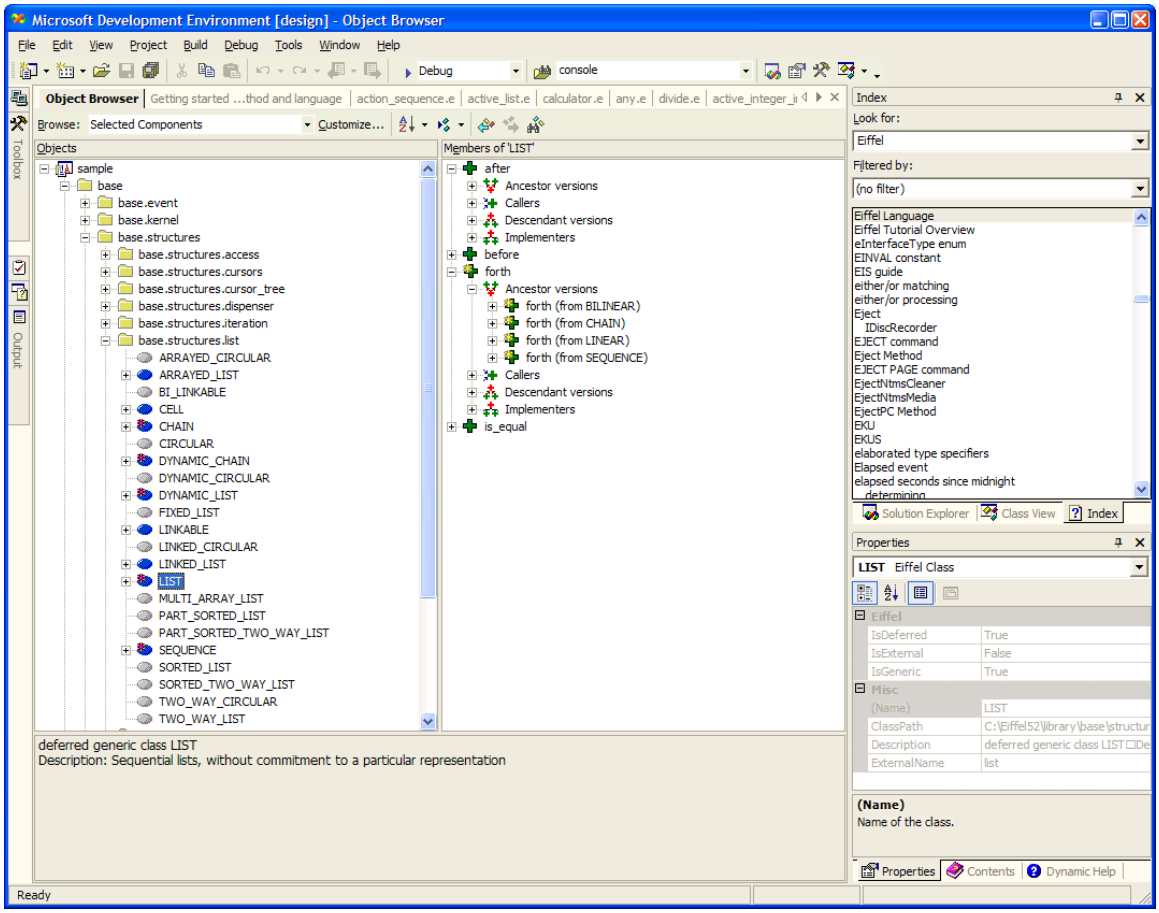
Eiffel project under Visual Studio .NET



The left pane shows a class text (CALCULATOR). The top-right pane shows the hierarchy of the project clusters; note that it uses the same graphical conventions for classes and clusters, standard for Eiffel, as in EiffelStudio. The bottom-right pane shows contextual help; it indexes the standard Eiffel documentation.

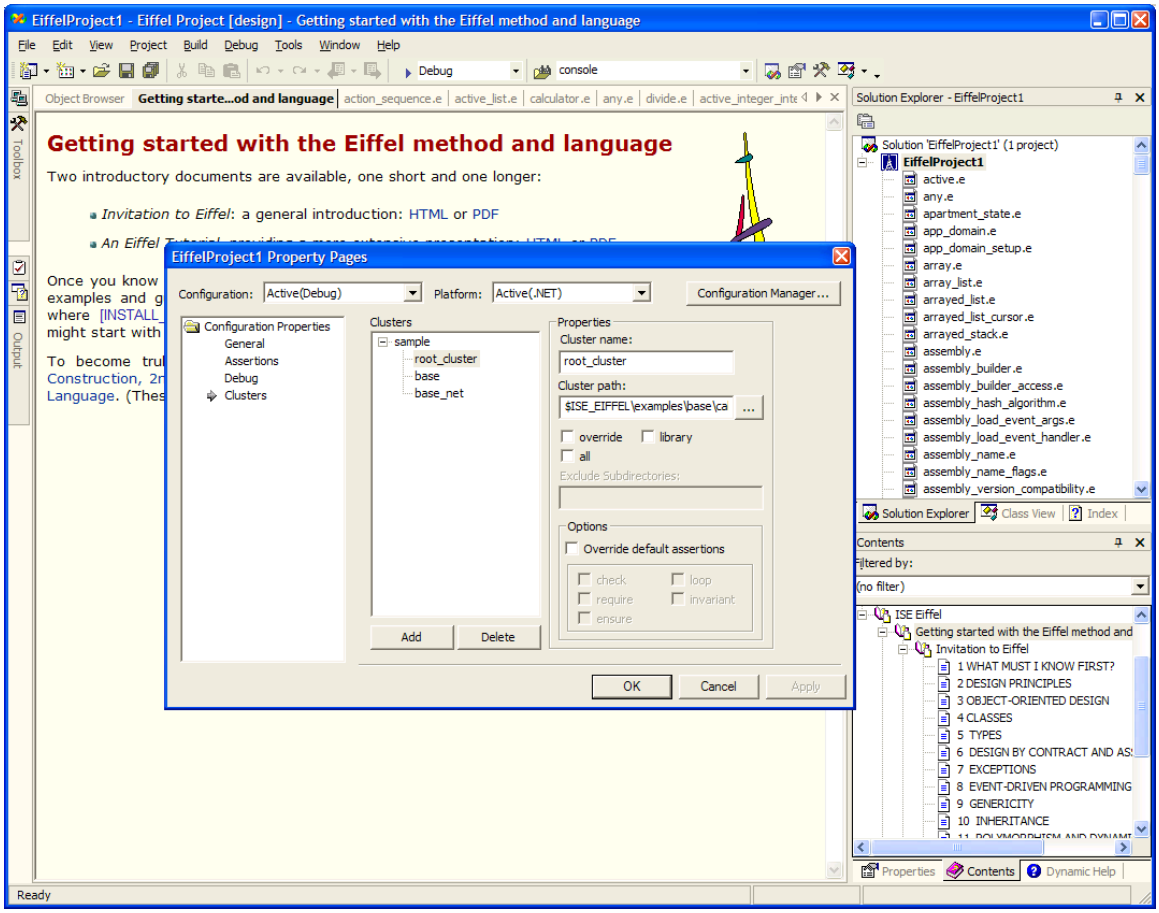
Here now is an example of using the Visual Studio .NET "object browser" (in fact a class browser) to display the inheritance structure and other inter-class relations of a project:

Object (class) browser under Visual Studio .NET



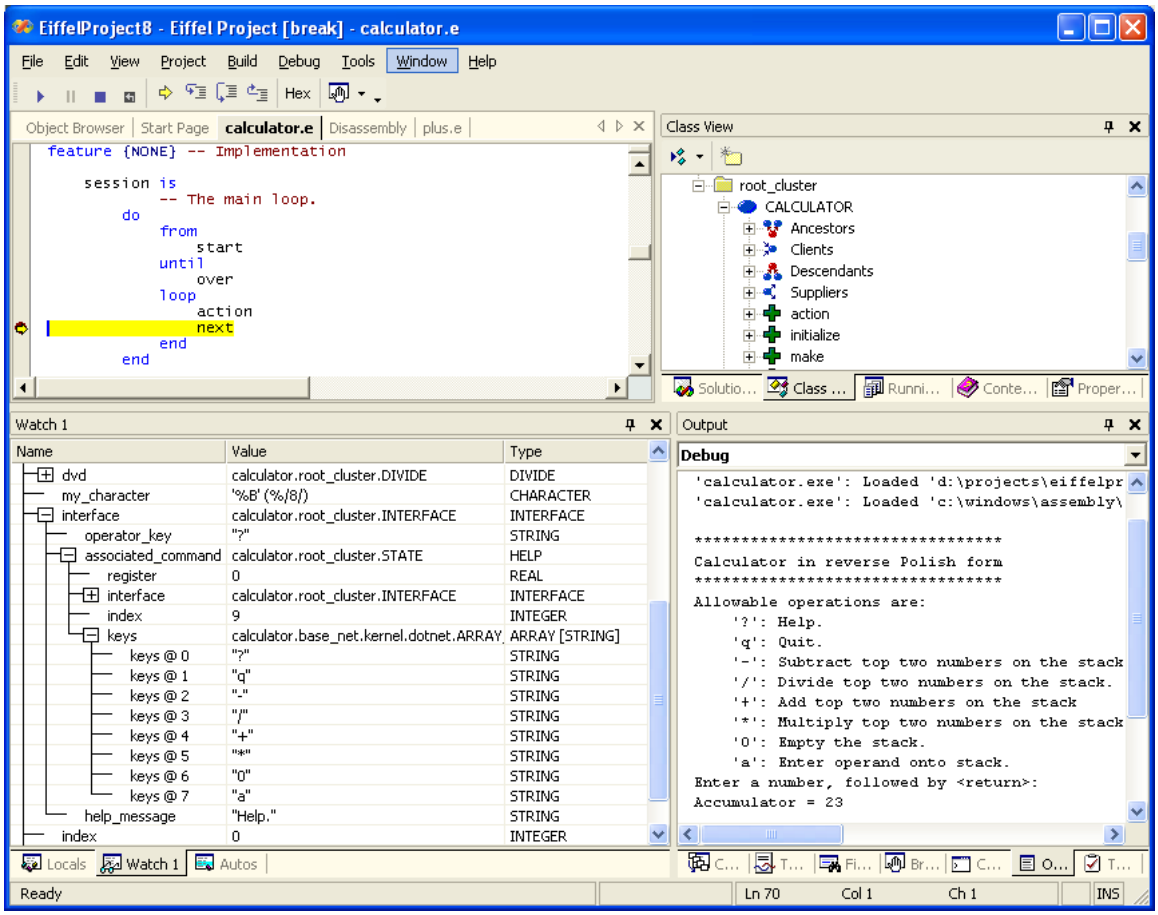
Next here is a Visual Studio .NET "property sheet" for displaying and editing the project properties, which in EiffelStudio would appear as "Project Settings" reflecting the contents of the system's "Ace file":

Property sheet under Visual Studio .NET



Our final example shows a debugging session under Visual Studio .NET for Eiffel:

Debugging an Eiffel system with the Visual Studio .NET debugger



The top-left pane shows the place in a routine text where the execution is currently stopped, and the enclosing class text, with the browsing mechanism in the top-right pane. At the bottom right is the execution's output. The bottom-left pane shows local variables, at different levels, their declared types (second column) and their values. You can use that bottom-left pane, using the Visual Studio .NET mechanisms, to define "watch lists" and to evaluate expressions on the fly.

These examples show that Visual Studio .NET users will be able to take full advantage of the Eiffel mechanisms, and that Eiffel users will for their part fully benefit from Visual Studio .NET.

5. TAKING ADVANTAGE OF .NET MECHANISMS IN EIFFEL

We have seen how you can use Eiffel to build .NET components. Since the compiler generates all the necessary metadata, other languages can reuse the Eiffel components in any way they like (heritance or client relationship). The next question is "how do I reuse existing .NET components in Eiffel?". Providing a complete and easy-to-use mechanism for this purpose is a key part of the Eiffel offering on .NET, delivering on the promise of multi-language interoperability. The "existing components" may be parts of a system written in another language; or, most importantly, they may be library components, such as the Microsoft-supplied fundamental .NET libraries that are one of the framework's principal attractions.

Strategy

For reusable components, the goal is clear: to enable Eiffel developers to combine the power of Eiffel libraries and non-Eiffel .NET libraries. The result is an unprecedented collection of reuse facilities:

On the Eiffel side, libraries such as EiffelBase (for fundamental data structures and algorithms) and EiffelVision 2 (for portable graphics) are the result of a decade and a half of continuous work and have reached a high level of quality and practicality.

On the .NET side, a rich set of advanced mechanisms is provided in particular by:

- The Base Class Library, providing basic types, collections, remoting services, threading services, security, IO access, and many other facilities.
- Windows Forms for Windows-oriented GUI building.
- Web Forms for Web User Interfaces, with types such as *DataGrid* and *HTMLImage*.
- ADO.NET for object-relational database programming.

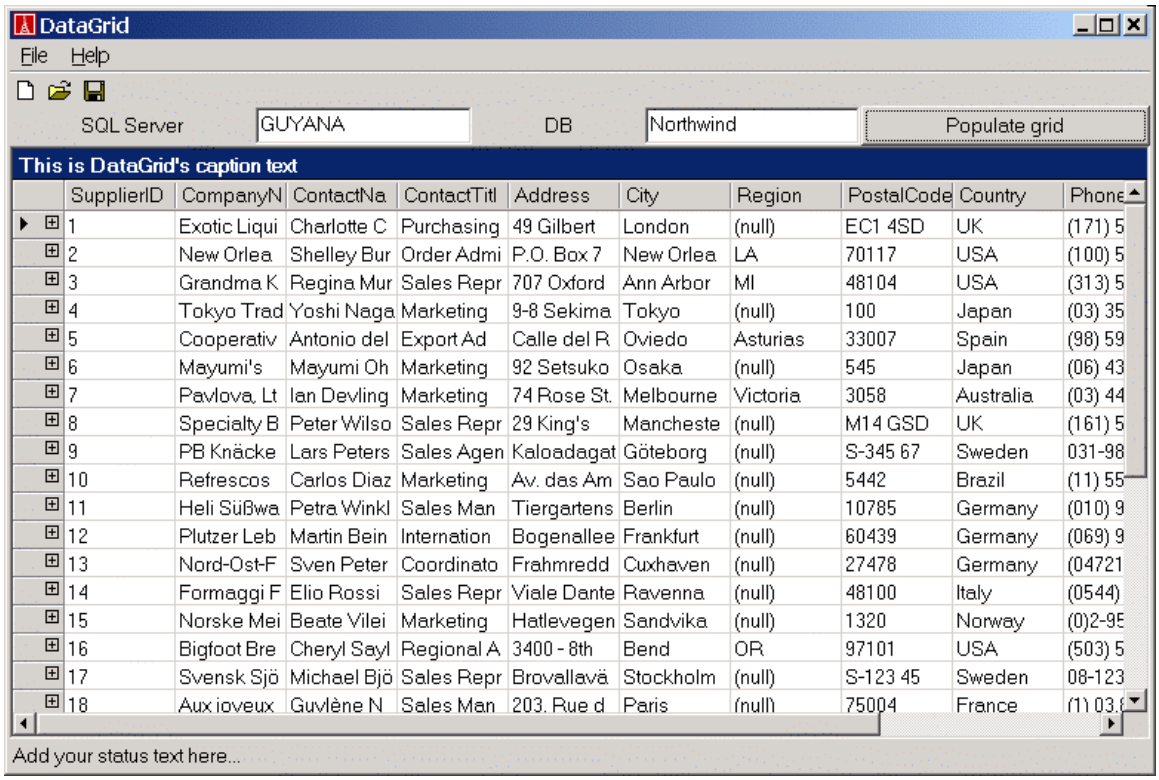
Eiffel for .NET provides access to both sides.

Eiffel libraries

Thanks for the availability of full Eiffel and to the reuse of .NET mechanisms, the basic Eiffel libraries are available to .NET developers, providing a considerable practical advantage. One of the most interesting parts of the offering is EiffelVision 2, an advanced graphical library providing an elegant GUI programming model and the ability to port an application, graphics included, to many other platforms without any change to the source code, and automatic adaptation to the native look-and-feel of the target platform.

Particularly interesting is the ability to combine Eiffel mechanisms such as EiffelVision to .NET mechanisms such as Windows Forms. For example you can embed, in a

possibly complex EiffelVision application, an advanced Windows Form control such as a Datagrid providing direct display of a database through ADO.NET. The figure below shows such a Datagrid displayed as part of an EiffelVision window.



.NET libraries

From the .NET side, Eiffel for .NET by default makes the Base Class Library, Windows Forms and Web Forms directly available to the Eiffel developer as if they were Eiffel classes. This means in particular that the tools of the environment will display the interface properties in a style consistent with Eiffel's, and that the classes can be used directly, without any need for special interface code.

This makes it possible to build powerful applications that tightly combine the benefits of these libraries with those Eiffel, as illustrated in the last figure by the combination of EiffelVision, Windows Forms and ADO.NET.

The Assembly Manager

The three .NET libraries mentioned are just examples — the most commonly needed ones — of non-Eiffel software that Eiffel for .NET makes available to any Eiffel class. The general mechanism for making *any* set of .NET classes available in this way is an ISE Eiffel tool called the Assembly Manager.

The principle of the Assembly Manager is simple. You can call the Assembly Manager from either EiffelStudio (the Eiffel-specific environment) or Visual Studio

.NET. You select the assembly to import; typically, it will include classes originally written in a language other than Eiffel, although you don't need to know what that language is, and it could in fact be Eifffel. The Assembly Manager lets you choose the assembly from those in the .NET *Global Assembly Cache*, which holds "shared assemblies" made available to all applications on the machine; or you may use Browse to find a private assembly. Once you have made that selection, the Assembly Manager will generate a set of XML files containing all the needed information about the classes of the assembly; this is made possible by the metadata-based reflection mechanisms of .NET. For global assemblies, the result is stored into an *Eiffel Assembly Cache*, including enough information to let Eiffel classes access the assembly's classes as if they were Eiffel classes, whether from EiffelStudio (the Eiffel-specific environment) or from Visual Studio .NET.

One of the tasks of the Assembly Manager is to remove overloading. For clarity, simplicity and compatibility with object-oriented principles, Eiffel maintains a one-to-one correspondence, within a class, between feature names and features (for a rationale, see [Meyer 2001]). The .NET model, however, permits overloading a name with several meanings. The Assembly Manager removes any ambiguity by generating unique names for any overloaded feature variant. The disambiguating algorithm will be presented below.

6. PRODUCING .NET SYSTEMS FROM EIFFEL

The preceding discussion has described the goals which we set ourselves at the start of the project in 1999, and which have now been achieved by the current implementation. We will now give the reader a view of the internals, to explain how we reached these goals.

Implementation strategy

At the start of the project, ISE organized the integration around three major milestones. The first step of the integration was to obtain a first usable version while avoiding the most delicate language aspects, especially multiple inheritance. This resulted in a language subset, Eiffel#, whose implementation became available in beta form in mid-2000 and as part of the first released version of Eiffel for .NET in July of 2001 (version 5.0). Eiffel# included support for Design by Contract and genericity and was sufficient to build real applications, but of course it was not the real thing. It enabled us, however, to provide an early product, gain in-depth experience with the implementation issues, and obtain invaluable feedback from our customers.

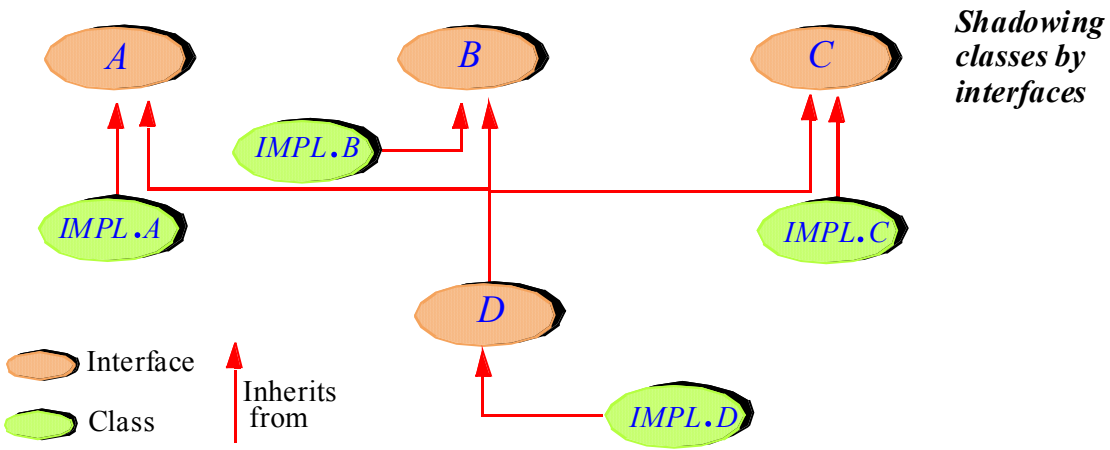
The second step was the implementation of full Eiffel, including multiple inheritance, agents and covariance. This was part of version 5.1 (December 2001).

The final step is to provide full Visual Studio .NET and ASP.NET integration. This is part of the latest release, 5.2, commercially available in May 2002.

Multiple inheritance

Multiple inheritance was, as noted, recognized from the start as a key implementation issue. The reader may indeed wonder how we can provide multiple inheritance on a platform that doesn't support it, especially with the requirement stated above that other languages should *see* the Eiffel multiple inheritance structure.

The solution used relies on the ability for a .NET class to inherit multiply from *interfaces* — completely abstract classes, without any implementation at all. In the generated code, the compiler shadows every class with interface. The following figure shows the result for the structure illustrated above; note that the counterpart of an original Eiffel class A is a .NET class called IMPL.A, whereas the "shadow" interface retains the name of the class, A, since it's what programmers in other languages will need to use:



Programmers using these classes from another .NET language, such as C# or Visual Basic.NET, do not need to know about the IMPL classes. They will declare the corresponding variables using types A, B, C, D. To create objects of the corresponding types, they will use a third set of generated classes: Create.A, Create.B and so on; this is because interfaces, such as A, cannot have constructors (creation procedures in Eiffel), so the CREATE classes provide the necessary mechanisms, one for each creation procedure of the corresponding Eiffel class. By using namespaces Impl and Create, this technique takes full advantage of .NET concepts.

Application packaging

.NET packages applications in an original way, using **assemblies** and modules rather than plain EXE or DLL files (executables or Dynamic Link Libraries). An assembly is made of a group of modules and corresponds to an application. A module may be either a DLL or an EXE. For that reason, the Eiffel compiler generates one assembly whose name is the name of the system as given in the system description file, or Ace (Assembly of Classes for Eiffel, written in a control language called Lace). You may specify whether the assembly should be an EXE or a DLL in the **msil_generation_type** default option as follows in the Ace file:

```
system
        sample
root
        ROOT_CLASS: "make"
default
        msil_generation (yes)
        msil_generation_type ("exe") -- "dll" to generate a DLL
…
```

In this example, the compiler generates a single file "sample.exe" containing both the assembly and the module

Another feature specific to .NET is the notion of namespace. Any .NET type is associated with a namespace that ensures the uniqueness of the type name in the system. You can define a default namespace for all the classes of the Eiffel system by using the following default ACE option:

```
system
        sample
root
        ROOT_CLASS: "make"
Default
        msil_generation (yes)
        msil_generation_type ("exe") -- "dll" to generate a DLL
        namespace ("MyApp")
…
```

In this example, all the classes of the Eiffel system will be generated in the namespace "MyApp.<*cluster_name*>" where <*cluster_name*> is the name of the cluster that contains the class. You may override the default namespace for each cluster as follows:

```
system
        sample
root
        ROOT_CLASS: "make"
default
        msil_generation (yes)
        msil_generation_type ("exe") -- "dll" to generate a DLL
        namespace ("MyApp")
cluster
        root_cluster: "c:\my_app"
                default
                        namespace ("Root")
                end
…
```

With this ACE file, all the classes part of the cluster *root_cluster* will be generated in the namespace "Root". Note that the name specified in the cluster clause is not appended to the namespace defined in the default clause.

Disambiguating overloaded names

We have noted above that feature names from non-Eiffel classes may require disambiguating if they have been overloaded.

The disambiguating algorithm is the following, which we would welcome other implementers adopting for no-overloading languages (so as to ensure commonality in the spirit of .NET and the Common Language Specification).

Let f1, f2, ..., fn be overloaded .NET functions with the same name (n >= 2)

For 1 <= i <= n, let Si be the signature of fi:

Si = [Ti1, Ti2, ..., Tim]
(im >= 0)

All the Si are different by the rules of overloading.

We say that a position u is unique for a function fi (for 1 <= u <= im) if there is at least one other function fj (1 <= j <= n, j /= i) such that u <= jm and Tju /= Tiu.

We determine a unique name Ni for fi as follows. Ni is of the form N_Ti1_Ti2..._Tiu (0 <= u <= im) where [Ti1, Ti2, ... ,Tiu] is the smallest initial subsequence of Si different from the corresponding subsequence for all other functions. By the rules of overloading such a subsequence exists and uniquely characterizes Si.

Informally, the algorithm appends as many signature type names as needed after the name of the function to obtain a unique name. So for example the following C# function:

public static *void WriteLine* (*String format*, *Object arg0*);
public static *void WriteLine* (*int value*);
public static *void WriteLine* (*String value*);

is translated into Eiffel as follows:

WriteLine_String_ Object (*format*: *STRING*; *arg0*: *SYSTEM_OBJECT*)
WriteLine_ Int32 (*value*: *INTEGER*)
WriteLine_ String (*value*: *STRING*)

By default the type names used by the algorithm do not include the namespaces. In the rare case of conflicts between type names in different name spaces, digits are appended to remove any remaining ambiguity.

Note that this algorithm only applies to features that are overloaded. Non-overloaded names will remain intact, except for optional adaptation to different letter case conventions. (Eiffel style rules prescribe clearly separating successive words in a feature name by underscores, as in write_line; this differs from the "camelCase" convention usually applied by C# and other .NET languages. Users can choose to retain the original names or convert them to the familiar Eiffel convention.)

Basic types

The Assembly Manager must also take care of mapping CLS types into their Eiffel equivalents, to guarantee the correct semantics. The following table shows the correspondence:

CLS Primitive Type (Description)	Eiffel Type
System.Char (2-byte unsigned integer)	CHARACTER
System.Byte (1-byte unsigned integer)	UNSIGNED_INTEGER_8
System.Int16 (2-byte signed integer)	INTEGER_16
System.Int32 (4-byte signed integer)	INTEGER
System.Int64 (8-byte signed integer)	INTEGER_64
System.Single (4-byte floating point number)	REAL
System.Double (8-byte floating point number)	DOUBLE
System.String (string, zero or more chars, null allowed)	STRING
System.Object (root of all class inheritance hierarchies)	SYSTEM_OBJECT
System.Boolean (true or false)	BOOLEAN

External Classes

Eiffel has for a long time provided extensive syntax for accessing mechanisms from other languages, in particular C, C++ and Java. Not only can you call routines written in those languages; you can also let them call back into an Eiffel system, through the *Cecil* library; and you can specify that an Eiffel routine is mapped into a C macro, that you want to use a certain C++ constructor or destructor for a particular class, that a C or C++ routine has a particular type signature in its language of origin, that a pair of getter-setter routines directly manipulate a certain field of a C *struct*, and so on. This has enabled the use of Eiffel as a **component**

combinatory, a tool widely open on the outside world and letting system developers take advantage of Eiffel's architectural mechanisms — classes, single and multiple inheritance, genericity, Design by Contract, information hiding, uniform access — to package components which may come from different languages.

A few extensions have been made to this "external" mechanism to account for the specific facilities of .NET:

.NET Function Kind	Eiffel External
Method	*"IL signature … use class_name"*
Static Method	*"IL signature … static use class_name"*
Field Getter	*"IL signature … field use class_name"*
Static Field Getter	*"IL signature … static_field use class_name"*
Field Setter	*"IL signature … set_field use class_name"*
Static Field Setter	*"IL signature … set_static_field use class_name"*
Constructor	*"IL signature … creator use class_name"*

The external features can be called from clients or descendants of the class the same way you would call any other Eiffel feature. So if your system includes a feature that needs user input, it can include the following code:

```
need_user_input is
        -- Take user input and do stuff.
    local
        io: SYSTEM_CONSOLE
        input: STRING
    do
        create io.make
        input := io.ReadLine -- calls System.Console.ReadLine()
        -- do stuff
    end
```

In this case, *ReadLine* is a static external, so you do not need to instantiate *io*. Instead you can use the syntax **feature** {*CLASS*}.*static_routine*, applicable only to external classes:

```
need_user_input is
        -- Take user input and do stuff.
    local
        io: SYSTEM_CONSOLE
        input: STRING
    do
        -- removed creation of io
        input := {SYSTEM.CONSOLE}.ReadLine
        -- do stuff
    end
```

The .NET Contract Wizard

As part of the Eiffel.NET development, we produced a new tool, the .NET Contract Wizard, which through the metadata mechanism enables users, interactively, to add Eiffel-like contracts to .NET components coming from arbitrary languages. This tool, which will be described in detail in a separate paper, makes it possible to apply some of the benefits of Design by Contract in languages other than Eiffel. This important extension was made possible by the metadata facilities of .NET.

7. CONCLUSION

Eiffel for .NET provides the combination of the two most exciting software technologies to have appeared in a decade. The combined power of the platform and the development environment should yield the dream environment for building the powerful Internet applications that society expects from us today.

The closeness of the integration enables developers to use the most advanced features of both technologies. The flexibility of the toolset — supporting both EiffelStudio on .NET, for developers coming from an Eiffel background, and Eiffel for Visual Studio .NET, for close integration with other .NET languages and use of common tools for editing, compiling, browsing, debugging — enables each company to adopt the development model that best fits its needs and its culture.

Eiffel on .NET provides flexibility, productivity, and high reliability. It is impossible in particular to overestimate the benefits of Design by Contract in a distributed environment, where looking for bugs after the fact can be an excruciating and money-wasting experience.

The level of reuse provided by the combination of Eiffel and .NET libraries provides an immediate and exceptional competitive advantage, letting companies leverage off quality reusable solutions resulting from thousands of person-years of quality-focused effort, and concentrate on their own added value to bring products to market quickly and successfully.

Together with the other benefits of the Eiffel method — seamless development, generic programming, information hiding and other software engineering principles, a powerful inheritance mechanism — Eiffel on .NET provides a best-of-breed solution for ambitious Internet software developers.

References

[ISE] Eiffel Web sites at http://www.eiffel.com/ and http://www.dotnetexperts.com.

[Meyer 1992] Bertrand Meyer: *Eiffel: The Language*, Prentice Hall, 1992.

[Meyer 1997] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[Meyer 2001] Bertrand Meyer: *Overloading vs. Object Technology*, in Journal of Object-Oriented Programming (JOOP), November 2001, also online at se.inf.ethz.ch/publications/joop/overloading.pdf.

About the authors

Raphael Simon is a senior engineer at Interactive Software Engineering (Santa Barbara, California), in charge of the Windows applications and tools division. **Emmanuel Stapf** is a senior engineer at ISE, head of the compiler and environment division. **Bertrand Meyer** is a professor of software engineering at the Swiss Federal Institute of Technology (ETH) in Zürich and scientific advisor to ISE, as well as an adjunct professor at Monash University in Melbourne (Australia).

To contact the authors, use info@eiffel.com.