

Frequently Asked Questions on the Eiffel Language

Classic Eiffel

[What is Eiffel?](#)

[Where does the name come from](#)

[How did Eiffel come about, and what is its history?](#)

[What is EiffelStudio?](#)

[Ok, so what is the difference between Eiffel and EiffelStudio?](#)

[Is Eiffel intended for any Specific Application area?](#)

[What about operating systems then? Where does Eiffel run?](#)

[Won't I have to forsake my existing software thus losing millions of dollars?](#)

[How fast is Eiffel's run-time performance?](#)

[What is Melting Ice Technology?](#)

[Is it true that Eiffel Compiles into C?](#)

[What about graphics?](#)

[What about relational databases?](#)

[What does the IDE look like?](#)

[What does an Eiffel Program look like?](#)

[What does a class look like?](#)

[A counter is great but how do I write a real system?](#)

[What about reusability?](#)

[Assertions look great, but how do they help me?](#)

[What about repeated inheritance?](#)

[Tell me about typing and binding](#)

[What is BON?](#)

[What is this Design by Contract mechanism, and how will it change my life?](#)

[What can Eiffel bring to me that other languages such as C++ cannot?](#)

[Why is Eiffel easier to use?](#)

[Why is multiple inheritance so clean in Eiffel?](#)

[What is multiple inheritance?](#)

[What is concurrent engineering?](#)

[What is genericity?](#)

Eiffel for .NET

[Are there performance issues with Eiffel for .NET?](#)

[Is Eiffel transparent in .NET?](#)

[How does Eiffel for .NET produce reusable components?](#)

[I have a bunch of classes in VB.NET and C# do I need to rewrite them in Eiffel?](#)

[What UML tools do we support?](#)

[How do you create a web service in Eiffel?](#)

[What about ASP.NET?](#)

Classic Eiffel Questions

What is Eiffel?

Actually, in this case it's short for "The Eiffel Development Framework™" - a comprehensive approach to software development. It consists of the Eiffel methodology for the beginning-to-end construction of robust, reusable software; the Eiffel language, which seamlessly supports and enforces the methodology; and EiffelStudio™, the environment that contains the Eiffel language and several productivity and quality related tools. The individual parts all fit together with and depend on each other, supporting each other's function in the pursuit of making the best software possible.

No other software development system has been designed to work in such a simple and powerful way.

The results of this approach and design are staggering. Productivity increases of 2 to over 10x. Cost of ownership 80% lower. Quality levels 10 times higher.

Eiffel takes companies' software to a level of efficiency and reliability far above the capabilities of other languages and development tools.

Where does the name come from?

Eiffel is named after the engineer Gustave Eiffel, an engineer who created the famous tower. The Eiffel Tower, built in 1887 for the 1889 World Fair, was completed on time and within budget, as will software projects written in Eiffel. If you look at that wonderful structure, you will see a small number of robust design patterns, combined and varied repeatedly to yield an extremely powerful, efficient structure - exactly like an Eiffel system built out of Eiffel Software's reusable libraries. Like many software systems today, the Eiffel Tower was initially conceived as a temporary structure; and like many a system built with Eiffel, it was able to endure far beyond its original goals.

How did Eiffel come about, and what is its history?

Eiffel was designed at Eiffel Software (then known as ISE) in 1985, initially as an internal tool to develop some of our products. We wanted a modern, object-oriented environment integrating the concepts of modern software engineering, and there was simply nothing available. The Eiffel 1 environment was first demonstrated in public at the first OOPSLA conference in October of 1986 where it attracted considerable attention, leading us to release it as a commercial product at the end of 1986. The technology spread rapidly over the following years, leading to a set of successful industrial projects in the US, Canada, Europe and the Far East. Right from the beginning Eiffel also impressed the academic community as an ideal way to teach software concepts at all levels, leading to its adoption by numerous universities around the world as the primary teaching language.

Successive versions of the environment appeared at the rate of about once a year. Eiffel recognition was given a large boost by the appearance in 1988 of the book *Object-Oriented Software Construction* by Bertrand Meyer, which quickly became the best-selling title in the field and was translated into eight foreign languages; the book used Eiffel as the natural vehicle to introduce concepts of object technology and Design by Contract. (The greatly expanded *Object-Oriented Software Construction, 2nd Edition* is now available; you can find it in bookstores or order a copy from the sources listed on our [products](#) page.)

The last iteration of the original technology was version 2.3, released in the Summer of 1990. The next version, Eiffel 3, resulted from the lessons of the initial version and was written entirely in Eiffel; it was bootstrapped from 2.3. Eiffel 3 introduced the Melting Ice Technology for fast recompilation, a fully graphical environment based on innovative user interface concepts, and considerable advances in libraries (graphics, networking...) and optimization of the generated code. The initial versions were available on Unix; since then they have been complemented by fully compatible releases on VMS, OS/2, Linux, and our best-selling Windows versions (Windows 3.1, Windows 95, Windows NT), making Eiffel one of the most widely portable solutions in the

software industry.

Today, the Eiffel technology continues to push the frontiers of software development technology forward. With the introduction of [EiffelStudio 5.2™](#), users have available to them the most efficient means for achieving highest-quality, robust, scalable, reusable software applications - on all major platforms, including Microsoft's new .NET framework™. And with [Eiffel ENViSioN!™](#), developers can use the power of the Eiffel language from within the popular Visual Studio .NET environment from Microsoft.

What is EiffelStudio?

EiffelStudio is the Integrated Development Environment (IDE) designed exclusively for the Eiffel Object-Oriented language. Seamlessly addressing the whole life cycle of software development, Eiffel Studio provides facilities that will help your application develop from initial design time, right through to time of deployment. Built in functionality such as Computer Aided Software Engineering (CASE) tool for the Business Object Notation (BON) method allow you to see and interact with the design of your system during development, and there is no need for reverse engineering as everything is done concurrently. Eiffel Studio also has superb browsing mechanisms for viewing information about your code and how that code performs and behaves whilst executing. From full feature browsing, to built in metrics and profiling, EiffelStudio can give you information on almost anything about your system that can aid you in developing and optimizing it to its full potential.

Centered around the Design by Contract™ methodology, EiffelStudio's fully featured debugger will allow your software to find the bugs for you, minimizing the huge cost of maintenance that systems designed with other languages have to endure. Coupled with a fully functional browsable editor, EiffelStudio allows you to navigate to any part of your system to track down and fix the so called 'hard to find' bugs, therefore reducing project costs even further.

What is the difference between Eiffel and EiffelStudio?

Eiffel is the language a developer uses to write great software. EiffelStudio is the environment and toolkit that surround the Eiffel language that he/she uses for creating large, sustainable, business-critical applications.

Is Eiffel intended for any specific application area?

Not specifically. Eiffel brings many benefits to serious application development, regardless of end-use. Eiffel has been used in many application areas, from financial applications to manufacturing to product configuration control to healthcare to telecommunication systems. It is also widely used for teaching purposes.

Eiffel shines particularly for ambitious systems that must be easy to adapt to changing market or user demands. With Eiffel you can quickly produce a basic version of a system, reliable and efficient, put it into users' hands early (while the competition is still trying to produce a "prototype"), and come up with new releases rapidly, all the time maintaining the standards of robustness that are the hallmark of the approach.

Eiffel scales up. Many a 500,000-line system started as a 50,000-line program. Through its abstraction and structuring facilities, Eiffel is one of the few environments that won't let you down when your project (or company) grows in size, scope and ambition.

What about operating systems then? Where does Eiffel run?

Eiffel is very portable, a feature that developers love. It runs just about everywhere, including Windows (classic and .NET), Unix, Linux, VMS and soon Mac OSX. This allows developers the flexibility to maintain their legacy code while developing new code on a completely different operating system.

Won't I have to forsake my existing software, thus losing millions of dollars?

Absolutely Not!

Eiffel is an open system; it is at its best when used as a combination technology to reuse software components written in various languages. In particular, Eiffel includes a sophisticated C and C++ interface, supporting:

- Calling C functions from Eiffel.
- Accessing C++ classes and all their components (functions or "methods", data members, constructors, destructors etc.) from Eiffel.
- Accessing Eiffel mechanisms from C or C++ through the Cecil library (C-Eiffel Call-In Library).
- Automatically producing a "wrapper" Eiffel class from a C++ class.

Eiffel makes it possible to move to modern software technology while reusing the best results of earlier practices.

How fast is Eiffel's run-time performance?

FAST. Eiffel Software has shown that it is possible to utilize the full power of modern object technology without sacrificing run-time performance. Various benchmarks show run-time efficiency similar to C and Fortran, and in many cases better.

What is Melting Ice Technology?

This describes Eiffel Software's unique incremental compilation technology, which combines compilation (for the generation of optimally efficient code) with bytecode interpretation (for fast turnaround after a change). The bulk of your software, including precompiled libraries, is "frozen", i.e. compiled; what you change gets "melted", i.e. the compiler will quickly generate some interpretable "bytecode" and stop there, making sure that the frozen part calls the melted part (and conversely) when appropriate.

Is it true that Eiffel Compiles into C?

Actually, the Eiffel compiler generates, as just noted, an internal form known as the "bytecode". The bytecode, as was also noted, can be interpreted directly. But it can also be translated into other forms.

To generate the final version of a system, the bytecode is optimized and translated into C, to take advantage of the presence of C compilers on just about every platform under the sun. This is the process known as "finalization", which performs extensive optimizations (routine inlining, static calls, array optimization), permitting the performance achievements mentioned above.

Using C as an intermediate language takes advantage of the platform-specific optimizations performed by C compilers and, most importantly, facilitates interoperability of Eiffel software and software written in C and C++.

What about graphics?

Eiffel offers a choice of graphical libraries for a variety of application scenarios.

For portable developments, use EiffelVision and EiffelVision2, which are high-level graphical libraries covering user interface objects (windows, dialogs, menus, buttons, dialog boxes etc.) as well as geometrical figures (polygons, circles and the like), which will run on all the supported platforms, adapting in each case to the native look-and-feel.

For platform-specific developments, to take advantage of the full set of "controls" or "widgets" available on a particular window system, use the platform-specific libraries:

- On Windows, WEL (the Windows Eiffel Library) gives you access to essentially all the Windows graphical API, including the most recent controls. A separate page describes how WEL combines the advantages of Windows and Eiffel. An extensive WEL tutorial is available on-line, as well as a general presentation of the design of WEL.
- On Unix, Linux and VMS we provide GEL, the GTK Eiffel Library.

Using EiffelVision and EiffelVision2 or one of the platform-specific libraries is not an exclusive proposition: you can mix-and-match the two levels, using EiffelVision for its abstract capabilities and, for example, WEL to take advantage of specific Windows controls. In fact, EiffelVision internally relies, for its implementation on each platform, on the corresponding platform-specific library, so you already have WEL if you are using EiffelVision on Windows.

What about relational databases?

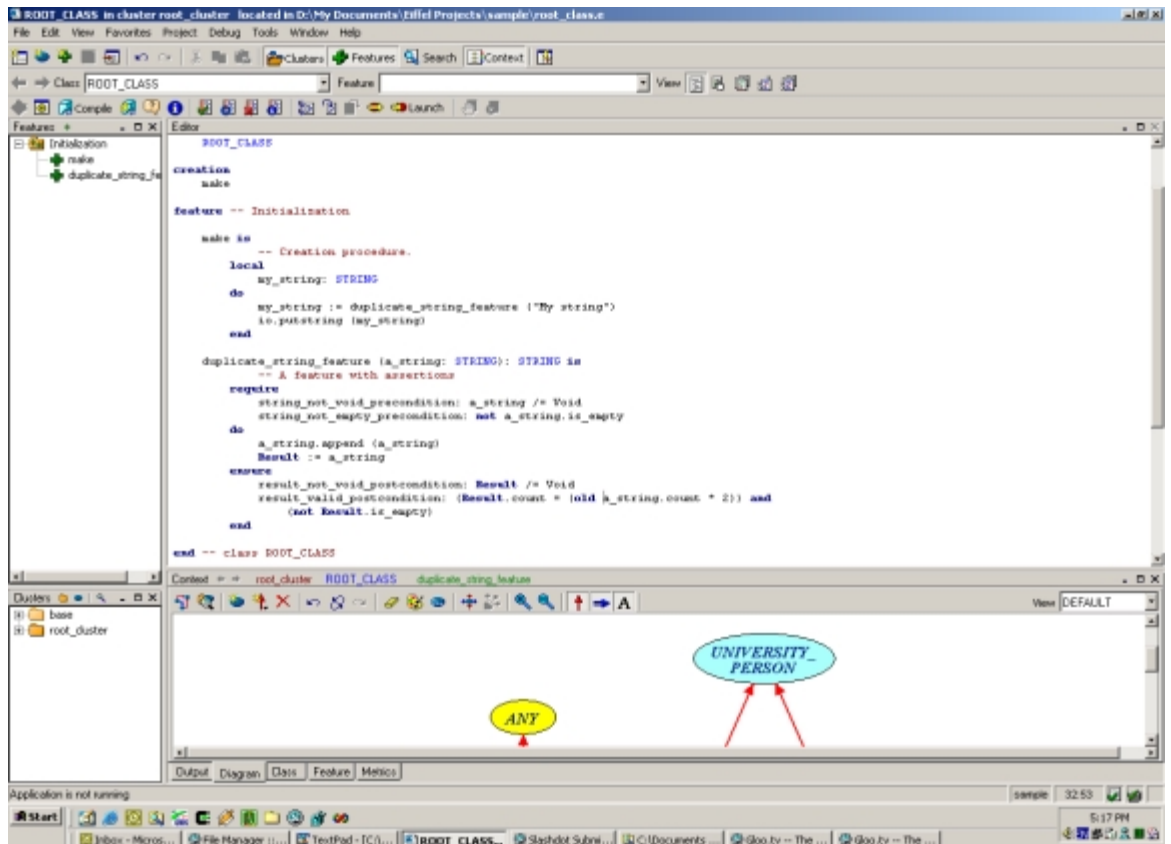
Eiffel Software provides the EiffelStore library for object-relational interfaces, with mappings currently available for ODBC (giving access to many dozens of database systems on Windows), Oracle, Sybase and Ingres.

Eiffel has also been interfaced with such object-oriented databases as Matisse (interface available from Eiffel Software), Versant and O2.

What does the IDE look like?

EiffelStudio is a fully graphical environment that includes all you would expect and more: analysis and design workbench with round-trip (or reversible) engineering, fast recompilation, editing, sophisticated browsing facilities, automatic documentation (see for example the descriptions of the "short form" below), an advanced debugging mechanism etc.

Just as an illustration, without further explanations (which you will find by browsing around our pages), here is a screenshot from EiffelStudio:



What does an Eiffel Program look like?

Actually we like to talk about "systems". A system is a set of classes, each covering a "data abstraction", that is to say a certain set of objects, from the external world or from the implementation. For example you may have classes AIRPORT, RADAR and RUNWAY in a flight control system, classes CUSTOMER and ACCOUNT in a banking system. In any system you can also have general-purpose classes such as LINKED_LIST and HASH_TABLE, although you would not normally write them but reuse them from a library such as EiffelBase.

What does a class look like?

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
end
```

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

```
my_counter.increment
...
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. If you understand this example, you already know a fair deal of Eiffel! Note how simple the syntax is. Semicolons between instructions are optional (they have been omitted above); no strange symbols, no complicated rules.

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 "information hiding". 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. Such information can be used by tools to help developers search for reusable classes. It is good practice to include at least a description entry providing an informal description of the purpose of the class.

A counter is great but how do I write a real system?

The principles scale up. A class can represent a counter, but it can also represent an assembly line or a factory.

The Eiffel mechanisms for abstraction, reliability and simplicity provide a power of expression unmatched in the software world.

What about reusability?

As Roland Racko wrote in Software Development:

"Everything about Eiffel is single-mindedly, unambiguously, gloriously focused on reusability -- right down to the choice of reserved words and punctuation and right up to the compile time environment".

We couldn't have said it better. Eiffel was designed from day one to be the vehicle for the new software industry, based on the reuse of high-quality components -- rather than on everyone reinventing the wheel all the time.

This saves organizations countless hours and thousands of dollars in wasted development time.

Eiffel has put these ideas into practice by providing a rich set of professional reusable libraries (several thousand carefully crafted classes): EiffelBase, EiffelVision, EiffelNet, EiffelWeb, EiffelParse, EiffelLex, WEL, MEL, PEL etc.

Assertions look great, but how do they help me?

Assertions radically improve the nature of software development. They have three major benefits:

- As a design aid. By working with Design by Contract, you build software together with the arguments that justify its correctness. This makes it much more realistic to produce bug-free software.
- As a testing and debugging mechanism. Using the Eiffel compiler, you select which assertions will be monitored at run time; you can set different levels (no check, preconditions only, preconditions and postconditions, everything) separately for each class. Then if an assertion is found at run time to be violated -- meaning a bug remains in your software -- an exception will interrupt execution. This is a tremendous help for getting software right quickly: testing and debugging are no longer blind searches; they are helped by a precise description both of what the software does (the actual executable texts, given by the do clauses) and of what it should do (the assertions).
- In automatic documentation. To give the users of a class a precise description of what a class provides, without giving out implementation details, you use the Eiffel notion of the short form of a class, which keeps the feature headers and comments as well as their assertions, but discards any implementation stuff. For example:

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

class interface
    COUNTER

feature -- Access
    item: INTEGER
        -- Counter's value.

feature -- Element change
    increment is
        -- Increase counter by one.
    ensure
        count_increased: item = old item + 1

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

```

reset is
  -- Reset counter to zero.
  ensure
    counter_is_zero: item = 0

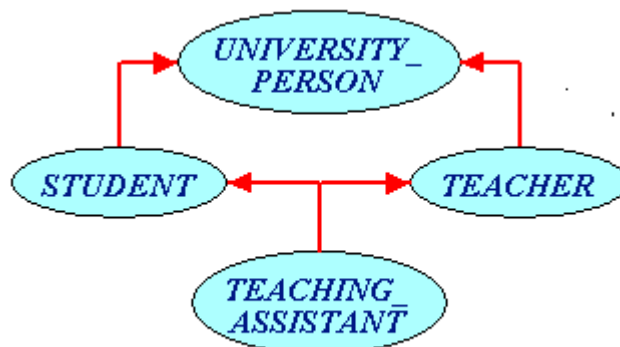
invariant
  positive_count: item >= 0
end

```

In the EiffelStudio environment, you get the short form at the click of a button. The short form is an ideal form of documentation: abstract yet precise; and best of all, produced by the environment, not written separately, so you don't need to do any work to get it! No more need to write "interface modules" and then repeat their information in the implementation part.

What about repeated inheritance?

Repeated inheritance is the case, (illustrated by the figure below), in which a class inherits from another through two or more paths. Only in Eiffel can you decide separately for each feature of the common ancestor, such as `birth_date` or `library_privileges` in the example, whether it gives one feature or two in the repeated descendant. In the figure, a teaching assistant has the same birth date whether viewed as an instructor or as a student (one feature), but has different library privileges under each of these capacities (two features). This flexibility is indispensable to specify what you want to share and what you want to replicate.



Tell me about typing and binding.

Eiffel is statically typed to ensure that errors are caught at compile time, not run time. For example if your system may mistakenly try to execute a request to compute the diagonal of a graphical object that happens to be a triangle, the Eiffel compiler will catch the error before it has had time to cause any damage. Most other object-oriented languages use some degree of "dynamic typing" in which such errors can escape the compiler. (Beware in particular of C extensions that are sometimes advertised as statically typed but still permit almost arbitrary type conversions.)

Eiffel is dynamically bound to guarantee that the right version of an operation will always be applied depending on the target object. For example if you apply the feature "take off" to an object representing some kind of plane, you have the guarantee that if there are several plane types, each with its own `take_off` feature, the appropriate one will always be automatically selected. (In contrast, some approaches by default use "static binding", which can result in disastrously incorrect behavior.)

What is BON?

BON is the acronym for Business Object Notation, an analysis and design method that is based on concepts close to those of Eiffel (seamlessness, reversibility, contracting) and defines simple, intuitive graphical conventions. BON is particularly notable for its ability to scale up when you

need to describe large and complex systems, keeping a view of the whole while zooming into the details of components at various levels of abstraction. The BON method is described in a widely acclaimed book by the authors of the method. Eiffel Software's EiffelCase (now the Diagram Tool within EiffelStudio) analysis and design workbench directly supports BON.

What is this Design by Contract mechanism, and how will it change my life?

Design by Contract (DBC) is a unique mechanism that demands the production of quality software. It ensures that your code will have substantially less errors because it follows 'the rules' of development. Our customers tell us that because of DBC they can dramatically lower the number of bugs and spend more time designing.

What can Eiffel bring to me that other languages such as C++ cannot?

Clean clear syntax, reusability, native Design By Contract, support for the whole software lifecycle, the ability to finish projects on schedule and enhanced productivity in all stages of development.

Why is Eiffel easier to use?

There are many reasons why Eiffel is easier to use. In addition to Design by Contract, Eiffel offers the following unique features:

- The simplest, most powerful language available
- Fully integrated with Visual Studio .NET
- Interoperates and shares code with any .NET language
- Enables use of multiple inheritance (the "Holy Grail" of O-O programming) in .NET

What is multiple inheritance?

Multiple inheritance is an inheritance mechanism whereby a software unit, known in Eiffel as a class, may inherit the features of many other classes. Most modern programming languages support single inheritance whereby a class can inherit from only one other class, but multiple inheritance offers the developer unrestricted inheritance through the ability to inherit as many as is desired. This has many advantages for the resulting software including improved reuse, better overall system design and architecture, greater flexibility, and easier maintainability and debugging.

Why is multiple inheritance so clean in Eiffel?

Eiffel tames the power of multiple inheritance through a renaming mechanism, which eliminates name clashes, and through a selection facility to remove any ambiguities resulting from multiple redeclarations. Without multiple inheritance you would lose much of the reusability benefits of the object-oriented method. For example, not all comparable elements are numeric (think of strings) and not all numeric elements are comparable (think of matrices). Without multiple inheritance you would not be able to select one of these properties when you need to, and both when you need to.

What is concurrent engineering?

Concurrent engineering allows the use of EiffelStudio to automatically generate BON diagrams to see and interact with the overall design of your system as you engineer it.

What is genericity?

Genericity is the support for type-parameterized class modules in the software text. In object-oriented circles these are known as generic classes. Such classes use generic parameters in the software text, which are then substituted for formal parameters when the class is actually used by a client class. The benefits of such a method is most fully realized in classes when you consider container objects such as arrays and lists. These types are designed to hold a number of arbitrary elements. Ideally these contained elements should be able to be any type of element, from books to customers to any type available in your system. Otherwise you would have to write a separate class definition for every type of element you wish to store, a painful and time consuming situation. Genericity solves this problem by assuming a generic type, which will be substituted by the actual type at runtime.

EIFFEL for .NET Questions

Are there performance issues with Eiffel for .NET?

In the end, no. Support for multiple inheritance and the inclusion of the base library classes currently does result in generated *assemblies* which are larger than those generated by other .NET languages; however, there are no significant performance differences at run-time.

Is Eiffel transparent in .NET?

Yes. Eiffel for .NET can be used in exactly the same way as classic Eiffel and no extra language constructs are required to produce applications. Since there are some elements of the .NET language which have no equivalents in Eiffel, such as the ability to define custom attributes, these have been extended to Eiffel for .NET so that the full features of .NET can be used.

Eiffel brings quite a few unique and powerful features to .NET. Most noticeably it is the only .NET language to offer multiple inheritance and genericity. These two mechanisms, described in more detail elsewhere in this FAQ, are an indispensable aid to creating fully reusable software in a truly object-oriented way.

Eiffel for .NET also brings the full benefits of Design By Contract to the software created, ensuring correctness of the software text, reliability and robustness. Whilst some other .NET languages do support contract mechanisms, Eiffel for .NET is the only one to support them natively, as an actual language construct. This is in accordance with classic Eiffel which considers contract use a vital and necessary element in the production of reliable, robust code. Eiffel for .NET also includes a Contract Wizard which enables existing .NET assemblies written in a non-contract language such as C# or VB.NET to be given contracts after they were written. This unique feature allows for assemblies to be made safer if necessary even if they were not given contracts at design time.

Eiffel also brings some existing libraries to .NET, so aside from being able to use other .NET libraries such as Windows.Forms for graphical elements you could instead use Eiffel's WEL or EiffelVision2 libraries. EiffelVision2 is a particularly unique graphical library in that it is multi-platform; thereby ensuring the compiled system will produce the same display and behavior on all supported platforms.

How does Eiffel for .NET produce reusable components?

Eiffel for .NET produces reusable components simply by virtue of the fact that it compiles to .NET Intermediary Language (IL) code which makes it available for use by any other .NET supported language. However, this method of reuse is only one type of reuse in action, namely the reuse of software components. There are many other levels and areas of potential reusability. The Eiffel language was developed with such levels of reuse in mind from the very beginning and these can benefit the developer in many ways. Multiple inheritance and genericity provide reuse within the actual software implementation, eliminating the need to repeat the same code in different

classes - if you need the functionality provided by a class then become a client or inherit from it. Likewise if much of your code does similar things with only small variations you can abstract it into a class or library. This practice is employed and can be seen in the reusable libraries provided with EiffelStudio or Eiffel ENViSion!

I have several classes in VB.NET and C#. Do I need to rewrite them in Eiffel?

No. Any code written in a .NET supported language such as VB.NET, C# or Eiffel for .NET automatically compiles to .NET intermediary language (IL) code. By virtue of this standard, independent modules are created (.dll's or .exe's) which are fully interoperable. This means a developer can reuse any code without having to rewrite or port it to another language.

What UML tools do we support?

EiffelStudio is capable of generating XMI (XML Metadata Interchange) information for any Eiffel system. The XMI notation makes it possible to exchange system information between any products which support this standard (e.g., Rational Rose). This can then be imported into such a product and UML information generated.

How do you create a web service in Eiffel?

Creating a web service using Eiffel for .NET is as simple as creating one in any other .NET supported language. The class which contains the methods you wish to expose on the Internet must inherit from the .NET type `System.Web.Services.WebService` so the `System.Web.Services` namespace must be included in your compiled Eiffel for .NET system. To create a web service, just inherit the generated class which corresponds to the `System.Web.Services.WebService` type.

To expose methods on the Internet they must then be given custom attributes declaring them as web methods. Exactly the same procedure can be used from within EiffelStudio.

What about ASP.NET?

One of the most useful aspects of .NET is the ability to write and call .NET code from within ASP pages. This is accomplished by specifying a language attribute tag in the ASP document which indicates the language you wish to use. To use the Eiffel language from within an ASP page all that is needed is to specify Eiffel as the language.

About Eiffel Software

Eiffel Software (a division of ISE) is the world leader in Eiffel pure object-oriented programming tools. Founded in 1985, Eiffel Software produces proven professional tools and component libraries for business-critical and enterprise software developments. Eiffel Software's products enable their customers to output more and higher-quality software in less time than with any other development tools available. Its users span the globe, in industries ranging from large financial institutions, to technology manufacturing, to government and defense contractors, to health care providers and more.