OOSC2 Reviews

Why Eiffel is better than C++ (for big projects)

by Paul Johnson
Email: paul.johnson@gecm.com

Table of Contents

1. Introduction

At present the most popular OO language in the world is C++. There are more environments which support it, more people who use it, and more products written in it, than for any other OO language. For many people, C++ *is* object-oriented programming.

Until recently a project manager contemplating the use of OO technology on a large (100,000 lines plus) software project might well have choosen C++ on pragmatic grounds. The current Eiffel vendors are comparatively small companies compared to C++ environment vendors such as Hewlett-Packard, DEC and Sun. The available Eiffel environments were slow and lacked important tools such as debuggers. These were good, pragmatic reasons for choosing C++. They were also very short term reasons. C++ is a deeply flawed language. This has not changed. Eiffel lacked useable development environments. This has changed.

2. The Criteria

Large projects need the following from a programming language:

Suport for modularity.
Readability and clarity.
Ease of debugging.
Efficiency.

Eiffel has a number of features which support these requirements. These are covered in the rest of this article.

3. Assertions

A module of any sort in any programming language has two interfaces:

The syntactic interface describes the argument data types, function names and return types.
The semantic interface describes the legal values of the argument data, and the results of calling each function.

Most programming languages are concerned only with the syntactic interfaces to modules, since this is what they need to generate correct code. Eiffel includes constructs for declaring the semantic interface. These constructs are known as *assertions*. For example, a routine to calcuate square roots might look like this:

```
square_root (x: REAL): REAL is
      -- Returns the positive square root of x.
   require
      positive_argument: x >= 0.0;
   do
      -- Implementation ommitted
   ensure
      correct_root: Result * Result = x;
      positive_root: Result >= 0;
   end;
```

This states that the routine takes a real argument, greater than or equal to zero. It returns the positive square root of the argument. *Result* is a special variable used to return values from functions.

As well as the *require* and *ensure* clauses, Eiffel also provides *invariant* clauses. These apply to entire classes, and are used to state relationships that always hold. For example, a stack class with an *empty* flag and a *count* of items held might have the following invariant clause:

```
invariant
count >= 0;
empty = (count = 0);
```

This states that the count of items on the stack can never be negative, and that the *empty* flag is true if and only if the count is zero.

When an assertion fails, Eiffel raises an exception. This exception can be caught by the function or passed back to the caller. If the exception is passed

all the way back to the top level function then the program fails Some implementations invoke a debugger. Others merely print a stack trace.

Afficionados of Ada will recognise this approach to run-time checks. However Ada is largely limited to checks on language level constructs (such as array bounds), and it is not possible to include most of these checks in the package specifications.

The ideas behind assertions come from algebraic formal languages such as Z and VDM. They try to use mathematical techniques to prove that a coomputer program is correct, but this has always been an impossibly expensive way to develop software in the real world. Eiffel offers a pragmatic compromise. Software can be formally specified, and at least some of the specification can be checked at run time to help with debugging.

Describing the semantic interface along with the syntactic interface has a number of important benefits.

> It documents the legal inputs of each function, and specifies the relationship between the inputs and outputs. A simple filter program can take an Eiffel class and list the interface, with function names comments, argument types, and assertions. This can serve as the reference documentaion for the class. All current Eiffel environments include a filter to generate this documentation automatically.
> By explicitly listing the *contract* between client and server classes, the assertions help to prevent misunderstandings between programmers, and hence simplify integration.
> When new classes are derived from existing ones by inheritance there are design rules which specify the relationship between the assertions in the parent and child classes. These rules ensure that the child can correctly be substituted for the parent in all cases.
> The assertions made by the programmer can be tested at run- time. This makes programs much easier to debug. Errors are caught when they occur, rather than when the programmer notices the wrong results.

The assertion mechanism in Eiffel is closely tied to the exception mechanism. When an assertion fails an exception is raised. This can be caught by the function or passed back to the caller. If the top level function fails then a stack trace is printed. Debuggers can also be used to catch exceptions.

C++ does not have any kind of exception mechanism. Most implementations include an *assert* macro which stops the program if it fails, but this is useful only for debugging. The other advantages of the Eiffel assertion mechanism are lost.

4. Garbage Collection

Eiffel programs need a garbage collector. This locates objects that are no longer used by the application and frees the memory for reuse. In C or C++, the programmer must do this manually by instructions such as *free* or *delete*.

All Eiffel environments come with at least one garbage collector, and one (Tower Eiffel) offers a range.

4.1 Modular Software

It is not possible to write a large modular program without garbage collection. Dynamic memory primitives require that each block of memory be explicitly deallocated by the programmer when it is no longer needed. But the programmer may want to have several different parts of the program accessing a particular block. Therefore every module in the program which uses this block of memory must know what every other area is doing with it. Hence the programmers of each of these program modules must know about the implementation of all the other modules. This violates the fundamental tenet of modular programming that the programmer of a module need only know about that module and the interfaces of the others being used.

For instance, if I write a graphical editor using an existing class library, I would create objects to represent the shapes on the screen. Now when I instruct the class library to remove an object from the screen, I need to know whether the library will delete the object from memory. If it does not then I must do so explicitly. Meanwhile some other part of the application may be keeping a pointer to this object. So before I can know whether my module should delete this object, I have to know what every other module might be doing with it.

5. Arguments Against Garbage Collection

There are two common arguments against the use of Garbage Collection (GC) in production software.

1. It is slow. We cannot afford the overhead of garbage collection (the efficiency objection).

   Modern garbage collection algorithms do not take very much CPU time. The precise amount is a function of the algorithm and the object usage patterns of the application, but 10-15% is a typical figure.

   At first this looks like a good reason to avoid GC. A program which uses GC will run slower than the same program without GC. However it is impossible to write a large program using GC and then write the same program without using GC. The programmer without GC will be forced to keep track of which objects are used by which other classes of the program. For a large, complex package the easiest way of doing this might actually be to write a garbage collector, and some programmers have done this (usually badly: garbage collectors are complicated and tricky to get right). Failing that the programmer must use other strategies, such as copying and comparing whole objects instead of just references. This causes the program to run more slowly, but the overhead is hidden while garbage collection overhead is up-front and measurable. It is quite possible for a large program written with

garbage collection to run faster than an equivalent one written using manual storage management.

2. You can't wait for five seconds while the machine picks up its garbage (the *stop the world* objection).

   The first garbage collection algorithms could not cope with changes in the data suring collection. For this reason the garbage collector had to stop all other computation while it sorted out the live data from the dead. Modern collectors do not suffer from this problem. They are *incremental*. The memory allocator does a limited amount of garbage collection every time it is called. Such algorithms take up more CPU time as more garbage is produced, but they never stop the machine for long.

   Some applications need a response from the computer within a defined time limit. If the computer fails to respond quickly enough then something bad may happen. Examples include process control and robotics. These can be called *hard* real-time applications. Other applications can take a statistical approach to their deadlines, so that a typical requirement might be *99% of responses within one second*. These can be termed *soft* real-time applications.

   GC is acceptable in soft real-time applications provided that it behaves predictably. *Stop the world* GC is not predictable, but incremental algorithms are. Interactive software can be considered as a soft real-time application, since the user will expect a consistently quick response for most inputs. Users become annoyed or alarmed if the machine appears to stop without reason, but minor variations in response time are acceptable.

   Writers of hard real-time applications often have to count up CPU cycles for each branch and loop in order to prove the maximum possible response time for the system. This kind of engineering is currently beyond the scope of Eiffel or any other object-oriented language. Research is being done into garbage collectors which can give guaranteed response times, but commercial implementations are not yet available.

5.1 Adding Garbage Collection to C++

There are three basic strategies for adding GC to C++. They are all inefficient and require the programmer to observe yet more coding standards.

1. **The Macro Processor Solution** GC algorithms need information about the structures of the objects they are collecting. C++ does not provide such information, but it can be obtained by using calls to macros instead of the normal C++ *class* and *struct* declarations. However it is impossible to use two class libraries together if they rely on different garbage collectors.

2. **Reference Counting** Reference counting is often implemented in C++ class libraries because of its simplicity. The NIH and Interviews libraries both do this. The computer keeps a count of pointers and references to each object. When that count reaches zero the object is destroyed. Unfortunately reference counting GC is inefficient and will not collect garbage involving "cycles" where two or more objects point at each other.
3. **Conservative Garbage Collection** Normal GC algorithms need information about the structures of the objects they are collecting. Conservative GC does not. Instead it scans the memory looking for anything that might be a pointer into the heap space. When one is found that area of heap is considered to be *used*.

   This is the nearest that C++ can come to having true garbage collection, but there are still problems. Conservative GC algorithms cannot be incremental, because incremental algorithms require co-operation from the application program. The nature of some C++ constructs, especially address arithmetic into arrays, force the GC routines to be very inefficient. It is still possible to have memory leaks and premature collection because the GC cannot reliably distinguish between pointers and other values.

6. Simplicity

C++ is one of the most complex languages ever produced. Many people imagine that since it is *only* an improved version of C, it will be easy for C programmers to learn. This is not the case. There is more complexity in the ++ part of C++ than in most whole languages. Many aspects of C++ fit poorly with the original features of C, or seriously compromise the object- oriented aspects of the language.

As programmers have used C++ they have found problems, and asked for them to be solved. Solving problems in a language usually means adding bits on to the language. And so C++ has gianed multiple inheritance, exception handling, templates and downcasting. This process is still continuing today. The ANSI committee charged with standardising C++ are not merely clarifying the existing de-facto standard: they are adding even more features. And with each new feature comes more complexty for programmers and compiler writers, and more chance for bugs to slip through unnoticed.

At one time *The C++ Report* carried a quiz column which listed a few lines of C++ and asked *what does this do?*. Finding the right answer required a deep knowledge of the language.

7. Finding Features

C++ programmers regularly have to look for the definition of some routine by tracing back through a tangled of inheritance graph. The use

of multiple inheritance and non-virtual functions in C++ can make it extremely difficult to decide which version of a feature is being called.

Eiffel environments include automatic tools to do this. The whole interface to a class, including all its ancestors, can be printed out or viewed in a browser. The implementaion details can be stripped out or not as necessary.

This may seem to be an environment issue rather than a language issue. However in C++ the decision as to which feature is called at run-time depends on a mixture of the dynamic type of the object and the context in which the object is used: virtual routines use the dynamic type of the object, while non-virtual routines use the static type. This makes it impossible to summarise a class.

8. Training

As noted above, C++ is a complex language while Eiffel is a simple one. It takes a few days for a programmer to learn to use Eiffel effectively. It may take that same programmer weeks to learn to use C++, and even then there will always be new subtleties.

Bertrand Meyer designed Eiffel around his ideas of how programs should be written. The object paradigm shines through the entire language. C++ was designed by Bjarne Stroustrup to be an efficient superset of C with object-oriented constructs. The language usually obscures the object paradigm, and forces programmers to worry more about how they *can* implement something instead of how they *should* implement it.

9. Efficiency

Eiffel compilers make two optimisations which C++ forces on the programmer.

1. By looking at an entire system during compilation, Eiffel can find routines which are not redefined. Non-redefined routines can be accessed by static calls instead of the function-pointer indirection technique used to implement *virtual* calls. C++ compilers work on one file at a time. Hence this information cannot be gathered. It is up to the programmer to decide which routines will never be redefined, and remove the *virtual* tag from the definition.
2. Once routines have been identified as non-redefined, they may be *inlined*. Inlining substitutes the text of a function for a function call. This avoids the time overhead of a function call at some cost in space. Again Eiffel compilers make inlining decisions automatically while C++ forces the programmer to decide.

Since the programmer can never predict the future, the only way to write expandable software is to make all functions virtual and never use inlining. This reduces the efficiency of the system.

As described in the section on garbage collection, C++ often forces the programmer to make many copies of data structures rather than simply passing pointers. This is an aspect of the lanbguage, but it rarely shows up in the benchmark tests to measure language efficiency. C++ is penny wise but pound foolish.

10. Summary

- Eiffel is a much simpler language than C++.
- Eiffel provides facilities for the creation of reliable, well-defined modules and packages.
- Eiffel's use of garbage collection prevents many bugs and improves modularity.
- Eiffel environments can generate reference documentation from source code, avoiding the need to maintain documents in parallel.
- Large Eiffel systems are frequently more efficient than the equivalent C++ systems because of the hidden costs of C++.

11. Futher Information

- *Eiffel Related Books*, there is no shortage of books to more about the Eiffel Language and Method. Some follow in the next few items..
- *Object-Oriented Software Construction* by Bertrand Meyer describes the principles of software engineering which led to the creation of Eiffel. You should read this even if you intend to stick with C++.
- *Eiffel: introduction to the language and method*, by Bertrand Meyer derived from chapter 1, An Invitation to Eiffel, of the book Eiffel: The Language (see next item),
- *Eiffel: The Language*, by Bertrand Meyer is a combined reference manual and tutorial, but new users will find it heavy going.
- *Eiffel: An Introduction* by Robert Switzer is probably the best tutorial for new users.
- *`Eiffel: An Advanced Introduction* by Alan A. Snyder and Brian N. Vetter is an online (with pdf and ps formats) starter to Eiffel - particular good if you have some OO knowledge (ie Java/C++/Python/Perl5).

Internet users can browse the **Eiffel home page** at

`http://www.cm.cf.ac.uk/`

and **Eiffel Liberty** at:

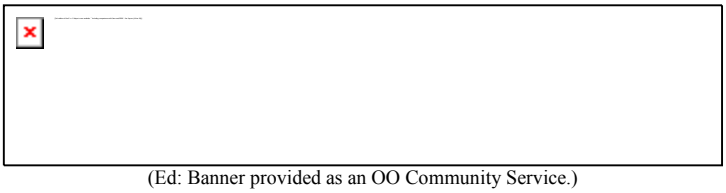
`http://www.elj.com/`

The Usenet group comp.lang.eiffel is dedicated to discussion of the language.

The **Eiffel Forum** at:

http://www.eiffel-forum.org/
is the *International Eiffel User Group* and is good place to learn more about Eiffel.

Paul Johnson can be contacted at paul.johnson@gecm.com, or telephone 01245 473331.



(Ed: Banner provided as an OO Community Service.)

[Eiffel Liberty *(New)* | GUERL | sOOap] [Top]

Page is http://www.elj.com/eiffel/projects/why/
Last Modified: 04 Mar 1998 (Created: 04 Mar 1998)