Study on Eiffel

Jie Yao Anurag Katiyar

May 11, 2008

Abstract

This report gives an introduction to Eiffel, an object oriented language. The objective of the report is to draw the attention of the reader to the salient features of Eiffel. The report explains some of these features in brief along with language syntax. The report includes some code snippets that were written in the process of learning Eiffel. Some of the more detailed tutorials, books and papers on Eiffel can be found at www.eiffel.com

Contents

1 Introduction

Eiffel is an object-oriented programming language. Its major goal is to allow programmers develop extensible, reusable and reliable software efficiently. Eiffel is widely used in many areas as a development platform. In particular, it's a very good language for teaching programming principles. The essence of Eiffel language is the set of fundamental principles, including design by contract, command-query separation, uniform-access principle, single-choice principle, open-closed principle, and option-operand separation. Many concepts initially introduced by Eiffel were applied into other object-oriented languages such as Java and C#. Originally, Eiffel was developed by Eiffel Software. This company named as Interactive Software Engineering Inc (ISE) was founded by Bertrand Meyer. Eiffel is named after the engineer Gustave Eiffel, an engineer who created the famous Eiffel tower. The language designers believe that the Eiffel has achieved its original goal. Any system built on Eiffel language is reusable and reliable, just like the design of the Eiffel tower.

This report is organized as follows. Section 2 talks about Eiffel's grammar and constructs. Section 3 presents presents basic language properties. Section 4 outlines the main design principles in Eiffel. Section 5 talks about the compilation process. Section 6 describes the exception handling. Section 7 discusses garbage collection. Section 8 describes Eiffel's approach to typing. Section 9 compares three major object-oriented languages: Eiffel, C++ and Java, and section 10 concludes our report.

2 Eiffel Constructs and Grammar

2.1 "Hello World"

Let us see the classic small example of the program that print the famous "Hello World" string in Eiffel notation. The example has a single class HELLOWORLD and single creation (constructor) procedure called make

```
class HELLOWORLD create make feature
      make is
            do print ("Hello World%N") end
end
```

2.2 Data Types

Eiffel supports two types of datatypes, the primitive datatypes (which are classes) and user defined classes. The following table lists the various types and their default initialization values in Eiffel

Type	Default Value
INTEGER,REAL,DOUBLE	Zero
BOOLEAN	False
CHARACTER,STRING	Null
Reference Types	Void reference
Composite expanded types	Same rules as above applied recursively to all fields

2.3 Classes

The basic construct in Eiffel is the class. A class represents entities, the attributes of those entities and the operations that those entities can perform. An example of Eiffel class is
class $TRUCK$
end

2.4 Libraries

To enable reusability, classes that frequently appear are organized into libraries, such as EiffelBase (also known as kernel or ELKS) for many basic types such as INTEGER,REAL, STRING, etc. Many collections such as LISTs and ARRAYs are also present in the base library. Eiffel has libraries to interface with CORBA and various databases. On MAC, Eiffel comes with MOTEL (full OO) interface to MAC OS. Libraries are not defined as part of the Eiffel language, they are simply collections of related classes.

2.5 Features

In Eiffel computational functions and procedures are together known as routines. Eiffel class attributes themselves can be fields or constants. These four entities are collectively known as Features. Let us see an example of a class with features.

```
class
    TRUCK
feature entities
    color: COLOR – a field
    velocity:INTEGER is – a function
            do
              Result := speed
            end
    wheels: INTEGER is 10 –a constant
    speed: INTEGER
    stop is – a procedure
            do
              speed := 0
            end
end–TRUCK
```

The features of a class are introduced by the keyword feature Comments are introduced by '–' Eiffel is not case sensitive.

2.6 Class relations and hierarchy

Two types of relations may exist between Eiffel classes. A class could be a client (has a relationship) of another class which allows its features to rely on the object of the other class. A class could be heir (is a relationship) of the other class.

2.7 Inheritance

In order to build new classes out of existing classes, we use inheritance in Eiffel as follows:

```
deferred  class
        VEHICLE
feature entities
        velocity:INTEGER is – a function
                do
                  Result := speed
                end
        wheels: INTEGER is
                deffered
                end
        speed: INTEGER
        stop is – a procedure
                deferred
                end
end–VEHICLE
```

```
class
    TRUCK
inherit
    VEHICLE
feature entities
    color: COLOR –a field
    wheels: INTEGER is 10 –a constant
    speed: INTEGER
    stop is – a procedure
            speed := 0
            end
end–TRUCK
```

This example shows single inheritance using the inherit clause. To redefine an inherited feature a class can use redefine clause. Defining a deferred feature is called as $effecting$ that feature. Eiffel has two specific classes ANY and $NONE$. ANY is at the top of inheritance hierarchy and exporting to ANY is equivalent to public in C++. ANY is automatically inherited by all classes. It is like Object in Java. $NONE$ is at the bottom of the inheritance hierarchy and exporting to $NONE$ is the equivalent of protected of C++. The *export* clause allows a class to control the level of export of its features (all exported,secret,selectively exported). In case of multiple inheritance, Eiffel solves the problem of two features with the same name being inherited from two different classes using the rename clause.

2.8 Genericity

Genericity helps in making program type safe without resorting to type casts. Eiffel has two types of generics, the normal unconstrained genericity and the constrained genericity. In order to use genericity in Eiffel, one has to create a generic class with formal generic parameters. These are generic types where the type is left open to be instantiated by actual generic types. Generics are most useful in the collection classes. Let us see an example of a $LIST$ the can store $INTEGER$s, $FLOAT$s and other objects. Such a generic $LIST$ class is declared as:

```
classLIST[T]
    ...
end
```

We can instantiate the actual lists as:
$list : LIST[INTEGER]$ –$LIST$ of $INTEGER$s
$list\_list : LIST[LIST[INTEGER]]$ –$LIST$ of $LIST$s of $INTEGER$s

2.9 Object Creation

Objects in Eiffel are created using instruction *create*. We first attach the reference variable(c) to the required Class(C) using
$c : C$
After this a call to the instruction *create* creates a new object of type C, initializes the fields to the defaults and attaches it to the reference c using the syntax
create c
We can also invoke a creation procedure with default values for the class fields during the construction of the object using the syntax
create c.some_procedure_name(some_default_value)

2.10 Exceptions

Exceptions in Eiffel are handled using the keywords rescue and retry. Let us see an example of exception handline in the program below

```
write_next_character(f:FILE) is
        --Write the available in last_character in to the file
        --retry 5 times
    require
        writeable:file.writeable
    local
        num_attempts:INTEGER
    do
        low_level_write_function(f,last_character)
    rescue
        num_attempts:=num_attempts+1
        if num_attempts< 5 then
          retry
        end
    end
```

An exception may cause a failure or it may be possible to "rescue" a routine from failure in the case of exception by equipping it with the rescue clause. A retry clause is only permitted in a rescue clause and its effect is to start again the execution of the routine without repeating the initialization of local entities.

2.11 Agents and Iteration

For some applications like numerical computation or iteration, operation instead of the objects may be more interesting. Hence operations are treated as objects and passed around to software elements, which can use them to execute the operations whenever they want. This separates the place of an operation's definition from the place of its execution; in addition the definition can be incomplete till the time of any particular execution. Agent objects are created to describe such partially or completely specified computations. Let as assume that we want to integrate a function $f(x : REAL) : REAL$ over the interval [0,1]. With our_integrator of a suitable type we can simply write the expression our_integrator.integral($\sim f(?), 0.0, 1.0$)
Note that the first $\sim f(?)$ the first argument to integral is an agent expression. The tilde sign tells us that we don't want to compute f yet. Instead we are passing to integral is an agent object enabling integral to call f whenever and with any value it wants to.

2.12 Tuples

Eiffel has a notion of tuples,arrays with variables of different types :
TUPLE[INTEGER,DOUBLE,EXAMPLE_CLASS] --tuple of length 3

2.13 Typing

A compiler can check statically that all type combinations will be valid, so that no run-time situation will occur in which an attempt will be made to apply an inexistent feature to an object. Static typing is one of the principal components of Eiffel's support for reliability in software development. The Feature Call rule in Eiffel allows it to follow a statically typed approach, where the applicability of operations to objects is verified at compile time rather than during execution. The rule states that 'A call to a feature in class X is only valid if the feature is made available using the feature clause to X'.
feature{<list of classes that are allowed to use this feature>}
feature{NONE} – the feature has become a private feature to the class it is defined in
feature{CLASS_A} – only CLASS_A is allowed to call this feature

2.14 Scope

Feature scopes are defined with the Feature Call rule. Eiffel has no global variables. But an attribute placed in a feature can be accessed by other features in the same class. Further, a feature can have local variables which are only accessible in the scope of the feature For example :

```
feature
        var1:INTEGER – can be used in entire class
        rout1 is
        local
            impossible:BOOLEAN – can only be used in rout1
        do
            impossible := item > 4
            print(impossible)
end
```

2.15 Memory Management

Eiffel provides a sophisticated garbage collector which efficiently handles the automatic reclamation process, while causing no visible degradation of a system's performance and response time. Here are some modes of the garbage collector

```
allocate_compact
                    – Enter 'memory' mode: will try to compact memory
                    – before requesting more from the operating system.
allocate_fast       – Enter 'speed' mode: will optimize speed of memory
                    – allocation rather than memory usage.
allocate_tiny       – Enter 'tiny' mode: will enter 'memory' mode
                    – after having freed as much memory as possible.
collection_off      – Disable garbage collection.
collection_on       – Enable garbage collection.
```

2.16 External software

In Eiffel some of the features can be **external** coming from languages such as C, C++, Java, C#, Fortran and others. For example a feature declaration may appear as

```
GetProcess_id(processid : INTEGER) : INTEGER is
                –Process id for the current process
    external
            "C" alias "getpid"
    end
```

The keyword external indicates that this Eiffel feature is an encapsulation of a C function whose name is *getpid*. Similar syntax exists to interface with C++ classes. This allows Eiffel as a system architecturing and software composition tool used at the highest level and which can provide extendibility, reusability and maintainability.

3 Fundamental Properties

In this section, we discuss Eiffel's fundamental properties. Some of them are common to object-oriented languages, while others are specific to Eiffel. We split these properties into two groups: "has" and "has no" depending on whether a certain property is supported by Eiffel or not. We will describe the important features in detail in following sections.

3.1 "Has" Properties

3.1.1 Multiple and repeated inheritance

Similar to C++, multiple inheritance is fundamental in Eiffel and is handled in an organized way. As a result of multiple inheritance, it's possible that a class inherits from another through multiple paths. In this case, the class creator can specify, for each repeatedly inherited feature, that it yields either one sharing feature or two replicas.

3.1.2 Dynamic binding

The default policy in C++ is static binding. Dynamic binding can only applied to routines when they are declared as 'virtual'. In contrast, dynamic binding is the default mechanism in Eiffel for routine calls. What's more, it can be achieved without any side effect on performance. The static binding certainly causes less overhead at run-time than dynamic binding. If routines never change at run-time, the Eiffel compiler will perform optimization automatically and apply static binding to them.

3.1.3 Type checking

Eiffel is a fully typed language. It has very strict type checks in assignments and it does not contain any unsafe type casts. Hence Eiffel is much more type safe compared to C/C++. As we know, type casts and other unsafe mechanisms are commonly seen in C programs, where as C++ is not a pure statically typed language.

3.1.4 Generics

Eiffel has support for Generics. In particular, it has two types of generics: one is unconstrained genericity and the other is constrained genericity. If the container of object can be an arbitrary type, this is unconstrained genericity. A generic class can also be declared with a generic constraint, then the corresponding types must satisfy some properties, such as the presence of a particular operation. This is known as constrained genericity.

3.1.5 Garbage collection

Garbage collection is common to object-oriented programming languages, including Eiffel. Eiffel has a good garbage collector to ensure good memory management at run-time, especially releasing the useless objects in an efficient way.

3.1.6 Consistency of the type system

Eiffel has a fully consistent type system in which every type, including basic types such as INTEGER, REAL etc., are defined by a class. The introduction of the notion of expanded class makes this possible and this is achieved without any effect on the efficiency of dealing with simple values such as integers, characters.

3.1.7 Assertions

The method and notation support writing the logical properties of object states, which are used to express the terms of the contracts. These properties can be monitored at run-time for testing and quality assurance. They are called as assertions. In Eiffel, assertions include routine pre-conditions which must be satisfied when a routine is called, routine post-conditions which must be ensured by the routine on exit, and class invariants which are global consistency conditions applying to every instance of a class.

3.1.8 Deferred classes

Deferred class is one of very important notions in Eiffel. It describes a abstraction which is not fully implemented, similar to abstract class in C++ or interface in Java. Two aspects are particularly important: the ability to define a partially deferred class, which contains

both implemented and non-implemented routines; and the ability to attach assertions to a deferred class and its deferred routines.

3.1.9 Renaming

Eiffel offers a powerful technique related to multiple inheritance called 'renaming'. A class can rename inherited routines and attributes. This is very useful in removing name clashes in multiple inheritance and providing locally adapted terminology when you inherit the right features but under the inappropriate names.

3.1.10 Exception Handling

Eiffel's exception handling is based on a strong emphasis of program correctness. Exception is defined with respect to assertion, and exception handling is constructed to observe the semantics defined by the assertion.

3.1.11 Automatic recompilation

One of the most important practical aspects of Eiffel is the automatic compilation mechanism. It will analyze the inter-class dependencies in an automatic way, as a result, there is no need for make files and include files.

3.1.12 Documentation

Similar to Java, Eiffel allows automatic generation of documentation from the source code. This acts as a substitution for separately developed and maintained software documentation. It produces full views of all classes and the whole systems.

3.2 "Has no" Properties

3.2.1 Main, Global variables and side effect operators

In Eiffel system, there is main program. Consistently, global variables are also absent from Eiffel. It is well know that global variables are detrimental to modularity and do harm to quality of the software. There are no side-effect expression operators which could cause confusing computation. There is no goto statement as well as no union types in Eiffel.

3.2.2 Inline expansion

Eiffel doesn't support inline expansion. In Eiffel, all optimizations are performed by the compiler instead of by the human user. Thus, the optimizer will automatically decide whether to expand certain routines in-line based on certain systematic criteria. For instance, if the routine is not subject to redefinition and dynamic binding, or the number of calls to the routine are below a threshold it will be considered for inlining. It's understandable that compiler can perform such operation in a more efficient and safe way.

3.2.3 Friend functions

A friend function is considered to be a dangerous violation of the object-oriented principles, there is nothing equivalent in Eiffel.

3.2.4 Others

Eiffel does not have pointers, type casts, pointer arithmetic, function pointers, malloc, free. All of these are known to well known unsafe factors in other languages.

4 Design principles in Eiffel

4.1 Design by Contract

DBC is a design methodology that enables the specification of mutual obligations between client and supplier classes. Preconditions are conditions the client has to fulfill in order that the supplier carries out the required operation properly. Postconditions are warranties by the supplier on the quality of the operation execution, provided that the preconditions are met. In addition to preconditions and postconditions, DBC also supports invariants. Invariants capture the deeper semantic properties and integrity constraints characterizing the instances of a class Contracts capture consistency conditions. In Eiffel, contracts are monitored at run-time. Any contract violation signals a constraint violation and consequently results in throwing an exception. Eiffel provides syntax for expressing preconditions (require), postconditions(ensure) and class invariants(invariant).
Let us look at a class *ACCOUNT* with some assertions.

```
indexing description : "Simple bank account"
class ACCOUNT
feature – Access
        balance : INTEGER –Current balance
        deposit_count : INTEGER is
                –Number of deposits made since opening
                do
                  if all_deposits /= VOID then
                        Result:= all_deposit.count
                  end
                end
feature – Element change
        deposit(sum:INTEGER) is – Add sum to account
                require
                        non_negative: sum >= 0
                do
                        if all_deposits = VOID then
                            create all_deposits
                        end
                        all_deposits.extend(sum)
                        balance:=balance+sum
                ensure
                        one_more_deposit: deposit_count= old deposit_count+1
                        updated: balance= old balance+sum
                end
feature{NONE} – Implementation
        all_deposits: DEPOSIT_LIST
                –List of deposits since account's opening.
invariant
        consistent_balance : (all_deposits /= Void) implies
                        (balance=all_deposits.total)
        zero_if_no_deposits : (all_deposits = Void) implies
                        (balance=0)
end–class ACCOUNT
```

Here a call to deposit is correct if and only if the value of the argument is nonnegative. The routine does not guarantee anything for a call that does not satisfy the precondition. The postcondition of the deposit routine expresses what the routine guarantees to its clients for calls satisfying the precondition. The notation old expression valid in postcondition, denotes the value that expression had on entry to the routine. The class variant is applied to all features and it must be satisfied on exit by any creation procedure and is implicitly applied to both the precondition and postcondition of every exported routine. The total of

all_deposits is checked to be compatible with the *balance*.

deposit	Obligation	Benefits
Client	(Satisfy precondition) Use a non negative argument	(From postcondition) Get deposits list and balance updated.
Server	(Satisfy postcondition) Update deposits list and balance	(From precondition) No need to handle negative arguments.

4.2 Command Query Separation

This principle states that every method should either be a command that performs an action, or a query that returns data to the caller, but not both. Moreover methods should return a value only if they are referentially transparent and hence possess no side effects. CQS makes it difficult to implement re-entrant and multi-threaded software correctly. In Eiffel commands have no result and may modify an object. They may only be procedures. In Eiffel queries are used to return the information about an object. The queries can be implemented either as an attribute or as a function.

4.3 Uniform Access Principle

The principle of Uniform access states that all services offered by a module should be available through a uniform notation, which does not reveal whether they are implemented through storage or through computation. In Eiffel there is no difference between a query implemented as an attribute and one implemented as a function. For example to obtain the speed of a truck a one can write $a.speed$ where $speed$ could be attribute or a function without arguments. The UA principle contributes to Eiffel's goal of extendibility and maintainability. The implementation can be changed without affecting clients and classes can be reused without having to know the details of its features' implementations.

4.4 Single Choice Principle

The principle of single choice states that whenever a software system must support a set of alternatives, one and only one module in the system should know their exhaustive list. Eiffel follows single choice principle. Eiffel uses a combination of inheritance, redefinition, polymorphism and dynamic binding to make possible a point of single choice, a unique location in the system which knows the exhaustive list of variants. Every client then manipulates entities of the most general type. For example the class $VEHICLE$ through dynamically bound calls of the form $vehicle.some\_vehicle\_feature(...)$ allows every client to manipulate the entities through the the most general type $VEHICLE$ even thought the actual account could be $TRUCK\_VEHICLE$ a variant.

5 Compilation Process in Eiffel

"Bytecode" denotes a low-level code that can be emitted by compilers and run directly through an interpreter. The Eiffel compiler also generates bytecode. The bytecode 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 process is known as 'finalization', which performs extensive optimizations (routine inlining, static calls, array optimization,dead-code removal), permitting performance achievements. Using C as an intermediate language takes advantage of the platform-specific optimizations performed by C compilers, and most importantly, facilitates interoperability with software written in C and C++. Only in development mode is the bytecode used for execution. Where optimal speed is required, the bytecode ceases to be executed, it is then used to generate optimized C and machine code. The Eiffel compiler takes care of optimizations of the object-oriented mechanisms: inheritance, information hiding, inlining, dynamic binding. The C compiler takes care of general-purpose, lower-level optimizations: loops, expressions, constants.
EiffelStudio is the Integrated Development Environment (IDE) designed exclusively for the

Eiffel Object-Oriented language. EiffelStudio supports gcc on most linux platforms and the native 'cc' compiler if its an ANSI C compiler. On Windows it supports Microsoft Visual C++ environment and the Bordland C++ compiler. Eiffel supports efficient compilation mechanisms through mechanisms such as the Melting Ice Technology of the EiffelStudio. The Melting Ice achieves immediate recompilation after a change, guaranteeing a recompilation time that's a function of the size of the changes, not of the system's overall size. Such a melt (recompilation) can immediately catch (along with any syntax errors) the type errors.

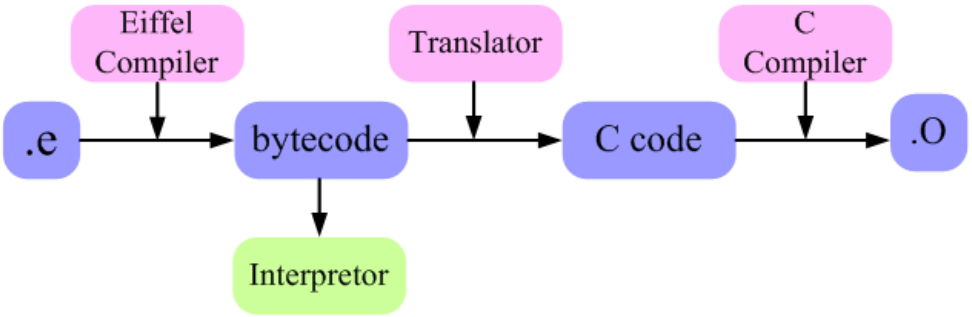

Figure 1: Figure showing the different stages of compilation in Eiffel

6 Exception Handling in the compiler

Eiffel has the termination model of exception handling, where control transfers into the scope of the handler and activation records separating the exceptional event from its handler are discarded. Eiffel allows resumption, but only of the failing block within the same routine as the handler which catches the exception. When an exception E is raised, the dispatcher determines what activation record contains a handler for E, and transfer control to that handler in that activation record. This search always involves traversal of a stack, either a special exception stack that is explicitly maintained by the normal-case code as it executes, or by walking up the stack of activation records. Eiffel uses threaded exception stacks which are based on the use of setjmp, longjmp, and a separate exception stack. Macros are used to obtain exception-handling in C itself. The major disadvantage to this technique is that it tends to impede optimization. Since C is used as an intermediate language, calls to setjmp tend to turn off optimization.

7 Garbage Collection for Eiffel

Running a program written with an object-oriented language means creating and destroying run-time objects. If dead objects are not collected, they pollute memory and shorten the execution of the program. Eiffel relies on the existence of a Garbage Collector. One important property of Eiffel garbage collector is that it not only frees memory for reuse by further object allocations, but actually returns memory to the operating system, freeing it for reuse by not just by Eiffel itself but by any other part of the application or even another program. Eiffel uses its own memory management mechanism. Instead of directly mallocing objects from the operating system, an application asks the kernel for memory chunks and allocates objects in these chunks. For Eiffel users, the principally relevant property of the garbage collector is that objects can move. This is totally invisible to the developer.

7.1 Garbage Collector Structure

The garbage collector actually uses several basic algorithms that can be used together or independently. The right algorithm is determined by the circumstances of each call, including for example the urgency of the memory need.

7.1.1 Mark and sweep

The mark-and-sweep garbage collection algorithm is considered to be the most effective one since it frees all memory can be freed. As we all know, reference counting can not handle circular structures, while two-space copying leaves half of memory unusable.
The mark-and-sweep consists of two steps. The first step recursively explores the live objects and marks them. The second step collects live objects and put unmarked chunks back to free list. It's often combined with compaction to provide the largest possible chunk of memory available.

7.1.2 Generation scavenging

Generation scavenging, another one of the algorithms, relies on the observation that young objects have a smaller probability of surviving than old objects. Hence the idea of splitting objects into several generations. The young generations will be collected more often than the old ones. Since most of the objects have a short life, it is likely that the biggest part of recycled memory will be obtained from the young generations. A main advantage of this technique is that a collection need not explore all the objects. Only old objects with references to young ones and local variables are recursively followed. A generation is collected when its occupation has reached a given value. When a generation is collected, all the surviving objects become older. When they have reached a given age, they are graduated to the next generation. The algorithm achieves the right tradeoff between low generation age (meaning that the old generation becomes too big) and high generation age (meaning too frequent scavenging).

7.2 Garbage Collector in Action

When the primary area is full, a cycle of garbage collection (generation scavenging) occurs. In the worst case, this cycle does not collect enough memory and a full collection (mark-and-sweep) is launched, possibly with a memory compaction step. Only as a last resort will the application asks the operating system for more memory, if it is still not possible to allocate a new object. The main algorithms are incremental, and their time consumption is negligible compared to the program run-time. Internal statistics keep track of the memory allocated and helps to determine the proper algorithm to call. The garbage collector can be diverted from its normal behavior through the speed option. In this case, instead of trying to collect all the available memory, it will ask for more memory earlier. It is thus fully under the developer's control to decide how best to resolve the space-time tradeoff involved. By default, each Eiffel application includes the garbage collector and uses it. But one can control the behavior of the memory management mechanism and tune it to fit to their specific needs through a set of procedures available from class MEMORY, which can be enabled to turn the garbage collector on and off, and launch full collection cycles at specific times.

8 Eiffel's approach to typing

Generally, static typing helps productivity and flexibility if it's done properly. C++ offers static typing, but it's too constrained for realistic software development. This the main reason why C++ programmers can't stop using type casts to cheat with the type system. Eiffel, in contrast, does not allow programmers bypass the type system. Surprisingly, this constraint doesn't lead to any limitation. As a matter of fact, Eiffel's typing system ensures safety, clarity and productivity. These quantities are achieved by the combination of features, including multiple inheritance, genericity, anchored declarations, assignment attempt and covariance.

8.1 Multiple inheritance

Multiple inheritance makes it possible to view an object from two or more viewpoints. For example, a TEACHING_ASSISTANT can be both a STUDENT and a FACULTY. Here

TEACHING_ASSISTANT class inherits from both classes: STUDENT and FACULTY. Unlike C++, Eiffle handles the multiple and repeated inheritance in a very careful and clean way, making multiple inheritance a realistic tool.

8.2 Genericity

As mentioned in section 2.8, Eiffel supports generic classes, which are similar to C++ templates. For instance, class LIST[G] describes lists of any type G. If you use MY_TYPE, e.g. LIST[MY_TYPE], the elements can be of any descendant type of MY_TYPE. This is called unconstrained genericity.
Differing from unconstrained genericity, the constrained genericity indicates that the corresponding types must satify certain properties. For example, if you define a class SORT_LIST [G→COMPARABLE], then only comparable types can be used, such as SORT_LIST[INTENGER]. This feature is absent in C++.

8.3 Anchored declarations

An anchored declaration, x: like y, is an interesting operation. It means that whenever y is redeclared in a descendant class, x follows y's new type automatically. Certainly, it saves a considerable amount of verbose re-declarations.

8.4 Assignment attempt

Assignment attempt is an operation special to Eiffel: $x? = y$. The normal assignment x := y, by the rules of typing, requires the type of y to be a descendant of the type of x. But in some cases, we may have no idea about the type of y at run time. Assume y is an object retrieved from network or database, but we may still want to make the assignment. Using assignment attempt in this scenario can ensure the assignment to be safe even though the type of the y object at run time may not compatible with that of x. Assignment attempt achieves this by doing this: it first looks at the actual object type, if that type is compatible with that of x performs the assignment; otherwise assigns to x a void value.
Though assignment attempt is not used very often in a well-written object-oriented system, it makes static typing more flexible.

8.5 Covariance

Sometimes, we may want to redefine an object to be descendent of its original type. Eiffel uses covariance to achieve this. It goes with the "anchored declaration" policy and is not available in C++.

As a result of above mechanisms, typing in Eiffel can be very helpful. First of all, every object with a type makes the source code easier to read and maintain. Secondly, though Eiffel is fully typed, it doesn't limited programmer's creativity. In addition, static typing helps Eiffel compilers generate highly optimized code. Finally, static typing enhances the safety of the software. This benefit is obtained in Eiffel without any negative effect.

9 Object Oriented Language Comparison

The following table compares three major O-O languages Eiffel,C++ and Java

Factor	Eiffel	C++	Java
How object-oriented	Purely OO	Hybrid	Purely OO
Age	Since 1985	Since 1983	Since 1995
Design by Contract and assertion	Design by Contract Language support	Nothing comparable Only assert instr	Nothing comparable Only assert instr
Static typing	Statically typed	Statically typed but C style cast allowed	Typed mostly statically but dynamic for containers
Compiler Technology	Combination of interpretation and compilation	Usually compiled	Mix of interpretation and on the fly compilation
Automatic Documentation	Documentation extracted automatically without extra programmer effort	No standard mechanism	JavaDoc: add special comments
Exception handling	Exception handling	Exception handling	Exception handling
Multiple Inheritance	Multiple inheritance	Multiple inheritance but with problems	Single inheritance but multiple interface
Automatic memory management	Garbage collection automatic memory management	No Garbage collection	Garbage collection
Openness and interoperability with legacy software	Standard language support for integration with C and C++	Good interoperability with C	Native methods

10 Conclusion

Eiffel is a language for efficient development of quality systems. It is more than a programming language. It covers the entire range of software development cycle right from Analysis, Design and Implementation to Documentation. With it's total object oriented approach it is a pure object oriented language with many design features that one would expect in an ideal O-O language. It allows programmers to develop extensible, reliable and reusable software. It is used in finance, education, healthcare as a development platform. As someone has aptly said 'I like Eiffel. Its flaws are more usable than most languages' features.'

REFERENCE

1. http://archive.eiffel.com/
2. http://wapedia.mobi/en/Eiffel_(programming_language)
3. Bertrand Meyer, An Eiffel Tutorial, July 2001.