Eiffel, the universe and everything. (somethings anyway)

Technical Report No.138

A. Mayes and R. Barrett

May 1992

Eiffel, the universe and everything.(somethings anyway)

Audrey Mayes and Ruth Barrett

18 May 92

1 Introduction

This document is a brief introduction to the programming language, Eiffel version 2.2, as currently available on the Sparc Workstations. It assumes a basic knowledge of object oriented concepts, terminology and techniques. Further details of the terminology and techniques associated with Eiffel can be obtained from Meyer [1]. Version 3 should be available during 1992. Any changes will be reported in a subsequent document. The remainder of this document has two sections. The first section contains the general information required to access and use the Eiffel environment. An example system supplied with the environment is used to demonstrate the available tools. The same example is used in the second section as an introduction to the syntax of the language.

2 What is Eiffel?

Eiffel is a true object oriented programming language which was devised by Meyer [1]. It was designed to have all the qualities he considered to be necessary in a software system. These qualities are correctness (compliance with the requirements and specification), robustness (the ability to function under abnormal conditions), extendibility (the ability to adapt to changing requirements), reusability (the ability to use some part of a system in a new application) and compatibility (being able to combine with other products).

A program, or system as it is called, is written as a group of classes. Each class provides a number of features which can be used by other classes as required. There are two kinds of features. These are attributes, which contain data items, and routines which perform some computation. The attributes of a class are the fields which contain the data. Attributes can be simple types, such as integer, real, character and boolean or class types, that is a type declared by another class declaration. The **root** class is responsible for setting the system going and is roughly equivalent to the main program in C or Modula-2.

During the running of the program these class definitions are used to produce objects. Thus objects are only found during the execution of the program and classes are only found in the program. This is comparable to the difference between programs and processes in procedural languages such as Modula-2 or C, in that programs and classes are what is written and processes and objects occur as a result of the execution of the code.

During the execution of the program, the objects communicate with each other by passing messages. These messages are the run time equivalent of one class using a function supplied by another class.

Another feature of Eiffel which is peculiar to object oriented languages is inheritance. The basic idea of inheritance is that a new class can be formed by adding extra facilities to an existing class or by combining two or more existing classes. The new class (called a descendant) formed has all the properties of the

existing class (called an ancestor) plus any extra properties declared in its own definition.

2.1 How to access the Eiffel system.

The Eiffel system can be accessed via the Sparc workstations. If you are unable to access the eiffel system the following should be added to the .login file:-

setenv EIFFEL /usr/local/Eiffel
/usr/local/Eiffel/bin — added to the set path command
/usr/local/man — added to the setenv MANPATH command

The Eiffel system requires the use of an editor. The default editor is 'vi'. This can be changed by adding the following to the .login file:-

setenv SDF_EDITOR <editor name eg. emacs>

When working with Eiffel it is best to use a workstation with 'Open Windows' and an xterm interface as some of the facilities (eg the browsers see subsection 2.3.4) do not work with a command tool interface. It is possible to remote login to the Eiffel environment from a terminal via sol but again some of the facilities do not work.

The sequence of events required to get access to the Eiffel environment on the workstations is:-

login
% openwin
% xterm & (wait for a new window)

If a print out is required, it is sometimes necessary to remote login to sol and then print the file.

The Eiffel system contains a library of basic classes. These are contained in the directory

/usr/local/Eiffel/library

A copy of all the library facilities provided can be obtained by printing out the short form, see Section 2.3.3, of each of the libraries.

2.2 Examples of Eiffel programs

All the source files have a .e extension (maximum 12 chars +.e). The directory

/usr/local/Eiffel/examples

contains some example programs. The simplest example is contained in the GUIDED_TOUR directory and will be used throughout this paper. The example can be copied into your area using the command:-

cp -r /usr/local/Eiffel/examples/GUIDED_TOUR .

2.3 THE UNIVERSE and other useful facilities of the Eiffel system

The universe of an Eiffel system is the set of classes which are needed to compile and assemble the system. The universe line of the System Descriptor File (SDF) is used to define the universe. The classes contained in the current directory and those in the kernel library are included automatically. The directories containing any other classes required by the system must be listed in the SDF. The kernel library contains classes which provide all the basic system needs such as input and output to the screen, arrays, strings and files. The group of classes found in a directory is called a cluster. This is a term used in the browsing tools (see section 2.3.4). It is suggested that clusters should contain logically related classes.

The remainder of this section gives some details of the facilities available in the Eiffel environment. More information can be obtained on each of the utilities by using the Unix "man" command eg. type

man es,

for information about the es command. The files in the directory:

/usr/local/Eiffel/doc

contain the same information.

2.3.1 ec

ec stands for Eiffel compiler. The **ec** command is used to compile classes. The following command is used:-

ec <classname> *(again classname is lower case with or without an extension)*

for example type "ec point1"

When using **ec** all the classes required by the class being compiled are checked to ensure that the latest version is used, supplier classes are then recompiled if necessary. If all these are in the same cluster, that is in the same directory, the SDF is not used. After running **ec**, one new directory is produced

for each .e file. These have the extension '.E' and contain files generated by the compiler.

2.3.2 es

es stands for Eiffel system. It is used to compile and assemble the system. The class name is not required when using **es**. eg. In the GUIDED_TOUR directory type

```
es
```

The first time the **es** command is called in a directory, the user is prompted to fill in the System Descriptor File (SDF). This is used to tell the system which system is to be assembled, where to find the basic library classes (in the line UNIVERSE:) and other information required by the compiler.

The only line which needs to be edited for the example is the ROOT: line. The "root_class_name" needs to be replaced by the actual name of the class to be compiled. In the GUIDED-TOUR example the name of the root class is session, so "root_class_name" should be replaced by "session".

The messages appearing on the screen during the execution of the **es** command show that it is not necessary to compile each class individually. The system checks and compiles each class used by the root class, as required in turn, before it compiles and assembles the root class. A runnable file, <root_class_name>, and a .eiffel file are produced as well as the ones produced by the compiler. The .eiffel file contains the amended system descriptor file. The program can be run by typing the name of the executable file, eg type "session" if using the above example.

The other example systems in the /usr/local/Eiffel/examples directory already have a .eiffel file with the correct root class name added. Beware! Some of the names of the root classes are not the same as the names of the directory they are in. The .eiffel file contains the file name to be used run the program. These examples take up a large amount of disc space, so may cause problems if more than one is resident in your area.

2.3.3 short

The complete class declaration contains more information than is needed by a programmer using the class. The short command is used to display only the required information. This information includes:–

1. the definition of each of the exported features, including the create function if one is defined,
2. any preconditions or postconditions associated with the features
3. the comments written by the programmer to accompany the feature declaration

4. the type of each of the exported attributes.

A call takes the form:–

short <file name>

The following is the result of running the short command on the class INTERACTION which is found in the GUIDED_TOUR directory.

```
class interface INTERACTION exported features

        over, get_request, one_command

feature specification
        over: BOOLEAN

        Create
                        -- Create a point

        get_request
                        -- Ask what the user wants to do next,
                        -- returning the answer in attribute 'request':
                        -- 'Up', 'Down', 'Left', 'Right' or 'Quit'.

        one_command
                        -- Get user request and execute it

end interface -- class INTERACTION
```

The amount of information produced by the short command is comparable to that included in a Modula-2 definition module or Ada package specification. This example also shows that the Create feature is part of the interface even though it is not included in the export list. See section 3.1.2 for information about the create feature.

2.3.4 eb and good –browsing tools

These two commands are used for browsing through the classes available to the system, ie those contained in the current directory and in the directories listed in the universe entry of the System Descriptor File. The **eb** command accesses a textual browser and the **good** command accesses a graphical browser. **good** is a viewing tool. It is used to graphically display the relationships between classes,

and to examine the code for the classes. Classes must be compiled before **good** can give any information about them.

eb is more interactive. It is possible to compile classes, edit existing classes, add new classes and to run the system from within **eb**. It is also possible to see a textual form of the relationships between classes.

2.3.5 et

et stands for Eiffel test, (not for green monsters of the universe!). This command is used to produce a test program which can be used to test classes interactively without having to write a special test driver. A call takes the form

et <class1>,<class2>,etc

All the classes listed must be in the current directory or one listed in the SDF ie must be in the current universe.

The test program is then automatically run, allowing all the features of the listed classes to be checked. That is the theory, but the easiest way to test and debug a program is to use the ALL_ASSERTIONS option in the system descriptor file as recommended by the author of the language.

2.3.6 ancestors

This command lists all the ancestors, ie. the classes from which features are inherited, of a compiled Eiffel class. A call takes the form

ancestors <classname or filename>

Indentations are used to indicate levels of inheritance.

2.3.7 flat

This gives an inheritance free version of the class, ie. it lists all the features inherited from other classes as well as those declared in the class. Inherited features are preceded by a statement –Feature from <class>. A call takes the form

flat <classname>

The class must be already compiled.

3 The Eiffel language

This section contains an introduction to the Eiffel language and its syntax.

3.1 The structure of an Eiffel class declaration

As stated before, an Eiffel program consists of components called classes. The class declaration contains both the definition and the implementation of the class, unlike Modula-2 which has separate definition and implementation modules. Below is a generalised class definition. The Eiffel reserved words are in bold type, the explanations are in italics:-

class CLASS_NAME *(capitals for class name is an Eiffel convention)*

export

(a list of the features which are available for use by other classes. They form the interface of the class. The root class will not have an export list unless it can be used as part of another system.)

inherit *(optional list of ancestor classes .)*

ANCESTOR_CLASS **rename** a **as** a1

(This facility is used to avoid name clashes and give more meaningful names to the properties of the new class)

ANCESTOR_CLASS **redefine** a

(used to put in a different piece of code to be used for the new class under the same name as the original feature.)

feature *(This section contains*
i) the type declarations for the attributes of the class which are equivalent to the fields of a record.
ii) the code used to implement the functions which form the interface and those required by the class itself)

a : CLASS1; *(a and b are attributes of the class CLASS_NAME .*

b : CLASS2; *They are instances of other classes CLASS1 and CLASS2)*

function_name (parameter list) **is**

local

(optional declaration of variables used in the function)

require

(optional declarations of preconditions)

do

(code required to perform the function)

ensure

(optional declaration of postconditions)

end;––function_name *(–– indicates comments)*

(more feature declarations)

invariant *(optional list of conditions which must be true after creation of the object and after every call to a routine.)*

end ––CLASS_NAME

3.1.1 the export list

Attributes are made accessible by exporting the name of the attribute. It is not necessary to supply a specific "get_attribute" routine. It is not possible, however, to assign a new value to the attribute outside the class, the compiler rejects the code giving an unhelpful message (see Section 3.5). The effect of this is that attribute and parameter-less functions appear the same. This gives more freedom to the implementor of the class to use whichever method they think is best and to change this without any change to the client class, eg a derived value could be held as an attribute or computed each time it is required, the frequency of access might determine which is the best implementation but will not change the way the class is used.

3.1.2 the create function

One feature that all Eiffel classes have, is a create function. This is a standard function which is exported by all classes. Attributes of the simple types, that is integer, real, char and boolean, are produced at runtime as actual instances of the type. Attributes of other classes, known as class types, are implemented as references to the corresponding object at runtime, not the actual object itself. The create procedure is called to produce the object and relate it to the reference held in the calling class. It is impossible to use any of the features supplied by a class until the create function has been called. A default create procedure exists which initialises all attributes of the class according to their type, as shown in the following chart [1].

type	Initial value
integer	0
real	0.0
boolean	false
character	null
class type	void reference

If these values are not suitable, a new create function can be defined.The feature is then listed in the interface displayed by the *short* command. One important use of a user defined create function is in the root class, which is used to start the system running.

The class SESSION, shown below, is the root class (the one responsible for driving the system) of the example in the GUIDED_TOUR directory. As this is a simple program, the root class only contains a create feature. It could contain other features as well and also have exported features if the system is part of a larger one.

```
class SESSION feature
        Create is       -- Execute sequence of interactive commands
                local
                        interface: INTERACTION
                do
                        from
                                interface.Create
                        until
                                interface.over
                        loop
                                interface.one_command
                        end
                end -- Create
        end -- class SESSION
```

3.1.3 system execution

When a system is executed, the create function of the root class is invoked. In the guided tour example, this means the create function of the class session. As can be seen above, the create function declares a local variable of class INTERACTION. The features exported by this class can then be called. The create feature is, of course, the first one called. The system then repeats the feature one_command until the boolean variable over is set to true.

3.1.4 syntax

Eiffel syntax is fairly simple. A semicolon is used as a statement separator. There is only one iterative statement. This is the loop statement an example of which can be seen in Section 3.1.2. There are two forms of conditional statement. These are

1. The if statement, which takes the form:-

   if *conditional* then *statements*

   else *statements*

   end

2. The inspect statement, a multi-branch instruction, which takes the form:-

   inspect *variable*

   when x then *statements*

   when y then *statements*

   else *statements*

   end

There is one other important piece of syntax. That is the dot notation as in interface.Create, see class SESSION above, is used to link the action required to a specific object. It means take the object named "interface" and apply feature "Create" to it. The same notation is used to access all features. The following code for the one_command feature of class INTERACTION shows the use of the dot notation when a feature requiring parameters is called.

```
one_command is
        -- Get user request and execute it
        do
        get_request;
        inspect request
        when Up then
                my_point.translate (0., 1.)
        when Down then
                my_point.translate (0., -1.)
        when Left then
                my_point.translate (-1., 0.)
        when Right then
                my_point.translate (1., 0.)
        when Quit then
                over := true
        end;
        my_point.display
```

end −−one_command

3.2 Types

There is no inbuilt facility to declare enumerated or subrange types. This is because they do not fit in with the Eiffel notion of type. Eiffel has only the four simple types plus class types. All the features in a class are eventually represented as one of the four simple types.

The way to implement an enumerated type is to represent each value by an integer, for example:

Monday: INTEGER is 1;
Tuesday: INTEGER is 2; etc

The class containing these declarations would also contain the features required to use this type. A class containing an enumerated type might be used by more than one other class in a system, that is they may contain an attribute of that class type. This would normally result in many instances of the class being produced. These instances would all be identical. For example a class CALENDAR would always contain the same information, such as May has 31 days. To avoid duplicating the information, the class can be declared as a **once** class [1] so that it is instantiated the first time it is called and any other objects can access this one instance.

3.3 Peculiarities of the Eiffel language or potential black holes.

3.3.1 "readline"

This procedure reads in a string of characters until a new line or end of file is input. This string of characters is put into a variable called laststring, which is of type STRING. STRING is a complex type which is predefined. This means that any variables of type string are implemented as references to the variable, not as actual instances of the variable. The statement

x:= io.laststring;

therefore binds x to the variable laststring, that is it always takes the value of the latest io.laststring, not the value it had at the moment the assignment to io.laststring was made.

In order to assign laststring permanently to a variable 'x', the statement

x := io.laststring.duplicate;

must be used. This makes a duplicate copy of the latest string input and assigns this to the reference x.

3.3.2 "readreal" and "readint"

These two input procedures read the input into a lastreal or lastint variable. This can then be assigned to the required variable. Real and integer are simple types, so just the normal assignment is needed. In common with other languages, a space is enough to separate a series of numbers, but the program will not go on to the next stage until a <return> is entered. In Eiffel this causes some problems such as :-

1. If two numbers are entered, separated by a space, when only one is required, the first one is allocated to the variable and the second one is stored in a buffer, waiting to be allocated. It is used for the next required input.

2. The return used to terminate the input of a number is also buffered and appears to lurk around waiting to terminate the next string input. This results in the user not being given an opportunity to enter the next textual information requested, a null value is inserted instead.

Both of these difficulties can be solved in the same way namely by forcing each input to be on a separate line. The way to do this is to follow each input of a number by an "io.next_line " instruction which reads the return and starts a new line on the standard input.

The instruction io.new_line forces the output to a new line but has no effect on the input so does not clear the return and can not be used in the same way.

This problem is a recognised "feature" of the Eiffel language. The situation arises because of the way readreal and readint are implemented. The required value is read into a variable, laststring, and then converted to the correct format and put into lastreal or lastint. The laststring variable is not cleared, leading to the above problem. It is unlikely that this will be changed as existing software would need to be altered to comply with the new implementation.

3.4 Reuse of existing classes

The short-flat form of a class is supposed to contain enough information to allow a programmer to understand and use the class and all its features. An attempt was made to reuse the demo-driver class. This class provides features to allow a user of the system to choose an option from a menu. Several problems were encountered. These were:-

1. The output did not appear on the screen in the expected order. This happened because the default output in the demo-driver class is set to the

standard error file which runs asynchronously with the normal standard output file, so it is pot luck which gets to the screen first. Two methods to deal with this situation were suggested by Applied Logic Developments, these were either to redirect the required output to the error file or to change the demo-driver class to write to the standard output instead. This second method is possible because the demo-driver is a high level class so other classes should not be effected. As a general rule, it is better not to change classes which have been provided.

2. The add_entry feature is supplied to add an entry to the menu. The comments state that

   " If it (an entry) contains upper case letters, they will be used as a tag for recognising the users input."

   This implies that the use of the tag is optional. This is not the case as the print_menu displays a heading stating:-

   "Select a comment by typing the first two letters in upper or lower case (? for help)"

   The print_menu comment is simply "Print menu". which gives no help with the structuring of the menu.

   After reading the code for all the features it is obvious that:-

   (a) In order to use the supplied print menu the upper case tag system must be used

   (b) A help facility must be added in order for some of the other features to work correctly. This is supplied by the feature

   complete_menu -- Add help command to menu.

   There is no instruction to say that this must be called before print_menu.

Problems of this nature make the reuse of classes difficult, if not impossible. The problem could be avoided by adding copious comments to tell the user exactly how to use the class and its methods or by making use of the features supplied by the language. As can be seen from subsection 3.1, Eiffel provides **require** and **ensure** to implement pre and post conditions. These could have been used in the demo-driver class to ensure that the class would be used correctly. Boolean variables would be set to true by a feature and tested as a precondition for another procedure. For example, the present print_menu feature has no preconditions. In order to ensure correct use the following could be added:-

```
print_menu
    require
        has_title --boolean variable to be set by new_menu
        menu_size /=0 --variable changed by add_menu
```

menu_complete -- boolean variable to be set by complete_menu

The comments supplied with the add_entry feature should be changed to reflect the real situation, that is to state that the use of a tag is essential if the print_menu feature is to be used.

All the changes use readily available facilities. If reuse of classes is to be a reality, these facilities should be fully exploited.

The poor quality of the documentation which accompanies supplied classes is not unique to Eiffel. D Esp [2] found the Smalltalk-80 library to be large and useful but inadequately documented.

This suggests that programmers need to be trained to document their work in a more complete way. This would not only make reuse easier but would also make it easier to make corrections to code written by another programmer.

3.5 Compiler error messages

The following list contains some of the less meaningful error messages that have been interpreted.

1. "is cannot be used as an identifier"

   really meant that an end statement was missing!

2. " ':= ' may not be used as identifier "

   occured when trying to change the value of an attribute outside its own class declaration but inside a client class. eg. changing the value of a point coordinate inside the interaction class in the GUIDED_TOUR example.

3.6 Warning

Several changes have been made since the Meyers' original book was published[1]. The main ones are:-

1. The library has been changed so the simple example (p72) does not work! Replace

   io.putstring_nl(" .....") with

   io.putstring (".....");

   io.new_line

2. The IF statement is the only conditional statement mentioned in the book. A new statement which is roughly equivalent to a case statement in Modula-2 or C has been introduced. This is the **inspect** statement. An example of this is in the code for the one_command feature in the class INTERACTION.

3. The SDF has also been changed.

There is a new book [3] but this relates to Eiffel version 3. There is conversion subsection to allow the comparison with previous versions.

References

[1] Bertrand Meyer. *Object-oriented Software Construction.* Prentice Hall, Hemel Hempstead, 1988.

[2] D.G.Esp. A beginners experience of smalltalk-80 for the evolutionary prototyping of an expert system. *IEE Colloquium on 'Applications and Experience of Object-Oriented Design'*, 18, 1991.

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