

Object Tools

<http://www.object-tools.com>

EiffelCocoa

Framework

Programming

Guide

“Think Different ... Think Eiffel!”

Powerful, flexible, simple, and elegant programming.

VERSION: 1.0

Revised: 31 December 2002

CONTENTS

1.	Welcome	2
1.1.	Why Eiffel for Cocoa?.....	3
2.	Creating a Cocoa application	8
3.	EiffelCocoa library organization	9
3.1.	The Cocoa Group	9
3.2.	The Common Group.....	11
3.3.	The Foundation Group	12
4.	The Application Kit Group	13
4.1.	User Interface Group	13
4.2.	Text Group	13
4.3.	Graphics Group	14
4.4.	Document Support Group.....	14
4.5.	Printing Group	14
4.6.	Pasteboards Group	14
4.7.	Operating System Services Group.....	14
5.	Cocoa to Eiffel Mapping	15
5.1.	Class Names	15
5.2.	Class Layout	15
5.3.	Feature Names	16
5.4.	Class Methods	17
5.5.	Scope	17
6.	The C/Objective-C Interface	18
6.1.	Cocoa calling Eiffel	18
7.	Status	21

1. WELCOME

Welcome to Object-Tools EiffelCocoa for OS X. EiffelCocoa provides a complete Eiffel interface to Apple's Objective-C Cocoa programming environment.

For full information on the Apple Cocoa framework, you should consult Apple's documentation. For information on creating, compiling, and debugging Eiffel projects in both ProjectBuilder and CodeWarrior, please consult the Eiffel for OS X manual. The EiffelCocoa library combines the strengths of Apple's Cocoa framework with the best of the Eiffel language including multiple inheritance, strong but convenient typing, generic and anchored types, and preconditions and postconditions to clearly express the expectations of the library interfaces.

If you have loaded this off a free CD, or from the Internet, you may use this under the Lite licence. With this licence you can use Eiffel for OS X on a trial basis. You should not use the Lite form for extended periods, and may not use it to release commercial software, or to deploy software for use of others within your company. The Lite licence limits the size of systems that you can compile, so once your application grows to a certain size, you will get a syntax error and not be able to generate code. We want to make it as easy as possible for you to use Eiffel, but in order to provide support and better future releases, we are dependent on your support.

Even if you do not have a licence, we will still support your suggestions and fix your problems, so please send them anyway. But for serious use, please purchase a licence, so we can provide the best support and most rapid updates.

The commercial licence entitles you to full compilation facilities, and to deploy and resell software created with Object Tools Eiffel for OS X. Contact Object-Tools at:

i.joyner@acm.org (Ian Joyner Australia)

or

fm@object-tools.com (Frieder Monninger Europe)

gfrank@object-tools.com (Gudrun Frank US)

to obtain a full licence.

Also see the Object Tools Web site at:

<http://www.object-tools.com>.

The full licence costs \$US149, which includes the full compiler, The Cocoa library, and the EiffelCarbon library. This is for a single seat licence. For site licences and University department licences, please contact Object-Tools. Full support is also not included with the Lite version, however, we will accept comments and problems that you encounter, especially since you will probably want any problems you have with your installation ironed out before you pay for a full licence.

If you do not have Internet access, please send International Money Order (personal cheques not accepted except in Australian dollars) to:

Object Tools
attn. Ian Joyner
14 Summerhaze Place
Hornsby Heights
Australia 2077
Phone +61 2 9477 3474

Object Tools
attn. Gudrun Frank
418 Parkview Way
Newtown, PA 18940
Phone 215-504 0854

Object Tools
attn. Frieder Monninger
Nordstr. 5
D 35619 Braunfels
Phone: 6472 911 030 Fax: 6472 911 031

Please enclose your name and address (email address) for us to return a full licensed version.

We hope that as it is, you can create some reasonable and exciting new Macintosh applications with EiffelCocoa. Our aim is to make building the most sophisticated Mac applications the easiest possible so that many new Mac applications can be quickly created and that the Mac can reestablish itself as the industry leader in innovations.

1.1. Why Eiffel for Cocoa?

Eiffel and Cocoa combine the best strengths of an industrial strength and sophisticated programming language and framework. Following are some

of the features of Eiffel with how they match well to Cocoa. As Anguish et al say in their book *Cocoa Programming*: “Using Cocoa is the most advanced and arguably the most productive way to program a Macintosh—it’s also the most fun.” Eiffel brings an added dimension of advancement, productivity, and fun.

Pure Object-orientation

Eiffel is an object-oriented language designed as an object-oriented language from the ground up, unlike C-based hybrids where much of the complexity and flaws of C must also be dealt with, leading to longer, more expensive development cycles. C was designed as a machine-oriented language, whereas Eiffel is designed as a problem-oriented language—the compiler knows that an object reference is an object reference and the programmer never needs be concerned with pointer referencing and dereferencing. The syntax of Eiffel is clean, easy to read, and brief.

Multiple inheritance

Eiffel provides full multiple inheritance of both interfaces and implementation. Because multiple inheritance has some bad implementations, it has been given an unfair bad reputation. Eiffel does not suffer from these problems—all conflicts are resolved within the inherit clause, and do not need to be addressed by special mechanisms in the code itself. Thus Eiffel’s, code remains clean and uncluttered by these mechanisms, and yet efficient.

Multiple inheritance of implementation is important—it means you can break your classes down into small pieces and place implementations where they can be accessed by many classes. In languages like Java, that only provide multiple inheritance of interface, default implementations must be provided by every class implementing the interface, or you have the complication of nested classes.

In Eiffel, the paradigm is simple—only classes and inheritance, there is no need for other concepts such as interfaces, protocols, categories, nested classes, enums, structs, etc.

Strong typing

Strong typing enables checking for as many errors in the code as possible at compile time. You will never get run-time errors due to “message not understood”. Where some strongly-typed languages might obstruct the programmer, Eiffel’s type system is very natural and does not have any need to resort to mechanisms such as casts for programmers to get around the inadequacies of the type system.

Another benefit of strong typing is run-time efficiency. First, Eiffel has a very efficient dispatch mechanism. After the first call on a routine, the time for dispatch is the same, and multiple inheritance also has no effect on effi-

ciency, with no searches through object tables to find methods. Second, many run-time checks that must be present in non-typed systems do not need to be generated in Eiffel, since the code has been proven correct at compile-time.

Correctness

Eiffel provides a mechanism to formally specify interfaces in the form of preconditions and postconditions. This mechanism has been used throughout the Eiffel for Cocoa library and turns Apple's statements about expectations on method arguments into run-time checkable code. Correct functioning of classes may also be monitored with the use of class invariants.

Efficiency

Eiffel generates machine code, using C as an intermediate language, so you may set optimizations that either GCC or the CodeWarrior C compilers use for the platform. Eiffel does not run on a virtual machine with inevitable slowness of this approach. As explained under strong typing, typing also results in more efficient generated code, both in dispatch mechanisms, and the reduced need for run-time checks. Eiffel's dispatch mechanisms are the implicit equivalent of Objective-C's explicit IMP mechanism. Unlike IMP, Eiffel's efficient dispatch always happens, is done in the background, and is never dangerous since polymorphism is not subverted.

Genericity

Data structures in Eiffel may be generic. For example there is one *ARRAY* class which holds elements of *INTEGERS*, *REALS*, or objects of any type. In languages without genericity, once elements are entered in a non-generic data structure, they lose their compile-time type information, and you must rely on run-time checks to ensure correctness.

Eiffel also provides constrained genericity, so that if you develop a data structure, you can restrict what kinds of objects may be passed into any instance of that data structure. This also enables you to make certain assumptions in the data structure itself about the capabilities of objects placed in it. For example, it can make routine calls on the entities without run-time checks and casts.

Anchored types

Anchored types allow the type of any feature declaration to be anchored to another feature using the *like other* construct. This allows subclasses to redefine the types of many features by just redefining the type of a single feature. This is very handy where a parent class has a routine to create an object of something of an as yet unknown type. All you need do is to anchor

the return type of the routine to another feature and the correct type will be created depending on the run-time type of the object. For example:

```

cell_type: CELL is deferred end

adopt_cell (objc_cell: POINTER): like cell_type is
do
    if objc_cell /= Nil then
        Result ?= from_cocoa.find (objc_cell)

        if Result = Void then
            create Result.adopt (objc_cell, True)
        end
    end
end -- adopt_cell

```

This is a pattern used in the delegate classes, so that for different kinds of *CELL* handled by subclasses, all the subclasses have to do is redefine the type of *cell_type*, the entire *adopt_cell* routine does not have to be redefined just to return the correct type.

Clean syntax

One example is in message passing. Where in Objective-C you have the complexity of Lisp-like nested brackets as in

```
[[[a b] c] d]
```

in Eiffel is:

```
a.b.c.d
```

and this becomes even more complicated with parameters:

```
[[[a b: [e f]] c] d]
```

is simply:

```
a.b (e.f).c.d
```

Also, Eiffel provides only the '.' operator to access everything, whereas C has redundancy between '.' and '->', reporting a syntax error if you use the wrong accessor syntax.

There are also no preprocessor directives to complicate the syntax as with # and @ directives.

Garbage collection

Programs you write in Eiffel for Cocoa are all garbage collected. You no longer have to be concerned with dangling pointers to objects that have

been mistakenly released, or memory leaks due to objects that have no references but have not been released.

The mechanisms for adopting Objective-C objects in Eiffel and for releasing them correctly are centralized in the *COCOA_ITEM* class.

One definition file

In Eiffel, the entire definition for a class is in the .e file. You do not have to separate interface and implementation in separate .h and .m files.

Wide availability

Eiffel is widely available, having several implementations for Windows, Unix, and .Net. It is also reasonably widely known, and even if you have not come across it before, it makes the most sophisticated aspects of OO more approachable than any other language. It is also well supported by a good range of books.

All these mechanisms add value to the already very sophisticated Cocoa programming environment. Eiffel provides the most approachable way to get into Cocoa programming and develop large-scale commercial applications.

2. CREATING A COCOA APPLICATION

You should use the provided examples as a basis. To create an application, create a class that inherits from *APPLICATION*:

```
class
    YOUR_APPLICATION
inherit
    APPLICATION
creation
    make

feature

    make is
        -- Force these classes to be included in compile:
        b: YOUR_CONTROLLER
    do
        application_make
    end
end -- class YOUR_APPLICATION
```

YOUR_APPLICATION and *make* are the class and routine referenced by the create clause in your *Project description.ep* file. All you should do here is to call *application_make*, which calls Cocoa's **NSApplicationMain** to get things going. After this Cocoa is in control and will call the appropriate routines in your program to handle events. This is why you must include a reference to at least one of the other Eiffel classes in the application class, so the Eiffel compiler knows it is used by your system.

The other classes you create will be some Cocoa glue classes which receive messages in response to events from Cocoa, and pure Eiffel classes which will be the bulk of your application, mainly implementing your application's model.

3. EIFFELCOCOA LIBRARY ORGANIZATION

The EiffelCocoa project is organized into groups. At this point, you will probably find it helpful to open up the Cocoa project in either Project-Builder or CodeWarrior. The main group is *Application Kit*, which contains most of the classes you will use, and this is the topic of the next section. The other groups, explained here, are *Cocoa*, containing high-level classes, *Common*, containing classes which are mainly abstract but do not fit into the hierarchy elsewhere, and *Foundation*, containing a few necessary classes from Apple's Foundation library.

3.1. The Cocoa Group

The Cocoa group contains some of the most basic classes used throughout Foundation and Application Kit. You will mainly be concerned with the *COCOA* and *COCOA_ITEM* classes.

COCOA_ITEM

The *COCOA_ITEM* class is the root class of all Cocoa framework classes corresponding to `NSObject` and `id`. *COCOA_ITEM* provides very few of the functions that `NSObject` provides—this is because they are mainly equivalents of the facilities that are inherited from the Eiffel *ANY* class. What *COCOA_ITEM* does provide is the basic connection between an Eiffel object and an Objective-C object. The mechanism for this is through the *item* attribute, containing a *POINTER* to the actual Objective-C object. It is this object that is passed to Cocoa to perform any methods on that object.

COCOA_ITEM also provides the interface between Eiffel's garbage collection and Objective-C's reference counting style of object disposal, ensuring that either an Eiffel object may be garbage collected after Cocoa deallocates the Cocoa object, or the corresponding Cocoa object is released when an Eiffel object is garbage collected.

Most of the classes that you create will be pure Eiffel classes and therefore not inherit from *COCOA_ITEM*. Only objects that must interface to Cocoa through an Objective-C object inherit from *COCOA_ITEM*.

COCOA

The *COCOA* class is inherited by all classes that need access to Cocoa facilities. These aren't necessarily Cocoa objects themselves, although *COCOA* is inherited by *COCOA_ITEM* itself, making these global facilities available to all Cocoa classes. You may also inherit *COCOA* from other classes to gain access to these facilities. Firstly, you can gain access to all the Cocoa facilities of your application through the *application* variable. Notice that this and many other attributes in this class are provided by Eiffel **once** functions. This means that only one copy is made and it does not take any room in all the objects that you create. Initialization is done only once, so after the first access, access is very fast.

COCOA provides a utility function *get_item* that accepts a *COCOA_ITEM* object and returns a Nil pointer if the Eiffel object is Void. This avoids having many if then else checks to avoid Void checks, which may become very nested if there is more than one such parameter being passed to Cocoa. It is used in the special circumstance where Cocoa will accept a Nil pointer. Normally, Objective-C objects are passed to Cocoa as follows:

```
w: WINDOW
do_something_to_a_window (w.item)
```

where *do_something_to_a_window* is declared as:

```
do_something_to_a_window (w: WINDOW) is
  require
    w /= Void
```

If there is no precondition that *w* must not be *Void*, then *do_something_to_a_window* should be called as:

```
do_something_to_a_window (get_item (w))
```

This normally reflects a comment in the Cocoa documentation that the argument passed in at this point may be Nil. Hence EiffelCocoa uses Eiffel assertions to reflect the Cocoa documentation in code so you don't have to double check the Cocoa documentation all the time.

Next are some utility create routines, *point_make*, *range_make*, *size_make*, and *rectangle_make*. These are a convenience to create these entities and pass them as parameters in a single line, as used in the *DELEGATE* classes.

set_ routines allow values to be passed back to Objective-C, but should be rarely needed, probably only where the value of a user interface field needs to be set from Eiffel directly.

The next set of routines gives access to the main modules of Cocoa. These are mainly where access to the class methods of Objective-C can be found, as well as other module specific utility routines.

Other routines accessing the mouse location, system time, and the global Nil pointer are also in here for convenient access. If you need access to any of these facilities, just inherit *COCOA* along with other classes you need to inherit. Multiple inheritance allows this clean approach without the need to access the facilities through an intermediate dummy object.

Taking possession of a Cocoa object is also enabled here through the *from_cocoa* attribute. To adopt a Cocoa object in an Eiffel class, inherit from *COCOA* and use the pattern:

```
from_cocoa.<type>.adopt (<pointer to obj-C object>)
```

where *<type>* is one of the routines in the *ADOPTERS* class.

3.2. The Common Group

The Common group contains the aforementioned *ADOPTERS* class and its paired generic class *ADOPTER*. These provide the connection between an Objective-C object and its Eiffel wrapper object. When an Objective-C object reference is passed into Eiffel, you should use the *from_cocoa* pattern as given above. This searches for the Eiffel wrapper, or creates it if it does not already exist. Aside from the adopter classes, there are several other top-level classes, described as follows.

DELEGATE

The *DELEGATE* class is the root class of all Eiffel object delegates. These are split into an interface class in Eiffel, providing the routine interfaces described in the Cocoa documentation for a class. The delegate classes receive the raw data from Cocoa and massage this into the Eiffel objects by adopting or calling a *_make* function and then call the delegate routine. You will provide a subclass of a delegate class to implement the delegate routines you need.

ENUMERATION

The *ENUMERATION* class is the root class for all C enums. They are all written with the pattern:

expanded class

```
COMPARISON_RESULT
```

feature

```
ascending: INTEGER is -1
```

```
same: INTEGER is 0
```

```
descending: INTEGER is 1
```

```
valid (order: INTEGER): BOOLEAN is
```

```

do
    Result := order = ascending or else
            order = same or else
            order = descending
end -- valid
end -- class COMPARISON_RESULT

```

where the routine *valid* is used in preconditions and postconditions. You should only use the values provided in these classes when calling a routine expecting one of the values as a parameter, not guess at the corresponding *INTEGER* or *STRING* (although that will work).

Note also that the corresponding value names in the Cocoa enums are global and hence very long, for example, `NSComparisonResultAscending`. The Eiffel *ENUMERATION* classes have removed this redundancy and hence the corresponding value name is simply *ascending*.

3.3. The Foundation Group

The Foundation group mainly provides classes that are involved in programming the model part of the MVC paradigm. The main idea of Eiffel for Cocoa is to program your application's model in Eiffel. It therefore makes sense to use the classes from the Eiffel libraries to program your models, not the Foundation classes, since they make better use of Eiffel's advanced concepts and provide a more seamless set of facilities when used in Eiffel. A small number of Foundation classes are used as parameters to the Application Kit. Thus the Foundation group provides some of Foundation's classes, but only the ones that are necessary to pass as parameters to the Cocoa. Also only necessary routines are provided, not the full interfaces of these classes.

NOTE: Eiffel has its own equivalents of the Foundation classes in the Eiffel Kernel library. These Eiffel classes should be used in your programs, not Apple's Foundation classes. For example, use Eiffel's *STRING*, not the Foundation equivalent *COCOA_STRING*, since it is more efficient because you will not be creating two objects, one Eiffel and one Objective-C and interfacing between them. *COCOA_STRING* is the equivalent of *NSString*, but since the class name *STRING* is already taken by the Eiffel *STRING* class, it is prefixed with *COCOA_*. A few other classes have also been prefixed with *COCOA_* to avoid a clash. Only use the Foundation classes to interface to Cocoa.

The group is organized according to the diagram in Apple's Introduction to the Foundation Framework.

4. THE APPLICATION KIT GROUP

The Application Group contains classes that mainly address the View part of the MVC paradigm. As with the Foundation Group, the Application Kit group is laid out following the diagram in Apple's Introduction to Application Kit, but with a little flattening out of the groups to make them easier to access. Also a few of the classes have been moved into conceptually related groups, rather than strictly following the inheritance hierarchy as in Apple's diagram. For instance, you will find the class *MOVIE_VIEW* in the Movies group, not in the Views group.

Also, unlike Foundation, the classes in the Application group are designed to be complete interfaces to Cocoa classes, since they provide higher-level functionality.

Within each group, you will generally find a 'group class'. For example, the Controls group has a class *CONTROLS* which contains various class methods. In this particular case, the class methods that relate to control colors from *NSColor* have been placed in here. There are similar color routines in the *TEXTS* class, and the names have been made consistent between these classes.

4.1. User Interface Group

The major group in the Application Kit is User Interface, since most classes in the Application Kit support user interface items such as Cells, Views, Controls, and Windows. Supporting groups which are not strictly user interface items, such as Text, Graphics, Documents, etc., are groups on their own.

In here, you will find the major classes *RESPONDER* and *APPLICATION*. Also the major user interface elements can be found in the subgroups Events, Cells, Controls, Windows. Panels, Browsers, Tables, Tabs, Menus, and Toolbars.

4.2. Text Group

The text group contains the subgroups Fonts and International Character Input, as well as all the classes needed to interface to Cocoa's text system.

4.3. Graphics Group

The graphics group contains the subgroups Images, Open GL, Movies, Colors, and all the classes to interface to Cocoa's text facilities.

4.4. Document Support Group

The documents support group contains classes to support document based applications.

4.5. Printing Group

The text group contains classes to support advanced features of printing.

4.6. Pasteboards Group

The text group contains classes for pasteboard handling. In the *PASTEBOARDS* class (accessed via *pasteboards* in the *COCOA* class) you can find access to shared singleton pasteboards. You can also create your own pasteboards. This group redesigns Cocoa's pasteboard concept a little, giving one class per pasteboard type. However, these are still only a direct access to the Cocoa pasteboard.

4.7. Operating System Services Group

The operating system services group contains the file system subgroup as well as a collection of classes supporting sound, help, spelling, etc.

5. COCOA TO EIFFEL MAPPING

This chapter explains some of the practical considerations in mapping Cocoa to Eiffel. As explained in the previous sections, the Foundation and Application Kit classes have been mainly laid out according to Apple's diagrams in the introduction documents, with some notable exceptions to make cleaner groupings and to ease navigation.

5.1. Class Names

All class names are equivalent to the Cocoa names, except 'NS' has been dropped. By Eiffel convention, all class names are in upper case with '_' denoting word breaks. The upper case convention follows the mathematical convention for types.

5.2. Class Layout

All libraries in this release of Eiffel for OS X use a similar layout, particularly using informative text in Eiffel's **Indexing** clause. Eiffel's indexing clause is optional and reasonably free form, but the convention we have chosen to use is as in the following example:

Indexing

Project: "Eiffel/OS X/Cocoa/Application Kit/User Interface/
Views/VIEW"

File: "View.e"

Description: "Glue class to NSView."

Cocoa_class: NSView

Author: "Ian Joyner"

Copyright: "© 2002 Ian Joyner, Harmonic Systems"

Version: 1.0

Date: "Monday 16th September 2002"

Keywords: view, superview, subview

The project item gives the group path to the current class. Information is given in the Description and Cocoa_class items as to which Cocoa entity this class maps. Keywords give some words which may in future be useful in a search engine, as over the Internet.

The features in a class are grouped according to Apple's documentation. Under the 'Method Types' section in the documentation for each class, the methods are functionally grouped under a heading. Eiffel classes follow this grouping, making use of the **feature** clause. Both Eiffel and the documentation usually put creation or initialization routines as the first group, and Eiffel usually uses the name *make* or *make_something* for creation routines. Such groups are flagged by a line such as:

```
feature {ANY} -- Key equivalents.
```

At the end of each class are a group of features named *platform_something*, which declare the external interface to the Objective-C functions. These features are not exported (exported to *NONE*).

5.3. Feature Names

The feature names are modifications of the Cocoa names, but are more consistent with Eiffel reasoning on names than Cocoa names. This is because Eiffel names are extremely consistent: for example, in the data structure classes data items are generally added by a '*put*' routine, even in a *STACK*, where it would usually be *push*. This makes names in Eiffel very easy to remember, and also aids in the classification for inheritance hierarchies, since the same name occurring in many classes is easy to see.

Secondly, whereas the Java implementation has followed Objective-C's convention of having very long names with parameters included in the names. For example *POP_UP_BUTTON_CELL* uses the feature name *insert_item* for *insertItemWithTitle:atIndex* in Objective-C and *insertItemAtIndex* in Java.

In the above case, the argument names which are tacked onto the Objective-C name become the names of the Eiffel arguments.

In cases where the Eiffel name has changed significantly to satisfy the requirements of consistency, brevity, and clarity, the original Objective-C name is included in the feature header comment, so that it is easy to cross reference to the Cocoa documentation, or to find the Eiffel feature from the Cocoa name:

```
flush is
    -- - flushGraphics
do
...

```

where this routine comes from the class *GRAPHICS_CONTEXT*, so the 'graphics' is implicit in the context.

Eiffel also uses the convention of '_' to denote word breaks, where white space would be used in natural language. This is for readability. Even though we have used shorter names in Eiffel, long names with an uppercase letter denoting the beginning of a new word are difficult to read. Histori-

cally, the use of white space to insert a break between words is a quite recent phenomenon. Words used to be run together, but about 700 years ago the white space was introduced, and monks which could not sight read songs before were suddenly able to do so. The simple white space to delimit written words also made reading much more accessible for the masses, and literacy rates improved.

5.4. Class Methods

Class methods are mostly implemented in the group class. Some class methods that return invariant data are implemented by Eiffel **once** routines. These are singleton or shared objects.

5.5. Scope

Objective-C provides three levels of scope for instance variables: `@public`, `@protected`, and `@private`. It is recommended not to make variables `@public` since they can then be changed beyond the scope of the class, nor to use `@private` since this restricts possible legitimate access in subclasses. That means that `@protected` covers most scopes. In Eiffel, scope is much more fine grained, `@protected` being broken up into multi-levels so you can design cooperating classes by use of the export mechanism, where $\{A, B, C\}$ covering a set of features (including constants, attributes, and routines), means that set of features is only exported to the related classes *A*, *B*, and *C*, and their subclasses.

`@public` is covered by the $\{ANY\}$ case, where the set of features is exported to the root class *ANY* and all its subclasses. However, this differs from `@public` since attributes remain read-only.

$\{NONE\}$ is the closest equivalent to `@private` meaning the features are exported to the *NONE* class, which notionally exists at the bottom of the inheritance hierarchy. However, since it rarely if ever makes sense to restrict visibility of inherited features in subclasses, $\{NONE\}$ does not restrict visibility in subclasses, only client classes.

Also, the Objective-C scheme only applies to instance variables, but not to methods, a trick with categories being used to make methods virtually private. Eiffel's mechanism applies to all features of a class.

6. THE C/OBJECTIVE-C INTERFACE

In each group, you will find a .m file. These files provide the mechanism to actually perform the calls from Eiffel on the Cocoa objects. NOTE: You should never have to be concerned with what is in these files. These mechanisms are completely hidden behind the Eiffel interfaces that you will use. In fact, they could easily change in future releases.

In the Cocoa group, you will find the Cocoa_Lib.h and Cocoa_Lib.m. The Cocoa_Lib.h file provides many defines which provide the interfaces to Cocoa, and you will not really need to be concerned with the details in this file.

6.1. Cocoa calling Eiffel

One of the main paradigms of an object-oriented framework is that the framework does all the event handling and calls your code to do the specific work in response to an event. Cocoa naturally calls Objective-C objects, and you need to provide small glue Objective-C objects in order to pass the calls onto Eiffel.

Fortunately, Eiffel provides a mechanism to do this called Cecil (C-Eiffel Call-In Library). Cecil is quite complicated in its mechanisms so these are simplified to what is required to integrate Eiffel and Cocoa. All the defines needed for this is contained in the header file Cocoa.h, and you will probably have to be somewhat familiar with these facilities. An example Objective-C class definition looks as follows:

```
#if defined (__MWERKS__)
#include <EiffelCocoa.h>
#else
#include <EiffelCocoa/Cocoa.h>
#endif

@interface ApplicationController: NSObject
{
@protected
eiffel_object;
IBOutlet NSBrowser *fsBrowser;
IBOutlet NSImageView *nodeIconWell; // Image well showing
the selected items icon.
IBOutlet NSTextField *nodeInspector; // Text field showing
the selected items attributes.
```

```
}

// Force a reload of the data.

- (IBAction)reloadData:(id)sender;

// Methods sent to us from the browser.

- (IBAction)browserSingleClick:(id)sender;
- (IBAction)browserDoubleClick:(id)sender;

@end

#import "AppController.h"
#import "FSBrowserCell.h"

@implementation AppController

cocoa_action_proc (reloadData, ereload_data)

EIF_INTEGER_FN erow_count;

// Use lazy initialization, since we don't want to touch the file
system too much.

- (int)browser:(NSBrowser *)sender numberOfRowsInColumn:(int)column
{
return call_eiffel_integer (erow_count)(Current, sender, column);
}

EIF_PROC ewill_display;

- (void)browser:(NSBrowser *)sender willDisplayCell:(id)cell
atRow:(int)row column:(int)column
{
call_eiffel_proc (ewill_display)(Current, sender, cell, row, column);
}

-(void) awakeFromNib
{
EIF_PROC make;
eiffel_create ("BROWSER_CONTROLLER");

// Get references to the routines we will call.

link_eiffel_proc (make, "make");
link_eiffel_proc (ereload_data, "reload_data");
link_eiffel_integer (erow_count, "row_count_from_cocoa");
link_eiffel_proc (ewill_display, "will_display_from_cocoa");
link_eiffel_proc (ebrowser_single_click,
"browser_single_click");
link_eiffel_proc (ebrowser_double_click,
"browser_double_click");

// Tell the browser to send us messages when it is clicked.
```

```
[fsBrowser setTarget: self];
[fsBrowser setAction: @selector(browserSingleClick:)];
[fsBrowser setDoubleAction: @selector(browserDoubleClick:)];

call_eiffel_proc (make)(Current, self, fsBrowser, [FSBrowserCell
class]);
}

cocoa_dealloc_proc
@end
```

In the class interface, include the line `eiffel_object` to set up a link to the Eiffel object. This should be in the `@protected` section since you want it accessible in subclasses, but not client classes.

In the implementation file, InterfaceBuilder action methods are linked to their Eiffel counterpart using the `cocoa_action_proc` define.

Each Eiffel routine that is called must first be linked. The natural place to do this is when the Cocoa object is initialized. You will notice above the `link_eiffel_` calls. There is one of these calls for each Eiffel type that can be returned and procedure calls which do not return anything.

`awakeFromNib` is a natural place to perform this linking when the object is created in InterfaceBuilder. However, some objects are created from within Eiffel. In this case an initialization routine is called, and the linking is performed on the first call only for the whole class.

7. STATUS

As of January 2003, Eiffel for Cocoa is still in early release cycle. While all care has been taken in creating the interfaces, since there are many thousands of them, it is impossible to exercise them all from a small set of test applications.

While there is a small amount of Objective-C to write to marshall messages from Cocoa to Eiffel, in the future we hope to minimize or remove the need for this.

Also, we have made pre- and post-conditions very tight. Since some values are passed in from Cocoa, and the documentation is not specific about allowed Nil values, or maybe integers less than zero, etc, exceptions may be generated. If you find this, please remove the assertion and report it to us.

If you find any problems, please let us know since it is our intention to have a rock solid release and to continue to improve being able to program the best framework with the best object-oriented language.
