Chair of
Software Engineering

Objective-C Frameworks to Eiffel Converter

Master Thesis

By: Matteo Cortonesi
Supervised by: Benjamin Morandi
Prof. Dr. Bertrand Meyer

Student Number: 05-917-455

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

EiffelVision [2] is an object-oriented framework for graphical user interface development, originally created by Eiffel Software. It is designed to be platform independent but it is not implemented for Mac OS X (although a GTK+/X11 based version exists). A necessary step towards this direction is the porting of Cocoa, an Apple framework used to create Mac OS X native applications, to Eiffel.

This led to the development of an automated tool that converts Objective-C frameworks to Eiffel.

Acknowledgments

I would like to thank my mentor, Prof. Bertrand Meyer, for giving me the opportunity to work in his company, Eiffel Software, in Santa Barbara (California, USA) where I spent a very pleasant time.

I would also like to say a special thanks to Emmanuel Stapf who hosted me for the first week in Santa Barbara. In addition, his support during the design, initial development and technical issues has been extraordinary. I enjoyed very much working and discussing with him.

Thanks to my supervisor Benjamin Morandi too for his continuous support and valuable feedback.

Many thanks also to Raphael Meyer, Annie Meyer, Isabelle Meyer for the support and Ian King for technical help.

Last but not least, thanks to my family and friends who supported me during my whole time at ETH.

Contents

1 Introduction **6**
1.1 Goal *6*
1.2 Related Work *6*
1.3 Outline *7*
2 Objective-C Overview **7**
2.1 Instance Methods *7*
2.2 Class Methods *8*
2.3 Declaring A Class Interface *8*
2.4 Creating Objects *9*
2.5 Memory Management *9*
2.6 Class Clusters *10*
3 Tool Architecture **10**
4 Parser **11**
5 Semantic Analyzer **11**
6 Code Generation **13**
6.1 Objective-C Identifiers *13*
6.2 C Structs *14*
6.2.1 Basic Type Fields *14*
6.2.2 Struct Fields *15*
6.2.3 Structs Comparison *17*
6.3 Objective-C Classes *17*
6.3.1 Mapping Of Objective-C Method Names *17*
6.3.2 Introducing The Top Level Classes *18*
6.3.3 Mapping Inheritance *18*
6.3.4 Wrapping Objective-C Objects *19*
6.3.5 Memory Management Invariant *19*
6.3.6 Mapping Procedures *19*
6.3.7 Mapping Queries With Expanded Return Type *20*
6.3.8 Mapping Queries With Structs As Return Type *21*
6.3.9 Mapping Queries With Objects As Return Type *22*
6.3.10 Mapping The Objective-C Class Type *33*
6.3.11 Mapping Objective-C Categories (Methods Grouping) *36*
6.3.12 Mapping Class Methods *36*
6.3.13 Object Creation *36*
6.3.14 Subclassing *40*
6.3.15 Callbacks *44*
6.4 Objective-C Categories *54*
6.5 Objective-C Protocols *55*
6.5.1 Introduction *55*
6.5.2 Mapping Objective-C Protocols *56*
6.5.3 Name Clashes *57*
7 Developers Guide **57**
8 Users Guide **57**
8.1 Wrapper Generator *58*
8.2 Generating A Cocoa Wrapper *58*

8.3 Using The Generated Cocoa Framework 58
9 Conclusion **59**
10 Future Work **60**
11 References **60**

1 Introduction

EiffelVision is an objective-oriented framework for graphical user interface development, originally created by Eiffel Software.

> The EiffelVision library offers an object-oriented framework for graphical user interface (GUI) development. Using EiffelVision, developers can access all necessary GUI components, called widgets (buttons, windows, list views) as well as truly graphical elements such as points, lines, arcs, polygons and the like – to develop a modern, functional and good-looking graphical interactive application.
>
> EiffelVision has played a major role at Eiffel Software and provided numerous Eiffel projects with a powerful, portable graphics development platform. EiffelStudio is totally reliant on EiffelVision for its graphical elements and overall interaction with the user.
>
> *Eiffel Software*

The existing implementation of EiffelVision for Mac OS X is based on GTK+ and runs under the X11 application. Therefore, it does not integrate well with the Mac OS X platform. The interface does not use the look and feel of Mac OS X, aqua, and many of the Mac OS X services are not available (standard keyboard shortcuts, file associations, icon in the dock, standard dialogs to open/save files, etc.). In addition, because GTK+ depends on a lot of other packages, we are not able to provide a simple binary installation package for EiffelStudio on Mac OS X.

1.1 Goal

Initially, the goal was to start from the existing work done by Daniel Furrer [6], to produce a final and solid Cocoa framework for Eiffel and, if time allowed it, completing the Eiffel Vision implementation.

Yet we quickly realized this was not an optimal choice. Not only we want to provide a solid Cocoa framework, we also want it to be maintainable and extensible such that it will not fall into disuse (just like what happened with other projects, see chapter 1.2 Related Work). This is especially true because Apple updates the frameworks every once in 1 or 2 years.

Hence, we decided to take a step back and aim at a higher goal: engineering a tool that automatically converts full Objective-C frameworks to Eiffel. This will allow an automated update of the Eiffel Cocoa wrapper with little to no user intervention. Likewise, it will allow the conversion of other Mac OS X or iOS (the OS used by iPhones, iPod Touches and iPads) frameworks.

Coming up with an Eiffel version of the Cocoa framework is probably the hardest step. EiffelVision can be implemented on top of it more easily.

1.2 Related Work

In the past, there have been a few attempts to make EiffelVision run on the Mac (see Vision4Mac [4], EiffelCocoa [5]). However, none of these projects has been completed and they have not been updated since long time.

A more recent attempt [6] has been made by Daniel Furrer. This, however, was not completely finished and further work remains to be done. For example, only about 20% of the 800 Cocoa classes are (partially) implemented. What is more, it has huge memory leaks because of no working memory management for the Objective-C objects, it is arduous to use and it does not provide a seamless developing experience.

1.3 Outline

In chapter 2 we give a brief description of the Objective-C programming language. This is only an informal, minimal overview of the language. It is not meant to be a full introduction to Objective-C. We recommend the reader unfamiliar with it to read Apple documentation [7] and [8].

In chapter 3 we present an overview of the architecture of the tool.

Chapters 4 and 5 discuss the parsing and semantic analyzation done by the tool.

In chapter 6 about code generation we describe in detail how we generate Eiffel classes to wrap Objective-C frameworks. This is the central chapter of this document.

Chapters 7 and 8 are guides for developers and users, respectively.

We conclude in chapter 9 and give an idea for future work in chapter 10.

Chapter 11 is a list of references.

2 Objective-C Overview

Before describing the phases of the wrapper generator in detail, we will give a brief overview of the Objective-C programming language used in Mac OS X and iOS. For a better and detailed description read [7].

2.1 Instance Methods

Instance methods are methods that can be called on instances of a class.

The basic syntax for calling a method on an object is this:

```
[anObject method];
[anObject methodWithArgument:anArgument];
```

Listing 1: calling an instance a method on an object.

Methods can return a value.

```
result = [anObject method];
result = [anObject methodWithArgument:anArgument];
```

Listing 2: instance methods returning a value.

Methods can take multiple arguments. In Objective-C, a method name can be split up into several segments. In the header, a multi-argument method looks like this:

```
- (NSString *)stringByReplacingOccurrencesOfString:(NSString *)target withString:
(NSString *)replacement;
```

Listing 3: declaration of a multi-argument instance method.

To call the method we write:

```
newString = [aString stringByReplacingOccurrencesOfString:@"PC" withString:@"Mac"];
```

Listing 4: calling a multi-argument instance method.

Note that these are not just named arguments. The method name is actually `stringByReplacingOccurrencesOfString:withString:` in the runtime system.

2.2 Class Methods

Methods can be called on classes too. These are called class methods.

```
NSArray *array = [NSArray arrayWithObject:anObject];
```

Listing 5: calling a method on a class object.

Class methods are sometimes used to create objects. In the header, their declaration starts with a plus before their name (compare with instance method in listing 3).

```
+ (NSArray *)arrayWithObject:(id)anObject;
```

Listing 6: declaration of a class method.

2.3 Declaring A Class Interface

The syntax for declaring a class is the following.

```
#import <Cocoa/Cocoa.h>

@interface Person : NSObject {
    NSString *firstName;
    NSString *lastName;
    NSInteger age;
}

@end
```

Listing 7: declaration of a class with some instance variables.

First, we import `Cocoa.h` which declares all the (public) classes for a Cocoa application (e.g. `NSObject`, the Cocoa root object).

Next, we declare a class named `Person` inheriting from `NSObject` with 3 fields: `firstName`, `lastName` and `age`.

Listing 8 shows how to add instance methods (`firstName`, `lastName`, `isAdult` and `setAge`).

```
#import <Cocoa/Cocoa.h>

@interface Person : NSObject {
    NSString *firstName;
    NSString *lastName;
    NSInteger age;
}

- (NSString *)firstName;
- (NSString *)lastName;
- (BOOL)isAdult;
- (void)setAge:(NSInteger)anAge;

@end
```

Listing 8: declaration of a class with instance variables and instance methods.

2.4 Creating Objects

There are two main ways to create an object. The first one is using class methods as we saw before in listing 5. The second one consists in two method invocations, `alloc` and an initializer (usually `init`) as shown in listing 9.

```
NSString *string = [[NSString alloc] init];
```

Listing 9: creating a `NSString` object.

The code in listing 9 shows a nested method call. `alloc` is a low-level method that allocates memory for the object and sets up its structure such that it can be used in the Objective-C runtime, while `init` performs some basic initialization. An initializer can also take several arguments.

2.5 Memory Management

In Objective-C you can either choose between programming with or without garbage collection. In some operating systems, e.g. iOS, garbage collection is not supported. Therefore we have chosen not to use it. The other option is reference counting. Cocoa provides facilities to abstract this task, namely the methods `retain` and `release`. `retain` increases the retain count by 1, while `release` decreases it by 1; if the retain count reaches 0 the `dealloc` method of the object will get called and the object will get deallocated.

Memory management in Objective-C is based on a single simple rule, i.e. *if you own an object, you must release it when you're done with it*.

You own an object whenever:

1) you retain the object,

2) you receive a reference to it by a function whose name

- starts with the keywords `alloc` or `new`,
- contains the keyword `copy`.

This definition does not exclude multiple ownership, i.e. an object can be owned by multiple clients.

2.6 Class Clusters

It is worth mentioning a design pattern used extensively in the Mac OS X Foundation framework because it will be relevant for later. The Apple documentation states the following.

> Class clusters group a number of private, concrete subclasses under a public, abstract superclass. The grouping of classes in this way simplifies the publicly visible architecture of an object-oriented framework without reducing its functional richness.
>
> [...]
>
> The abstract superclass in a class cluster must declare methods for creating instances of its private subclasses. It's the superclass's responsibility to dispense an object of the proper subclass based on the creation method that you invoke—you don't, and can't, choose the class of the instance.

Let's consider the `NSNumber` class cluster as an example.

```
NSNumber *aChar = [NSNumber numberWithChar:'a'];
NSNumber *anInt = [NSNumber numberWithInt:1];
NSNumber *aFloat = [NSNumber numberWithFloat:1.0];
NSNumber *aDouble = [NSNumber numberWithDouble:1.0];
```

Listing 10: instantiating objects of a class cluster.

Each returned object – `aChar`, `aInt`, `aFloat` and `aDouble` – may belong to a different private subclass (and in fact it does). The header files of the private subclasses are not available.

3 Tool Architecture

As stated in chapter 1.1, goal of the tool is to convert Objective-C frameworks to Eiffel.

Objective-C frameworks are stored in */System/Library/Frameworks* and they consist of a folder containing several header files that define symbols, declare classes and other Objective-C entities.

Figure 1: Conversion workflow.

As shown in Figure 1, the tool takes a set of framework files contained in a framework folder (not necessarily Cocoa) as an input and starts parsing the Objective-C code with a custom parser we built. The parser outputs a forest of Objective-C trees. These are then fed into a semantic analyzer that will add information to the trees. Finally, the Objective-C to Eiffel compiler will generate the Eiffel classes representing the Objective-C entities. This is the most crucial and complicated step.

4 Parser

We did not opt for a full-fledged parser that requires a grammar for Objective-C because it was too complex and time-consuming for the available time.

In particular, we didn't use Gobo because a ready to use Objective-C grammar for Geyacc was missing. Moreover, the part we were interested in was just a small percentage of the full grammar. This would have caused an overhead for the understanding of the tools and issues for the tweaking of the grammar.

For this reason, we decided to build a custom parser from scratch.

The parser is able to parse class interfaces, categories, protocols, instance- and class methods declarations and properties – all the typical Objective-C entities.

It is also able to correctly parse header files that contain preprocessor statements (such as `#if`, `#define`, etc.), structs, typedefs and enums.

5 Semantic Analyzer

The code generator needs to be able to distinguish between several types. E.g.

- Pointers to instance objects,
- pointers to structs,
- structs,
- basic types (enums included),
- pointers to Objective-C selectors,
- pointers to Objective-C class objects,
- function pointers,

- Objective-C blocks.

These types will be treated differently when they are returned or passed to a function. E.g. an integer is passed by value, whereas an object is passed by reference.

This is extremely complicated to achieve by simply parsing. Therefore, we implemented a semantic analyzer too.

In order to find out about the semantic value of a type we use Objective-C type encodings, a character string identifying types such as basic types (`int`s, etc.), pointers, tagged structures, unions or class names – any type, in fact, that can be used as an argument to the C `sizeof()` operator. To get the type encoding of a specific type we use the `@encode(type)` compiler directive (see [8] for more information). The type encodings are summarized in Table 1.

Code	Meaning
`c`	A `char`
`i`	An `int`
`s`	A `short`
`l`	A `long`, `l` is treated as a 32-bit quantity on 64-bit programs.
`q`	A `long long`
`C`	An `unsigned char`
`I`	An `unsigned int`
`S`	An `unsigned short`
`L`	An `unsigned long`
`Q`	An `unsigned long long`
`f`	A `float`
`d`	A `double`
`B`	A C++ `bool` or a C99 `_Bool`
`v`	A `void`
`*`	A character string (`char *`)
`@`	An object (whether statically typed or typed `id`)
`#`	A class object (`Class`)
`:`	A method selector (`SEL`)
[array type]	An array
{name=type...}	A structure

Code	Meaning
(name=type...)	A union
bnum	A bit field of *num* bits
^type	A pointer to *type*
?	An unknown type (among other things, this code is used for function pointers)

Table 1: Objective-C Type Encodings.

In order to resolve the type encodings, we first parse the entire system of header files collecting all the types we need to encode. Next, we generate and compile an Objective-C file with all the encode statements with the types we parsed as argument and we load the executable as a dynamic library in the tool to read the type encodings. After this step, we know the type encoding of each type name. This is particular useful with structs, because the type encoding gives information about the struct fields too. For instance, `@encode(CGRect)` would return the string "`{CGRect={CGPoint=dd}{CGSize=dd}}`", i.e. a struct named `CGRect` with two fields of type `CGPoint` and `CGSize` respectively, both of which have two fields of type double (type encoding *d*). `CGRect` is in fact declared as follows:

```
typdef struct CGRect {
    CGPoint origin;
    CGSize size;
} CGRect;
```

Listing 11: declaration of type definition for `CGRect`.

6 Code Generation

As explained in chapter 3, the responsibility of the code generator is to convert the parsed Objective-C framework into Eiffel classes. The main goal here is to end up with a generated Eiffel code that provides a seamless user experience to the developer that will be using it. In particular, we don't want the user to worry about the Objective-C memory management.

A fundamental problem is the mapping of Objective-C entities to Eiffel. In the following chapters we will introduce all the Objective-C entities and present their mapping to Eiffel.

6.1 Objective-C Identifiers

Class/struct names and identifiers in Objective-C use the camel case naming convention. Identifiers are mapped to names following the underscore convention. Class/struct names, in addition, are capitalized. Table 2 shows examples.

	Objective-C Example	Eiffel Example
Class name	`MyClass`	`MY_CLASS`
Struct	`MyStruct`	`MY_STRUCT`
Identifier	`anArgument`	`an_argument`

Table 2: Class/struct names and identifiers mapping examples.

6.2 C Structs

C structs are not classes in Objective-C. In Eiffel, however, they are mapped to classes. Every mapped struct inherits from `MEMORY_STRUCTURE`. This class defines several facility features that we use to wrap the C struct. In particular:

- an attribute `item` of type `POINTER` that points to the C struct,
- a `structure_size` feature returning an `INTEGER` that specifies the size of the memory allocated pointed by `item` in bytes. This feature is redefined in our wrapper class. We can find out about the size of a C struct with the `sizeof()` operator,
- a creation procedure `make` that automatically allocates `structure_size` bytes and sets `item` to point to that allocated memory. When the Eiffel object is disposed the memory pointed by `item` is freed automatically.
- a creation procedure `make_by_pointer` that simply initializes `item` with the passed argument. The memory pointed by `item` is shared and it is not freed when the Eiffel object is disposed.

For each C struct field we generate an Eiffel getter and setter (exported to every client) and private externals to get and set the values of the C struct. We consider, for example, a struct field named `field`.

We support two kinds of field types:

1) Basic types (`int`s, `float`s, etc.)

2) C structs.

6.2.1 Basic Type Fields

We consider a struct tagged as `MyStruct`. Let `field` be a `MyStruct` field and let its type be `BASIC_TYPE` (e.g. an `int`, `float`, etc.). We generate a private getter and setter (we use the *c_* prefix to denote these are C externals and to distinguish them from the actual getter and setter) as shown in Listing 12.

```
feature {NONE} -- Implementation

c_field (a_struct_pointer: POINTER): BASIC_TYPE
        -- Return the field value.
    external
        "C inline use <Cocoa/Cocoa.h>"
    alias
        "return (((MyStruct *) $a_struct_pointer)->field);"
    end

c_set_field (a_struct_pointer: POINTER; a_c_field: BASIC_TYPE)
        -- Set the corresponding C struct field with `a_c_field'.
    external
        "C inline use <Cocoa/Cocoa.h>"
    alias
        "((MyStruct *) $a_struct_pointer)->field = $a_c_field;"
    end
```

Listing 12: private getter and setter for `field`.

In the getter we use the passed pointer to a `MyStruct` struct to access the field and return it with the usual C syntax. In the setter we do pretty much the same thing except that we set the field with the passed argument.

Next, we generate the actual getter and setter exported to every client. This features simply call the externals we described earlier in Listing 11 passing `item` as an argument (see listing 13).

```
field: BASIC_TYPE assign set_field
        -- Return the struct field.
    do
        Result := c_field (item)
    end

set_field (a_field: BASIC_TYPE)
        -- Set `field' with 'a_field'.
    do
        c_set_field (item, a_field)
    ensure
        field_set: field ~ a_field
    end
```

Listing 13: exported getter and setter for `field`.

6.2.2 Struct Fields

We now consider fields that are structs themselves. Let `MyStruct` be a struct with a field named `field`. Furthermore, let `field` be of type `MyStruct2`, an other struct. We first consider the case of setting the field. Listing 14 shows the private external and the exported setter.

```
feature -- Setters

set_field (a_origin: MY_STRUCT_2)
        -- Set `field' with 'a_field'.
    do
        c_set_field (item, a_field.item)
    ensure
        field_set: field ~ a_field
    end

feature {NONE} -- Implementation

c_set_field (a_struct_pointer: POINTER; a_c_field: POINTER)
        -- Set the corresponding C struct field with `a_c_field'.
    external
        "C inline use <Cocoa/Cocoa.h>"
    alias
        "((MyStruct *) $a_struct_pointer)->field = *((MyStruct2 *) $a_c_field);"
    end
```

Listing 14: private and public setters for `field`.

The external simply takes the struct pointed by the passed argument (`a_c_field`) and copies it to the corresponding field.

The getter is implemented using the same copy behavior – the returned struct is an object backed by a copy of the actual struct. This is needed because we do not want 2 Eiffel object to point in the same memory area: it would be trickier to free the memory occupied by the struct when multiple objects are pointing to it. Listing 15 shows the getter implementation.

```
feature -- Getters

field: MY_STRUCT_2 assign set_field
        -- Return the struct field.
    do
        create Result.make
        c_copy_field (item, Result.item)
    end

feature {NONE} -- Implementation

c_field_copy_address (a_struct_pointer: POINTER; result_pointer: POINTER)
        -- Set `result_pointer' with the address of a copy of the field.
    external
        "C inline use <Cocoa/Cocoa.h>"
    alias
        "[
            size_t size = sizeof(((MyStruct *) $a_struct_pointer)->field);
            memcpy($result_pointer, &(((MyStruct *) $a_struct_pointer)->field),
size);
        ]"
    end
```

Listing 15: private and public getters for `field`.

6.2.3 Structs Comparison

In order to retain the same behavior as in C, we redefine `is_equal` to compare the memory pointed by the item attributes of the 2 structs (see listing 16).

```
is_equal (other: like Current): BOOLEAN
        -- Is `other' attached to an object considered
        -- equal to current object?
    do
        Result := item.memory_compare (other.item, structure_size)
    end
```

Listing 16: `is_equal` function to compare 2 structs.

6.3 Objective-C Classes

Objective-C classes are mapped to Eiffel classes. However, some of their functionalities such as class methods and categories, cannot be directly mapped to an Eiffel class. In the following chapters we describe the mapping for each Objective-C entity.

6.3.1 Mapping Of Objective-C Method Names

Method names in Objective-C have the following form.

```
setFirstName:lastName:age:
```

Listing 17: an Objective-C method name.

We convert the words to underscore syntax and we replace the colons with 2 underscores if the colon is not the last character in the name or with 1 underscore if the colon is the last character. This ensures the mapping function is injective. Listing 18 shows the method name in listing 17 after its conversion.

```
set_first_name__last_name__age_
```

Listing 18: method name converted to Eiffel style.

The double underscore is also used to make the labeled arguments more readable. The trailing underscore is still required to distinguish between method names that are equal except for a trailing colon. An example is given in listing 19.

```
setNeedsDisplay:
setNeedsDisplay
```

Listing 19: 2 similar Objective-C method names.

We also have to make sure method names don't have conflicts with Eiffel keywords. Cocoa has, in fact, several method names that generate conflicts, such as `class`, `prefix`, `result`, `attribute`, `copy` and `print`. To solve this problem we append the `_objc` suffix to those method names.

6.3.2 Introducing The Top Level Classes

The generated Eiffel classes inherit (directly or indirectly) from a set of auxiliary, pre-written top level classes. We introduce these classes in Figure 2.

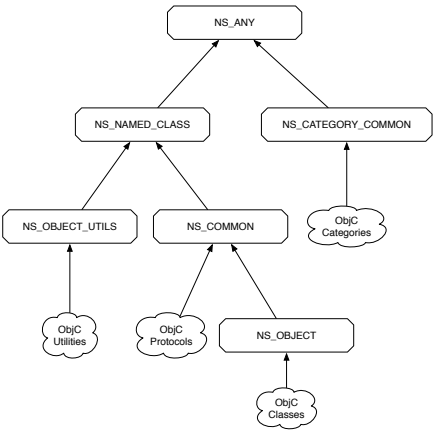

Figure 2: Top level classes. The clouds represent a set of classes. Arrows denote the inheritance relationship.

We give a brief description of these classes. We will describe them in more depth as needed in the following chapters.

`NS_ANY` is the top most class of the generated framework, it inherits from `ANY`. The `NS` prefix comes from the naming convention [7] used in the Cocoa framework.

`NS_NAMED_CLASS` is an auxiliary class that declares features useful when descending classes need to know their original Objective-C name.

`NS_CATEGORY_COMMON` is a common ancestor for all eiffel classes representing Objective-C categories.

`NS_OBJECT_UTILS` is a common ancestor for all eiffel classes containing Objective-C class methods. Note that `NS_OBJECT_UTILS` is not a pre-written class, it is generated by the tool.

Finally, `NS_COMMON` contains mechanisms inherited by `NS_OBJECT` – the main Cocoa root class – and every eiffel class representing Objective-C protocols. Note that `NS_OBJECT` is not a pre-written class, it is generated by the tool.

6.3.3 Mapping Inheritance

The inheritance chain of Objective-C classes is retained in the generated wrapper. That is to say if Objective-C class `ClassA` inherits from `ClassB`, the corresponding Eiffel class `CLASS_A` will inherit from `CLASS_B` too.

6.3.4 Wrapping Objective-C Objects

`NS_COMMON` declares an attribute `item` of type `POINTER`. This attribute points to the wrapped Objective-C object. For example, the attribute `item` of an instance of `NS_STRING` will point to the corresponding `NSString` Objective-C object.

6.3.5 Memory Management Invariant

In order to satisfy the memory management rule (see chapter 2.5) we define the following invariants:

Invariant 1: *An Eiffel wrapper object owns the Objective-C object pointed by* `item`.

Invariant 2: *An Eiffel wrapper object releases the Objective-C object pointed by* `item` *when it is disposed.*

With the term *Eiffel wrapper object* we mean an instance of an Eiffel class that inherits from `NS_OBJECT`.

If invariant 1 and 2 hold, we will not have memory leaks according to the memory management rule described in chapter 2.5.

There is a special class that does not need to satisfy the invariants: `OBJC_CLASS`. We present this class in chapter 6.3.10.

6.3.6 Mapping Procedures

For each Objective-C procedure we generate 2 features. The first one consists in an external feature exported to `NONE` whose name is prefixed with `objc_`. It has the arguments of the original Objective-C procedure along with a pointer pointing to the Objective-C object the method has to be invoked on. Let's consider the method of class `NSText` in listing 20.

```
- (void)setTextColor:(NSColor *)aColor
```

Listing 20: an Objective-C procedure.

Listing 21 shows the generated external for it.

```
objc_set_text_color_ (an_item: POINTER; a_color: POINTER)
        -- Auto generated Objective-C wrapper.
    external
        "C inline use <AppKit/AppKit.h>"
    alias
        "[
            [(NSText *)$an_item setTextColor:$a_color];
        ]"
    end
```

Listing 21: external procedure generated for the `setTextColor:` Objective-C procedure.

The wrapper generator converts the name of the Objective-C method (see chapter 6.3.1) and completes the header of the external with the name of the framework that declares the method (in this case `AppKit/AppKit.h`). Finally it

generates the Objective-C code that calls the method on the passed argument (`an_item`).

This feature is exported to `NONE` because it is part of the implementation and thus not part of the interface. The second generated feature is shown in listing 22 and it is exported to `ANY`.

```
set_text_color_ (a_color: detachable NS_COLOR)
        -- Auto generated Objective-C wrapper.
    local
        a_color__item: POINTER
    do
        if attached a_color as a_color_attached then
            a_color__item := a_color_attached.item
        end
        objc_set_text_color_ (item, a_color__item)
    end
```

Listing 22: `set_text_color_` procedure.

As we can see from the code, the argument is declared as `detachable`, this is in fact always the case (except for basic types or structs). This is because Cocoa always accepts `Void` pointers as arguments. The routine first checks whether the argument is `attached`. If it is it sets a local variable (`a_color__item`) to the Objective-C object wrapped by `a_color`. Then it simply calls the external feature with `item` (the attribute pointing to the wrapped Objective-C object) as first argument and `a_color__item` as second argument. If `a_color` was `Void` `a_color__item` would have been equal to the `default_pointer` (i.e. `NULL`).

The argument considered in this example (`a_color`) was of pointer type (`NSColor *`), therefore the external in listing 17 declared the argument as `POINTER`. If the argument was a basic type (e.g. `int`, `float`, `double`, etc.) the type would have been an Eiffel basic type too (e.g. `INTEGER_32`, `REAL_32`, `REAL_64`) and the second argument of the external call in listing 22 could have been passed as is without invoking `item` on it (this is because basic types in Eiffel are expanded: they are passed by value, not by reference). Note that even if the type is not explicitly a basic type, but rather an `enum` or `typedef`, the wrapper generator is smart enough to figure that out and to use the correct Eiffel basic type for it. Lastly, structs are handled as pointer types.

The code shown above does not break invariant 1 nor 2.

6.3.7 Mapping Queries With Expanded Return Type

Queries that return basic types such as `int`, `float`, `double`, etc. are mapped similarly to chapter 6.3.6. Let's consider the method of class `NSView` in listing 23.

```
- (BOOL)isHidden;
```

Listing 23: an Objective-C function returning a basic type.

Listing 24 shows the generated external and wrapping function for it.

```
feature {NONE} -- Implementation

objc_is_hidden (an_item: POINTER): BOOLEAN
        -- Auto generated Objective-C wrapper.
    external
        "C inline use <AppKit/AppKit.h>"
    alias
        "[
            return [(NSView *)$an_item isHidden];
        ]"
    end

feature

is_hidden: BOOLEAN
        -- Auto generated Objective-C wrapper.
    do
        Result := objc_is_hidden (item)
    end
```

Listing 24: generated external and wrapping function for the `isHidden` Objective-C function.

The wrapping function `is_hidden` simply returns the value returned by the external. The external returns the result of the Objective-C function. Because the return type is a basic C type, we don't need to wrap it: we can just pass it to Eiffel.

The code shown above does not break invariant 1 nor 2 because the returned data is an object.

6.3.8 Mapping Queries With Structs As Return Type

Let's consider the method of class `NSView` in listing 25.

```
- (NSRect)frame;
```

Listing 25: an Objective-C function returning a struct.

Listing 26 shows the generated code for it.

```
feature {NONE} -- Implementation

objc_frame (an_item: POINTER; result_pointer: POINTER)
        -- Auto generated Objective-C wrapper.
    external
        "C inline use <AppKit/AppKit.h>"
    alias
        "[
            *(NSRect *)$result_pointer = [(NSView *)$an_item frame];
        ]"
    end

feature

frame: NS_RECT
        -- Auto generated Objective-C wrapper.
    do
        create Result.make
        objc_frame (item, Result.item)
    end
```

Listing 26: generated code for the Objective-C function `frame`.

The wrapping function creates a new instance of the wrapped struct (`create Result.make`). Then it calls the external passing `item` and the pointer of the newly created struct wrapper (`Result.item`). The external copies the struct returned by the Objective-C function to the memory pointed by `Result.item`.

Invariant 1 and 2 only apply to Eiffel objects wrapping Objective-C objects. Structs are not Objective-C objects. The code is, however, memory leaks free because the memory allocated for the returned struct-wrapping Eiffel object will be freed when the Eiffel object is disposed.

6.3.9 Mapping Queries With Objects As Return Type

Let's consider the method of class `NSArray` in listing 27.

```
- (id)objectAtIndex:(NSUInteger)index;
```

Listing 27: an Objective-C function returning an object.

This method takes a `NSUInteger` (i.e. an `unsigned long` on 64 bit machines) as argument and returns an identifier (`id`), that is to say an object of type `NSObject`. The generation of the external is similar to the previous cases. Listing 24 shows the generated external for that method.

```
objc_object_at_index_ (an_item: POINTER; a_index: NATURAL_64): POINTER
        -- Auto generated Objective-C wrapper.
    external
        "C inline use <Foundation/Foundation.h>"
    alias
        "[
            return (EIF_POINTER)[(NSArray *)$an_item objectAtIndex:$a_index];
        ]"
    end
```

Listing 28: external generated for the `objectAtIndex:` Objective-C function.

The external simply returns the address of the object returned by the Objective-C function. What the wrapping function needs to do is to create an Eiffel object to wrap the returned Objective-C object. This is, however, not always needed because there might be already an Eiffel object in the runtime wrapping that object. We want the returned Objective-C object to be wrapped by at most 1 Eiffel object. When this is satisfied we say the identity property holds. The following example illustrates the problems that might arise when this property does not hold.

```
array: NS_MUTABLE_ARRAY

add_object_at_beginning_of_array (an_object: NS_OBJECT)
    do
        array.insert_object__at_index_ (an_object, 0)
    ensure
        object_added: attached array.object_at_index_ (0) as object and then
                      object = an_object
    end
```

Listing 29: example illustrating the identity property problem.

The code inserts a `NS_OBJECT` object in a `NS_MUTABLE_ARRAY` array at index 0. The postcondition checks whether the object has been inserted correctly. Yet if the code of `object_at_index_` creates a new Eiffel object to wrap the returned `NS_OBJECT` the postcondition will fail. This might be very surprising for a developer using the framework.

To solve this problem we need Objective-C objects to keep track of which Eiffel object is currently wrapping them. We use Objective-C associative references to achieve this. Associative references simulate the addition of object instance variables to an existing Objective-C class. (see [7]).

We add two routines that make use of associative references in the `NS_ANY` class. The first one (`objc_set_eiffel_object`) sets the Eiffel object wrapping an Objective-C object, the second one (`objc_get_eiffel_object`) gets the Eiffel object that is wrapping an Objective-C object. We first describe `objc_set_eiffel_object` shown in listing 30.

```
objc_set_eiffel_object (a_pointer: POINTER; an_object: POINTER)
        -- [...]
    external
        "C inline use <assert.h>"
    alias
        "[
            EIF_OBJECT object_to_associate = NULL;
            if ($an_object != NULL) {
                assert(!objc_getAssociatedObject($a_pointer, NULL));
                object_to_associate = eif_create_weak_reference($an_object);
            } else {
                EIF_OBJECT associated_object = (EIF_OBJECT)objc_getAssociatedObject
($a_pointer, NULL);
                assert(associated_object);
                eif_free_weak_reference(associated_object);
            }
            objc_setAssociatedObject($a_pointer, NULL, (id)object_to_associate,
OBJC_ASSOCIATION_ASSIGN);
        ]"
    end
```

Listing 30: the `objc_set_eiffel_object` procedure.

The first argument is a pointer to an Objective-C object, the second one is a `POINTER` to an Eiffel object. If `an_object` is not the `default_pointer`, the routine associates it with the Objective-C object pointed by `a_pointer`. It also needs to use the `eif_create_weak_reference` function to get a weak reference to the Eiffel object that stays valid in time because the Eiffel garbage collector may move the object around. `eif_create_weak_reference` does not prevent the garbage collector to collect the Eiffel object if it is not referenced anymore. Otherwise, i.e. `an_object` is the `default_pointer`, the routine deletes the association and uses `eif_free_weak_reference` to delete the weak reference for the Eiffel object. In other words, this routine sets or unsets a reference from an Objective-C object to an Eiffel object.

The second routine is shown in listing 31.

```
objc_get_eiffel_object (a_pointer: POINTER): detachable ANY
        -- [...]
    external
        "C inline use <objc/runtime.h>"
    alias
        "[
            EIF_OBJECT associated_object = (EIF_OBJECT)objc_getAssociatedObject
($a_pointer, NULL);
            if (associated_object != NULL) {
                return eif_access(associated_object);
            } else {
                return NULL;
            }
        ]"
    end
```

Listing 31: the `objc_get_eiffel_object` function.

This function takes only 1 argument, i.e. a pointer to an Objective-C object. It returns the Eiffel object associated with the Objective-C object pointed by `a_pointer`. If there is no Eiffel object associated it simply returns `Void`.

Now that we have a way to set, unset and get references from Objective-C objects to Eiffel objects we can generate a wrapping function for methods that return objects just like the one shown in listing 27. Listing 32 shows the code.

```
object_at_index_ (a_index: NATURAL_64): detachable NS_OBJECT
        -- Auto generated Objective-C wrapper.
    local
        result_pointer: POINTER
    do
        result_pointer := objc_object_at_index_ (item, a_index)
        if result_pointer /= default_pointer then
            if attached objc_get_eiffel_object (result_pointer) as
existing_eiffel_object then
                check attached {like object_at_index_} existing_eiffel_object as
valid_result then
                    Result := valid_result
                end
            else
                check attached {like object_at_index_} new_eiffel_object
(result_pointer, True) as valid_result_pointer then
                    Result := valid_result_pointer
                end
            end
        end
    end
```

Listing 32: the wrapping function generated for `objectAtIndex:`.

First, the external feature is called and the returned object is assigned to `result_pointer`. If it is the `default_pointer` the function will return `Void`. Otherwise we check whether the object pointed by `result_pointer` already has an Eiffel object associated with it using the `objc_get_eiffel_object` function. If that is the case we simply return the associated Eiffel object. If not, we create a new Eiffel wrapper object with the `new_eiffel_object` function and return it.

The `new_eiffel_object` function is declared in `NS_ANY` and it takes 2 arguments. The first one is the Objective-C object that needs to be wrapped, the second one is a boolean that indicates whether the Objective-C object needs to be sent a `retain` message or not. This is needed because of invariant 1 (see chapter 6.3.5) which states the following.

Invariant 1: *An Eiffel wrapper object owns the Objective-C object pointed by* `item`.

We recall the memory management definition of ownership for convenience of the reader.

You own an object whenever:

1) you retain the object,

2) you receive a reference to it by a function whose name

- starts with the keywords `alloc` or `new`,
- contains the keyword `copy`.

If the object assigned to result_pointer was returned by an Objective-C function whose name starts with alloc, new or contains the keyword copy we do not need to retain the object pointed by it, otherwise yes. This ensures invariant 1 is satisfied. In the example above (listing 32) the function name is objectAtIndex:, we therefore need to retain the object pointed by result_pointer. This is why the second argument of new_eiffel_object is True. The wrapper generator is able to determine whether to retain an object or not just by looking at the function name.

We will now describe the new_eiffel_object function. We will start with a simplified version and give the full version in the next chapters.

```
new_eiffel_object (a_pointer: POINTER; retain: BOOLEAN): detachable NS_OBJECT
        -- [...]
    do
        check attached {like new_eiffel_object} internal.new_instance_of
(get_eiffel_type (a_pointer)) as eiffel_object then
            if retain then
                eiffel_object.make_with_pointer_and_retain (a_pointer)
            else
                eiffel_object.make_with_pointer (a_pointer)
            end
        end
    end
```

Listing 33: the new_eiffel_object function.

As we explained in the paragraph above, the first argument of new_eiffel_object is a pointer to an Objective-C object that needs to be wrapped, the second one specifies whether that object needs to be retained or not. First, the function creates a new Eiffel object of the right type using the {INTERNAL}.new_instance_of function. The get_eiffel_type is a function that returns the Eiffel type corresponding to the Objective-C type (we will describe it later in this chapter). After the Eiffel object of the right kind has been created, we still need to associate it with the corresponding Objective-C object and vice-versa and – if that is the case – retain the Objective-C object. To do this we make use of the make_with_pointer and make_with_pointer_and_retain procedures defined in NS_COMMON. Listing 34 shows a preliminary version of make_with_pointer (we will present the full code in following chapters).

```
make_with_pointer (a_pointer: POINTER)
        -- Initialize `Current' with `a_pointer'.
    require
        a_valid_pointer: a_pointer /= default_pointer
    do
        item := a_pointer
        objc_set_eiffel_object (item, $Current)
    ensure
        item_set: item = a_pointer
    end
```

Listing 34: the make_with_pointer creation procedure.

The function sets item to the passed Objective-C pointer and then associates the Eiffel object ($Current) with the Objective-C object.

`make_with_pointer_and_retain` simply retains the Objective-C object before passing it to the `make_with_pointer` procedure (see listing 31).

```
make_with_pointer_and_retain (a_pointer: POINTER)
        -- Initialize `Current' with `a_pointer' and send
        -- it an Objective-C retain message.
    require
        a_valid_pointer: a_pointer /= default_pointer
    do
        make_with_pointer (objc_retain (a_pointer))
    ensure
        item_set: item = a_pointer
    end
```

Listing 35: the `make_with_pointer_and_retain` creation procedure.

In order to satisfy the memory management invariant 2 (i.e. *an Eiffel wrapper object releases the Objective-C object pointed by* `item` *when it is disposed*) we need to implement the `dispose` feature called by the garbage collector just before the object is collected. Listing 36 shows a preliminary version of the function defined in `NS_COMMON` (the full function code will be presented in the following chapters).

```
dispose
        -- <Precursor>
    do
        objc_set_eiffel_object (item, default_pointer)
        objc_release (item)
    end
```

Listing 36: the `dispose` procedure.

The `dispose` procedure first invokes the `objc_set_eiffel_object` in order to dissociate the Eiffel object from the Objective-C object. `objc_set_eiffel_object` will also invoke `eif_free_weak_reference()` to invalidate the weak reference we previously created when we created the object. Next, it simply releases the Objective-C object pointed by `item`. This ensures invariant 2 holds.

Let us now go back to the `get_eiffel_type` function (listing 37).

```
get_eiffel_type (a_pointer: POINTER): INTEGER
        -- [...]
    require
        a_valid_pointer: a_pointer /= default_pointer
    do
        Result := mapping.item (objc_class_objc (a_pointer))
    end
```

Listing 37: the `get_eiffel_type` function.

`get_eiffel_type` first finds out the Objective-C class of the object pointed by `a_pointer` using the `objc_class_objc` function. Then it retrieves and returns the Eiffel type associated with that Objective-C class using a mapping from Objective-C classes to Eiffel types (`mapping`). Listing 38 shows the `mapping` function.

```
mapping: HASH_TABLE [INTEGER, POINTER]
        -- [...]
    local
        classes_mapper: CLASSES_MAPPER
        computed_mapping: HASH_TABLE [STRING, POINTER]
        dynamic_type: INTEGER
    once
        create classes_mapper.make
        classes_mapper.compute_mapping
        computed_mapping := classes_mapper.mapping
        create Result.make (4096)
        from
            computed_mapping.start
        until
            computed_mapping.after
        loop
            dynamic_type := internal.dynamic_type_from_string
(objc_class_name_to_eiffel_style (computed_mapping.item_for_iteration))
            check valid_dynamic_type: dynamic_type /= -1 end
            Result.put (dynamic_type, computed_mapping.key_for_iteration)
            computed_mapping.forth
        end
    end
```

Listing 38: the `mapping` function.

`mapping` is a hash table mapping Objective-C `Class` objects (the type of objects returned by the `objc_class_objc` function shown in listing 37) to integers representing Eiffel types that can be used to instantiate new Eiffel objects with the `{INTERNAL}.new_instance_of` function. The function utilizes `classes_mapper`, an instance of `CLASSES_MAPPER`, to get a mapping from Objective-C `Class` objects to strings representing Objective-C class names. It then converts those strings with the `{INTERNAL}.dynamic_type_from_string` function to an integer that can be later be used with the `{INTERNAL}.new_instance_of` function.

We will now describe how `CLASSES_MAPPER` computes the mapping. The `compute_mapping` function is shown in Listing 39.

```
compute_mapping
        -- Compute `mapping'.
    local
        managed_pointer: MANAGED_POINTER
        c_mapping: POINTER
        count: INTEGER
        i: INTEGER
        class_object: POINTER
        string_pointer: POINTER
        c_string: C_STRING
        pointer_bytes: INTEGER
    do
        pointer_bytes := {PLATFORM}.pointer_bytes
        c_mapping := objc_get_mapping ($count)
        create managed_pointer.own_from_pointer (c_mapping, count * pointer_bytes)
        from
            i := 0
        until
            i >= count
        loop
            class_object := managed_pointer.read_pointer (i * pointer_bytes)
            string_pointer := managed_pointer.read_pointer ((i + 1) * pointer_bytes)
            if string_pointer /= default_pointer then
                create c_string.make_shared_from_pointer (string_pointer)
                mapping.put (c_string.string, class_object)
            end
            i := i + 2
        end
    end
```

Listing 39: the `compute_mapping` procedure.

`compute_mapping` simply invokes `objc_get_mapping` to get a C array of elements following the format [`a_objc_class_object`, `a_objc_class_name`, `a_objc_class_object2`, `a_objc_class_name2`, ...], i.e. the `i`-th and `i`-th plus 1 elements of the array are the pointers to the Objective-C Class object and Objective-C class name, respectively (for `i` an even number). The details of the code are left to the reader.

Before we explain how `objc_get_mapping` computes the mapping we recall the concept of class clusters (chapter 2.6).

> Class clusters group a number of private, concrete subclasses under a public, abstract superclass. The grouping of classes in this way simplifies the publicly visible architecture of an object-oriented framework without reducing its functional richness.
>
> [...]
>
> The abstract superclass in a class cluster must declare methods for creating instances of its private subclasses. It's the superclass's responsibility to dispense an object of the proper subclass based on the creation method that you invoke—you don't, and can't, choose the class of the instance.

Because of class clusters, not all returned Objective-C objects have a type known to the wrapper generator. The wrapper generator only knows classes declared in the header files of the framework. Which generated Eiffel class should be instantiated if we receive an object whose type is private, i.e. not in the header files? Let's consider the following example. `NSNumber` declares a `numberWithInt:` function. The returned object is, however, an instance of `NSCFNumber` – a private subclass. The wrapper generator is not aware of the `NSCFNumber` class and thus it did not generate the `NS_CF_NUMBER` Eiffel class wrapper. The only meaningful Eiffel class to instantiate to wrap that Objective-C object would be `NS_NUMBER`. There are some more complicated cases where public class clusters inherit from other class clusters. For instance, `NSMutableArray` is a class cluster and it inherits from `NSArray` – a class cluster again. Therefore, a private Objective-C class should be mapped to the closest Objective-C public ancestor. In order to generate this mapping, `objc_get_mapping` needs to know the list of all classes declared in the header files (the public classes) and the list of all classes registered in the runtime (including private classes). The list of all public classes can be easily known because the wrapper generator can generate code in the `classes_mapper.e` file to populate an array with such information. The list of all private classes can be extracted from the list of all Objective-C classes registered in the runtime system using the `objc_getClassList` Objective-C function. Listing 40 shows the `objc_get_mapping` function. The code has been split on more pages to facilitate the reading. Only the first portion of the (repetitive) code used to populate the array is shown.

```
objc_get_mapping (out_count: POINTER): POINTER
        -- [...]
    external
        "C inline use <objc/runtime.h>, <assert.h>"
    alias
        "[
            int parsed_classes_count = 360;
            Class parsed_classes[parsed_classes_count];
            parsed_classes[0] = objc_getClass("NSObject");
            parsed_classes[1] = objc_getClass("NSEnumerator");
            parsed_classes[2] = objc_getClass("NSValue");
            parsed_classes[3] = objc_getClass("NSNumber");
            parsed_classes[4] = objc_getClass("NSArray");
            parsed_classes[5] = objc_getClass("NSMutableArray");
            parsed_classes[6] = objc_getClass("NSAutoreleasePool");
            parsed_classes[7] = objc_getClass("NSBundle");
            parsed_classes[8] = objc_getClass("NSDate");
            parsed_classes[9] = objc_getClass("NSCalendar");
            parsed_classes[10] = objc_getClass("NSDateComponents");
            parsed_classes[11] = objc_getClass("NSString");
            parsed_classes[12] = objc_getClass("NSMutableString");
            parsed_classes[13] = objc_getClass("NSSimpleCString");
            parsed_classes[14] = objc_getClass("NSConstantString");
            parsed_classes[15] = objc_getClass("NSCharacterSet");
[...]
```

```
    int runtime_classes_count = objc_getClassList(NULL, 0);
    assert(runtime_classes_count > 0);
    Class *runtime_classes = malloc(sizeof(Class) * runtime_classes_count);
    objc_getClassList(runtime_classes, runtime_classes_count);
    void **mapping = malloc(2 * sizeof(void *) * runtime_classes_count);
    int i, j;
    Class runtime_class, superclass;
    BOOL found;
    for (i = 0; i < runtime_classes_count; i++) {
        runtime_class = runtime_classes[i];
        mapping[2*i] = runtime_class;
        for (j = 0, found = NO; j < parsed_classes_count; j++) {
            if (runtime_class == parsed_classes[j]) {
                found = YES;
                break;
            }
        }
        if (found) {
            mapping[2*i + 1] = (void *)class_getName(runtime_class);
        } else {
            superclass = runtime_class;
            while (!found) {
                superclass = class_getSuperclass(superclass);
                if (superclass == nil) {
                    break;
                }
                for (j = 0, found = NO; j < parsed_classes_count; j++) {
                    if (superclass == parsed_classes[j]) {
                        found = YES;
                        break;
                    }
                }
            }
            if (found) {
                mapping[2*i + 1] = (void *)class_getName(superclass);
            } else {
                // This is a private class and it is not part of any class cluster.
                mapping[2*i + 1] = NULL;
            }
        }
    }
    free(runtime_classes);
    *(int *)$out_count = 2 * runtime_classes_count;
    return mapping;
    ]"
end
```

Listing 40: the `objc_get_mapping` function.

The details of the code are left to the reader.

Previously in this chapter we presented a simplified version of the `new_eiffel_object` function. We will now present the full version that takes into account the special case of class objects (see listing 41). The added code has been highlighted.

```
new_eiffel_object (a_pointer: POINTER; retain: BOOLEAN): detachable NS_OBJECT
        -- [...]
    do
        if objc_class_objc (a_pointer) = a_pointer then
                -- If `a_pointer' represents a class object
            create {OBJC_CLASS} Result.make_with_pointer (a_pointer)
        else
            check attached {like new_eiffel_object} internal.new_instance_of
(get_eiffel_type (a_pointer)) as eiffel_object then
                if retain then
                    eiffel_object.make_with_pointer_and_retain (a_pointer)
                else
                    eiffel_object.make_with_pointer (a_pointer)
                end
            end
        end
    end
```

Listing 41: the updated `new_eiffel_object` function.

The new code first checks whether the Objective-C object pointed by `a_pointer` is a `Class` object (if we query a `Class` object for its `Class` object, they will return a reference to themselves). Next we create a new instance of `OBJC_CLASS`, we do not care whether to retain it or not since it is a singleton. `OBJC_CLASS` is a custom class that wraps an Objective-C `Class` object. It is described in chapter 6.3.10.

Lastly, we need to handle the case of queries with return type `SEL` (Objective-C selectors) in a special way because they are not object. Listing 36 shows the code generated for a method named `selector` returning an object of type `SEL`.

```
selector: detachable OBJC_SELECTOR
        -- Auto generated Objective-C wrapper.
    local
        result_pointer: POINTER
    do
        result_pointer := objc_selector (item)
        if result_pointer /= default_pointer then
            create {OBJC_SELECTOR} Result.make_with_pointer (result_pointer)
        end
    end
```

Listing 42: code generated for an Objective-C function returning a selector.

The code simply creates and initializes a new instance of `OBJC_SELECTOR`, a custom class, with the returned selector pointed by `result_pointer`. `OBJC_SELECTOR` simply declares some features like `make_with_pointer`, `is_equal` and `item` (they are only exported to `OBJC_SELECTOR` and `NS_ANY`). It does not, however, inherit from `NS_ANY` because we do not need the facilities declared there. We do not have to worry about memory management when dealing with selectors because they are never deallocated.

6.3.10 Mapping The Objective-C `Class` Type

Objective-C Class objects are a bit particular because they are not explicitly described as objects in the documentation. The only definition the documentation gives about Class objects is the following.

```
typedef struct objc_class *Class;
```

Listing 43: declaration of the `Class` type.

I.e. `Class` is a type definition for a pointer to an `objc_class` struct (an opaque type, i.e. the interface does not declare the fields of the struct). However, variables of type `Class` seem to have an objective nature because we can send them Objective-C messages (in particular all Objective-C messages declared in the `NSObject` protocol). Therefore we decided to treat them as objects in Eiffel too and to manually create the `OBJC_CLASS` inheriting from `NS_OBJECT`.

`Class` objects have a name, an optional superclass and can be allocated and registered (registered classes are known to the Objective-C runtime and instances of them can be created). If a class has not been allocated it cannot be registered.

```
name: STRING
        -- The name of the class represented by `Current'.

superclass_objc: detachable OBJC_CLASS assign set_superclass_objc
        -- The superclass of this class. If the class is registered it is guaranteed
to have a superclass.
    do
        if registered then
            create Result.make_with_pointer_and_retain (objc_class_get_superclass
(item))
        else
            Result := internal_superclass_objc
        end
    ensure
        registered_implies_superclass_not_void: registered implies Result /= Void
    end

registered: BOOLEAN
        -- Is the Objective-C class represented by `Current' registered in the
Objective-C runtime?

allocated: BOOLEAN
        -- Has the Objective-C class represented by `Current' already been allocated?
```

Listing 44: attributes of `OBJC_CLASS`.

It also provides 2 creation procedures: `make_with_pointer` and `make_with_name` (see listing 45).

```
make_with_pointer (a_pointer: POINTER)
        -- Initialize `Current' with `a_pointer'.
    local
        c_string: C_STRING
    do
        item := a_pointer
        create name.make_from_c (objc_class_get_name (item))
        create c_string.make (name)
        allocated := True
        registered := objc_get_class (c_string.item) /= default_pointer
    end

make_with_name (a_name: STRING)
        -- Initialize `Current' with `a_class_name'
    require
        a_valid_name: not a_name.is_empty
    local
        c_string: C_STRING
    do
        name := a_name
        create c_string.make (name)
        item := objc_get_class (c_string.item)
        registered := item /= default_pointer
        allocated := registered
    ensure
        name_set: name = a_name
    end
```

Listing 45: creation procedures of `OBJC_CLASS`.

`make_with_pointer` initializes `item`, sets `name` (see the Objective-C `objc_class_get_name` function) and the `allocated` and `registered` attribute by checking whether the passed class exists in the Objective-C runtime (see the Objective-C `objc_get_class` function). Similarly, `make_with_name` sets the `name` and the `registered` attribute by checking whether the `a_name` argument corresponds to the name of an Objective-C class registered in the runtime. The `allocated` attribute is also set accordingly.

Because Objective-C classes are singleton objects we do not have to worry about disposal. Therefore, the `dispose` feature is redefined and it does not do anything.

`OBJC_CLASS` also declares methods to allocate and register a class. They are shown in listing 46.

```
allocate
        -- Creates a new Objective-C class and metaclass.
    require
        not_allocated: not allocated
        has_superclass_objc: superclass_objc /= Void
    do
        check attached superclass_objc as attached_superclass_objc then
            item := objc_allocate_class_pair (attached_superclass_objc.item, {create
{C_STRING}.make (name)).item, 0)
        end
        allocated := True
    ensure
        allocated: allocated
    end

register
        -- Register this class in the Objective-C runtime such that it can be used.
    require
        allocated: allocated
        not_registered: not registered
    do
        objc_register_class_pair (item)
        internal_superclass_objc := Void
        registered := True
    ensure
        registered: registered
    end
```

Listing 46: `allocate` and `register` procedures of `OBJC_CLASS`.

The `allocate` and `register` methods simply call their Objective-C counterpart and the set the `allocated` and `registered` attributes accordingly.

Lastly, we also declare a feature to add methods to an Objective-C class. Listing 47 shows the code.

```
add_method (a_selector: OBJC_SELECTOR; an_implementation: POINTER; types: STRING):
BOOLEAN
        -- Add a new method to this class with a given name and implementation.
        -- Return `True' if the method was added successfully.
    require
        allocated: allocated
        not_registered: not registered
    local
        c_string: C_STRING
    do
        create c_string.make (types)
        Result := objc_class_add_method (item, a_selector.item, an_implementation,
c_string.item)
    end
```

Listing 47: `add_method` function of `OBJC_CLASS` to add methods to an Objective-C class.

This function will be very useful when implementing a solution for Objective-C callbacks (i.e. function calls from Objective-C to Eiffel).

6.3.11 Mapping Objective-C Categories (Methods Grouping)

Objective-C categories are used for 2 purposes: to group methods or to add methods to a class without subclassing it. In this chapter we are going to map the first use case of categories, i.e. methods grouping. We distinguish the 2 use cases by checking in which framework the category has been defined. If it was defined in the same framework as the original class we assume it is used to group methods (in practice this is always the case), otherwise it is used to extend a class declared in an other framework.

It is relatively easy to group features in Eiffel. We can just use the `feature` keyword followed by a comment. We can use the Objective-C category name as a comment.

6.3.12 Mapping Class Methods

For Objective-C class methods of a given class we generate an Eiffel class with the original class name converted to Eiffel capital case underscore syntax with the `_UTILS` suffix (it stands for utilities). These classes inherit from their corresponding `_UTILS` class parent (according to the Objective-C hierarchy). The parent of the `_UTILS` class of the top-most class in the Objective-C hierarchy – `NSObject` – is `NS_NAMED_CLASS`.

6.3.13 Object Creation

Objects in Objective-C are usually created by sending an `alloc` method to the class object of the type you want to instantiate, followed by an initialization method (whose name begins with `init`). Listing 48 shows an example.

```
NSNumber *aNumber = [[NSNumber alloc] initWithInt:1];
```

Listing 48: instantiating an `NSNumber` object.

Unlike in Eiffel, initialization in Objective-C consists in assigning the returned initialized object to a variable (just like in listing 48). This is because the initializer does not necessarily need to return the same object it was called on (see, for example, class clusters). Listing 49 shows a way not to do initialization.

```
// Wrong! Do not do this!
NSNumber *aNumber = [NSNumber alloc];
[aNumber initWithInt:1];
```

Listing 49: bad way to create a new `NSNumber` object.

Our goal is to provide an easy way for Eiffel developers to instantiate this kind of objects. We don't want the Eiffel developer to struggle with `alloc` and assigning the result of `init` methods. The Eiffel way would be the one shown in Listing 50.

```
create n.make_with_int_ (1)
```

Listing 50: Eiffel way to create an object.

For this reason, we do not generate any `alloc` method and we replace the prefix `init` of all Objective-C initializers with `make` (there is no way to tell whether a method is used as an initializer other than checking whether it start with `init`, this is in practice always the case). Listing 51 shows how the initializer looks like.

```
make_with_int_ (a_value: INTEGER_32)
        -- Initialize `Current'.
    do
        make_with_pointer (objc_init_with_int_(allocate_object, a_value))
    end
```

Listing 51: code generated for the `initWithInt:` Objective-C function.

The initializer calls `make_with_pointer` to initialize `item` and to set up the reverse Objective-C association with the passed argument which happens to be a pointer to the object returned by the initializer `objc_init_with_int_` (shown in Listing 52).

```
objc_init_with_int_ (an_item: POINTER; a_value: INTEGER_32): POINTER
        -- Auto generated Objective-C wrapper.
    external
        "C inline use <Foundation/Foundation.h>"
    alias
        "[
            return (EIF_POINTER)[(NSNumber *)$an_item initWithInt:$a_value];
        ]"
    end
```

Listing 52: generated function for the Objective-C `initWithInt:` function.

The first parameter of the initializer must be an instance of the Objective-C class. The `allocate_object` function, declared in `NS_COMMON`, returns just that. Listing 53 shows its implementation (the code is incomplete for didactic purposes, we will present the full code incomplete the following chapters).

```
allocate_object: POINTER
        -- Allocate an Objective-C instance of `Current' and return a pointer
        -- to its address.
    local
        l_objc_class: OBJC_CLASS
    do
        create l_objc_class.make_with_name (get_class_name)
        check l_objc_class_registered: l_objc_class.registered end
        Result := objc_alloc (l_objc_class.item)
    end
```

Listing 53: the `allocate_object` function.

The function creates an `OBJC_CLASS` object using the name returned by `get_class_name`. `get_class_name` is declared in `NS_NAMED_CLASS` and it is redefined by inheriting classes to return the Objective-C class name associated with

the current Eiffel wrapper object. Next, we allocate and return an instance of that class using the `objc_alloc` function.

What we achieved with this is an easy way to instantiate objects which is familiar to Eiffel users (see listing 50). Listing 54 shows how inconvenient it would have been to create objects using the Objective-C style.

```
create_a_number
        -- Only for demo purposes. This is not supported by the generated wrapper.
    local
        n: NS_NUMBER
        ns_number_utils: NS_NUMBER
    do
        create ns_number_utils
        n := ns_number_utils.alloc.init_with_int_ (1)
    end
```

Listing 54: example that shows the inconvenience of using Objective-C style object creation in Eiffel.

The convenience, however, comes with a little price to pay. There are certain situations where creating objects in this manner could result in inconsistencies or crashes. We first explain this problem and present a solution later.

In Objective-C nothing prevents an initializer to always return the same object. This does in fact sometimes happen. For example, the address of the object assigned to `number` in listing 55 will always be the same.

```
number = [[NSNumber alloc] initWithInt:1];
```

Listing 55: creating a `NSNumber` instance.

This is because `NSNumber` stores some pre-cached `NSNumber` objects with frequently used constants like the numbers [0-12]. If the argument passed to the initializer is a number within that range, the initializer will return the cached `NSNumber` object. Even `[NSNumber alloc]` does not really allocate a new object, it simply returns a singleton placeholder instance (which is in fact of type `NSPlaceholderNumber`). Using this technique `NSNumber` can create objects more efficiently. Now, after the execution of the code in listing 56 there will be 2 Eiffel objects pointing to the same `NSNumber` object.

```
create_two_numbers
        --
    local
        n1, n2: NS_NUMBER
    do
        create n1.make_with_int_ (1)
        create n2.make_with_int_ (1)
            -- Inconsistency in the system
    end
```

Listing 56: creation of 2 `NS_NUMBER` instances with the same initialization value (i.e. `1`).

Moreover, the second object creation will override the existing association between the Eiffel object `n1` and the Objective-C one. This might lead to crashes when the 2 objects will be disposed.

We do not want the user to always check whether the object returned by an Objective-C initializer is a singleton or not. Therefore we check this automatically in the `make_with_pointer` procedure as shown in listing 57. The added code has been highlighted.

```
make_with_pointer (a_pointer: POINTER)
        -- Initialize `Current' with `a_pointer'.
    require
        a_valid_pointer: a_pointer /= default_pointer
    do
        item := a_pointer
        if attached objc_get_eiffel_object (item) then
            is_shared_objc_object := True
        else
            objc_set_eiffel_object (item, $Current)
        end
    ensure
        item_set: item = a_pointer
    end
```

Listing 57: the updated `make_with_pointer` creation procedure.

The `make_with_pointer` procedure first checks whether there's already an Eiffel object associated with the Objective-C object. If this is the case it does not override the association and it sets the `is_shared_objc_object` attribute to `True` such that the `dispose` feature can be aware of this and it will not reset the association. Listing 58 shows the code of `dispose`. The added code has been highlighted.

```
dispose
        -- <Precursor>
    do
        if not is_shared_objc_object then
            objc_set_eiffel_object (item, default_pointer)
        end
        objc_release (item)
    end
```

Listing 58: the updated `dispose` procedure.

These little changes solve to problem of instantiating (using the `create` keyword) Eiffel objects that wrap Objective-C singletons. This has, however, the consequence of breaking the identity property, i.e. we might have multiple Eiffel objects in the system pointing to the same Objective-C object. Listing 59 highlights the problem with an example.

```
array: NS_MUTABLE_ARRAY

cause_trouble
        --
    local
        n1, n2: NS_NUMBER
    do
        create n1.make_with_int_ (1)
        create n2.make_with_int_ (1)
        add_object_at_beginning_of_array (n2)
    end

add_object_at_beginning_of_array (an_object: NS_OBJECT)
        --
    do
        an_array.insert_object__at_index_ (an_object, 0)
    ensure
        object_added: attached an_array.object_at_index_ (0) as object and then
                      object = an_object
    end
```

Listing 59: example that shows how the identity property does not always hold.

The procedure `add_object_at_beginning_of_array` seems to satisfy the postcondition. This, however, will not be the case in the example above because the object returned by `an_array.objcect_at_index_ (0)` will be the object `n1` created in the `cause_trouble` procedure instead of `n2`, i.e. the object passed to `add_object_at_beginning_of_array`.

Unfortunately, we cannot provide a better solution because each object instantiated with the `create` keyword in Eiffel will be a different one. Therefore, it is impossible to wrap an Objective-C singleton if this has to be instantiated with the `create` keyword. This is not a problem if, instead, a wrapped Objective-C function returns the object. For this reason, the identity property will generally hold. In order for the identity property to hold all the time we would need to prevent the users to create Eiffel objects wrapping Objective-C singletons. We chose not to do this in order not to limit functionality.

6.3.14 Subclassing

It is a rather common pattern to subclass classes of a framework. This chapter describes what needs to be done under the hood when users decide to subclass classes of the generated wrapper.

When a user subclasses a class of the generated wrapper we need to create a subclass of the corresponding Objective-C class as well. This can easily be done because of the dynamic nature of the Objective-C runtime system. Listing 60 shows the updated `allocate_object` function. The added code has been highlighted.

```
allocate_object: POINTER
        -- Allocate an Objective-C instance of `Current' and return a pointer
        -- to its address.
    local
        l_objc_class: OBJC_CLASS
    do
        create l_objc_class.make_with_name (get_class_name)
        if is_subclass_instance and not l_objc_class.registered then
            l_objc_class.superclass_objc := create {OBJC_CLASS}.make_with_name
(wrapper_objc_class_name)
            l_objc_class.allocate
            l_objc_class.register
        end
        check l_objc_class_registered: l_objc_class.registered end
        Result := objc_alloc (l_objc_class.item)
    end
```

Listing 60: the updated `allocate_object` function.

The function uses the `is_subclass_instance` function declared in `NS_NAMED_CLASS` to check whether the current object is a subclass instance of a class of the generated wrapper (it does so by comparing the class name of the current object with `wrapper_objc_class_name`, the original name of the wrapped Objective-C class). If this is the case and the Objective-C subclass has not been created yet (i.e. it is not registered in the runtime), it allocates and registers it. For example, if we instantiate an Eiffel class named `MY_VIEW` which inherits from the generated class `NS_VIEW`, a new class named `MY_VIEW` will be allocated and registered in the Objective-C runtime too. `MY_VIEW` (in the Objective-C runtime) will be a subclass of `NSView`. To make sure `allocate_object` will get called the user has to ensure the original initializer will be executed. If the user redefines an initializer he can use the `Precursor` keyword to call the parent implementation.

With subclassing comes a problem about memory management. Let's consider the Eiffel class `MY_VIEW` shown in listing 61.

```
class
    MY_VIEW
inherit
    NS_VIEW
create
    make
feature -- Setting and Access
    set_data (a_data: like data)
            -- Set `data' with `a_data'.
        do
            data := a_data
        end

    data: detachable STRING assign set_data
end
```

Listing 61: the code of the `MY_VIEW` class.

`MY_VIEW` declares the attribute `data` to store a string. Clients can save a string in it with the setter `set_data`. Let's now consider the code in listing 62.

```
array: NS_MUTABLE_ARRAY

add_my_view_to_array
        --
    local
        my_view: MY_VIEW
    do
        create my_view.make
        my_view.data := "Saved data"
        array.insert_object__at_index_ (my_view, 0)
    end

check_saved_data
        --
    do
        check
            attached array.object_at_index_ (0) as my_view and then
            attached my_view.data as data and then
            data.is_equal ("Saved data")
        end
    end
```

Listing 62: example that highlights the problem of memory management when creating and using subclasses of the generated wrapper.

The function `add_my_view_to_array` instantiates `my_view`, it saves some data in it and adds it to an array. Let's assume after the execution of `add_my_view_to_array` the garbage collector executes a full collecting cycle. The Eiffel object initially referenced by `my_view` is not referenced anymore. Therefore it is collected releasing the corresponding Objective-C object. The corresponding Objective-C object will not, however, be deallocated because the array retained it when it was added (every collection class in Cocoa – e.g. arrays, dictionaries, etc. – retains its elements when they are added and releases them when they are removed or when the collection is deallocated). Now, if we execute the `check_saved_data` procedure the `object_at_index_` function will create a new Eiffel object to wrap the result. The `data` attribute of the newly created Eiffel object will obviously be `Void`. Ergo the data was lost.

To solve this problem we need to make sure Eiffel instances of subclasses stay in the system until their Objective-C counterpart is not owned by anybody else except the Eiffel object itself (i.e. its retain count is 1). To make an Eiffel object stays in the system even though it is not referenced we can use the C function `eif_protect()`. We will protect the Eiffel object when its retain count becomes greater than 1 and we will unprotect it with `eif_wean()` when its retain count falls back to 1. To do this we need to override the `retain` and `release` methods of subclasses. We only present the implementation of the `retain` method in listing 63 because the code of `release` is similar.

```
id retain_imp(id self, SEL _cmd) {
    if ([self retainCount] = 1) {
        EIF_OBJECT associated_object = (EIF_OBJECT)objc_getAssociatedObject(self,
NULL);
        assert(associated_object);
        EIF_OBJECT object_to_associate = eif_protect(eif_access(associated_object));
        objc_setAssociatedObject(self, (void *)-1, (id)object_to_associate,
OBJC_ASSOCIATION_ASSIGN);
    }
    // Call super implementation
    SEL retain_selector = @selector(retain);
    Method super_retain_method = class_getInstanceMethod([self superclass],
retain_selector);
    IMP super_retain_imp = method_getImplementation(super_retain_method);
    return super_retain_imp(self, retain_selector);
}
```

Listing 63: the `retain_imp` function.

The first thing to notice is that the function is not declared as an Objective-C function but as a C function with 2 parameters. In fact, the implementation of every Objective-C function is a C function accepting at least 2 parameters. The first one is a pointer to the current object (`self`) and the second one is a pointer to the selector of the method that has been called (`_cmd`). `retain_imp()` has not been declared inside an Objective-C class because we need to redefine it for classes we do not know at compile time. Instead, it is contained in the external `objc_callbacks.m` file and declared in `objc_callbacks.h`. We are still able to reference this function from the Eiffel code because it will be compiled as a C library together with the Eiffel executable.

The function first checks whether the retain count of the Objective-C is 1. If this is the case it means the function will increment it to 2 and therefore we need to protect the Eiffel object. We retrieve the Eiffel object associated with the Objective-C object using `objc_getAssociatedObject()` and we protect it with `eif_protect()`. Next, we store the new `EIF_OBJECT` reference in the Objective-C object with a different key (`-1`). The Eiffel function `objc_get_eiffel_object` needs to be updated to first try to get the Eiffel object associated with the key `-1` (if it does not find any reference it will try to retrieve the other reference stored with the key `NULL`). For more information about associative references and keys in Objective-C see [7]. At this point, `retain_imp` simply calls the parent implementation which will increment the retain count by 1.

We now need to add the implementation of `retain_imp()` to every `retain` method of the subclasses. We do this by updating the `allocate_object` function. The added code has been highlighted in listing 64.

```
allocate_object: POINTER
        -- Allocate an Objective-C instance of `Current' and return a pointer
        -- to its address.
    local
        l_objc_class: OBJC_CLASS
    do
        create l_objc_class.make_with_name (get_class_name)
        if is_subclass_instance and not l_objc_class.registered then
            l_objc_class.superclass_objc := create {OBJC_CLASS}.make_with_name
(wrapper_objc_class_name)
            l_objc_class.allocate
                -- Add a custom retain method
            create selector.make_with_name ("retain")
            check l_objc_class.add_method (selector, retain_imp, "@@:") end
                -- Add a custom release method
            create selector.make_with_name ("release")
            check l_objc_class.add_method (selector, release_imp, "v@:") end
            l_objc_class.register
        end
        check l_objc_class_registered: l_objc_class.registered end
        Result := objc_alloc (l_objc_class.item)
    end
```

Listing 64: the updated `allocate_object` function.

The added code uses the `add_method` feature of `OBJC_CLASS` to add the new implementation of `retain`. `retain_imp` is simply a function that returns a pointer to the `retain_imp()` function in listing 63. The third argument of `add_method` is the type encoding of the method. This is needed by the Objective-C runtime system. For more information about type encodings see [8].

One more thing to consider when subclassing are methods redefinitions. This is quite straightforward to achieve in Eiffel using the `redefine` keyword. If, however, the redefined method is called by an Objective-C function the corresponding Eiffel version will not get called. The chapter 6.3.15 about callbacks discusses and solves this issue.

6.3.15 Callbacks

Like in many other object-oriented frameworks, the Hollywood principle applies: *do not call us, we will call you*. That is to say the user does not generally add functionality to an application by instantiating classes of the framework and calling their methods. Contrarily, he inherits from them and redefines existing methods that act as placeholders to add functionality. The framework will automatically call those pieces of code. In Cocoa, for example, in order to implement a view that draws a custom shape, the user needs to inherit from `NSView` and redefine the `drawRect:` method. The user does not need to call `drawRect:` directly, the framework will do that automatically when it is needed.

Unfortunately, if an Eiffel object inheriting from `NS_VIEW` redefines `draw_rect_` it will not automatically receive calls coming from the Objective-C Cocoa framework. This does not only apply to redefined methods. It applies to every Eiffel method we would like to receive Objective-C calls from. This is a quite frequent pattern, for instance when specifying the function to call when a `NS_TIMER` fires or to receive `NS_EVENT`s from the Cocoa Event System like a mouse down event.

The first idea was to use trampolines [9]. This, however, would have been a highly platform dependent solution because we would have needed to deal with assembly code and low-level arguments passing in the x64 architecture which is much more complicated compared to the x86 architecture.

The second idea was to generate for each possible callback a C function that packs the arguments in an array and passes that array to a corresponding eiffel function that will unpack the arguments and will call the actual Eiffel method. This, however, was not feasible because it would have generated too much code (1 additional C and Eiffel function for each method).

Both the first and second idea have been re-though several times to find possible optimizations. However, the biggest downside they share is that they only generate support for callbacks of routines generated by the tool. If a user wanted an Objective-C function to call back an Eiffel function that was not already generated by the tool (this is a recurrent pattern in Cocoa) he would have needed to write all the C and Eiffel code to create a bridge for the callback. This is a poor developing experience.

Therefore, we finally though of a solution to make the process of adding Eiffel functions that can be called from Objective-C as seamless as possible.

Listing 65 shows how to make an Eiffel feature callable from Objective-C.

```
class
    MY_VIEW
inherit
    NS_VIEW redefine make, draw_rect_ end
create
    make
feature {NONE} -- Initialization
    make
            --
        do
            add_objc_callback ("drawRect:", agent draw_rect_)
            Precursor
        end
feature -- Drawing
    draw_rect_ (a_rect: NS_RECT)
            --
        do
                -- Draw a shape
        end
```

Listing 65: the code of the `MY_VIEW` class.

The code declares a class named `MY_VIEW` which inherits from `NS_VIEW` and redefines the `draw_rect_` method to draw a custom shape. In order to redirect calls from the `drawRect:` method of the associated Objective-C object to `draw_rect_` the user simply needs to redefine an initializer (e.g. `make`) and use the `add_objc_callback` procedure to specify the name of the Objective-C selector they want to redirect the calls from and an Eiffel agent pointing to the routine they want to be called. When the user is done adding callbacks, he needs to invoke the parent implementation of the creation procedure using the `Precursor` keyword. The user does not have to worry about anything else. The callback system will take care of the memory management, arguments wrapping/unwrapping, return values, etc. At the moment, callbacks support arguments of basic type (`int`, `double`, etc.),

objects and structs. The supported return types are objects and basic types except doubles. It is, however, easy to extend the callback system to support other return types.

We now describe the callback system. The add_objc_callback procedure is declared in NS_COMMON. When a callback is added using that procedure it is saved in the objc_callbacks hash table declared in NS_COMMON. The hash table is indexed by Objective-C selectors and the elements are tuples. Each tuple contains the Eiffel agent to call back, its return type, an array with the type of the arguments and the Objective-C type encoding of the method. Listing 66 shows the add_objc_classback function.

```
add_objc_callback (selector_name: STRING; eiffel_callback: ROUTINE [ANY, TUPLE])
        -- Add an Objective-C callback associated to an eiffel function to the
current class.
    require
        is_subclass_instance: is_subclass_instance
    local
        selector: OBJC_SELECTOR
        return_type: detachable TYPE [detachable ANY]
        arguments_types: ARRAY [TYPE [detachable ANY]]
        argument_type: TYPE [detachable ANY]
        arguments_tuple: TYPE [detachable ANY]
        objc_encodings: STRING
        i: INTEGER
    do
        create selector.make_with_name (selector_name)
        create arguments_types.make_empty
        create objc_encodings.make_empty
        if attached {PREDICATE [ANY, TUPLE]} eiffel_callback as
eiffel_predicate_callback then
            return_type := {BOOLEAN}
            objc_encodings.append (objc_encoding_for_eiffel_type (return_type))
        elseif attached {FUNCTION [ANY, TUPLE, detachable ANY]} eiffel_callback as
eiffel_function_callback then
            return_type :=
eiffel_function_callback.generating_type.generic_parameter_type (3)
            objc_encodings.append (objc_encoding_for_eiffel_type (return_type))
        else
            check attached {PROCEDURE [ANY, TUPLE]} eiffel_callback end
            objc_encodings.append ("v")
        end
        objc_encodings.append ("@:")
        arguments_tuple := eiffel_callback.generating_type.generic_parameter_type (2)
        from
            i := 1
        until
            i > arguments_tuple.generic_parameter_count
        loop
            argument_type := arguments_tuple.generic_parameter_type (i)
            arguments_types.force (argument_type, i)
            objc_encodings.append (objc_encoding_for_eiffel_type (argument_type))
            i := i + 1
        end
        objc_callbacks.force ([[return_type, arguments_types], objc_encodings,
eiffel_callback], selector.item)
    end
```

Listing 66: the `add_objc_callback` procedure.

The procedure first determines what the return type of the Eiffel callback is. Next, it does the same for the arguments. Finally, it adds the information in the `objc_callbacks` hash table. In order to compute the Objective-C type encoding of each Eiffel type we use the `objc_encoding_for_eiffel_type` function shown in listing 67.

```
objc_encoding_for_eiffel_type (eiffel_type: detachable TYPE [detachable ANY]): STRING
    do
        create Result.make_empty
        if
            attached {TYPE [BOOLEAN]} eiffel_type or
            attached {TYPE [CHARACTER_8]} eiffel_type or
            attached {TYPE [INTEGER_8]} eiffel_type
        then
            Result.append ("c")
        elseif attached {TYPE [INTEGER_16]} eiffel_type then
            Result.append ("s")
        elseif attached {TYPE [INTEGER_32]} eiffel_type then
            Result.append ("i")
        elseif attached {TYPE [INTEGER_64]} eiffel_type then
            Result.append ("q")
        elseif attached {TYPE [NATURAL_8]} eiffel_type then
            Result.append ("C")
        elseif attached {TYPE [NATURAL_16]} eiffel_type then
            Result.append ("S")
        elseif attached {TYPE [NATURAL_32]} eiffel_type then
            Result.append ("I")
        elseif attached {TYPE [NATURAL_64]} eiffel_type then
            Result.append ("Q")
        elseif attached {TYPE [REAL_32]} eiffel_type then
            Result.append ("f")
        elseif attached {TYPE [REAL_64]} eiffel_type then
            Result.append ("d")
        elseif attached {TYPE [detachable OBJC_CLASS]} eiffel_type then
            Result.append ("#")
        elseif attached {TYPE [detachable OBJC_SELECTOR]} eiffel_type then
            Result.append (":")
        elseif attached {TYPE [detachable CG_POINT]} eiffel_type then
            Result.append ("{CGPoint=dd}")
        elseif attached {TYPE [detachable CG_SIZE]} eiffel_type then
            Result.append ("{CGSize=dd}")
        elseif attached {TYPE [detachable NS_RANGE]} eiffel_type then
            Result.append ("{_NSRange=QQ}")
        elseif attached {TYPE [detachable CG_RECT]} eiffel_type then
            Result.append ("{CGRect={CGPoint=dd}{CGSize=dd}}")
        elseif attached {TYPE [detachable NS_DECIMAL]} eiffel_type then
            Result.append ("{?=b8b4b1b1b18[8S]}")
        elseif attached {TYPE [detachable NS_AFFINE_TRANSFORM_STRUCT]} eiffel_type then
            Result.append ("{?=dddddd}")
        elseif attached {TYPE [detachable AE_DESC]} eiffel_type then
            Result.append ("{AEDesc=I^^{OpaqueAEDataStorageType}}")
        elseif attached {TYPE [detachable CG_AFFINE_TRANSFORM]} eiffel_type then
            Result.append ("{CGAffineTransform=dddddd}")
        elseif attached {TYPE [detachable CA_TRANSFORM3D]} eiffel_type then
            Result.append ("{CATransform3D=dddddddddddddddd}")
        elseif attached {TYPE [detachable CV_SMPTE_TIME]} eiffel_type then
            Result.append ("{CVSMPTETime=ssIIIssss}")
        else
                -- Assume it's an Objective-C wrapped object
            Result.append ("@")
        end
    end
```

Listing 67: the `objc_encoding_for_eiffel_type` function.

For each Eiffel type the function returns the specific Objective-C type encoding. Structs have a particular type encoding so each struct has its dedicated if. The `ifs` for the structs are generated automatically by the wrapper generator based on the struct declarations it finds. For more information about type encodings see [8].

We now give an overview of the path a callback runs through. Let's consider the situation shown in figure 3.

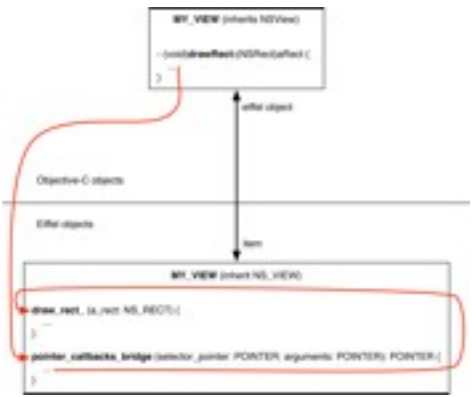

Figure 3: overview of the path of a callback.

An Eiffel object of type `MY_VIEW` and subclass of `NS_VIEW` is wrapping the corresponding instance of the Objective-C class `MY_VIEW` (subclass of `NSView`). The implementation of `drawRect:` has been installed with a C function with variable arguments that packs all its arguments in a C array and forwards them to the `pointer_callbacks_bridge` function of the corresponding Eiffel object. `pointer_callbacks_bridge` will then unpack the arguments and wrap them, if necessary, taking care of memory management issues. Finally, `pointer_callbacks_bridge` will call the `draw_rect_` agent with the unpacked arguments. Note that the C function used to hijack the call to `drawRect:` and redirect it to `pointer_callbacks_bridge` is the same for all callbacks. It is defined in the external `objc_callbacks.m` file. The `pointer_callbacks_bridge` name starts with *pointer* because that is its return type (`void *`). This means it can be used for callbacks returning objects, selectors and basic types except `doubles`. It also can't be used for callbacks returning structs. We designed the callbacks system to be easily extendible. Therefore, it is relatively easy to add support for callbacks `doubles`. In the following paragraphs we will describe the callbacks system in more detail.

In order to install the hijacking function in the callbacks we need to update the `allocate_object` function as shown in listing 68. The added code has been highlighted.

```
allocate_object: POINTER
        -- Allocate an Objective-C instance of `Current' and return a pointer
        -- to its address.
    local
        l_objc_class: OBJC_CLASS
    do
        create l_objc_class.make_with_name (get_class_name)
        if is_subclass_instance and not l_objc_class.registered then
            l_objc_class.superclass_objc := create {OBJC_CLASS}.make_with_name
(wrapper_objc_class_name)
            l_objc_class.allocate
                -- For each callback
            across objc_callbacks as objc_callbacks_cursor loop
                    -- Add a callback hijacker
                create selector.make_with_pointer (objc_callbacks_cursor.key)
                callbacks_hijacker := pointer_callbacks_hijacker
                check l_objc_class.add_method (selector, callbacks_hijacker,
objc_callbacks_cursor.item.objc_encoding) end
            end
                -- Add a custom retain method
            create selector.make_with_name ("retain")
            check l_objc_class.add_method (selector, retain_imp, "@@:") end
                -- Add a custom release method
            create selector.make_with_name ("release")
            check l_objc_class.add_method (selector, release_imp, "v@:") end
            l_objc_class.register
        end
        check l_objc_class_registered: l_objc_class.registered end
        Result := objc_alloc (l_objc_class.item)
    end
```

Listing 68: the updated `allocate_object` function.

The added code loops through all the user-added callbacks. For each callback we use the `add_method` procedure of `OBJC_CLASS` to add a method with the selector of the callback and our hijacking function as implementation (`pointer_callbacks_hijacker`). `pointer_callbacks_hijacker` is simple a C external that returns a C function pointer to the `pointer_callbacks_hijacker` function defined in `objc_callbacks.m`. Another change we need to do concerns `make_with_pointer`. The added code has been highlighted in listing 69.

```
make_with_pointer (a_pointer: POINTER)
        -- Initialize `Current' with `a_pointer'.
    require
        a_valid_pointer: a_pointer /= default_pointer
    do
        item := a_pointer
        if attached objc_get_eiffel_object (item) then
            is_shared_objc_object := True
        else
            objc_set_eiffel_object (item, $Current)
        end
        if is_subclass_instance then
            objc_connect_callbacks_bridge (item, $pointer_callbacks_bridge, 1)
        end
    ensure
        item_set: item = a_pointer
    end
```

Listing 69: the `make_with_pointer` creation procedure.

The code basically stores a function pointer to the Eiffel feature `pointer_callbacks_bridge` in the wrapped Objective-C object such that we can call that Eiffel function from Objective-C.

We can now present the code of the C function declared in `objc_callbacks.m` that redirects Objective-C calls to the Eiffel callbacks bridge feature. Listing 70 shows the first part of the function.

```
void * pointer_callbacks_hijacker (id self, SEL _cmd, ...) {
    va_list variableArguments;
    va_start(variableArguments, _cmd);
    NSMethodSignature *methodSignature = [self methodSignatureForSelector:_cmd];
    int argumentsCount = [methodSignature numberOfArguments];
    void **arguments = NULL;
    if (argumentsCount > 2) {
        arguments = malloc(sizeof(void *) * (argumentsCount - 2)); // Freeing is done in eiffel
code.
    }
    int i;
    for (i = 2; i < argumentsCount; i++) {
        const char *argumentType = [methodSignature getArgumentTypeAtIndex:i];
        BOOL argumentRead = YES;
        if (argumentType[0] == '@') {
            arguments[i - 2] = (void *)va_arg(variableArguments, void *);
        }
        switch (argumentType[0]) {
            case 'c':
            case 'i':
            case 's':
            case 'C':
            case 'I':
            case 'S': {
                int *value_pointer = malloc(sizeof(value_pointer)); // Freeing is done in eiffel
code.
                *value_pointer = va_arg(variableArguments, int);
                arguments[i - 2] = value_pointer;
                break;
            }
            case 'q':
            case 'Q': {
                long long *value_pointer = malloc(sizeof(value_pointer)); // Freeing is done in
eiffel code.
                *value_pointer = va_arg(variableArguments, long long);
                arguments[i - 2] = value_pointer;
                break;
            }
            case 'f':
            case 'd': {
                double *value_pointer = malloc(sizeof(value_pointer)); // Freeing is done in eiffel
code.
                *value_pointer = va_arg(variableArguments, double);
                arguments[i - 2] = value_pointer;
                break;
            }
            case '@':
            case '#':
            case ':': {
                void *object = va_arg(variableArguments, void *);
                arguments[i - 2] = object;
                break;
            }
            default: {
                argumentRead = NO;
                break;
            }
        }
```

Listing 70: the first part of the `pointer_callbacks_hijacker()` function.

The function returns a pointer (`void *`) and it accepts at least 2 arguments. The implementation of every Objective-C instance method is actually a C function accepting at least 2 arguments. The first one is a reference to the object the selector has been called on (`self`), the second one is the selector itself (`_cmd`). For more information see [8]. Because of this reason the first 2 arguments of `pointer_callbacks_hijacker` are declared just like that. However, `pointer_callbacks_hijacker` can accept a variable number of arguments. This way we can use only one function to redirect every kind of callback (except those returning structs or `doubles`).

The first thing the function does is to retrieve the method signature associated with the called method. Next it loops though all arguments and retrieves them using the C `va_arg` macro. Because `va_arg` needs the type of the argument to retrieve we need to query the Objective-C runtime system to get the type encoding of each argument. Based on the type encoding we can specify a type for `va_arg` and store a pointer to the value it returns in the `arguments` array that we previously allocated (note that the array is freed later by some Eiffel code it is passed to). Structs arguments are a slightly special case. The code of the function does not know a priori the structs declared in the Objective-C framework. Therefore, when the wrapper generator is translating an Objective-C framework to Eiffel it will modify the `objc_callbacks.m` file and add the code to extract the structs it found.

The final part of the `pointer_callbacks_hijacker` function is shown in listing 71 (we omitted the part that extracts structs with `va_arg`).

```
typedef void * (*eiffel_callback_bridge_type)(EIF_REFERENCE, EIF_POINTER,
EIF_POINTER);
eiffel_callback_bridge_type eiffel_callback_bridge = (eiffel_callback_bridge_type)
objc_getAssociatedObject(self, (void *)1);
// Get the EIF_OBJECT.
// First try to get the EIF_OBJECT created with eif_protect.
EIF_OBJECT eiffel_object = (EIF_OBJECT)objc_getAssociatedObject(self, (void *)-1);
if (eiffel_object == nil) {
        // If there is no eiffel object created with eif_protect() associated with
self
        // then there is one created with eif_create_weak_reference().
        eiffel_object = (EIF_OBJECT)objc_getAssociatedObject(self, NULL);
}

return eiffel_callback_bridge(eif_access(eiffel_object), _cmd, arguments);
```

Listing 71: the final part of the `pointer_callbacks_hijacker()` function.

The code first retrieves the address of the Eiffel function to forward the arguments to (`pointer_callbacks_bridge`, see Figure 3). Then it retrieves the Eiffel object associated with the current Objective-C object. Finally, it invokes the Eiffel function `pointer_callbacks_bridge` passing the selector and the arguments array. It is important to note that we also return the value returned by that Eiffel function. This allows us to support callbacks that return data. If a callback does not return any data the returned data is simple ignored.

The `pointer_callbacks_bridge` function (defined in `NS_COMMON`) is too long to be listed here. We only give a brief description of it. The first thing it does is to retrieve the information associated with the passed selector by querying the `objc_callbacks` hash table described earlier in this chapter (note that every Eiffel

wrapping object has its own dedicated hash table). Then it iterates though the passed `arguments` array extracting them. Arguments of basic type con be easily assigned to Eiffel types like integers, reals, booleans, etc. However, structs and objects need a special treatment because they need to be wrapped by an Eiffel object. If an Eiffel object is already wrapping the extracted Objective-C object then it is reused (this way we avoid to create multiple Eiffel objects pointing to the same Objective-C object and therefore satisfying the identity property). Next, we call the actual Eiffel callback (we have this information because we extracted it from the `objc_callbacks` hash table). If the Eiffel callback is a procedure we do not need to do additional tasks. If, instead, it is a function we need to convert the returned value to `POINTER` type (this is the return type of `pointer_callbacks_bridge`) and to return that. The C function `pointer_callbacks_hijacker` defined in the `objc_callbacks.m` will return that value to the Objective-C caller of the callback.

6.4 Objective-C Categories

In this chapter we will explain how we mapped Objective-C categories that have been defined in a framework different than the one of the original class (i.e. the purpose of the category was extending the original class).

Let us consider the example of `NSString`. `NSString` is declared in the Foundation framework. However, the AppKit framework declares a category on `NSString` called `NSStringDrawing`. `NSStringDrawing` declares methods such as `drawInRect:withAttributes:`, i.e. methods to draw strings. These methods do not really belong to the `NSString` class, yet they are not declared in a subclass. They are used as utility methods for `NSString`. Therefore, we decided to map categories to utility classes in the manner described in the following paragraph.

First, we convert the Objective-C category name to Eiffel underscore style appending the `_CAT` suffix that stands for category. Next, we generate the Eiffel class inheriting from `NS_CATEGORY_COMMON`. For each method declared in the category we generate an external as described in chapter 6.3. We also generate a wrapper function. Listing 72 shows an example for the `drawInRect:withAttributes:` method.

```
draw_in_rect__with_attributes_ (a_ns_string: NS_STRING; a_rect: NS_RECT; a_attrs:
NS_DICTIONARY)
        -- Auto generated Objective-C wrapper.
    do
        objc_draw_in_rect__with_attributes_ (a_ns_string.item, a_rect.item,
a_attrs.item)
end
```

Listing 72: generated code for the Objective-C `drawInRect:with:Attributes:` category method.

The first argument of the function is the `NS_STRING` object we want to call the method on, the others are the method arguments. The wrapper function generation is the same as the one described in chapter 6.3, except the first argument of the external call is not `item` but rather the `item` attribute of the first argument. All the considerations about memory management of chapter 6.3 apply here too.

6.5 Objective-C Protocols

6.5.1 Introduction

Objective-C protocols are similar to Java interfaces: they declare methods that others are expected to implement. Concretely, a protocol is a list of methods declarations, unattached to a class definition. Methods declared in a protocol can be required or optional. Required methods must be implemented by classes adopting the protocol, optional methods do not need to be implemented. Listing 73 shows an example of protocol declaration.

```
@protocol MyProtocol

- (void)requiredMethod;

@optional
- (void)anOptionalMethod;
- (void)anotherOptionalMethod;

@required
- (void)anotherRequiredMethod;

@end
```

Listing 73: example of protocol declaration.

Classes can adopt a protocol with the syntax shown in listing 74.

```
@interface ClassName : ItsSuperclass <protocol list>
```

Listing 74: example syntax to adopt protocols for classes.

Categories adopt protocols in a similar manner as classes (see listing 75).

```
@interface ClassName (CategoryName) <protocol list>
```

Listing 75: example syntax to adopt protocols for categories.

A class (or category) can adopt more than one protocol. Names in the protocol list are separated by commas as shown in listing 76.

```
@interface ClassName : NSObject < Protocol1, Protocol2 >
```

Listing 76: a class adopting 2 protocols.

A class or category that adopts a protocol must implement all of its required methods, otherwise the compiler issues a warning.

Protocols can incorporate other protocols using the same syntax that classes use to adopt a protocol (listing 49).

```
@protocol ProtocolName < protocol list >
```

Listing 49: example syntax to adopt protocols for protocols.

6.5.2 Mapping Objective-C Protocols

For each Objective-C protocol we generate a deferred Eiffel class. The Eiffel class name is the Objective-C protocol name converted to underscore capital case syntax along with the `_PROTOCOL` suffix. Objective-C protocols that do not incorporate other protocols inherit from `NS_COMMON`, otherwise they inherit from the protocol classes they incorporate.

Required methods are generated the same way instance methods are (see chapter 6.3). Optional methods are generated in a similar manner but with a slight exception. Let's consider the example shown in listing 78.

```
optional_method
        -- An optional method
    require
        has_optional_method: has_optional_method
    do
        [...]
    end
```

Listing 78: generated code for an optional method of an Objective-C protocol.

The generated optional method has a precondition: `has_optional_method` (the name is generated by simply prepending the prefix `has_` to the feature name). This is a feature clients of the protocol can override if they wish to implement `optional_method`. The default implementation of `optional_method` is to simply call `optionalMethod` on the wrapped Objective-C object. In order to know if the wrapped Objective-C object implements the optional method or not we need to query it at runtime in the `has_optional_method` feature. Listing 79 shows the default implementation of `has_optional_method`.

```
feature -- Status Report

has_optional_method: BOOLEAN
        -- [...]
    do
        Result := objc_has_optional_method (item)
    end

feature {NONE} -- Implementation

objc_has_optional_method (an_item: POINTER): BOOLEAN
        -- [...]
    external
        "C inline use <Foundation/Foundation.h>"
    alias
        "[
            return [(id)$an_item respondsToSelector:@selector(optionalMethod)];
        ]"
    end
```

Listing 79: generated code to check whether an optional protocol method has been implemented or not.

The `objc_has_optional_method` feature shown in listing 79 uses the dynamic nature of Objective-C to check whether the wrapped Objective-C object implements the optional method. It does so by using the `respondsToSelector:` method of `NSObject`.

6.5.3 Name Clashes

It is very likely to have features from different classes or protocols with the same name. Therefore, classes inheriting from different protocols and classes will be very likely to have feature name conflicts. The generator automatically picks a version of the feature from the inherited classes and protocols and un-defines it in the other parents.

7 Developers Guide

In this chapter we are going to give a brief description of the wrapper generator folder.

The wrapper generator folder contains an Eiffel project along with other folders. The `templates` folder is used to store Eiffel classes (e.g. the `NS_COMMON` class) to be used as templates when generating an Objective-C wrapper. The `objc_wrapper` folder is a generated and ready-to-use Eiffel version of the Cocoa framework. This is the folder the wrapper generator automatically creates when it is executed. The `testing` folder in the wrapper generator folder contains tests about structs, objects creation, class clusters, memory management, subclassing and callbacks. All the tests have been run under Mac OS X 10.6.4 64-bit and they all passed. We run the tests setting the `debug` feature in `NS_COMMON` to see when objects are retained/released by Eiffel objects. The `examples` folder contains two example applications, namely `EmptyCocoaApp` – a starting point for creating Cocoa applications using Eiffel – and `CircleApp` – an example Cocoa application to demonstrate to use of the generated Cocoa framework for Eiffel.

The wrapper generator has been created and tested using Mac OS 10.6.4 and EiffelStudio 6.7.8.4353. In this version there is a garbage collector bug that can cause unexpected crashes of Eiffel programs using a converted Objective-C framework.

The wrapper generator tool is internally organized in 2 parts: the parser and the generator. The parser consists of several classes for Objective-C and Eiffel entities. For each of these entities there are visitors that provide an easy and convenient way to traverse them. The generator consists of several visitor classes that are used to visit Objective-C entities and output Eiffel classes.

8 Users Guide

This chapter explains how to use the wrapper generator and the generated Cocoa framework.

8.1 Wrapper Generator

When using the wrapper generator to translate an Objective-C framework to Eiffel you can specify the Objective-C framework to translate in the `SHARED_CONFIGURATION` file.

8.2 Generating A Cocoa Wrapper

There are some known parsing problems when parsing the Cocoa framework. The following paragraph describes a workaround to make it parse anyway.

Make a copy of the `/System/Library/Frameworks` folder and change the `frameworks_path` setting in `SHARED_CONFIGURATION` to the new location of the frameworks folder. In order for the parser to correctly parse the header files make the following changes.

- In `AppKit/NSMatrix.h`, line 216: add an end of line before `@end`.
- In `QuartzCore/CIImage.h`: inline all method declarations.
- In `AppKit/NSTextField.h`, line 62, add an end of line before `@end`.
- In `AppKit/NSTextView.h`: rewrite the interface of `NSTextView` such that it will be on a single line after it is preprocessed.
- In `AppKit/NSBezierPath.h`: inline all method declarations.
- In `QuartzCore/CALayer.h`, line 714: inline method declaration.
- In `QuartzCore/CAOpenGLLayer.h`: inline all method declarations.
- In `QuartzCore/CALayer.h`, line 359: split property declaration in 2.

After running the wrapper generator execute the command `finish_freezing -library` with a terminal in the `objc_wrapper/Clib` folder. Then set the right header files in the externals of the generated structs (the wrapper generator cannot guess them), import all needed frameworks headers in objc_callbacks.h, set the right field names in the classes wrapping structs (contained in the structs folder) and substitute the following files.

- `objc_wrapper/Foundation/ns_string.e` with `templates/Optional classes to manually add/ns_string.e`. This version of `NS_STRING` has been manually changed to support seamless conversion between Eiffel and Cocoa strings.
- `objc_wrapper/AppKit/ns_layout_manager.e` with `templates/Optional classes to manually add/ns_layout_manager.e`. A method of that class has been uncommented in order to use it in the example application `CircleApp`.

8.3 Using The Generated Cocoa Framework

When using the generated Cocoa framework you should pay attention to the following points.

- Objective-C identifiers conflicting with Eiffel keywords or existing methods from `ANY` have been appended with an underscore symbol (_).
- Subclassing structs is not supported.
- Be aware of Objective-C retain cycles. This is usually not a problem.

- Always check the Apple documentation when subclassing wrapped classes to see if there are any special methods you must redefine (especially for class clusters).
- Do not call `deep_twin` or `deep_copy` on wrapped objects.
- Invoking `copy` on wrapped objects is not yet supported.
- To add a callback from Objective-C to an Eiffel function use the `add_objc_callback` function in a creation procedure. When you are done adding callbacks you must invoke the parent implementation of the creation procedure using the `Precursor` keyword. Make sure the signature of the routine pointed by the Eiffel agent is compatible with the signature of the Objective-C callback. Note that callbacks returning `double`s or structs are not supported yet.
- Be aware that fields of wrapped structs are always returned as a copy. Therefore the code `a_rect.size.width := 20` is not going to change the width of `a_rect`.
- Be aware of a rare bug affecting `va_arg` in callbacks. It seems like the `va_arg` implementation for 64 bit machines is buggy.

9 Conclusion

Thanks to the tool we developed we opened the door to a whole new world for Eiffel developers. They can finally use Cocoa or any other Objective-C framework to reach a huge amount of new users and devices (e.g. Mac, iPods, iPhones, iPads, etc.) producing native applications that excel in interface design. This is a crucial step for every product: a user interface that is unattractive or convoluted can make even a great application an awful experience to use. But a beautiful, intuitive, compelling user interface enhances an application's functionality and inspires a positive emotional attachment in users.

Figure 4 shows the screenshot of a very simple demo Cocoa application we wrote exclusively in Eiffel. It can be found in `examples/CircleApp`.

Figure 4

10 Future Work

EiffelVision 2 does not have a Cocoa-based implementation yet. This is definitely something that could be done in the future.

Other improvements to the wrapper generator could be the following.

- Adding support for copying objects implementing `NS_COPYING_PROTOCOL`.
- Adding support for callbacks returning `double`s and structs.
- Do not return structs fields by copy (this requires a retain count mechanism for structs too).

11 References

[1] Bertrand Meyer: Object-Oriented Software Construction, 2nd edition, Prentice Hall, 1997.

[2] EiffelVision 2, http://docs.eiffel.com/book/solutions/eiffelvision-introduction

[3] EiffelStudio, http://eiffelstudio.origo.ethz.ch/

[4] Vision4Mac, Carbon implementation http://vision4mac.origo.ethz.ch/

[5] EiffelCocoa, http://www.maceiffel.com/

[6] Daniel Furrer: EiffelVision for Mac OS X, 2009; online at: http://se.ethz.ch/projects/daniel_furrer/project%20plan.pdf

[7] Apple, The Objective-C Programming Language; online at: http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf

[8] Apple, Objective-C Runtime Programming Guide; online at: http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/ObjCRuntimeGuide.pdf

[9] Eiffel callback trampolines; online at: http://www.eiffelroom.org/node/405

[10] Cocoa Fundamentals Guide; online at: http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/CocoaFundamentals/Introduction/Introduction.html