Chair of Software Engineering

EiffelVision for Mac OS X

Master Thesis

Daniel Furrer
ETH Zurich
dfurrer@ethz.ch

April 16, 2009 - September 16, 2009

Supervised by:
Benjamin Morandi
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

EiffelVision is an object-oriented framework for graphical user interface development, originally created by Eiffel Software. Although generally designed to be platform independent, there is currently no native version for Mac OS X. While the GTK+/X11 based version (used on Linux) works on Mac OS X as well it has many problems which cannot be overcome with the current architecture. For this reason we developed an implementation of EiffelVision which is based on Cocoa - the native graphics framework of Mac OS X. The development of this implementation also led to an Eiffel based Cocoa framework which can be used to write native applications on Mac OS X and inspired the development of an iPhone library for Eiffel.

Acknowledgments

My gratitude goes to my supervisor Benjamin Morandi and to Emmanuel Stapf from Eiffel Software for their continuous support, interesting discussions and valuable feedback, to Prof. Bertrand Meyer for giving me the possibility to work on this interesting topic, and to my family and friends who supported me during my whole time at ETH.

Contents

Chapter 1

Introduction

EiffelVision is an object-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*

Currently EiffelVision on Mac OS X has many problems, many of which can be attributed to it's GTK+/X11 based nature. It is not able to integrate with the Mac OS X platform as a whole and could never do so with the current approach. One example is that copy&paste commands in EiffelVision could not use the regular key-bindings and would sometimes miss line breaks. Other examples of problematic areas with this implementation include multi-screens setups and the handling of file associations. Because of the dependency on GTK+, which depends on a lot of other packages itself, we were not able to provide a simple binary installation package for EiffelStudio on Mac OS X.

1.1 Goal

The native programming environement for Mac OS X is called Cocoa and it is centered around the Objective-C programming language. It was the goal of this thesis to write a Cocoa based implementation of EiffelVision thus ultimately getting rid of the GTK+ dependency of EiffelVision on Mac OS X.

1.2 Outline

In chapter 2 we present an overview of the architecture and how we can split up the system in three major layers. Chapters 3, 4 and 5 discuss our solutions for each of these layers as well as the implementation issues we encountered. We conclude in chapter 6 and present some ideas for future work in chapter 7.

1.3 Related Work

1.3.1 Vision4Mac

Vision4Mac is a Carbon based EiffelVision2 implementation that was never completed. Carbon was originally a legacy API which allowed pre-OS X applications to be ported to OS X without a major rewrite. For a while it was unclear what the future of Carbon would be as Apple continued to add new functionality. With it's latest operating system Apple has dropped support for user interface programming using Carbon which effectively makes this implementation obsolete.

1.3.2 MacEiffel

MacEiffel comes as two Eiffel libraries called EiffelCocoa and EiffelCarbon that wrap Cocoa respectively Carbon. The project has not seen any major updates since 2002 and could not be used as the base of our work, because it was not available under an open source license.

1.3.3 Other Objective-C/Cocoa Bridges

Several other language bindings for Cocoa exist, most notably the D/Objective-C bridge[1] and monoobjc[3] for bridging between .NET and Objective-C.

Chapter 2

Architecture

2.1 Overview

The whole system consists of three major layers. We begin with a short overview of the responsibilities of each layer and then proceed by explaining them in more details in the next three chapters.

1. **Bridging between Eiffel and Objective-C**
   This lowest layer takes care of the direct interfacing with the Objective-C language. Our solution handles the gap between Eiffel and Objective-C by mapping Eiffel objects to Cocoa objects and the other way around. Two important concerns are handled in this layer: Memory management and callbacks from Objective-C to Eiffel. We also provide high-level Eiffel abstractions of Objective-C's language primitives where necessary.

2. **Wrapper for Objective-C GUI classes** (Cocoa)
   This is a set of classes that wrap most of Cocoa's Objective-C classes to Eiffel classes. Using the classes in this layer is similar to the way their counterparts are used in Objective-C. There are a few places however, where we replaced Cocoa design patterns by something which makes more sense in the context of Eiffel.

3. **Cocoa based EiffelVision implementation**
   The final layer is the implementation of EiffelVision for Cocoa based on the previous two layers. Thanks to the previous abstractions the interaction with Objective-C and Cocoa is usually straight forward. The focus here is on implementing the features required by EiffelVision with the facilities provided by the Cocoa framework. This entails the mapping between widgets and widget structures, the transformation of coordinates, events, etc.

Chapter 3

The Eiffel/Objective-C Bridge

The objc_base library is the core of the bridge between Eiffel and Objective-C. It presents a low-level interface to the Objective-C language and its run-time system, including classes for accessing and manipulating Objective-C primitives such as classes, selectors and messages. It handles the mapping between Eiffel and Objective-C objects, the callbacks from Objective-C to Eiffel, the conversion of types and memory management.

3.1 Duality of Classes (and Objects)

For every Objective-C class we created an associated Eiffel class which offers mostly the same functionality. Moreover, each Objective-C object will have its dual wrapper object in Eiffel, and we will map the calls between the two sides so that we can call Objective-C messages from Eiffel features and vice versa. Our Cocoa framework can thus be used like any other Eiffel framework and the binding to Objective-C happens transparently, without any intervention or required knowledge of the user. In the following sections we will explain the details of how this mapping works. More information about the generation fo those wrapper classes can be found in section 4.1.

3.2 Foundation Framework

Also part of this layer is the equivalent of the Foundation framework on Mac OS X. It contains the core classes which are, strictly speaking, not part of the language itself, but work closely with it and provide the basis for all other frameworks. The Foundation framework contains the root class `NSObject`, some basic protocols like `NSCopying` and various standard library classes such as `NSArray`, `NSDictionary` or `NSSet`.

3.3 Short Comparison of Objective-C and Eiffel

Objective-C is the language of choice for user interface programming on Mac OS X (or perhaps the only choice) and we will use many of the basic and also some of the more advanced features that it offers. It is important to understand the basic concepts and the differences with respect to Eiffel to understand some of the design decisions we had to take. A very short overview is presented in table 3.1.

	Eiffel	**Objective-C**
Object-oriented	inheritance, dynamic binding, polymorphism	
Inheritance	multiple	single (protocols, categories)
Type system	static strong (including genericity)	dynamic(/static)[a] weak (no genericity)
Memory management	garbage collector	reference counting or garbage collector
Other	Design by Contract agents Eiffel method	delegation pattern[b] reflection dynamic loading

[a]Objective-C's type system was originally dynamic but some optional static type checks have been added.
[b]Not to be confused with C# delegates

Table 3.1: Comparison chart of Eiffel and Objective-C

Both languages are object-oriented and they both appeared around the same time in 1986. Eiffel was developed as a new programming language, free of concerns for supporting legacy code, whereas Objective-C is designed as a superset of C to which many concepts from Smalltalk have been added. Both languages claim to be the holy grail of reusability and share the core principles of object-orientation, namely: inheritance, polymorphism and dynamic binding.

Objective-C uses Smalltalk's message analogy, which means that instead of talking about methods being called on objects we say that a message is sent to an object. Throughout this text we will use the term *message* when talking about Objective-C and *feature* when talking about Eiffel. In Objective-C, classes are objects themselves and thus may implement messages as well (similar to `static` members in C++ or Java). These static methods are used to create objects or get references to global objects and values.

The simple example in listing 3.1 demonstrates how a class message is used to create an object and how a message can be sent to that object. The syntax of Objective-C is inspired by Smalltalk and might look strange to programmers used to the dot-syntax at first. Another peculiarity is the use of named parameters. For example, the full name of the first message in listing 3.1 is

`createWithInitial:andOverdrawLimit:`.

Listing 3.1: Objective-C syntax

```
1  BankAccount* account =
2    [BankAccount  createWithInitial: 50 andOverdrawLimit: -1000];
3  [account deposit: 50];
```

Eiffel works with *references* and uses a garbage collector to automatically free unused memory. In Objective-C the programmer deals with *pointers* directly and two ways of memory management are available: Reference Counting and Garbage Collection. Garbage collection was added to Objective-C recently and is not supported on some platforms, such as the iPhone.

3.4 Calling an Objective-C Function from Eiffel

The Eiffel standard defines the `external` keyword to be used for the integration of code written in other languages into Eiffel source code. We have a simple patch that adds support for '`Objective-C`' to ISE's Eiffel compiler. Its usage is illustrated in listing 3.2.

Listing 3.2: Calling an Objective-C function

```
1  array_object_at_index (target: POINTER; a_index: INTEGER):
       POINTER
2    external
3      "Objective-C inline use <Foundation/NSArray.h>"
4    alias
5      "return [(NSArray*)$target objectAtIndex: $a_index];"
6    end
```

3.5 Names

By convention Objective-C programmers use camelCase in the naming scheme for identifiers whereas Eiffel [5] propagates the more readable spaced_out naming style. `NSWindow` becomes `NS_WINDOW` and `setFrame:` becomes `set_frame`. Where Objective-C uses named parameters in some messages such as `initWithContentRect :styleMask:backing:defer:` we will often drop the named parameters and simply call the feature `init`.

3.6 Types

When going from Objective-C to Eiffel the dynamic and weak type information is replaced by static, strong typing and generic parameters are added where appropriate. Although the Objective-C compiler does not enforce the type constraints as rigorously as the Eiffel compiler, this enforcement is actually not

a problem in practice: We did not run into any limitations with static checking. Moreover we were able to benefit from all the advantages of static typing. Thanks to the Eiffel type system the original type annotations can often be made more precise: First, through the use of genericity (`NSArray` becomes `NS_ARRAY [G]`), second, by adding the void-safety keywords (`detachable` or `attached`) where appropriate.

3.7 Other Language Constructs

3.7.1 Protocols

An Objective-C protocol is a collection of messages, similar to an interface in Java. An class conforms to a protocol if it implements the specified messages. Protocols can be mapped to a deferred classes in Eiffel and conforming to that protocol turns into writing an effective class that inherits from the protocol's deferred class.

3.7.2 Categories

A category is an Objective-C construct to add methods to a class at runtime. The dynamic nature of categories has implications for the separation of concerns. It allows, for example, the UIKit framework to add drawing methods to the Foundation framework's `NSString` without imposing any dependencies on `NSString`. There is no such concept in Eiffel, but fortunately categories are only used sparingly in Cocoa. There is only the single case of `NS_STRING` in our wrapper, where we use inheritance instead.

3.7.3 Contracts

Eiffel allows us to add contracts to features and classes. It is a good idea to take advantage of this. There are several cases where contracts are useful in the bridge:

- to check the range of parameters, especially if the type of a parameter or the type of the result was an `enum` in Objective-C.
- to check some input parameters or parameter combinations for validity if explicitly stated in the documentation of the original Objective-C message.

We do not generally need assertions to check void-values as our wrapper is fully void-safe[7], i.e. we have type annotations where necessary and the type system guarantees all calls to be valid.

3.8 Memory Management

The memory spaces of Eiffel and Objective-C are conceptually separated, which means that the memory management systems of either side do not follow references into the other world. Therefore some manual bookkeeping is necessary in this layer. Since garbage collection for Objective-C is not supported on some platforms and because a Objective-C garbage collector would not help us much, we quickly decided to rely on reference counting. Thus we need to make sure that an Eiffel wrapper object will initialize a valid pointer to an Objective-C object at creation time, maintain that pointer during the lifetime of the Eiffel object and properly release the Objective-C object when the Eiffel object gets released.

The basic features that we've come up with to implement the idea depicted here are listed in the following paragraphs. They are implemented in the root class `NS_OBJECT`:

`make_from_pointer (a_ptr: POINTER)`
: Create a new Eiffel object whose Objective-C counterpart was just initialized via an external call. This creation procedure is usually not exported to the outside of the framework but used by descendants.

`share_from_pointer (a_ptr: POINTER)`
: Create a new Eiffel object with the given Objective-C object and retain[1] the reference. This call must be used when the Objective-C object was not created by the Eiffel wrapper. Again this creation procedure is usually not exported, but in this case it is often used by functions of other classes when returning an object as the result of a query.

`dispose`
: When the Eiffel object is collected by the garbage collector we can release the pointer to the Objective-C object. Objective-C takes care of freeing the object if the reference count is zero.

3.8.1 Reverse lookup: Finding the dual Eiffel object

Since the pointer to the dual Objective-C object is stored as an attribute of the Eiffel object, it is easy to do a lookup in that direction. But how can we find out the dual Eiffel object, given an Objective-C pointer?

One simple idea is to keep a reference to the Eiffel object as an attribute of the Objective-C object. This would create problems with the memory menagement described above unless we find a smart way to releasing that reference. We cannot wait for the Objective-C object to be destroyed because that can only happen after its Eiffel dual has called `dispose`. If the reference to the Eiffel object is not freed that would prevent the garbage collector from collecting the Eiffel object.

[1] `retain` and `release` are Objective-C messages that increase/decrease the reference count of an object.

Our current solution is to use a hash-map with pointers to Objective-C objects as keys and weak references to Eiffel objects (IDENTIFIED) as values. Objects can be put into this hash-map when they are created

3.9 Callbacks and Inheritance

3.9.1 The Problem when Inheriting from an Objective-C Class

Since Cocoa is an object-oriented framework it is necessary – especially for the more advanced customizations of the standard widgets – to redefine certain methods through subclassing. We want to support a similar inheritance structure on the Eiffel side as on the Objective-C side. Moreover we would like to be able to customize the behaviour of a message by making a redefinition of a feature in Eiffel in just the same way as we would in Objective-C. One example is the `mouseDown:` message of `NSResponder` which is redefined by `NSButton` and many other classes. The most straight forward and transparent way to do this is if we can allow the user of our wrapper to inherit from NS_BUTTON and redefine the `mouse_down` feature.[2]
This is not trivial since overriding a method in the Eiffel system will not be visible to the Objective-C runtime. But before we can even think about solving this problem we need to make a short digress and take a look how simple callbacks work.

3.9.2 Calling an Eiffel Feature from Objective-C

We have seen how to call C and Objecitve-C code from Eiffel, but how can we call Eiffel code? CECIL (short for C-Eiffel Call-In Library) is a library used to do just that [6]. Listing 3.3 shows how such a feature call may look like from the C side and listing 3.4 shows how the respective Eiffel features could look like. For simplicity, the object and feature to be called are directly passed to the C function here.

Listing 3.3: A C function calling an Eiffel feature

```
1 void call_eiffel (EIF_OBJECT obj, void (*ep) (EIF_REFERENCE,
      EIF_INTEGER))
2 {
3   // Call feature 'ep' of Eiffel object 'obj' with argument 123
4   (ep) (eif_access(obj), 123);
5 }
```

[2] A similar approach will be used whenever Objective-C *delagetes* need to be made available to Eiffel. See chapter 4.

Listing 3.4: Calling the C-callback from Eiffel

```
1  execute_callback
2    do
3      call_eiffel (Current, $call_me)
4    end
5
6  call_me (a_int: INTEGER)
7    do
8      print_string("Callback with argument " + a_int.out)
9    end
10
11 call_eiffel (obj: ANY; ep: POINTER)
12   external
13     "C inline use %"c_file.h%""
14   end
```

3.9.3 Our Solution

Objective-C is more dynamic than many other programming languages in that it provides an API for the creation and manipulation of classes at runtime [4]. Moreover it provides simple ways for accessing runtime-specific features, such as getting a function pointer to the precursor of a message. This made it possible to come up with an interesting design, in which we centralize the callbacks to just one Eiffel singleton class (`OBJC_CALLBACK_MARSHAL`) and a single Objective-C file (`objc_callback_marshal.m`).

Create a new subclass of the desired Objective-C class and redefine its messages to point to a common C function. In that C function inspect the object's class and the selector being called, pass all the parameters to the callback feature of `OBJC_CALLBACK_MARSHAL`. (Special care has to be taken if Objective-C passes struct values on the stack.) The Eiffel feature can then access its Objective-C precursor, if needed, by determining the superclass and calling the respective method.

On a side note: We did not use this design from the beginning. Our initial approach was to manually create Objective-C classes for the features with the need of a callback. This would mean that a .h and .m file had to be written, defining a subclass of the original Objective-C class with facilities to forward certain messages to a given Eiffel feature and return the result. An instance of that class would then be initialized with a reference to the Eiffel object[3]. This approach turned out to be time-consuming and inflexible as it showed that the number of places where this pattern was needed was much greater than anticipated.

[3]Note that it is problematic to keep a regular Eiffel reference in the Objective-C object, because this prevents the Eiffel object from being garbage collected (See 3.8)

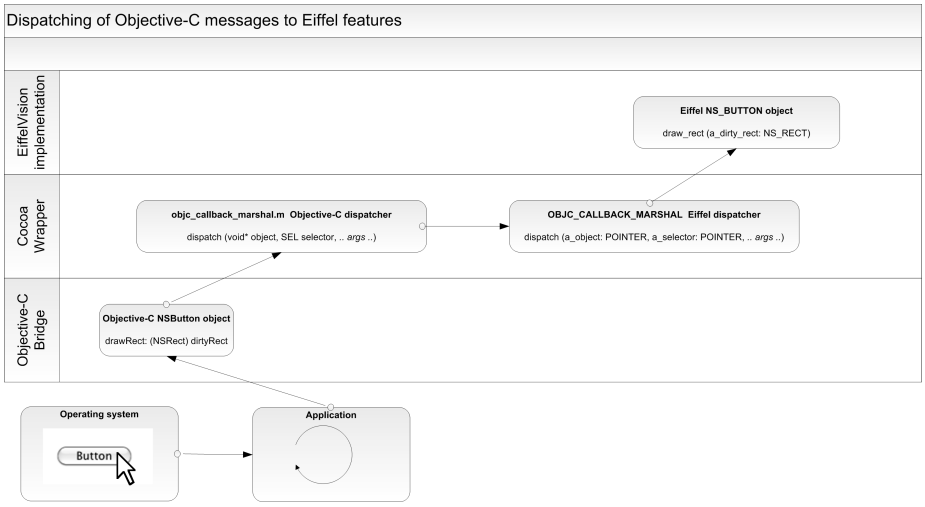

Figure 3.1: Illustration of the dispatching process

Chapter 4

Application Kit Wrapper

The Application Kit Wrapper library is a set of classes that wrap most of Apple's ApplicationKit – around 170 classes also known under the name Cocoa – to Eiffel classes. Using the classes and features of this layer is similar to the Objective-C pendants and anybody who has experience with Cocoa should feel right at home. The minor differences are due to the discrepancies between the two languages as explained in the previous chapter and because we replaced a few of Cocoa's design patterns with other solutions.

4.1 Wrapper Generator

Creating all the wrapper classes is a tedious and time consuming task. For this reason we wrote a simple Objective-C wrapper generator. A Python program which, given an Objective-C class name, parses the header file[1] of that class and Apple's documentation to generate initial Eiffel wrapper features. Manual tuning and refining is still needed because there are parts of the code that cannot be inferred automatically. Such tasks include the addition of assertions, deciding whether a type should be attached or detachable, generic type parameters and some other type mappings.

Let's have a look at how a feature of class `NSButton` gets mapped:

1. The wrapper will generate an Eiffel class called `NS_BUTTON_API` that contains all the calls to Objective-C features of the given Objective-C class. These features are usually only used by `NS_BUTTON` but the separation into another file helps avoiding clutter. (Listing 4.1)

2. `NS_BUTTON` – also generated by the wrapper generator – is the Eiffel abstraction of `NSButton`: The class that will be used by the EiffelVision implementation or other applications. (Listing 4.2)

[1] Although there exist Objective-C grammars for both YACC and ANTLR are available it was decided that those were too complex for our purposes. Our parser is thus based on simple regular expression matching. Further ideas are presented in section 7.2.

3. After the initial work by the wrapper some manual customizations may be necessary. In our example we have added another note to the description of the feature and a postcondition. (Listing 4.3)

Listing 4.1: A low-level feature as generated by our script

```
1  frozen set_title (a_button: POINTER; a_string: POINTER)
2      -- - (void)setTitle:(NSString *)aString
3    external
4      "Objective-C inline use <Cocoa/Cocoa.h>"
5    alias
6      "[(NSButton*)$a_button setTitle: $a_string];"
7    end
```

Listing 4.2: A wrapper feature as generated by our script

```
1  set_title (a_title: NS_STRING)
2      -- Sets the title displayed by the receiver when in
           its normal state and, if necessary, redraws the
           button's contents.
3    do
4      {NS_BUTTON_API}.set_title (item, a_title.item)
5    end
```

Listing 4.3: A wrapper feature after manual tuning

```
1  set_title (a_title: NS_STRING)
2      -- Sets the title displayed by the receiver when in
           its normal state and, if necessary, redraws the
           button's contents.
3      -- This title is always shown on buttons that don't
           use their alternate contents when highlighting or
           displaying their alternate state.
4    do
5      {NS_BUTTON_API}.set_title (item, a_title.item)
6    ensure
7      title_set: a_title.is_equal (title)
8    end
```

4.2 Inheritance for Wrapped Classes

To detect a click on a button we need to redefine `mouseDown:` of `NSButton` in Objective-C. The redefinition then needs to forward this message to the correct Eiffel feature (using the callback mechanism). The Eiffel class `NS_BUTTON` takes care of this by creating a subclass of `NSButton` using `OBJC_CLASS` in a `once` feature of `NS_BUTTON` as sketched in listing 4.4. In practise we often need to bind many more features and can use calls to the superclass to do this in an elegant way. We can then create an object of this class by calling `button_class.create_instance`.

Listing 4.4: Binding Objective-C messages to Eiffel features

```
1  class NS_BUTTON
2
3    button_class: OBJC_CLASS
4        -- Binds Objective-C messages to Eiffel features
5      once
6        create Result.make_with_name ("EiffelWrapperButton")
7        Result.set_superclass (create {OBJC_CLASS}.
             make_with_name ("NSButton"))
8        Result.add_method ("mouseDown:", agent mouse_down)
9        Result.add_method ("mouseUp:", agent mouse_up)
10       -- ...
11       Result.register
12     end
13
14   make
15       -- Create a new button
16     do
17       item := button_class.create_instance
18     end
```

4.3 The Target-Action Mechanism

The target-action mechanism is an implementation of the well-known command pattern. This pattern is used in Cocoa to send certain event notifications as an application specific message to the appropriate object. One example where it is used is `NSButton`, which sends a notification when the button was pressed[2].

Classes implementing this pattern implement two messages, `setTarget:` and `setAction:` which can be used to set the target, an object to which a message will be sent, and the action, the name of the message to be called on that object.

In Eiffel we can use the powerful notion of agents to support this pattern. We therefore only have a `set_target` feature which takes a `PROCEDURE` as argument. We can then call `button.set_target (agent my_application.do_stuff)` if we want feature `do_stuff` of object `my_application` to be called when `button` is pressed.

Internally we have to set up a simple Objective-C class which implements a single message and forwards it to an Eiffel object that can then invoke the agent. The creation of this Objective-C class and its binding to Eiffel are very similar to what we have seen in listing 4.4.

Similar invocations are used elsewhere in Cocoa and we apply the same mapping (e.g. for `NS_TIMER` and `NS_NOTIFICATION_CENTER`).

[2]Not to be confused with just clicking a button, meaning that a `mouseDown:` message is sent to the button object. Pressing a button is more abstract and can also happen when the user presses certain keys.

4.4 The Delegation Pattern

The delegation pattern – an object that acts on behalf of another object – is used frequently in Cocoa and often so where it is not common practice in Eiffel. One example is `NSWindow` which has a `setDelegate:` message. Instead of implementing similar features itself (or in addition to) the window is sending messages such as `windowDidResize:` or `windowWillClose:` to its delegate. The same behaviour is achieved in Eiffel using agents and even more flexibility is gained by using lists of agents. At this point we did not reach a decision whether to use agents or delegate classes in our Cocoa wrapper and so we use both approaches.

Chapter 5

Cocoa EiffelVision Implementation

Thanks to the abstractions described in the previous two chapters we do not need to deal with the low-level details of Objective-C or Cocoa anymore when implementing EiffelVision on Mac OS X. What remains to do in this final layer is to provide the necessary mapping between widgets and widget structures by implementing the EiffelVision features in terms of the wrapper classes from the lower layer. Sometimes this is as easy as forwarding a call but in other places we will see that a one-to-one mapping may not make any sense[1], be hard or impossible. In this chapter we give a brief overview of EiffelVision and its internals in general. We proceed by describing the problems encountered and the respective solutions.

5.1 A Quick Introduction to EiffelVision

EiffelVision uses the bridge pattern which makes it easy to change the implementation on different platforms. This means that every platform-dependent component consists of two classes, plus an implementation for each platform. The interface (e.g. `EV_BUTTON`) – the class used by applications using EiffelVision – only defines the available features and delegates any calls to the implementation object which is coupled to it. The implementation class (`EV_BUTTON_IMP`) is a subtype of the implementation interface (`EV_BUTTON_I`) and implements the platform specific features.

[1] For example because it would contradict fundamental principles of GUI design on Mac OS X.

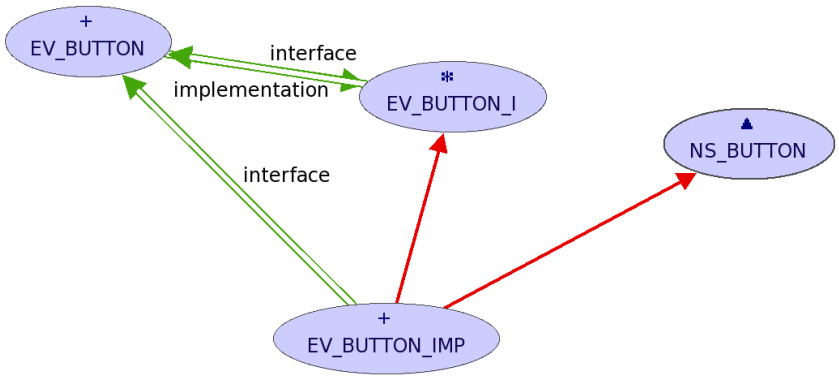

Figure 5.1: A simplified BON diagram illustrating our button implementation

5.2 Event Handling

Cocoa and EiffelVision both send events to their widgets when certain things happen. In Cocoa[2] every class whose objects can receive events inherits from `NS_RESPONDER` which has messages like `key_down (a_event: NS_EVENT)`. By redefining those messages in a subclass custom behavior can be implemented. In EiffelVision there is no global event class. The widgets define certain actions as a list of agents to be called in case of an event. An example of such an agent list is `key_press_actions` which is of type `LIST [PROCEDURE [ANY, TUPLE [key: EV_KEY]]]`. So what that remains to do in this case is that we have to redefine `key_down`, extract from the `NS_EVENT` to the information needed by the `key_press_actions` and call them.

Initially we experimented with an approach where we inspect the events in the application's main event loop in `EV_APPLICATION_IMP` to translate and forward them to the corresponding EiffelVision widget. Since we would only get a pointer to the Objective-C object from Cocoa we had to rely on a mechanism similar to what we described in section 3.8.1 (Reverse lookup: Finding the dual Eiffel object). This worked well in general but there was one problem with modal windows: A modal Cocoa window runs with its own event loop which is not accessible from the outside and so the events did not get forwarded in that case.

5.3 The Layout Engine

The layout engine for widgets which is used by EiffelVision is strongly inspired by GTK+. It promotes the automatic management of size and position of child widgets through their parents. The different widget containers have different constraints[3] to do this and we will look at one example in the following section.

[2]When talking about Cocoa in this chapter we usually mean our wrapped Cocoa classes. We do not use any Objective-C code in this layer and the reader should be familiar with the underlying differences by now.

[3]An example of a constraint is the minimum width needed to display a widget

Cocoa offers only a very simple interface for positioning widgets. The exact size of a widget and its coordinates relative to those of the parent widget (also called superview) need to be specified. Our EiffelVision implementation needs to calculate and update those positions when required while makeing sure the constraints hold.

We took many parts of the layout engine from the Windows implementation of EiffelVision.

5.3.1 Vertical box

The vertical box (`EV_VERTICAL_BOX`) is one of the most used containers. It can contain any number of widgets and displays those widgets in a single column. In its simplest configuration it devides the space along the y axis equally between all the widgets it contains. Equations 5.1 and 5.2 show the applicable constraints in this situation.

$$box.min\_height = |box.C| \cdot \max_{c \in box.C} \{c.min\_height\} \quad (5.1)$$

$$box.min\_width = \max_{c \in box.C} \{c.min\_width\} \quad (5.2)$$

where $box.C$ is the set of children of box.

5.4 Coordinate Systems

Cocoa is based on the Quartz rendering library which uses a standard cartesian coordinate system and places the origin in the bottom-left corner of the screen with the y-axis pointing upwards. Most other graphic toolkits (including EiffelVision) place the origin in the top-left corner with the y-axis pointing downwards. A coordinate transformation is needed and it is important that this transformation is handled transparently with respect to the rest of the implementation.

The following equations can be used for this transformation:

$$view.x_{cocoa} = view.x_{vision} \quad (5.3)$$

$$view.y_{cocoa} = view.superview.height - view.height - view.y_{vision} \quad (5.4)$$

We wrote a class called `EV_NS_VIEW` which exports the feature `cocoa_set_size` that takes care of those transformations and internally uses `NS_VIEW`'s `set_frame`. All our widget classes inherit from `EV_NS_VIEW` to access this functionality.

The other major components where coordinate transformations are needed are `EV_DRAWABLE_IMP` and its descendants. These classes handle the drawing of primitives such as lines, strings and images.

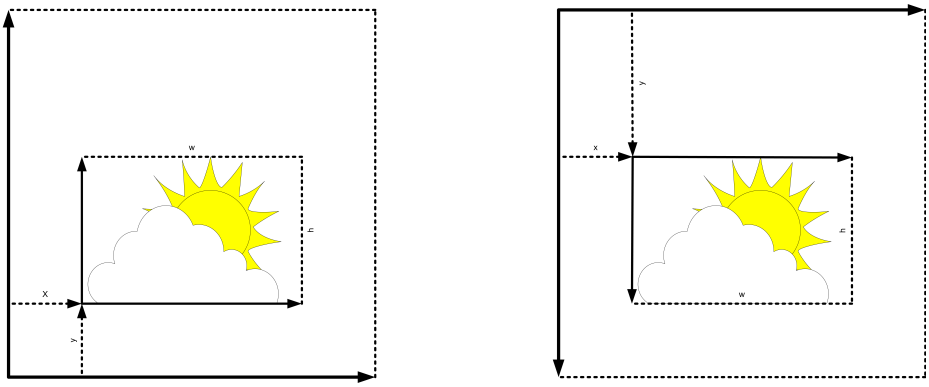

Figure 5.2: The coordinate system of Cocoa (left) and EiffelVision (right)

5.5 Other Platform Differences

There are some parts of EiffelVision for which we either have no Cocoa implementation or for which we only have a unsatisfactory solution. Some of these gaps are due to fundamental differences between the platforms. For these gaps it will be necessary to change the interface of EiffelVision. These differences include:

- On Mac OS X there is an application icon but no window icons.
- There is only one menu bar at the top of the screen (not one per window)
- Keyboard shortcuts include the command key which is only found on Mac keyboards.
- Many drawing modes used by EiffelVision (e.g. XOR) have no equivalent in Quartz
- Colors can't always be converted to RGB since they can be patterns
- Some dialogs (called panels in Cocoa) cannot be run modally and are of a one-per-application kind

Chapter 6

Conculsions

6.1 How well does it work?

In this section we give a high-level overview of what applications and features are working with our EiffelVision implementation and what does not work yet. A more detailed account by widget is given in appendix **??**.

Many simple example applications that are distributed with EiffelVision work well. The same is true for the applications we wrote. Here are a few examples:

widgets A demo application that displays (almost) all EiffelVision widgets and lets the user manipulate them.

edraw A simple drawing application written using EiffelStudio.

estudio The graphical starter for EiffelStudio

wizard We wrote a new applications wizard for creating an native Mac OS X application. This wizard can be selected when creating a new project with EiffelStudio on Mac OS X.

In EiffelStudio some basic use cases work. A project can be opened, compiled and run. A new project can be created. But there are issues with editing text, some of the widgets are not resized correctly.

Our libraries are fully void-safe[7] and have been tested using 32bit mode on Mac OS X 10.5 and 64bit mode on Mac OS X 10.6.

6.2 Summary

We have written an implementation of EiffelVision using the Cocoa framework which works well for many applications we tested. The architecture of our solution is based on three layers: An Eiffel/Objective-C bridge, a Cocoa wrapper

library, and the EiffelVision implementation. Each layer provides fundamental abstractions that make it easier to build the next one. We are convinced that this architecture makes understanding parts of the system easier and that it provides the necessary abstractions for further development.

We have shown how Objective-C can be integrated into Eiffel by creating Eiffel wrapper classes based on Objective-C classes. We have explained how identifiers, types, protocols, categories and other constructs can be mapped from Objective-C to Eiffel. Problems with memory management that arise when using wrapper classes have been identified and solved. We have shown and implemented solutions for calling Objective-C messages from Eiffel features and vice versa. And finally, we have shown how we can use inheritance in our Eiffel wrapper classes to redefine Objective-C messages.

We have written a tool to facilitate the creation of Eiffel wrapper classes based on Objective-C header files and documentation. Using this tool, we have created Eiffel wrapper classes for most of the functionality provided by Cocoa and the Foundation framework on Mac OS X.

While our EiffelVision implementation it is not yet fully functional we have implemented many important parts.

Chapter 7

Future Work

7.1 EiffelStudio

Although it was our goal to make EiffelStudio run using our EiffelVision implementation, much work remains to be done in this area. More work is needed to adapt the Smart Docking library, which depends on EiffelVision, but also uses some platform dependent code.

7.2 Objective-C Wrapper Generator

The Objective-C wrapper generator is very handy when wrapping new Objective-C classes, but it is missing some cases such as the proper handling of `typedef`s and it has problems when `struct`s are returned from a function on the stack. Also it is currently written in Python which makes it hard to interoperate with Eiffel directly, for example to get type information. Converting it to Eiffel and using the Eiffel Wrapper Generator should be investigated. It would certainly pay off to improve the wrapper generator, in case there is a need to wrap more Objective-C libraries.

7.3 Eiffel-Cocoa Wrapper

Although we designed our Cocoa wrapper as a separate library, so that it would be easy to create native OS X GUI applications, not much work has gone into exploring this scenario. We created a wizard application for EiffelStudio that creates a very basic Cocoa GUI application. However, at the moment we do not support the loading of user interface descriptions used for Cocoa development. Support for loading those files and integration of Apple's Interface Builder could make EiffelStudio a serious alternative to Apple's developer tools.

7.4 iPhone Library

Our work on the Eiffel/Objective-C language bridge inspired the development of an Eiffel library for iPhone development[1]. Some basic example applications with simple drawing and touch response can be run on the iPhone or on the iPhone simulator on Mac OS X. The iPhone library is, however, far from complete: Most functionality of the underlying UIKit framework is not wrapped at the moment and a better integration of the platform specific tools into EiffelStudio would be desirable as well.

[1] More information is on http://dev.eiffel.com/IPhone_Development

Appendix A

Development

A.1 Layout Inspector

The Layout Inspector (Figure A.1) is a graphical debugger for the GUI structures of EiffelVision2. It allows browsing through the tree of widgets and containers and displays relevant information for the selected widget, such as its position, size, and some layout attributes. On Mac OS X it also highlights the selected widget visually as can be see in in the screenshot below. Moreover it can be used to inspect a widget in the EiffelStudio debugger with the click of a button.

A.2 Tests

When we started with our project there were no test for EiffelVision. Several reasons made us think it was a good idea to add tests as our implementation progressed:

- There is no exact specification of the behavior of EiffelVision, in the form of a software requirements specification or any other form of extensive documentation. Contracts are a good way to make the specification of a feature clearer, but in many cases the exact semantics remained somewhat unclear. Unit tests serve as simple, minimal use cases, which can be compared across platforms and implementations.
- Tests help us detect regressions during development.
- Unit tests will lead to a more stable EiffelVision across platforms in the long term.

Testing a GUI is not easy, as it often involves the simulation of GUI events that are hard to describe exactly and even harder to keep in sync with changes in the framework. Furthermore, detecting the success of an operation on a high level can be equivalently hard. We focused our attention on simple unit tests

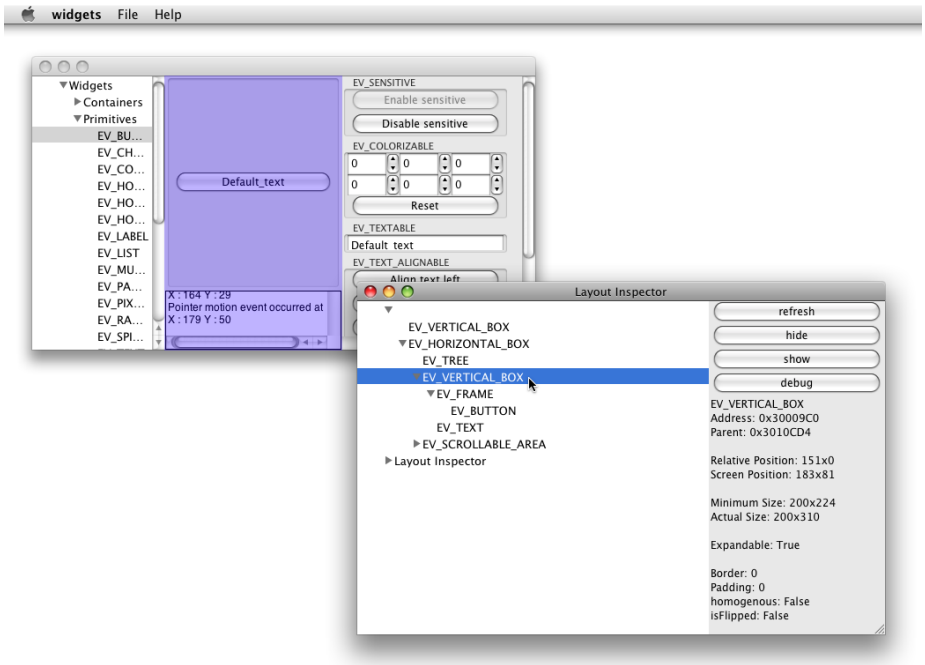

Figure A.1: The Layout Inspector on Mac OS X

that in most cases work directly with the API of a component and do not need any user input.

We used the Eiffel testing framework and suggested some changes that were promptly made part of its API. Our tests can be found in the `test` directory of the EiffelVision project (and also EiffelCocoa).

A.3 Developer's Guide

A.3.1 Index

All our code can be found in the subversion repository of EiffelSoftware under `https://svn.origo.ethz.ch/eiffelstudio/trunk/Src/experimental`. These are the subdirectories where the most important components can be found:

Component	Directory
Eiffel/Objective-C bridge	objc_base
Eiffel/Objective-C bridge tests	objc_base/tests
Cocoa wrapper	cocoa
Cocoa wrapper tests	cocoa/tests
Wrapper generator	cocoa/development
EiffelVision implementation	vision2/implementation/cocoa
EiffelVision tests	vision2/tests
Layout inspector	vision2/implementation/cocoa/testing

A.3.2 Setup

This section describes how to compile EiffelStudio or your own application using EiffelVision. An updated guide can be found online at http://dev.eiffel.com/EiffelVision_Cocoa.

1. Get the latest EiffelStudio for the Mac. If you have MacPorts installed you can do this by typing

```
port install eiffelstudio65
```

2. Check out the latest source code from the trunk:

```
svn co https://svn.origo.ethz.ch/eiffelstudio/trunk
```

3. Set your EIFFEL_SRC and ISE_LIBRARY like this:

```
export ISE_LIBRARY=/path_to_your_checkout/trunk/Src
export EIFFEL_SRC=/path_to_your_checkout/trunk/Src
```

4. Compile the C-parts of the library

```
cd $ISE_LIBRARY/experimental/library/objc_base/Clib
finish_freezing -library
cd $ISE_LIBRARY/experimental/library/cocoa/Clib
finish_freezing -library
```

5. Now open the .ecf file of your project and add the following line to your target:

```
<variable name="vision_implementation" value="cocoa"/>
```

   alternatively you can add another target which will allow you to easily compile both, a GTK+ and a Cocoa version of your application:

```
<target name="cocoa" extends="classic">
      <variable name="vision_implementation" value="
          cocoa"/>
</target>
```

If you want to compile Eiffel Studio you can use the provided target "bench_cocoa".

Make sure that you use `ec -experiment` (or `estudio -experiment`) to compile.

Appendix B: Status of our EiffelVision implementation

This chapter describes the status of the Cocoa based EiffelVision.

Contents

The list may miss some important functionality, which is part of the implementation but not directly exposed through features (e.g. the resizing behavior of the widgets).

General

- Pick&drop is not supported.

References

Apple Human Interface Guidelines (HIG), 2008-06-09

Status of our EiffelVision implementation

Windows

EV_WINDOW
EV_POPUP_WINDOW
EV_TITLED_WINDOW

NSWindow (style (titled, resizable), visibility, title, border, shadow, toolbar button, show in window list)

Supported

1. Title
   (title, set_title, remove_title)
2. Menubar
   [See Menus for the issue with Application Menus/Windows]
   (menu_bar, set_menu_bar, remove_menu_bar)
3. Window styles
   (disable_border, enable_border, is_border_enabled) (enable_user_resize, disable_user_resize, user_can_resize)
4. Arranging windows
   [Slightly different semantics than on other platforms]
   (is_minimized, is_maximized, raise, lower, minimize, maximize, restore)
5. Actions
   show_actions, hide_actions, move_actions

Not supported

1. Maximum size constraints (set_maximum_width, set_maximum_height, set_maximum_size, maximum_width, maximum_height)
2. Accelerators (accelerators)
3. Window icon (-> Dock icon) (icon_name, icon_pixmap, set_icon_name, remove_icon_name, set_icon_pixmap)
4. show_relative_to_window
5. close_request_actions

Dialogs

EV_DIALOG
EV_STANDARD_DIALOG
EV_FILE_DIALOG
EV_FILE_OPEN_DIALOG
EV_FILE_SAVE_DIALOG
EV_COLOR_DIALOG
EV_DIRECTORY_DIALOG
EV_FONT_DIALOG
EV_PRINT_DIALOG

NSPanel
NSOpenPanel (canChooseFiles, canChooseDirectories)
NSSavePanel
NSColorPanel

Status of our EiffelVision implementation

`NSFontPanel`
`NSPrintPanel`

Supported

1. Running modally
   (show_modal_to_window, is_modal, blocking_window)
2. Actions
   (ok_actions, cancel_actions)
3. Accessing files
   (file_name, file_title, file_path, valid_file_name, valid_file_title)
4. Start directory
   (start_directory, set_start_directory)
5. Multiple file selection
   (multiple_selection_enabled, enable_multiple_selection, disable_multiple_selection)

Not supported

1. Changing closable status (is_closable, disable_closable, enable_closable)
2. File filters (filter, filters, selected_filter_index)
3. Color dialog (all features)
   [On OS X a floating color panel per Application is usually used]
4. Font dialog
5. Print dialog

Containers

`EV_CONTAINER`

Supported

1. Basic operations (count, has, item, extend, replace)
2. new_item_actions
3. connect_radio_group, unconnect_radio_group

Boxes

`EV_BOX`
`EV_HORIZONTAL_BOX`
`EV_VERTICAL_BOX`

Supported

1. All features are available
   (is_homogenious, set_homogenious, border_width, set_border_width, padding, set_padding, is_item_expanded, set_child_expandable, pointer_offset)

Frame and Cell

`EV_FRAME`
`EV_CELL`

Status of our EiffelVision implementation

Supported

1. Basic operations, background color

Not supported

1. Title alignment (EV_TEXT_ALIGNABLE)
   [Not supported by the underlying Cocoa control]
2. Title font (EV_FONTABLE)

Note

a. (Frame) styles are somewhat different from other platforms

Fixed

`EV_FIXED`

Supported

1. Positioning and sizing items (set_item_position_and_size, set_item_x_position, set_item_y_position, set_item_position, set_item_width, set_item_height, set_item_size)

Split Area

`EV_SPLIT_AREA`
`EV_HORIZONTAL_SPLIT_AREA`
`EV_VERTICAL_SPLIT_AREA`

Supported

1. Setting items (set_first, set_second)
2. Splitter position (set_split_position, split_position)

Not supported

1. Size constraints (minimum_split_position, set_minimum_split_position, maximum_split_position, set_maximum_split_position)
2. Setting expanded-status (enable_item_expand, disable_item_expand, set_split_position, prune)

Note

a. Resizing is buggy (doesn't resize the children correctly)

Table

`EV_TABLE`

Supported

1. Mostly implemented but not well tested.
   (row_spacing, set_row_spacing, column_spacing, set_column_spacing, border_width, set_border_width, item_column_position, item_column_span, item_row_position, item_row_span)

Status of our EiffelVision implementation

Scrolling Area

`EV_VIEWPORT -> NSClipView`

`EV_SCROLLABLE_AREA -> NSScrollView` (setDocument)

Supported

1. All viewport features
   (x_offset, y_offset, set_x_offset, set_y_offset, set_offset, set_item_width, set_item_height, set_item_size)
2. Scrollbar hiding
   (is_horizontal_scroll_bar_visible, show_horizontal_scroll_bar, hide_horizontal_scroll_bar, is_vertical_scroll_bar_visible, show_vertical_scroll_bar, hide_vertical_scroll_bar)

Not supported

1. Accessing step size
   (horizontal_step, set_horizontal_step, vertical_step, set_vertical_step)

Tabs

`EV_NOTEBOOK`
`EV_NOTEBOOK_TAB`

`NSTabView` (add, select, orientation)
`NSTabViewItem`

Supported

1. Basic features
   (extend, remove, item_text, item_tab, select_item, set_item_text)

Not supported

1. Bugs in the layouting and displaying of child containers
2. Positioning tabs
   (tab_position, pointed_tab_index, set_tab_position)
3. Detecting selection
   (selection_actions, selected_item, selected_item_index)
4. Tab pixmaps
   [Not straight forward to implement on OS X]

Primitives

`EV_PRIMITIVE`

Buttons

`EV_BUTTON`
`EV_CHECK_BUTTON`
`EV_TOGGLE_BUTTON`
`EV_RADIO_BUTTON`

Status of our EiffelVision implementation

`EV_RADIO_PEER`

`NSButton` (type (normal, switch, toggle, radio), style, title, image, key)
`NSButtonCell`

1. Buttons are represented using one class in Cocoa. Get different styles by configuring a button through function calls.

Supported

1. Actions
   (select_actions)
2. Default button
   (is_default_push_button, enable_default_push_button, disable_default_push_button, enable_can_default)
3. Alignment
   (text_alignment, align_text_center, align_text_left, align_text_right)
4. Selection
   (is_selected, enable_select, disable_select, toggle)
5. Radio button grouping
   (is_selected, peers, selected_peer)
   [Other classes inheriting from EV_RADIO_PEER should work as well]

Not supported

1. Limited support for setting pixmaps
   [Mac OS X does not support pixmaps on the default button type]
2. Setting the font
   [Goes against Apple's HIG]

Combo Box

`EV_COMBO_BOX`
`EV_COMBO_BOX_ACTION_SEQUENCES`
EV_LIST_ITEM, EV_LIST_ITEM_LIST, EV_LIST_ITEM_LIST_ACTION_SEQUENCES

`NSComboBox` (insertItem)

Supported

1. (selected_item, select_item, deselect_item, clear_selection)
2.

Not Supported

1. (drop_down_actions, list_hidden_actions)
2. is_list_shown
3. Pixmaps for items

Note

a. Both implementations inherit from their respective Text Field.

Status of our EiffelVision implementation

Lists

EV_LIST
EV_LIST_ITEM
EV_LIST_ITEM_LIST
EV_CHECKABLE_LIST
EV_MULTI_COLUMN_LIST
EV_MULTI_COLUMN_LIST_ROW

NSTableView
NSTableHeaderView
NSTableDataSource
NSTableColumn

Supported

1. Basic operations
   (extend, remove, selected_item)
2. Selection and multiple selection
   (select_actions, multiple_selection_enabled, enable_multiple_selection, disable_multiple_selection)
3. Item pixmaps

Not supported

1. Checkable lists
2. ensure_item_visible, select_item, deselect_item, clear_selection, selected_items, deselect_actions
3. Item tooltips
4. Multi-column list specific
   (Column_count, row_height, title_shown, column_title, set_column_title, set_column_titles, column_width, set_column_width, resize_column_to_content, set_column_widths, show_title_row, hide_title_row, expand_column_count_to, update_column_width, set_text_on_position, set_row_pixmap, remove_row_pixmap, column_title_changed, column_width_changed, column_alignment_changed)
   [column_title_changed, column_width_changed: Why are these functions not implemented as action sequences?]
   column_title_click_actions, column_resized_actions
5. Text alignment
   (align_text_left, align_text_center, align_text_right, column_alignment, set_column_alignments)

Notes

a. Unclear how the checkable can be implemented

Grid

EV_GRID
EV_GRID_ROW
EV_GRID_ITEM
EV_GRID_CHECKABLE_LABEL_ITEM
EV_GRID_LOCKED_COLUMN
EV_GRID_ARRAYED_LIST

EV_GRID_DRAWABLE_ITEM
EV_GRID_LABEL_ITEM
EV_GRID_COLUMN
EV_GRID_LOCKED_ROW
EV_GRID_LOCKED
EV_GRID_DRAWER

EV_HEADER
EV_HEADER_ITEM

Not supported

The grid is almost fully implemented in a platform independent way and it is probably the single most complex widget. Because of those two reasons many things don't work yet.

Menu

EV_MENU
EV_MENU_BAR
EV_MENU_ITEM
EV_MENU_SEPARATOR
EV_RADIO_MENU_ITEM
EV_CHECK_MENU_ITEM
EV_MENU_ITEM_LIST

NSMenu
NSMenuItem (title, key, hidden, enables, image, submenu, separatorItem, key, state(on/off))
NSMenuView (display)

NSApplication (mainMenu, windowsMenu)

Supported

1. Basic operations
   (set_text, parent, select_actions)
2. Check and radio items
   (is_selected, enable_select, disable_select)

Not supported

1. Positioning
   (width, height, screen_x, screen_y, x_position, y_position, minimum_width, minimum_height)
2. Custom showing
   (show, show_at)
3. Icons
   (pixmap, set_pixmap)
4. (item_select_actions)

Notes

a. A menu bar is usually associated with an application instead of a window in Cocoa. This can be overcome by manually setting the right menu when a window receives focus.
b. In EV several classes for menu items exist: simple, a separator, a radio or a check item. One class in Cocoa handles all these and the difference is achieved through setting some properties.
   Menu items can be made checkable by toggling their 'state' in Cocoa. Radio buttons do not exist in the menus but can be created by manually setting the on/off pictures for the state and a bit of logic.

Tree

```
EV_TREE
EV_TREE_ITEM
EV_TREE_NODE
EV_TREE_NODE_LIST
EV_CHECKABLE_TREE
```

NSOutlineView
NSOutlineViewDataSource (collapse, expand)

Supported

1. Basic operations
   (extend, remove)
2. Selection
   (select_actions, selected, selected_item)

Not supported

1. Checkable tree
2. Actions
   (deselect_actions, expand_actions, collapse_actions)
3. Item tooltips
   (EV_TOOLTIPABLE)
4. Item icons
   (EV_ITEM)
5. (ensure_item_visible, is_selected, is_expanded, set_expand)

Notes

a. Data is loaded "on demand" via a delegate (the data source) in Carbon.

Tool Bar

```
EV_TOOL_BAR
EV_TOOL_BAR_SEPARATOR
EV_TOOL_BAR_BUTTON
EV_TOOL_BAR_TOGGLE_BUTTON
EV_TOOL_BAR_RADIO_BUTTON
EV_TOOL_BAR_DROP_DOWN_BUTTON
```

Status of our EiffelVision implementation

Supported

1. Basic Toolbar functionality
   (inserting buttons and separators)
2. Basic Toolbar Button functionality
   (has_text, text, set_text, has_pixmap, pixmap, set_pixmap, select_actions)

Not Supported

1. Vertical Toolbar
   (enable_vertical, disable_vertical, is_vertical)
2. Vertical button style
   (has_vertical_button_style, enable_vetrtical_button_style, disable_vertical_button_style)
3. (drop_down_actions)

Note

a. Toolbars on Mac OS X are somewhat different
b. Not sure what `EV_TOOL_BAR_DROP_DOWN_BUTTON` should do. It has no immediate features.

Progress Bar

`EV_PROGRESS_BAR`
`EV_HORIZONTAL_PROGRESS_BAR`
`EV_VERTICAL_PROGRESS_BAR`

`EV_GAUGE`

`NSProgressIndicator` (min, max, value, style)

Supported

1. **Full support**[a]
   (value, set_value, step, set_step, leap, set_leap, step_forward, step_backward, leap_forward, leap_backward, change_actions)
2. (is_segmented, enable_segmentation, disable_segmentation)

Notes

a. Only vertical Progress Bars are supported by Cocoa.
   It is unclear how a horizontal Progress Bar could be implemented.

Range

`EV_RANGE`
`EV_HORIZONTAL_RANGE`
`EV_VERTICAL_RANGE`

`NSSlider` (orientation, min, max, tickMarks)

Status of our EiffelVision implementation

Full support
(value, set_value, step, set_step, leap, set_leap, step_forward, step_backward, leap_forward, leap_backward, change_actions)

Scroll Bar
`EV_SCROLL_BAR`
`EV_HORIZONTAL_SCROLL_BAR`
`EV_VERTICAL_SCROLL_BAR`

`NSScroller`

Full support
(value, set_value, step, set_step, leap, set_leap, step_forward, step_backward, leap_forward, leap_backward, change_actions)

Note

a. The scroll-knob does not have the correct size

Spin Button
`EV_SPIN_BUTTON -> NSStepper (min, max, increment)`

Full support[a]
(value, set_value, step, set_step, leap, set_leap, step_forward, step_backward, leap_forward, leap_backward, change_actions)

Note

b. Needs more testing

Text Field, Password Field and Text Area
`EV_TEXT_FIELD`
`EV_PASSWORD_FIELD`
`EV_TEXT`
`EV_TEXT_COMPONENT`

`NSTextView, NSText, NSTextStorage, NSTextContainer`
`NSTextField` (editable, selectable, color, background color, style, border)

Supported

1. Basic functionality
   (set_text, append_text, prepend_text)
2. (is_editable, set_editable)

Not supported

1. Actions
   (change_actions, return_actions)
2. (capacity, set_capacity, insert_text, insert_text_at_position)
3. Text alignment
   (text_alignment, align_text_left, align_text_center, align_text_right)

4. Cursor position and text selection
   (caret_position, set_caret_position, has_selection, selection_start, selection_end, select_region, select_from_start_pos, deselect_all, delete_selection)
5. Clipboard interaction
   (cut_selection, copy_selection, paste)
6. Font settings
   (EV_FONTABLE)

Rich Text

```
EV_RICH_TEXT
EV_RICH_TEXT_CONSTANTS
EV_RICH_TEXT_BUFFERING_STRUCTURES

RTF_FORMAT
EV_PARAGRAPH_FORMAT
EV_CHARACTER_FORMAT
```

Not Supported

Separator

```
EV_SEPARATOR
EV_VERTICAL_SEPARATOR
EV_HORIZONTAL_SEPARATOR
```

Full Support

Label

```
EV_LABEL
```

Supported

1. Most functionality
   (text, set_text, font, set_font, set_background_color)

Not Supported

1. Text alignment
   (text_alignment, align_text_left, align_text_center, align_text_right)

Pixmap and Drawing Area

```
EV_DRAWING_AREA
EV_PIXMAP
EV_PIXEL_BUFFER

EV_BITMAP
```

Supported

1. Drawing
   (draw_point, draw_text, draw_rotated_text, draw_text_top_left, draw_ellipsed_text, draw_ellipsed_text_top_left, draw_segment, draw_straight_line, draw_pixmap, draw_sub_pixmap,

draw_sub_pixel_buffer, draw_arc, draw_rectangle, draw_ellipse, draw_polyline, draw_pie_slice, fill_rectangle, fill_ellipse, fill_polygon, fill_pie_slice)

2. Drawing styles
   (line_width, set_line_width, dashed_line_style, enable_dashed_line_style, disable_dashed_line_style)
3. Draw actions
   (expose_actions, redraw, redraw_rectangle, clear_and_redraw, clear_and_redraw_rectangle, flush, clear_rectangle)
4. Font setting
   (EV_FONTABLE)
5. Color setting
   (EV_COLORIZABLE)

Not Supported

1. Clipping
   (clip_area, set_clip_area, set_clip_region, remove_clipping)
2. Drawing styles
   (drawing_mode, set_drawing_mode, tile, set_tile, remove_tile)

Properties

```
EV_PICK_AND_DROPABLE
EV_PIXMAPABLE
EV_POSITIONABLE
EV_SELECTABLE
EV_SENSITIVE
EV_COLORIZABLE
EV_DESELECTABLE
EV_DRAWABLE
EV_FONTABLE
EV_TEXTABLE
EV_TOOLTIPABLE
EV_POSITIONED
EV_TEXT_ALIGNABLE
EV_TAB_CONTROLABLE
```

Other

Application

```
EV_ANY

EV_APPLICATION
```

Supported

1. Basic Actions
   (post_launch_actions, idle_actions, destroy_actions, uncaught_exception_actions)
2. Basic runloop
   (process_underlying_toolkit_event_queue)

3. Actions
   (pointer_motion_actions, pointer_button_press_actions, pointer_double_press_actions, key_press_actions, key_press_string_actions, key_release_actions, focus_in_actions, focus_out_actions)
4. (windows)

Not supported

1. Actions
   (pick_actions, drop_actions, cancel_actions, pnd_motion_actions, file_drop_actions, system_color_change_actions)
2. Other
   (pick_and_drop_source, ctrl_pressed, shift_pressed, alt_pressed, caps_lock_on, is_display_remote, process_graphical_events, sleep, lock, try_lock, unlock, tooltip_delay)

EV_ENVIRONMENT

Docking

EV_DOCKABLE_SOURCE
EV_DOCKABLE_TARGET

Not supported

Uncategorized

EV_DYNAMIC_LIST
EV_ITEM_LIST
EV_WIDGET_LIST

EV_TIMEOUT
Full support

EV_ACCELERATOR

EV_BEEP

EV_CLIPBOARD

EV_COLOR
Full support

EV_FONT
Full support

EV_HELP_CONTEXTABLE

EV_ITEM

EV_SIMPLE_ITEM

Bibliography

[1] The D/Objective-C Bridge. http://michelf.com/weblog/2007/d-objc-bridge/, 2007.

[2] Eiffel software mailing list. mailto:eiffel_software@yahoogroups.com, 2009.

[3] Monoobjc: A .NET/Objective-C Bridge. http://monoobjc.net, 2009.

[4] Inc. Apple. Objective-C 2.0 Runtime Reference. http://developer.apple.com/mac/library/documentation/Cocoa/Reference/ObjCRuntimeRef/Reference/reference.html, 2007.

[5] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition.* Prentice-Hall, 1997.

[6] Bertrand Meyer. Eiffel: The language, third edition. ongoing work.

[7] Bertrand Meyer, Alexander Kogtenkov, and Emmanuel Stapf. Avoid a Void: The eradication of null dereferencing. 2009.