Chair of Software Engineering

A Mac OS X EiffelVision port based on a generated Cocoa wrapper

Bachelor Thesis

Emanuele Rudel
ETH Zurich
erudel@student.ethz.ch

October, 2011 - February, 2012

Supervised by:
Benjamin Morandi
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

inf | Informatik
Computer Science

Abstract

EiffelVision 2 is a framework for developing graphical user interfaces with the Eiffel programming language. Despite being a cross-platform library, a native implementation for the Mac platform does not exist yet.

The goal of this project is to develop the basic functionalities of EiffelVision 2 by identifying and applying recurrent patterns to map the widgets from EiffelVision 2 to Cocoa. The port relies on an existing Cocoa wrapper framework.

Acknowledgments

I would like to thank my supervisor Benjamin Morandi for the precious and continuous feedback received throughout the research. I would also like to extend my deepest gratitude to Prof. Bertrand Meyer for giving me the opportunity to work on this exciting topic and to Emmanuel Stapf from Eiffel Software for the technical help.

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

Contents

Chapter 1

Introduction

EiffelVision 2[1] is the main graphical user interface library available for the Eiffel programming language. It runs on Windows and all major versions of Unix, including Mac OS X. The Unix version of EiffelVision is implemented using GTK+[2], a multi-platform toolkit written in C. This port runs natively on the X window system. Since X is not the default window system of Mac OS X, EiffelVision runs on a simulator of the X environment called X11. The disadvantage of this solution is that the user interface is not consistent with the rest of the operating system.

Cocoa is a set of object-oriented frameworks for the Mac OS X and is the main application environment for this operating system. The purpose of this project is to provide an implementation of EiffelVision based on Cocoa. The port significantly relies on the Cocoa AppKit framework, which contains all the graphical elements to build a native Mac OS X application. This enables applications to run natively on Mac OS X without the aid of X11.

This report presents the strategies adopted to map the widgets implemented in EiffelVision to the ones available in the AppKit library. It also discusses possible extensions, workarounds and the issues to achieve a complete widgets mapping.

[1] Referred to as EiffelVision throughout the document
[2] GTK+ is an acronym for GIMP Toolkit. http://www.gtk.org/

Chapter 2

Related work

In 2009 Daniel Furrer made an attempt to port the EiffelVision library to Mac OS X [2]. He developed the Cocoa wrapper libraries and, relying on them, a part of the EiffelVision framework, which was used as a starting point for this work. Furrer's EiffelVision implementation and the one of this work, from now on, are referred to as the legacy port and the current port. Although Cocoa frameworks were automatically wrapped in Eiffel, manual adjustments were performed after the code generation. The adjustments involved adding generics (not present in Objective-C) and adding contracts that could not be inferred automatically because of a lack of specifications. These enhancements do yield more robust code; however, they turn out to be time-consuming operations that in the end do not heavily affect the EiffelVision implementation. Moreover, the approach is not flexible as it needs to be performed to every new class added to the system and each time the APIs are updated.

Another project, developed in 2010 by Matteo Cortonesi, focuses on the automatic generation of Eiffel wrappers for Objective-C libraries [5]. The main features of the generator are:

- There are no manual adjustments, i.e. the wrapped Objective-C libraries do not have contracts and generics.
- Delegates and protocols are supported.
- A robust memory management is present. At the time of writing, no memory leaks have been observed.

The first feature makes this tool the best choice to build a Cocoa wrapper in Eiffel. Apple releases a new operating systems about once every two years. Each release introduces new APIs, deprecates old ones, and modifies existing ones. For that reason it is essential that the Cocoa wrapper can be quickly updated.

Chapter 3

Background

3.1 Port Architecture

Porting EiffelVision to Mac OS X is a procedure that requires a well-structured foundation. The port is divided into three layers which make the interaction between EiffelVision and Cocoa possible:

1. The first and lowest layer connects the Eiffel and Objective-C programming languages. It allows to reference an object from the Cocoa to the Eiffel environment and vice versa. Moreover, this layer takes care of memory management (there is no garbage collector in Objective-C) and callbacks from Objective-C to Eiffel.

2. The second layer wraps the Cocoa libraries in Eiffel. This layer is generated using an automatic library converter [5]. Given a set of Objective-C frameworks as input, the converter outputs a set of Eiffel classes that wrap the Objective-C classes of the frameworks.

3. The third layer implements the Mac OS X version of EiffelVision. The previous two layers, also called abstraction layers, hide the implementation details of interfacing Cocoa with Eiffel. Thanks to this layer architecture, the port can use the Cocoa libraries as if they were natively written in Eiffel. The work of this thesis focuses on the third layer and relies on the already available abstraction layers.

EiffelVision organizes its classes in a set of clusters according to their roles and functionalities [3]. The next chapter describes for each cluster the similarities and differences between Cocoa and EiffelVision, along with the strategies adopted to bridge the gap between the two worlds.

The port should satisfy the two following requirements:

1. A widget must reflect the state specified by the application. This means that a button, for example, is displayed with the title, color, position

and size that EiffelVision assigned to it. EiffelVision exploits the bridge pattern to support multiple platforms, hence the set of possible features applicable to a widget is defined by an interface class. For each feature of the interface class, an equivalent feature — or the closest possible to an equivalent — in the Cocoa frameworks must be called.

2. A widget must notify the application of events triggered by the user. Events are triggered in the Cocoa environment using three distinct methods and must be forwarded to the EiffelVision widget, which deals with events using action sequences. The corresponding techniques are described in Section 4.1.

3.2 AutoLayout

AutoLayout is an automatic layout system that controls the appearance of objects in the user interface. AutoLayout has been introduced in Mac OS X 10.7 and it is not complete yet. The current Mac OS X is in fact in a transitional state from the old model (not discussed in this thesis) to AutoLayout. Even though it is not always possible to obtain the same exact behavior that EiffelVision offers to other platforms, future releases of Mac OS X are likely to expand the AutoLayout APIs and fill these gaps.

The idea behind automatic layout is to encode simple human statements in layout constraints. A layout constraint expresses a linear relationship between two widgets — called *views* in the Cocoa environment.

Cocoa offers a convenient way to express constraints using ASCII art and allows developers to define complex layouts with just a few lines of code. ASCII art has the great advantage over plain code of being extremely easy to read for humans. The statement "this text field should be horizontally aligned to the right of this button and have a fixed distance of 7 pixels from it" can be written as:

```
H:[button]-7-[text_field]
```

AutoLayout also allows to bind the size of two or more widgets. The next layout constraint expresses the same statement above, and additionally requires that the text field and the button have the same width:

```
H:[button]-7-[text_field(button)]
```

A container is represented in ASCII using the pipe character. In AutoLayout the position of a widget is defined by the padding between the widget and its container in the following way:

```
H:|-20-[button]-100|
V:|-14-[button]-100|
```

Once constraints have been assigned to a container, the Cocoa layout engine takes care of displaying and resizing the widgets while respecting all the given

constraints. Since adding and removing constraints are expensive operations, Cocoa allows to change a subset of the properties of a layout constraint after the constraint's creation. Therefore, whenever possible, the port modifies the layout constraint's properties instead of replacing the whole constraint.

Constraints are a set of linear equations and for that reason it is important that they are neither ambiguous (underspecified) nor unsatisfiable (overdetermined). In EiffelVision it is not always possible to define an unambiguous set of constraints (e.g. in Section 4.3.1).

In AutoLayout the minimum size of a widget is computed taking into account all the constraint it holds and the subviews in the widget, if any. On a widget that inherits from `NS_VIEW`, the `fitting_size` method determines the minimum size of the widget.

3.3 Cocoa Wrapper Callbacks

A recurring situation in the port is the need to subclass and redefine methods of a wrapped Cocoa class. The goal is to obtain, at runtime, an instance of the Cocoa class with the new functionalities implemented in the port, i.e., implemented using the Eiffel language. Without any adjustments, it is impossible for the Objective-C object to call the Eiffel feature because the object does not know the address of the redefined feature. A simple solution to this problem is proposed in [5] and it requires three steps:

- In the inheritance clause of the wrapped Cocoa class, say `NS_A_CLASS`, redefine the `make` feature and the feature that needs to be redefined, e.g., `do_something`:

```
class MY_CLASS
inherit
  NS_A_CLASS
    redefine
      make,
      do_something
    end
  [...]
end
```

- Implement the `make` procedure as follows:

```
make
  do
    add_objc_callback ("doSomething", agent
        do_something)
    Precursor
  end
```

- Implement `do_something` in the subclass of `NS_A_CLASS`.

The `add_objc_callback` feature intercepts all the Objective-C calls to `doSomething` and redirects them to the redefined `do_something` feature. This process is called hijacking and is described thoroughly in Cortonesi's work [5]. Using this approach, it is also possible to hijack calls to Objective-C methods that do not actually exist. On the other hand, the agent to which the call is hijacked must be defined.

Chapter 4

Cocoa EiffelVision Implementation

The basis for this work is a combination of the legacy EiffelVision port [2] and the wrapper for the Mac OS X 10.7 libraries generated with the automatic library converter [5]. To make the legacy port compatible with the newly generated Cocoa wrapper, the following basic changes had to be performed:

- Replacement of deprecated (obsolete) methods since the legacy port is based on the Mac OS X 10.6 libraries.
- Renaming of methods. Cortonesi's report briefly explains the naming conventions adopted and why they are important.
- Renaming of inheritance clauses for the classes that use categories or protocols.
- Removal of generics in arrays and other Cocoa containers.

The remainder of this chapter describes the continuation of the development based on the new Cocoa wrapper.

4.1 Events Cluster

The EiffelVision library supports a wide range of user initiated events such as single and double mouse clicks, pick and drop, dragging operations and keystrokes. Event handling is elegantly implemented using agents: each event keeps a list of agents that take a number of parameters such as the location on the screen where the event happened. Whenever an event is triggered, EiffelVision traverses the list and calls the agents.

Cocoa has at least three different ways of handling events. In the first method, a widget inherits from a class that responds to events (`NS_RESPONDER`)

and implements its custom behavior by redefining those features that capture events of interest. This method is mainly adopted to intercept mouse and keyboard events.

The second one, available only to a subset of classes, is similar to the EiffelVision action sequences, with the difference that a widget can subscribe at most one action per event. The mapping from EiffelVision to Cocoa of this method is described in Section 4.3.3.

In the third method, an object shifts the responsibility to a delegate object which is capable of responding to a set of events.

The second method and third method are usually adopted when responding to one and multiple events, respectively. Consequently, depending on the widget being implemented, the method that fits best is chosen.

4.2 Kernel Cluster

The kernel cluster lays the foundations of the EiffelVision framework. The main class of the kernel cluster is the application class, which provides the starting point of all EiffelVision applications. Although the legacy port already provided an implementation, there was a major issue with handling modal windows. In order to understand and fix the problem, it is important to recall what the application class does. The main job of the application class is to run a possibly endless loop that wait for events. As soon as an event is detected, the application forwards the event to the appropriate widget, i.e., the widget the user clicked on. A Cocoa application object (an instance of `NS_APPLICATION`) receives events from the window server and processes them one at a time. If the main loop is busy handling an event, other events arriving from the window server are stored in an event queue and processed later.

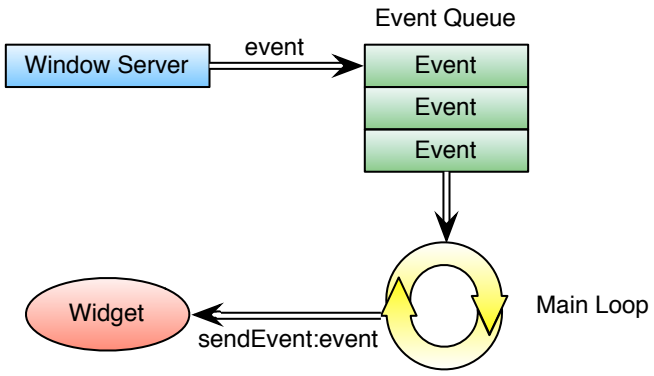

Figure 4.1: Main loop of the `NS_APPLICATION` class

The implementation of the EiffelVision application main loop in the legacy port is shown in Listing 4.1.

Listing 4.1: *Eiffel*: `EV_APPLICATION_IMP` main loop: legacy implementation

```
1  process_underlying_toolkit_event_queue
2        -- Process Cocoa events
3      local
4        event: detachable NS_EVENT
5        l_loop_pool: NS_AUTORELEASE_POOL
6      do
7        create l_loop_pool.make
8        from
9          event := next_event ({NS_APPLICATION_API}.
               ns_any_event_mask, Void, default_run_loop_mode,
               true)
10       until
11         event = Void
12       loop
13         send_event (event)
14         update_windows
15         event := next_event ({NS_APPLICATION_API}.
               ns_any_event_mask, Void, default_run_loop_mode,
               true)
16       end
17       l_loop_pool.release
18     end
```

The legacy port had the issue that the application could not capture events triggered outside modal windows because in Cocoa modal windows have their own main loops. In other words, the application was stuck in the main loop of the last shown modal window. As the name suggests, the feature's task in Listing 4.1 is to fetch events from the event queue and forward them to the EiffelVision widget. The code, however, just receives events from the Cocoa event queue and forwards them (line 13) to other Cocoa objects, which will later notify the corresponding EiffelVision widgets.

The mistake in Listing 4.1 of the old port is trying to mimic the Cocoa behavior of the `NS_APPLICATION` `run` method in the EiffelVision environment instead of the other way around. The current port is hence just calling `run` on the Cocoa application object and thus solves the problem of handling modal windows.

Listing 4.2: *Eiffel*: Application main loop: new implementation

```
1  process_underlying_toolkit_event_queue
2      -- Process Cocoa events
3    do
4      application.run
5    end
6  [...]
7    application: NS_APPLICATION
```

4.3 Widgets Cluster

The widgets cluster contains classes used to create objects the users can interact with. A widget is either a container — an object that can contain other widgets — or a primitive; a primitive cannot container other widgets. Primitives are used for the communication between the user and the application.

4.3.1 Containers Cluster

Containers play a major role in this thesis. The legacy port relies on a two-step mechanism triggered upon insertion or deletion of a widget into a container:

1. The container recursively computes the size of its widgets.
2. Should a widget have a different size compared to the previous one, it notifies the parent container which in turn adjusts its size.

The drawbacks of this mechanism are:

- Containers look different and may have different ways of computing the size necessary to display their children. Even primitives may have their own way of computing the widget's size. Consequently, the code is scattered all over the EiffelVision port.
- The method is not efficient. Consider for example the case in which the user resizes a window. Each time the window's boundary changes, the widgets in the whole hierarchy have to recompute the size needed to be properly displayed. Of course the proposed improvement is going to have to perform the same computations when resizing the window; the difference is that the latter defers the task to the Cocoa frameworks, which highly optimize computations with respect to the legacy port.
- Changes are applied lazily, i.e., the changes are postponed until the widgets are about to be displayed to the user. Although in the end the outcome is correct, it adds unnecessary complexity to the overall process.
- In some cases, the legacy port needs the widget to be in a container to compute the height properly (see Section 4.3.3). This requirement is not always satisfied because the height's computation might be triggered before the widget is inserted into a container.

The following subsections describe implementations of different types of containers to overcome those limitations. To explain the solution adopted to handle the size of containers it is necessary to introduce AutoLayout (Cocoa automatic layout constraints) [1]. A brief overview of AutoLayout is given in Section 3.2.

Vertical and Horizontal Boxes

Vertical and horizontal boxes are containers that stack widgets from top to bottom and from left to right, respectively. Widgets expand by default to fill the available space. Suppose that four widgets w_i with zero padding should be inserted into a box. The corresponding layout constraints (actually 8 combined in one ASCII art) are shown in Listing 4.3.

Listing 4.3: Layout constraints for horizontal and vertical boxes

```
V:|[w_1]-0-[w_2(w_1)]-0-[w_3(w_1)]-0-[w_4(w_1)]|

H:|[w_1]-0-[w_2(w_1)]-0-[w_3(w_1)]-0-[w_4(w_1)]|
```

Applying layout constraints is a much more involved process than just defining the above ASCII layout for a number of different reasons:

- Widgets can only be inserted one at a time and can be assigned to an arbitrary position.
- Widgets may be removed. If the removed widget is referenced in constraints (e.g. `w_1` is removed in Listing 4.3) it is necessary to reflect the changes.
- Some widgets may not be resizable. This causes a problem with vertical boxes because widgets such as buttons and text fields have a fixed height. In this case, the Unix and Windows EiffelVision versions leave a padding of equal size between widgets of fixed height. This behavior cannot be achieved in Cocoa because the layout engine does not allow to resize paddings equally (only widgets can be bound to have the same size, as explained in Section 3.2). In the current port if a widget cannot change its height, then the height of its container is fixed too. It suffices to have one resizable widget into a container to resize the container itself, as shown in Figure 4.2.

Cells, Frames, Notebooks and Windows

Cells, frames, notebooks and windows are simple containers because they can display at most one item at a time. These widgets have the same layout constraints even if their graphical representation differ. Notebooks are widgets that can hold multiple widgets, each assigned to one tab; only one item at a time is displayed on the screen. The constraints to satisfy for an item inserted in one of the first three type of containers are to stick to the container's borders:

```
V:|-border-[item]-border-| -- the border is 0 if
H:|-border-[item]-border-| -- not set or not changeable
```

Like cells, frames and notebooks, windows can hold at most one item. Yet they differ from them because in the actual implementation windows also allocate space for an upper and a lower bar which can neither be removed nor replaced. The layout constraints to satisfy become:

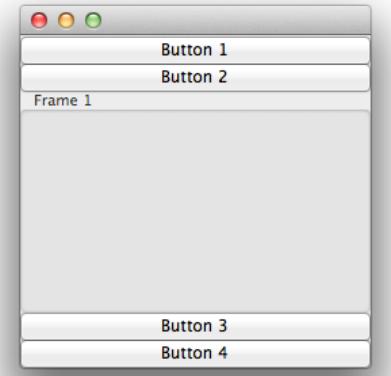

Figure 4.2: An example of a vertical box layout. The window's height can be resized since it has a widget (a frame) whose height is not fixed.

```
V:|-border-[upper_bar][item][lower_bar]-border-|
H:|-border-[item]-border-|
```

Viewports and Scrollable Areas

Viewports and scrollable areas are containers that may hold one widget whose size is bigger than the container itself. While viewports simply clip the invisible part of the item they contain, scrollable areas allow the user to access the whole widget by providing horizontal and vertical scroll bars.

The scrollable area is a widget for which each feature has a direct Cocoa equivalent. The scrollable area widget corresponds to the `NSScrollView` class.

There is no need to apply layout constraints to the item contained in one of the two widgets because viewports and scrollable areas automatically clip the item if it is bigger than the container.

Split Areas

Split areas are containers that give users the possibility to adjust the size of two widgets stacked either horizontally or vertically. Such behavior has the benefit of letting users decide on which widget to focus. They are often used to compare two similar resources, e.g., differencing and merging tools.

Cocoa offers a `NSSplitView` class which is a generalization of the EiffelVision split area, as it can contain more than two widgets. Each item of the split area must stick to the borders of its container (where the splitter is also considered to be a border) and thus the constraints to apply are:

```
V:|-0-[item]-0-|
H:|-0-[item]-0-|
```

Fixed Containers

Fixed containers are the natural equivalent of the Cocoa `NSView` class. In both Cocoa and EiffelVision, widgets can be positioned anywhere inside the container and can also overlap (in which case clipping occurs). The visibility order of widgets is determined by the sequence in which they are added to the container, i.e., the last item will be the top-most widget. A widget does not expand to fill the available space because it could potentially cover widgets behind it.

The constraints to define the position and size of an element are listed below:

```
V:|-y_offset-[item(height)]
H:|-x_offset-[item(width)]
```

4.3.2 Primitives Cluster

Primitives are widgets users typically interact with; they do not contain other widgets, although certain primitives allow to reference other items (e.g. a list may contain list items). Some primitives share common patterns to achieve a mapping of the functionalities from EiffelVision to Cocoa. These patterns are described in the next section because even though they are mostly used in the primitive cluster, other type of widgets can take advantage of them too.

The complete list of the primitives' implementation status can be found in Appendix A.

4.3.3 Mechanisms

This section describes the different implementation solutions adopted throughout the EiffelVision port to map the widgets to Cocoa. Each solution is accompanied with a practical example of how it has been employed in the port. At the end of this section, a summary shows which widgets adopted which strategies.

Target-Action Mechanism

The target-action mechanism was briefly mentioned in Section 4.1 as a common Cocoa pattern to respond to user actions such as clicking on a button. The target-action mechanism is implemented in `NS_CONTROL` and its descendants. In the port, this mechanism is used to respond to a single event per class. A Cocoa widget that wants to execute an action upon user invocation must specify both a target and an action (the feature) to be called on the target. In the port, the target-action mechanism is typically adopted subclassing the wrapped Cocoa class as shown below in the simplified button implementation.

Listing 4.4: *Eiffel*: Target-action mechanism in button implementation

```
1  class EV_BUTTON_IMP
2
3  inherit
4    [...]
5    NS_BUTTON
6      redefine
7        make
8      end
9
10 create
11   make
12
13 feature {NONE} -- Initialization
14
15   make
16     do
17       add_objc_callback ("did_press_button:", agent
             did_press_button)
18       Precursor {NS_BUTTON}
19       set_target_ (Current)
20       set_action_ (create {OBJC_SELECTOR}.make_with_name("
             did_press_button:"))
21       [...]
22     end
23
24   did_press_button (sender: NS_BUTTON)
25     do
26       select_actions.call([])
27     end
28 end
```

In the creation procedure of Listing 4.4, an instance of `EV_BUTTON_IMP` takes the responsibility of responding to user actions. While Cocoa requires to specify both a target and an action to apply on the target (the latter known as selector), Eiffel just uses the notion of agents. The target is thus set to `Current` through `set_target_` and the action to `did_press_button` through `set_action_`. Cocoa will call `did_press_button` whenever the user clicks on the button and this feature then triggers the `select_actions` action sequence.

Data Sources and Delegates

In Cocoa, data sources and delegates are objects that help another object — typically a widget — controlling the data model and responding to events, respectively. Data sources and delegates are required to implement a set of methods that Cocoa frameworks are automatically going to call whenever it is needed. This is also known as the Hollywood principle "don't call us, we will call you".

Data sources act as controllers between the application models and the graphical representation of these models. Consider, for example, the process of populating an `EV_LIST` in EiffelVision. The data and the Cocoa widget are

already available in the form of a sequence of list items and a `NS_TABLE_VIEW`. The data source only asks to specify the number of rows to be displayed and the object (the list item) associated to each row. It then automatically takes care of displaying and redrawing the list's content. The widget still needs to be manually updated by calling `reload_data` as changes to the list's model are applied; Cocoa then calls the data source's methods and applies the changes accordingly.

Although custom drawing for a list item is achievable, it is not needed for the EiffelVision port.

Listing 4.5 illustrates how to setup the delegate and the data source of a `NS_TABLE_VIEW`. The setup only consists of specifying which object, at runtime, implements the data source and the delegate. In the same way as the target-action mechanism, delegates and data sources have to hijack the Objective-C methods they are required to implement.

Listing 4.5: *Eiffel*: List implementation: delegate and data source setup

```
1  class EV_LIST_IMP
2  inherit
3    NS_TABLE_VIEW_DELEGATE_PROTOCOL
4    NS_TABLE_VIEW_DATA_SOURCE_PROTOCOL
5    [...]
6  feature {NONE} -- Initialization
7    make
8      do
9        add_objc_callback ("numberOfRowsInTableView:", agent
             number_of_rows_in_table_view_)
10       add_objc_callback ("tableView:
             objectValueForTableColumn:row:", agent
             table_view__object_value_for_table_column__row_)
11       add_objc_callback ("tableViewSelectionDidChange:",
             agent table_view_selection_did_change_)
12       [...]
13       table_view.set_delegate_ (Current)
14       table_view.set_data_source_ (Current)
15     end
16
17   table_view: NS_TABLE_VIEW
18 end
```

As mentioned before, the class `EV_LIST_IMP` must specify who is in charge to respond to the data source and delegate protocol calls. In this case it is `Current` because it provides the data model to be displayed on the screen.

Once the setup is complete, the data source and the delegate (which happen to be the in the same class in this case) implement the following features:

```
feature -- Data Source

  number_of_rows_in_table_view_ (a_table_view:NS_TABLE_VIEW)
      : INTEGER_64
    do
      Result := count
```

```
    end

  table_view__object_value_for_table_column__row_ (
      a_table_view: detachable NS_TABLE_VIEW; a_table_column:
       detachable NS_TABLE_COLUMN; a_row: INTEGER_64):
      detachable NS_OBJECT
    do
      Result := create {NS_STRING}.make_with_eiffel_string (
          i_th (a_row.as_integer_32 + 1).text)
    end

feature -- Delegate

  table_view_selection_did_change_ (a_notification:
      NS_NOTIFICATION)
      -- The selection of the table view changed
    do
      select_actions.call ([])
      if attached selected_item as l_item then
        l_item.select_actions.call([])
      end
    end
```

The first feature (line 3) tells the table view `a_table_view` to instantiate a number of rows equal to the number of items in the list. The second feature (line 8) returns the object associated to the row at index `a_row`. The value must conform to `NS_OBJECT` and thus instead of the `EV_LIST_ITEM` object only its text content as a `NS_STRING` is returned. The table view is smart enough to create a graphical element containing the text specified for each row. In fact, there is no drawing code involved in the list implementation.

The last feature, `table_view_selection_did_change`, is triggered when the selected row in the table view changed. It then calls the EiffelVision action sequences for both the list and the list item.

Flipped View

Cocoa is based on a classic Cartesian coordinates system with the origin $(0, 0)$ placed on the bottom-left of the plane, while in EiffelVision the origin lies on the upper-left corner. In the legacy port the following simple coordinate transformation was applied to convert EiffelVision coordinates to Cocoa ones:

$$view.x_{cocoa} = view.x_{vision} \tag{4.1}$$

$$view.y_{cocoa} = view.superview.height - view.height - view.y_{vision} \tag{4.2}$$

An issue that often arises using Equation 4.2 is that the view may not have been added to a container yet and thus its superview attribute is void. In that case, it is not possible to determine the correct value for the y-coordinate of the Cocoa widget. Furthermore, the above equations do not hold for widgets that

have graphical elements on the borders like frames and notebooks because they require even more sophisticated coordinate transformations.

A simpler and more robust approach is to flip the Cocoa coordinates — and therefore avoid to compute the coordinates transformation — by subclassing the `NS_VIEW` class and redefining the origin's location. Listing 4.6 shows `EV_FLIPPED_VIEW`, a class identical to `NS_VIEW` except for the origin's location.

Listing 4.6: *Eiffel*: Implementation of `EV_FLIPPED_VIEW`

```
1  class EV_FLIPPED_VIEW
2  inherit
3    NS_VIEW
4      redefine
5        make,
6        is_flipped
7      end
8
9  feature {NONE} -- Initialization
10   make
11     add_objc_callback ("isFlipped", agent is_flipped)
12     Precursor
13   end
14 feature -- Access
15   is_flipped: BOOLEAN = True
16 end
```

The primitive (or container) widgets with coordinate system can now inherit from `EV_FLIPPED_VIEW` instead of `NS_VIEW`. Equation 4.1 and 4.2 are therefore not needed anymore. Flipping coordinates also facilitates the task of drawing points, lines and text on a canvas.

Summary

The mechanisms covered in the above sections are used in a number of classes spread not just over the primitive cluster, but also across the kernel and container clusters. The table below indicates in which classes the aforementioned patterns have been implemented. Although not reported in Table 4.1, descendants of the listed classes adopt the same patterns as the classes themselves.

	Target-Action	Data Sources, Delegates	Flipped View
EV_BUTTON_IMP	✓		
EV_HORIZONTAL_SCROLL_BAR_IMP	✓		
EV_VERTICAL_SCROLL_BAR_IMP	✓		
EV_COMBO_BOX_IMP		✓	
EV_LIST_IMP		✓	
EV_MULTI_COLUMN_LIST_IMP		✓	

EV_TEXT_FIELD_IMP		✓	
EV_DRAWING_AREA_IMP			✓
EV_GRID_IMP			✓
EV_TOOL_BAR_IMP			✓
EV_WINDOW_IMP		✓	✓
EV_BOX_IMP			✓
EV_CELL_IMP			✓
EV_FIXED_IMP			✓
EV_FRAME_IMP			✓
EV_APPLICATION_IMP		✓	

Table 4.1: Classes that implement at least one of the three solutions, ordered by cluster.

4.4 Items Cluster

An item is an object that displays information inside certain primitives. Widgets that represent data using items range from menu bars and tool bars to multi-column lists and trees.

Menu items are fundamentally different with regard to geometric attributes (such as position and size) which makes it impossible to achieve a complete mapping between EiffelVision and Cocoa. In the former, menu items inherit from `EV_POSITIONED`, which is "an abstraction for objects that have geometric attributes". In the latter, menu items do not have a geometric position because the menu bar is fixed by the operating system at the top of the screen and does not interact with the rest of the GUI. A solution to map geometric attributes in Cocoa is not available yet.

4.5 Support Cluster

The support cluster provides a set of helper classes used throughout EiffelVision. Helper classes add functionalities for default and constant features of existing widgets. The classes in this cluster are not discussed since the interesting behavior of widgets is implemented in other clusters.

4.6 Properties Cluster

This cluster contains classes that define common properties for EiffelVision widgets and items. The properties available are:

Colorizable The widget can change color.

Dockable Source The widget represents the source for a pick and drop action.

Dockable Target The widget represents the target for a pick and drop action.

Drawable A widget onto which graphical primitives can be drawn.

Fontable The widget can change font.

Pick and Dropable The widget can be both a source and a target for a pick and drop action.

Pixmapable The widget can have a pixmap.

Sensitive The widget may ignore user input.

Textable The widget has a text label.

Tooltipable The widget has a tooltip.

Except for the pick and drop mechanism, which is not supported in the current port, most functionalities of the property classes are platform independent and thus implemented in the port in a similar way to the GTK version.

4.7 Platform-specific Classes

The port also contains platform-specific abstractions, such as `EV_NS_VIEW` and `EV_FLIPPED_VIEW`. `EV_NS_VIEW` is a class for adding `NS_VIEW` functionalities to EiffelVision. The port has completely rewritten this class in order to replace the layout handling of the legacy port with automatic layout constraints. The Unix and Windows ports put platform-related classes in the support cluster, hence this port adopts the same grouping rule.

In this work, delegates are implemented either in the class that needs to respond to certain events or in a separate class. In the latter case, for a matter of convention, we have decided to place delegate classes in the support cluster.

Chapter 5

Guides

5.1 Developer Guide

This guide is intended to help developers to quickly get familiar with the Cocoa EiffelVision implementation and continue working on it.

The port relies on the wrapped Cocoa frameworks generated by the automatic library converter [4] using Mac OS X 10.7. This is the minimal version required to be able to run the Cocoa EiffelVision library. The library converter automatically generates wrappers for the *AppKit*, *Foundation*, *CoreData* and *QuartzCore* libraries. They are made available in the *objc_wrapper* library of the Eiffel Verification Environment repository[1]. To generate more up to date frameworks or to add new Objective-C frameworks, follow the developers guide in [5]. Assuming that EiffelStudio is already installed, the steps necessary to start working on the port are the following:

1. Define two new environment variables in ∼/.profile.

```
export EIFFEL_SRC=path/to/eve/Src
export ISE_LIBRARY=$EIFFEL_SRC
```

2. Check out the latest source code from the EVE branch.

```
$ svn co https://svn.origo.ethz.ch/eiffelstudio/
    branches/eth/eve/Src/ $EIFFEL_SRC
```

3. From a console, compile the C code for the objc_wrapper library.

```
$ cd $EIFFEL_SRC/library/objc_wrapper/Clib
$ finish_freezing -library
```

[1]The wrapped Cocoa frameworks are available at https://svn.origo.ethz.ch/eiffelstudio/branches/eth/eve/Src/library/objc_wrapper/

4. Download the python script from https://svn.origo.ethz.ch/eve/scripts/compile_es/compile_ec.py and execute it to compile the workbench.

```
$ python compile_ec.py --target bench_cocoa
```

5. Open EiffelStudio and add the *ec* project located in `$EIFFEL_SRC/Eiffel/Ace`. Choose the *bench_cocoa* target and open the project.

Running the project causes EiffelStudio to launch natively on the Mac, but it is not fully working yet.

5.2 User Guide

The user guide explains how to run a project using this port. Please note that it is also necessary to perform the first three steps of the developers guide above because the Cocoa implementation is not part of the official EiffelStudio release.

Open the .ecf file of the project to be compiled using the Cocoa EiffelVision version and add the following line to the target:

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

Alternatively, open the project with EiffelStudio and click `Project Settings` under the `Project` menu. Then add a new variable to the target as depicted in Figure 5.1.

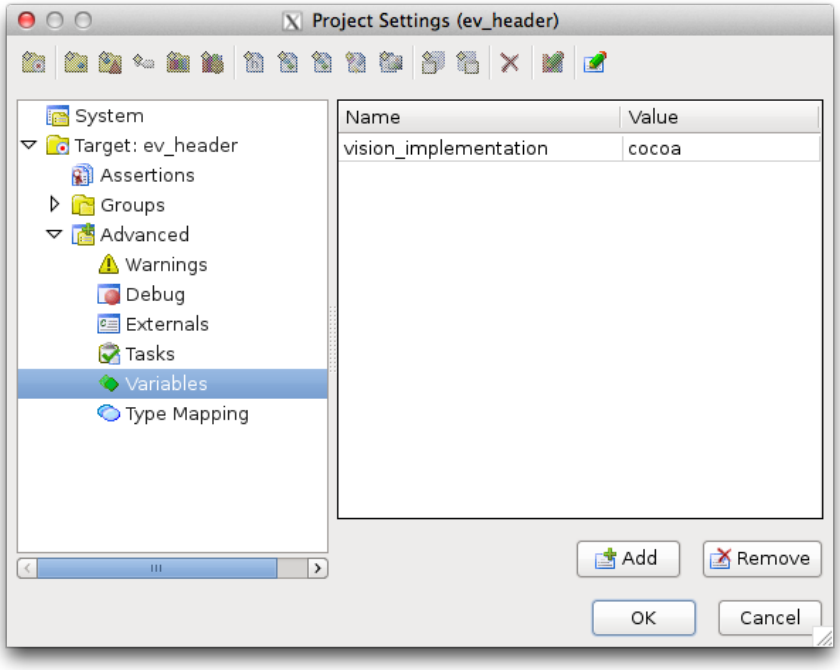

Figure 5.1: Adding a new variable to the project's target.

Chapter 6

Conclusion

This work focused on improving the legacy port by taking a more systematic approach under two different aspects:

- The layout system for containers and widgets has been simplified and extended to support resizing.
- A number of methods to map widgets have been defined and applied in a consistent manner throughout the port.

As a consequence, the complexity of the port reduced both in terms of structure (throwing away abstract helper classes) and code size. The current port, while adding more functionalities, has lost about 10% lines of code with respect to the legacy port. Although a solid foundation for the Cocoa EiffelVision implementation is provided, there are situations in which a mapping from EiffelVision to Cocoa is simply not achievable with the current support from Cocoa. Some of those issues have already been identified in [2]:

- EiffelVision provides an icon for each window, while Cocoa only allows one icon per application.
- Keyboard shortcuts often include the command key, which is only available on Mac keyboards. These shortcuts are therefore not recognizable in EiffelVision.
- The font, color and print panels are one per application in the Cocoa environment, while EiffelVision allows to have multiple instances of these panels.

Possible future work include completing the functionalities of widgets and proposing new solutions to situations in which there is a substantial difference between EiffelVision and Cocoa.

Appendix A

Development Status

This appendix provides a description of the implementation status of each EiffelVision class. The goal is to give future developers an overview of what is left to implement or what could be improved. Where a mapping from EiffelVision to Cocoa is difficult to achieve, a brief explanation of why that is the case is provided.

The whole event cluster has been successfully implemented and its classes are not discussed in this section because the implementation is trivial.

The pick and drop mechanism is currently not supported on Mac OS X.

Note that even if a class is (in part) implemented, it cannot be assumed that its behavior is correct because testing a GUI library is a difficult and time consuming task. The lack of documentation makes it even harder to understand what is assumed to be the correct behavior of a widget and its functionalities.

A.1 Kernel Cluster

EV_ACCELERATOR_IMP

All the functionalities of this class have been implemented. As already mentioned before, the command key is not recognized by EiffelVision.

EV_ANY_IMP

All the functionalities of this class have been implemented.

EV_APPLICATION_IMP

The functionalities of this class are only partially implemented.

- Detection of pressed keys such as *ctrl*, *shift*, *alt*, *caps* is not working. In Cocoa the so called modifiers keys are specified for each keyboard or mouse event triggered by the user. The idea behind this approach is that in practice it is useful to know which modifier key is pressed only in combination with another key or mouse button.
- `system_color_change_actions` is never called because `NS_APPLICATION` cannot detect changes of the screen color's depth.
- Cocoa does not allow to change the tooltip delay and therefore `set_tooltip_delay` cannot be mapped.

EV_BITMAP_IMP

The functionalities of this class are only partially implemented.

- `set_default_colors` only assigns the default background color to the bitmap. The foreground color property does not exist in Cocoa.

EV_CHARACTER_FORMAT_IMP

The functionalities of this class are only partially implemented.

- Features exported to `EV_RICH_TEXT_IMP` are not functional because this class is not fully implemented yet.

EV_CLIPBOARD_IMP

All the functionalities of this class have been implemented. The Cocoa clipboard allows any kind of object to be copied in the clipboard, hence the port restricts the clipboard to just contain strings.

EV_COLOR_IMP

All the functionalities of this class have been implemented. Most features are platform independent and are equivalent to the GTK implementation.

EV_ENVIRONMENT_IMP

All the functionalities of this class have been implemented, but the image formats supported could be further extended to more than just PNG.

EV_FONT_IMP

The functionalities of this class are only partially implemented. An important addition to the implementation in [2] is the conversion of font size from pixel to points and vice versa; before that, a bug caused the text to be displayed very small (1 or 2 points tall). The missing functionalities are listed below.

- `set_family (a_family: INTEGER)` has to map the integer argument to a string because they are handled as such in Cocoa.
- `update_font_face` could not be implemented because of a lack of specifications.

EV_INTERMEDIARY_ROUTINES

The functionalities of this class have not been implemented. It may be a helper class borrowed from the GTK implementation in the legacy port.

EV_PARAGRAPH_FORMAT_IMP

The functionalities of this class are only partially implemented.

- `new_paragraph_tag_from_applicable_attributes` could not be implemented because of a lack of specifications.

EV_POINTER_STYLE_IMP

The functionalities of this class are only partially implemented. In Cocoa, the hotspot of a pointer cannot be changed after the creation procedure. It is therefore necessary to create a new cursor — equal to the previous one — each time `set_x_hotspot` or `set_y_hotspot` are called.

EV_TIMEOUT_IMP

The functionalities of this class are only partially implemented. The Cocoa timeout, when fired, does not call the timeout action on the proper target. It might be necessary to implement the target-action mechanism by subclassing `NS_TIMER`.

A.2 Widgets - Container Cluster

EV_BOX_IMP

The functionalities of this class are only partially implemented.

- `set_border_width` has been disabled because it only works properly for a single box in a container. When multiple widgets are present, it must be first checked which borders are touched by the box.

EV_CELL_IMP

All the functionalities of this class have been implemented.

EV_CONTAINER_IMP

The functionalities of this class are only partially implemented. In Cocoa, there is no notion of radio groups for a container (the only way to achieve this is by creating a `NS_MATRIX`, but it only works for a restricted set of widgets) and thus the functionalities offered in EiffelVision cannot be mapped to Cocoa.

EV_DYNAMIC_LIST_IMP

All the functionalities of this class have been implemented.

EV_FIXED_IMP

The functionalities of this class are only partially implemented.

- `set_item_position` has been disabled because otherwise the `EV_GRID` is not drawn correctly. This problem occurs because the grid widget tries to set the item's position way out of the visible rectangle (e.g. `set_item_position (a_widget, 15000, 15000)`.

EV_FRAME_IMP

All the functionalities of this class have been implemented. Note that in Cocoa the background color can only be set under certain conditions.

EV_HORIZONTAL_BOX_IMP

The functionalities of this class are only partially implemented.

- Expandable widgets in a horizontal box do not have the same width because if an item is removed the layout constraints can potentially break (if the removed item is present in layout constraints of other widgets). This can be fixed, but it requires a more involved implementation and thorough testing.
- As a consequence of the above issue, items cannot be set to be non-expandable.

EV_HORIZONTAL_SPLIT_AREA_IMP

All the functionalities of this class have been implemented.

EV_MENU_ITEM_LIST_IMP

The functionalities of this class have been implemented, but the radio group features need to be tested more thoroughly.

EV_NOTEBOOK_IMP

The functionalities of this class are only partially implemented.

- `pointed_tab_index` In Cocoa the notebook widget does not offer this functionality. It may be possible, however, to compute the cursor coordinates and check it they are in a notebook tab region
- Notebook tabs do not support pixmaps in Cocoa.

EV_POPUP_WINDOW_IMP

All the functionalities of this class have been implemented.

EV_SCROLLABLE_AREA_IMP

All the functionalities of this class have been implemented.

EV_SPLIT_AREA_IMP

The functionalities of this class are only partially implemented.

- `disable_item_expand` and `enable_item_expand` have been disabled because they are not working properly yet. The layout constraints need to be fixed.
- `set_split_position` cannot be mapped because Cocoa does not offer this functionality.

EV_TABLE_IMP

Some functionalities of this class have been implemented in the legacy port, but it is not complete yet. This class, however, is a combination of horizontal and vertical boxes and is thus not essential for the basic implementation of EiffelVision.

EV_TITLED_WINDOW_IMP

The functionalities of this class are only partially implemented.

- EiffelVision allows to set a pixmap for each window, while Cocoa allows only one pixmap per application. A mapping is not achievable in this case.

EV_VERTICAL_BOX_IMP

The functionalities of this class are only partially implemented.

- Expandable widgets in a vertical box do not have the same height because some widgets in Cocoa have a fixed height.
- Widgets that can expand their height do not have all the same height because of the issue above.

EV_VERTICAL_SPLIT_AREA_IMP

All the functionalities of this class have been implemented.

EV_VIEWPORT_IMP

The functionalities of this class are only partially implemented. Even if in section 4.3.1 the mapping is said to be trivial, there are issues with the `EV_GRID` widget.

- `set_x_offset` and `set_y_offset` have been disabled because, once again, the `EV_GRID` widget tries to set offsets way out of the visible range. Since the grid widget is widely used, it has been decided, for now, to disable these two functionalities. This might cause problems to other widgets.

EV_WIDGET_LIST_IMP

All the functionalities of this class have been implemented.

EV_WINDOW_IMP

The functionalities of this class are only partially implemented. In EiffelVision each window can have its own menu, while in Cocoa there is only one menu per application. We proposed a solution that tries to mimic the one-menu-per-window EiffelVision behavior. The port detects the Cocoa event of a window becoming the main one (i.e. the one that will respond to user actions) and changes the application menu to the one held by the window.

- `set_maximum_width` and `set_maximum_height` have not been implemented yet. Note that in Cocoa the maximum size can only be set once using `set_max_size_`, and thus these two features should be somehow called together otherwise only one of the two dimensions will be set.
- Interaction with accelerators (called shortcuts or key bindings in Cocoa) has not been implemented yet.
- `hide` is crashing whenever calling the Cocoa equivalent. The functionality has been disabled but it needs to be fixed.

A.3 Widgets - Dialogs

Dialog widgets could not be further implemented from the legacy port for a lack of time. Moreover, it may be extremely difficult — if not impossible — to map some of these widgets (e.g. the color, font and print dialogs) because they are one per application in the Cocoa environment, while EiffelVision supports multiple instances of the mentioned widgets.

A.4 Widgets - Primitive Cluster

EV_BAR_ITEM_IMP

This class has been implemented, although it does not offer any functionalities.

EV_BUTTON_IMP

The functionalities of this class are only partially implemented.

- `enable_default_button` causes the application to crash. The causes of this bug are currently not known yet, but it might have something to do with the animation of the button (the pulsing effect) performed in another GUI thread.

EV_CHECK_BUTTON_IMP

All the functionalities of this class have been implemented.

EV_CHECKABLE_LIST_IMP

All the functionalities of this class have been implemented.

EV_CHECKABLE_TREE_IMP

The checkable tree has not been implemented yet. The strategy to implement this widget is to adopt the mechanism described in Section 4.3.3 using the `NS_OUTLINE_VIEW_DATA_SOURCE_PROTOCOL` and `NS_OUTLINE_VIEW_DELEGATE_PROTOCOL` .

EV_COMBO_BOX_IMP

The functionalities of this class are only partially implemented. In contrast to EiffelVision, Cocoa allows at most one selected item per combo box and thus `selected_items` always returns a list with one item, which is equal to the result of `selected_item`.

EV_DRAWING_AREA_IMP

The functionalities of this class have been implemented, although `redraw_rectangle` is not efficient because of an unresolved conflict with the grid widget.

EV_GAUGE_IMP

All the functionalities of this class have been implemented. All features are platform independent.

EV_GRID_IMP

The functionalities of this class are only partially implemented. The grid is a very difficult widget to map even if most of the implementation is done in the interface class, i.e. it is platform independent. In Figure A.1 it is showed how the widget is built in the interface class.

Two reasons of why the mapping would work better if the widget was completely platform dependent are given here:

- The `EV_GRID` mainly contains two viewports: one for displaying the headers and one for drawing the grid items. The widget re-implements the whole scrolling machinery because only the second viewport needs to be scrolled vertically, while both the first and second viewport must scroll together horizontally. The platform independent code to adopt this behavior works well with the GTK and WEL implementations, but it is not working with the Cocoa one. For the Cocoa implementation it would suffice to put a `NS_TABLE_VIEW` inside a `NS_SCROLL_VIEW` and the behavior described is automatically handled by the Cocoa frameworks.
- The EiffelVision implementation of the grid has an additional platform-independent helper class (`EV_GRID_DRAWER_I`) in charge of drawing the grid items and line separators, which would not be necessary in the Cocoa

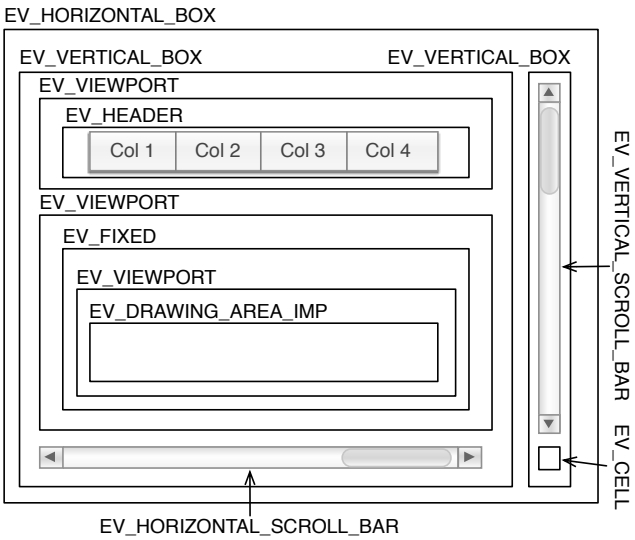

Figure A.1: Structure of the `EV_GRID` widget class

implementation as the appearance of the items is handled by the table delegate and the line separators are automatically drawn by the system.

The `EV_GRID` widget, implemented specifically for Mac OS X, would require much less code and look simpler in terms of structure of the widget. Figure A.2 depicts a possible implementation of the grid widget for this port. Such implementation is only possible if the current implementation of the grid is moved from the interface class, `EV_GRID_I`, to the platform dependent (Unix, Windows and Mac) classes, `EV_GRID_IMP`.

EV_HEADER_IMP

The functionalities of this class are only partially implemented.

- `call_item_resize_actions` is not triggered when the Cocoa widget is resized.
- Cocoa header items do not support pixmaps, therefore `set_pixmap` is not working and `pixmaps_size_changed` is never called.

EV_HORIZONTAL_PROGRESS_BAR_IMP

All the functionalities of this class have been implemented.

EV_HORIZONTAL_RANGE_IMP

All the functionalities of this class have been implemented.

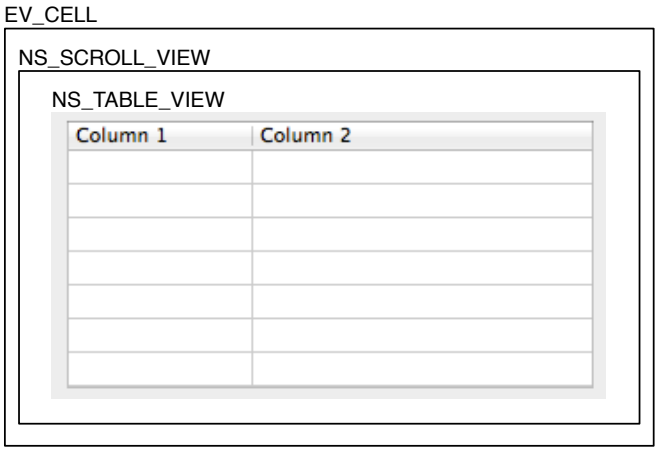

Figure A.2: Structure of the `EV_GRID` widget class using a platform dependent implementation

EV_HORIZONTAL_SCROLL_BAR_IMP

All the functionalities of this class have been implemented.

EV_HORIZONTAL_SEPARATOR_IMP

All the functionalities of this class have been implemented.

EV_LABEL_IMP

The functionalities of this class are only partially implemented.

- `set_font` has been disabled because the font received as formal argument does not always have the correct size. This bug is caused by the incomplete implementation of `EV_FONT_IMP`.

EV_LIST_IMP

The functionalities of this class are only partially implemented. Selection and deselection of (possibly multiple) items has not been implemented yet.

EV_LIST_ITEM_LIST_IMP

The functionalities of this class are only partially implemented. Insertion of text and a pixmap has not been implemented yet.

EV_MULTI_COLUMN_LIST_IMP

The features of this class have not been implemented yet. The legacy port uses the delegate and data source of a `NS_OUTLINE_VIEW` while it should actually implement the `NS_TABLE_VIEW` ones.

EV_PASSWORD_FIELD_IMP

All the functionalities of this class have been implemented.

EV_PND_DEFERRED_ITEM_PARENT

The functionalities of this class have not been implemented because of a lack of specifications.

EV_PRIMITVE_IMP

The functionalities of this class are only partially implemented.

- It is not clear, from the available documentation, what are the functionalities of `enable_tabable_to` and `disable_tabable_to`.

EV_PROGRESS_BAR_IMP

The functionalities of this class are only partially implemented.

- `enable_segmentation` and `disable_segmentation` have been disabled because a crash occurs when animating the progress bar. This bug could happen for the same reason as the bug in `EV_BUTTON_IMP`.

EV_RADIO_BUTTON_IMP

All the functionalities of this class have been implemented.

EV_RANGE_IMP

The functionalities of this class are only partially implemented. The Cocoa widget does not call `change_actions` when the value of the slider changes.

EV_RICH_TEXT_IMP

The functionalities of this class are only partially implemented. More research must be done to understand how rich text is handled in the Cocoa frameworks.

EV_SCROLL_BAR_IMP

All the functionalities of this class have been implemented.

EV_SEPARATOR_IMP

All the functionalities of this class have been implemented.

EV_SPIN_BUTTON_IMP

The functionalities of this class are only partially implemented. To create a spin button in Cocoa a `NS_TEXT_FIELD` and a `NS_STEPPER` must be bound in order to reflect accordingly the changes.

- The Cocoa stepper does not trigger `change_actions`. The target-action mechanism described in Section 4.3.3 should be used.

EV_TEXT_COMPONENT_IMP

All the functionalities of this class have been implemented.

EV_TEXT_FIELD_IMP

The functionalities of this class are only partially implemented.

- `capacity` is not available in Cocoa, which assumes a possibly infinite sequence of characters in a text field.
- Insertion or removal of text at a given position is not functional yet, as well as selecting portions of the text.

EV_TOGGLE_BUTTON_IMP

The functionalities of this class are only partially implemented.

- `set_pixmap` is not applicable to a toggle button yet. Changing the bezel style of the Cocoa button, however, should solve the problem.

EV_TOOL_BAR_IMP

The functionalities of this class are only partially implemented.

- `remove_item` has not been implemented yet. Particular care for the layout constraints must be taken.

EV_TREE_IMP

The functionalities of this class have only been partially implemented because of a lack of time. The strategy to follow is to implement the `NS_OUTLINE_VIEW` delegate and data source using the mechanism described in Section 4.3.3.

EV_VERTICAL_PROGRESS_BAR_IMP

All the functionalities of this class have been implemented.

EV_VERTICAL_RANGE_IMP

The functionalities of this class are only partially implemented and they need to be thoroughly tested.

EV_VERTICAL_SCROLL_BAR_IMP

All the functionalities of this class have been implemented.

EV_VERTICAL_SEPARATOR_IMP

All the functionalities of this class have been implemented.

A.4.1 Widgets - Other Classes

EV_MENU_IMP

The functionalities of this class are only partially implemented.

EV_PIXMAP_IMP

The functionalities of this class are only partially implemented.

- `set_with_default` should set the pixmap to the default EiffelVision2 logo, however it is not specified where this pixmap can be found.
- `raw_image_data` needs to read the bitmap values and write them as data in the result.

EV_SCREEN_IMP

The functionalities of this class are only partially implemented.

- `widget_at_position` and `widget_imp_at_pointer_position` have not been implemented yet, but a suggestion of how these features could be implemented is given. In Cocoa, to find the widget at a certain position, it is necessary to find the active window and to perform a `hit_test` on a given coordinate point; the method will return the farthest view in the hierarchy.
- The fake pointer button press features have not been implemented yet. To fake a keystroke in Cocoa, one needs to create a new instance of a `NS_EVENT` and then forward it to the `NS_APPLICATION` application by calling `application.post_event__at_start_ (event, True)`.

EV_WIDGET_IMP

The functionalities of this class are only partially implemented.

- `pointer_position` has not been implemented yet. It must be first determined which coordinate systems the widget and the screen are using.
- `is_displayed` is partly based on the legacy layout implementation and thus does not always return the correct result.
- `on_key_event` needs more testing.

A.5 Items Cluster

Note that, as already mentioned in section 4.4, some items that inherit from `EV_POSITIONED` do not have an equivalent mapping in Cocoa.

EV_CHECK_MENU_ITEM_IMP

All the functionalities of this class have been implemented.

EV_HEADER_ITEM_IMP

The functionalities of this class are only partially implemented.

- Header items cannot hold a pixmap in Cocoa.

EV_ITEM_IMP

All the functionalities of this class have been implemented.

EV_LIST_ITEM_IMP

The functionalities of this class are only partially implemented.

- `is_selected`, `enable_select` and `disable_select` are not working properly.

EV_MENU_BAR_IMP

All the functionalities of this class have been implemented.

EV_MENU_ITEM_IMP

The functionalities of this class are only partially implemented.

- The `select_actions` action sequence must be triggered using the target-action mechanism described in Section 4.3.3.

EV_MENU_SEPARATOR_IMP

All the functionalities of this class have been implemented.

EV_MULTI_COLUMN_LIST_ROW_IMP

Previously implemented in [2], but testing is still necessary.

EV_PND_DEFERRED_ITEM

All the functionalities of this class have been implemented.

EV_RADIO_MENU_ITEM_IMP

All the functionalities of this class have been implemented.

EV_TOOL_BAR_BUTTON_IMP

The functionalities of this class are only partially implemented.

- `select_actions` must be triggered using the target-action mechanism described in Section 4.3.3.
- `set_vertical_button_style` is not supported yet.

EV_TOOL_BAR_DROP_DOWN_BUTTON_IMP

There are no functionalities to implement for this widget.

EV_TOOL_BAR_RADIO_BUTTON_IMP

All the functionalities of this class have been implemented.

EV_TOOL_BAR_SEPARATOR_IMP

All the functionalities of this class have been implemented.

EV_TOOL_BAR_TOGGLE_BUTTON_IMP

All the functionalities of this class have been implemented.

EV_TREE_ITEM_IMP

There are no functionalities to implement for this widget.

EV_TREE_NODE_IMP

Previously implemented in [2], but testing is still necessary.

A.6 Properties Cluster

EV_COLORIZABLE_IMP

The functionalities of this class are only partially implemented.

- Setting the background color in Cocoa is a procedure that may change from one widget to another and therefore must be done at a deeper level in the class hierarchy.

EV_DOCKABLE_SOURCE_IMP

Currently not supported.

EV_DOCKABLE_TARGET_IMP

Currently not supported, although this class has actually no functionalities.

EV_DRAWABLE_IMP

The functionalities of this class are only partially implemented. Drawing arcs, ellipsoid text, rotated text, ellipses and polylines is not supported yet. This is in partly due to the fact that the wrapper does not support `doubles` as arguments

and return values. In most drawing functions, however, `doubles` as arguments and return values are required.

EV_FONTABLE_IMP

All the functionalities of this class have been implemented. Note that the font can be set to the Cocoa widget only at a more specific level.

EV_PICK_AND_DROPABLE_IMP

Currently not supported.

EV_PIXMAPABLE_IMP

All the functionalities of this class have been implemented. In a similar way to the fontable property, also the pixmap must be later applied to the Cocoa widget.

EV_SENSITIVE_IMP

The property of being sensitive or insensitive is not very well defined in Cocoa. Some widgets can be set to be enabled or disabled, other widgets must be set to either respond to or ignore user events. It is therefore not really meaningful to talk about achieving a mapping for this widget.

EV_TEXTABLE_IMP

The functionalities of this class have been implemented, but the Cocoa text label must be specified at the widget level.

EV_TOOLTIPABLE_IMP

All the functionalities of this class have been implemented.

A.7 Support Cluster

EV_APPLICATION_DELEGATE

The application delegate takes care of terminating the application when the last window is closed, in order to match the behavior in the Unix and Windows implementations.

EV_BEEP_IMP

The functionalities of this class have been implemented.

EV_COCOA_KEY_CONVERSION

The functionalities of this class have been implemented in the legacy port. This class provide functionalities for converting key codes between Cocoa and EiffelVision.

EV_COMBO_BOX_DELEGATE

This helper class allows Cocoa combo boxes to be notified whenever the selection changes.

EV_FLIPPED_VIEW

The `EV_FLIPPED_VIEW` is needed by Cocoa widgets to flip their coordinates system and match the EiffelVision one.

EV_ITEM_LIST_IMP

This class has been implemented and it is the abstraction for widgets that contain a list of items.

EV_MODEL_PRINT_PROJECTOR_IMP

A class that makes a standard projection of a model on a printer device. The implementation is identical to the GTK version.

EV_NOTEBOOK_TAB_IMP

The functionalities of this class are only partially implemented.

- A Cocoa notebook tab cannot be assigned a pixmap, therefore `set_pixmap` and `remove_pixmap` have not been mapped.

EV_NS_RESPONDER

This class has been implemented in the legacy port. It provides functionalities for capturing Cocoa key events and forwarding them to EiffelVision.

EV_NS_VIEW

The `EV_NS_VIEW` class abstracts the functionalities of `NS_VIEW` to apply to EiffelVision widgets, such as the position (relative to both their superviews and screen coordinates) and the size attributes.

`EV_NS_VIEW` also implements the functionalities for dealing with layout constraints. A brief overview of the applicable constraints is given in the list below.

- `set_ATTRIBUTE_constraint`, where `ATTRIBUTE` is one of `left`, `top`, `right` or `bottom`, applies a fixed padding constraint to the current widget and its superview, e.g. `H:|-10-[button]`.
- `set_minimum_ATTRIBUTE_constraint` are equivalent to the previous features, but they allow the padding to be greater or equal the given argument, e.g. `V:|-(>=10)-[frame]`.
- `set_fixed_width_constraint` and `set_fixed_height_constraint` block the size of the current widget. The former feature applies a constraint of the type `H:[button(100)]` to set the width of a button to be 100 pixels.
- `set_minimum_width_constraint` and `set_minimum_height_constraint` are equivalent to `set_fixed_width_constraint` and `set_fixed_height_constraint`, respectively, but they allow the widget to expand its size. The latter feature applies a constraint of the type `V:[vertical_box(>=200)]` to set the minimum height of a vertical box to 200 pixels.
- The `EV_FIXED_IMP` widget also uses the `set_position_constraints` to place widgets anywhere in the container. This feature is actually a combination of `set_left_padding` and `set_top_padding`.
- The operations for updating the constraints must be called whenever the padding of a widget changes (by calling `set_padding`).

EV_NS_WINDOW

This class has been implemented in the legacy port and it abstracts `NS_WINDOW` functionalities for EiffelVision windows and dialogs.

EV_PIXEL_BUFFER_IMP

The functionalities of this class are only partially implemented. Writing to a buffer data is not supported because the Cocoa wrapper currently does not handle byte streams as arguments and return values.

EV_PRINT_PROJECTOR_IMP

All the functionalities of this class have been implemented.

EV_RADIO_PEER_IMP

This class has been implemented in the legacy port and needs to be tested.

EV_REGION_IMP

The features of this class have not been implemented yet.

EV_SINGLE_CHILD_CONTAINER_IMP

This class abstracts common functionalities for containers that can hold at most one item, i.e. `EV_CELL_IMP` and `EV_WINDOW_IMP` (and respective descendants).

EV_STOCK_COLORS_IMP

The functionalities of this class have been implemented, although the specifications are not really clear.

EV_STOCK_PIXMAPS_IMP

The functionalities of this class are only partially implemented, because not all stock pixmaps are included in Mac OS X.

EV_TABLE_CHILD_IMP

This class has been implemented in the legacy port and needs to be tested.

EV_TEXT_FIELD_DELEGATE

This is a helper class that detect changes in a Cocoa text field.

Bibliography

[1] Apple Inc.: Cocoa AutoLayout Guide. http://developer.apple.com/library/mac/documentation/UserExperience/Conceptual/AutolayoutPG/AutolayoutPG.pdf (2011)

[2] Daniel Furrer: EiffelVision for Mac OS X. http://se.inf.ethz.ch/old/projects/daniel_furrer/report.pdf (2011)

[3] Eiffel Software: EiffelVision Library Reference Manual. http://docs.eiffel.com/book/solutions/eiffelvision-library-reference-manual

[4] Matteo Cortonesi: Objective-C Frameworks to Eiffel Converter. http://objc-frameworks-to-eiffel-converter.origo.ethz.ch/

[5] Matteo Cortonesi: Objective-C Frameworks to Eiffel Converter. http://se.inf.ethz.ch/old/projects/matteo_cortonesi/report.pdf (2011)