Application development with Edje

Title:	Application development with Edje
Subtitle:	From the very basics
Author:	Andres Blanc
Contact:	andresblanc@gmail.com
Version:	1.0
Date:	2008/04/03
Copyright:	This book is distributed under the terms of the "Attribution" Creative Commons license. See: http://creativecommons.org/licenses/by/3.0/.

Abstract

An introduction to GUI based application development. Starts from the most basic concepts and introduces the EFL libraries that deal with each of them. It covers Edje, Ecore, Evas and EWL. This book is intended to cover all the concepts needed by a novice programmer to create a full blown Edje based application.

Contents

Chapter 1

Book overview

- About Graphical User Interfaces. If we look through the code of some of the many open source applications available, we can find most of them share a common structure. This chapter introduces the reader to said structure, to the concept of an Event loop and how does it deal with the interaction between form and function.

  - Decomposing the frontend. There are different approaches to the creation of Graphical User Interfaces, some provide flexibility while others shorter development time. In the end, the rest of the application will have to go through the same mechanism to deal with the interface.

  - Introduction to Edje. Edje is a compromise between flexibility and development time. Comparing it to a plain canvas or a fully featured toolkit can show us how it fits in the middle ground and provides advantages for designers and developers alike.

  - The foundations. Edje allows low level control of the interface without forcing the developer through a low level API. The interface objects are implemented as Evas objects, but through a completely new language friendly to designers and independent from the application code.

  - Convenient libraries To load an Edje interface the application needs to setup an Evas canvas first. This would look as a burden for the developer, dealing with issues specific to the underlying system. Thankfully Evas is supported by an additional library that resolve most of these issues, Ecore.

- The foundations in practice. Practical examples of the Englighenment Foundation Libraries required for a common Edje application.

  - Working with the canvas. Including new Edje objects inside the previously setup canvas can be explained with this simple example.

  - Interacting with the objects. There are different channels for the application to interact with the interface. For the sake of brevity I will include an example

of each one to serve as an introduction for the practical examples to come in the next chapters.

– Building a framework. Hopefully the reader's mind won't be filled already with preconceptions about this subject. In any case, only the foundations of what could be used to form a complex framework are going to be reviewed in this chapter.

* Simpler library intialization. We have seen why and how to initialize the necessary EFL libraries. This chapter introduces the implementation of a convenience function to do it in one call along with control and intialization of configuration and theme files.

* Simpler window setup. Altought setting up a window is not a complex task there is no reason for not using a standarized method that automates some aspects of window creation. Some other aspects are left for the designer to manipulate, the end result is less code which often means less bugs.

* Simpler theme management. The EFL are programmed in a incredible flexible way and they cover many uses and platforms. The functions explained in this section makes some assumptions about the programmer's intentions. If they match with yours, using them can mean simpler application code, less development time and less bugs.

• Introduction to widgets. Interfaces need to resolve two problems, presenting information to users and taking orders from them. The past chapters presented enough information to solve the first. In the following chapters we will review the second.

– Widgets with Edje. Since this is a book about Edje it might be a good idea to detail the way Edje simplifies custom widget creation by reviewing the list of tasks presented in the previous chapter and how Edje helps with each task.

– The Ewl as a shortcut. The Enlightened Widget Library is a piece of software older than Edje itself. But almost as soon as Edje was made aviable the library developers started using it as their theme engine. Since widget's from this library can be included as another Edje object we can use it as a shortcut for the most complex widgets.

– Implementing widgets with Edje. It is possible to fully implement widgets using the Edje library alone. But this approach grows more unmanageable as the widget becomes more complex. In this chapter we will review a simple widget, written using Edje alone and a more complex widget that shares the same foundations with Edje objects.

* Introduction to Smart Objects. We have seen Smart Objects in the previous chapters and we know they are used to implement Edje objects. In

this chapters we will analyze the surface code of a Smart Object based widget, the minimap.

* The foundations of a Smart Object. If you felt that basing Edje widgets in Smart Objects seems simple enough, I hope that impression does not fade away because of the amount of code presented in this chapter. Altought at a first look it might seem complex creating an Smart Object is quite simple and logical.

– Laying out widgets. TODO.

– Creating list based widgets. TODO.

• Widgets by Example. TODO,

  – The text entry. TODO

  – The combox. TODO.

  – The kinetic list. TODO.

  – The media viewport. TODO.

  – Flexible toolbars. TODO.

Chapter 2

About Graphical User Interfaces

So... you want to create a GUI application? I assume so since you choosed this book as instructive, or at least bathroom, material. You could google "GUI" and "library" to feel overwhelmed by the large number of development libraries avilable. As you look through the source of your (open source) favorites you will realize that all of them, and the applications that use them, share a common structure. In this chapter we will review that structure.

At this point it is convenient to note that the concepts seen in this book and the applications resulting from them translate painlessly to special purpose computers or embed devices running under alternative architectures like ARM or PPC. Edje and related libraries are not only efficient but portable.

The structure of the averange graphical application is built around a concept known as event (or signal) driven execution. Event driven applications are persistent and depend on a gate guardian to call the appropriate functions in the appropriate moment. This gate guardian is known as the Event loop (or main loop).
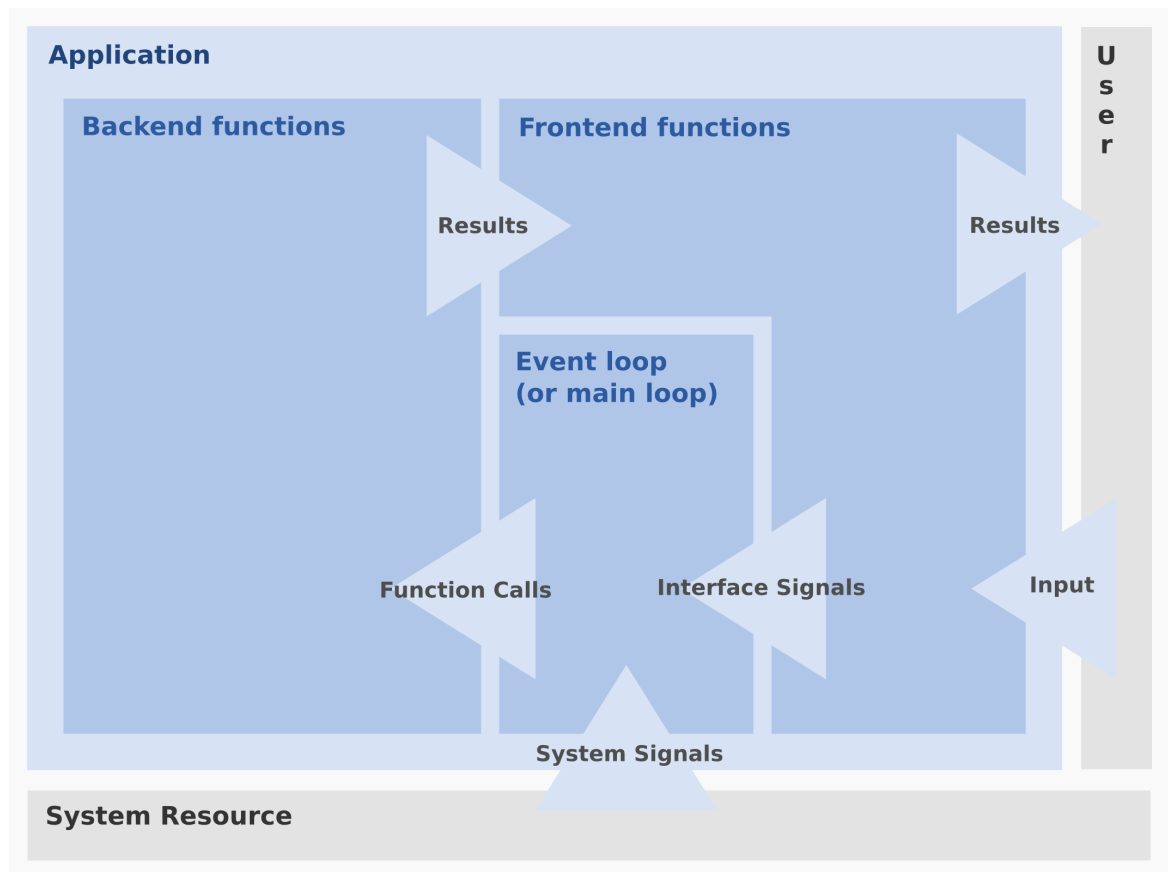
The functions that form a event driven application (with a GUI) can be split among two groups. The first group is known as the backend, these functions deal with the actual purpose of the application, crunching numbers, decoding media files and so on. The second group is known as the frontend, the prupose of these functions is to present the results from the backend to their human overlords and to receive orders from them.

Between the backend and the frontend is where the Event loop lives, its mission is to connect both ends of the application. Not only between them but also with their environment. The Event loop mantains a list of signals to look out for and functions related to them. When a signal is received, the Event loop looks it up in a list and executes the corresponding function or functions.

For the application to work, the Event loop needs to be aware of events in the interface, thus it is usually provided by the same library that provides the GUI elements. It also needs to be aware of events in the system where the application is running. Even when the concept is simple, creating a portable and properly abstracted event loop is no simple task.

2.1 Decomposing the frontend

There are many libraries that aid creation of a GUI. From a plain canvas consisting of primitive design objects, like a line or a rectangle, to complex layout schemes and predefined interface elements, the latter known as "toolkit" or widget library. As opposed to the first, in a toolkit the canvas is just another widget.

Regardless of the method of choice, the resulting GUI has to provide the same resources to the rest of the application. A mechanism to present information to the user, a mechanism to know when the user interacts with the interface and a mechanism to retreive information that resulted from said interaction.

In the case of the plain canvas the application developer must assemble the interface elements, know as widgets, using primitive objects. A very simple text entry widget could consist of a rectangle and a string of text. Besides assembling the widget, the developer has to instruct the Event loop to call a given function on a given interface event on either primitive object. It is possible to discriminate between, for example, a click in the rectangle from a click in the string.

In the case of a toolkit library the application developer would simply include a predefined "text entry" widget. The events from this object would be dealt with in terms of the object as a whole. It wouldn't seem that there is much difference between using a canvas or using a toolkit library until we consider all the possiblities, like focus, overflowing text, copying, pasting, selecting, etc.

Deciding which approach to use is, of course, up to the developer to decide as each one provide capabilities useful for different types of applications. But as we will find out further ahead in this book, these examples only represent opposite extremes and there is an alternative approach that sits quite comfortably in the middle. Edje.

2.2 Introduction to Edje

Allow me to begin this chapter with a quote from the introduction of Edje's API Reference. After all, I cannot expect to give Edje a better introduction than its creator:

> Edje is a complex graphical design and layout library. [..]

> [..] Edje should serve all the purposes of creating visual elements (borders of windows, scrollbars, etc.) and allow the designer the ability to animate, layout and control the look and feel of any program using Edje as its basic GUI constructor. This library allows for multiple collections of Layouts in one file, sharing the same image database and thus allowing a whole theme to be conveniently packaged into 1 file and shipped around.

Edje [..] separates the layout and behavior logic. Edje files ship with an image database, used by all the parts in all the collections to source graphical data. [..] Each part collection consists of a list of visual parts, as well as a list of programs. A program is a conditionally run program that if a particular event occurs (a button is pressed, a mouse enters or leaves a part) will trigger an action that may affect other parts. In this way a part collection can be "programmed" via its file as to hilight buttons when the mouse passes over them or show hidden parts when a button is clicked somewhere etc. The actions performed in changing from one state to another are also allowed to transition over a period of time, allowing animation.

[..] This separation and simplistic event driven style of programming can produce almost any look and feel one could want for basic visual elements. Anything more complex is likely the domain of an application or widget set that may use Edje as a convenient way of being able to configure parts of the display.

As we have seen in the past chapters, there are roughly two methods for creating Graphical User Interfaces. In both cases it had to be implemented through a programming language. In one case, an API was used by the application developer to assemble interface elements from more primitive objects, in the other case the API was used to include objects already defined by a library. Any changes to an nterface object beyond simplistic style modifications had to be submitted by the designer to the developer. Layout changes were only slightly less difficult than behavior changes. The idea of working on the interface objects and their composition in a live environment, like a web developer does, was pretty much unthinkable.

This is where the Edje library fits in, it liberates the designer and the developer from eachother. The artists uses the Edje Data Collection language to manipulate primitive objects. EDC is in some ways comparable to Cascading Style Sheets but its free of the framework that markup imposes. From the other side, the developer only has to include the resulting object and setup the callbacks (by the main loop) to the backend functions.
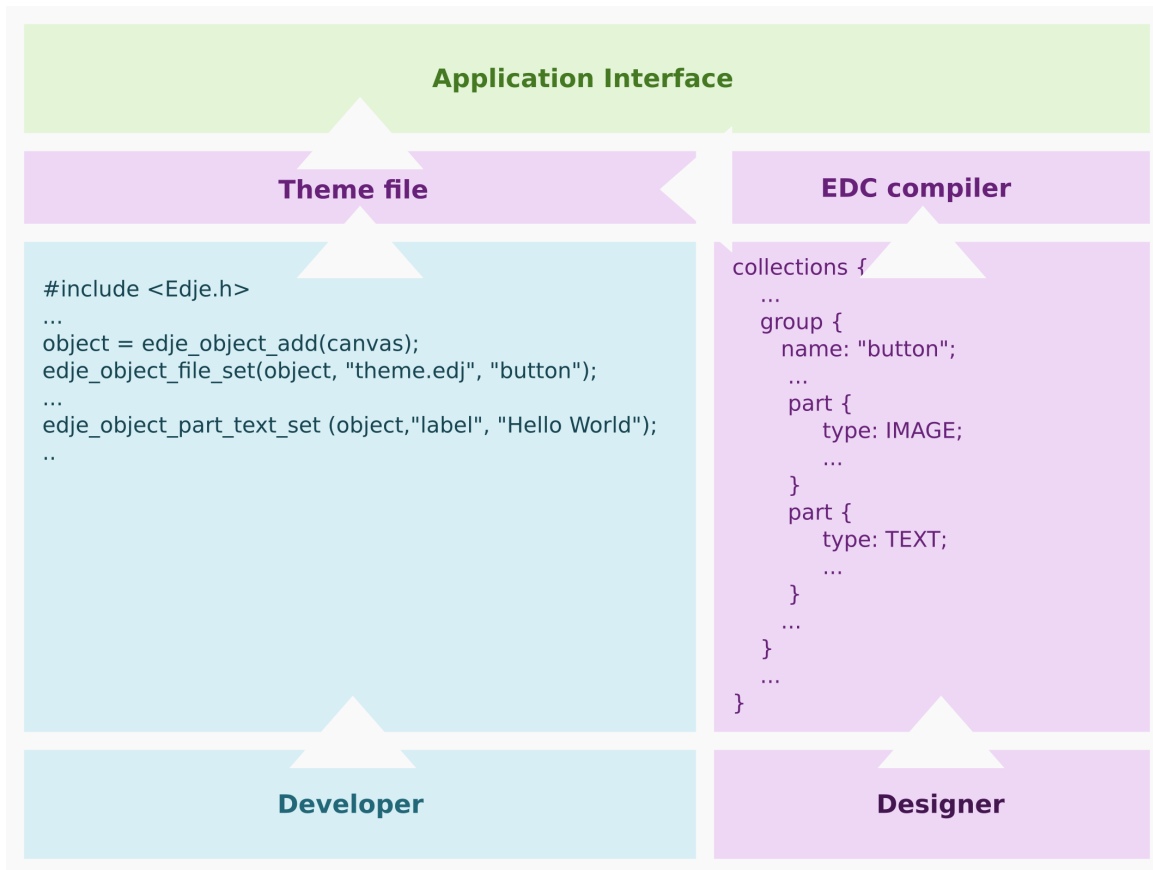
Except for the usage of nested blocks, the sintax of an EDC file is similar to CSS. What really sets them appart is that with EDC the designer it's free to create and layout design elements as he sees fit. With CSS the designer is limited to applying style and layout properties to a structure of objects defined by the markup. With Edje each design object, know as "part", is created by the designer and the final interface object composed by those parts, known as "group", is used by the developer. The resulting theme file can consist of multiple groups representing multiple interface object.

The application developer will find out that the Edje API is small, since the developer is not expected to alter the composition of a group. The API focus on high level manipulation of groups, like forcing a maximum size, but provides some functions to alter the content of a part when it's necessary to transmit information, like altering a paragraph of text with a message.

If we compare Edje to both extremes of GUI development we can see it provides the flexibility of developing your own interface objects from a plain canvas, yet remain almost as simple as including a predefined object from a toolkit. Of course Edje has shortcomings of its own and we will explore them in this book as well.

2.3   The foundations

From a developer's point of view, we cannot expect to understand how Edje works without going through a brief introduction about the Evas first. Luckily for this writer, an excellent introduction to Evas has already been written in the API Reference.

> Evas is a clean display canvas API for several target display systems that can draw anti-aliased text, smooth super and sub-sampled scaled images, alpha-blend objects much and more.
>
> It abstracts any need to know much about what the characteristics of your display system are or what graphics calls are used to draw them and how. It deals on an object level where all you do is create and manipulate objects in a canvas, set their properties, and the rest is done for you.
>
> Evas optimises the rendering pipeline to minimise effort in redrawing changes made to the canvas and so takes this work out of the programmers hand, saving a lot of time and energy.
>
> It's small and lean, designed to work on embedded systems all the way to large and powerful multi-cpu workstations. It can be compiled to only have the features you need for your target platform if you so wish, thus keeping it small and lean. It has several display back-ends, letting it display on several display systems, making it portable for cross-device and cross-platform development.

When using the Evas API directly, the developer uses function calls n the format object_line_add and object_image_add to include the different primitives in our canvas. Each of these primitives would be included in the form of an "Evas object".

But Evas is not limited to simply rendering primitive objects. The most common use for a canvas is to assemble multiple primitives like lines or rectangles into figures like charts or diagrams. In order to maintain coherency among all the primitives the developer was forced to implement functions that abstracted manipulation of the figure from the manipulation of its components. One of these "workarounds" is now known as Evas Smart Objects.

Smart objects are implemented by the developer to create new Evas object types. Functions like add, del, hide, show are implemented using the regular Evas API to affect each primitive. This collection of functions is grouped into a new Evas Smart Class structure that consist of a list of pointers to the functions, the object type name and version. The instances resulting from these classes would be manipulated by the canvas in the same way it does for any other object.

Edje is implemented as a more abstract type of smart object. The list of primitives to manipulate is not hard coded into the functions forming the Smart Class. Edje has

functions that analyze the structure of a compiled theme file and get the list of primitives and their properties from a given "group" inside the file.

In the end Edje interfaces can be seen as an illustration over a canvas. Space they can share with other primitive and smart objects. Without an Evas canvas there is no Edje theme.

2.4 Convenient libraries

The normal process to get a canvas up and running can be bothersome. Evas supports multiple rendering engines, like the software, xrender and opengl flavors of X11 and framebuffer devices. But before any rendering can be done the developer has to complete an Evas_Engine_Info structure with the required information about the target engine. This forces the developer to research the different functions to get that information for each target. Alternatively he can use a shortcut available for most of them.

As you might have realized by at this point, I intend to quote the official API reference at every chance I get. This one comes straight from the "The Ecore Main Loop" page:

> Ecore is a clean and tiny event loop library with many modules to do lots of convenient things for a programmer, to save time and effort.
>
> It's small and lean, designed to work on embedded systems all the way to large and powerful multi-cpu workstations. It serializes all system signals, events etc. into a single event queue, that is easily processed without needing to worry about concurrency. A properly written, event-driven program using this kind of programming doesn't need threads, nor has to worry about concurrency. It turns a program into a state machine, and makes it very robust and easy to follow.

At the beginning of its life, the Ecore library was used as a Event loop and loop management is still one of the modules that composes Ecore.
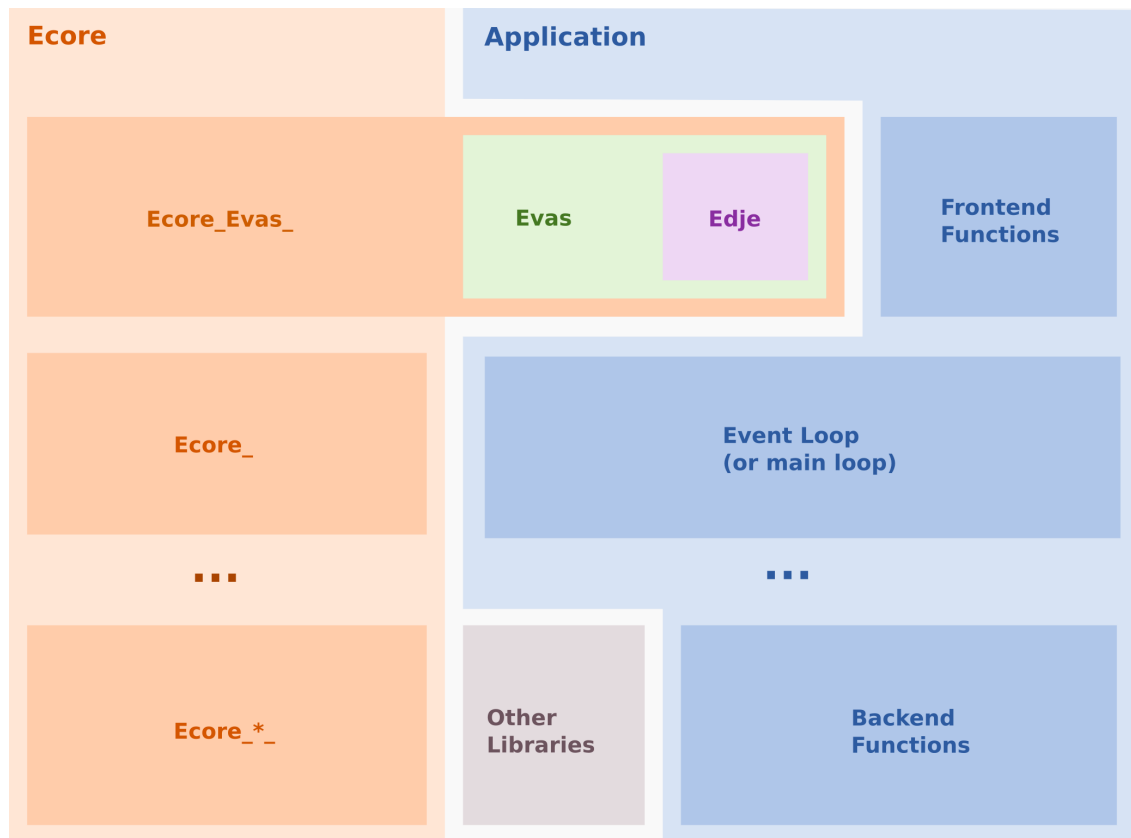
Today, Ecore encompasses a long list of modules properly namespaced and prefixed with "Ecore_". From what we have seen in the previous chapters, there are two modules that jump right out of the list. The first Ecore_Evas and second named Ecore. The first provides convenient functions to setup the Evas canvas and the later provides the loop management functionality described above.

The developer needs an Evas canvas to render the Edje interface and Ecore_Evas is the simplest way to get an Evas canvas up and running. This wrapper is intended to support every backend that Evas supports with its respective (and sometimes unique) attributes. It trivializes initialization to a couple of lines and multiple engine support to an application reload.

Chapter 3

The foundations in practice

By pointing out that the Enlightenment Foundation Libraries are designed in a Object Oriented manner I wish not to raise the wrath of OO purists but to simplify the reader's mental image of how the EFL C Application Programming Interface is structured.

Now that the pitchforks are back in the barn and torchs have been put off, allow me to put it in more clear terms using a simple example:

```
Evas_Object *button = NULL;
button = edje_object_add(evascanvas);
edje_object_file_set(button, "theme.edj", "button");
```

This is a simple C snippet that could be translated into a more (sintactically speaking) OO language like Python as:

```
button = Evas_Object()
button.file_set("theme.edj","button")
```

The differences between the code snippets could be written off as "sintactic sugar" but it serves to demonstrate the structure which the EFL C API follows. If we dissected the last function call we could split it into three groups. First, the class of the object to manipulate, `edje_object`, second, the method to call `file_set` and third the pointer to the object instance (`button`, along with the parameters `"theme.edj","button"`) or generically speaking, `class_method(instance,parameters)`.

The first source snippet in the following tutorial is, of course, the first exception. The following function calls deal with the library itself. Anyway, displaying a simple Edje object in a window is a task with a few well defined steps that begin by initializing the necessary libraries:

```
#include <stdlib.h>
```

```
#include <stdio.h>
#include <Evas.h>
#include <Ecore.h>
#include <Ecore_Evas.h>

int main() {
    if (!ecore_init()) return EXIT_FAILURE;
    if (!ecore_evas_init()) return EXIT_FAILURE;
    ...
```

All initialize-able Enlightenment Foundation Libraries do so in the format `library_name_init()`. Both Ecore and Ecore_Evas need to be initialized before being used and both will return success or failure using standard C values (0 is failure, any other number is success), hence the `if`.

This would be a good moment to note that the assembled, and commented, version of the source code of these examples can be found in CVS repository of the Enlightenment project: `docs/devwithedje/src`

In order to render a canvas its necessary to create a canvas wrapper that will host it and to store a pointer to it for future reference:

```
    ...
    Ecore_Evas  *ecore_evas = NULL;
    ...
    ecore_evas = ecore_evas_software_x11_new(NULL, 0, 0, 0, 800, 600);
    if (!ecore_evas) return EXIT_FAILURE;
    ...
```

While the software_x11 engine is being used in this example, its possible to use any other supported engine by simply changing the second line to `ecore_evas_enginename_new()`. A list of supported engines and their parameters can be found in the Official API reference.

Once the canvas wrapper has been setup we need to change its state to visible:

```
    ...
    ecore_evas_title_set(ecore_evas, "Example Application");
    ecore_evas_name_class_set(ecore_evas, "testapp", "Testapp");
    ecore_evas_show(ecore_evas);
    ...
```

While the first two lines are optional, it is useful to see the way we set the title name and class of the window that will host our canvas. The third function sets the given canvas wrapper to visible, this can be reversed later with `ecore_evas_hide()`.

The canvas wrapper is ready to go, but the actual canvas that will be use to draw our Edje objects is nowhere to be found:

```
...
Evas *evas = NULL;
...
evas = ecore_evas_get(ecore_evas);
...
```

The function ecore_evas_get() returns a pointer to the canvas housed in the canvas wrapper, this is the pointer we need in order to include our Edje objects later.

The execution loop for the program can also be conveniently handled by Ecore:

```
...
ecore_main_loop_begin();
...
```

Once ecore_main_loop_begin(..) has been called, both the canvas wrapper and the canvas itself will be drawn in their current state (a 800x600px empty window in this case). Ecore will continue to loop until an event handled by it occurs.

Once the execution of the main loop has finished it's a good practice to shut down any library we initiated:

```
...
ecore_evas_shutdown();
ecore_shutdown();
}
...
```

3.1 Working with the canvas

Now that we know how to setup the enviroment to display the Edje objects we will review how to include and interact with the objects themselves. The following code is platform independent and can be merged with the example code provided before or with the equivalent for any other platform.

Just like the last time we begin by including the necessary header files and initializing the libraries:

```
...
#include <Edje.h>
...
int main() {
    ...
    if (!edje_init()) return EXIT_FAILURE;
    ...
```

After a pointer to the canvas has been aquired, we need a pointer to a valid Evas object
to insert our Edje object:

```
...
Evas_Object *edje = NULL;
...
edje = edje_object_add(evas);
edje_object_file_set(edje, "testfile.edj", "testgroup");
...
```

Both functions are specific to Edje, in the first case we use edje_object_add to create
a pointer to an Evas object and edje_object_file_set to add the contents represented
by "testgroup" in "testfile.edj".

As with any other Evas object we need to instruct Evas to make it visible, but not before
adjusting the object inside the canvas or since we are only showing one object, adjust
the canvas to the size of our object:

```
...
Evas_Coord width, height;
...
evas_object_move(edje, 0, 0);
edje_object_size_min_get(edje, &width, &height);
evas_object_resize(edje, width, height);
ecore_evas_resize(ecore_evas, width, height);
evas_object_show(edje);
...
```

First we use evas_object_move to move our Edje object to the left-most, up-most
corner of the canvas. The function edje_object_size_min_get, returns the minimal
possible size of the object and evas_object_resize changes the current size of the object
to those values.

Before the end, we resize the canvas to the same values the object has with ecore_evas_resize
and (finally) instruct Evas to show the object.

3.2   Interacting with the objects

To understand how to interact with Edje or any other Evas based object we need to
review the basics of how the main loop manager handles events. Ecore maintains a list
of pointers to functions to be called when a signal of a given type is received. These
functions are known as signal "handlers".

By default, Ecore awareness is limited to system singals like HUP or KILL. Additional libraries or modules like Ecore_Evas register new signal types for the event loop to be aware of. In the specific case of Evas the new signal types deal with the interaction between the user and the Evas objects displayed in the canvas.

The developer can manipulate the list of handlers as well as creating new signal types. The latter among other subjects like timers and pollers exceed the scope of this book and are properly documented by the API reference and the EFL Cookbook.

We will begin by setting up a simple signal handler that will be called any time the application is closed:

```
Ecore_Event_Handler* close = NULL;
...
int
good_bye(void *data, int type, void *event)
{
    //Removing handler for no reason other than API showoff
    if (ecore_event_handler_del(close))
        printf("Handler deleted\n");

    printf("Good bye! \n");
    ecore_main_loop_quit();
    ecore_evas_shutdown();
    ecore_shutdown();
    edje_shutdown();
}
...
int main() {
...
    close = ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT,
                                    good_bye,"data");
    ...
    ecore_main_loop_begin();
    ...
```

This example moves the library shutdown procedure from the main function to the "good_bye" function. Then before the main loop is initiated we add the handler for this signal type. The last parameter it's a pointer to any kind of data you want to pass to the handler function, in this case is just a string containing "data".

Interaction with the interface works in a similar way. The Edje library registers its own handler function in the Ecore loop. This handler will be called for every signal coming from the interface. Therefore, in order to react to interface events we need to register our functions as "callbacks" in this handler's list:

```
    ...
    void
    colorize(void *data, Evas_Object *o, const char *emission,
             const char *source)
    {
        Evas_Coord x,y;
        evas_pointer_canvas_xy_get(evas,&x,&y);
        if(x > 255) x = 255;
        if(y > 255) y = 255;                    //R,   G,   B,   A,
        edje_color_class_set("main color", 190, x,   y,   255,
                                                255, 255, 255, 255,
                                                255, 255, 255, 255);
    }
    ...
    int main() {
        ...
        edje_object_signal_callback_add(edje, "mouse,move", "*",
                                        colorize,"data");
        ...
        ecore_main_loop_begin();
        ...
```

The resulting application changes the color of every part using the "main color" color class every time the mouse moves inside the interface. If you use the EDC theme included in the assembled example the result will be a small rectangle in the middle of the canvas that changes of color as we move our mouse around.

The function that produces this effect is our handler, or callback, `colorize`. We call functions from the Evas API `evas_pointer_canvas_xy_get` and the Edje API `edje_color_class_set`. The first call gets the current coordinates of the mouse pointer position. The second call uses those values to alter the color class "main color".

Before the main loop begins we use `edje_object_signal_callback_add` to register our callback. The real handler keeps its own list of functions to call. This list's index is a combination of the signal identification and the name of the source that emitted it. In this particular case, the function `colorize` will be called when the signal "mouse,move" is emitted by any object in the interface. Edje string matching supports wildcards for both the name and source of the signal.

3.3   Building a framework

Hopefully the reader has not been introduced to the concept of software frameworks by web based monsters like Ruby on Rails. Personally I think RoR is great and my point is

that a framework does not necessarily mean large software libraries. A framework can be seen as library of functions determined by the similiarity in the profile of the applications that use it. In a framework, shorter development time means either more specific profiles or more complex library code.

In this section and the following subsections we will develop an application framework for applications with a specific profile: "A X11 desktop application that doesn't require any exotic manipulation of its theme or configuration files". A list of the tasks the framework must perform follow:

- **Configuration (Using Ecore_Config).** – Initialization and shutdown of the necesary services.
    - Saving configuration changes on exit.
    - Recall the previously saved values on initialzation.
    - Control that the necessary configuration and theme files exist.

- **Interface Management** – Create windows with their properties lifted from a given Edje object.
    - Allocate, load and display Edje objects from a "current" theme file, set by the user or a "default" theme file setup by the application developer.

The example application using this framework will be an Edje group viewer, this application will receive a filename and a group name, it will display them inside a viewport managed with a minimap. The last signal emitted by the object is going to be printed to a given text string. Without further ado, here it is, the main file:

```c
#include "../lib/framework.c"
#include "../lib/viewport.c"
#include "../lib/minimap.c"
#include <string.h>
#include <limits.h>

int arguments_parse(char path[], char group[], int argc, char **argv);

int main(int argc, char **argv) {

    Ecore_Evas  *mainWindow = NULL;
    Evas        *mainCanvas = NULL;
    Evas_Coord   width, height;
    Evas_Object *mainLayout, *toView, *viewport, *minimap;
    char         path[PATH_MAX],group[100];

    application_name_set("Plain Edje Viewer");
```

```
    if (!simpler_init())
        return EXIT_FAILURE;

    if(!arguments_parse(path, group, argc, argv))
        return EXIT_FAILURE;

    mainWindow = simpler_window_new("window/main",NULL);
    if (!mainWindow)
        return EXIT_FAILURE;
    mainCanvas = ecore_evas_get(mainWindow);
    mainLayout = ecore_evas_data_get(mainWindow,"layout");
```

The precedeing code snippet shows two of the three shortcuts our framework provides. The first function `simpler_init` is used to group all the library intialization functions along with the configuration initialzation functions. Following the library initialzation comes `simpler_window_new` which is used to create a window based on values provided by a given Edje group. The group itself is saved inside the Ecore_Evas canvas wrapper as a data pointer and can be retrieved with `ecore_evas_data_get`. As you might notice we use of a function named `arguments_parse` to get the path of the requested file and group from the array of arguments.

Notice that the call to `simpler_window_new` does not have any filename attached to it. But how does the application know where to look this group up? This is the work of Ecore_Config and we will review how is it used in the next section of the book. In any case, the following code snippet show a little more Edje related action:

```
    toView = edje_object_add(mainCanvas);
    edje_object_file_set(toView, path, group);
    edje_object_size_min_calc(toView, &width, &height);
    if(width <= 0)
        ecore_evas_geometry_get(mainWindow, NULL, NULL, &width, NULL);
    if(height <= 0)
        ecore_evas_geometry_get(mainWindow, NULL, NULL, NULL, &height);
    evas_object_resize(toView, width, height);
    evas_object_move(toView,0,0);
    evas_object_show(toView);

    viewport = viewport_add(mainCanvas);
    viewport_theme_set(viewport, simpler_object_add(mainCanvas, "widget.mi
    viewport_target_set(viewport,toView);
    edje_object_part_swallow(mainLayout,"swallow.viewport", viewport);

    minimap = minimap_add(mainCanvas);
```

```
    minimap_theme_set(minimap, simpler_object_add(mainCanvas, "widget.minimap"));
    edje_object_part_swallow(mainLayout,"swallow.minimap", minimap);
    minimap_viewport_set(minimap,viewport);

    ecore_main_loop_begin();
}
```

You might notice the code snippet is split in three bigger sections. In these sections we include an Edje object to view, we setup the viewport for the object and the minimap for the viewport respectively. Since the application is going to display a group of an arbitrary name from an arbitrary file we don't use the shortcut functions to load and display the target. But among these know functions `ecore_evas_geometry_get` stands out. This function is used to load the current size of the window as the object width or height in case it does not have either one.

The following two code sections setup each widget, we won't review the internals of these yet. We will explore the function `simpler_object_add` that is used to load the theme groups for these widgets. In order to mantain our santity (and to save some trees) some additional functions or pieces have been left out of the book. The complete, commented sources for this framework and the "Plain Edje Viewer" application can be found in the CVS repository of the Enlightenment proyect under `e17/docs/devwithedje/src`. Rest assured the omitted functions wont add much to your knowledege about Edje.

3.3.1 Simpler library intialization

We have seen initialization functions previously in this chapter and other that an increased level of verbosity there is nothing special about the use we have give them here:

```
Evas_Bool
simpler_init()
{
    char *path;

    if (ApplicationName == NULL)
    {
        fprintf(stderr, "Error: The application's name was not set.\n");
        return FALSE;
    }

    if (!evas_init())
    {
        fprintf(stderr, "Error: Evas failed to initialize.\n");
```

```
            return FALSE;
        }
        ...
```

The rest of the initialization functions can be found in the CVS repository, I decided to show this particular piece because it shows a characteristic of the framework that can become a problem. Having an application's name setup is a requeriment of Ecore_Config. It uses that name to search for or create a configuration directory and file. Unless it was explicitly altered by the developer the default location of the configuration file will be $HOME/.e/apps/ApplicationName/config.eet. Once the application's name has been set, we are safe to work with Ecore_Config:

```
        ...
        ecore_config_theme_default("theme/default", "default");
        ecore_config_theme_default("theme/current", "default");
        if (ecore_config_load() != ECORE_CONFIG_ERR_SUCC)
        {
            fprintf(stderr,"Warning: Could not load config from ~/.e/apps/%s/c
        }
        path = ecore_config_theme_with_path_get("theme/default");
        if (path == NULL)
        {
            fprintf(stderr,"Error: The default theme, '%s.edj', was not found
            return FALSE;
        }
        return TRUE;
    }
```

An excellent introduction to Ecore_Config already exists in the EFL Cookbook and I won't replicate here. But I will resume the functions used in this code sample. The first function called is `ecore_config_theme_default` and it's used to setup the default value for a given theme configuration item, don't worry, the specifics of "theme configuration items" are explained further down this book. In any case, setting up a default value is useful to ensure a valid value in case the configuration file is non-existent or incomplete.

Right after setting some default values we use `ecore_config_load` to load the configuration file found in the default location mentioned earlier. It is a good practice to not use custom file locations for configuration files because it eases portability of the application. The user will also apreciate the additional feeling of certainlity of how an application is going to behave. Still, the alternative function `ecore_config_file_load` could have been used to load a configuration file from a custom location.

Before finishing sucessfully we will have to control that at least the default theme file can be found somewhere. Finding at least one theme file is pretty much mandatory for any Edje based application.

3.3.2 Simpler window setup

Setting up a window isn't complex by itself, but why would we manually setup these properties, copy and paste these big blocks of code and manually replace the parameters for them to adjust to our new application when we can use an automated method that also gives more freedom to the designer?

In this example all the properties for the window including the maximum and minimum sizes, name and class are setup by a "group" in an Edje theme file. It's up to the developer's personal taste to decide how much control a designer will have, thus altering the behaviour of this function was left as an excercice for the reader:

```
Ecore_Evas*
simpler_window_new(const char *groupName, Ecore_Evas *parent)
{
    Ecore_Evas        *window;
    Evas              *canvas;
    Evas_Object       *layout;
    Window_Properties *prop;
    Evas_Coord        width, height;

    if (parent == NULL)
    {
        window = ecore_evas_software_x11_new(NULL, 0, 0, 0, 0, 0);
        ecore_evas_data_set(window, "is_main", "yes");
    }
    else
    {
        window = ecore_evas_software_x11_new(NULL, ecore_evas_software_x11_window_
        ecore_evas_data_set(window, "is_main", "no");
    }

    if (window == NULL)
    {
        fprintf(stderr, "Error: Couldn't initiate the canvas wrapper.\n");
        return NULL;
    }

    canvas = ecore_evas_get(window);
    if (canvas == NULL)
    {
        fprintf(stderr,"Error: There is no canvas in the window.\n");
        return NULL;
```

```
    }

    layout = simpler_object_add(canvas, groupName);
    if (layout == NULL)
    {
        fprintf(stderr, "Error: Couldn't load the layout object.\n");
        return NULL;
    }
    ecore_evas_data_set(window,"layout",layout);
    ...
```

By now you might be wondering if there is a real reason to abstract functions as simple and wildly known such as these? There is no simple answer other than that these function set properties that could cause unexpected behaviour later in the application. I spent an hour of my life trying to figure out why the size calculation of the canvas didn't reflect the size changes that ocurred in this very initalization function.

Now that I have justified the first part of the function we might as well analyze it! Most of these function have been seen before except for two. The first function is `simpler_object_add`, it is a part of the framework that acts as a wrapper for `edje_object_add` and `edje_file_set`. The specifics of this function are going to be explained in the next chapters. The second function is `ecore_evas_data_set`, it is a standard Ecore_Evas function and it's used to attach pointers to arbitrary data to a given key, in this case we save the layout Edje object in the "layout" data pointer. The next functions will probably be altered by most application developers, and it can be split in three big sections of code:

```
    ...
    prop = _window_prop_get(layout);
    ecore_evas_title_set(window, prop->title);
    ecore_evas_name_class_set(window, prop->name, prop->class);
    ecore_evas_size_min_set(window, prop->minw, prop->minh);
    ecore_evas_size_max_set(window, prop->maxw, prop->maxh);

    evas_object_move(layout, 0, 0);
    edje_object_size_min_get(layout, &width, &height);
    evas_object_resize(layout, width, height);
    ecore_evas_resize(window,  width, height);
    evas_output_size_set(ecore_evas_get(window), width, height);

    ecore_evas_callback_resize_set(window, _resize_window);
    ecore_evas_callback_delete_request_set(window, _close_window);
    ecore_evas_callback_destroy_set(window, _close_window);
```

```
        ecore_event_handler_add(ECORE_EVENT_SIGNAL_EXIT, _close_application, window);

        ecore_evas_show(window);
        return window;
    }
```

We can begin our analysis with the first section of code. There we setup the various properties of the window, including titile, name, class, maximum and minimum size. Some of these properties could have been setup before with `ecore_evas_software_x11_new` if we wanted them to be hard coded in the framework. In this case we needed to load an Edje object to memory and a canvas is needed to initialize the objects.

The second section of code looks quite familiar since it is basically a carbon copy of code written in "The foundations at practice", this section is used to synchronize the sizes of the window and the Edje group used as a layout object.

In the last section we set a couple of callbacks. The first callback resizes the layout object along with the window, thus all the inserted objects that are relative to the layout are going to be resized when the window is resized. The rest of the callbacks are used to shutdown the application in different situations. The function, `_close_window` is quite simple, and will calle `_close_application` when the user closes the main window. A main window is any window without a parent, the framework assumes you will only create one parentless window per application.

3.3.3   Simpler theme management

In the previous section we have seen how to create a window based on parameters lifted from a given Edje object refered to as the layout object. If you take a look you will see that the layout object was aquired differently than the common Edje object, the function is names `simple_object_add` and its used in the main function to load the themes for the viewport widget and the minimap widget as well.

In this chapter we will review all the functions called to make this simpler method of aquiring Edje objects possible:

```
    Evas_Object *
    simpler_object_add(Evas *canvas, const char *group)
    {
        Evas_Object *o;
        o = edje_object_add(canvas);
        if(simpler_object_file_set(o, ecore_config_theme_with_path_get("theme/current")
        {
            return o;
```

```
    }
    else
    {
        if(simpler_object_file_set(o, ecore_config_theme_with_path_get("the
        {
            return o;
        }
    }
    return NULL;
}
```

We begin by the most popular `simpler_object_add`. All this function really does is to allocate and initate the object in the choosen canvas with `edje_object_add`. The rest of the work is done by `simpler_object_file_set` and `ecore_config_theme_with_path_get`.

As I have said earlier we are going to use Ecore_Config to manage the application's settings. An introduction to Ecore_Config can be found in the "EFL Cookbook" downloaded for free in the Enlightenment project website. This particular setting is used to manage a theme file name. Altought this sounds like basically managing a string the function"ecore_config_theme_with_path_get" show us how it actually work, better expressed in pseudocode:

```
ecore_config_theme_with_path_get(key):
    theme_file_name = search_config_file_for(key)
    possible_paths = ecore_config_theme_search_path_get()
    for each path in possible_paths:
        if theme_file_name in path:
            return path + theme_file_name
        else
            continue
```

What this means is that a `ecore_config_theme_*` property is quite more advanced than a simple path in the form of a string. Now that we have that cleared up lets hit the next function in the calling stack:

```
Evas_Bool
simpler_object_file_set(Evas_Object *o, const char *path, const char *grou
{
    if(path != NULL)
    {
        if(!ecore_file_exists(path))
        {
            fprintf(stderr, "Warning: Failed to find the theme file '%s'.\r
```

```
                return FALSE;
        }
        else
        {
            if(!edje_file_group_exists(path, group))
            {
                fprintf(stderr, "Warning: Failed to find group '%s' in theme file
                return FALSE;
            }
            else
            {
                edje_object_file_set(o, path, group);
                evas_object_show(o);
                return TRUE;
            }
        }
    }
    else
        return FALSE;
}
```

This function is also quite simple. It basically calls *edje_object_file_set* after running
some checks to see if the parameters are valid. After that, it sets the object to be
displayed with `evas_object_show` mainly because the default value hides the object and
it will not be used as often by the callers of this function.

These, and a couple of more functions are used by every application fitting the profile
of this framework. I wrote this code expecting you will want to modify it, learning a lot
in the process, so I commented and structured the full source in Enlightenment's CVS
server exactly for this prupose.

Chapter 4

Introduction to widgets

Graphical User Interfaces do not only to display information, they convey information. Interface elements have a meaning of their own and this meaning alters the user's perception on the information displayed, for better or for worse. A flexible interface design system means the designer can add more meaning to the information. Features like multiple states and transitions extend this capacity to the point where the designer's creativity is the limit.

As the application matures the number of elments in the interface will grow. These elements will be grouped by some common property or prupose. Functions to deal with these groups as a unit are also going to be created. This is not an unique process and anyone creating a GUI from scratch would go through it. These groups and the functions that deal with them is what is commonly know as "widgets". Widgets work as a small application inside our own.

In order to work a widget needs access to the drawing device to request that the primitive objects that conform it be drawn. Most widgets will want to interact with their environment, thus they need to be familiar with the Event loop managing the application. As we would do with a plain interface, the widget will register callbacks for its own functions that deal with events in the primitive objects that conform it.
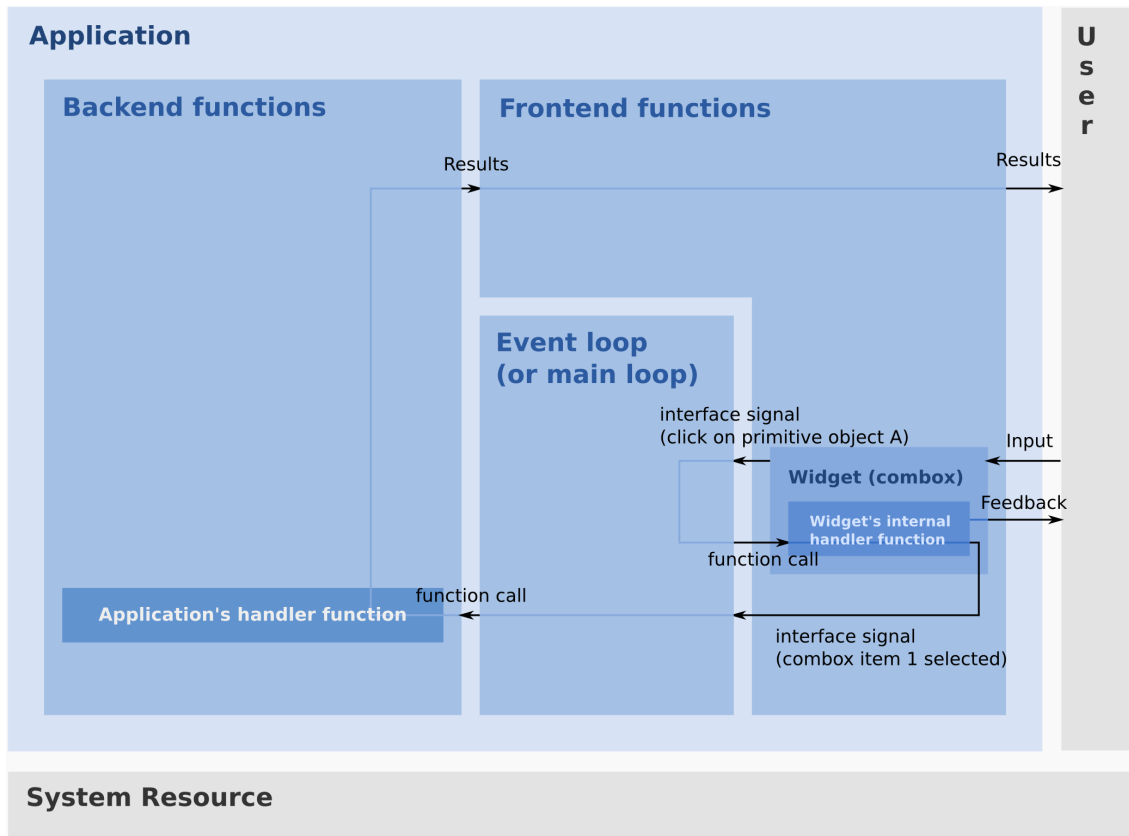
In order to be possible for the rest of the application to interact effectively with a widget it has to provide the Event loop with new signal types representing abstracted version of its interface signals. It basically intercepts what would be otherwise normal interface signals and (sometimes) replace them with its own abstracted versions.

In general terms, a developer implementing a widget from the scratch would have to program functions to:

- Manipulate the canvas primitives to draw the widget.

- Register new signal types to emit as a widget.

- Display information and retreive changes.

- Handle events ocurring in the environment that affect it.

- Handle events ocurring on its primitives.

Depending on the complexity of the widgets, the amount of functions represented by the last item can become quite large.

4.1 Widgets with Edje

We have seen there are roughly 5 kinds of functions that have to be implemented in order to create a widget from scratch. Although the way groups are split might not seem to correspond with the amount of work each one might entail, the divisions concord with the shortcuts Edje provides. For the cases where the number of functions in one group seems to clearly outweighs the others, Edje provides additional shortcuts not directly related to the design concepts and design elements we have seen in the previous chapters.

I will begin with **Manipulate the canvas primitives to draw the widget** since it's the easiest to explain. With a plain canvas approach to widget creation, this would mean the developer must manually create each rect, line, etc. that form its widget. Instead, Edje draws the primitive objects based on the designer's specifications in the Edje Data Collection file, not a single primitive has to be created or manipulated by the application developer.

Addressing the task **Handle events occurring on its primitives** using Edje can be done with two different, but compatible, approaches. The first approach is also available in the other low level libraries. To intercept all the signals coming from the primitive objects that form the widget, filter the signals and (sometimes) emit a new kind of signal. This approach is useful in many situations and that's why it is possible to use it in Edje even when it provides an alternative, not available among its low level cousins. With Edje designer can write small "programs" in his EDC files. An Edje program is not as complex as the name might make us think. Unless the designer uses a script block (rare) the average Edje program would look like this:

```
program {
    name:       "playthemusic";
    signal:     "mouse,down,1";
    source:     "*";
    action:      action: SIGNAL_EMIT "PLAY" "button";
}
```

There is more to Edje programs than just this, but this example will be enough to illustrate the point. Since Edje programs are limited to their own group, this program is limited to a hypothetical "button" group. What the program does is to emit the signal "PLAY" when the left button is pressed over any part that forms the "button" group. The application only has to listen for the PLAY signal. When and why that signal will be emitted is completely up to the designer to decide. He could change "mouse,down,1" to "mouse,wheel,*" and cause the music to play when the user uses the mouse wheel over any part of the button.

The previously illustrated program shows that **Register new signal types to emit as a widget** will not be as common as it would with another canvas library. Most of the time,

it will be designer who implements new signal types. Sadly, there is little that Edje can provide concerning **Handle events occurring in the environment that affect it**.

To **Display information and retrieve changes** is quite simple. As with any other low level library, the developer will have to alter some primitive in order to display the requested information. Most of the time this means changing a string of characters, complex diagrams and similar objects go beyond the scope of Edje and enter the realm of Evas programming. Plain Evas objects, of course, integrate perfectly with Edje interfaces.

A difference between a regular canvas lib and Edje concerns conveying subtle information by changing the "state" of the interface. Designers can implement any arbitrary number of states for any part or group and change them using simple Edje programs. The same programs can also be triggered by the application. For example, to disable a button in a plain canvas the developer would have to call functions that gray out the background and text. With Edje the developer just emits the signal "DISABLED" to the button and lets the designer implement this however he wants.

4.2   The Ewl as a shortcut

Creating custom widgets even with the facilities Edje provides can become a very complex task with more advanced widgets. Thankfully for us a working widget library for those specially-hard widgets already exists. Altought the goal of the Enlightened Widget Library are not exactly in synch with the goals of plain Edje application development, hard work has been done by the EWL developers to make including their widgets in a plain Edje interface possible.

But before we go into further implementation details it might be a good idea to quote the introduction to the Ewl API reference:

> The Enlightened Widget Library (Ewl) is a high level toolkit providing all of the widgets you'll need to create your application. Ewl is built on the Enlightenment Foundataion Libraries and makes heavy use of the provided technologies. The goal of Ewl is to abstract the use of these backends and to present an easy to use object model to the end programmer. The expansive object oriented style API provides tools to easily extend widgets and containers for new situations.
>
> Overall, Ewl is similar in design and functionality to other common toolkits such as GTK+ and QT. The APIs differ, but the overall concepts and ideas are similar. If you are familiar with these other toolkits getting into Ewl should be relatively simple.

Simply put, the goal of the Ewl developers is to abstract Edje as on of many rendering backends. This is not a bad thing on its own, but for developers wanting to use Edje

capabilities to the max the Ewl can become limiting. Great care is taken by the Ewl developers to not limit theme designers and this has resulted in a library that does *not* limit the themer more often than it does.

In any case, the Enlightened Widget Library provides an special type of container know as the "embed" container. Widgets included in this container can be swallowed into a SWALLOW part in a common Edje interface. Once the widget has been included its appereance can be altered in a per-theme basis in order for the widget to fit the rest of the application as cleanly as possible.

You might not want to use Ewl to create simple interface objects like buttons and check boxes. But it can become an invaluable time saver for complex widgets like trees and file management dialogs. A great resource to learn how to include Ewl widgets into an Edje interface is the source code ewl_embed_test.c which is logically divided and toroughly commented.

4.3   Implementing widgets with Edje

4.3.1   Introduction to Smart Objects

4.3.2   The foundations of a Smart Object

4.4   Laying out widgets

4.5   Creating list based widgets

Chapter 5

Widgets by Example

5.1 The text entry

5.2 The combox

5.3 The kinetic list

5.4 The media viewport

5.5 Flexible toolbars

This will be at the top of every page.