Interfaces gráficas de usuario con Elixir

Graphical user interfaces with Elixir

Ignacio Nicolás López

Director: Manuel Montenegro Montes

Facultad de Informática

UNIVERSIDAD COMPLUTENSE

Trabajo de Fin de Grado en Ingeniería Informática

Curso 2019 - 2020

Resumen **5**

Palabras clave **5**

Abstract **6**

Keywords **6**

I Introducción **7**
Antecedentes 7
¿Qué es Elixir? 7
Motivación 7
Objetivos 8
Plan de trabajo 8

II Introduction **11**
Background 11
What is Elixir? 11
Motivation 11
Objectives 12
Work plan 12

III Herramientas y tecnologías **15**
Erlang 15
Elixir 15
wxWidgets 15
Elm 15
WxHelper 16
Git 16
Visual Studio Code 16
Google Drive 17
Documentos de Google 17
Google Meet 17

IV Arquitectura básica **18**

V Arquitectura general de la librería **25**
Los árboles de diferencia 25
Módulo Builder 281
Módulo Comparator 31
Módulo GUIUpdater 34
Módulo Elixir_elm 35

VI Conclusiones y trabajo futuro **37**
Conclusiones y objetivos 37
Trabajo futuro 38

VII Conclusions and future work **39**
Conclusions and objectives 39
Future work 40

Bibliografía **42**
Libros: 42
Páginas web: 42

Resumen

En este Trabajo de Fin de Grado se desarrolla ElixirElm, una biblioteca de desarrollo de interfaces gráficas de usuario para el lenguaje Elixir.

El desarrollo de interfaces gráficas en Elixir se realiza de una forma demasiado alejada de un paradigma funcional, por lo que resulta lógico buscar alguna alternativa que elimine esta falta de naturalidad en el desarrollo. Así pues, ElixirElm tiene como propósito otorgar a los programadores de aplicaciones gráficas en Elixir una capa de abstracción que solucione esta problemática. El objetivo de esta capa de abstracción es mantener el paradigma funcional de programación que tiene Elixir, pero que la librería WxErlang, utilizada para el desarrollo de interfaces en Elixir, contradice forzando su uso mediante un paradigma imperativo “orientado a objetos”.

El nombre de la biblioteca es fruto de juntar el nombre del lenguaje Elixir, lenguaje en el que está programada y lenguaje objetivo de la librería, y Elm, otro lenguaje funcional de programación de aplicaciones web, cuya arquitectura es el objetivo de la capa de abstracción que otorga ElixirElm.

ElixirElm permitirá a los programadores que hagan uso de ella programar interfaces gráficas de propósito general con un conjunto básico de componentes visuales, manteniendo la coherencia con el resto de su programa. Nuestra biblioteca para la construcción de interfaces de usuario hará uso internamente de la biblioteca WxErlang, pero el programador tan solo tendrá que interactuar con ElixirElm.

Palabras clave

Elixir, WxWidgets, WxErlang, Elm, GUI, biblioteca, librería, interfaz gráfica, lenguaje funcional.

Abstract

In this Final Degree Project, ElixirElm, a library for developing graphical user interfaces, is developed.

The development of graphical interfaces in Elixir is done in a way that does not fit very well in a functional language, seeming only logic to try and find any alternative that bypasses that lack of naturalness in the development. Said that, ElixirElm has as purpose to grant the programmers of graphical application in Elixir an abstraction layer that gives a solution to this problematic. The goal of this abstraction layer is to maintain the functional programming paradigm of Elixir, that the library WxErlang, used for the building of the interfaces in Elixir, contradicts by forcing its imperative “object oriented” paradigm upon the programmers.

The name of the library comes from joining the name of the programming language Elixir: language in which the library is programmed and the target of the library as well, and the name of the programming language of Elm: another functional language for web applications, whose syntax is the one to achieve by the abstraction layer of ElixirElm.

ElixirElm will allow its programmers to program graphical user interfaces of general purpose with a basic collection of visual components, maintaining the coherence with the rest of the programmers’ code. Behind the scenes, the library will use WxErlang for the construction of the interfaces, while the programmer will only have to interact with the functions of ElixirElm.

Keywords

Elixir, WxWidgets, WxErlang, Elm, GUI, library, graphical user interface, functional language.

I Introducción

El objetivo de este Trabajo de Fin de Grado es la creación de una librería que facilite el manejo de interfaces gráficas de usuario en el lenguaje Elixir. Esta librería estará basada en la arquitectura utilizada en el lenguaje Elm.

A lo largo de este capítulo se desarrollará la explicación de los antecedentes que motivaron este trabajo, así como los objetivos que se fueron planteando para este durante el desarrollo del mismo. Además, se incluye una descripción en detalle del plan de trabajo en cada fase del desarrollo.

Antecedentes

¿Qué es Elixir?

Para entender las necesidades que motivaron la realización de este proyecto es importante explicar este lenguaje de programación ya que el trabajo que está orientado a sus necesidades.

Elixir es un lenguaje de programación funcional orientado a procesos y que está construido sobre otro lenguaje de características similares: el lenguaje Erlang. Entre sus principales características está el soporte para metaprogramación, mediante el cual se permite ampliar la sintaxis del propio lenguaje con el fin de incorporar construcciones específicas de un dominio.

Motivación

Pese a todo el potencial de Elixir, este lenguaje tiene ciertas carencias a la hora de construir interfaces gráficas de usuario (GUIs). Para llevar a cabo esta tarea, Elixir actualmente utiliza una biblioteca heredada del lenguaje sobre el que está construido, la biblioteca WxErlang. Además, esta biblioteca está adaptada a su vez desde la original WxWidgets de C++. Esto implica que, al estar WxWidgets basado en un paradigma imperativo orientado a objetos, la creación de GUIs en Elixir utilizando WxErlang se realiza de un modo poco natural.

Como contraposición, Elm es un lenguaje de programación funcional orientado a la programación de GUIs para navegadores web. Este lenguaje implementa una sencilla forma de programar GUIs, pero carece de otras características clave de Elixir. Entre las principales desventajas de Elm está su ámbito, que se limita principalmente a las aplicaciones web.

Así pues, en este trabajo se pretende integrar las facilidades que otorga la arquitectura de Elm a la hora de desarrollar las GUIs al lenguaje de Elixir.

Objetivos

Como conclusión a los antecedentes, el objetivo principal sobre el que se basa este TFG es el de desarrollar una librería que otorgue una capa de abstracción sobre WxErlang que proporcione al programador un modelo más adecuado para el desarrollo de GUIs en Elixir.

A raíz de este objetivo principal se han planteado una pequeña lista con otros objetivos que detallan el desarrollo de la librería. Estos objetivos son:

- **Aproximar la sintaxis resultante de la capa de abstracción al lenguaje Elm**, ya que esta es bastante simple y fácil de utilizar. En este modelo, el programador deberá de especificar el modelo de la interfaz y programar funciones para la inicializar o modificar un VDOM.
- **Otorgar soporte para los elementos básicos de una interfaz gráfica de usuario**. Así pues, se implementará soporte para los elementos que se han considerado básicos en cualquier interfaz de usuario. Estos elementos son: cuadros de texto (static texts en WxErlang), cuadros de entrada de texto (text controls), contenedores de elementos (sizers) y listas de elementos (list boxes).
- **Añadir la posibilidad de creación y uso de diálogos modales**, permitiendo al programador la capacidad de crear y mostrar estos diálogos a partir de una ventana o de otro diálogo con el fin de profundizar en la capacidad de construcción de GUIs de la librería.

Plan de trabajo

Durante el desarrollo del TFG, el trabajo a realizar se organizó en fases de 15 días a través de reuniones entre el alumno y el profesor director. Cabe mencionar que durante los periodos de exámenes hubo excepciones con la regularidad de las reuniones.

En estas reuniones se establecían metas a alcanzar pasados 15 días y se revisaban las establecidas en la anterior reunión. Dependiendo del estado de esas metas, se decidía si se debía profundizar en estas o apuntar a otras.

Estas metas se pueden englobar en los siguientes bloques de trabajo:

1. Familiarización con las tecnologías

Pese a tener experiencia con otros lenguajes funcionales, era necesario dedicar las primeras tres fases para familiarizarse con todas las tecnologías que entrarían en juego en los próximos bloques de trabajo. Este bloque está descrito en el capítulo III.

Durante este bloque, los esfuerzos se centraron principalmente en la entrada en contacto con Elixir y con la construcción de GUIs con WxErlang. Esto se consiguió, primero, a través de la realización de pequeños ejercicios para comprender la sintaxis general del lenguaje, para luego realizar ejemplos sencillos de interfaces de usuario con WxErlang, donde se entró en contacto con los detalles más

específicos de la librería, como las constantes y clases con las que se trabajará en los próximos bloques.

2. Codificación de los elementos de interfaz en árboles de diferencia

En este bloque se reúnen las fases que se centraron en crear una estructura de datos recursiva que describiese una interfaz de usuario. Esta estructura debía de ser capaz de recoger los atributos que son necesarios para la instanciación de un elemento de la interfaz con WxErlang, y así representar en detalle las características de cualquier elemento visual.

A lo largo de este bloque y dependiendo de las metas de cada fase, los campos de esta estructura de datos fueron cambiando y adaptándose a las necesidades de la fase, hasta finalmente asentarse en los árboles de diferencia definitivos con los que cuenta ElixirElm (ver capítulo V, apartado de Los árboles de diferencia).

3. Construcción de una GUI a través de un árbol de diferencia

Tras haber codificado con éxito los elementos de la interfaz en árboles de diferencia se requería poder construir y mostrar por pantalla esta interfaz con la información que otorgaba el árbol de diferencia. Este bloque contiene tan solo una fase de quince días, ya que una vez que se contaba con una estructura de datos para la representación de GUIs, tan solo fue necesario programar un módulo que explorase la estructura e instanciase los elementos. Este módulo está desarrollado en más detalle en el capítulo V, en la descripción del Módulo Builder.

4. Desarrollo de un módulo para hallar diferencias entre interfaces

Este bloque de trabajo también se apoya en el segundo bloque, resultando en un bloque que se extendió en tan solo una fase de quince días. Con la implementación de este módulo fue necesaria una modificación en la estructura de los árboles de diferencia, para que pudiesen representar de una forma más cómoda la diferencia entre dos interfaces.

Es importante mencionar que esta es una primera versión del módulo de comparación, y a medida que avance el desarrollo de la interfaz será actualizada. La descripción en profundidad de este módulo se puede encontrar en el capítulo V, en el apartado referente al Módulo Comparator.

5. Desarrollo de un módulo para aplicar cambios a interfaces

Era necesario un módulo que a raíz de las diferencias halladas gracias al del bloque anterior, actualizase la interfaz que se estaba mostrando por pantalla. El desarrollo del módulo en cuestión ocupó dos fases de quince días cada una, ya que la dificultad del módulo y la poca documentación con la que cuenta WxErlang ralentizaron dicho desarrollo.

Tras explorar las posibilidades dentro de las actualizaciones que se pudieran llegar a necesitar fue necesario realizar una revisión de las capacidades descriptivas de

las diferencias que se encontrasen entre interfaces. Así pues, se decidió actualizar tanto la estructura de árboles de diferencia como el módulo de diferenciación.

Se puede encontrar una especificación más concreta consultando el apartado del Módulo GUIUpdater, dentro del capítulo V.

6. Desarrollo del módulo de manejo de eventos y GUI

Una vez completados los anteriores módulos, quedaba por implementar un módulo que facilitase el uso de los eventos por parte del programador. El desarrollo de este módulo demostró ser complicado, extendiéndose a lo largo de treinta días y conllevando múltiples modificaciones tanto en todos los módulos anteriores como en la estructura de datos de los árboles de diferencia.

Una vez que este módulo estuvo finalizado, la librería ya podía ser utilizada, pero no contaba con la sintaxis objetivo. Este módulo queda explicado en profundidad en el último apartado del capítulo V.

7. Adaptación de la sintaxis de Elm a ElixirElm

Pese a tener una versión funcional de la biblioteca, todavía era necesario adaptar la sintaxis de Elm a la biblioteca para satisfacer los objetivos del TFG. Durante 7 días se desarrollaron unas modificaciones en el módulo de construcción de GUIs. Estas modificaciones quedan detalladas en el apartado cuarto del capítulo IV.

8. Expansión de los módulos para la inclusión de listas

Con el objetivo de ampliar la funcionalidad de ElixirElm, se propuso ampliar el conjunto de elementos representables para que se incluyan los componentes gráficos para representar listas con elementos seleccionables. Durante este bloque, que se extendió tan solo 7 días, se modificaron todos los módulos programados hasta el momento para adaptar las funcionalidades al nuevo elemento.

9. Expansión de los módulos para la inclusión de diálogos modales

Por último, se decidió hacer la incorporación a la biblioteca de los diálogos modales. Esta incorporación se implementó en 30 días, separados en dos fases, en las que se modificaron los módulos para incluir las funcionalidades necesarias que permitieran el uso de diálogos.

II Introduction

The goal of this project is to create a library that simplifies the way in which graphical user interfaces are developed in the programming language of Elixir. This library will be based on the architecture used in the language Elm.

Throughout this chapter, we shall put in context the background that has motivated the development of this project, as well as the objectives that were set for this project along the early stages of its development. Also, this chapter includes a detailed description of the work plan in every phase of the development.

Background

What is Elixir?

In order to understand the needs that motivated the realization of this project it is important to explain the Elixir programming language, for the project is oriented to strengthen its shortcomings.

Elixir is a functional programming language, oriented to processes which it is built on top of another language of similar characteristics: Erlang. Among its main peculiarities is the support for metaprogramming, which allows the expansion of the language's own syntax with the purpose of incorporating domain specific constructions.

Motivation

In spite of all the potential that Elixir has, it lacks of essential characteristics when building Graphical User Interfaces (GUIs). In order to fulfill this tasks at present, Elixir makes use of a library inherited from the language on top of which Elixir is built. This library is WxErlang. Furthermore, this library is adapted at the same time from the original WxWidgets for the language C++. This means that, as WxWidgets is based on an imperative, object oriented paradigm, the GUIs creation in Elixir using WxErlang is made in a non natural way.

As opposed to that, Elm is another functional programming language, oriented to the programming of GUI applications for web browsers. This language implements a simple method in order to programming this GUIs, but lacks other key functionalities that Elixir has. Among the main downsides of Elm is its scope, which is limited to web applications.

Having said that, this project aims at the integration of the facilities that the Elm architecture grants when developing GUIs into the language of Elixir.

Objectives

In conclusion to the background, the main objective of this project is to develop a library for Elixir that gives a layer of abstraction over WxErlang, providing the programmer a more adequate model for the GUIs development in this language.

From this main objective there was established a list containing the objectives that gave detail to the development of the library. Those objectives are:

- **To approximate the resulting abstraction layer syntax to Elm's**, as its syntax is fairly simple and easy to understand. In the structure granted by this layer, the programmer must specify the GUI's model as well as program functions for the initialization and modifications of a VDOM.
- **To grant support for the basic elements that most GUIs use.** Thus, support will be implemented for the elements that have been considered basic in any user interface. This elements are as follows: boxes for text output (named static texts in WxErlang), boxes for text input (text controls), elements for containing and arranging other elements (sizers) and lists of selectable elements (list boxes).
- **To add the possibility of creation and usage of modal dialogs**, giving the programmer the capability to create and show these dialogs from either a window or from another modal dialog, so as to deepen the library's capabilities of constructing GUIs.

Work plan

Along the development of the project, the work was organized in phases of fifteen days, which started with a meeting between the student and the project's director, where goals for the next fifteen days were set and the goals for the previous fifteen were reviewed. Depending on the state reached on those goals, it was decided whether to go deeper into those goals or to aim for another. It is also worth mentioning that during the examination periods there were exceptions on the periodicity of this meetings.

These goals can be grouped into the following work periods:

1. Familiarization with technologies

Although I had experience with other functional programming languages, it was necessary to dedicate the first three phases to become acquainted with the languages, libraries and other technologies that were going to be used in the development of this project. This period of work is further explained in the third chapter.

Along these phases, efforts were mainly put into the understanding of Elixir and the creation of GUIs with the WxErlang library. That was achieved through solving firstly simple exercises that could exemplify the general use of the language's syntax, and

then more complicated ones that explained the use of WxErlang in the creation of simplistic GUIs. On those late exercises the use of the library's constants and specific modules used on the whole development of ElixirElm came into play.

2. Codification of the GUI elements into difference trees

   In this period are the phases that were focused on creating a recursive data structure that could describe any GUI. This data structure should be capable of contain every necessary attribute that is essential to the instantiation of any element of a GUI with WxErlang, thus representing in detail the main characteristics of any of those visual elements.

   Throughout this work period, and depending on the goals set on each of the phases it contains, the data fields these trees hold were in constant change so that the tree could adapt to the needs of the phase, until it finally settled on the difference trees with which ElixirElm counts. For further information about the difference tree, read chapter fifth.

3. Building a GUI from a difference tree

   Once the elements of a GUI were successfully codificated in the difference trees, the next step was to build and show in the screen the GUI whose information was coded on the difference tree. This work period contains only one phase of fifteen days, as having debugged a definitive version of the data structure for the representation of GUIs made it easier to program a module that could explore and instantiate the elements it represented. This module is explained in detail in the second section inside the fifth chapter.

4. Development of a module to find differences between interfaces

   This work period also relies heavily on the structure developed in the second one, resulting in a development that lasted only one phase of fifteen days. With the implementation of these modules it was necessary to make modifications to the difference trees' structure, so that they could represent in a more comfortable way the differences between two graphical interfaces.

   It is important to mention that in this period the first version of the comparison module was developed, and that as the development of the project was progressing it will be updated to fit the project's needs. A detailed description of this module can be found on the third section of the fifth chapter.

5. Development of a module to apply changes to graphical interfaces

   It was necessary to develop a module that, parting from the differences found in the previous module, could update the GUI that was currently been displayed on screen. Due to the slowdown suffered from the difficulty of the development of said

module as well as the lack of documentation for WxErlang, this period was extended for two phases of fifteen days each.

After exploring the possibilities of future updates on the module it was necessary to review the descripting capabilities of the differences found between graphical interfaces. Therefore, it was decided that the difference tree as well as the differentiation module had to be updated to fit the updating module.

There can be found a more concrete specification of the updating module on the fourth section in fifth chapter.

6. Development of the event and GUI handling module

Once the previous modules were completed, an implementation of a module that would facilitate the use and management of the events by the programmer was needed as well as the dynamic updating of any changes to the GUI. The development of this module proved to be complicated, being extended through an entire month and producing multiple modifications on every module programmed to date as well as on the difference tree structure.

Once the module was finished, the library of ElixirElm was ready for use, but it still lacked the desired syntax. This last module is further explained in the last section of the fifth chapter.

7. Adapting Elm's syntax to ElixirElm

Although already having a functional version of ElixirElm, it was still necessary to adapt the syntax of the language Elm to the library in order to fulfill the objectives of the project. The modifications needed on the building module were completed in half a phase of fifteen days. These modifications are detailed in the fourth section of chapter fourth.

8. Extension of the modules in order to include selectable-item lists

With the target of expanding the functionality of ElixirElm, the inclusion of lists in the set of representable elements of the graphical component was proposed. During this seven day phase, all modules programmed had to include all necessary modifications regarding lists that already existed for the rest of visual elements.

9. Extension of the modules in order to include modal dialogs

As the last modification to the library, we decided to include modal dialogs. This inclusion was implemented through two phases of thirty days in total, in which all the modules of the library were extended to include the functionalities needed for these new elements.

III Herramientas y tecnologías

A lo largo de este capítulo se irán describiendo cada una de las tecnologías empleadas en el desarrollo de este TFG.

Erlang [2],[4]

Erlang es un lenguaje de programación concurrente, funcional, dinámica y fuertemente tipado y de código abierto. Es importante mencionar este lenguaje, ya que el lenguaje principal de la librería a desarrollar se basa en este. Este lenguaje funcional cuenta con las características de inmutabilidad de los datos, ajuste de patrones y tolerancia a fallos entre las más relevantes.

El papel principal de este lenguaje es el de referencia, ya que a partir de él se construye Elixir, el lenguaje principal del TFG y que hace uso de la máquina virtual de Erlang para ejecutar programas. Además, la forma de construir interfaces gráficas de la librería será a través del módulo wxErlang de Erlang, teniendo que hacer llamadas a este dentro de Elixir.

Elixir [1]

Elixir es un lenguaje de programación funcional concurrente orientado a procesos, un paradigma bastante diferente al de los lenguajes de programación más comunes como Java, el cual es un lenguaje imperativo orientado a objetos. Como mencioné anteriormente, este lenguaje está construido sobre Erlang, por lo que comparte gran parte de sus características además de ejecutarse en la máquina virtual propia de Erlang. Aun así, la sintaxis de ambos lenguajes se diferencia en aspectos clave, como el acceso a módulos o el uso de macros.

Este lenguaje es el más importante a mencionar dentro de este capítulo, ya que es en el que se ha programado el trabajo y entorno al cual giraban los objetivos de este.

wxWidgets [6]

WxWidgets es un entorno de trabajo de código abierto desarrollado en el lenguaje de programación C++. Este entorno permite el desarrollo de interfaces gráficas de usuario con una gran capacidad de integración y muy buena portabilidad entre sistemas al basar las interfaces en las librerías nativas de los sistemas.

De este entorno surge wxErlang [5], una implementación en Erlang de la librería original. Como el entorno original está orientado a objetos, cada clase está representada como un módulo y los objetos creados por dicha clase tienen su propio tipo.

La implementación en Erlang de wxWidgets está descrita como «un mapeo 1:1 de la original» [5]. Esto tiene ciertas implicaciones: la más importante es la estructura de los

programas que hacen uso de esta librería. Esta estructura es directamente imperativa, lo cual choca de frente con el paradigma funcional de Erlang, y por tanto de Elixir, creando la motivación para este TFG.

Elm [3]

Este lenguaje de programación de interfaces gráficas de usuario para navegadores web es, de forma parecida a Erlang y Elixir, un lenguaje funcional, con inmutabilidad en los datos pero, a diferencia de los anteriores, es estáticamente tipado. Este lenguaje forma interfaces gráficas en HTML a través de un DOM virtual.

Es importante hacer mención a este lenguaje de programación, porque, pese a no haber programado ningún fragmento de código en él, el uso de los DOM virtuales y la sintaxis de los programas en Elm son los aspectos principales en los que se basa la capa de abstracción que ofrece la librería ElixirElm que se desarrolla en este TFG.

WxHelper

WxHelper es un módulo de Elixir, obtenido como dependencia a través de un repositorio de GitHub. Este módulo facilita el uso de wxErlang en Elixir definiendo una función por cada constante de wxWidgets, haciendo que su acceso sea mucho más sencillo.

Aun así, se requirió hacer una breve modificación para ignorar las constantes referentes a OpenGL, porque daban lugar a fallos de compilación.

Git

Git es un sistema de control de versiones de código a través de repositorios. Permite principalmente realizar cambios, revertirlos, crear bifurcaciones y juntar dichas bifurcaciones en el repositorio.

Para gestionar el repositorio se ha utilizado GitHub, una herramienta web que permite el almacenamiento y seguimiento de los repositorios. Esta herramienta es especialmente útil para que varias personas compartan acceso a los repositorios y ha demostrado ser muy útil para la revisión del código por parte del tutor durante el proceso de desarrollo del TFG.

Visual Studio Code

Visual Studio Code es el entorno de desarrollo integrado, o en inglés Integrated Development Environment (IDE), que he decidido utilizar para llevar a cabo la programación de la librería.

La decisión de escoger este IDE fué relativamente fácil e inmediata, ya que tiene la capacidad de instalar complementos que asisten en el coloreado de la sintaxis del código en Elixir, autocompletado de palabras reservadas o módulos programados en dicho lenguaje, funcionalidades para mantener un control de versiones en repositorios Git y la

capacidad de abrir una consola de comandos dentro del IDE, que permite compilar y ejecutar código en Elixir.

Google Drive

Se trata de un sistema de almacenamiento en la nube que permite compartir archivos con otras personas a través de correo electrónico o del mismo sistema.

El uso principal que ha tenido Google Drive en el TFG ha sido el de mantener una carpeta compartida con el tutor, donde se almacenaba documentación relevante, las actas de las reuniones y en las últimas etapas, los distintos capítulos de la memoria.

Documentos de Google

Documentos de Google es una plataforma web que recoge una colección de aplicaciones web de ofimática que permiten la creación de archivos de texto enriquecido, presentaciones de diapositivas, hojas de cálculo y demás, que pueden ser editadas y accedidas por varios usuarios al mismo tiempo.

Se ha utilizado para tanto la redacción de las actas en las reuniones alumno-tutor como para la redacción de los capítulos de la memoria del TFG.

La capacidad de hacer ediciones simultáneas que faciliten la corrección de textos y de compartirlo con varias personas han sido las principales características que han llevado a escoger esta plataforma como sistema de edición.

Google Meet

Esta aplicación de Google implementa un servicio de videotelefonía que permite la creación de salas de conferencia para realizar reuniones de forma telemática entre dos o más personas. Cabe hacer una mención especial a este servicio, ya que es a través de él como se realizaron las últimas reuniones debido a la imposibilidad de hacerlas presenciales.

IV Arquitectura básica

Llegado el momento de hacer uso de la librería *ElixirElm* dentro de un programa, el programador deberá implementar ciertas funciones básicas (`name()`, `initial_model()`, `render()` y `update()`). Una vez implementadas, simplemente hará una llamada a la función `start_gui()` del módulo `Elixir_elm` para iniciar la GUI, pasando por parámetro el nombre del módulo donde se hayan implementando las funciones anteriores. A continuación se explican las funciones a implementar, todas ellas alrededor de un ejemplo simple de un contador, con dos botones y un cuadro de texto, que es el que se muestra en la siguiente figura:

El modelo de funcionamiento de la biblioteca requiere que se mantenga un modelo de la interfaz, que podrá ir variando a lo largo de la ejecución del programa a través de mensajes enviados por los eventos de la interfaz. Estos mensajes deberán ser definidos por el programador al igual que el cambio en el modelo que estos suponen.

En esta GUI de ejemplo el modelo será simplemente un número entero para indicar el valor del contador en un momento dado.

Los mensajes serán de dos tipos, el mensaje de `:decrement` asignado al botón de "-" e `:increment` asignado al botón de "+".

Función `name()`

Esta función sirve para dar título a la ventana que contendrá la interfaz. Tanto los módulos que hagan el papel de ventana principal como los que hagan de diálogos modales deberán implementar este método.

Esta función no admite parámetros y deberá devolver un valor del tipo `String`. A continuación se incluye un pequeño ejemplo de esta función.

```
defmodule Example do
...
  @spec name :: String.t()
  def name(), do: "Esto es un título"
...
end
```

Función `initial_model()`

El propósito de esta función es el de devolver un map que contendrá el estado inicial del modelo de la interfaz. `initial_model` no admite ningún parámetro y puede devolver cualquier información que el usuario requiera almacenar en el modelo.

```
defmodule Example do
  @type model() :: any()
...
  @spec initial_model :: model()
  def initial_model() do
    0
  end
...
end
```

En el ejemplo anterior hemos especificado que modelo inicial es el valor 0, indicando el valor inicial del contador.

Función `update()`

Esta función define las transformaciones que sufre el modelo cada vez que se produce un evento de la interfaz. En particular, en esta función se recogen los distintos mensajes que debe recibir la vista para modificar el modelo.

Estos mensajes pueden ser de cualquier tipo, y la vista no deberá de hacer uso de ningún otro que no haya sido contemplado en esta función. La función recibirá por parámetro el modelo antes de que se produjera el evento y el mensaje asociado a dicho evento. Por último, el valor de retorno será el modelo con las modificaciones necesarias.

```
defmodule Example do
  @type model() :: any()
...
  @spec update(model(), any()) :: model()
  def update(model, :increment), do: model + 1

  def update(model, :decrement), do: model - 1
...
end
```

En la función anterior hacemos que cada vez que se reciba un mensaje increment, el cual se produce cuando se pulsa el botón “+” de la interfaz, el modelo (un número entero cualquiera) se incrementará en uno y será retornado por la función. A su vez, cuando se

reciba un mensaje decrement al hacer un click sobre el botón “-” se devolverá el modelo se decrementado en uno.

Una vez ejecutada esta función, el modelo resultante servirá para crear un árbol de diferencia con ese estado actual y, posteriormente, aplicar los cambios necesarios que se necesiten aplicar en la interfaz (en este ejemplo, cambiar el valor del contador).

Funciones constructoras

Antes de explicar el funcionamiento de la función `render()`, es importante hablar de las funciones constructoras de los elementos de la interfaz. Estas funciones del módulo `Builder` simplifican la creación de elementos de la vista y devuelven un elemento del tipo `VDom.t()` a partir del cual se construirá la interfaz. Existe una función de este tipo por cada elemento de interfaz implementado en la librería, y estas funciones se organizan en dos tipos según sus características: elementos visuales (botones, textos estáticos, controles de texto y listas) y contenedores. Ambas son similares en los dos primeros argumentos que reciben: una lista con las características del elemento (etiqueta, dimensiones, etc), y el identificador numérico único del elemento en cuestión.

Sin embargo, las constructoras de los elementos visuales podrán recibir una lista opcional de pares evento-mensaje, mientras que los contenedores recibirán una lista con las vistas anidadas en ellos. Además de estos, existe una constructora adicional reservada a los diálogos, que recibirá las opciones del cuadro de diálogo, el identificador, las vistas anidadas en el diálogo y el tipo de evento que lanzará al ser cerrado.

Función `render()`

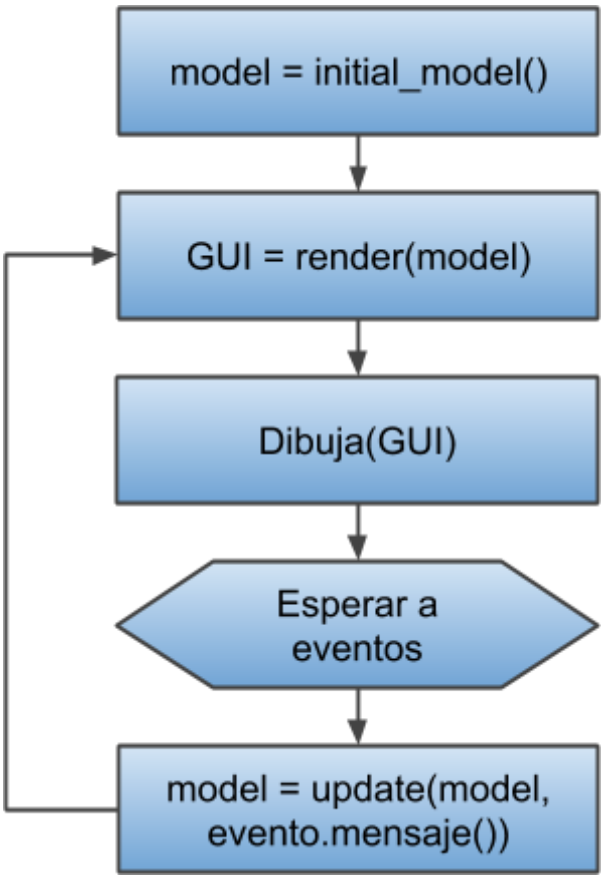
La función `render()` se hará cargo de, dado un modelo, crear una estructura para la vista de la interfaz. La función recibirá como parámetro el modelo (que podrá ser de cualquier tipo) para devolver un `VDom.t()` con la estructura de la vista.

```
defmodule Example do
  @type model() :: any()
...
  @spec render(model()) :: VDom.t()
  def render(model) do
    Builder.sizer([orientation: :vertical], 0, [
      Builder.button([label: "+"], 1, on_click: :increment),
      Builder.static_text([label: Integer.to_string(model)], 2),
      Builder.button([label: "-"], 3, on_click: :decrement)
    ])
  end
...
end
```

La función render, y por tanto el árbol de diferencia que genera, son intrínsecamente dependientes del modelo de la interfaz, que se pasa por parámetro a la función. Es necesario que la interfaz que se vaya a construir esté incluida en un sizer principal, que haga las funciones de elemento padre a lo contenido en él. En el caso de nuestro ejemplo, el valor del modelo tan solo modificará el texto que haya en la salida de texto de la interfaz, que será necesario transformar en un valor string y especificando ese valor como el atributo label del static text.

Interacciones entre las funciones

Una vez el programador haya programado todas las funciones requeridas por ElixirElm, estará preparado para ejecutar su programa. El programador deberá llamar a la función start_gui, y la librería empezará a hacer las llamadas necesarias a las funciones anteriores. El flujo de la librería es el siguiente: primero, se llamará a initial_model(), para tener el valor inicial del modelo, se llamará a la función render() con ese modelo inicial, con el árbol de diferencias devuelto por render() la librería mostrará por pantalla la interfaz y se quedará a la espera de cualquier evento producida por esta. Una vez que se haya recibido un evento, se extraerá el mensaje de este y se llamará a la función update() con el modelo actual y el mensaje recibido para recibir un nuevo modelo, el cual será pasado por parámetro de nuevo a la función render() para obtener el nuevo árbol de diferencias. Con ese nuevo árbol se aplicarán los cambios necesarios a la interfaz y se volverá a quedar a la espera de cualquier evento. A continuación se incluye un gráfico explicativo:



Comparación entre Elixir_elm y wxWidgets

La estructura final del código (implementando una función `start()` para lanzar la ejecución del programa) haciendo uso de la librería Elixir_elm es la siguiente:

```
defmodule Example do
  @type model() :: any()

  def start() do
    Elixir_elm.start_gui(__MODULE__)
  end

  @spec name :: String.t()
  def name(), do: "Esto es un título"

  @spec initial_model :: model()
  def initial_model() do
    0
  end

  @spec render(model()) :: VDom.t()
  def render(model) do
    Builder.sizer([orientation: :vertical], 0, [
      Builder.button([label: "+"], 1, on_click: :increment),
      Builder.static_text([label: Integer.to_string(model)], 2),
      Builder.button([label: "-"], 3, on_click: :decrement)
    ])
  end

  @spec update(model(), any()) :: model()
  def update(model, :increment), do: model + 1

  def update(model, :decrement), do: model - 1
end
```

Mientras que la estructura de un programa equivalente sin hacer uso de Elixir_elm es:

```
defmodule LongExample do

  def start() do
    :wx.new()
    frame = :wxFrame.new(:wx.null(),
        WxHelper.wxID_ANY(),
```

```
      "Esto es un título")
  panel = :wxPanel.new(frame)
  container = :wxBoxSizer.new(WxHelper.wxVERTICAL())
  top = :wxButton.new(panel, 1, label: "+")
  inside_lbl = :wxStaticText.new(panel, 9, "0")
  bot = :wxButton.new(panel, 2, label: "-")
  :wxSizer.add(container, top)
  :wxSizer.add(container, inside_lbl)
  :wxSizer.add(container, bot)
  :wxPanel.setSizer(panel, container)
  :wxSizer.setSizeHints(container, panel)
  :wxSizer.setSizeHints(container, frame)
  :wxFrame.show(frame)
  :wxSizer.setSizeHints(container, panel)
  :wxSizer.setSizeHints(container, frame)
  :wxSizer.layout(container)

  :wxButton.connect(top, :left_down,
    id: 1,
    lastId: 1,
    callback: fn _, _ -> increment_label(inside_lbl) end
  )

  :wxButton.connect(bot, :left_down,
    id: 2,
    lastId: 2,
    callback: fn _, _ -> decrement_label(inside_lbl) end
  )
end

def increment_label(label) do
  current = label |> :wxStaticText.getLabel() |> List.to_string()
  {current, _} = Integer.parse(current)
  label |> :wxStaticText.setLabel(Integer.to_string(current + 1))
  :ok
end

def decrement_label(label) do
  current = label |> :wxStaticText.getLabel() |> List.to_string()
  {current, _} = Integer.parse(current)
```

```
    label |> :wxStaticText.setLabel(Integer.to_string(current - 1))
    :ok
  end
end
```

Las dos versiones son claramente diferentes. La diferencia más obvia de todas es la extensión de ambos programas: el programa que hace uso de ElixirElm ocupa un total de veintinueve líneas de código, mientras que el programa que programa diréctamente con WxErlang ocupa cincuenta líneas. Además de la extensión, cabe remarcar que el primer programa es más sencillo de entender que el segundo, donde se requiere un conocimiento de WxErlang si se quiere entender cómo se construye la interfaz. Pero la diferencia más importante se da en la sintaxis y la estructura del código, llegando al punto en el que ambos programas parecen haberse realizado en dos lenguajes de programación totalmente distintos, el primero en un lenguaje funcional y de más alto nivel, mientras que el segundo parece haberse programado en un lenguaje imperativo, y que al no poder mantener un modelo como tal, complica la modificación de la interfaz a través de los eventos.

V Arquitectura general de la librería

En este capítulo se detalla la estructura de ElixirElm. Esta divide su estructura interna en 4 módulos: `Builder`, `Comparator`, `GUIUpdater` y `Elixir_elm`. También es importante mencionar que ElixirElm implementa una variante de los VDOM: los árboles de diferencia. Tanto los módulos de la librería como estos árboles se explican más en detalle a continuación.

Los árboles de diferencia

Un árbol de diferencia es una estructura de datos de uso interno de la librería. Estos árboles contienen, por un lado, la información de los elementos de la interfaz gráfica. Por otro lado, almacenan los cambios que se producen en los nodos cada vez que se actualiza la interfaz gráfica. La información contenida dentro de cada nodo es la siguiente:

- **El tipo**: un átomo que indica que tipo de elemento representa el nodo actual (botón, cuadro de texto, etiqueta, etc.). Este atributo es importante mantenerlo, ya que la instancia del objeto wxErlang que mantenemos dentro de la estructura de datos no indica de qué tipo es.

- **El identificador numérico del elemento**: cada uno de los elementos debe estar representado por un identificador numérico único para permitir el uso de eventos. Si dos elementos tienen el mismo valor para el resto de atributos menos para este, se considerarán dos elementos totalmente distintos.

- **Una lista con los atributos del elemento**: esta lista contiene las peculiaridades del elemento (márgenes, orientación, etc). Es importante mantener esta lista para simplificar la comparación entre elementos equivalentes de árboles.

- **Su instancia del SizerItem**: una vez que se instancia y agrega un elemento a la interfaz, este es referenciado por un objeto que contiene dicho elemento. A través de esta instancia se puede acceder al elemento en cuestión, y se pueden actualizar los parámetros que no son propios del elemento, sino del contexto en el que se encuentra, como los márgenes, el tamaño o la alineación en la ventana.

- **La referencia al elemento padre que los contiene**: esta es una referencia al elemento superior donde está construido. En el caso de elementos en las ventanas normales, se referencia al panel principal de la ventana que contiene dicho elemento, y en el caso de los elementos de un diálogo, se hará referencia a este.

- **La posición ordinal que ocupan en su contenedor**, importante para detectar cambios en la ordenación entre él y sus hermanos.

- **La lista de los eventos que lanza**: una lista con pares átomo-mensaje. El átomo hace referencia al evento que lanza el mensaje al que va unido.

- **La lista de diferencias:** cada vez que hay algún cambio en la interfaz, se ha de mantener una correspondencia entre los dos estados de la misma (el estado actual y el estado siguiente). En particular, hay que monitorizar los elementos que se eliminan del árbol, los que se añaden, y aquellos que cambian o se reordenan en su padre. Este componente del nodo mantiene una lista de átomos que refleja las acciones a tomar para realizar una transformación desde el estado actual al estado siguiente. Todas estas diferencias tienen asignadas un valor atómico específico de los siguientes:

    - **`:remove`:** esta diferencia es asignada a un nodo del árbol cuando el elemento no se haya encontrado en su padre equivalente de la interfaz objetivo. Esta diferencia implica que el elemento será destruido a la hora de aplicar los cambios. Esta diferencia es excluyente a cualquier otra.

    - **`:new`:** un nodo tendrá esta diferencia si el elemento de la interfaz objetivo no se ha encontrado dentro del mismo padre en la interfaz original, ya sea porque es un elemento nuevo o porque ha sido desplazado a ese nuevo padre. Los nodos marcados con esta diferencia tendrán que ser instanciados por primera vez. Esta diferencia también es excluyente a cualquier otra.

    - **`:changed`**: si un nodo contiene esta diferencia significa que los atributos de este elemento en la interfaz original han cambiado. Cualquier cambio en los atributos de un elemento gráfico supone ser marcado de esta forma. A la hora de aplicar los cambios necesarios a los atributos de este elemento se calcularán aquellos que hayan cambiado y que repercutan en la vista.

    - **`:sizer_changed`**: esta diferencia surge al comparar el sizer principal de dos estados de un mismo diálogo. Un diálogo se marca con esta diferencia si cualquiera de sus elementos ha sufrido un cambio.

    - **`:diff_in_child`**: esta diferencia es específica de los elementos de tipo sizer, y es similar a la anterior. Un diálogo se marca con esta diferencia si cualquiera de sus elementos ha sufrido un cambio, marcándolo para que sus hijos sean explorados.

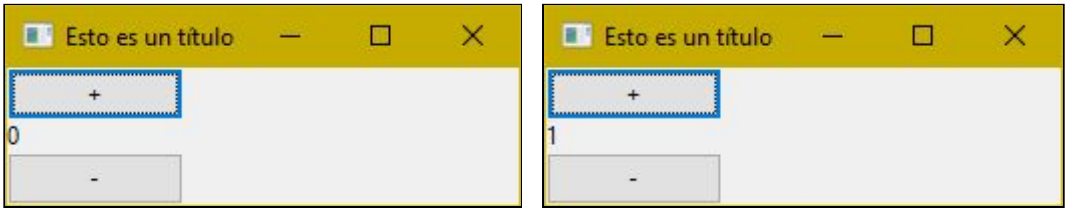
    - **`:shift_child`**: esta diferencia, también específica de los sizer, indica que los hijos de este han sufrido una reordenación, se han eliminado algunos y/o se han creado nuevos. Por la forma en la que se añaden elementos a los sizer, la reordenación de un elemento requiere que se destruya la instancia de sizer item (que no la instancia del elemento en sí) y se cree una nueva.

- `:connection`: esta diferencia surge porque una de las diferencias entre dos estados de un mismo elemento sea que el estado final contiene eventos adicionales que el anterior no incluía. Es importante remarcar que distintos eventos con el mismo tipo (ver la sección del módulo Builder de este mismo capítulo) sobreescribirán a sus anteriores. Esta diferencia, al contrario que las anteriores y al igual que la siguiente, se representa con una tupla con el átomo `:connection` y una lista de los eventos que deben ser conectados.

- `:disconnection`: de manera contraria al anterior, un nodo es marcado con esta diferencia si alguno de los eventos en el estado original de la interfaz no se encuentra en el final. Su funcionamiento es análogo al anterior, y no son excluyentes entre sí.

- **La lista con los hijos del elemento:** lista que contiene los sub-árboles de diferencia del nodo.

Para ilustrar mejor la representación de las interfaces gráficas por parte de los árboles de diferencia, a continuación se incluye el valor de este árbol a la hora de comparar los siguientes estados de una ventana correspondiente al ejemplo visto en el capítulo IV:



El término que corresponde a comparar el original (con el valor 0) al nuevo (valor 1) será:

```
%VDom{
  attributes: [orientation: :vertical],
  difference: [:diff_in_child], //Indica que sus hijos han cambiado
  events: [],
  id: 0,
  instance: {:wx_ref, 47, :wxSizerItem, []},
  position: -1,
  sub_dom: [
    %VDom{
      attributes: [label: "+"],
      difference: [],
      events: [left_down: :increment],
      id: 1,
      instance: {:wx_ref, 40, :wxSizerItem, []},
      position: 0,
      sub_dom: [],
      top_panel: {:wx_ref, 0, :wx, []},
      type: :button
    },
    %VDom{
      attributes: [label: "1"],
      difference: [:changed], //Indica que sus atributos han cambiado
```

```
      events: [],
      id: 2,
      instance: {:wx_ref, 0, :wx, []},
      position: 1,
      sub_dom: [],
      top_panel: {:wx_ref, 0, :wx, []},
      type: :static_text
    },
    %VDom{
      attributes: [label: "-"],
      difference: [],
      events: [left_down: :decrement],
      id: 3,
      instance: {:wx_ref, 45, :wxSizerItem, []},
      position: 2,
      sub_dom: [],
      top_panel: {:wx_ref, 0, :wx, []},
      type: :button
    }
  ],
  top_panel: {:wx_ref, 36, :wxPanel, []},
  type: :sizer
}
```

En el caso de esta comparación, habrá cambiado el atributo label de la salida de texto (id = 2) de 0 a 1, manteniendo el resto de elementos invariantes. Para representar este cambio, primero se añadirá a la lista de diferencias de la salida de texto como `:changed`. Después de esto, se marcarán los sizer que contengan dicha salida de texto con la diferencia de `:diff_in_child` (en este caso será solamente el sizer de id = 0), que indicará que habrá que explorar sus hijos para encontrar los modificados. Los atributos almacenados en este árbol de diferencia serán los atributos objetivos de la interfaz

Módulo `Builder`

El módulo `Builder` se encarga de dos tareas: simplificar la creación de los elementos de la GUI a través de los árboles de diferencia, usados simplemente como VDOMs y a través de estos VDOMs, instanciar y mostrar en pantalla estos elementos.

Creación de los elementos de la GUI

Esta tarea se realiza a través de las funciones constructoras `sizer/3`, `button/3`, `static_text/3`, `text_ctrl/3`, `list_box/3` y `dialog/4`, introducidas en el capítulo IV, en el apartado Funciones constructoras.

El objetivo de estas funciones, además de simplificar la creación de estos árboles de diferencia, es el mantener una sintaxis similar al lenguaje de programación Elm. Así pues, a cada elemento le corresponde una función en cuyos argumentos se podrán especificar los atributos, hijos y eventos que surjan de interactuar con dicho elemento.

Las funciones constructoras varían entre ellas en los parámetros que admiten, siendo en orden estos parámetros:

1. **Opciones:** por este parámetro el programador deberá especificar en una lista de pares clave atómica y valor las características predeterminadas de los elementos gráficos que quiera modificar. Las claves válidas para estas características pueden ser:
    - **`:proportion`**, para indicar con un valor entero la proporción con la que se debe dibujar ese elemento respecto a los demás contenidos en su mismo sizer.
    - **`:border`**, con un valor entero que indique el tamaño del borde del elemento en píxeles.
    - **`:border_directions`**, para indicar los lados del elemento que tendrán un borde (los valores se deben incluir en una lista, y pueden ser: `:up`, `:down`, `:left` y `:right`).
    - **`:alignments`**, para indicar el alineamiento del elemento en su contenedor (los valores deben estar en una lista, y pueden ser: `:left`, `:right`, `:top`, `:bottom`, `:centre`, `:horizontal` y `:vertical`).
    - **`:size`**, para indicar el tamaño del elemento en píxeles con una lista de dos elementos (la lista debe ser del estilo de `[width: valor, height: valor]`).
    - **`:label`**, para los botones indicará el texto del botón, mientras que para los elementos de entrada y salida de texto indicará lo que se debe mostrar en dicho elemento. En ambos casos, la clave debe ir asignada a un valor string.
    - **`:orientation`**, tan solo para especificar la orientación de los sizers (con los posibles valores `:horizontal` o `:vertical`).
    - **`:choices`** es una opción únicamente para las listas de elementos seleccionables. Esta opción debe ir asignada a una lista con los elementos de tipo string que el programador quiera mostrar en la lista.
2. **Id:** cada elemento gráfico debe tener asignado a él un identificador numérico único, que otorgue a la biblioteca la capacidad de diferenciarlo de otros con apariencia similar.

3.1 **Eventos de elementos gráficos generales** (opcional): todos los elementos gráficos a excepción de los sizers admiten este último parámetro. El valor por defecto será una lista vacía, pero el programador podrá especificar los distintos eventos que debe lanzar su elemento. Estos eventos se especificarán como una lista de pares clave-valor, donde el valor será el mensaje que tendrá que

interpretar la función update. Sin embargo, el evento solo podrá ser de los tipos:

- Interacción con el ratón: estos eventos se lanzarán cuando se haga sobre el elemento solamente un click izquierdo, representado con `:on_click`, doble click izquierdo, representado con `:on_double_click`. En el caso del botón derecho del ratón serán `:on_right_click`, y `:on_double_right_click` respectivamente. Estos eventos devolverán directamente al programador su mensaje asociado.

- Interacción con la entrada de texto: este evento está asociado al átomo `:on_modify` y sólo puede ser lanzado por los text controls cuando el texto de entrada que contienen cambie. Este evento devolverá al programador una dupla cuyo primer elemento será el mensaje asociado al evento y como segundo elemento tendrá el nuevo valor contenido en la entrada de texto.

- Interacción con listas de elementos: los eventos de este tipo sólo pueden ser lanzados por los list boxes. El átomo que se debe asignar a este evento es `:on_selection`. El evento devolverá al programador una dupla similar a los eventos de `:on_modify`, pero en lugar de devolver el valor de la entrada, devolverá un número entero indicando el elemento de la lista seleccionado (empezando en 0).

3.2 Hijos del elemento: este tercer atributo es particular de los sizer y los dialog. En este atributo se recogen los elementos que se añadirán en dicho sizer o dialog. Los elementos deben ser del tipo VDOM, que se pueden obtener a través de estas mismas funciones constructoras

4 Tipo de finalización para el diálogo (opcional): al igual que los eventos en los elementos gráficos, este parámetro es una lista opcional. En él se puede especificar qué acción se debe llevar a cabo en caso de cerrar el diálogo pinchando en la equis de la ventana y puede ser de dos tipos, excluyentes entre ellas:

- Salida del diálogo por defecto: este tipo de salida cerrará el diálogo, perdiendo cualquier información que haya almacenada en su modelo. Es la salida por defecto, representada por :default y sin mensaje asignado. Si no se incluye el cuarto parámetro en la función dialog este será el tipo de salida que tendrá el diálogo.

- Salida del diálogo con valor de retorno: cuando el diálogo se cierre usando este tipo de salida, se hará una llamada a la función update de la ventana que haya creado el diálogo en cuestión con el mensaje asignado al evento. El átomo que representa este tipo de salidas es `:close_and_return`.

Instanciación de los elementos de la GUI

La instanciación de los elementos se hará a través de la función `draw/3`, que tomado un árbol de diferencia, un nombre para la ventana y el PId del proceso que maneja la llegada de eventos de la GUI, recorrerá el árbol instanciando los elementos que hay en él, para después insertarlos en una ventana con el título otorgado por parámetro.

Tras haber instanciado todos los elementos del árbol de diferencia y haberlos colocado en una ventana, esta se mostrará por pantalla, finalizando el proceso de instanciación.

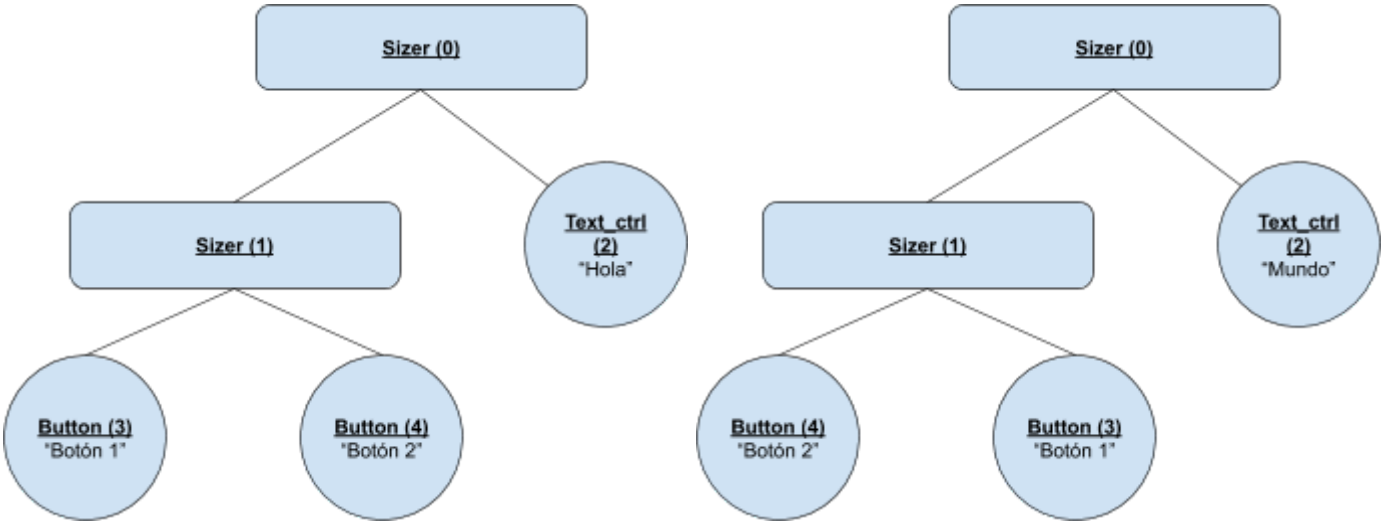
Módulo `Comparator`

El módulo `Comparator` contiene las funciones que permiten construir árboles de diferencia. Esto se logra a través de la función `get_diff/2` que recibe dos árboles de diferencia (usados simplemente como VDOMs) por parámetro, y devuelve otro árbol que contendrá la unión de los elementos de cada VDOM con las diferencias, si hubiera, entre los dos.

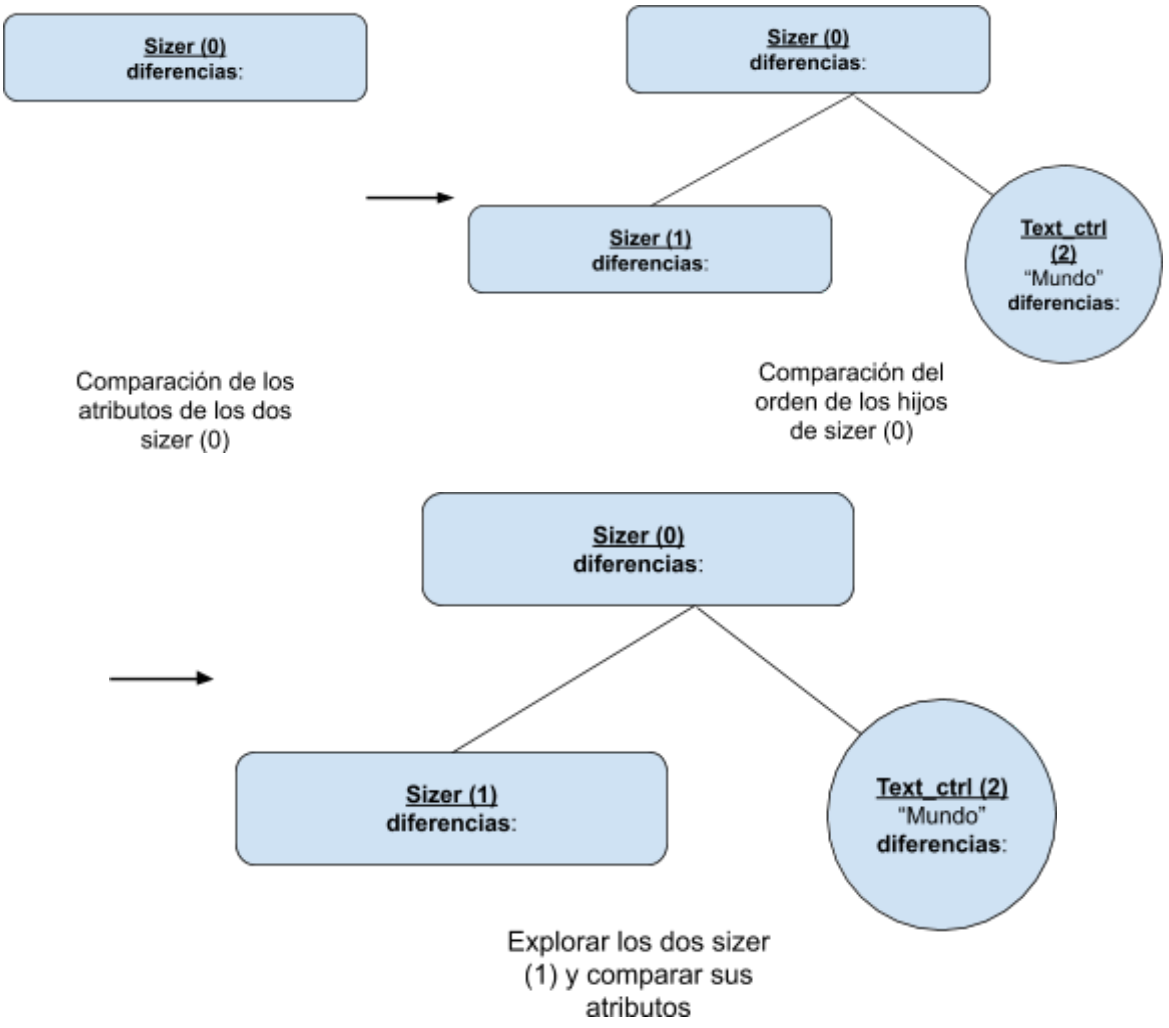
Estas diferencias se almacenan como un marcaje en cada uno de los nodos del árbol de diferencia, almacenado en cada nodo como una lista de las acciones que habrá realizado el primer VDOM para llegar a ser el segundo. Una lista de diferencias vacía indicará que tanto ese nodo como todos los hijos de este no tendrán que realizar ningún cambio, lo que acelerará el proceso de actualizar la GUI.

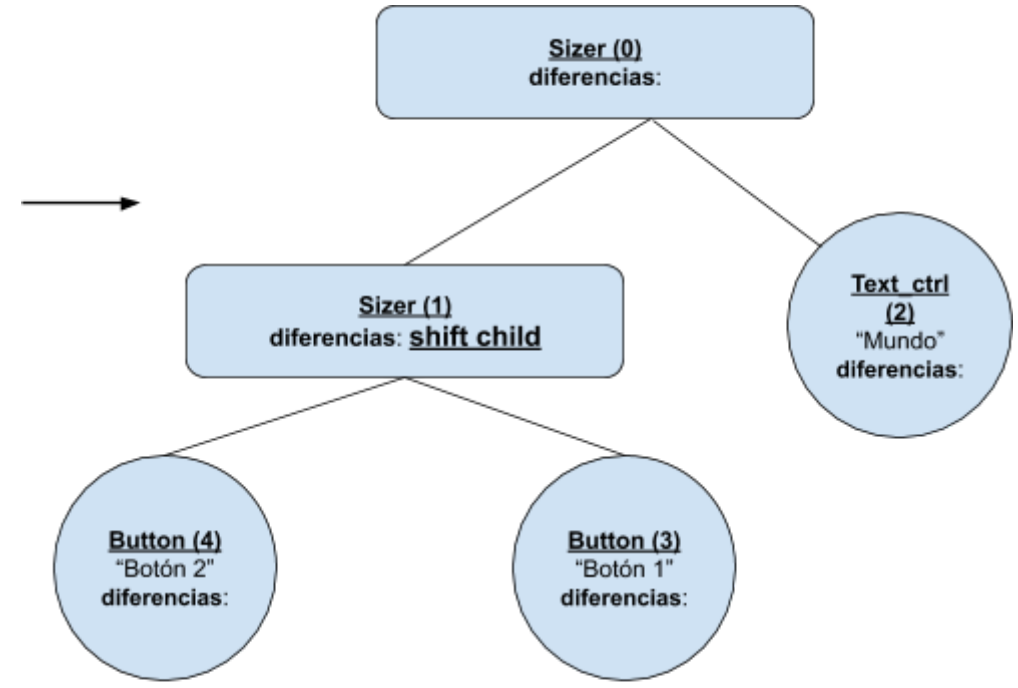
El árbol de diferencia se construye a lo largo de la exploración simultánea en profundidad de los árboles orignal y objetivo. En cada nodo sizer, primero se comparan los atributos, marcando como changed si se han encontrado diferencias entre estos, en cuyo caso se guardarán los atributos del árbol objetivo. Si no se han encontrado diferencias entre los atributos, se guardarán en el árbol final los atributos del original. Una vez se hayan comparado los atributos de los sizer, se pasará a comparar los hijos. Esta comparación se realiza buscando los elementos existentes en el original dentro del objetivo, explorando cada uno y marcando el padre si estos han sufrido algún cambio (diff in child) o, si han sido eliminados (hijo marcado como removed) o su posición dentro del padre ha cambiado (shift child). Una vez terminada la comparación con los hijos existentes en el original, los elementos que solo existan en el objetivo serán marcados como new y se añadirán al árbol de diferencias.

Así pués, de estos dos árboles:

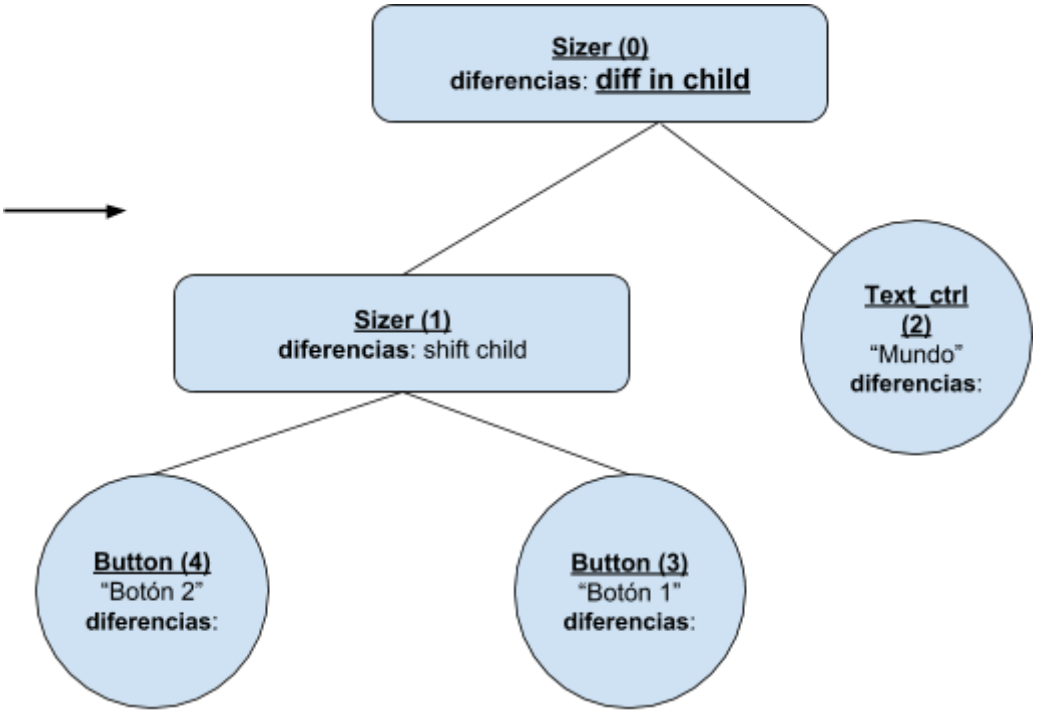


La construcción del árbol de diferencias entre el original (derecha) y el objetivo (izquierda) se realizará de la siguiente forma:

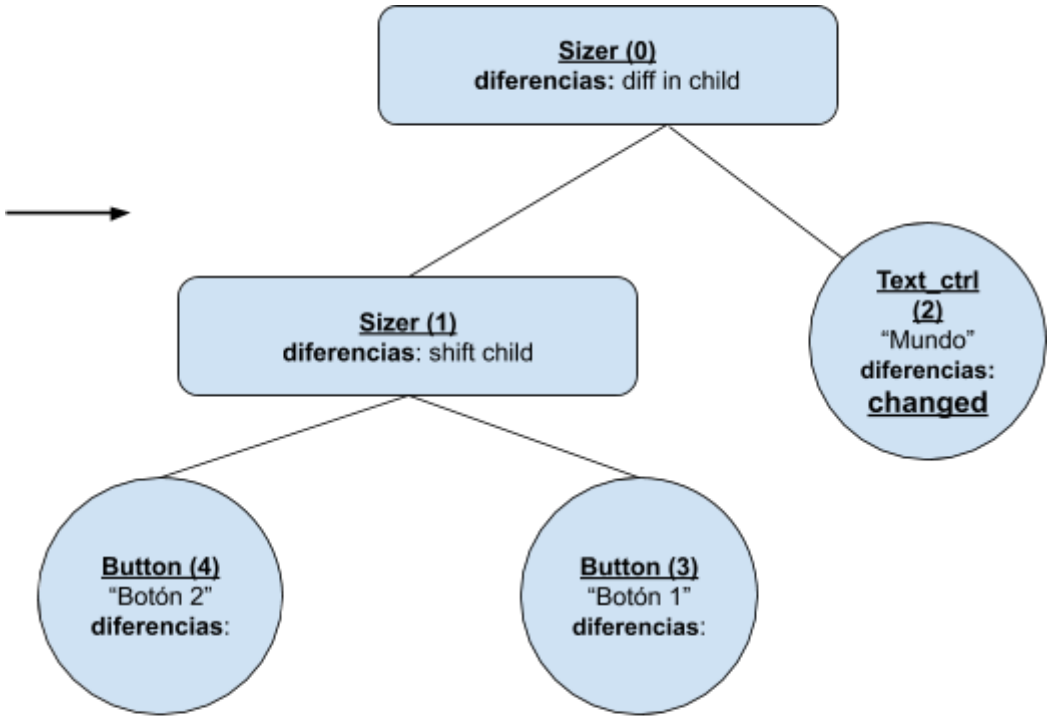




Comparación del orden y exploración de los hijos de sizer (1)



Fin de la exploración de sizer (1)



Comparación de los atributos de los dos Text_ctrl (2)

Este módulo es el único que no hace uso de la librería de WxErlang, siendo genérico para cualquier librería gráfica a la que se quiera aplicar.

Módulo `GUIUpdater`

`GUIUpdater` se centra en aplicar las modificaciones necesarias a un árbol de diferencias. Esto se logra haciendo uso de la función `apply_changes/3`, que recibirá el árbol sobre el cual se van a aplicar los cambios, el árbol de diferencias obtenido del módulo Comparator, y el PId del proceso que maneja los eventos de la GUI, para devolver un nuevo árbol de diferencias que representará el VDOM de la GUI después de aplicar los cambios necesarios y con las listas de diferencias de los nodos vacías. Esta función recorre los nodos de ambos árboles, y dentro de cada uno de estos nodos del árbol de diferencias, recorrerá la lista que contiene estas diferencias para aplicarlas una a una.

Para aplicar las diferencias que son relativas a elementos internos de los elementos de ventana (cuadros para introducir o mostrar textos, botones o listas), como cambiar etiquetas, estilos, o eventos, es necesario acceder a la instancia del elemento. Esto se hace obteniendo el objeto Window que contiene la instancia que se guarda en el árbol de diferencia y aplicando a este objeto el cambio necesario.

En el caso de los elementos sizer el proceso es parecido si se quieren cambiar, reorientar o reordenar los hijos, pero será necesario obtener la referencia al objeto Sizer de la instancia.

En cambio, los cambios sobre los atributos relativos al aspecto de un elemento con respecto al resto de elementos de la ventana, como ampliar márgenes, o cambiar alineamientos se realizarán sobre la instancia del objeto SizerItem.

Módulo `Elixir_elm`

`Elixir_elm` es el módulo principal de la librería. Este y el módulo `Builder`, donde se encuentran las funciones constructoras, son los únicos módulos con los que el programador tendrá que interactuar. En el caso de `Elixir_elm` será para hacer una llamada a la función `start_gui/1` con el nombre de un módulo por parámetro, y este módulo se encargará empezar la ejecución de la GUI.

Dentro de `Elixir_elm` se encuentran, además de la función que inicia la GUI, dos funciones recursivas, que se lanzarán cada una en un proceso distinto.

El bucle que se lanza en un segundo proceso sirve para convertir los eventos que lanzan los elementos en WxWidgets a una tupla átomo-mensaje más sencilla de procesar. Esta tupla se enviará al bucle del proceso principal para que haga los cambios necesarios en la GUI.

El bucle que se ejecutará en el proceso principal sirve para, una vez recibido un mensaje producido por un evento, que se actualice el modelo, se genere el árbol con la apariencia con el modelo actualizado, se cree un árbol de diferencias entre el árbol antiguo y el nuevo, y por último, a través de este último árbol de diferencias, se actualicen los elementos de la GUI.

El código de este bucle es el siguiente:

```
def loop(module, model, dom, receiver) do
  receive do
    :exit ->
      :wx.destroy()
      false

    {:gui_event, :exit_modal} ->
      dom.instance |> :wxDialog.endModal(0)
      true

    {:gui_event, message} ->
      model = module.update(model, message)
      command = model |> get_command()
      model = model |> get_model()
      new_dom = module.render(model)
      dtree = Comp.get_diff(dom, new_dom)
      IO.inspect(dtree)
      dom = GUIUpdater.apply_changes(dom, dtree, receiver)
```

```
        if(dom.type == :sizer) do
                        dom.instance  |>  :wxSizerItem.getSizer()  |>
:wxSizer.layout()
          dom.top_panel |> :wxPanel.layout()
        end

        if(execute_command(command, receiver, dom)) do
          loop(module, model, dom, receiver)
        else
          true
        end
    end
  end
```

Hay un elemento particular dentro de este módulo, los comandos. Los comandos son instrucciones que el programador da a la librería con el objetivo de crear o destruir ventanas y diálogos. Para mandar un comando a la librería, el programador deberá de devolver por la función update una tupla con el modelo y el comando a ejecutar. Los comandos actuales de la librería son:

- **Show modal**: este comando está compuesto por una tupla, en la que el programador incluirá el átomo `:show_modal`, con el árbol de diferencias de un diálogo que quiera mostrar. Una vez enviado este comando a la librería, un diálogo modal acorde al árbol de diferencias otorgado por el programador es mostrado por pantalla.

- **Return modal**: está compuesto por una tupla con el átomo `:return_modal` y el mensaje que se quiere devolver. Sirve para que, a través de un evento provocado por un elemento del diálogo, dicho diálogo se cierre, haciendo una llamada a update con el modelo de la ventana o diálogo que mostró el que se acaba de cerrar y el mensaje que da el comando.

- **Close modal:** este comando está tan solo compuesto por el átomo `:close_modal`, y sirve para cerrar un diálogo modal sin devolver nada.

- **Exit**: este comando, compuesto tan solo por el átomo `:exit`, termina la ejecución de la interfaz.

VI Conclusiones y trabajo futuro

En este capítulo final de la memoria se desarrollan las conclusiones y resultados que surgen a raíz de este TFG y del desarrollo del mismo. Además, también se relata una posible dirección a la que pudiera orientarse el trabajo sobre la biblioteca ElixirElm en un futuro.

Conclusiones y objetivos

Tanto el objetivo principal que se planteó para este TFG como los objetivos derivados de este, han sido cumplidos en su totalidad. A continuación se detalla la revisión de cada uno de esos objetivos:

1. **Desarrollar una librería que otorgue una capa de abstracción sobre WxErlang que proporcione al programador un modelo más adecuado para el desarrollo de GUIs en Elixir**

Este objetivo se cumplió satisfactoriamente, puesto que la abstracción sobre el modelo de desarrollo de GUIs que otorga la librería permite que el programador mantenga la coherencia del paradigma funcional dentro del código de su GUI. El programador no necesita interactuar con la biblioteca WxErlang en ningún momento, ya que se han implementado sustituciones de las constantes que usa ElixirElm para facilitar más aún su uso.

2. **Aproximar la sintaxis resultante de la capa de abstracción al lenguaje Elm**

   Consideramos como cumplido este objetivo, ya que la sintaxis de los programas que utilizan ElixirElm requieren, al igual que en Elm, la implementación de funciones que modifiquen un DOM virtual a través de cambios en un modelo, que procesen cambios en un modelo a través de eventos y funciones que den el DOM inicial de una interfaz.

3. **Otorgar soporte para los elementos básicos de una interfaz gráfica de usuario.**

   En la versión final de ElixirElm, el programador puede desarrollar cualquier GUI que contenga elementos para entrada/salida de texto, botones, listas de elementos, contenedores y diálogos modales. Consideramos que estos elementos son la base para que se puedan desarrollar GUIs con el suficiente potencial descriptivo, dando por cumplido este objetivo.

4. **Añadir la posibilidad de creación y uso de diálogos modales**

   Damos el objetivo por cumplido, ya que se incluye la capacidad de diseñar y mostrar diálogos modales a través de eventos lanzados desde la GUI. Esto

también permite la creación de diálogos sobre otros diálogos de manera indefinida.

Trabajo futuro

ElixirElm es una biblioteca con gran potencial para el futuro, ya que la abstracción que otorga a los programadores de Elixir que la utilicen es atractiva tanto por la gran ayuda que supone a estos, como por la capacidad de mejora de su versión actual. A continuación se incluyen algunos ejemplos de mejora para esta biblioteca:

- Ampliación de elementos visuales soportados

  Pese a que los elementos que ya contiene son más que suficientes para implementar GUI de múltiples propósitos, sería necesario añadir más elementos para la posible construcción de interfaces más complejas. Este proceso sería relativamente sencillo, ya que, como pude ver a la hora de incluir las listas de elementos en la biblioteca (sección tercera, punto octavo del primer o segundo capítulo en español o inglés respectivamente), la adición de elementos a la biblioteca es relativamente sencillo. Esto es así gracias al ajuste de patrones de Elixir, que permite adaptar las funcionalidades de construcción, comparación y actualización de los nuevos elementos con facilidad.

  Las posibles dificultades que pudiesen presentarse a la hora de llevar a cabo este proyecto surgirían de cara a la biblioteca de WxErlang, ya que sería necesario comprender en detalle las características y peculiaridades de cada elemento a añadir.

- Adaptación de ElixirElm a otras bibliotecas de GUIs

  Este posible proyecto es mucho más ambicioso que el anterior, ya que incluiría un gran trabajo de modificación en todos los módulos de la biblioteca. Aun así, se trata de algo bastante plausible, ya que el uso de los árboles de diferencia para representar GUIs es un método tan genérico que no depende de la biblioteca gráfica que se utilice.

  Para lograr esta meta, se podría, tanto hacer genéricos los módulos que existen hasta el momento, como crear los mismos módulos que ya existen para WxErlang pero haciendo uso de otras librerías. Desde el lado del programador, este seleccionaría la biblioteca que desea utilizar, y ElixirElm la utilizará para crear la GUI que el programador haya diseñado.

  Este proyecto podría ser una ampliación válida de este TFG, sobre todo si fuese desarrollado por las mismas personas, porque cuentan con el conocimiento del funcionamiento de la biblioteca. Aún así, podrían surgir dificultades al tener que tratar con los distintos funcionamientos internos de las bibliotecas a adaptar, ya que tan solo la documentación y familiarización con estas bibliotecas requeriría mucho trabajo.

VII Conclusions and future work

In this final chapter of the report, the conclusions derived from the development of the library ElixirElm are detailed. Also, the possible directions in which further work on this project could be taking in the future.

Conclusions and objectives

The main objective that was proposed for this project, as well as the objectives that derived from it were fully accomplished. Below is detailed the review of each and every one of those objectives:

1. **To develop a library for Elixir, that gives a layer of abstraction over WxErlang, providing the programmer a more adequate model for the GUIs development in Elixir**

   This objective is completely accomplished, and that is because the abstraction over the GUIs' development model that ElixirElm grants allows the programmer to maintain cohesion between the functional paradigm of his GUI and the rest of his/her code. The programmer will not need to interact directly with the WxErlang library at any moment as there are implemented substitution to its constants to further simplify its use.

2. **To approximate the resulting abstraction layer syntax to Elm's**

   This objective is considered as accomplished due to the fact that the syntax of the programs that use ElixirElm require, as Elm does, to implement functions that process changes in a model through events, others that modify a VDOM through those said changes and functions that return the initial state of the GUI initial DOM.

3. **To grant support for the basic elements that most GUIs use.**

   With the final version of ElixirElm the programmer is able to develop any GUI that contains for either input or output of texts, buttons, selectable items lists, elements containers and modal dialogs. We consider this elements to be the base for developing graphical user interfaces with the sufficient descriptive potential, assuming this objective as fulfilled.

4. **To add the possibility of creation and usage of modal dialogs**

   This objective is accomplished as well. This is considered this way because the programmer is able to design and show modal dialogs in his GUI through events. This also allows the creation of nested structures of dialogs indefinitely.

Future work

ElixirElm is a library with great potential for future development as the abstraction that it gives the Elixir programmers is very attractive to them because both the great help it grants when developing GUI applications and the capabilities of improvement this library in its actual version has. Examples of future improvements on the library are listed below:

- Extending the support to more visual components

  Despite the fact that the visual components ElixirElm contains are enough for implementing multiple purpose GUIs, it will be necessary to add more of this components in order to allowing the programmers to build more complicated interfaces.

  This process would be relatively easy due to the fact that, as shown on section three, subsection eight of the introduction (first chapter in Spanish, second in English), the implementation of the lists with selectable items was relatively simple, making me believethat the inclusion of more of those components would be easy as well. This ease comes from the usage of pattern matching in the functions that are necessary to modify, needing only to add clauses for every new component added to the library.

  There can also be difficulties when delivering this task. Those difficulties could come from the need to fully understand the functioning of every new element added in the WxErlang library, probably making it tedious for every new one of those components.

- Generalization of ElixirElm to other graphical libraries

  This extension is much more ambitious than the previous one, as with this task will come a great deal of modifications to every module programmed in the library, as well as the creation of new others. However, it is something quite possible, because the use of the difference trees to represent GUIs is a generic method to do so, and does not depend on the library it is using to draw those GUIs.

  To achieve these goal there is the possibility to either transform the existing modules into a generic version of them, or to create the same modules that already exist for WxErlang for the new libraries. Then, the programmer would have only to choose with which library his GUI should be built and ElixirElm will make use of that library.

This project could be a valid extension of the library, especially if it were to be developed by the same director or student, as they would already have the necessary knowledge about the functionality of the library and its modules. On the other hand, difficulties may arise from the need to adapt to the different features of the new libraries to which ElixirElm will be adapted, as familiarizing with this new technologies would require a great deal of time and effort.

Bibliografía

Libros:

[1] Dave Thomas, Programming Elixir ≥ 1.6, The Pragmatic Bookshelf, 978-1-68050-299-2, 2018.

[2] Francesco Cesarini, Simon Thompson, Erlang Programming, O'Reilly, 978-0-596-51818-9, 2009.

[3] Jeremy Fairbank, Programming Elm, The Pragmatic Bookshelf, 978-1-68050-285-5, 2019.

Páginas web:

[4] Ericsson AB, Erlang Programming Language, https://erlang.org.

[5] Ericsson AB, Erlang/OTP 23.0, https://erlang.org/doc/.

[6] Julian Smart, Vadim Zeitlin, Robin Dunn, Stefan Csomor, Bryan Petty, Francesco Montorsi, Robert Roebling et al, wxWidgets: Documentation, https://docs.wxwidgets.org/3.0/.

[7] Friedel Ziegelmayer, Elixir v1.10.3, https://hexdocs.pm/elixir/.

[8] Nathan Doctor, Jake Hoffner, Phil DeJarnett, Dan Nolan, Corewars, https://www.codewars.com.