Politecnico di Milano

Dipartimento di Elettronica, Informazione e Bioingegneria

LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

A markup language for graphical user interfaces in secure environments

Thesis of:

Luca Cioria

Matricola:

780295

Advisor (Politecnico di Milano):

Prof. Stefano Zanero

Advisor (Politecnico di Torino):

Prof. Antonio Lioy

Advisor (University of Illinois at Chicago):

Prof. Jon Solworth

To anyone I ever learnt from.

ACKNOWLEDGEMENTS

My professors, both Stefano from Italy and Jon from Chicago, for the support they gave me in writing this thesis. Stefano again, since his support goes far beyond this simple text. Michele for the latex template, without which I would have been lost. Gabe, my friend from Paris, for leaving Italy just in time to let me complete this work. My great friends and coworkers from buildo, Giovanni Gabro Claudio Andre and Dani, for their patience, complicity and understanding of my being Luca.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

TABLE OF CONTENTS (Continued)

CHAPTER **PAGE**

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In questa tesi proporremo un sistema di sviluppo di interfacce grafiche innovativo, che cerca di rispondere a dei requisiti di semplicità e di sicurezza. Il lavoro è da inquadrare nel più ampio progetto di Ethos OS, un sistema operativo che cerca di reinventare la semantica dei sistemi operativi stessi per ottenere delle proprietà di sicurezza superiori. In quest'ottica è chiaro che il presente lavoro porrà una particolare attenzione agli aspetti di sicurezza, in particolare al livello di design del linguaggio.

Il sistema proposto è basato su un linguaggio di markup chiamato *Em*, che è in grado di descrivere documenti strutturati contenenti componenti interattivi. Questi documenti, quando processati dall'applicativo client di rendering, divengono interfacce grafiche interattive. La logica dell'applicazione, sia per motivi di sicurezza che di semplicità, è interamente separata dal documento Em ed è implementata a livello remoto su un server. L'interfaccia è quindi classificabile come un'interfaccia remota, in maniera similare a quanto è HTML e un browser web.

Un'altra caratteristica fondamentale di Em è il valore semantico che viene fornito all'interfaccia grafica. Infatti il layout dei componenti e il loro stile grafico sono astratti in due documenti diversi. Il documento di layout usa un linguaggio chiamato *Ex*, mentre il documento di stile usa il linguaggio *Es*.

Alla base del design di questi tre linguaggi è una approfondita analisi, svolta nel capitolo 3, dei paradigmi di interazione tipici delle interfacce grafiche moderne. Dopo una

classificazione di queste interazioni, un confronto è fatto con sistemi di sviluppo di interfacce grafiche esistenti. A questo punto una soluzione alternativa è proposta, e paragonata a quelle esistenit in base a completezza di funzionalità e semplicità di utilizzo, mirando ad ottenere una buona copertura della prima minimizzando il più possibile la seconda.

Saranno analizzati anche i possibili rischi dal punto di vista della sicurezza relativi a un tale sistema grafico, concentrandosi sui problemi che Em, Ex ed Es possono risolvere attraverso il loro design.

Inoltre, per validare quanto proposto, un'implementazione prototipale è stata sviluppata. Questa implementazione copre una parte rilevante delle funzionalità proposte, scelte in modo opportuno per confermare la fattibilità di tutti gli aspetti principali che stanno alla base di Em, Ex ed Es.

CHAPTER 1

INTRODUCTION

This thesis is developed as part of the efforts for the Ethos project, developed at the University of Illinois, Chicago. Ethos is an experimental operating system, in which the focus shifts from features and flexibility to more advanced OS semantics and security. This semantics (as defined by its system calls) includes security services, such as authentication, authorization and isolation, and its abstractions, with strong typing enforced at all levels.

Currently, no GUI system is fully implemented in Ethos, and this thesis works in the direction of fulfilling this need. Previous work has been done, on which the proposed design builds upon, and a prototype implementation of the GUI has also been realized (explained in greater detail in chapter 2). However the current design and implementation is very limited, and has been found to be not sufficient, in particular for all aspects pertaining layout, style and interactivity. For this reason, we present here a more complete and substantially different design, alongside with a prototype implementation which we hope will serve as seed for a complete one.

Our proposed GUI system for the Ethos OS is designed following the philosophy of the Ethos project, which tries to reinvent the semantics underlying an operating system so as to optimize its security and simplicity. Based on this consideration, the two main points the design is based on are:

- simplicity and minimality of concepts over flexibility, as a way to enforce good practices and reduce security risks;
- security by design, which is translated in an effort to address common security flaws in the design of the language itself, freeing the developer or many responsibilities.

The thesis will be structured in these chapters:

1. **State of the art** A quick overview on the current research related to the main topics addressed by this thesis, followed by a description of the work previously done on this project.
2. **Design** An in depth view of the formal design process that led to the definition of the Em, Ex and Es languages, based on the analysis of common UI interaction paradigms and their typical implementations in existing toolkits..
3. **Em** An introduction to the proposed Em language, followed by its complete specification.
4. **Ex** The definition of the layout system for Em, and its associated language: Ex.
5. **Es** The definition of the style system for Em, and its associated language: Es.
6. **Interaction** An explanation of how an interactive GUI can be created with the proposed languages and a server-client infrastructure.
7. **Security by design** An analysis of how Em, Ex and Es perform security wise with respect to common threats related to UIs.

8. **Implementation** A prototype implementation of a renderer for Em, Ex and Es.

9. **Conclusions** Conclusions regarding the effective success of the proposed design, its contribution and the points that could be subject of further development.

CHAPTER 2

STATE OF THE ART

2.1 Related work

In this section, we explore the state of the art and previous works related to the subject under analysis. Because of the broad scope of this thesis, this section will be divided in two subsections, each one referring to a specific area of research.

semantic and adaptive interfaces This area focuses on how to add semantics to a user interface, so as to make automatic adaptation to the environment possible.

secure user interfaces Research on security threats directly tied to GUIs, and also on other more general risks that are of particular relevance for user interfaces.

2.1.1 Semantic and adaptive interfaces

The need for a semantic interface derives mainly for its ability to adapt to different situations. Extensive research has been done, for instance, on the adaptivity of interfaces to custom user profiles. A typical example is to optimize the experience for motion impaired people while preserving the same interface definition for standard users. This problem is typically tackled with the separation between semantics and visual representation, also a key component of our project.

We consider the semantics of a UI to be strictly tied with its interaction paradigms, as explained in chapter 3. The final output of these paradigms is a set of UI components that

enforce visual formalisms. This approach has been already followed in the literature, in [1] an extensive discussion is made on the validity of such formalisms. Other formalisms can be studied and applied to the design of a UI, such as audio or movement based ([2]), however we will concentrate only on the visual part for this work.

Having a formal semantics for GUI applications gives us the flexibility to adapt such interface at runtime based on the current needs. As discussed in [3], the separation of application logic and interface is the first necessary step. We can then develop a system that flexibly and automatically renders the interface on the device based on a UI model, such as in [4] and [5]. Another approach is to use this semantic information to help the designer develop specialized versions of the interface for particular devices or users, such as in Trident [6] and Mobi-D [7]. Our approach is to provide adaptive flexibility to the end user device and preferences, making it simple for the programmer to implement adaptive behaviors, but without going as far as automatic generation.

2.1.2 Secure user interfaces

As proposed for the first time in [8], security and human computer interaction should be seen as a mixed branch of research when it comes to secure user interfaces. An interface can be considered secure, at its basic level, if the users build trust in it with time and use. Therefore, even for graphical applications that do not expose their users to threats, security can be compromised if the GUI is not trusted by its users.

This being said, we should think how to isolate the security concerns most related to GUI applications. First, we should see this problem in the context of the Trusted Path problem ([9, 10]), which places the GUI as one component out of many that is to be trusted.

The weakness between computer and human does not only pertain to non-technical social engineering attacks, involving naive users fooled into clicking on malicious links of components. It can also be relevant for security-conscious users, with more advance visual spoofing techniques.

When thinking about the security of a user interface, we concentrate on those risks that are naturally tied to a GUI environment, excluding all the security flaws that have their origin at a different level. Therefore, we concentrate of flaws such as visual spoofing, click-through and similar attacks based on creating false or confusing graphical elements and interactions. In [11], the authors analyze typical security flaws in user interfaces, and elaborate an automatic way to find such threats. The identified threats will be analyzed in detail in chapter 9.

2.2 Previous work under the Ethos project

As mentioned in chapter 1, this thesis is the continuation of a previous effort to develop a graphics system for the Ethos OS. Some of the ideas developed in the following chapters are taken from the previous Em design, while others are completely original. In the following list we will talk about the main connection points between this thesis and the previous work done on the Em language.

markup The idea of using a markup language was already present and has been adopted. The declarative nature of Em fits perfectly with its definition as a markup language.

style Style was already separate from Em, and defined the Es language. However, the layout information was mixed with the structure of the document. This separation has been taken further in the new Em, separating clearly structure, style and layout in three different languages.

layout Layout in the previous Em was part of the Em markup and based on the idea of frames. However, no real layout logic was developed, and only the most basic layouts could be implemented. The new layout system, now separate from the Em markup, is simple but very generic and flexible, providing the power to create more complex layouts.

commands To edit an Em document specific commands were designed, in particular Insert, Remove and Update. This command based approach has been kept intact with the EmCommand object. To specify where to apply a command, a selection syntax is used. While already present in the previous Em, the current syntax is much more flexible and human readable, being based on attribute selectors.

text The previous Em was very focused on text layout, taking heavily from the latex typesetting system as inspiration. While the focus now shifted to a more global view on UI design, we kept this great typesetting flexibility with a more generic approach.

implementation The previous prototype implementation was developed in Java, using the HTML rendering capabilities of Swing components. It was a remote renderer

that communicated in JSON. Also, a very basic Em editor was developed using the same technologies. The new prototype is web based, and therefore uses a browser to render the final UI. No editor has been developed in the scope of this thesis.

CHAPTER 3

DESIGN

3.1 Introduction

In this chapter we explain in details the formal design process that was behind the definition of Em, Es and Ex. As discussed in chapter 1, the Em language, as well as its style and layout languages, are designed with a focus on security and simplicity. It is important to remind that simplicity is not only relevant for the developer, but it is also a good heuristic of the number of security flaws, as we have seen in section 2.1.2.

Starting from these considerations, what we will be optimizing in the design of Em is the ratio between the expressive power of the UI markup language and its complexity, measured in number of elements and properties defined.

The objective is to design a UI toolkit that can implement 90% or more of typical UI paradigms, with only 10% or less of the conceptual complexity. We will proceed in this order:

1. in section 3.2 a semi-formal categorization of UI paradigms will be presented, in order to better understand the structure of user interfaces and the role each component has;

2. for each one of these categories, that we call building blocks, we will analyze how the relative paradigms are implemented in three major UI toolkits (Java SWING, Android and HTML browsers);
3. after having a clear understanding of the role and typical implementation of such paradigms, we will draft the definition of the Em components that are able to perform similar functions;
4. as a last step, we will evaluate the design decisions taken in step 3 on a qualitative scale, to see how much expressive power has been preserved and how much complexity has been minimized.

3.2 UI building blocks

There are various ways to think about the building blocks of a UI. Commonly we reason in terms of the *widgets* that compose it. With *widgets* we refer to the set of UI components, such as text boxes, buttons or menu bars.

However, this way of thinking has a major issue which makes it not optimal for our analysis. It implies some underlying choices of how users are supposed to interact. We want to rethink UI components based on their function, and how they can be performed. For this reason we think instead about building blocks, as illustrated in figure 1.

These building blocks are not directly related with UI components, but rather with the interactions that the user performs on them. In other words, we shift our focus from widgets to basic user interactions, which we call *UI paradigms*, so as to be free to reinvent how these interactions can be implemented.

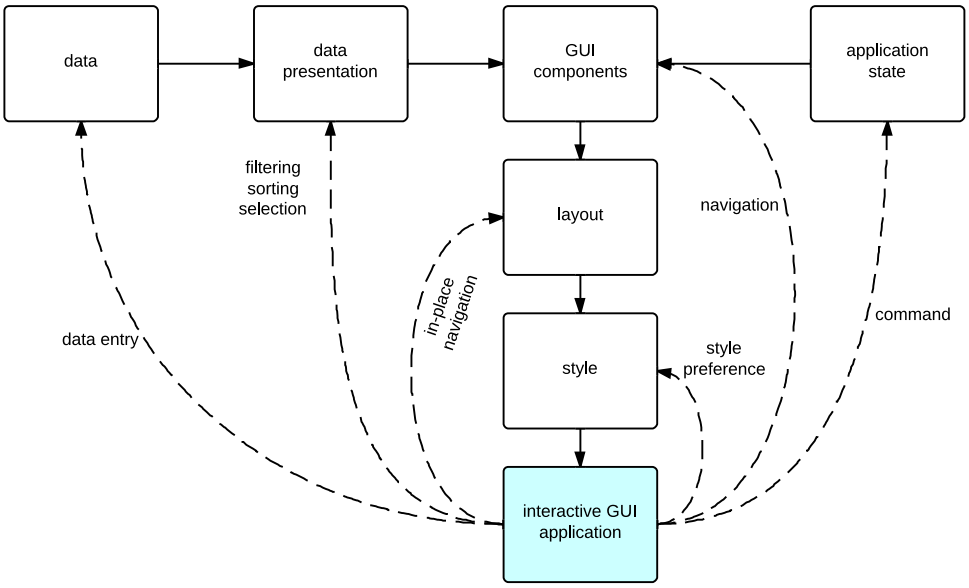

Figure 1: building blocks of a UI

In the following sections we will explain the meaning of each building block, and the corresponding user interactions. Further details of these interactions will be provided in sections 3.4 to 3.8.

3.2.1 Data and application state

The first two building block are *data* and *application state*. *Data* represents the persistent data of the application, such as the documents it elaborates and its settings, and its basic interaction is *data entry*. *Application state* represents all those information that

are not necessarily persistent and define the current, temporary state of the application. The basic interaction with the application state is the *command*. The difference between persistent and non persistent data should not be interpreted strictly as what is saved when the application is closed, but rather as what the user would commonly expect to be saved.

3.2.2 Data presentation

Data presentation represents the act of deciding what data, and with which characteristics, should be displayed on screen. Its first two basic actions are *filtering* and *sorting*. Example components that provide these interactions are sortable tables, pagination controls, search boxes and many others. Further details and examples on filtering and sorting interactions will be provided in section 3.2.2.

Selection is another basic interaction related to *data presentation*, which lets users select and deselect elements, defining a temporary set of *selected elements* that is typically used to specifically target other commands. For example, the list of files in a file browser exposes this interaction. Figure 2 (taken from *Finder*, the OSX file browser) shows the set of selected elements, represented by the blue background color).

3.2.3 GUI components

GUI components are the widgets that compose the interface. Their presence on the screen represents the current *position* in the application. This position is called in different ways depending on the context, such as *page* in web browsers, or *window* in traditional desktop applications. The basic interaction on UI components is therefore *navigation*, the act of deciding which components are present on the screen. Examples of navigation

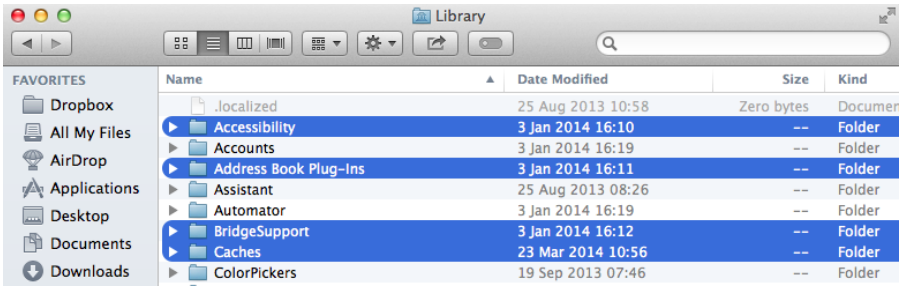

Figure 2: selection interaction

components are links in web browsers, or more in general buttons that, when clicked, trigger a change of the current position in the application.

3.2.4 Layout and style

The layout defines the position and size of UI components, which often adapts dynamically to the size and form factor of the container (display). The basic interaction for layout is *in-place navigation*, similar to navigation as explained in section 3.2.3 but only affecting the size and position (and not the presence) of UI components.

The style defines the appearance of UI components in terms of color, font type, size, icons and all those details that affect neither the layout nor the behavior of the application. Its basic interaction is the definition of *style preferences*, such as setting the desired font size in a text editor.

3.3 Comparison with other UI toolkits

In this section we will examine in detail the interactions related to the building blocks we defined. For each interaction, we will talk about its most common implementations, and

analyze qualitatively how the test toolkits perform. Finally, we will suggest a list of new components (Em components) that tries to satisfy most needs with reduced complexity.

3.3.1 Test UI toolkits

We have chosen 3 existing and commonly used UI toolkits to serve as reference point for the definition of Em components. These toolkits are:

Swing Java Swing is a traditional UI toolkit, developed for desktop applications. It is one of the most used and proven Java UI toolkits.

Android Android is a mobile operating system of recent conception, and its UI toolkit is therefore mobile oriented. Various advanced features and peculiar interaction paradigms exposed by this toolkit will serve well as comparison in our analysis.

HTML HTML is the markup language of the web and, along with CSS (the style definition language) and Javascript (the client side programming language) provides a complete UI toolkit. The running environment for applications developed in HTML is the browser. This toolkit will prove to be the most similar to Em, being both based on a markup language and with separate style definition.

3.3.2 Qualitative analysis

We define here a scale to measure how well an interaction is performed by a UI toolkit. This scale will be used to measure how Em performs in reproducing the interactive behavior of common components.

0 interaction not available, no fallback interaction provided

1 interaction not completely available, fallback interaction of lower quality provided

2 interaction available

3 superset of the interaction available, providing a greater expressive power

3.4 Data entry actions

As seen in section 3.2.1, with data we refer to all the information that is present in the UI and felt as persistent. This means that the user, when interacting with such information, expects to have it saved somewhere and available in the future.

The basic interaction with data, *data entry*, includes all actions that modify in some way (add, delete or update) data.

The following list presents the main types of data entry, and for each type the most common interaction paradigms found in GUI applications. Examples of some typical implementations of these interactions are given in parenthesis.

text edit textual data

- **simple edit** edit text without formatting (traditional text editor, simple text box)
- **formatted edit** edit text with formatting (word processors)

options select a value from a predefined set

- **selected** select one of two options (checkboxes)
- **alternatives** select one option from 2 or more (radio buttons, dropdown lists)

numbers edit a number

text entry write the characters that represent the number (simple text edit)

step edit edit a number by adding or removing a constant value (+/- buttons)

analog entry drag a component that will generate a number (slider, other draggable elements)

ordering edit the order of a list

index numbers edit directly the numbers corresponding to the index position of each element (simple text edit)

move commands edit a position by moving the element up or down (up/down buttons)

visual reposition drag the element and drop it in its new position (drag and drop list elements)

We will now see how these paradigms are implemented in the test toolkits:

Swing This traditional toolkit well supports all data entry interactions. We have text boxes, rich text boxes (that rely on HTML formatting to display styles), checkboxes, radio buttons, dropdown lists, generic buttons and sliders. We also have programmatic support to implement drag and drop actions and to further extend the capabilities of the previous components. While being able to support all the interactions, this comes at the expense of a high implementation complexity (as it is typical of traditional UI toolkits).

Android While the Android toolkit is much leaner than Swing, and focused on mobile applications, it still supports most of the interactions described. Rich text edit is not easily available (but can be implemented with much additional work). Radio buttons are not available as they are not a good fit for mobile applications. Drag and drop can be implemented, and it is a natural interaction in touch devices.

HTML With HTML we have some major differences, since this language was originally conceived for simpler, document oriented applications. We have data entry components out of the box as parts of forms. However, we do not have sliders, rich text edit and support for drag and drop. They can be implemented in client side logic (Javascript), but as usual with great added complexity in the code.

As we have seen, most interactions can be accomplished by editing simple text and giving commands. For simplicity, these are the only two interactions that Em will focus on, and they will work as follows:

EmInput The default text input box, that serves all simple text input needs. It has an important advanced feature, *validation* of its contents by regular expressions. Since a regular expression defines a list of possible values, we can use this feature backwards and implement dropdown lists with EmInput, simply enabling autocompletion on the allowed values. Not only this removes the need for dropdown lists (and radio buttons), but it gives us the flexibility to chose a value from an infinite lists (since the completion list is dynamically generated from the regular expression).

EmButton A simple button, perfect for sending commands (step edit, move commands..)

EmSelectableButton An extension to the concept of EmButton, that is selectable. It can serve as checkbox with minimal added complexity.

Since Em will not support client side logic (the reasons for this choice are explained in chapter 9), there is no way to implement a slider or a drag and drop action. We feel that these two elements are not absolutely necessary, and that the fallback solutions provided are worth the simplification in concepts and codebase. Here we can see the previous list annotated with the Em component to be used for each task, and its quality vote (as defined in section 3.3.2) in square brackets.

text :

simple edit EmInput [2]

formatted edit not available [0]

options :

selected EmSelectableButton [2]

alternatives EmInput [3]

numbers :

text entry EmInput [2]

step edit EmButton [2]

analog entry not available [1]

ordering :

index numbers EmInput [2]

move commands EmButton [2]

visual reposition not available [1]

In conclusion, Em supports with varying degrees of quality all interactions except for in place formatted text editing. The choice of not supporting this interaction was taken because of its complexity and high level of customizability necessary, which is against the main philosophy of the Em language. However, Em supports very well the display of formatted text and documents, and editing can be accomplished with direct manipulation of the Em markup with a dedicated editor. We do no feel this is ultimately a real loss of expressive power, more a different way of achieving a similar objective.

3.5 Filtering, sorting and selection actions

In this section we talk about all the interactions related to manipulating how data is presented, and more in general interacted with. First of all, we need to divide data in two main categories:

record Conceptually a single data object. It might be a document, an email, a contact in the address book or a file in a file browser application.

list An aggregation of records, not necessarily of the same kind even though this is commonly the case. It can optionally have an order.

As described in section 3.2.2, the three basic interactions pertaining to data presentation are filtering, sorting and selection, which can be defined as follows:

filtering Deciding which elements from a list are visible. The decision can be taken based on the content of records in the list (e.g. search emails by subject) or on their index in the list (e.g. show only the first 20 emails).

sorting Deciding in which order elements from a list are displayed, based on some arbitrary decision or a sortable property common to the records in the list.

selection Defining the *selected set* of records. The operation is performed by toggling elements between a normal state and a selected state. This does not directly affect how they are presented, but it is useful to define a target for further data presentation transformations or generic commands.

Figure 3: example of search box filtering

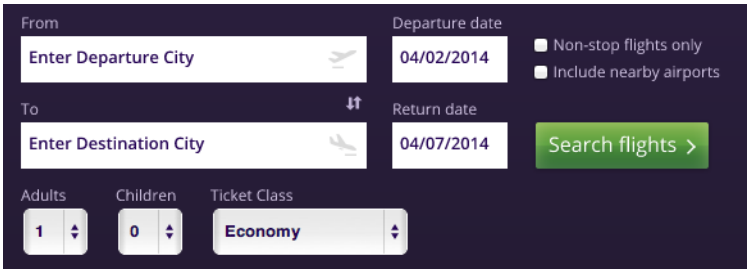

Figure 4: example of generic filtering form

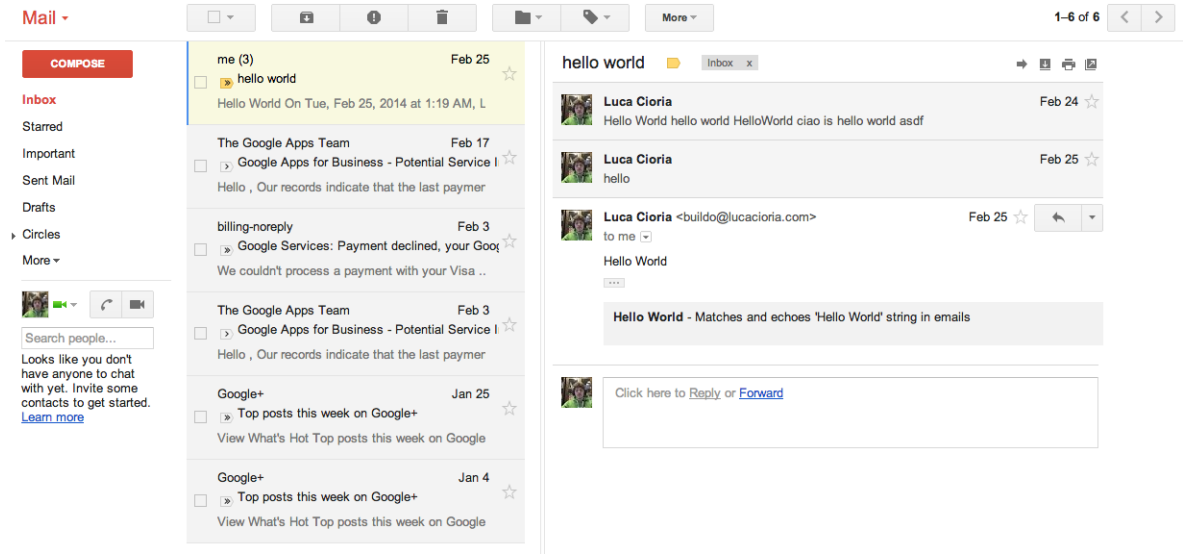

Figure 5: example of generic filtering form

In the following list we describe the main type of each interaction and the typical UI paradigms associated with it.

filtering based on content filter elements in a list based on a common property they have

- **search boxes** define a search query (as in image 3)
- **general filter forms** define a complex search by multiple parameters (structured components, typically a form with checkboxes, dropdown lists, sliders and text inputs, as in image 4)
- **selection state** use the selection state of other data as filter (as in image 5, where the selection in the list of conversation determines the filtering on the list of emails on the right)

filtering based on index filter elements in a list based on their position or nesting level

- **hidden overflow** hide the overflow of content, only displaying an interval of elements (scrolling components)
- **pagination** show one page at a time, always displaying a contiguous interval of data of the same length (paginated views with next/previous buttons)
- **folding** toggle the visibility of a nested section (folding/unfolding) in a nestable list (tree view component)

sorting change the order of display of elements in a list

headers in tabular views (where each line defines a record and each column a property) click on the column header to sort the list based on that column (advanced table components)

select sorting field use a custom, separate way to select the sorting field (drop-down list)

selection toggle the selected state of records

basic select an element by clicking on it (selectable components)

toggle change the selection state of an element by clicking on it (selectable components)

advanced advanced behaviors such as exclusive selection in list, multiple selection, contiguous election (selectable components with modifier keys)

We will now see how these paradigms are implemented in our test toolkits:

Swing Filtering based on content is performed with the usual input elements (illustrated in section 3.4). Hidden overflow is provided out of the box by components such as JScrollPane, while pagination has to be implemented manually. Swing provides a JTreeView component that works well for very simple cases of folding, but it is not flexible to accommodate for more general uses. Also, Swing provides an advanced and complex table component with sorting, filtering and many other capabilities. For what concerns selection, the selection behavior depends on the specific component being used, it is defined programmatically and can be customized. As we can see,

Swing tackles these problems by introducing a new specific component every time, that works well for its specific use case but is not flexible or generic as a concept.

Android The basic input components are also available in the Android UI toolkit. However, Android proposes a more advanced component, called the ActionBar, that can easily provide filtering based on a dropdown list as one of its features. Hidden overflow is used to great extend, and usually it is applied to the whole screen of the mobile device. Pagination is not specifically addressed, and neither are sortable tables or foldable trees. Selection behavior is on a per component basis, as it was the case for Swing.

HTML As seen in section 3.4, HTML provides the most typical input fields (form input tags). The hidden overflow is a natural behavior in browsers, since by default the page scrolls vertically. No specific component is present for paging, advanced tables (with sorting) or tree views. All these components can be implemented, with great additional effort, with client side logic in Javascript. Also, since there are no predefined interactive components, the selection behavior is undefined and needs to be reimplemented each time.

In these 3 toolkits, filtering, sorting and selection behaviors are not provided as natural parts of the language, but rather implemented up to a certain level in each specific components. On the same line, Em does not provide special features or guidelines to implement generic filtering based on content, or pagination. However, it does provide a flexible implementation of hidden overflow with scrolling (which can be both vertical and horizontal)

in a way similar to HTML. It is sufficient to have the content of an `EmFrame` be larger than the frame itself, and enable scrolling, as explained in detail in chapter 6.8, where we talk about the layout.

Em offers an interesting approach to the folding problem, with the use of sections. They will be better explained later in the current chapter, and they're complete definition provided in section 5.4.2, but at a minimum level sections allow us to organize nested content, and provide folding out of the box. They also provide convenience ways to access content, and enumerate it.

For what concerns selection, Em tackles the very common problem of selecting elements from a group in an innovative and flexible way. The `EmSelectableButton` (the only selectable element in Em, see 5.4.12) has a `nextSelectable` property, which can point to another `EmSelectableButton`. Using this property we can form a chain of buttons, and in this chain we automatically get one of these behavior:

exclusive selection only one element in the chain can be selected

one selected as in exclusive selection, but always enforcing the selection of one element

multiple selection multiple elements can be selected, with groups selection (using the SHIFT key) and toggling (using the CTRL key) automatically enabled

We have therefore three components used for filtering, sorting and selection:

EmFrame the basic Em container, can provide scrolling and hide overflowing content

EmSection a more advanced container that provides nesting levels, folding and additional features

EmSelectableButton a selectable component with the ability to form a chain with advanced selection functions provided out of the box

To sum up, here is a list of the features Em exposes and their qualitative vote:

filtering based on content :

search boxes EmInput [2]

general filter forms EmInput, EmSelectableButton, EmButton [2]

selection state EmSelectableButton [2]

filtering based on index :

hidden overflow EmFrame [3]

pagination to be implemented manually [1]

folding EmSection [3]

sorting :

headers to be implemented manually [1]

select sorting field to be implemented manually [1]

selection :

basic EmButton [2]

toggle EmButton [2]

advanced EmButton [3]

In conclusion, Em supports with varying degrees of quality all interactions, and performs better (in terms of ease of use) than most toolkits in handling selections and scrolling.

3.6 Application state and commands

In the previous two sections we have seen the actions associated with one side of the information that goes into define our UI, part of the data and data presentation building blocks. We talk now about the application state, information that is usually not expected to be persistent between sessions. This is the information about what the current application tasks are. Examples of application state are the song currently being played in a music player, or the email currently being read.

The basic interaction related to the application state is the *command*. A command is an action that the user performs and has the effect of changing in some way (even hidden) the application state, which does not fall in the previous categories (data entry, filtering, sorting and selection). There are two types of commands, and in the following list we have their definition and typical interactions.

global command a command that has a global effect, and is not tied to a specific element

button the easiest implementation, a single button triggering the command

menu element a menu bar is a set of buttons nested and visually organized, it reduces visual clutter and aggregates similar commands close to each other

keyboard shortcut a keyboard shortcut is a combination of keypresses that trigger a command

gesture mostly used on multitouch devices, a gesture is a hand movement that is recognized and has a command associated (e.g. pinch to zoom)

contextual command a command that changes its effect based on the element from which it is triggered

contextual menu a floating menu, usually displayed on right click in traditional interfaces, that shows a list of buttons which might be contextual to the element from which the menu was generated

We will now see how these paradigms are implemented in our test toolkits:

Swing Swing offers good support for menu bars with the `JMenu` component. It is a standalone element which can contain buttons, also with icons. Keyboard shortcuts can also be programmatically implemented, and gestures are generally not available (but there are specific extensions for them). Contextual menus are also available in their typical implementation (floating menu).

Android Menus in android are available in various typical implementations, such as the ActionBar (which contains a menu for the current activity). Keyboard shortcuts could be implemented programmatically but are not generally useful since mobile devices tend not to have physical keyboards. Gestures are an idiomatic interaction paradigm and natively integrated in the toolkit. Contextual menus can be implemented by

visualizing a custom option list on a secondary action, typically a long touch on an item.

HTML only single buttons are available out of the box, no specific support for menus is provided, neither global nor contextual. They can be implemented in Javascript with great added complexity. Gestures are not in general available, and shortcuts can be implemented, with some limitations, in client side logic.

The three test toolkit have very different approaches in organizing commands into menus, managing gestures and shortcuts. This is due to their fundamental difference, from standard desktop applications, to mobile applications and to web pages. However, the concept of commands and their organization in some form of menu is often useful and should be tackled in a more generic way. We use the `EmMenu` component to create a standard and powerful way to define menus. These are the two Em components needed to provide commands and menus:

EmButton a single button, which fires an event when clicked that can be associated to a command on the server

EmMenu a menu built around a group of EmButtons. It can have a title, and nested menus. What makes EmMenu different from a typical menu bar is its tight association with EmSections. In fact there can only be one menu per section, and it should contain commands that are related to that section. All menus are rendered in a single top menu bar (an not where they are defined), but only the menus corresponding to

visible sections are showed. With this particular approach, we have a simple way to define a dynamic menu bar which changes based on the context (the open sections), and therefore fulfills to a good extent both the global and the contextual command needs.

For what concerns keyboard shortcuts, keypresses are events sent to the server which can then run commands. Gestures are not supported. To sum up, here is a list of the features Em exposes and their qualitative vote:

global command :

button EmButton [2]

menu element EmMenu [3]

keyboard shortcut EmEvent [2]

gesture not available [1]

contextual command :

contextual menu EmMenu [2]

In conclusion, Em will support well shortcuts, buttons and global menus, and also provides an easy way to have dynamic, contextual menus. Even though it is not practical to have a section for each visible item, this is not a limit since we can use the selection state on a list of components and define contextual commands, in the section containing the list, based on the current selection.

3.7 Navigation

As introduced in section 3.2.3, GUI components (or widgets) are the components that are drawn on a display to create the interactive user interface. We focus here on **navigation** interactions, whose purpose is to determine which components are visible and which are not, thus changing our *position* in the application. A position change can be drastic (as in a change of web page, where every single component could change) or minor (as in the appearance of a pop-up window), but both fall under the same category. However, if we only change the position or size of a component, we talk about *in-place navigation* (see section 3.8).

These are the two basic navigation interactions and their common implementations:

view change when the view (set of UI components on the screen) changes radically

- **link** in document based languages links are commonly used to open (or load) a different document
- **switching container** in traditional toolkits switching components (tabs, multiple views) are used to switch between different views
- **windows** in multiple windows environments, a view change can be obtained creating another window and giving it focus

modular view when the view remains conceptually the same, but certain parts are hidden or made visible

collapsible panel a panel that can be interactively collapsed by the user (e.g. clicking on an icon in the panel top bar)

We will now see how these paradigms are implemented in our test toolkits:

Swing In this traditional toolkit there is no explicit concept of navigation, but there are specific layout components that can be used to simulate page based navigation. The most common one is `CardLayout`, which operates swapping different containers (of type `JPanel`) programmatically. Links are then just buttons that tell the card layout to select a different card (page). Windows are well supported with the `JFrame` container. There is no explicit support for collapsible panels but they can be implemented either programmatically or using container components such as `JSplitPane`.

Android In Android, views and navigation are a fundamental part of the framework since most mobile application, because of their smaller size, are navigation based rather than implemented as a single view. In this environment, linking is performed using the `Intent` object to open another `Activity`, which defines a view. Also, navigation operations are recorded in a stack and can be implicitly navigated back using the standard back button. Collapsible panels, and more in general views that can be customized, are not a common paradigm in mobile applications and there is no direct support for it, even though they could be implemented as usual with custom logic.

HTML In HTML links and pages are basic concepts, and backwards navigation is provided by browsers saving the history of the visited URLs (which uniquely define a page).

However, there is no native support for switching containers and windows, which can be recreated programmatically only with considerable effort. Collapsible panel are also not directly available, but can be implemented. Also, navigation is stateless and it is therefore more complicated to keep application wide state information in sync within page changes.

As we have seen, Swing and HTML provide two very different approaches on navigation. Swing is concentrated on the current window, and it is very flexible on customizing and navigating within it. However, it does not offer a simple way to link to other views or keep track of the history of visited views. On the contrary, HTML does exactly that, but it does not offer a simple way to customize the current view, organize it in panels or switch panels without changing page. Since both behaviors are desirable in different circumstances, we have tried to provide both solutions with minimal complexity in Em, using the two following elements:

EmSection a frame (rectangular container, see section 5.4.1 for details) with a specific semantic meaning. This means that its function is not only to layout components, but also to group components that are conceptually correlated. For instance a single panel could be a section, and its subpanels could be nested sections. Since an Em UI is defined in a single document, there is no direct way to switch to a different page. In fact, the concept of pages is not present. However, a similar effect can be easily obtained by defining each page as a root section, and making them visible one at a time (similar to the `CardLayout` component in Swing). In addition, sections

further provide a few additional features. They are collapsible by users, they can have titles and also appear in a table of contents, that is automatically generated based on their nesting structure. This table of contents can act as site map, or main navigation point for the whole application, and it is rendered as a global navigation menu. Furthermore, sections have automatic numbering (each section knows its nesting level) and can be easily styled. The `EmSection` component will be explained in much greater detail in section 5.4.2.

EmLink a pointer to a section. Links are an efficient way to decide which sections are visible. When clicked, a link will make the destination section become visible. Also, if so desired, it will hide all other sections at the same nesting level, thus providing a switching behavior. The client keeps a history of clicked links, and can navigate backwards in a similar way to the browser history. Also, differently from HTML, navigation is stateful and application state is not lost when moving through sections, easing server side development.

Differently from Swing, Em does not provide windows. This is done intentionally, as they might lead to confusion because content could overlap and be hidden. Since this issue is related to the security of a user interface, more details can be found in chapter 9. To sum up, here is a list of the features Em exposes and their qualitative vote:

view change :

link EmLink [3]

switching container EmSection + EmLink [3]

windows not available by design

modular view :

collapsible panel EmSection [2]

In conclusion, Em supports well all main navigation interactions, with a very flexible and semantically relevant approach.

3.8 In-place navigation

In this section we will again talk about navigation, but at the finer level of component layout. In the previous section we defined navigation as the set of interactions that affect the presence of GUI components. In the current section we will instead focus on the position and size of these components. The process of computing, for each element, its position and size, is called *layout*.

These are the two basic in-place navigation interactions, along with their common implementations:

rearrange panels Move panels and rearrange them on the screen.

draggable panels Visually drag a panel to its new desired position, based on drag and drop.

move commands Rearrange panels with explicit move commands, such as the swapping of two panels.

stretch panels Resize panels maintaining their current position.

stretchable panels Drag the border element of a panel extending/reducing it to a new size. Optionally the angles can be dragged, too, thus modifying both height and width at the same time.

change flow Change the flow of elements on the screen, keeping their order the same. For example, we have a change of flow when switching between a single column view to a multi column view, or from a list view to a grid view.

change flow command An explicit command that applies the new layout.

responsive layout The layout changes dynamically based on the screen size and orientation (typical on mobile devices).

We will now see how these paradigms are implemented in our test toolkits:

Swing Being designed to implement complex desktop UIs, Swing has some powerful components that can be rearranged by drag and drop. `JToolBar` is an example of a draggable, and dockable, component which can also stand alone in its own floating window. While this component is not a generic container, this behavior can be programmatically defined on other containers. Stretch panels can be partially implemented with a stretchable container, the `JSplitPane`, which can contain two panels and makes their separation border draggable. For what concerns the flow change, it is more complicated since each flow is usually achieved with a specific layout container. Therefore multiple container definitions need to be used and then switched

on the fly. Also, responsive layout is not available directly but can be implemented with great added complexity.

Android In Android, the idea of a custom view with draggable elements is not really present, mainly due to the small size of mobile devices. It could be implemented but it is not relevant for the platform. The change of flow works in a similar way to Swing. Responsive layout can be efficiently implemented thanks to the ability of Android to select the appropriate UI definition file based on the current device screen.

HTML In HTML there is no concept of a panel, and no drag and drop behavior is available by default. In can be implemented in Javascript with great effort. Changing the flow is on the contrary easy, thanks to the CSS layout definitions which can be changed on the fly without touching the elements and their containers. Also, responsive layout is well supported thanks to media queries, which can adapt style properties to screen sizes.

We can see that traditional toolkits are quite powerful in handling these kinds of interactions, while HTML stands far behind in the features it provides. Em stands in the middle, starting from the HTML approach of separate layout files (Ex files, explained in chapter 6) which are easily customized, but also providing resizable panels out of the box. No support for generic draggable elements is included, as this would complicate too much the interface while providing, in our opinion, only a limited benefit.

The main component responsible for these features is the `EmFrame`.

EmFrame + Ex `EmFrame` is the basic Em container, which supports the layout property `resizable`, used to define which side of the component can be resized by the user by dragging it (see section 6.8 for details). Combined with the other Ex layout rules, this is a simple but powerful way to resize components on the screen without the need for custom client side logic.

For what concerns responsive layouts, they can be simply achieved by conditional inclusion rules for Ex and Es files by the rendered. This means that a few different versions of Ex and Es files can be provided, each one with its optimal width range. Then the renderer will select the correct files based on the width of the current screen (or the renderer window). This works in a similar way to how HTML and CSS and media queries operate.

The following is a qualitative evaluation of how Em performs in layout related navigation interactions.

rearrange panels :

draggable panels not available [1]

move commands EmButton [2]

stretch panels :

stretchable panels EmFrame [2]

change flow :

change flow command EmButton + Ex language [2]

responsive layout conditional inclusion of Ex/Es files [2]

In conclusion, Em supports well all the basic layout interactions but it is not very powerful in complex, customizable UIs, which would need client side logic to be implemented.

3.9 Style preference

The style of GUI components defines how each element appears, without interfering with its layout. Examples of style properties are colors, font sizes and font type. Also, even though the size of components is not in itself a style property, the overall zoom factor of an application (its resolution on screen) can be considered a style property.

The main interaction related to style is the act of changing it by editing *style preferences*. We also consider zooming the interface as a separate interaction, since it is of common use and can be accomplished efficiently with specific actions.

change style preference Change a style preference.

settings panel A standard panel with settings to change style properties.

style file A specific file, not part of the UI, where style settings can be manually written.

zoom interface Resize all components on screen proportionally.

settings panel The zoom factor can be part of the custom settings panel.

global command The zoom factor can be changed by a global command available to all applications, such as in web browsers (e.g. using the CTRL +/- shortcuts).

We will now see how these paradigms are implemented in our test toolkits:

Swing No direct support for styling is available, if necessary it must all be implemented programmatically and on a per component basis. No support for zooming is available, either.

Android Style files are provided, but they are only used when writing the application. They are not accessible by the end user, and only the settings panel approach can be followed.

HTML Style is provided by CSS files (cascading style sheet), which however cannot be directly edited by the user. As usual, the settings panel approach is necessary. There is the possibility of writing extensions for browsers that modify the CSS files, giving the possibility to experienced users to change the appearance of web pages. This operation is not user friendly, as it is not meant to be performed by the end user. Global zooming is available directly as a browser actions, and it is performed fully by the renderer engine.

We can see that only the settings panel approach is really available in the test toolkits. This limits drastically the ability of users to change the style of applications to their need. For example, an application might not be adequate for color blind users, or have a font style that is difficult to read, and the responsibility for these choices would fall entirely on the developer.

In Em, we suggest an approach based on style files, similarly to CSS and HTML. The main difference is that the developer can easily mark certain style properties as user ed-

itable, providing them a name. A settings panel is automatically generated by the renderer, in a consistent way for all applications. Also, global zoom can be implemented by the renderer in a similar way that browsers do.

The main responsible for these features is the Em renderer and the Es style system, explained in greater details in chapter 7.

The following is a qualitative evaluation of how Em performs in style related interactions.

change style preference :

settings panel custom or automatic with renderer + Es files [3]

style file Es files [2]

zoom interface :

settings panel not available, but not needed [2]

global command available through the renderer [3]

In conclusion, Em (using its style language, Es) defines a powerful style system that has the advantage of being easily accessible by the end user.

CHAPTER 4

ARCHITECTURE

4.1 Introduction

This chapter will provide a general introduction to the design of Em, Ex and Es GUIs and how they integrate with the application logic. A general description of the three languages is given, and for each one there will be a dedicated chapter. Although the following chapters are conceptually distinct, they should be read in order since each builds on the previous one.

Em, Ex and Es are three languages that work together to specify interactive GUIs. Em defines the hierarchical structure of the elements in the GUI. It also defines the possible interactions with the user (clicks...). Ex specifies the layout, where and how big the elements and containers are, and how they adapt to different devices. Es is a styling language to add colors, fonts, borders and a few other graphical properties to the interface.

4.1.1 declarative languages

Em, Ex and Es are declarative languages: you cannot implement any program logic (except for the predefined actions) in these languages, the GUI only acts as a frontend for the program logic. This is done for two main reasons: firstly, client side logic is a security threat. The interface functioning could be in some way altered so as to show unexpected behaviors. Moreover, having client side logic makes it much more difficult

for the server (program logic) to keep track of the application state, introducing potential risks (the server executes actions while believing the remote interface to be in a particular state, which has been modified by malicious client side code).

The second reason is separation of concerns, and the resulting ease of use and clear organization of the application. As we keep structure, layout and style clearly separated, we have the program logic centralized on the server and not mixed with the interface definition.

4.1.2 application workflow

This list explains the basic execution workflow of an application that uses Em, Ex and Es as GUI fronted. The architecture is server/client, where the client only displays the interface and sends interaction events (clicks...) back to the server, where all the real logic is coded, as explained in 4.1.1. Even though these two components are separate, it does not mean that they should be physically apart (remote). The server and the client could easily be running on the same, local machine.

1. SERVER sends complete Em, Ex and Es documents to CLIENT
2. GUI is rendered by CLIENT
3. GUI LOOP
   (a) user interactions trigger events that are sent to the SERVER
   (b) is the event is relevant, the SERVER responds with a series of edit commands over the Em,Ex and Es documents

(c) the CLIENT applies the edits and renders the updated GUI

4.2 Structure, Layout and Style: three levels of GUI design

Em, Ex and Es provide a clean and effective way to separate the most semantic aspects of an interface from those related only to style and visual appearence. There are three levels at which we operate:

1. STRUCTURE (Em): defines a hierarchy of Em elements with semantic meaning. Elements that are related to each other will generally be under the same parent in the hierarchy. The structure by itself does not provide a clean layout of the UI, but it provides the general organization of the elements. An example of structure would be a panel containing text fields and buttons (e.g.: a data insertion form). We know they belong to the panel, but we don't know where and how exactly they will be drawn on the screen.

2. LAYOUT (Ex): specifies the actual positioning of all elements. it is connected with the structure by means of classes and IDs. This file is not strictly semantic, but it is necessary to create a consistent and meaningful UI, and also one that is able to adapt to different display sizes. If very different targets are to be taken into consideration (laptop, tablet, smartphone) a layout file specific for each one of those can be written. The end user will have some limited control over the layout (resizing panels and similar actions)

3. STYLE (Es): describes the more stylistic aspects of the application: colors, fonts and all those visual characteristics that are not essential but can be very beneficial. The end user will have a lot of control over the styling part, being able to change size, fonts, colors to better fit her needs.

4.2.1 Em: structure

As previously mentioned, the structure is a hierarchical representation (tree) of all the elements in the interface. Every internal node of the tree represents a rectangular container, called **frame**, while all leaves are **components** (labels, buttons... see 5.4 for details). Frames are the building blocks of the interface: they group related components together, such as the text fields in a form, and can be in turn part of a bigger group. Frames are also the key elements in laying out the interface, as explained later (see 4.2.2 for details).

The end user has no control over this structure, it is defined by the programmer and is the same across all devices and users.

4.2.2 Ex: layout

The layout is specified assigning properties to the frames in the structure. These properties are applied using selectors (explained later in 6.2) and are a declarative expression of how the frames should behave. We do not, in general, specify exact positions and dimensions, but rather write general properties such as *expand horizontally* or *all buttons should have the same width*, using the available commands that will be explained in depth in a later chapter.

While the layout can adapt to changes in display size, it should be tailored for the target device category (laptop, tablet, phone..) in order to optimize the user experience (for example, switching from a 2 columns layout on a laptop to a single column one on a smartphone).

The user does not have direct control over the layout file, but certain properties can optionally be customized (panel sizes and positions).

4.2.3 Es: style

The style file is structured exactly as the layout file, assigning properties to frames and components using selectors. The difference is in the kind of properties, which in this case are explicit requests regarding the visual appearence of such elements. For example, we might write that the buttons have a *red background*, or that the font will be *helvetica*.

The end user will have a lot of control over the styling part, being able to change size, fonts, colors to better fit her needs.

4.3 Syntax

The following chapter will provide specifications and extensive examples of Em, Ex and Es. As explained before, these documents are declarative and can easily be represented by a text format such as JSON. In an actual implementation, JSON may or may not be used as an encoding format. Other formats (XML, custom binary formats..) could be used instead for various reasons, but for the scope of this thesis JSON is a suitable choice, since it easy to read and short. I will now provide a general introduction to JSON and how it will be used in the coming chapters.

Json is a simple data format, that was invented for the javascript language (JSON stands for javascript object notation). it is only able to represents hash tables and array, and variables are not typed.

Hash tables are contained in a pair of curly braces, and key/value pairs are written as `key: value, key: value`, using columns to separate keys from values and commas to separate pairs.

Arrays are contained in square brackets, and elements are separated by commas: `[element, element, element]`.

The following example is a JSON representation of a hash table with two keys (aKey and anotherKey). The first one points to an array of strings, while the second points to a number.

```
{
    "aKey": ["elements", "of", "the", "array"],
    "anotherKey": 34
}
```

To keep the examples in this document lightweight, a simplified syntax will be used where double quotes are optional (unless the string contains spaces, commas or other special characters). The examples becomes:

```
{
    aKey: [elements, of, the, array],
```

```
    anotherKey: 34
}
```

This is just a syntax to represent data, in each chapter additional information will be provide about how to encode Em elements and Ex, Es properties with JSON.

CHAPTER 5

EM: THE STRUCUTRE

5.1 Introduction

The Em document is the one specifying the hierarchical structure of the GUI, listing all the UI components that will be present. I also include its interactive and state properties and all the necessary information to connect to the Ex and Es files. More precisely, for each element contained in the Em file, we can specify:

- (required) **type** of the element (EmFrame, EmButton, . . .)
- **id** and **classes** for each element (used to connect with the Ex and Es files, see 6.2 for more details)
- enabled/disabled state for each available **event**
- **state variables** specific to the element (e.g.: text of a button)

Before proceeding any further, we will now introduce a syntax that will consistently be used to provide examples of Em code.

5.2 Em syntax

As explained in 4.3, we use JSON to encode Em elements. Each element will be a JSON hash table. The element type is defined in the *type* attribute. To specify the hierarchy, the *children* attribute is used (an array of children, between square brackets). This is a very simple example of an Em document that displays two buttons:

```
{type: EmFrame, id: myContainer, children: [
    {type: EmButton, id: okButton, text: "ok"},
    {type: EmButton, id: cancelButton, text: "cancel"}
]}
```

5.3 EmDocument

Before proceeding in explaining in details each available Em element, we will introduce EmDocument. This serves as a container for all other elements, and defines the whole interface. Also, three additional functions are performed by the EmDocument:

- contains links to Es and Ex files, which define layout and style for the EmDocument but are conceptually and physically separate files.
- provides the signature of the linked Es and Ex files.
- provides the signature of the content, generated by the server, used to verify the integrity of the EmDocument.

5.4 Em elements

Up to now we talked about the general structure and design of an Em, Ex and Es application, but to actually implement a working UI we need a few basic UI elements to interact with. We will now list all available Em components, this section will serve as a reference of all components, their properties and events. For each element we will specify:

type the type of the element

parent element the element from which it inherits (in the Object Oriented sense of inheritance). This means that all events and properties are inherited.

frame-like if the element is frame-like or not. Refer to chapter 6 for a definition of frame-like, since this is only relevant for layout purposes

properties a list of properties, with their valid values, default value and read-only state.

events a list of available events that the element can send to the remote server. For more details of events and user interaction, refer to chapter 8.

5.4.1 EmFrame

```
parent element: none
frame-like: true
properties: none
events: none
```

EmFrames are the main building block for laying out interfaces. They serve as generic rectangular containers, and all their children will be rendered inside their bounds. They do not have any particular property or event associated with them, and do not inherit from any other element. All their power comes from the layout and style properties that can be associated with them.

5.4.2 EmSection

```
parent element: EmFrame
frame-like: true
```

```
properties: (see table)

events: none
```

TABLE I: EmSection properties

name	type	default	read only
title	String	empty	NO
newScope	Boolean	False	NO
sectionNestingLevel	Integer	-	YES
sectionNumbering	Integer[]	-	YES

Text documents have been historically divided in parts hierarchically, with these parts having names such as chapters, sections, paragraphs, etc... These structural dividers have a few things in common:

- they have a title and a content
- they are nested in order (a subsection cannot be directly contained in a chapter, but only in a section). This implies that the type of divider can be inferred by the nesting level.

Considering these two factors, I will start defining a new Em element called **EmSection**. An EmSection has a title and a content. The type (chapter, section, document...) is not specified explicitly but inferred based on the nesting level.

```
{type: EmSection, title: "section title", children: [
    ...
]}
```

This concept is similar to the one of frames, being a hierarchical representation of the UI. However, frames are meant to be used for layout and the frame hierarchy is not directly relevant for the user. The section hierarchy, on the contrary, is explicit and very semantic, and users can interact with it easily since they are used to its presence. Certain interactions will be natural, for example the ability to collapse/expand sections or the presence of a table of contents where each line is a link to the corresponding section.

We should think of sections as an additional hierarchy, specified inside the main Em hierarchy, that highlights the important semantic relations between groups of elements. Sections could be used to divide text in the usual way, or to divide GUI areas (menu, toolbars, content area...), or more generally to do both. In fact, while we have been referring to sections as a way to organize and divide text, after some considerations we realize that there is no real problem in having sections work with other UI elements as well (such as frames, buttons etc...).

However, while an EmFrame and an EmSection are conceptually different, the second is actually extending (with additional semantics and properties) the first one. An EmSection is actually a subclass of an EmFrame, and can be used for layout in the same way. Therefore we can replace a normal EmFrame with an EmSection to add semantics, and

keep the same layout properties. Further explanations regarding the layout of EmFrame (and EmSection) will be given in chapter 6.

Sections are an optional hierarchical organization of the UI (both interactive and text based), specified by the programmer but accessible and comprehensible by the end user. This is clearly different from frames, which are mainly devoted to layout organization, and the two structures can naturally coexist in the same document.

5.4.3 Section scoping

In complex documents, we can imagine a UI which uses sections to be organized (similar to a what a sitemap is) and at the same time uses sections in a text area to organize text. These two things are clearly in conflict, since we want the sections in the text area to be restricted to the text area itself. That is why we introduce the concept of section scoping, with the attribute `newScope`, which has the effect of resetting the nesting level for its children:

```
{type: EmSection, title: "content area", children: [ /* level 1 */
    {type: EmSection, title: "text area", newScope: true, children: [ /* level 2 */
        {type: EmSection, title: "chapter 1", children: [...]} /* level 1 */
        {type: EmSection, title: "chapter 2", children: [...]} /* level 1 */
    ]}
    {type: EmSection, title: "menu area", children: [ /* level 2 */
        ...
    ]}
```

```
]}
```

Using section scoping, we can refer to the element nesting level as relative to its scope, which is the distance in terms of sections from the first EmSection ancestor having `newScope: true`. Section scoping will be useful mainly for two reasons: directly using the number representing the nesting level and applying layout and styles based on it.

5.4.4 Section numbering

It is often useful to auto generate text based on the section number. For example in LaTeXdocuments chapter are automatically numbered and the document structure is always consistent because it is computed and not entered manually.

To access a section number two properties are available over an EmSection element:

- `sectionNumbering [array[int]]` returns an array of integers, using the normal numbering for sections, starting from nesting level 1 up to the current section. An example would be `[3,2,2]`, which would represent what is usually written as `3.2.2` next to the section title in LaTeXdocuments.
- `sectionNestingLevel [int]` this is simply the length of the array above, and represents the section nesting level. The main purpose of this element is to allow easy styling of sections.

5.4.5 Section headers

Sections are, similarly to frames, invisible elements unless something is put inside them. Differently to sections in LaTeX, in Em the title is not displayed directly (it can be used to generate a table of contents, or accessed from other elements...). If we need to have a real section title (as we do when using sections in a traditional text environment) we can simply define it as its first child. Details on how this affects style definition will be provided in 7.3.

5.4.6 Advanced section behaviors

As previously hinted, sections are more than just titles and headers. They can give structure to the whole interface, both text and UI, and provide a natural way for users to interact with this structure.

Two main advanced interactions are available: the table of contents and collapsing / expanding of sections.

The **table of contents** of an Em document can be generated by visiting its section hierarchy. It will then be presented in a consistent way in the client, for example a nested menu which gives a quick view of the document structure and provides links to the corresponding sections. This behavior is not optional and such menu (or similar view) should always be present in all applications that use sections, as to enforce their semantic value for the user.

The **collapsing and expanding** of sections will, on the contrary, be optional and configurable by the property `userCollapsible: true/false (defaults to false)` and con-

trolled programmatically using the style property `sectionCollapsed: true/false` (more details in the style chapter at 7.3). When a section is collapsed, all its children except the first one will be hidden. It is required to use the first child as section header for the collapsing to operate correctly.

5.4.7 EmText

`parent element: none`

`frame-like: false`

`properties: (see table)`

`events: none`

TABLE II: EmText properties

name	type	default	read only
text	String	empty	NO

EmText is a generic text container which contains a single string. it is useful to associate styles to strings, and behaves similarly to the HTML `span` element, but restricted to containing only strings. It is not visually rendered unless property value of certain elements (such as EmLabel and EmParagraph), and it is not a frame-like component.

5.4.8 EmLabel

`parent element: none`

`frame-like: true`

`properties: (see table)`

`events: none`

TABLE III: EmLabel properties

name	type	default	read only
text	String	empty	NO
textElements	EmText[]	null	NO

Displays a single line of text, without wrapping. It behaves as an EmText, meaning it has a text property, but can also contain other EmText elements to provide inline formatting. The final output will be a concatenation of the string contained in `text` and the contained EmText elements. It is a frame-like component.

5.4.9 EmParagraph

`parent element: EmLabel`

`frame-like: true`

`properties: none`

```
events: none
```

Similar to an EmLabel but with one major difference: text can wrap to new lines. Also, EmParagraph support additional style attributes (line spacing etc..) as we will see in chapter 7. Since it inherits from EmLabel, we have access to the text and textElements properties (as defined in the EmLabel section).

5.4.10 EmButton

```
parent element: EmLabel

frame-like: true

properties: (see table)

events: selectionStart, selectionEnd, secondarySelectionStart,
        secondarySelectionEnd
```

TABLE IV: EmButton properties

name	type	default	read only
enabled	Boolean	True	NO
interactionState	NORMAL, PRESSED	NORMAL	YES

An EmLabel that additionally provides and manages click events. Clearly it is a frame like components, since it extends a frame like component (EmLabel). The user can interact

with it in various ways. On a computer, for example, the user can click, double click and right click. On a touch device, she can touch and long touch. In general, all these actions have something in common: they have a begin phase (click down or start touching) and then an end phase (click up or finish touching). This component behavior will depend on the device/client it is rendered by, but for the server application this won't make any difference because the server side events are predefined. We have 4 events available, 2 different interactions (primary, secondary) and two events for each one (start, end). it is the client responsibility to map these events to some kind of user interaction specific to that device.

The button can be enabled or disabled, and its interaction state can be accessed through the interactionState property (this is mainly useful for styling the button differently based on its state). The value of interactionState is PRESSED only after a user has started an interaction with the button, and she hasn't finished it yet (mouse down, touching..).

5.4.11 EmLink

```
parent element: EmLabel
frame-like: true
properties: (see table)
events: none
```

This component defines a link to another section in the document. Pressing it will make the interface go to the section having the id specified in the linkTo property, called the target section. This means that the target section, and all its ancestor sections, will be

TABLE V: EmLink properties

name	type	default	read only
enabled	Boolean	True	NO
linkTo	String		NO
interactionState	NORMAL, PRESSED	NORMAL	YES

expanded. Also, if the parent section to the target section has the style property oneChild-Expanded set to true, we can simulate a tab panel behavior (see Chapter on Es for more details).

It is a frame like components, since it extends a frame like component (EmLabel). The user can interact with it in various ways, like with the EmButton. The main difference from an EmButton is that EmLink does not have events, and therefore does not generate remote commands.

The button can be enabled or disabled, and its interaction state can be accessed through the interactionState property in the same way we can do with EmButton.

5.4.12 EmSelectableButton

```
parent element: EmButton

frame-like: true

properties: (see table)

events: selectedStateChanged(selectedState)
```

TABLE VI: EmSelectableButton properties

name	type	default	read only
selectedState	SELECTED, DESELECTED	DESELECTED	YES
nextSelectable	id: EmSelectableButton	null	NO
groupSelection	ONE, EXCLUSIVE, MULTIPLE	EXCLUSIVE	NO
stateChangingInteraction	SELECTION_START, SELECTION_END, SECONDARY_SELECTION_START, SECONDARY_SELECTION_END	SELECTION_END	NO

Extends EmButton adding a new state property, selectedState. The values it can take are the usual `SELECTED, DESELECTED`. The state changes when an interaction of type stateChangingInteraction is performed (defaults to SELECTION_END, which on a normal computer would be a click up). After the state is changed, an event selectedStateChanged is raised, which is dispatched along with the new selectedState.

An EmSelectableButton can be used for example as a checkbox, provided two different styles are associated with its two different states.

EmSelectableButtons can be chained, forming and ordered list, using the nextSelectable property (which can contain the id of another EmSelectableButton). In a chain, the selection behavior is different. It is specified in the first element of the chain with the `groupSelection` property, and can be one of the following:

EXCLUSIVE only one element in the chain can be selected, or none.

ONE same as EXCLUSIVE, but exactly one element has to be selected. If none is selected, the first one in the chain is automatically selected and the `selectedStateChanged` event is raised.

MULTIPLE zero or more elements can be selected. The effect of the `stateChangingInteraction` depends of the currently pressed modifier key:

mo modifier the element is selected, all others are deselected

SHIFT key all elements between the last selected one and the current one are selected

CTRL key selection for the current element is toggled

5.4.13 EmInput

```
parent element: EmLabel
frame-like: true
properties: (see table)
events: inputChanged
```

This component defines a textbox to be used for user input. it is a subclass of EmLabel, where the text property can be directly changed by the user by typing into the text box. This element behaves as you would expect a normal form text field to behave.

It is a frame like components, since it extends a frame like component (EmLabel). The EmInput can be enabled or disabled, and its interaction state can be accessed through the

TABLE VII: EmInput properties

name	type	default	read only
enabled	Boolean	True	NO
validation	String	.*	NO
validationState	PRISTINE, ERROR, VALID	PRISTINE	YES
interactionState	NORMAL, FOCUSED	NORMAL	YES

interactionState property in the same way we can do with EmButton, but in this case it can be NORMAL or FOCUSED, when the user is writing in the EmInput text box.

The validation property specifies the validation pattern, as a regular expression, that should be respected by the text property. If the text does not match the validation pattern, the validationState property is set to ERROR. This can be used to style the EmInput (for example with a red border) when the text is not valid. If the text is invalid, but the user hasn't typed yet (usually when the text is still empty), the validationState property is set to PRISTINE.

It has one event, inputChanged, which is raised every time the user changes the value of the text property (by typing in something). The event raised will contain, alongside the usual EmEvent data, a text property with the value of the EmInput text property (more details on events in the Interaction chapter).

5.4.14 EmMenu

`parent element: none`

```
frame-like: false

properties: none

events: none
```

TABLE VIII: EmMenu properties

name	type	default	read only
title	String	empty	NO

This component is a way to create menus in Em. Menus are a hierarchical collection of buttons. They could theoretically be created only using EmButton and some layout/style properties. However, since menus are ubiquitous and very useful components, the EmMenu element provides a simple way to define them.

Each menu is associated with an section. There can be only one menu element for each section, and it should be its child. The menu should contain buttons that perform commands related to that section.

Menus can be nested, to define the usual hierarchy we can find in menu bars. To specify the title of each submenu, and of the main menu, we use the title property. The leaves of the menu hierarchy are EmButton elements. They behave as usual buttons, with the only difference of being shown inside the menu structure.

Menus are all shown in a general menu bar (similar, for example, to the typical OSX menu bar). However, since we have (up to) one menu per section, we must show multiple menus in the menu bar. Only the menus corresponding to sections that are expanded will be shown, and they will be displayed in order, visiting the section tree breadth first (see the examples Chapter to better understand how menus can be used).

CHAPTER 6

EX: THE LAYOUT

6.1 Introduction

In the previous chapter we saw how the structure of the UI is written using Em. The nodes of the tree representing the structure, as we already illustrated, are EmFrame elements. These elements are special in the sense that they have a whole set of properties to lay them out.

Layout is therefore a process that takes frames (and **frame-like** elements, as we will explain later) annotated with layout properties (the focus of this chapter) and outputs, for each frame, its position and size.

Before going any further, we need to introduce an important concept: **selectors**. Selectors are the connection between the Em document and the Ex (and Es) document. They are a way to select a subset of elements in the structure tree, so that we can apply layout properties to them.

6.2 Selectors

Selectors are a **query syntax** that lets us select a subset of all the elements in the structure. It is, as everything else, encoded and expressed here in JSON. To be specific, a selector is a **JSON array**.

Selectors filter the elements based on their id, classes, type and, more in general, any property or child/parent relation.

A selector is an array because it is actually an array of sub selectors, where each sub selector is a hash table:

```
{selector: [{sub selector 1}{sub selector 2} ... {sub selector N}]}
```

To match, the first sub selector must match on an element and every successive sub selector must match on a child of the previously selected one, thus creating a chain. Each sub selector defines a selection (subset of elements in the tree). By default, the final selection will correspond to the selection of the last sub selector.

Each sub selector works in a very simple way: it is a hash table where for each key (property name) we specify a list of allowed values. All properties must match at least one of their allowed values. Here are a few examples of how selectors work, explaining also a few more advanced features.

all elements with class title:

```
[{class: title}]
```

all elements with class title OR class subtitle:

```
[{class: [title, subtitle]}]
```

all elements of type EmButton AND class formButton OR submitButton:

```
[{type: EmButton, class: [formButton, submitButton]}]
```

all elements of type EmButton AND with property checked = true:

```
[{type: EmButton, checked: true}]
```

all elements:

```
[{}]
```

As you can see, for every property a list of possible values is specified (OR). If any of the values matches, the selector is satisfied. If we want to match all values (AND) we can define an array instead of a single value. This example matches all elements of class (*EmButton* AND *selected*) OR *link*:

```
[{type: EmButton, class: [[button, selected], link]}]
```

Up to now we didn't chain any selector, this example matches all buttons inside (direct descendant) an element of class myForm:

```
[{class: myForm}{type: button}]
```

If we need to match indirect descendants, we can insert an empty selector in between the two elements, and give it the property `_limit: 0`, which means to match as many elements as possible. The default value for this property is 1 (matches exactly one element). We can also specify a range, in the form `[a, b]`, which defines the minimum and maximum number of elements to match. 0 always means infinite.

This example matches all buttons which have a descendant of class myForm. In this case, we search for buttons. Then we match their parents recursively against `{}` (empty

selector, matches everything) until we find an element that also matches the next rule, class: myForm.

```
[{class: myForm}{_limit: 0}{type: button}]
```

Lastly, we can select objects based on their **_position** at their level, which is equal to how many left siblings they have. Therefore if we want to select the first child of an element with id="myID" we can do:

```
[{id: myID}{_position: 0}]
```

Consistently with the `_limit` property, we can assign an array to the `_position` property in order to select a range of elements.

One last feature of selectors is the ability to include in the final selection the selection defined by any sub selector in the chain, not just the last one. This is done with the property **_select**, which is by default true for the last one and false for every other sub selector. If we want to select all frames that directly contain a button as child, we can do:

```
[{type: EmFrame, _select: true}{type: EmButton, _select: false}]
```

First, we selected the first sub selector, then we also remove from the selection the last sub selectors. As many sub selectors as needed can be added to the selection.

6.3 Ex document syntax

As usual, our syntax is JSON based. The Ex file is a JSON array that contains hash tables. Each hash table is a pair of properties to assign, an a selector to define the subset of Em elements to apply these properties to. This is an example of a complete Ex document:

```
[{
    selector: [{id: myButton}],
    value: {widthPolicy: FILL}
},{
    selector: [{type: EmFrame}],
    value: {minHeight: 120}
}]
```

6.4 rule precedence and cascading inheritance

The layout properties are **not cascading** (they only apply to the elements selected) and therefore if a property value is assigned to a frame, it will not propagate to its children.

To keep the layout files easy to understand and to parse, precedence rules are reduced to the simple order of declaration: declared later, higher precedence (overrides preceding rules). Therefore it is important to write rules from the most generic ones to the most specific ones. This also has the advantage of keeping the layout files clean and organized.

To determine which property value will be associated to an element, the client will, in order:

1. search for matching selectors, respecting precedence as previously defined
2. apply the property value if the previous search is successful, otherwise use the default value

6.5 Frames and frame-like components

Frames (`EmFrame`) are the main element used for layout. They are rectangular, transparent (by default) containers that can be both laid out and provide layout rules for their children elements. To better explain, a frame can have attributes that specify its size (`width: 100`) or attributes that specify how elements behave inside it (`flowDirection: VERTICAL`).

Frame-like components can be laid out just like frames, but they cannot contain any other element (they are leaves in the structure tree). Therefore they are compatible with all those properties that are not related to layout of inner elements. Frame-like components are almost all UI controls: buttons, labels, ... In 6.8, for each property, it will be specified if it applies only to frames or, more in general, to all frame-like components.

6.6 Sections

As explained in 5.4.2, EmSection is a subclass of EmFrame that adds semantic values and additional behavior to the EmFrame element. Therefore every time we talk about properties compatible with an EmFrame, we imply (consistently with Object Oriented concepts) that they also apply to EmSection elements.

6.7 Units of measurement

In the following section, we will list and explain in depth every available layout and style property. However, before proceeding, a short explanation of our units of measurement is necessary.

6.7.1 Sizes

Size (font size, frame heights..) is usually measured in pixels. This however is dependent of the dpi (pixel density) of the device in use. We can use a density independent pixel (by choosing a standard density, say 160 dpi). This is what has been done, for instance, in the Android project, where the `dip` unit is used. However, it is still dependent on the average distance of the user from the screen. On a mobile phone things are supposed to be smaller, because we look at it more closely than a tablet, laptop, desktop monitor, television and so on..

In Es, sizes are measured with a standard pixel, with defined dpi and average distance (160 dpi and 30 inches), so that each device is then able to adapt the interface in a consistent and predictable way.

We can imagine the user adjusting her preferred viewing distance, and having the UI adapt to accommodated for different preferences.

Today this approach has one minor drawback, the loss of pixel perfection in the rendering. This can result in less sharp, blurry interfaces. This problem is however less and less relevant, as average pixel density is increasing rapidly and will not be a problem anymore in the coming years.

6.7.2 Coordinates

Coordinates in Es will always be given as `X, Y`, with X being the horizontal distance from the left, and Y the vertical distance from the top. Therefore the `0, 0` coordinate will indicate the top-left corner.

No absolute coordinates exist, as we always refer to the current layout scope (parent Em element, which is always a frame). This is necessary to have a portable and adaptive interface, independent of the current display, and generates much more reusable code.

6.8 Layout properties

Here are listed all the available layout properties. These properties can be applied to every frame or frame-like element. Frame-like elements are those rectangular UI components that behave like frames, with the only major exception that cannot contain children. Examples are buttons, text fields etc. . .

For each property, the value type is specified (either *double* or a list of CONSTANTS). Next, the compatible elements are specified. If *frame-like*, it means frames and frame-like elements. When only *frames* is written, it means that the property is meant to affect the element children and therefore it is not compatible with frame-like elements.

width (double) (frame-like) [no default] width

height (double) (frame-like) [no default] height

x (double) (frame-like) [no default] x value of the top-left corner

y (double) (frame-like) [no default] y value of the top-left corner

minWidth (double) (frame-like) [0] the minimum width of the element.

maxWidth (double) (frame-like) [MAX_WIDTH] the maximum width of the element.

widthPolicy (WRAP, FILL) (frame-like) [WRAP] WRAP means wide enough con contain everything. FILL means as wide as possible, respecting widthWeight. Min and Max

width constraint are always respected, therefore by specifying the same max and min width, we can enforce a fixed width (FILL or WRAP does not matter in this case).

widthWeight (double) (frame-like) [1] weight of this element as a filler. If two elements are on the same row and have widthPolicy=FILL, then they will fill proportionally to their weight.

minHeight (double) (frame-like) [0] the minimum height of the element.

maxHeight (double) (frame-like) [MAX_HEIGHT] the maximum height of the element.

heightPolicy (WRAP, FILL) (frame-like) [WRAP] WRAP means high enough con contain everything. FILL means as high as possible, respecting heightWeight. Min and Max height constraint are always respected, therefore by specifying the same max and min height, we can enforce a fixed height (FILL or WRAP does not matter in this case).

heightWeight (double) (frame-like) [1] weight of this element as a filler. If two elements are on the same column and have heightPolicy=FILL, then they will fill proportionally to their weight.

margins only the left margin is explained. Top, bottom and right have similar properties.

minMarginLeft (double) (frame-like) [0] the distance the element keeps to whatever is on its left side. The distance is kept outside the element.

marginLeftPolicy (FIXED, EXPAND) (frame-like) [FIXED] if it is FIXED, then the minMarginLeft value is used. If it is EXPAND then that value is treated as a minimum but will be expanded as far as possible.

marginLeftWeight (double) (frame-like) [1] if marginLeftPolicy is EXPAND, this is the weight of expansion on the expansion line. This is similar to the widthWeight concept.

flow This is a complex but powerful and generic concept. It specifies how elements inside a frame are supposed to flow and how many elements should be in each row/column.

flowLimitVertical (integer) (frames) [0] specifies how many elements can be stacked together vertically at most. Use 0 to remove limit.

flowLimitHorizontal (integer) (frames) [0] same for horizontal.

verticalFlowWrapping (boolean) (frames) [true] if true, the flow of elements wraps inside the container, meaning that it goes to a new line if it reaches the side.

horizontalFlowWrapping (boolean) (frames) [true] same for horizontal.

flowOrigin (TOP_LEFT, TOP_RIGHT, BOTTOM_LEFT, BOTTOM_RIGHT) (frames) [TOP_LEFT] the corner the flow starts form. As we can see in image 6, the first one from the left has direction=HORIZONTAL, origin=TOP_RIGHT, verticalFlow=0, horizontalFlow=3. If we resize the container so that child 3 is out, we obtain the second situation. The figure has horizontalFlowWrapping=false,

and child 3 is not displayed. The last figure has horizontalFlowWrapping=true and therefore child 5 is out.

flowDirection (VERTICAL, HORIZONTAL) [HORIZONTAL] the direction of the flow.

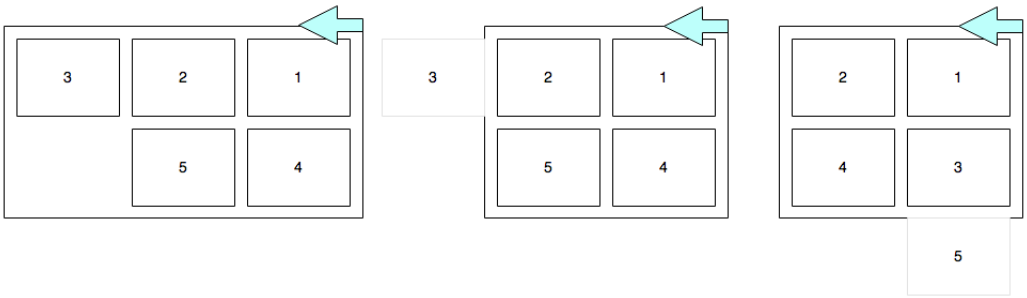

Figure 6: Example of a container with 5 children

padding also known as inset or internal margin. Simply a double value for each side. IMPORTANT: if padding is applied on the left or top side, then the origin of the coordinates inside this element are shifted by the padding. This is consistent with the box model definition given previously. Also, the effective internal available height and width of the element are reduced. Only left padding is listed here, similar properties exist for right, top and bottom padding.

leftPadding (double) (frames) [0] padding value for the left side.

overflow if the content of the frame overflows, this sets the policy to apply.

verticalOverflowPolicy (SCROLL, HIDDEN) [SCROLL] Should be self explanatory. It must be noted that this policy is applied after all other layout calculations are performed, the overflow policy does not change the layout in any way.

horizontalOverflowPolicy (SCROLL, HIDDEN) [SCROLL] same for horizontal.

user resizable this property makes the border of the container resizable by users. This operation is performed by dragging the border to its new desired position.

leftResizable (boolean) (frames) [false] whether the left border can be dragged to resize the frame.

rightResizable (boolean) (frames) [false] whether the right border can be dragged to resize the frame.

topResizable (boolean) (frames) [false] whether the top border can be dragged to resize the frame.

bottomResizable (boolean) (frames) [false] whether the bottom border can be dragged to resize the frame.

6.8.1 Constraints on layout attributes

Up to now, we only described layout attributes without the possibility of an attribute depending on others. There are times where, for example, the width of a button depends on the width of other buttons (so that they may all be the same), or the height of cells in

a table is related to each other (so that cells are all aligned). To achieve this we introduce the concept of constraints and groups.

To work with groups and constraints we use two special attributes

inGroup (string) [none] a group is a set of attributes that are linked together by a common group name. They must all have the same type (integer, double, rgb color etc..). When an attribute is part of a group, it contributes its value (if specified or computable) to the group itself.

aggregationFunction (min(), max(), avg()) [max()] after having populated the group with various attributes, we can use the group to set the value of compatible (same type) attributes. The aggregate function is a function that takes a set of values (all those contained in the group) and outputs only one value.

6.8.1.1 Examples of groups

The simplest example is the master/follower one. One element sets an attribute value and other elements follow it, using the same value. In this case the label follows the width of the button:

```
Em:
    [{type: button, id: myButton},
    {type: label, id: myLabel}]
Es:
    [{
```

```
        selector: [{id: myButton}],
        value: {width: [20, inGroup("my_group")]}
    },
    {
        selector: [{id: myLabel}],
        value: {width: min("my_group")}
    }]
```

In this other example we use the max function. We have 3 buttons on a row and we want them to be the same width, therefore we have to chose the max width. We do not set the width explicitly, it'll be set when the contentWidth is computed. We can omit the aggregation function because `max()` is the default function. We have to both set the inGroup attribute and use the group, because in this case the elements both take part in the group and get their values from it.

```
Em:
    [{type: EmButton, id: myButton1, class: buttons},
    {type: EmButton, id: myButton2, class: buttons},
    {type: EmButton, id: myButton3, class: buttons}]
Es:
    [{
        selector: [{class: buttons}],
        value: {width: ["my_group", inGroup("my_group")]}
```

```
}]
```

CHAPTER 7

ES: THE STYLE

7.1 Introduction

The style file (Es) is the last component we need to specify a complete interface. It is not mandatory (just like the layout file) and, when not provided, each element will simply have all its properties initialized to their default values. The complete list of properties and their defaults will be provided later in the chapter.

We will not go over the syntax and rules precedence of the style file because the same rules that were illustrated for the layout file still apply here, with one major difference: style properties are **cascading**.

As mentioned before, a property is cascading if, by default, it inherits its value from its ancestors. With cascading properties we can rewrite the procedure (see 6.4) the client will follow to determine the final value of a property at runtime:

1. search for matching selectors, respecting precedence as previously defined
2. apply the property value if the previous search is successful, otherwise use the value provided by the parent element
3. if no parent element is defined, use the default value for that property

7.2 Units of measurement

We will now introduce the units we need to specify style properties. We already defined sizes and coordinates in the layout chapter, and they are still valid for compatible properties in the style file.

7.2.1 Colors

Colors are specified as `RGB` (red, green and blue) hexadecimal values, just like in CSS and many other visual environments. For example, the red color is written as `#ff0000`. Semitransparent colors are not allowed, in the philosophy of keeping the interface clean and understandable, avoiding complex visual effects.

7.2.2 Style properties

Here are listed all the available style properties. These properties can be applied to all elements, and it will be the the element responsibility to use or ignore the property value appropriately. For example, a text label will probably care about text related style properties, while an image container would ignore those property (since it does not have any text). However it will still propagate them to its children, therefore a label that is child of the image container would use them to style its text rendering.

text and font related styles :

color (rgb color) [#000000] text color

font (font name) [opensans] text font, to be chosen from a fixed list of compatible fonts

fontSize (double) [12] font size

fontWeight (LIGHTER, LIGHT, NORMAL, BOLD, BOLDER) [NORMAL] font weight

fontStyle (NORMAL, ITALIC) [NORMAL] self explanatory

fontDecoration (NONE, UNDERLINED) [NONE] self explanatory

wordSpacing (double) [1] space between words. 1 is normal spacing, 2 is double that, 0.5 half that..

letterSpacing (double) [1] space between letters

frame-like elements related style :

backgroundColor (rgb color) [#000000] background color

borderWidthLeft (double) [0] width of the border stroke. The border is not part of the box model, and will be rendered centered between margin and padding.

borderColorLeft (rgb color) [#000000] border color

borderStyleLeft (SOLID, DASHED) [SOLID] border style

similar border properties exist for right, top and bottom borders.

section related styles (sections are explained later) :

userCollapsible (Boolean) [true] whether a section can be collapsed by the user or not

oneChildExpanded (Boolean) [false] whether a section must have one and only one subsection expanded at all times

collapsed (Boolean) [false] whether a section is collapsed or not. When collapsed, all children except the first one are hidden.

7.3 Styling sections

As explained in 5.4.2, sections are transparent containers that behave like frames. They can be used for layout and can be styled (with borders, backgrounds. . .). However, there is one particular situation for which additional considerations are in order: section headers. As explained before, sections to not display a header (title) anywhere in the UI (the `title` property is used only to generate the table of contents). The first child of an EmSection element can be used as header (meaning it will not become invisible when collapsing the section). Therefore, if we want to style section headers, we will then need to define an appropriate selector to style the first child of every matching section.

Section styling is based on the nesting level, relative to the current scope. To style a section we need to select it based on its nesting level and specific scope, and then select its first child. Here is an example:

```
[{
    selector: [{id: sectionHead}{_limit: 0}
               {type: EmSection, nestingLevel: 1}{_position: 0}],
    value: {color: #ef41b7}
}]
```

In this example, we first define the selection scope (by limiting the selector to all children of the `sectionHead` element). Then we match all elements of type `EmSection` with `nestingLevel = 1` (which are the top level sections) and restrict the selection to their respective first children using the `_position` attribute. Finally, a style property (`color` in this case) is specified and applied to all matching section headers.

CHAPTER 8

INTERACTION

8.1 Introduction

As mentioned in chapter 3, the user can interact with the GUI (using mouse clicks, touches and keyboards) and each of these interactions results in an event being raised and sent to the server, which in return replies with an edit command to update the GUI.

Events are predefined, each component has a set of available events that are always off by default. For a complete list of available events for each Em element, refer to chapter 5. The programmer can selectively enable relevant events. If an element has at least one enabled event, it is required to have an id so that, when the event is raised, the client can communicate to the server which element raised it.

8.2 User interaction: EmEvent and EmCommand

We will now explain in more details events and edit commands, using a simple *hello world* example to illustrate the concept. In the example we have two buttons, one saying ẅriteäand the other d̈estroy.̈ Pressing the first one will write ḧello worldïn a label, while the second one will eliminate the label all together.

First, we define the button and label elements, enabling the onClick events:

```
{type: EmFrame, children: [
    {type: EmButton, id: writeButton,   text: "write",   onClick: true},
```

```
    {type: EmButton, id: destroyButton, text: "destroy", onClick: true},
    {type: EmLabel,  id: label1,        text: "initial text"}
]}
```

When the user clicks the write button, a message is sent to the server containing the event name (and optionally additional informations regarding how the event was performed), the time of execution (as milliseconds since epoch) and the id of the element that raised it. This information constitutes the *EmEvent* object:

```
{type: EmEvent, elementId: writeButton,
 eventName: onClick, time: 1366934712412}
```

When the server receives this message containing an EmEvent, it can decide to do one of two things: ignore the event or reply with an *EmCommand* to modify the rendered interface. The following EmCommand updates the interface by replacing the value of a property on a selected element. In this case, we change the text property of the EmLabel element with id label1:

```
{type: EmCommand, commandType: update, selector:[{id: label1}],
 data: {text: "hello world"}}
```

When the client renderer receives this EmCommand, it parses it and recognize it to be an **update command** based on the commandType field. It then and applies the edit contained in data to all elements matching the provided selector. The data field is a simple object that specifies, for each property names, its new value.

The destroy button will behave in a similar way, sending the following EmEvent when clicked:

```
{type: EmEvent, elementId: destroyButton,
 eventName: onClick, time: 1366915243434}
```

The server will respond with a **delete command**, which is similar to the update command but without the data field:

```
{type: EmCommand, commandType: delete, selector:[{id: label1}]}
```

As it did for the update command, the client receives this EmCommand and applies it, deleting all elements matching the selector and all their children.

There is one last type of command, the **create command**, which is used to insert new elements in the structure tree. This command is specified as follows:

```
{type: EmCommand, commandType: create, selector:[{id: label1}],
 position: before,
 data: {...the element to insert...}
}
```

The already specified syntax is used, with a commandType of create. A combination of selector and position field is used to specify where to insert the element contained in the data field. First, the selector identifies a set of elements. Then the position property is used to specify the insertion point relative to these elements. It can be one of the following:

before inserts as left sibling

after inserts as right sibling

firstChild inserts as first child

lastChild inserts as last child

When the selector matches more than one element, the new element is duplicated and inserted at every location, unless it contains a unique id, in which case an error is raised.

Based on these three simple types of EmCommand, all edits to the interface can be performed.

CHAPTER 9

SECURITY BY DESIGN

9.1 Introduction

This project is part of the Ethos OS, which is characterized by its focus on security by design. The goal is to minimize security risks by changing the design principles themselves, making various attacks not possible.

The situation can be interpreted as the typical trusted path problem in secure communications, where we include the user interface as the last element in such path. Also, we should not limit our analysis to technical exploits at different levels of this path, but also include social engineering attacks, where both naive and tech savvy users can be, for instance, fooled into clicking on malicious buttons or giving personal information.

In this chapter we will quickly go through a list of possible attacks that are performed against applications with a remotely rendered GUI and see how we invalidate these exploits with our framework. **However, we should note that the greatest effort in the design of Em, Es and Ex was put on simplicity and minimality of both implementation code and concepts.** This is because we believe that most security risks in modern UIs arise from their excessive complexity, and therefore we tried to minimize such complexity while maintaining most of the features required to develop a functional interactive GUI application.

This being said, we do not pretend to be exhaustive in the following list, or to analyze these threats in depth, but to provide a general overview and a possible starting point for further analysis, when the implementation will be more solid.

9.1.1 Trust and security risks

Key elements that should be considered as part of the trusted path are the developer, the server, the communication channel, the UI rendering engine and the user machine. In this first list we show two possible attacks that should be prevented, even though they are not directly solved by the Em design, pointing to existing solutions (if present) to the problem.

false server identity The situation in which a server pretends to be the legitimate server and provides a similar UI (phishing). Approaches to solve this problem are based on private/public key matching and have either to be managed by an authority (as it is done in SSL and HTTPS), or be on a case by case acceptance of the server.

connection channel is not trusted Since we receive remote updates to our UI, we should also verify the authenticity of these updates. This problem, being similar to the previous one, can be tackled with the same methodologies.

In general, these risks are not currently completely solved. This is because there is no way to establish the authenticity of the first connection if we assume the attacker has access to our client machine (or the network) since our first attempt of connection. We will focus more on those aspects of security that are directly correlated with GUIs, or that the Em language can solve thanks to its design. The considered situations are:

malicious intentions of the developer Even on a trusted server, the application might be malicious. For instance, the application might be intended to do harm to the user machine (DoS or virus injection), or have access to its data. In such a scenario, the security risk is mitigated by a trusted UI rendering engine. Not having client side code, but only a minimal interaction logic provided by the framework, is already a big step towards solving this problem.

visual spoofing of the client This attack is performed drawing components similar to those of the UI renderer (chrome elements such as a status bar, and address bar or menus) and hiding the original ones so that the user is fooled into interacting with these false replacement. A possible solution here is the inability to edit the chrome of the UI renderer, and to standardize its appearance as much as possible.

internal visual spoofing This attack consists in creating confusing interfaces that lead the user into performing unintended actions. Examples of internal visual spoofing are visually similar components or symbols, elements that are superimposed or partially overlapping, colors that are difficult to distinguish or that hide components' borders. Possible solutions revolve around limiting the visual expressiveness of the language, for instance limiting the available fonts (to avoid similar symbols), not allowing overlapping of interactive components and enforcing certain visual clues to clearly delimit functional areas of the GUI.

9.1.2 Em design and security

We will now briefly see how Em and its environment can help solve these issues:

9.1.2.0.1 Em renderer

The renderer is the application that receives the Em, Ex and Es documents and renders the interactive UI. Since Em does not allow for client side code, the renderer is responsible for implementing all the necessary interaction logic. If this code is secure, and no other custom logic can be executed, we minimize the risk of malicious applications. Unless exploitable bugs in the renderer are present, it will be impossible to do harm to the client machine, and for this reason it plays a central role in securing the GUI application. Another important security related aspect to consider is the renderer's chrome. The chrome is the set of all UI elements that are not part of the rendered UI. In web browser, for example, the chrome typically includes the address bar, status bar and the browser menus. The Em renderer should have a distinct separation of the chrome from the rendered application area, and give no way for the developer to change the appearance of the chrome itself. In this way, we can avoid visual spoofing attacks such as reproducing a fake address bar, or status bar, with false information.

9.1.2.0.2 Em, Ex and Es

In addition to what already explain in the previous paragraph, the Em, Ex and Es languages have two other properties that were designed to optimize security, in particular avoid visual spoofing and confusion of the user by complex UI.

no overlapping containers no overlapping is allowed by Ex, therefore no click through attacks are possible. Also in this way we avoid confusing UIs that use overlapping and

pop-up windows to mask their real functions or spoof other typical OS UI elements, such as a system wide confirmation windows.

limited interactions since interactions are limited to a predefined set, no special behaviors that might be confusing can be implemented. In particular, interactions are only clicks on buttons, dragging to resize frame borders and typing in text input fields or triggering commands through keyboard shortcuts.

CHAPTER 10

PROTOTYPE IMPLEMENTATION

10.1 Introduction

To validate the design suggested in this thesis, a prototype implementation has been realized. The implementation, while not able to cover all presented features, tries to prove the feasibility of the design by implementing a subset that is significant. Also, the prototype does not communicate for now with the Ethos OS. However, since the communication protocol is generic and implemented in JSON, this is not a limitation.

The main technologies used for the prototype are:

Javascript The main programming language used is Javascript. The choice is forced, since this is the only language used for browser based development. Also, being a dynamic and flexible language, it proved as a good choice for a prototyping environment.

AngularJs The AngularJs framework ([12]), an open source Javascript framework developed by Google, has been used to implement the Em components, thanks to its modular design which fits well with the Em language.

JSON JSON is a simple data interchange format, and it has been used to encode the Em elements. More details on how JSON is used are given in section 4.3.

Bootstrap Bootstrap is a CSS framework that was used to streamline the graphical development of the basic UI components.

Node To simulate the server, which in a real implementation would run in the Ethos environment, we used a simple web server written in *Node.js*. Node.js is a server side Javascript interpreter ([13]). It is used with the *Connect* library as the HTTP middleware framework ([14]).

10.2 Demo Application

The prototype was developed to provide all the necessary features to implement a demo application. This application is a document viewer able to show Em documents. A screenshot of the application is visible in figure 7. The application is structured in 3 panels:

list of documents On the left we have the list of documents. The title of each document is show, and it is rendered in bold if the document is unread. Each document can be opened pressing the *open* button. Also, multiple documents can be selected for bulk operations with the checkboxes on the left. The only bulk operation available is *mark selected as read*, which is show in the main menu.

document view On the bottom right, we have the actual document being displayed. This is a generic Em document, a few textual examples are provided. Also, since Em can intermix textual and interactive components, the documents displayed can provide interactions, such as buttons and forms.

document control On the top right, we have the document controls. On the left, the tagging control lets us add a tag to each document, that is then displayed in blue in

the list underneath it. On the right, two buttons let us mark a document as read, or as important, as shown in figure 8.

Also, the standard top menu bar is present. As explained in section 5.4.14, the actions displayed in the menu are defined using the EmMenu component inside a section. In the demo application, we have two EmMenu instances:

list menu the menu relative to the list of documents section, it contains the *mark selected as read* action.

document menu the menu relative to the document control section, it contains the *mark as read* and *mark as important* actions.

10.3 Features implemented

To develop a functional demo application, a subset of all the Em, Ex and Es features has been implemented and it is here described.

Em :

EmDocument basic functionality including Es and Ex files linking

EmFrame all functionality

EmSection basic functionality including section title

EmText all functionality

EmLabel all functionality

EmMenu all functionality

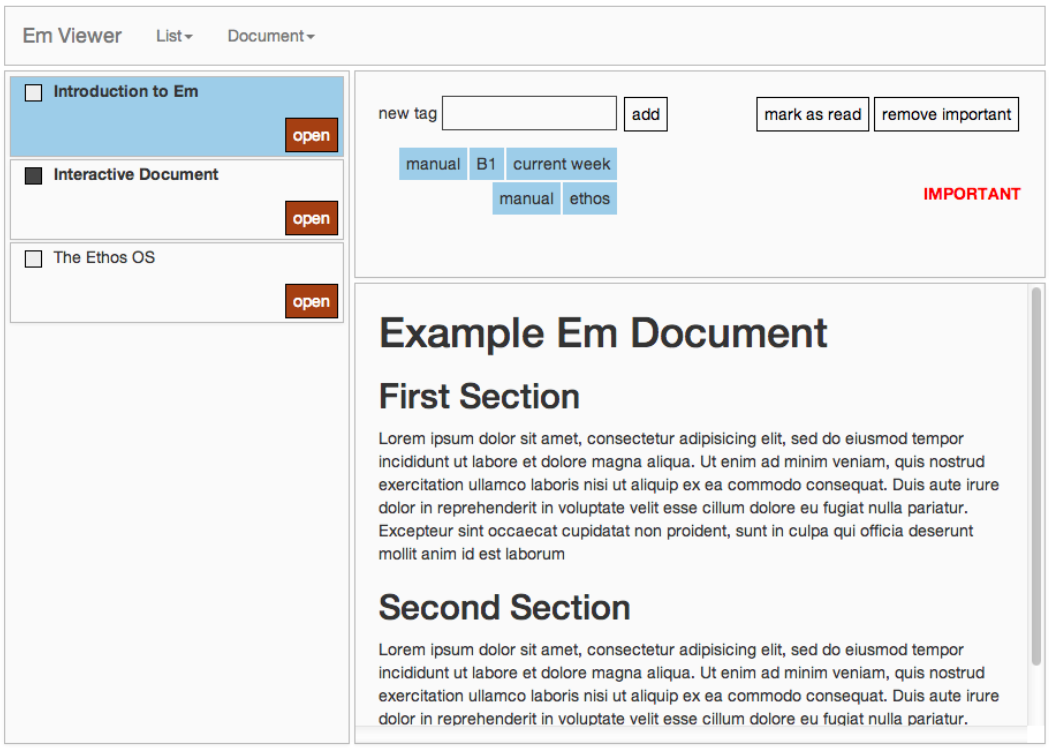

Figure 7: demo application

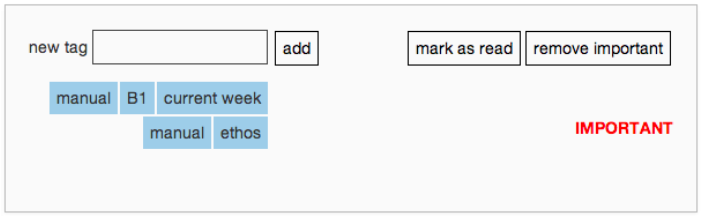

Figure 8: mark as important control

EmButton basic functionality including primary interaction

EmSelectableButton all functionality

EmLink basic functionality including linkTo property

EmInput basic functionality including text input

Ex :

selectors basic selectors by type and class

layout complete implementation of the layout engine

Es :

selectors basic selectors by type and class

text and font style color, size and weight are implemented

frame-like styles background color, border style and width are implemented

section styles collapsed and oneChildExpanded properties are implemented

10.4 Results and future work

Even though the demo application was not of high complexity, the prototype was successful in showing the feasibility of the current Em, Ex and Es design. The technological choices made have proven to be adequate for the task at hand. In particular, the Angular.js framework has been a key component thanks to its modular design and its directive system. Directives are self contained components which were used to implement each Em element. Further work is needed to complete the validation of every aspect of the

design, and to move towards a stable implementation. In particular, the next step will be to interface the prototype with a server running under the Ethos operating system.

CHAPTER 11

CONCLUSIONS

11.1 Conclusions

The objective this thesis had, as explained in chapter 1, was to lay the foundation for a new graphical user interface for the Ethos operating system, following the philosophy of that project. The final design, as tested with the prototype implementation, was able to satisfy in a preliminary way the requirements we set. However, validation of the chosen approach is necessary to further judge its quality and efficacy. In addition to continuing the development of Em, Ex and Es as languages, and the implementation of a stable renderer, it will be necessary to work on the integration, at the implementation level, of this work with the Ethos operating system.

CITED LITERATURE

1. Nardi, B. A. and Zarmer, C. L.: Beyond models and metaphors: Visual formalisms in user interface design. In System Sciences, 1991. Proceedings of the Twenty-Fourth Annual Hawaii International Conference on, volume 2, pages 478–493. IEEE, 1991.
2. Pirhonen, A., Brewster, S., and Holguin, C.: Gestural and audio metaphors as a means of control for mobile devices. In Proceedings of the SIGCHI conference on Human factors in computing systems, pages 291–298. ACM, 2002.
3. Nichols, J. and Faulring, A.: Automatic interface generation and future user interface tools. In ACM CHI 2005 Workshop on The Future of User Interface Design Tools, 2005.
4. Schlungbaum, E. and Elwert, T.: Automatic user interface generation from declarative models. In CADUI, volume 96, pages 3–17. Citeseer, 1996.
5. Gajos, K. and Weld, D. S.: Supple: automatically generating user interfaces. In Proceedings of the 9th international conference on Intelligent user interfaces, pages 93–100. ACM, 2004.
6. Vanderdonckt, J.: Knowledge-based systems for automated user interface generation: the trident experience. In Proceedings of the CHI, volume 95. Citeseer, 1995.
7. Puerta, A. R.: A model-based interface development environment. Software, IEEE, 14(4):40–47, 1997.
8. Johnston, J., Eloff, J. H., and Labuschagne, L.: Security and human computer interfaces. Computers & Security, 22(8):675–684, 2003.
9. Epstein, J., McHugh, J., Pascale, R., Orman, H., Benson, G., Martin, C., Marmor-Squires, A., Danner, B., and Branstad, M.: A prototype b3 trusted x window system. In Computer Security Applications Conference, 1991. Proceedings., Seventh Annual, pages 44–55. IEEE, 1991.
10. Ye, Z. E., Smith, S., and Anthony, D.: Trusted paths for browsers. ACM Transactions on Information and System Security (TISSEC), 8(2):153–186, 2005.
11. Chen, S., Meseguer, J., Sasse, R., Wang, H. J., and Wang, Y.-M.: A systematic approach to uncover security flaws in gui logic. In Security and Privacy, 2007. SP'07. IEEE Symposium on, pages 71–85. IEEE, 2007.
12. Angular.js. `http://angularjs.org/`, 2014.
13. Node.js. `http://nodejs.org/`, 2014.
14. Connect. `http://www.senchalabs.org/connect/`, 2014.

15. Schilit, B., Adams, N., and Want, R.: Context-aware computing applications. In Mobile Computing Systems and Applications, 1994. WMCSA 1994. First Workshop on, pages 85–90. IEEE, 1994.

16. Bootstrap. `http://getbootstrap.com/`, 2014.

17. Android. `http://www.android.com/`, 2014.

VITA

Luca Cioria

Education

B.S., Engineering of Computing Systems
Politecnico di Milano
2011

M.S., Computer Science (*current*)
University of Illinois at Chicago, Chicago, IL
2014

Alta Scuola Politecnica diploma (*expected*)
Politecnico di Milano, Politecnico di Torino
2014