

Universiteit Antwerpen
Departement Wiskunde en Informatica



Performance Analysis of Telecommunication Systems

Adaptive Graphic User Interface

Author: Héctor Toll Vallejo

Supervisor: Vincenzo De Florio

Date: September 2011

A word of thanks

I would like to thank my supervisor Vincenzo De Florio. Without his advice, support and encouragement I would never have realized this thesis.

Table of Contents:

1 Introduction	7
1.1 Motivation	7
1.2 Goals	8
1.3 Text structure.....	9
2 Autonomic computing and QoE.....	11
2.1 Autonomic computing	11
2.1.1 Definition	11
2.1.2 Model	13
2.1.3 Characteristics.....	15
2.1.4 Benefits	16
2.2 Quality of Experience.....	17
3 Problem analysis and solutions.....	19
3.1 Problem description.....	19
3.2 Problem analysis	21
3.3 Our solution	25
3.3.1 First Design.....	25
3.3.2 Encountered problems	30
4 Final implementation.....	33
4.1 Tools used	33
4.2 GUI's Hierarchy.....	34
4.3 Classes	37
4.4 Drawing elements.....	48
4.5 Saving/Loading the GUI	49
4.6 Modifying the GUI.....	51
4.7 Quality of Experience Rules	52
4.7.1 Recording background actions.....	53
4.7.2 First QoE failure detection: Slider.....	54
4.7.3 Second QoE failure detection: Menues	56
4.7.4 Third QoE failure detection: Email.....	57
4.7.5 Fourth QoE failure detection: Date format	60

4.7.6 Fifth QoE failure detection: Date order	61
4.7.7 Sixth QoE failure detection: Text Area	63
4.7.8 Seventh QoE failure detection: Clean button.....	64
4.7.9 Eighth QoE failure detection: Validate button	66
5 Evaluation	67
5.1 The system.....	67
5.2 Experiments and their statistics	69
5.2.1 First subject: Experienced user	69
5.2.2 Second subject: Not experienced user	71
5.2.3 About the results	73
6 Related work	74
7 Futur work and Conclusion	76
7.1 Future work.....	76
7.2 Conclusion	77
8 Bibliography	79

1 Introduction

1.1 Motivation

Nowadays when we use commercially available user interfaces, e.g. those typical of a Smartphone, a PC, laptop or any other, communication takes place between the user side and the computer side: at the one end, e.g. keys are being pressed, screen portions are being touched, words are typed in, text fields are filled, options are selected, etc — at the other end, screens are being filled in with widgets, images and textboxes, sounds are emitted, etc. We focus here on the user-to-computer flow of actions, and observe that a large portion of these actions is simply ignored by computers.

Some UI, like for Google search, are anticipative, thus they try to predict what the user is expecting (in this case, what he or she is looking for), but the general approach with UI's is simply to ignore whatever user action is not associated with a function of the underlying system. By doing so the computer side deliberately chooses not to take into account most of the user context. This means we do not know how comfortable the computer experience is, how adequate the UI is to the current user and how fluent the person in command is in the conversation with the computer system he or she is using.

We want to highlight the importance to take into account all these actions and we also think that no system should be an immutable entity focused only in its main functionalities. We believe that taking into account just a very little portion of the actions taken by the user (thus wasting not too many resources) we can improve considerably the user's experience. Furthermore the current computers have become powerful machines, flexible and intelligent agents able to

enhance our lives, for which reason the general approach should be fully exploit their potential an capabilities.

It is for these reasons that our approach is to create a systematic way to address these problems. Since it is unfeasible to register all actions generated by users due to the large variety of possible input events, we propose to register a part of the domain input (e.g. number of clicks on a certain button, number of times a menu is selected and so on) that the user may generate while using a GUI in order to change automatically those aspects of the User Interface that could be improved for a better experience of that user.



Figure 1.1.1: Example of a User Interface.

1.2 Goals

Before starting this thesis we set a group of objectives. These objectives are:

General

- Demonstrate it is feasible to enhance the user's experience by taking into account some user actions typically ignored.

- Demonstrate through a scientific methodology how this improvement is possible.
- Develop this work as a proof of concepts.

Particular

- Create a group of simple QoE rules in order to show the validity of our concepts.
- Create an adaptive Graphical User Interface in order to show the usefulness of well applied QoE rules.

1.3 Text structure

This text has been structured in order to be suitable for the reader. That means that first, we introduce the reader to the theoretical terms and concepts, then we analyse the main problem and its possible solutions, we continue explaining the most important implementation parts, after that we evaluate and test the final result and finally we state lessons learned, conclusions and possible improvements.

The sections description is:

- **Theoretical study** (*section 2*): Prior to any analysis or implementation is important to understand the different technical concepts related to this thesis. Information about autonomic systems and about quality of experience is provided in this section.

- **Analysis** (*section 3*): Before starting any system implementation, we need to do a deep study about the main problem in order to find out the best way to solve it. Here we review the main problem characteristics and the different designs/solutions adopted throughout the realization of this thesis.
- **Implementation** (*section 4*): This section describes the implementation of the final system. Specifically, it describes and defines those parts of the software implementation that have been significant for the accomplishment of our objectives. It explains what tools we have used, how we storage the GUI in a formal way, how we draw elements, what QoE rules have been developed and what elements make up the application.
- **Test/Evaluation** (*section 5*): In section 5 the system is tested. We evaluate the quality and effectiveness of our solution and also we test the system responses according to the actions carried out by the users. Here, two different experiments and their statistics are shown. We considere two different subjects in order to check if the changes in the GUI are adjusted to the user needs. The first experiment is realized with an experienced user and the second one is realized with a not experienced user.
- **Conclusions** (*section 6 and section 7*): Almost at the end of the text we provide a brief summary about related work, introduce possible extensions on this thesis, and summarize the concepts and skills acquired during its realization and all conclusions that have been reached.

2 Autonomic computing and QoE

2.1 Autonomic computing

In this sub-section we are going to explain one of the most important concepts related to this thesis. What we mean by autonomic computing? What kind of model follows it? What are its characteristics? Which are its benefits? All these concepts will be detailed here. Readers acquainted with these concepts can skip this subsection and still read on subsection 2.2.

2.1.1 Definition

Autonomic Computing refers to the self-managing features of computing resources, adapting to unpredictable changes while hiding intrinsic complexity to operators and users. An autonomic system makes decisions on its own, using high-level policies; it will constantly check and optimize its status and automatically adapt itself to changing conditions [1].

“These autonomic systems are focused on managing themselves without direct human intervention. A well-known inspiration for this functionality is the human central nervous system. Autonomic controls use motor neurons to send indirect messages to organs at a sub-conscious level. These messages regulate e.g. temperature, breathing and heart rate without conscious intervention. The implications for computing are immediately evident: a group of organized, "smart" computing components that provide us with what we need when we need it, without a conscious mental or even physical effort” [2].

An autonomic computing framework might be seen as composed by Autonomic Components (AC) interacting with each other. Although a variety of architectural frameworks based on “self-regulating” autonomic components has been proposed [1], we are going to introduce here a particular set of ACs as they are directly related to the kind of autonomic system we will use to represent our problem. These ACs are called: Monitoring (M), Analyser (A), Planner (P), Executer (E) and Knowledge (K).

In order to reach a better understanding on their mutual relations, we are going to explain in detail each of them.

- M = At Monitoring stage sensors are responsible for detecting and determining inputs. Such sensors are very important on autonomic systems because the more of them we have, the more complex rules we can add. Besides that here is where all kind of information, important or not, is transmitted to the Analyser stage.
- A = It collects events generated at the previous stage, it analyses them and if they are relevant it stores them.
- P = Planner’s work consists of checking data saved at Analyser stage and if that data fulfil some conditions then it schedules response-actions to be executed.
- E = It is responsible for executing all those actions scheduled by Planner so it will cause changes to the system.
- K = Knowledge block contains information generated through system’s life. It is the fountain of knowledge from which the system recognizes past patterns, correlates current with past events and extracts data to decide if changes need to be applied.

Depending on the general policies and rules defined as inputs for the self-management process, IBM defined different functional areas [1]:

- **Self-Configuration:** Automatic configuration of components.
- **Self-Healing:** Automatic discovery, and correction of faults.
- **Self-Optimization:** Automatic monitoring and control of resources to ensure optimal functioning.
- **Self-Protection:** Proactive identification and protection from arbitrary attacks.

Specifically the functional areas where we will focus our work will be on self-configuration and self-optimization.

2.1.2 Model

In order to understand what model an Autonomic system follows we are going to introduce an important concept.

Control loops

A basic concept applied in Autonomic Systems are closed control loops. This well-known concept stems from Process Control Theory. Essentially, a closed control loop in a self-managing system monitors some resource (software or hardware component) and autonomously tries to keep its parameters within a desired range. If these parameters exceeds from the desired range then the system reacts changing some system aspects [1].

Conceptual model

If seen as conceptual model an autonomic system is divided into different building blocks:

- Sensors (S_i): The sensing capability, which enables the system to observe its external/internal operational context.
- Purpose: The intention and goals the system tries to reach.
- Know-How: Determines which actions to implement and how to apply them without external intervention (e.g. knowledge configuration, interpretation of sensory data...).
- Logic: Responsible for taking the right decisions to serve its *Purpose*, it is influenced by the observation of the operational context based on the sensor information.

This model highlights the fact that the operation of an autonomic system is purpose-driven. This includes, its mission (e.g. the service it is supposed to offer), the policies (e.g. what defines the basic behaviour) and its adaptability [1].

On subsection 3.2 it will be provided a correlation between concept model elements and the problem at hand.

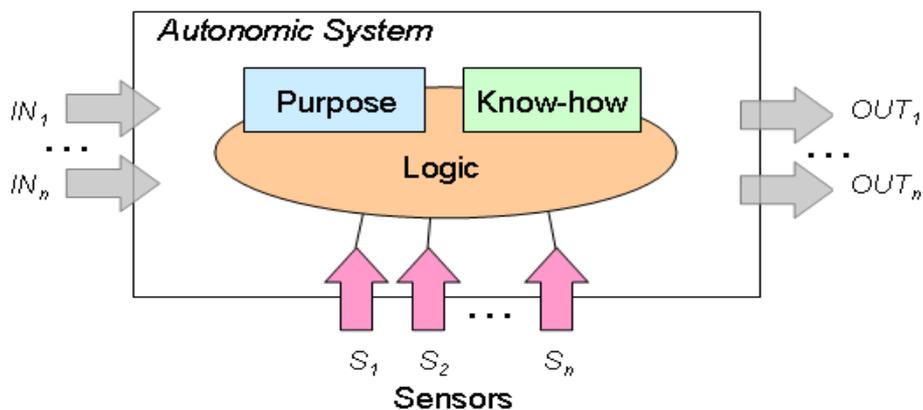


Figure 2.1.2.1: Autonomic System Model.

2.1.3 Characteristics

Even though the purpose and thus the behaviour of autonomic systems vary from system to system, every autonomic system should be able to exhibit a set of characteristics to achieve its intention. The following list suggests seven defining properties of an autonomic system [2]:

1. An autonomic computing system needs to "know itself", that means its components should be identifiable. An autonomic system will need detailed knowledge of its components, current status, and ultimate capacity.
2. An autonomic computing system must configure and reconfigure itself under varying conditions.
3. An autonomic computing system always looks for ways to optimize its workings. It will monitor its elements and adapt workflow to reach predetermined system goals.
4. An autonomic computing system must be able to adapt to routine and extraordinary events that might cause some of its parts to function in a different way than desired. It must be able to discover those components that give rise to confusion, then find an alternate way of using resources or reconfiguring the system to keep functioning according to what user is expecting (this refers to QoE - more information on this is provided in section 2.2).
5. An autonomic computing system must be an expert in self-protection. It must detect, identify and protect itself against various types of attacks.

6. An autonomic computing system must know its environment and the context surrounding its activity, and act accordingly.
7. An autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden (without involving the user in that action).

On section 5 we will review which of these characteristics our autonomic system has been reached.

2.1.4 Benefits

We can consider two different kind of benefits: Short-term and long-term according to their benefits immediacy [2].

Short-term benefits

- Reduced dependence on human intervention to maintain complex systems.
- Favourable relation between costs and user happiness.
- Simplified user experience through a more responsive system.
- Stability.
- High availability.

Long-term benefits

- Constructing autonomic federated systems: Embedding autonomic capabilities in client, servers, middleware... and the network itself.
- Achieving end-to-end service level management.
- Allowing different entities to collaborate on complex problem solving.
- Quality of Experience maximization due to a large iterative learning process.

2.2 Quality of Experience

Quality of experience (QoE), also known as quality of user experience, is a subjective measure of a customer's experiences with a service (Graphical application, phone call, TV broadcast, call to a Call Center, web browsing...). Quality of Experience systems will try to measure metrics that user will directly perceive as a quality parameter (e.g. reaction time when some message appears on GUI).

For instance, if we analyse a vendor's or purveyor's offering from the standpoint of the customer or end user, we want to know what mix of goods, services, and support, the user thinks will provide him or her with the perception that the total product is providing him or her with the experience he or she desired and/or expected. If the vendor/purveyor has not actually provided that, then the QoE tries to determine what changes are needed to be made to enhance his or her total experience (for instance, if a given user is not able to read the values arranged on a slider, then the QoE determines that slider size should be increased).

Quality of Experience is related to but differs from Quality of Service (QoS), which attempts to objectively measure the service delivered by the vendor: with QoS measurement is most of the time not related to user, but to technology services offered. A vendor/purveyor may be living up to the terms of a contract's language, thus rating high in QoS, but, the users may be very unhappy, thus causing a low QoE.

Although QoE is perceived as subjective, it is the only measure that counts for service users. Being able to measure it in a controlled way helps companies/programmers understand what may be wrong with their services/applications [5].

3 Problem analysis and solutions

3.1 Problem description

Here we are going to describe in detail the problem briefly presented in the introduction.

This thesis investigates some problems in the context of autonomic systems in order to make easier the interaction between the user and the system.

Within that context it is required some application, in particular a Graphical User Interface, where we can demonstrate (as a proof of concepts) that if we monitor the actions produced by the user and we analyse the QoE of these actions, then (when actions indicate a loss of QoE) we can plan and execute changes of some aspects of the GUI in order to facilitate user tasks and improve again QoE. These concepts and ideas will be developed and explained later.

Through a GUI we usually can push buttons, enable a set of options, select different menus, fill in text fields, move sliders to fix values and much more. When users interact with these widgets/components is because they are expecting a specific effect or response. If the system detects that some set of actions carried out by the user do not make sense or are different than the normal way (functionally valid but semantically not valid), this indicate that the user does not know how to interact properly. For example, in that case the system could detect this sequence of actions and change correspondig widgets so that the combination used by the user will be correct (“the client is always right” – they say in business). Other examples could be:

- When the user often presses the wrong button due to the fact that e.g. that button is too close to a text field then the system could reshape the GUI and move it away from the text field.
- If the user rarely uses some part of the screen (e.g. widgets that never are used) then the system could remove elements contained within that part and put other elements in its place.

As a general remark, before determining which conditions cause what changes, we have to consider the environment and the context for which a GUI is oriented. Because of the variety of commercially-available user terminals a GUI may be deployed onto several different surroundings. For instance, we cannot apply the same rules on a GUI deployed onto a touch phone than a GUI deployed onto a conventional PC.

In brief, if we detect QoE changes from recurring actions of the users; unproductive actions, specific actions repeated more than n times, etc, such patterns will be interpreted as a loss of QoE therefore, once detected them, we have to show that indeed we can reshape some aspect of the GUI in order to help the user. That means, in an autonomous way the system has to make appear or disappear widgets or parts of the GUI, it has to create shortcuts if they are useful, it has to disable checkboxes, create new ways to select values (e.g. use a slider rather than a text field), turn into a bigger size some parts of the screen ...

3.2 Problem analysis

If we analyse deeply the system needed, we can clearly understand it as an autonomic system.

Inside this system we can identify the five different autonomic sub-components commented in section 2.1: Monitoring (M), Analyser (A), Planner (P), Executer (E) and finally Knowledge (K).

These sub-components are related and integrated in a control loop that every cycle checks if adaptation is needed.

The following figure captures and shows this process.

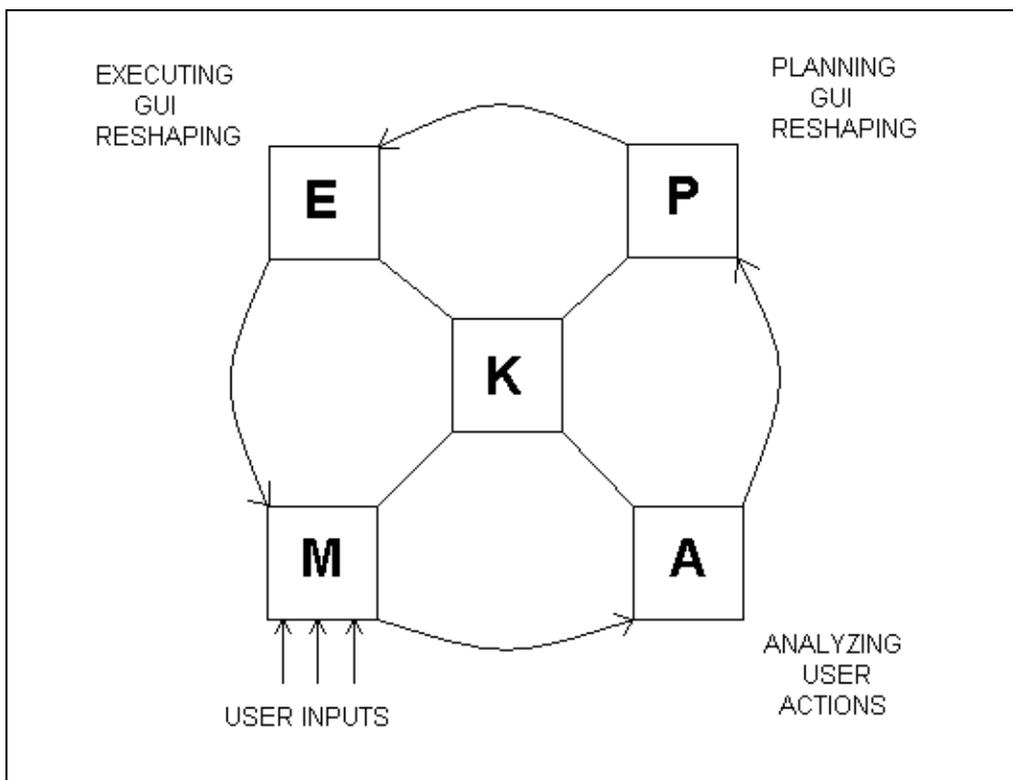


Figure 3.2.1: Autonomic system control loop.

More specifically the process that describes the loop applied to our problem can be summarized as follows.

- The system is “reading” data permanently (*Monitoring* stage).
Examples:
 - The user is pushing buttons.
 - The user is selecting menus.
 - The user is entering characters.

- From that read data the system is deriving some knowledge (*Analyser* and *Knowledge* stages).
Examples:
 - The user is typing too fast and that is interpreted by the system as a stress symptom.
 - The user is always selecting the same menus and the system interprets that the user is losing time unnecessarily.

- Such knowledge is used to plan a response (*Planner* stage).
Example:
 - The system is scheduling a reshaping of the GUI.

- Lastly, the changes are executed (*Executer* stage).
Examples (applied to the examples before):
 - The GUI is transformed into a more friendly GUI with different buttons or different colors in order to reduce stress.
 - The system creates a shortcut on the GUI in order to save time the next time the user wants to select that menu.

We will link now building blocks of the concept model with the design we want to implement.

Sensors

We will use different widgets in order to validate our concepts. Interaction between these widgets and the user give us the input information to determine QoE's user. Next table shows the most important sensors used in our system. Information provided by this sensors will be used in our rules with the purpose of deciding if a change is needed.

Widget	<i>Number of clicks</i>	<i>Times Scroll bar</i>	<i>Focus on</i>	<i>Characters entered</i>	<i>Number characters</i>
Slider	Sensor 1		Sensor 2		
Clean button	Sensor 3				
Name TextField			Sensor 4	Sensor 5	Sensor 21
Validate Button	Sensor 6				
Text Area	Sensor 8		Sensor 7		
Day TextField			Sensor 9	Sensor 10	Sensor 22
Month TextField			Sensor 11	Sensor 12	Sensor 23
Year TextField			Sensor 13	Sensor 14	Sensor 24
Username TextField			Sensor 15	Sensor 16	Sensor 25
Domain TextField			Sensor 17	Sensor 18	Sensor 26
Check button	Sensor 19				
Add Date Button	Sensor 20				

Purpose

Purpose determines the main goal system and within the context we face it is very clear. Our main purpose is optimizing Quality of Experience of the user.

Know-how

At this block knowledge is determined about how our system should operate by itself. This block decides which sensory information is important. For instance, our system provides several sensors (more than those presented on the above table) but just a part of them are considered. The Know-how block knows how to interpret data information.

Logic

Here it is defined our semantic block. Such block consists of a set of rules whose functionality consists of detect QoE failures with the aim of applying changes to the system.

Examples of these rules may be (see on above table sensor references):

(Sensor21 = 0) and (Sensor3 > 4) → move Clean button to the right

(Sensor1 > 4) → increase slider size

(Sensor10 = 'char') or (Sensor12 = 'char') or (Sensor14 = 'char')

→ Date field error

. . .

We have to point out the fact that Artificial Intelligence support would be required in order to deriving new knowledge and increase our knowledge database but that is out of the scope of this work, we want to highlight again that this work is a proof of concepts.

The reader may find similarities with other areas like control theory and natural systems in the loop presented above. Ideas presented in these areas could possibly be used in the software engineering area. The major problem with the current autonomic components is that this control loop presented here is often hidden, abstracted or internalized.

3.3 Our solution

Now we will explain which were our first designs and ideas to define the current system and which were our problems related to them.

3.3.1 First design

At the beginning, we had a set of files and base codes with which we tried to start developing our solution. The origin of these files was from an unfinished prototype which had to be completed.

These base files consisted of:

- A very simple Tcl/Tk GUI instrumented such that it creates a transcript file that lists some basic actions carried out by user.
- A scanner of the transcript language file created with the Tcl/Tk GUI.
- A Makefile to produce the executable file.

- A parser of the transcript language. When it is compiled using the Makefile, it creates an executable file that checks with the transcript file if some QoE failure rule have to be triggered.

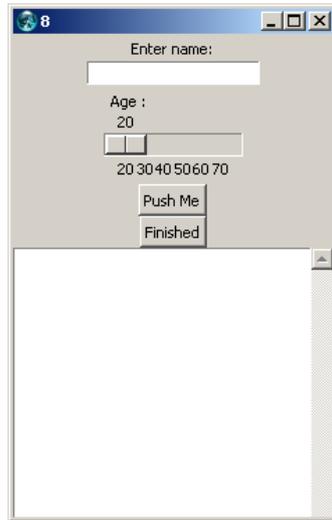


Figure 3.3.1.1: GUI programmed with Tcl/Tk.

Taking these files we had yet a number of incomplete aspects:

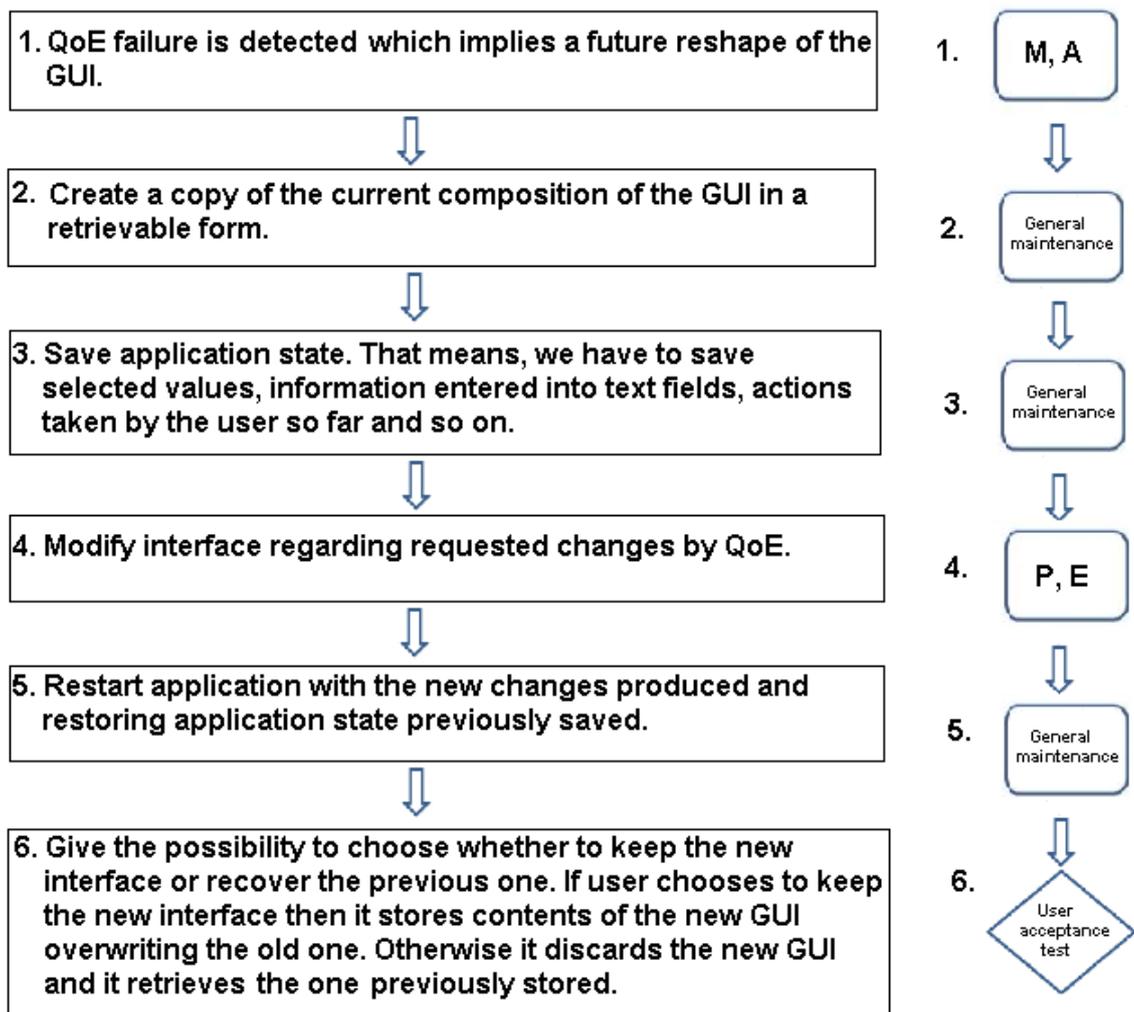
- 1) A too simplistic GUI.
- 2) Too few and too simple QoE "detection rules".
- 3) No adaptation of the GUI is associated with the QoE detection rules. That is, once the QoE failure are detected then they are just reported back and no changes to the GUI are applied regarding those QoE failures.

From this point of view, our original assessment was first, to study the problem and the code and then, extend and design improvements on 1) and 2) and finally, design a proof of concepts for 3). The positive aspect we saw in this is that we would work with a number of well-defined and independent "units": pieces of 1), 2) and 3). If we got difficulties in one unit, we could switch to another one.

It is very important to highlight the fact that the system was not a complete system.

Thus we decided to improve and add more complex components to Tcl/Tk interface we had, and we also wanted to design new and more sophisticated QoE detection rules in order to change the GUI in a more useful way. From that moment, our main goal was focused on finding a way to reach adaptation of the GUI if any QoE failure was activated.

In order to make effective this adaptation one of the possible solutions that we chose is summarized in the following chronological step list.



The diagram on the left represents a description of each step. The diagram on the right represents the functional logic associated with each step (M, A, P and E mean Monitoring, Analyzer, Planner, Executer respectively)

After thinking about it carefully we concluded that the best way to store the GUI in some concise form and in order to be retrieved and modified without problems, it would be serializing the contents of the window.

If the reader is not familiar with the term *serialize*, now we are going to detail some aspects related to this term. Otherwise, please skip the next text box.

The words serialization and “serialize” have several different meanings, depending on the context. In the context of this thesis, *serialize* essentially means “to store the state of an object.” Conversely, *deserialize* means “to restore the state of an object.” This is most often associated with the save/load features of an application, but serialization actually has several uses aside from saving and loading, for example [7]:

- * *Generic cloning of objects.* Any objects which can be serialized can be cloned using the serialization interface, without requiring an intermediary data file.
- * *Clipboard support.* Serialization can be used to store a copy of an object in a system clipboard.
- * *Network transport.* Serialization can be used to transport over arbitrary streams (files are also covered by this abstraction).

Regarding the format in which we would store the contents of the interface, we analyzed different options.

At first we studied the possibility to do it through a library with which we could serialize our object to XML. Using a XML format we could use different tags easily identifiable and replaceable in case of modification, as well as being an agile way to handle and understand the information stored. So the idea was to create a class that stores all information related to the interface. This class would contain general parameters, as well as an array of widgets with specific information for each item (size, color, position...).

However we observed that the GUIs form a special hierarchy. Such a hierarchy can be represented by an n-ary tree where each node represents one element of the screen. Using such a structure the way to access the elements and cause changes in the interface is much more efficient. That was how we finally decided that we would represent the GUI through a tree that would be serialized using basic serialization libraries (more details about how the structure and the GUI are stored can be found on section 4.2).

Since many files we had available were programmed in C, we decided to use any C library to serialize the tree. The chosen library was the libc11n. Libc11n is released under the umbrella of the s11n Project. That means libs11n is the conceptual forefather of c11n, and its architecture is based entirely on s11n's [7].

On the other hand, we needed that our serialized tree containing the GUI information could be retrieved and displayed on the screen consistently. Since this serialization was done in C, we just could deserialize the tree using libc11n if we did it also in C code or another language that was capable of interpreting it.

Searching on Internet we found out a kind of peculiar language: L language. *“L is a compiled-to-byte-code language with the unusual twist that it compiles to Tcl byte codes and by doing so leverages the entire Tcl runtime. L is designed to peacefully coexist with Tcl rather than replacing Tcl. L functions may call Tcl procs and vice versa. They may also coexist in the same source file. L is a static weakly typed language with int, float, string, struct, array, and hash as first-class objects”* [8]. The L syntax is completely based on C with a tiny bit of C++ thrown in.

It is for the above consideration that using the language L we had the bridge we needed between Tcl/Tk GUI and serialization with libc11n C library.

3.3.2 Encountered problems

Although with this design we initially felt able to reach an acceptable solution, we finally found different problems that made it difficult to continue.

These problems were:

- Excessive coexistence of different programming languages in a single system that led to compatibility problems between them.
- Serializing objects like tree doesn't work correctly with very big sizes. Searching on Internet we found out a forum where other libc11n's users had the same problems.
- Too many difficulties and big effort implementing when we want to serialize or deserialize objects with libc11n.

- ❑ It was not possible to integrate the library libc11n inside L language.
- ❑ Combination Tcl-L was not so good as we thought.
- ❑ There are not many help/information/examples on Internet about L programming and about the correct use of the libc11n library.

As a result of these problems we decided to change radically the support with which we wanted to implement our design. We left out all files from which we started to develop our solution. That means we left aside the libc11n library, the L language, the base codes, the implemented C new code... Then we started again in order to re-implement all code completely from scratch and using only one language: Java.

With Java we have graphical tools and useful libraries like Swing and AWT(Abstract Window Toolkit) with which we can handle graphical issues related to our GUI. These two libraries have important features.

AWT features include [10]:

- A rich set of user interface components.
- Graphics and imaging tools, including shape, color, and font classes.
- Layout managers, for flexible window layouts that don't depend on a particular window size or screen resolution.
- Data transfer classes, for cut-and-paste through the native platform clipboard.

- A robust event-handling model.

Swing features include [10]:

- All the features of AWT.
- 100% Pure Java certified versions of the existing AWT component set (Button, Scrollbar, Label, etc.).
- A rich set of higher-level components (such as tree view, list box, and tabbed panes).
- Pluggable Look and Feel.

Apart from graphic control with Java we also have native serialization library, whose packet implements the `java.io.Serializable` interface.

Benefits to programmers include:

- Reducing time taken to write code to save and restore objects or application states.
- Eliminating complexity of save and restore operations, and avoiding the need for creating a new file format.
- Making it easier for objects to travel over a network connection[9].

On the next section we will detail each part of the implementation of the final solution.

4 Final implementation

4.1 Tools used

Having once decided to implement the whole code using Java language we had to choose with what IDE (Integrated development environment) to implement our solution in order to make programming easier.

Two options we considered here, NetBeans IDE or Eclipse IDE. After studying the two options in depth we decided to use NetBeans IDE version 6.9.1. The main reason was that NetBeans comes with a professional and easy to use GUI builder.

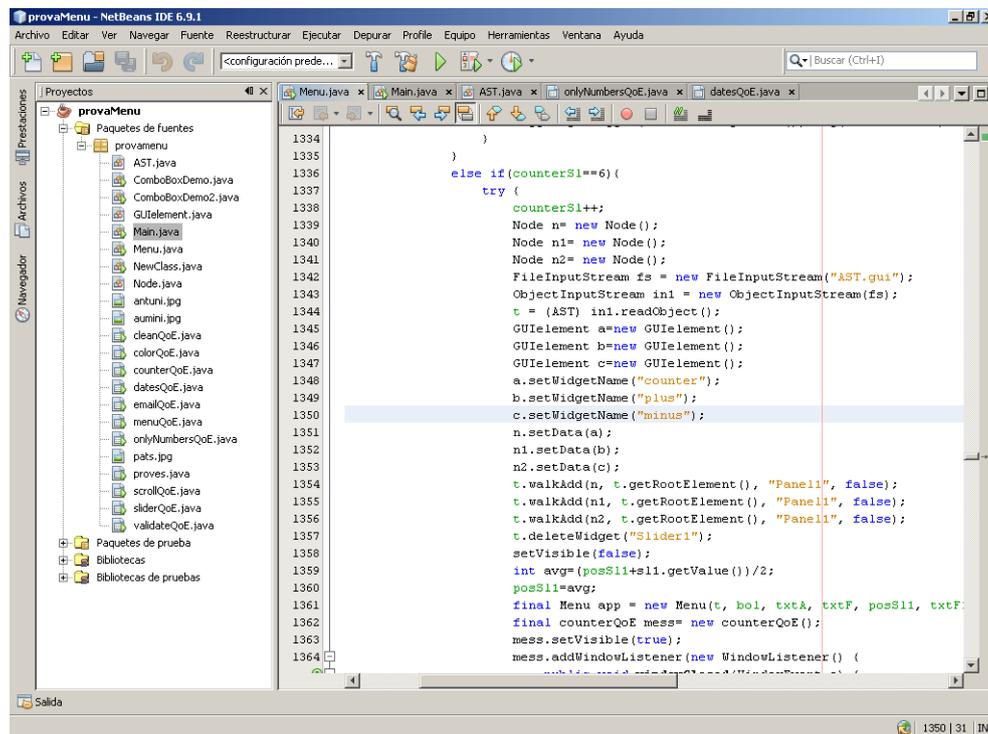


Figure 4.1.1: NetBeans IDE.

Furthermore, as explained in the previous section we used different typical Java modules for each purpose:

- Graphical modules:
 - `java.awt` (AWT)
 - `javax.swing` (Swing)

- Serialization modules:
 - `java.io.Serializable`

4.2 GUI's Hierarchy

As we mentioned in sub-section 3.3.1, if we look at the way in which typically a graphical window is composed, we observe that its structure can be represented as a tree-like hierarchy. That is, we have a window and within it we can find various widgets such as panels, buttons, sliders, text fields, text areas, menus... Each of these widgets may contain or not contain other elements within it, e.g. a menu bar may contain drop-down menus or menu items and a drop-down menu may contain other menus. Likewise a panel may contain buttons, text fields, sliders,..

The following figure shows the hierarchy just discussed.

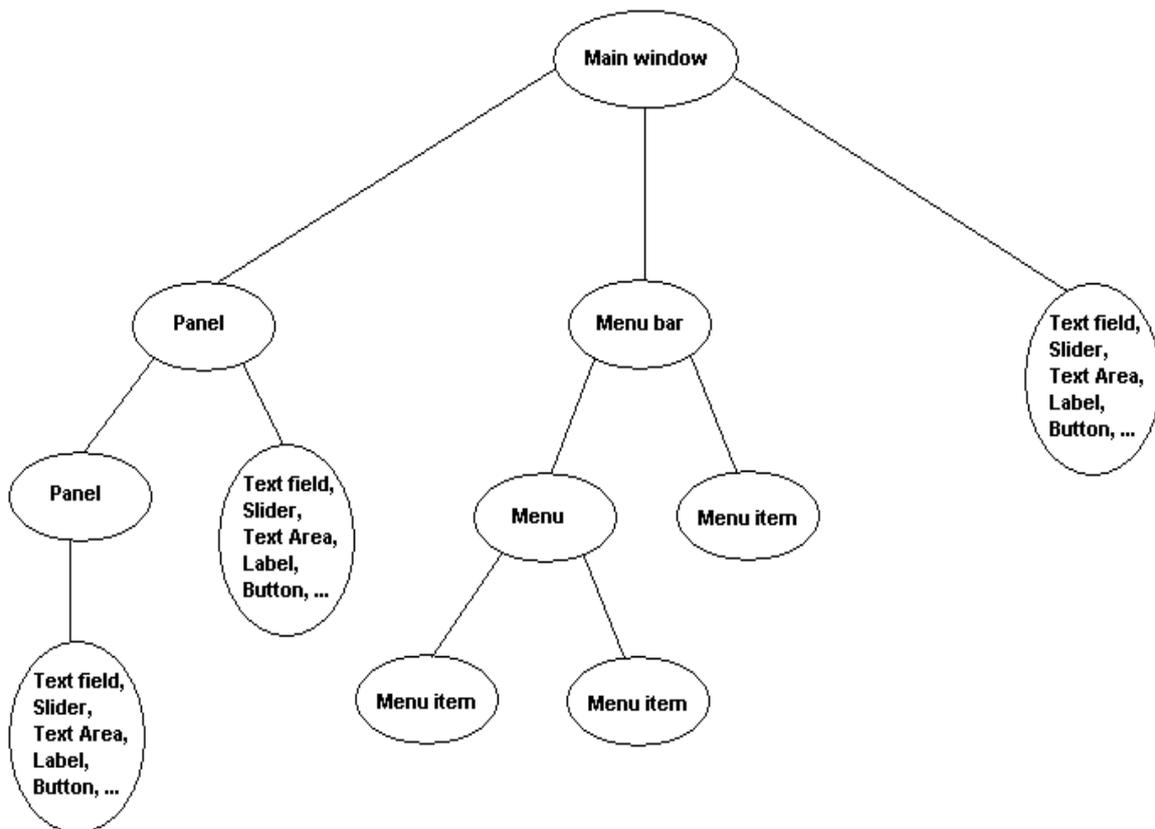


Figure 4.2.1: Window's hierarchy.

Therefore given this structure we can represent a GUI using an n-ary tree (i.e. a tree where each node has an unlimited number of children). For example, suppose you have a GUI that consists of a main window with a panel and a menu bar. The panel contains a button, a slider and a panel with a text field inside. Finally the menu bar contains 2 menus with a menu item each.

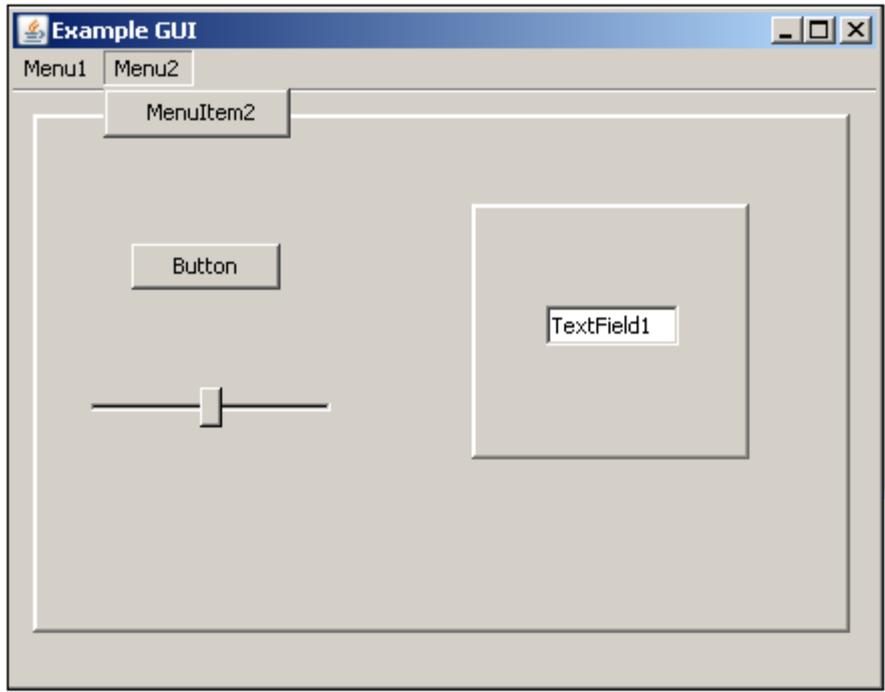


Figure 4.2.1: Example GUI.

The tree structure of such a GUI would be as follows:

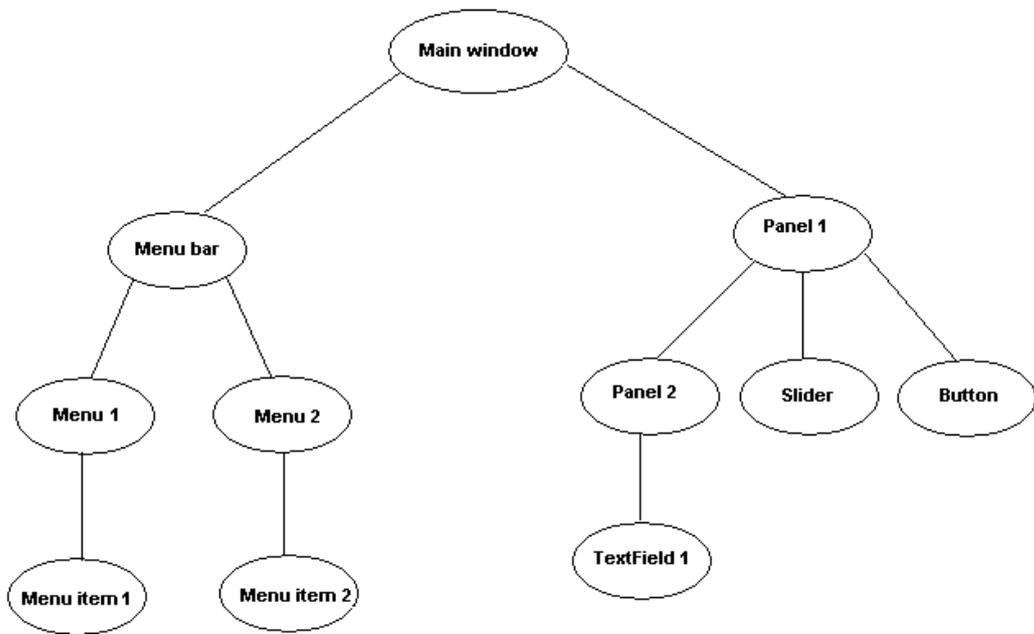


Figure 4.2.2: Window's hierarchy of the screen before.

Apart from the fact that using this tree we have the hierarchy of containers, each node contains information related to the widget configuration that is represented by that node. This basic information consists of: the position, size, color, type of widget, boolean activation... To store all this information the `GUElement.java` class was created (in the next section we will see its content).

Thus, each time there is a QoE failure giving rise to a change in the GUI, we will access to a particular node (i.e. the widget) and execute some actions. Such actions include:

- Modify parameters (position, color, size, ..) .
- Move element to another area on the screen.
- Delete element.

A comment regarding the fact of using a tree to represent our GUI is that every time we need to move a GUI element (e.g. a sub-menu inside another menu) we simply have to cut a tree branch and paste it into another, reducing considerably the complexity of such operations.

4.3 Classes

In this section we show the schema classes represented by our system and we explain briefly the contents of each of them.

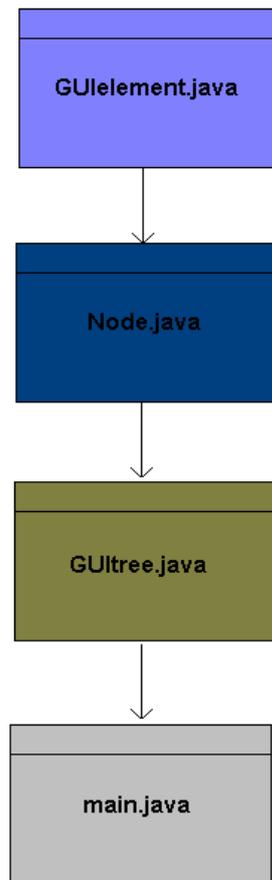


Figure 4.3.1: Schema file.

□ GUltree.java

This is the class that represents our GUI therefore the most used type data in the main.java class. The data structure contained in this class is basically a special node type created by us, which works as a root node and 2 lists used to save information about data that have to be shown in some widgets (these widgets will be explained on next sub-sections). The node contained in this class is the parent of the remaining nodes and the first to visit on almost all searches.

The type data definition contained in it and most important functions are:

```
public class AST implements Serializable {

    private Node rootElement;
    private DefaultListModel domains;
    private DefaultListModel dates;
    /* NOTE: dates and domain are params used with
       specific widgets. */

    /* Default constructor */
    public AST() {
        . . .
    }

    /* Return the root Node of the tree
       Output= rootElement */
    public Node getRootElement() {
        . . .
    }

    /* Return the domains attribute.
       Output=domains */
    public DefaultListModel getDomainList() {
        . . .
    }

    /* Return the dates attribute
       Output=dates*/
    public DefaultListModel getDateList() {
        . . .
    }

    /* Set the root Element for the tree */
    public void setRootElement(Node rootElement) {
        . . .
    }

    /* Set the domain list */
    public void setDomainList(DefaultListModel list) {
        . . .
    }
}
```

```

/* Set the date list */
public void setDateList(DefaultListModel list) {
    . . .
}

/* Returns the Tree GUI as a List of Node objects.
The elements of the List are generated from a pre-order
traversal of the tree.
Output = List<Node> element*/
public List<Node> toList() {
    . . .
}

/* Returns the Tree GUI only from the n param Node as a List of
Node objects. The elements of the List are generated from a pre-
order traversal of the tree.
Output = List<Node> element*/
public List<Node> toListNoRoot(Node n) {
    . . .
}

/* Returns a String representation of the Tree where appears each
widget name contained on the tree. The representation format is
“ [ { grandparent, [ parentA,...,parentN ] }, { parentA, [ childA1,...] },
{ childA1, [...] }, { parentN, [childN1,...] }, { childN1, [...] } ] “
The elements are generated from a pre-order traversal of the Tree.
Output = String of the widgets name contained in the tree */
public String toString() {
    . . .
}

/* Returns a String representation of the SubTree contained at
node n. It uses the same representation of the above function The
elements are generated from a pre-order traversal of the Tree.
Output = String of the widgets name contained in the Subtree */
public String toStringNoRoot(Node n) {
    . . .
}

```

```

/* Moves the node containing the widget name1 to the node
containing the widget name2 */
public void moveWidgetTo(String name1, String name2){
    . . .
}

/* Deletes the node containing the widget name1 and links name1's
parent with name1's children */
public void deleteWidget(String name1){
    . . .
}

/* Search the widget with name name and returns his node.
Output = Node element containing widget name passed
as param */
public Node searchWidget(String name){
    . . .
}

/* Adds node n1 as a child to the node containing the widget
name. It starts searching from the node element and puts on b
a boolean "true" if the widget is found and "false" if it is not found.
It is a recursive method */
public void walkAdd(Node n1, Node element, String name,
                    boolean b){
    . . .
}

/* Deletes the node containing the widget name1. It starts
searching from the node element and puts on b a boolean true
if the widget is found and false if it is not found. It is a recursive
method */
private void walkDelete(Node element, String name1, boolean b){
    . . .
}

```

```
/* Walks the Tree in pre-order traversal. This is a recursive method,
and is called from other methods (e.g. from toList() method with the
root element as the first argument). It appends found nodes to the
second argument list, which is passed by reference as it recurses
down the SubTree contained in element param.
```

```
Output= List<Node> list created from Node passed as param */
```

```
private void walk(Node element, List<Node> list)
{
    . . .
}
}
```

□ Node.java

This class defines the node type we mentioned in the class before, corresponding to the root of the tree and therefore is the class directly related to GUITree.java. Also it is the class that defines how the tree nodes are related to each other.

The data type contained in it is a GUIelement type which contains parameters regarding each widget and a linked node list corresponding to the children of the related node.

The definition of the class and its most relevant functions is:

```
public class Node implements Serializable {

    public GUIelement data;
    public List<Node> children;

    /* Default constructor. */
    public Node() {
        . . .
    }
}
```

```

/* Convenience constructor to create a Node with an instance
of GUElement */
public Node(GUElement data) {
    . . .
}

/* Return the children of Node. The Tree is represented by
a single root Node whose children are represented by a
List<Node>. Each of these Node elements in the List
can have children.
Output= children attribute */
public List<Node> getChildren() {
    . . .
}

/* Sets the children attribute of a Node object which executes
this method with children param */
public void setChildren(List<Node> children) {
    . . .
}

/* Returns the number of immediate children of the Node which
executes this method.
Output= number of children of the node */
public int getNumberOfChildren() {
    . . .
}

/* Adds a child to the list of children for the associated Node
element. If the Node which executes this method has not children, it
will create a new List<Node> in such Node */
public void addChild(Node child) {
    . . .
}

/* Inserts the Node child at the specified position index in the
children list attribute. Will throw an
ArrayIndexOutOfBoundsException
if the index does not exist */
public void insertChildAt(int index, Node child)
    throws IndexOutOfBoundsException {
    . . .
}

```

```

/* Remove the Node element at index index of the
   List<Node> */
public void removeChildAt(int index)
    throws IndexOutOfBoundsException {
    . . .
}

/* Return the GUIelement data attribute
   Output= data */
public GUIelement getData() {
    . . .
}

/* Set the GUIelement data attribute with the data param */
public void setData(GUIelement data) {
    . . .
}

/* Return a list String of the name widgets contained in the
   node and its children. It works like toString() method
   stated in the class before but applied to the Node wich
   executes this method, not applied to the whole tree
   Output= String of the widgets name contained in
   the Node tree */
public String toString() {
    . . .
}

/* Return a List Node of the node ans its children
   wich executes this function.
   Output= List<Node> of the node and its children*/
public List<Node> toListNode() {
    . . .
}

/* Walks the tree from the element node in order to put
   in the list param a List Node of the node children */
private void walk(Node element, List<Node> list) {
    . . .
}
}

```

□ GUElement.java

It is the last class directly related to the representation of our graphical interface, besides being the minimum accessible unit within the tree and as we have repeated several times it contains information related to each widget. Exactly this information is: X and Y positions of the widget, sizes, height and width, widget name, color and functionality booleans.

The content is as follows:

```
public class GUElement implements Serializable {  
  
    public int posX;  
    public int posY;  
    public String WidgetName;  
    public String color;  
    public int SizeX;  
    public int SizeY;  
    public boolean enabled;  
  
    /* Default constructor */  
    public GUElement() {  
        . . .  
    }  
  
    /* Return enabled attribute */  
    public boolean getEnabled(){  
        . . .  
    }  
  
    /* Set enabled attribute */  
    public void setEnabled(boolean b){  
        . . .  
    }  
  
    /* Return posX attribute */  
    public int getPosX(){  
        . . .  
    }  
}
```

```

/* Return posy attribute */
public int getPosy(){
    . . .
}

/* Return WidgetName attribute */
public String getWidgetName(){
    . . .
}

/* Return color attribute */
public String getColor(){
    . . .
}

/* Return Sizex attribute */
public int getSizeX(){
    . . .
}

/* Set SizeX attribute */
public void setSizeX(int i){
    . . .
}

/* Return SizeY attribute */
public int getSizeY(){
    . . .
}

/* Set SizeY attribute */
public void setSizeY(int i){
    . . .
}

/* Set posx attribute */
public void setPosx(int i){
    . . .
}

/* Set posy attribute */
public void setPosY(int pos){
    . . .
}

```

```
/* Set WidgetName attribute */
public void setWidgetName(String text){
    . . .
}

/* Set color attribute */
public void setColor(String color){
    this.color=color;
}
}
```

□ Main.java

Such class is the main program. Within it all the graphic elements that make up the main window are defined and it specifies how they have to be drawn. It also controls all operations performed by the user so that it responds to a change in the GUI if it detects a QoE failure. Due to its large size and extension we are not going to show its content here however, on the following subsections we will discuss its most important features.

As an important comment, we want to highlight that this classes together are a library from which represent and handle other GUIs and not just our GUI. It is true that in some classes there are attributes and functions whose purpose are only oriented to our GUI but if we want to represent another GUI with these classes, these functions and attributs could be simply ignored.

4.4 Drawing elements

Once explained the data structure where we store the GUI and the related classes, we are going to discuss here about how we proceed in order to paint the elements contained in the tree.

In brief the part of the code responsible for this task is based on two nested loops. The first one walks all nodes in the tree, keeping the reference of the father and the second one walks through all child nodes of the node kept by the top loop. If a node has children each child is checked for what type of widget it is, accessing to the `WidgetName` field of each element. When there is a match and depending on the type of widget found we will define specific parameters of that widget and it will be added to the father that, as we said before, will correspond to the node kept in the upper loop. In most cases, we will restore parameters as X and Y position on the screen, X and Y size and color. As one may clearly see, this is part of the Executer module of the MAPE-K loop.

Let us review in pseudocode what was explained before in order to make it a little clearer.

```
. . .
Tree= read_tree
While exist_nodes(Tree) do
    Father=get_node(Tree)
    Childs=get_childs(Father)
    While exist_childs(Childs) do
        Child=get_node(Childs)
        If (widget_name(Child)=button) then
            Set_posX(button, get_posX(Child))
            Set_posY(button, get_posY(Child))
            Set_sizeX(button, get_sizeX(Child))
```

```

        Set_sizeY(button, get_sizeY(Child))
        . . .
        Add(button, widget(Father))
    Else If (widget_name(Child)=slider) then
        Set_posX(button, get_posX(Child))
        Set_posY(button, get_posY(Child))
        Set_sizeX(button, get_sizeX(Child))
        Set_sizeY(button, get_sizeY(Child))
        . . .
        Add(button, widget(Father))
    . . .
    Take_out (Child, Childs)
End_while
    Take_out (Father, Tree)
End_while
    . . .

```

4.5 Saving/loading the GUI

As mentioned at the beginning of this section, to save and retrieve the contents of our GUI we apply serialization particularly, we use the package `java.io.Serializable`.

The mechanism used by this package is quite simple when compared with others. Any object whose class implements the `java.io.Serializable` interface can be made persistent with only a few lines of code. No extra methods need to be added to implement the interface. We only need to add the “implements” keyword to our class declaration, to identify our classes as serializable [9]:

```
public class GUIelement implements Serializable
```

“Now, once a class is serializable, we can write the object to any OutputStream, such as to disk or a socket connection. To achieve this, we must first create an instance of java.io.ObjectOutputStream, and pass the constructor an existing OutputStream instance “ [9].

```
// Write to disk with FileOutputStream  
FileOutputStream f_out = new FileOutputStream("our_object.gui");  
  
// Write object with ObjectOutputStream  
ObjectOutputStream obj_out = new ObjectOutputStream (f_out);  
  
// Write object out to disk  
obj_out.writeObject ( GUItree );
```

Note that any Java object that implements the serializable interface can be written to an output stream this way (including those that are part of the Java API). Furthermore, any objects that are referenced by a serialized object will also be stored. This means that arrays, vectors, lists, trees and collections of objects can be saved in the same fashion (without the need to manually save each one). This can lead to significant time and code savings.

“Reading objects back is almost as easy. The one catch is that at runtime, you can never be completely sure what type of data to expect. A data stream containing serialized objects may contain a mixture of different object classes, so you need to explicitly cast an object to a particular class” [9].

```

// Read from disk using FileInputStream
FileInputStream f_in = new FileInputStream("our_object.gui");

// Read object using ObjectInputStream
ObjectInputStream obj_in = new ObjectInputStream (f_in);

// Read and cast to a GUItree from our ObjectInputStream
GUItree gui = new GUItree ();
gui = (GUItree)obj_in.readObject();

// Do something with gui....

```

4.6 Modifying the GUI

Each time a modification of the GUI is required the general steps to achieve it are the same. The process which follows determines the whole functionality of E component (Executer).

First, we have to load the tree that contains our GUI in the manner described in the previous subsection and store it in a copy. Then we have to take the node content regarding the widget to change (we do this using the searchWidget function). Once we have the node, either we remove it, or we add it into another, or we modify any of its parameters (color, size, position,...). When the desired changes have been done, the next step consists of save the new content by overwriting the previous one (by means of serialization). Finally, to see the new changes in the interface, we have to create a new instance of the main window passing as parameters those values that represent the current state of the application (entered text, selected values, values shown,...) to finally stop the execution of the

original instance of the window. Thus when we execute the new main window, the tree will be loaded again painting all the elements that form the interface but now with the new changes applied. As a remark, when the old interface is being replaced with the new one, it is disabled in order to avoid inconsistencies if the user tries to interact with it.

The following pseudocode represents what we have just explained:

```
Tree=load_gui

Node=searchWidget(name_widget, Tree)

Modify_param(Node, param)

Replace_node(Node, Tree)

Save_gui(Tree)

New_main_Window=new mainWindow(current_state)

Delete(old_main_window)
```

4.7 Quality of Experience Rules

This section details each and every one of the Quality of Experience Rules currently programmed in our application. What follows attempts to offer to the reader a better understanding about how we measure certain user actions. Such actions are very important because they give us significant information related to what we suppose that would help to improve interaction with the application.

4.7.1 Recording background actions

All actions carried out by the user considered as relevant are registered in our system. In particular, and given the way our QoE failure detections are oriented, we record actions related to specific widgets. Actions also detail information such as number of times we change the direction of slider movement, number of times we click a button, number of times we enter information incorrectly in a text field, number of times we scroll to a certain direction... Such information is collected by our sensors at M stage of the MAPE-K.

Then, these parameters are stored and used to measure the Quality of Experience for each user. If at any time any of these values exceeds a certain limit then it is triggered a QoE failure so that the interface changes according to what that parameter value means. Interpretation of the meaning of each parameter is carried out by the semantic module whose functionality is directly associated with the A step/component. Such module establishes rules which determine system behaviour (i.e what changes apply). Furthermore, a relevant fact is that with this module we have a useful tool to detect “QoE failure”. If we would add a publish/subscribe mechanism, once detected a QoE failure, this knowledge could be shared with third parties. Therefore any external service subscribing to it could also be informed of a QoE failure.

Since the interface changes are based on assumptions about what we believe that could be improved or at least about what we think could become more useful for the user and due to the wide range of possibilities related to users and preferences, these changes do not occur unconditionally. Each time one of these improvements are applied, first of all they are shown to the user to give him or give her the opportunity to check what produces these new changes. Due to what we mention here, some user could dislike these changes, therefore our system always offers the possibility to dismiss these

changes. Thus, immediately after applying those changes a dialog is opened in which both options are offered: to keep the new interface or return to the previous one.

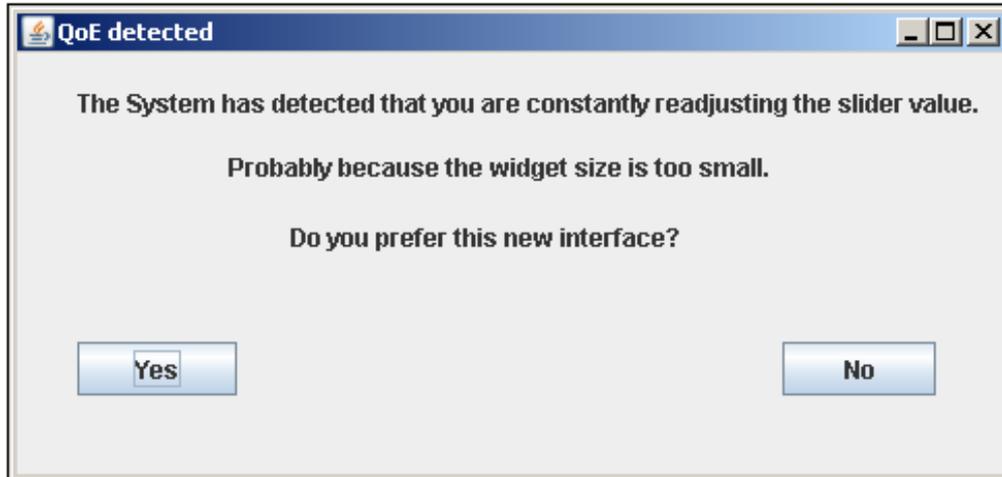


Figure 4.7.1.1: Dialog where user can choose either keep the new interface or reject it.

Let us look at the detection rules mentioned.

4.7.2 First QoE failure detection: Slider

The first QoE failure detection we are going to explain is related to the Slider. The user has a slider with which he or she can select his or her age simply by moving right or left and drop on the desired point.

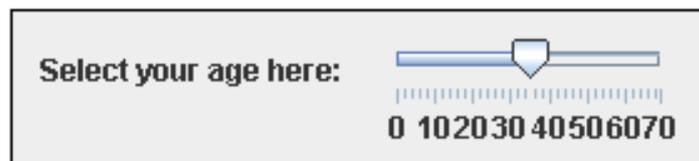


Figure 4.7.3.1: Slider before QoE failure detection.

Depending on the kind of user that uses this widget and the input terminal used it is possible that he or she thinks that the distance between the different values are too short and therefore difficult to see. We have to think about those people who have vision problems and try to offer an easier and enjoyable reading. Thus, when the system detects that the slider is adjusted too many times, it increases its size.

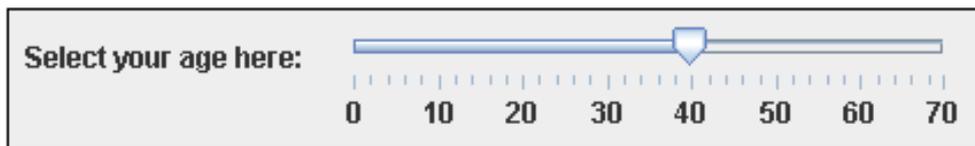


Figure 4.7.3.2: Slider after QoE failure detection-1.

If even so the system still detect that the user is adjusting the slider value then the system offers an alternative even more clear to him or her. We shall consider users that are able to see values arrangement but they do not know exactly how to interact with slider. In that case when it detects another QoE failure on the same widget the system offers the possibility of using a counter. This counter has two buttons, one to increase the value and another to decrease it. Each time user presses one of these buttons the value resulting from increase or decrease is shown on the counter.

In addition, to match the desired value with the initial value shown on the counter, we take the last two values selected with the slider and average them. That average will be the number that appears on the counter.

e.g.: $v_1=30$,
 $v_2=26$,
counter $_{value}=(30+26)/2=28$



Figure 4.7.3.3: Slider after QoE failure detection-2.

4.7.3 Second QoE failure detection: Menues

The second QoE failure detection which will be discussed is related with the menues.

Using menues users can select the different drop-down menues shown until they get to the desired option.

These drop-down menues sometimes tend to be a little unpleasant in the sense that we often have to select too many submenues until can be selected a particular menu.

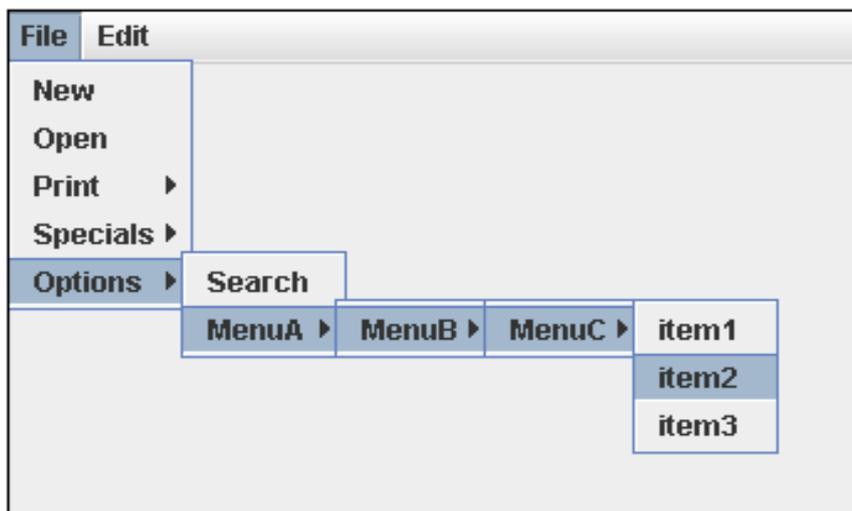


Figure 4.7.6.1: Selection menues before QoE failure detection.

To avoid stress to the user and to be as effective as possible saving time to the user, when the system detects there are frequently used options from a particular menu, it creates a shortcut of that menu in the menu bar.

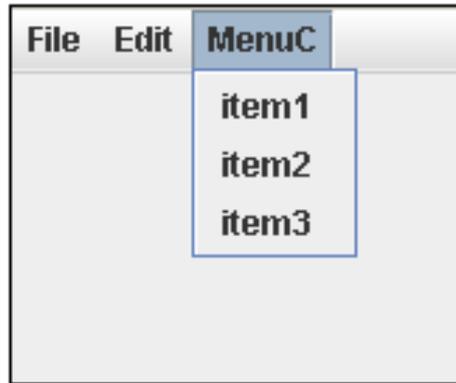


Figure 4.7.6.2: Selection menus after QoE failure detection.

In the example in Figure 4.7.6.1 we can see how each time user wants to select one option from the MenuC he or she has to across other menus. If the menu options are used occasionally then no change will be produced but if these options are used frequently then that fact will be detected and the system will create a shortcut to the MenuC like in Figure 4.7.6.2. In order not to place too many menus as a shortcut the system just will select the three last most frequently used menus.

4.7.4 Third QoE failure detection: Email

The user has a text field where he or she shall enter a email address. Once the email has been entered it is checked for correctness by pressing the check button.



Figure 4.7.7.1: Email widget before QoE failure detection.

When the check button is pressed the system checks the basic structure of introduced mail reporting an error if it is not correct. This error may be due to the lack of point, the lack of domain, lack of @ and so on.



Figure 4.7.7.2: Message error when mail is not valid.

Specifically, when the system detects that the user often forgets to enter the @ then the QoE failure is triggered changing the appearance of the GUI so that the @ will be implied. Therefore the system will create two text fields separated by a label. In the text field on the left the user shall enter the username of the email, the label corresponds to @ (do not write anything) and on the right text field he or she shall enter the domain.



Enter your email here: @

Figure 4.7.7.3: Email widget after QoE failure detection.

Also in order to facilitate user tasks, the system remembers the last email addresses entered correctly.



Enter your email here: @

- gggmail.com
- ggggmail.com
- gggggmail.com
- ggggggmail.com
- hhotmail.com
- hotmail.com

Figure 4.7.7.4: Text field remembering the last correct emails introduced.

In addition, the system tries to be proactive so that when user writes the first letters of the domain, all registered email addresses that start with those letters are shown, so the user can select directly the right one avoiding to type the whole domain. In this sense, this behaviour has similarities with google interface (at different scales, of course). When we type in a text on the text field of google main page, google's system tries to be anticipative offering the most commonly used searches starting with the characters written. Obviously, our domain is much more reduced because it is just focused on latest mails entered.



The image shows a web form for entering an email address. The label "Enter your email here:" is followed by a text input field containing "hector". To the right of this field is an "@" symbol, followed by another text input field containing "gggg". A dropdown menu is open below the second input field, showing three suggestions: "ggggmail.com", "ggggggmail.com", and "gggggggmail.com". To the right of the dropdown is a "Check" button.

Figure 4.7.7.5: The system tries to be anticipative when the user is typing the domain.

4.7.5 Fourth QoE failure detection: Date format

This QoE rule is based on the type of values you enter in the date fields.

The application has three text fields corresponding to the day, month and year in which the user must enter the date. Although it is more common that the user enter values for day, month and year in numeric format someone could enter them in alphabetical format.



The image shows a date input widget. The label "Enter date:" is followed by three text input fields. The first field contains "1", the second contains "8", and the third contains "2011". Below each field is a label: "Day", "Month", and "Year" respectively.

Figure 4.7.8.1: Date widget after QoE failure detection.

If any user has the bad habit for instance to introduce characters on the date text fields rather than numbers (e.g. August instead of 08) and the system detects that the user produces the same mistake repeatedly, then it triggers the QoE failure. Once the fault is triggered a message appears above the date text fields remembering that in these fields only can be entered numbers.

REMEMBER: You only can enter numbers.

Enter date:

Day Month Year

Figure 4.7.8.2: Date widget after QoE failure detection.

4.7.6 Fifth QoE failure detection: Date order

The fifth QoE failure detection that we will discuss in this section is also related to how we type in the date.

The user through the use of the application is entering dates. He or she must first enter day, month and year and click the Add button below in order to register that date (assuming correct entered values). These dates are introduced on different days so that they should appear in chronologically order (see figure below).

Enter date:

Day Month Year

1/2/2011

1/3/2011

1/5/2011

1/9/2011

1/11/2011

Figure 4.7.9.1: Date widgets before QoE failure detection.

However, not all users use the same order of date. Other countries commonly use month/day/year. Thus if the system detects a QoE failure because the last date entered is not consistent with the other, it proposes an alternative. The QoE failure detection can be due to either because in the month field the user has entered a value greater than 12 or because the new date is earlier than the last one. In such cases, the system offers the possibility of exchanging the text fields day and month and re-registers the previous dates but now with the new order. Taking the example above if we try to enter the value date 2/1/2011 the system will detect a value earlier than the others thus it will trigger the QoE failure proposing the change represented in the figure below (the same would happen if we tried to add the date 1/13/2011 because we would give a value to the month field greater than 12).

The screenshot shows a date entry interface. At the top, it says "Enter date:" followed by three input fields containing "1", "2", and "2011". Below these fields are labels "Month", "Day", and "Year". To the left of the date list is a blue button labeled "+Add". To the right is a list of dates: 2/1/2011, 3/1/2011, 5/1/2011, 9/1/2011, 11/1/2011, and 1/2/2011.

Figure 4.7.9.2: Date widgets after QoE failure detection.

4.7.7 Sixth QoE failure detection: Text Area

Now we are going to explain the QoE failure detection related to the text area that shows the validated values.

As mentioned above each time the user presses the validate button and if information has been entered, the text area displays a text with that information.

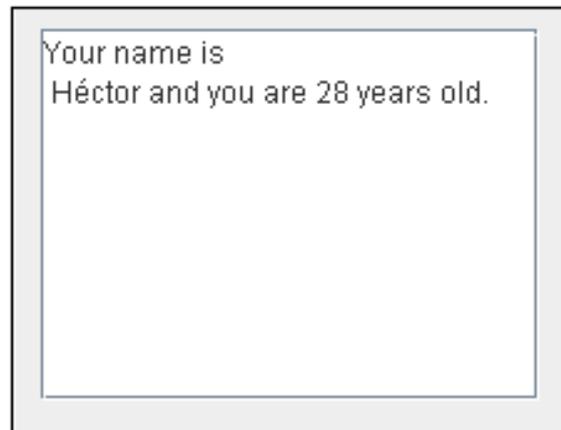


Figure 4.7.5.1: Text Area with a short name before QoE failure detection.

If the entered name is short we can safely view the full description of the selected name and age. However, if the name entered is too long the text area just let us see a part of the description forcing the user to scroll to the right if he or she wants to read the whole description.

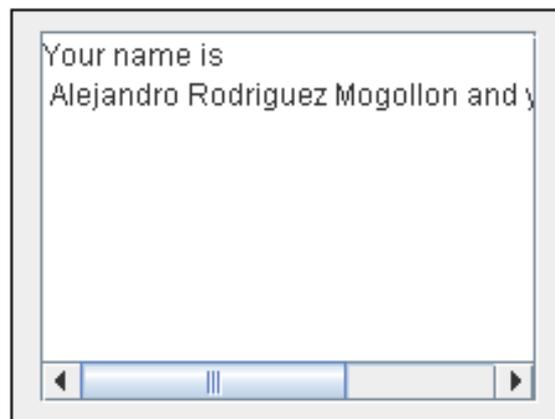


Figure 4.7.5.2: Text Area with a large name before QoE failure detection.

Thus, if the user scrolls horizontally repeatedly, then QoE failure will be detected and give rise to a reshape of the GUI by doing the text area larger. This process will be applied each time the user scrolls until the text area size were enough to get the whole description. As an additional comment the text area increases its size until a max value, from that point if the user still scroll, the size will not be modified. This makes sense because if we allowed to increase indefinitely the text area, it would exceed the limits of our main window.

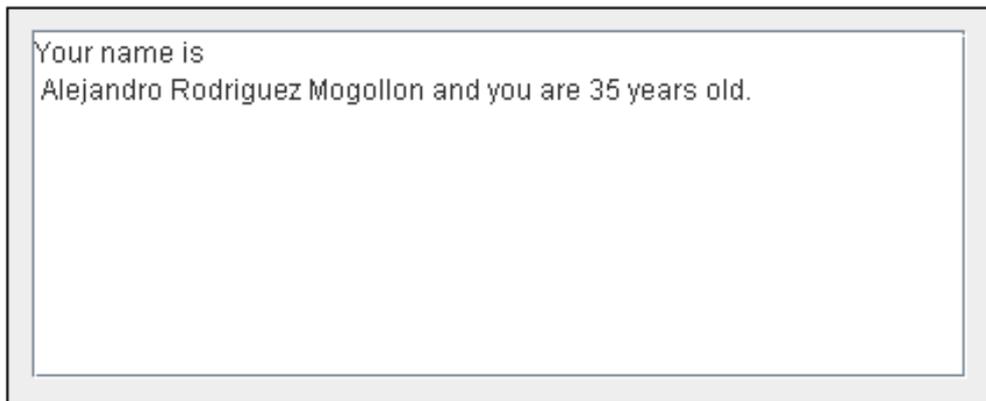


Figure 4.7.5.3: Text Area with a large name after QoE failure detection.

4.7.8 Seventh QoE failure detection: Clean button

This QoE failure detection we are going to explain is related to the clean button. The functionality of this button is basically clear the contents of text field. Thus if the user when typing his or her name makes a mistake, by pressing this button will delete everything that had been introduced.



Figure 4.7.2.1: Clean button before QoE failure detection.

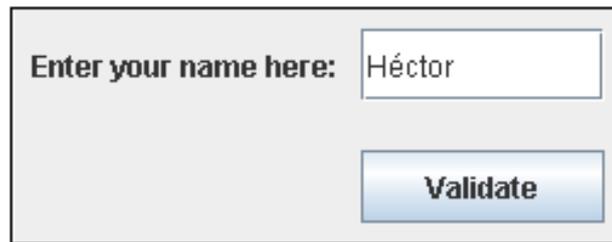
However, if the system detects that throughout the execution and repeatedly, the user presses the button to clean the text field when the content of text field is empty, then it triggers the QoE failure. Since the functionality of the button just like it is labeled and positioned should not lead to confusion, we assume that the explanation about why the user presses the button when the text field is empty is because this button is too close to the text field so that the user is really looking for selecting the text field to fill it. In response to this what the system does is to slightly shift to the right the button, giving to the user the opportunity to return it where it was if he or she wishes.



Figure 4.7.2.2: Clean button after QoE failure detection.

4.7.9 Eighth QoE failure detection: Validate button

The last QoE failure detection is related to the validate button. Once the user has selected his or her age and introduced his or her name, he or she can select the validate button to validate that information. When the user clicks this button, the selected name and age is displayed on a text area.



A screenshot of a web form. On the left, the text "Enter your name here:" is displayed. To its right is a text input field containing the name "Héctor". Below the input field is a blue button with the text "Validate" in white.

Figure 4.7.4.1: Validate button before QoE failure detection.

However, if the user presses the validation button repeatedly when has not been filled in all the information yet (e.g. text field is empty) then the system in order to avoid wasting resources will disable the button. Once disabled the button and as a preventive measure, it will be re-enabled only when the user introduces the name in the text field. By doing so we drive the user to understand that first he or she has to fill in his/her name and then click on the validate button.



A screenshot of a web form. On the left, the text "Enter your name here:" is displayed. To its right is an empty text input field. Below the input field is a greyed-out button with the text "Validate" in grey.

Figure 4.7.4.2: Validate button after QoE failure detection.

5 Evaluation

5.1 The system

Once finished the system we can evaluate its properties, quality and effectiveness.

Regarding the characteristics commented on subsection 2.1.3 we are going to see which of them have been reached.

1. *An autonomic computing system needs to "know itself"*: Our system has a detailed knowledge of its components. All widgets are identified properly so that at all times we can check their current status, know their limits, when they are not used correctly...
2. *An autonomic computing system must configure and reconfigure itself under varying conditions*: Depending on what kind of user are using our application, the system will react in one way or other, reconfiguring its elements according to the actions carried out by the user.
3. *An autonomic computing system always looks for ways to optimize its workings*: If a determined user causes a QoE failure and the system changes accordingly but even so, the user still making the same error then the system will change in a different way in order to optimize communication user-interface.
4. *An autonomic computing must be able to adapt routine and extraordinary events that might cause some of its parts to function in a different way than desired*: For instance, when

the system detects a routine event like the user always clicking on the same submenu, then a shortcut of that submenu will be created on the main window.

5. *An autonomic computing system must detect, identify and protect itself against various types of attacks:* Due to oriented purpose of our application, this characteristic is not supported by our prototypic system.

6. *An autonomic computing system must know its environment and the context surrounding its activity, and act accordingly:* It is well known in our system the context and environment related to it. We are within a context that expects user actions, interactions with the interface and the system reacts depending on the value of these actions/interactions.

7. *An autonomic computing system will anticipate the optimized resources needed while keeping its complexity hidden:* Implementation related to changes produced are performed without involving users in that implementation.

Although our autonomic system achieves most of the typical properties an autonomic system has to reach, sometimes the usefulness of some solutions adopted are not as good as we would like. What we mean is that despite the tests carried out by ourselves were quite satisfactory in some cases the system acts in a different way than expected/disered.

That is due to the limitations to the system. We have to consider limitations due to time constrain (this thesis began later than usual), limitations of our approach (it is just a proof of concepts not a final product), the lack of infrastructures...

5.2 Experiments and their statistics

As an important issue for the reader, the experiments/tests detailed here are not a full-fledged experimentation due to above mentioned infrastructure limitations (as a MSc thesis, this can't be much more than a proof of concepts).

Despite this, we carried out a limited analysis. To be precise, we considered two different subjects: an experienced user and a not experienced user. We detail their relevant characteristics and then we comment what they did, what they were expecting and what actually happened. Finally, we evaluate if the alternatives offered by the system are adjusted to the needs of each kind of user.

At the end of this section, we will compare both results and discuss them and we will continue providing indications about what is acceptable, what needs to be changed and what could be changed in order to make it better.

5.2.1 First subject: Experienced user

Age: 29.

Gender: Male.

Technology experience: Total experience (Computer Sciences student).

Important considerations: He does not mentioned physical defects such as vision problems or similar.

Since the subject was an experienced user accustomed to using buttons, text fields, e-mails and so on he did not prompt many QoE failures.

Nevertheless, we are going to explain which were his actions and opinions regarding each widget and each change applied to the GUI.

Entering name: The subject entered his full name without making errors on the text field enabled for this purpose.

Selecting age: He readjusted repeatedly the slider because he was not able to read correctly the values arranged on the widget therefore the first QoE failure was triggered making the widget bigger. With the new slider size, he selected his exact age perfectly. He believed that the change was quite useful.

Cleaning information: He pushed the clean button repeatedly to check its work. When he saw the button move to the right he thought it could be a good change for people who always clicks on incorrect parts of the screen.

Validating name and age: He used the validate button correctly in order to validate name and age selected.

Showing values selected: Once introduced his name and age and pushed validate button he could not read the whole description in the text area enabled for this purpose, in consequence he scrolled to the right in order to read the whole description. When the scroll movement was repeated the text area changed turning into a larger widget, even so he could not read the whole description scrolling to the right again. When the same QoE failure was triggered making the text area even larger, the subject could read the entire description. The user thought that this change was very useful and comfortable.

Entering mail: He introduced different mails correctly and without triggering any QoE failure but he thought it was very comfortable and useful that the system remembered the last mails entered.

Selecting menus: The subject clicked many times on different options of the same submenu. When he saw a shortcut of that submenu was created on the menu bar of the main window his feelings were quite good. He believed that was an efficient change avoiding to waste time.

Entering date: No errors were generated filling in the date fields.

5.2.2 Second subject: Not experienced user

Age: 53.

Gender: Female.

Technology experience: Subject not used to using computers (just to check email once a month).

Important considerations: She has vision problems (she has to wear glasses).

Selecting age: First of all the subject tried to select her age but due to her vision problems he had to readjust many times slider value giving rise to a resize of the slider. Although once the QoE failure was triggered the slider turned into a big enough size, the subject still had problems selecting her age. When the second QoE failure was triggered and the widget changed to a new widget using a counter instead of a slider then she was able to select her age properly. The subject believed that this change was very useful. If the widget had not changed, she had never been able to select a proper value.

Validating name and age: Before enter her name, she tried to validate information pushing validate button many times, for which reason the QoE was triggered disabling the validate button. The subject needed a few minuts to understand what was happening.

Once she checked that if she entered information on the text field the validate button was enabled then she understood that first it was necessary to enter her name. In a sense, this gave rise to a loss of QoE.

Entering name: After problems mentioned above, the subject introduced her name on the right widget.

Cleaning information: This subject did not push clean button.

Showing values selected: When she pushed on the validate button (with all necessary information entered) the whole description displayed on the text area could be read (her name was short).

Entering mail: She tried to enter her mail several times making different errors (forgetting '.', forgetting '@',...) when the system detected she checked the email too many times without '@' then the mail widget changed with an '@' in the middle of the username and domain. Once the change was applied she entered her mail address correctly. Furthermore the system had remembered the last domains entered and she could select one of them avoiding errors.

Selecting menus: No menus were selected.

Entering date: The subject tried to enter the month in alphabetical way giving rise to QoE failure. Once the failure was detected, the system created a label. Such label clarified that date fields only could be filled in with numbers. That helped the subject because afterwards she only entered numbers. She was used to use format mm/dd/yyyy and although date fields are labeled with their appropriate type, she caused a QoE failure giving rise to an exchange between fields day and month. That was very useful to her because she could enter date as preferred.

5.2.3 About the results

Comparing both results – though obtained with a sample population of just 2 because of time restrictions – it is shown how for a not experienced user the QoE rules have been very useful. The experienced user has not experienced too many QoE failures but even so he has evaluated most of the changes applied by the system as useful, comfortable and efficient.

When the system applied changes reshaping some part of the GUI the experienced user understood very quickly how he had to interact with the interface. On the other hand, when the same changes were applied with a not experienced user, she needed a few minutes to understand what was happening and how she had to interact with the system.

We also have detected that some aspects of the system could be improved. Specifically, some QoE failure detections are triggered too late. For instance, when the user is scrolling to the right because it is impossible to read the whole text displayed, the text area is not resized until several scrollings. Other aspect to change could be for example when the system detects that a set of actions carried out by the user are contradictory or unusual then the system should react indicating what the user has to do with the widgets related to those actions.

All things considered including the above imperfections, calling for specific improvements. Nevertheless, we can conclude that the tests performed were quite satisfactory and positive.

6 Related work

Before detailing any conclusion reached with this thesis, we should examine some other interesting works on adaptive GUIs.

Most of the work related to adaptive user interfaces found out on internet is focused on the task of “information filtering”, in which the aim is to provide the user with material that she/he will find informative or useful.

“One example is Pazzani, Muramatsu, and Billsus (1996) SYSKILL&WEBERT, which recommends web pages on a given topic that the user is likely to find interesting. The user marks suggested pages as desirable or undesirable and the system uses these as training data to develop a model of his preferences. They incorporate a common scheme known as content-based filtering, as the basis for selection and learning.

Another example of an adaptive user interface was developed by Hermens and Schlimmer (1995), who developed an adaptive system for filling out repetitive forms. Their interface suggests values for various fields in the form, but these are defaults that the user can always override. Once the user completes the form, the system interprets the entries as opportunities for learning and uses them to revise its existing predictive rules. These rules predict a default value for a concrete field based on fields entered earlier in the form and in previous forms. Experiments carried out by them showed that the system reduced keystrokes by 87 percent in eight months. In short, this adaptive system learns a grammar that predicts the order and content of the notes of the users focusing on reducing keystrokes and helping users organize their thoughts” [12].

Cypher (1991) describes EAGER, a system that learns iterative procedures from observation in a HyperCard setting, then highlights the actions and it anticipates for the approval of the user. This kind of work are called “programming by demonstration” and briefly, it is based on constructing personalized user interfaces by observing the behavior of the user [13].

The last example we will comment here is the work developed by Leung, Morisson, Wringe and Zou (2006), who developed an adaptive system to adapt Eclipse menu elements to each user. They proposed an architecture that modifies the Eclipse menu system, hiding infrequently used menu elements, and predicting the next menu elements that a user is likely to click. Moreover, they developed adaptive algorithms that perform a cost-benefit analysis for making modifications to the menus system, and determine the optimal changes to make [14].

As an important remark, the reader has to know that this list does not exhaust the work on adaptive user interfaces, which is an active area with many ongoing research efforts and therefore, other works focused on adaptive user interfaces can be found.

7 Future work and Conclusion

7.1 Future work

Since this work is not a finished product, it remains open to possible extensions and improvements. Some of them could be:

- As we have mentioned several times throughout this text, the work developed here is just a proof of concepts. The main extension/improvement that could be applied to our work would be turn it into a final product, applying the basis of our system and all concepts learned. They could be implemented in a real environment like on an internet web page. A special kind of web page where it would be useful could be on an online travel agency. The web page system would register user's actions, storing his or her likes and dislikes. The web page would be a reliable application that would adapt preferences, needs and improvements to each user. Throughout user's interaction with the system, the Graphical User Interface of the web page would be more suitable, all widgets and their features (buttons, selectors, text fields, colors, shapes, positions...) would be created from what the user are expecting and from errors learned, avoiding that the user could repeat again the same errors. Furthermore the system would be anticipative to users, filling out all information on the forms. The system would suggest values from previous values selected by the user but these values always could be rejected by the user.

- Another interesting improvement to apply would be implementing advanced concepts from Artificial Intelligence. It could be developed an expert system with a very big knowledge database. Inside this system, knowledge would be introduced as rules. These predefined rules would be based on other user's experience. Every time the user would use our system, the knowledge base would grow learning more with every interaction user-system. From that knowledge base the system could predict future user actions, possible user preferences, needed changes to apply to the system...
- A very ambitious extension would be to extrapolate the system to a large-scale. We would register all interactions with GUIs carried out by the user, not only with our system but also with all kind of user interfaces which the user interacts with, including interfaces over internet like web pages. Every time the user uses any GUI it would storage what they do and the database about user behaviour would grow. The total knowledge acquired about that user would accrue on a way to determine exactly what would be better to that user so that any application were able to apply changes on its user interface in order to make easier user life.

7.2 Conclusion

In this thesis we considered the typical functionality of current systems. Most of them by nature are based on predefined assumptions about their defined states and their intended platforms. They are just focused on the functionality of the underlying system, thus they do not take into account many actions that could be reasoned upon in order to learn about the current user's QoE and improve one or more services offered by the system. Systems actually should not be "blind" machines with little or no knowledge

about their environments. We proposed a different way to address these problems in the specific context of Graphical User Interfaces. We have decomposed the problem in the well-known autonomic components with the aim of solve some problems related to the user-to-GUI interaction. We have developed such a GUI with several widgets, where each widget was associated with one or more QoE failure detection rules. Furthermore such rules aimed to prove that in fact it is possible to reshape a GUI or modify some functionality system aspects, enhancing the total user experience as a main goal. Under the limited experiments carried out here, we have demonstrated that in practice the total experience of both subjects was improved. In other words, we have proved that registering a portion of the domain user actions and applying rules then, it is feasible to increase user's benefits, what it means that our work is useful and also that all goals fixed at the beginning of this thesis have been achieved.

As final comment I would like to conclude highlighting that this work has been an educational, constructive, innovative and above all rewarding experience.

8 Bibliography

- [1] http://en.wikipedia.org/wiki/Autonomic_Computing.
- [2] <http://www.research.ibm.com/autonomic>.
- [3] <http://www-03.ibm.com/press/us/en/pressrelease/464.wss>.
- [4] <http://en.wikipedia.org/wiki/Dependability>.
- [5] http://en.wikipedia.org/wiki/Quality_of_experience.
- [6] “An adaptive OSGi robotic application”, Pintens Pieter-Jan.
- [7] Libc11n manual.
- [8] “The L Programming Language or Tcl for C Programmers”, Oscar Bonilla, Tim Daly, Jr., Larry McVoy.
- [9] <http://www.javacoffeebreak.com/articles/serialization/index.html>.
- [10] <http://edn.embarcadero.com/article/26970>.
- [11] <http://discuss.itacumens.com/index.php?topic=38945.0>
- [12] “Experimental study of adaptive user interfaces”, Pat Langley
- [13] “Machine learning for adaptive user interfaces”, Pat Langley
- [14] “Developing an adaptive user interface in Eclipse”, Alex Leung, Scott Morisson, Matt Wringe and Ying Zou

