

UNIVERSITY OF PATRAS
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING
ELECTRONICS AND COMPUTERS FIELD



DESIGN OF A LOG-FILE MANAGEMENT TOOL FOR EYE-TRACKING DATA

DIPLOMA THESIS OF
SAMUEL HERRERO GARCÍA
STUDENT OF THE UNIVERSIDAD DE VALLADOLID

SUPERVISOR: PROF.N.AVOURIS

DIPLOMA THESIS NUMBER
PATRAS 4, JULY 2011

It is certified that Diploma Thesis with the title:

DESIGN OF A LOG-FILE MANAGEMENT TOOL FOR
EYE-TRACKING DATA

Of the student of the University of Valladolid

Herrero García
(Surnames)

Samuel
(Name)

was presented in public at the Department of Electrical and Computer Engineering of the University of Patras, Greece on July 4, 2011.

The supervisor

The head of the Electronics and Computers division

N. AVOURIS

E.CHOUSOS

Abstract

In this project I have tackled the design and implementation of a management tool for the *eye-tracking* data. The main purpose is related with the huge amount of data and metrics the user have to deal with so as to provide a *graphical user interface (GUI)* that will enable shortening as a filter does.

Normally, the main labour after recording with the *Tobii* monitor of the department generally is usual to write all the information as data output *.tsv* file format readable. The overall issue is that commonly have large files, making endless work of finding the data and metrics the user is looking for. Due to this, is why we have tried to find a solution by implementing a *User Interface (UI)* based on the *Python* language programming. For this special task, I have used one of the most used graphical developing libraries, called *wxPython*. I have used *IDLE* as an integrated development environment (*IDE*) for *Python*, which has been very useful as it has text editor with syntax highlight and interactive shell.

Likewise, there are a lot of different *libraries, parsers, and modules* implemented in the code of the application, in order to manage in a correct way other equally important tasks that differ from the design.

Intentionally, and throughout the reading of this thesis, is explained in different chapters how this useful tool can provide the user by a clear and neat way, a shorten list of metrics available in a *.tsv* format file. There are mainly five chapters in which are divided this thesis: *Introduction, Theory and Analysis, Design, Development and Implementation*, and finally there is the *Evaluation* done with *Results and Conclusions*.

Keywords: eye-tracking, graphical user interface (GUI), Tobii, user interface (UI), Python, wxpython, IDLE, IDE, .tsv format, module, parser.

Acknowledgments

Firstly, I would not know how to thank all those who have been with me not only during these 5 months, but also during this long but beautiful stage, so I believed that there is no better way to shape in this thesis a few words of gratitude.

Thanks to Irina Chounta, my supervisor, she has been always in a great mood, you have been a great support during these months, constantly taking care and supervising my work. I am very much indebted to her, for her motivation, encouragement and guide, resulted in becoming interested in a new language programming as Python.

Likewise, I would like to thank all the members of the HCI group I have been enrolled with, because they have allowed me to work in their laboratory and making me very comfortable; Ioannis Ioannidis, a person that have been very kind in all moments, spreading happiness during the days I was upset with Python, providing anything I need, and of course, because of our musical exchange; Christos Sintoris who I have seen a lot of days supervising the work of my Spanish colleges, and who has been always interested in how I was going on; Filio Vogiantzi, who has helped me with every kind of administrative papers. I want also to thank Professor Nikolaos Avouris (my Erasmus coordinator in Patra), Dimitros Raptis and Eleftherios Papachristos for their good reception when I came to Patra.

Similarly, I would like to thank the University of Valladolid for giving me the opportunity to enjoy the experience lived abroad. Thanks to all the people I've met from other countries, all my Erasmus friends, I've lived an unforgettable experience and all thanks to an excellent company serves from them. I will never forget each one of you.

Finally, I thank of course the most important part, never stopped supporting me and pushing me to the finish: my family. Thanks to my parents, María Luz, and José María. Mam you have always made me see things as they are, aware every time what it takes to achieve them, and dad, also thank you for being at every good or bad moment supporting me in everything. Undoubtedly, you have given me everything. And of course, my sister Berta (thank you for making me smile), grandmother (this report goes for you, I did it!), and other members of the family. I love you

Contents

Abstract.....	5
Acknowledgments	7
CHAPTER 1	12
INTRODUCTION	12
1.1 MOTIVATION	12
1.2 BACKGROUND	13
CHAPTER 2	15
THEORY AND ANALYSIS	15
2.1 INTRODUCTION	15
2.2 WHY PYTHON	15
2.3 WXPYTHON	16
2.3.1 IDLE WXPYTHON	17
2.3.2 WXWIDGETS AS A GUI LIBRARY	18
2.4 EYE-TRACKER MONITOR.....	18
CHAPTER 3	21
DESIGN	21
3.1 INTRODUCTION	21
3.2 CREATING AND USING THE <i>TOP-LEVEL</i> WINDOW OBJECT	23
3.2.1 WORKING WITH WX.FRAME	23
3.3DESIGNING THE NEW APP GUI.....	28
3.2.1 INITIAL WINDOW	28
3.2.2 SECOND WINDOW	31
3.2.3 THIRD FRAME	34
3.3 UNDERSTANDING THE APP OBJECT LIFECYCLE AS PART OF THE DESIGN	37
3.4 HOW THE APP WORKS	38
CHAPTER 4	40
DEVELOPMENT AND IMPLEMENTATION	40
4.1 INTRODUCTION	40
4.2 FEATURES IMPLEMENTED	40
4.2.1 PYTHON MODULES AND LIBRARIES DESCRIPTION	40
4.2.2 WORKING IN A EVENT-DRIVEN ENVIRONMENT	70
CHAPTER 5	75
EVALUATION.	75
5.1 INTRODUCTION	75
5.2 SELECTING SUBJECTS IN <i>TA</i>	76

5.2.1 CRITERIA FOR SELECTING SUBJECTS	76
5.2.2 EXPERTS AS SUBJECTS.....	76
5.3 EVALUATION	77
5.4 RESULTS.....	78
5.5 CONCLUSIONS	80
CHAPTER 6.....	81
CONCLUSIONS	81
References	¡Error! Marcador no definido.
GLOSSARY	83

LIST OF FIGURES

Figure 1. Raymond Dodge's Photochronograph	13
Figure 2. Python logo	15
Figure 3. Python performance model in regard to other programming languages	16
Figure 4. wxPython logo	17
Figure 5. Screenshot of the IDLE interface of Python	18
Figure 6. Tobii T60 Eye tracker	19
Figure 7. Technical Specification of T60 Eye tracker	20
Figure 8 How to design a GUI. Disorder frames vs. clear Window	22
Figure 9. Interconnections between wx.Frame() methods	24
Figure 10. the extended style wx.help.FRAME_EX_CONTEXTHELP	27
Figure 11. Thestyle=wx.DEFAULT_FRAME_STYLE wx.FRAME_TOOL_WINDOW.	27
Figure 12. The first window (frame) of the application and the related mechanics	30
Figure 13. Elements positioning in the second window (frame)	32
Figure 14. The wx.BoxSizer() functionality	32
Figure 15. Screenshot from the second window of the application	34
Figure 16. Schematic of the wxPython application structure	35
Figure 17. Screenshot of the third window of the application	37
Figure 18. The life-cycle of the application	38
Figure 19 Example of a <i>tsv</i> format file	39
Figure 20. Modules and libraries used in the application	40
Figure 21 SWIG wrapper for Python	42
Figure 22Size Matters in time Boost vs. SWIG	42
Figure 23 Runtime test. Boost	
Figure 24 Panel method in wx module	44
Figure 25 Button in different platforms	49
Figure 26 Button with <i>wx.ToolTip</i> method	50
Figure 27 <i>wx.ListCtrl()</i> method with two different lists	54
Figure 28 <i>wx.CheckBox</i> with two different states	55
Figure 29 Three different instances of <i>wx.TextCtrl</i> method	57
Figure 30 Alert message implemented with Message Dialog	59
Figure 31 Error message implemented with Message Dialog	60
Figure 32 Information message implemented with Message Dialog	60
Figure 33 Question message implemented with Message Dialog	61
Figure 34 <i>wx.FileDialog</i> method with <i>os.getcwd</i>	66
Figure 35 A schematic of the event handling cycle, showing the life of the main program, a user event, and dispatch to handler fuctions	71
Figure 36 Even handling process, starting with the event triggered, and moving through the steps of searching for a handler	72

LIST OF TABLES

Table 1. Description of the parameters used by the constructor	25
Table 2. Styles of the wx.Frame() method	28
Table 3 wx.SizerFlags in WX module	47
Table 4 <i>wx.ListCtrl</i> window styles	51
Table 5 <i>wx.ListCtrl</i> events handling	51
Table 6 wx.CheckBox window styles	55
Table 7 OS module styles windows	65

CHAPTER 1

INTRODUCTION

1.1 MOTIVATION

The motivation for this thesis was the limited number of tools that manipulate data recorded from eye tracking monitors during various studies regarding analyzing interfaces, measuring usability, gaining insight into human performance as well as an actual control medium within a human-computer dialogue. The two areas have generally been reported separately; but the aim purpose seeks to tie them together. For usability analysis, the user's eye movements while using the system are recorded and later analyzed retrospectively, but the eye movements do not affect the interface in real time. As a direct control medium, the eye movements are obtained and used in real time as an input to the user-computer dialogue. The main purpose that will be discussed later, will try to facilitate the user a graphical way, to obtain in such a simple and accurate way, the huge amount of data collected during the recording with the *Eye-Tracker* monitor.

Interestingly, the principal challenges for both retrospective and real time eye tracking in *Human Computer Interaction (HCI)* turn out to be analogous. For retrospective analysis, the problem is to find appropriate ways to use and interpret the data; here becomes the main goal of this *Thesis*, empowering the individual through a *user graphical interface (GUI)*, and through a tool such as this application, so as to analyze the data in an efficient, fast and clear way, trying to facilitate the comprehension of the data obtained. And for real time use, the problem is to find appropriate ways to respond judiciously to eye movement input, and avoid over-responding; it is not nearly as straightforward as responding to well-defined, intentional mouse or keyboard input, but this last part we will not mention it rather than, because it has not come to appreciate neither been studied in depth.

These uses of eye tracking in *HCI* have been highly promising for many years, but progress in making good use of eye movements in *HCI* has been slow to date. We see promising research work, but we have not yet seen wide use of these approaches in practice or in the marketplace. We will describe the promises of this technology, its limitations, and the obstacles that must still be overcome. Work presented in this *Thesis* and elsewhere shows that the field is indeed beginning to flourish.

1.2 BACKGROUND

The study of eye movements pre-dates the widespread use of computers by almost 100 years (for example, *Javal, 1878/1879*). Beyond mere visual observation, initial methods for tracking the location of eye fixations were quite invasive – involving direct mechanical contact with the cornea. *Dodge and Cline(1901)* developed the first precise, non-invasive *eye tracking* technique, using light reflected from the cornea. Their system recorded only horizontal eye position onto a falling photographic plate and required the participant's head to be motionless. Shortly after this, *Judd, McAllister & Steel(1905)* applied motion picture photography to record the temporal aspects of eye movements in two dimensions. Their technique recorded the movement of a small white speck of material inserted into the participants' eyes rather than light reflected directly from the cornea. These and other researchers interested in eye movements made additional advances in *eye tracking* systems during the first half of the twentieth century by combining the corneal reflection and motion picture techniques in various ways (see *Mackworth & Mackworth, 1958* for a review).

Raymond Dodge's
Photochronograph
(1871-1942)

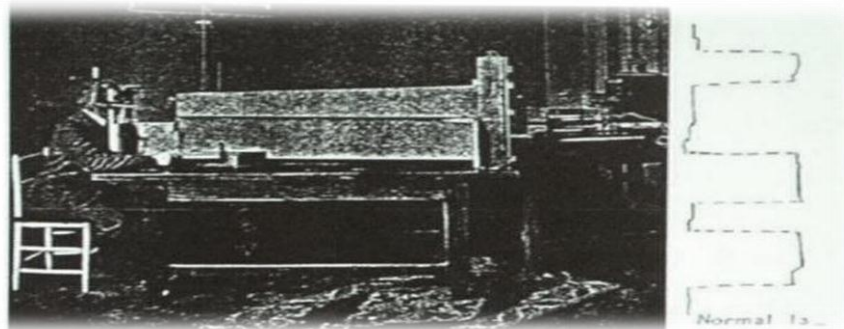


Figure 1. Raymond Dodge's Photochronograph

Much of the relevant work in the 1970s focused on technical improvements to increase accuracy and precision and reduce the impact of the trackers on those whose eyes were tracked. The discovery that multiple reflections from the eye could be used to dissociate eye rotations from head movement (*Cornsweet and Crane, 1973*) increased tracking precision and also prepared the ground for developments resulting in greater freedom of participant movement. These developments translated into innovation, was an essential precursor to the use of eye tracking data in real-time as a means of *human-computer interaction* (*Anliker, 1976*). Nearly all eye tracking work prior to this used the data only retrospectively, rather than in real time (in early work, analysis could only proceed after film was developed). The technological advances in eye tracking during the 1960s and 70s are still seen reflected in most commercially available *eye tracking* systems today.

Psychologists who studied eye movements and fixations prior to the 1970s generally attempted to avoid cognitive factors such as learning, memory, workload, and deployment of attention. Instead their focus was on relationships between eye movements and simple visual stimulus properties such as target movement, contrast, and location. Their solution to the problem of higher-level cognitive factors was “to

ignore, minimize or postpone their consideration in an attempt to develop models of the supposedly simpler lower-level processes, namely, sensorimotor relationships and their underlying physiology” (Kowler, 1990, p.1). But this attitude began to change gradually in the 1970s. While engineers improved eye tracking technology, psychologists began to study the relationships between fixations and cognitive activity. This work resulted in some rudimentary, theoretical models for relating fixations to specific cognitive processes. Of course scientific, educational, and engineering laboratories provided the only home for computers during most of this period.



So eye tracking was not yet applied to the study of *human-computer interaction* at this point. Teletypes for command line entry, punched paper cards and tapes, and printed lines of alphanumeric output served as the primary form of human-computer interaction. As *Senders (2000)* pointed out, the use of eye tracking has persistently come back to solve new problems in each decade since the 1950s. *Senders* likens eye tracking to a Phoenix raising from the ashes again and again with each new generation of engineers designing new eye tracking systems and each new generation of cognitive psychologists tackling new problems. The 1980s were no exception. As personal computers proliferated, researchers began to investigate how the field of *eye tracking* could be applied to issues of human-computer interaction. The technology seemed particularly handy for answering questions about how users search for commands in computer menus. The 1980s also ushered in the start of eye tracking in real time as a means of *human-computer interaction*. Early work in this area initially focused primarily on disabled users. In addition, work in flight simulators attempted to simulate a large, ultra-high resolution display by providing high resolution wherever the observer was fixating and lower resolution in the periphery (*Tong, 1984*). The combination of real-time eye movement data with other, more conventional modes of user-computer communication was also pioneered during the 1980s.

In more recent times, *eye tracking* in *human-computer interaction* has shown modest growth both as a means of studying the usability of computer interfaces and as a means of interacting with the computer. As technological advances such as the Internet, e-mail, and videoconferencing evolved into viable means of information sharing during the 1990s and beyond, researchers again turned to *eye tracking* to answer questions about usability and to serves a computer input device.

CHAPTER 2

THEORY AND ANALYSIS

2.1 INTRODUCTION

This chapter arguments on the need of a tool that allows the user to collect information obtained from *.tsv* files (text files that use *tab* as separator)where the *Eye-Tracker* data are logged. According to users' experience, the *.tsv* files were difficult to use and manipulate, mainly because of their long size. In common scenarios of use, the time of recording from the *Eye-Tracker* monitor, took about mostly half an hour, traduced to take files with a huge amount of data. Besides, trying to sort some metrics, or even values, becomes increasingly tedious.

The language programming chosen in this project is Python. It is a simple but very efficient actual language that provides a lot of useful utilities. Ease of file handling is one of the reasons that this language is used for the purposes of this diploma thesis. The *User Graphical Interface (GUI)* is provided with the *wxPython* toolkit.

2.2 WHY PYTHON

The current *Python* version installed and used to develop this Project is from the *2.7.xseries*, being the *2.7.1* the selected one.

Python is an *interpreter*, *general-purpose high-level programming language* whose design philosophy emphasizes code readability. *Python* aims to combine "remarkable power with very clear syntax", and its standard library is large and comprehensive. Its use of indentation for block delimiters is unique among popular programming languages.



Figure 2. Python logo

Python supports multiple programming paradigms, primarily but not limited to *object-oriented*, *imperative* and, to a lesser extent, functional programming styles. It features a fully *dynamic* type system and automatic memory management, similar to that of *Scheme*, *Ruby*, *Perl*, and *Tcl*. Like other dynamic languages, *Python* is often used as a scripting language, but is also used in a wide range of non-scripting contexts.

The reference implementation of *Python* (*CPython*) is free and open source software and has a community-based development model, as do all or nearly all of its alternative implementations. *CPython* is managed by the non-profit *Python Software Foundation*.

Python interpreters are available for many operating systems, and *Python* programs can be packaged into stand-alone executable code for many systems using various tools.

Moreover, the tool used to make the executable code is the *py2exe* programme that allows packaging all the source files into the *.exe* file.

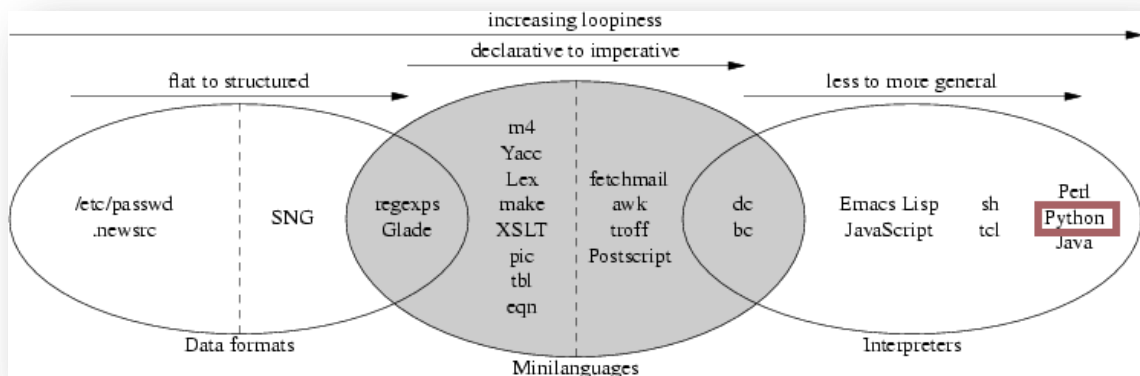
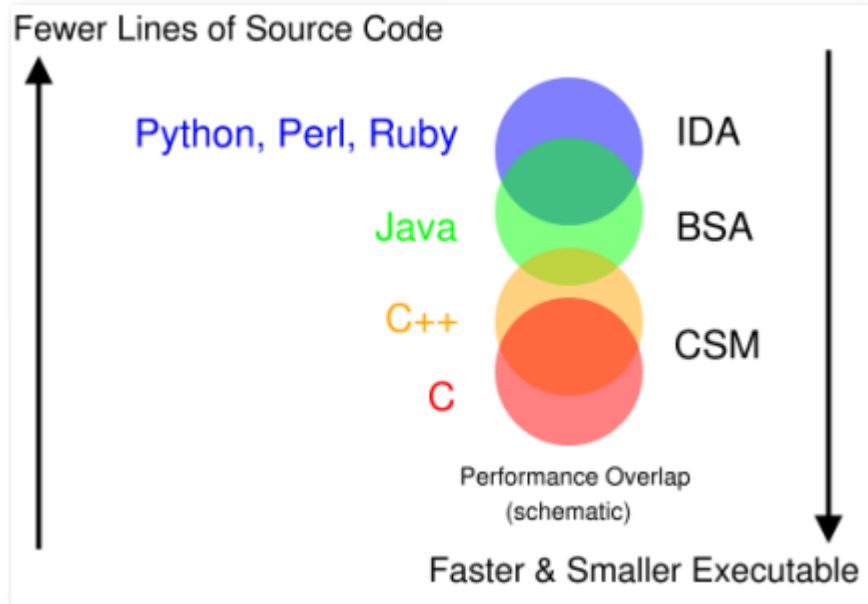


Figure 3. Python performance model in regard to other programming languages

2.3 WXPYTHON

WxPython is a *GUI* toolkit for the *Python* programming language. It allows *Python* programmers to create programs with a robust, highly functional graphical user interface, simply and easily. It is implemented as a *Python* extension module (native code) that wraps the popular *wxWidgets* cross platform *GUI* library, which is written in C++.

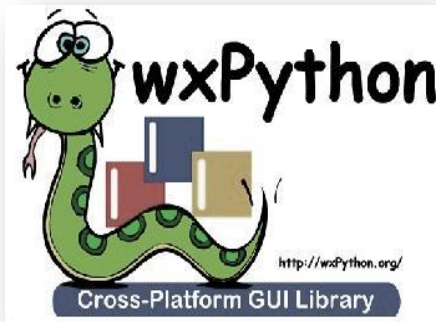


Figure 4. wxPython logo

As we have said, like *Python* and *wxWidgets*, *wxPython* is *Open Source* which means that it is free for anyone to use and the source code is available for anyone to look at and modify. Or anyone can contribute fixes or enhancements to the project.

WxPython is a cross-platform toolkit. This means that the same program will run on multiple platforms without modification.

Currently supported platforms are 32-bit *Microsoft Windows*, most *Unix* or *Unix-like systems*, and *Macintosh OS X*. The platform where the *User Graphical Interface (GUI)* has been developed is *Windows 7 Home Premium* with the *Service Pack 1* installed. The version is the 2.8.11.0, one of the most famous and extended released of the *wxpython* developments.

For *Python* language used, *wxPython* programs are simple, easy to write and easy to understand.

2.3.1 IDLE WXPYTHON

IDLE is an integrated development environment for *Python*, which is bundled in each release of the programming tool since 2.3. It is not included in the python package included with many Linux distributions. It is completely written in *Python* and the *Tkinter GUI toolkit* (wrapper functions for *Tcl/Tk*).

Its main features are:

- Multi-window text editor with syntax highlighting, auto completion, smart indent and other.
- *Python* shell with syntax highlighting.
- Integrated debugger with stepping, persistent breakpoints, and call stack visibility.

IDLE is often criticized for its non-standard keyboards shortcuts (unique to *IDLE*, with only a manual schema editor available) and lack of line numbering as an option. Nevertheless, this lack of keyboards shortcuts or even the line numbering, has not been an impediment for the development of this project. The version installed for the *Python's Integrated Development Environment (GUI)* is the 2.7.1.



Figure 5. Screenshot of the IDLE interface of Python

2.3.2 WXWIDGETS AS A GUI LIBRARY

WxWidgets gives a single, easy-to-use *API* for writing this *GUI* application on even multiple platforms that still utilize the native platform's controls and utilities. It is linked with the appropriate library for the platform used (Windows/Unix/Mac) and compiler (almost any popular C++ compiler), and the application done adopts the look and feel appropriate to that platform. In this Project, the application has been running in different platforms, in order to know the different look for each one.

On top of great *GUI* functionality, *wxWidgets* gives: online help, network programming, streams, clipboard and drag and drop, multithreading, image loading and saving in a variety of popular formats, database support, HTML viewing and printing, and different other utilities.

The version installed for the development of this *Project* has been the 2.8.11 series.

2.4 EYE-TRACKER MONITOR

The device that provides information and data to be treated and analyzed in our *App* for *eye tracking* comes from the monitor used in the *Human Computer Interaction (HCI)* group in the *University of Patras*.

Furthermore, without this tool could not be possible to develop any *log-file* in order to give further information about the data gathered throughout all the process of *eye tracking*.

Below is detailed some information about the monitor placed in this department as a major part of the framework of this *Project*.

The exact model of the *Tobii monitor* used for the analyzing data *log-file tool* developed comes from the *T Series*, being the model *T60* the one for *eye-tracking*.

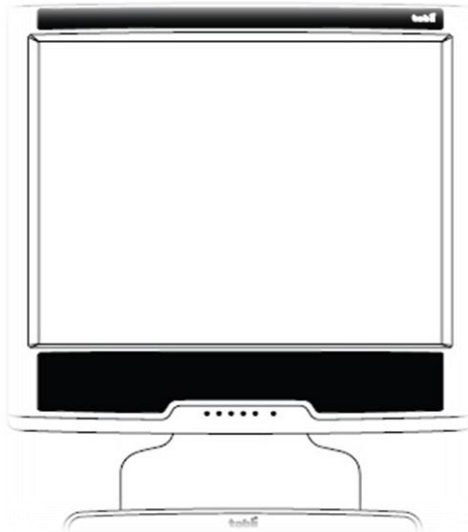


Figure 6. Tobii T60 Eye tracker

Tobii T60 Series® brings user-centered design into eye tracking technology, making eye tracking studies easily accomplished. The *T60 Series* gives accurate eye gaze data and provides an even more natural environment for the subject to ensure realistic responses.

Since it is fully plug-and-play, the eye tracker can be set up in minutes. Time and cost efficient automatic tracking enables *eye tracking* in a lot of different areas of application. In our case, it has been used as a method of working that will allow the technical user in futures studies to easier understand and use the data collected.

Tobii T60 Eye Tracker® is integrated into a 17" TFT monitor. It is one of the most efficient monitors in the actual market for all forms of eye tracking studies using screen based stimuli.

Overview & Specification of the T60 Eye-Tracker monitor [1]

Below is detailed some specifications of the *Tobii* monitor used in the *Human Computer Interaction (HCI) Department* of the *University of Patras*.

✓ Plug-and-Play

Tobii T60Eye Tracker® carries out long lasting, robust calibrations in seconds. Tracking is fully automatic and started by issuing a simple command. Advanced technology and software saves the need for expertise. Large studies can be conducted in an-efficient manner.

✓ Non-intrusive

Tobii T60Eye Tracker® has no visible or moving '*tracking devices*' that might affect the subject. The exceptionally large freedom of head movement allows respondents to behave naturally in front of the screen. You can perform long and accurate studies without fatigue.

✓ High tracking quality

High accuracy and excellent head movement compensation ensures high quality data throughout your entire population. *Tobii T60 Eye Tracker®* tracks basically everyone, regardless of ethnic origin, age, glasses or contact lenses. Drift compensation guarantees high tolerance to varying light conditions.

Technical Specification	
Tobii T60 Eye Tracker	
Accuracy	0.5 degrees
Drift	< 0.3 degrees
Freedom of Head Movement	44x22x30cm
Data Rate	60Hz
Binocular Tracking	Yes
Bright/Dark Pupil Tracking	Both - automatic optimization
TFT Display	17" TFT, 1280x1024Pixels
Eye Tracking Server	Embedded
Weight	~10kg(22lbs)
User Camera	Built in
Speakers	Built in

Figure 7. Technical Specification of T60 Eye tracker

CHAPTER 3

DESIGN

3.1 INTRODUCTION

This chapter focuses on the new *User Interface UI* design of the application. The design has been based on several comments recorded from user experts and even from experience in relation with the actual market in the design of applications.

It should be mentioned that the purpose of the development of this *App* is based on the idea of allowing the normal or even technical user to figure out in an easy way how it works, based on the main goal of enabling them to sort a number of metrics and values, that are recorded in huge *.tsv* files, as output data from *eye-tracking*.

Furthermore, the simplicity of the *App* it's related with the *wxpython language programming* that has been very useful in order to show in a clear but also successful and orderly way the contents in a *log-file management tool*.

Below is detailed as a point of view of *Pablo Garaizar from Deusto University [2]*, some of the main points a *User Interface (UI)* should be contained in each *App*.

✓ 1.- Effective Visual Communication for Graphical User Interfaces

The use of typography, symbols, color and other static and dynamic graphics are commonly used to express facts, concepts and emotions. This forms consistent graphic design-oriented information that helps people understand complex information. Effective visual communication system is based on some basic principles of *graphic design*.

✓ 2.- Design Considerations

Three factors may be considered for the design of a correct user interface, *development factors, feasibility factors and factors of acceptance*.

Development factors help to improve visual communication. This includes toolkits and libraries of components, supports rapid prototyping, and adaptability.

Feasibility factors into account human factors and express a strong visual identity. This includes people skills, product identity, a clear conceptual model, and multiple representations.

As *factors of acceptance* is the policy of the corporation, international markets, and documentation and training.

✓ 3.- The Visible Language

Visible Language refers to all the graphical techniques used to communicate the message or context. This includes:

- *Layout and Layout*: formats, aspect ratios, and meshes (grids); organization: either 2D or 3D.
- *Typography*: selection of fonts and styles, including fixed and variable width.
- *Color and Texture*: color, texture and luminance provide complex information and pictorial reality.
- *Images*: signs, icons and symbols, from real to abstract photographically.
- *Animation*: a dynamic or kinetic display: very important in the use of video-related images.
- *Sequencing*: sequencing the roughly total visual sequencing logic.
- *Visual Identity*: additional rules that provide consistency and unique set of user interface.

✓ 4.- Principles of User Interface Design

There are three fundamental principles relating to the use of language visible:

- *Organize*: provide the user with a clear and consistent conceptual structure.
- *Economize*: do the most with the least amount of items.
- *Communicate*: adjust the presentation to the user's capabilities.

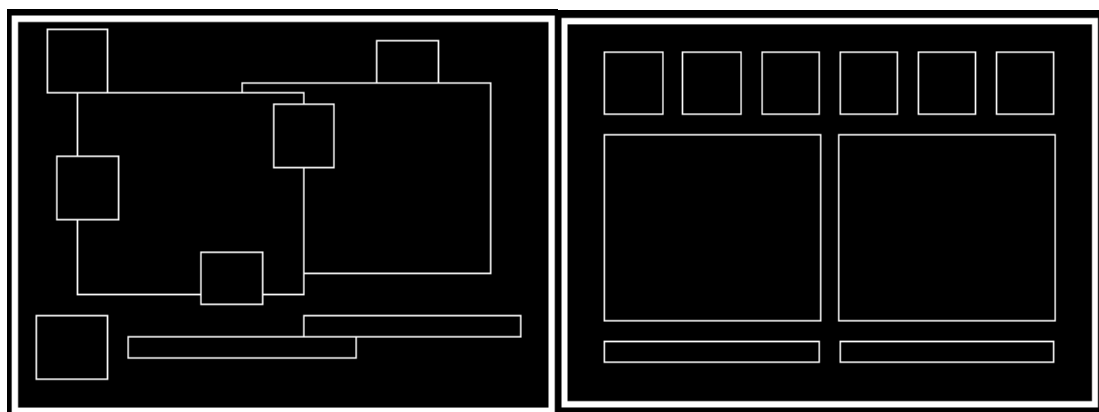


Figure 8 How to design a GUI. Disorder frames vs. clear Window

3.2 CREATING AND USING THE *TOP-LEVEL* WINDOW OBJECT

In order to describe the design in this eye-tracking tool, we attempt a brief review of the most basic aspects in the development of this application.

A *top-level* window object is a *widget* (usually a *frame*) that is not contained by another widget in the application –it is what a typical user would point to and says, “*That’s the program.*”[1] The *top-level* window object is usually the *main window*, in this case, all the modules created, as part each one of the entire *log-file management tool App*, and contains *widgets* and interface objects that the *user interacts* with.

The *application* must have at least one *top-level* window object. The *top-level* window object is usually a subclass of the class *wx.Frame* from *wxpython* library, although it can also be a subclass of *wx.Dialog()*. Besides there are defined a lot of custom subclasses of *wx.Frame* inside the application throughout the developing of this *Project*. However, there are a number of *pre-defined wx.Dialog* subclasses that provided many of the typical dialogs that were very useful and helpful to encounter in the *application*.

As a matter of fact, there is some naming confusion here, due to overloading of the word “top”. A generic “*top-level*” window is any *widget* in the application that doesn’t have a parent container. So this is why the *application* programmed must have at least one of these, but it can have as many as we would like. Only one of these windows, however, can be explicitly defined by *wxPython* as the main *top-window* by using *SetTopWindow()*. If there is not specify a *main window* with *SetTopWindow*, then the first frame in the *wx.App*’s *top-level window* list is considered to be the top window.

Therefore, explicitly specifying the *top window* is not always necessary –as it is not needed to if, for example, there is only one top window. Repeated calls to *SetTopWindow()* will replace the current *top-window*- an application can only have one *top-window* at a time.

3.2.1 WORKING WITH WX.FRAME

In this subsection we discuss the most important element of each of the modules created for this *application*: *wx.Frame()* method derived from the *wxpython* library.

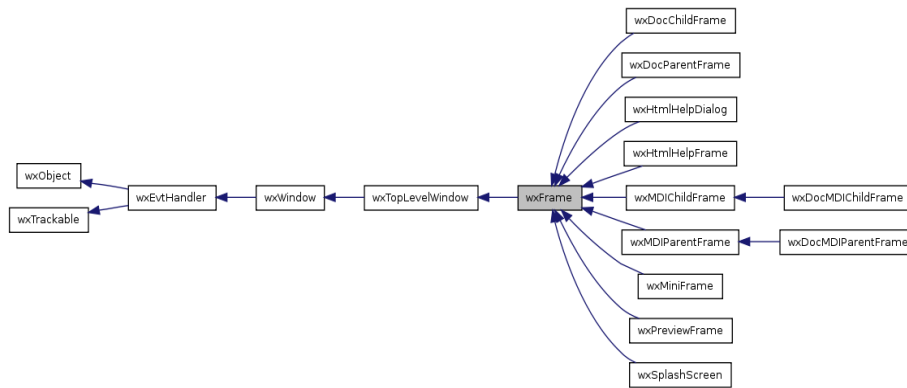


Figure 9. Interconnections between wx.Frame() methods

In *wxPython* parlance, a frame is the name given to what a *Graphical User Interface (GUI)* user normally calls a “window”[3]. That is to say, a frame is a container where the user can generally move freely around on the screen, and which often includes such decorations as a *title bar*, *menu bar*, and *resize targets* in the corners. **The class *wx.Frame* is the parent class of all frames in *wxPython*.** There are also a few specialized subclasses of *wx.Frame* that you may use. This section will give an overview of the *wx.Frame* family.

When subclasses of *wx.Frame* are created, the `__init__()` method of in the class should call the parent constructor *wx.Frame.__init__()*. The signature of that constructor is as follows.

```

wx.Frame(parent, id=-1, title="Eye-Tracking", pos=wx.DefaultPosition,
         size=wx.DefaultSize, style=wx.DEFAULT_FRAME_STYLE,
         name="frame")
    
```

This constructor takes several parameters. In normal use, however, at least some of the defaults are reasonable options. Afterwards we will see parameters similar to this constructor again and again in other widget constructors used in this *App* as part of its design.

Below is shown a table that contains description of each of the parameters[3]:

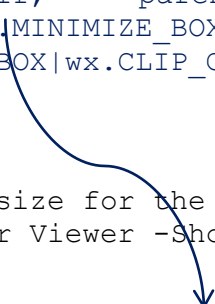
Parameter	Description
parent	The parent window of the frame being created. For top-level windows, the value is None. If another window is used for the parent parameter then the new frame will be owned by that window and will be destroyed when the parent is. Depending on the platform, the new frame may be constrained to only appear on top of the parent window. In the case of a child MDI window, the new window is restricted and can only be moved and resized within the parent.
id	The wxPython ID number for the new window. You can pass one in explicitly, or pass -1 which causes wxPython to automatically generate a new ID. See the section "Working with wxPython ID" for more information.
title	The window title—for most styles, it's displayed in the window title bar.
pos	A wx.Point object specifying where on the screen the upper left-hand corner of the new window should be. As is typical in graphics applications, the (0, 0) point is the upper left corner of the monitor. The default is (-1, -1), which causes the underlying system to decide where the window goes. See the section "Working with wx.Size and wx.Point" for more information.
size	A wx.Size object specifying the starting size of the window. The default is (-1, -1), which causes the underlying system to determine the starting size. See the section "Working with wx.Size and wx.Point" for more information.
style	A bitmask of constants determining the style of the window. You may use the bitwise or operator () to combine them when you want more than one to be in effect. See the section "Working with wx.Frame styles" for usage guidelines.
name	An internal name given to the frame, used on Motif to set resource values. Can also be used to find the window by name later.

Table 1. Description of the parameters used by the constructor

These are the parameters as passed to the parent constructor method, `wx.Frame.__init__()`. The argument signature of the constructor to our class can, and often will, be different. This allows to conveniently ensuring default values for our own frame by not allowing them to be modified by a call to the constructor.

```
class MyFrame(wx.Frame):
    def __init__(self, parent, mytitle, mysize):
        wx.Frame.__init__(self, parent, wx.ID_ANY, mytitle,
                           size=mysize, style=wx.MINIMIZE_BOX|wx.SYSTEM_MENU|
                           wx.CAPTION|wx.CLOSE_BOX|wx.CLIP_CHILDREN)

app = wx.App(0)
    # set title and size for the MyFrame instance
mytitle= "Eye Tracker Viewer -Shorting list of Values-"
width = 850
height = 700
style=wx.RESIZE_BORDER
MyFrame(None, mytitle, (width, height)).Show()
app.MainLoop()
```



3.2.1.1 Working with wx.Frame styles as design in the App GUI

As it is described above, the `wx.Frame()` method is the first thing that should be instanced as a frame container where the user can generally move freely around on the screen. That is why this method gathered a lot of metrics or parameters, to which it is allowed to change the general perception of the window.

With this aim, there are parameters for this method, called *bitmask*. The `wx.Frame` constructor takes a *bitmask* as a style parameter. Every `wxPython` widget object takes a similar style parameter, although the exact values that are defined are different for each type of *widget*. This section will discuss the styles used for `wx.Frame`. At least some of this is applicable to other `wxPython` widgets used and instanced in this *GUI Application*.

Firstly we introduce a new term coined in computer jargon and widely used today in relation with `wx.Frame` constructor and its style parameters so as to design this *Log-File tool*. [3]

➤ What is a BITMASK?

A bitmask is a way of compactly storing information about system attributes that is especially useful when there are a limited number of attributes with boolean values and the values are more or less mutually independent. In wxPython, bitmasks are used to manage a number of different attributes throughout the framework, most notably style information. In a bitmask, the individual attributes are assigned constant values corresponding to powers of two, and the value of the bitmask is the sum of all the attributes which are “turned on”. In binary notation, the power of two system guarantees that each attribute corresponds to a single bit in the total sum, allowing all of the attribute state to be compactly stored in a single integer or long value. For example, if attribute $a=1$, $b=2$, $c=4$, and $d=8$, then any combination of the group has a unique sum that can be stored in an integer. The pair a and c would be 5 (binary 0101), while b , c , and d would be 14 (binary 1110). In wxPython, the attributes have symbolic constants, so you don’t need to worry about the individual bit values.

As a matter of fact, developing the *Graphical User Interface (GUI)* for this *Application* has been very useful to introduce the *bitmask* attribute. Styles are defined for all `wxPython` widgets by passing a *bitmask* to the style parameter of the constructor. Some widgets also define a `SetStyle()` method, allowing the changing in the style after the widget is created. All the individual style elements used have a predefined constant identifier (such as `wx.MINIMIZE_BOX`). To add multiple styles together, it is established the Python bitwise OR operator|. For example, the constant `wx.DEFAULT_FRAME_STYLE` is defined as a combination of basic style elements:

wx.MAXIMIZE_BOX		wx.MINIMIZE_BOX	
wx.RESIZE_BORDER		wx.SYSTEM_MENU	
wx.CLOSE_BOX		wx.CAPTION	

The next four figures show a few common frame styles:

Figure 11 was created with `wx.DEFAULT_STYLE`. Figure 10 is a frame created using the *non-resizable* style.



Figure 10.the extended style `wx.help.FRAME_EX_CONTEXTHELP`



Figure 11. The style=`wx.DEFAULT_FRAME_STYLE | wx.FRAME_TOOL_WINDOW`.

Below is detailed a list with the most important styles of the method `wx.Frame` as an important issue in our *App*:

Style	Description
<code>wx.DEFAULT_FRAME_STYLE</code>	As you might expect from the name, this is the default if no style is specified. It is defined as <code>wx.MAXIMIZE_BOX wx.MINIMIZE_BOX wx.RESIZE_BORDER wx.SYSTEM_MENU wx.CAPTION wx.CLOSE_BOX</code> .
<code>wx.FRAME_SHAPED</code>	Frames created with this style can use the <code>SetShape()</code> method to create a window with a non-rectangular shape.
<code>wx.FRAME_TOOL_WINDOW</code>	Makes the frame look like a toolbox window by giving it a smaller titlebar than normal. Under Windows a frame created with this style does not show in the taskbar listing of all open windows.
<code>wx.MAXIMIZE_BOX</code>	Adds a maximize box on the frame, using the system parameters for the look and placement of the box. Also enables maximize functionality in the system menu if applicable.
<code>wx.MINIMIZE_BOX</code>	Adds a minimize box on the frame, using the system parameters for the look and placement of the box. Also enables minimize functionality in the system menu if applicable.
<code>wx.RESIZE_BORDER</code>	Adds a resizable border to the frame.
<code>wx.SIMPLE_BORDER</code>	A plain border without decoration. May not work on all platforms.
<code>wx.SYSTEM_MENU</code>	Adds the system menu (with close, move, resize, etc. functionality, using system look and feel) and the close box to the window. The availability of resize and close operations within this menu depends on the styles <code>wx.MAXIMIZE_BOX</code> , <code>wx.MINIMIZE_BOX</code> and <code>wx.CLOSE_BOX</code> being chosen.

Table 2. Styles of the `wx.Frame()` method

3.3DESIGNING THE NEW APP GUI

As mentioned previously, the development of this *App* has to keep in mind several points of how a *user interface(UI)* should be implemented.

Never before, anyone has developed any kind of *App* related with the issue concerning with this *Project*, so there was no prior experience.

The *App* program consists basically in *three main frames*, interconnected with each one. Apart from this, there are a lot of *events* referred to *binding objects*, which show *message dialogs*, as it has been an important and deep concern for allowing the user to understand the mechanics of the application.

3.2.1 INITIAL WINDOW

The first, main frame consists mainly in a *browse-window* where the user can open a *.tsv* file with the data recorded from the *Eye-Tracker monitor*. This frame, as all the frames described below, has in common the style of the background. It had to be a clear, minimal window of a neutral color with a gray's flush. The gray scale is coded with the hexadecimal `#EEEEEE` and provides a clear view of all the *buttons*, *text path*, *panel*, *GridBagSizer* and *message dialogs* that it contains.

Firstly, as an introduction to this window, the common user will discover how easy it works by the opportunity of browsing a *.tsv* file. This main screen, is composed of a number of methods all collected in the module *menuv3*.

By way of explanation, there is a *Panel* that contains a *GridBagSizer* method, a very useful tool that comes from the *wxpython widgets*. The main purpose of this method is to allow a perfect disposition of all the widgets that are going to be placed in the panel.

There is also a *TextControl* method instanced which permits gathering the path that previously the user has chosen from the *button Browse*. After clicking this button, it appears a new *message-dialog*, very common in all type of *App*, showing the current folder with the files that are going to be selected.

Moreover, there are methods linked to all the buttons, called *SetToolTip()*, which are imported from the library of *wxpython*, and shows by dragging the mouse above them a little message as a short description of the binding method related with.

As a common issue in the development of the design for all frames in this *Project*, it has been settled a *non-resizable frame*, with just a close and minimize button placed on the upper-right corner of each frame.

Graphically and followed by these words is detailed the *menuv3.py* module as the main and first frame shown to the user.

As a result, here are detailed the designing methods used in this initial frame:

- ✓ *wx.Panel()*
- ✓ *wx.GridBagSizer()*
- ✓ *wx.TextControl()*
- ✓ *wx.messageDialogs()*
- ✓ *wx.SetToolTip()*
- ✓ *wx.Frame()*
- ✓ *wx.ShowModal()*
- ✓ *wx.Destroy()*

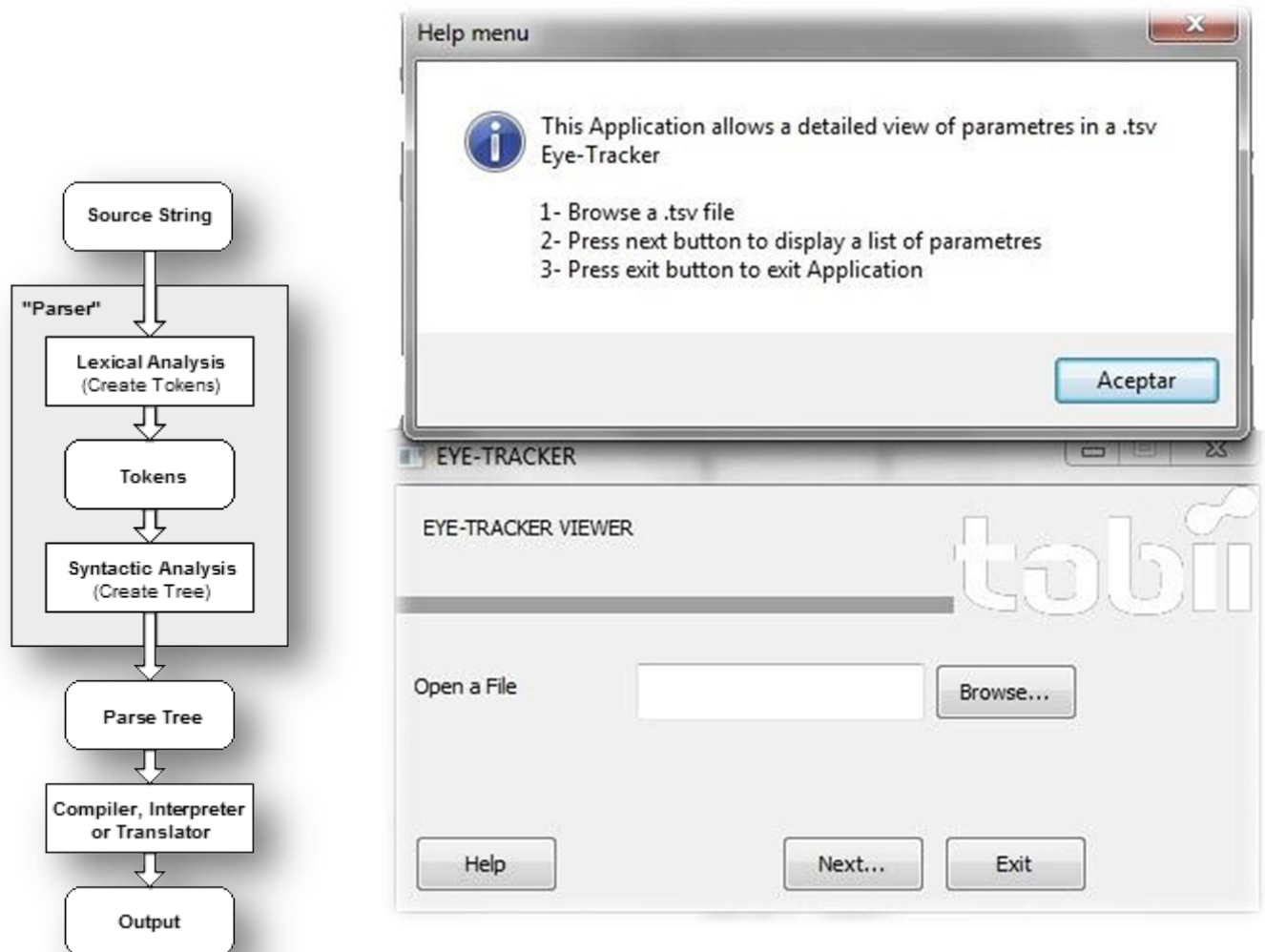


Figure 12. The first window (frame) of the application and the related mechanics

3.2.1.1 Parser TSV among Initial and Second frame

Despite not having any graphics implementation of this *Parser*, it should highlight the importance of including it in this chapter as part of the design, since without it, could not reach the *Second window module*.

In computing, a *parser* is one of the components in an interpreter or compiler, which checks for correct syntax and builds a *data structure* (in the case that occupies in this *Project* one hierarchical structure related with *metrics* from the *eye-tracking .tsv file*) implicit in the input tokens. The parser often uses a separate lexical analyzer to create tokens from the sequence of input characters. Parsers may be *programmed by hand* or may be *(semi-)automatically generated* (in some programming languages) by a tool. In this case it is *programmed by hand*.

This parser works so that once the user has selected the metrics collected from the selected path -remember it has to be a file with extension *.tsv*, with a large amount of

information and data- so that will be introduced in the *second window* as part of the list parameters of the *eye-tracking* file.

3.2.2 SECOND WINDOW

Likewise the initial window, it has tried to follow the same pattern trying to provide the user a clear and helpful frame, in order to figure out how the program works. It still has the same background color, with mainly two lists, one with the metrics gathered -from the *.tsv file* that in *menuv3.py* module, the user chose- and the other list with the parameters list selected from the list on the left.

Apart from these two lists that mostly occupy the entire window, there are some more *widgets* as buttons that allow full interoperability between them. There are three buttons that divide the two lists, one for sorting the list on the left by name, and other two, that simply offers the user to *Add or Delete* items from the list on the right.

The “*list*” method is contained in the code as *wx.ListCtrl()* from the *wxpython library* with also a linked method connected with. Is remarkable to say that one of the main goals on the designing of this Project was apart from the *usability* of it, but also provide *interaction* between Interface and User. These aspects were considered as ones of the most important goals while developing of the *Project*.

In addition, there are new methods included in the code of the module *SecondWindow.py* such *wx.ListCtrl()*, already mentioned before as the most remarkable method in the design of this frame, the *wx.Centre()* method that allows the frame to be positioned in the central part of the screen, *wx.messageDialogs()* as part of the *pop-ups* and *alert* messages, *wx.showModal()* and *wx.Destroy()* methods in related with this alerts or advising messages that permits them to appear and disappear when an *Ok* or *Cancel* button is clicked.

Briefly below it's shown the mainly methods used for the design of this frame:

- ✓ *wx.ListCtrl()*
- ✓ *wx.Centre()*
- ✓ *wx.messageDialogs()*
- ✓ *wx.showModal()*
- ✓ *wx.Destroy()*
- ✓ *wx.SetColumnWidth()*
- ✓ *wx.BoxSizer()*
- ✓ *wx.SetToolTip()*
- ✓ *wx.Frame()*

Similarly, as it has been detailed in the list above, one of the main characteristics of this frame it's connected with the placement of the items throughout the window.

Thanks to the method called *wx.BoxSizer()* from the *wxpython library*, we are provided by a highly useful design tool, that permits by a correct and orderly way to place buttons, lists and all kind of widgets that *wxpython* provides.

The way of going positioning each of the elements is described below, as a part of a *jigsaw puzzle* in which each of the elements are positioned within the *sub-windows*, *vertically or horizontally* and are added to parent window in a *hierarchical* way.

It has been very useful for the development of this frame, and third window also, though it takes a long time to think about the correct way of positioning each item along the *wx.BoxSizer()*.



Figure 13. Elements positioning in the second window (frame)

Below is described in a graphical way how the *wx.BoxSizer()* method works as a form of designing tool for this *secondwindow.py* module.

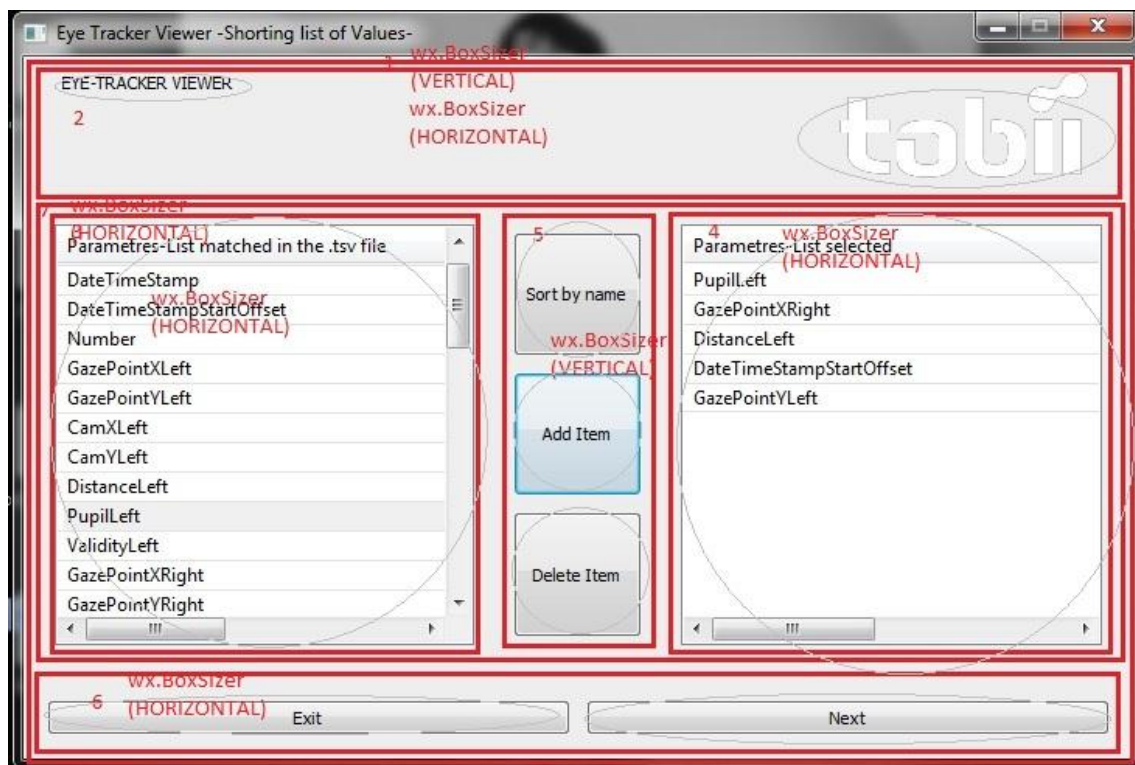


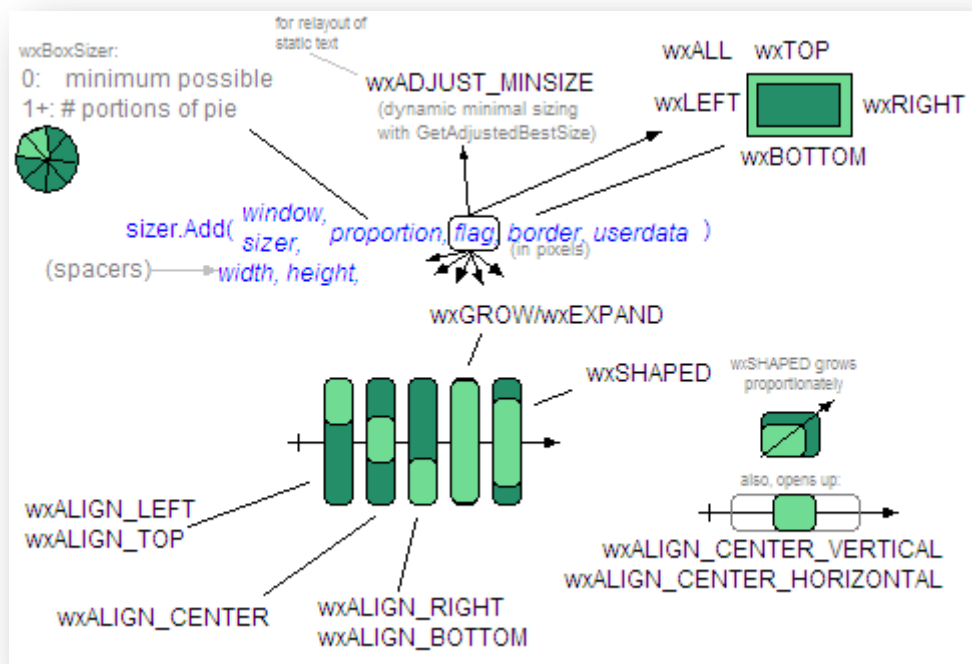
Figure 14. The *wx.BoxSizer()* functionality

As shown in the illustration above, we can see the importance of making a preview of how you can arrange each *boxSizer*, and as each one has a hierarchy among them. The preview was established in a separate sheet, as an outline or sketch.

Not only about the graphical way of explaining the designing of this frame is important but also the *flags* inside the `wx.BoxSizer()` method from *wxpython library* have to be mentioned as a description of how can the elements and the style of itself could be placed.

For instance, all buttons and even the two lists placed in the `wx.BoxSizer()` method are part of smaller structures, as `wx.BoxSizers()` which are positioned horizontally or vertically in a hierarchical way. Besides, they are added into the parent `wx.BoxSizer()` with the `wx.sizerAdd()` method instanced. As a result, we have a main *sizer* (n° 1 **in figure 14**) as the parent frame placed with *VERTICAL* flag; inside we have disposed three frames (n° 2, 6 and 7 **in figure14**). Number n°2 is a *HORIZONTAL* sizer with title of the App and logo. Number n°6 is also a *HORIZONTAL* sizer which contains *Exit* and *Next* buttons. Finally number n°7 is a *HORIZONTAL* sizer which itself contains another three sizers (n°3, 4 and 5 **in figure 14**). Number n° 3 and number n° 4 are added as `wx.ListCtrl()` methods, and n° 5 is another *sizer* with *VERTICAL* flag which contains three buttons.

Here it is a summary of how this method works while the elements are added to each `wx.BoxSizer()` instanced as `wx.SizerAdd()` method:



Furthermore, below is shown one of the few *alert messages* instanced in the *SecondWindow.py* module, as part of the design of this frame, which allows helpful displays as the user can figure out how is going through the App:

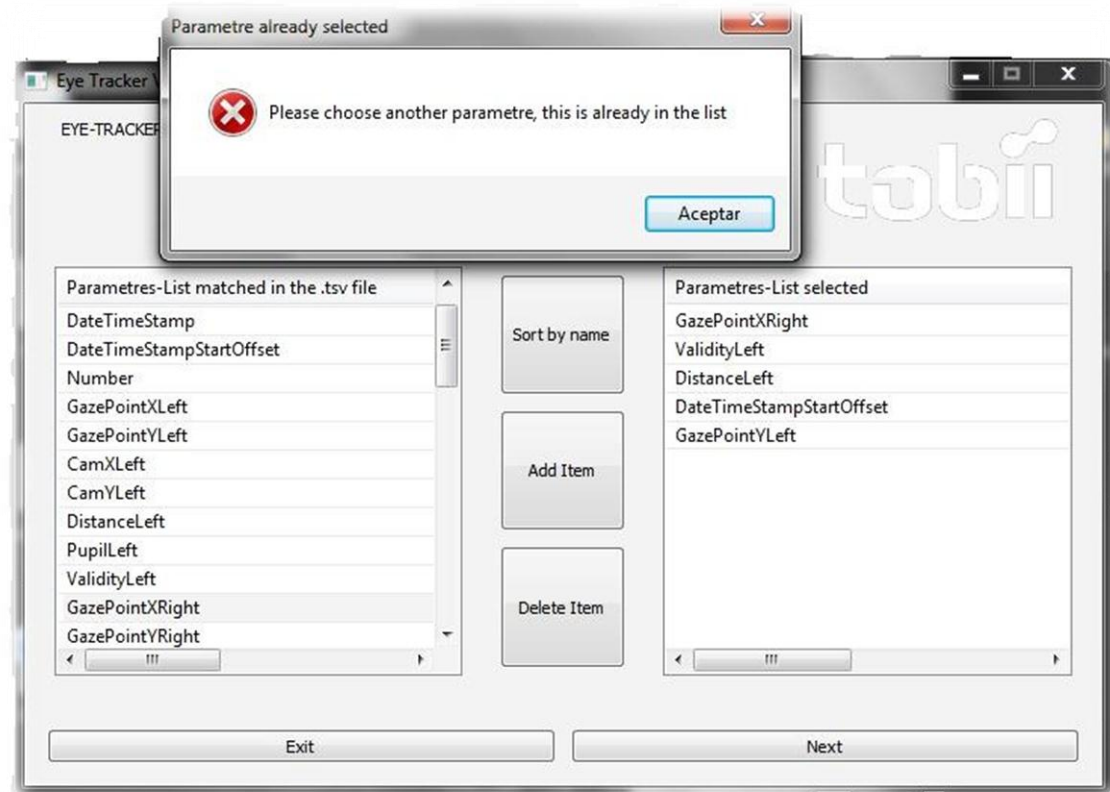


Figure 15. Screenshot from the second window of the application

3.2.3 THIRD FRAME

As a result, and as a consequence of the compiling and running *menuv3.py* and *secondwindow.py* modules, it can be displayed the third and last frame.

Therefore becomes a window that contains the previous window, a set of parameters chosen by the user, and arranged in a list. It is thus that it can be seen if the number of metrics collected exceeds the length of the list; an event resulted in a scrollbar, which allows access to all the items simply by dragging the mouse.

The arrangement of all elements is parallel to the *SecondWindow.py* module according to what was described in the preceding paragraph. So it is, that has been developed by inserting a *wx.BoxSizer()* method. As stated earlier, this method has clearly each *wxPython* widgets as it is provided to us. This is the mainly reason why finally it has been used this valuable tool.

Mainly we observe that on top of this frame, we find two *text boxes* provided by the method *wx.TextCtrl()* of *wxPython* library, where it controls two times. It is as much as one entry into the starting time which the user wants to use to begin scanning the data of each one of the metrics that you then select. On the other hand, immediately afterwards, is the second box, which will be introduced from the typing keyboard the last time.

It should highlight the importance of these two *text boxes* that serve as data entry time. This is mainly to provide a filter for all metrics selected and gathered into the list below; by adapting into maximum data information the user wants to define.

In addition, now we are going to explain an issue not so much involved in the design, but if an indirect way that influences on it. These are *events* produced by introducing the wrong value of time, or for example if the initial time is higher than the final time, even if times are not introduced in the range defined by the *file in .tsv format*.

This kind of events are instanced thanks to `wx.messageDialog()` widget from `wxpython` library. It displays a numerous *alert-messages*, with *pop-ups*, related with the `mainLoop()` method. It simply creates an instance of our application class, and then calls its `MainLoop()` method. `MainLoop()` is the heart of the application and is where events are processed and dispatched to the various windows in the application. Fortunately `wxWidgets` insulates us from the differences in event processing in the various GUI toolkits.

As the diagram below shows, the *application object* “owns” both the top-level window and the *main event loop*. The top-level window manages the components in that window, and any other data objects we assign to it. That window and its components trigger events based on user actions, and receive event notifications to make changes in the display.

A schematic of the basic `wxPython` application structure, showing the relationship between the application object, the top-level window, and the main event loop, is presented:

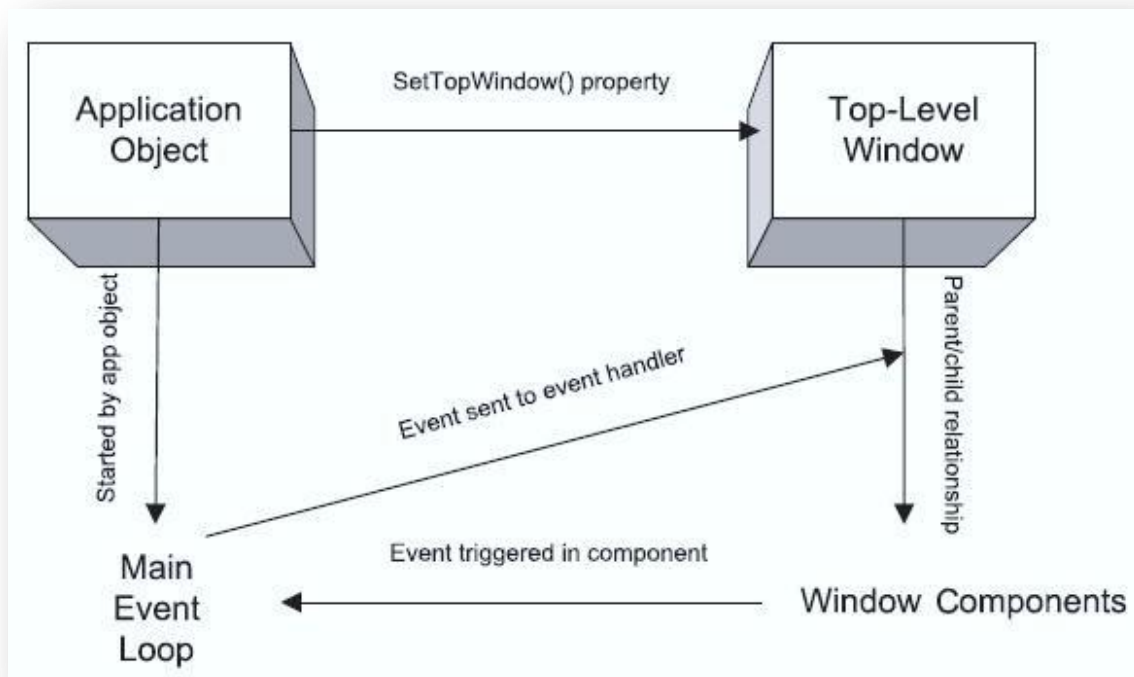


Figure 16. Schematic of the wxPython application structure

Nevertheless, after this subchapter, it will be explained how the lifecycle of the main program works, as a review of the importance of some methods, and as an important issue on the designing of this *App*.

Likewise some of the methods related with the development in the design of this frame, are connected with also a lot of methods used previously.

Worth noting the *wx.CheckBox()* method, as a widget used for binding an event that allows the user once finished to open the file chosen previously in a *browse button* placed in one *sizer*. Each of these different methods used as *widgets* from the *wxpython* library, are blinded with different type of events, which take into account the different method calling.

Furthermore there are plenty of options within each of the calls to the various events that take place at any time. Following is an example of how an alert message can be displayed in the screen, and the parameters taken for detailing how it should work.

```
defexitEvent(self, event):
    """Exit Program"""
    exitInfo = """Are you sure you want to exit?"""
    exitBox = wx.MessageDialog(self, message=exitInfo,
                              caption='Exit Eye-Tracker App',
                              style=wx.ICON_EXCLAMATION | wx.STAY_ON_TOP | wx.OK | wx.CANCEL)

    result = exitBox.ShowModal()
    exitBox.Destroy()
```

Briefly below it's shown the mainly methods used for the design of this frame:

- ✓ *wx.ListCtrl()*
- ✓ *wx.Centre()*
- ✓ *wx.messageDialogs()*
- ✓ *wx.showModal()*
- ✓ *wx.Destroy()*
- ✓ *wx.SetColumnWidth()*
- ✓ *wx.BoxSizer()*
- ✓ *wx.SetToolTip()*
- ✓ *wx.checkBox()*
- ✓ *wx.Show()*
- ✓ *wx.Frame()*

The overview of this third frame contained in the *ThirdWindow.py* module is detailed below:

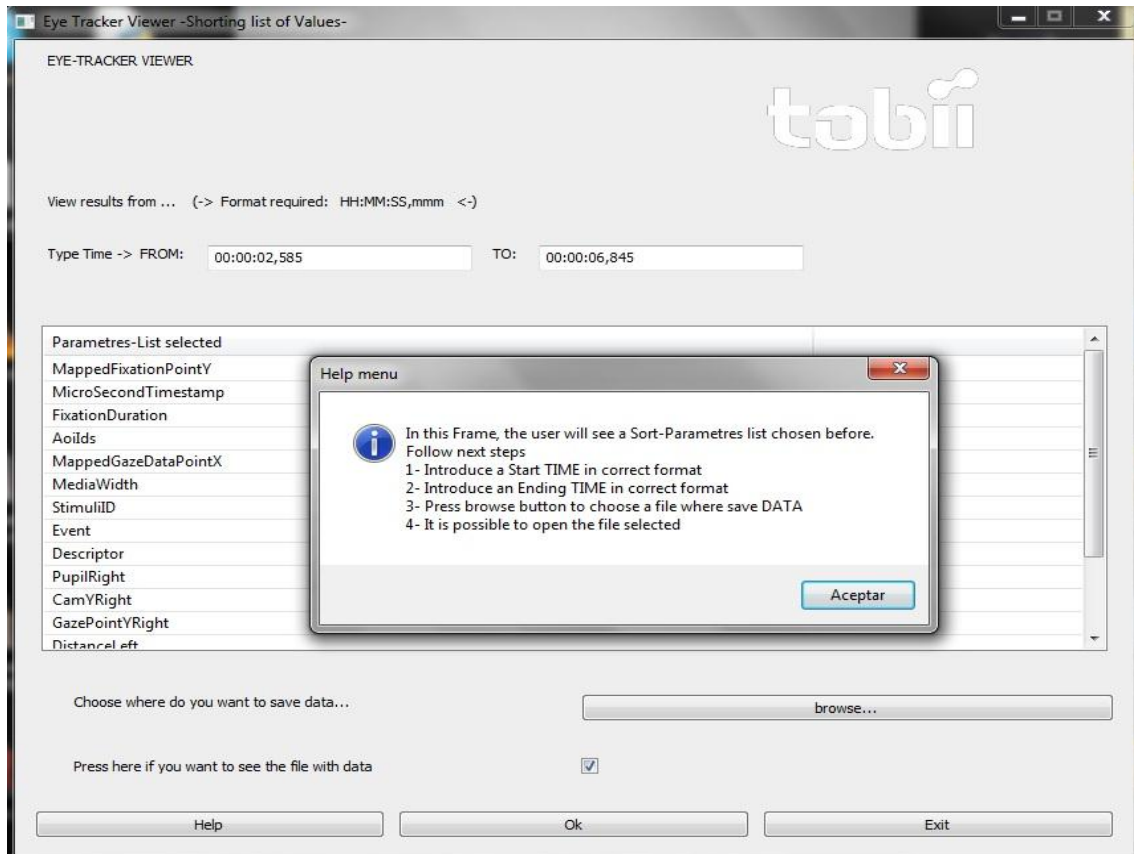


Figure 17. Screenshot of the third window of the application

3.3 UNDERSTANDING THE APP OBJECT LIFECYCLE AS PART OF THE DESIGN

The lifecycle of the *wxPython* application object begins when the application instance is created and ends just after the last application window is closed. This does not necessarily correspond to the beginning and ending of the *Python* script that surrounds our *wxPython* application. The script may choose to do some activity before creating the *wxPython* application, and may do further cleanup after the application *MainLoop()* exits. All *wxPython* activity, however, must be performed during the life of the application object. As it is mentioned, this means that our main frame object cannot be created until after the *wx.App* object is created. (This is one reason why it is recommend creating the top-level frame in the *OnInit()* method -doing so guarantees that the application already exists-.)

As below is shown, creating the application object triggers the *OnInit()* method and allows new window objects to be created. After *OnInit()*, the script calls *MainLoop()*, signifying that *wxPython* events are now being handled. The application continues on its merry way, handling events until the windows are closed. After all top-level windows are closed, the *MainLoop()* function returns to the calling scope and the application object is destroyed. After that, the script can close any other connections or threads that might exist.

A major event in the *wxPython* application lifecycle, including the beginning and ending of both the *wxPython* application and the script which surrounds it is shown below.

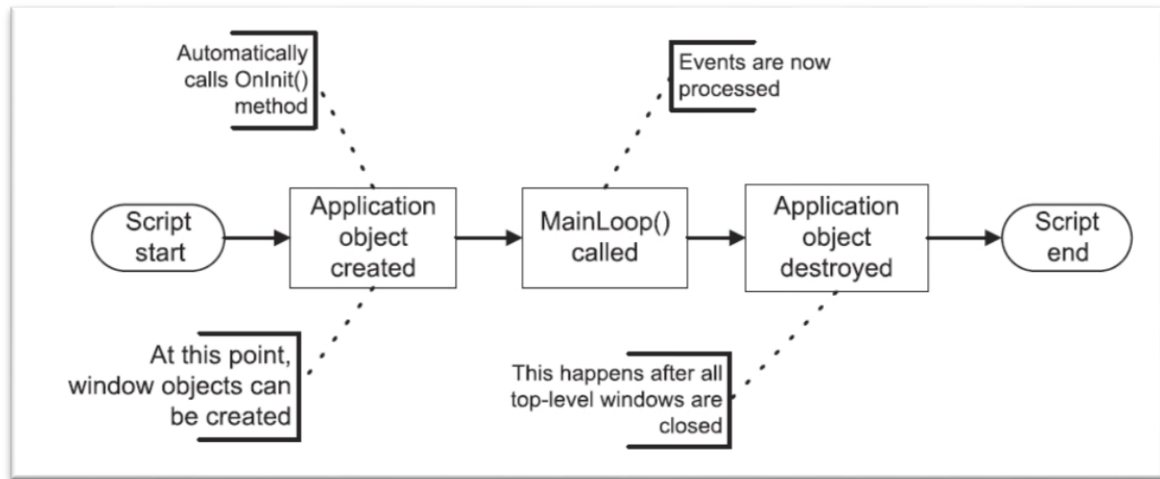


Figure 18. The life-cycle of the application

One reason to be aware of the main application *life-cycle* is that, while active, a *wxPython* application will take control of certain system functions, such as the standard output streams.

3.4 HOW THE APP WORKS

This application was developed with the purpose of making a simple graphical interface as well as functional. The main idea was to create an application that enables the users to search data in a fast and selective way, so as to using files much more manageable and small.

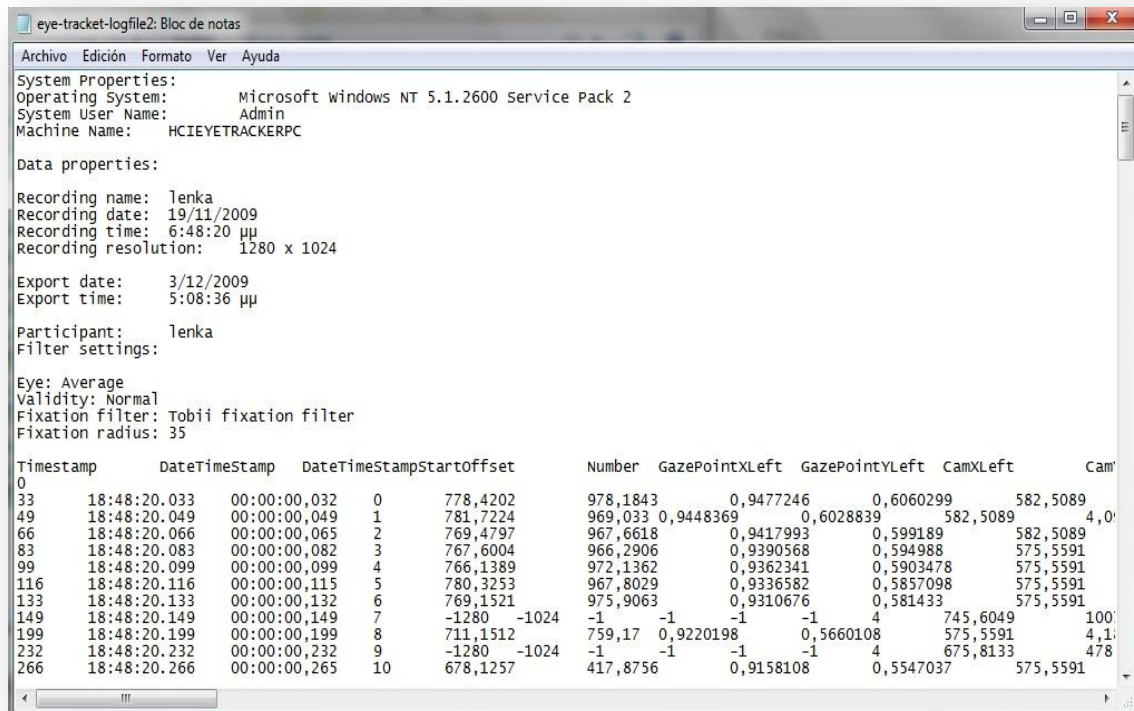
Here it is a brief resume of how the *log-file management tool* based on *wxpython* programming language works:

- At first the user is presented with a window that operates as a *browser-file*. This is the first step in which we will select the file in *.tsv* result of *eye-tracking* monitor *Tobii*. In this first frame, there are *help button* (always attempted to provide a helpful guide at all times to facilitate understanding of the program), as well as an *exit button* and one *next button* to continue reading the *metrics*.
- Then, and depending on a small time delay (variable according to the size of the *.tsv file* chosen in the *initial window*), we can see the *second window*. This frame works as a first filter of selected metrics. The main idea was to resume in a clear and fast way by using *lists*, and also buttons for *adding* or even *removing* parameters, so that in this way could obtain as much data to read as the user wanted. Likewise there is a button in the middle of the window, used to form alphabetical order of all the metrics found in the file *Tobii eye-tracking* selected.

DESIGN OF A LOG-FILE MANAGMENT TOOL FOR EYE-TRACKING DATA

It has been supposed that it was also a good way to establish a certain order, and thus facilitate the interaction between *user and interface*. In this way it is, and just by pressing the next button we will continue through next frame, or if the user deems it opportune, you can exit the application by selecting the *exit button* and suddenly the *ok button* in the *alert box* will appear.

- Finally, we reach the third and last window. This frame provides the user a last filter following *two text boxes*, which serve as time entry, and even more so summary data collected in the *metrics*. It is here where it has tried to be more careful in the sense that the entry of data such as time, is sensitive to any error in both the format introduced, as in variables such as longer than the permitted in *ending time* less than *starting time*... all error messages alerted to call the attention of the user. It enables the user a *list* of previously selected and gathered parameters, also appears on *browse-file button* (in case of failure to select any *file* the user is notified), a *checkbox widget* that just in case is ticked the *App* will open the selected file with *Windows Notepad* (it will depends on the operating system) as well as a *help button* for help and guidance.



System Properties:
Operating System: Microsoft Windows NT 5.1.2600 Service Pack 2
System User Name: Admin
Machine Name: HCIEYETRACKERPC

Data properties:
Recording name: lenka
Recording date: 19/11/2009
Recording time: 6:48:20 µµ
Recording resolution: 1280 x 1024

Export date: 3/12/2009
Export time: 5:08:36 µµ

Participant: lenka
Filter settings:

Eye: Average
Validity: Normal
Fixation filter: Tobii fixation filter
Fixation radius: 35

Timestamp	DateTimeStamp	DateTimeStampStartOffset	Number	GazePointXLeft	GazePointYLeft	CamXLeft	CamYLeft
0							
33	18:48:20.033	00:00:00,032	0	778,4202	978,1843	0,9477246	0,6060299
49	18:48:20.049	00:00:00,049	1	781,7224	969,033	0,9448369	0,6028839
66	18:48:20.066	00:00:00,065	2	769,4797	967,6618	0,9417993	0,599189
83	18:48:20.083	00:00:00,082	3	767,6004	966,2906	0,9390568	0,594988
99	18:48:20.099	00:00:00,099	4	766,1389	972,1362	0,9362341	0,5903478
116	18:48:20.116	00:00:00,115	5	780,3253	967,8029	0,9336582	0,5857098
133	18:48:20.133	00:00:00,132	6	769,1521	975,9063	0,9310676	0,581433
149	18:48:20.149	00:00:00,149	7	-1280	-1024	-1	-1
199	18:48:20.199	00:00:00,199	8	711,1512	759,17	0,9220198	0,5660108
232	18:48:20.232	00:00:00,232	9	-1280	-1024	-1	-1
266	18:48:20.266	00:00:00,265	10	678,1257	417,8756	0,9158108	0,5547037

Figure 19 Example of a *tsv* format file

CHAPTER 4

DEVELOPMENT AND IMPLEMENTATION

4.1 INTRODUCTION

Mainly, this project has been focused basically in the development and implementation of the application. Most of the time spent in the project has been used for this task. Due to this, it has been decided to present a detailed report of the features that have been designed and implemented in the application, some of them has been already mentioned in the previous chapter, but nevertheless, is a must need to go back to discuss it in a greater depth.

4.2 FEATURES IMPLEMENTED

Here below there are described the features used in the development and implementation of this *log-file management tool for eye-tracking data*.

4.2.1 PYTHON MODULES AND LIBRARIES DESCRIPTION

Python usually stores its library (and thereby mine site-packages folder) in the installation directory. So, as I had installed Python to *C:\Python*, the default library is resided in *C:\Python\Lib*, and third-party modules are stored in *C:\Python\Lib\site-packages*. Here is a depth description of the different modules or libraries used in the *log-file App*.

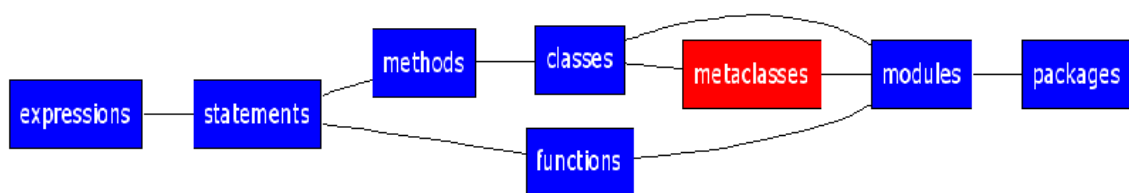


Figure 20. Modules and libraries used in the application

4.2.1.1 WX Python library

WxPython is a *GUI* toolkit for the *Python* programming language. It allows *Python* programmers to create programs with a robust, highly functional graphical user interface, simply and easily. It is implemented as a *Python extension module* (native code) that wraps the popular *wxWidgets* cross platform *GUI library*, which is written in *C++*.

Thank to this *module*, there are a lot of features implemented. More or less, we must thank this module all about the design of the application. That is, all the methods used

and related to the implementation of the *graphical log-file*, inherited from the *wx python library* as part of the *widgets* that have been used.

As a matter of fact, there have been changes throughout the *importing of modules* since the first Python code was released. Here is a brief review, taken from the book *wxpython in Action*[3]

✓ **OLD STYLE IMPORTS**

The first thing we need to do is import the main wxPython package, which is named wx:

```
➤ import wx
```

Once that package is imported, you can refer to wxPython classes, functions, and constants using the wx package name as a prefix, like this:

```
➤ class App(wx.App):
```

Since the old naming convention is still supported, I have encountered wxPython code written in the “old style”. So, I am going to digress briefly explaining the “older style” and why it was changed. The old package name was wxPython and it contained an internal module named wx. There were two common ways to import the needed code -I imported the wx module from the wxPython package:

```
➤ from wxPython import wx #ALMOST DEPRECATED
```

Or, I could import everything from the wx module directly.

```
➤ from wxPython.wx import * # ALMOST DEPRECATED
```

*Both import methods had serious drawbacks. Using the second method of import * is generally discouraged in Python because of the possibility of namespace conflicts. The old wx module avoided this problem by putting a wx prefix on nearly all of its attributes. Even with this safeguard, import * still had the potential to cause problems, but many wxPython programmers preferred this style, and it is normal to see it used quite often in older code. One downside of this style was that class names began with a lowercase letter, while most of the wxPython methods begin with an uppercase letter -the exact opposite of the normal Python programming convention. However, I have tried to avoid the namespace bloat caused by import * by doing from wxPython import wx, and this is the reason why now had to type “wx” twice for each class, function, or constant name—once as the package prefix and once as the “normal” prefix, such as wx.wxWindow. This got old fast. Many wxPython programmers saw this dilemma as a wart that should be removed, and eventually, it was.*

One more thing to know about importing *wxPython*: I must import *wx* before I import anything else from *wxPython*. In general, the order of imports in *Python* is irrelevant, meaning you can import *modules* in any order. However, *wxPython*, although it looks like a single *module*, is actually a complex set of modules (many of which are automatically generated by a tool called the *Simplified Wrapper and Interface Generator*, or *SWIG*) that wrap the functionality provided by the underlying *wxWidgets* C++ toolkit.

4.2.1.1.1 SWIG

To build Python extension modules, *SWIG* uses a layered approach in which parts of the extension *module* are defined in C and other parts are defined in *Python*. The C layer contains low-level wrappers whereas *Python* code developed in this *Application* is used to define high-level features.

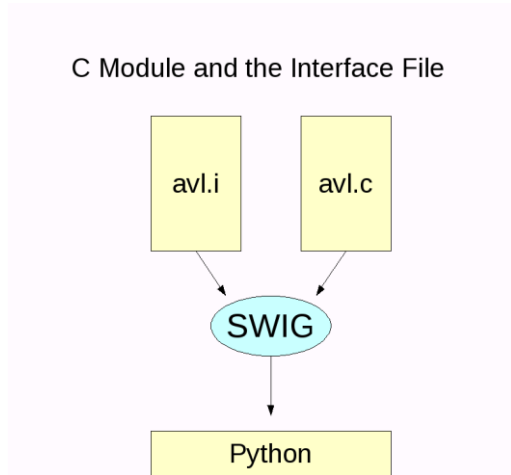


Figure 21 SWIG wrapper for Python

This layered approach recognizes the fact that certain aspects of extension building are better accomplished in each language (instead of trying to do everything in C or C++). Furthermore, by generating code in both languages, you get a lot more flexibility since you can enhance the extension module with support code in either language.

There are researches, that explained why is helpful this kind of wrapper, and not for example *boost wrapper*. For the development of this *App* that have no long files, is shown below a graphic that shows the differences in terms of time, and measures how quickly *SWIG* goes versus *BOOST* one [4]:

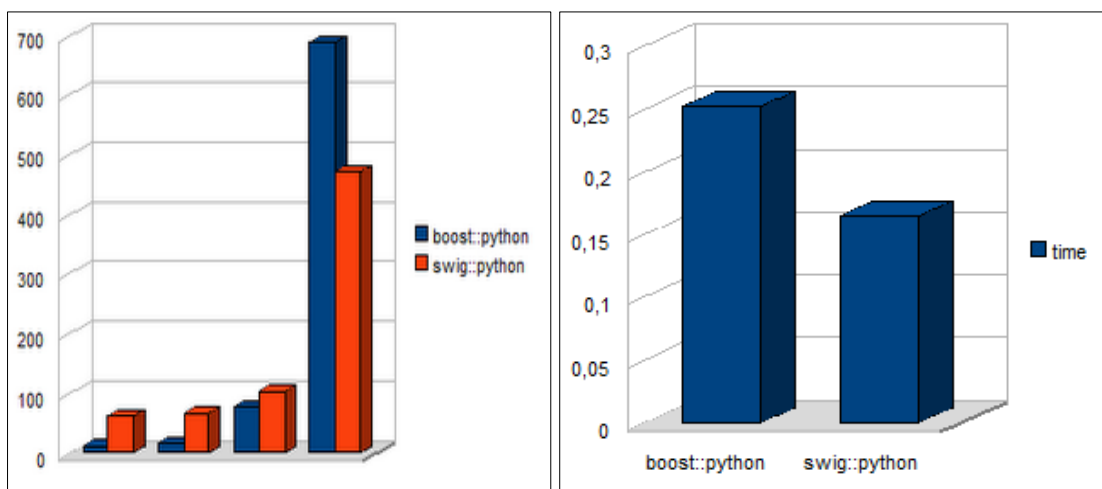


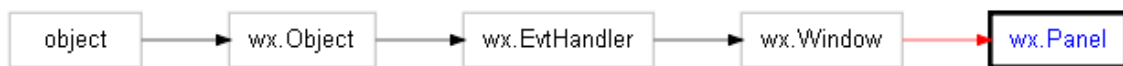
Figure 22Size Matters in time Boost vs. SWIG Figure 23 Runtime test. Boost vs. SWIG

4.2.1.1.2 Widgets provide in WX Module

Nearly all of the *Graphical User Interface (GUI)* made in the development of this *log-file management tool for eye-tracking data* can be filled by *wx module* for *Python*. This is the main reason for the need of writing in an extended way which are the features implemented with this *module* throughout this entire *App*. In the following paragraphs I will talk about some *wxPython* toolkits look like, such as *buttons*, *checkboxes*, *list boxes*, *text controls*, etc. Some of the information written below has been collected from the *wxPython GUI toolkit* webpage [5].

1. **Wx.Panel() method**

Inheritance diagram for *wx.Panel*:



This method has been used for the development and design of a *Panel* contained in the *initial frame*, gathered in *menuv3.py* package.

A panel is a window on which controls are placed. It is usually placed within a frame. It contains minimal extra functionality over and above its parent class *wx.Window*; its main purpose is to be similar in appearance and functionality to a dialog, but with the flexibility of having any window as a parent. In it there are placed a lot of widgets as buttons, text control, a GridBagSizer, static text, etc.

➤ **Parameters:**

- ✓ *parent (wx.Window)*
- ✓ *id (int)*
- ✓ *pos (wx.Point)*
- ✓ *size (wx.Size)*
- ✓ *style (long)*
- ✓ *name (string)*

➤ **Returns:**

- ✓ *wx.Panel*

And here it is the code implemented in *menuv3.py* package:

```

panel = wx.Panel(self, -1, size=(500, 300),
style=wx.DEFAULT|wx.NO_FULL_REPAINT_ON_RESIZE)
  
```

Below is shown the general appearance of the main *panel* contained in the first frame of the “Eye-Tracking Data Extractor” Application:

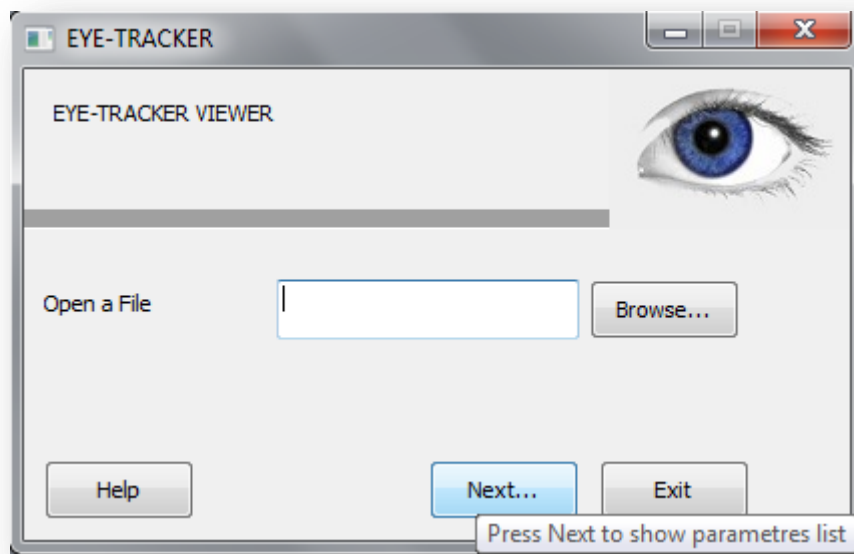


Figure 24 Panel method in wx module

2. *Wx.GridBagSizer()* method

Inheritance diagram for *wx.GridBagSizer*:



This method has been used for the development and design of a virtual grid where can be laid out items in the *initial frame* gathered in *menuv3.py* package.

A *wx.Sizer* that can lay out items in a virtual grid like a *wx.FlexGridSizer* but in this case explicit positioning of the items is allowed using *wx.GBPosition*, and items can optionally span more than one row and/or column using *wx.GBSpan*.

➤ **Methods:**

✓ `__init__(vgap=0, hgap=0)`

Constructor, with optional parameters to specify the gap between the rows and columns.

➤ **Parameters:**

✓ *vgap* (*int*)

✓ *hgap (int)*

➤ Returns:

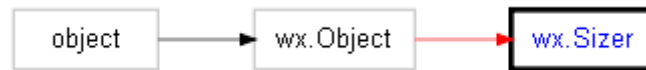
✓ *wx.GridBagSizer*

And here it is the code implemented in *menuv3.py* package:

```
sizer = wx.GridBagSizer(0, 0)
```

3. Wx.Sizer() method:

Inheritance diagram for *wx.Sizer*:



This method has been used in all the frames developed in the three main packages of this “*Eye Tracking Data Extractor*” such as *menuv3.py*, *SecondWindow.py* and *ThirdWindow.py*.

Wx.Sizer is the abstract base class used for laying out sub-windows in a window. It is not allowed to use *wx.Sizer* directly; instead, I have to use one of the *sizer* classes derived from it. As explained in the previous method above, the currently classes are: *wx.BoxSizer*, *wx.StaticBoxSizer*, *wx.GridSizer*, *wx.FlexGridSizer* and *wx.GridBagSizer*.

The layout algorithm used by *sizers* in *wxWidgets* is closely related to layout in other *GUI toolkits*, such as *Java’s AWT*, the *GTK toolkit* or the *Qt toolkit*. It is based upon the idea of the individual sub-windows reporting their minimal required size and their ability to get stretched if the size of the parent window has changed. This will most often mean that while I programmed the design of the frames, I did not set the original size of a dialog in the beginning, rather the dialog will be assigned a *sizer* and this *sizer* will be queried about the recommended size. The *sizer* in turn will query its children, which can be normal windows, empty space or other *sizers*, so that a hierarchy of *sizers* can be constructed.

What makes *sizers* so well fitted for use in *wxWidgets* is the fact that every control reports its own minimal size and the algorithm can handle differences in font sizes or different window (dialog item) sizes on different platforms without problems, as part of the tested *App*. If e.g. the standard font as well as the overall design of *Motif widgets* requires more space than on *Windows*, the initial dialog size will automatically be bigger on *Motif* than on *Windows*.

Sizers may also be used to control the layout of custom drawn items on the window. The *Add*, *Insert*, and *Prepend* functions return a pointer to the newly added *wx.SizerItem* method. Just add empty space of the desired size and attributes, and then use the *wx.SizerItem*.

➤ **Methods:**

✓ `__init__()`

The constructor.

Note that `wx.Sizer` is an abstract base class and may not be instantiated.

✓ `Add(item, proportion=0, flag=0, border=0, userData=None)`

Appends a child to the *sizer* `wx.Sizer`. It is an abstract class, but the parameters are equivalent in the derived classes that I have instantiated to use it so they are described here:

- ***window*:** The window to be added to the *sizer*. Its initial size (either set explicitly by the user or calculated internally when using `wx.DefaultSize`) is interpreted as the minimal and in many cases also the initial size.
- ***sizer*:** The (child-)sizer to be added to the *sizer*. This allows placing a child sizer in a sizer and thus to create hierarchies of sizers.
- ***width and height*:** The dimension of a spacer to be added to the *sizer*. Adding spacers to sizers gives more flexibility in the design of dialogs; for example in the development of the *third frame*, there are *horizontal boxes* with two *buttons* at the bottom of a dialog: so the main purpose is to insert a space between the *buttons* and make that space stretchable using the *proportion flag* and the result will be that the *left button* will be aligned with the left side of the dialog and the *right button* with the right side - the space in between will shrink and grow with the dialog.
- ***proportion*:** Although the meaning of this parameter is undefined in `wx.Sizer`, it is used in `wx.BoxSizer` to indicate if a child of a *sizer* can change its size in the main orientation of the `wx.BoxSizer` – where *0* stands for not changeable and a value of more than zero is interpreted relative to the value of other children of the same `wx.BoxSizer`. For example, in the development of the *Eye-Tracking data Extractor*, there is in the *SecondWindow.py module* a horizontal `wx.BoxSizer` with three children, two of which are supposed to change their size with the sizer. Then the two stretchable windows get a *value of 1* each to make them grow and shrink equally with the *sizer's horizontal* dimension.
- ***flag*:** This parameter can be used to set a number of *flags* which can be combined using the binary *OR operator* `|`. Two main behaviors are defined using these flags. One is the *border around a window*: the *border* parameter determines the *border width* whereas the flags given here determine *which side(s)* of the item that the border will be added. The other flags determine how the sizer item behaves when the space allotted to the sizer changes, and is somewhat dependent on the specific kind of sizer used:

- ***border***: Determines the *border width*, if the *flag parameter* is set to include any border flag.
- ***userData***: Allows an *extra object* to be attached to the sizer item, for use in derived classes when sizing information is more complex than the proportion and flag will allow for.
- ***flags***: A *wx.SizerFlags* object that enables you to specify most of the above parameters more conveniently.

<ul style="list-style-type: none"> • <i>wx.TOP</i> • <i>wx.BOTTOM</i> • <i>wx.LEFT</i> • <i>wx.RIGHT</i> • <i>wx.ALL</i> 	These flags are used to specify which side(s) of the sizer item that the <i>border</i> width will apply to.
<ul style="list-style-type: none"> • <i>wx.EXPAND</i> 	The item will be <i>expanded</i> to fill the space allotted to the item.
<ul style="list-style-type: none"> • <i>wx.SHAPED</i> 	The item will be <i>expanded</i> as much as possible while also maintaining its aspect ratio
<ul style="list-style-type: none"> • <i>wx.FIXED_MINSIZE</i> 	Normally <i>wx.Sizers</i> use <i>wx.Window.GetMinSize</i> or <i>wx.Window.GetBestSize</i> to determine what the minimal size of window items should be, and will use that size to calculate the layout. This allows layouts to adjust when an item changes and its best size becomes different. If it is already a window item stay the size it started with then use <i>wx.FIXED_MINSIZE</i> .
<ul style="list-style-type: none"> • <i>wx.ALIGN_CENTER</i> • <i>wx.ALIGN_LEFT</i> • <i>wx.ALIGN_RIGHT</i> • <i>wx.ALIGN_TOP</i> • <i>wx.ALIGN_BOTTOM</i> • <i>wx.ALIGN_CENTER_VERTICAL</i> • <i>wx.ALIGN_CENTER_HORIZONTAL</i> 	The <i>wx.ALIGN</i> flags allow you to specify the alignment of the item within the space allotted to it by the <i>sizer</i> , adjusted for the <i>border</i> if any.

Table 3wx.SizerFlags in WX module

➤ Parameters:

- ✓ *item*: The item to add to the sizer.
- ✓ *proportion* (int)
- ✓ *flag* (int)
- ✓ *border* (int)
- ✓ *userData* (PyObject)

➤ Returns:

✓ *wx.SizerItem*

Here it is the code implemented in the *SecondWindow.py* module, as a result of various *sizers* used, which allow to display in the frame two *list boxes*, different *buttons*, the *title*, the *logo display* on the upper right side, everything gathered into a *BoxSizer*, that we already talked about in the previous chapter, but also I will explain it in a greater depth, as an important method of the *wx python* module.

```
# use a vertical boxsizer as main layout sizer
sizer_v = wx.BoxSizer(wx.VERTICAL)

# use a vertical boxsizer to put into main layout sizer
sizer_v2Title = wx.BoxSizer(wx.HORIZONTAL)
sizer_v3Cajas = wx.BoxSizer(wx.VERTICAL)
sizer_v4ListParametres = wx.BoxSizer(wx.HORIZONTAL)
sizer_v5NextCancel = wx.BoxSizer(wx.HORIZONTAL)

# use a vertical sizer for the buttons
sizer_v2Title.Add(text1, 1, flag=wx.EXPAND|wx.LEFT, border=15)
sizer_v2Title.Add(icon, 1, flag=wx.EXPAND|wx.LEFT, border=150)

sizer_v3Cajas.Add(sort_parametres, 1, flag=wx.ALL|wx.EXPAND, border=5)
sizer_v3Cajas.Add(Add_Item, 1, flag=wx.ALL|wx.EXPAND, border=5)
sizer_v3Cajas.Add(Delete_Item, 1, flag=wx.ALL|wx.EXPAND, border=5)
sizer_v4ListParametres.Add(self.lc,1,flag=wx.ALL|wx.EXPAND, border=10)
sizer_v4ListParametres.Add(sizer_v3Cajas,0,flag=wx.ALL|wx.EXPAND,
                           border=10)

sizer_v4ListParametres.Add(self.lc2,1,flag=wx.ALL|wx.EXPAND,border=10)
sizer_v5NextCancel.Add(Exit_button,1,flag=wx.ALL|wx.EXPAND, border=5)
sizer_v5NextCancel.Add(Next_button,1,flag=wx.ALL|wx.EXPAND, border=5)
```

```
# add the rest + sizer_h to the vertical sizer
sizer_v.Add(sizer_v2Title,0,flag=wx.ALL|wx.EXPAND|wx.RIGHT,
border=10) sizer_v.Add(sizer_v4ListParametres,2,flag=wx.ALL|wx.EXPAND|w
x.RIGHT,
                           border=10)
sizer_v.Add(sizer_v5NextCancel,0,flag=wx.ALL|wx.EXPAND,border=10)

self.SetSizer(sizer_v)
self.Centre()
```

4. Wx.Button() method:

Inheritance diagram for *wx.Button*:



Along the code implemented in the “*Eye-Tracking Data Extractor Application*” there are several buttons, instanced with the `wx.button()` method. A button is a control that contains a text string, and is one of the most common elements of the *Graphic User Interface (GUI)*. It may be placed on a dialog or panel (as it has been done in most of the *modules*), or indeed almost any other window.

Here are some examples as how a button can be displayed in different platforms:



Figure 25 Button in different platforms

➤ **Methods:**

- ✓ `__init__(parent, id=-1, label="", pos=wx.DefaultPosition, size=wx.DefaultSize, style=0, validator=wx.DefaultValidator, name=wx.ButtonNameStr)`

Create and show a button. The preferred way to create standard buttons is to use a standard *ID* and an empty label. In this case *wxWidgets* will automatically use a stock label that corresponds to the *ID* given. These labels may vary across platforms as the platform itself will provide the label if possible. In the case we are dealing with, the platform where the code has been implemented is *Windows 7 Home Premium*. In the image shown above this paragraph matches with the first button displayed.

➤ **Parameters:**

- ✓ *parent* (*wx.Window*)
- ✓ *id* (*int*)
- ✓ *label* (*string*)
- ✓ *pos* (*wx.Point*)
- ✓ *size* (*wx.Size*)
- ✓ *style* (*long*)
- ✓ *validator* (*wx.Validator*)
- ✓ *name* (*string*)

➤ **Returns:**

- ✓ *wx.Button*

It is worth mentioning in this section on a widely used tool in relation to the method described above. I will write a brief resume, as just an introduction in order to provide an overall idea.

Wx.SetToolTip():

The main purpose of this method is to attach a tooltip to the window.

➤ **Parameters:**

✓ *tip (wx.ToolTip)*

Above we can see how this useful tool is displayed in the Application, particularly, is implemented in the third frame of the “*Eye-Tracking Data Extractor*”:

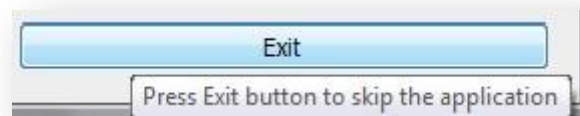


Figure 26 Button with *wx.ToolTip*method

As a result, here it is the code for this method implemented in the *ThirdWindow.py* package:

```
Exit_button = wx.Button(self, wx.ID_ANY, "Exit")
Exit_button.SetToolTip(wx.ToolTip("Press Exit button to skip the
application"))
```

5. **Wx.ListCtrl() method:**

Inheritance diagram for *wx.ListCtrl()*:



A *list control* presents lists in a number of formats: *list view*, *report view*, *icon view* and *small icon view*. In any case, elements are numbered from zero. For all these modes, the items are stored in the control and must be added to it using *InsertItem()* method.

A special case of report view quite different from the other modes of the list control is a virtual control in which the items data (including text, images and attributes) is managed by the main program and is requested by the control itself only when needed which allows to have controls with millions of items without consuming much memory.

To intercept events from a list control, I have used the event table macros described in *wx.ListEvent()*, which we will discuss with greater depth hereafter.

➤ **Window styles:**

Window Style	Description
<code>wx.LC_LIST</code>	Multicolumn list view, with optional small icons. Columns are computed automatically, i.e. you don't set columns as in <code>wx.LC_REPORT</code> . In other words, the list wraps, unlike a <code>wx.ListBox</code> .
<code>wx.LC_REPORT</code>	Single or multicolumn report view, with optional header.
<code>wx.LC_VIRTUAL</code>	The application provides items text on demand. May only be used with <code>wx.LC_REPORT</code> .
<code>wx.LC_ICON</code>	Large icon view, with optional labels.
<code>wx.LC_SMALL_ICON</code>	Small icon view, with optional labels.
<code>wx.LC_ALIGN_TOP</code>	Icons align to the top. Win32 default, Win32 only.
<code>wx.LC_ALIGN_LEFT</code>	Icons align to the left.
<code>wx.LC_AUTOARRANGE</code>	Icons arrange themselves. Win32 only.
<code>wx.LC_EDIT_LABELS</code>	Labels are editable: the application will be notified when editing starts.
<code>wx.LC_NO_HEADER</code>	No header in report mode.
<code>wx.LC_SINGLE_SEL</code>	Single selection (default is multiple).
<code>wx.LC_SORT_ASCENDING</code>	Sort in ascending order (must still supply a comparison callback in <code>SortItems</code>).
<code>wx.LC_SORT_DESCENDING</code>	Sort in descending order (must still supply a comparison callback in <code>SortItems</code>).
<code>wx.LC_HRULES</code>	Draws light horizontal rules between rows in report mode.
<code>wx.LC_VRULES</code>	Draws light vertical rules between columns in report mode.

Table 4 *wx.ListCtrl* window styles

Here above there is a list with the most commons events bind to the `wx.ListCtrl()` method. Further on, we will discuss about *Events*, and we will talk about the events of this method, as it is considered one of the main methods used in the development of the “Eye-Tracking Data Extractor” Application.

➤ **Event handling:**

EventName	Description
<code>wx.EVT_LIST_BEGIN_DRAG(id, func)</code>	Begin dragging with the left mouse button.
<code>wx.EVT_LIST_BEGIN_RDRAG(id, func)</code>	Begin dragging with the right mouse button.
<code>wx.EVT_LIST_BEGIN_LABEL_EDIT(id, func)</code>	Begin editing a label. This can be prevented by calling <i>Veto</i> .
<code>wx.EVT_LIST_END_LABEL_EDIT(id, func)</code>	Finish editing a label. This can be prevented by calling <i>Veto</i> .
<code>wx.EVT_LIST_DELETE_ITEM(id, func)</code>	Delete an item.
<code>wx.EVT_LIST_DELETE_ALL_ITEMS(id, func)</code>	Delete all items.
<code>wx.EVT_LIST_ITEM_SELECTED(id, func)</code>	The item has been selected.
<code>wx.EVT_LIST_ITEM_DESELECTED(id, func)</code>	The item has been deselected.
<code>wx.EVT_LIST_ITEM_ACTIVATED(id, func)</code>	The item has been activated (ENTER or double click).
<code>wx.EVT_LIST_ITEM_FOCUSED(id, func)</code>	The currently focused item has changed.

Table 5 *wx.ListCtrl* events handling

➤ **Methods:**

- ✓ `__init__(parent, id=-1, pos=wx.DefaultPosition, size=wx.DefaultSize, style=wx.LC_ICON, validator=wx.DefaultValidator, name=wx.ListCtrlNameStr)`

➤ **Parameters:**

- ✓ *parent* (*wx.Window*)
- ✓ *id* (*int*)
- ✓ *pos* (*wx.Point*)
- ✓ *size* (*wx.Size*)
- ✓ *style* (*long*)
- ✓ *validator* (*wx.Validator*)
- ✓ *name* (*string*)

➤ **Returns:**

- ✓ *wx.ListCtrl*

Here are other *methods* from the *Class API*. I thought it would be appropriate that should be named as an important part in the realization of the application, and which are described in the implementation of the code. Then we will see an example used in the *SecondWindow.py* package of the Application.

✓ **Append(entry)**

Append an item to the list control. The entry parameter should be a sequence with an item for each column.

- *Parameters:*
 - *entry* (*tuple*)

✓ **DelleteAllColumns()**

✓ **DeleteAllItems()**

Deletes all items in the list control.

- *Returns:*
 - *Bool*

✓ **DeleteColumn(col)**

Deletes a column.

- *Parameters:*
 - *col* (*int*)

- *Returns:*

- Bool

✓ **DeleteItem(item)**

Deletes the specified item. This function sends the wxEVT_COMMAND_LIST_DELETE_ITEM event for the item being deleted.

- *Parameters:*

- item (long)

- *Returns:*

- Bool

```
self.lc = wx.ListCtrl(self, wx.ID_ANY,  
    style=wx.LC_REPORT|wx.SUNKEN_BORDER|wx.LC_HRULES)  
  
self.lc.InsertColumn(0, "Parametres-List matched in the .tsv  
file")  
self.lc.SetColumnWidth(0, 580)  
  
def loadList(self):  
    # clear the listctrl  
    self.lc.DeleteAllItems()
```

```

        # load each data row
        for ix, line in enumerate(self.states):
            # set max_rows, change if need be
            # also sets/updates row index starting at 0
            index = self.lc.InsertStringItem(max_rows, line)

max_rows = 100

defonAdd_Item(self, event):
    ix_selected = self.lc.GetNextItem(item=-1,
        geometry=wx.LIST_NEXT_ALL,
        state=wx.LIST_STATE_SELECTED)
    state = names0[ix_selected]

    s2.insert(0, state)
    index=self.lc2.InsertStringItem(0, s2[0])
self.lc2.SetStringItem(index, 0, s2[0])

defonDelete_Item(self, event):
    ix_selected3 = self.lc2.GetFocusedItem()
self.lc2.DeleteItem(self.lc2.GetFocusedItem())

defonSelect(self, event):
    # wx.LIST_STATE_SELECTED get the selected item
    ix_selected=self.lc.GetNextItem(item=-1,
        geometry=wx.LIST_NEXT_ALL, state=wx.LIST_STATE_SELECTED)

    state = names0[ix_selected]
    s1=[]
    s1.append(state)

```

As a result of the code shown above, there is the frame related with the second frame of the *SecondWindow.py* package that displays the *wx.ListCtrl()* method implemented:

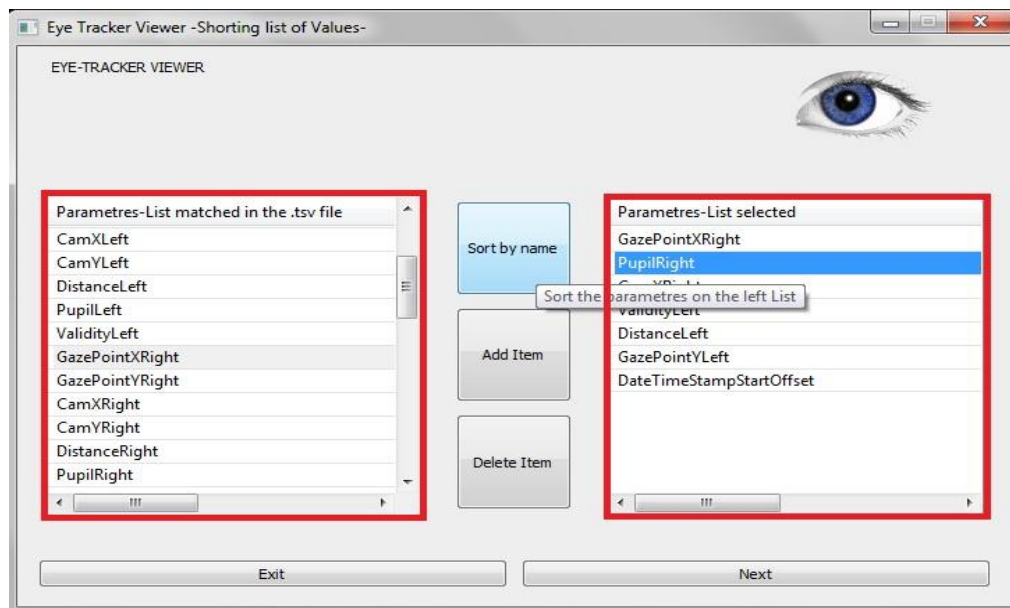


Figure 27 *wx.ListCtrl()* method with two different lists

6. Wx.CheckBox() method:

Inheritance diagram for *wx.CheckBox*:



This special method has been instanced in the *ThirdWindow.py* package, as a result of the need of having a *check box*, that when is ticked, and previously the user has chosen a file where to save the data resulted of the *metrics* and *time* selected, it will be displayed the *.tsv file* with the data written on it.

A checkbox is a labeled box which by default is either on (checkmark is visible) or off (no checkmark).

➤ Window styles:

Window Style	Description
<i>wx.CHK_2STATE</i>	Create a 2-state checkbox. This is the default.
<i>wx.CHK_3STATE</i>	Create a 3-state checkbox. Not implemented in wxMGL, wxOS2 and wxGTK built against GTK+ 1.2.
<i>wx.CHK_ALLOW_3RD_STATE_FOR_USER</i>	By default a user can't set a 3-state checkbox to the third state. It can only be done from code. Using this flags allows the user to set the checkbox to the third state by clicking.
<i>wx.ALIGN_RIGHT</i>	Makes the text appear on the left of the checkbox.

Table 6wx.CheckBox window styles

The method is instanced with the *wx.CHK_2STATE*, as it is the default, and we don't need any more states.

➤ Control Appearance:

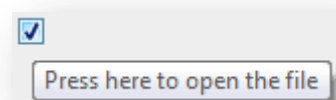


Figure 28wx.CheckBox with two different states

➤ Methods:

- ✓ `__init__(parent, id=-1, label="", pos=wx.DefaultPosition, size=wx.DefaultSize, style=0, validator=wx.DefaultValidator, name=wx.CheckBoxNameStr)`

Creates and shows a `wx.CheckBox` control as we can see in the picture above, and we can see implemented in the code below.

➤ Parameters:

- ✓ `parent` (`wx.Window`)
- ✓ `id` (`int`)
- ✓ `label` (`string`)
- ✓ `pos` (`wx.Point`)
- ✓ `size` (`wx.Size`)
- ✓ `style` (`long`)
- ✓ `validator` (`wx.Validator`)
- ✓ `name` (`string`)

➤ Returns:

- ✓ `wx.CheckBox`

Here is the example of the code implemented for creating this *widget*, as the result of implementing the `wx.checkbox()` method.

```
view_button=wx.CheckBox(self,wx.ID_ANY,style=wx.CHK_2STATE)
view_button.SetToolTip(wx.ToolTip("Press here to open the file"))
```

➤ Wx.StaticText() method:

Inheritance diagram for `wx.StaticText`:



Along the “*Eye-Tracking Data Extractor*” Application there have been instanced a lot of static text methods, in order to placed text in the different frames. Mostly, it has served to design a clear and visual way in which the windows are.

Basically, a static text control displays one or more lines of read-only text.

Significantly, in the great majority of instances, these methods have been positioned within other methods as in the case of the `wx.BoxSizer()` we have already discussed before. It has been possible with the `wx.Add()` method, just choosing the right place in the *Box Sizer*.

➤ **Control Appearance:**

This is how it looks the static text in the *ThirdWindow.py* package, where the lines of text related with this method are framed in red color. Depending on the platform, we will see different visual effect on the text, as in this example, it has been run under *Windows 7 Home Premium*:

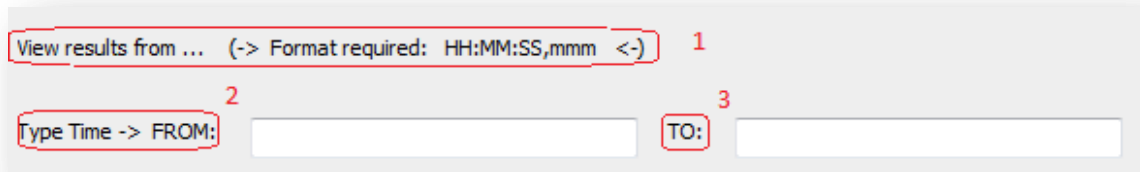


Figure 29 Three different instances of *wx.TextCtrl* method

➤ **Method from the Class API:**

- ✓ `__init__(parent, id=-1, label="", pos=wx.DefaultPosition, size=wx.DefaultSize, style=0, name=wx.StaticTextNameStr)`

➤ **Parameters:**

- ✓ *parent* (*wx.Window*)
- ✓ *id* (*int*)
- ✓ *label* (*string*)
- ✓ *pos* (*wx.Point*)
- ✓ *size* (*wx.Size*)
- ✓ *style* (*long*)
- ✓ *name* (*string*)

➤ **Returns:**

- ✓ *wx.StaticText*

And here below is detailed the code related with this method, taken from the *ThirdWindow.py* package, as it has been used throughout the four frames implemented in this *Application*. Note that there are placed into a *Box Sizer* detailed in next page.

```
# create titles for each boxsizer
text2 = wx.StaticText(self, -1, 'View results from ...(->
Format required: HH:MM:SS,mmm<-) ')

text5 = wx.StaticText(self, -1, 'Type Time -> FROM: ')
text6 = wx.StaticText(self, -1, ' TO: ')

text3 = wx.StaticText(self, -1, ' Choose where do you
want to save data... ')

text4 = wx.StaticText(self, -1, ' Press here if you want
to see the file with data ')

```

```
sizer_Title.Add(text1, 0, flag=wx.EXPAND|wx.LEFT, border=15)
sizer_Results2.Add(text2, 0, flag=wx.ALL|wx.EXPAND, border=5)

sizer_Results3.Add(text5, 0, flag=wx.ALL|wx.EXPAND,
border=5) sizer_Results3.Add(text6, 0, flag=wx.ALL|wx.EXPAND,
border=5) sizer_Save.Add(text3, 1, flag=wx.EXPAND|wx.ALL,
border=5)

sizer_ViewFile.Add(text4, 1, flag=wx.ALL|wx.EXPAND, border=5)

```

7. Wx.Dialog() method:

Inheritance diagram for *wx.Dialog*:



During the realization of this *project* has been used on numerous occasions this *method*. It is considered one of the most important because it has been possible to provide a variety of ways with a better understanding of what goes on in the program. This is mainly because there is a window with information in the form of *warning*, *advice* or even to offer the user a *second chance* (see the example given by this method in use to close the *application*, provided they do not close it because of user error, ensuring that you are ending it).

This class represents a *dialog* that shows a *single or multi-line message*, with a choice of *OK*, *Yes*, *No* and *Cancel buttons*.

➤ **Methods in Class API:**

- ✓ `__init__(parent, message, caption=wx.MessageBoxCaptionStr, style=wx.OK|wx.CANCEL|wx.CENTRE, pos=wx.DefaultPosition)`

The Constructor, use *ShowModal()* method from the *wx module* in *python* in order to display the dialog.

➤ **Parameters:**

- ✓ *parent (wx.Window)*
- ✓ *message (string)*
- ✓ *caption (string)*
- ✓ *style (long)*
- ✓ *pos (wx.Point)*

➤ **Returns:**

- ✓ *Wx.MessageDialog*

And other methods included in the *wx.MessageDialog()* are the *wx.ShowModal()* and also it has to be mentioned the *wx.Destroy()*.

The *ShowModal()* method allows the Application to mainly show the dialog, returning one of the *wx.ID_OK*, *wx.ID_CANCEL*, *wx.ID_YES* or *wx.ID_NO*. On the other hand the *Destroy()* method hide and kill the process routine of the Message Dialog is running in.

Below we can see some examples used in the “*Eye-Tracking Data Extractor*” Application along all the *packages implemented*:

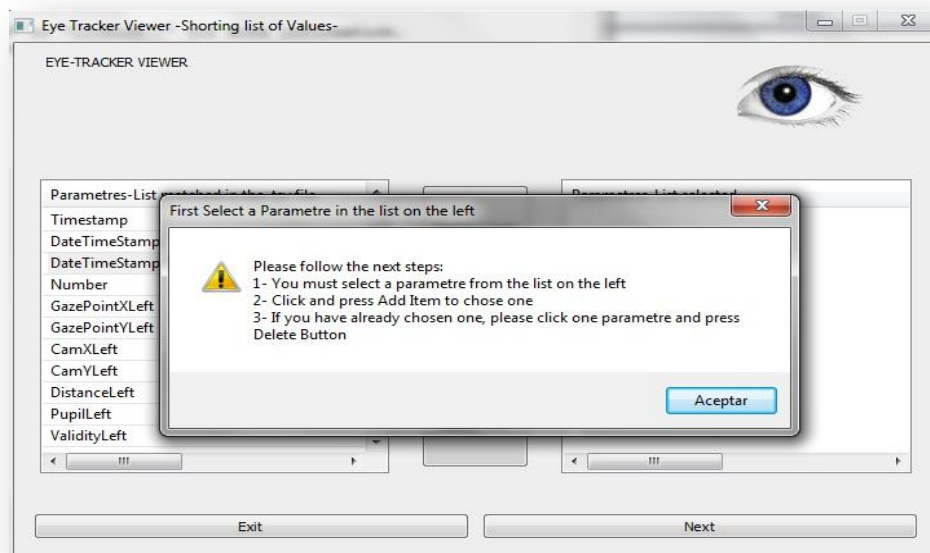


Figure 30 Alert message implemented with Message Dialog

DESIGN OF A LOG-FILE MANAGMENT TOOL FOR EYE-TRACKING DATA

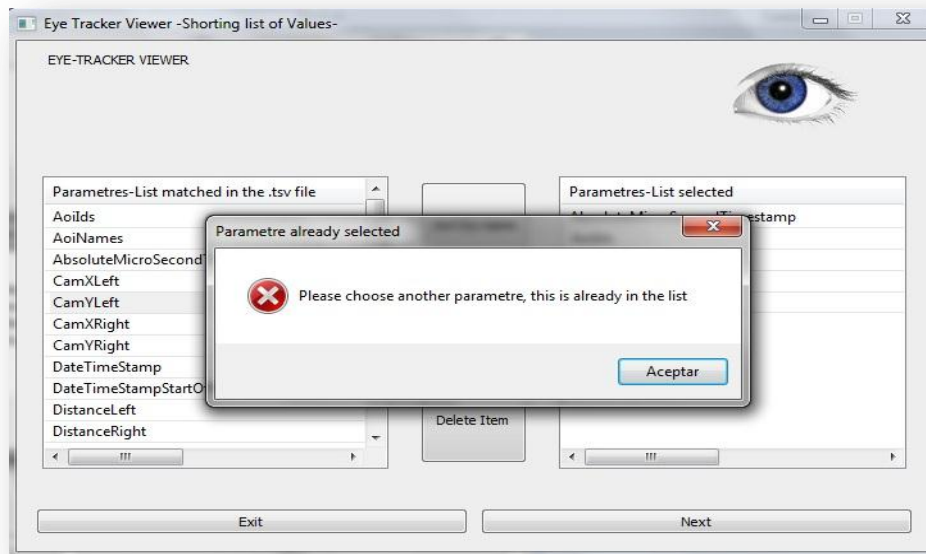


Figure 31 Error message implemented with Message Dialog

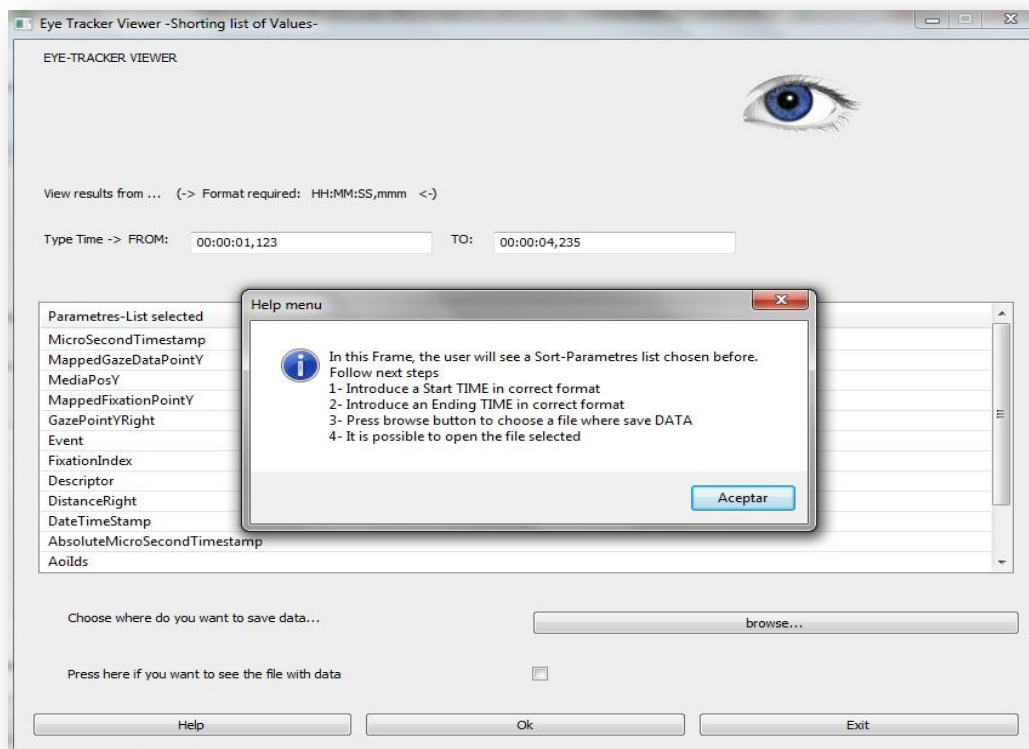


Figure 32 Information message implemented with Message Dialog

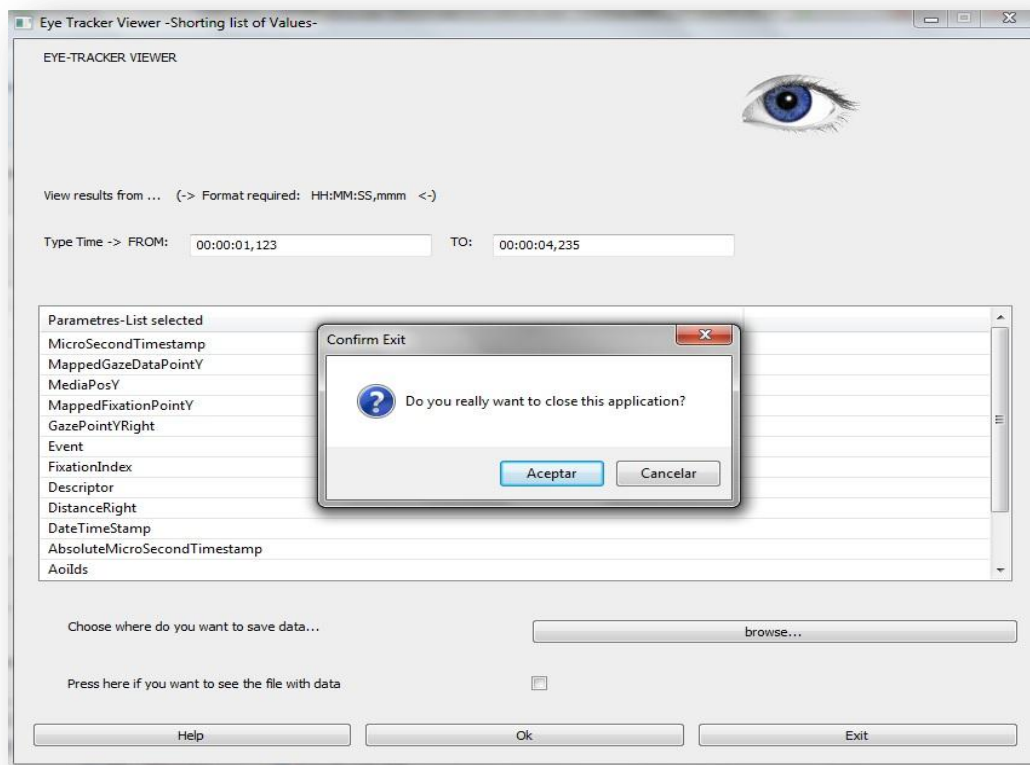


Figure 33 Question message implemented with Message Dialog

The code related with the figures displayed above is described below:

1. Alert message dialog displayed in the first figure:

```
def onDelete_Item(self, event):

    ix_selected3 = self.lc2.GetFocusedItem()
    if ix_selected3==-1:
        alertInfo = "Please follow the next steps:" + "\r1- You
        must select a parametre from the list on the left" + "\r2-
        Click and press Add Item to chose one" + "\r3- If you have
        already chosen one, please click one parametre and press
        Delete Button"

        alertBox = wx.MessageDialog(self,message=alertInfo,
        caption='First Select a Parametre in the list on the
        left',style=wx.ICON_EXCLAMATION | wx.STAY_ON_TOP | wx.OK)

        result = alertBox.ShowModal()
        alertBox.Destroy()

        if result == wx.ID_OK:
            alertBox.Destroy()
```

2. Error message dialog displayed in the second figure:

```
def onAdd_Item(self, event):

    ix_selected = self.lc.GetNextItem(item=-1,
    geometry=wx.LIST_NEXT_ALL, state=wx.LIST_STATE_SELECTED)

    state = names0[ix_selected]

    for data in s2:
        if str(state) in s2:
            data2 = 1
    dlg = wx.MessageDialog(self, "Please choose another parametre,
    this is already in the list ", "Parametre already selected ",
    wx.OK|wx.ICON_ERROR)

    result = dlg.ShowModal()
    dlg.Destroy()
```

3. Information message for the help window display in the *ThirdWindow.py* package:

```
def onHelpButton (self,event):

    d=wx.MessageDialog(self, "In this Frame, the user will
    see a Sort-Parametres list chosen before. Follow next
    steps"+"r1- Introduce a Start TIME in correct format" +
    "r2- Introduce an Ending TIME in correct format"+"r3-
    Press browse button to choose a file where save DATA" +
    "r4- It is possible to open the file selected", "Help menu",
    wx.OK)

    d.ShowModal()
    d.Destroy()
```

4. Finally the code related with the *exiting App* when is pressed the close cross in the upper right side of the *third frame*. It is displayed with the *interrogation icon*, and there are two possibilities gathered into *two buttons*.

```
def OnClose(self, event):  
    """Exit App"""  
    dlg = wx.MessageDialog(self,  
        "Do you really want to close this application?",  
        "Confirm Exit", wx.OK|wx.CANCEL|wx.ICON_QUESTION)  
  
    result = dlg.ShowModal()  
    dlg.Destroy()  
  
    if result == wx.ID_OK:  
        dlg.Destroy() & self.Destroy()
```

4.2.1.1.3 Importing OS Module

This module is provided in the *Python Standard Library* and it has been used mainly for *themenuv3.py package*, as a way to provide the way to open a new window, so the user can open the *.tsv file*, and read all the metrics that in future frames will be used.

Likewise, this module provides a portable way of using operating system dependent functionality. So as a matter of a fact if I just want to read or write a file I used the *open() method*, or even if I want to manipulate paths, the *os.path* module.

The *method* used in the development of this “*Eye-Tracking Data Extractor*” *Application* has been the *os.getcwd()* one. Below I will explain how this method works, and I will provide the user a brief resume with examples code, and an image so as to figure out the simple but useful tool that the *OS module* provides. Furthermore, I will explain one method from the *WX Module*, *wx.FileDialog()*, that even this module has been described in the last subchapter, has considered appropriate to explain in this one, as it relates to the method detailed below.

➤ Method used:

✓ `os.getcwd()`

Return a string representing the current working directory.

➤ Related method used from the wx module:

- Inheritance diagram for *wx.FileDialog*:



- Description of this method:

This method represents the file chooser dialog.

Pops up a *file selector box*. The path and filename are distinct elements of a full *file pathname*. If path is “”, the current directory will be used. If filename is “”, no *default filename* will be supplied. The wildcard determines what files are displayed in the file selector, and file extension supplies a type extension for the required filename.

- Windows styles:

Window Style	Description
<i>wx.FD_DEFAULT_STYLE</i>	Equivalent to <i>wx.FD_OPEN</i> .
<i>wx.FD_OPEN</i>	This is an open dialog; usually this means that the default button's label of the dialog is "Open". Cannot be combined with <i>wx.FD_SAVE</i> .
<i>wx.FD_SAVE</i>	This is a save dialog; usually this means that the default button's label of the dialog is "Save". Cannot be combined with <i>wx.FD_OPEN</i> .
<i>wx.FD_OVERWRITE_PROMPT</i>	For save dialog only: prompt for a confirmation if a file will be overwritten.
<i>wx.FD_FILE_MUST_EXIST</i>	For open dialog only: the user may only select files that actually exist.
<i>wx.FD_MULTIPLE</i>	For open dialog only: allows selecting multiple files.
<i>wx.FD_CHANGE_DIR</i>	Change the current working directory to the directory where the file(s) chosen by the user are.
<i>wx.FD_PREVIEW</i>	Show the preview of the selected files (currently only supported by wxGTK using GTK+ 2.4 or later).

Table7 OS module styles windows

- Method from the Class API:
- ✓ `__init__(parent,message=wx.FileSelectorPromptStr,defaultDir="",defaultFile="",wildcard=wx.FileSelectorDefaultWildcardStr,style=wx.FD_DEFAULT_STYLE, pos=wx.DefaultPosition)`
- Parameters:
- ✓ *parent* (wx.Window)
- ✓ *message* (string)
- ✓ *defaultDir* (string)
- ✓ *defaultFile* (string)
- ✓ *wildcard* (string)
- ✓ *style* (long)
- ✓ *pos* (wx.Point)

Here is the code and an image of how these two methods from different modules (OS, WX) are running in the first frame of the *Application*:

DESIGN OF A LOG-FILE MANAGMENT TOOL FOR EYE-TRACKING DATA

```
def loadEvent(self, event):  
    """Create a load dialogue box, load a file onto transferArea,  
    then destroy the load box."""  
  
    loadBox = wx.FileDialog(self, message="Open",  
        defaultDir=os.getcwd(), defaultFile="", style=wx.OPEN)  
    # When the user clicks 'Browse', do this:  
        if loadBox.ShowModal() == wx.ID_OK:  
            global fileName  
            fileName = loadBox.GetPath()  
            fileName2=loadBox.GetFilename()  
            #write the path to the textbox  
            tc2.SetValue(fileName)  
            loadBox.Destroy()
```

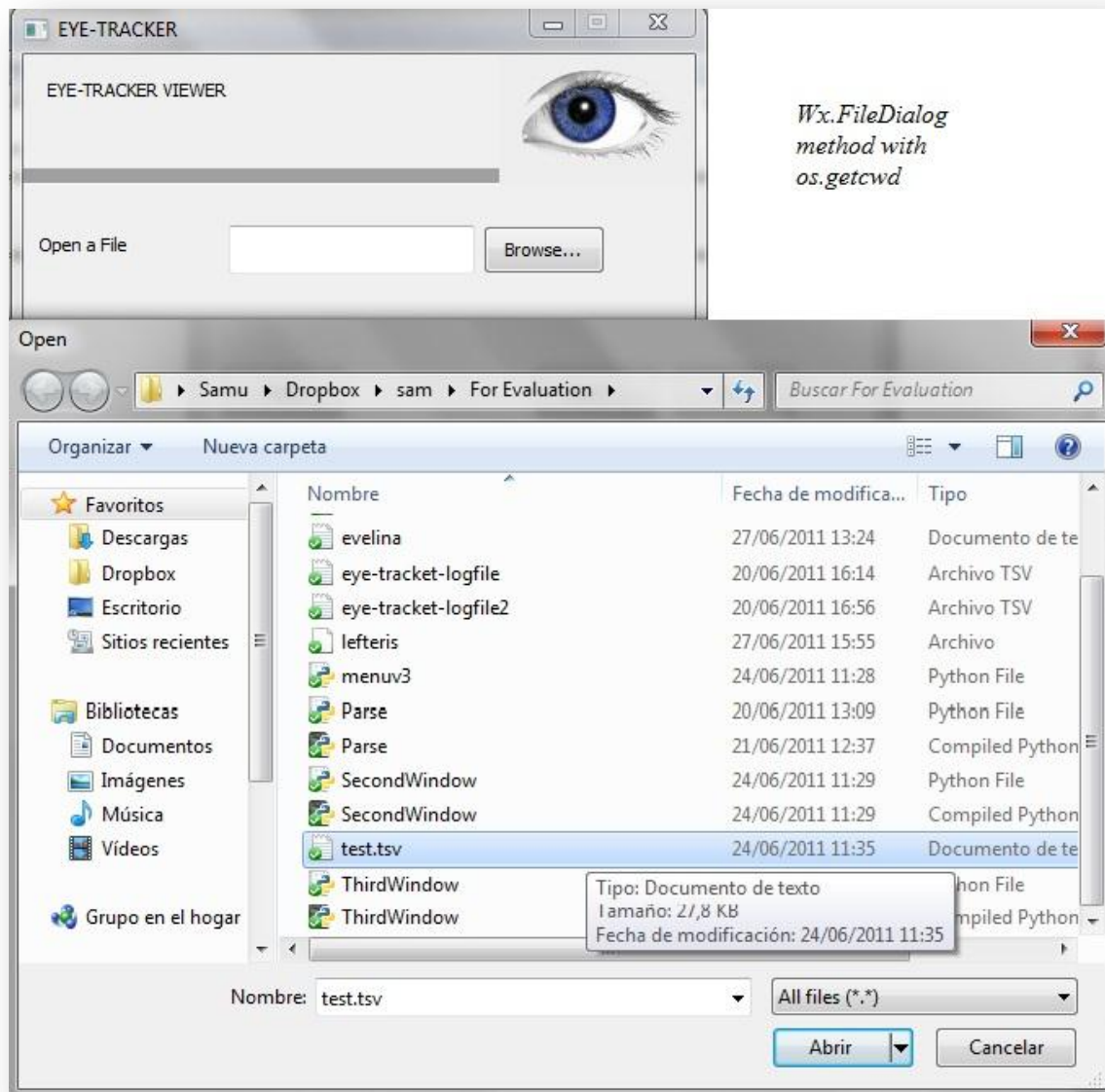


Figure 34 *wx.FileDialog* method with *os.getcwd*

4.2.1.1.4 Importing STRING Module into Parser.py Package

During the realization of this project called “*Eye-Tracking Data Extractor*” Application it has been very useful to have this *module*, importing a very useful tool, because it allows working with *text strings*.

The *string module* contains a number of useful constants and classes, as well as some deprecated legacy functions that are also available as methods on strings. In addition, *Python*’s built-in string classes support the sequence type methods that most of them has been instanced in this *App* such as *str()*, *unicode()*, *list()*, *tuple()*, *bytearray()*, *buffer()*, *xrange()*, and also some other string-specific methods.

Significantly, I will give a brief resume of the *re module* for string functions based on regular expressions.

➤ *string.split()* method:

- Method from the Class API:

string.split(s[, sep[, maxsplit]])

- Brief resume of how it works:

Return a list of the words of the *string s*. If the optional second argument *sep* is absent or None, the words are separated by arbitrary strings of whitespace characters (space, tab, newline, return, form feed). If the second argument *sep* is present and not None, it specifies a *string* to be used as the word separator. The returned list will then have one more item than the number of non-overlapping occurrences of the separator in the string. The optional third argument *maxsplit* defaults to 0. If it is nonzero, at most *maxsplit* number of splits occur, and the remainder of the string is returned as the final element of the list (thus, the list will have at most *maxsplit+1 elements*).

The behavior of split on an empty string depends on the value of *sep*. If *sep* is not specified, or specified as None, the result will be an empty list. If *sep* is specified as any *string*, the result will be a *list* containing one element which is an empty *string*.

This method gathered in the *STRING module* has provided with an enormous use, as it has been instanced in the *Parser.py package*. The main purpose has been to offer a way to collect all the data and try to *parse* it later. In this way, it has been gathered all the metrics of a *.tsv format file* (chosen by the user in the first window), and matched depending on whether the chosen *metrics* are available and put them graphically on the list.

Code gathered from the *Parser.py package* where the data is split from the *.tsv format file* chosen by the user:

```

import csv, operator, wx, string

portfolio = open(str(fileName2), "rb")

    for data in portfolio:
        data1= data.strip()
        names=data1.split('\t')

        if "Timestamp" in names:
            names0.append("Timestamp")

        if "DateTimeStamp" in names:
            names0.append("DateTimeStamp")

        if "DateTimeStampStartOffset" in
names:

names0.append("DateTimeStampStartOffset")

```

➤ **Re.match() method:**

- Method from the Class API:

re.match(string[, pos[, endpos]])

- Brief resume of how it works:

This method has been mainly used in the development of the *ThirdWindow.py* package. The main purpose of the using of it lies on the hand of be able to make time search, and whether they match the times listed in the *metric* called “*DateTimeStampStartOffset*”.

If zero or more characters at the beginning of string match this regular expression, return a corresponding *MatchObject()* instance. Return *None* if the string does not match the pattern; note that this is different from a zero-length match.

The optional *pos* and *endpos* parameters have the same meaning as for the *search()* method.

Here below is the code implemented in the last *Window*, and as we can see, the time introduced by the user, it has matched with and accuracy of decimals of second, as it has been the best way to approach the exact time gathered in the *.tsv* file format chosen by the user in the *Menuv3.py* package.

```
import string, re, wx, global
if re.match(value1[:10],names1[z][:10]):

    for names1[j] in portfolio:

        DateTimeStampStartOffset2=str(names1[j]).strip()

        names2=DateTimeStampStartOffset2.split('\t')
        portfolio2.write('\t'+names1[j]+'\\n')
```

4.2.2 WORKING IN A EVENT-DRIVEN ENVIRONMENT

4.2.2.1 Programming in an event-driven environment

Event handling is the fundamental mechanism that makes *wxPython* programs work. A program that works primarily via event handling is called event driven. In this chapter, we will discuss what an event-driven application is, and how it differs from a traditional application. We'll provide an overview of the concepts and terminology involved in GUI programming for the development of the "*Eye-Tracker Data Extractor*" Application, covering the interaction between the user, the toolkit, and the program logic. We'll also cover the lifecycle of a typical event-driven program.

An event is something that happens in the system which the application can respond to by triggering functionality. The event can be a *low-level user action*, such as a mouse move or key press, or a *higher level user action* given a specific meaning by *wxPython* because it takes place inside a *wxPython* widget, such as a button click or a menu selection. The event can also be created by the under lying operating system, such as a request to shut down. It has been even created own objects to generate own events. A *wxPython* application works by associating a specific kind of event with a specific piece of code, which should be executed in response (it will be shown below examples of the implemented code in the *Project*). The process by which events are mapped to code is called *event handling*.

4.2.2.2 What is event-driven programming

An event-driven program is mainly a control structure that receives events and responds to them. The structure of the *wxPython* program developed in this *Thesis* (or of any event-driven program) is fundamentally different from that of an ordinary *Python script*.

Atypical *Python script* has a specific starting point and a specific ending point, and the programmer controls the order of execution using conditionals, loops, and functions. The program is not linear, but its order is often independent of user action.

From the user's perspective, a *wxPython* program spends much of its time doing nothing. Typically, it is idle until the user or the system does something to trigger the *wxPython* program into action. The *wxPython* program structure is an example of an *event-driven program architecture*.

Think of the main loop of an event-driven system as analogous to an operator at a customer service call center. When no calls are coming in, the operator is, as it is said, standing by. Eventually, an event occurs, such as the phone ringing. The operator initiates a response process, which involves talking to the customer until the operator has enough information to dispatch the customer to the proper respondent for her call. The operator then waits for the next event.

Below is shown a simple diagram outlining the major parts of an *event-driven program*:

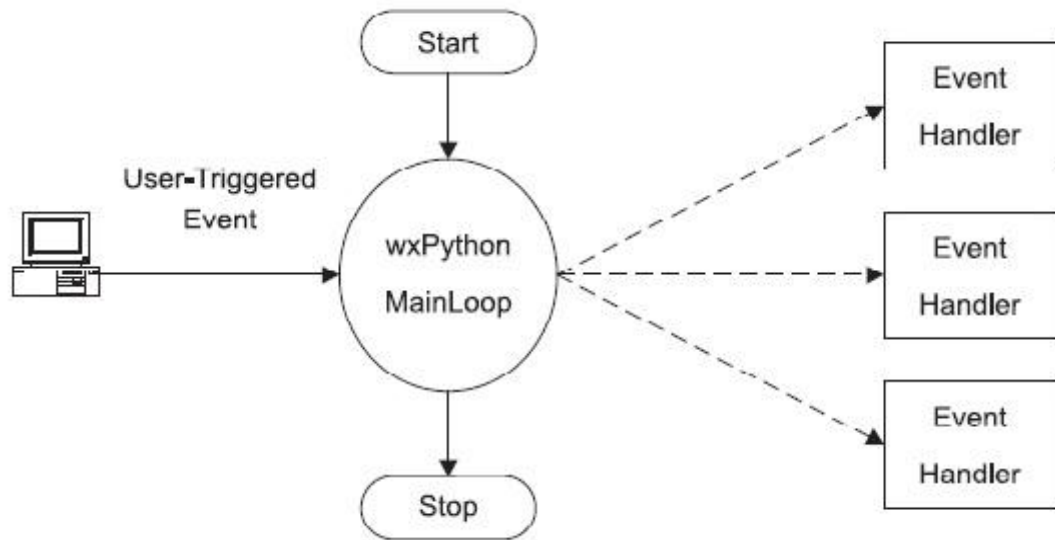


Figure 35 A schematic of the event handling cycle, showing the life of the main program, a user event, and dispatch to handler functions

Although each event-driven system is somewhat different, there are many similarities between them. The primary characteristics of an event-driven program structure are as follows [3]:

- ✓ After the initial setup, the program spends most of its time in an idle loop, where it does little or no information processing. Entering into this loop signifies the beginning of the user-interactive part of the program, an exiting the loop signifies its end. In *wxPython*, this loop is the method *wx.App.MainLoop()*, and is explicitly invoked in our script. The main loop is automatically exited when all top-level windows are closed.
- ✓ The program contains events that correspond to things that happen in the program environment. Events are typically triggered by user activity, but can also be the result of system activity, or arbitrary code elsewhere in the program. In *wxPython*, all events are instances of the class *wx.Event* or one of its subclasses. Each event has an event type attribute that allows different kinds of events to be distinguished. For example, a mouse up and mouse down event are both delivered as instances of the same class, but have a different event type.
- ✓ As part of the idle loop, the program periodically checks to see whether anything requiring a response has happened. There are two mechanisms by which an *event-driven* system may be notified about events. The more popular method, used by *wxPython*, posts the events to a central queue, which triggers processing of that event. Other *event-driven* systems use a polling method, where possible raisers of events are periodically queried by the central process and asked if they have any events pending.

- ✓ When an event takes place, the event-based system processes the event in an attempt to determine what code, if any, should be executed. In *wxPython*, native system events are translated to *wx.Event* instances and then given to the method *wx.EvtHandler.ProcessEvent()* for dispatching out to the proper handler code.

Figure 35 presents a basic overview of the process.

- ✓ The component parts of the event mechanism are event binder objects and event handlers. An event binder is a predefined *wxPython* object. There is a separate event binder for each event type. An event handler is a function or method that takes a *wxPython* event instance as an argument. An event handler is invoked when the user triggers the appropriate event.

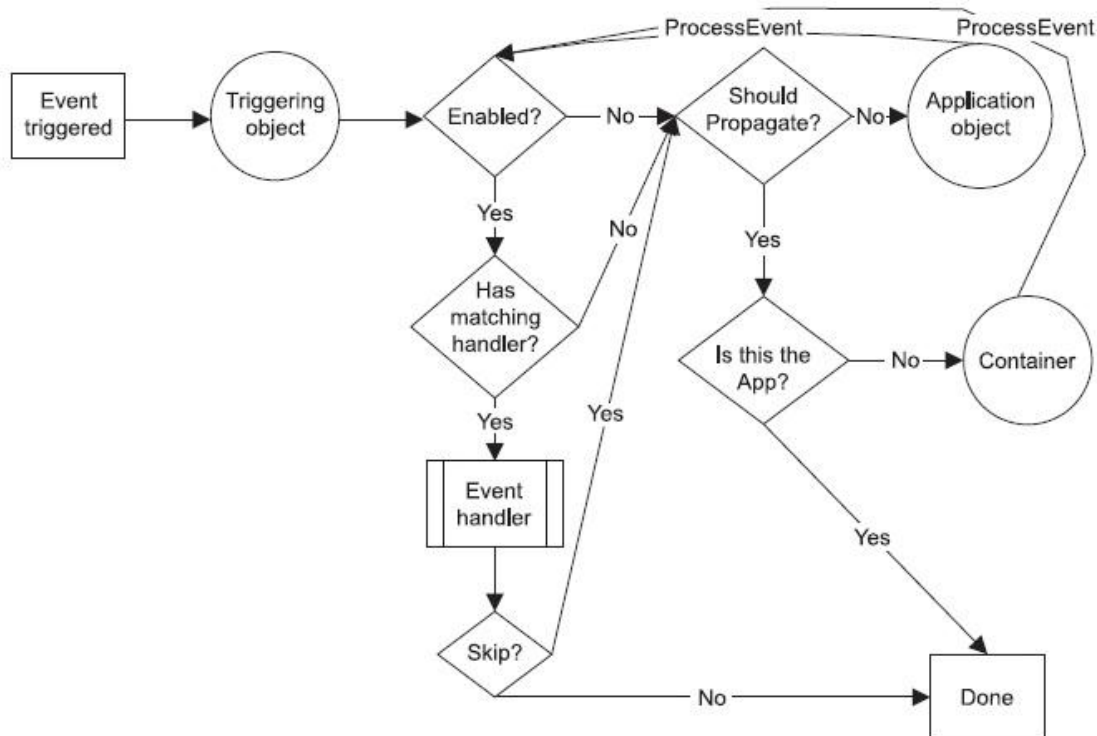


Figure 36 Even handling process, starting with the event triggered, and moving through the steps of searching for a handler

4.2.2.3 Binding Events in the Application

It is significant line item for all methods used in the development of binding events. This is the super class *wx.EvtHandler*, which handles all types of events produced by any interruption. Let us call interruption, as previously described in this *Thesis*, to any caused by the click of a mouse, by pressing a button, by marking a check box, etc. Detailed below, the super class *wx.EvtHandler* method is described and methods used for each of these interruptions as well as of code that describes each one.

➤ **Wx.EvtHandler**

Inheritance diagram for wx.EvtHandler:



○ Description:

- A class that can handle events from the windowing system. *wx.Window* and therefore all window classes are derived from this class.
- When events are received, *wx.EvtHandler* invokes the method listed in the event table using itself as the object.
- When using multiple inheritance it is imperative that the *wx.EvtHandler* (-derived) class be the first class inherited such that the “*self*” pointer for the overall object will be identical to the “*self*” pointer for the *wx.EvtHandler* portion.

○ Methods from the Class API:

Bind(event, handler, source=None, id=-1, id2=-1)

- Bind an event to an event handler.

- Parameters:

- **Event** (*wx.Event*): One of the *wx.EVT_** objects that specifies the type of event to bind,
- **Handler** (*PyObject*): A callable object to be invoked when the event is delivered to self. Pass *None* to disconnect an event handler.
- **Source** (*wx.Window*): Sometimes the event originates from a different window than self, but you still want to catch it in self. (For example, a button event delivered to a frame). By passing the source of the event, the event handling system is able to differentiate between the same events types from different controls.
- **id**(*int*): Used to specify the event source by ID instead of instance.
- **id2**(*int*): Used when it is desirable to bind a handler to a range of IDs, such as with *wx.EVT_MENU_RANGE*.

Below is detailed part of the code that has been implemented in the development of the “*Eye-Tracker Data Extractor*” Application, where there are detailed some of the methods connected with the *Widgets* used in the *wx module* of *Python*.

```
# Bind button clicks, introducing time in text control ...

self.Bind (wx.EVT_BUTTON, self.onSave, save_button)
self.Bind (wx.EVT_CHECKBOX, self.onClick, view_button)
self.Bind (wx.EVT_BUTTON, self.onHelpButton, Help_button)
self.Bind (wx.EVT_BUTTON, self.exitEvent, Exit_button)
self.Bind (wx.EVT_TEXT, self.On_DataTime, time1)
self.Bind (wx.EVT_TEXT, self.On_DataTime, time2)
self.Bind (wx.EVT_BUTTON, self.onOK_Button, OK_button)

# Below should be all the methods connected with the bindings..

defonSave()
defonClick()
defonHelpButton()
defexitEvent()
defOn_DataTime()
defonOK_Button()
```

CHAPTER 5

EVALUATION

5.1 INTRODUCTION

The main reason to make an evaluation of the program lies in the novelty of it. It must be said that this was the first time it is used a tool like the *"Eye-Tracking Data Extractor"* implemented with the *Python language* to allow the user to collect *metrics* and information on a *.tsv format file* recording, as a result of the Tobii Eye Tracker Monitor.

Significantly, it has sought an assessment *method* to help us refine the possible future improvements in the *App program*. We have used the *Think Aloud method*, in a scene of two *ordinary users*, and secondly in a scenario with *two expert users*.

Here below is explained how this method works, in order to give a brief resume and figure out the main purpose of using it:

Think Aloud (TA)[6] studies provide rich verbal data about reasoning during a problem solving task. Using *TA* and protocol analysis, we can identify the information that is concentrated on during problem solving and how that information is used to facilitate problem resolution. From this, inferences can be made about the reasoning processes that were used during the *problem-solving* task. In the past, the validity of data obtained from *TA* studies has been suspect because of inconsistencies in data collection and the inability to verify findings obtained from the slow, laborious process of protocol analysis. But nowadays, with this evaluation made in this *Project*, it describes a means of obtaining more accurate verbal data and analyzing it in a standardized step-by-step manner.

5.2 SELECTING SUBJECTS IN TA

5.2.1 CRITERIA FOR SELECTING SUBJECTS

Both, subjects and tasks must thus be selected that the effect of possible disruptive effects of thinking aloud is minimized. The cognitive process in which we are interested should occur when the task is presented to the subject, disruption of the process by thinking aloud should be minimized and so should synchronization problems and working memory overload. Both in scientific research and in knowledge acquisition one does not always have a choice. Research may be directed at a particular kind of persons and we need a random sample of those because the results must be generalized over all persons of this kind. In knowledge acquisition it is often difficult to get access to an expert and one often cannot choose. Two important properties of subjects with regard to the applicability of the think aloud method are the degree of experts and verbalization skills.

In this way, for the realization of this evaluation, we have chosen two different groups of subjects. The first is in relation to a *common subject* or user, and the second a *subject or expert user*.

5.2.2 EXPERTS AS SUBJECTS

If the *think aloud* method is used for the elicitation of expert knowledge several problems are likely to occur. Expert knowledge is often partially '*compiled*' in the sense that experts are able to perform a task very well, but that they cannot explain how they found the right answer (*'I just saw that it had to be this'*).

The *think aloud* method makes some of this knowledge visible. On several occasions we observed that *experts* were able to make their knowledge explicit in a discussion afterwards about their *think aloud* protocols and our analysis of the protocols. However, as with *regular subjects*, *experts* that perform a task as a routine and very fast are unable to verbalize their thoughts during this performance. This is the reason why the *experts* have evaluated the user interface and provided comments, while the users worked in the scenario and used this protocol, called *Think Aloud (TA)* [6].

5.3 EVALUATION

As it has been explained in the previous subchapter, the main idea lies in providing both groups of users a scenario, in which the “*Eye-Tracking Data Extractor*” Application was evaluated.

- There have been **four participants** in total:
 - *Two typical or normal users.*
 - ✓ They have been working on a scenario and used the *Think Aloud Protocol* evaluation method. Comments and conclusions were drawn regarding their activity.
 - .
 - *Two experts evaluator subjects*
 - ✓ They have evaluated the *User Interface (UI)* and provided comments.
- **Scenario created** for two **normal users**:

By using the Eye-Tracking Data Extractor complete the following:

1. *Export to a text file the metrics* named
 - ✓ *AOIDS*
 - ✓ *MappedFixationPointX*
 - ✓ *MappedFixationPointY*

from the logfile named “*eyetracker-logfile.tsv*”
2. *Gather all the metrics chosen for this time period*
 - ✓ *FROM: 00:00:01,123*
 - ✓ *TO: 00:00:05,246*
3. *View your results.*

5.4 RESULTS

Here below are resume the results of both two groups. The *normal users* group tried to explain as the way of the *Think Aloud evaluation protocol*, their reactions during the process of going through the scenario provided. On the other hand, the *experts group*, commented writing down their impressions about how the *Application* works, and even what should be done in future developments.

➤ Normal users group:

I. FIRST USER.

✓ Reactions collected:

- *It was a little difficult to understand at the end that it was necessary to insert a filename so the program could save the data.*
- *Also believes that the file should write the metrics in different columns.*
- *Overall it was an easy to use this application*

II. SECOND USER.

✓ Reactions collected:

- *Too slow after pressing “next” button in the first frame. No indication of loading the file*
- *In the parameters list in the second frame, Timestamp and DateTimeStamp were not clearly displayed. Also the word “parameter” was misspelled*
- *The produced file is not very easy to read and understand. Metrics should not be written one after the other, no spaces between results and the title of the next metric.*

➤ Experts users group:

III. FIRST USER.

✓ Reactions collected:

○ Firstframe:

- *The progress bar is missing. So there is no system’s feedback while a file is opened*

○ Secondframe:

- *“sort by name” should be placed over the first text box since it refers only to this.*
- *A “back” button redirecting to the first frame should be provided*

- Thirdframe:
 - *A “back” button redirecting to the third frame should be provided*
 - *Feedback should be provided through a progress bar or hourglass while the text file is produced*
 - *Interface is not well organized in terms of space*

IV. SECOND USER.

- ✓ *Reactions collected:*
 - *Next button should be deactivated if no file has been chosen.*
 - *Feedback while waiting for .tsv file to load*
 - *Mouse over explanation for every metric would be useful*
 - *"Sort by name" should be placed above the metric list*
 - *Help should be relative to the individual tasks. If delete is pressed and no item was selected should not read "you must select a parameter from the list on the left"*
 - *Multiple item selection should work also for adding multiple items to the right list*
 - *Exit button should not be so prominent. The possibility to press it by accident is bigger.*
 - *Information about total running time of the selected file should be given to users.*
 - *The timestamp format should be displayed by default (e.g 00:00:00,000) in the textbox. User then can only change values.*

5.5 CONCLUSIONS

By using the method of evaluation, it was possible to know which are the impressions and reactions of different users studied. Note the importance of using a fully recognized and valid method such as the *Think Aloud protocol* for evaluation, thus giving greater accuracy and depth to the study.

In this way have been collected in written form evaluations on the use of this application by even a normal user as also an expert one. Thanks to these users for future improvements of the program (remember that for the realization of this project has not been able to count on any previous version or a tool like this), it will be possible to make changes and improvements in many facets of design and functionality so that the user will find this application more complete.

It should be noted that some of these improvements have been solved, in order that did not require so much time to implement them (e.g changes in the spelling of a button, or fixing the disposition of a button). Still, it remains open availability of the *Project* for future use and improvement.

CHAPTER 6

CONCLUSIONS

The motivation of this thesis and the application implemented resulted from the need to develop a tool to facilitate the reading of data collected after making a recording with the *Eye Tracker* monitor.

It is a tedious, lengthy and cumbersome labor that the user has for their own resources to get information to highlight the metrics that are important to use. The main reason lays on the *tsv format files* resulted of the record. This is the main reason why this *Project* has been developed.

Never before had disposed of this tool. In this way has attempted to the first version of an application that would provide the user a clear and effective way to solve the problem resided in summarized in the way as possible all data collected and displayed in a new *tsv format file*.

By using the *Python language*, more specifically the graphic tool provided by the *wxmodule*, has been able to learn a new programming language. Likewise, it has been a great help as a language to be so intuitive, but the simplicity sometimes has resulted in complex algorithms, or search for methods to facilitate the work.

Therefore it has had a modern graphical tool, according to a modern programming language that is rising nowadays. It also highlights the intention of offering a simple and nice graphic interface for the user, through packages or modules imported by *Python widgets*.

As further improvements can be handled with the evaluation method described in the previous chapter, as well as new windows to improve the efficiency in obtaining the metrics, to offer the user the data collected, etc. It is thought to include the recorded video of the *Eye-Tracker* monitor in the application itself and thus give the user a graphical way to choose the times between which he wants to gather the data.

Referencies

- [1] “*Tobii Product Description TX Series Eye Trackers.*” Revision 2.1, June 2010, [http://www.tobii.com/en/eye-tracking-integration/global/products-services/hardware/eye-tracking-academy/\(tobii\)](http://www.tobii.com/en/eye-tracking-integration/global/products-services/hardware/eye-tracking-academy/(tobii))
- [2] Pablo Garaizar Sagarmiaga. University of Deusto (Vasque Country. SPAIN). “*Review in designing of User Interfaces*” 2006-28-08, Web site: [http://blog.txipinet.com/2006/08/28/28-estudio-sobre-el-diseno-de-guis-iii-la-importancia-del-diseno/\(deusto\)](http://blog.txipinet.com/2006/08/28/28-estudio-sobre-el-diseno-de-guis-iii-la-importancia-del-diseno/(deusto))
- [3] Noel Rappin and Robin Dunn “*Wxpython in Action*” Manning Publication Co. – Greenwich (2006), ISBN 1-932394-62-1
- [4] René-Lévesque Ouest, 2008-12-12 Montréal, Québec, Canada, *Les Innovations VLAM inc.* “*Benchmark study between Boost and SWIG wrappers for Python*”, <http://foobrac.blogspot.com/2008/12/boostpython-vs-swigpython-benchmark.html>
- [5] Robin Dunn and Harri Pasanen, Developers of the *wxpython* programming language
Web site: <http://www.wxpython.org/>
- [6] Maarten W. van Someren, Yvonne F. Barnard, Jacobijn A.C. Sandberg, “*The Think Aloud Method*” “*A practical guide to modeling cognitive processes*”, Department of Social Science Informatics, University of Amsterdam, Published by Academic Press, London, 1994, ISBN 0-12-714270-3, Copyright M.W. van Someren, Y.F. Barnard and J.A.C. Barnard, Web site: <http://www.staff.science.uva.nl/>

GLOSSARY

HCI:	H uman C omputer I nteraction
TSV:	T ab S eparate V alues
GUI:	G raphical U ser I nterface
APP:	A pplication
API:	A pplication P rogramming I nterface
UI:	U ser I nterface
CPU:	C omputer P rocessing U nit
IDLE:	I ntegrate D evelopment E nvironment for Python
IDE:	I ntegrate D evelopment E nvironment
GNU:	G NU's N ot U NIX
SWIG:	S implified W rapper and I nterface G enerator
TA:	T hink A loud protocol method of evaluation
OS:	O perative S ystem
ID:	I dentifier
VGAP:	V ertical G ap
HGAP:	H orizontal G ap
AWT:	A bstract W indow T oolkit
GTK:	G imp T ool K it
QT:	Q T widgets T ool K it
TFT:	T hin F ilm T ransistor
HTML:	H yper T ext M arkup L anguage
PC:	P ersonal C omputer
RAM:	R andom A ccess M emory
SDK:	S oftware D evelopers K it
URL:	U niversal R esource L ocator
USB:	U niversal S erial B us
VM:	V irtual M achine
WST:	W eb S tandard T ools

