UNIVERSITÀ DEGLI STUDI DI PADOVA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA BIOMEDICA

DESIGN OF NEUROPHYSIOLOGICAL SIGNAL ANAYSIS SOFTWARE

GIACOMO BASSETTO

SUPERVISOR: PROFESSOR STEFANO VASSANELLI

CO-SUPERVISOR: DOTT. MUFTI MAHMUD

PADOVA, 30 SEPTEMBER 2010

“Ah-ah” (Nelson — The Simpsons)

ACKNOWLEDGEMENTS

My thanks to prof. Stefano Vassanelli and dott. Mufti Mahmud, for the opportunity they gave letting me to have this experience.

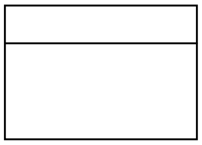
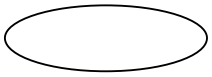
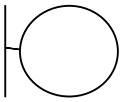
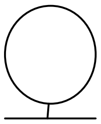
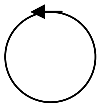
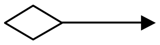
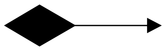
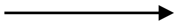
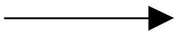
Special thanks to Elisabetta, Marta, Stefano and Azzurra, because they proved to be a valid company as long all this period, even during those days of hardest work.

Finally, last but not for their importance, thanks with love to all my family, who support and bear (mi sUpportano e sOpportano) me every day.

INDEX

LIST OF FIGURES

LIST OF SYMBOLS

	Class / Interface
	Use Case
	Boundary
	Physical Entity
	Program Control / Logic
	Aggregation
	Composition
	Association
	Inheritance

ABSTRACT

Advances in neuronal probe technology to record brain activity make now possible very deep spatial and temporal resolution in signal recording. In order to be able to infer significant conclusions from experimental recordings, sophisticated signal processing tools have been adapted by different laboratories. However, in the scientific community till date there is no unique software that performs all the signal processing and analyses and is shared among the laboratories. Though there are a few packages available incorporating few techniques together, they rather are very specific to their data and kind of analyses they want to perform, rather than being open for all the laboratories. Keeping this in mind, recently, the NeuroChip Laboratory of the University of Padova, Italy has proposed such a software (named "SigMate") incorporating most of the tools in one single application, in order to render the analysis faster (avoiding time consuming file conversion to analyze the same set of data with different tools) and more accessible, as scientists will deal with one single application instead of with a growing number of small and performing atomic operations programs. However, this software package is developed in Matlab (http://www.mathworks.com/) following a functional programming approach. To facilitate the inter-platform operability and reusability of the software package object oriented design is required. In this work, we have tried to provide object oriented design of some of the modules of SigMate. Due to the lack of time all the modules could not be converted and eventually, in the time course, as a future direction, the software would be completely transformed to object oriented platform for being adapted by the scientific community.

CHAPTER 1: INTRODUCTION

Progresses in microelectrode array technology now allow simultaneous recording from large populations of neurons, thus giving the possibility to understand the complex relationship between neurophysiology and behavior. The arising drawback is that each recording produces a continuously growing amount of data. Processing these massive amounts of data to extract critical biological information from hundreds of neurons in noisy recording situations is a major challenge in many experimental settings. For this reason visualizing and processing the large amounts of data generated by modern recording systems requires efficient computer software. Very few software tools exist to date that integrate all the needed processing steps in one comprehensive package (a brief description of them is given below) [2] [3] [4] [5]. This is coupled with a significant lack standardized set of tools that enables replicating experimental data analysis across labs to facilitate objective comparison between scientific findings. As result, the sharing of experimental data across the web, common in many fields of biology, is compromised.

Moreover, although many academic and commercial neurophysiological signal processing software have been developed, almost no one of them is comprehensive of all the steps required in extracellularly recorded signal analysis. Therefore, it is required to use multiple packages to analyze the same dataset, which makes the analysis cumbersome and time consuming.

1.1 NEUROSCOPE

NeuroScope [3] is a GNU GPL distributed software designed to help neurophysiologists to process and view recorded data in an efficient and user-friendly manner. This software package consists of several applications and tools designed to assist the experimenter in extracting and exploring data collected in experiments ranging from acute recordings in anesthetized animals to complex chronic recordings where brain signals are recorded from freely moving animals as they perform behavioral tasks in automated apparatuses.

This software is distributed both in binary and source forms. Developers assure they have been very careful in developing high-quality and documented code. Having this software been developed within a neuroscience laboratory, the feature set was directly chosen and defined by experimenters. Moreover, constant and direct user feedback made easy and efficient user interfaces.

1.2 NEUROQUEST

The program [4] is a Matlab developed package studied to give support for spike detection and analysis on experimental data. This tool is very user friendly, and a typical use can be summarized as follows. The user examines the dataset and then can select segments of the data for further analysis. The following stage consists in denoising the data to enhance the neural yield. After that, spikes are detected. Detected spikes are extracted from the data and sent to the spike sorting algorithm. Several parameters can be specified in the spike sorting GUI such as spike length, type of the array, clustering method, and type of feature for spike sorting.
The obtained spike trains are then further analyzed using the primary spike train analysis tools such as ISIH, PSTH, and cross-correlogram.

1.3 SIGTOOL

The package runs in the MATLAB programming environment and has been designed to promote the sharing of laboratory-developed software across the worldwide web. It provides features as spike and waveform analysis for recorded neural signals.
Waveform analysis exposes useful and common functionssuch as waveform averaging (mean and median), auto- and cross-correlation, power spectral analysis, coherence estimation, digital filtering (feedback and feedforward) and resampling. Spike-train analyses include interspike interval distributions, Poincaré plots, event auto- and cross-correlations, spike-triggered averaging, stimulus driven and phase-related peri-event time histograms. Developers let users the freedom of writing their own extensions that will be added to the sigTOOL interface on-the-fly, without the need to modify the core sigTOOL code.

CHAPTER 2: SOFTWARE DESIGN

2.1 DESIGN PATTERNS

Designing object-oriented software is hard, and designing *reusable* object-oriented software is even harder. For this reason we adopted some of the design patterns described by the gang of four [1] while projecting our software.

2.1.1 WHAT DESIGN PATTERNS ARE?

In software engineering, a design pattern is a reusable general solution to a commonly occurring problem. It is not a library or a software reusable component. Rather it is a model to apply to solve a problem that might take place in different situations when designing and developing software. Object oriented design patterns show interactions and relationships among classes or objects, without specifying final classes involved in the application.

In general, a pattern has four essential elements:

1. The **pattern name** is a handle we can use to describe a design problem, its solutions, and consequences in a word or two. It lets us design at a higher level of abstraction.
2. The **problem** describes when to apply the pattern. It explains the problem and its context. It might describe specific design problems such as how to represent algorithms as objects. It might describe class or object structures that are symptomatic of an inflexible design. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply the pattern.
3. The **solution** describes the elements that make up the design, their relationships, responsibilities, and collaborations. The solution doesn't describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, the pattern provides an abstract description of a design problem and how a general arrangement of elements (classes and objects in our case) solves it.

4. The **consequences** are the results and trade-offs of applying the pattern. The consequences for software often concern space and time trade-offs. They may address language and implementation issues as well. Since reuse is often a factor in object-oriented design, the consequences of a pattern include its impact on a system's flexibility, extensibility, or portability.

2.2 OBSERVER

2.2.1 DESCRIPTION AND APPLICABILITY

The Observer Pattern [1] is a design pattern used when we need to control the state of different objects mainly when one of the following situations occurs:

- an abstraction has two aspects one dependent on the other, thus encapsulating these aspects in separate objects makes the code more reusable.
- a change to one object requires changing others, and the number of how many objects need to be changed is not known priory.
- an object should be able to notify other objects without making assumptions about who these objects are.

2.2.2 STRUCTURE

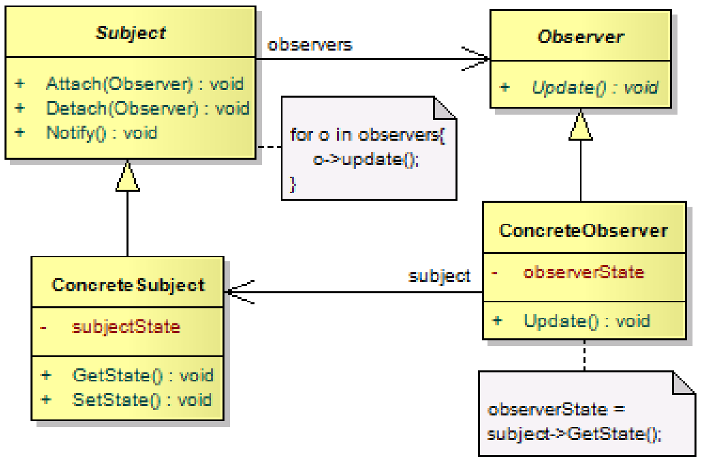


Fig. 2.1 Observer Pattern

- **Subject**
  - knows its observers. Any number of Observer objects may

observe a subject.

  - provides an interface for attaching and detaching Observer objects.

- **Observer**
  - defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**
  - stores state of interest to ConcreteObserver objects.
  - sends a notification to its observers when its state changes.
- **ConcreteObserver**
  - maintains a reference to a ConcreteSubject object.
  - stores state that should stay consistent with the subject's.
  - implements the Observer updating interface to keep its state consistent with the subject's.

2.2.3 CONSEQUENCES

The Observer pattern lets programmers vary subjects and observers independently. Subjects can be reused without reusing their observers, and vice versa. Observers can be added without modifying the subject or other observers.

Further benefits and liabilities of the Observer pattern include the following:

1. *Abstract coupling between Subject and Observer.* All a subject knows is that it has a list of observers. As the subject doesn't know the concrete class of any observer, the coupling between subjects and observers is abstract and minimal.

   Because Subject and Observer aren't tightly coupled, they can belong to different layers of abstraction in a system.

2. *Support for broadcast communication.* Unlike an ordinary request, the notification that a subject sends needn't specify its receiver. The notification is broadcast automatically to all interested objects

that subscribed to it. This gives the freedom to add and remove observers at any time. It's up to the observer to handle or ignore a notification.

3. *Unexpected updates.* Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject.

   This problem is aggravated by the fact that the simple update protocol provides no details on *what* changed in the subject. For this reason we added additional information on the update methods in our design.

2.3 ITERATOR

2.3.1 DESCRIPTION AND APPLICABILITY

This pattern allows programmers to traverse through all the element of a collection, regardless its specific implementation [1].

Use the Iterator pattern

- to access an aggregate object's contents without exposing its internal representation.
- to support multiple traversals of aggregate objects.
- to provide a uniform interface for traversing different aggregate structures (that is, to support polymorphic iteration).

2.3.2 STRUCTURE

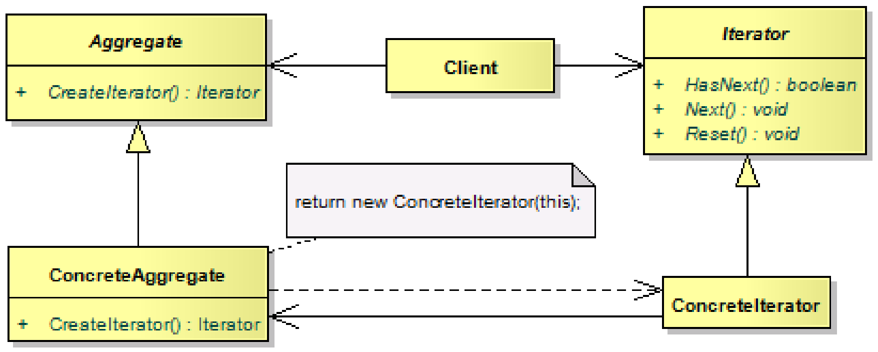


Fig. 2.2 Iterator Pattern

- **Iterator**
  - defines an interface for accessing and traversing elements.
- **ConcreteIterator**
  - implements the Iterator interface.
  - keeps track of the current position in the traversal of the aggregate.
- **Aggregate**
  - defines an interface for creating an Iterator object.
- **ConcreteAggregate**
  - implements the Iterator creation interface to return an instance of the proper ConcreteIterator.

2.3.3 CONSEQUENCES

The Iterator pattern has three important consequences:

1. *It supports variations in the traversal of an aggregate.* Complex aggregates may be traversed in many ways: just replace the iterator instance with a different one. Iterator subclasses can be defined to support new traversals.
2. *Iterators simplify the Aggregate interface.* Iterator's traversal interface obviates the need for a similar interface in Aggregate, thereby simplifying the aggregate's interface.
3. *More than one traversal can be pending on an aggregate.* An iterator keeps track of its own traversal state. Therefore more than one traversal in progress can exist at once.

2.4 BRIDGE

2.4.1 DESCRIPTION AND APPLICABILITY

Decouple an abstraction from its implementation so that the two can vary independently.

When an abstraction can have one of several possible implementations, the usual way to accommodate them is to use inheritance. An abstract class defines the interface to the abstraction, and concrete subclasses implement it in different ways. But this approach is not always flexible enough. Inheritance binds an implementation to the abstraction permanently, which makes it difficult to modify, extend, and reuse abstractions and implementations independently [1].

Use the Bridge pattern when

- it is desired to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
- both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets programmers combine the different abstractions and implementations and extend them independently.
- changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
- there is a big proliferation of classes. Such a class hierarchy indicates the need for splitting an object into two parts.
- an implementation is shared among multiple objects, and this fact should be hidden from the client.

2.4.2 STRUCTURE

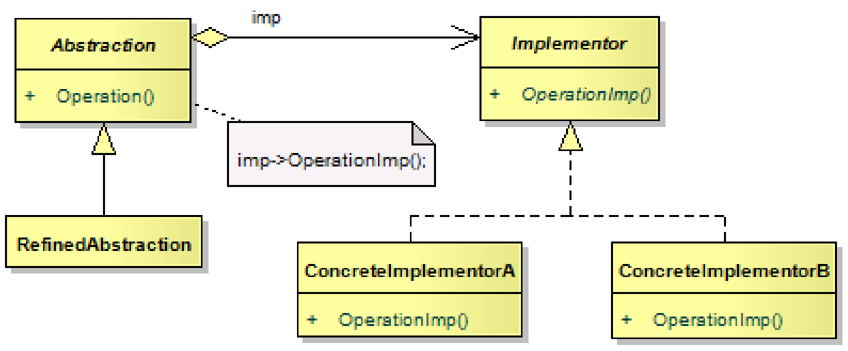


Fig. 3.3 Bridge Pattern

- **Abstraction**
  - defines the abstraction's interface.

    - maintains a reference to an object of type Implementor.
- **RefinedAbstraction**
    - Extends the interface defined by Abstraction.
- **Implementor**
    - defines the interface for implementation classes. This interface doesn't have to correspond exactly to Abstraction's interface; in fact the two interfaces can be quite different. Typically the Implementor interface provides only primitive operations, and Abstraction defines higher-level operations based on these primitives.
- **ConcreteImplementor**
    - implements the Implementor interface and defines its concrete implementation.

2.4.3 CONSEQUENCES

1. *Decoupling interface and implementation.* An implementation is not bound permanently to an interface. The implementation of an abstraction can be configured at run-time. It's even possible for an object to change its implementation at run-time.

   Decoupling Abstraction and Implementor also eliminates compile-time dependencies on the implementation. Changing an implementation class doesn't require recompiling the Abstraction class and its clients.

   Furthermore, this decoupling encourages layering that can lead to a better-structured system. The high-level part of a system only has to know about Abstraction and Implementor.
2. *Improved extensibility.* Abstraction and Implementor hierarchies can be extended independently.
3. *Hiding implementation details from clients.* Clients can be shield from implementation details, like the sharing of implementor objects.

CHAPTER 3: INTRODUCTION TO THE SOFTWARE

SigMate [2] is designed to perform various processing and analysis on the neuronal data files. The use case diagram of the software package can be seen in Figure 3.1. The main functionalities included at present are: data display (2D and 3D) with zooming, panning and data cursor options, artifact removal (including baseline correction), noise characterization with noise estimation, file operations (including file splitting, file concatenating and file column rearranger), latency estimation with the possibility to detect the cortex layer activation order automatically, and the spike train analysis package. The rest of the features are in-house developed algorithms, rigorously tested on datasets recorded using standard micropipette and EOSFETs from anesthetized rats.

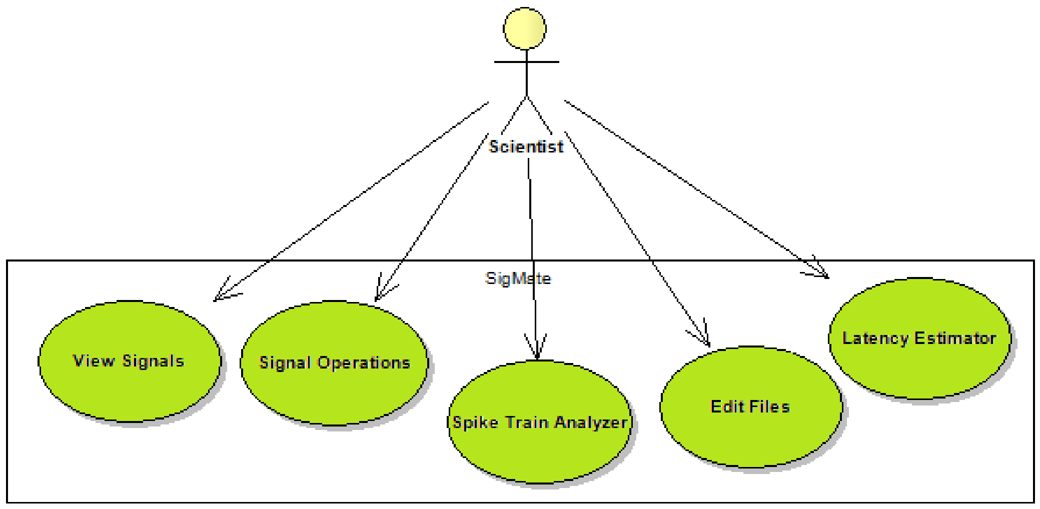


Fig 3.1: System Use Case Diagram

3.1 DATA DISPLAY

This is the main functionality of the application, the first module presented to the user once he launches the application. Providing the user with the flexibility in viewing the data in 2D and 3D, it also gives the possibility to perform averaging of single sweeps, estimate the noise, perform +/- averaging, and calculate the mean square and root mean square. The noise estimation information is used in further signal operation and when saving the signal.

3.2 ARTIFACT REMOVAL

This module performs artifact removal as well as baseline correction. It expects the control signals (recorded without providing stimulation) and signals with response upon stimulation. Peaks-and-valleys in the signal are detected using an in-house algorithm and uses their average to determine an estimation of the signal. The average of the estimation is used for the baseline correction. Artifact removal is obtained by subtracting the estimation of the control signal from the signal itself.

3.3 NOISE CHARACTERIZATION

Quality of recorded signals and quantification of the noise present in the signal are assessed using this module, designed for this purpose. It uses *in-house* algorithms to detect the first steady-state (the part of the signal before the evoked response), the second steady-state (the part of the signal from the end of response until the end of signal) and fit mathematical model to calculate the measurement error (ME) present in the signal. First order statistical information such as mean and standard deviation of the ME are used to quantify the noise. Moreover the module performs calculations to retrieve the distribution of the noise and its estimation, which are finally shown to the user.

3.4 FILE OPERATIONS

Few basic file operations are being incorporated in the software package that are often time consuming for the scientists who use different software for signal recording and performing signal processing and analysis. These operations include: file splitter, that splits a multi-sweep file into single sweeps based on the recording sampling frequency, file concatenator, that merges multiple single-sweep files into a multi-sweep file, and file column rearranger, whose purpose is to retain only the selected channels discarding the unselected ones.

3.5 LATENCY ESTIMATION

With the latency estimation module scientists can select a set of files and then detect events present in the signals contained in these files and calculate the latencies from the starting of the evoked response. The module first detects the response-onset, then the latencies are calculated from this response by detecting the various signal events present in each signal. Automated detection of the layer of recording is determined based on the signal characteristics and events just detected. Finally the order of activation of different cortical layers in the barrel column due to the whisker stimulation is determined by sorting the layer-wise average of the calculated latencies of the second event.

CHAPTER 4: DATA DISPLAY AND PROCESSING

During our analysis of the system, we noticed that data visualization is one of the major and common operations that the scientists want to perform regardless whatever operation they want to perform later on their data set. For this reason we found logical to begin our design from this aspect of the application, and then expand the design to include the various functionality.

4.1 ANALYSIS

First of all, we have to understand what the user expects and what he/she has to do to visualize a recorded signal.

Our analysis brought us defining the following points:

- We incorporated two different kinds of recording techniques to obtain neuronal signals from the brain (by a micropipette or by a transistor chip) in our design due to the diverse characteristics of the recorded signals.
- Based on the recording method signals can be recorded through multiple channels. The number of these channels can be very high, hundreds or thousands in the case of a transistor chip, and for this reason the user would like to decide which of them have to be visualized (and even processed).
- The user must be able to set a directory path and to choose file, since data are stored in physical files. Eventually, he/she can load many files and keep them in memory for later analysis or operation over the whole set of data.

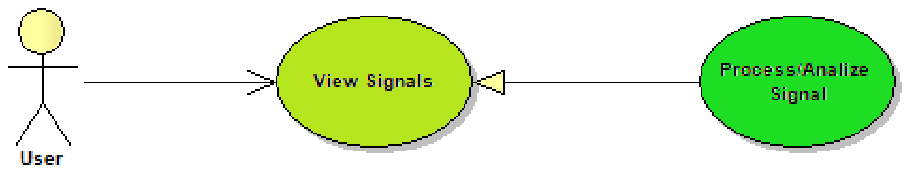


Fig. 4.1 "Data Display & Data Processing" Use Cases

4.1.1 USE CASE DESCRIPTION

Use Case: Data Display

Description: the user selects a recording to visualize

Pre-Conditions: the user must have set the type of the recordings, which channel he wants to see and specified a directory path where to find the files

Post-Conditions: the selected signal can be seen on the screen

Main Flow: select a recording type, choose channels of interest, set the directory path and finally select a specific signal file

Exception Flows: user could have made a wrong selection on channels, meaning no channel has been selected or the selected channel is not present in the signal file. If this situation occurs, a message is shown to the user to inform him/her that he/she must correct his/her settings before continuing.

Use Case: Process/Analyze Signal

Description: after having selected a signal to visualize, the user can perform an analysis or some process operation on data.

Pre-Conditions: same as Data Display

Post-Conditions: same as Data Display, with the addition of saving and/or displaying the result to the user

Main Flow: same as Data Display, and then select the preferred operation

Alternative Flow: before selecting the preferred operation, the user may choose to use a single file or multiple files

Exception Flow: same as Data Display.

4.1.2 DOMAIN MODEL

Now that we have a description of the use case we can analyze what kinds of objects compose our domain. For the moment we focus only on the main flow of the “Data Display” use case, and leave exceptions handling and “Process/Analyze Signal” for a later phase of our design.

As the purpose of the application is dealing with multichannel signals, we begin with defining two entities: *"Signal"* that is composed of *"Data Channels"*. Signals are saved into physical files, thus in some way recordings are loaded by the application from the hard drive into system memory – *"File Reader"* is responsible of this operation. We also need an object capable of displaying data to the user – this is the aim of *"Data Plotter"*. *"Recording Type"* is an enumeration of constants, while *"Data File"* represents an existing file.

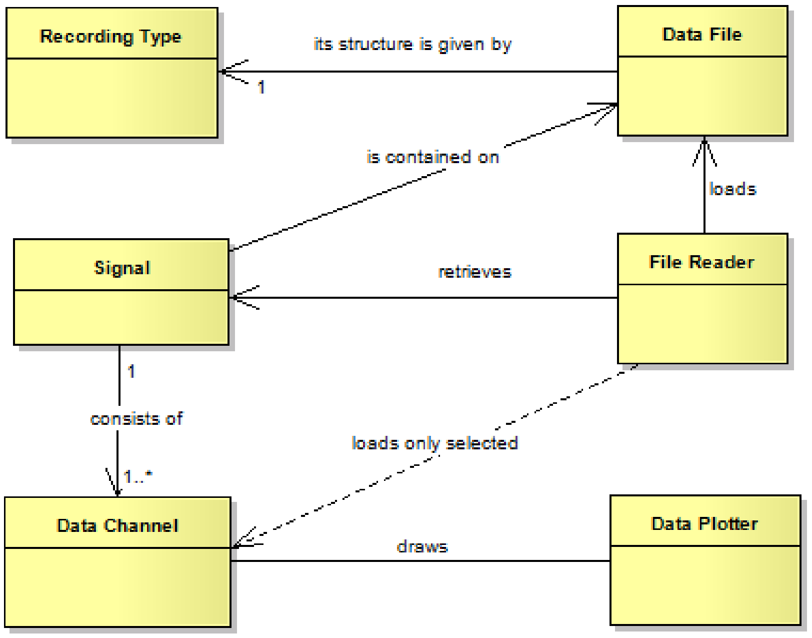


Fig. 4.2 Class Diagram of the Domain Model

4.2 DESIGN

4.2.1 COMMUNICATION DIAGRAM

The use case description helps us to define the way objects interact each other. As a matter of fact we realize from our previous analysis that the problem can be solved in three temporally subsequent steps, as explained in the main flow description of the use case. We added a boundary object UI (User Interface) to model the interaction between the user and the application. By now it is intended to be something that allows this exchange of information. We will focus on it in the next chapter (Figure 4.3).

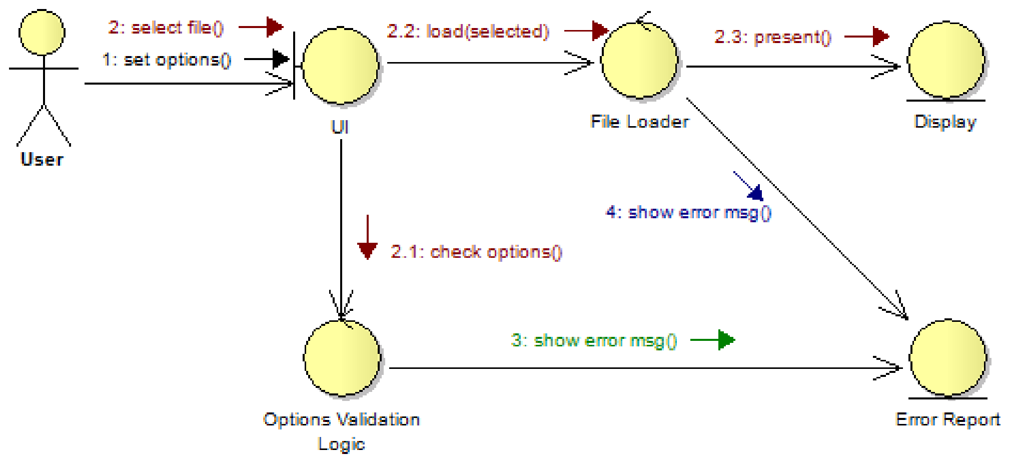


Fig. 4.3 Communication Diagram for Data Displaying

Paths 3 and 4 are exceptional flows when an error occurs in options validation or while loading a file.

The diagram in figure 4.4 extends the previous one in order to include also data processing/analysis logic.

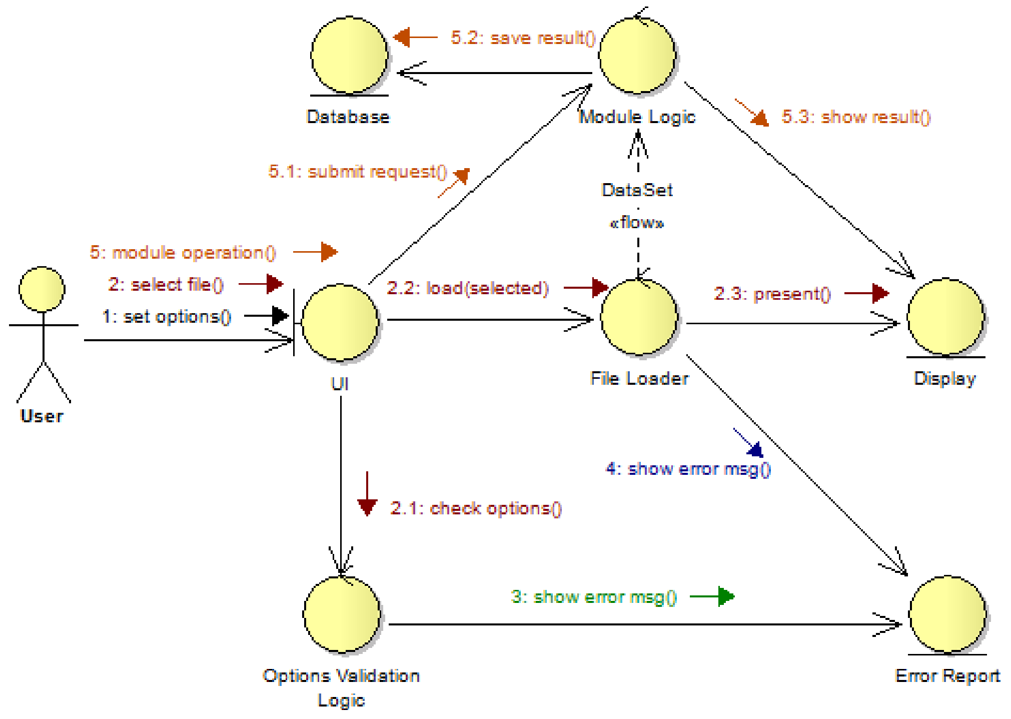


Fig 4.4 Communication Diagram including Operations on Signals

4.2.2 CLASS DIAGRAM

The chronological order of the operations that emerges from the *Communication Diagram* also underlines a logical subdivision of the problem, thus suggesting one possible solution about what classes and what objects we need to reach our target.

In chapter 4 we said that SigMate consists in many modules, each one responsible of one (or one set of similar) operation(s). For this reason we created an interface named *IModule.* Each concrete module that composes the whole application must implement this interface.

Through interfaces *IFileReader*, *IOptionSelector* and *IFileBrowser* the different application properties can be selected. The basic output, i.e. signal visualization, is responsibility of *IDataDisplay* interface. *IModule* is an aggregation of these interfaces all together.

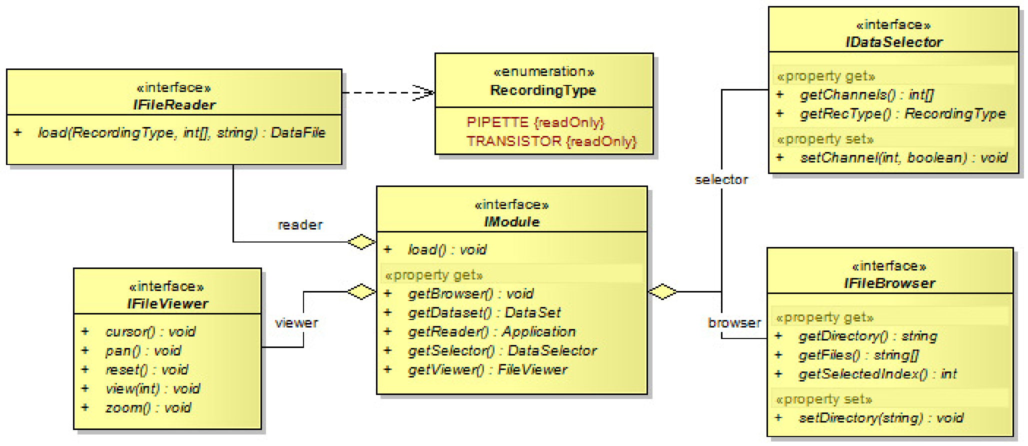


Fig. 4.5 Initial Class Diagram

We adopted the *Observer Pattern* to extend functionality of *IModule*. This way, a class implementing this interface, only has to care about task logic, leaving to external observers the duty of showing or saving results. This way we avoid heavy coupling between classes, since each *IModule* knows its observers only through their interfaces.

There are mainly two events that occur during the life cycle of a module:

- Data Selection
- Task Completed

Thus we created two interfaces that allow a module to be observed by other objects in order to respond properly at those events.

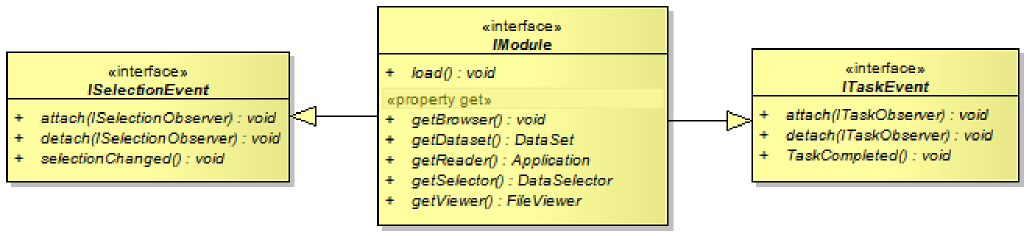

«interface»
ISelectionEvent
+ attach(ISelectionObserver) : void
+ detach(ISelectionObserver) : void
+ selectionChanged() : void
«interface»
IModule
+ load() : void
«property get»
+ getBrowser() : void
+ getDataset() : DataSet
+ getReader() : Application
+ getSelector() : DataSelector
+ getViewer() : FileViewer
«interface»
ITaskEvent
+ attach(ITaskObserver) : void
+ detach(ITaskObserver) : void
+ TaskCompleted() : void

Fig. 4.6 The Observable Interface of IModule

CHAPTER 5: INTERACTING WITH THE USER

User interface behaves like a bridge between the user and the application logic. Through it, commands given by users can be dispatched to the underlying classes that perform application tasks. Figure 5.1 has already illustrated this concept.

5.1 ANALYSIS

SigMate interacts with the user through a simple to understand graphical interface. Following, there is a brief description (Figure 5.1).

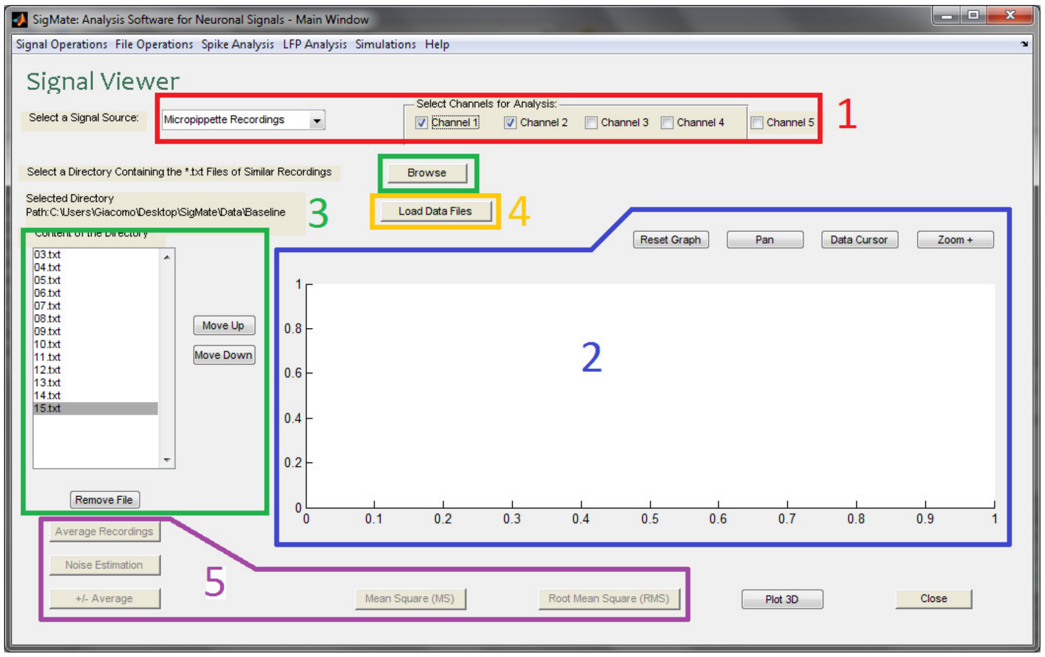


Fig. 5.1 SigMate Graphical User Interface of the first module (for signal visualization and performing basic operations)

1. Signal options panel: recording type and channels can be selected by the user;
2. Data display panel: when a file is chosen, it is displayed on this panel. The four buttons allow the user to zoom, to translate, and to show a cursor. One resets the graph to its default behavior;
3. File navigation panel: folder navigation and file selection can be done here. Two buttons allow moving a file inside the list. Another

button allows file removing;

4. A button to load all the files in the list into the memory, to perform tasks over the whole amount of data;
5. Module logic panel: this section varies from each module to the other. It contains buttons and other kinds of controls to perform operations depending on the module.

Use Cases for this problem emerges from points 1, 3, 4 and 5 of the previous list. Our domain model simply consists in four classes representing those features, plus one class that coordinates and manages the whole user interface.

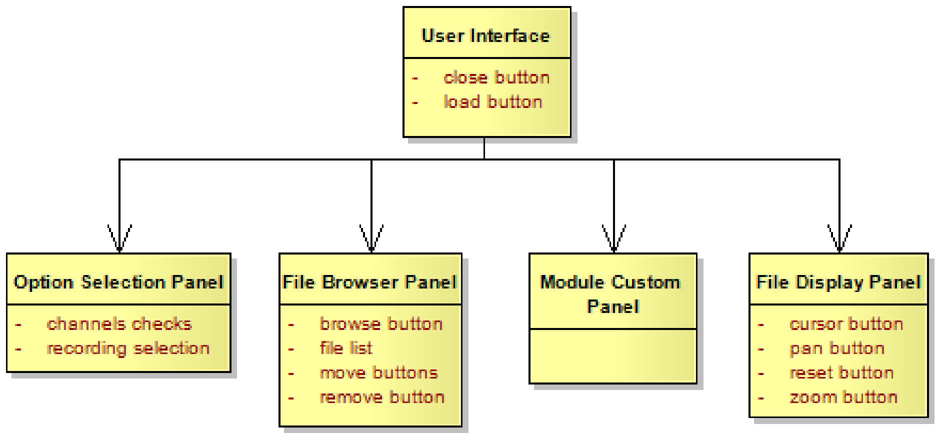


Fig. 5.2 Domain Model for User Interaction

5.2 DESIGN

5.2.1 CONSIDERATION ABOUT MATLAB GUI SYSTEM

Matlab provides programmers a nice tool, GUIDE, to create GUI (Graphical User Interface(s)), but it is not intended to be used in an object oriented programming logic. This is a strong limitation as other objects cannot see the resulting GUI as a whole, but rather as a collection of static functions.

We can think to adapt the various objects that compose our system to interact with a single figure, but this implies that for each different figure we must create a different version of the same objects. It would result in the creation of a lot of similar classes and would make both design and code hard to understand. More, if we wish to write this software in another programming language but Matlab, we would need to restructure the whole design, and many classes designed for a Matlab implementation will become useless.

Our solution consists in the creation of general interface classes, basing on our domain model. Implementations of these abstractions deal with the specific entities the platform we are writing these classes for uses to paint a GUI. This way, GUI logic is independent and its physical implementation results completely transparent to the user.

5.2.2 COMMUNICATION DIAGRAM

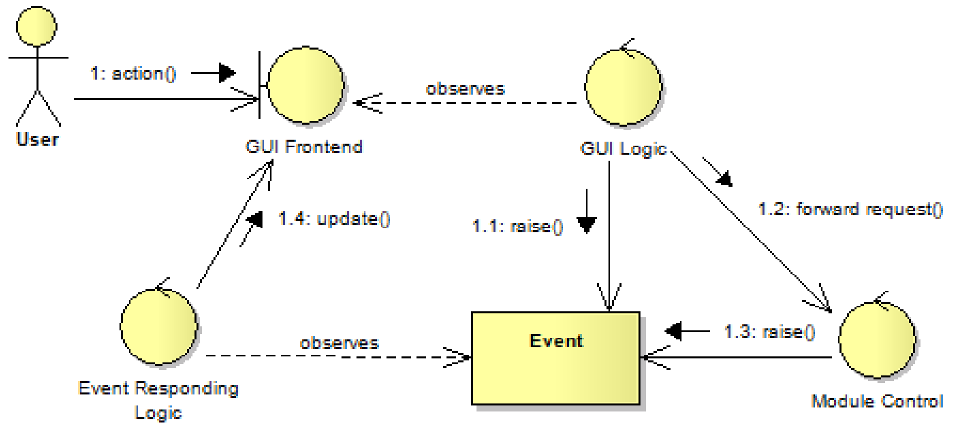


Fig. 5.3 Communication Diagram

GUI logic is realized following the observer pattern, where the observable object is the GUI front end. It is responsible of raising events or of forwarding requests towards the underlying module. It can also trigger events, for example, when a task is completed. Events are observed by a *Event Responding Logic*, which updates the GUI if necessary (for example disabling or enabling buttons or displaying results).

In the next figure (Figure 5.4) we underline how entities found in domain model are one to one mapped to our design. We notice that both *GUI Frontend* and *GUI Logic* are made up of five subsystems each.

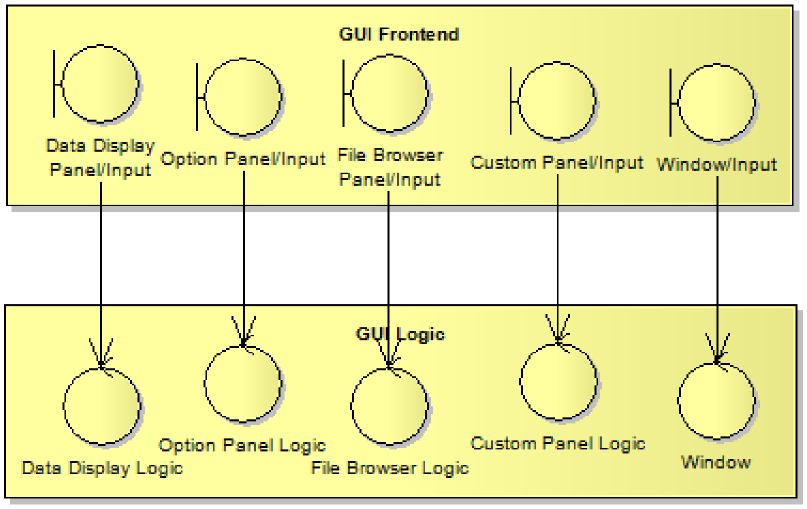


Fig. 5.4 GUI Figure and GUI Logic in Detail

Each one of the components of *GUI Logic* raises specific events and notifies messages to *Module Control*, forwarding user's requests.

5.2.3 CLASS DIAGRAM

Once we have planned both the *Domain* and *Communication Diagrams*, the class design for this part of the application descends naturally from them.

We have already defined the behavior of the logic in chapter 5, providing the interfaces *IFileBrowser*, *IOptionSelector*, *IDataDisplay* accessible from the *IModule* interface. Instead of creating classes for the GUI logic that eventually would interact with implementations of these interfaces, we preferred adopt the *Adapter Pattern* to solve this problem.

We first defined an *IUserPanel* interface, shown in the next figure:

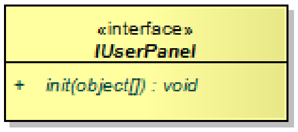


Fig. 5.5 IUserPanel interface

The only method *init(object[] controls) : void* is called when the panel has to be initialize. C*ontrols* variable contains references to the objects present in the window, like buttons, lists, check boxes etc.

In our MATLAB version, this method is called every time a figure is created, since every handle of the figure is destroyed when the figure is closed and recreated when it is opened. In a Windows Forms implementation, the method would be called only when the window is created, and not every time it is hidden and shown.

Each base panel of our application (numbers 1, 2 and 3) is managed by a class implementing both *IUserPanel* and one of the above cited interfaces (Figure 5.6). An *IModule* using these classes doesn't matter how they are realized, since it can access them only through their interface, as defined in the previous chapter.

Methods which name sounds like *on_something()* are handles to events triggered by objects of the GUI. Updating logic (changes of the enable state of a control, update of a list etc.) is specified in their bodies.

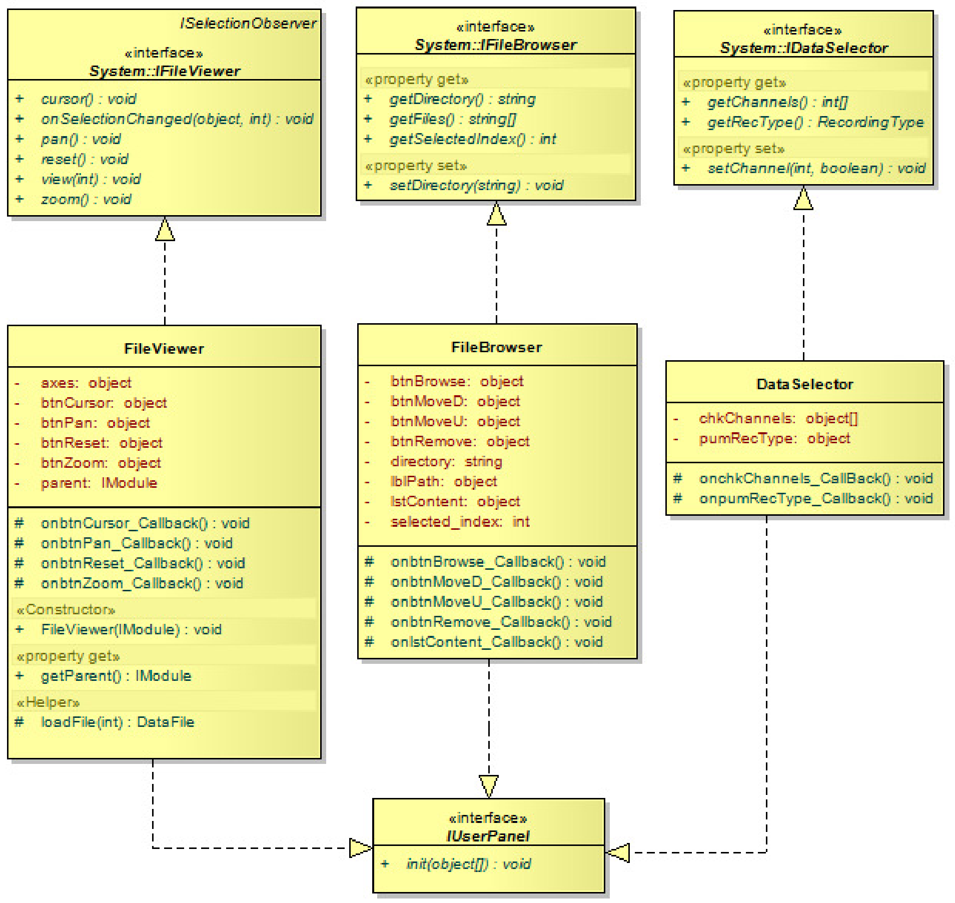


Fig. 5.6 Class Diagram for User Interaction

Figure 5.7 shows a base implementation of *IUserInterface*. It defines variables and behavior shared among all the GUIs of our program. Each graphical user interface must inherit from this abstract class.

In our logic each subclass maintains a reference to the particular module it is the boundary for and not vice versa, because modules are at a medium layer between the user interface and the file system and they are not intended to know how interact with the user. This way the final application deals only with GUI objects, being them responsible of the communication with the underlying layer.

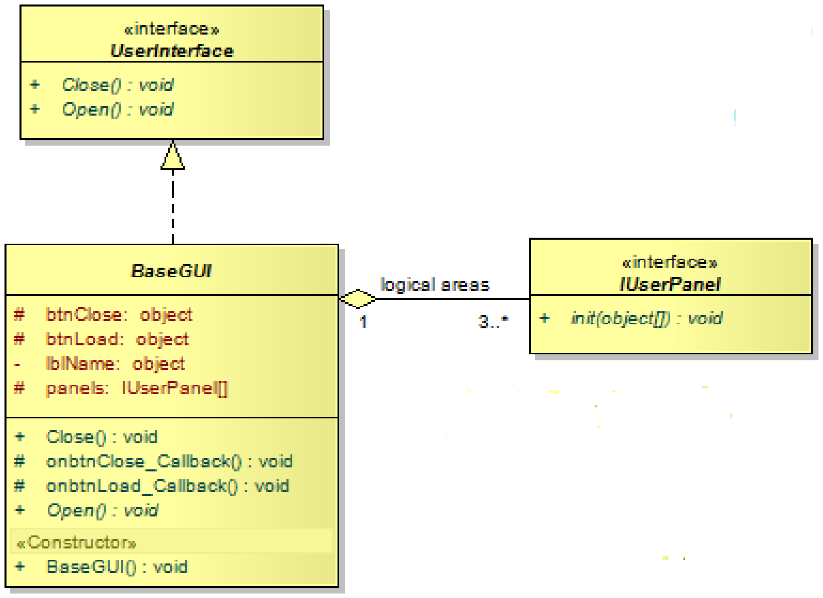


Fig. 5.7 Base GUI Class Diagram

5.3 CUSTOM CONTROL PANELS

We also included the design of some panels that often occurs in SigMate.

The first one is the *Statistic Panel*, which contains commands to execute statistical operations like averages, mean squares and noise estimation. It is primarily used in the main module of the program, but we can find it also when using LFP analyzer. Moreover, the underlying logic is shared among every module of the program. This is the reason because it is our first implementation as we will see in the case of study.

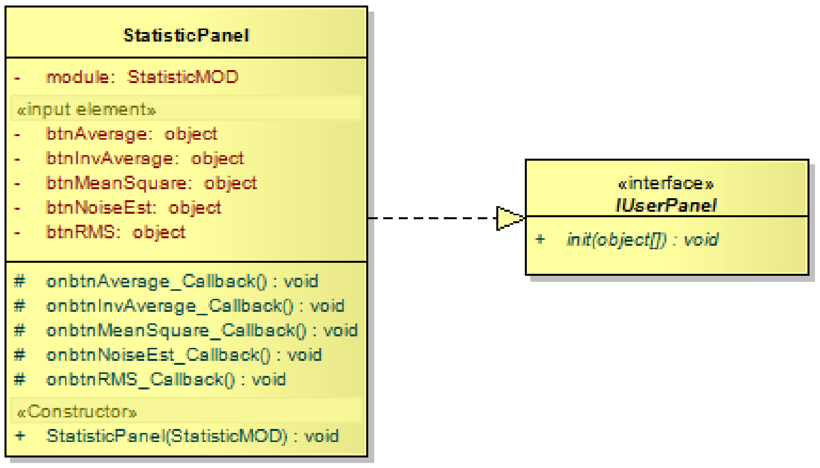


Fig. 5.8 Statistic Panel

CHAPTER 6: PUTTING IT ALL TOGETHER

SigMate consists of multiple modules, each one performing a specific set of operations. By now six of these modules already exist, but their number is going to grow in the next future. For this reason we need to design our software in order to be flexible enough to allow easy integration of new modules into the existing system. In this chapter there won't be the analysis step, since it only consists in one single *Domain Diagram*. The *Use Case Diagram* has already been explained in chapter 4.

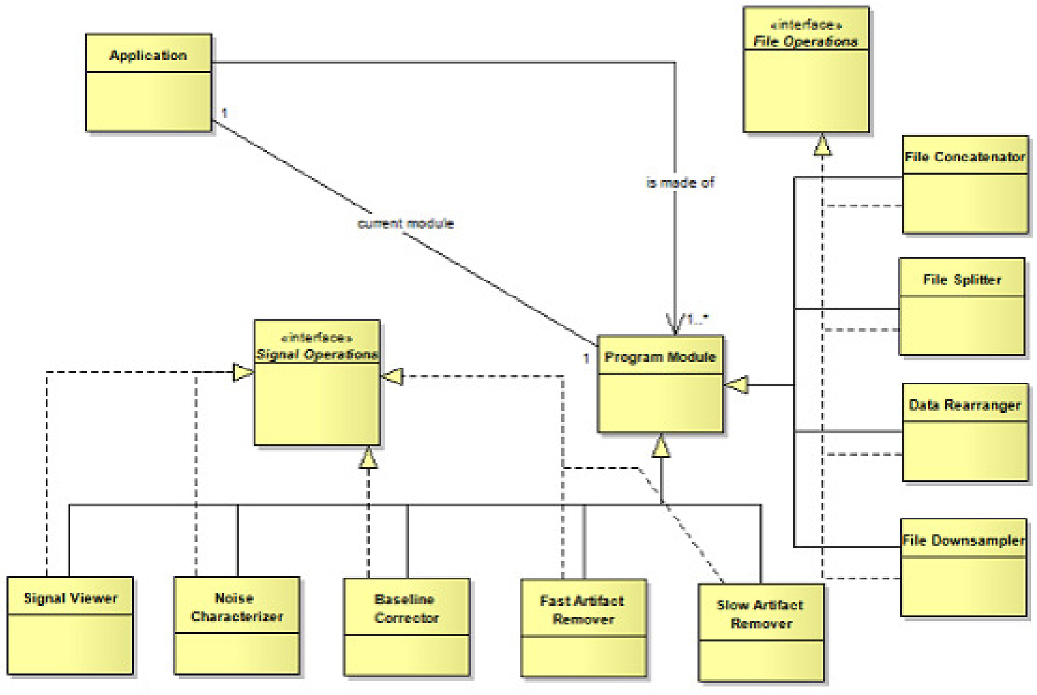


Fig. 6.1 Domain Model of the Application

From this domain model we can infer two interfaces, which are our starting point during the design phase.

As the application is nothing more than a set of features coded in different modules, we modeled it as a collection of plug-ins. From this interface we can register a plug-in or choose which one we want to be executed.

The second one is an interface we named, as the reader can expect, *IPlugin*. We kept it as simpler as possible, because our goal is to make SigMate a very flexible tool able to adapt to the many different needs of scientists working on the branch of neuronal signal analysis. It exposes three methods, two of which are simple access methods. The remaining one is responsible of the execution of the plug-in.

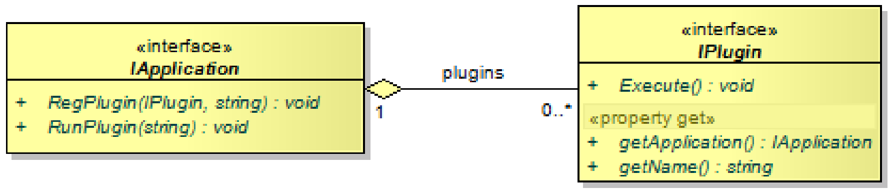


Fig. 6.2 IApplication and IPlugin Interfaces

6.1 COMMUNICATION DIAGRAM

In the last chapter we said that *the application can interact with logic only through user interfaces*. Now, since each plug-in is a sort of little program, we can say that *interaction with logic is reached only through user interface and the responsible of the good cooperation between these two entities in carried by a plug-in*.

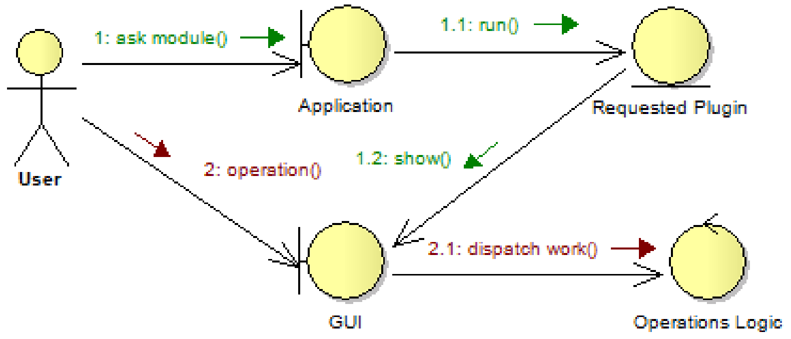


Fig. 6.3 Communication Diagram

The role of the application is managing the coordination of the various features, accepting user's input and giving him/her back the module requested.

In our MATLAB implementation we directly mapped these concepts into classes, making the application a collection of plug-ins.

This class exposes a method to load them in which the loading routine is hard coded. The drawback of this solution is that every time we add a new module we must update the code of our program. This is not a real problem since MATLAB is a script language and code is compiled for every execution of the program. If we wish port our application in another language, we would have to think of a different way to solve the problem.

Solutions are different if the module is released with a compiled library or with a script text file. This is not the place where discuss about this matter, but our suggestion is to maintain a file where names and locations of modules are registered. This solution requires an careful design of the plug-in loading engine, but once the code has been written and compiled no more modifications are needed.

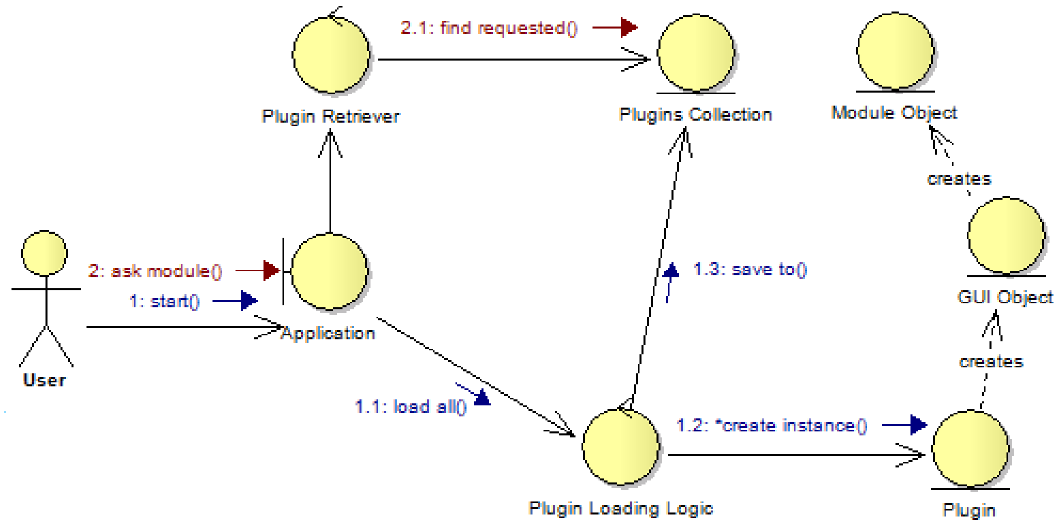


Fig. 6.4 Modules Loading Logic

6.2 DESIGN OF THE APPLICATION CLASS

There are many ways to realize the application class and we chose to adopt objects polymorphism common between almost the object oriented program languages existing at the time. This choice was due by the fact that the main window of the program is the only one having a menu bar for the navigation through the various features. Implementing both *IApplication* and *IUserInterface* in one class everything is strictly bounded inside an already existing object.

Although it may seem these behaviors are heavily coupled, the application logic is very simple to manage, and we think that this solution is simpler and makes the design more understandable, rather than the creation of lots of micro classes.

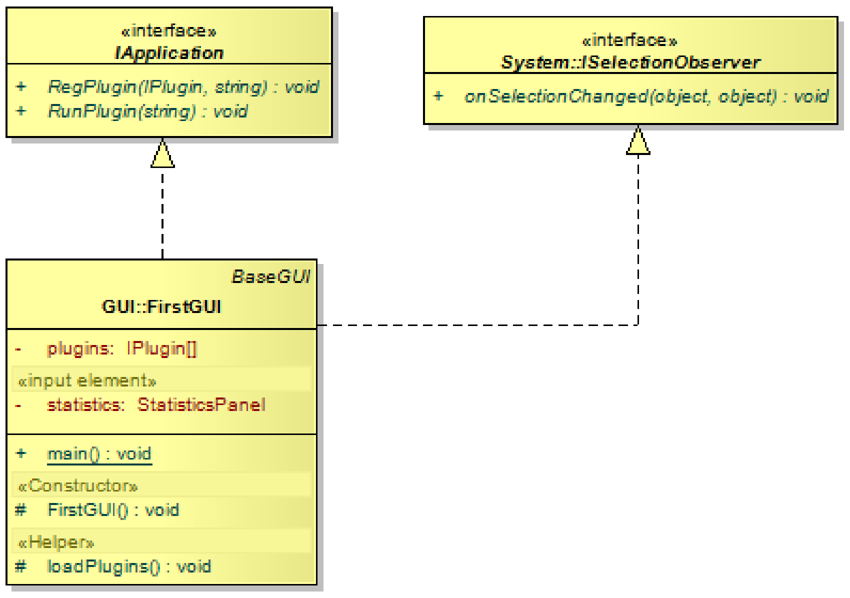


Fig. 6.5 Application Object Class Diagram

The selection of a specific module is communicated to the application by menu items callbacks present in the figure file. This is the only one case in which one figure stores a reference to its GUI class in order to notify changes in selections. As a matter of fact the creation of a graphical panel responsible of the management of the menu bar input would result hard to maintain because of the continuously growing number of SigMate's features. This approach implies the insertion of a new method in the interface of such object every time a new module is developed, making its interface variable over time (thus arising incompatibility problems with previous versions of the program).

With our solution, instead, we only makes changes on the main window of the program, in order to insert new menu items, and on the body of the *onSelectionChanged* method of the application object.

CHAPTER 7: CASE OF STUDY – IMPLEMENTATION OF THE MAIN MODULE

In this case of study we will show a matlab implementation of the main module of the application. Orange italic sentences represent parts of code we omitted because not essential to understand the working logic of our design. Each class is preceded by a short comment.

DataFile

This class is responsible of keeping in memory data just loaded from file and retrieves them when needed.

```
classdef DataFile < handle

    properties (Access = private)
        data %matrix containing data from file
        channels %channels description
        data_pts %number of samples
        sample_f %sample frequency
    end
    methods
        %PROPERTIES GETTERS
        function f = getSampleFreq(obj)
            f = obj.sample_f;
        end
        function N = getNumSamples(obj)
            N = obj.data_pts;
        end
        function d = getChannelsDesc(obj)
            d = obj.channels(2:length(obj.channels));
        end
        %ACCESS METHODS
        function time = getTime(obj)
            time = obj.data(:,1);
        end
        function data = getData(obj,varargin)
            ch = varargin{1};
            id = zeros(1,length(ch));
            for i = 1:length(ch)
                id(i) = find(obj.channels == ch(i));
            end
            data = obj.data(:,id);

            if length(varargin) == 3
                s = varargin{2};
                c = varargin{3};
                data = data(s:(s + c - 1),:);
            end
        end
        %CONSTRUCTOR
        function obj = DataFile(time,data,channels)
            obj.data_pts = size(time,1);
            obj.sample_f = (time(2) - time(1)) / obj.data_pts;
            obj.data = [time, data];
            obj.channels = [0 channels];
        end
    end
end
```

ISelectionEvent

We reported the code only for this interface, as *IProcessedEvents* behaves exactly in the same way.

```
classdef ISelectionEvent < handle

    properties (Access = protected)
        handlers %array of pointers to event handlers associated with this
object
    end
    methods
        function attach(obj,handler)
            obj.handlers = [obj.handlers(:), handler];
        end
        function detach(obj,handler)
            x = find(obj.handlers == {handler});
            if (x) %the specified handler observes this object
                for i = x:length(obj.handlers)-1
                    obj.handlers{i} = obj.handler{i+1};
                end
                obj.handlers = [obj.handlers(1:length(obj.handlers)-1)];
            end
        end
        function selectionChanged(obj, sender)
            for i = 1:length(obj.handlers)
                h = obj.handlers{i};
                h.onSelectionChanged(sender);
            end
        end
        %CONSTRUCTOR
        function obj = ISelectionEvent()
            obj.handlers = {};
        end
    end
end
```

StatisticMOD

This module performs various operations, like averaging or estimating noise on data signals. The code for these operations has been omitted, but the reader must keep in mind that when an operation is completed a

ProcessedDataEvent is triggered, thus the application can display results to the user.

Notice that the *DataSet* class appearing in *load* method is an extension of the *DataFile* class. It can perform operation of merging multiple data files and handles all of them in a single structure.

```
classdef StatisticMOD < IModule

    properties (Access = protected)
        eavg, oavg, avg,
        nest, rms, ms
    end
    %
    methods
        %CONSTRUCTOR
        function obj = StatisticMOD(selector, browser, viewer)
            obj = obj@IModule();
            obj.selector = selector;
            obj.browser = browser;
            obj.viewer = viewer;
            %
            obj.reader = FileReader();
        end
        %STATISTIC OPERATION
        function average(obj)
        function invertedAverage(obj)
        function estimateNoise(obj)
        function meanSquare(obj)
        function rootMeanSquare(obj)
        %LOAD
        function load(obj)
            files = obj.browser.files;
            ch = obj.selector.channels;
            dataset = DataSet(ch);
            for i = 1:length(files)
                f = files{i};
                M = obj.reader.load(obj.browser.directory, f, ch);
                dataset.vertcat(M);
            end
        end
    end
end
```

FileBrowser, DataSelector and FileViewer

These three classes are both GUI panels and implementations of the respective interfaces.

```
classdef FileBrowser < IFileBrowser & IUserPanel

    properties (Access = private)
        btnBrowse, btnRemove, btnMoveD, btnMoveU,
        lstContent, lblPath, path
    end
    properties (Dependent, Access = public)
        directory
    end
    properties (Dependent, SetAccess = private)
        files
        index
    end
    methods
        %IUSERPANEL
        function init(handles)
            % initialization routine initializes each private property with
            % a corresponding handler of one figure object
        end
        %IFILEBROWSER
        function files = get.files(obj)
            files = get(obj.lstContent, 'String');
        end
        function index = get.index(obj)
            index = get(obj.lstContent, 'Value');
        end
        function path = get.directory(obj)
            path = obj.path;
        end
        function set.directory(obj, value)
            obj.path = value;
            text = ['Selected Directory Path: ', value];
            set(obj.lblPath,'String', text);

            newpath = strcat(regexprep(obj.path,'\','/'),'/*.txt');
            dir_struct = dir(newpath);
            file_names = sortrows({dir_struct.name}');
            set(obj.lstContent, 'String', file_names);
            set(obj.lstContent, 'Value', 1);

            % GUI objects update

        end
        function remove(obj)
            selected = obj.index;
            prev_str = get(obj.lstContent, 'String');
            if ~isempty(prev_str)
                prev_str(selected) = [];
                set(obj.lstContent, 'String', prev_str);
                set(obj.lstContent, 'Value',
min(selected,length(prev_str)));
            end

            % GUI objects update

        end
        function moveU(obj)
            if (~isempty(obj.files) && (obj.index > 1)&&(obj.index <=
length(obj.files)))
                listbox_contents = obj.files;
                temp = listbox_contents{obj.index};
                listbox_contents{obj.index} = listbox_contents{obj.index-1};
                listbox_contents{obj.index - 1} = temp;
                set(obj.lstContent, 'String', listbox_contents);
                set(obj.lstContent, 'Value', min(index -
1,length(listbox_contents)));
            end

            % GUI objects update

        end
        function moveD(obj)
            if (~isempty(obj.files) && (obj.index >= 1)&&(obj.index <
length(obj.files)))
                listbox_contents = obj.files;
                temp = listbox_contents{obj.index};
                listbox_contents{obj.index} = listbox_contents{obj.index +
1};
                listbox_contents{index + 1} = temp;
                set(obj.lstContent, 'String', listbox_contents);
                set(obj.lstContent,'Value', min(index +
1,length(listbox_contents)));
            end

            % GUI objects update

        end
    end
end

classdef DataSelector < IDataSelector & IUserPanel

    properties (Access = private)
        pumRecType
        chkChannels
        my_module
    end
    properties (Dependent, SetAccess = private)
        channels
        rec_type
    end
    methods
        %IUSERPANEL
        function init(obj, handles)
            obj.pumRecType = handles{1};
            obj.chkChannels = handles{2:length(handles)};
        end
        %IDATASELECTOR
        function channels = get.channels(obj)
            channels = [];
            for i = 1:length(obj.chkChannels)
                ch = obj.chkChannels{i};
                if (get(ch, 'Value') == get(ch, 'Max'))
                    channels = [channels, i];
                end
            end
        end
        function rec_type = get.rec_type(obj)
            t = get(obj.pumRecType, 'Value');
            switch (t)
                case 2
                    rec_type = 'micropipette';
                case 3
                    rec_type = 'transistor';
            end
        end
        %HELPER
        function enable(obj, flag)
            if flag
                state = 'On';
            else
                state = 'Off';
            end
            for i = 1:length(obj.chkChannels)
                ch = obj.chkChannels{i};
                set(ch, 'Visible', state);
            end
        end
    end
end

classdef FileViewer < IUserPanel & IFileViewer

    properties (Access = private)
        axes, btnCur, btnPan
        btnReset, btnZoom
    end
    properties (Access = public)
        my_module
    end
    methods
        %IUSERPANEL
        function init(obj,varargin)
            % initialization routine initializes each private property with
            % a corresponding handler of one figure object
        end
        %IFILEVIEWER
        function cursor(obj)
            button_state = get(obj.btnCur,'Value');
            if button_state == get(obj.btnCur,'Max')
                % Toggle button is pressed, take appropriate action
                datacursormode on;

                 % GUI objects update

            elseif button_state == get(obj.btnCur,'Min')
                % Toggle button is not pressed, take appropriate action
                datacursormode off;
            end
        end
        function pan(obj)
            button_state = get(obj.btnPan,'Value');
            if button_state == get(obj.btnPan,'Max')
                % Toggle button is pressed, take appropriate action
                pan on;

                 % GUI objects update

            elseif button_state == get(obj.btnPan,'Min')
                % Toggle button is not pressed, take appropriate action
                pan off;
            end
        end
        function reset(obj)
```

```
             % GUI objects update

             datacursormode off;
            zoom off;
            pan off;
            axis('tight');
        end
        function zoom(obj)
            button_state = get(obj.btnZoom,'Value');
            if button_state == get(obj.btnZoom,'Max')
                % Toggle button is pressed, take appropriate action
                zoom on;

                % GUI objects update

            elseif button_state == get(obj.btnZoom,'Min')
                % Toggle button is not pressed, take appropriate action
                zoom off;
            end
        end
        function view(obj,file_name)
            % load the file
            % display data on the axes panel
        end
        function onSelectionChanged(obj, sender)
            if (isa(sender,'IBrowser'))
                index = sender.index;
                name = sender.files{index};
                path = sender.directory;
                file_name = [path, name];
                obj.view(file_name);
                obj.view(data);
            end
        end
    end
end
```

MainGUI and IApplication

IApplication is implemented as dictionary storing key-value pairs, where keys are plugins' names and values are plugins themselves.

```
classdef IApplication < handle

    properties
        % dictionary containing all plugins.
        plugins
    end
    methods
        function regPlugin(plugin, name)
            % register plugins in a dictionary
        end
        function runPlugin(name)
            % find the selected plugin and run it
        end
    end
end
```

This is the main class of the application, being it the first GUI displayed to the user. The application starts from the static method *main*. A new instance of the application is created, then the program looks for the existent plugins and load them. Finally, the main window is displayed to the user.

```
classdef MainGUI < IApplication & GUI.BaseGUI

    properties (SetAccess = private)
        my_module % the specific module associated with this GUI
    end
    methods
        %CONSTRUCTOR
        function obj = MainGUI()
            obj@IApplication();
            obj@GUI.BaseGUI();
            % panels creation
            selector = GUI.DataSelector();
            browser = GUI.FileBrowser();
            viewer = GUI.FileViewer();
            obj.panels = {selector, browser, viewer};
            % module creation
            obj.my_module = StatisticMOD(selector, browser, viewer);
        end
        %IUSERINTERFACE
        function open(obj)
            hfig = MainFIG();
            handles = guidata(hfig);
            %create references to the GUI manager and the module
            handles.my_GUI = obj;
            handles.my_module = obj.my_module;
            %store handles in the figure
            guidata(hfig, handles);
            %
            % initialize every panel with its object handles
            %
            obj.hfig = hfig;
        end
        function close(obj)
            selection = questdlg('Do you really want to exit?',...
                     'Exit Request Confirmation',...
                     'Yes','No','Yes');
             switch selection,
                 case 'Yes',
                     delete(obj.hfig)
                 case 'No'
                     return
             end
        end
    end
    methods (Static)
        function main()
            app = FirstGUI();

            % load all plugins in the application

            app.open();
        end
    end
end
```

As the user can have notice, many of our interfaces contains themselves some implementation code. This is due to MATLAB, since it does not have a specific construct to represent pure abstract interfaces. They are simply abstract classes. For this reason, instead of create lots of interface-base class pairs, we preferred to fuse these two entities in a single object.

We moved GUI objects' callbacks out from classes contained in the GUI package. For the sake of simplicity we created them with the GUIDE programming tool, which automatically sets objects' callbacks to static methods contained in the figure's corresponding m-file, and, moreover callback methods themselves cannot be instance methods but only static ones.

By storing a reference to the module in the figure *handles* object, the figure itself can dispatch methods directly to the underlying module with a single line of code, avoiding this problem at all.

Figure File

```
function varargout = MainFIG(varargin)

gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @FirstGUI_OpeningFcn, ...
                   'gui_OutputFcn',  @FirstGUI_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin && ischar(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
%%

%
% code generated by GUIDE
%
function FirstGUI_OpeningFcn(hObject, eventdata, handles, varargin)
function varargout = FirstGUI_OutputFcn(hObject, eventdata, handles)

% --- Executes on button press in btnClose.
function btnClose_Callback(hObject, eventdata, handles)

    handles.my_GUI.close();
%%

% --- Executes on button press in btnBrowse.
function btnBrowse_Callback(hObject, eventdata, handles)
    path = uigetdir('D:\MatlabWorks','Select a Directory');
    if(path)
        mod = handles.my_module;
        mod.browser.directory = strcat(path,'\');
    else
        msgbox('Please Select a Directory for Analysis', 'Directory Not
Selected','Error');
    end
%%

function pumRecType_Callback(hObject, eventdata, handles)
    mod = handles.data_selector;
    t = get(hObject, 'Value');
    switch t
        case 2
            mod.selector.enable(1);
        case 3
            mod.selector.enable(1);
        otherwise
            msgbox('Please Select a Signal Source', 'Signal Source Not
Selected','Error');
            mod.selector.enable(0);
        end
%%

% --- Executes on selection change in lstboxContent.
function lstboxContent_Callback(hObject, eventdata, handles)
    mod = handles.my_module;
    mod.selectionChanged(mod.browser);
%%

function btnLoad_Callback(hObject, eventdata, handles)
    handles.my_module.load();
%%

% --- Executes on button press in btnAverage.
function btnAverage_Callback(hObject, eventdata, handles)
    handles.my_module.average();
%%

% --- Executes on button press in btnNoiseEstimation.
function btnNoiseEstimation_Callback(hObject, eventdata, handles)
    handles.my_module.estimateNoise();
%%

% --- Executes on button press in btnInvertedAverage.
function btnInvertedAverage_Callback(hObject, eventdata, handles)
    handles.my_module.invertedAverage();
%%

% Same thing for all the other callbacks

function btnMeanSquare_Callback(hObject, eventdata, handles)
function btnRootMeanSquare_Callback(hObject, eventdata, handles)
function btnRemoveFile_Callback(hObject, eventdata, handles)
function btnMoveUp_Callback(hObject, eventdata, handles)
function btnMoveDown_Callback(hObject, eventdata, handles)
function tbZoom_Callback(hObject, eventdata, handles)
function tbDataCursor_Callback(hObject, eventdata, handles)
function btnResetGraph_Callback(hObject, eventdata, handles)
function tbPan_Callback(hObject, eventdata, handles)


function sigPlotter_Callback(hObject, eventdata, handles)
function plot3D_Callback(hObject, eventdata, handles)
function fileOperators_Callback(hObject, eventdata, handles)

% Menu callbacks call IApplication.runPlugin(plugin_name) on the object
handles.my_module, passing as plugin_name the name of the plugin
corresponding to the specified menu item

function lfpCharacterization_Callback(hObject, eventdata, handles)
function estimateLatency_Callback(hObject, eventdata, handles)
function characterizeNoiseMenu_Callback(hObject, eventdata, handles)
function manualMenu_Callback(hObject, eventdata, handles)
function aboutMenu_Callback(hObject, eventdata, handles)
function creditsMenu_Callback(hObject, eventdata, handles)
function menuSpike_Callback(hObject, eventdata, handles)
function spikeDetectionAndSorting_Callback(hObject, eventdata, handles)
function csdAnalysis_Callback(hObject, eventdata, handles)
function lfpmnuShapeCharacterization_Callback(hObject, eventdata, handles)
function fastArtifactRemover_Callback(hObject, eventdata, handles)
function slowArtifactRemover_Callback(hObject, eventdata, handles)
```

7.1 CONCLUSIONS AND FUTURE DIRECTIONS

From the case of study, this design has proved to be implementable. We choose MATLAB for our implementations because the existing code is written in this language, since it offer a simple way even to code complex algorithms. Scientists and researchers find this language friendlier than other just because it allows them to focus on the mathematical solution of a problem rather than on other aspects typical of programming languages.

However, we think we will port SigMate on another platform in the future. Our ideal target is C#, since this language combines software speed with a well documented library provided with the development environment. As a matter of facts speed is very important in such applications, since they have to deal with large amount of data and computations take often a lot of time.

Consequently, we are also planning to develop a math library, designed with particular care for our purposes, able to exploit parallel computation capabilities of modern hardware. Thus, as computational times would reduce of several magnitude orders, great advantages would be possible for analyzing and understanding recordings.

REFERENCES

[1] E. Gamma, R. Helm, R. Johnson, J. M. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," *Addison-Wesley, New Jersey, 1995.*

[2] M. Mahmud, A. Bertoldo, S. Girardi, M. Maschietto, S. Vassanelli "SigMate: A MATLAB-based Neuronal Signal Processing Tool," in: *Proceedings of the 32nd Annual International Conference of the IEEE Engineering in Medicine and Biology Society (IEEE EMBC2010)*, Buenos Aires, Argentina, September 2010, pp. 1352-1355.

[3] L. Hazan, M. Zugaro, G. Buzsáki, "Klusters, NeuroScope, NDManager: A free software suite for neurophysiological data processing and visualization", *Journal of Neuroscience Methods,* 155, 2006, pp. 207–216.

[4] K. Y. Kwon, S. Eldawlatly, K. G. Oweiss, "NeuroQuest: A Comprehensive Tool for Large Scale Neural Data Processing and Analysis", *Proceedings of the 4th International SaD1.18 IEEE EMBS Conference on Neural Engineering*, Antalya, Turkey, April 29 - May 2, 2009, pp. 622-625.

[5] M. Lidierth, "sigTOOL: A MATLAB-based environment for sharing laboratory-developed software to analyze biological signals", *Journal of Neuroscience Methods*, 178, 2009, 188–196.

[6] ________, "UML Applied – Object Oriented Analysis and Design using the UML," Training Material, ariadnetraining.co.uk (www.ariadnetraining.co.uk)