

Facultés Universitaires Notre Dame De la Paix (FUNDP Namur)  
Faculté d'Informatique

École Polytechnique de Montréal  
Département de Génie Logiciel

Team Leader: Yann-Gaël GUÉHÉNEUC



## TAUPE 2.0 - Developer's Guide

---

Benoît DE SMET  
Lorent LEMPEREUR

February 12, 2011

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Good to Know</b>	<b>4</b>
2.1	About this Document . . . . .	4
2.2	System Requirements . . . . .	4
2.3	Version . . . . .	4
2.4	Javadoc . . . . .	5
2.5	Directories . . . . .	5
2.6	Choices . . . . .	6
<b>3</b>	<b>Program Design</b>	<b>7</b>
3.1	Packages . . . . .	7
3.2	Data . . . . .	8
3.3	The Commands . . . . .	10
3.4	Preferences . . . . .	12
3.5	Groups . . . . .	14
3.6	Computation . . . . .	16
3.7	Parsers . . . . .	18
3.8	The graphical visualisation system . . . . .	20
3.9	The “AOI Maker” module . . . . .	22
<b>4</b>	<b>Maintainability</b>	<b>24</b>
4.1	How to Modify a Set of Parameters . . . . .	24
4.2	How to Add a New <i>command</i> . . . . .	24
4.3	How to Add a New Computation . . . . .	24
4.4	How to Add a New Group of Subjects . . . . .	26
4.5	How to Add a New Parser . . . . .	28
4.6	How to Add a Printer . . . . .	29
4.7	How to Add an Element to the Preferences . . . . .	29
4.8	How to Add an Option for the <i>Graphical Visualisation</i> . . . . .	30
<b>5</b>	<b>Compiling</b>	<b>32</b>
<b>6</b>	<b>Test Cases</b>	<b>33</b>
<b>7</b>	<b>Licence</b>	<b>34</b>
7.1	GNU GPL . . . . .	34
7.2	Icons . . . . .	34
<b>8</b>	<b>Contacts</b>	<b>36</b>
	<b>Index</b>	<b>37</b>
	<b>References</b>	<b>38</b>

## List of Figures

1	Directories . . . . .	6
2	Package diagram . . . . .	7
3	Class diagram - Package <code>laigle.taupe.viewer.data</code> . . . . .	9
4	Class diagram - Package <code>laigle.taupe.viewer.command</code> . . . . .	10
5	Class diagram - Package <code>laigle.taupe.viewer.configuration</code> . . . . .	12
6	Class diagram - Package <code>laigle.taupe.viewer.utils.group</code> . . . . .	14
7	Class diagram - Package <code>laigle.taupe.viewer.computation</code> . . . . .	17
8	Class diagram - Package <code>laigle.taupe.viewer.parsers</code> . . . . .	19
9	Class diagram - Package <code>laigle.taupe.viewer.graphics</code> . . . . .	21
10	Class diagram - Package <code>laigle.taupe.viewer.aoimaker</code> . . . . .	23

## 1 Introduction

This document aims to help developers to understand and modify the TAUPE 's source code.

The first section gives a set of preliminary tips for the developers.

The following section, using class diagrams and package diagrams, explains how the software is designed and which design have been taken.

Section 4 explains how add or modify some features into the software.

Section 5 describes how to compile the whole software.

TAUPE contains a set of test cases, as explained in Section 6. TAUPE follows the terms of a licence mentionned in Section 7.

Section 8 gives some links to contact the team that developed TAUPE .

## 2 Good to Know

### 2.1 About this Document

The source of this document (written in L<sup>A</sup>T<sub>E</sub>X) are available at <http://www.ptidej.net/research/taupe/>. Please maintain this document while you modify the software.

### 2.2 System Requirements

#### Java

The software is written in Java<sup>1</sup> using the *Java SE 1.6*.

#### Libraries

The following libraries are needed to develop the current source code:

- **jdom 1.1:** Used to parse the XML files from the *Eye-link II* system<sup>2</sup>.
- **junit 4.8.3:** Used for the test cases.

### 2.3 Version

The whole software was created at the *Ecole Polytechnique de Montreal*<sup>3</sup> in the *Ptidej Team*<sup>4</sup>.

0.0 The first version consists of the raw data of the software (fixations and saccades) and a graphical user interface to visualise the data from Eye-link®II. It was written by Yann-Gaël GUÉHÉNEUC. [1]

1.0 The version of the software written by several students in internship (mainly by Bertrand VAN DEN PLAS) at the *Ptidej Lab*. The version 1.0 allowed the user to apply some algorithms to some data.

---

<sup>1</sup><http://www.oracle.com/technetwork/java/javase/downloads/index.html>

<sup>2</sup>[http://www.sr-research.com/EL\\_II.html](http://www.sr-research.com/EL_II.html)

<sup>3</sup><http://www.polymtl.ca>

<sup>4</sup><http://www.ptidej.net>

2.0 During Fall 2010, the whole software was rewritten by Benoit DE SMET and Lorent LEMPEREUR. As explained below, this last version is able to parse the files generated by different kinds of eye-tracking systems and contains many new features. It is the current version of the software. The development of this new version lasted a couple of months under the supervision of Yann-Gaël GUÉHÉNEUC.

## 2.4 Javadoc

We ask to the developers who modify the source code to maintain the whole *Javadoc*. A particular syntax is used:

- Most of the methods uses preconditions about the input parameters. The preconditions are expressed by a comment after the parameter's description. For example, if a developer wants to express that a parameter should be strictly positive:

```
1  /**
2  * Setter
3  * @param price the new price; price > 0
4  */
5  void setPrice(int price) {
6      if (price <= 0) {
7          throw new IllegalArgumentException("The price must be
8              strictly positive.");
9      }
10     this.price = price;
11 }
```

- Each class must express its author (with the `@author` annotation) and its version (`@since`) as described in Section 2.3.

## 2.5 Directories

The project directory is currently organised following the structure shown in FIGURE 1:

- `doc` the generated javadoc, some `README` files concerning the usage of the software including this document.
- `lib` all the libraries
- `src` the source files
  - `icons` the set of icons used in the software (images)
  - `laigle` (*Laboratoire de Génie Logiciel Expérimental* or *Experimental Software Engineering Laboratory*) the main package
    - \* `taupe` the package that contains the application's code
    - \* `tests` the classes which define the unit tests
- `rsc` some ressources like `edf2xml` or some examples...
  - `tests` the input files for the unit tests

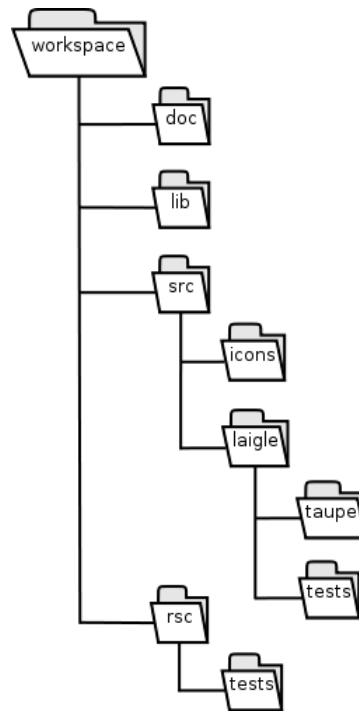


Figure 1: Directories

## 2.6 Choices

- All the design patterns are based on the definitions from [2].
- As illustrated in Section 2.4, the preconditions are checked using the `IllegalArgumentException` class. Actually, this exception is more often used for the method that have a `public` or `package` visibility according to [3]. For `private` methods, the developers may use an assertion (`assert`).
- The identifier of each variable always uses the Camel Case convention and never the character “\_”. For example, “max value” is written `maxValue`, not `max_value`.
- The choice of *Java SE 1.6* pertains to the possibility of adding the `@Override` annotation to the methods implemented from interfaces. It also allows the usage of the Nimbus look and feel<sup>5</sup>.
- Some classes are loaded with the *Java*’s reflection. TAUPE must know the content of some packages to load according to two situations: TAUPE is in development (for example, a developer works in *Eclipse* or *Netbeans* with the `.java` files) or TAUPE is a JAR file. When the software is in development, TAUPE lists some packages using the file system. When TAUPE is a JAR file, it uses the classes `java.util.jar.JarFile` and `java.util.jar.JarEntry` to browse the all JAR file. For this reason, the JAR file cannot be renamed. Although TAUPE was renamed, the JAR file’s name used would be the nearest of `TAUPE.jar` thanks to the *edit distance* algorithm from *Levenshtein*.

<sup>5</sup><https://nimbus.dev.java.net/>

## 3 Program Design

### 3.1 Packages

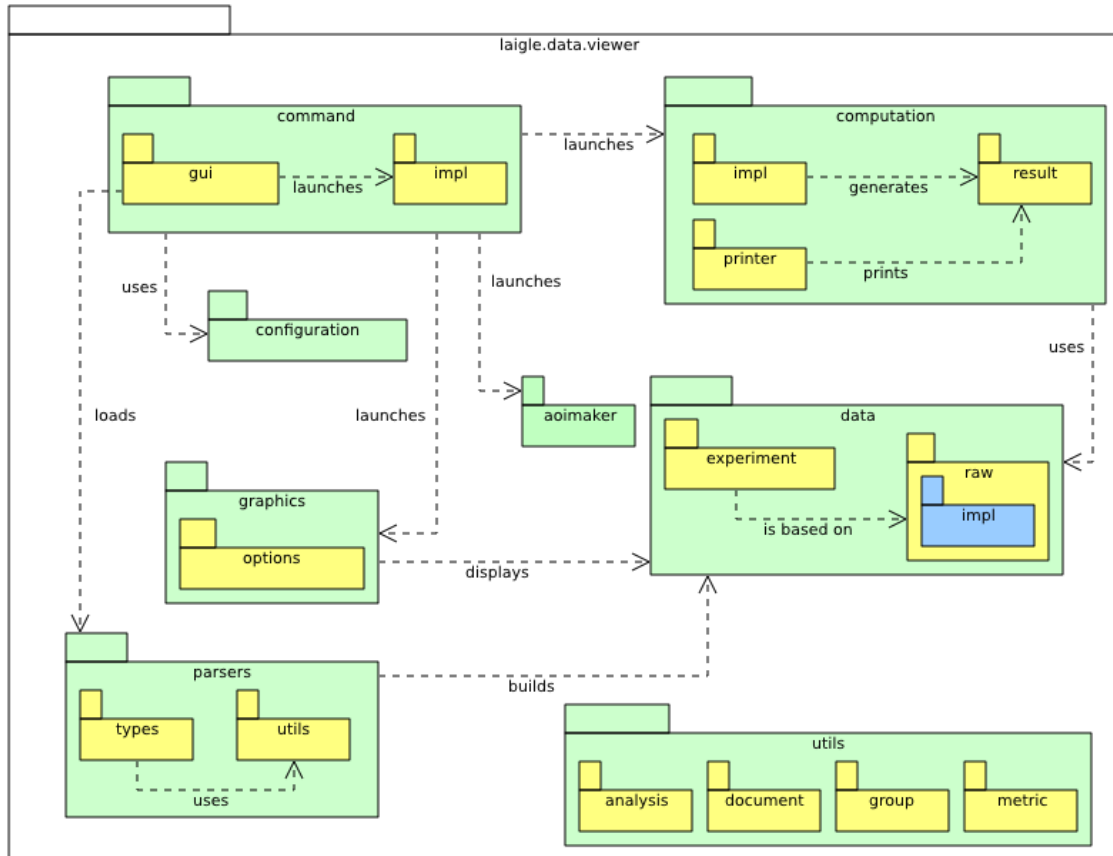


Figure 2: Package diagram

FIGURE 2 shows the project's structure. The next sections describe more precisely the content of the packages.

## 3.2 Data

This section describes the package `laigle.taupe.viewer.data` (see [FIGURE 3](#)).

### Description

- **fixation**: A fixation is a position of the eye during a gaze.
- **saccade**: A saccade is a movement of the eye between two fixations.
- **experiment**: The experiment represents the whole system we want to analyse with the software. Actually, it is a set of questions and their answers. When the user is using the software, this one handles up to only one experiment at a time.
- **question**: A question (a `Question`'s instance) is related to one image which defines the whole question.
- **subject**: A subject (a `SubjectData`'s instance) is a person which answered to a set of questions during the time of the experiment. They are defined by a set of characteristics like their name or their level of study. Each subject is linked to a file which is generally generated by an eye-tracking system and that contains the whole set of data about the subject's answers. If the file is called `Subject01.xml`, then it maybe exists a file named `Subject01.subject` which describes the subject's characteristics and whether their answers are wrong or not.
- **area of interest**: An area of interest (a `AreaOfInterest`'s instance) is an area on a question's image. This area can be relevant or not. It also can be "ignorable", it means that the system will not take account of this area. Those areas are defined in a text file for each question. For example, if a question is related to an image called `foo.png`, then the file which defines the areas is named `foo.aoi`.
- **answer**: An answer is what a specified subject has answered to a specified question. This answer can be correct or not (determined by the experiment's supervisor) and is related to a set of fixations and a set of saccades that the subject did during the time of the experiment. All the answers for a subject are in the same file.
- **scene**: This concept represents the link between a question, its areas of interest, and the fixations of each answers for this question.

### Choices

The software handles one and only one experiment at a time, therefore the `Experiment`'s instance follows the *singleton* pattern.

The *observer* design pattern is used with the `Experiment`'s singleton to notify the whole software about this singleton's changes.

The software uses an `Integer` to represents the relevance of an area of interest because we assumed that other developers would be interested by using a `weight` with the concept of areas of interest. The `Experiment`'s attribute named `minDuration` is used to define the minimal duration (in milliseconds) of a fixation. Some eye-tracking systems (like *GazeTracker<sup>TM</sup>*, for example) sometimes allow to choose this duration. So, TAUPE is able to set this duration too. Therefore, it is easier for the user to make different analysis based on a same set of files. Developers manipulates fixations and saccades with the methods declared respectively in `IFixation` and `ISaccade`.

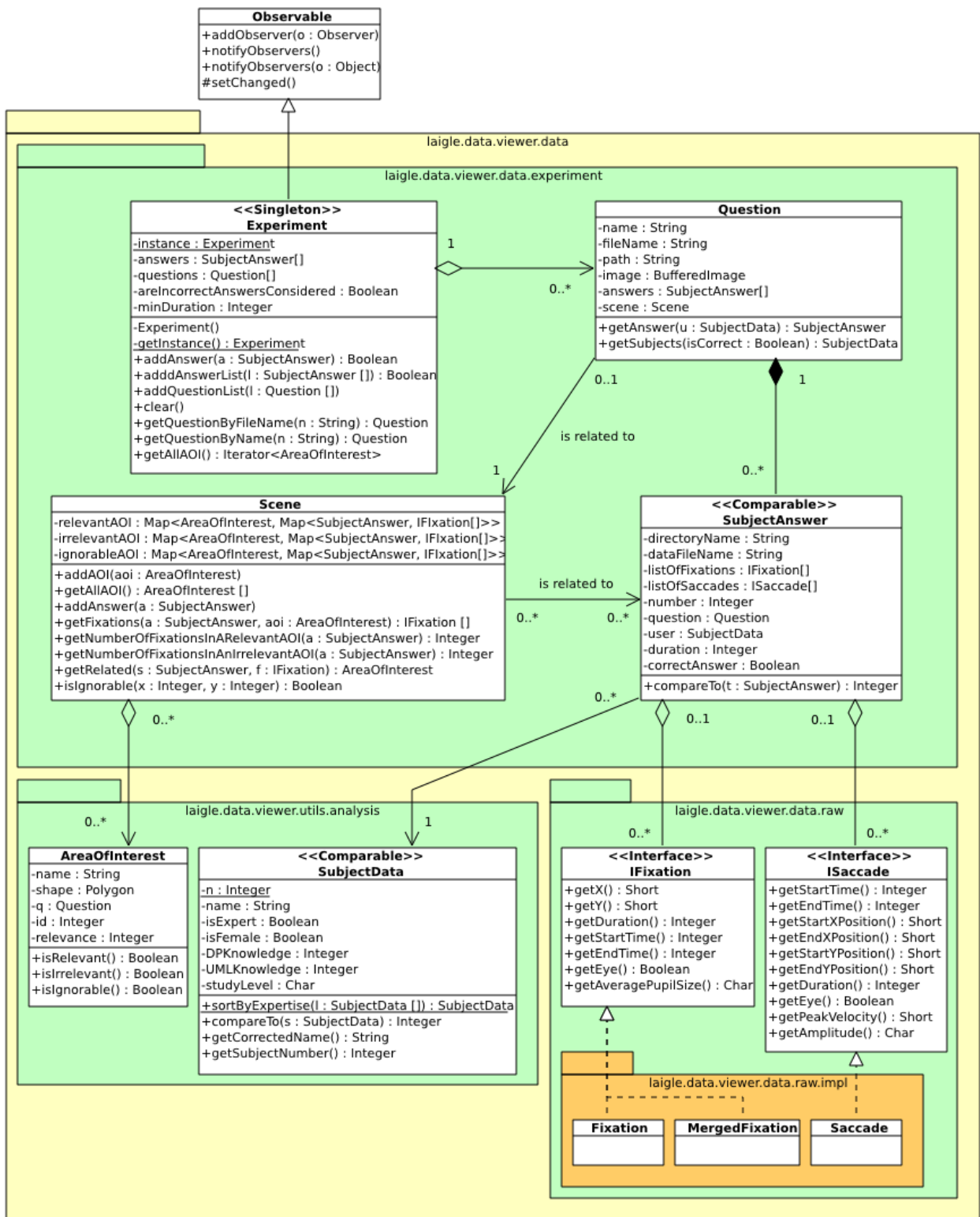


Figure 3: Class diagram - Package `laigle.taupe.viewer.data`



## Description

- **command:** The commands are coded through the abstract class `AbstractCommand` and its sub-classes. The reason why the class extends the `Runnable` interface is that it must be executed in parallel with the graphical user interface in a separate thread. Regarding the `Observer` interface, the notifications are thrown to the GUI through the *observer* design pattern.
- **parameter:** An instance of `Argument` can be a simple value or a set of values. If a parameter is an instance of `FileArgument`, it represents a file or a directory in the file system. To help the developer, the class `ArgumentGUIFactory` is able to convert a parameter into a graphical element thanks to the Swing API. The parameters are displayed by the panel coded in the class called `OptionPanel`.
- **notification:** The notifications in the list are described by the class `Notification`, stored in the instances of `ListModificationModel` and displayed through the class `ListNotificationRenderer`. These classes redefine the default Swing classes.
- 

## Choices

TAUPE, the set of *commands*, and the graphical user interface (GUI) are designed following the classical design pattern *Model-View-Controller* (MVC).

In the package `laigle.taupe.viewer.command`, the main model is represented by the class `AbstractCommand`, the view by the class `CommandView` and the controller by the class `CommandController` that describes the whole set of event listeners related to the main frame of the software.

The choice of the *observer* design pattern for the notifications is justified by the ease to send a notifications to the main GUI.

The interface named `ListenerProvider` is used to define the event listeners that describe how the model is modified by the Swing elements of the `Argument`'s instances.

Some user interfaces are *singleton* to avoid multiple instantiation of the unmodifiable user interface. For example, the "About View" displays every time the same elements.

The process to modify the set of parameters is described in Section 4.1. The process to add a new *command* is described in Section 4.2.

### 3.4 Preferences

This section describes the package `laigle.taupe.viewer.configuration` (see FIGURE 5). The “Preference” system allows the developer to define a default configuration for TAUPE . Moreover, it allows the user to use this default configuration and to modify its values. The configuration’s data displayed through the software are stored in the software’s cache memory and are recorded into a `.properties` file.

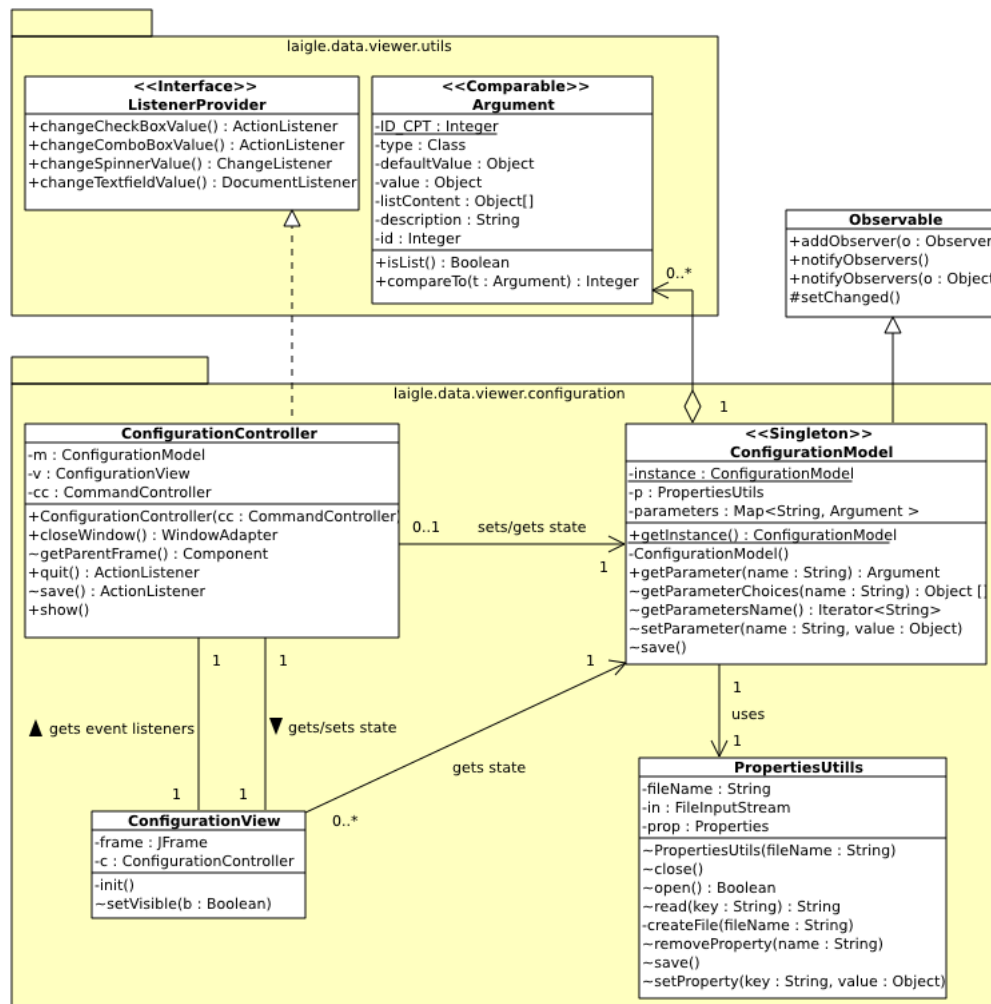


Figure 5: Class diagram - Package `laigle.taupe.viewer.configuration`

#### Description

The class named `PropertiesUtils` is used to record the preferences permanently *i.e.*, in a `.properties` file. The classes `ListenerProvider` and `Argument` are already defined in Section 3.3.

#### Choices

The GUI related to the “Preference” system is designed with the MVC where the model is stored in the `ConfigurationModel`’s instance, the view is displayed thanks to the `ConfigurationView` class and the controller, which handles the events about the modification of the configuration by

the user, is defined by the `ConfigurationController`'s instance.

To allow the package `laigle.taupe.viewer.command` to use easily the configuration in the commands' parameters and the parser's chooser, the class `ConfigurationModel` is designed with the singleton design pattern.

The process to modify the set of preferences is described in Section [4.7](#).

### 3.5 Groups

This section describes the package `laigle.taupe.viewer.utils.group` (see FIGURE 7). The notion of group allows TAUPE 's users to classify subjects in some (non-exclusive) subsets of subjects in accordance with their characteristics.

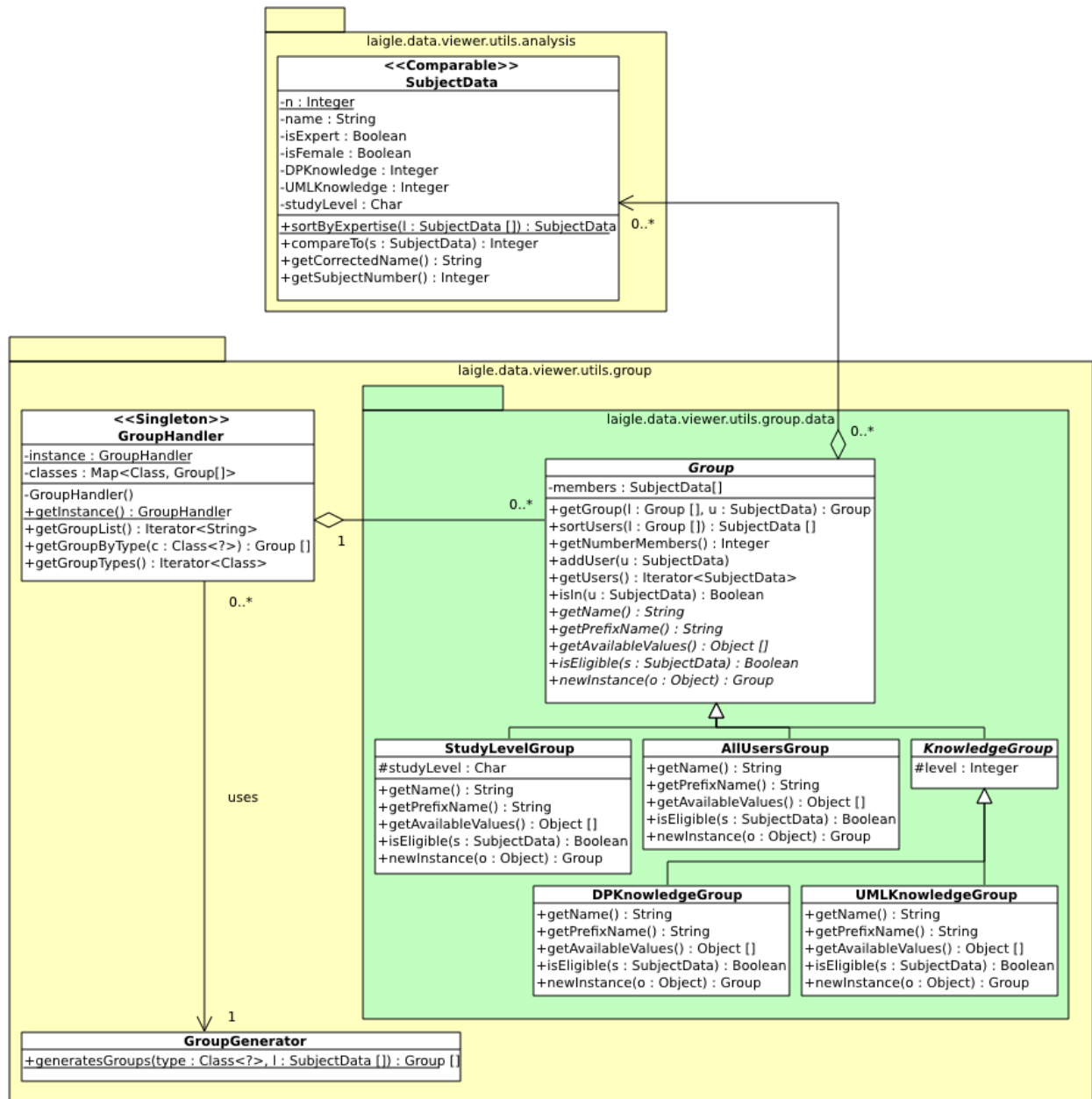


Figure 6: Class diagram - Package `laigle.taupe.viewer.utils.group`

#### Description

- **group**: A group of subjects. A group is defined by its type and characteristics. For example, groups that classify the subjects based on their UML knowledge are defined by the class

UMLKnowledge and inner attribute `level1`, which contains the level of its members.

- **subject:** Already defined in Section [3.2](#)

### Choices

The kinds of groups are defined by all the sub-classes of `Group`. Those classes are loaded by the Java's system called *reflection*. The *singleton* represented by the class `GroupHandler` is used to handle the whole set of groups. The *singleton's* instantiation loads the whole set of groups and subjects are added inside each one in a convenient way using *GroupGenerator*.

The process to add a new kind of group is described in Section [4.4](#).

### 3.6 Computation

This section describes the package `laigle.taupe.viewer.computation` (see [FIGURE 7](#)). This package represents one *command* of the software and is called using the class named `ComputeResults`.

#### Description

- **result**: The data produced by the classes which implement the interface named `ICompute`, here called `computation`. All the results extend the abstract class called `AbstractResult`.
- **printer**: The tool that generates an output from a result. All the printers implement the interface called `IPrinter`.

#### Choices

The hierarchy of results is represented using the *composite* design pattern. The class `ContainerResult` plays the role of *Composite* and the class `AbstractResult` is the *Component*.

The printers generate some output from the results. To isolate the role of the printing and the role of the browsing, the module uses the *visitor* design pattern. The result hierarchy plays the role of *Element*, it describes, through the method `accept`, how to browse and handle the hierarchy. For example, the method `accept` defined in `ContainerResult` creates a new directory (through the related `visitor` method of the *printer*) and launches `accept` from each its children into the new directory. Therefore, the hierarchy browsing is currently a *depth first search*. The printers play the role of *visitor*, they define a set of methods called `visitXXX` that describe how to print a specified result. They are called by the Java's reflection system.

The method to generate the results is `compute(GroupHandler gh, Experiment e)` from the `ICompute`'s sub-classes. The `GroupHandler`'s singleton is used to sort and classify the results and the `Experiment`'s singleton contains all the data needed by a computation.

This module is launched from the *command* represented by the class `laigle.taupe.viewer.command.impl.ComputeResult`, which allows TAUPE 's users to choose the desired computations and the desired printers. The way to add a new *computation* is described in [Section 4.3](#). The way to add a new *printer* is described in [Section 4.6](#).



### 3.7 Parsers

This section describes the package `laigle.taupe.viewer.parsers` for the eye-tracker's files (see FIGURE 8).

#### Description

- **parser:** A tool which is used to read some files generated from eye-tracking system and to fill the cache data about the related experiment. All the parsers extend the class named `AbstractParser` and must be member of the package `laigle.taupe.viewer.parsers.types`.

#### Choices

The parsers implement the interface `Runnable` to be executed in parallel with the GUI. They follow the *observer* design pattern to easily notify the GUI.

The list of parsers in `laigle.taupe.viewer.parsers.types` are loaded using the Java's reflection system and with the class called `Package`. The package `laigle.taupe.viewer.parsers.utils` contains a set of tools to parse some kind of files:

- `AbstractParser` uses a `RelevantSlide`'s instance to know whether or not a file must be taken in account.
- `AbstractParser` uses a `SubjectParser`'s instance to get some information in the `.subject` files.
- `AbstractParser` uses an `AOIParser`'s instance to get the areas of interest about the questions.
- `AbstractParser` uses an `OffsetParser`'s instance to load the information about a hypothetical offset on the questions's image.
- `GazeTrackerParser` uses an instance of a `ILineParser`'s child to parse different kinds of line.

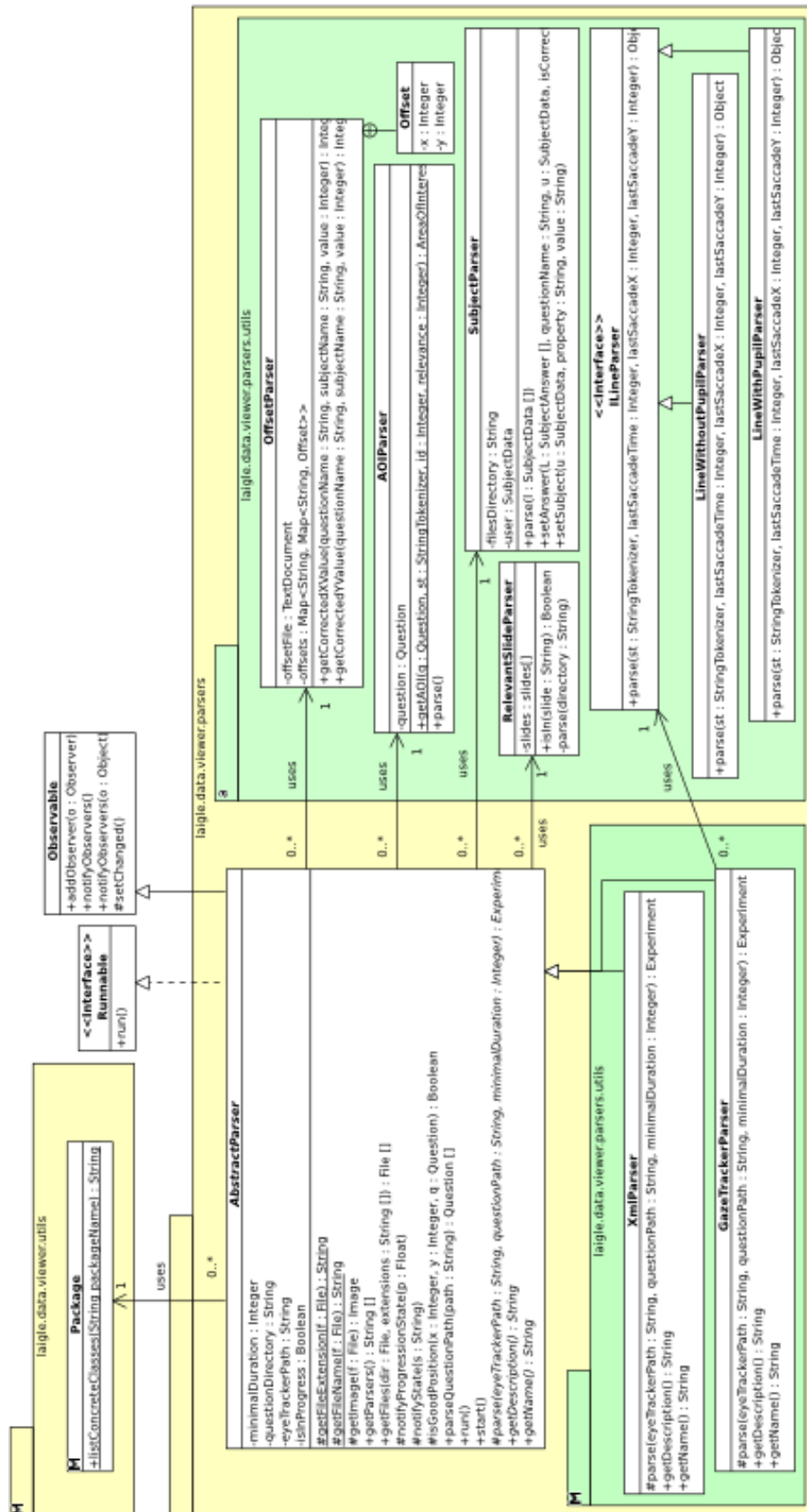


Figure 8: Class diagram - Package laigle.taupе.viewer.parsers

### 3.8 The graphical visualisation system

This section describes the package `laigle.taupe.viewer.graphics` (FIGURE 9). Actually, the graphical visualisation is a *command* of the program. It allow the user to visualise the whole experiment.

#### Description

- **drawer**: An option is the *visual system* which is able to draw something on the GUI's panel. For example, a drawer is able to draw the images associated with questions. All the drawers extend the class named `AbstractDrawer` and are members of the package `laigle.taupe.viewer.graphics.options`.

#### Choices

This feature follows the *MVC* design pattern where `ExperimentView` is the view, `ExperimentDomain` is the model (more precisely the model domain, as described in [4]), and `ExperimentController` is the controller that defines all the event listeners.

`QuestionView` represents the main panel in the GUI where is drawn all the `AbstractDrawer`'s executions.

The “graphical visualisation” system is launched by the instantiation of the controller.

`ExperimentDomain` implements the interface `Runnable` to notify the progression of the drawing to the `QuestionView`'s instance using the *observer* design pattern.

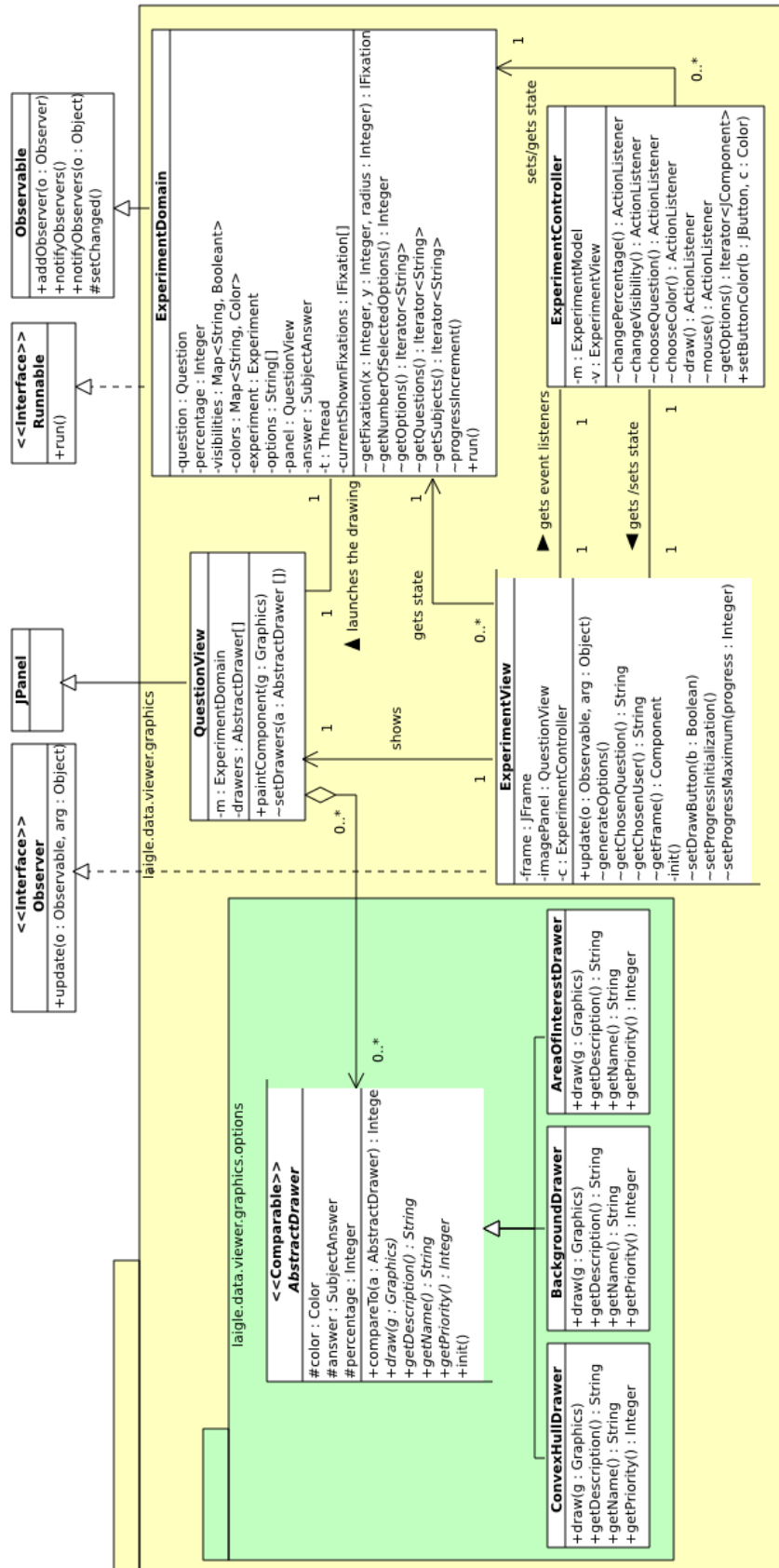


Figure 9: Class diagram - Package laigle.taupe.viewer.graphics

### 3.9 The “AOI Maker” module

This section describes the package `laigle.taupe.viewer.aoimaker` (see FIGURE 10). This module is a *command* of the program. It is used to create the files that contains the description of the areas of interest using a graphical interface.

#### Choices

This module contains a `public static void main(String[] args)` method that allows it to be launched without launching the whole software. Therefore, the developer may start the module by this `main` method or by the `AOIMakerController` instantiation.

The “AOI Maker” follows the design pattern *Model-View-Controller* (MVC) where the class named `AOIMakerController` is the controller, `AOIMakerModel` is the model, and the main view is `AOIMakerView`. `AOIMakerView` shows a set of panels represented by the `BottomPanel`’s sub-classes. These sub-classes’ displays are exclusive, so the methods `showXXX()` defined in `AOIMakerView` are used to show one *bottom panel* at a time. `AOIComboBoxModel` is a part of the model and redefines the default Swing system.

The *observer* design pattern is used to make easier the communication between the model and the view.

The class named `AOIMakerModel` implements `Runnable` to write the result file in parallel with the GUI.

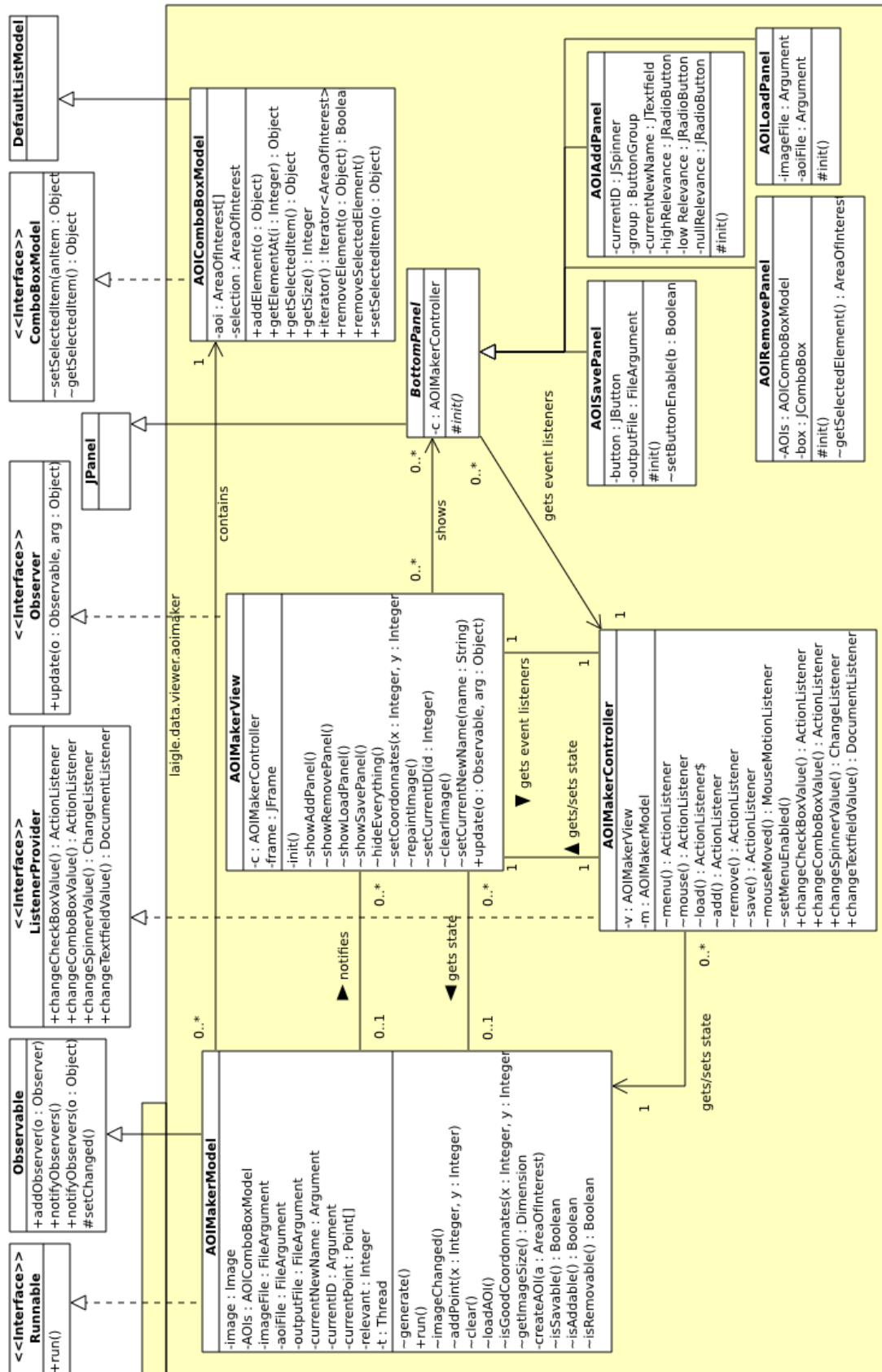


Figure 10: Class diagram - Package laigle.taupe.viewer.aoimaker

## 4 Maintainability

This section explains how developers can add or modify some features of the software. The following list of possible modifications is not comprehensive but is a good start to understand the software following recurring scenarios.

### 4.1 How to Modify a Set of Parameters

As mentioned in Section 3.3, each *command* has a set of parameters that allows TAUPE 's users to give some arguments to the command's execution.

To add some parameters to a command, a developer must modify the *protected* method named `initArguments()` in the related command's class (which must extend the class named `AbstractCommand`). This method just add a set of instances of `Argument` to the attribute called `arguments`. These arguments will be automatically handled by the graphical user interface (GUI). Then, the developer can use them in the method `run()` of the desired *command*.

#### Advice

- Declare the name of the new argument as a *constant* in the related command's class. It will make the new argument easier to access.

### 4.2 How to Add a New *command*

If a developer wants to add a new feature to the software, she must add a class that extend `AbstractCommand` and put it in the package `laigle.taupe.viewer.command.impl`. There are four abstract methods to implement:

[`String getName()`] This method gives the name of the new feature. The name is displayed in a `JButton` in the main frame.

[`String getDescription()`] This method gives the description of the new feature. The description is displayed in a `JLabel` in the panel which contains the arguments.

[`void initArguments()`] The developer must implement this method as described in Section 4.1.

[`void execution()`] This method is the actual content of the feature that will be executed in a new *thread*. This method must notify the software about its progression using the method `notifyView`.

The GUI will automatically take into account of a new *command*. To add a feature which is not represented by a freezed (disabled) button at the software's start, the developer must modify the method `CommandController.initUnfreezableElements` to add the name of the command's class to the attribute `unfreezableElements`.

### 4.3 How to Add a New Computation

The *computation* package is described in Section 3.6. There are three steps to add a new algorithm. Let us assume that the developer wants to add the addition as an algorithm to the software.

**Step 1** Implement the result generated by this new algorithm. A result is a class which extends the class `AbstractResult`. An instance of this new class must contains all the result data.

The developer can use the class `ContainerResult` to describe its result as a hierarchy. All the sub-classes of `AbstractResult` implement the method `accept`:

**boolean accept(p : IPrinter, path : String)** This method describes how to browse the related result with the printer `p`. For instance, `ConvexHullResult.accept` just calls the `visit` method from the printer but `ContainerResult.accept` calls `visit` on each of its sub-results (*depth first search*).

```

1 package laigle.taupe.viewer.computation.result;
2
3 public class AdditionResult implements AbstractResult {
4
5     private int result;
6
7     @Override
8     public boolean accept(IPrinter p, String path) {
9
10        ....
11
12        return p.visit(this, path);
13    }
14 }

```

**Step 2** Create a new class that implements the interface `ICompute` and put it in the package `laigle.taupe.viewer.computation.impl`. There are two methods to implement:

**[getName() : String]** This method gives the name of the new algorithm. This name is displayed in the `JComboBox` in the feature named `ComputeResults`.

**[compute(gh : GroupHandler, e : Experiment) : AbstractResult]** The actual algorithm to be called on an *experiment* (defined in Section 3.2). The developer can use a `GroupHandler` to sort the sub-results.

In this case:

```

1 package laigle.taupe.viewer.computation.impl;
2
3 public class AdditionComputation implements ICompute {
4
5     @Override
6     public String getName() {
7         return "Addition";
8     }
9
10    @Override
11    public AbstractResult compute(GroupHandler gh,
12        Experiment e) {
13
14        AdditionResult a = new AdditionResult(...);
15        ...

```

```

15     return a;
16   }
17 }

```

**Step 3** Create the printing method for each *printer* for this new result. In this case:

```

1  package laigle.taupe.viewer.computation;
2
3  public interface IPrinter {
4
5      ...
6
7      @Override
8      public boolean visit(final AdditionResult r, final
9                          String path);
10
11     ...
12 }

```

### Advices

- Do not use the configuration system (the `ConfigurationModel`'s singleton). Instead, add some parameters to the `ComputeResults` class.
- The order of those steps is not mandatory but it is strongly suggested to create the result first because the two other steps depend on the result's structure.

## 4.4 How to Add a New Group of Subjects

The notion of group is defined in Section 3.5. To add a new group in the system, the developer just has to create a new class which extends the abstract class named `Group`. She must add this new class in the package `laigle.taupe.viewer.utils.group.data`. There are five abstract methods to implement to make the group effective:

[**String getName()**] This method gives the name of the group according to its members' characteristics.

[**String getPrefixName()**] A short version of the method `getName()`'s result.

[**Iterator<Object> getAvailableValues()**] The possible values of the characteristic of the group's members.

[**Object newInstance(Object o)**] This method plays the role of constructor with an argument.

[**Boolean isEligible(s : SubjectData)**] This method checks whether or not the subject `s` can be added to the group according to its characteristics.

Let us assume that a developer wants to create three groups related to the gender of their members, she will implement a new class named `GenderGroup`:

```
1 package package laigle.taupe.viewer.utils.group.data;
2
3 public class GenderGroup extends Group {
4
5     protected char gender;
6
7     public static final char MALE = 'M';
8     public static final char FEMALE = 'F';
9     public static final char UNKNOWN = 'U';
10
11     public GenderGroup() {
12         super();
13         this.gender = GenderGroup.UNKNOWN;
14     }
15
16     public GenderGroup(char gender) {
17         super();
18         this.gender = gender;
19     }
20
21     @Override
22     public String getName() {
23         String result = "";
24
25         switch (this.gender) {
26
27             case GenderGroup.FEMALE :
28                 result = "Female";
29                 break;
30
31             case GenderGroup.MALE :
32                 result = "Male";
33                 break;
34
35             case GenderGroup.UNKNOWN :
36                 result = "Unknown";
37                 break;
38
39         }
40         return result;
41     }
42
43     @Override
44     public String getPrefixName() {
45         String result = "";
46
47         switch (this.gender) {
```

```

48
49     case GenderGroup.FEMALE :
50         result = "F";
51         break;
52
53     case GenderGroup.MALE :
54         result = "M";
55         break;
56
57     case GenderGroup.UNKNOWN :
58         result = "U";
59         break;
60     }
61     return result;
62 }
63
64 @Override
65 public Iterator<Object> getAvailableValues() {
66     final List<Object> l = new ArrayList<Object>();
67     l.add(GenderGroup.MALE);
68     l.add(GenderGroup.FEMALE);
69     l.add(GenderGroup.UNKNOWN);
70     return l.iterator();
71 }
72
73 @Override
74 public boolean isEligible(final SubjectData s) {
75     if (s == null) {
76         throw new IllegalArgumentException("s cannot be null");
77     }
78     return this.gender == GenderGroup.FEMALE && s.isFemale()
79         || this.gender == GenderGroup.MALE && !s.isFemale();
80 }
81
82 @Override
83 public Group newInstance(final Object o) {
84     if (o == null) {
85         throw new IllegalArgumentException("o cannot be null");
86     }
87     return new GenderGroup((Character) o);
88 }
89 }

```

#### 4.5 How to Add a New Parser

The parsing system is described in Section 3.7. If a TAUPE's user acquires a new eye-tracking system or if the structure of the generated eye-tracking files changes, a developer must add a new parser.

When a new parser is added, it is represented by a new sub-class of the abstract class named `AbstractParser` and this new class must be in the package `laigle.taupe.viewer.parsers.types`. There are three abstract methods to implement:

[**String getName()** ] The parser's name is shown to the user when she must choose a parser.

[**String getDescription()** ] This methods gives the description of the typical input files required by this new parser. The description is displayed on the panel which allows TAUPE users to choose the parser.

[**Experiment parse(String eyeTrackerPath, String questionPath, int minDuration)** ] This method performs the actual parsing. The methods `notifyProgressState` and `notifyState` must be used to notify the software about the parsing's progression.

### Advices

- The abstract class `AbstractParser` defines a lot of usefull methods to help the parsing of files.
- It is easier to implement the `parse` method following those steps:
  1. Get a `Experiment`'s singleton and set its `minDuration`'s value
  2. If needed, create an instance of `RelevantSlideParser`.
  3. Use the *protected* method `parseQuestionPath` implemented in `AbstractParser` to get the questions from the directory related to the `questionPath`'s value. Then, add those questions to the experiment.
  4. Use the *protected* method `getFiles` implemented in `AbstractParser` to get the files generated by the eye-tracking device (in the directory related to the value of the parameter named `eyeTrackerPath`).
  5. For each of these files, parse its content and add the resulting instances of `SubjectAnswer` to the experiment.

## 4.6 How to Add a Printer

Printers are described in Section 3.6 and a complete example is given in Section 4.4. A developer who wants create a new type of output for a set of results must implement the interface `IPrinter`. There are as methods to implement as there are children of `AbstractResult`. These `visit` methods typically create per file by result and `ContainerResult.visit` creates a directory for its children. The developer must put the new printer into the package `laigle.taupe.viewer.computation.printer`.

## 4.7 How to Add an Element to the Preferences

It is easy to add a new element in the preference (described in Section 3.4). A developer must add a new `Argument`'s instance into the attribute named `parameters` of `ConfigurationModel`. Usually, these arguments are defined in the constructor of `ConfigurationModel`. Then, the GUI will automatically take into account of the new argument.

## 4.8 How to Add an Option for the *Graphical Visualisation*

The *graphical visualisation* system is already described in Section 3.8. If a developer wants to add a new *drawer*, she must create a new class which extends the abstract class `AbstractDrawer` and put it in the package `laigle.taupe.viewer.graphics.options`. The GUI will automatically take into account of the new drawer. There are four methods to implement:

[**String getName()** ] This method gives the name that is displayed as the label of its related checkbox in the GUI.

[**String getDescription()** ] This method gives the checkbox's *tooltip* for this drawer.

[**int getPriority()** ] If this drawer's priority equals zero, then the drawer will draw before all the drawers that have a higher priority. The developer must take into account of the priority of each *drawer* to avoid a problem of juxtaposition.

[**void draw(g : Graphics)** ] This method obtain information about the answer to draw using the *protected* attribute `AbstractDrawer.answer`. The *protected* attribute `color` can be used in this method.

Let us assume that a developer wants a *drawer* that is able to write the number of fixations of an answer on the left top corner of the panel. She will write the following class named `NumberOfFixationsDrawer`:

```

1 package laigle.taupe.viewer.graphics.options;
2
3 public class NumberOfFixationsDrawer extends AbstractDrawer {
4
5     @Override
6     public String getName() {
7         return "Number_of_fixations";
8     }
9
10    @Override
11    public String getDescription() {
12        return "It_writes_the_number_of_fixations_on_the_left_top_
13            corner_of_the_image.";
14    }
15
16    @Override
17    public int getPriority() {
18        return 2;
19    }
20
21    @Override
22    public void draw(Graphics g) {
23        if (g == null) {
24            throw new IllegalArgumentException("g_cannot_be_null");
25        }
26        g.setColor(this.color);

```

```
27     g.drawString(answer.getNumberOfFixations()+"_fixations",  
28                   0, 0);  
29 }
```

## 5 Compiling

To compile the sources and to generate the `.jar` file, a developer must use the *Apache Ant*<sup>6</sup> tool:

```
1 ant
```

or

```
1 ant run
```

If the developer wants to modify the compilation's preferences, she must modify the file named `build.xml` that describes how the `ant` command is configured. For example, if the software needs a new *library*, the `classpath` attribute must be updated and the `jar` task must contain a new `zipfileset` element.

---

<sup>6</sup><http://ant.apache.org>

## 6 Test Cases

The test cases related to TAUPE are written with *JUnit*<sup>7</sup> and are stored in the package `test`. The directory `rsc/tests` contains the files that are parsed and used by the tests. The class named `Initialisation` defines the method `setUp()` that parses the input files. To make the development of tests easier, the test classes can extend this class.

According to the 4<sup>th</sup> version of *JUnit*, it is recommended to add the annotation `org.junit.Test` to each test method. It is now useless to extend the class `junit.framework.TestCase`.

For example, if a developer wants to write a test that checks whether the value of `Experiment.minDuration` is 0, she should write the following class:

```
1 package test;
2
3 public class TestExperiment extends Initialisation {
4
5     @Test
6     public void testMinDuration() {
7         Assert.assertEquals(0, this.e.getMinDuration());
8     }
9
10 }
```

---

<sup>7</sup><http://www.junit.org>

## 7 Licence

### 7.1 GNU GPL

TAUPE follows the terms of the GNU General Public Licence as published by the Free Software Foundation. A copy of this licence is given in `./COPYING-TAUPE.txt` or can be found at <http://www.gnu.org/licenses/gpl.txt>.

The following text must be added at the top of each source file in the software.

```
1 /*
2  * This file is part of TAUPE (Thoroughly Analysing the
3  * Understanding Programs through Eyesight).
4  *
5  * TAUPE is free software: you can redistribute it and/or
6  * modify
7  * it under the terms of the GNU General Public License as
8  * published by
9  * the Free Software Foundation, either version 3 of the
10 * License, or
11 * (at your option) any later version.
12 *
13 * TAUPE is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty
15 * of
16 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See
17 * the
18 * GNU General Public License for more details.
19 *
20 * You should have received a copy of the GNU General Public
21 * License
22 * along with TAUPE. If not, see <http://www.gnu.org/licenses
23 * />.
24 */
```

### 7.2 Icons

Here follow the licences related to each icon used in TAUPE .

Directory	File	Licence
icons/	about.png	GNU GPL
icons/	about_small.png	GNU GPL
icons/	add.png	GNU GPL
icons/	clear.png	GNU GPL
icons/	directory.png	GNU GPL
icons/	draw.png	GNU GPL
icons/	exit.png	GNU GPL
icons/	eye.png	Creative Commons <sup>8</sup>
icons/	file.png	GNU GPL
icons/	load.png	GNU GPL
icons/	load_small.png	GNU GPL
icons/	notification0.png	GNU GPL
icons/	notification1.png	GNU GPL
icons/	notification2.png	GNU GPL
icons/	ok.png	GNU GPL
icons/	preferences.png	GNU GPL
icons/	preferences_small.png	GNU GPL
icons/	process.png	GNU GPL
icons/	question.png	GNU GPL
icons/	remove.png	GNU GPL
icons/	save.png	GNU GPL
icons/	ShowGraphicalExperiment_small.png	GNU GPL
icons/	subject.png	GNU GPL
icons/	unknown.png	GNU GPL
icons/command/	AOIMaker.png	GNU GPL
icons/command/	GetCacheSummary.png	GNU GPL
icons/command/	ComputeResults.png	GNU GPL
icons/command/	ShowGraphicalExperiment.png	GNU GPL

<sup>8</sup>Creative Commons(Attribution-NonCommercial 3.0 Unported (CC BY-NC 3.0))

## 8 Contacts

### Team

Alphabetically:

- **De Smet Benoît** Main developer (FUNDP)  
ben.desmet@gmail.com
- **Guéhéneuc Yann-Gaël** Team Leader (Ptidej)  
yann-gael.gueheneuc@polymtl.ca at .umontreal.ca
- **Lempereur Lorent** Main developer (FUNDP)  
lorent.lempereur@gmail.com

### Links

- **École Polytechnique De Montreal** Software Engineering Department  
<http://www.polymtl.ca/gigl/>
- **Facultés Notre-Dame-De-La-Paix (FUNDP Namur)** Faculty of Computer Sciences  
<http://www.info.fundp.ac.be>
- **Ptidej**  
<http://www.ptidej.net>

## Index

answer, 8  
AOI maker, 22  
area of interest, 8, 18, 22  
argument, 24  
  
calculation, 24  
command, 10, 16, 20, 22, 24  
compilation, 4, 32  
component, 16  
composite, 16  
composite design pattern, 16  
computation, 16, 24  
configuration, 12, 29  
contact, 4, 36  
controller, 11, 12, 22  
  
depth first search, 16, 25  
design pattern, 6  
directory, 5  
domain, 20  
drawer, 20, 30  
  
element, 16  
experiment, 8, 25  
  
fixation, 8  
  
group, 14, 26  
  
icon, 5  
index, 20  
  
Java, 4  
javadoc, 5  
  
laigle, 5  
licence, 4, 34  
ListenerProvider, 11  
  
model, 11, 12, 20, 22  
MVC design pattern, 11, 12, 20, 22  
  
notification, 11  
  
observer design pattern, 11, 18, 20, 22  
offset, 18  
  
package, 7  
parameter, 11, 24  
parser, 13, 18, 28  
  
precondition, 5, 6  
printer, 16, 26, 29  
priority, 30  
  
question, 8, 18  
  
reflection, 16, 18  
result, 16, 24  
runnable, 11, 18, 20, 22  
  
saccade, 8  
scene, 8  
singleton, 8, 11, 13, 15, 16  
subject, 8, 14, 18  
  
test case, 4, 33  
  
view, 11, 12, 20  
visitor, 16  
visitor design pattern, 16  
  
XML, 4

## References

- [1] Y.-G. Guéhéneuc, “Taupe: Towards understanding program comprehension,” in *Proceedings of The Conference of the Center for Advanced Studies on Collaborative Research (CASCON’06)*, October 2006, pp. 1 – 13.
- [2] R. J. Erich Gamma, Richard Helm and J. Vlissides, *Design Patterns Elements of Reusable Object-Oriented Software*. Addison-Wesley Pub Co, 1995.
- [3] J. D. from Sun, “Programming With Assertions,” <http://download.oracle.com/javase/1.4.2/docs/guide/lang/assert.html#usage-conditions>, accessed November 28 in 2010.
- [4] J. Deacon, “Model-View-Controller (MVC) Architecture,” 2009.