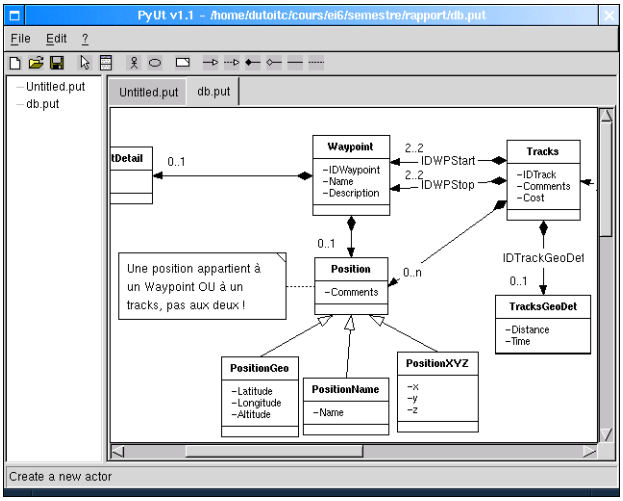
PyUt Developers documentation

PyUt Team

July 17, 2002



Contents

1 Introduction

1.1 What is *PyUt* ?

PyUt stands for Python UML Tool and supports actually class diagram edition, including notes. It also supports use cases diagrams. *PyUt* is developped on Sourceforge, under the terms of the GPL license see file LICENSE on CVS and distributions for more informations on the GPL license.

1.2 Motivation

As *PyUt* is getting stronger and more developped, it becomes necessary to document it for developpers, so that new developers on *PyUt* will be able after reading this documentation to understand how *PyUt* works and how he can works on it by itself.

1.3 Audience

This document is written exclusively for developers who want to participate to *PyUt*.

1.4 Scope of this document

This document will try to answer to the following questions :

- What are all the developed modules ?
- How *PyUt* works ?
- How plugins works and how to make them ?
- Which technologies are used ?

1.5 Others documents

PyUt developers should consider reading all *PyUt* documents, and among them :

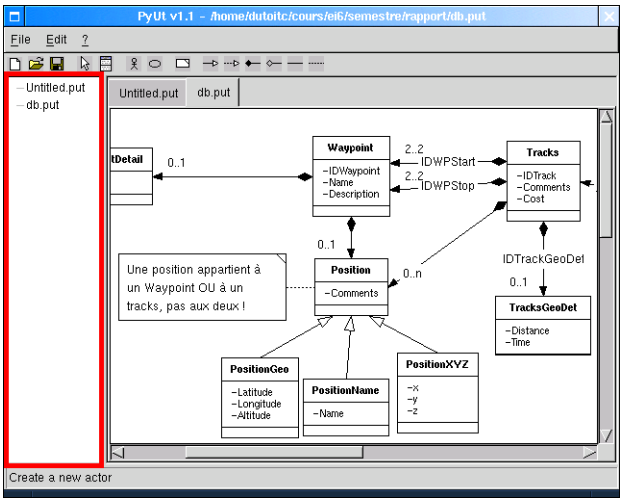
- *PyUt* Quality document, situated on CVS in `pyutDoc/quality/quality.pdf`
- *PyUt* User documentation, situated on CVS in `pyutDoc/usr/pyutxx` and in *PyUt*, in the Help/Index menu
- *PyUt* template file, on `pyut/src/templates/template.py`
- ... all documentation on *PyUt*'s website : `http://pyut.sf.net`

1.6 Contacts

For any question, feel free to contact *PyUt*'s team, on sourceforge website :
`http://www.sf.net/projects/pyut` or `http://pyut.sf.net`

You can also join directly *PyUt*'s contributors (see file CREDITS in CVS, in `pyut/CREDITS`).

2 Project tree



Here is a screenshot of *PyUt*. On the left you can see the project tree. A project tree contain all the currently opened files.

2.1 How it works

The project tree is handled in the file `FileHandling.py`. It is in relation with the class diagram notebook (on the right of the screen). It is a wxTreeCtrl control, with a root node named "Root" and child nodes, which name is filename, associated with a python data (pyData) which is the UML class diagram frame instance.
When a file is opened or created, it is loaded in a new frame, attached as root node in the project tree and inserted in the class diagram notebook.

2.2 Future

In the future, the project tree could contain files nodes (like as today), which could contain diagrams frames of many kind : class diagrams, collaboration diagram, sequence diagram, ...
A file will be a project file, and could have others kind of information : URL, file links, MIME links, ActiveX controls, contacts and projects informations.

3 XMI

3.1 What's XMI

XMI is an interchange format for metadata that is defined in terms of the Meta Object Facility (MOF) standard.

It is a set of XML tags settled by OMG as a way to describe in standard terms the structure of data about data ("metadata"). This is useful in communicating between CASE tools, and in describing a "metadata repository".

3.2 What's MOF

The MOF (Meta Object Facility) is the OMG's adopted technology for defining metadata and representing it as CORBA objects. It MOF supports any kind of metadata that can be described using Object Modeling techniques.

3.3 What is the objective of XMI?

The objective of XMI is to allow the exchange of objects from the OMG's Object Analysis and Design Facility. These objects are more commonly described as UML (Unified Modeling Language) and MOF . Finally, the XML Metadata Interchange Format (XMI) specifies an open information interchange model that is intended to give developers working with object technology the ability to exchange programming data over the Internet in a standardized way.

3.4 Who is behind XMI?

IBM and Unisys originally sponsored XMI as a result of their joint work on the OMG UML and MOF standards and others.

3.5 *PyUt* and XMI, which relation ?

PyUt has intentend to gives his users ability to use their class diagrams, use case diagram ... with commercial UML tools like rational Rose. That's why *PyUt* propose the Export/Import XMI functions. With Rose, to be able to export and import XMI, we should install UnisysRoseXMLTools. In *PyUt* we use version 1.0 for XMI. Actually we can read an XMI file (import), the export function doesn't finished in the time wich this lines was written.

3.6 Where we can find XMI specification ?

OMG UML Ressource page : `http://www.omg.org`

UniSys : `http://www.unisys.com`

IBM : `http://www.ibm.com`

4 IOPython, Roundtrip Python

This section describe the IOPython plugin (located in the 'Plugins' directory). This plugin does roundtrip for Python's language : Python code generation and Python reverse engineering.

4.1 Python roundtrip example

The figures 1 and 2 (pages 7 and 8) shows an example of Python code generation. The figures 3 and 4 (pages 9 and 10) shows an example of Python reverse engineering.

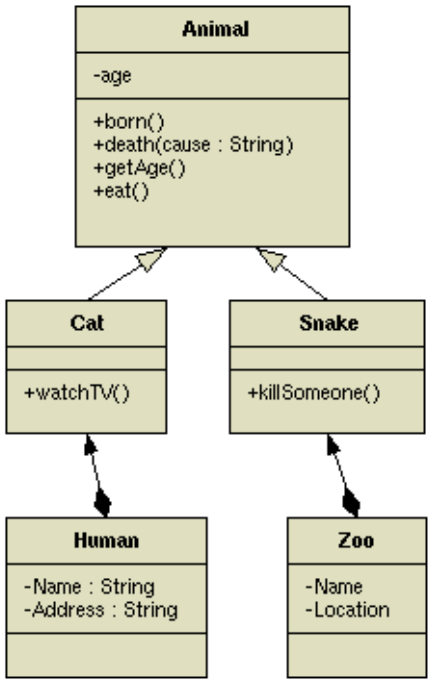


Figure 1: UML class diagram before python code generation

4.2 Python code generation

4.2.1 Current status

The Python code generation is fully-working and has no features request at this time.

4.2.2 How it works

The Python code generation plugin analyzes all classes and generates the code classes by classes. For each classes, it generate all methods code, putting it into a dictionary, as this one

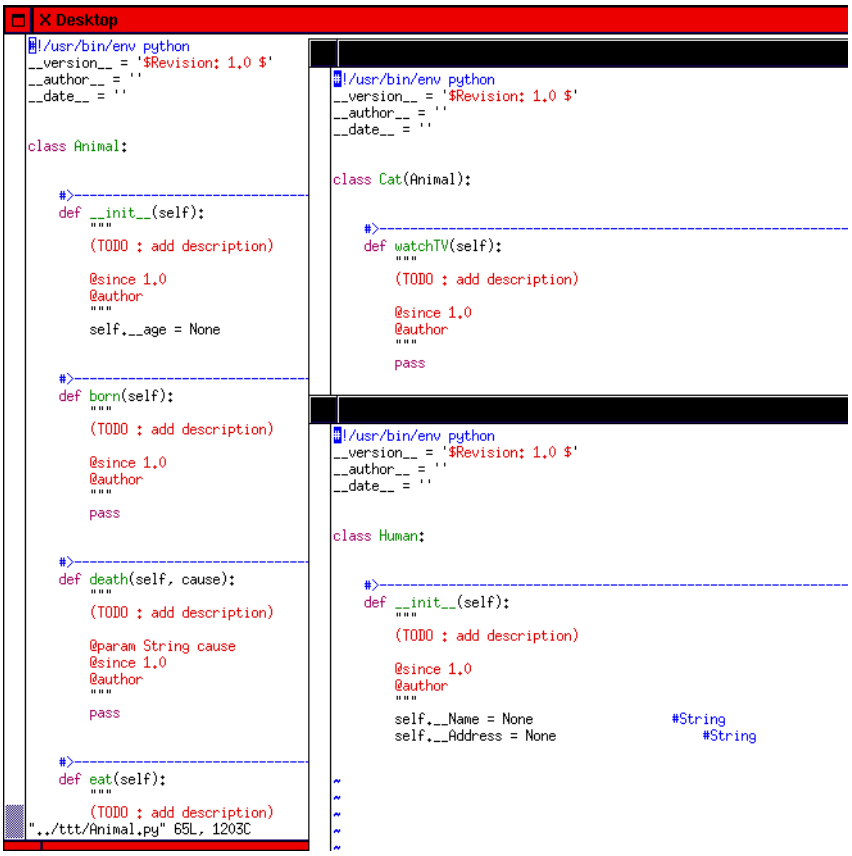


Figure 2: Extract of Python generated code

will be used to sort methods and generate after-time the constructor. Each generated class code is written to a file in a specified directory and is a fully Python 2.2-compliant.

4.2.3 Languages conventions

The generated code has been tested for Python 2.2, but this could work with previous version of Python. The generated comments in the code are pydoc-compatible, with some enhanced tags, the sames as in JavaDoc.

4.3 Python reverse engineering

4.3.1 Current status

The Python reverse engineering is well-working. It let us select many files, analyze them and let us select which classes we want to select.

```
# Python reverse engineering example classes
# Author : C.Dutoit

class Vehicle:
    def __init__(self, gpsDrive):
        self._size = None
        self._gpsDrive = gpsDrive

    def goto(self, latitude, longitude):
        self.__gpsdrive.goto(latitude, longitude)


import Genius
class Bicycle(Vehicle):
    def __init__(self, size):
        self._size = size

    def repair(self):
        Genius.command("REPAIR", self)


import divination
class Plane(Vehicle):
    def __init__(self, capacity):
        self._size = divination.divination(capacity)


    def goto(self, latitude, longitude):
        if (latitude  != WTC_Latitude and \
            longitude != WTC_Longitude):
            self.__gpsdrive.goto(latitude, longitude)
```

Figure 3: Python code to reverse

4.3.2 How it works

This plugin uses Python self-inspection capabilities, which is a good method to reverse Python code, but this method introduce the following disagrement :

- Only inheritance can be reversed, no agregation nor composition will ever be reversed, until the method used change for parsing the file directly. (Analyze it by knowing its grammar, ...)

4.3.3 To be done, future

At last subsection, we saw that Python retro-inspection is a bad idea. One better will be to integrate `pyreverse`, a free open-source Python reverse-engineering program, which can export datas as XMI file format. PyUt 1.1 can import XMI file format yet.

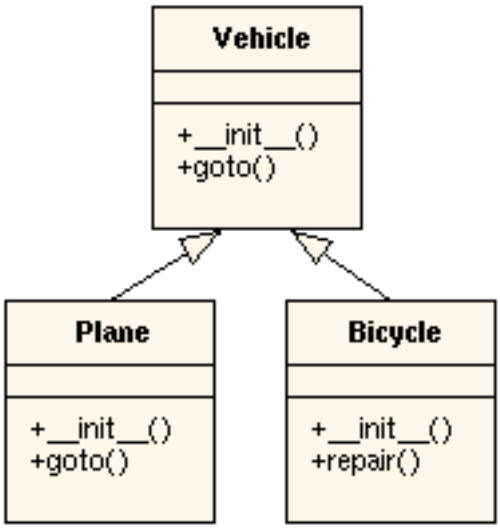


Figure 4: UML class diagram after python code reverse-engineering

4.4 Conclusions

Python roundtrip works now and we can work with it. Only reverse engineering could be improved to reverse agregation and composition links (plus bonus like UMLNotes and others).

5 Java generation plugin

This section describes the IoJava plugin (located in the `plugins` directory). That plugins generates the `*.java` files from the current diagram.

5.1 Example

The following example shows the Java code generation given for a special *PyUt* classes diagram (see fig. 5).

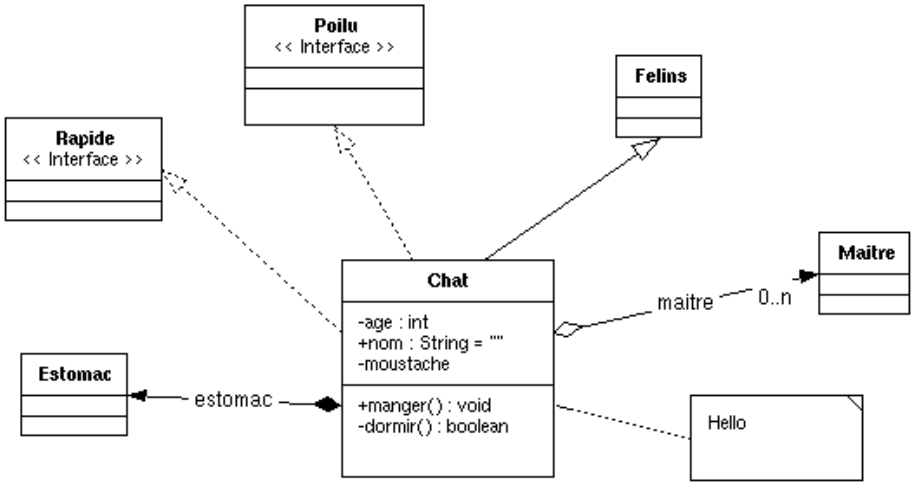


Figure 5: Example of a diagram generation

All the differents classes are generated in a different file. In this example, we'll only look at the class *Chat*. Some comments generated have been deleted in this example to see only the java instructions.

```
public class Chat extends Felins implements Poilu, Rapide {

    // ------
    // Fields
    // ------

    private int age;
    public String nom = "";
    private  moustache; // Warning: no type
    private Maitre maitre[];
    private Estomac estomac;

    // -------
    // Methods
    // -------

    public void manger(int croquette) {
```

```
    }

    private boolean dormir(int duree , String lieu) {
    }

}
```

5.2 Current status

The python code generation works well with good diagrams. This means that the diagram has to respect some conventions. An interface must be declared through the stereotype field of a normal class with the keyword '`Interface`'.

When a diagram contains errors, some comments are directly added in the generated code, but the user doesn't see anything during the process.

5.3 How it works

When your diagram is correct, use the menu `File`→`Export`→`Java` for the generation of code. A dialog box asks you in which directory you would like to have your files. Choose one and click ok.

It is possible that there are some warnings in the generated files. You should look in your files and search the comments '`// Warning`'.

5.4 To be done, future

For the acutal status of *PyUt*, there are not a lot of things to change or add in that plugin.

Perhaps we could add some messages for the user during the generation, like:

```
3 warnings in Chat.java
```

In the same way we could add warings when a class has more than one inheritance relation.

5.5 Conclusions

The plugin for Java Generation works well, but we should add some comments for the user during the process.

6 IoJavaReverse, reverse engineering Java

This section describes the IoJavaReverse plugin (located in the 'Plugins' directory). This plugin does reverse engineering for Java language.

6.1 Java reverse engineering example

Here is a good example of Java reverse engineering :

```
/**
 * Nom       : testReverse.java
 * But       : Fichier de test pour le reverse JAVA de Pyut
 * Auteur    : C.Dutoit - dutoitc@hotmail.com
 * Date      : 5 juin 2002
 */

import java.util.*;
import java.awt.*;
import java.awt.event.*;
import java.awt.image.ImageObserver;

/**
 * Commentaire de classe, l1
 * et ligne 2
 */
class testReverse extends classExtend implements interfaceImplement1,
                                                 interfaceImplement2 {
    // Commentaire 1
    private int intPrive = 8;

    // Commentaire 2
    public double doublePublic=12;

    protected float floatProtected = 16;


    /**
     * Constructeur
     */
    public testReverse(boolean reseau, Ping pong) {
        if(pong.getComm() == null)
            System.out.println("mode réseau");

        // l'espace de jeu
        score = new Score();
        jeu =new Jeu(reseau, pong.getParam(), pong.getComm(), score);
        chat = new Chat(pong.getComm());

        // ajout d'un ecouteur de fenetre
        addWindowListener (new FenetreEcouteur());
    }//testReverse
}// testReverse
```

Figure 6: Java code to reverse 1/2

6.2 Java reverse engineering

6.2.1 Current status

The Java reverse engineering is currently working not so bad. It does reverse classes, classes methods and parameters, classes inheritance and implementation. It does not implement

```
}// testReverse

//--------------------------------------------------------------------------

/**
 * autre classe
 */
class classeTest2
{
    // Autre commentaire
    private monType mavar1;
    public monType2 mavar2 = 18;
    protected monType3 mavar3=22;


    public boolean fonctionBool()
    {
        //...
    }// fonctionBool

    public classeTest2(int x){
        //...
        /*..*/
        /*lkfajélsdkjdfléj
         */
    }

    public Param getParam(double a, int x, float y = 3)
    {
        return param;
    }
```

Figure 7: Java code to reverse 2/2

internal classes reverse-engineering (not enough time to implement that and too many questions on how to draw them) and it does not support methods with parameters like "float a=cos(3.19)".

6.2.2 How it works

The Java reverse engineering plugin asks the user for a file to reverse, then it reads it into a list and parses it by calling specialized methods, recursively.

6.2.3 To be done, future

This has to be done in the future :

- Reverse Documentation in notes, class commentaries
- Reverse methods parameters like "float a cos(3.18)".
- Reverse internal classes
- Reverse many files at one time

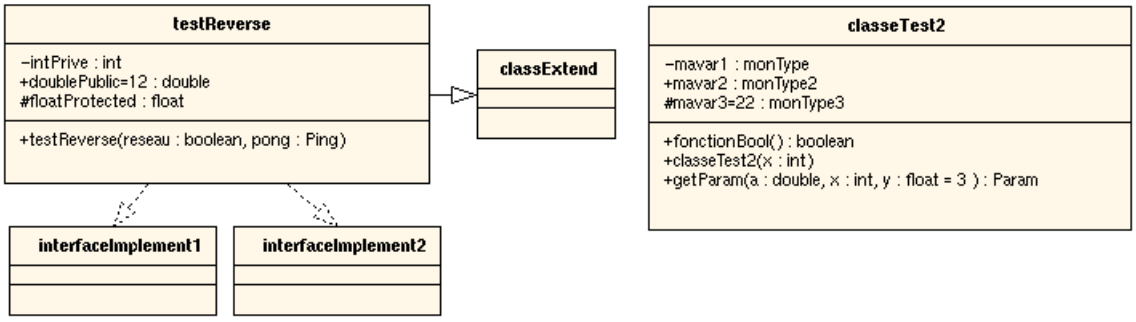


Figure 8: Reversed Java code

6.3 Conclusions

This plugin, Java reverse engineering, is a full-working plugin. It does enough of reverse engineering for the common working. The reverse engineering is done by parsing manually the source file.

7 Internationalization - i18n

This section describes how you can translate *PyUt* in your own language, and what to do when you add some new code with strings that will have to be translated.

7.1 Adding new code

If you add some code and want to make it translatable, you have to convert each string like this:

`"Hello"` → `_("Hello")`

The char '`_`' is a function defined in the gettext package. You don't have to import the package for using '`_`' because it is installed in the globals builtins in `lang.py`. For further informations, please refer to the python documentation, chapter about gettext.

If you add a new file `*.py` which contains translatables strings, you have to refer it in the following Makefile:

`../pyut/src/i18n/Makefile`

Just add the name of your new files at the end of the files list, on the line begining with:

```
$(python) i18n/pygettext.py
```

7.2 Creating your own translation

The function '`_`' will search in a dictionary if there is an existing translation for the given string and will return it. If not found, it will return the original string. There is a dictionary for each different language. Now we will see how to create a dictionary.

7.2.1 Extracting the strings

For creating this dictionary, you'll first have to extract all the strings from all the differents sources, merging them into a file called `Pyut.pot`. A tool called `pygettext.py` will do this for us. In the directory `../pyut/src/i18n/`, you can use the Makefile to do the extraction. Just type the following command under Linux or Cygwin:

```
make
```

Now you have a `Pyut.pot` file in `../pyut/src/` which contains all the strings of the dictionary that you can translate. Do not edit this file.

7.2.2 Translation of the strings

Each language has his own directory. Just make a new directory for your langage in '`src`', called for example '`fr`' for french. In your new directory, you have to create an other new one called '`LC_MESSAGES`'. Then you can copy the Makefile which is in the '`en`' in your own directory. This will be usefull for compiling the dictionary.

Copy the `Pyut.pot` file in your directory and rename it to `Pyut.po`. Edit this file and complete all the `msgstr` lines. See the french translation for example. If you leave a string empty, the original string will be used.

After that, you have to convert the `Pyut.po` file into a `Pyut.mo` with the `msgfmt.py` program. This time you can use the Makefile in your directory like this:

```
make mo
```

Your dictionary is ready to be used. Now jump into section 7.3 for the integration of your translation files into Pyut.

7.2.3 Updating an old dictionary

If you just have to update an existing dictionary with the new strings from Pyut, create the `Pyut.pot` file like we saw in the section 7.2.1

Now, merge it with your old `Pyut.po` file in your language directory. For this, go into your language directory and use this command:

```
make merge
```

Edit the new `Pyut.po` file, and find all the strings '`fuzzy`'. This is a comment added by the program `msgmerge` each time it has written a translation for you. Delete the comment line and correct the translation.

Then you can translate all the empty lines. Don't forget to make the `Pyut.mo` file when the translation is over (see section 7.2.2).

7.3 Use your translation file with Pyut

You just have to edit the file `lang.py` and add your language to the Python dictionary object `LANGUAGES`.

```
LANGUAGES = {DEFAULT_LANG: "English", "fr": "Francais", "..": ".."}
```

Now run Pyut, and changed the language to yours, exit, and restart.

8 Design Patterns

Design patterns are code templates proposing good solutions to some of the most common programming problems. They are the basic building blocks of computer engineering. Knowing them is the key of a simple and efficient communication between engineers, and a step towards well structured code.

We will now present some patterns used in *PyUt*.

8.1 Flyweight

This pattern has been described in the first group project about *PyUt*, last year. We provide it here one more time, exactly as it was :

> A Flyweight is a design pattern useful when a program manipulates a lot of little objects. These objects can be turned to Flyweights if their inner-data is common to all the instances. All you have to do is have a pool of instances, and manage it. If an instance is needed, first look in the pool for the needed one. If it's not yet existing, instantiate it, put it in the pool and return it. This is possible if the state of the object is unmutable, thus the `FlyweightString` derives from `PyutUnmutableObject`.
>
> To implement it, we use a functionnality of Python 2.2 which is the `__new__` operator. It lets us redefine what happens at the time of an object's creation. We use a weak references dictionnary to store the instanciated objects. This one is a special dictionnary that doesn't increment the reference count of an object, letting it be destroyed and removed from the dictionnary when no other references exist.
>
> The `FlyweightString` is the base of these classes :
>
> **PyutType** : type of a parameter or field
>
> **PyutModifier** : modifier of a method or parameter
>
> **PyutStereotype** : stereotype of a UML object
>
> **PyutVisibility** : visibility of a parameter or field

8.2 Singleton

This pattern has been described in the first group project about *PyUt*, last year. We provide it here one more time, exactly as it was :

> A Singleton is a design pattern ensuring that only one instance of the singleton class exists in the program. This is made by redefining the `__new__` operator. The first time the constructor is called, an instance is created and stored as a class field of the singleton. Next calls to the constructor simply return the class field, which is the only instance.
>
> This is useful for objects that must be shared accross the program, like in our case :

OglLinkFactory : factory for OglLinks (cf{conception.ogl.links} (voir avec CD))

PluginManager : class that loads the plugins (cf{conception.plugins} (voir avec CD))

Mediator : class that controls the interactions between the application's components (cf{conception.interactions} (voir avec CD))

8.3 Mediator

This pattern has been described in the first group project about *PyUt*, last year. We provide it here one more time, exactly as it was :

> All interactions between the graphical interface (GUI) parts and the data structure are made by a mediator (Mediator Pattern). This one knows (store references) all the parts of the program. For example, if the user clicks on a tool of the UML toolbar, the toolbar calls the mediator with the selected action as argument. The mediator is responsible to do the job, or delegate it to the right recipient. The advantage of this pattern is that it separates the parts of the application, reducing coupling between them. If we choose to remove the toolbar and put its buttons everywhere else, we would just have to adapt the mediator. Which is in fact what we did when we decided to remove the mobile toolbar to put its button in the main toolbar.

8.4 Observer

This pattern uses two types of actors : an observer and an observable. The observers want to be notified when a change occures in the observable. They register themselves to the observable, and this one will notify them when its state changes.

We use this pattern to let the `OglObjects` notify their links when their size is changed. Once notified, the links can redraw themselves correctly.

8.5 Factory

A factory is an object that will construct another object for you, according to the parameters you give it. For example, our `OglLinkFactory` will instantiate and return you an `OglLink` child, based on the type of link that you ask for. You don't need to know anything about the class you need, just ask the factory and it'll give you the good one.

8.6 Template Method

This one is used by the plugins architecture. The `PyutPlugin` class is the father of all plugins. It defines the skeleton of an algorithm, letting the children implement the details of it.

Let's take the `doExport` method as an example :

```
def doExport(self):
    # if this plugin can export
    if self.getOutputFormat() != None:
        # set user options for export
        if not self.setExportOptions():
            return None

        # write the file
        self.write(self.__oglObjects)
```

Now, imagine that we have created a text export plugin, derived from `PyutPlugin`. Calling the `doExport` (inherited) method will result in :

- call of the `getOutputFormat` of your text plugin
- call of the `setExportOptions` of your text plugin
- call of the `write` of your text plugin

Thus, you have redefined three methods, and *PyUt* just has to call `doExport` to run the whole export process :

- if the plugin can export
  - set the export options (interactive gui process)
  - if the user cancels now
    - stop the export
  - write the file

The algorithm is really in `PyutPlugin`, but the details of the implementation are delegated to the childrens.

9 Refactoring

Refactoring consist to rework source code in order to make it more clean or to adapt it to a different model, the whole process making an improvement to the global structure.

In *PyUt*, some modifications have been done in the modelisation with the objective to simlpify and make the program more extensible (for functionality add-on, for example).

9.1 OGL part

The need of *refactoring* has just came up with the graphical objects extensions (note, use case, ...). In the first version of *PyUt*, a diagram was designed to contain only class diagrams. So we only had a class named `OglClass` which was inherited from `wxRectangleShape`. When the time came to create the note graphical object, the problem was the code redundance. Almost all code in `OglNote` was the same that in `OglClass`. This was the sign of a bad design.

The first design (*PyUt* 1) of OGL data structures (without the links) is presented in figure 9. As you can see, it just consists to a class for an UML class object.

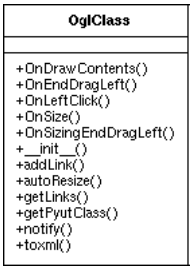


Figure 9: OGL design before refactoring

As shown by this diagram, there is no structure for a simple extension. The main new design was to define a common base for all graphical objects. The new design of OGL part is shown in figure 10.

As you can see, all global OGL operations like links management, data layer link and shape management have been introduced in a base class called `OglObject`. This base class inherits from `wxRectangleShape` to make resizing and link management easier.

Now all OGL objects like a note may inherit form `OglObject` and the only work you'll have to do is to tell to your new object how it should be drawn.

9.2 Data part

In the same idea used in the OGL part, data part has also been modified to improve extensibility. The first design of *PyUt* data layer was a base class called `PyutObject`. This is a good design. Now the problem is that in data layer, we also keep the links. In the same way seen in graphical layer, the only object that was linked in the first version of *PyUt* was `PyutClass`. Now, with the introduction of notes, actors and use cases, it makes three more classes to be

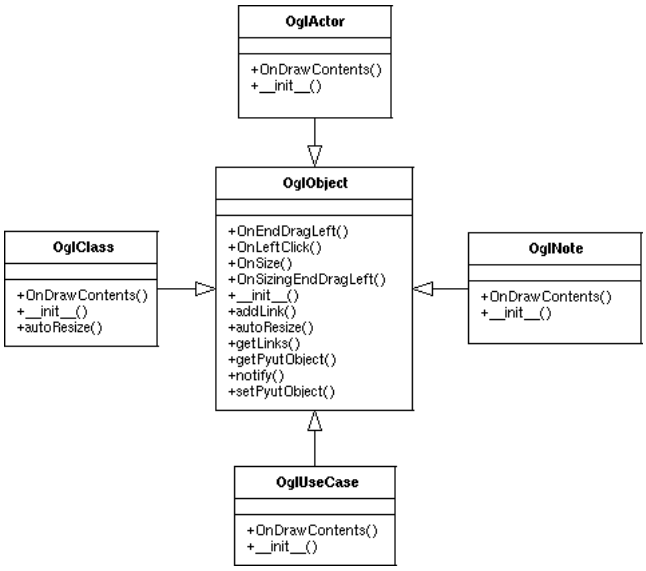


Figure 10: OGL design after refactoring

linked to other objects. The new data layer design for this version is shown in figure 11 (only intersting part).

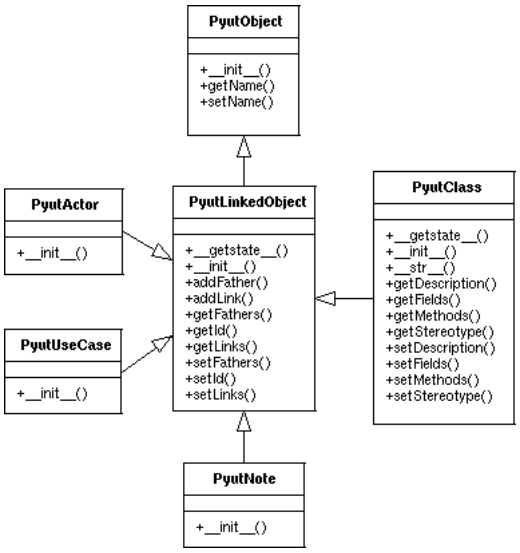


Figure 11: Data layer after refactoring

As you can see, a new class named `PyutLinkedObject` has been defined and handle all kind of links. Now if you want to make an object that may be linked, just inherit this class to get this support.

A second modification is the attribution of an ID to the `PyutLinkedObject` in order to make possible to have more than two objects with the same name, feature that where not possible in the first version. Each time an linked object is instanciated, an ID is affected to this object. A link do not use anymore the name to know which object is connected to another but the identification number that is affected to all linkable object.

10 Glossary

Class diagram notebook : Situated on the right of *PyUt*'s main screen. Handle all opened UML diagrams.

Code generation : Ability for a program to generate a source code from a model, a diagram, class diagram in our case.

Plugin : Little program which can be installed as complement to a program, which enhance its capabilities.

Project tree : Situated on the left of *PyUt*'s main screen. Handle all opened files.

pydoc : As JavaDoc for Java, pydoc is an auto-documentation system.

Reverse engineering : Ability for a program to read a source code and find/display its structure, for our program as class diagram.

Roundtrip : Ability to generate code and reversing it.

wxTreeCtrl : A standard wxPython treeview control.