PyUt
Python UML Tool
Conception

Team Burgbacher
EI5, eivd

27th February 2002

Contents

Abstract

PyUt is a little graphical UML editor featuring editing, printing and export for UML class diagrams. This document presents the conception and possible future developments.

1 General Information

Project Name PyUt - Python UML Tool

Reference Web Site *http://pyut.sf.net*

Source files Available through CVS access at sourceforge

Professor J. Zuber <*jzuber@aicos.com*>

Team Members

- Laurent Burgbacher <*lb@alawa.ch*>
- Nicolas Dubois <*n_dub@altavista.com*>
- Cédric Dutoit <*dutoitc@vtxnet.ch*>
- Nabil Hamadi <*hamadi12@yahoo.fr*>
- Dève Roux <*droux@eivd.ch*>
- Philippe Waelti <*pwaelti@urbanet.ch*>

2 Conception

2.1 Data structure

We've chosen to develop a multi-layer structure, to ease code reutilization. The lowest layer is called *data layer*. The *graphical layer* comes on top of it.

The data layer is a class hierarchy, having `PyutObject` or `PyutUnmutableObject` as parent. There is absolutely no graphical considerations at this level. Two things are stored in this structure :

Classes : Classes descriptions, with their
- fields
- methods
- stereotype
- parents

Links : All others non-hierarchical links, namely
- aggregation
- composition
- association

On top of this comes the graphical layer. Mainly composed of two objects (`OglClass` and `OglLink`), knowing how to draw themselves in the diagram. Each of them contains an object of the data layer, and is responsible of its drawing. Moreover, they contain the graphical information needed for this job (size, position). See the annexes for a class diagram.

Most of the classes in the data layer are very simple. Indeed, they only store some data, and do quite no manipulation to it. They heavilly use Python's list type. We will present the important classes now.

2.1.1 PyutObject

Each of our UML objects has a name. Starting from this deduction, we've chosen to have a base class giving us the functionnality to store the name and get it back, which is `PyutObject`.

2.1.2 PyutUnMutableObject

Some UML objects contains strings that could be shared between them, like the type of a parameter. In a big project, imagine the number of parameters or fields being of type `integer`. In order to prevent the duplication of these strings in memory, we created another base object, the `PyutUnmutableObject`. The difference with `PyutObject` is that there's no `setName` method. The name is fixed at the time of instanciation. See the next section for the sharing method.

2.1.3 FlyweightString

A Flyweight[Coo00] 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

2.1.4 Singleton

A Singleton[Coo00] 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 section 2.2.2)

PluginManager : class that loads the plugins (cf section 2.5 on page 9)

Mediator : class that controls the interactions between the application's components (cf section 2.3 on the following page)

2.2 OGL

OGL (Object Graphics Library) is a library included in wxPython. It offers the basic blocks needed to do vectorial drawings. We mainly used the `wxRectangleShape` (for the UML classes) and `wxLineShape` (for the UML links).

2.2.1 UML Classes

A UML class (`OglClass` in PyUt) is derived from `wxRectangleShape`. This class offers some features like graphical resizing (edge dragging), move (drag and drop) and custom content drawing (by overloading its `OnDrawContent` method). A UML class must also know its associated UML links, to be able to notify them when a move occurs (Observer Pattern[Coo00]).

A UML class also knows its XML representation. This let us save it very easily, by calling the `toXML` method, grouping the results in a root node object and save it to the disk (cf section 2.4 on page 6).

2.2.2 UML Links

A UML diagram is not only composed by classes but also of links between those different classes. UML introduces these main links :

- Inheritance
- Implementation
- Association
- Composition
- Aggregation

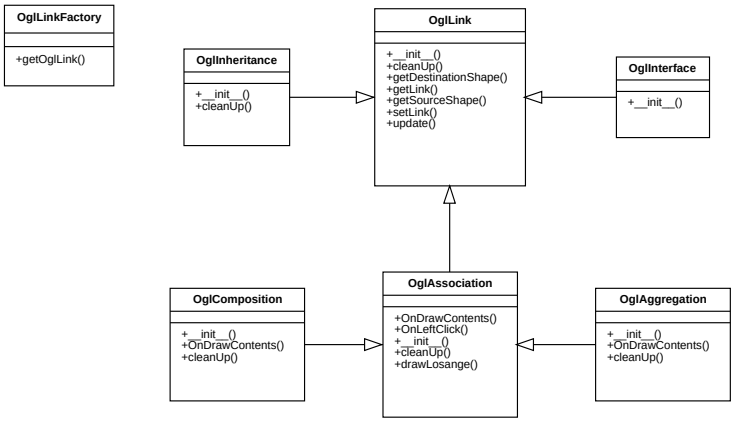

Figure 1: UML Diagram for OGL links

In the OGL part of PyUt, you can find the definition of the different classes that are used to represent all of those links. A graphical link is associated with a corresponding `PyutLink`. The UML diagram of the conception of OGL links follows in figure 2.2.2.

Here are the detailed conception for each class :

OglLink : This classes is the base for all other link. It's responsible for position management and reference keeping on the associated PyutLink.

OglInheritance : Implements the inheritance links. The only thing to add to the basic OglLink is the empty arrow.

OglInterface : Implements the implementation links, this link must be drawn with a dashed line and an empty arrow.

OglAssociation : Implements an association link that must provide the possibilities of label attachment for cardinalities and meaning of link. Association link has no arrows.

OglComposition : Implements composition links, they are specialized association link because we just have to append an arrow to the end of the line and a filled diamond on its beginning.

OglAggregation : Implements aggregation links, just identical to composition links except for the base diamond which is empty.

To create an OGL link, you have to use the factory called `OglLinkFactory`. The conception of this class is build over the *factory pattern*[Coo00]. A factory is a singleton. As a client, you just ask to the factory a kind of product (OGL links) and the factory build it for you. Then the factory returns the built object to you.

2.3 Interactions

All interactions between the graphical interface (GUI) parts and the data structure are made by a mediator (Mediator Pattern[Coo00]). 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.

2.4 XML

XML was chosen as the main file format for multiple reasons :

- easy to undertake
- portable
- human readable

To gain some disk space, XML being a "fat" format, we compress it using the zlib (gzip algorithm, in the standard Python library).

2.4.1 Saving / Loading

The save process is rather simple :

- a root element is created
- for each oglObject in the diagram:
  - call oglObject.toXML() (this is a recursive call)
  - this returns a new node representing the oglObject
  - append this new node as a child of the root element
- save the root element to disk (python library)

The load process is a little more complicated :

- load the xml file
- parse it (python library)
- for each node in the parsed tree:
  - instantiate the corresponding object
  - add it to the diagram
  - store it in a dictionnary
- for each item in the dictionnary
  - restore the links

Since the links are stored by the name of the destination, all classes must be instanciated before we can recreate the links.

2.4.2 XML format

A diagram is represented by the tag `Pyut`, with contains :

GraphicClass

- Attributes
  - **width** : width of a class in pixels
  - **height** : height of a class in pixels
  - **x** : x position in pixels
  - **y** : y position in pixels
- Contains
  - 0..n `Class`

Class

- Attributes
  - **name** : name of the class
  - **stereotype** : stereotype of the class
- Contains
  - 0..n `Method`
  - 0..n `Field`
  - 0..n `Father`

Method

- Attributes
  - **name** : name of the attribute
  - **visibility** : visibility of the attribute
- Contains
  - 0..n `Modifier`
  - 0..n `Param`
  - 0..1 `Return`

Param

- Attributes
  - **name** : name of the param
  - **type** : type of the param
  - **defaultValue** : default value

Modifier

- Attributes
  - **name** : name of the modifier

Return

- Attributes
  type : type of the return value

Field

- Attributes
  visibility : visibility of the field
- Contains
  - 0..n `Param`

Father

- Attributes
  name : name of the father

Link

- Attributes
  bidir : 1 if the link is bidirectionnal (not used yet)
  cardDestination : cardinality of the destination
  cardSrc : cardinality of the source
  destination : name of the destination class
  name : name of the link
  type : type of the link

An example being better than a lot of theory, here's a little PyUt xml file :

```
<?xml version="1.0" ?>
<Pyut>
    <GraphicClass height="98" width="96" x="120" y="72">
        <Class name="Sample" stereotype="abstract">
            <Method name="method1" visibility="+">
                <Return type="int"/>
                <Param name="value" type="int"/>
            </Method>
            <Field visibility="-">
                <Param defaultValue="0" name="field1" type="int"/>
            </Field>
            <Father name="Object"/>
        </Class>
    </GraphicClass>
</Pyut>
```

2.5 Plugins

To allow a simple plugin mecanism, we offer a `PyutPlugin` class that must be derived and put in the `plugins` directory. At runtime, the `PluginManager` will scan this directory, and load the plugins. A template can be found in the `plugins` directory (`PluginTemplate.py`).

Each plugin must overload some informative methods (returning the name, author, input/output format of the plugin) and one or two of the `read` and `write` methods. Two template methods (`doImport` and `doExport`, see Template Method Pattern[Coo00]) initialize the file and data structures that `read` and `write` will receive as argument. They also call the `setImportOptions` or `setExportOptions` methods, that can be used to ask the user for options regarding the plugin. This is used in our XML export plugin, to ask the user if he wants a pretty xml file (indented) or not.

In `AppFrame` (the main PyUt frame), the `OnImport` and `OnExport` methods just ask the user for a filename, and call the plugin to do the job. The import and export menus and submenus are dynamically generated from the `PluginManager` information at runtime.

2.6 Introspection

As a sample UML class diagram, we decided to let PyUt create its class diagram alone. This is done by self introspection. At runtime, when the user selects *Add Pyut hierarchy* from the *Edit* menu, the program will search for the classes given in a list (in the file `PyutDataClasses.py`), find the methods' names and extract their parameters and fathers. Then, it will create the OGL classes and add them to the diagram.

Since Python is dynamically typed, we can't get the associations by self introspection, thus the only given link is paternity.

3 Development State

3.1 Known bugs

- Name clash in XML export when two classes have the same name. This causes the links to be badly instanciated.
- UML standard is not respected regarding the links
- When saving or exporting, if a name is entered without extension, the user is not prompted for overwriting issues.
- Accented letters don't get saved correctly, because the xml writer doesn't support the encoding attribute.

3.2 Not implemented

- Links can't be reorganized yet.
- UML Notes are missing
- Links from one class to the same (links to self) are not implemented

3.3 Extending

Since PyUt is very modular (GUI is separated from the data, good plugins support...), it will be easy to extend it. Reverse engineering could be added, and would not be too hard for a language like Python. Just see the self introspection example (cf section 2.6 on the preceding page) which really is a reverse engineering of PyUt's classes. Code generation could be added with an export plugin. And the code itself could be stored in classes derived from `PyutMethod`. Code documentation could also be added, with just a new textbox in the edit dialogs.

References

[Coo00] James W. Cooper. *Java Design Patterns*. Addison-Wesley, 2000.