# TexdocGraphic

Tommaso Previero
<t.previero@gmail.com>

May 9, 2010

# Contents

# Chapter 1

# GUI - Graphical User Interface explaination

## 1.1 Why a TEXdoc GUI?

This program is born to be a simply way for search into the documentation of LATEX. Many users have really no idea of what is a shell and how it works. The `texdoc` command that is available once installed the system into your computer is really powerful, but if someone doesn't knows how to access to it became useless. That's the why of this program!

Building an interface that should help the users to use this command showing them, in a way that they know, all the features offerted by `texdoc`.

The interface is written in java, so anyone can use it in his own system, no matter wich. This document was created to help users to find what they're searching about this program, i hope that will be useful for someone.

## 1.2 The graphic and its components

The first thing, and maybe the most important to talk about, is the way the program presents itself to a user. The GUI can be found in the picture 1.1, we go on in this chapter referring to this picture. The components are

1. a text field where the user have to insert the word to search into the documentation
2. the button to push for start the search of the inserted word
3. where will be stored the results of the search (only if checkbox 'List' is selected)
4. the modes available for search a document
5. a button that will open the selected element of the results list

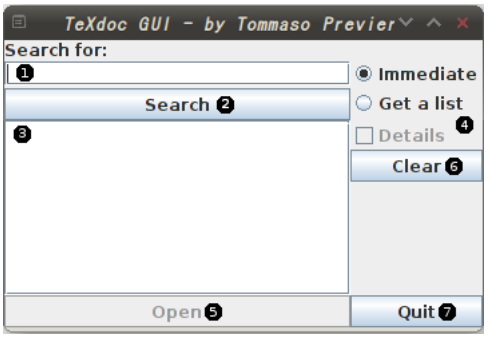


Figure 1.1: The GUI

6. a button that once clicked will clear the GUI

7. a button for exit the application

The esier way to understand how the GUI of this program works is performing a search. The first thing to do when using this program is always (if we have something to search, of course) inserting what we are going to search into the component (1).

Once we have insert the word to search we have to choose the way we want to search for that thing, and two possibilities are at our disposal, both can be set using the component (4). The first is to choose the *Immediate* mode, the second choosing the *List* mode. Using the first the program will not show a list of results, it will just open the first document that will corresponds to the given word in component (1), anyway that search is always optimized, so you'll never get a document that doesn't satisfies some precise parameters. Otherwise you can use the *List* mode, using this you'll get a list of all files that, in some way, satisfies the requests. In this case a list will appears into the component (3) and you'll be able to choose which file want to open.

When the *List* mode is selected another option will became available, the *Details*. Selecting this will allow you to get more information for the documents into the results list. The information availables will be the points assigned to the document by `texdoc` and othe useful information where are specified. To see these information just stay with the mouse icon over the element into the results list and wait util a tooltip will be shown.

Once decided between the two mode, pushing the component (2) will start the search. If the program don't find any document for the given word or meet some problem during the process, a message box is shown with a specified error message. Otherwise, if the search succesful, depending on which mode you have selected, the program will open the requested document or fill the results list. If the mode *List* was selected now you have to choose which document want to open, to open a file contained in the

results list just double-click it or singol click and then push the component (5).

Now you can start a new search pushing on component (6) erasing all the the GUI and restarting, or simply changing the word into component (1) (of course even the modes in component (4)) and then research.

You can always keep open your documents opened before even if a new search have begin. When you have done can close the program pushing the component (7). None of the opened documents will be closed at the exit of the program.

## 1.3 Java implementation

To learn more about the java implementation of the GUI you can see the *sources/TexdocGraphic.java* file to know how it's really implemented or the *doc/TexdocGraphic.html* to see the documentation made with `javadoc`. For other classes can read also section 2.1.

# Chapter 2

# Program Logic

## 2.1 The java classes

How we've already said the program was made in java, so behind the GUI there are a few classes that realize all the functionality we need. Of course in an OOP there's no way to do everything and do it well without interfaces and, in our case, enumeration.

The interfaces are

- Command
- CommandMode
- CommandOption
- CommandRuler
- CommandOutputInterpreter

There also a few enumeration that are

- TexdocMode
- TexdocRule

And last, but not last, our classes

- TexdocCommand
- TexdocOptions
- OutputInterpreter
- TexdocOutputInterpreter

There are also other few classes that helps out to do some stuff like throwing errors of some type and of course the GUI class already mentioned before. By the way we need only these classes, enumeration and interfaces to realize all we want. In the picture 2.1 you can see how them are linked each other in UML (Unified Modeling Language). It all starts with a `Command`, it's the center of all our speculation. In a shell a command would be something like

```
commandName [-param1 -param2 ...  -paramN] [text]
```

In this case we have a command that is `commandName`, a list of parameters, and some text to pass to the command. Understanding this you'll probably have understand how all the program work. In fact there is an interface `Command` that describes all the method a shell command should have. It must have a way to add or remove parameters, to set the text passed to the command, an of course to run itself.

Adding or removing parameters will be simply using a `CommandOption` that is only a list of parameters, but only those that are allowed an so present into a `CommandMode`, that stores all the possibles parameters for a `Command`.

Every of the interfaces mentioned have the specified class for the command `texdoc` that realize them. For the interface `Command` there's `TexdocCommand`, for `CommandOption` there's `TexdocOptions` and for `CommandMode`, the enum that stores all the possibles parameters for the command `texdoc`, there is `TexdocMode`.

Using these simple classes and an enum we're able to run a `texdoc` command, but what if the the `Command` that we're going to run isn't consistent? Like when tryin' to execute the command

```
texdoc -m -M -I nosense
```

the answer are consistent looking only the first parameter, but what if these answer arrive to our program and were interpreted in the wrong way? Better avoid this. Here come the why of `CommandRuler` interface and `TexdocRule` class.

A `CommandRuler` is an entity that contains all the possibles combination of parameters for a `Command`, so we can establish if a `Command` is consistent using a set of possibles `CommandOption`. Shift this to the command `texdoc` an you'll have the `TexdocRule` enum, containing all the possibles allowed combination used by the program. There's no worry now for the program to execute some command not in the list, and so not allowed.

Once we have launch the `Command` it will generate some output, so we need something to get this output and elaborate for us. In this case we suppose the `Command` will generate only output and don't need other inputs than parameters already given. Here is the why of our `OutputInterpreter`,

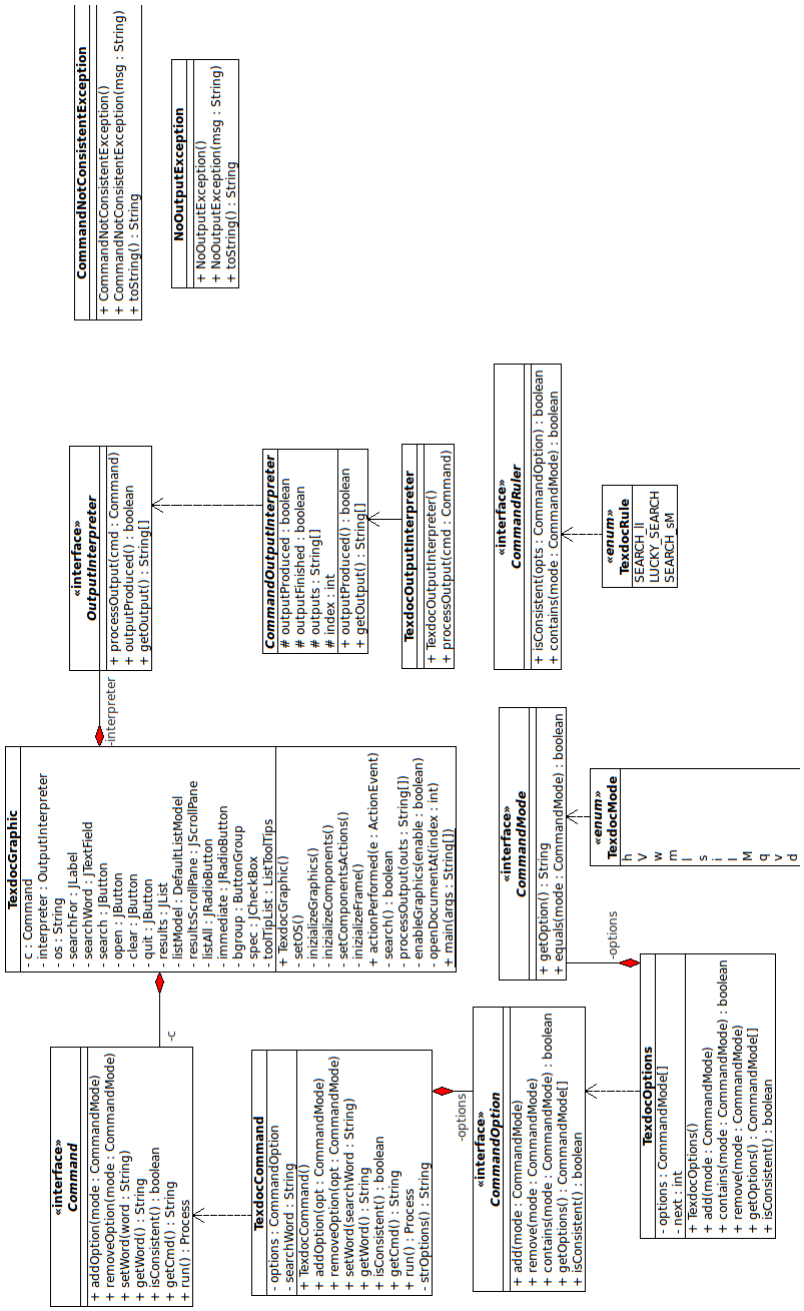


Figure 2.1: The UML

it will take the `Command`, execute it, and process its output. In specific if our `Command` is a `texdoc` command we need some specific management, and so come `TexdocOutputInterpreter`. However each `Command` have something in common, and that's the why of the abstract class `CommandOutputInterpreter`.

This is all we need to run and manage a command for our purposes.

# Chapter 3

# The sources

## 3.1 Organization of the program folders

Here we'll present you how are organized the folders of the program, so you can find what you're looking for easily. The main directory of the project is splitted as follow.

```
/
├── sources
│   ├── Command.java
│   ├── CommandOption.java
│   └── ...
├── compiled
│   ├── Command.class
│   ├── CommandOption.class
│   └── ...
├── dist
│   └── TexdocGraphic.jar
├── doc
│   ├── index.html
│   ├── Command.html
│   ├── CommandOption.html
│   └── ...
├── makeJar
├── makeDoc
├── MANIFEST.MF
└── COPYING
```

There are parts that are quite important to modify or compile by yourself the program and will be treated separately into the next sections, for other not so important will be treated here.

### 3.1.1 The *MANIFEST.MF* and *COPYING* files

The *MANIFEST.MF* file contain the definition of how the program will be launched once have been compressed into a `.jar` file. Here its content.

```
Manifest-Version:  1.0
Created-By:  Tommaso Previero
Main-Class:  TexdocGraphic
```

In this way when command `java` starts executing it knows what to do.

The *COPYING* file instead contains the whole license that have been used to distibute the software. If you are planning to modify the sources and redistribuite the program by yourself you have to copy this file and redistribuite it too.

### 3.1.2 The *compiled* folder

Into this folder will be stored all the compiled class obtained by compiling the `.java` contained into the *sources* folder. The compilation of the sources can be done with a script that you can find well argued at section 3.2.

## 3.2 The *makeJar* file

This script, if launched from a shell, will perform the clean of previously versions, compilation of the sources, and the creation of the `.jar` archive. Here is how it's implemented.

```
echo "Removing all the .class and .jar builded previously..."
rm compiled/*.class
rm dist/TexdocGraphic.jar

echo "Compiling sources..."
javac -sourcepath sources/ -d compiled/ sources/TexdocGraphic.java -target 1.5 -source 5

echo "Creating TexdocGraphic.jar file..."
cd compiled
jar cvfm ../dist/TexdocGraphic.jar ../MANIFEST.MF *.class
cd ..

echo "Setting the privileges for the TexdocGraphic.jar file..."
chmod 755 dist/TexdocGraphic.jar

echo "Done."
```

This script starting removing all the `.class` contained into the *compiled* folder and the `TexdocGraphic.jar` into the *dist* folder.

After that starts compiling all the `.java` contained into the *sources* directory and put the results `.class` files into the *compiled* folder, using some other specific options like the target `VM` for generate specific `.class` for a version and the source that tells the compiler which is the source version of java.

Then starting create the `.jar` file that will be our final program. This process will put the generated file into the *dist* folder, and using the *MANIFEST.MF* file to tells the program how to start once executed.

Once set the privileges for the file `.jar` created the creation of our program is completed. Now you can try your program by executing a

```
java -jar dist/TexdocGraphic.jar
```

from the main folder of the program.

## 3.3 The *makeDoc* file

Another interesting script is the *makeDoc* file. This file can be used to create the documentation for all the classes stored into the *sources* folder. If the java classes that are contained into this directory are commented using the `javadoc` format you can easily create the respective documentation using this script. Here is how it's implemented.

```
echo "Removing the documentation previously builded..."
rm -r doc

echo "Building new documentation..."
javadoc -private -sourcepath sources/ -classpath compiled/ -d doc sources/*.java

echo "Done."
```

As first the script starting removing the folder *doc* that contains all the documentation generated before.

After that begin to build the new documentation using `javadoc`, setting the visibility to private so every private member or method will be displayed into the documentation we're going to create. Telling `javadoc` that the destination folder is *doc* it will place all the thing generated into this directory, and of course it will create it if isn't present (we've just deleted it). We have also to tell the command where to find the `.java` classes and, for better documentation, where to find the `.class` files. So compile the whole project before starting build the documentation!

## 3.4 The *sources* folder

Now that you know a little better how the project is structured and how it works you can try to modify and improve it. All the things you need to begin to customize the project is contained into the *sources* folder. If you are able to program in java language and understand the sources, you will be now able to modify the project. All the sources are (at least i hope) commented in every single operation, so it shouldn't be hard to understand them.

Anyway all the project is under `SVN` even if it's not currently available on internet. So if you want to help in some way please contact me at `<t.previero@gmail.com>`, maybe we can start to distribute this software over internet.