TEXdocGraphic

Tommaso Previero
<t.previero@gmail.com>

September 18, 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. the menu of the application
2. a text field where the user have to insert the word to search into the documentation
3. the modes available for search a document
4. the button to push for start the search of the inserted word
5. a button that once clicked will clear the GUI

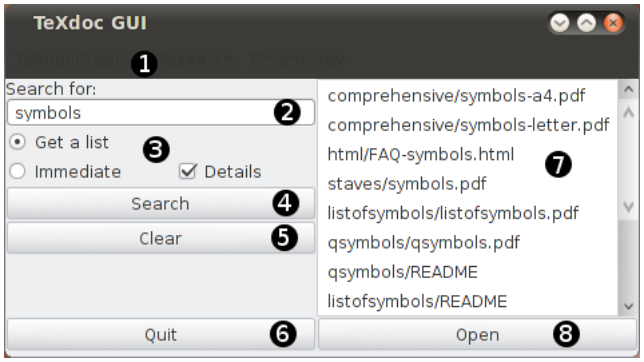


Figure 1.1: The GUI

6. a button for exit the application

7. where will be stored the results of the search (only if checkbox 'List' is selected, otherwise is not shown)

8. a button that will open the selected element of the results list

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 (2).

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 (3). 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 (2), 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 (7) 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 other useful information where are specified. To see these information just stay with the mouse icon over the element into the results list and wait until a tooltip will be shown.

Once decided between the two mode, pushing the component (4) 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 successful, 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 (8).

Now you can start a new search pushing on component (5) erasing all the the GUI and restarting, or simply changing the word into component (2) (of course even the modes in component (3)) 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 (6). 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`. The API of this program can also be found in internet at *http://texdocgraphic.altervista.org/*. For other classes can read also section 2.2.

Java language was choosen so anyone can use this program on his system but in the future the project can be rewrite in some other language. Doing so there will be no need to use the JVM that is a little heavy for some system (if they have it).

Chapter 2

Program Logic

2.1 The logic behind

The logic behind the program is really simple. There is a `Command` that can be executed, a `OutputInterpreter` that will catch the output of the command and make it usable for the program. All the other things are useful only for control the execution of the program and we can even not use them. Of course if we don't use other things the program is not well structured, and it's not good to put thousands rows in a file, so in the next section we'll explain thing a little better.

2.2 The java classes

How we've already said the program is 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

- ICommand
- ICommandMode
- ICommandOption
- ICommandRuler
- ICommandOutputInterpreter

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. Recently added there are also some classes used for show a splashscreen at the starts of the application, some for keep trace of a chronology of the searched files, and others used at the start of the application that will treated separately in the next section.

By the way we need only these classes, enumeration and interfaces to realize all we want. 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 `ICommand` 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 `ICommandOption` that is only a list of parameters, but only those that are allowed an so present into a `ICommandMode`, that stores all the possibles parameters for a `ICommand`.

Every of the interfaces mentioned have a specified class for the command `texdoc` that realize them. For the interface `ICommand` there's `TexdocCommand`, for `ICommandOption` there's `TexdocOptions` and for `ICommandMode`, 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 `ICommand` 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 `ICommandRuler` interface and `TexdocRule` class.

A `ICommandRuler` is an entity that contains all the possibles combination of parameters for a `ICommand`, so we can establish if a `ICommand` is consistent using a set of possibles `ICommandOption`. 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 `ICommand` it will generate some output, so we need something to get this output and elaborate for us. In this case we suppose the `ICommand` will generate only output and don't need other inputs than parameters already given. Here is the why of our `IOutputInterpreter`, it will take the `ICommand`, execute it, and process its output. In specific if our `ICommand` is a `texdoc` command we need some specific management, and so come `TexdocOutputInterpreter`. However each `ICommand` 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.

2.3 Other classes

As mentioned before there are some other classes behind the GUI. For example there is the `SplashScreen` class that it's only used to show a splashscreen at the start of the application while all the components are loaded.

More important are the `IChrono` interface and the `Chrono` class. These two are used to store things inside a directory created by the program into the user's home (the directory is called *.tgchrono*). An `IChrono` object can be used to store many kind of things, everithing that implements the interface `Serializable`. This method is used to store the last searched words and the opened documents by the program and also for other stuffs useful for the program. The `Chrono`, saving thing on a file, will need something that works like a 'writer to' and 'reader from' the specified file. Here comes the `IReader` and the `IWriter` that defines the methods useful for read from and write to a store support. If this support is a file a `Reader` and a `Writer` will be used.

Other two very important classes are the `Starter` and the `TexdocBinChooser`. The first is used to 'rebound' the start of the application. Sometimes happend that at the start of the program for some reason in the environment variable PATH is not defined the path of the `texdoc` command, so the program check its existence and if it's not defined call the `Starter` that will add the correct path and restart the application. The problem is that somewhere we must have the correct path where is the `texdoc` command, and here comes the `TexdocBinChooser`. This class is a simple window that is shown the first time we start the application and ask to us to choose the directory where is stored the `texdoc` command. This information is also stored into a file in the *.tgchrono* directory with the others program's information.

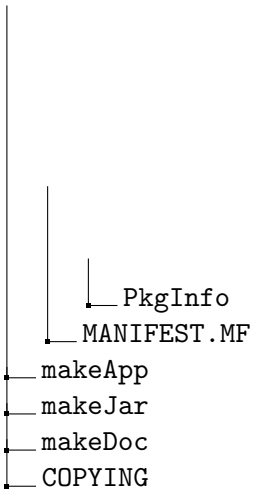
Chapter 3

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 split as follow.

```
/
├── sources
│   ├── ICommand.java
│   ├── ICommandOption.java
│   └── ...
├── compiled
│   ├── Command.class
│   ├── CommandOption.class
│   └── ...
├── dist
│   ├── TeXdocGraphicRXX.app
│   └── TexdocGraphicXX.jar
├── doc
│   ├── index.html
│   ├── Command.html
│   ├── CommandOption.html
│   └── ...
└── res
    ├── images
    │   ├── splash.png
    │   ├── icon.icns
    │   └── ...
    └── Mac
        ├── Info.plist
        └── JavaApplicationStub
```



In the next sections we'll discuss a bit the file and the foders of the program.

3.2 The *res/MANIFEST.MF* and *COPYING* files

This two files are not so important to understand the program. But we need them for tell the application wich class contains tha main method and for the license.

3.2.1 The *res/MANIFEST.MF* file

The *res/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 wich class contains the main method.

3.2.2 The *COPYING* file

The *COPYING* file 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.

3.3 The folders of the application

Here are presented the folders that are available into the root directory of the project and their use.

3.3.1 The *sources* folder

In this folder are stored all the class written in java of the application. 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.

All the project is under `SVN` and you can have your working copy on *http://code.google.com/p/texdocgraphic/*.

3.3.2 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 3.4.1.

3.3.3 The *doc* folder

This folder is used to save the documentation of the `.java` classes stored into the *sources* folder. Each file into the *sources* is commented using the `javadoc` format, so you can create the documentation of the whole program using the *makeDoc* script that is described in 3.4.3.

3.3.4 The *res* folder

Used to store useful and unchanged thing between a revision of the code and another, like images and other stuffs like the things used to create the `.app` and even the *MANIFEST.MF*

3.3.5 The *dist* folder

Folder where the scripts *makeJar* and *makeApp* stores compiled applications. See the 3.4.1 and the 3.4.2 for more.

3.4 The make files

This files are used to compile the program or to create its documentation.

3.4.1 The *makeJar* file

This script, if launched from a shell, will perform the clean of previously versions removing the `.jar` into the *dist* folder and all the `.class` into the *dist* folder, compilation of the sources, and the creation of the new `.jar` archive with the compiled sources. After the creation of the new `.jar` into the *dist* folder this script sets its privileges. If this script is called without any argument if some error occours in the compiling process no output is produced about the problem, it just fails. If you want all the output like the compiler messages and others then call it with the `verbose` argument like

this:

```
./makeJar verbose
```

3.4.2 The *makeApp* file

You can use this script to build a `.app` for a `MacOSX` system. As first step this script will recompile the sources by calling the *makeJar* script. Then it use the `.jar` generated into the *dist* folder to create a `.app`. For doing this we'll need some file into the *res/Mac* folder. When the script have done a folder named *TeXdocGraphicRXX.app* (where 'XX' will be replaced with the current revision of the project) will be available in the *dist* folder. This is an application for a `MacOSX`. If the compiling process fails somewhere the creation of the application fails and no application is created.

3.4.3 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!