## ScriptRepositoryGUI

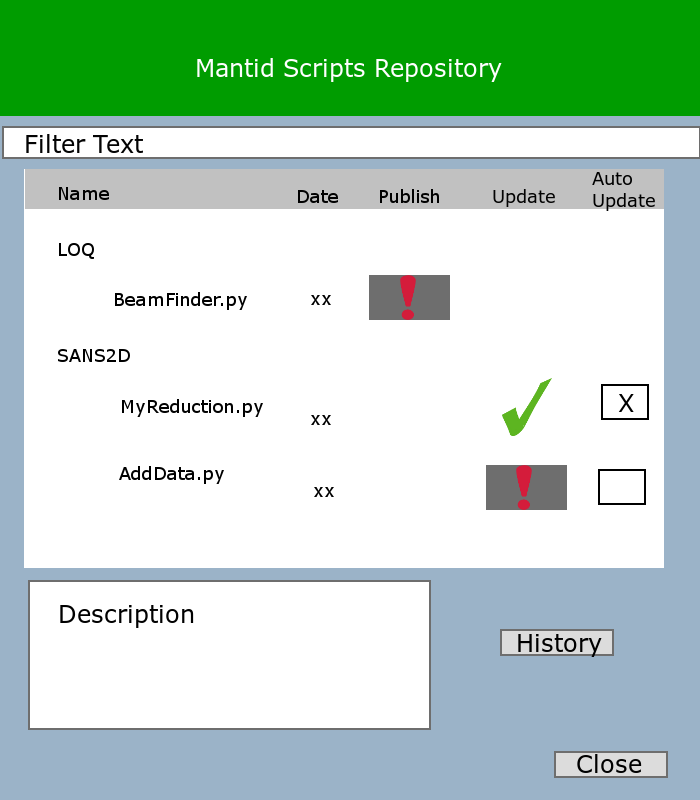


Figure 1 ScriptRepositoryGUI

### UML

#### Diagram For ScriptRepository User interface

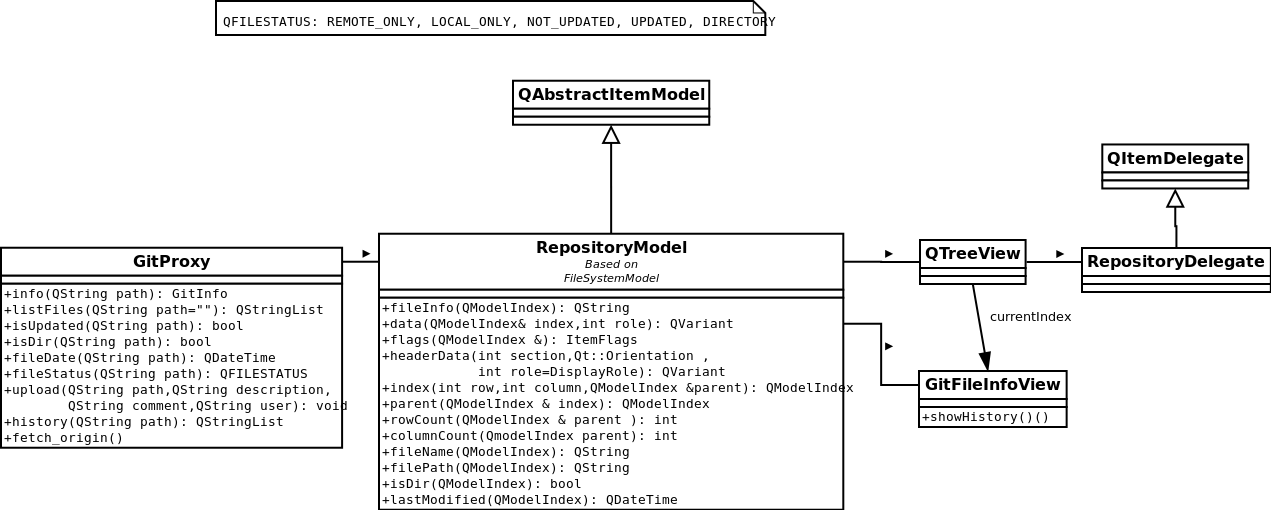


Figure 2ScriptRepositoryGUI UML

#### Diagram For Script Repository Implementation

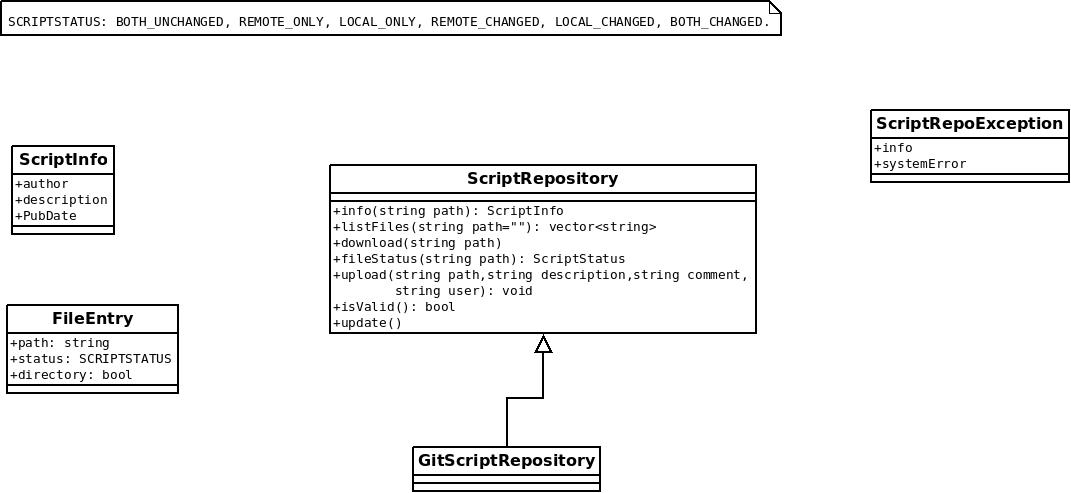


Figure 3ScriptRepository Implmentation UML

### Detailed Design

#### Description of the Implementation

GitScriptRepository implements the interface of ScriptRepository. In what follows, a description of the goal of each method is given, and how to obtain it though the git command line.

##### Update:

Connects to the remote repository, and check for updates. If necessary, it may create the local repository (clone), specially, when first called.

*This operation requires internet connection.*

So, the update method may be separated in two main functionalities:

* Installation

When installing the git repository, the user do not want to have all the files downloaded in his folder, so it is necessary to do a clone of the git repository without checkout:

###### git clone –n [url] [local\_path]

Besides, when using the repository, the user should not see .pyc files created automatically from the python, so, a .ignore file must be present as well.

**OBS: After doing this, git will create an “empty” folder (everything will be inside the .git folder). And a git status will say that you have deleted all the files.**

* Checking up updates

It is necessary to check the remote repository for updates. Git allows to check remotes changes through:

###### git fetch origin

The problem here is that **we can not merge the updates**. The reason is that we do not want to create new files, nor we want to change files of the users if they have not requested to do so. One possibility is to use the command:

###### git reset –soft FETCH\_HEAD

This command, change the master to point to the origin, but it does not change the current index. By doing this,

###### git status

will be able to see that new files are being added from remote, and will be able to see if the file has being remotely changed or locally changed.

##### ListFiles:

List all files that are available at the repository and the local files, marking the information about the current state of the file. The files support the following states:

* BOTH\_UNCHANGED: The local file and the remote file are the same.
* REMOTE\_ONLY: The user has not downloaded this file.
* LOCAL\_ONLY: The user has created this file, but not publish it.
* REMOTE\_CHANGED: The user downloaded the file, but a new version of this file is available.
* LOCAL\_CHANGED: The user downloaded the file and edited it.
* BOTH\_CHANGED: Both Remote and local changes are valid.

The answer here is equivalent to the output of:

###### git status

##### Download:

Create a copy of the file presented at the remote repository inside the working directory. Technically speaking, all the files are already presented inside the folder under the .git folder. So, what download must to is just checkout the file from the master branch to the working directory.

This is equivalent to the following command:

###### git checkout master [path]

The user will see the file inside the local repository.

##### FileStatus:

Return the status of the given file path.

Here it is worth to mention how we will deal with directories.

The directory is BOTH\_UNCHANGED, if all its children are BOTH\_UNCHANGED. It is REMOTE\_ONLY if at least one of its children is REMOTE\_ONLY and those that are direrents are BOTH\_UNCHANGED. The same rule is valid for REMOTE\_CHANGED, LOCAL\_CHANGED, REMOTE\_CHANGED. If there is at least one REMOTE\_CHANGED and one LOCAL\_CHANGED among its children, it will receive BOTH\_CHANGED.

##### Info ( file\_path):

Return information about the file: description, author, date of last change.

Git does not have support on entry description on its own, and besides, the description of the documents should go with them as well. Remembering that the purpose of this repository is to share *MantidScripts* our suggestion is to provide the following rules for the files description:

* Python File: Description is its \_\_doc\_\_() or the group of comments on the very top of the document.
* Python Module: Description is the \_\_doc\_\_() of the module, or the very top of the comments inside the \_\_init\_\_.py file or the documentation of a General Folder.
* General Folder: The description is the content of a file called README with any extension.
* README files: its contents.
* Other files: no description
* **Executables: will we allow executables at the repository?**

##### Upload:

There are a couple of issues related to the upload of the users *MantidScripts:*

1. It requires a read+write transport layer, which require a certificate (ssl) and the grant of the permission to the user.
2. We should not break the agreement with github creating a generic user and certificate for all the mantid users, this would cause also to loose traceability of the changes
3. The current version of libgit2 (library used for the current implementation) does not support write mode for linux through ssh.

A workaround for these issues could be the setup of an intermediate, a web server that could receive the files to upload and them push them to the git repository.

Currently, we propose that the upload should be done with the following parameters:

* Path: that identifies where the file/folder is.
* Description of the document: may be already inside the document (see info).
* Comment for the current version.
* Author: a string that could identify or give us the possibility to identify who changed the document, although, the privacy must be considered.

## ScriptSharingService

The ScriptSharingService is responsible for doing things in the background. Its main purpose are:

* cloning the repository at the installation of Mantid (if necessary)
* Periodically fetching the origin repository
* Update the files automatically, and/or notify users about new versions available.

For the second option, it would be possible to add this service to the Operate System chron table, so to, force the execution of the service in a periodic base.

This service will need to interact with the Mantid user properties, in order to keep information about the list of files that the user wants to be notified about its changes, the list of files that the user wants to download automatically, etc.

# The Upload web service

The uploads of user files and scripts will be done indirectly through a dedicated upload web service. It will be set up a web service that receive a file, and the git path, and some information that are necessary to insert the file inside the git repository.

TODO: WHY THIS IS NECESSARY?

# For future

* Add folder special icons to show up-to-date, not in repository
* Add right-click menu to upload.

# READERS: DO NOT READ AFTER THIS POINT (NOTES FOR THE AUTHOR)

Notes for me: From a Top-Botton approach.

Using a Model-View-Control scheme provided by Qt framework.

http://doc.qt.digia.com/qt/model-view-programming.html

tips:

QStandardItemModel

QTreeView

QStyledItemDelegate <-QAbstractItemDelegate, [QItemDelegate](http://doc.qt.digia.com/qt/qitemdelegate.html)

http://doc.qt.digia.com/qt/itemviews-spinboxdelegate.html

QTreeView -> will show the remote/local files, and custom delegates will be created to handle the information of Not Updated, Not published, and to provide the following actions:

upload, download, update.

Model:

RepositoryModel:

File, Download, Date, published, update status, history, description

http://doc.qt.digia.com/qt/itemviews-simpletreemodel.html

Delegate:

RepositoryDelegate:

column: 0-> Download -> Button or empty widget

column: 1-> File/Folder name -> Default ->line edit

column: 2-> Date of last modified ->QDateTimeEdit

column: 3-> published -> button or emtpy widget

column: 4 -> update status -> button or checked widget

View:

QTreeView