MIND development process

Abstract

This document describes the workflow to develop on MIND technology and tools from GitHub sources. It indicates how to configure Git and the basic commands to be used.

Document History

Date	Author	Version	Change Reason/Description
July 27th, 2012	Valerie Bertin	Draft	First Version
August 3rd, 2012	Valerie Bertin	Draft	New version after Vincent Lorquet's comments
August 6th, 2012	Valerie Bertin	Draft	Next steps paragraph
August 7th, 2012	Valerie Bertin	V1.0	Terminology paragraph

Table of contents

Contents

1 Introduction

This document describes the workflow to follow to contribute to MIND sources. The development process is based on a branching policy.

As MIND technology and core tools sources are now hosted on GitHub, this document describes how to configure Git Software Configuration Management tool and gives the main Git commands to use.

2 Organization of the document

This document is organized as follows:

- First, some references about Git SCM are given.
- Then, MIND on GitHub is presented.
- In a third part, the branching policy to contribute to MIND is explained.
- The fourth part gives the main Git commands to be used to start with MIND on GitHub and Git SCM.
- The fifth part summarizes the workflow.
- Finally, the last part gives next steps.

3 About Git

3.1 Overview

Git is a free and open-source, distributed version control system. It is designed to handle a wide range of project types and sizes, in a fast and efficient way.

Every Git clone is a self-contained repository with complete history and full revision tracking capabilities. Thanks to this feature, many operations can be carried out even if a network access is not available. Branching and merging are fast and easy to do.

3.2 References and documentation

The Git homepage: http://git-scm.com/

The Git reference: http://gitref.org/

The Git community book: http://git-scm.com/book

The Git community book (in French): http://www.alexgirard.com/git-book/index.html

A simple guide for getting started with Git: http://rogerdudler.github.com/git-guide/

An interactive cheat sheet: http://www.ndpsoftware.com/git-cheatsheet.html

EGit, Git for Eclipse users: http://www.eclipse.org/egit/

EGit tutorial: http://unicase.blogspot.fr/2011/01/egit-tutorial-for-beginners.html

4 Terminology

Git uses a lot of dedicated terms and naming conventions. Here is a non-exhaustive list of the main terms often used in the rest of the document.

- ***Repository***: a repository is a complete source database managed under Git.
- ***Clone***: a clone is a copy of a repository. The copy is self-contained with complete history and full revision capabilities.
- ***Fork***: from the user point of view, a fork is similar to a clone. A fork can only be performed on the same server. It is an optimization of the clone mechanism.
- ***Remote*** repository: as Git is a distributed system, several clones of a repository can exist. Thus referring to another clone than the local one is known as referring to a remote repository.
- ***Origin***: when creating a clone of a remote repository, a link from the local clone to the remote repository is created. In this case, the remote repository is called origin. In the Subversion environment, the origin is the central repository.
- ***Upstream***: when developing in a large community, there is often a remote repository which is the reference. Most of the time, developers are not allowed to commit directly in it. Thus they have to fork this remote repository in a developer area and clone it in their local development area. In the Git terminology, this remote reference repository is known as upstream.
- ***Commit***: a commit corresponds to the storage of a change in the local repository.
- ***Patch***: a patch is a set of commits representing a (coherent) set of changes.
- ***Merge***: a merge action consists in applying modifications from a branch to another one (patch application). It is the same meaning as in Subversion.
- ***Rebase***: a rebase action consists in un-doing the local commits, updating the current branch to the latest revision of the reference and re-applying local commits. In other words a rebase action applies local commits on top of the latest revision of the reference (in the local area).
- ***Push***: a push command synchronizes a remote repository with the local repository, i.e. local committed changes are reported on the remote repository. In the Subversion environment, a commit corresponds to the Git sequence of commit + push.
- ***Fetch***: a fetch command retrieves the changes from a remote repository and stores them in a local cache.
- ***Pull***: a pull command retrieves the changes from a remote repository and applies them on the local repository (i.e. fetch + merge).

5 MIND on GitHub

5.1 About GitHub

GitHub (https://github.com) is a web-based hosting service for software development projects that uses the Git SCM tool. GitHub offers both paid plans for private repositories, and free accounts for open-source projects.

5.2 MIND location

MIND technology and core tools are hosted on GitHub under the organization named MIND-Tools (https://github.com/MIND-Tools).

5.3 MIND structure

There are 8 repositories, corresponding to the 8 modules that exist in the SVN repository of the OW2 MIND forge:

- maven (MIND parent pom),
- mind-release (Maven script to generate an integrated release of MIND tools - non-functional in current state),
- mindc (MIND compiler modules),
- minded (MIND IDE - Eclipse plugin),
- mindoc (documentation from ADL - MIND plugin),
- mindtest (testing framework - non-functional in current state),
- sandboxes (developments of each contributor),
- web (web site).

5.4 Creating an account on GitHub

In order to be able to collaborate on MIND, each contributor has to create a GitHub account.

- Go to https://github.com/ and select ***Signup and pricing***.
- Click on ***Create a free account*** button.
- Choose your username and password, give your email address and click on ***Create an account*** button.

Your GitHub account is created.

Send an e-mail with your GitHub username to valerie.bertin@st.com in order to be added as a MIND-Tools organization member.

6 Branching policy

The development process on MIND is based on a branching policy, and relies on a naming convention.

6.1 Objectives and principles

A branching policy is a set of rules, defining operations that must be done to perform versioning and when these operations must be applied.

Branching policy allows one to ensure the permanent coherency and validity of a given Git repository. It follows that such a repository can be used as a basis for any new development or release, leading to an easier development (well-known base for each new development) and an easier back tracking of bugs (bugs that appear in a new development may have been introduced in modified sources).

The life of a branch is the following:

- Branch creation when a new development or a bug fix begins.
- Rebase from master to stay in line with the master (regularly and at least before asking for merge agreement).
- Request for merge agreement.

- Peer code review and update. Remarks from the reviewers are taken into account and/or discussed. Some modifications can happen during the review.
- Delivery on master.

The Git / GitHub commands to be used all along a branch life are given in section 0.

6.2 Naming convention

For MIND contributions, we propose to create a new branch for any new development or bug fix with a conventional name, which will precise the origin and type of the change.

- Bug fix, the name is fix-<bug tracker name>-<bug number>
- Development, the name is dev-<developer name>-<dev-name>
- Release, the name is rel-<version>

The checking of naming convention has to be implemented (under study).

6.3 Branch tracker

A branch tracker would be useful, to know for each branch, why it exists (new development, bug fix, release), how it is implemented, and to follow modifications.

Solutions to implement a branch tracker are under study (in particular, where to host it). A link with the bug or feature request tracker is necessary, as well as a link with the code review.

6.4 Code review and delivery policy

Integration documents and code review remarks should also be attached in the branch tracker. Code review will be based on the pull request mechanism existing on GitHub. The exact flow and constraints of code reviews has to be defined precisely.

6.5 Continuous integration

In order to ensure that the code developed in a branch does not introduce any regression, some continuous integration has to be implemented. Solutions are under study to be able to check branches, at least when a merge agreement is requested. The master branch has to be checked on a regular basis.

6.6 Release policy

The release policy will have to be defined, in particular, when a release has to be made (annual roadmap, on demand (new features or important bug fixes available), etc…) and the naming convention.

7 How to begin with MIND on GitHub

Warning: This section is not a Git manual. It only indicates main Git commands to be able to begin with Git.

Information given below is for Linux users. This information is applicable with EGit for Eclipse users.

In the rest of the document, the commands to type are highlighted in grey, preceded by the character "➢".

```
➢ git <command>
```

Text between "<" and ">" has to be replaced with your text.

Text between "[" and "]" is optional.

7.1 Git Help

To obtain some help:

```
➢ git help [<command>]
```

7.2 Git Configuration

There are two kinds of configurations:

- Global configuration specific to a user, the same for all Git repositories: (in ~/.gitconfig)
- Local to a repository (in .git/config at the root of a project)

To configure identity (username), type the two following commands, once and for all:

```
➢ git config --global user.email "<your email address>"
➢ git config --global user.name "<user name>"
```

These commands set the default email (respectively name) for Git to use when you commit, so that your email (respectively name) can properly labels the commits you make.

To enable by default the coloring in all Git commands, use:

```
➢ git config --global color.ui auto
```

All these commands update the global configuration file (~/.gitconfig).

In this global configuration file, you can also define aliases and preferences (editor, merge and diff tool,…) by editing it or with other commands such as:

```
➢ git config --global core.editor <your favorite editor>
```

The global configuration file looks like:

```
[user]
        email = <email-address>
        name = <username>

[alias]
        st = status
        ci = commit
        co = checkout

[color]
        ui = auto

[core]
        pager = <favorite pager>

[diff]
        tool = <favorite diff tool>
```

Figure 1 - Example of global configuration file

7.3 Forking the MIND-Tools repository

From the MIND reference GitHub server (MIND-Tools organization – https://github.com/MIND-Tools), each developer has to create a fork (copy) of the MIND-Tools repositories he needs on his GitHub account (① on Figure 3). He will then be able to modify source code without any impact on the reference repository.

To do this, go to the MIND-Tools page (https://github.com/MIND-Tools), select the repository you want to fork and click the ***fork*** button. Then select where you want to fork this repository to (your GitHub account).

7.4 Cloning your fork

You have successfully forked the repositories you need, but so far they only exist on GitHub. To be able to work on them, you need to clone these repositories to your local machine (② on Figure 3).

The following code clones your fork of the repository into the current directory in the terminal you are typing commands.

```
➢  git clone <URL of repository to clone>
```

Repositories forked on your GitHub account (https://github.com/<github-username>) are accessible via several URLs:

- Read-only: *git://github.com/<github-username>/<repo>.git*. You can use this to clone, fetch and pull a repository, but you won't be able to push.
- Authenticated HTTPS: *https://github.com/<github-username>/<repo>.git*. Allows you to push - if you're allowed to for the given repository.
- SSH: *git@github.com: <github-username>/<repo>.git*. Allows you to push - if you're allowed to for the given repository. Uses SSH keys.

In the lines above, *<github-username>* is your GitHub username and *<repo>* is the name of the repository you want to clone.

7.5 Status

Current status in your local repository is given by the command:

```
➢ git status
```

The status gives information on the commits history, the next commit in preparation (called ***staging area***) and the local copy. It precisely gives what has changed and proposes required commands to update the staging area and do a commit. In the case of Figure 2, it indicates that:

- two files have been modified since last commit (README.txt and LICENSES.txt),
- README.txt has been added to the staging area(it will be in the next commit).

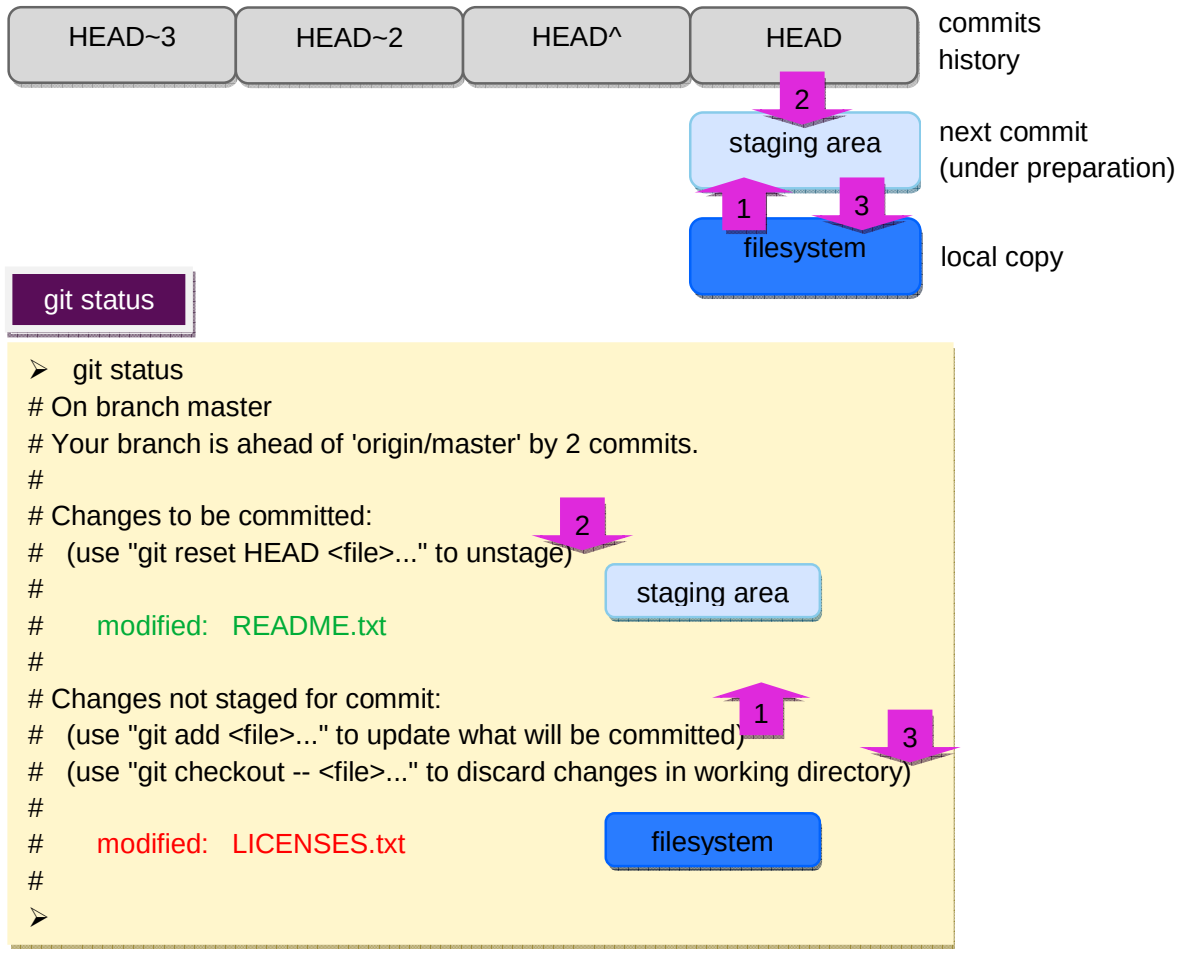

Figure 2 - Git status (commits / staging area / local copy)

To see modifications in file system as a patch, use the command:

```
➢ git diff [--color] [<path to a given file>]
```

If you give a *<path to a given file>,* only the differences on this file will be displayed.

git diff

```
diff --git a/LICENSES.txt b/LICENSES.txt
index cbf8250..8a96e20 100644
--- a/LICENSES.txt
+++ b/LICENSES.txt
@@ -1,3 +1,4 @@
+// Extra line for demo
```

To obtain the differences between the staging area and the history, the command is:

```
➢ git diff --cached.
```

git diff --cached

```
diff --git a/README.txt b/README.txt
index db6d900..daa7a11 100644
--- a/README.txt
+++ b/README.txt
@@ -1,4 +1,4 @@
-
+# Extra line for demo
```

7.6 History

The following command allows you to display history of commits. By default, the header of each commit (revision, date, author, message) is displayed.

```
➢ git log
```

Several options can be used to filter and format the output of this command.

For example, to display the 3 last commits as patches:

```
➢ git log -3 -p
```

To display revisions graph for the current (active) branch (summary view):

```
➢ git log --graph --color --decorate --pretty=oneline --abbrev-commit
```

A GUI allows to view history as well as modifications in each commit or to search for information, in a simple and intuitive way.

```
➢ gitk
```

The following command allows you to display the content of any revision as a patch:

```
➢ git show <revision>
```

In the line above, *<revision>* can be an absolute reference (SHA1) or a relative reference.

In cryptography, SHA1 (Secure Hash Algorithm) is a cryptographic hash function whose output is a 40-character checksum hash. Git assigns a unique SHA1 hash to each object, based on its contents.

The relative identifiers of revisions are the following:

- HEAD: last commit
- HEAD^: parent of HEAD
- HEAD^^: second parent of HEAD (for merge commit with several parents)

- HEAD~1: parent of HEAD (identical to HEAD^)
- HEAD~2: grandparent of HEAD
- HEAD~3: great-grandparent of HEAD
- and so on…

To compare current state with a given revision, the only identifier of this revision (SHA1) must be given to the ***git diff*** command:

```
➢ git diff <SHA1>
```

Two commits can be compared:

```
➢ git diff <revision1> <revision2>
```

7.7 Developing with branches

Git branches are convenient to use: it is easy and quick to create a new branch and to change branch to work on. Branches allow work in isolation on different aspects of a project.

7.7.1 Creating a new branch

To create a new branch (③ on Figure 3) and activate it, use:

```
➢ git checkout –b <new-branch>
```

To create a branch without activating it (staying on the current branch), use:

```
➢ git branch <new-branch-2>
```

To display existing branches:

```
➢ git branch
```

To change the current branch:

```
➢ git checkout <branch-to-activate>
```

All these operations are local ones.

7.7.2 Current branch and status

To verify the current branch, you can use two commands.

```
➢ git branch
```

This command indicates by a star (“*”) the current branch.

```
➢ git status
```

This command indicates ***#On branch <branch>***.

7.7.3 Commits

A commit (④ on Figure 3) is done in 2 steps:

- Update of the *staging area* with modified files to commit.
- Commit from the *staging area*.

The staging area corresponds to the next commit in preparation. It contains the modifications that will be committed (see Figure 2).

7.7.3.1 Update of the staging area with modified files to commit

A file is added to the staging area with:

```
➢ git add <file>
```

To add all modified files to the staging area:

```
➢ git add –u
```

Note: ***git diff*** run after this command gives an empty output. In effect, ***git diff*** shows the difference between the working tree and the staging area. To obtain the differences between the staging area and the history, the command is ***git diff --cached***. (see section 7.5).

7.7.3.2 Commit from the staging area

The command is:

```
➢ git commit
```

A text editor is opened. The content of the staging area (files included in the commit) is displayed as comments (it will not be part of the commit message). The commit message can then be typed above these comments. Save and quit. The commit is stored.

The following command can also be used:

```
➢ git commit -m “<commit comment>”
```

7.7.3.3 Commit using a GUI

A GUI can be used to prepare staging area and do the commit.

```
➢ git gui
```

7.7.3.4 Partial commits

It is possible to do partial commits, that is: to commit only some of the modifications made in a given file.

Remark: commit is a local operation. It does not perform any access to the network or remote repository.

7.7.3.5 Cancel some modifications (before commit)

If, you want to revert your changes after a ***git add***, in order not to commit them, the following command can be used:

```
➢ git reset
```

This command cancels the modifications in the staging area, but without deleting the modifications in the working tree.

To delete definitively the modifications in file <*file*> in the local copy, do a checkout:

```
➢ git checkout <file>
```

The checkout updates the local copy with state in the staging area, corresponding to the last commit.

The following command cancels modifications in the staging area (***git reset***) AND modifications in local copy (***git checkout***), by putting all files in the state of the last commit.

```
➢ git reset --hard
```

To cancel a commit corresponding to a given revision:

```
➢ git revert <revision>
```

To get a file as it was at a given revision:

```
➢ git checkout <revision> -- <file>
```

7.7.3.6 Stash

If you are working on a branch, and you want to switch branches to work on something else (for example, hot fix), but you do not want to commit half-done work in the current branch, you can use the following command to record the current state of the working directory and the staging area, in order to be able to go back to this point later.

```
➢ git stash
```

This command saves your local modifications away and reverts the working directory to match the HEAD commit. You then have a clean working directory to switch branch.

The modifications stashed away by this command can be listed with:

```
➢ git stash list
```

7.7.4 Merge of branches

To merge <*branch1*> into <*branch2*> do

```
➢ git checkout <branch2>
➢ git merge <branch1>
```

Branch <*branch1*> is not modified by the merge. A merge is an asymmetrical and non-destructive operation.

If Git detects some conflicts when merging two branches, they have to be resolved before the commit can be done. It is not possible to create a commit if a conflict is not resolved.

Merging and conflicts resolution can be done with mergetools such as opendiff, kdiff, meld, etc…

```
➢ git mergetool
```

Git launches a specialized tool (opendiff, kdiff, meld, etc…) to do an interactive 3-way merge on each file. This tool allows to compare the two versions of the file and to produce the merged version. When the tool is left, mergetool updates the staging area.

When conflicts are resolved, the commit can be done with an automatic message.

Conflicts resolution can also be done manually, in two steps:

- Modification of files (conflicts are indicated between <<<<<<<< and >>>>>>>>).
  One of the two versions can be selected (*--ours* to keep the version of the file you modified, *--theirs* to keep the version of the file of the branch into which you are merging):

```
➢ git checkout --ours <file>
➢ git checkout --theirs <file>
```

- Addition or deletion in staging area (***git add*** / ***git rm***) then commit.

Note: if a merge is too problematic to be resolved and you want to cancel it to start again from preceding state, the following command restores the working tree and the staging area in the state of the last commit.

```
➢ git merge --abort
```

7.7.5 Deletion of a branch

When a branch is merged into another one, it can be deleted (pointer on it is deleted so you cannot any longer access it, history is kept):

```
➢ git branch -d < branch-to-delete>
```

You cannot delete the current branch. If you want to delete it, you must change active branch.

Deletion is local. No branch is deleted on a remote repository.

Warning: If no merge has been done, commits will be definitively lost if branch is deleted.

7.8 Remotes

A remote is a repository stored on another computer, for example on GitHub's server. Git supports multiple remotes.

7.8.1 Add/delete a remote repository

It is possible to get the work of another developer by adding his remote repository in the list of known remote repositories:

```
➢ git remote add <other-dev-remote-name> <other-dev-remote-url>
➢ git remote show
```

To delete a remote repository:

```
➢ git remote rm <remote-name>
```

7.8.2 Configure remotes

When a repository is cloned, it has a default remote (the one used to clone the local repository when it was created) called ***origin***. In the case of MIND, this remote ***origin*** points to your fork on GitHub, not the original repository it was forked from. To keep track of the original repository, it **is mandatory** to add another remote usually named ***upstream***. So, type the following command:

```
➢ git remote add upstream https://github.com/MIND-Tools/<repo>.git
```

This command assigns the original repository to a remote (on MIND-Tools organization) called ***upstream***.

7.8.3 Information on remotes

Information on a remote repository is given by:

```
➢ git remote show <remote-name>
```

7.8.4 Work on a remote branch

A checkout of a remote branch allows to work on it. To create a local branch <*dev*> from remote branch < *origin/dev*>:

```
➢ git checkout -t <origin/dev>
```

This command associates the local branch with the remote branch.

7.9 Synchronization with remote repositories (upstream and origin)

To contribute to the reference repository on MIND-Tools organization, it is necessary to synchronize the local and the ***upstream*** remote repositories. To stay in line with code on reference repository (***upstream***) and get code provided by other developers, it is necessary to downward synchronize. To backup local code and share it with other developers, it is necessary to upward synchronize.

7.9.1 Downward synchronization (from upstream remote repository to local copy)

The command ***git fetch*** allows you to get data from ***upstream*** remote repository (changes not present in your local repository). Git gets commits and branches on this remote repository but does not modify local branches and files in your working tree.

To synchronize all remote branches:

```
➢ git fetch --all
```

To synchronize remote branches of the specified remote:

```
➢ git fetch <remote>
```

The ***git pull*** operation (⑤ on Figure 3) is equivalent to a fetch AND a merge with the remote repository. As every merge operation, some conflicts can happen. They must be resolved in the same way as any other merge.

```
➢ git pull <remote> <branch>
```

The branch <*branch*> of the remote repository <*remote*> is merged in the current branch.

It is important that branches of your local repository (including the master one) be updated regularly to benefit from the developments / bug fixes available in the reference repository.

7.9.2 Upward synchronization (from local copy to origin remote repository)

Work will be pushed on the origin repository (⑥ on Figure 3). This allows to back-up the work and to share it with other developers.

```
➢ git push
```

The remote repository is updated only if no modification happened since last synchronization. If it has been updated since last pull, the push operation will fail. A pull has to be done before the push.

```
➢ git push <remote> <branch>
```

The branch <*branch*> of the <*remote*> repository is updated with the current branch, except if it has diverged.

The complete form for push is:

```
➢ git push <remote> <source>:<destination>
```

7.10 Delivering a development

The delivery of a development / bug fix on the reference repository (***upstream***) must follow the following steps:

7.10.1 Updating the local branch

The local branch must be updated by synchronizing the local master branch with the reference remote repository (***upstream***) and then by rebasing the branch from the local master, in order to have a code as close as possible to the one in the reference master (pull commands - ⑤ on Figure 3).

7.10.2 Publishing the branch

The branch has to be published on your GitHub fork by pushing (push command - ⑥ on Figure 3).

7.10.3 Submitting the code for review

The branch is submitted for review to be merged on the reference repository (MIND-Tools organization) by making a pull request through GitHub web interface. Click on ***Pull request*** button (⑦ on Figure 3) and select the branch you want to merge and the repository in which you want to merge. It is important to do a pull request on a branch in order to keep the commit history of the branch.

Several reviewers have to be nominated, ideally one from each organization involved in the contributions to MIND technology and core tools. The list of reviewers has to be defined.

7.10.4 Code review

The code review (⑧ on Figure 3) is made on web interface of GitHub MIND-Tools organization. Everyone can comment on a pull request. Select ***Pull request*** index to see the patches waiting for agreement. Patches can be commented line by line and approved/rejected. A patch can be re-submitted if modified according to reviewers' remarks. A patch can be approved only if validation is successful regarding at least the QA (Quality Assurance) defined inside STMicroelectronics.

7.10.5 Merging the development on reference master

When all the reviewers have approved the development/fix, then the code is merged (⑨ on Figure 3) on reference master by the maintainer (STMicroelectronics). This can be done directly from the GitHub web interface, when no conflict happens (this should be the case if branches are up-to-date).

8 Workflow summary

The workflow to contribute to MIND sources is presented on Figure 3.

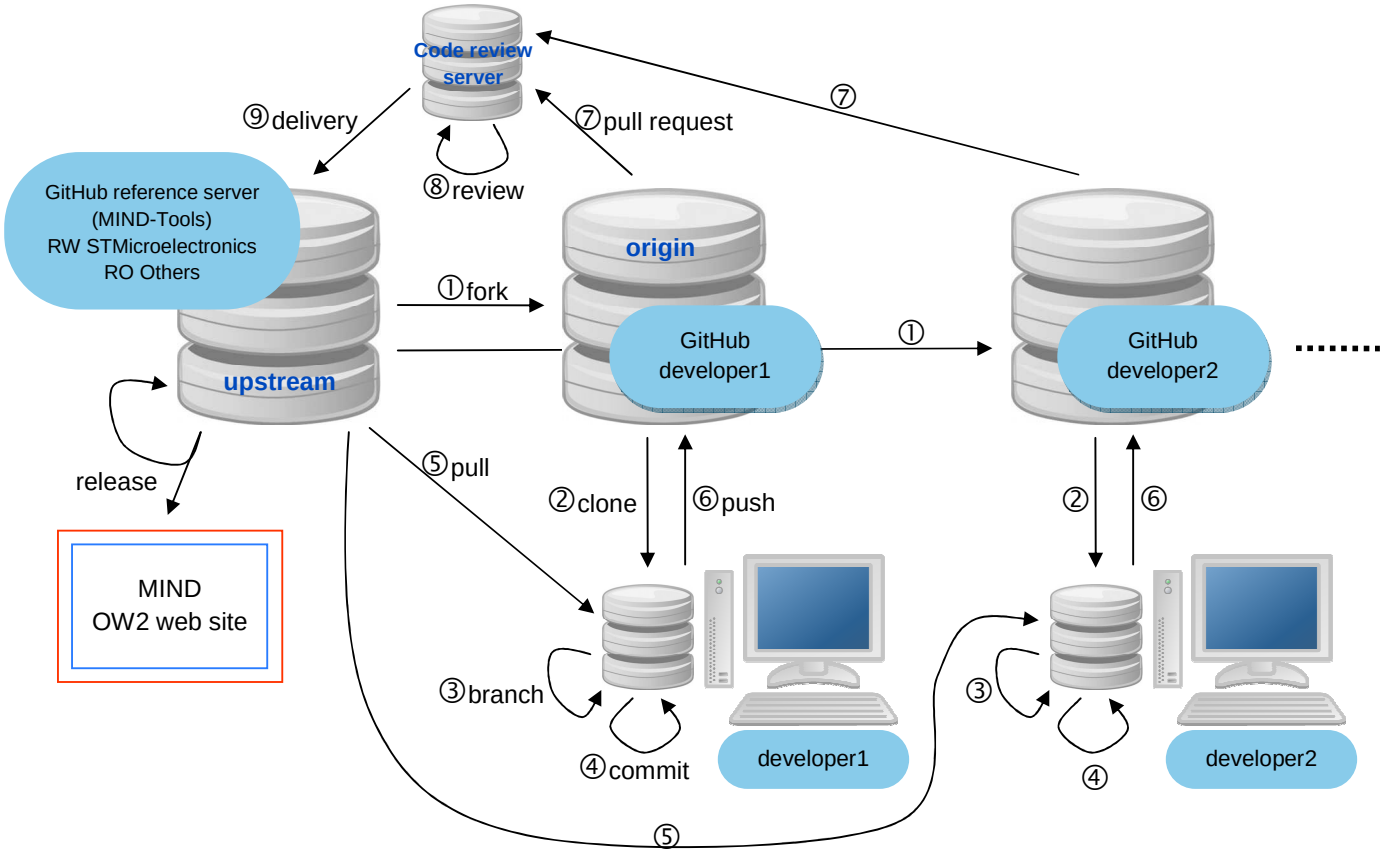

Figure 3: MIND workflow overview

1. From the GitHub reference server (MIND-Tools organization), each developer creates a ***fork*** on his GitHub account.
2. He then ***clones*** his fork on his local machine.
3. He can then develop and/or fix bugs by creating a ***branch*** for each development …
4. … and ***committing*** changes in the branch.
5. During development, he can ***pull*** from GitHub reference server to get updates from GitHub reference server and ***rebase*** his branches.
6. He can also ***push*** his branches on the fork on his GitHub account, in order to back-up his code and to share it.
7. When a branch is ready to be delivered on GitHub reference server, he has to perform a ***pull request***.
8. Then, the reviewers ***review*** the branch.
9. When accepted, the branch is ***delivered*** on GitHub reference server by the maintainer.

Releases are also prepared and deployed by the maintainer, according to the release policy.

9 Next steps

Some parts linked to the workflow are still under study/implementation:

- Checking of branch naming convention,
- monitoring and delivery of branches through the branch tracker,
- exact flow and constraints of code reviews,
- Continuous integration,
- Release scripts.

This document will be updated as soon as a solution meets the requirements.

The development process presented in this document will be presented during a meeting to be planned in September 2012 (tentative), at STMicroelectronics Grenoble site.