# Git Repository Management and Workflow

Version: November 2016

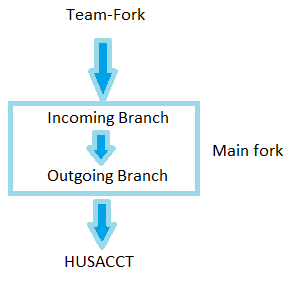
To work efficiently and effectively with teams, it’s recommended to have a proper git workflow. This document will describe the workflow used by the HUSACCT 2016 team. In the end, it was a very effective workflow.

Useful explanation on the organization of branches is available at:   
<http://nvie.com/posts/a-successful-git-branching-model/>

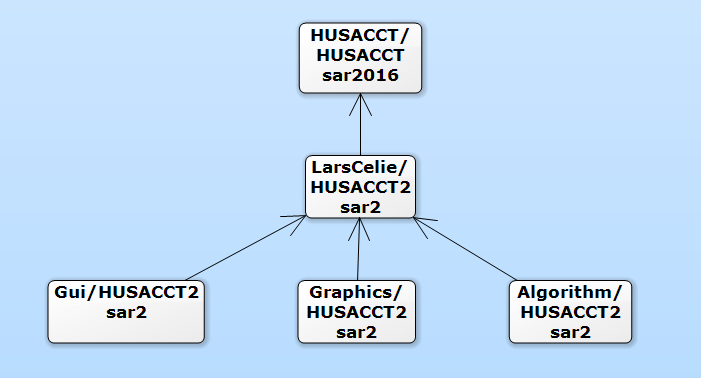
# 1. Setup

There are many open-source projects on the internet. And most likely you will not have push rights to contribute to these repositories. If you do not have push rights (‘Collaborator’ on GitHub) then simply cloning the repository will not work. So to contribute to any open-source repository, you need to create a fork of that main repository (called ‘project fork’ in this document). It is recommended to assign one or two team members that are responsible for managing Git.

For the Git managers, it’s also recommended to create 2 branches of the appropriate development branch (in our case the sar2016 branch) on the fork: an incoming and an outgoing branch. The incoming will accept all changes from the team forks and the outgoing branch will hold all the updates verified and to be accepted into HUSACCT main. This will also help when the main HUSACCT repository has updates that the project fork does not yet have. The stream can be inverted. In the case that the project fork is both ahead and behind, there won’t be an issue because it is separated into branches. They can then be merged into each other easier.



To avoid common issues as much as possible and to make sure the Git Managers won’t have too much work to do, it’s also recommended to create another fork per team on project fork. This would look something like this:



There are several reasons why this is useful:

1. Every team will work will pull requests. (This is good for practicing Git management)
2. These pull requests will notify if there are merge conflicts.
3. Pull requests can be checked by automation and integration tools like Travis-CI.
4. This creates a leadership responsibility per team for the repository owner.
5. If the pull request isn’t mergable or the tests fail, they can be denied and have to solved by the team that submitted it.

# 2. Workflow git managers

## 2.1 Incoming changes from teams

The pull requests will be made towards the project repository and it’s up to the Git Managers to decide to accepts them or not. There are a couple of things that you need to check to verify that the code you’re about to accept is good:

1. GitHub says the pullrequest is mergable.
2. Travis-CI says the test have passed.
3. Manually check if the files have not completely been overridden. (If this is the case, don’t use EGit and use SourceTree. This might also be an issue with line endings on windows. CLRF)
4. There is a proper description on the tasks in the pull requests. This helps to see what was added and when.

If all these things are okay, then you can go ahead and accept it, if they are not, simply reject the pull request and inform the team what they have to improve.

## 2.2 Incoming changes from HUSACCT/Main fork

It’s possible that HUSACCT’s development continues on the main fork, while your project’s development also continues. If this is the case, you need to integrate the changes made into your project fork. The easiest way to do this is by using the command:

|  |
| --- |
| **Syntax:** *git pull <repository url> <branch>*  **Example:** *git pull http://github.com/HUSACCT.git sar2016* |

This will perform a cross repository pull and will get all the updates from HUSACCT and put it into your repository.

# 3 Workflow developers

As a normal developer, there are only a few things you have to worry about.

## 3.1 Pull before commiting

To prevent any issues or perhaps duplicate work, pull often because you may be behind. Pulling often is the number one solution to preventing unnecessary merge conflicts.

## 3.2 Test beforing pushing

To ensure that all your commits haven’t broke anything, you should run the Junit tests to verify. If you don’t and you break something, it’ll get back to you anyway because of the rejected pull requests. So do it right, or get blamed for not doing it.

Also make sure that the commits you made reflect what you changed. There is a weird bug that happens with Eclipse and EGit that sometimes override the whole file while only a few lines were changes. This bug however replaces every line of code and changes the author. This messes up the Git history. Check your commits to detect this early. The best option to this issue is discarding the commit when it’s not pushed yet and preferably using Source Tree.

## 3.3 Test After pushing

Travis-CI will check every push you made. If by some reason you did not test your commits before pushing and you broke something, you will find out soon enough by Travis.

## 3.4 Merge conflicts

It’s almost inevitable though. IntelliJ editor and SourceTree have pretty good merge tools to help you with this. I personally used ‘KDiff3’ (Yes it’s an old program but still works great!) which can be used with Source Tree. There’s more information on how to set this up here:

http://stackoverflow.com/questions/16800280/how-to-interactively-visually-resolve-conflicts-in-sourcetree-git/16824546#16824546

# 4. Workflow Team repository owners

The team repository owners are normal developers as well. But you are also in charge of creating Pull Requests. Before doing this though, you need to make sure your team repository is up to date with the project fork and there are no conflicts. So pull the changes (see 2.2) and solve potential merge conflicts (see 3.4)

Create a pull request to the project repository on GitHub and provide a good description on what you’ve done.

# 5. Travis

Travis-CI is a continuous integration tool. It helps you build your (open-source) projects. Every push and pull request made to the repository result in a signal to travis to build and test. This is useful because you don’t have to manually test incoming changes.

Using travis for your repository is easy and is only three steps. (Two because one is already done!)

1. Create an account on travis. (http://travis.ci-org) Travis will redirect you to login via GitHub. With your account, you can synchronize your repositories and enable travis on a specific repository.
2. Add a .travis.yml file to your project. This is already done, no worries here.
3. Push changes to your project.

Travis will now check every push and notify you if the status of the build will change.

## Customizing Travis

The .travis.yml file looks like this:

|  |
| --- |
| **language:** java **before\_script:** - **"export DISPLAY=:99.0"** - **"sh -e /etc/init.d/xvfb start"** *# this is a virtual screen* - sleep 3 *# give xvfb some time to start* **script:** ant main **jdk:** - oraclejdk8 **notifications:  slack:** husacct:YpeL7dIr0OEKvPRCVIV03XCd |

It’s pretty self-explanatory. The power in building what you want is in the script tag. This will execute the build.xml ant script, with the ‘main’ target.

Another remark is the slack channel code. This will most likely change if another slack is used or a different channel is used. Ask your slack owner to configure slack for the correct channel.

If you want to customize more, you can find more information here:

https://docs.travis-ci.com/user/getting-started/