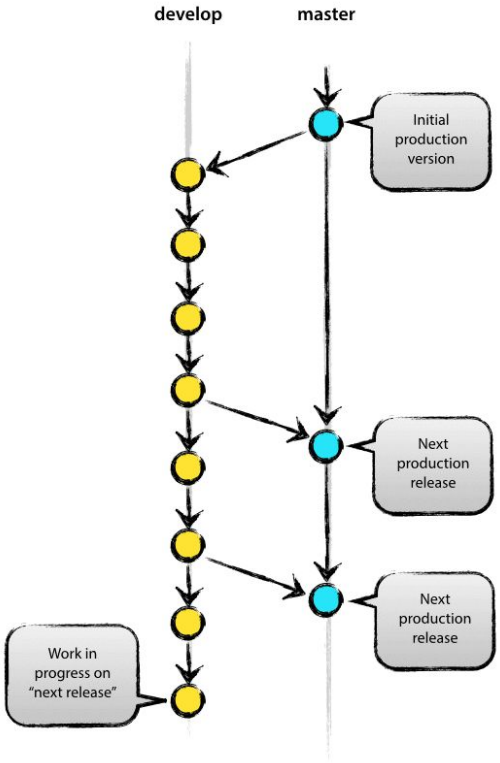
Workflow

All release branches are going to be merged to master. Initially, those things specific to each old release will be included with a macro. But further that will be changed to use a runtime variable.
Then, a develop branch will be created from master.

Once that is done, git workflow will be as follows:

The central repo holds two main branches with an infinite lifetime:

- master
- develop



1. master branch HEAD will always reflect the production-ready state.
2. master branch at origin should be familiar to every git user.
3. There is a branch called develop, parallel to the master branch.
4. origin/develop is the main branch where the source code of HEAD always reflects a state with the latest delivered development changes for the next release.
5. origin/develop is where any automatic nightly builds are built from.
6. Once source code in develop branch reaches a stable point and is ready to be released, all of the changes should be merged back into master somehow and then tagged with a release number. Thus, each time changes are merged back into master, there si a new production release by definition. We must be strict at this.
7. Next to the main master and develop branches, the development model uses a variety of supporting branches to aid parallel development between team members, ease tracking of features, prepare for production releases and to asist in quickly fixing live production problems. Unlinke the main branches, these branches always have a limited life time, since they will be removed eventually. These branches may be the following:

- Feature branches
- Release branches
- Hotfix branches

Feature branches

Feature branches are used to develop new features for the upcoming or a distant future release. A feature branch exists as long as the feature is in development, but will eventually be merged back into develop to definitely add the new feature to the upcoming release, or discarded.

Basic rules:

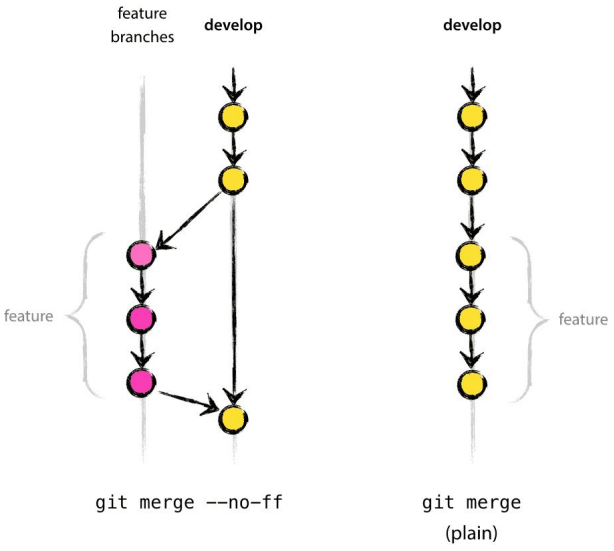
1. **Feature branches always must branch off from develop** with following command:

$ git checkout –b feature-*[my feature]* develop
Switched to a new branch feature-*[my feature]*

2. **Feature branches merge back into develop, and then deleted**, when that feature is being incorporated in an upcoming release. In that case, that will be done with the following commands:

$ git checkout develop
Switched to branch ‘develop’
$ git merge –no-off feature-*[my feature]*
Updating
(Summary of changes)
$ git branch –d feature-*[my feature]*
Deleted branch feature-*[my feature]* (was xxx)
$ git push origin develop

It is important to remark the no-off flag. It causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward, thus avoiding lose of information about the historical existence of a feature branch and groups together all commits that together added the feature, as shown in the image.



3. **Feature branches name will always be “feature-*”.**

Release branches

Release branches support preparation of a new production release. They allow for last minute minor modifications, like comments for running doxygen without warnings, minor bug fixes, preparing meta-data for a release (version number, build dates, etc). Doing this work on a release branch, the develop branch is cleared to receive feature for the next big release.

The key moment to branch off a new release branch from develop is when develop reflects the desired state of the new release. All features that are targeted for this release must be merged in to develop before to branch off a new release branch.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number. Up until that moment, the develop branch reflected changes for the “next release”, but it is unclear whether that “next release” will evenutally be 02.00.00 or 05.’7.01 until next release branch is started.

Basic rules:

1. **Must branch off from develop** with the following command:

```
$ git checkout -b release-1.2 develop
Switched to a new branch "release-1.2"
$ update-version 1.2
Files modified successfully, version bumped to 1.2.
$ git commit -a -m "Bumped version number to 1.2"
[release-1.2 74d9424] Bumped version number to 1.2
1 files changed, 1 insertions(+), 1 deletions(-)
```

On that command sequence, update-version is assumed to be a script or batch file which modifies properly the files to update the project to version 1.2

2. **Must merge back into develop and master and deleted** when the state of the release branch is ready to be a real release. Following up proposed commands:

```
$ git checkout master
Switched to branch 'master'
$ git merge --no-ff release-POC-1.2
Merge made by recursive.
(Summary of changes)
$ git tag -a POC-1.2
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff release-POC-1.2
Merge made by recursive.
```

(Summary of changes)
$ git branch -d release-POC1.2
Deleted branch release-1.2 (was xxx).

3. Release branches must be named release-*[product-version]*

Hotfix branches

Hotfix branches are like release branches, since they are also meant to prepare a new production release, but uplanned. They come from the necessity to act immediately upon an undesired state of a live production version. When a critical bug in a production version must be resolved immediately, a hotfix branch may be branched off from the corresponding tag on the master branch that marks the production version.

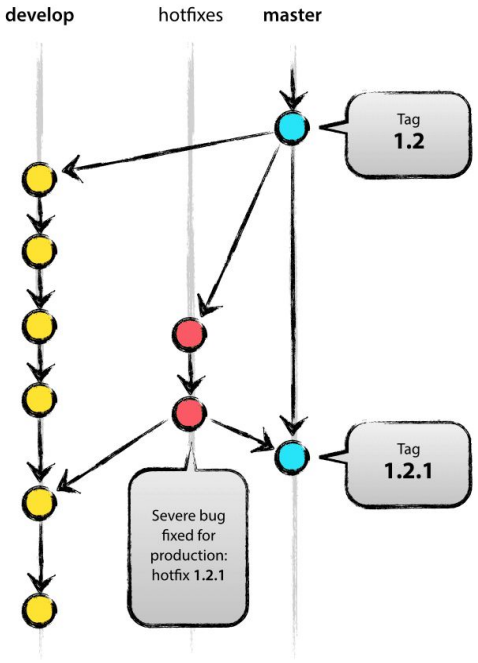
The essence is that work of team members, on the develop branch, can continue, while another person is preparing a quick production fix.

Basic rules:

1. **A hotfix branch must branch off from master**, following up the proposed commands:

$ git checkout -b hotfix-1.2.1 master
Switched to a new branch "hotfix-1.2.1"
$ update-version 1.2.1
Files modified successfully, version bumped to 1.2.1.
$ git commit -a -m "Bumped version number to 1.2.1"
[hotfix-1.2.1 xxx] Bumped version number to 1.2.1
1 files changed, 1 insertions(+), 1 deletions(-)
$ git commit -m "Fixed severe production problem"
[hotfix-1.2.1 xxx] Fixed severe production problem
5 files changed, 32 insertions(+), 17 deletions(-)

2. **A hotfix branch must merge back to develop and master**, when the bugs are fixed, and then deleted. Following up the proposed commands:



$ git checkout master
Switched to branch 'master'
$ git merge --no-ff hotfix-1.2.1
Merge made by recursive.
(Summary of changes)
$ git tag -a 1.2.1
$ git checkout develop
Switched to branch 'develop'
$ git merge --no-ff hotfix-1.2.1

Merge made by recursive.
(Summary of changes)
$ git branch -d hotfix-1.2.1
Deleted branch hotfix-1.2.1 (was xxx).

3. **A hotfix branch is always named hotfix-*[product-mastertag].***