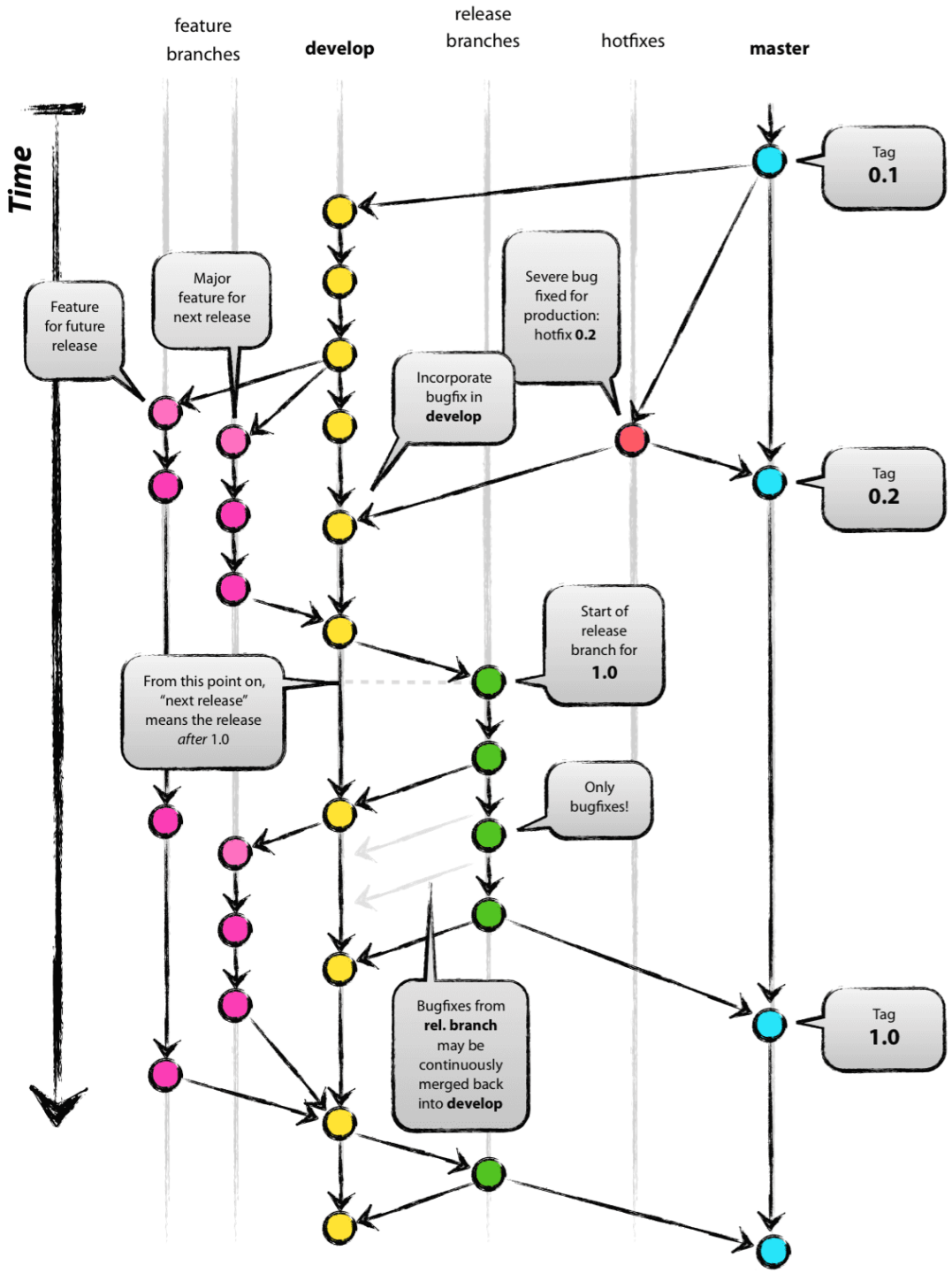
Contributing - GitHub Branch Management

A GitHub branching model is described in a paper by Vincent Driessen at http://nvie.com/posts/a-successful-git-branching-model/

To avoid total anarchy in management of the HRE GitHub repository, it is intended to use something similar – this may go somewhat against the free-for-all nature of GitHub, but having observed the difference between badly managed and well-managed open-source developments, it is apparent HRE needs some form of control.

The following is Vincent's original total diagram showing the branching strategy and release management approach – this is a whole lot more complex than HRE needs, but the intent is to use similar basic ideas, as follows.



Vincent's document focuses around using the Git tool (https://git-scm.com/) for versioning and manipulation of all items. You may not want to use Git, given it is based on a command-line interface, but that's up to you – it is somewhat dependent on the items in each branch being only source code, which might not work for HRE. You can review his full paper for all the Git commands that perform the various functions he describes.

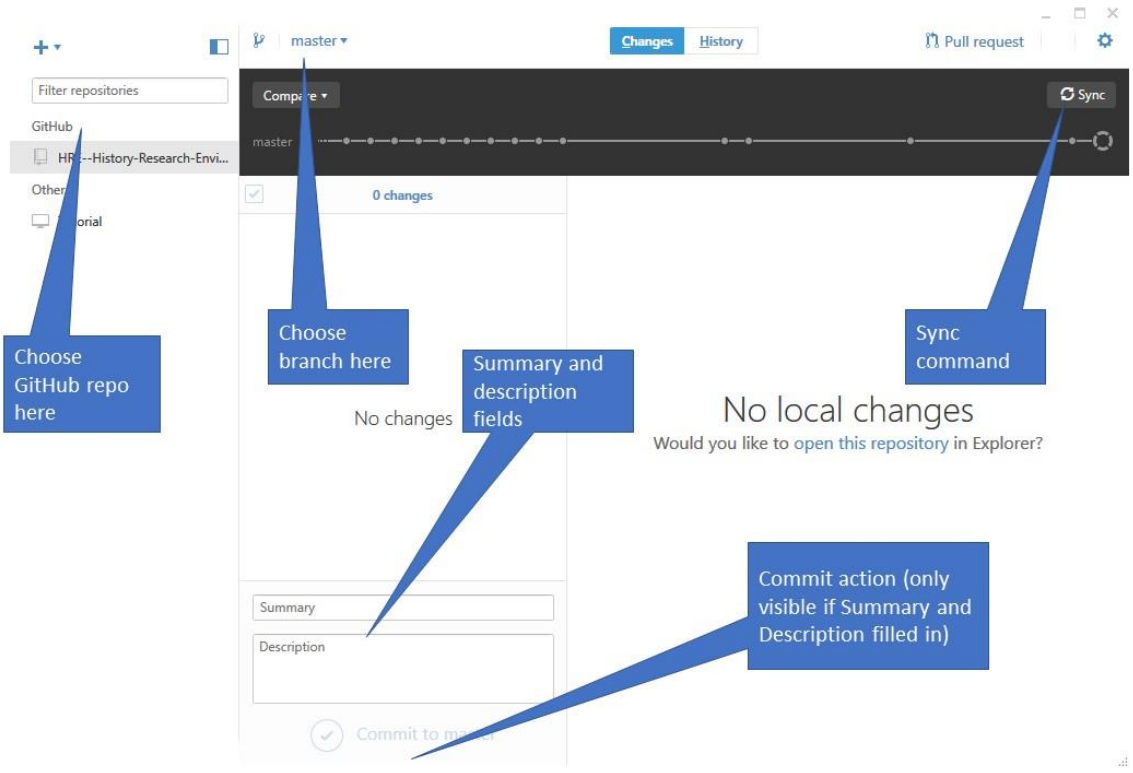
GitHub Desktop

For HRE it may make more sense to use GitHub Desktop, which you can download from https://desktop.github.com/ - the link to download is on the site, but beware it is for 64-bit Windows only.

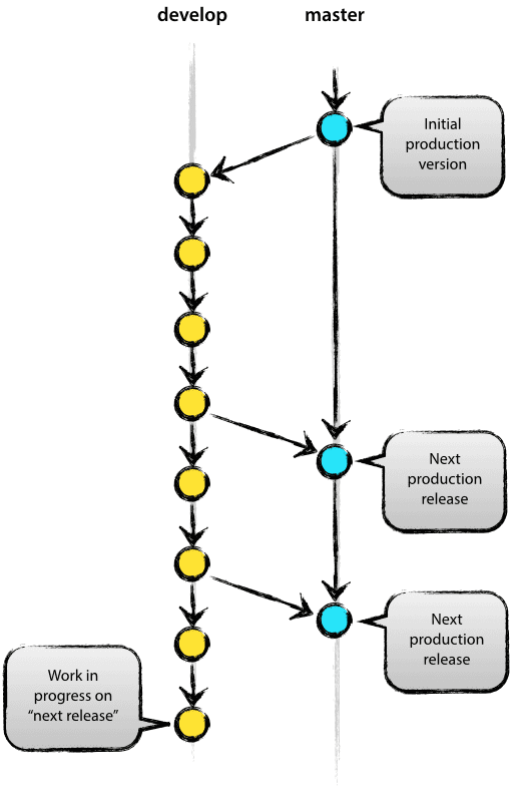
GitHub Desktop allows you to manipulate any GitHub repository on your desktop using the normal Explorer interface, then synchronise your changes with the online GitHub, and vice-versa.

Once you've installed it, link it to the HRE repository and click 'sync' (top right of screen) – this will copy the HRE repo structure onto your hard disk. You can continue to make changes direct on GitHub, and the changes will then be repeated on your PC every time you use the 'Sync' button OR make changes on your PC version and have them loaded back into the GitHub repository.
There is a trick to getting local changes committed onto GitHub. Here's how to do it (see diagram below):

- Select the HRE repository on the left top pane (assuming you have added it there)
- At the top you will see a drop down menu (by default set to master) - change it to the branch you're trying to do the commit to
- Click on the changes tab (top middle) – which is the default anyway
- The left part of the screen should show those files on your local repo (your PC) that you're trying to commit. If there are more than one, ensure they're all ticked
- Now type something in the 'summary' and 'description' sections at bottom of the screen
- At this stage (and ONLY now) the 'Commit to master' item below the description section gets highlighted. Click on it and it will perform the commit of your desktop repo
- Now click on 'sync' on the top right corner of your Github Desktop and it will sync/commit to the online repository.



The main branch concepts



The main HRE repository should hold two main branches with an infinite lifetime:

- master
- develop.

The `master` branch should be familiar; parallel to the `master` branch, another branch now exists called `develop`.
We consider `master` to be the main branch where the code HRE always reflects a *production release*.

We consider `develop` to be the main branch where the code of `HRE` always reflects a state with the latest delivered development changes for the next release. Some would call this the "integration branch".
When the code in the `develop` branch reaches a stable point and is ready to be released, all of the changes should be merged back into `master` and then tagged with a release number.
Therefore, each time when changes are merged back into `master`, this is a new production release *by definition*. We should be very strict at this.

Supporting branches

The following 3 other branches are areas we probably don't need to go to yet, but as they were well described in Vincent's original paper, I've left the basic rules/description here for reference as well.

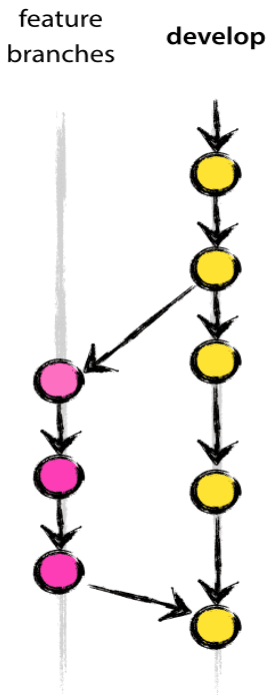
Next to the main branches `master` and `develop`, the model uses a variety of supporting branches to aid parallel development between team members, ease tracking of features, prepare for production releases and to assist in fixing identified problems. Unlike the main branches, these branches always have a limited life time, since they will be removed eventually.

The different types of branches we may use are:

- Feature branches
- Release branches
- Hotfix branches

Each of these branches have a specific purpose and are bound by strict rules as to which branches may be their originating branch and which branches must be their merge targets. The branch types are categorized by how we *use* them.

Feature branches



May branch off from:
`develop`
Must merge back into:
`develop`
Branch naming convention:
`feature-name plus an ID number identifying the originating develop version#.`

Feature branches (or sometimes called topic branches) are used to develop new features for an upcoming or a distant future release. When starting development of a feature, the target release in which this feature will be incorporated may well be unknown at that point.
The essence of a feature branch is that it 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 (in case of a disappointing experiment).
Feature branches typically exist in developer repos only, not in the HRE repo.

Release branches

May branch off from:
`develop`
Must merge back into:
`develop` and `master`
Branch naming convention:
`release-*.`

Release branches support preparation of a new production release. They allow for last-minute dotting of i's and crossing t's. and minor bug fixes. By doing this work on a release branch, the `develop` branch is cleared to receive features for the next big release.
The key moment to branch off a new release branch from `develop` is when develop (almost) reflects the desired state of the new release. At least all features that are targeted for the release-to-be-built must be merged in to `develop` at this point in time. All features targeted at future releases may not—they must wait until after the release branch is branched off.

It is exactly at the start of a release branch that the upcoming release gets assigned a version number—not any earlier. Up until that moment, the `develop` branch reflected changes for the "next release", but it is unclear whether that "next release" will eventually become 0.3 or 1.0, until the release branch is started. That decision is made on the start of the release branch and is carried out by the project's rules on version number bumping.

Creating a release branch

Release branches are created from the `develop` branch. For example, say version 1.1.5 is the current production release and we have a big release coming up. The state of `develop` is ready for the "next release" and we have decided that this will become version 1.2 (rather than 1.1.6 or 2.0). So we branch off and give the release branch a name reflecting the new version number:

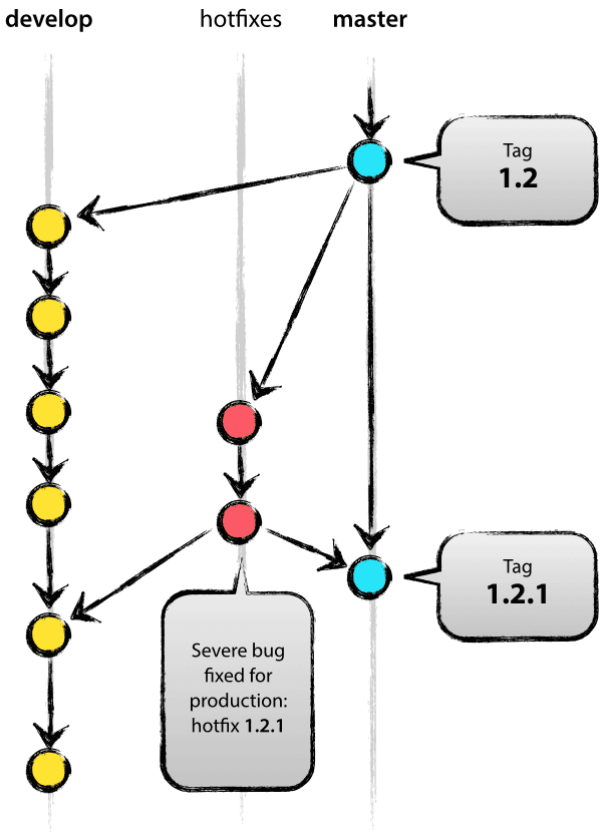
After creating a new branch and switching to it, we bump the version number. Then, the bumped version number is committed.
This new branch may exist there for a while, until the release may be rolled out definitely. During that time, bug fixes may be applied in this branch (rather than on the `develop` branch). Adding large new

features here is strictly prohibited. They must be merged into `develop`, and therefore, wait for the next big release.

Finishing a release branch

When the state of the release branch is ready to become a real release, some actions need to be carried out. First, the release branch is merged into `master` (since every commit on `master` is a new release *by definition*, remember). Next, that commit on `master` must be tagged for easy future reference to this historical version. Finally, the changes made on the release branch need to be merged back into `develop`, so that future releases also contain these bug fixes.

Hotfix branches



May branch off from:
`master`
Must merge back into:
`develop` and `master`
Branch naming convention:
`hotfix-*`

Hotfix branches are very much like release branches in that they are also meant to prepare for a new production release, albeit unplanned. They arise 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.

Creating the hotfix branch

Hotfix branches are created from the `master` branch. For example, say version 1.2 is the current production release running live and causing troubles due to a severe bug. But changes on `develop` are yet unstable. We may then branch off a hotfix branch and start fixing the problem. Then, fix the bug and commit the fix in one or more separate commits.

Finishing a hotfix branch

When finished, the bugfix needs to be merged back into `master`, but also needs to be merged back into `develop`, in order to safeguard that the bugfix is included in the next release as well. This is completely similar to how release branches are finished.
The one exception to the rule here is that, **when a release branch currently exists, the hotfix changes need to be merged into that release branch, instead of** `develop`. Back-merging the bugfix into the release branch will eventually result in the bugfix being merged into `develop` too, when the release branch is finished. (If work in `develop` immediately requires this bugfix and cannot wait for the release branch to be finished, you may safely merge the bugfix into `develop` now already as well.)