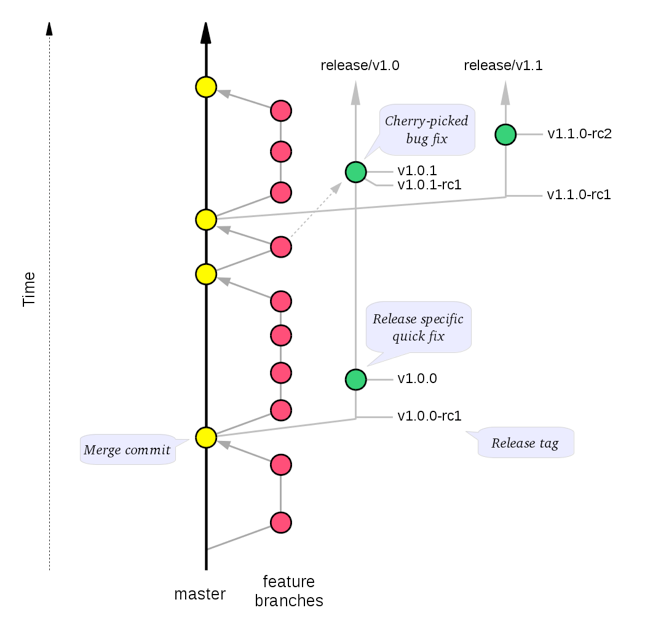
The main idea is that development should go through a **stable mainline** and pass all integration tests (**Continuous Integration**).

**Branches**

In this model, there are three types of branches: the **master branch**, **feature branches** and **release branches**.



**Main branch**

There is only one eternal, canonical branch: **master**.

For most purposes, the master branch is considered *stable*. In other words, if you check out the master branch you can expect that:

* It builds on all supported platforms/targets.
* All unit tests pass (as well as static tests, linter checks and the like).
* A “standard run” of the software works (e.g. if it’s a GUI application, you should be able to launch it and do some basic operation).

## Feature branches

All development is done in dedicated (relatively short lived) feature branches. This is where most of the action takes place, including:

* Feature development.
* Code review.
* Integration testing.

A feature branch branches off from master, and once development is finished and all the integration criteria have been met, it is merged back to the master branch.

A feature branch is typically called **feature/ticket\_description**, where ticket is a reference to the corresponding ticket in the project issue tracker (if applicable), and description is a very short description (up to five words or so) of the purpose of the branch.

Example: **feature/ABR-226134-granular-user-roles**

## Release branches

A release branch is essentially a fork of the master branch; it is never merged back to the master branch.

The main intent of a release branch is to freeze the code, and to have a point to fall back to if a hotfix is required in the future (e.g. if a customer needs a critical bugfix without consuming all the changes that have been made on master).

Given a regular **major.minor.patch** version numbering scheme (e.g. [semantic versioning](http://semver.org/)), a release branch should be named **release/X.Y**, where X is the major version number and Y is the minor version number.

Example: **release/9.0**

## Release tags

In addition to release branches, release tags are created for each actual release (this may include release candidates that are intended for QA or beta testing, as well as public or customer releases). The release tags are made in the corresponding release branch.

The commit that represents a specific release is tagged with a tag named **X.Y.Z**, optionally suffixed with a textual identifier, such as **-rc**, **-beta** or **-some-customer**.

Example: **9.0.368-rc.42**

# Feature development

As mentioned earlier, all development takes place in feature branches – no development is done directly on master.

There are several reasons for this:

* It makes it easy to do code review and integration tests before the code hits master, thereby guaranteeing that the master branch is always in a good shape.
* Having a single way of doing things makes life easier for everyone. No ambiguities. No special cases.
* The information about what commits belong to a certain feature or task is clear.

The details of how the development cycle works depends on what CI and code review tools you use, but here is an example.

### Development

1. A feature branch is created locally, based on the latest master from origin.
2. Iterate until done:
   1. The feature is developed, and commits are made to the local feature branch.
   2. The developer runs unit tests and other checks locally, and fixes any regressions if necessary.
   3. If necessary, the feature branch is rebased locally on the latest master from origin.

### Integration

1. The branch is pushed to a centralized repository that is used for code review and CI testing.
2. Iterate until stable:
   1. Reviewers point out issues that the developer then fixes.
   2. Builds are done and tests are run on the CI server, and the developer fixes any reported issues.
   3. Once the branch is considered good, the commit history is [cleaned up](https://git-scm.com/book/en/v2/Git-Tools-Rewriting-History) as needed, and the branch is force pushed to the remote.
   4. If necessary, the branch is rebased on the latest master, and force pushed to the remote.
3. The branch is merged to the master branch using a non-fast-forward merge commit.

# Creating a release

There are essentially two kinds of releases:

* Standard releases, that include new features from master.
* Patch releases, that fix bugs or problems in earlier releases.

The main difference between the two is that standard releases are based on master, while patch releases are based on previous release branches. Otherwise the processes for the two are very similar.

## Standard release

Once it has been decided what version number to use and what commit to base a release on, a new release branch is created from the selected commit, with a name based on the selected version number (as described earlier, i.e. **release/X.Y**, where X and Y are the major and minor version numbers, respectively).

The initial purpose of the release branch is to feature-freeze the release, while keeping the master branch open for continued development.

### Release stabilization

To stabilize the release branch, the following process is iterated until a final release is ready:

1. Create a new release tag on the release branch. The tag is named **X.Y.0-rc.N**.
   * **.0** (zero), because this is the first release version from the vX.Y release branch.
   * **-rc.N**, where N is an increasing number (starting at 1), indicates that this is a release candidate.
2. Build the release using your build / release process (e.g. trigger a build factory in the CI system).
3. Put the build through the required testing process (e.g. QA and/or thorough system testing).
4. Fix any bugs that are found, and commit the fixes to the release branch (see below).

When no more bugs are found, tag the tip of the release branch with **X.Y.0** (it is no longer a release candidate), and if necessary make a final build (e.g. to get the release number correct in the release package etc).

### Fixing bugs on the release branch

Any bugs that are found in a release candidate can be fixed in one of two ways.

#### Proper fix on master

The preferred way to fix a release bug is to implement the fix on master, using the conventional feature development process, and then cherry-pick the bug fix commit(s) to the release branch.

The advantages are that the fix will automatically be included in future releases, and it will be subject to the standard integration process (i.e. all the regular quality measures apply).

#### Quick-fix on the release branch

If the fix is too complex to implement properly in time for the release (e.g. if the bug is a symptom of a bigger architectural problem), or if the corresponding fix on master would be incompatible with the release branch, one option is to make a quick-fix on the release branch.

When doing so, it is important to create a ticket for fixing the bug properly on master, and preferably start working on it right away.

## Patch release

A patch release is created from an existing release branch. For instance, if **9.2.0** needs a critical patch, the patch is applied on top of the **release/9.2** branch, and then the regular release stabilization process is performed (except that tags now use the name of the new version number, for instance **9.2.1-rc.1**).