A complete Mercurial branching strategy

3 simple rules: default, stable, feature-branches

This is a **complete collaboration model** for Mercurial. It boils down to **3 simple rules** and **shows you all the actions** you may need to take, except for the basics already found in other tutorials like

- Mercurial in workflows (official guide, 15 minutes)
- hg init science (slides 12 to 23)

Adaptions optimize the model for special needs like maintaining multiple releases, grafting micro-releases and an explicit code review stage.

Contents

1 3 simple rules

Any model to be used by people should consist of simple, consistent rules. Programming is complex enough without having to worry about elaborate branching directives. Therefore this model boils down to **3 simple rules**:

1. you do **all the work on** `default`[1] - except for hotfixes.
2. on `stable` you only do **hotfixes**, **merges** for release[2] and **tagging** for release. Only maintainers[3] touch stable.
3. you can use arbitrary **feature-branches**[4], as long as you don't call them `default` or `stable`. They always **start at default** (since you do all the work on default).

These rules are local. Use ***`pull`*** *and* ***`push`*** *to transfer changes and* ***`merge`*** *to join diverging work on your branch.*[5]

[1] `default` is the default branch. That's the named branch you use when you don't explicitely set a branch. Its alias is the empty string, so if no branch is shown in the log (`hg log`), you're on the default branch. *Thanks to John for asking!*

[2] If you want to release the changes from `default` in smaller chunks, you can also graft specific changes into a release preparation branch and merge that instead of directly merging default into stable. This can be useful to get real-life testing of the distinct parts. For details see the extension Graft changes into micro-releases.

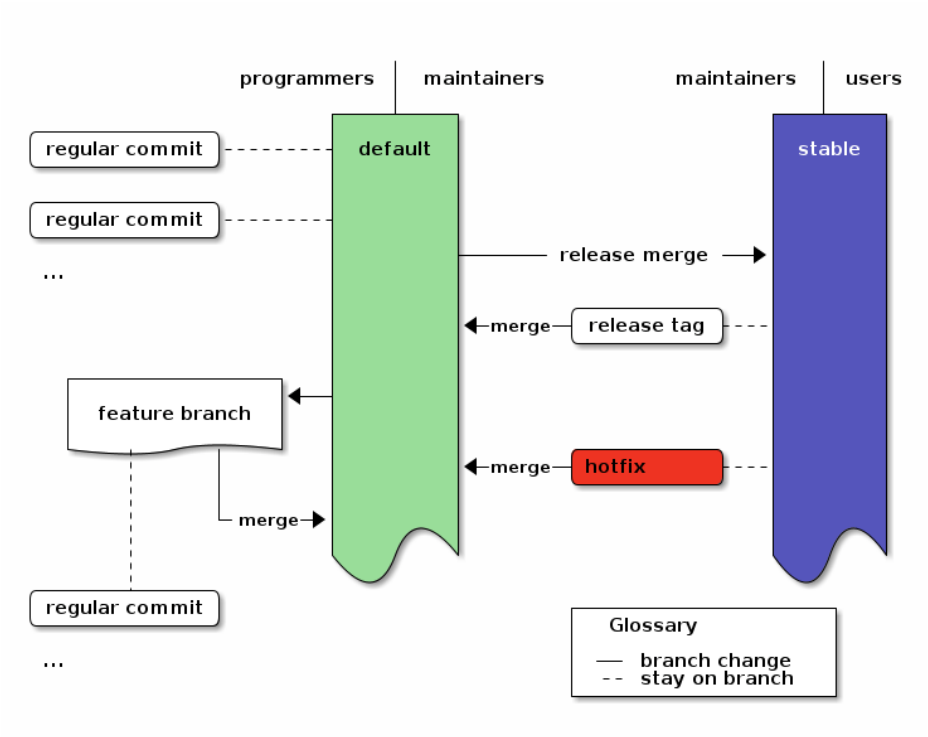
[3] Maintainers are those who do releases, while they do a release. At any other time, they follow the same patterns as everyone else. If the release tasks seem a bit long, keep in mind that you only need them when you do the release. Their goal is to make regular development as easy as possible, so you can tell your non-releasing colleagues "just work on default and everything will be fine".

[4] This model does not use bookmarks, because they don't offer benefits which outweight the cost of introducing another concept: If you use bookmarks for differenciating lines of development, you have to define the canonical revision to clone by setting the `@` bookmark. For local work and small features, bookmarks can be used quite well, though, and since this model does not define their use, it also does not limit it.
Additionally bookmarks could be useful for feature branches, if you use many of them (in that case reusing names is a real danger and not just a rare annoyance) or if you use release branches:
"What are people working on right now?" → `hg bookmarks`
"Which lines of development do we have in the project?" → `hg branches`

[5] You can use any kind of repository structure and synchronization scheme. Usually you'll use `hg pull` and `hg push` to transfer changes and `hg merge` to join diverging work on the branch. The practical actions only require that you synchronize your repositories with the other contributors at some point.

1.1 Diagram

To visualize the structure, here's a 3-tiered diagram. To the left are the actions of **programmers** (commits and feature branches) and in the center the tasks for **maintainers** (release and hotfix). The **users** to the right just use the stable branch.[6]



2 Practial Actions

Now we can look at all the actions you will ever need to do in this model:

- **Synchronize**
  - transfer changes: `hg pull` and `hg push`
  - join other work on the same branch: `hg merge`
- **Regular development**
  - commit changes: `(edit); hg ci -m "message"`
  - continue development after a release: `hg update; (edit); hg ci -m "message"`

[6] Those users who want external verification can restrict themselves to the tagged releases - potentially GPG signed by trusted 3rd-party reviewers. GPG signatures are treated like hotfixes: reviewers sign on stable (via `hg sign` without options) and merge into default. Signing directly on stable reduces the possibility of signing the wrong revision.

- **Feature Branches**
  - start a larger feature: `hg branch feature-x; (edit); hg ci -m "message"`
  - continue with the feature: `hg update feature-x; (edit); hg ci -m "message"`
  - merge the feature: `hg update default; hg merge feature-x; hg ci -m "merged feature x into default"`
  - close and merge the feature when you are done: `hg update feature-x; hg ci --close-branch -m "finished feature x"; hg update default; hg merge feature-x; hg ci -m "merged finished feature x into default"`
- **Tasks for Maintainers**
  - *Initialize (only needed once)*
    - create the repo: `hg init reponame; cd reponame`
    - first commit: `(edit); hg ci -m "message"`
    - create the stable branch and do the first release: `hg branch stable; hg tag tagname; hg up default; hg merge stable; hg ci -m "merge stable into default:  ready for more development"`
  - apply a hotfix[7]: `hg up stable; (edit); hg ci -m "message"; hg up default; hg merge stable; hg ci -m "merge stable into default:  ready for more development"`
  - do a release[8]: `hg up stable; hg merge default; hg ci -m "(description of the main changes since the last release)" ; hg tag tagname; hg up default ; hg merge stable ; hg ci -m "merged stable into default: ready for more development"`

That's it. All that follows are a detailed example which goes through all actions one-by-one, adaptions to this workflow and the final summary.

[7] Here a hotfix is defined as a fix which must be applied quickly out-of-order, for example to fix a security hole. It prompts a bugfix-release which only contains already stable and tested changes plus the hotfix.

[8] If your project needs a certain **release preparation phase** (like translations), then you can simply assign a task branch. Instead of merging to stable, you merge to the task branch, and once the task is done, you merge the task branch to stable. An Example: Assume that you need to update *translations* before you release anything. *(next part: init: you only need this once)* When you want to do the *first release* which needs to be translated, you update to the revision from which you want to make the release and create the "translation" branch: `hg update default; hg branch translation; hg commit -m "prepared the translation branch"`. All translators now update to the translation branch and do the translations. Then you merge it into stable: `hg update stable; hg merge translation; hg ci -m "merged translated source for release"`. After the release you merge stable back into default as usual. *(regular releases)* If you want to start translating the next time, you just merge the revision to release into the translation branch: `hg update translation; hg merge default; hg commit -m "prepared translation branch"`. Afterwards you merge "translation" into stable and proceed as usual.

3 Example

This is the output of a complete example run [9] of the branching model, including all complications you should ever hit.

We start with the full history. In the following sections, we will take it apart to see what the commands do. So just take a glance, take in the basic structure and then move on for the details.

```
hg log -G
@    changeset:   15:855a230f416f
|\   tag:         tip
| |  parent:      13:e7f11bbc756c
| |  parent:      14:79b616e34057
| |  summary:     merged stable into default: ready for more development
| |
| o  changeset:   14:79b616e34057
|/|  branch:      stable
| |  parent:      7:e8b509ebeaa9
| |  parent:      13:e7f11bbc756c
| |  summary:     merged default into stable for release
| |
o |    changeset:   13:e7f11bbc756c
|\ \   parent:      11:e77a94df3bfe
| | |  parent:      12:aefc8b3a1df2
| | |  summary:     merged finished feature x into default
| | |
| o |  changeset:   12:aefc8b3a1df2
| | |  branch:      feature-x
| | |  parent:      9:1dd6209b2a71
| | |  summary:     finished feature x
```

[9] The output is complete except for *user* and *date*, because those were unnecessarily repetitive. To run the example and check the output yourself, just copy-paste the following your shell: `HGPLAIN=1 LC_ALL=C sh -c 'hg init test-branch; cd test-branch; echo 1 > 1; hg ci -Am 1; hg branch stable; hg tag v1 ; hg up default; hg merge stable; hg ci -m "merged stable into default: ready for more development"; echo 2 > 2; hg ci -Am 2; hg up stable; echo 1.1 > 1; hg ci -Am hotfix; hg up default; hg merge stable; hg ci -m "merge stable into default: ready for more development"; hg up stable; hg merge default; hg ci -m "merge default into stable for release" ; hg tag v2; hg up default ; hg merge stable ; hg ci -m "merged stable into default: ready for more development" ; hg branch feature-x; echo x > x ; hg ci -Am x; hg up default; echo 3 > 3; hg ci -Am 3; hg merge feature-x; hg ci -m "merged feature x into default"; hg update feature-x; hg ci --close-branch -m "finished feature x"; hg update default; hg merge feature-x; hg ci -m "merged finished feature x into default"; hg up stable ; hg merge default; hg ci -m "merged default into stable for release"; hg up default; hg merge stable ; hg ci -m "merged stable into default: ready for more development"; hg log -G'`

```
| | |
o | |  changeset:   11:e77a94df3bfe
|\| |  parent:      10:8c423bc00eb6
| | |  parent:      9:1dd6209b2a71
| | |  summary:     merged feature x into default
| | |
o | |  changeset:   10:8c423bc00eb6
| | |  parent:      8:dc61c2731eda
| | |  summary:     3
| | |
| o |  changeset:   9:1dd6209b2a71
|/ /   branch:      feature-x
| |    summary:     x
| |
o |  changeset:   8:dc61c2731eda
|\|  parent:      5:4c57fdadfa26
| |  parent:      7:e8b509ebeaa9
| |  summary:     merged stable into default: ready for more development
| |
| o  changeset:   7:e8b509ebeaa9
| |  branch:      stable
| |  summary:     Added tag v2 for changeset 089fb0af2801
| |
| o  changeset:   6:089fb0af2801
|/|  branch:      stable
| |  tag:         v2
| |  parent:      4:d987ce9fc7c6
| |  parent:      5:4c57fdadfa26
| |  summary:     merge default into stable for release
| |
o |  changeset:   5:4c57fdadfa26
|\|  parent:      3:bc625b0bf090
| |  parent:      4:d987ce9fc7c6
| |  summary:     merge stable into default: ready for more development
| |
| o  changeset:   4:d987ce9fc7c6
| |  branch:      stable
| |  parent:      1:a8b7e0472c5b
| |  summary:     hotfix
| |
o |  changeset:   3:bc625b0bf090
| |  summary:     2
| |
o |  changeset:   2:3e8df435bcb0
```

```
|\|  parent:      0:f97ea6e468a1
| |  parent:      1:a8b7e0472c5b
| |  summary:     merged stable into default: ready for more development
| |
| o  changeset:   1:a8b7e0472c5b
|/   branch:      stable
|    summary:     Added tag v1 for changeset f97ea6e468a1
|
o  changeset:   0:f97ea6e468a1
   tag:         v1
   summary:     1
```

4 Action by action

Let's take the log apart to show the actions contributors will do.

4.1 Initialize

Initializing and doing the first commit creates the first changeset:

```
o  changeset:   0:f97ea6e468a1
   tag:         v1
   summary:     1
```

Nothing much to see here.

Commands:

```
hg init test-branch; cd test-branch
(edit); hg ci -m "message"
```

4.2 Stable branch and first release

We add the first tagging commit on the stable branch as release and merge back into default:

```
o    changeset:   2:3e8df435bcb0
|\   parent:      0:f97ea6e468a1
| |  parent:      1:a8b7e0472c5b
| |  summary:     merged stable into default: ready for more development
| |
| o  changeset:   1:a8b7e0472c5b
|/   branch:      stable
|    summary:     Added tag v1 for changeset f97ea6e468a1
```

```
|
o  changeset:   0:f97ea6e468a1
   tag:         v1
   summary:     1
```

Mind the tag field which is now shown in changeset 0 and the branchname for changeset 1. This is the only release which will ever be on the default branch (because the stable branch only starts to exist after the first commit on it: The commit which adds the tag).

Commands:

```
hg branch stable
hg tag tagname
hg up default
hg merge stable
hg ci -m "merged stable into default: ready for more development"`
```

4.3 Further development

Now we just chuck along. The one commit shown here could be an arbitrary number of commits.

```
o    changeset:   3:bc625b0bf090
|    summary:     2
|
o    changeset:   2:3e8df435bcb0
|\   parent:      0:f97ea6e468a1
| |  parent:      1:a8b7e0472c5b
| |  summary:     merged stable into default: ready for more development
```

Commands:

```
(edit)
hg ci -m "message"
```

4.4 Hotfix

If a hotfix has to be applied to the release out of order, we just update to the stable branch, apply the hotfix and then merge the stable branch into default[10]*. This gives us changesets 4 for the hotfix and 5 for the merge (2 and 3 are shown as reference).*

[10] We merge the hotfix into default to define the relevance of the fix for general development. If the hotfix also affects the current line of development, we keep its changes in the merge. If the current line of development does not need the hotfix, we discard its changes in the merge. We do this to ensure that it is clear in future how to treat the hotfix when merging new changes: let the merge record the decision.

```
o    changeset:   5:4c57fdadfa26
|\   parent:      3:bc625b0bf090
| |  parent:      4:d987ce9fc7c6
| |  summary:     merge stable into default: ready for more development
| |
| o  changeset:   4:d987ce9fc7c6
| |  branch:      stable
| |  parent:      1:a8b7e0472c5b
| |  summary:     hotfix
| |
o |  changeset:   3:bc625b0bf090
| |  summary:     2
| |
o |  changeset:   2:3e8df435bcb0
|\|  parent:      0:f97ea6e468a1
| |  parent:      1:a8b7e0472c5b
| |  summary:     merged stable into default: ready for more development
```

Commands:

```
hg up stable
(edit)
hg ci -m "message"
hg up default
hg merge stable
hg ci -m "merge stable into default: ready for more development"
```

4.5 Regular release

To do a regular release, we just merge the default branch into the stable branch and tag the merge. Then we merge stable back into default. This gives us changesets 6 to 8[11]*. The commit-message you use for the merge to stable will become the description for your tag, so you should choose a good description instead of "merge default into stable for release". Userfriendly, simplified release notes would be a good choice.*

```
o    changeset:   8:dc61c2731eda
|\   parent:      5:4c57fdadfa26
| |  parent:      7:e8b509ebeaa9
| |  summary:     merged stable into default: ready for more development
| |
| o  changeset:   7:e8b509ebeaa9
```

[11] We can also merge to stable regularly as soon as some set of changes is considered stable, but without making an actual release (==tagging). That way we always have a stable branch which people can test without having to create releases right away. The releases are those changesets on the stable branch which carry a tag.

```
| |  branch:      stable
| |  summary:     Added tag v2 for changeset 089fb0af2801
| |
| o  changeset:   6:089fb0af2801
|/|  branch:      stable
| |  tag:         v2
| |  parent:      4:d987ce9fc7c6
| |  parent:      5:4c57fdadfa26
| |  summary:     merge default into stable for release
| |
o |  changeset:   5:4c57fdadfa26
|\|  parent:      3:bc625b0bf090
| |  parent:      4:d987ce9fc7c6
| |  summary:     merge stable into default: ready for more development
```

Commands:

```
hg up stable
hg merge default
hg ci -m "merge default into stable for release"
hg tag tagname
hg up default
hg merge stable
hg ci -m "merged stable into default: ready for more development"
```

4.6 Feature branches

Now we want to do some larger development, so we use a feature branch. The one feature-commit shown here (x) could be an arbitrary number of commits, and as long as you stay in your branch, the development of your colleagues will not disturb your own work. Once the feature is finished, we merge it into default. The feature branch gives us changesets 9 to 13 (with 10 being an example for an unrelated intermediate commit on default).

```
o    changeset:   13:e7f11bbc756c
|\   parent:      11:e77a94df3bfe
| |  parent:      12:aefc8b3a1df2
| |  summary:     merged finished feature x into default
| |
| o  changeset:   12:aefc8b3a1df2
| |  branch:      feature-x
| |  parent:      9:1dd6209b2a71
| |  summary:     finished feature x
| |
o |  changeset:   11:e77a94df3bfe
```

```
|\|  parent:      10:8c423bc00eb6
| |  parent:      9:1dd6209b2a71
| |  summary:     merged feature x into default
| |
o |  changeset:   10:8c423bc00eb6
| |  parent:      8:dc61c2731eda
| |  summary:     3
| |
| o  changeset:   9:1dd6209b2a71
|/   branch:      feature-x
|    summary:     x
|
o    changeset:   8:dc61c2731eda
|\   parent:      5:4c57fdadfa26
| |  parent:      7:e8b509ebeaa9
| |  summary:     merged stable into default: ready for more development
```

Commands:

- Start the feature

  ```
  hg branch feature-x
  (edit)
  hg ci -m "message"
  ```

- Do an intermediate commit on default

  ```
  hg update default
  (edit)
  hg ci -m "message"
  ```

- Continue working on the feature

  ```
  hg update feature-x
  (edit)
  hg ci -m "message"
  ```

- Merge the feature

  ```
  hg update default
  hg merge feature-x
  hg ci -m "merged feature x into default"`
  ```

- Close and merge a finished feature

  ```
  hg update feature-x
  hg ci --close-branch -m "finished feature x"
  hg update default; hg merge feature-x
  hg ci -m "merged finished feature x into default"
  ```

Note: Closing the feature branch hides that branch in the output of `hg branches` *(except when using* `--closed`*) to make the repository state lean and simple while still keeping the feature branch information in history. It shows your collegues, that they no longer have to keep the feature in mind as soon as they merge the most recent changes from the default branch into their own feature branches.*

Note: To make the final merge of your feature into default easier, you can regularly merge the default branch into the feature branch.

Note: We use feature branches to ensure that new clones start at a revision which other developers can directly use. With bookmarks you could get trapped on a feature-head which might not be merged to `default` *for quite some time. For more reasons, see the bookmarks footnote[4].*

The final action is to have a maintainer do a regular merge of `default` into `stable` to reach a state from which we could safely do a release. Since we already showed how to do that, we are finished here.

5 Adaptions

This realizes the successful Git branching model[12] with Mercurial while maintaining one release at any given time.

[12] If you look at the Git branching model which inspired this Mercurial branching model, you'll note that its diagram is a lot more complex than the diagram of this Mercurial version. The reason for that is the more expressive history model of Mercurial. In short: The git version has 5 types of branches: feature, develop, release, hotfix and master (for tagging). With Mercurial you can reduce them to 3: default, stable and feature branches:

- Tags are simple in-history objets, so we need no special branch for them: a tag signifies a release (down to 4 branch-types - and no more duplication of information, since in the git-model a release is shown by a tag *and* a merge to master).
- Hotfixes are simple commits on `stable` followed by a merge to `default`, so we also need no branch for them (down to 3 branch-types). And if we only maintain one release at a time, we only need one branch for them: stable (down from branch-type to single branch).
- And feature branches are not required for clean separation since mercurial can easily cope with multiple heads in a branch, so developers only have to worry about them if they want to use them (down to 2 mandatory branches).
- And since the default branch is the branch to which you update automatically when you clone a repository, *new developers don't have to worry about branches at all.*

So we get down from 5 mandatory branches (2 of them are categories containing multiple branches) to 2 simple branches without losing functionality. And new developers only need to know two things about our branching model to contribute:

> "If you use feature branches, don't call them `default` or `stable`. And don't touch `stable`".

If you have special needs, this model can easily be extended to fullfill your requirements. Useful extensions include:

- multiple releases - if you need to provide maintenance for multiple releases side-by-side.
- grafted micro-releases - if you need to segment the next big changes into smaller releases while leaving out some potentially risky changes.
- explicit review - if you want to ensure that only reviewed changes can get into a release, while making it possible to leave out some already reviewed changes from the next releases. Review gets decoupled from releasing.

All these extensions are orthogonal, so you can use them together without getting side-effects.

5.1 Multiple maintained releases

To use the branching model with multiple simultaneously maintained releases,[13] you only need to change the hotfix procedure: When applying a hotfix, you go back to the old release with `hg update tagname`, fix there, add a new tag for the fixed release and then update to the next release. There you merge the new fix-release and do the same for all other releases. If the most recent release is not the head of the stable branch, you

[13] If you want to adapt the model to multiple very distinct releases, simply add multiple release-branches (i.e. release-x). Then hg graft the changes you want to use from default or stable into the releases and merge the releases into stable to ensure that the relationship of their changes to current changes is clear, recorded and will be applied automatically by Mercurial in future merges.[14] If you use multiple tagged releases, you need to merge the releases into each other in order - starting from the oldest and finishing by merging the most recent one into stable - to record the same information as with release branches. Additionally it is considered impolite to other developers to keep multiple heads in one branch, because with multiple heads other developers do not know the canonical tip of the branch which they should use to make their changes - or in case of stable, which head they should merge to for preparing the next release. That's why you are likely better off creating a branch per release, if you want to maintain many very different releases for a long time. If you only use tags on stable for releases, you need one merge per maintained release to create a bugfix version of one old release. By adding release branches, you reduce that overhead to one single merge to stable per affected release by stating clearly, that changes to old versions should never affect new versions, except if those changes are explicitely merged into the new versions. If the bugfix affects all releases, release branches require two times as many actions as tagged releases, though: You need to graft the bugfix into every release and merge the release into stable.[15]

[14] If for example you want to ignore that change to an old release for new releases, you simply merge the old release into stable and `use hg revert --all -r stable` before committing the merge.

[15] A rule of thumb for deciding between tagged releases and release branches is: If you only have a few releases you maintain at the same time, use tagged releases. If you expect that most bugfixes will apply to all releases, starting with some old release, just use tagged releases. If bugfixes will only apply to one release and the current development, use tagged releases and merge hotfixes only to stable. If most bugfixes will only apply to one release *and not to the current development*, use release branches.

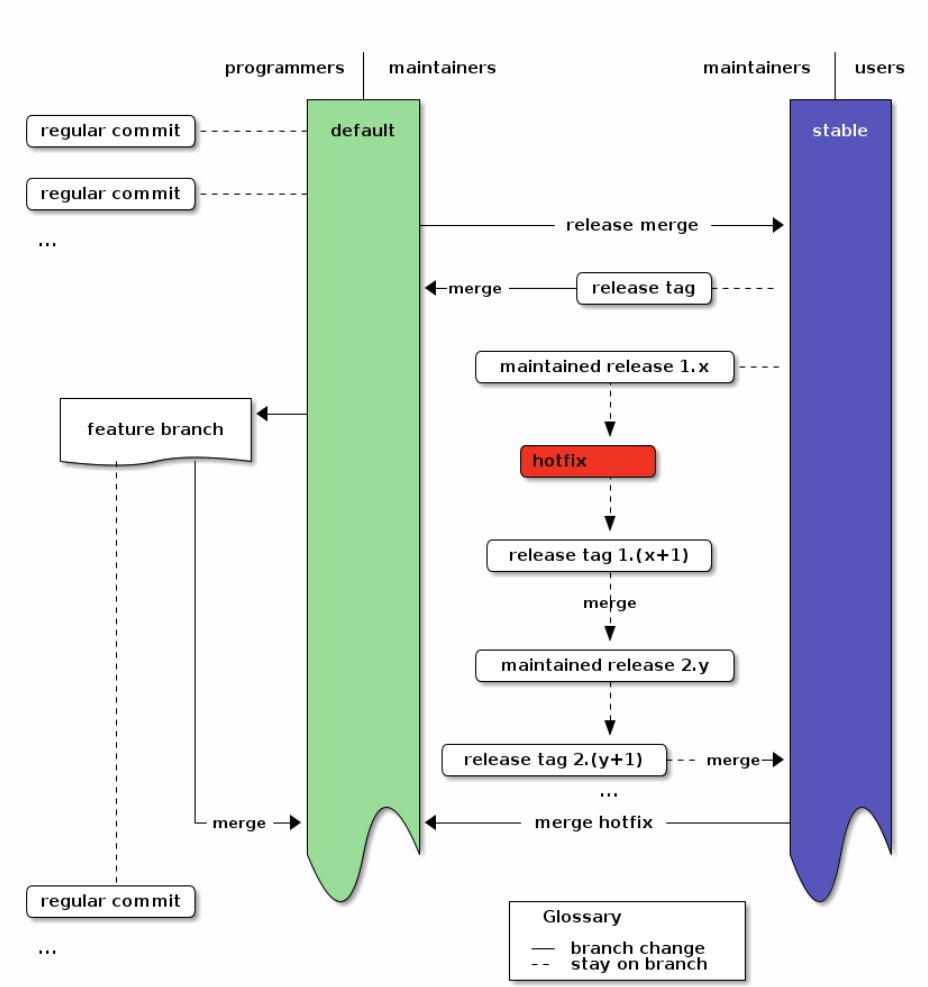
also merge into stable. Then you merge the stable branch into default, as for a normal hotfix.[16]

With this merge-chain you don't need special branches for releases, but all changesets are still clearly recorded. This simplification over git is a direct result of having real anonymous branching in Mercurial.

```
hg update release-1.0
(edit)
hg ci -m "message"
hg tag release-1.1
hg update release-2.0
hg merge release-1.1
hg ci -m "merged changes from release 1.1"
hg tag release-2.1
... and so on
```

In the Diagram this just adds a merge path from the hotfix to the still maintained releases. *Note that nothing changed in the workflow of programmers.*

[16] Merging old releases into new ones sounds like a lot of work. If you get that feeling, then have a look how many releases you really maintain right now. In my Gentoo tree most programs actually have only one single release, so using actual release branches would incur an additional burden without adding real value. You can also look at the rule of thumb whether to choose feature branches instead.[15]



An overview of the branching strategy with maintained releases.

5.2 Graft changes into micro-releases

If you need to test parts of the current development in small chunks, you can graft micro releases. In that case, just update to stable and merge the first revision from default, whose child you do not want, and graft later changes[17].

[17] If you want to make sure that every changeset on `stable` is production-ready, you can also start a new release-branch on stable, then merge the first revision, whose child you do not want, into that branch and graft additional changes. Then close the branch and merge it into stable. You can achieve the same with much lower overhead (unneeded complexity) by changing the requirement to "every tagged revision on `stable` is production-ready". To only see tagged revisions on stable, just use `hg log -r "branch(stable) and tag()"`. This also works for incoming and outgoing, so you can use it for triggering a build system.

Example for the first time you use micro-releases[18]:

> You have changes 1, 2, 3, 4 and 5 on default. First you want to create a release which contains 1 and 4, but not 2, 3 or 5.
>
> hg update 1 hg branch stable hg graft 4
>
> As usual tag the release and merge stable back into default:
>
> hg tag rel-14 hg update default hg merge stable hg commit -m "merge stable into default. ready for more development"

Example for the second and subsequent releases:

> Now you want to release the change 2 and 5, but you're still not ready to release 3. So you merge 2 and graft 5.
>
> hg update stable hg merge 2 hg commit -m "merge all changes until 2 from default" hg graft 5
>
> As usual tag the release and finally merge stable back into default:
>
> hg tag rel-1245 hg update default hg merge stable hg commit -m "merge stable into default. ready for more development"

The history now looks like this[19]:

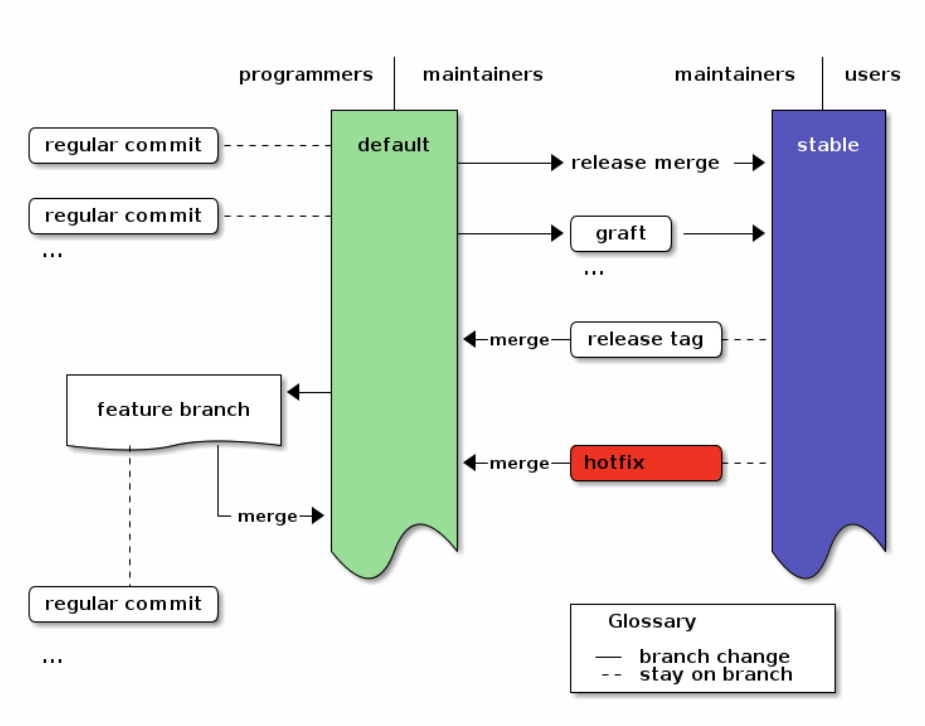
```
@    merge stable into default. ready for more development (default)
|\
| o  Added tag rel-1245 for changeset 4e889731c6ca (stable)
| |
| o  5 (stable)
| |
| o    merge all changes until 2 from default (stable)
| |\
o---+  merge stable into default. ready for more development (default)
| | |
| | o  Added tag rel-14 for changeset cc2c95dd3f27 (stable)
| | |
| | o  4 (stable)
| | |
o | |  5 (default)
| | |
o | |  4 (default)
| | |
```

[18] To test this workflow yourself, just create the test repository with `hg init 12345; cd 12345; for i in {0..5}; do echo $i > $i; hg ci -Am $i; done`.

[19] The short graphlog for the grafted micro-releases was created via `hg glog --template "{desc} ({branch})"`.

```
o | |  3 (default)
|/ /
o /  2 (default)
|/
o  1 (default)
|
o  0 (default)
```

In the Diagram this just adds graft commits to stable:



An overview of the branching strategy with grafted micro-releases.

Grafted micro-releases add another layer between development and releases. They can be necessary in cases where testing requires actually deploying a release, as for example in Freenet.

5.3 Explicit review branch

If you want to add a separate review stage, you can use a review branch[20, 21] into which you only merge or graft reviewed changes. The review branch then acts as a staging area for all changes which might go into a release.

To use this extension of the branching model, just create a branch on default called `review` in which you merge or graft reviewed changes. The first time you do that, you update to the first commit whose children you do not want to include. Then create the review branch with `hg branch review` and use `hg graft REV` to pull in all changes you want to include.

On subsequent reviews, you just update to review with hg update nextrelease, merge the first revision which has a child you do not want with `hg merge REV` and graft additional later changes with `hg graft REV` as you would do it for micro-releases..

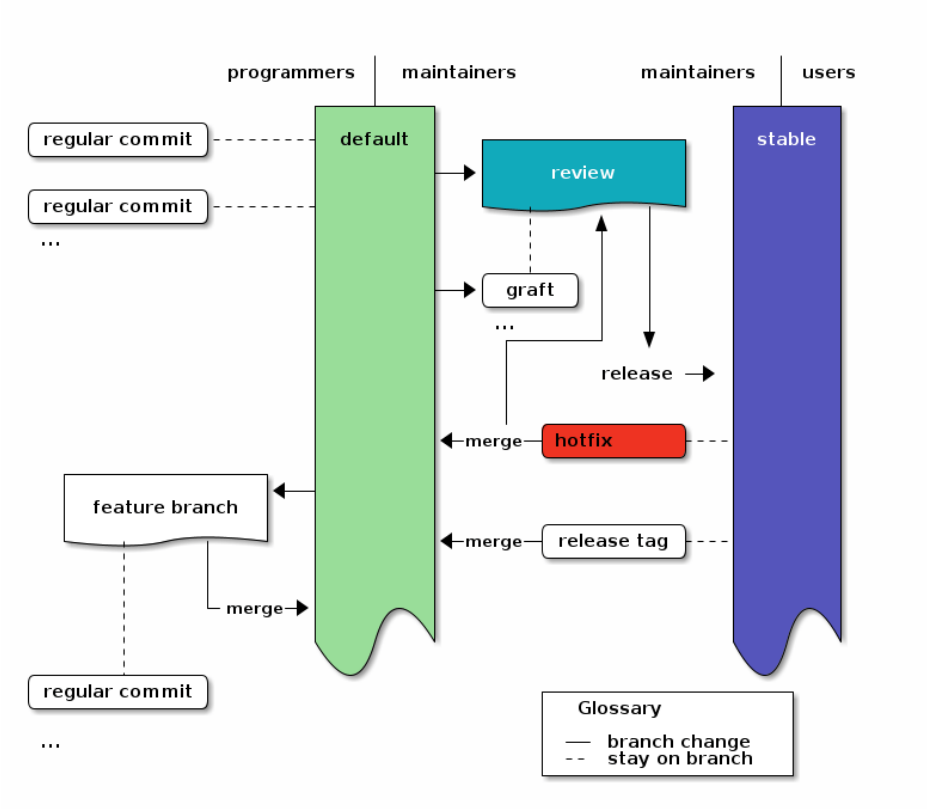
In both cases you create the release by merging the `review` branch into stable.

A special condition when using a review branch is that you always have to merge hotfixes into the review branch, too, because the review branch does not automatically contain all changes from the default branch.

In the Diagram this just adds the `review` branch between `default` and `stable` instead of the release merge. Also it adds the hotfix merge to the `review` branch.

[20] The review branch is a special preparation-branch,[8] because it can get discontinous changes, if maintainers decide to graft some changes which have ancestors they did not review yet.

[21] We use one single review branch which gets reused at every review to ensure that there are no changes in stable which we did not have in the review. As alternative, you could use one branch per review. In that case, ensure that you start the review-* branches from `stable` and not from `default`. Then merge and graft the changes from default which you want to review for inclusion in your next release.



An overview of the branching strategy with a `review` branch.

6 Frequently Asked Questions (FAQ)

6.1 Where does QA (Quality Assurance) come in?

In the default flow when the users directly use the stable branch you do QA on the default branch before merging to stable. QA is a part of the maintainers job, there.

If your users want external QA,[6] that QA is done for revisions on the stable branch. It is restricted to signing good revisions. Any changes have to be done on the default branch - except for hotfixes for previously signed releases. It is only a hotfix, if your users could already be running a broken version.

There is also an extension with an explicit review branch. There QA is done on the review branch.

7 Summary

This realizes the successful Git branching model with Mercurial.

We now have nice graphs, examples, potential extensions and so on. But since this strategy uses Mercurial instead of git, we don't actually need all the graphics, descriptions and branch categories in the git version - or in this post.

Instead we can boil all of this down to **3 simple rules**:

1. you do all the work on `default` - except for hotfixes.
1. on `stable` you only do hotfixes, merges for release and tagging for release. Only maintainers touch stable.
1. you can use arbitrary feature-branches, as long as you don't call them `default` or `stable`. They always start at default (since you do all the work on default).

They are the rules you already know from the starting summary. Keep them in mind and you're good to go. And when you're doing *regular development*, there is only one rule to remember:

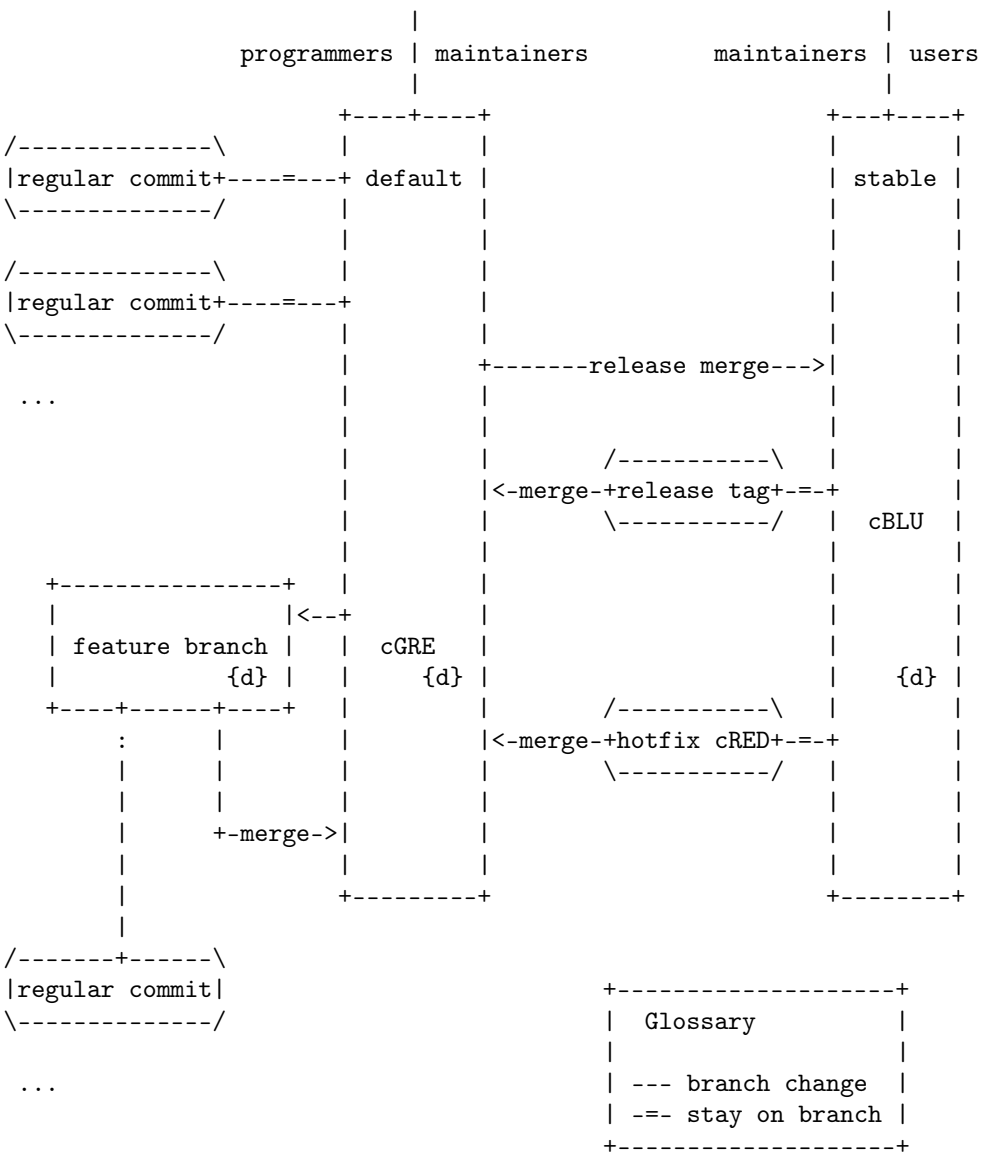
> *You do all the work on default.*

That's it. Happy hacking!

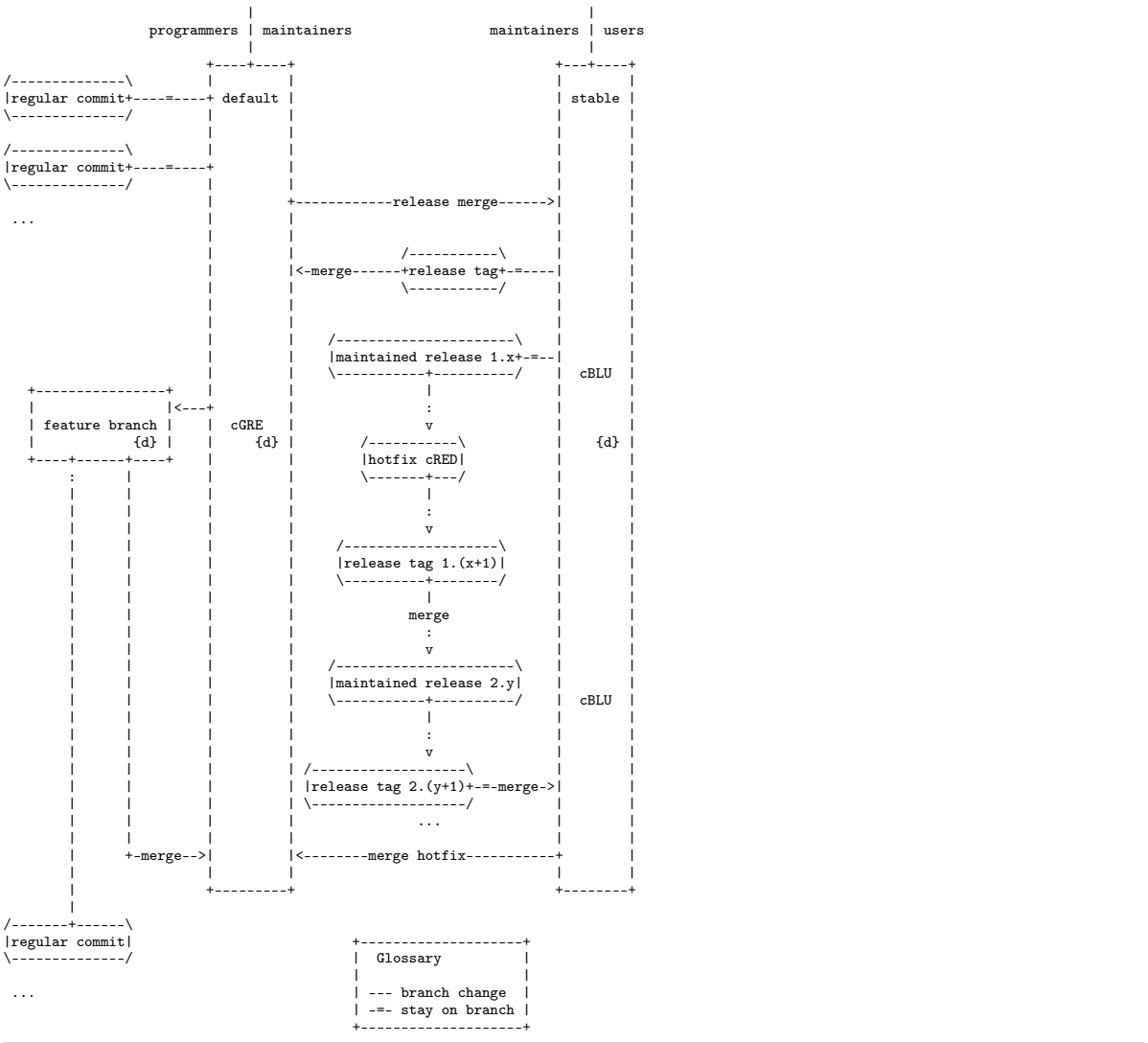
8 Appendix: Diagrams

Diagrams for the hg branching strategy

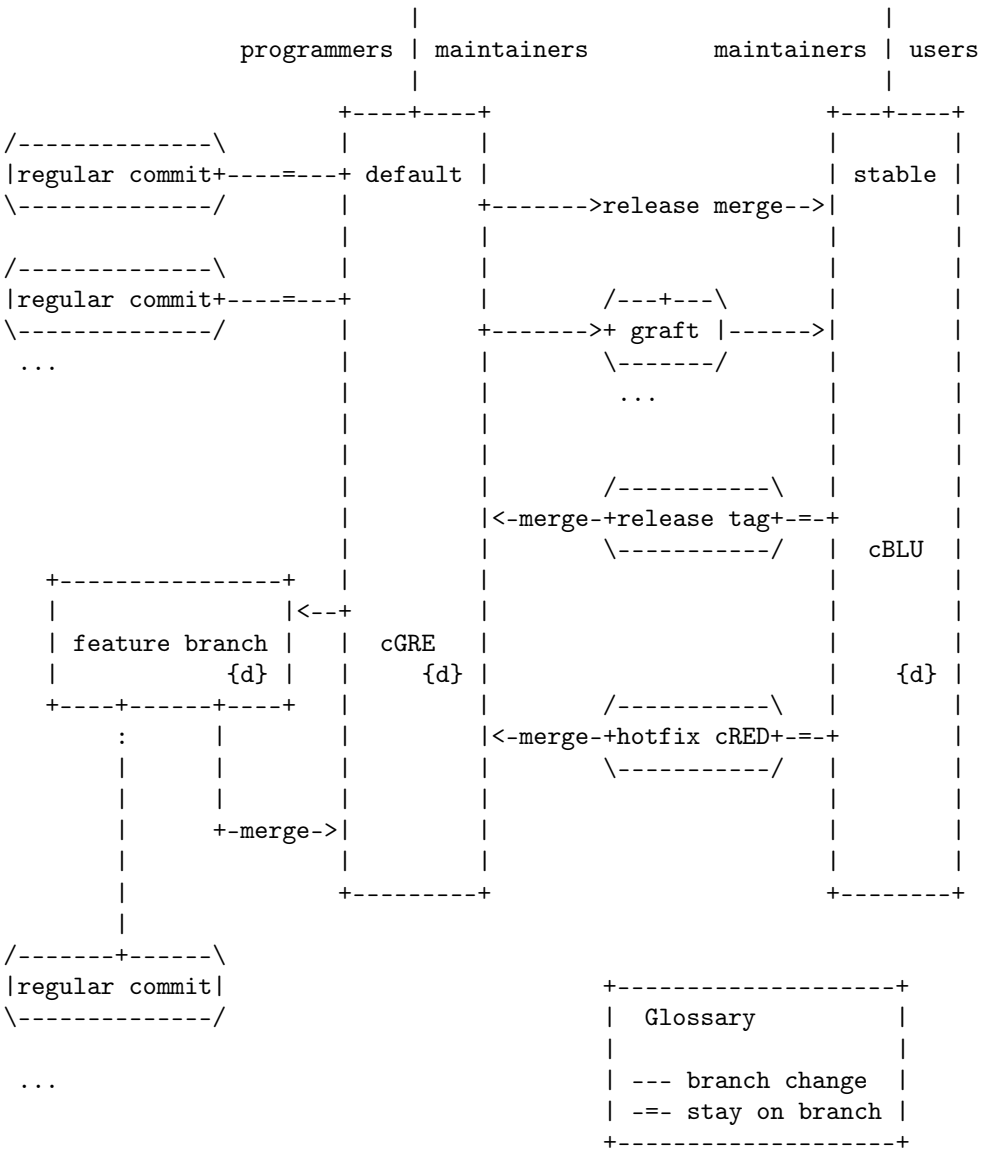
8.1 Base workflow



8.2 Maintain several releases side-by-side



8.3 Graft releases



8.4 Use a review branch