Adapt Learning: Gitflow process

Document control

Abstract:	Presents Totara Social's design goals to ensure subsequent design and development meets the needs of end-users.		
Author:	Fabien O'Carroll, Sven Laux	**Version:** 0.1	**Date:** 02 / 10 / 2013

Summary of Changes:	**Versions**	**Date**	**Description**
	0.1	02 / 10 / 2013	Initial draft for review

Contents

Purpose of document

This document has been written to outline and define the version control process we will be using for the Adapt Learning open source project. It is aimed at anyone that would like to get involved by contributing code. We will be using Git to version control the codebase.

Our aim is to keep a clean but informative history of commits, allowing us to revert mistakes easily.

Glossary of terms

This section describes the basic terms we use and assume knowledge of. Full documentation of Git terms can be found at: http://git-scm.com/docs

Term	**Description**
Repository	A Git repository is essentially a collection of branches, all related to the same code base.
Commit	A commit in Git can be thought of as a snapshot of the repository at any given point. Commits can be referenced by SHA1 checksums, tags or branches.
Commit message	A commit message is the message that describes the changes made in a commit. viewable with `git log`. Please see commit section below.
Branch	A branch in Git can be thought of as a sequence of commits which can be separate from all other commits. A branch may be merged or rebased onto another branch to introduce its commit history to other branch.
Master	The 'master' branch is the parent branch of all other branches, including 'develop'. In our process, it will receive only 2 types of commits: *Hotfix*es and major updates. The 'master' branch will always be contain stable code and will be what the public will receive.
Develop	The 'develop' branch is a branch spun from the 'master' branch. It is where all feature branches will be derived from. The code contained in 'develop' should always be stable. This branch will be the source of the latest codebase, including nightly builds for example. 'Develop' will receive all *Hotfixes* as well as 'master'.
Feature	A feature branch is a branch used to develop new functionality. Very large feature additions may require multiple feature branches to be created.
Hotfix	A hotfix branch is used for critical bug fixes. This type of branch is taken directly from 'master' and, once the work is completed, merged directly to 'master' and 'develop' using a --squash merge.
Release / release candidate	A release branch is used when working on large features which are split up into separate branches, or to test integration
Tag	A tag is the term used to define a textual label that can be associated with a specific revision. In the context of the Adapt Learning project, we use it to identify stable versions of the code base.

What is Gitflow?

Git flow is a workflow for developing with the Git version control system. It is how we structure our repository, our commits and branches.

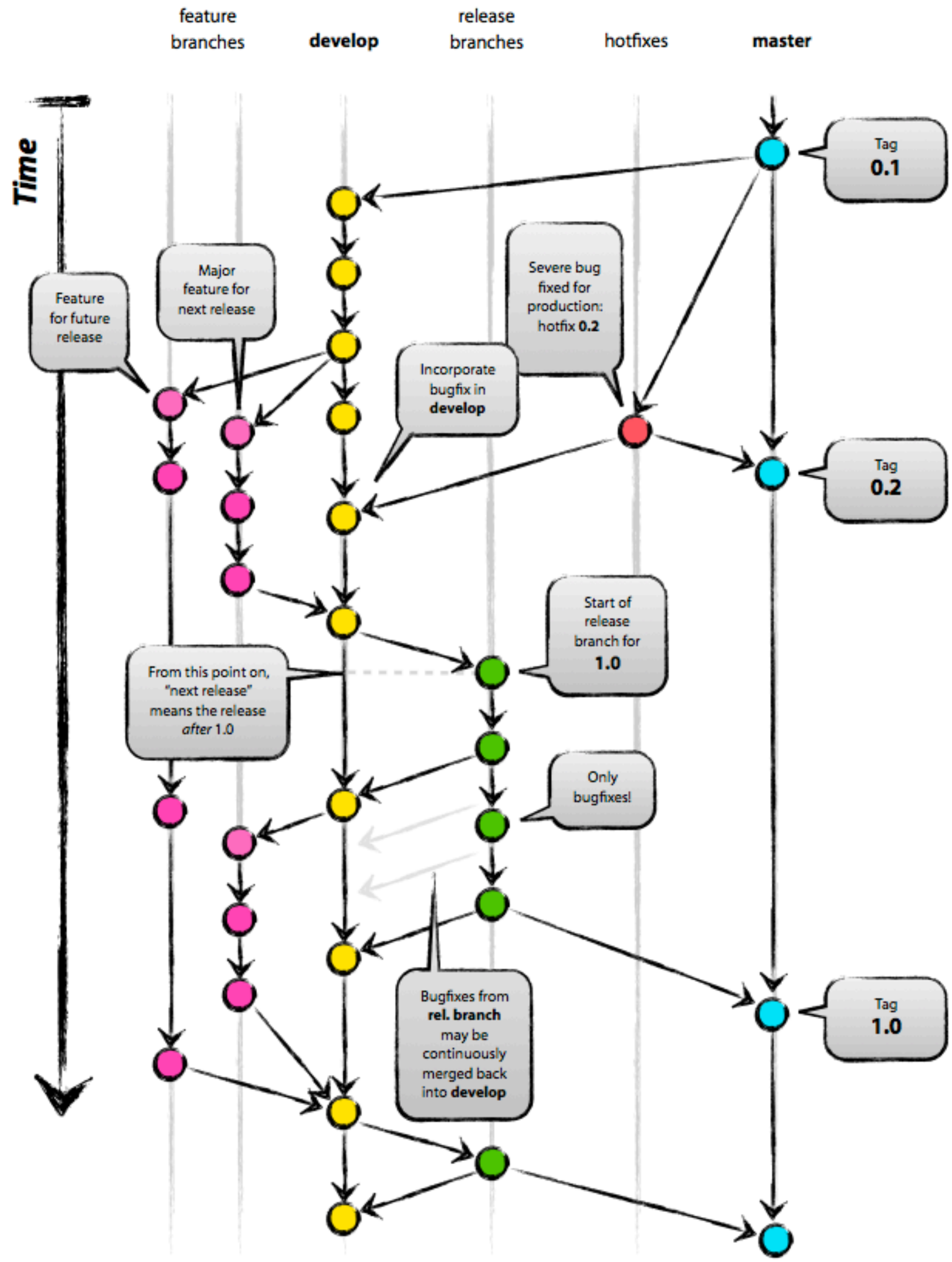
For the most part this document is based this on Vincent Driessen's article “A successful Git branching model” at http://nvie.com/posts/a-successful-git-branching-model/ and thus we will be using feature and hotfix branches, a main develop branch and we will up-merge local branches and tags.

The diagram on the next page (taken from the above URL with thanks to Vincent) shows an overview of this process.

The only difference we apply to the process outlined by Vincent is that we will **not** use the --no-ff merge. This means that when merging feature branches into master and develop, we are breaking the commit history up into manageable chunks. This is based on and incorporates the advice on the article at https://sandofsky.com/blog/git-workflow.html

We will use tags only in the master branch. The develop branch will always aim to be stable and bugs will have tickets and a branch created to fix them. We will enforce this by running tests pre and post merge of the develop branch and code review for all merge requests.

On the positive side, **not** using --no-ff and using an interactive rebase means that if a bug is introduced we do not have to deal with a large amount of irrelevant commits as the commits will be sequenced in logical steps.


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix **0.2**
Incorporate bugfix in **develop**
Tag 0.2
Start of release branch for **1.0**
From this point on, "next release" means the release *after* 1.0
Only bugfixes!
Bugfixes from **rel. branch** may be continuously merged back into **develop**
Tag 1.0

How do we use Gitflow?

Repositories

There are two repositories:

- Authoring tool
- Output framework

The reason for this is to keep the commit histories clear and relevant. It also allows developers to on components in isolation, which is most relevant to the output framework (as they will not need to clone the whole authoring tool repository).

When the development of the authoring tool has begun, this document will be updated to inform developers how to checkout both of the repository's in relation to each other (e.g. whether they need to be checked out as siblings or parent/child etc.)

We will also produce futher documentation inlcuding specific instructions on how to get started and how to work with the authoring tool in particular, as this requires both codebases.

Branches

Master and develop branches

There will be two long-running branches - master and develop - these will be protected branches, meaning only a select few will have the ability to push changes directly to these branches. All additions to these branches must be done via a merge request (pull-request), this allows code review of everything that enters the framework and ensure both master and develop branches stay stable.

Feature branches

When developing new functionality or features, or simply refactoring/optimizing core code you should create a feature branch, feature branches are branched from the develop branch and named after their ticket number or a brief description of the work being done.

Naming convention:

Feature branches should always be prefixed with “feature/”, for example “feature/2345” or “feature/scorm_tracking”.

Hotfix branches

When a serious bug is found or a fix is needed immediately, you should create a hotfix branch. Hotfix branches are spun directly from ‘master’, tested as soon as possible and merged into the master branch using a --squash merge. This gives us a commit checkpoint of the fix being implemented. Hotfix branches should also be merged directly into ‘develop’.

Naming convention:

Hotfix branches should always be prefixed with “hotfix/”, for example “hotfix/mcq_multiple_attempts”

Creating branches

With the exception of Hotfixes, branches should only ever be spun off the ‘develop’ branch. The command line for creating a branch is:

```
$ git fetch origin develop

$ git checkout -b <fix_type>/<ticket_number> develop
```

Keeping updated

When working on a branch you should often up-merge from the parent branch. This means you merge any changes in ‘develop’ into your branch. It avoids conflicts when the time comes for your branch to be merged back in. The relevant commands are listed below:

If your branch is a local branch:

```
$ git fetch origin master

$ git rebase origin/master
```

If your branch is remote:

```
$ git pull origin master
```

Deleting branches

Hotfix branches should be deleted as soon as they are merged into both ‘master‘ and ‘develop’. Feature branches can be deleted once they have been tested, peer reviewed, merged into develop and retested.

Committing

We aim to follow the 50/72 rule for committing messages (http://stackoverflow.com/questions/2290016/git-commit-messages-50-72-formatting).

Key points:

- To keep each commit concise and direct
- To including why a change was made
- Capitalized, short (50 chars or less) summary
- More detailed explanatory text, if necessary. Wrap it to about 72 characters or so.
- To avoid mixing whitespace changes with functional changes as this will make the code review process longer.
- To speak in the imperative tense, not past, “Update” not “Updated”
- Further paragraphs come after blank lines
- Bullet points are okay, too

- Typically a hyphen or asterisk is used for the bullet, preceded by a single space, with blank lines in between, but conventions vary here
- Use a hanging indent

In some contexts, the first line is treated as the subject of an email and the rest of the text as the body. The blank line separating the summary from the body is critical (unless you omit the body entirely); tools like rebase can get confused if you run the two together.

Further reading about best practice for Git commit messages:
https://wiki.openstack.org/wiki/GitCommitMessages

http://ablogaboutcode.com/2011/03/23/proper-git-commit-messages-and-an-elegant-git-history/

Merging

As mentioned before, your private branches should be cleaned up using rebase -i if they are large features with many checkpoints, or just merged with --squash this means that each feature will come as a single chunk if appropriate, or manageable chunks if it is too large.

```
$ git fetch origin develop

$ git checkout develop

$ git merge ––squash feature/1234
```