G Git Workflow

Contents

G.2

Introduction

Git is a very successful software version control tool, mainly because of the way it implements *branch* and *merge* operations – they are extremely "cheap" in terms of time and storage, are simple for users to perform, and are now considered core parts of a programmer's *daily* workflow.

A *Git workflow* is a set of procedures that every team member has to follow in order to implement a prescribed software development process. There is no such thing as the "definitive" workflow. Workflows are arbitrary, and may be prescribed by the software team, or a company's software standards, or the size of a project, etc.

A common workflow model is shown graphically on the next page. Details of the workflow model are given in the following sections.

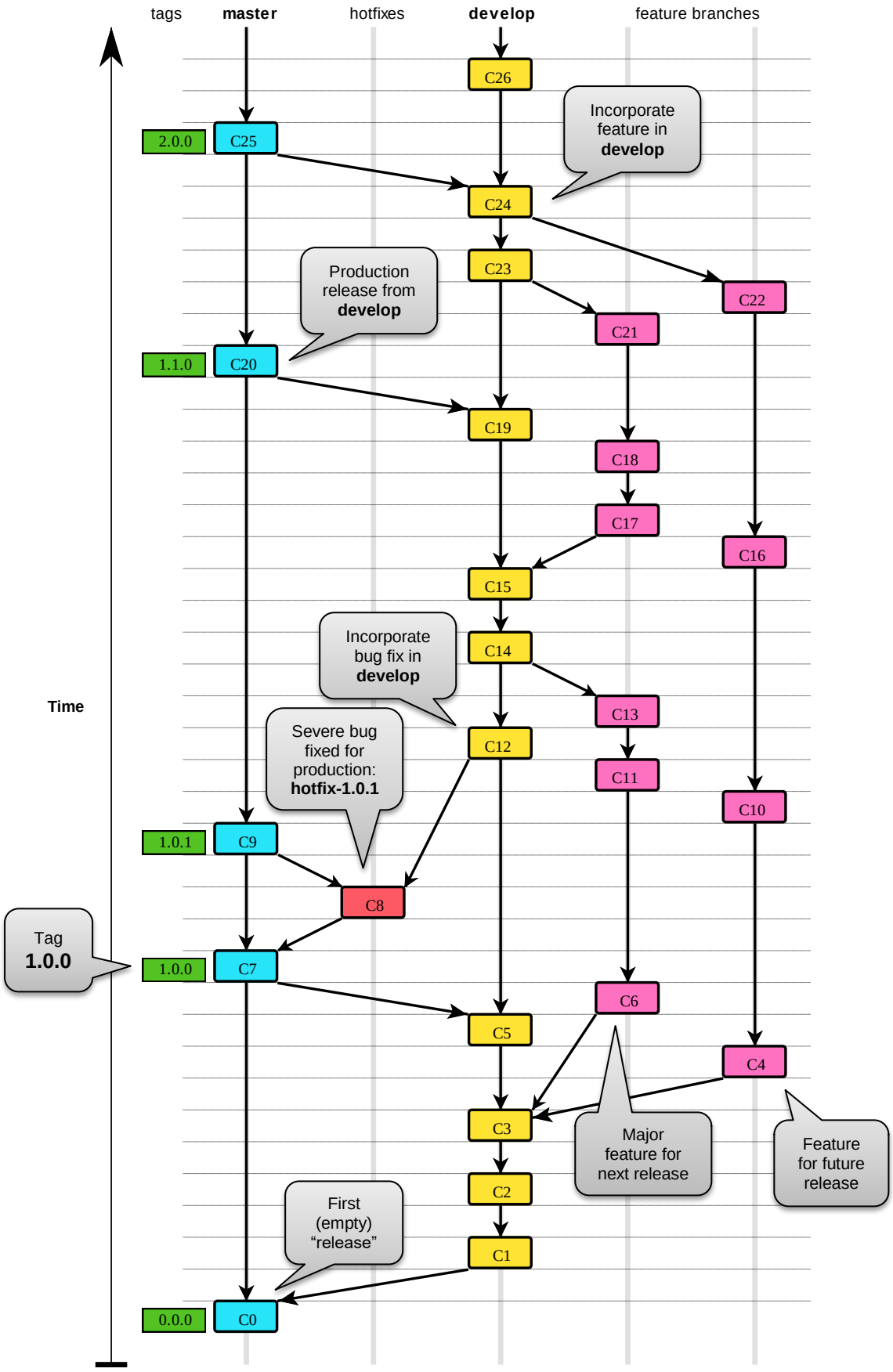

tags
master
hotfixes
develop
feature branches
Time
C26
Incorporate feature in develop
2.0.0
C25
C24
C23
Production release from develop
C22
C21
1.1.0
C20
C19
C18
C17
C16
C15
Incorporate bug fix in develop
C14
C13
Severe bug fixed for production: hotfix-1.0.1
C12
C11
C10
1.0.1
C9
C8
Tag 1.0.0
1.0.0
C7
C6
C5
C4
C3
Major feature for next release
Feature for future release
C2
First (empty) "release"
C1
0.0.0
C0

Figure G.1

G.1 The Main Branches

There are two main branches with an infinite lifetime:

- `master`
- `develop`

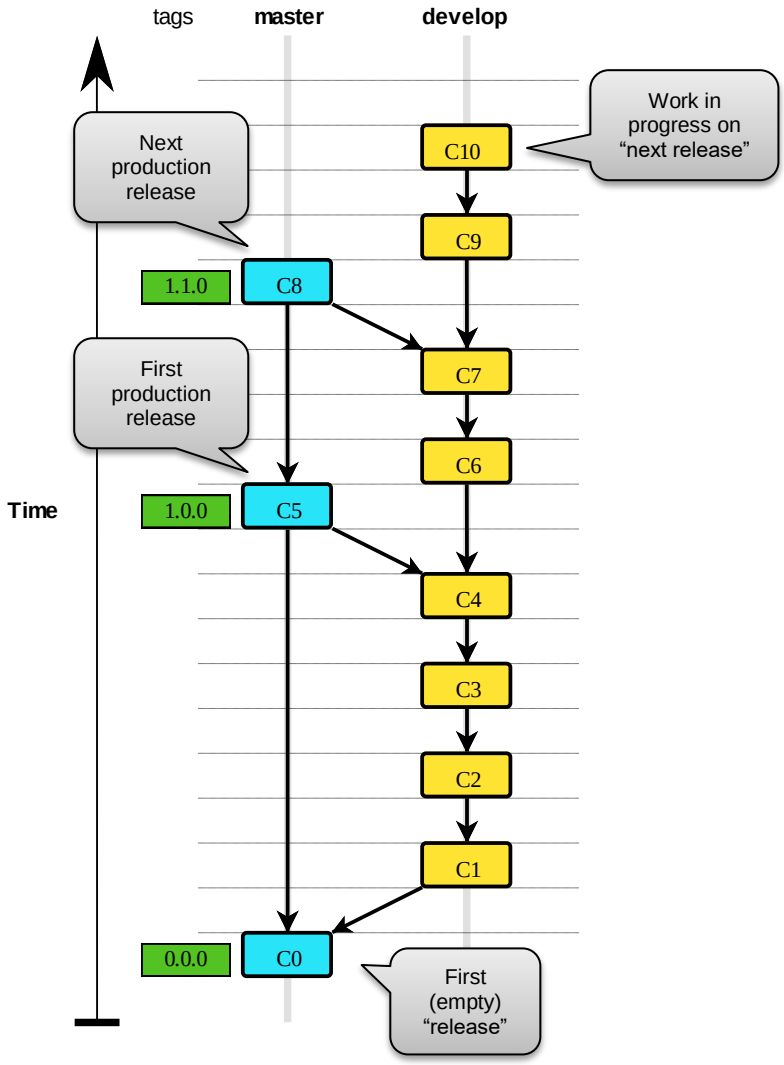


Figure G.2

The `master` branch at `origin` is a "default" branch of Git (the name can be changed on creation of a repository, but most users don't bother to change it). The `develop` branch needs to be created by the user, and runs in "parallel" to the `master` branch. The `origin/master` branch only holds *production-ready* commits, with each commit being *tagged* with a version number. The `HEAD` of this branch always reflects the latest release.

The `origin/develop` branch is the main branch used for development purposes. The source code at `HEAD` always reflects the latest state of the development changes.

When the source code in the `develop` branch reaches a stable point and is ready to be released, all of the changes are *merged* back into `master` and then tagged with a version number.

Although Git does support an "octopus" *merge* operation (a merge which is more than a 3-way merge), this workflow requires a "normal" 3-way merge from the `develop` branch into the `master` branch.

G.2 Supporting Branches

In addition to the main `master` and `develop` branches, the workflow model uses two supporting branches to aid parallel development between team members, ease tracking of features, and to assist in quickly fixing live production problems. Unlike the main branches, these branches always have a limited lifetime, since they will lie dormant after a merge (or optionally be removed entirely).

The supporting branches are:

- `feature/{name}`
- `hotfix-{version ID}`

Each of these branches has a specific purpose and are bound to strict rules as to which branches may be their originating branch and which branches must be their merge targets.

These branches are not "special" from a technical perspective – they are normal Git branches. The branch types are categorized by how we use them.

G.2.1 Feature Branches

May branch off from:

`develop`

Must merge back into:

`develop`

Branch naming convention:

`feature/{name}`

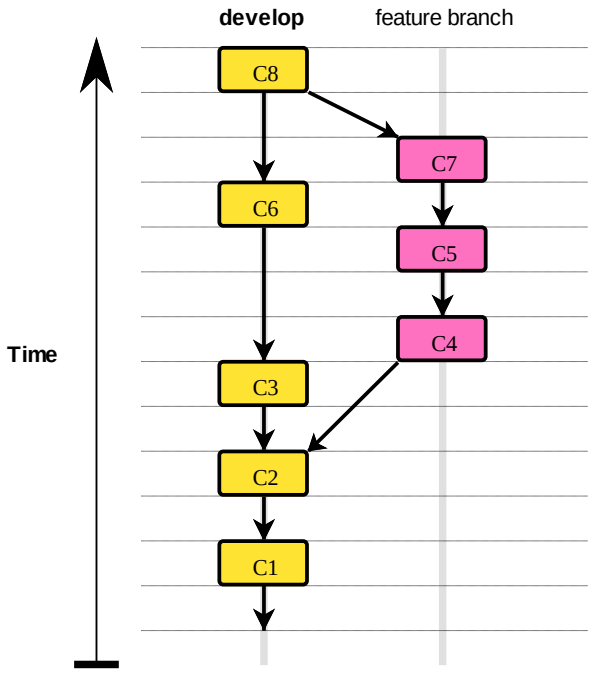


Figure G.3

Feature branches are used to develop new features for the 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 the case of a disappointing experiment).

Creating a Feature Branch

When starting work on a new feature, branch off from the `develop` branch.

Ending a Feature Branch

Finished features may be merged into the `develop` branch to definitely add them to the upcoming release.

Always use the "no fast forward" option (`--no-ff` flag) when merging back into `develop`. The `--no-ff` flag causes the merge to always create a new commit object, even if the merge could be performed with a fast-forward. This avoids losing information about the historical existence of a feature branch and groups together all commits that together added the feature. Compare:

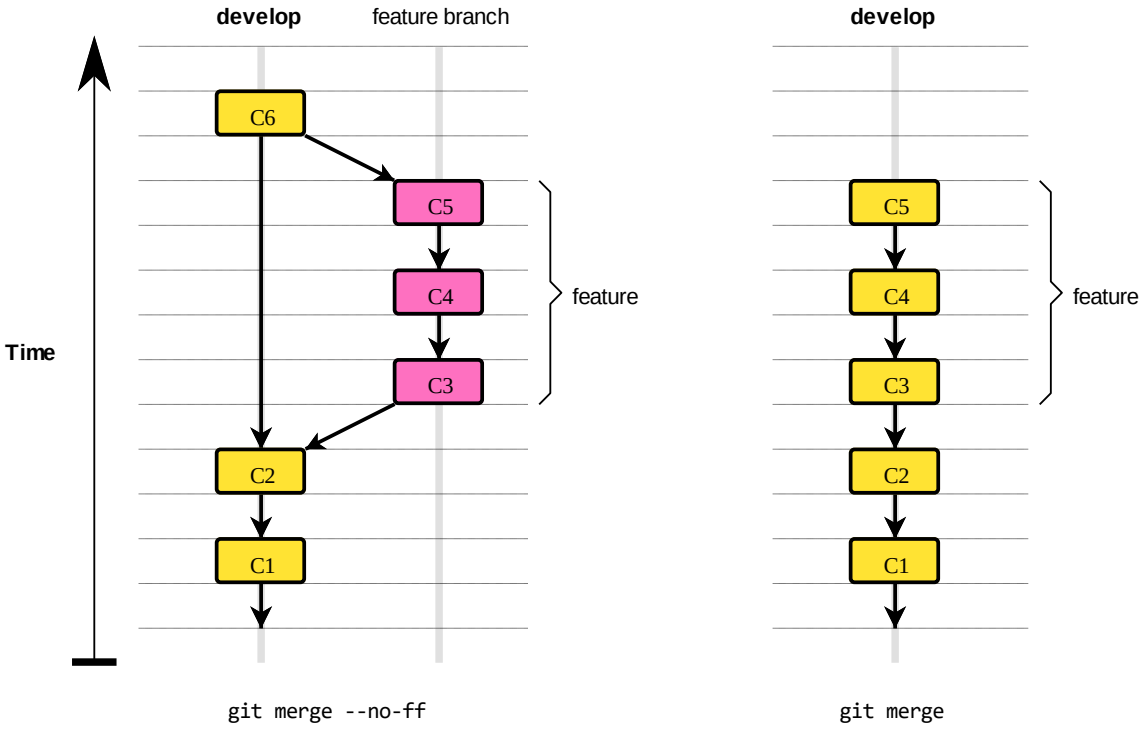


Figure G.4

In the latter case, it is impossible to see from the Git history which of the commit objects together have implemented a feature – you would have to manually read all the log messages. Reverting a whole feature (i.e. a group of commits), is time consuming in the latter situation, whereas it is easily done if the `--no-ff` flag was used. This workflow will create a few more (empty) commit objects, but the gain is much bigger than the cost.

G.2.2 Hotfix Branches

May branch off from:

`master`

Must merge back into:

`develop` and `master`

Branch naming convention:

`hotfix-{version ID}`

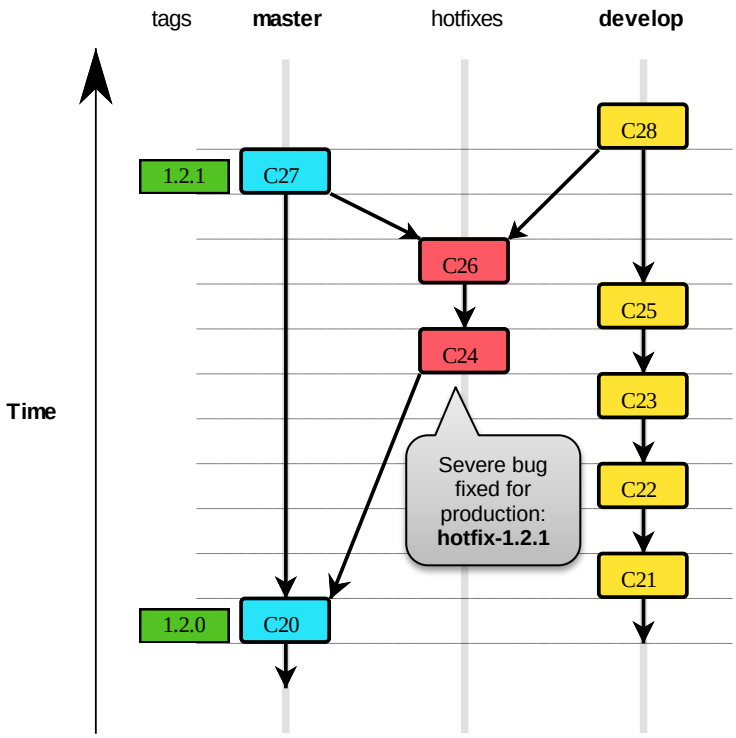


Figure G.5

Hotfix branches 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 idea behind a `hotfix` branch is that work on the `develop` branch (which may be many commits ahead of the current release), can continue while another person is preparing a quick production fix.

Creating a Hotfix Branch

`Hotfix` branches are created from the `master` branch, since changes on `develop` are potentially unstable. When branching off, the `hotfix` branch name needs to take on the next version ID, but with the *patch* "bumped" by 1, e.g. in fixing a bug in 1.2.0, the hotfix branch name is `hotfix-1.2.1`.

Ending 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 ensure that the bugfix is included in the next release as well.

Always use the "no fast forward" option (`--no-ff` flag) when merging back into the `master` branch to ensure the merge always creates a new commit object. When `hotfix` is merged into `master`, the `master` should be tagged with the new version ID.

There is no need to use the "no fast forward" option when merging back into the `develop` branch (`hotfix` branched from `master`, not `develop`, so there is no possibility of a fast-forward merge in this case).

G.3 Versioning

Utilize semantic versioning for your releases.

For Embedded Software, the software modules and programs we write will also attain the following attributes:

Hardware Interaction

Type	Abbrev.	Description	Example
None	N	No interaction with the hardware.	A data structure such as a FIFO.
Direct	D	Writing or reading directly to or from H/W, which will always respond.	Writing to a PORTx register.
Polling	P	H/W can only be accessed when ready, and the H/W status is determined by polling status flags.	Reading received data from the UART.
Interrupts	I	H/W will assert an interrupt when it is ready.	Received UART data is processed in an interrupt service routine.

Software Framework

Type	Abbrev.	Description	Example
Polling	P	S/W has “busy waiting” loops while polling H/W status flags.	Waiting for data to be received by the UART.
Interrupts	I	S/W is built around interrupt service routines (ISRs) and may require “critical sections”.	An ISR stores received UART data in a FIFO for later processing.
RTOS	R	S/W is multi-threaded and relies on RTOS functions and ISRs that may “suspend” or “block” the caller.	A FIFO module that “blocks” callers if it is full, empty, or being accessed by another thread.
Any	A	The S/W does not rely on a particular framework.	A module that performs data processing, such as finding the average or median.

Semantic Version Control

For Embedded Software, we will slightly modify the semantic version control rules:

Given a version number MAJOR.MINOR.PATCH, increment the:

1. **MAJOR** version when you:
   make incompatible API changes
   or
   change the software framework
2. **MINOR** version when you:
   add functionality in a backwards-compatible manner
3. **PATCH** version when you:
   make backwards-compatible bug fixes.

EXAMPLE G.1 UART Module

A module which implements communications via a UART has the following version numbering, where the abbreviation "H-S" stands for the Hardware Interaction-Software Framework:

H-S	Version ID	Summary
P-P	1.0.0	Uses H/W polling in a S/W architecture where interrupts are not used (or off).
P-P	1.1.0	Same as above, but adds additional functionality, but which can be "used in a 1.0.0 role" if necessary.
I-I	2.0.0	Uses ISRs to respond to H/W events in a S/W architecture that uses interrupts. There are critical sections.
I-R	3.0.0	Uses ISRs to respond to H/W events in a S/W architecture that uses an RTOS. There are operating system calls within the module implementation.

G.4 Summary

There is no definitive workflow model, but the one presented here is ideal for development in small teams. It forms an elegant mental model that is easy to comprehend and allows team members to develop a shared understanding of the branching and releasing processes.

Keep Figure G.1 handy and use it as a quick reference!

G.5 References

https://nvie.com/posts/a-successful-git-branching-model/
(Accessed 2018-07-14)

https://github.com/chrisjlee/git-style-guide (Accessed 2018-07-14)

https://git-scm.com/book/en/v2/Git-Branching-Branches-in-a-Nutshell
(Accessed 2018-07-14)

Exercise

1.

Recreate the workflow structure and commits shown in Figure G.1.

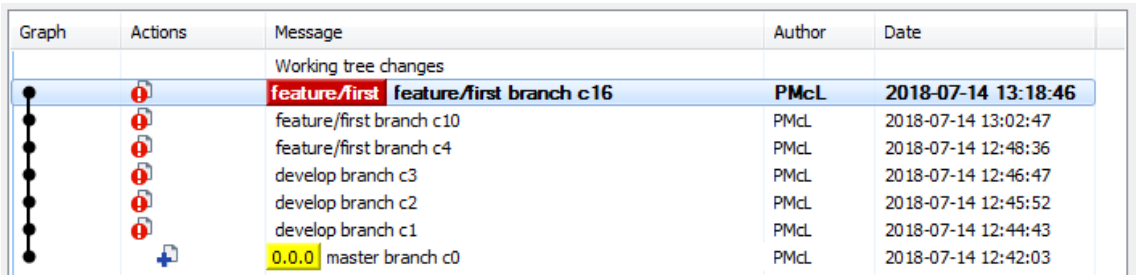
G.14

Answer

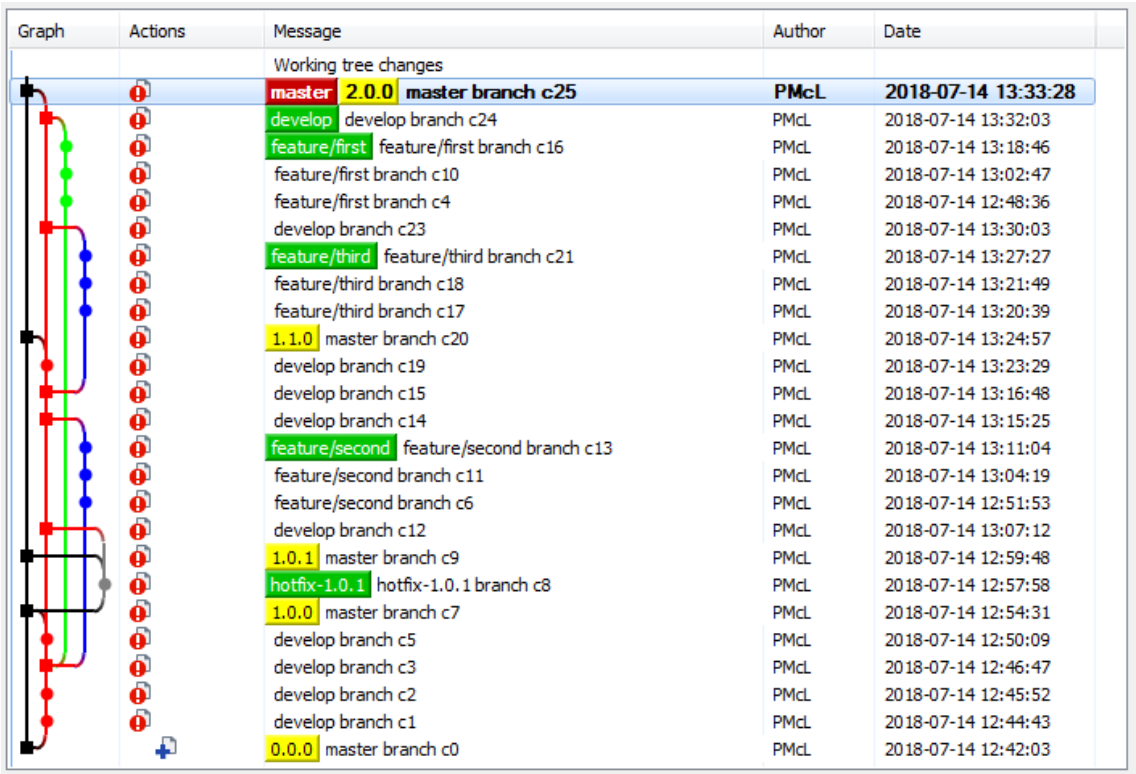
1.

Some “snapshots” of the revision log are shown below using TortoiseGit:

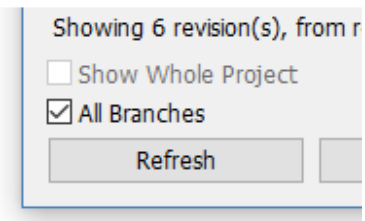
After commit 16:



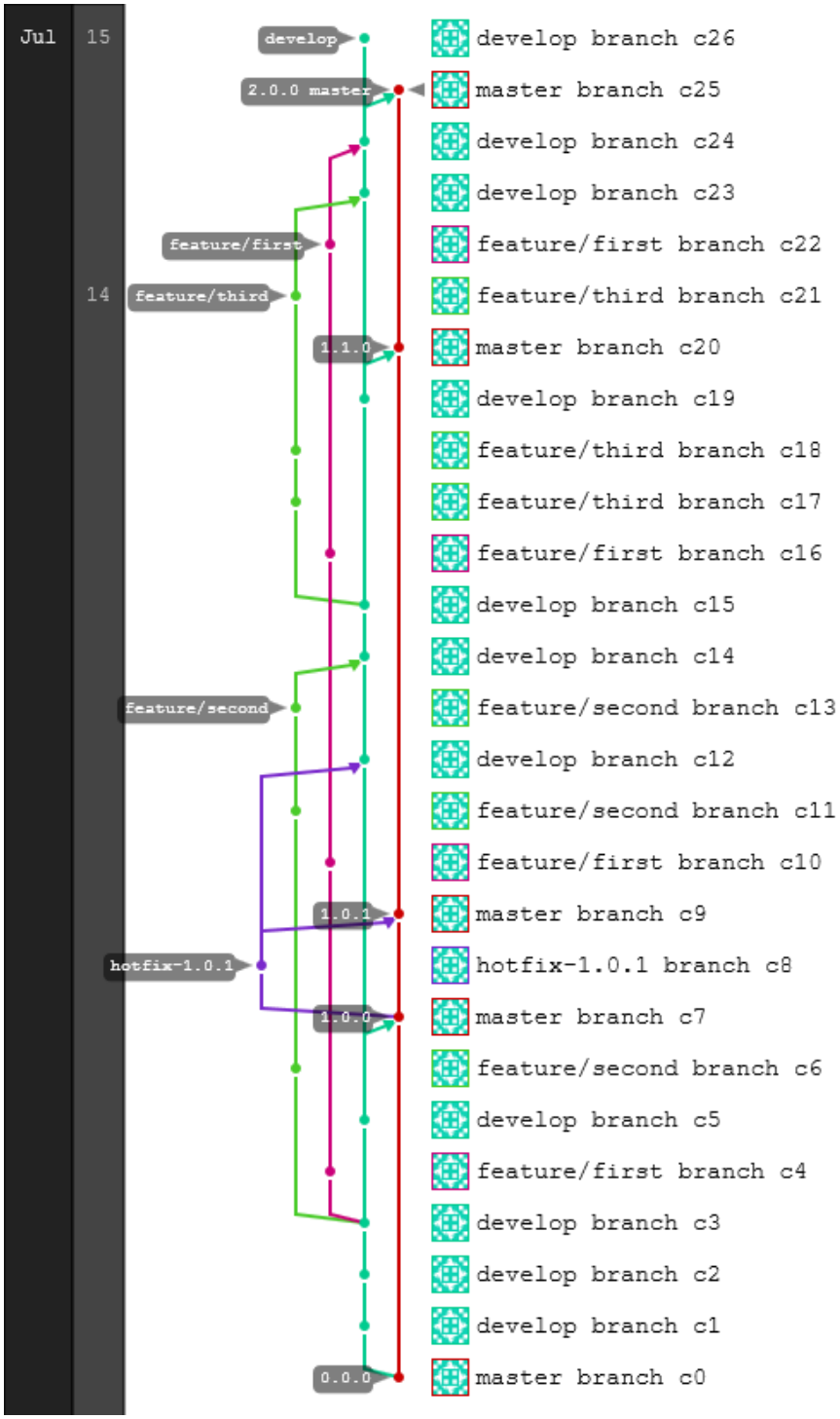
After commit 25 and the 2.0.0 tag:



Note that a revision log in TortoiseGit only shows the “parent” commits, unless you check the option for “All Branches” in the bottom left of the Log Messages dialog box:



A snapshot of the graph in GitLab is shown below:



Note that GitLab shows the entire history of commits, not just the parents (this graph was generated from the perspective of the `master` branch).

The solution can be cloned from the Git repository at:

http://git.pmcl.net.au/48434_Public/GitWorkflow.git