SE 234 Advance Software Development

#3 Git WorkFlow and Git History

CAMT
College of Arts, Media and Technology
Chiang Mai University

Lect Passakorn Phannachitta, D.Eng.

passakorn.p@cmu.ac.th

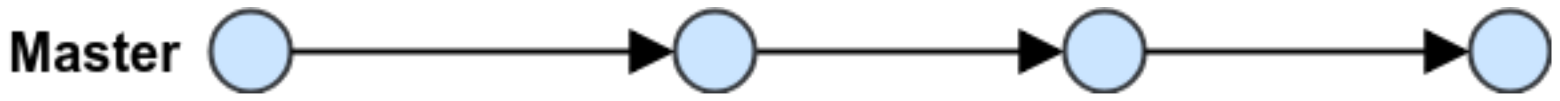
College of Arts, Media and Technology

Chiang Mai University, Chiangmai, Thailand

References

- Driessen, V. (2010). A successful Git branching model. Retrieved December, 2020, from http://nvie.com/posts/a-successful-git-branching-model/
- Jacobs, S. (2017). Telling stories with your Git history. Retrieved December, 2020, from https://about.futurelearn.com/blog/telling-stories-with-your-git-history
- Programster's Blog (2017). Git Workflows, 2020, from https:// blog.programster.org/git-workflows

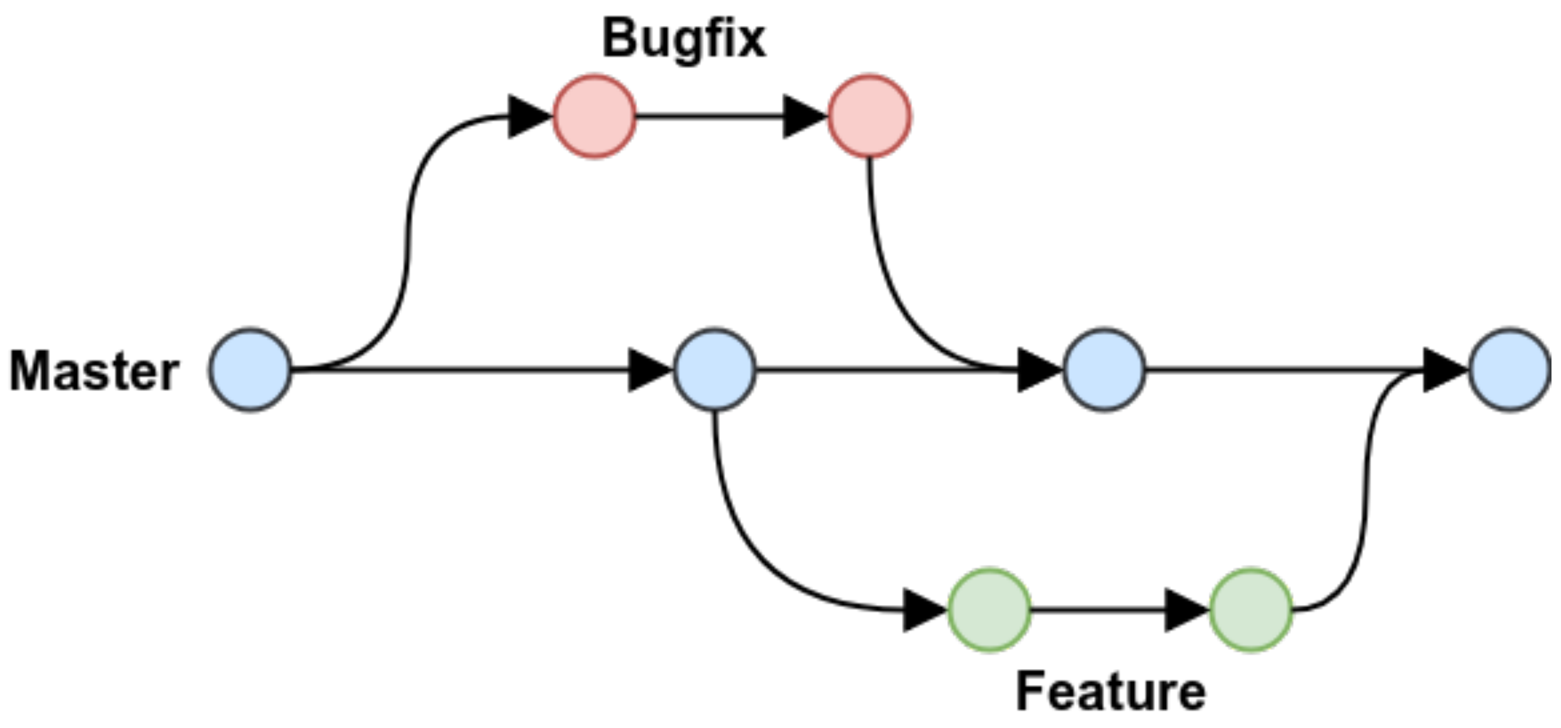
At the moment



What can be the problem?

How to fix it?

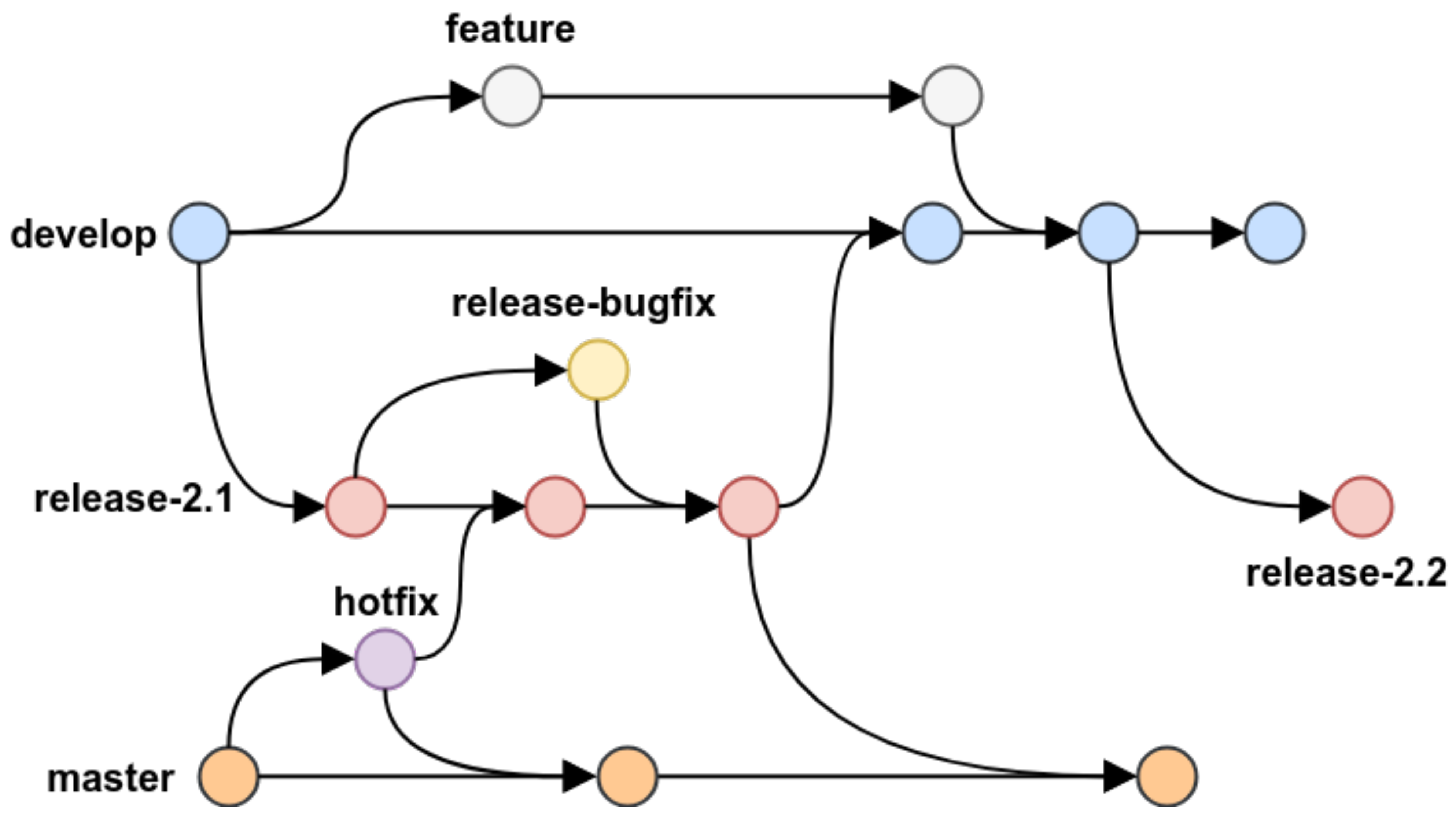
Add a few more branches



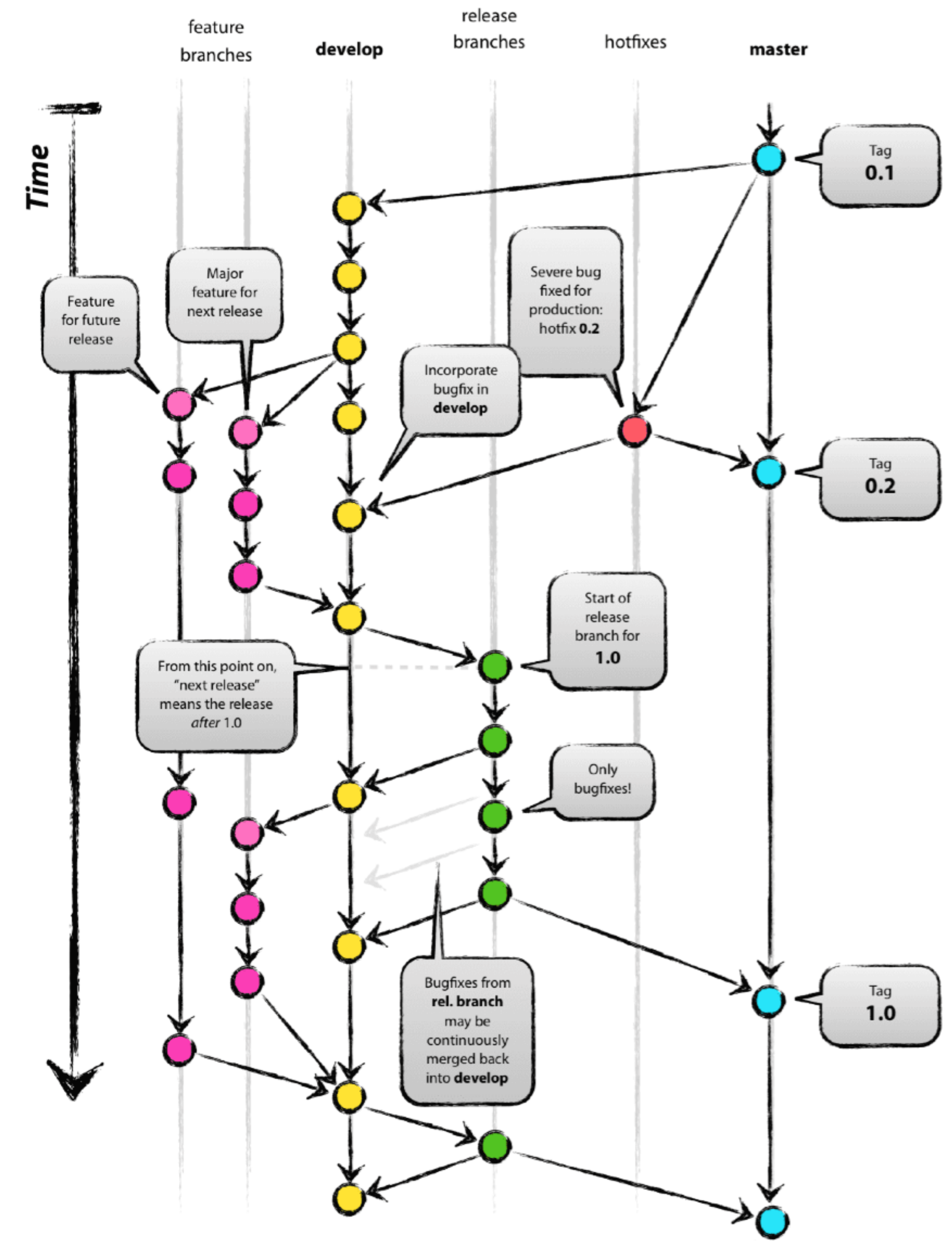
What can be the problem?

How to fix it?

The famous GitFlow



The famous GitFlow

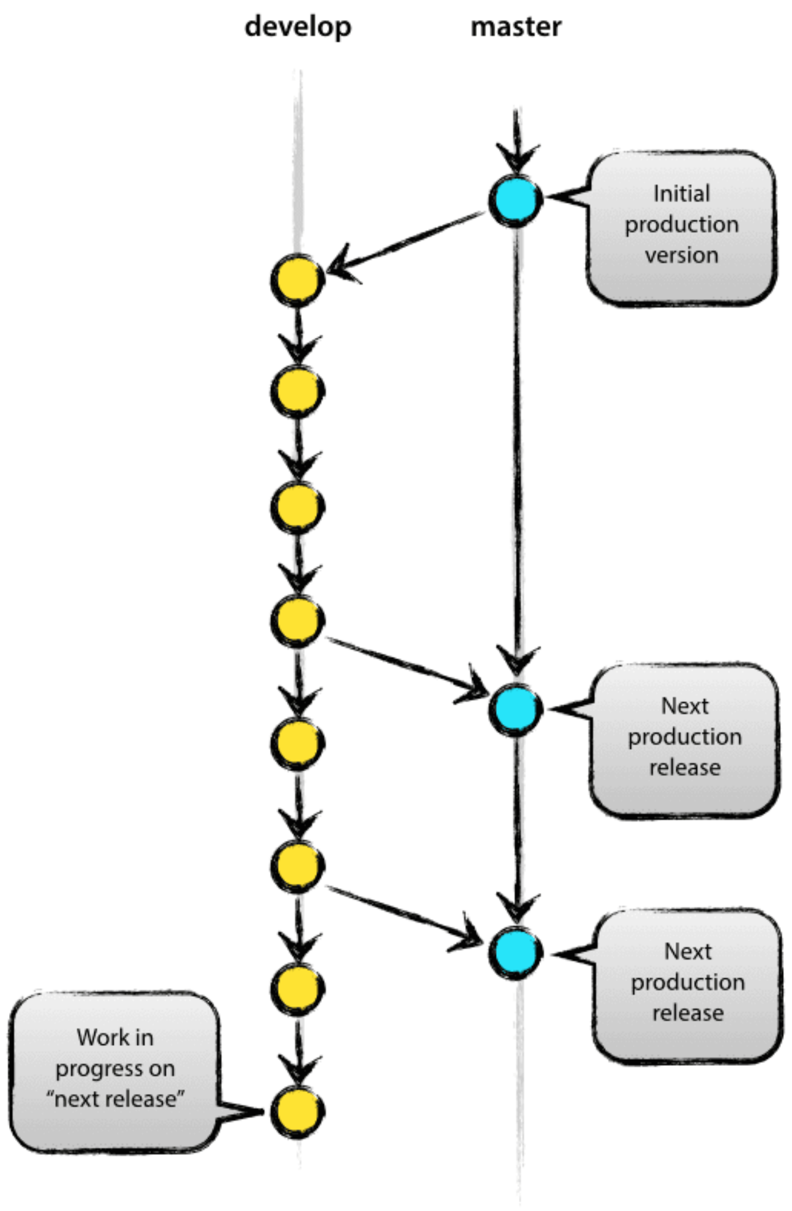


feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Severe bug fixed for production: hotfix 0.2
Major feature for next release
Feature for future release
Incorporate bugfix in develop
Tag 0.2
Start of release branch for 1.0
From this point on, "next release" means the release after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

A successful Git branching model

- Begin with the main branch**es,** i.e., origin/master and origin/develop branches
  - Master's head always point at a production-ready state.
  - Develop's head reflects a state with the latest delivered development changes for the next release.
- When the source 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**.

A successful Git branching model


develop
master
Initial production version
Next production release
Next production release
Work in progress on "next release"

A successful Git branching model

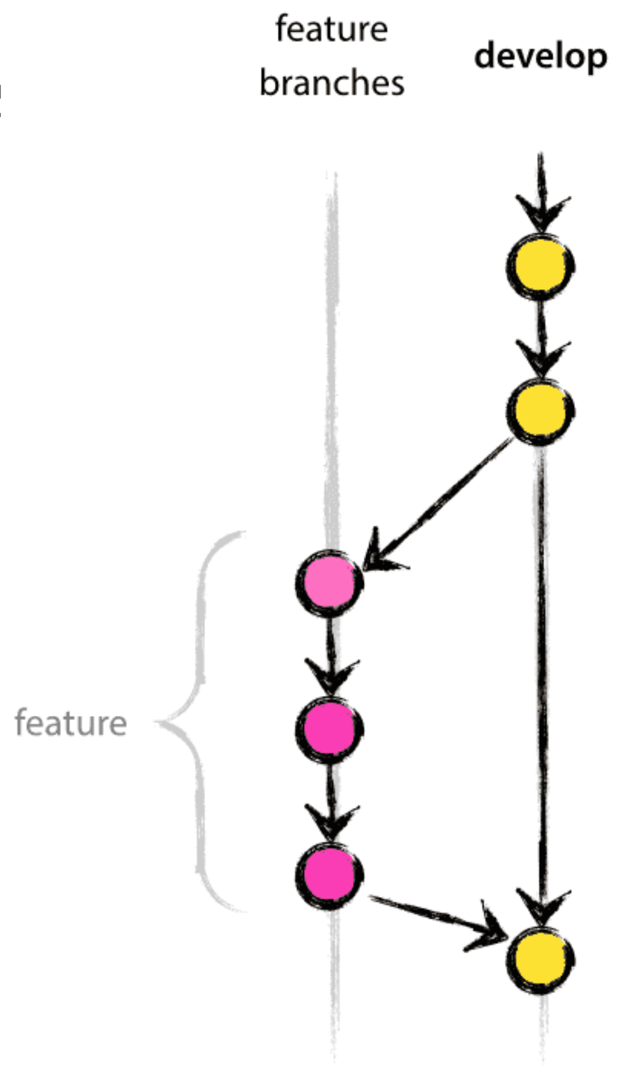
- Create a variety of supporting branches to
  - Aid parallel development between team members;
  - Ease tracking of features;
  - Prepare for production release;
  - Facilitate the possibility to quickly fix live production problems.
- These branches have a limited lifetime.

A successful Git branching model

- Types of supporting branches that are commonly used —
  - Feature branches
  - Release branches
  - Hotfix branches
- They have a specific purpose and are bound to different rules —
  - When and how they are created
  - When and how they are merged back

A successful Git branching model

- Feature branches
  - Generally branch off from develop and merge back to develop.
  - It typically exist in developer repos only, not in origin.
  - Recommend to merge **without** performing a fast-forward.
    - To retain all the histories in the git tree.



A successful Git branching model

- Release branches
  - Generally branch off from **develop** and merge back to **develop** or **master**.
  - Branch naming for convention: release-*
  - Release branches support preparation of a new production release.
    - The develop branch will be cleared to receive features for the next big release.

A successful Git branching model

- Release branches
  - A new **release** branch should be branched from **develop** when **develop** (almost) reflects the desired state of the new release.
  - A version number of the upcoming release branched should be assigned at the time the release branch is created.

A successful Git branching model

- Release branches — an example scenario
  - Let’s say a version 1.1.1 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
  - This shall be the time a new branch named release-1.2 is created.
  - If there is any bug fix, it should be applied on this branch.

A successful Git branching model

- When a release is ready
  - The **release** branch is merged into **master**, mainly because every commit on master is <u>a new release</u> by definition.
  - The commit on **master** must be <u>tagged</u> for future reference to this version.
  - Finally the changes related to <u>bug fix</u> that are made on the **release** branch need to be merged back into **develop**, so that future releases also contain these bug fixes.

A successful Git branching model

- Hotfix Branches
  - Generally branch off from **master** and merge back to **develop** and **master**.
  - Branch naming for convention: hotfix-*
  - Similar to **release** by means that **hotfix** is belong to the preparation for a new release, but **release** is made made to some plans.
    - Hotfix arises from the necessity to act immediately upon an undesired state of a live production version.

A successful Git branching model

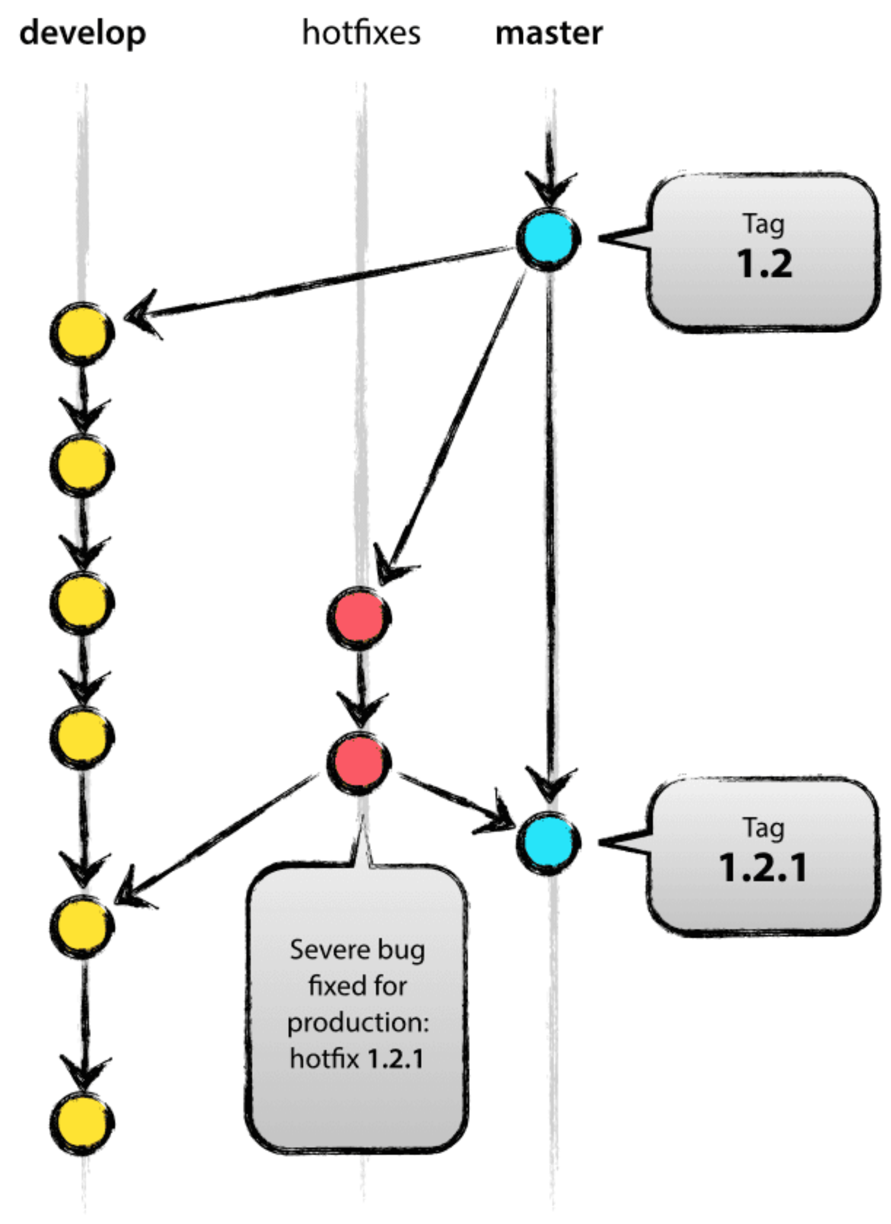
- Hotfix Branches
  - When a critical bug in a production version must be resolved immediately, a **hotfix** branch may be branched off from the **master** branch corresponding to the current production version.
  - Generally, a special team is assigned to quickly fix the bug.
  - The work on such team must not slow down or block the other teams that are working on **develop** and **master.**

A successful Git branching model

- Hotfix Branches — An example scenario
  - Let's say version 1.2 is the current production release running live and causing troubles due to a severe bug.
  - The changes on **develop** are yet unstable.
  - We may then branch off a **hotfix** branch and start fixing the problem.
  - At the time the fix is finished we have to merge back to both **master** and **develop** or **release.**
    - To safeguard that the fix is also included in the next release.

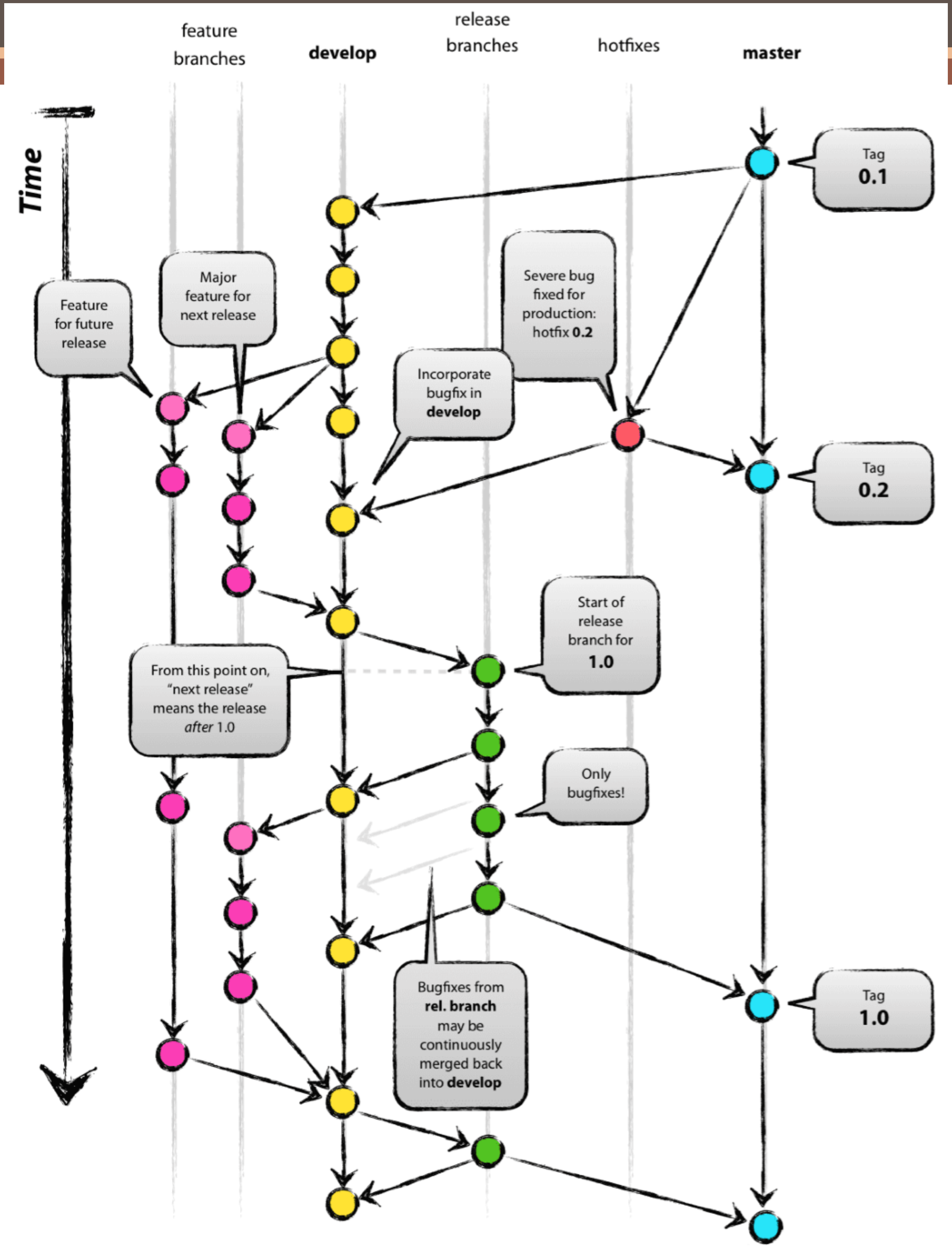
A successful Git branching model

- Hotfix Branches



A successful Git branching model

- Putting all together



feature branches
develop
release branches
hotfixes
master
Time
Tag 0.1
Severe bug fixed for production: hotfix **0.2**
Major feature for next release
Feature for future release
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

Telling stories with Git history

- Seb Jacobs suggests 5 principles to enable us to explicitly see the stories of our particular project.
  - Although our code should be self-documenting, it doesn't tell the story of why the code is the way it is or how it came to be.
  - Git history is a living, ever-changing, searchable record that tells the story of how and why our code is the way it is.

Telling stories with Git history

- Principle 1 — Atomic commits
  - It maybe a lot easier to make a commit like —

```
commit: [REDACTED]
Date:   [REDACTED]
   Allow educators to invite users onto courses.
61 files changed, 937 insertions(+), 81 deletions(-)
```

  - A better story of what has happened maybe told by —

```
a6455f8 Record when enrolment is created via an invitation.
b529f6d Allow invited users to enrol on courses.
b5bb6e4 Allow invited users to see the course description page.
c829cbc Send enrolment invitation emails in batches of 1000.
5feaccf Allow educators to invite users onto courses.
```

  - Atomic commits provide you a better sense of what is going on, thus increasing the value of each commit.

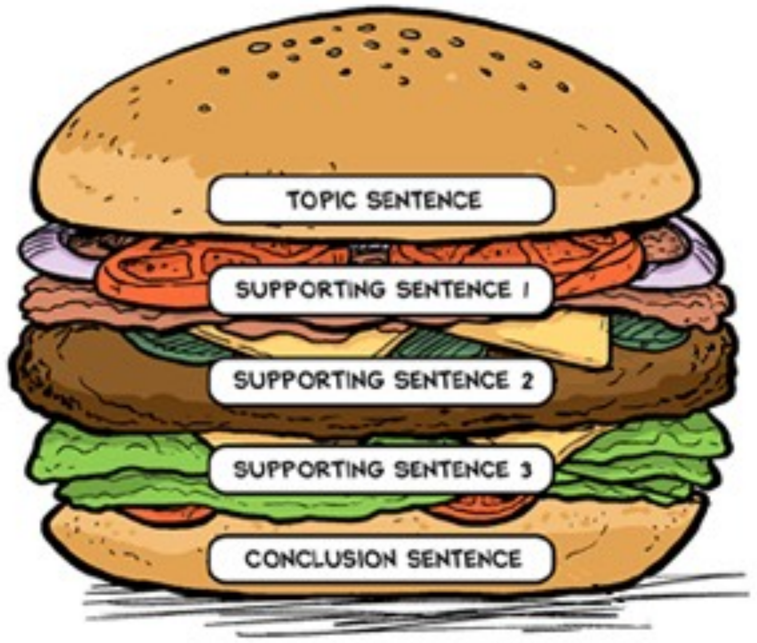
Telling stories with Git history

- Principle 2 — Write useful commit messages
  - Writing the message is to explain why you have made the change in the first place.
    - E.g., to satisfy a user requirement, to fix a bug, or to make another change easier to make in the future.
    - An example of a good template —

```
Short one line title.

An explanation of the problem, providing context (this may be as simple
as a reference to the user story).

Longer description of what the change does.

An explanation of why the change is being made.

Perhaps a discussion of alternatives that were considered.
```

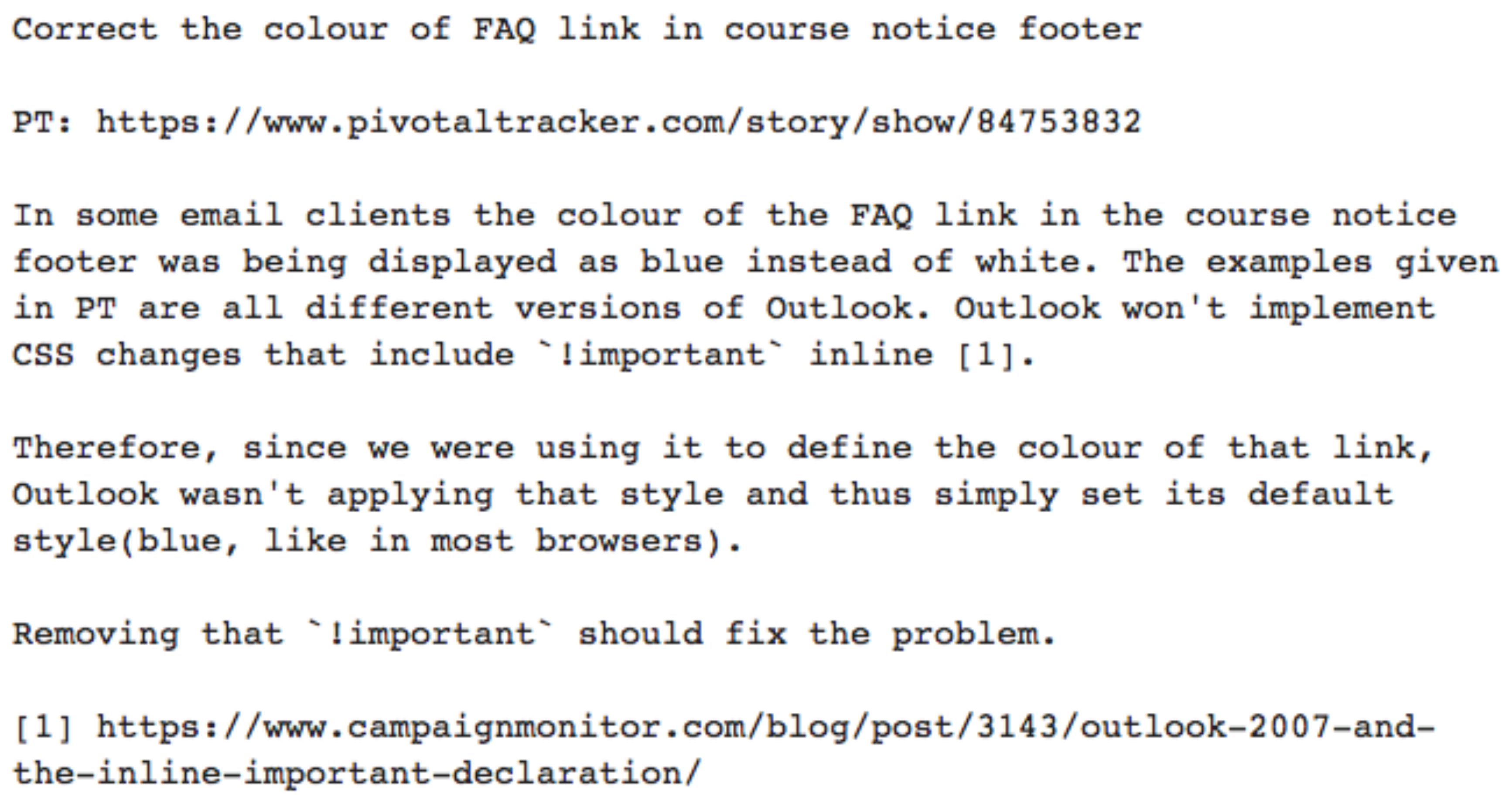
Telling stories with Git history

- Principle 2 — Write useful commit messages
  - Title should be clearly and concisely explained what the change is all about
  - In body should be written using the same principle of general writing, e.g. the hamburger model
    - The topic sentence should explain the value of the changes, rather than focussing on the implementation details
    - Explaining alternative solutions and providing external references are also recommended.



Telling stories with Git history

- Principle 2 — Write useful commit messages
  - Example — with a clear headline, it outlines the problem, the developer’s intent and also provides context around the change.



```
Correct the colour of FAQ link in course notice footer

PT: https://www.pivotaltracker.com/story/show/84753832

In some email clients the colour of the FAQ link in the course notice
footer was being displayed as blue instead of white. The examples given
in PT are all different versions of Outlook. Outlook won't implement
CSS changes that include `!important` inline [1].

Therefore, since we were using it to define the colour of that link,
Outlook wasn't applying that style and thus simply set its default
style(blue, like in most browsers).

Removing that `!important` should fix the problem.

[1] https://www.campaignmonitor.com/blog/post/3143/outlook-2007-and-
the-inline-important-declaration/
```

Telling stories with Git history

- Principle 3 — Revise history before sharing
  - We can modify the commit histories by use the **rebase** command in the **interactive** mode
  - The purpose is to re-order, reword, and refactor your commits until **they tell the clearest story possible**.
    - E.g. some commit messages do not useful information for someone else to read. These are such as, commit because of a typo where you found and fix it by yourself.
  - The trick is when you are going to send out a pull request, if some particular commits are not supposed to be in the message, simply remove it from the history.

Telling stories with Git history

- Principle 4 — Single purpose branches
  - Although you are working on one single feature/user story, you do not necessarily need to do everything in one branch.
  - It appears to be important to think about the **purpose** and **scope** of your feature branch.
  - By splitting up your feature branches, you not only reduce the pain of merging each branch, you also deliver subtask sooner, and make your Git history more readable.

Telling stories with Git history

- Principle 4 — Single purpose branches
  - From

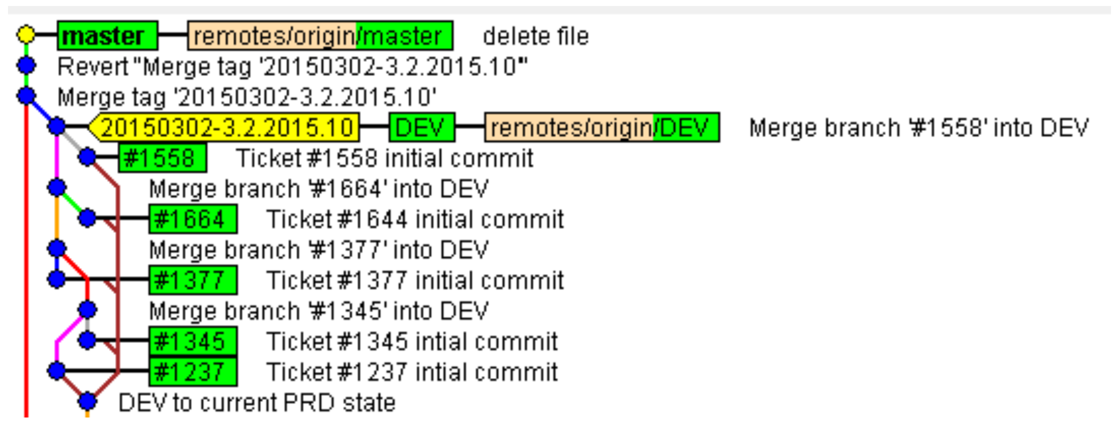
```
5ce95fb Notify educators when an invitation has been accepted.
5ce95fb Refactor specs around enrolment invitations.
ee95245 Extend enrolment invitation to educators.
cfb2fb4 Tidy up whitespace in enrolment invitations spec.
```

  - Changed to

```
* 0564508 Merge branch 'educator-enrolment-invitations'
|\
| * 5ce95fb Notify educators when an invitation has been accepted.
| * ee95245 Extend enrolment invitation to educators.
| |
|/
* 5ce95fb Refactor specs around enrolment invitations.
* cfb2fb4 Tidy up whitespace in enrolment invitations spec.
```

Telling stories with Git history

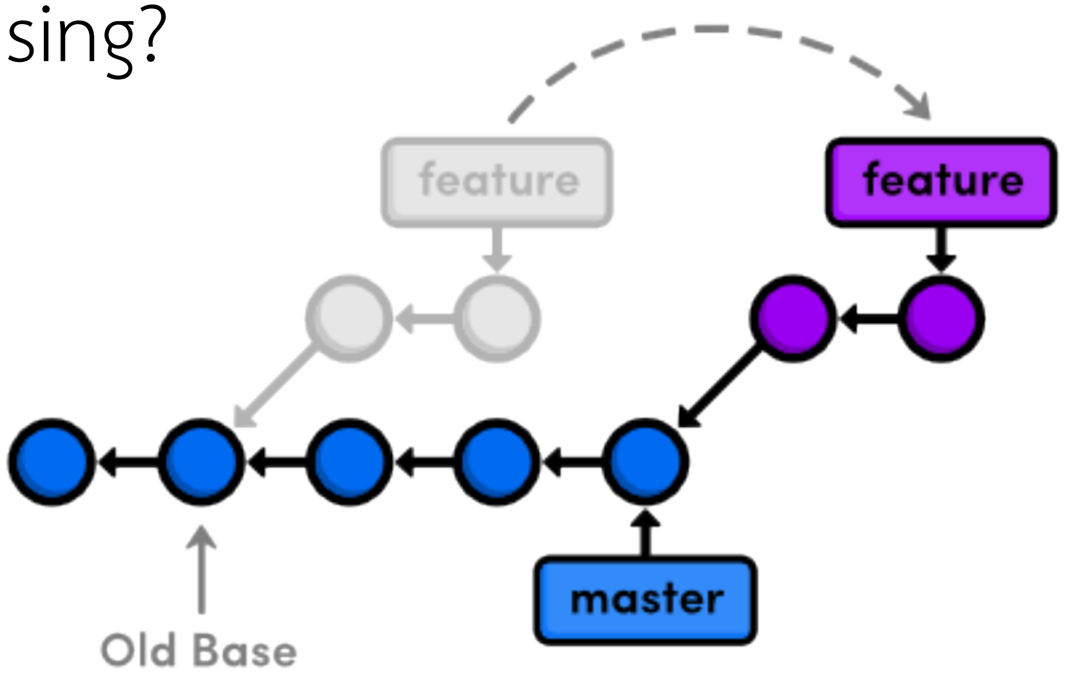
- Principle 5 — Keep your history linear
  - Often, merging changes into master can result in your history becoming tangled and difficult to read.
  - This becomes even more of an issue when you have several feature branches being developed in parallel.



```
master  remotes/origin/master  delete file
Revert "Merge tag '20150302-3.2.2015.10"
Merge tag '20150302-3.2.2015.10'
20150302-3.2.2015.10  DEV  remotes/origin/DEV  Merge branch '#1558' into DEV
#1558  Ticket #1558 initial commit
Merge branch '#1664' into DEV
#1664  Ticket #1644 initial commit
Merge branch '#1377' into DEV
#1377  Ticket #1377 initial commit
Merge branch '#1345' into DEV
#1345  Ticket #1345 intial commit
#1237  Ticket #1237 intial commit
DEV to current PRD state
```

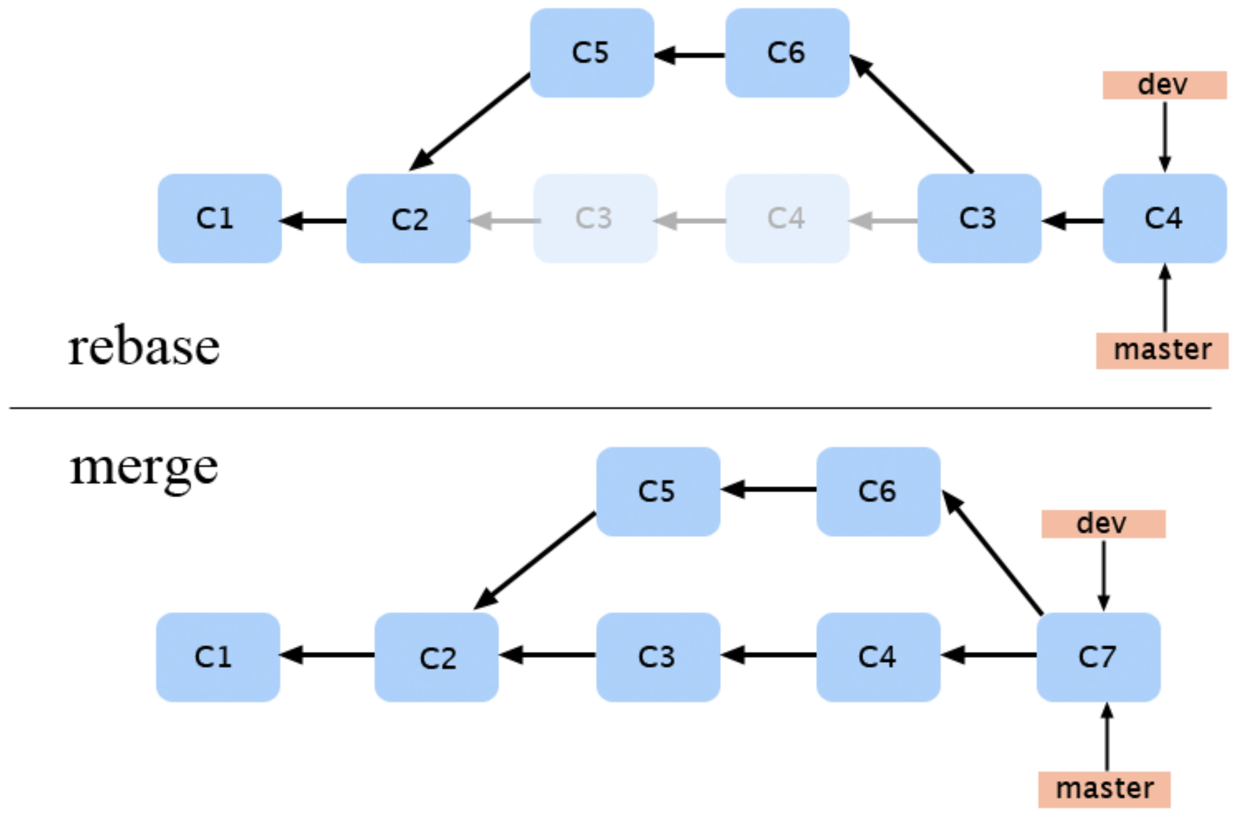
Telling stories with Git history

- Principle 5 — Keep your history linear
  - When it comes to merging, we may need try to to preserve our merges commits and **rebase** our feature branches before merging.
  - What is rebasing?



Telling stories with Git history

- Principle 5 — Keep your history linear
  - The difference between merge and rebase



Telling stories with Git history

- Principle 5 — Keep your history linear
  - The purpose of the rebase command is mainly for history refactoring —
    - E.g., a Git tree is the story book (it maybe) written by many authors about how your project was made.
      - The book will be useful when you start maintaining your software or want the reader to be able to reengineer your project.
      - Of course, you would not publish the very first draft of a book.

Telling stories with Git history

- Principle 5 — Keep your history linear
  - Rebase vs Merge — simple selection criteria.
    - Rebase local changes you have made but have not shared yet before you push them in order to clean up your story
    - But never rebase anything you have pushed somewhere — It is needed to be merged.

Telling stories with Git history

- Principle 5 — Keep your history linear
  - If everything is done correctly

```
*   ce91a05 Merge branch 'reprint-statements'
|\
| * ae43ad0 Disable reprint link for refunded purchases.
| * 0b1abb0 Allow admins to flag purchases for re-printing.
* | 35d0357 Put dates formats in the pattern library
* |   275206c Merge branch 'fulfilment-attempt'
|\ \
| * | 7aae45b Populate `fulfilled?` for existing purchases.
| * | 8e461b1 Display purchase fulfilment attempts to admins.
* | | 1adc0a9 Reduce padding around the course run date
| |/
|/|
```

  - Will be changed to

```
*   ce91a05 Merge branch 'reprint-statements'
|\
| * ae43ad0 Disable reprint link for refunded purchases.
| * 0b1abb0 Allow admins to flag purchases for re-printing.
|/
*   35d0357 Put dates formats in the pattern library
*   275206c Merge branch 'fulfilment-attempt'
|\
| * 7aae45b Populate `fulfilled?` for existing purchases.
| * 8e461b1 Display purchase fulfilment attempts to admins.
| * 44cbfd0 Introduce Fulfilment attempts
|/
*   1adc0a9 Reduce padding around the course run date
```

What's next

- MSR (Mining software repository)
- E.g.,
  - https://github.com/ishepard/pydriller
  - https://medium.com/thg-tech-blog/analysing-source-control-history-with-rust-ba766cf1f648

Assignment 1 — 8 points of total grades

- Recall the final assignment we had in the SE233 class where Pikachu Volleyball game was made
- Your task is to redo plus combine yours with that of several group members and use git to control the entire process.

1. Group of **four** or **five** members (not subject to further negotiation)
2. A student have to take a full implementation for their "**Two additional features**" For a particular member who did not submit such features or their original submitted or having the same features, select each two per from following list:
    1. Any kind of special attack
    2. Speed up player 3 seconds
    3. Slow down enemy 3 seconds
    4. Speed up ball 3 seconds
    5. Slow down ball 3 seconds
    6. Instant counter
    7. Instant respawn enemy
    8. Add ball 5 seconds

Criteria — Due Jan 8, 2021

1. Commit frequently, e.g., when any subtask is completed.
   On the other hand, push only when an entire feature is completed. At the end, there must be at least one commits per feature. **(1 points)**
2. The code for all the (pushed) version must be executable, i.e., can run through Maven. Also versioning label must be making sense **(1 point)**
3. The branch structures must be complied with the model discussed in this lecture
   1. Have dev **(0.5 point)**
   2. Have bugfix **(0.5 point)**
   3. Merge through merge request **(0.5 point)**
   4. Merge with Master only required **(0.5 point)**
4. Write the commit messages that can tell the story of the entire project development. **(1 point)**
5. Not a one or two days project **(1 points)**
6. GitLab said that tasks are evenly distributed as **(2 points)**

Question Times