Git Introduction

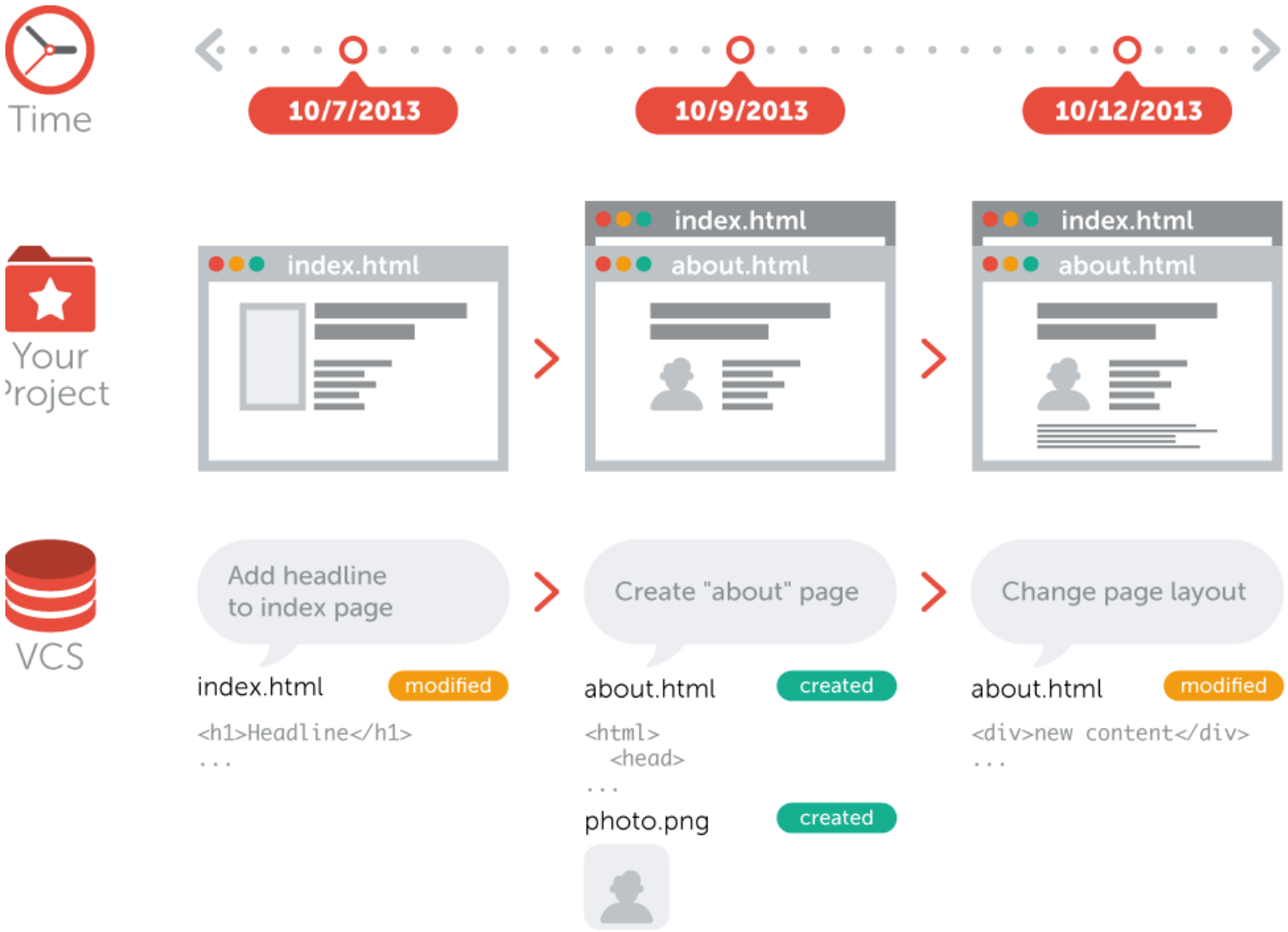
Author: Yaozhi Zhao

2018-11-28

Agenda

- What is Version Control?
- Why use Version Control?
- Muscle Control System
- Version Control System
- Git Installation
- Git terminology
- Basic operations
- Basic Workflow with Git

What is Version Control?



Why use Version Control?

- Collaboration
- Storing Versions
- Restoring Previous Version
- Understanding What happened
- Backup

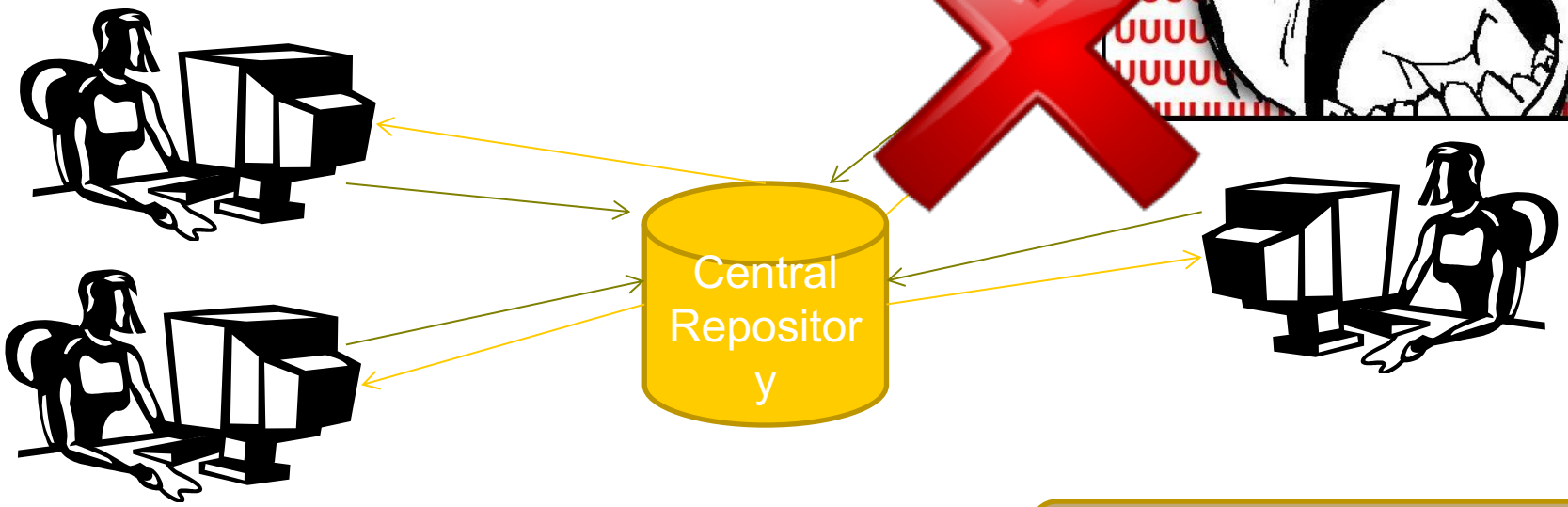
MCS

- Muscle Control System
- IM/Mail
- Folder
- Archive
- Cloud Disk

VCS Classify

Centralized	• cvs (Concurrent Version System) • Svn (Subversion) • Visual SourceSafe • Perforce • etc
Distributed	• Bitkeeper • Monotone • Mercurial(hg) • Git • etc

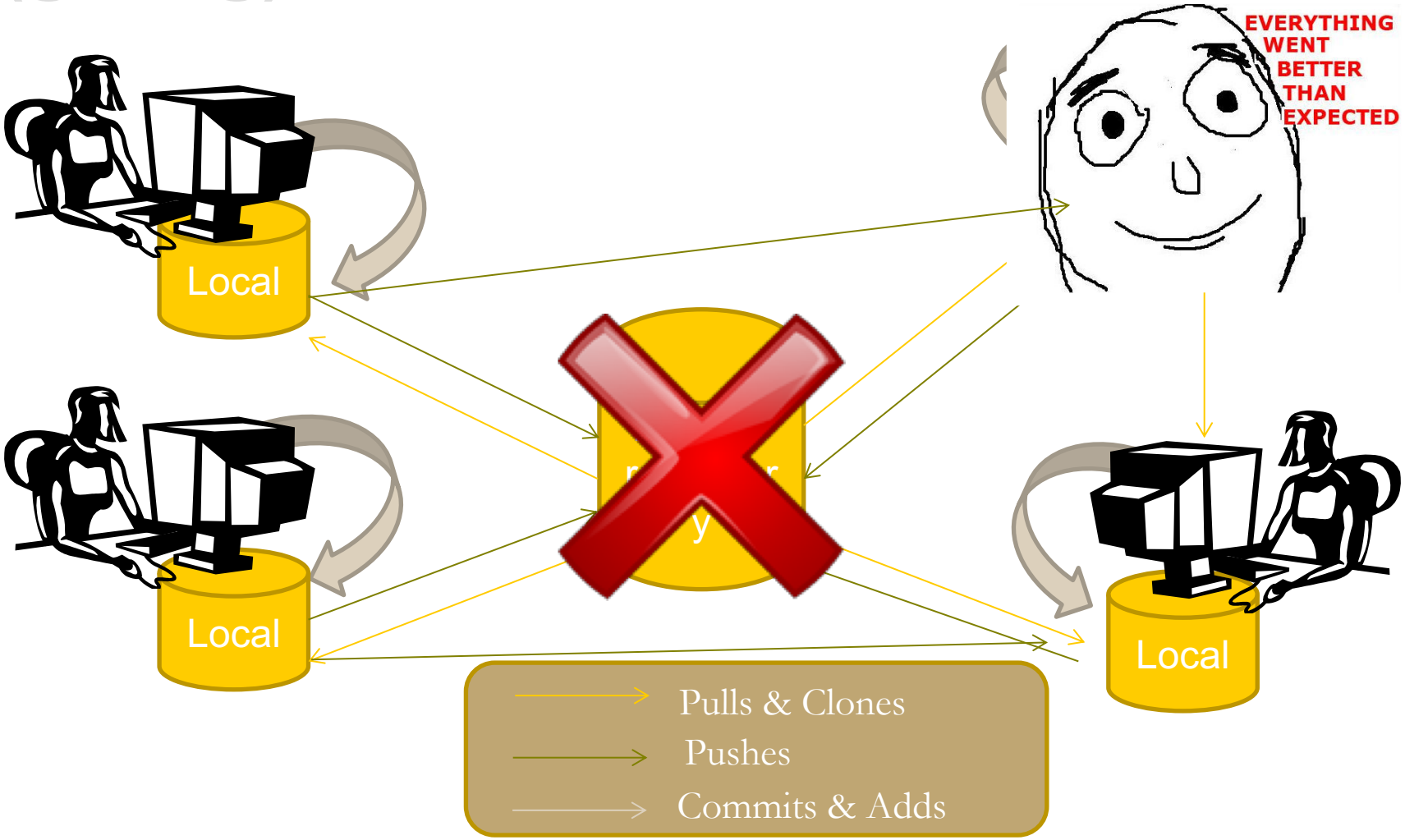
Traditional centralized VC model (CVS, Subversion)



Checkouts & Updates

Commits & Adds

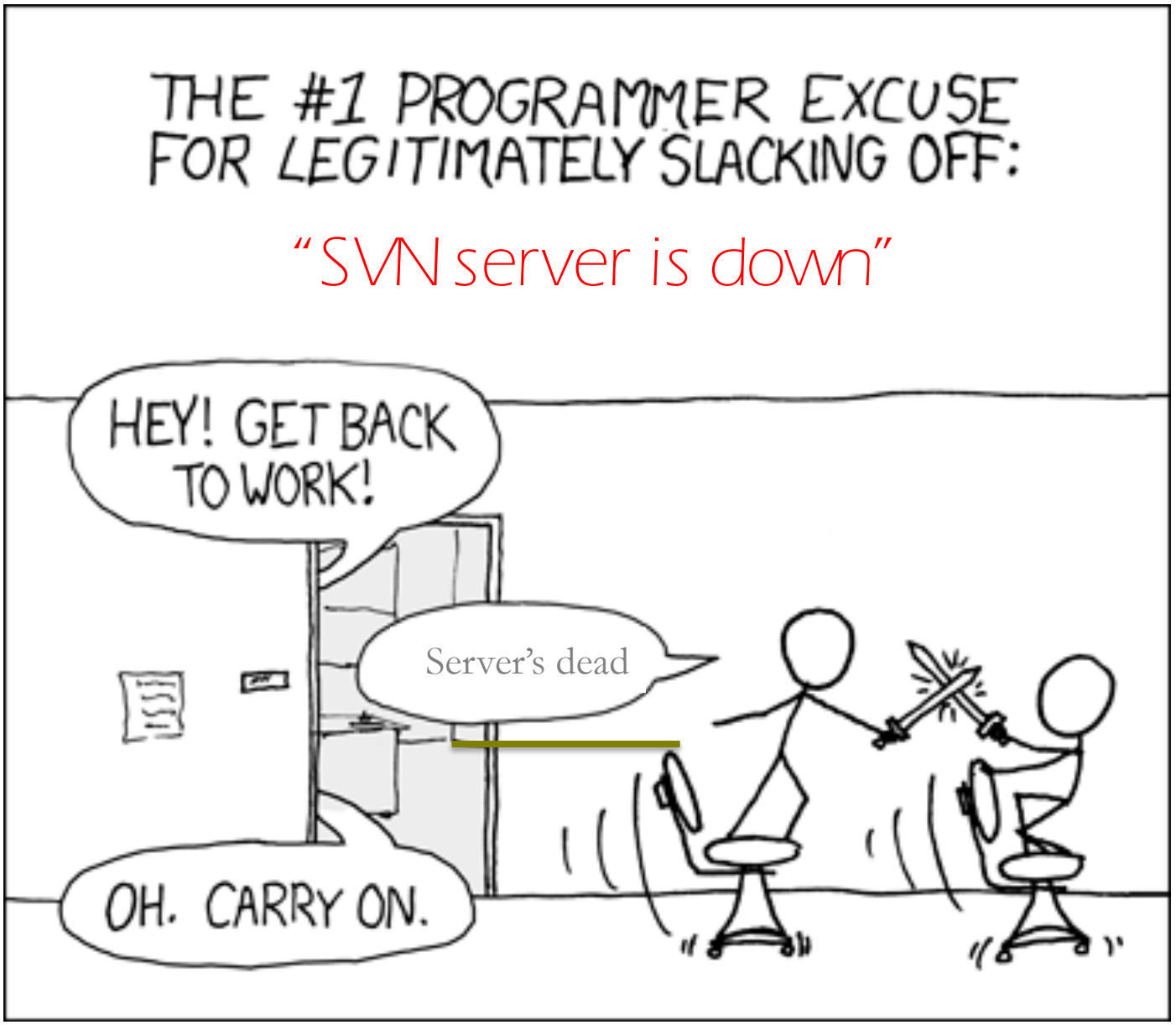
Distributed version control system (git, hg)



SVN vs Git

Distributed
Lightweight
SHA-1 hash key
Anyone
Extremely Fast

Centralized
Heavy
Global Revision Number
Authorized
According to network

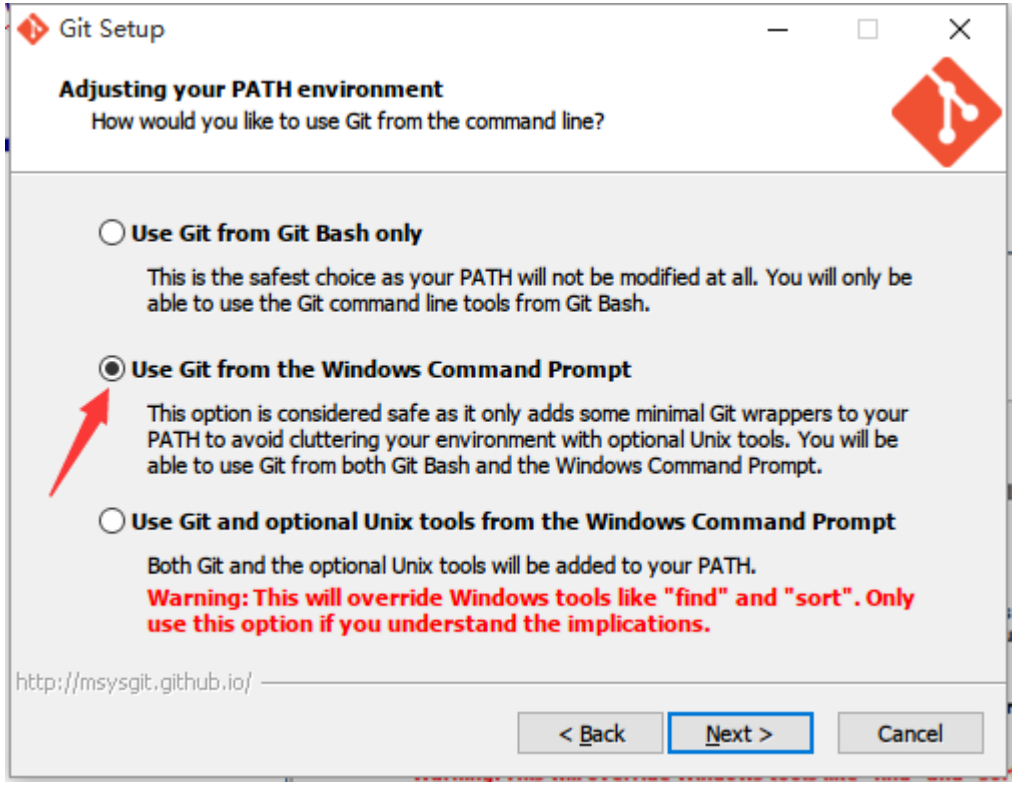

THE #1 PROGRAMMER EXCUSE FOR LEGITIMATELY SLACKING OFF:
"SVN server is down"
HEY! GET BACK TO WORK!
Server's dead
OH. CARRY ON.

Git Installation

- **Windows:**
  - **https://msysgit.github.io/**
- **Linux**
  - **User Package Manager**
    - RedHat Series: yum install git
    - Debian Series: apt-get install git
    - Archlinux Series: pacman -S git
- **Mac OSX**
  - **An old version of git was delivered with OS.**
  - **Use homebrew install newer version:**
    - brew install git

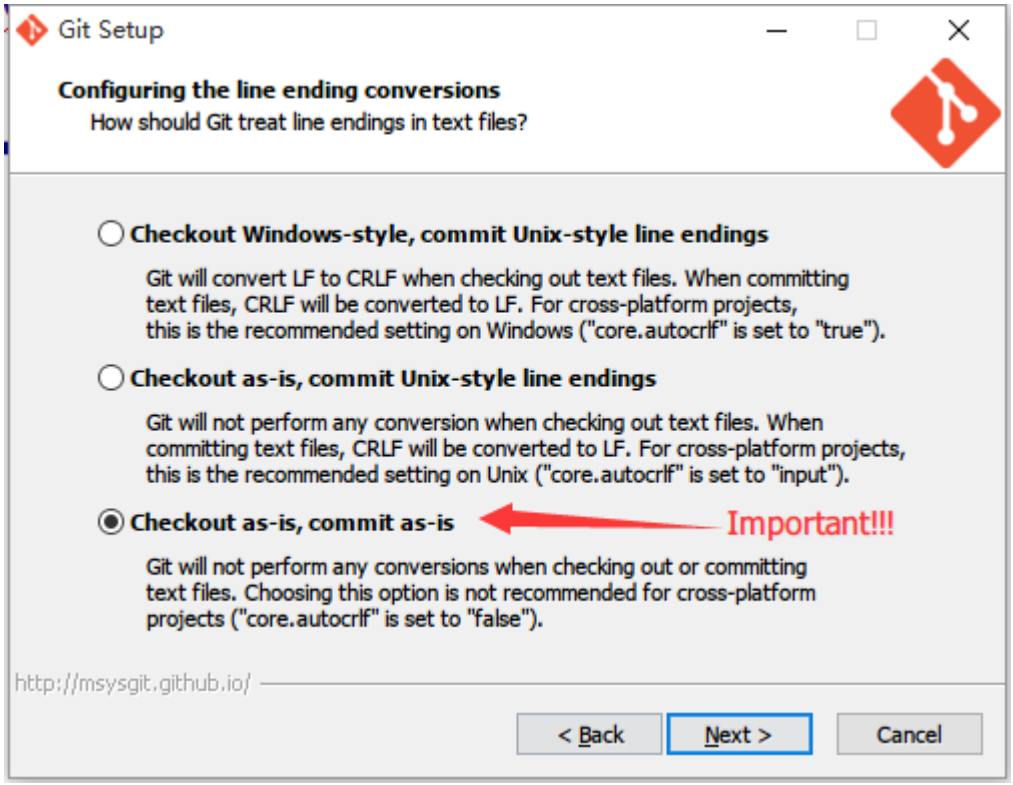
Git Installation

- **Windows:**



Git Installation

- **Windows:**



Git GUI Client

gitk, git-gui (delivered with git)

TortoiseGit (Windows only)

Git Extensions (Windows only)

SourceTree (Windows/Linux/Mac)

GitKraken (Windows/Linux/Mac)

GitHub Client (windows/Mac)

Git terminology (1/4)

- **Git Directory**
  - The .git directory that holds all information in the repository
  - Hidden from view
- **Working directory**
  - The directory in which .git resides
  - Contains a "snapshot" of the repository
  - Will be changed constantly eg. when reverting or branching
  - Your changes will be made to this directory
- **Stage**
  - Changes have to be added to the stage from the working directory in order to be saved into a commit
  - Could be thought of as a “loading bay” for commits

Git terminology (2/4)

- **Branch**
  An alternate line of development
- **Working copy**
  The branch you are in now that you make your changes in
- **Master**
  The default branch of the repository

Git terminology (3/4)

- **Commit**
  - A set of changes that have been saved to the repository
  - Can be reverted and even modified to some extent
  - Identified by a hash of the changes it contains
- **Tag**
  - A certain commit that has been marked as special for some reason
  - For example used to mark release-ready versions

Git terminology (4/4)

- **HEAD**

  The latest revision of the current branch

  Can be used to reference older revisions with operants

  HEAD^^2 == 2 revisions before latest

  HEAD^ == 1 revision before latest

  HEAD~3 == 3 latest revisions

  HEAD^^2..HEAD == 2 revisions before the latest to the latest

- **Origin**

  Default name for a remote repository when cloning an existing repository

Basic operations

- **Initializing a repository**
- **Basic workflow**

  Commands:

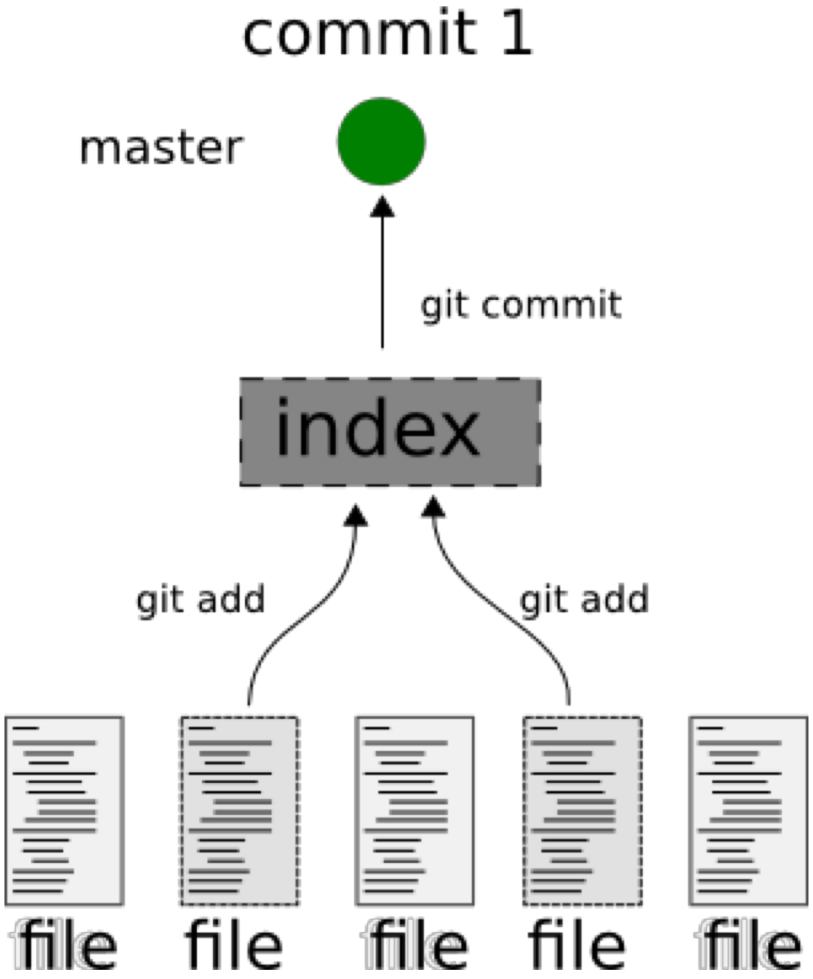
  - add
  - commit
  - revert
  - reset
  - remote
  - push & pull & fetch

Initializing a repository

- **Initializing an empty repository**
  Go to wanted directory
  Write $ git init
- **Cloning from existing repository**
  Go to wanted directory
  $ git clone URL

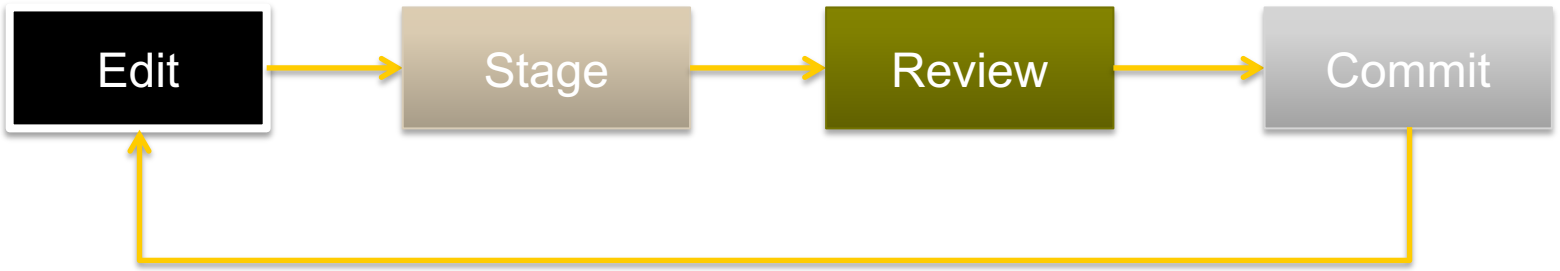
- **Remember to set your user name and options**
  git config –global user.name ”my name”
  git config –global user.email my@email.com

Visualization of basic workflow



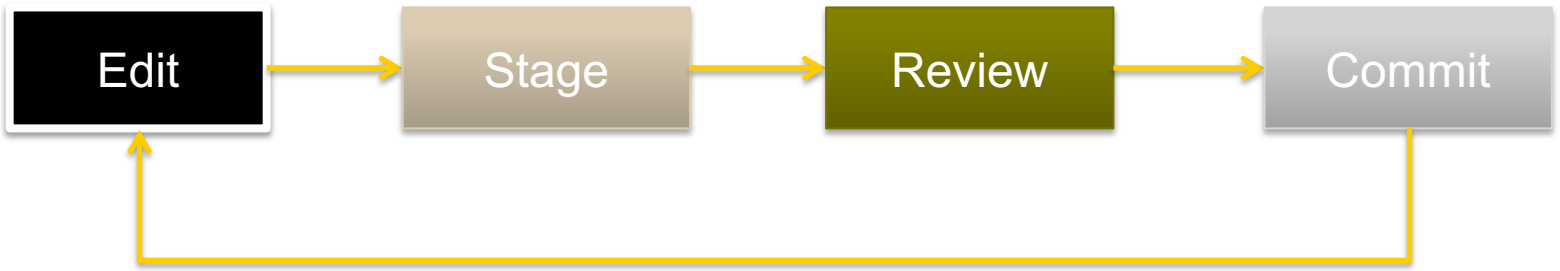
Basic Workflow with Git

- **Edit Files**
- Add them to be versioned (*staging commits*)
- Review changes
- Commit changes to the repository



Basic Workflow with Git

- Edit Files
- Add them to be versioned (*staging commits*)
- Review changes
- Commit changes to the repository



Stating commits (git add)

- **Adds your files to the index so that they are ready to be committed**
- **Only added files can be committed**
- **You can add individual files or all**

  $ git add .

  $ git add filename

  $ git add part/filename

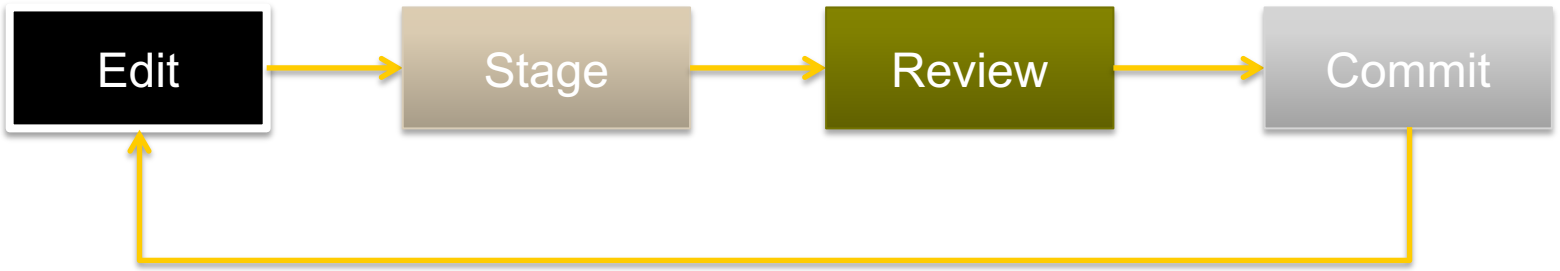
Assigment 1: Let’s create a file

- **Create a file that has your first name in it. Call it 'name' or whatever you want**

- **Make Git track your file (stage the change)**

Basic Workflow with Git

- Edit Files
- Add them to be versioned (*staging commits*)
- **Review changes**
- Commit changes to the repository



Git status

- **Shows the branch you are on**
- **Shows staged commits (files added to the index)**
- **Files can still be removed from the index**
- **Usage:**

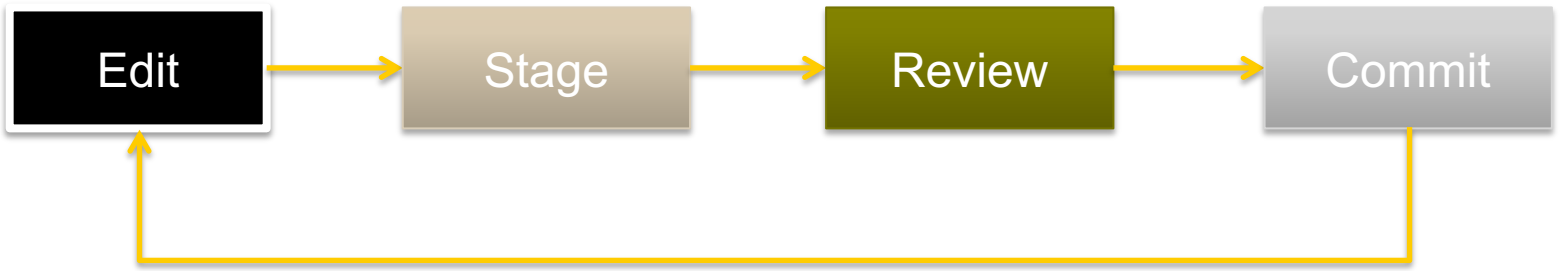
  $ git status
- **If there are any files that need removing from the index**

  $ git reset HEAD <filename>

  Unstages file from current HEAD

Basic Workflow with Git

- Edit Files
- Add them to be versioned (*staging commits*)
- Review changes
- **Commit changes to the repository**



Commit

- **$ git commit**

  Opens up a text editor for you to change the commit message

  Good to review all changes in this stage too

- **When you have added all the files you want, you can commit them to the repository using**

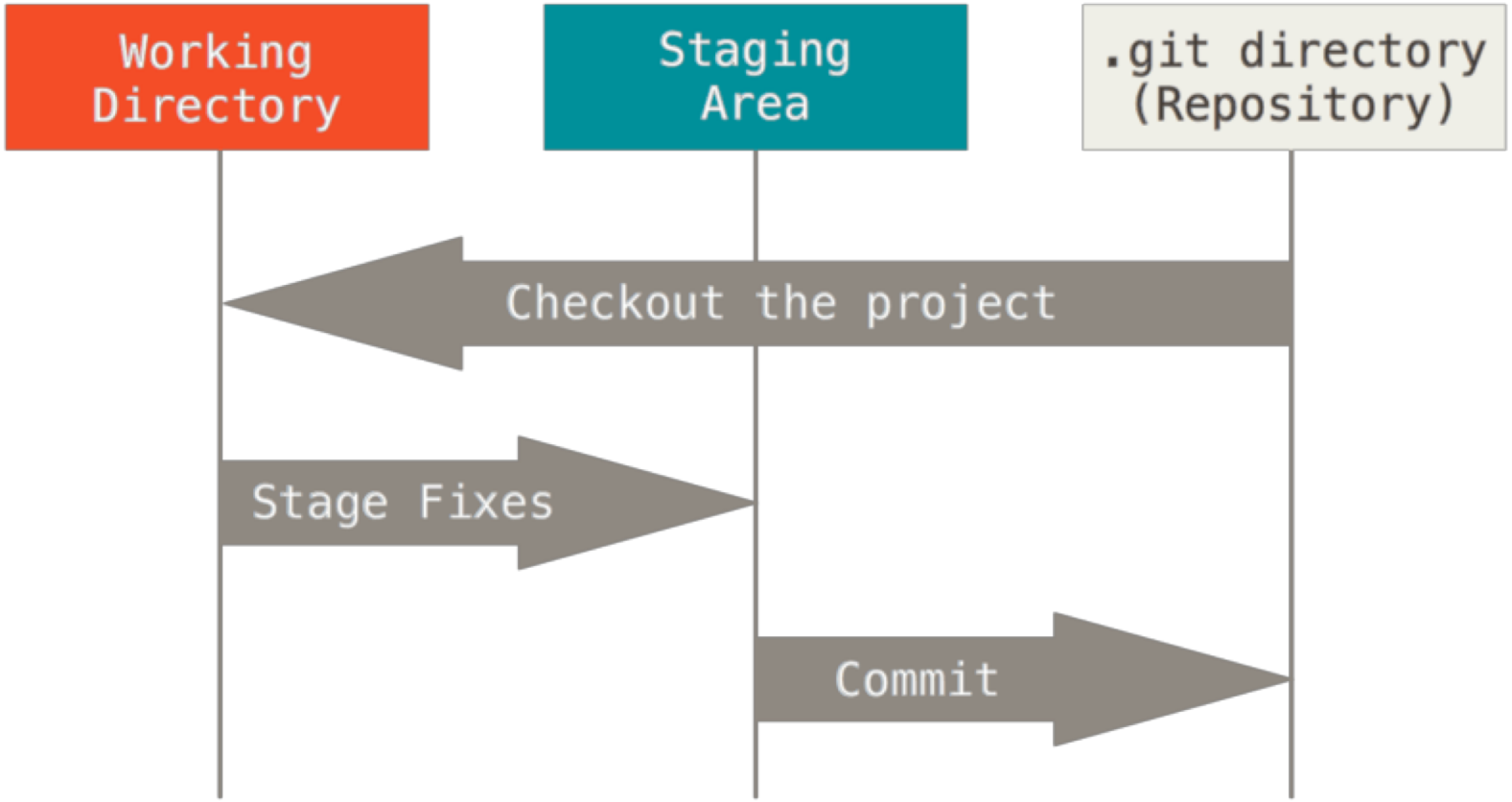
- **$ git commit -m "commit message“**

  Inserts a commit message straight away, does not open an editor

Assignment 1: Commit

- **Adds your changes to your local repository**
  Only changes added to the index will be notified
  => Only added files will be versioned
- **Opens a dialog where you can insert a commit message**
- **Commit messages consists of two parts:**
  One short line describing the commit in general
  Longer more detailed description about the commit
- **Short and long description are separated by a newline**
- **Only the short description is required**
- **Empty commit message will cancel the commit**
- **Lines beginning with # are regarded as commits**
- **You can view the commits in a repository by using:**
  $ git log

Git Stage Conception



Comparing differences between files

- **Git makes heavy use of *NIX software called diff**
- **Usage**

  $ diff <file1> <file2>

- **Sometimes you need to know about what changes have occured in a file between two revisions**

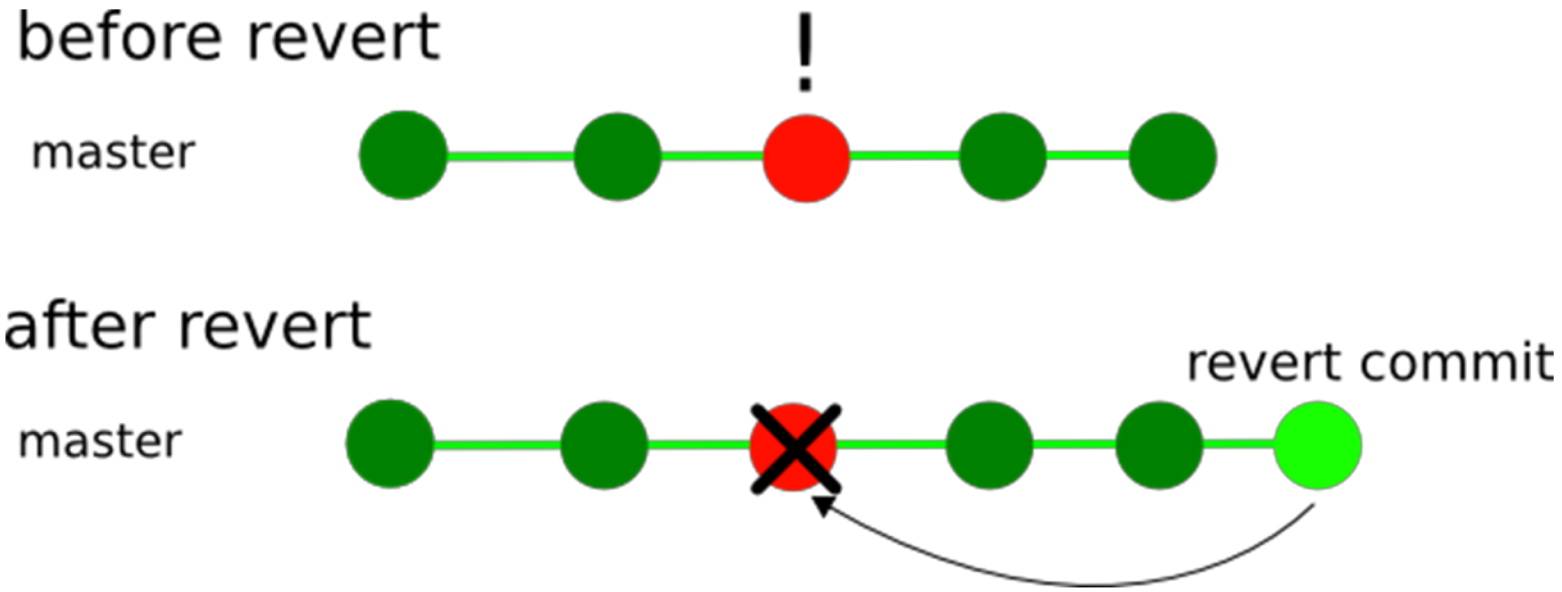
  $ git diff <start commit>..<end commit> -- file

  E.g. $ git diff HEAD^^..HEAD^ -- index.php

Revert

- **You can undo commits by using revert**
- **Revert undoes a single commit and leaves other commits intact**
- **This means that you can revert a change that broke something, but reverting it doesn't affect commits made after the "faulty commit"**
- **Adds a commit that undoes the given commit**
- **usage example:**
  $ git revert <commit-id>

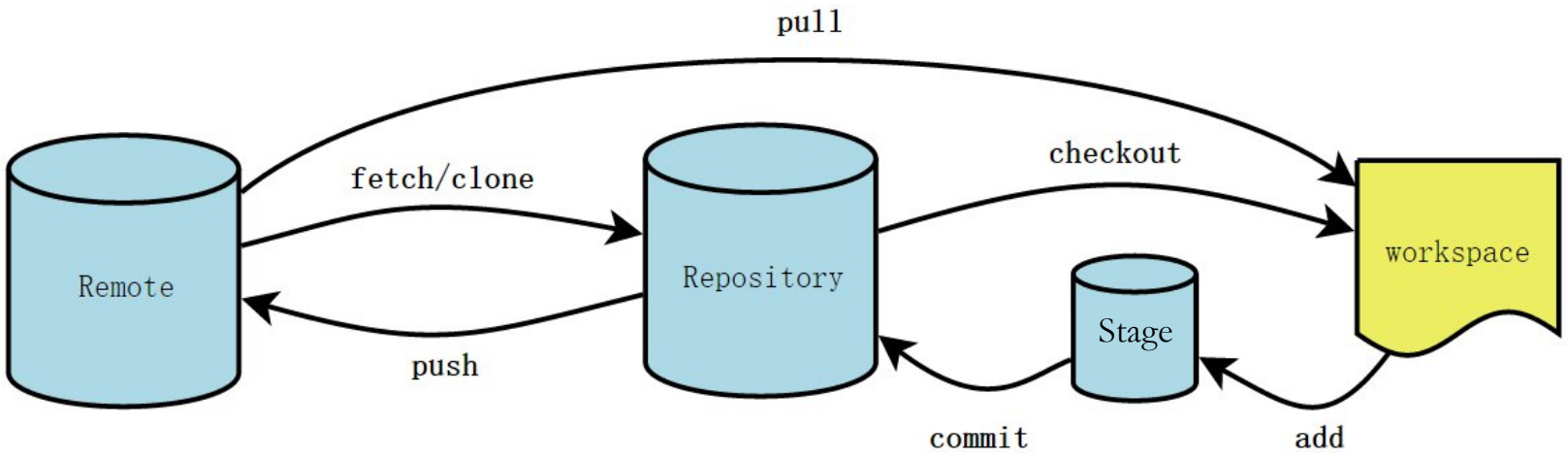
Revert



Assigment 2: Answers

- $ git diff <id of the second commit> <id of the third commit>
- $ git revert <id of the second commit>
- $ git add name
- $ git reset HEAD name

Git Remote


pull
fetch/clone
checkout
Remote
Repository
workspace
Stage
push
commit
add

Remote

- **Remotes are remote git repositories**
- **You can pull and push changes to remote repositories**
- **You can also fetch branches from the remote repository**

- **Remotes have to be added by using:**

  $ git remote add <name> <url>

- **You can list all remotes with**

  $ git remote

- **You can list details about a specific remote with**

  $ git remote show <name>

Creating a central repository

- **While technically any repository can be used as a central repository, it is sometimes better to create a ”bare” repo to act as a central unit**

  If you use a full git repository as a central repo, you will get errors
- **Bare repositories contain just the contents of the .git folder, and no working directory**
- **They cannot be locally modified**

  Instead all changes must be pushed to them
- **Creation of a bare repo:**

  $ git init --bare

Cloning remote repositories

- **You can easily clone remote repositories with git**
- **Every clone is always a full copy of the remote repository**

  => Full backup too if push comes to shove
- **Usage:**

  $ git clone <url>

  The remote repository is automatically called origin

  To use something else as a name for the origin, use

  $ git clone –o <name> <url>

Assignment 3: Create a remote repository & two remotes that use it as origin

- **Create a new folder**

  Create a headless repository there
- **Create another folder elsewhere**

  Clone the first repository
- **Create another folder elsewhere**

  Add the first repository as a remote repository (= don't clone it)

Assignment 3: Answers

```
$ mkdir newrepo
$ cd newrepo
$ git init --bare
$ cd ..
$ mkdir local1
$ cd local1/
$ git clone ../newrepo
$ cd ..
$ mkdir local2
$ cd local2/
$ git init
$ git remote add remote ../newrepo
```

Pushing changes

Sends your changes from your current branch to a remote repository

If the repository has changed since you last pulled from it you will be shown an error and you have to pull and merge the changes before you can push

- There might be conflicts!
- The error message usually is something like “no fast-forward commits”

The changes will be taken from your current branch by default

Usage:

- $ git push <remote name> <branch on remote>
- If you are in a cloned repo you need not specify remote names or branches unless encountered with an error
- If you are on a repo with an added remote you need to specify

Pulling changes from remote

Similar to push, but the other way around

Pulls changes from the remote to the current branch

If any changes have been made locally to a file that has changed at the remote end, this causes a conflict

- Conflicts need to be resolved by editing the conflicted files
- After the conflicted file has been fixed, commit your changes and then push to the remote
- Git pull tries to merge conflicts automatically

Usage

- $ git pull <remote-name> <remote-branch>
- If you are using a cloned repo, you don't need to specify a remote name or branch
- If you use an added remote you need to remember to specify both

Fetching changes

Similar to pull, but doesn't try to merge the changes instantly

Usage

$ git fetch <remote-name> <remote-branch>

When using fetch, you have to merge the changes manually in any conflicted files

You merge the changes by using git merge

$ git merge <filename>

Or if automatic merge fails, use the default merge tool of your system

$ git mergetool <filename>

Assignment 4: Pushing & Pulling & Conflict

- **Go to the cloned repo**
  - Create new empty file called class.file
  - Commit your changes
  - Push to the remote repository
- **Go to the empty repository with the added remote**
  - Pull changes from the remote end
- **Add a line to class.file in the second repo**
  - Commit your changes
  - Push to remote
- **Go to first repo**
- **Edit class.file and and add new line too**
  - Commit your changes
  - Try to push your changes to the remote

Assignment 4: Answers

```
$ cd ~/local1
$ touch class.file
$ git add class.file
$ git commit –m ”added class.file”
$ git push

$ cd ~/local2
$ git pull remote master
$ echo ”line” > class.file
$ git add class.file
$ git commit –m ”added a line to class.file”
$ git push remote master

$ cd ~/local1
$ echo ”line too” > class.file
$ git add class.file
$ git commit  -m ”added a new line too”
$ git push
$ git pull
$ git mergetool class.file
$ git commit –a
$ git push
```

Branches

- **Branches are one of Git's killer features**
- **Branches always have a starting point in a commit in another branch**
- **Branches are "parallel lines of development"**
  - You can commit, revert (etc.) in a branch without having to worry about it affecting other branches
  - Until you merge of course.
- **You can move changes between branches by using git merge**
- **Branches are extremely flexible!**

Branches: SVN vs. Git

- **Branches are fundamentally different in git and SVN**

  SVN Branches are, essentially, just folders, that increase the global revision number

  Git branches are fully parallel lines of development that are completely autonomous

  => Makes switching between branches really fast

  => Makes it easier to manage branches

  In git there's no predefined trunk or branches

  Instead, the trunk is more or less just some branch that has been designated as such

  Allows for flexible strategies

Feature branches

- **"Feature branches" are commonly used:**
  - Branches that are specific to one feature
  - When development of that feature is done, the branch is cleaned and merged to a "development branch"

Creating branches

- **$ git branch <name>**
  creates a new branch called <name>

- **$ git branch**
  shows a list of branches

Switching between branches

- **Switching is done with a checkout**
  Checkout == Change Branch
- **NOT to be confused with SVN's checkout**
  SNV's Checkout == Git Clone

- **You can always see the branch you are in by typing**
  $ git branch

Checking out

- **$ git checkout <name>**
  changes in to a branch called <name>

- **$ git checkout -b <name>**
  creates and checkouts to a branch called <name>

Assignment 5: Branching

- **Go to the original repository that was created in Assignment 1**
- **Create a new branch.**

  You can call it “family”.
- **Checkout to that branch.**
- **Create a new file and call it ‘father’.**

  Insert your father’s full name to the file and commit.
- **Checkout back to the ‘master’**

  Verify that the ‘father’ file only exists in the branch ‘family’

Assignment 5: Answers

- $ git checkout -b family
- <create file father, and write your fathers name in it>
- $ git add father
- $ git commit -m “Added file father”
- $ git checkout master
  $ ls

Merging branches

Adds changes to the current branch (by default)

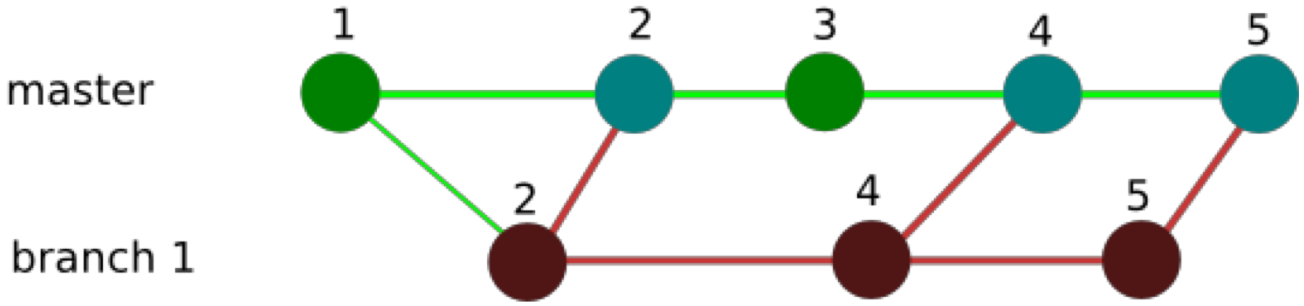
The changes are usually taken from other branches

Merges changes that have been committed after the two branches diverged

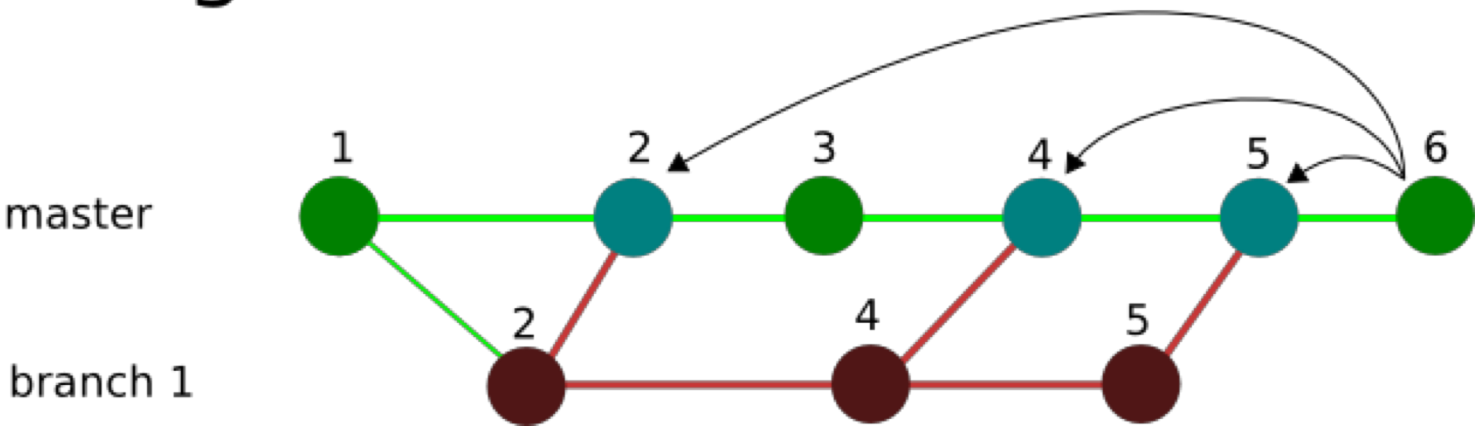
Two kinds of merges

- *Fast Forward merge*
- *Non-Fast Forward*

merge with fast forward



merge with --no-ff



Merging branches: Usage

- **$ git merge <branch>**
  merges changes from <branch> to the current branch
- **$ git merge --no-ff <branch>**
  merges changes from <branch> to the current branch and always creates an extra commit about the merge

  This makes it easier to undo merges

Tags

There are 2 kinds of tags "lightweight tags" and "tag objects"

Lightweight tags

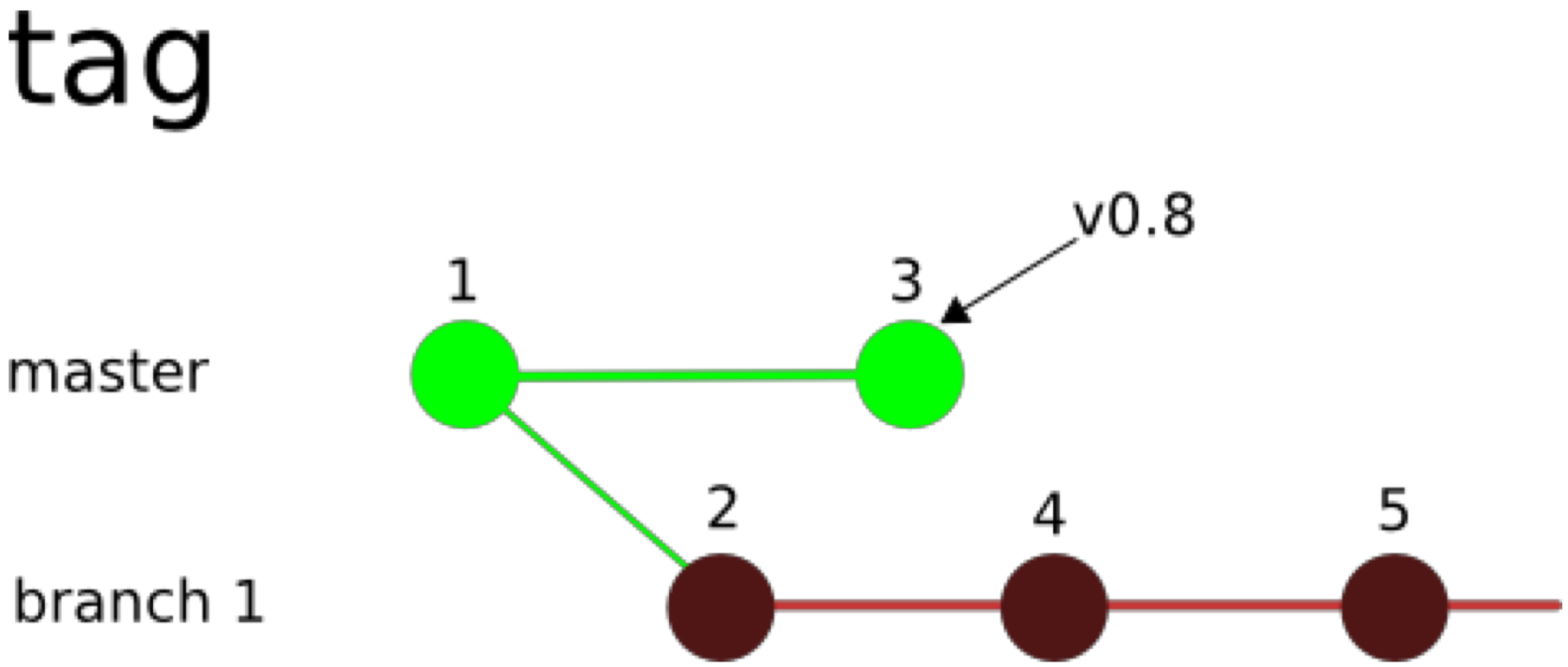
- Simple aliases for tags
- "branch that never moves"

Tag objects

- Added as objects to the git database
- Can be signed using gpg
- Require tag messages

Tags create a sort of a frozen branch, which to which you can perform a checkout

tag visualization



Usage

- **Lightweight tags**

  $ git tag <tagname> <commit>
  creates a tag called <tagname> that points to <commit>

- **Tag objects**

  $ git tag -a <tagname> <commit>
  creates a tag object and opens a text editor where you can type a tag message

  tag objects can be signed

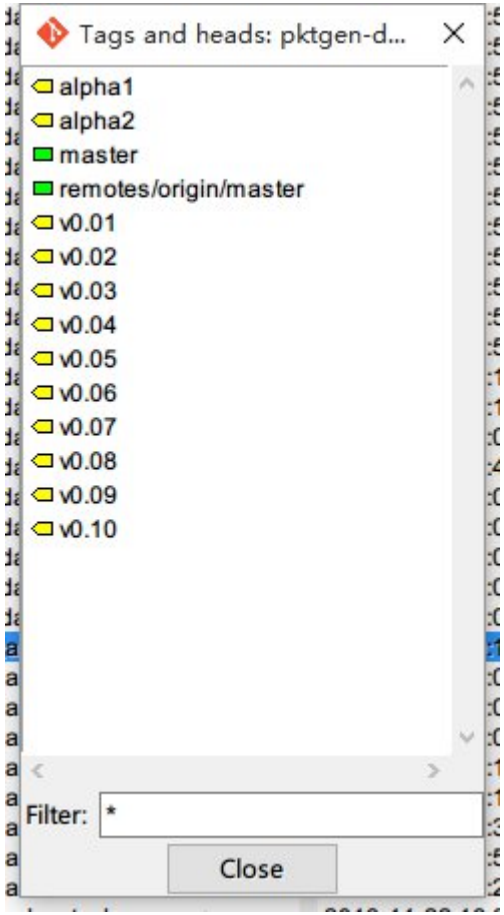
- **List all tags**

  $ git tag

Git tag

- Delete tag
  - git tag -d <tag name>

Show tag with gitk



Assignment 6: Merging

- **Merge your ‘family’ branch to ‘master’.**
  Verify that the master branch contains the contents of the family branch
- **Create a new lightweight tag to master called release-1.0**
  Verify that the tag exists

Assignment 6: conflict

- **Create new branch 'test'**

  Create file called fatha.txt and add your fathers second name on the *first line*. Commit changes.

- **Checkout to master branch**

  Create file called fatha.txt and add your fathers first and last name on the *first line*. Commit changes

- **Merge the branch test to master**

  Resolve conflict using git mergetool

  Retry merge

Assignment 6: Answers

- $ git checkout master
- $ git merge family

- $ git tag version-1.0
- $ git tag

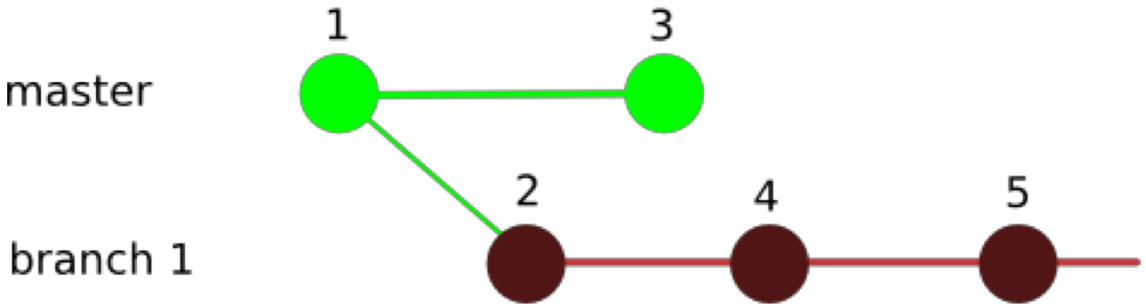
Rebase

- **Changes the "starting point" of a branch**
- **Can be used in feature branches when the "development branch" changes drastically and it affects the feature branches**
- **Usage:**

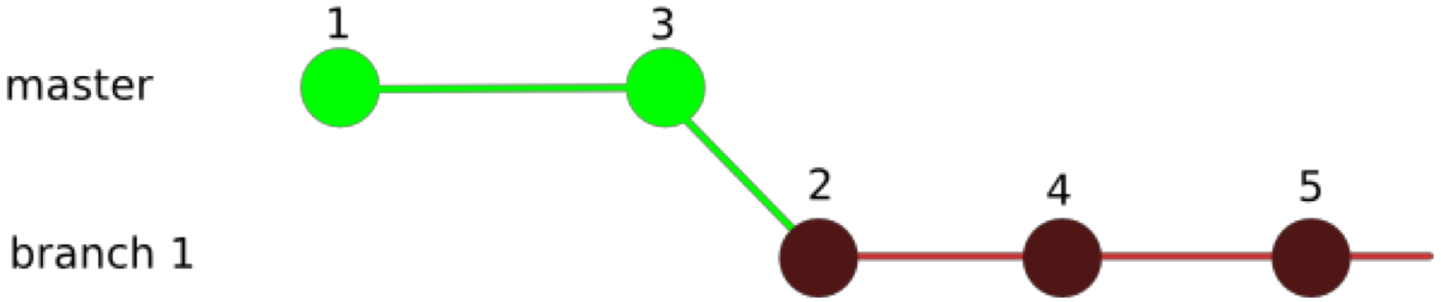
  $ git rebase <branch>

  sets the starting point of the current branch to <branch>

before rebase



after rebase



Common branch types

- **master branch**
  Every commit is a release
- **Release branch**
  Use tags to mark releases
  Used for release specific changes before merging to master
- **Feature branches**
  Work is done in feature branches
- **Development branch**
  Features branches are merged here before going to release
- **Testing branches**
  Optional, you can also use tags instead in the release branch
- **Hotfix branches**
  Branched from release branch
  Merged into development before going to release

Branching example

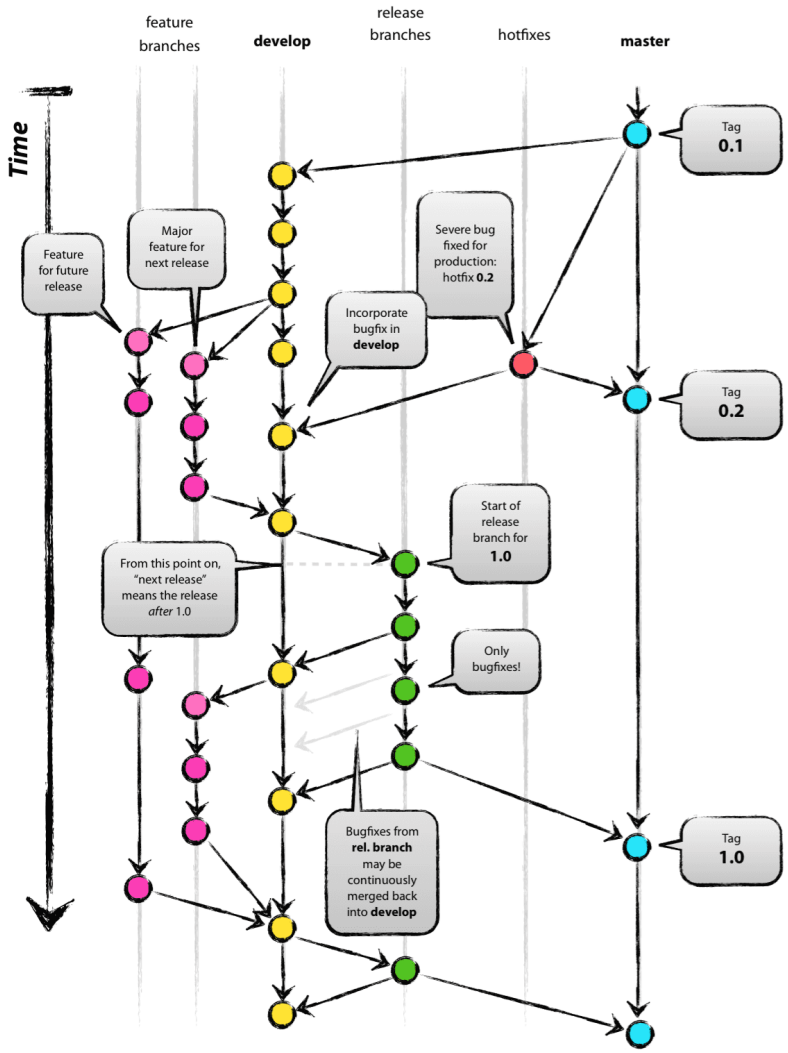
- **2 "main" branches: master and develop**
  master
    Production ready state
    1 commit = 1 release
  develop
    approved changes
    nightly builds
    "integration branch"
- **Feature branches**
  Branches where work is being done
  Specific to one feature! (Bug #1234 for example)
  Unit tested and *reviewed* before merged into development
  Removed after merge to development
  use --no-ff when merging to development!

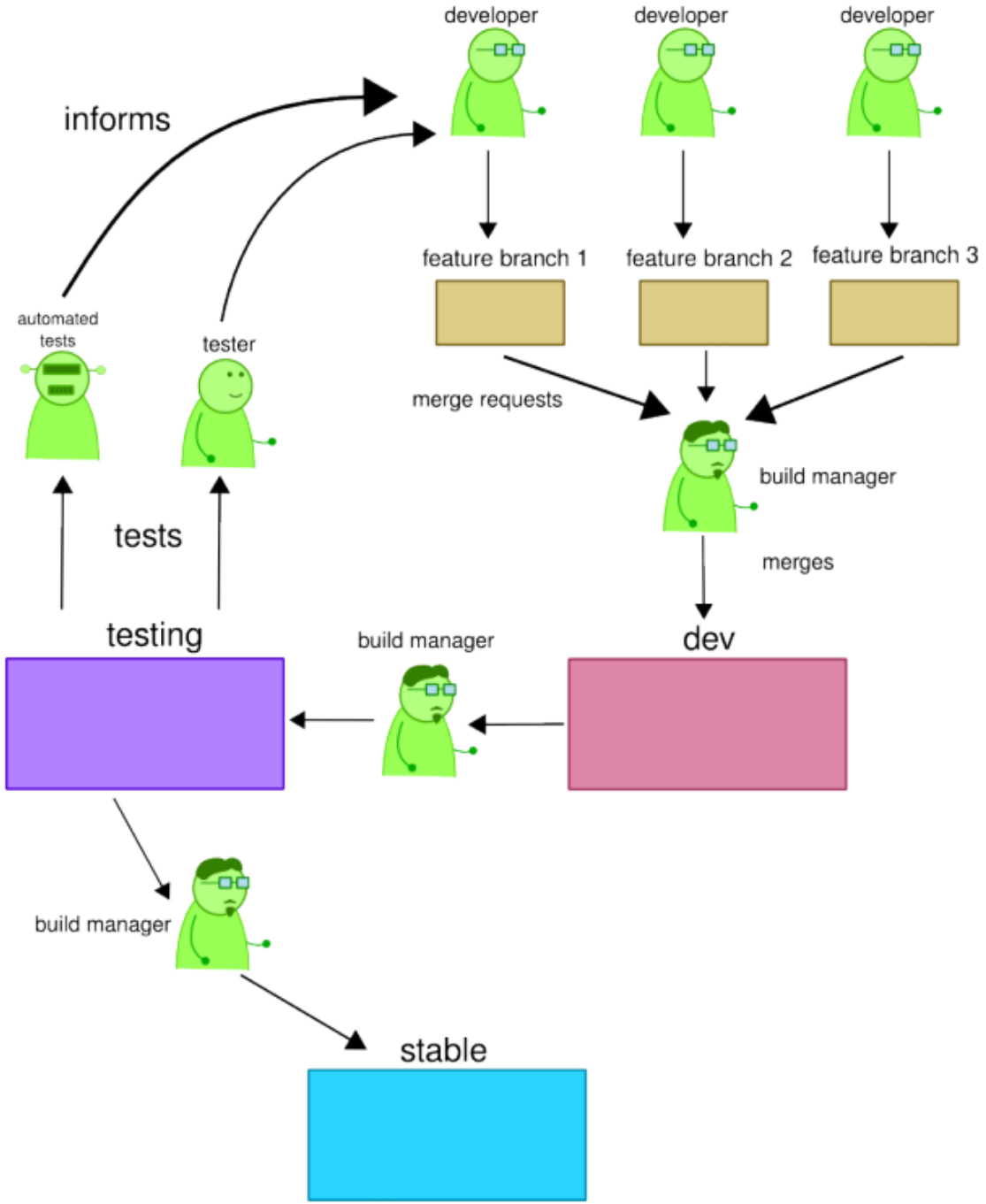
Branching continued

- **Hotfix branches**
  - Critical bugs in production
  - Branched from release
  - Merged to development and master
- **Release branches**
  - Release specific changes
  - Small bug fixes
  - Used so that development can be used for new features
  - When the release is done, merge to master

Gitflow

Gitflow


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


developer
developer
developer
informs
feature branch 1
feature branch 2
feature branch 3
automated tests
tester
merge requests
build manager
tests
merges
testing
build manager
dev
build manager
stable

Git reset - Advanced stuff

- **Specifically, git reset is a tool for resetting HEAD to a desired stage**
- **Most commonly it is used to unstage commits**

- **Reset general usage**

  $ git reset --<mode> [<commit>]

  Most common modes:

  soft: Resets repository HEAD position to <commit>, leaving your changed files marked as ”Changes to be committed”

  hard: Resets the index and working tree. Any changes to tracked files in the working tree since <commit> are discarded.

  mixed: default action without the <mode> identifier. Resets the index but not the working tree (i.e., the changed files are preserved but not marked for commit) and reports what has not been updated.

Git reset – Advanced stuff

- **Permanently removing commits**

  => Primarily should not be used, instead use *revert*

  => If you are absolutely sure,

  $ git reset --hard HEAD~3

  Removes last 3 commits

- **Undo merge or pull**

  $ git reset --hard ORIG_HEAD

  Git stores the local head in ORIG_HEAD-file after merges or pulls

  Restore status to Original HEAD

  => undoes whatever happened

Blame it on others

- **blame allows you to see who has modified a file on a line by line basis**
- **blame prints out the entire file accompanied by the name of the user who made changes to that specific file**
- **usage example:**
  $ git blame <filename>

.gitignore

- You can make sure that no unnecessary files can be added to your repository by using a .gitignore file
- it contains filenames and paths that cannot be added to the repository and will not be shown in git status when they change
- it should be placed in your projects root directory
- it can be tracked by git just like any other file

Example .gitignore file

#ignore all .serializedconf files
**.serializedconf*

#ignore all .pyc files
**.pyc*

#ignore all files in build dir
*Build/**

Patching

- **Patches are basically diff –files that contain the differences between two files**
- **Git allows you to create patches easily**
- **Example patch:**

```
--- hello.c      2006-04-07 10:05:08.000000000 +0000
+++ hello2.c     2006-04-07 10:12:47.000000000 +0000
@@ -1,5 +1,6 @@
#include <stdio.h>
-void main() {
- printf("Hello word");
+int main() {
+ printf("Hello world");
+ return 0;
}
```

Create a patch

- **Usage**

  $ git format-patch master --stdout > name.patch

  Or alternatively you can just use git diff to generate a patch for single files between two revisions:

  $git diff –p <STARTING REVISION>..<END REVISION> <file> > name.patch

Apply a patch

- **1: Test its contents**

  $ git apply --check fix_empty_poster.patch

  If there are no errors, you can apply it with

- **2: Apply**

  $ git am --signoff < fix_empty_poster.patch

Git submodules

- Git submodules allow you to include external Git repositories in your Git repository
- Submodules are always a reference to a single commit in the external repository
- Git submodules have few quirks which should be taken into account when using them

Adding a submodule

- **You can add a submodule to a repository using**

  $ git submodule add http://URL-TO-A-REPO/ SUBMODULENAME
- **Example:**

  $ git submodule add URL jquery
- **After the submodule has been added, you have a new file in your repository: .gitmodules**

  [submodule "jquery"] path = jquery
  url = https://github.com/jquery/jquery
- **All submodules have similar entries in the .gitmodules file**
- **You can check what commit your submodule points to by using**

  $ git submodule status

Cloning a repository with submodules

- **When you clone a repository with submodules in it, you get empty directories for the submodules**
- **In order to "fill" the submodules you have to use**

  $ git submodule init

  Which will initialize the submodules
- **After initialization you have to use**

  $ git submodule update

  to update the reference to the submodule to match the parent repo
- **$ git submodule update**

  has to be called whenever a submodule reference changes in the parent repo

Dos and Don'ts with submodules

Super = project containing the submodule(s)

ALWAYS commit changes to the submodules before committing changes to the super.

ALWAYS commit changes in the submodules to a public repository

DO NOT add the submodule directory to tracked files using
$ git add submoduledir/

ALWAYS add the submodule without the trailing slash eg.
$ git add submoduledir

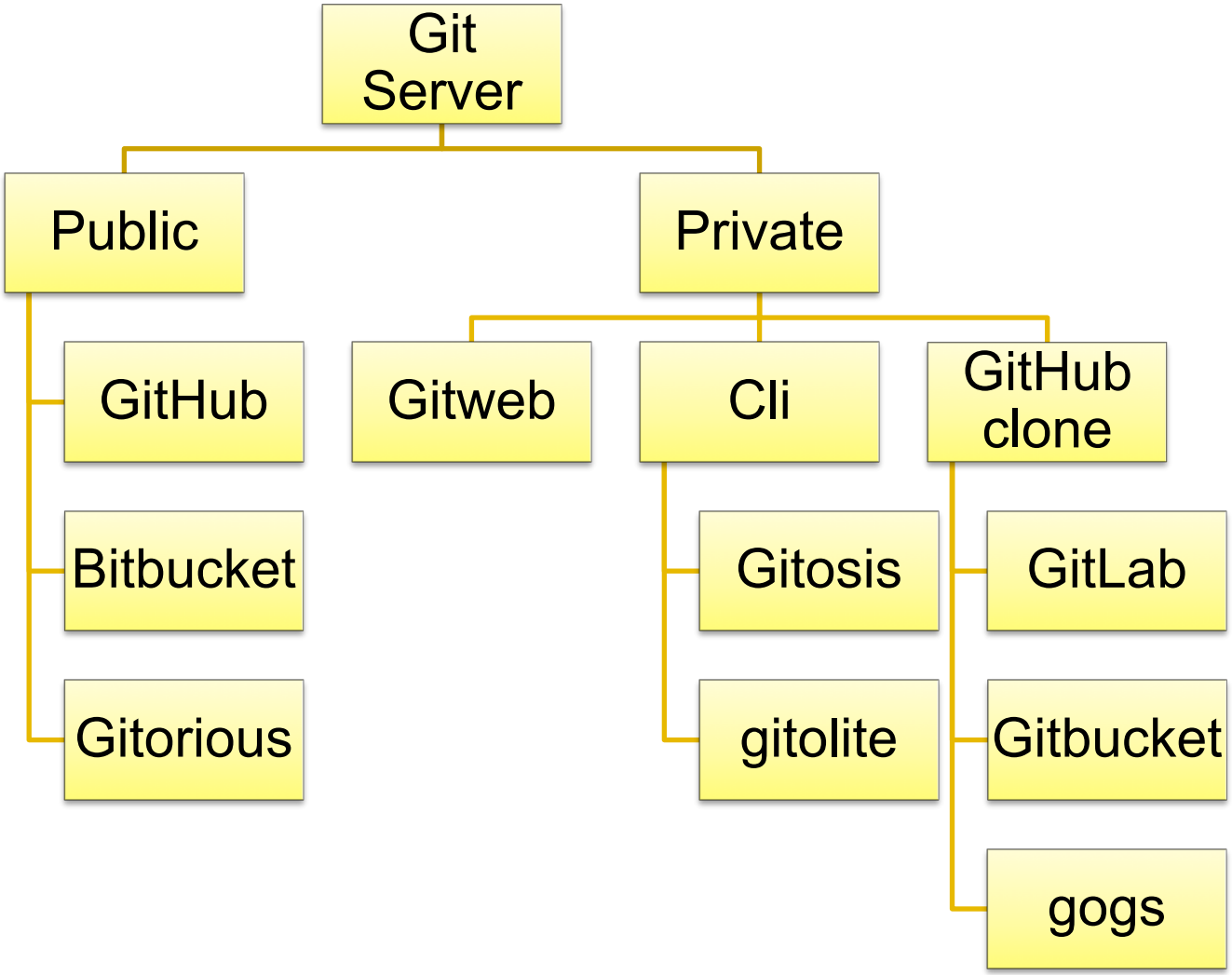
- **Otherwise Git thinks that you want to remove the submodule and add the contents of the submodule to super**

Repository authentication

- **Git itself does not contain a method of creating private repositories**
- **Instead, 3rd party software is needed to manage the repository access rights**
  - Gitolite
  - Gitorious
  - Gerrit (auditing, already explained)

- **Usually done through ssh**
- **90% authentication is handled with SSH Keys**
  - BUT normal ssh keys DO NOT work with TortoiseGIT
  - TortoiseGIT can use Putty keys
  - PuttyGen is installed with TortoiseGit

Git Server


Git Server
Public
Private
GitHub
Bitbucket
Gitorious
Gitweb
Cli
GitHub clone
Gitosis
gitolite
GitLab
Gitbucket
gogs

Reference

- **ProGit** https://progit.org/
- **GitMagic** http://www-cs-students.stanford.edu/~blynn/gitmagic/
- **Learn Version Control with Git** http://www.git-tower.com/learn/git/ebook
- GitFlow http://nvie.com/posts/a-successful-git-branching-model/

Bonus

- Git Play with SVN
- git-svn provide ability to access SVN repository
- git svn clone <url> ➔ svn checkout <url>
- Git svn rebase ➔ svn update
- Git svn dcommit ➔ svn commit
- WARNING: Single commit line required!
- Rebase your git branch into single line before dcommit!

Partnering for Smart City & IoT Solutions

驱动智慧城市创新　共建物联产业典范