267424 Custom Git-Gradle

Web Age Solutions Inc.
USA: 1-877-517-6540
Canada: 1-866-206-4644
Web: http://www.webagesolutions.com

For customizations of this book or other sales inquiries, please contact us at:

USA: 1-877-517-6540, email: getinfousa@webagesolutions.com
Canada: 1-866-206-4644 toll free, email: getinfo@webagesolutions.com

Web Age Solutions Inc.
439 University Ave
Suite 820
Toronto
Ontario, M5G 1Y8

Table of Contents

Chapter 1 - Introduction to Git

Objectives

In this chapter, we will discuss:

- What is Git
- Some Basic Git Concepts

1.1 What is Git

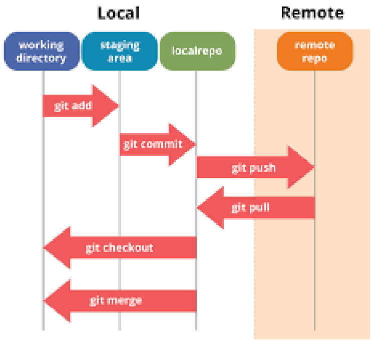
- Brief History
    - The Linux kernel was initially stored in a proprietary VCS called BitKeeper
    - In 2005, there was a falling-out when BitKeeper became a pay-to-play product
    - Up until then, BitKeeper had been provided free to the Linux kernel team
    - Linus Torvalds decided to create his own VCS
        - Git

1.2 Git's Design Goals

- Essentially, Git was designed with a view towards running the Linux development process.
- Which is...
  - Big
  - Distributed responsibilities
  - Ability to selectively apply patches
  - Spread across thousands of developers
  - Non-linear development (thousands of parallel branches)
  - Did we mention big?

1.3 Git's Design Goals (cont'd)

- Torvalds had specific design goals for git
  - Speed
  - Simplicity
  - Fully Distributed Development
  - No dependency on centralized availability



1.4 Branching and Merging

- Centralized Systems have one central repository
  - Developer checks out a working copy, does work, then commits changes back to central repository
  - What if developers are working on different things? e.g.
    - Different features being added for next release
    - Bug fix to released code
    - Bug fix to a legacy version (n-2)

1.5 Branching and Merging (cont'd)

- Solution is “Branching”
  - System has multiple streams of development, e.g.
    - Main branch
    - Release Branch
    - Development Branch
    - Feature Branches
  - Then, you also need to be able to copy changes from one branch to another
    - Merge
  - These have historically been “hard”

1.6 Centralized Version Control

- Even with branches, you have the problem of working offline
  - Developers now work from home, on airplanes, at coffee shops, etc
  - We’d like to be able to store changes as we work
    - Commit early, commit often!
    - Allowing rollback out things that don’t work
    - At the same time, we don’t want to “break the build”

SUBVERSION

1.7 Distributed Version Control

- Proprietary
  - Sun WorkShop TeamWare – 1990s
  - BitKeeper -1998
- Started to appear as open-source in mid-2000’s
  - Arch -2001
  - Monotone – 2003
  - Darcs - 2003
  - Git – 2005
  - Mercurial (Hg) – 2005
  - Bazaar (2005)

1.8 Git Basics

- Git doesn’t track files, it takes snapshots
  - Traditional systems track files individually, store deltas
  - git takes snapshots of the whole directory, stores the whole snapshot
  - With some optimizations so as not to duplicate data!
- Git maintains a database of your file tree
  - Generally only adds data to the database
  - The data added for a commit naturally forms a chunk that can be moved around

1.9 Git Basics (Cont'd)

- Branching is easy and cheap
  - No longer a rare occurrence, it’s part of daily workflow
- Many “workflows” are possible in git
  - You can keep on doing things as though you had a central repo
  - Feature branches?
  - “Gitflow”
  - “Forking” workflow

1.10 Git Basics (cont'd)

- Push
  - Takes changes committed to the local repository (branch) and applies them to a remote repository (branch)
- Pull
  - Takes changes committed to a remote repository (branch) and applies them to a local repository (branch)
- Push and Pull are the mechanisms that allow distributed collaboration
- Important - You will have a distributed workflow!
  - It isn't possible to "commit" to a central repository
  - All commits are done against a local repository
  - Then pushed to a remote repository (or pulled)

1.11 Getting Git

- http://git-scm.com
- But have a look first – you might already have it
  - Shipped with many Linux distributions
  - Included in Apple Xcode
  - Included in Microsoft Visual Studio
  - Embedded in Eclipse, Netbeans, etc
  - But may require a local install as well

1.12 Git on the Server

- Easiest way is to just allow ssh access to the server
  - “Server” functionality is built into the cmd-line tools
  - Can also access repos in shared file systems
    - But often, ssh will be faster
- Git protocol
  - built-in but not authenticated
  - Won’t route through proxies, etc
- Http/https
  - Easy to setup for read/pull
  - Push can be done, but rarer – usually use ssh

1.13 Git Repository Managers

- Non-Exhaustive List
  - GitLab
  - Atlassian Stash
  - GitHub Enterprise
  - GitBlit
  - scm-manager
  - Perforce
  - There are probably others...

1.14 Git on Somebody Else's Server

- The rise of Git has coincided with the emergence of hosted Git repositories
  - GitHub
  - BitBucket
- These repositories have brought workflow and collaboration along with them, e.g.
  - Pull requests
  - Code review
  - Documentation sites
- Also, there are in-house versions of git repos
  - Atlassian Stash
  - GitHub Enterprise

1.15 Summary

- DVCS is the new way!
- Git is one of the leading options
- There’s a lot to learn!
- Workflow decisions to be made
- You can use hosted repositories or in-house

Chapter 2 - Basic Git Operations

Objectives

In this chapter, we will discuss:

- Definitions
- Getting Started with Git

2.1 Using Git

- Git is fundamentally a Linux command line utility
  - 'git'
- Same functionality is often put into IDEs
- Git for Windows ships with
  - Bash shell that runs under Windows
    - derived from Cygwin
  - GUI
  - Explorer integration (context menu)

2.2 Definitions

- Developer's Work Area
  - Terminology is a little loose.
  - A folder on disk can contain a repository
  - As a hidden folder called ".git"
  - Typically thought of as three areas:
    - Working Copy
    - Repository
    - Staging Area

2.3 Definitions (cont'd)

- Repository
  - A repository is an area that stores a version-controlled image of a folder on disk
  - Recall that Git doesn't track files exactly, it stores snapshots of a repository
  - Repository contains a complete version history
  - Casually, we might say a folder contains a git repository
  - A repository can also exist on a remote server (or actually just anywhere besides 'here')
  - Repository can be 'bare', meaning it doesn't have a working copy

2.4 Repository (cont'd)

- 'git init' creates a repository in the current directory
  - stored in a hidden folder called '.git'
- 'git init --bare' creates a bare repository in the current directory
  - Bare repository is used as a remote repository
  - Similar to the server-side repo in cvs or svn.
  - Doesn't have '.git' folder - repository files are in current directory.

2.5 Definitions (cont'd)

- Working Copy
  - The files and folders contained in the developer's work area, apart from the Git repository
  - We edit and manipulate files in the Working Copy
  - We use git commands (or a gui) to move files between the working copy, staging area and repository

2.6 Definitions (cont'd)

- Staging Area
  - Sometimes called the "Index"
  - Basically the list of files that are part of the next commit
  - 'git add <file>' adds a file to the staging area
  - 'git add .' or 'git add --all' adds everything
  - 'git rm --cached <file>' removes from the index
  - 'git status' shows status of staging area and working copy

2.7 Commit

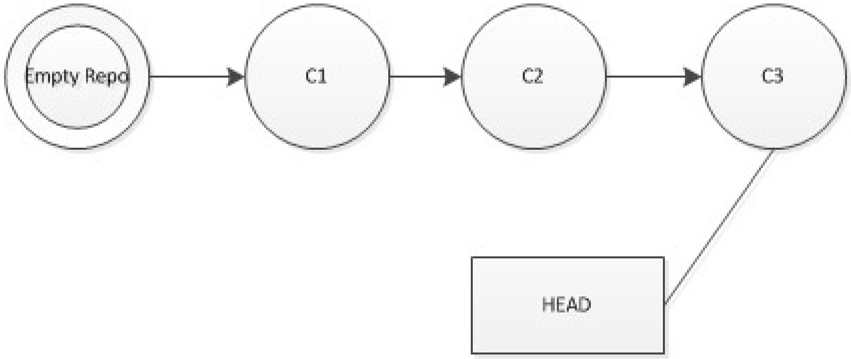
- A snapshot of the working area at a point in time
- Add files to the staging area, then commit --> Files go to repository
- 'git commit' sends files to the repository from the staging area
    - Requires a 'commit message' that is stored with the commit
        - Displays an editor to edit the message
            - 'git config core.editor <editor-exe>' configures which editor
            - Ships with 'vim'
        - 'Use git commit -m "message here"' to supply message on cmd line

2.8 Commit (continued)

- Each commit, when complete, is identified by an SHA-1 hash value
  - e.g. d11b35f57881f7665a16a3195eeaf722fd156e67
  - We can usually use a shortened version of the hash
- Each commit has a "parent" commit, so the history is viewed as a chain of commits.
  - Actually, some commits have two parents - a "merge" commit
- The commit represents a snapshot of the working copy
- We can return to any snapshot at any time
  - 'git checkout <commit-id>'
- The current parent commit is called the "HEAD".

2.9 How to Think About Commits

- The repository is a collection of snapshots, or "Commits"
- Each commit has a parent
- The current parent is the "HEAD"



2.10 Viewing History

- 'git log'
  - Show history
- 'git log -p'
  - Show history with patches

2.11 Configuring Git

- There are a few things git needs to know...
  - Developer's name and email
  - Preferred editor
  - PGP Signing key
    - What? - we'll get to that...
  - Default behaviors
    - e.g. what to do about end-of-line characters
  - Files of interest, files to ignore
  - Activity "hooks"
    - Code that we can execute as part of git's operations

2.12 Configuration Scope

- We can configure Git at a few different scopes
  - System Global - Settings for all users
    - 'git config --system <attr-name> <attr-value>'
  - Global - Settings for the current user (all repositories)
    - 'git config --global <attr-name> <attr-value>'
  - Local
    - 'git config <attr-name> <attr-value>
    - Per-repository

2.13 User Identification

- Think about this for a minute - in a distributed version control system, we have a little problem
  - We can't count on operating system ids or logins, because we're going to be moving commits from one repository to another
  - Repositories could be on different systems (remotes)
  - These systems may not be directly connected
  - Systems may not share an authentication store
- Git needs to record the user's identity information with each commit
- At the very least, you need to be rigorous about email address standards
  - If you have multiple email addresses, be very careful!
- You might need cryptographic verification of contributors' identity

2.14 User Identification (cont'd)

- To configure user id across all the user's repositories

```
git config --global user.name "Jane Doe"
git config --global user.email jane@doe.com
```

- To configure for a local repository only (execute in your working copy)

```
git config user.name "Jane Doe"
git config user.email jane@doe.com
```

2.15 GPG Signing

- An obvious problem with the user identification!
  - Anybody could do 'git config --global user.name billg@microsoft.com'
- Even if it were practical to have git look at user authentication, what about moving commits between different repositories?
  - The only information is what's in the repository.
  - System context can't move with the commit data.
- The solution is GPG signing of commits and tags

2.16 Gnu Privacy Guard

- GPG is Gnu Privacy Guard
- Open-source implementation of RFC4880, PGP (Pretty Good Privacy)
- It's a little out-of-scope for this document

2.17 GPG Basics

- Dual-Key Encryption
- Generate a key pair
- Publish your public key to one or more key servers
- Keys are identified by their "Key Fingerprint" or "Key ID"
  - Key ID is the last 8 digits of the Key Fingerprint
- There is no central authority, unlike X.509 certificates
- "Web of Trust"
  - People "sign" other people's keys, certifying that they know that person and that person owns that key

2.18 GPG and Git

- Git can calculate and record a digital signature for a commit or a tag
- Advantage - committer is positively identified
- Identification is recorded in the repository, can't be repudiated
- To use:
  - Setup GPG
  - Generate a key pair
  - 'git config --global user.signingkey <key-id>' or
  - 'git config user.signingkey' (per-repository)
  - On commit, 'git commit -S'
    - Note Capital-S!

2.19 .gitignore

- Quite often, there are files that we don't want to put in version control
- e.g.
  - Build artifacts
  - Binaries
  - Editor's temporary files
  - Generated source code
- '.gitignore' lets us exclude files from 'git add' etc
- Contains a list of patterns to ignore
- Version-controlled and distributed when we clone a repo
- For non-shared, edit "$GIT_DIR/info/exclude"
  - $GIT_DIR in a working copy is ".git"
  - In a bare repository is just the repository folder

2.20 Other Useful Configurations

- 'man git-config' - tells you the options
- core.editor - Preferred text editor
- commit.template - points to a file for commit messages

2.21 Summary

- Git is a powerful version control system
- Basic concepts
  - Commit records a snapshot of everything that's staged
  - Entire commit history is stored in a repository
  - Working copy is a selected snapshot, expanded out into the file system

Chapter 3 - Branching, Merging and Remotes

Objectives

In this chapter, we will discuss:

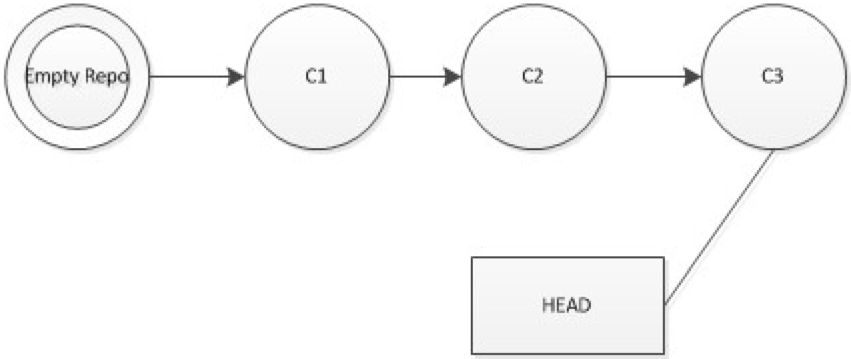
- Branching
- Merging
- Dealing with Remote Repositories

3.1 Branching

- A repository usually corresponds to a project or a development product
- Quite often, there are multiple versions of a product that are "current" at once
  - e.g. v1.0 is in support, 2.0 in development
- Also, there might be multiple development streams going
  - e.g. experimental features that might end up being abandoned
- Traditionally, branches have been difficult to maintain
  - Not so in git!

3.2 Branches in Git

- Recall that the Git repository stores a series of snapshots

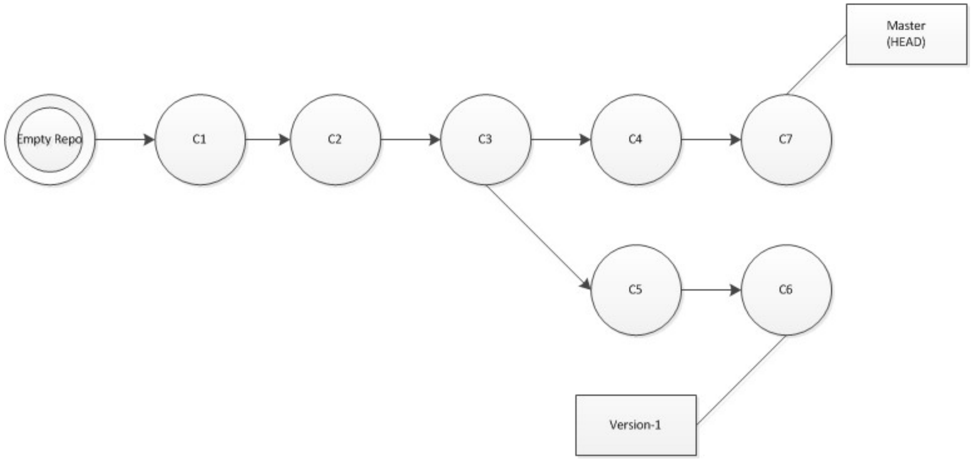


- HEAD is an alias for the "current parent" commit
  - in other words, the commit that will be recorded as the parent of the next commit
  - Usually the last commit

3.3 Branches in Git (cont'd)

- We can actually have more than one "HEAD"
  - I.e. branches
- A "Branch" is just a named alias for another "HEAD"
- So really, HEAD is an alias for the current branch
- Other branches point to other commits
- When you create a repo, git creates a branch called 'master'

3.4 Branches in Git (cont'd)



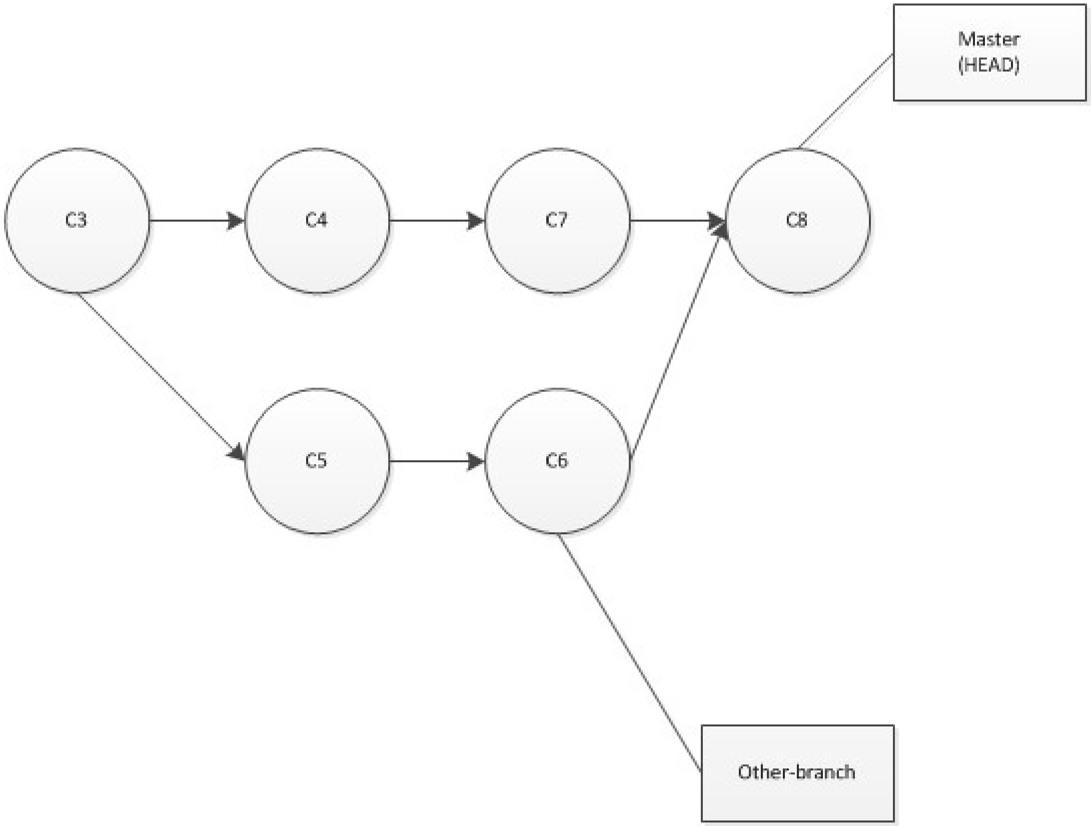
3.5 Branches in Git (cont'd)

- Create a branch:
  - 'git branch <new-branch-name>'
- Switch working copy to a branch
  - 'git checkout <branch-name>'
- Do both
  - 'git checkout -b <branch-name>'

3.6 Merge

- Branches represent different paths of development
  - e.g. V1.x, V2.x, etc
- So, there are changes that happen on each path
- Sometimes, we want to copy changes onto another path
  - e.g. a bug fix in V1.x path, copied onto v2.x
- "Merging" copies all the changes on one branch to another branch
  - doesn't actually end the branch

3.7 Merge (cont'd)



3.8 Merge (cont'd)

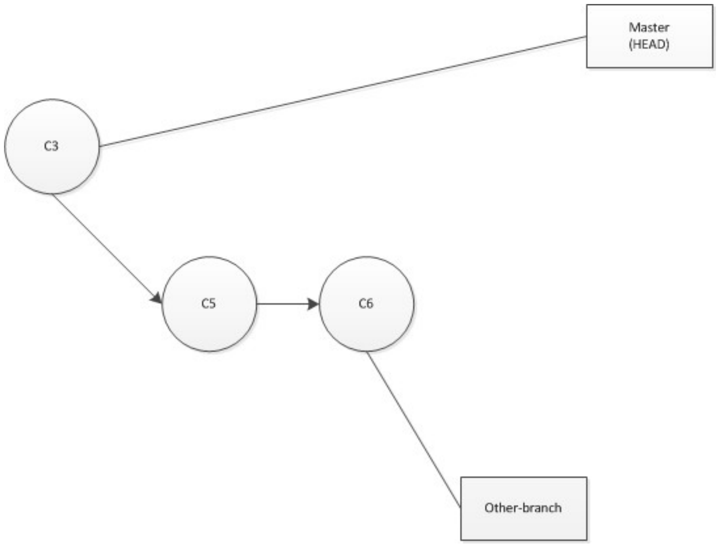
- What's involved in merging?
  - Added files
  - Deleted files
  - Updated files
- Possible conflicts
  - File added on both paths
  - File deleted on one path, but updated on the other
  - File updated on both paths

3.9 Merge (cont'd)

- Checkout your target branch
  - 'git checkout master'
- Merge from the desired branch
  - 'git merge bug-fix'
- Conflicts need to be resolved
- git uses an external tool for this
  - to run 'git mergetool'
- For text files, the tool can often figure things out
- When it can't, it asks the user to resolve the conflicts
  - 'git add <file>' to confirm that you're done.
  - git commit' to finish off the merge

3.10 Fast Forward Merge

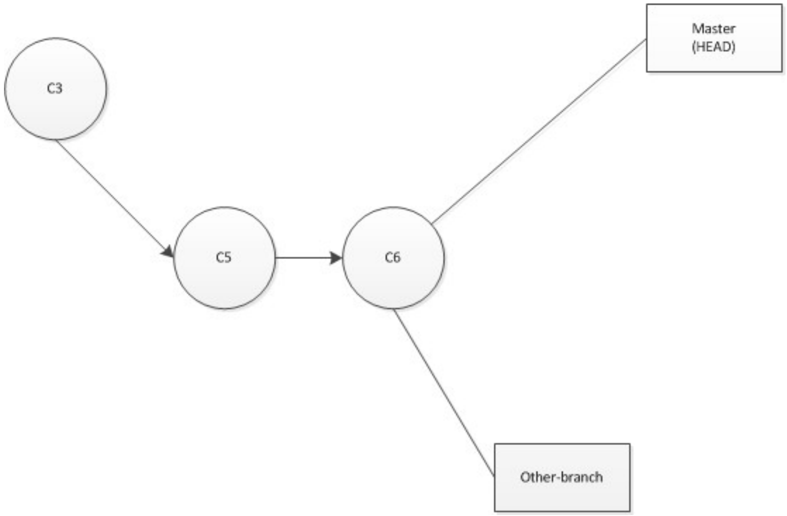
- When the “branch to merge” is a direct descendant of the current HEAD



- Git will simply advance the HEAD

3.11 Fast Forward Merge

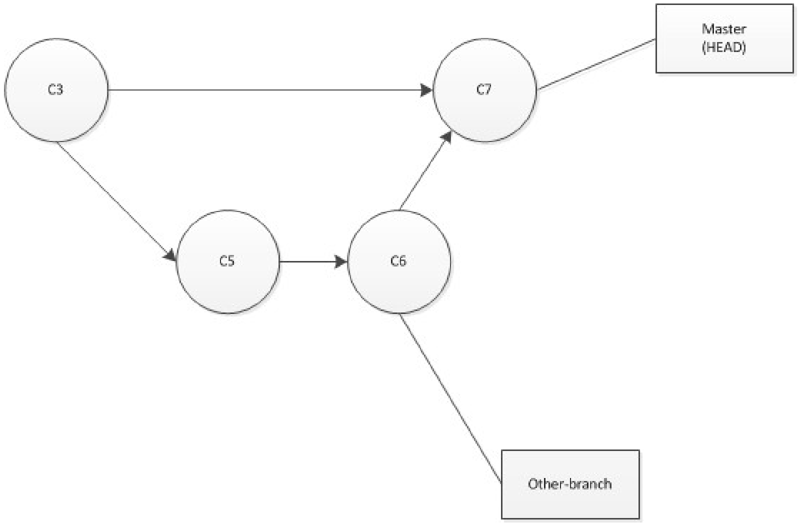
- This is a “Fast Forward” merge



- There's no commit message for a fast-forward merge
  - i.e. no indication that a merge actually happened

3.12 --no-ff

- To make sure that there **is** a commit message, specify '--no-ff' when you issue the merge command



3.13 More Than One Repository

- Git allows for "remote" repositories
- "Remote" could actually be on the same machine, or really remote
- 'git clone <repo-url>' to create a new local repository that has all the information from an original repository
- Remote repositories are referenced by a name
  - 'origin' is setup automatically by 'git clone'
- "Remote branches" are branches in a remote repository
  - 'origin/master' refers to the 'master' branch in the 'origin' repo

3.14 Working with Remotes

- 'git remote add <remote-name> <remote-url>'
  - adds a new remote repository
- 'git remote remove <remote-name>'
  - removes a remote repository
- git remote set-url <remote-name> <url>

3.15 Fetch and Pull

- 'git fetch <remote-name>' gets everything from the remote repository that you don't already have
    - Recall that commits are identified by a hash code, so there's no overlap
        - and actually, everything in git is stored by hash code
- You can then merge a remote branch onto your current branch
- Very common case is where you essentially want to work on a remote branch
    - e.g I have a local branch called 'master' and I'd like to keep it updated with a remote branch like 'origin/master'
    - 'git fetch origin' followed by 'git merge origin/master'
    - So common, this is called a 'tracking branch'
    - 'git pull origin' is a shortcut

3.16 Push

- 'git push <remote-name> <branch>' sends our current branch to the remote server branch
- Pushes the current branch to the named branch on the named remote
  - Think of it like a merge to the remote branch.
    - Hence, subject to conflict resolution

3.17 Pull Requests

- A pull request is simply a request that we send to another developer to "pull" from a branch on our public repository
    - Conceptually, could be a phone call or an email
- Central repository managers often have a built-in capability for pull requests
- Git also has a command to generate a pull request that can be sent over email.
    - git request-pull <starting commit> <url> [<ending commit>]

3.18 Tagging a Commit

- A branch is a pointer to the last commit on that branch
  - This will be the parent of the next commit
- The branch gets “moved forward” every time we do a commit
- A commit is a snapshot of the repository at a particular instant
- Sometimes, we'd like to flag and preserve a particular commit/snapshot as “special”
  - e.g. for a release version
- For this, git offers 'tags'
- There are two kinds of tags in git, 'lightweight' and 'annotated'

3.19 Lightweight Tags

- Just like a branch, except..
  - It doesn't get moved forward on commits
  - It's just a pointer to a particular commit
  - No additional information
- Creating a lightweight tag:

```
git tag v1.0
```

- The tag points to the head of the current branch

3.20 Annotated Tags

- Has additional information stored with it
  - Identity of the tagging user
  - Date
  - Checksum
  - A tagging message
  - Can be signed and verified with GPG
- Creating an Annotated Tag:

```
git tag -a v1.1
```

- Git will open an editor for you to enter a tagging message
  - You can supply the message with '-m <message>'

3.21 Sharing Tags

- Tags aren't included by default when you push to a different repo
  - That's because we're normally pushing a branch
- To push a tag,

```
git push origin v1.0
```

- To push all your tags,

```
git push origin --tags
```

3.22 Checking Out a Tag

- A tag is essentially short-form for a commit id
- So...you can check out a tag, but you can't really do anything with it
  - You end up in “detached head” state
  - You would lose any commits you made, because they aren't referenced by any branch
- Create a branch starting from the tagged commit

```
git checkout -b v1-branch v1.0
```

  - Creates a new branch called 'v1-branch' that starts from tag 'v1.0'

3.23 Summary

- Branches are not overly special - just a moving pointer to a commit
- Current branch is called "HEAD"
- Remotes contain other versions and data that we can pull from or push to
- Tags let us mark a commit for posterity

Chapter 4 - Git Work Flows

Objectives

In this chapter, we will discuss:

- Centralized Repository Work Flow
- Integration Manager Work Flow
- Localized Work Flows

4.1 Work Flows

- Usually, we have more than one developer
  - Developer has a local repository
- We can have more than one repository
- Repositories can have different purposes
  - git really doesn't care what you use a repository for
- How to manage collaboration, publishing, release, maintenance, etc?
  - There are many possibilities, or work flows

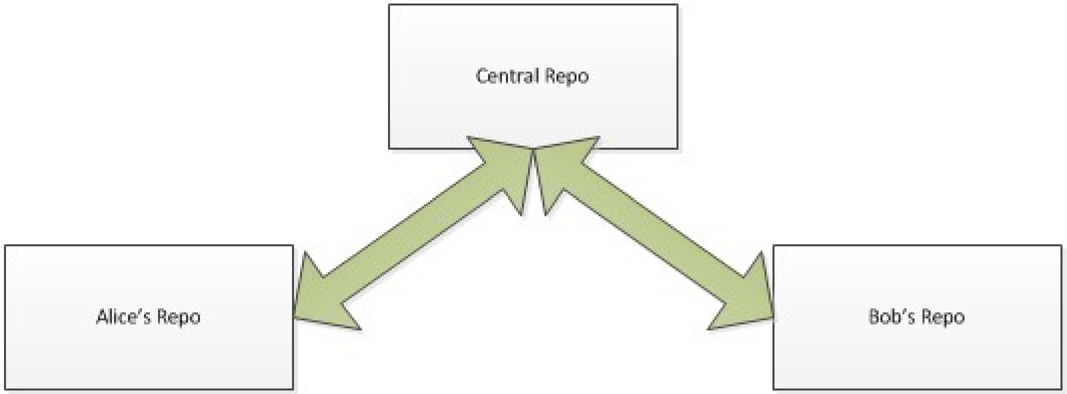
4.2 Local Work Flow

- Traditionally, branching and merging has been hard
- Developers usually worked on a "development" branch
    - common variation - "develop on trunk"
- With git (and other dvcs) branching is much "cheaper"
- Now, it's common to do our work on a branch, then merge back to the mainline when a chunk of work is "done"
- Sometimes known as "feature branches"
- Especially useful combined with remote tracking branches

4.3 Feature Branches

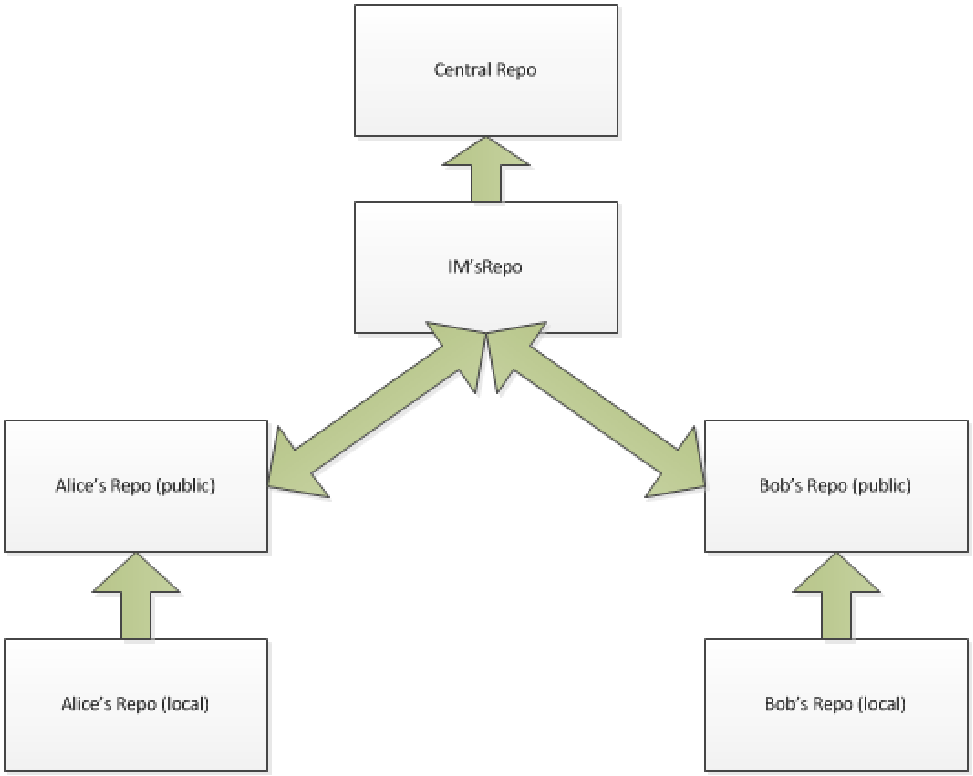
- Develop locally on a feature branch
  - 'git checkout -b new-feature'
  - add/commit...
- Update local master branch
  - 'git checkout master'
  - 'git pull origin'
- Merge
  - 'git merge new-feature'
- Then push to somewhere - subject to overall workflow

4.4 Centralized Workflow



- Use a centralized repository just like you currently do with a non-distributed VCS
- Individual developers are responsible to merge to the remote VCS' master (or other) branch
- Individual developers will see the latest work by other developers, and need to deal with it.

4.5 Integration Manager Work Flow



4.6 Integration Manager Work Flow (cont'd)

- Developers work in their local repo
- Developers publish to a repo that the Integration Manager can see
- Developers tell IM to pull revisions
- IM Pulls from Developer's repos, integrates
- Developers don't have to work together
- IM makes all the integration decisions

4.7 Other Work Flows Are Possible

- Dictator/Lieutenants
- Deployment Stream
- GitFlow
- etc.

4.8 Summary

- Given combinations of branches and multiple repositories, many different work flows are possible
- You need to consider and document the workflow that you'll use

Chapter 5 - Introduction to GitFlow

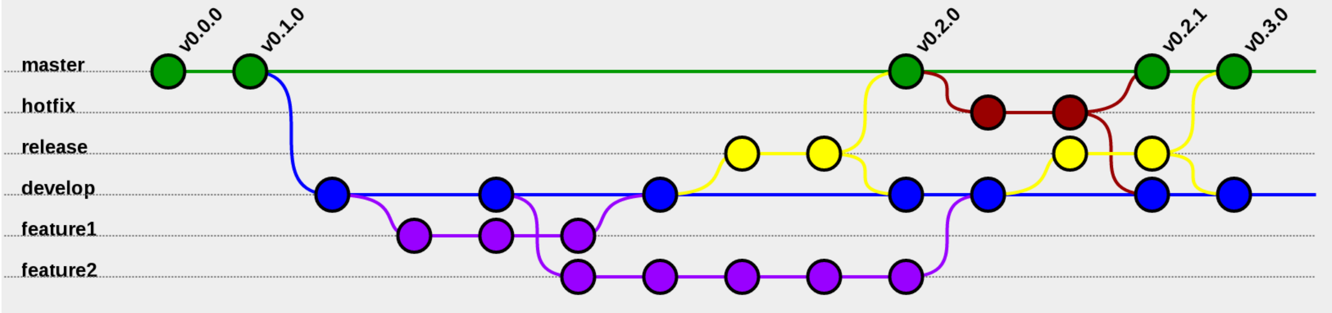
Objectives

Key objectives of this chapter

- Understanding GitFlow
- Understanding Features
- Understanding Releases
- Understanding Hotfixes

5.1 What is GitFlow

- GitFlow is a branching model for Git.



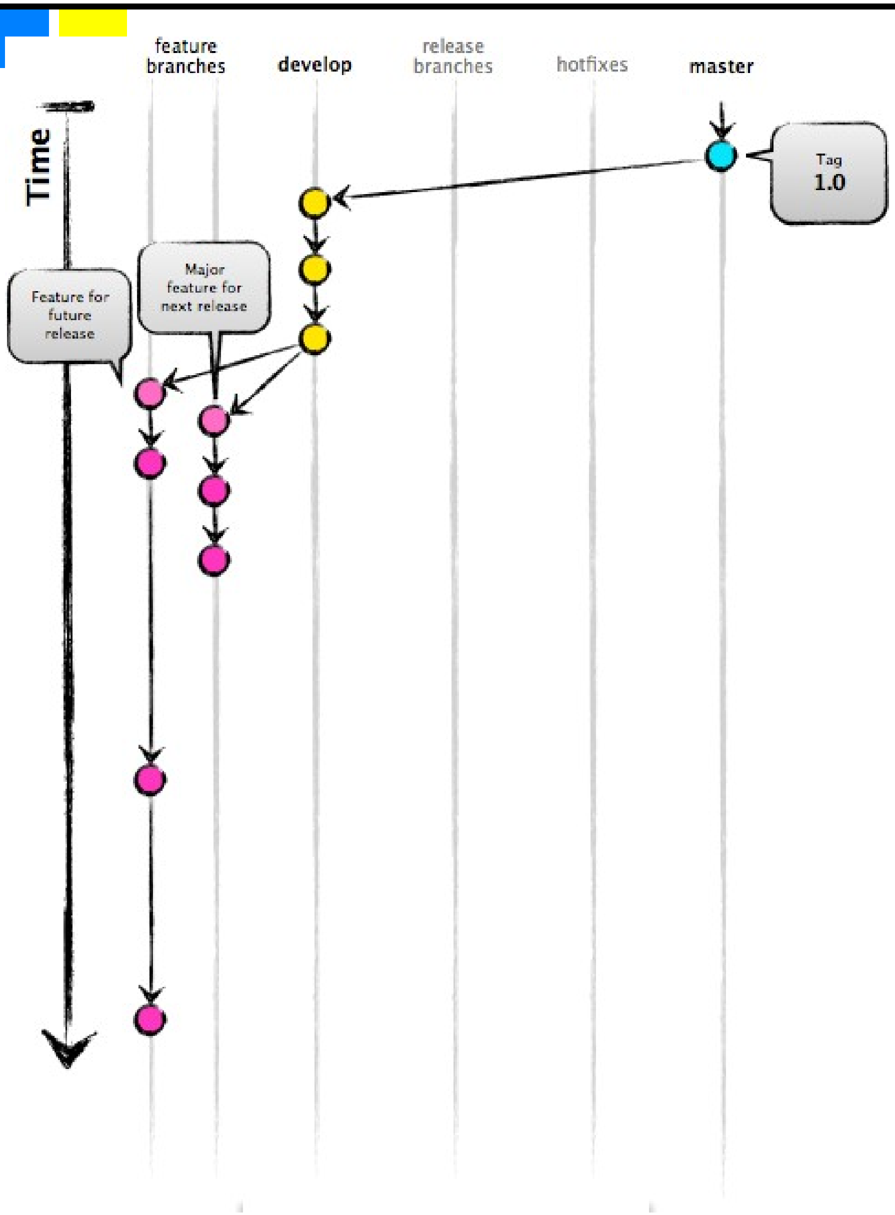
- Created by Vincent Driessen and widely adopted by many organizations
- A set of git extensions to provide high-level repository operations.
- The model is scalable.
- Well suited to collaboration due to the isolation of code across branches
- Maps to the Continuous Integration (CI) and Continuous Delivery (CD) models of code being consistently buildable and deployable in the release and master branches

5.2 Benefits

- Parallel development
  - Isolates new development work from finished work
  - Code is merged back into main body of code when developers are happy that code is ready for release.
- Collaboration
- Release Staging Area
  - New development work is merged back into the develop branch
  - 'Develop' branch is a staging area for all completed features that haven't yet been released.
- Support for hotfixes
  - Hotfix branches – branches made from a tagged release which can be used to make an emergency change.

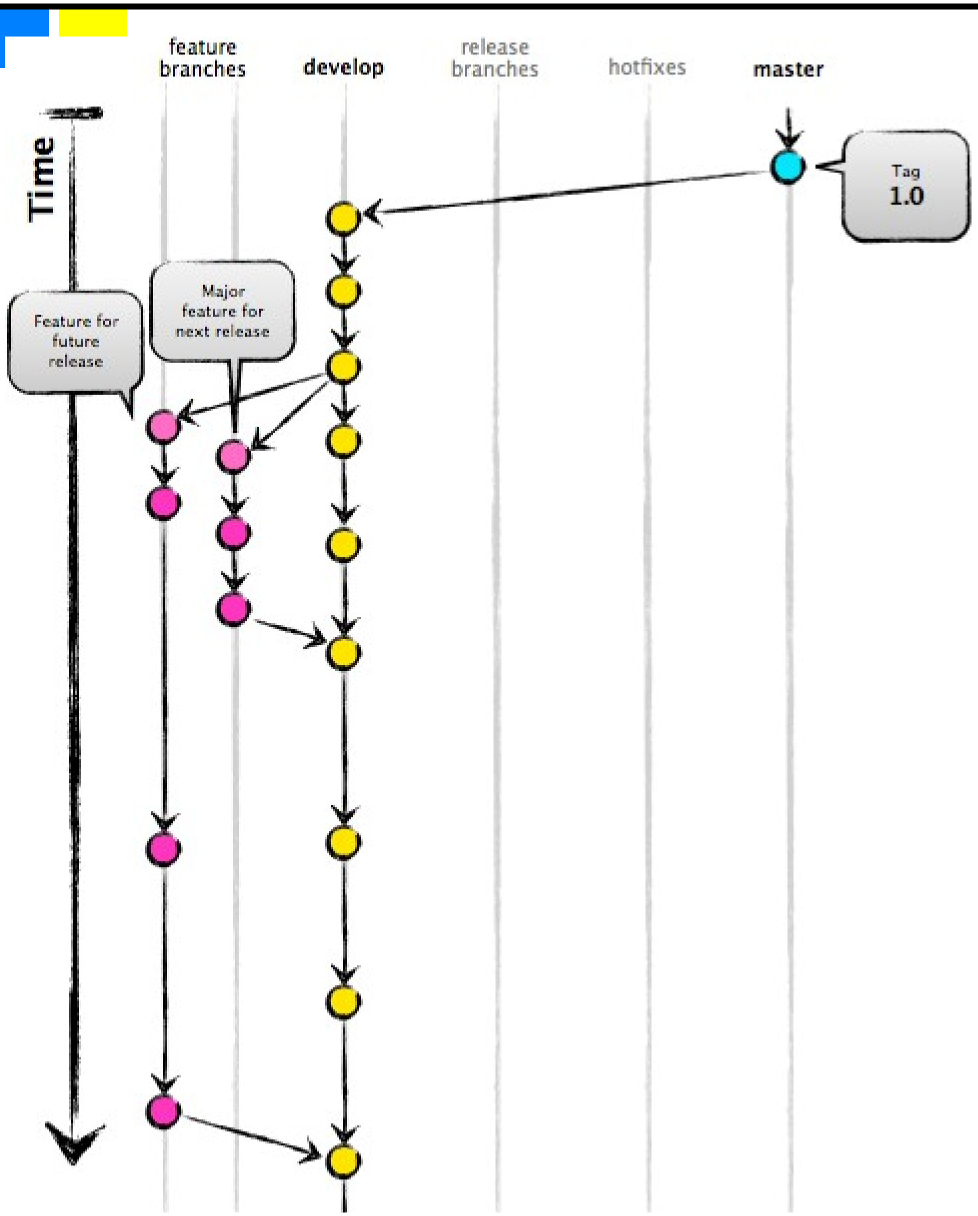
5.3 How GitFlow works?

- New development are built in **feature** branches.


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Major feature for next release
Feature for future release

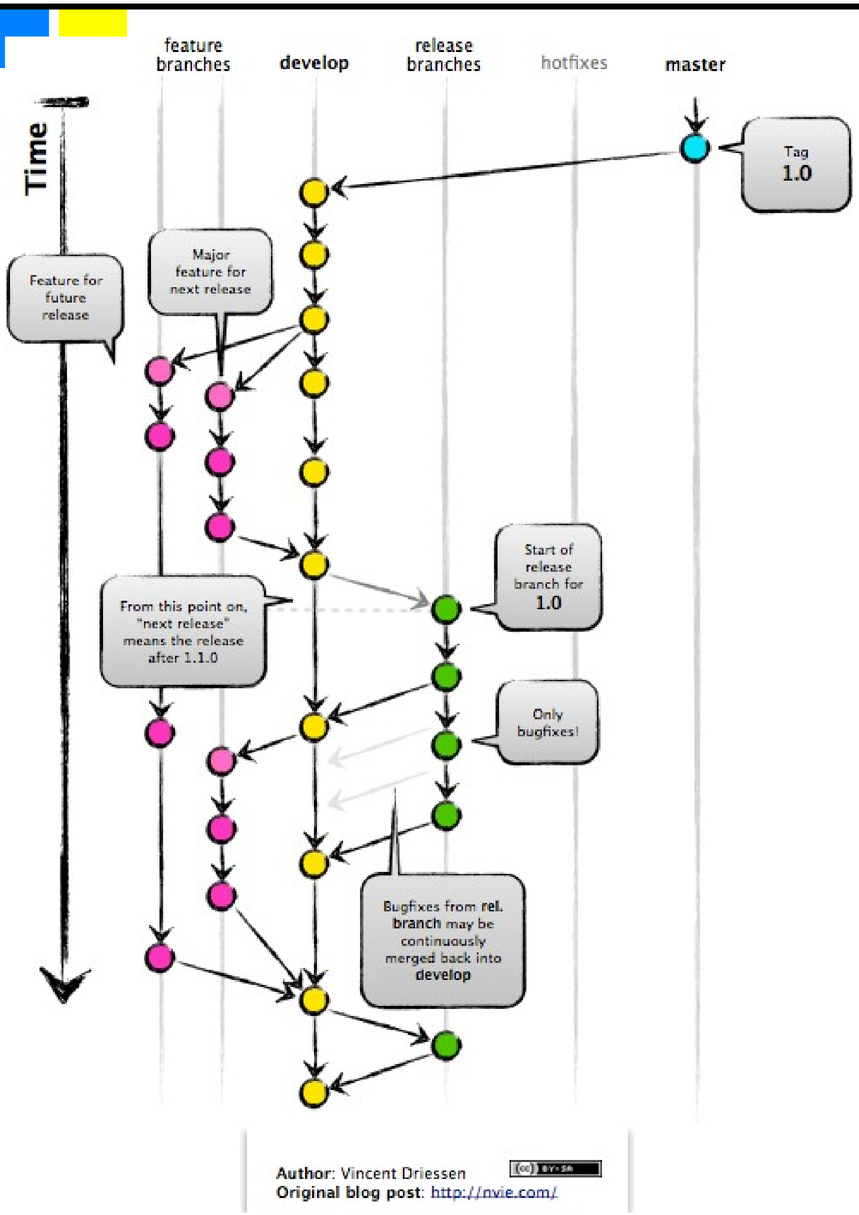
5.4 How GitFlow works? (Contd.)

- Feature branches are branched off of the **develop** branch.
- Finished features are merged back into the **develop** branch.


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Major feature for next release
Feature for future release

5.5 What is GitFlow? (Contd.)

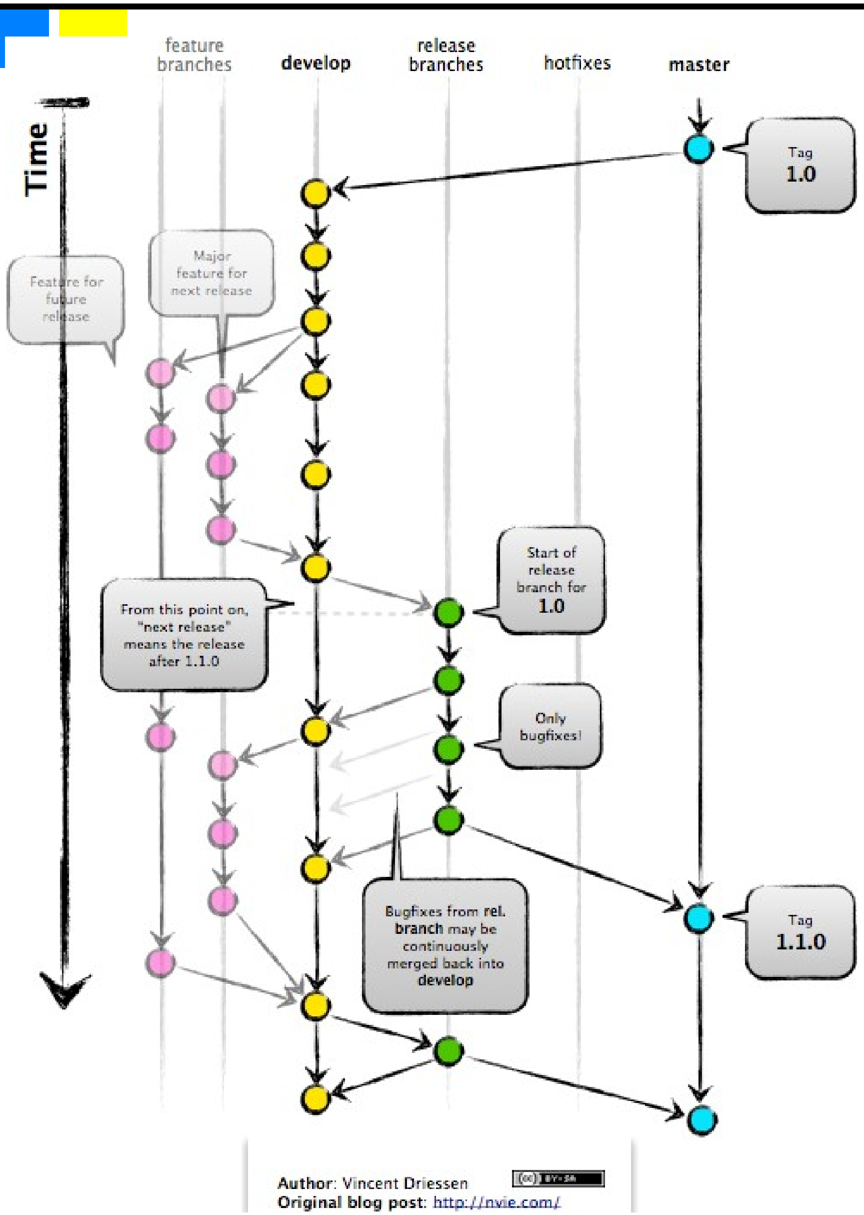
- When features are completed, a **release** branch is created off of **develop** branch.


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop

Author: Vincent Driessen
Original blog post: http://nvie.com/

5.6 How GitFlow works? (Contd.)

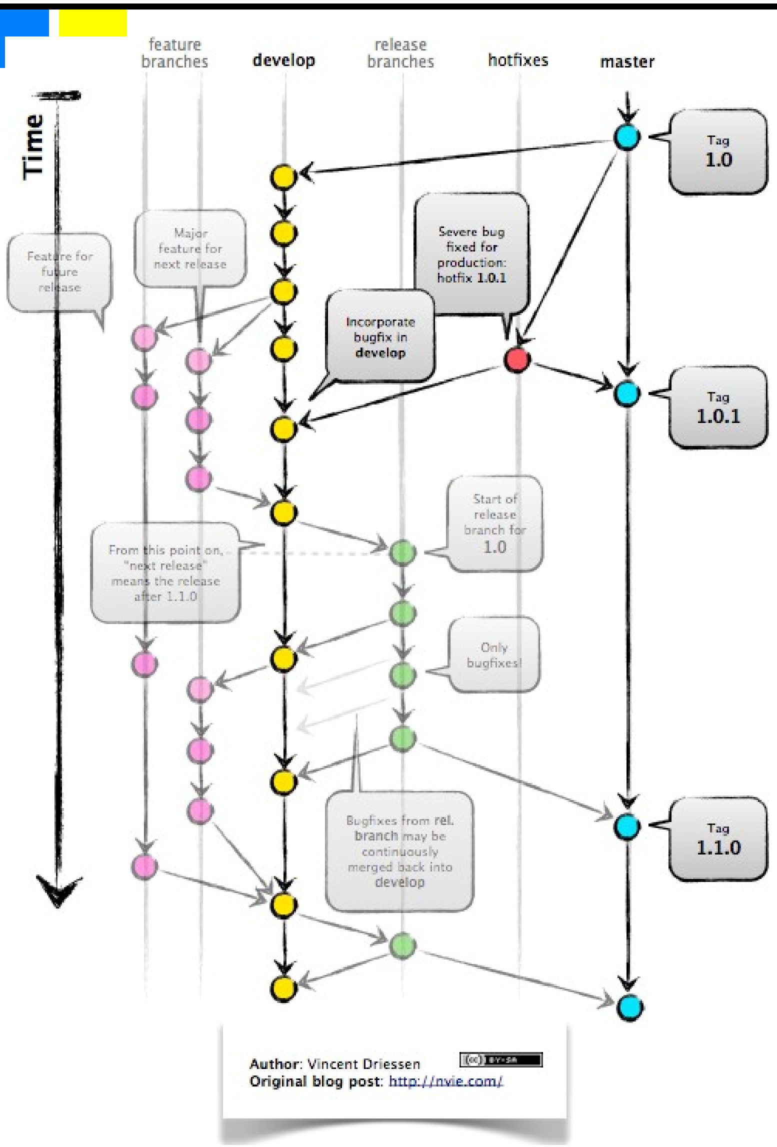
- The code in the release branch is deployed, tested, fixed, redeployed, and retested.
- Release branch is merged into master and into develop branch.


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0
(cc) BY-SA

Author: Vincent Driessen
Original blog post: http://nvie.com/

5.7 How GitFlow works? (Contd.)

- The master branch contains released code only.
- The only commits to master are merges from **release** branches and **hotfix** branches.
- Hotfix branches are branched directly from a tagged release in the master branch.
- Hotfixes, when finished, are merged back into both master and develop to ensure the hotfixes aren't accidentally lost when next release occurs.


feature branches
develop
release branches
hotfixes
master
Time
Tag 1.0
Feature for future release
Major feature for next release
Severe bug fixed for production: hotfix 1.0.1
Incorporate bugfix in develop
Tag 1.0.1
Start of release branch for 1.0
From this point on, "next release" means the release after 1.1.0
Only bugfixes!
Bugfixes from rel. branch may be continuously merged back into develop
Tag 1.1.0
Author: Vincent Driessen
Original blog post: http://nvie.com/
BY-SA

5.8 GitFlow Extension

- All the above can be implemented using Git's inbuilt branching and merging
- To simplify a little, there's an “extension” to Git called “GitFlow”
- The extension is shipped by default with “Git for Windows”
- Available through package managers
  - e.g. 'sudo apt-get install git-flow'

5.9 Initializing GitFlow

- Initialized inside an existing git repository

```
git flow init
```

- Recommended to use the default values for branches

5.10 Features

- New development is organized in the form features
- Typically exist in developers repository only
- Start a new feature

```
git flow feature start FEATURE_NAME
```

- Finish up a feature
    - Merges feature into develop branch
    - Removes the **feature** branch (unless you use -k switch to keep it)
    - Switches back to **develop** branch

```
git flow feature finish FEATURE_NAME
```

- Published feature
    - Useful for pushing a feature to the remote repository so it can be used by other users

```
git flow feature publish FEATURE_NAME
git flow feature pull origin FEATURE_NAME
```

5.11 Release

- Finalized features are merged into a release branch
- Allow for minor bug fixes
- Start a release
  - **Release** branch is created from the **develop** branch

```
git flow release start RELEASE [BASE]
e.g.
git flow release start REL_1.0 develop
```

- Finish up a release
  - Merges the release branch back into 'master'
  - Tags the release with its name
  - Back-merges the release into 'develop' branched
  - Removes the release branch unless you use -k switch to keep it

```
git flow release finish RELEASE -m "tag"
```

- Publish a release
  - For collaboration, a release can be published to a remote repository

```
git flow release publish RELEASE
```

5.12 Hotfixes

- Changes required to fix an undesired state of a live production version
- Hotfixes can be branched off from the corresponding tag on the master branch that marks the production version
- Create a hotfix branch

```
git flow hotfix start VERSION [BASENAME]
```

- e.g.

```
git flow hotfix start 'REL_1.0.1' develop
```

- Finish a hotfix
- Finishing a hotfix gets merged back into **develop** and **master** branches.
- Master merge is tagged with the hotfix version

```
git flow hotfix finish VERSION
```

- e.g.

```
git flow hotfix finish REL_1.0.1
```

5.13 Summary

- GitFlow is a branching model for git
- It is a set of extensions for git.
- The branching model supports features, releases, and hotfixes.

Chapter 6 - Introduction to Gradle

Objectives

In this module we will discuss

- What is Gradle
- Build Scripts and Tasks
- Multi-project Build
- Testing
- Integrating with Eclipse

6.1 What is Gradle

- Gradle is a flexible general purpose build tool like ANT.
- Powerful dependency management
- Full support for your existing Maven or Ivy repository infrastructure
- Ant tasks are also supported
- Groovy language is supported for writing scripts
- It is free and an open source project, and licensed under the Apache Software License (ASL)
- Gradle can increase productivity, from single project builds to huge enterprise multi-project builds.

6.2 Why Groovy

- ANT uses declarative XML based style
- Uses DSL based on Groovy which makes it more powerful.
- Gradle's main focus is on Java projects, but it's still a general purpose build tool
- Groovy offers transparency for Java people.
- Groovy's base syntax is the same as Java's

6.3 Tasks

- Tasks are units of work in Gradle
    - Similar to functions
- Each Task is a sequence of Actions
- Actions in a Task are executed sequentially
- Tasks are *normally* executed sequentially

6.4 Task Dependency

- A task can depend on one or more tasks
- If a dependent task already exists then following syntax can be used

```
task <task_name>(dependsOn: dependentTask)
```

- Tasks with dependencies cannot run until after their dependencies have completed

6.5 Build Script

- build.gradle file
- Written in Groovy (or Kotlin)
- Composed of tasks.
  - Similar to ANT target.

6.6 Sample Build Script

```
task hello1 << {
   println 'Hello World!'
}

task hello2 {
   doFirst {
      print 'This is '
   }
   doLast {
      println 'a test!'
   }
}
```

6.7 Multi-Project Build

- Allows projects to share common configuration
  - Prevents duplication and maintains consistency across projects
  - Common features in root project *settings.gradle*
- Projects are arranged hierarchically
- Example: Parent project that contains multiple web applications

6.8 Plugins

- A plugin is a collection of tasks
  - Reusable
- Two types of plugins:
  - Script – build scripts
  - Binary – classes that implement the plugin interface
- Plugins can be defined in build script to make more tasks available to Gradle
- e.g. apply plugin: 'java'

6.9 Dependency Management

- A project can make use of additional libraries that are not already part of current project.
- Java projects can connect to various repositories, such as Maven Central and JCenter.
- Repositories can be defined like this:

```
repositories {
    mavenCentral()
}
```

- Dependencies can be defined like this:

```
dependencies {
    testCompile "junit:junit:4.12"
}
```

6.10 Gradle Command-Line Arguments

- gradle tasks : displays tasks defined in build script, including plugin provided tasks
- gradle -q : suppresses log messages and runs default tasks
- gradle build: compiles code, generates jar file, and runs unit tests.
- gradle clean: cleans the build directory
- gradle test: runs unit tests

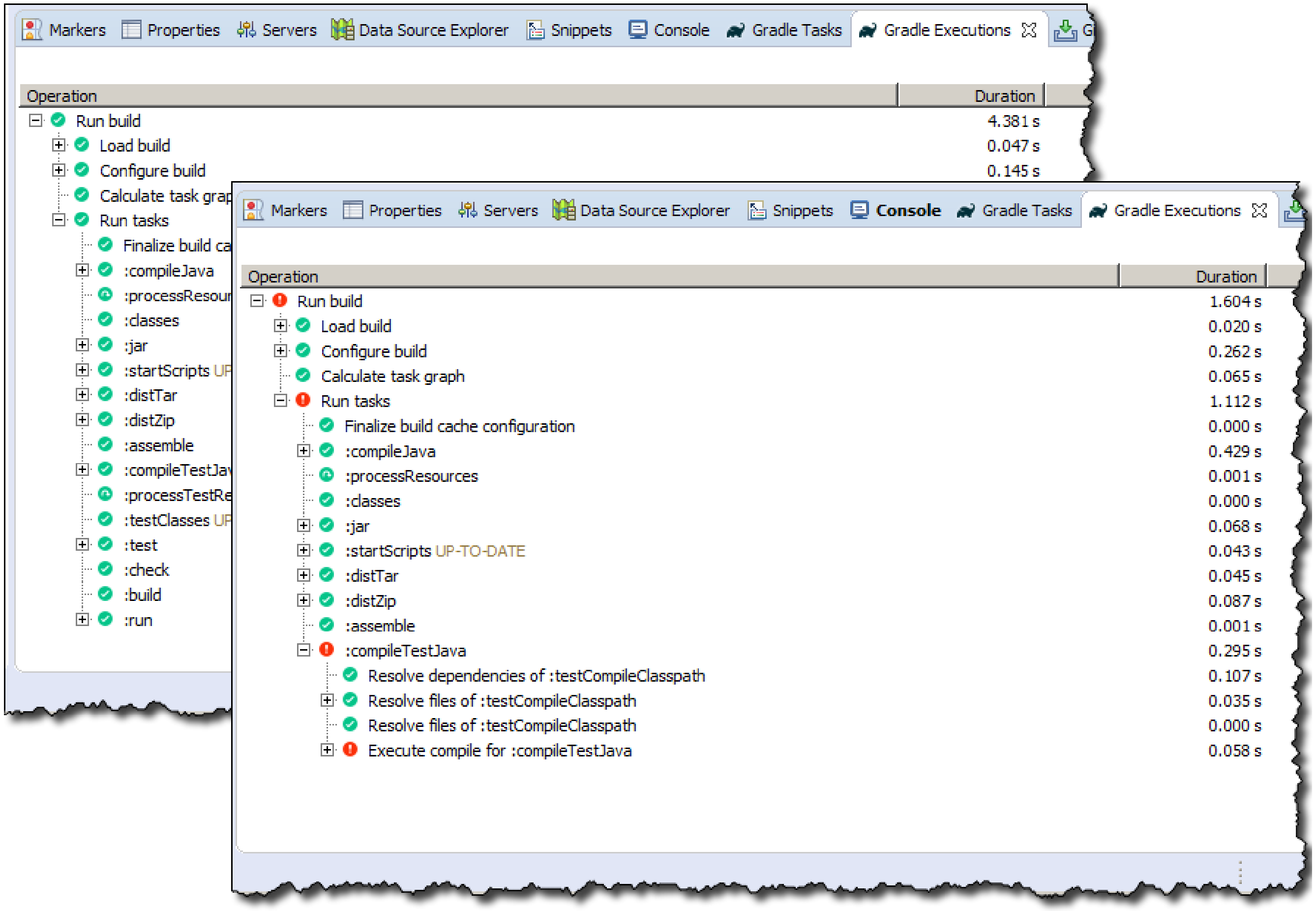
6.11 Testing

- Gradle natively supports JUnit
- Test task executes JUnit test suites
- Test locations
  - Code **src/test/java**
  - Resource **src/test/resources**
- Gradle produces and XML report of text runs

6.12 Eclipse Integration

- Provided through the Eclipse Buildship plugin
  - Now included in base Eclipse distribution
- Create different build configurations to run different tasks
- Run Gradle tests
- Tools to quickly examine the progress of build process

6.13 Successful and Unsuccessful Builds



```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  Gradle Tasks  Gradle Executions  G

Operation                                   Duration
⊟ ✓ Run build                                4.381 s
   ⊞ ✓ Load build                            0.047 s
   ⊞ ✓ Configure build                       0.145 s
     ✓ Calculate task grap
   ⊟ ✓ Run tasks
       ✓ Finalize build ca
     ⊞ ✓ :compileJava
       ✓ :processResour
       ✓ :classes
     ⊞ ✓ :jar
     ⊞ ✓ :startScripts UP
     ⊞ ✓ :distTar
     ⊞ ✓ :distZip
       ✓ :assemble
     ⊞ ✓ :compileTestJa
       ✓ :processTestRe
       ✓ :testClasses UP
     ⊞ ✓ :test
       ✓ :check
       ✓ :build
     ⊞ ✓ :run
```

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  Gradle Tasks  Gradle Executions

Operation                                                   Duration
⊟ ! Run build                                               1.604 s
   ⊞ ✓ Load build                                           0.020 s
   ⊞ ✓ Configure build                                      0.262 s
     ✓ Calculate task graph                                 0.065 s
   ⊟ ! Run tasks                                            1.112 s
       ✓ Finalize build cache configuration                 0.000 s
     ⊞ ✓ :compileJava                                       0.429 s
       ✓ :processResources                                  0.001 s
       ✓ :classes                                           0.000 s
     ⊞ ✓ :jar                                               0.068 s
     ⊞ ✓ :startScripts UP-TO-DATE                           0.043 s
     ⊞ ✓ :distTar                                           0.045 s
     ⊞ ✓ :distZip                                           0.087 s
       ✓ :assemble                                          0.001 s
     ⊟ ! :compileTestJava                                   0.295 s
         ✓ Resolve dependencies of :testCompileClasspath    0.107 s
       ⊞ ✓ Resolve files of :testCompileClasspath           0.035 s
         ✓ Resolve files of :testCompileClasspath           0.000 s
       ⊞ ! Execute compile for :compileTestJava             0.058 s
```

6.14 Summary

- In this module we discussed
  - What is Gradle
  - Build Scripts and Tasks
  - Multi-project Build
  - Testing
  - Integrating with Eclipse

Chapter 7 - Rewriting History

Objectives

In this chapter, we will discuss:

- Altering the Commit History
- Squashing Commits
- Rebasing
- The Reset Command

7.1 Rewriting History

- Rewriting history sounds like it's a bad thing
- Think of it more like editing
    - You wouldn't publish the first draft of your novel, would you?
- It's common to change or alter your commit history before publishing to a remote repository
- In particular, squash commits so you have one set of changes

7.2 Squashing Commits

- DVCS encourages us to "commit early and often"
- Developers will often have a multitude of branches and commits
  - Feature branches
  - Experimental branches
  - Things that didn't work out
  - Tests, regressions, etc
- We tend to commit work-in-progress, not just finished product
  - This doesn't fit with commandments like "Never break the build"

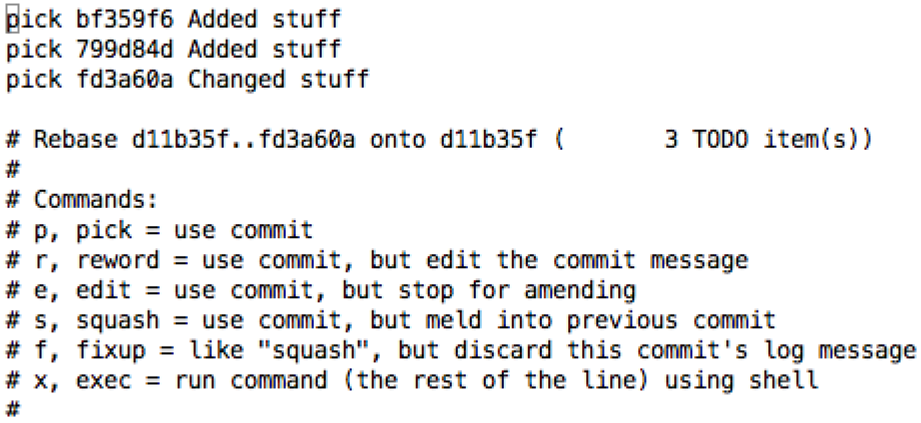
7.3 Squashing Commits (cont'd)

- When sending work for integration, the integration manager doesn't want to see every step
  - she just wants to evaluate the accumulated work
  - Much like in the old days we'd look at a patch
- So, we'd like to be able to take a series of commits and rewrite them into a single commit
  - I.e. "Squash" them
- Then we can push that single commit to a repository for the IM to pull from.

7.4 Squashing Commits (cont'd)

- Figure out what commit we want to start from
  - 'git log'
- Then we use the 'rebase' command
- 'git rebase -i <commit-id>'
  - generates a script file and opens an editor to edit that file
    - Each commit is listed
    - You edit the script to either leave the commit as-is, or squash (add to previous commit)
  - Executes when we exit the editor

7.5 Squashing Commits (cont'd)



```
pick bf359f6 Added stuff
pick 799d84d Added stuff
pick fd3a60a Changed stuff

# Rebase d11b35f..fd3a60a onto d11b35f (       3 TODO item(s))
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's log message
# x, exec = run command (the rest of the line) using shell
#
```

7.6 Rebase vs Merge

- Merge does a three-way merge (two-in, one-out)
- Rebase calculates patch files, then applies successive patches
  - Sometimes a cleaner history
  - Also lets us replay a set of changes onto a branch
  - 'git rebase <other-branch>
    - replays every change from the point the branches diverged
- Many, many possibilities

7.7 Amending Commits

- Extremely common to say "Oops!" after a commit
- e.g. "missed a file", or "should have a better commit message"
- 'git commit --amend' causes whatever was going to be in this commit to be added to the last commit.

7.8 Reset

- Current location of the HEAD can be changed
- 'git reset <commit-id>'
  - '--soft' just changes HEAD
  - '--mixed' updates the index, but not the working copy
  - '--hard' updates the working copy

7.9 Summary

- Although it sounds bad, rewriting history is very common
- Squashing commit is a regular occurrence.