

Laboratorio di Programmazione

Prof. Marco Bertini marco.bertini@unifi.it http://www.micc.unifi.it/bertini/



Code versioning: techniques and tools

Software versions

- All software has multiple versions:
 - Each time you edit a program
 - Versions within a development cycle
 - Test release with debugging code
 - Alpha, beta of final release
 - Variations for different platforms
 - Hardware and software
 - Different releases of a product

Version control

- Version control tracks multiple versions of code.
- In particular, allows:
 - old versions to be recovered
 - multiple versions to exist simultaneously
- Typically multiple users can contribute to software development and version control systems allow them to collaborate:
 - multiple versions of multiple users, merging their contribute
 - tracks who did what



 Version control tracks multiple versions of code In general version control (or revision control, or source control) is about managing multiple versions of documents, programs, web sites.
 It works best on text documents but can manage also binary files such as images. to collaborate:

- multiple versions of multiple users, merging their contribute
- tracks who did what

Why using version control?

- Because it is useful
 - You will want old/multiple versions
 - Without version control, can't recreate project history
 - Allows to go back in history, to solve bugs introduced since the last version of the code
- Because everyone does
 - A basic software development tool.
 Beware of those who do not use it.
 - If you need to share coding responsibilities or maintenance of a codebase with another person, you need version control.

Why using version control?

- When working by yourself:
 - It gives you a "time machine" for going back to earlier versions
 - It gives you great support for different versions of the same project
- When working with others:
 - It greatly simplifies concurrent work, merging changes

Code base

- A Code Base does not just mean code! It also includes:
 - Documentation
 - Build Tools (CMake files, Makefiles, etc.)
 - Configuration files
- All these files may change over time and older versions have to be kept.

Code base

Manage these things using a version control system (VCS)

A version control system is a system which allows for the management of a code base.

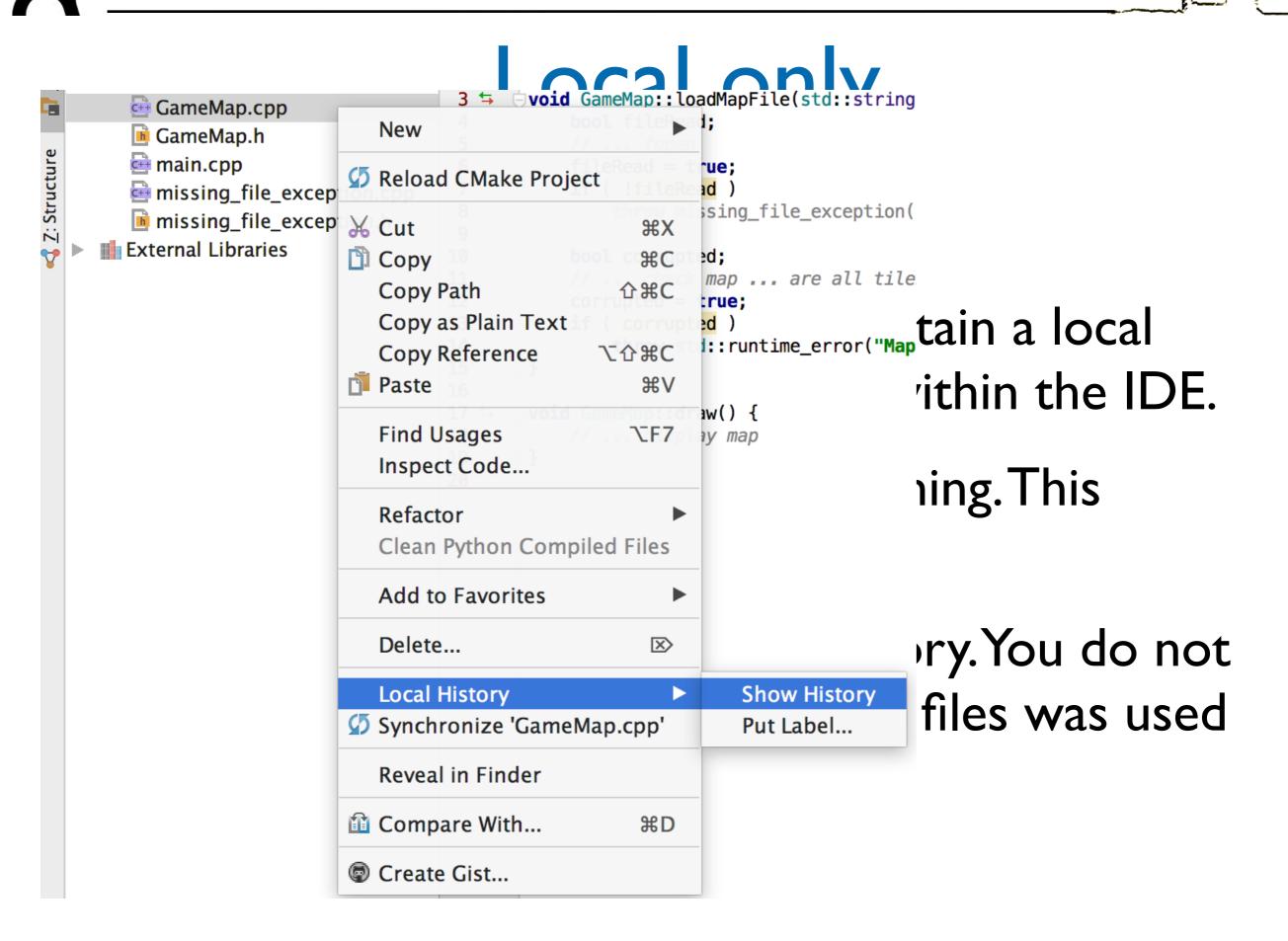
- Documentation
- Build Tools (CMake files, Makefiles, etc.)
- Configuration files
- All these files may change over time and older versions have to be kept.

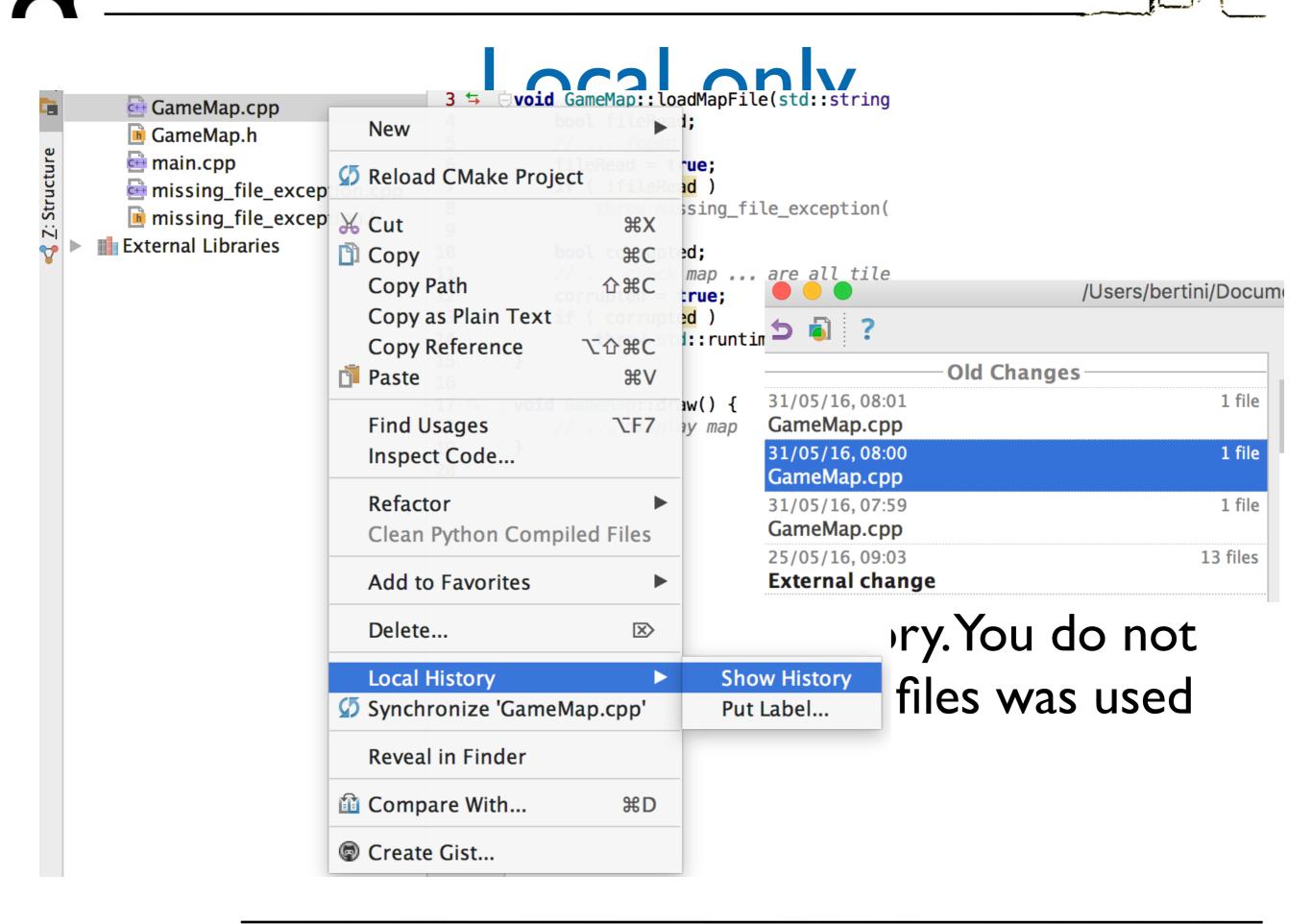
Types of Version Control Systems

- Local only keeps a local database of changes in your local machine filesystem.
- Centralized (Subversion, CVS), require a connection to a central server and "checkout"
- Distributed (Git, Mercurial) allow for local systems to be "mirrors" of the central repo. You don't need to be connected to the central server to get work or commits done.



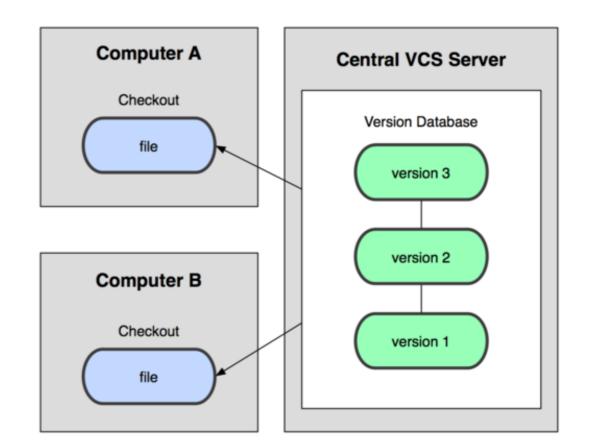
- IDE like CLion and Eclipse maintain a local history of each file developed within the IDE.
- Pros: you don't have to do anything. This versioning is automatic.
- Cons: each file has its own history. You do not know which versions of several files was used at a certain moment.





Centralized

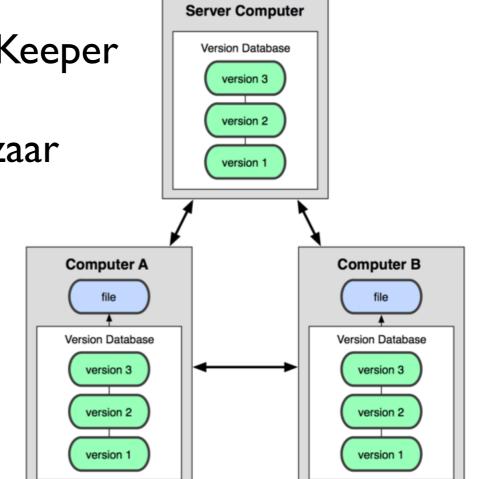
- Traditional version control system
 - Server with database
 - Clients have a working version
- Examples
 - CVS
 - Subversion
 - Visual Source Safe
- Challenges
 - Multi-developer conflicts
 - Client/server communication



Distributed

- Authoritative server by convention only
- Every working checkout is a repository
- Get version control even when detached
- Backups are trivial

- Examples
 - Git
 - Mercurial
 - BitKeeper
 - Bazaar



Overview of the process

- Files are kept in a **repository**
- Repositories can be local or remote to the user
- The user edits a copy called the **working copy**
- Changes are **committed** to the repository when the user is finished making changes
- Other people can then access the repository to get the new code
- Can also be used to manage files when working across multiple computers

Branching

- Branches allows multiple copies of the code base within a single repository.
- Different customers have different requirements
 - Customer A wants features A, B, C
 - Customer B wants features A & C but not B because his computer is old and it slows down too much.
 - Customer C wants only feature A due to costs
- Each customer has their own branch.
- Different versions can easily be maintained

Branching

Common practice:

When releasing a 1.0 version of a system, and start working on version 2.0, you want to keep them separated in the VCS: 1.0 for bug fixing and 2.0 for features development.

- Customer B wants features A & C but not B because his computer is old and it slows down too much.
- Customer C wants only feature A due to costs
- Each customer has their own branch.
- Different versions can easily be maintained

Basic features of a VCS

- Check-in and check-out of items to repository
- Creation of baselines (labels/tags)
 - e.g. "Version 1.0 released!"
- Control and manipulation of branching
 - management of multiple versions
- Overview of version history
 - Allows to see who changed what

Check out / check in

- If you want to make a change the file needs to be checked out from the repository.
- When changes are completed the new code is **checked-in**.
- A **commit** consists of a set of checked in files and the diff between the new and parent versions of each file.
- Each check-in is accompanied by a user name and other meta data.
- Check-ins can be exported from the Version Control System the form of a **patch**.

Revision

- Consider
 - Check out a file
 - Edit it
 - Check the file back in
- This creates a new version of the file
- With each revision, system stores
 - The diffs for that version (typically for efficiency, the VCS doesn't store entire new file, but stores diff with previous version)
 - The new file version number
 - Other metadata
 - Author
 - Time of check in
 - Log file message



- There are occasions when multiple versions of a file need to be collapsed into a single version.
- E.g. a feature from one branch is required in another, or two developers worked on the same file.
- This process is known as a merge.



• There are occasions when multiple versions of

a file need to be collapsed into a single

 I be Edt Wew Window Help I be Edt Window Help I be Edt Wew Window Help I be Edt Window Help I be Edt Window Help I be Edt Window Help I be Cost Artwork Experimental I be window Help I be Cost Artwork Experimental I be window Help I be Cost Artwork Experimental I be window Help I be Cost Artwork I be window Help I be window Help
Image: Second
Z:\Docs\Araxis Docs\Artwork\Experimental 💌 🗳 Z:\Docs\Araxis Docs\Artwork\Experimental 💌 🗳 Z:\Docs\Araxis Docs\Artwork\Experimental 💌 🗳
Z:\Docs\Araxis Docs\Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Z:\Docs\Araxis Docs\Artwork\Experimental 2:\Docs\Araxis Docs\Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Z:\Docs\Araxis Docs\Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Z:\Docs\Araxis Docs\Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Artwork\Experimental 39 (Local Improbability Field 40 41 In order to test this hypot Artwork\Experimental 40 41 In order to test this hypot 41 In order to test this hypot 41 In order to test this hypot
 39 (Local Improbability Field 40 41 In order to test this hypot 39 (Local Improbability Field 40 41 In order to test this hypot 39 (Local Improbability Field 40 41 In order to test this hypot
41 In order to test this hypot 41 In order to test this hypot 41 In order to test this hypot
이야 한 것은
42 the LIF hypothesis were cor 42 the LIF hypothesis were cor 42 the LIF hypothesis were cor
43 substantial enough as to en 43 substantial enough as to en 43 substantial enough as to en 1 on tho
45 We all have an intuitive un 46 do stunningly well'. We kno 3 46 do stunningly well'. We kno 46 do stunningly well'. We kno
47 47 47 47
48 It would seem that all life 48 It would seem that all life 48 It would seem that all life
49 rarely sees plants sufferin 49 rarely sees plants sufferin 49 rarely sees plants sufferin
50 not bright) will often but 50 not desperately bright) # * 50 not bright) will often but *
51 accessible when the squirre 51 accessible when the squirre 51 accessible when the squirre 0
52 Intelligent people are part 52 Intelligent people are part 52 Intelligent people are part
53 LIFE, but by and large they 53 LIFE, but by and large they 53 LIFE, but by and large they
54 are said to be DULL). 54 are said to be DULL). 54 are said to be DULL).
55 55 55
56 56 56 56 56 56 56 56 56 56 56 56 56 5
57 Humanities students often s 57 Humanities students often s 57 an exam raises the likeliho
58 therefore intelligent. Quit 58 therefore intelligent. Quit 58 breakthroughs are likely to
59 against the obvious consequ 59 against the obvious consequ 59 to this). Software engineer
60 therefore greatly raise the 60 therefore greatly raise the 60 disaster such as a hard dis
61 the computer lab burning do 61 the computer lab burning do 61
62 62 62 Humanities students often s
A 63 So far, much anecdotal evi* 63 1. Strong statements affect / 63 therefore intelligent. Quit
A 64 Marmalade Rule [1]. Even so 64 2. The effect of such state 64 against the obvious consequ A 65 65 3. LIFE seems to centre on 65 therefore greatly raise the
3 68 3. LIFE seems to centre on 68 6. Such precautions reduce 3 60 4 4 5 68 1. Strong statements affect
ess F1 for help. Right-click in a file or folder comparison window for a context men: Default ANSI code page 0 removals · 2 insertions · 2 changes · 0 conflicts Ln 63 of 112 🛒

Merging

- I. Start with a file, e.g. v. I.5
- 2. Bob makes changes A to v.1.5
- 3. Alice makes changes B to v.1.5
- 4. Assume Alice checks in first
- 5. Current revision is v.1.6 = apply(B, v.1.5)
- 6. Now Bob checks in
- 7. System notices that Bob checked out v.1.5, but current version is v.1.6
- 8. Bob has not made his changes in the current version!

- 9. The system complains
- 10. Bob is told to update his local copy of the code
- II. Bob does an update
- I 2. This applies Alice's changes B to Bob's code
- I 3. Two possible outcomes of an update:
 - Success
 - Conflicts



• Assume that:

• apply(A, apply(B, v. I.5)) = apply(B, apply(A, v. I.5))

- Then then order of changes didn't matter
- Same result whether Bob or Alice checks in first
- The version control system is happy with this
- Bob can now check in his changes
 - Because apply(B, apply(A, v. I.6)) = apply(B, v. I.6)



- Assume
- $apply(A,apply(B,I.5) \neq apply(B,apply(A,I.6))$
- There is a conflict
 - The order of the changes matters
 - Version control will complain
- Arise when two programmers edit the same piece of code
 - One change overwrites another



- System cannot apply changes when there are conflicts:
 - Final result is not unique
 - Depends on order in which changes are applied
- Version control shows conflicts on update
- Conflicts must be **resolved by hand**

Conflicts

- Conflict detection is based on "nearness" of changes
 - Changes to the same line will conflict
 - Changes to different lines will likely not conflict
- Note: Lack of conflicts does not mean Alice's and Bob's changes work together



Merging conflicts

• Merging is syntactic

- Semantic errors may not create conflicts
 - But the code is still wrong
- You are lucky if the code doesn't compile
 - It is worse if it does ... run unit tests to check nothing is broken!





History

- Came out of Linux development community
- Linus Torvalds, 2005
- Initial goals:
 - Speed
 - Support for non-linear development (thousands of parallel branches)
 - Fully distributed
 - Able to handle large projects like Linux efficiently

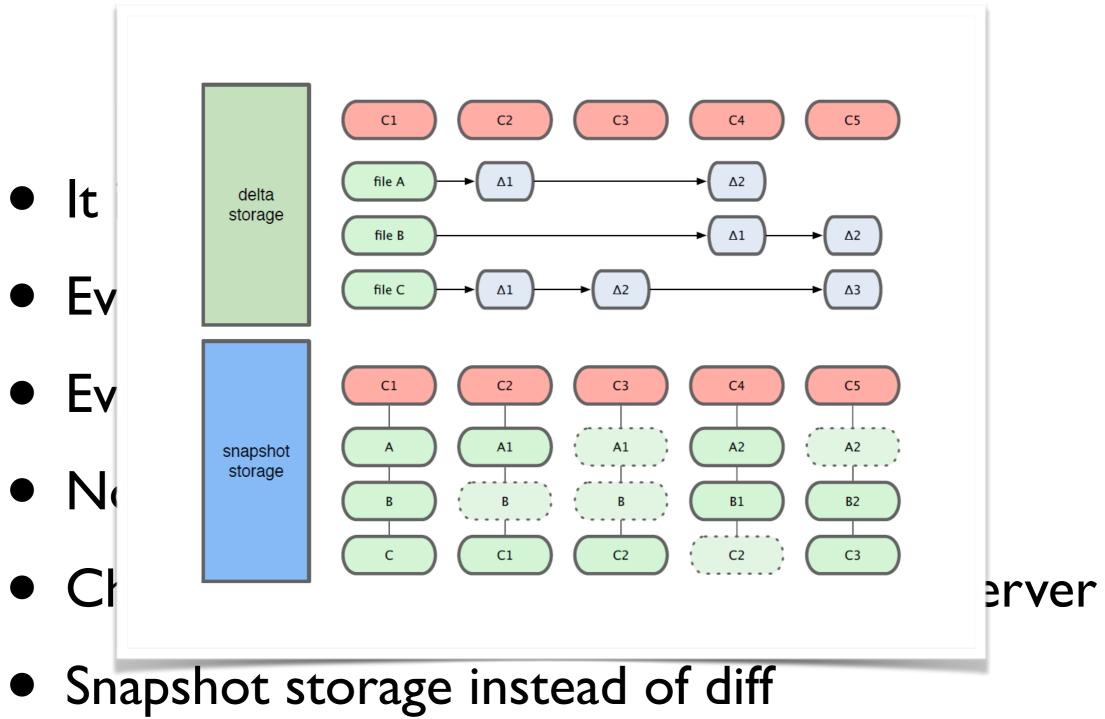
Problem example

- The Linux kernel runs on different processors (ARM, x86, MIPS). These can require significant differences in low level parts of the code base
- Many different modules
- Old versions are required for legacy systems
- Because it is open source, any one can download and suggest changes.

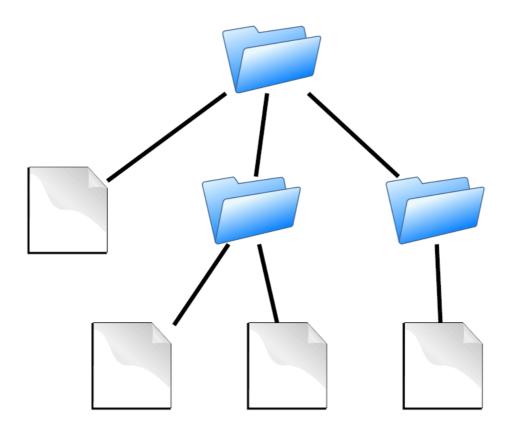
Features

- It is distributed
- Everyone has the complete history
- Everything is done offline
- No central authority
- Changes can be shared even without a server
- Snapshot storage instead of diff

Features







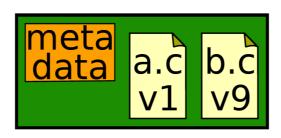
- Contains:
- directories
- files



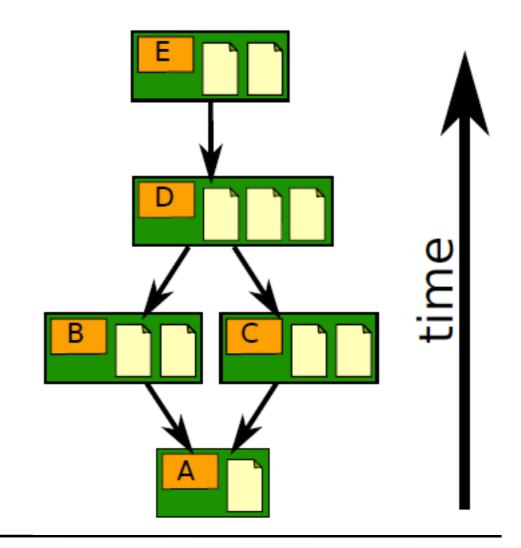


- files
- commits

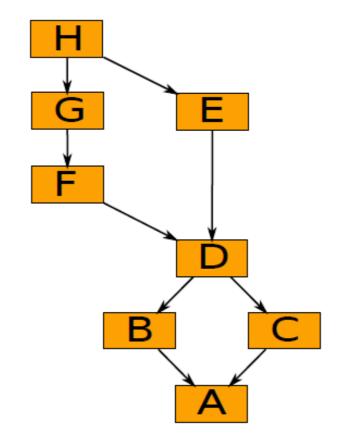




- ancestry relationships
- records history of changes

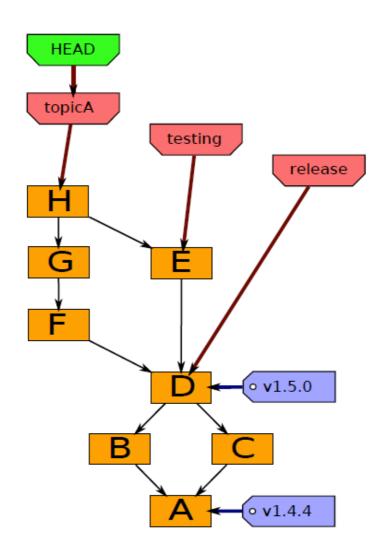


Ancestry relationships



 form a directed acyclic graph (DAG)

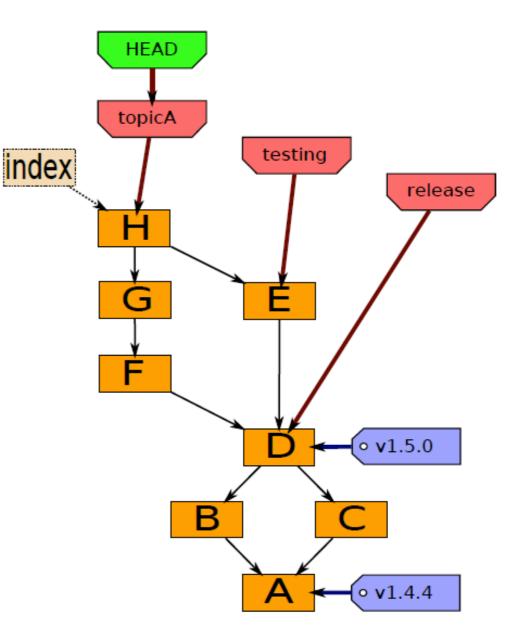
Ancestry graph features



- HEAD
 - is current checkout
 - usually points to a branch



Git component



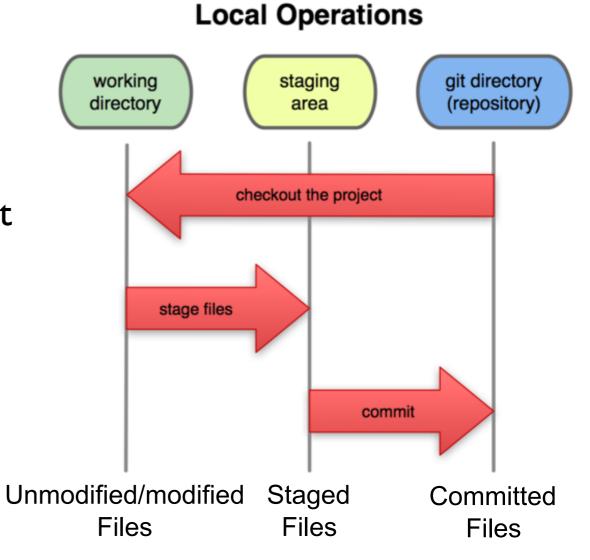
- Index
 - "staging area"
 - what is to be committed

Getting started

Index HEAD



- The HEAD
 - last commit snapshot, next parent
- Index
 - Proposed next commit snapshot
- Working directory
 - Sandbox





- Init a repo(sitory): init to start a new project or clone an existing project
 - will create a ".git" directory. This is your local repo.
- Edit files
- Stage the changes (add files to repo)
- Review your changes
- Commit the changes

You can work as much as you like in your working directory, but the repository isn't updated until you commit something

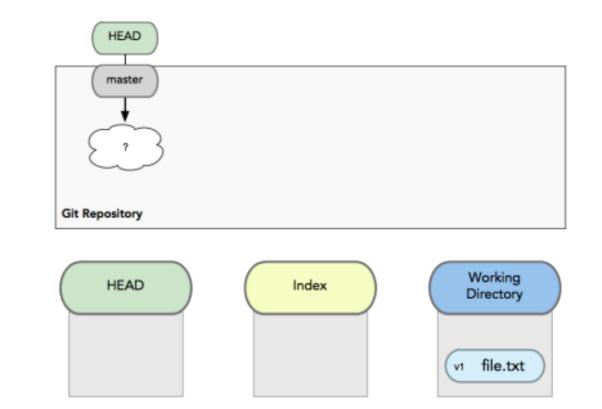
- Init a repo(sitory): init to start a new project or clone an existing project
 - will create a ".git" directory. This is your local repo.
- Edit files
- Stage the changes (add files to repo)
- Review your changes
- Commit the changes

What not to track

- It's important to tell Git what files you do not want to track
- Temp files, executable files, etc. do not need version control (and can cause major issues when merging!)
- We add the filenames to the special file .gitignore. We store this file in the repository

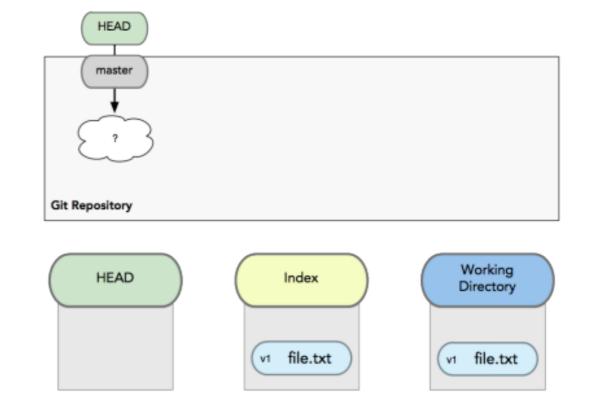
Getting started: edit file

- A basic workflow
 - Edit files
 - Stage the changes
 - Review your changes
 - Commit the changes



Getting started: stage

- A basic workflow
 - Edit files
 - Stage the changes
 - Review your changes
 - Commit the changes

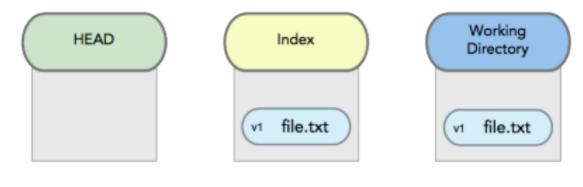


git add

Getting started: review

- A basic workflow
 - Edit files
 - Stage the changes
 - Review your changes





• Commit the changes

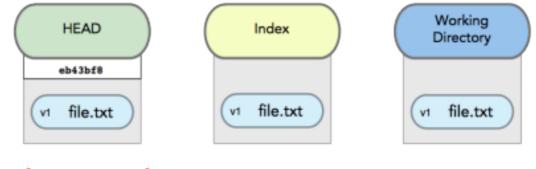
git status

zachary@zachary-desktop:~/code/gitdemo\$ git add hello.txt
zachary@zachary-desktop:~/code/gitdemo\$ git status
On branch master
Changes to be committed:
(use "git reset HEAD <file>..." to unstage)
#
modified: hello.txt
#

Getting started: commit

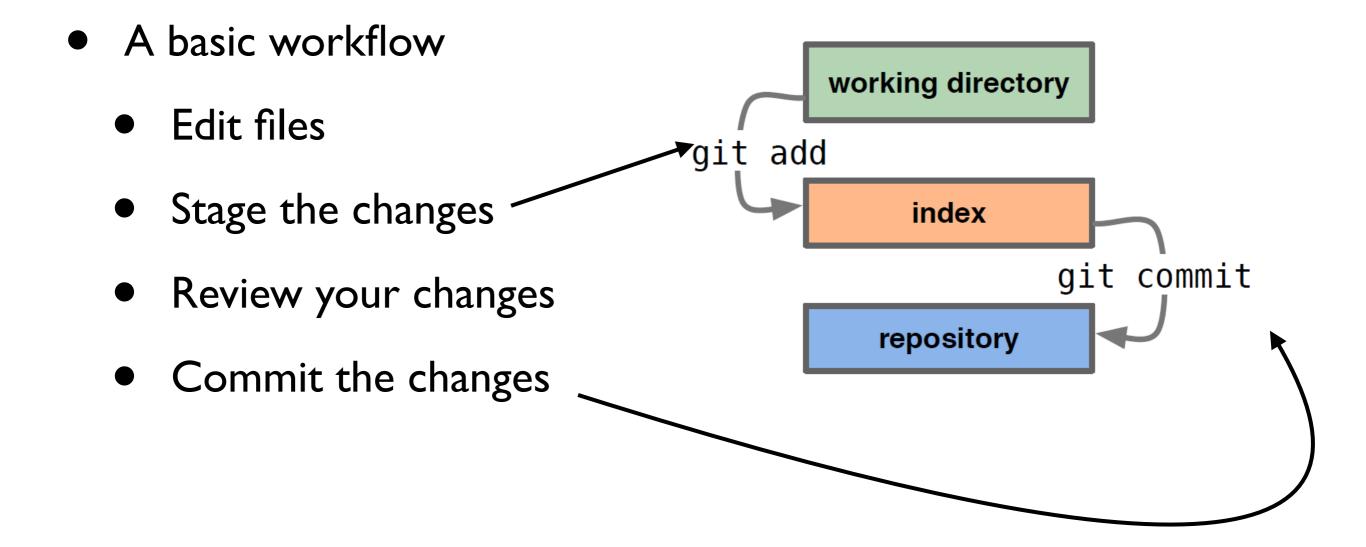
- A basic workflow
 - Edit files
 - Stage the changes
 - Review your changes
 - Commit the changes





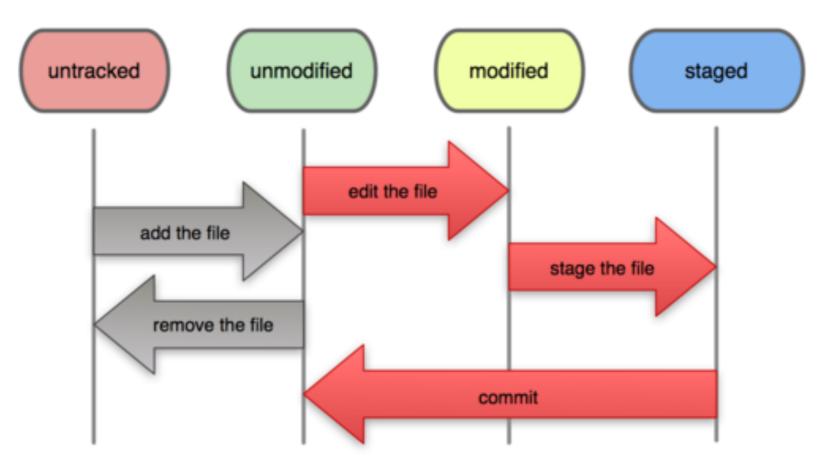
git commit





File life lifecycle

File Status Lifecycle



Files outside Git

Commits and graphs

- A commit is when you tell git that a change (or addition) you have made is ready to be included in the project
- When you commit your change to git, it creates a commit object, that represents the complete state of the project, including all the files in the project
- The very first commit object has no "parents"
- Usually, you take some commit object, make some changes, and create a new commit object; the original commit object is the parent of the new commit object
 - Hence, most commit objects have a single parent
 - You can also merge two commit objects to form a new one, in this case the new commit object has two parents
 - Hence, commit objects form a directed graph
- Git is all about using and manipulating this graph

Commits and graphs

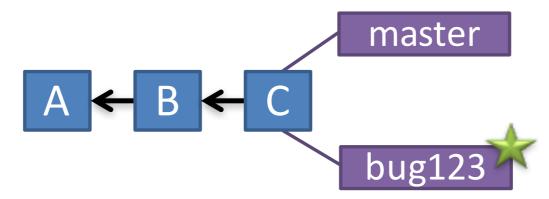
- A head is a reference to a commit object
- The "current head" is called HEAD (all caps)
- Usually, you will take HEAD (the current commit object), make some changes to it, and commit the changes, creating a new current commit object
- This results in a linear graph: $A \rightarrow B \rightarrow C \rightarrow ...$ \rightarrow HEAD

Good practice

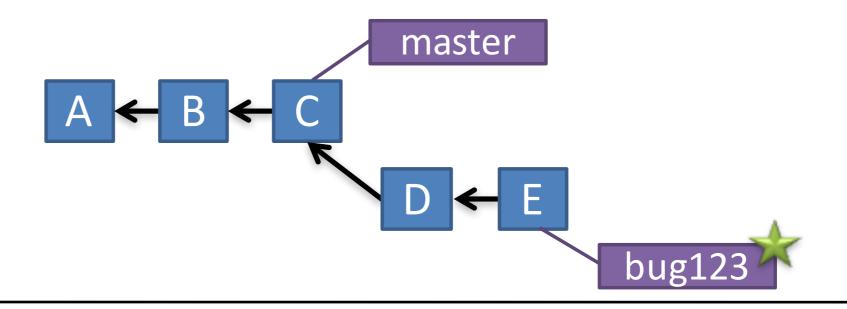
- In git, "Commits are cheap." Do them often.
- When you commit, you must provide a oneline message stating what you have done
 - Terrible message: "Fixed a bunch of things"
 - Better message: "Corrected the calculation of median scores"
- Commit messages can be very helpful, to yourself as well as to your team members

- Branch annotates which commit we are working on
- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch

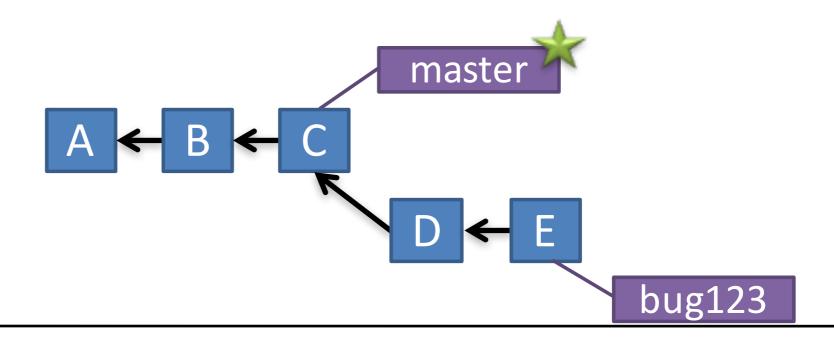
- Branch annotates which commit we are working on
- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch



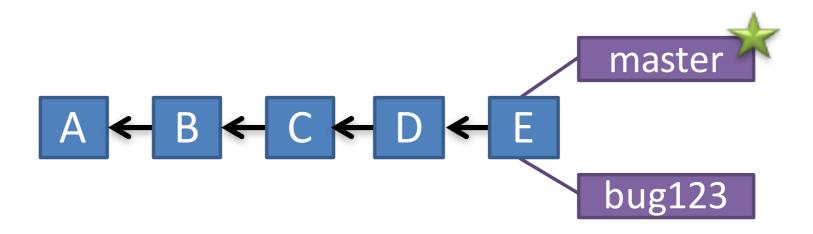
- Branch annotates which commit we are working on
- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch



- Branch annotates which commit we are working on
- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch



- Branch annotates which commit we are working on
- E.g. we can work on development, create a new branch to handle a bug, write code in the branch and then merge to the master branch



Retrieve old commit

- Use checkout to select a committed version of the project or to branches
 - allows to go back in time, e.g. to see when a bug was introduced
 - we can also just evaluate the difference between current ad older versions of code base

Working with remote

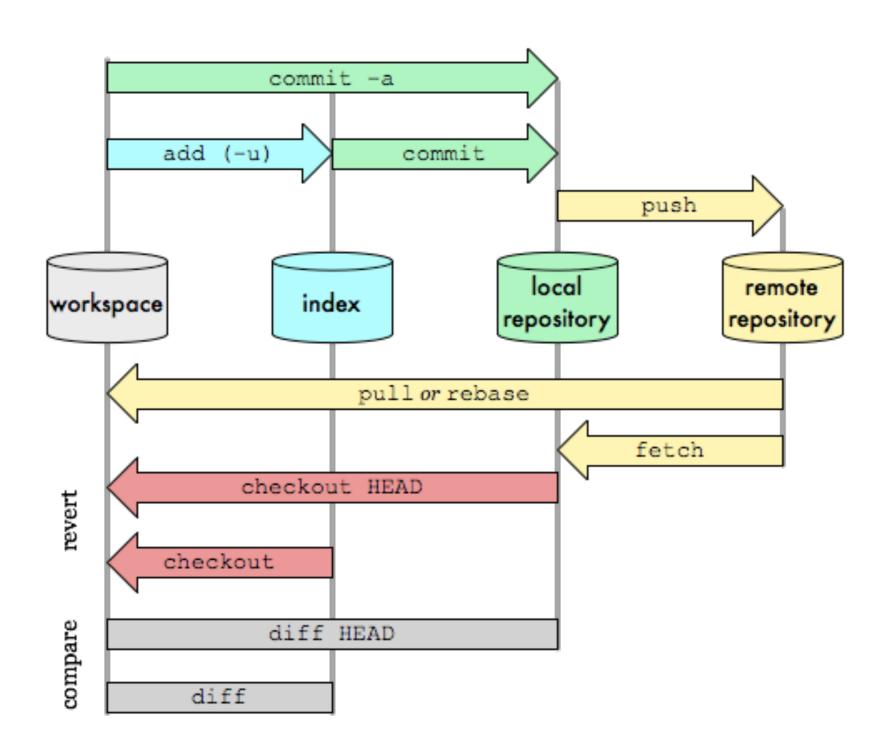
- Add and Commit your changes to your local repo
- **Pull from remote** repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
- **Push** your changes **to** the **remote** repo
- Fetch to retrieve from remote without merging with current code.

Working with remote

- Add and Commit your changes to your local repo
- **Pull from remote** repo to get most recent changes (fix conflicts if necessary, add and commit them to your local repo)
- **Push** your changes **to** the **remote** repo Good practice: Pull then Push

Push will update the remote server. If you are out of date, Git will reject that push.

Git at a glance





Git and CLion

Check git install

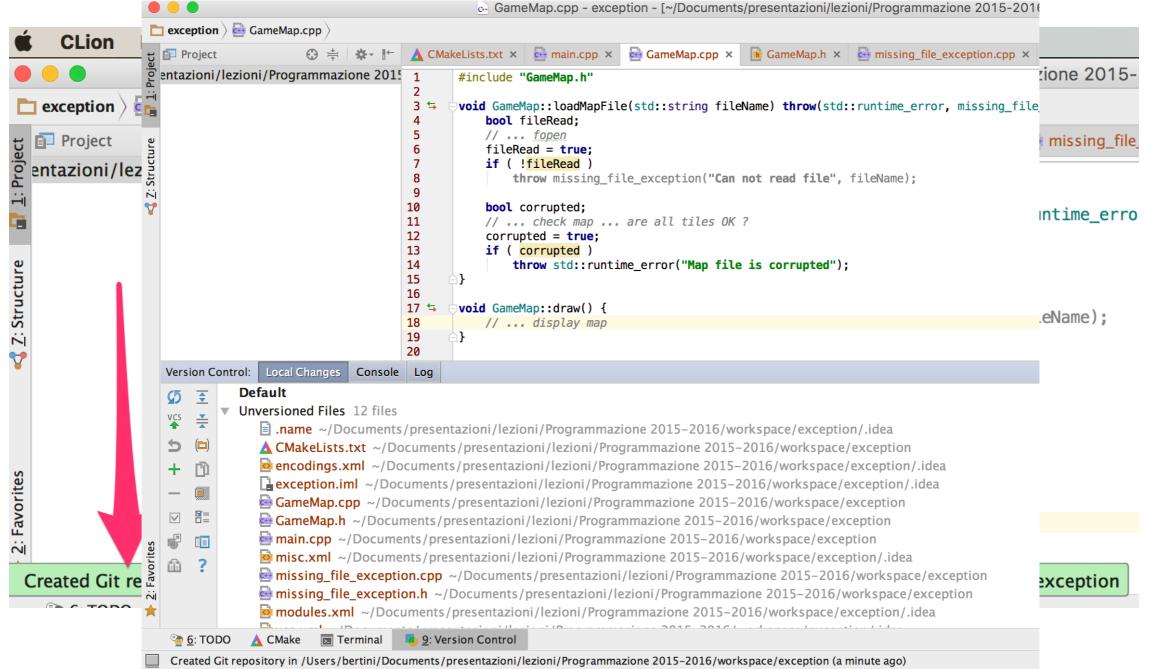
	Preferences
Q Search	Version Control > Git G For current project
Background	Path to Git executable: /usr/bin/git SSH executable: Built-in Commit automatically on cherry-pick Varn if CRLF line separators are about to be committed Warn when committing in detached HEAD or during rebase Update method: Branch default Auto-update if gush of the current branch was rejected
Changelist Conflicts GitHub CVS Git Mercurial Perforce Subversion	Allow force push Protected branches: master

 OSX command line development tools include git. Linux and Windows require to install it.

	CLion	File	Edit	View	Naviga	ite Co	ode	Refactor	Run	Tools	VCS	Window	Help	
					c	Game	eMap.c	pp - exce	ption - [[,]	~/Docum	Loc	al History		►
ex	ception)									Ena	ble Versio	n Control Integratio	on
P	Project			- I	* - I⊂	🛕 CMa			main.cp		VCS	6 Operatio	ns Popup	^ V
ent	azioni/l	ezioni,	/Progra	mmazio	ne 2015	1 2		lude " Ga i				ly Patch		
						3 ≒ (4		GameMap bool fil	eRead;	pFile(s	Che		n Version Control	•
						5 6		// for fileRead	= true;			ort into Ve wse VCS F	ersion Control Repository	► ►
						7 8 9		if (!fi throw	-	.on("Can n	ot read file", fi	leName);		
						10 11		bool cor		2.50	11 <i>+</i>	iles OK ?		
						12		corrupte			e all l	ILES UN ?		
						13		if (cor						
						14		throw	v std::r	untime_e	error("	Map file	<pre>is corrupted");</pre>	
							≙}							
						16 17 ≒ (void	GameMap	udrow()	r				
						17 → (18	F							
						19 (} [′]	// 111 0.	speay m	ap				
						20								

	Enable Version Control Integration
Select a version c	ontrol system to associate with the project root:
Version control se	ettings can be configured in 'Settings Version Co
?	Canc Git Mercurial
	Perforce
	Subversion
	TFS

🐇 CI	Lion	File	Edit	View	Naviga	ate	Code Refactor Run Tools VCS Window Help
						Ga Ga	GameMap.cpp - exception - [~/Documents/presentazioni/lezioni/Programmazione 2015
🖿 exce	ption $ angle$	📴 Gan	neMap.c	$\left< qq \right>$			
ty o entazi	oject		(⊕ ≑	☆ - 1←	🔥 (CMakeLists.txt × 🖨 main.cpp × 🗗 GameMap.cpp × 🕞 GameMap.h × 🗗 missing_fi
2: Favorites 🏹 <u>7</u> : Structure 📊 <u>1</u> : Proje	ioni/le	zioni/	Progra	mmazio	one 2015	4 5 7 8 9 10 11 12 13 14 15 16	<pre>#include "GameMap.h" void GameMap::loadMapFile(std::string fileName) throw(std::runtime_err bool fileRead; // fopen fileRead = true; if (!fileRead) throw missing_file_exception("Can not read file", fileName); bool corrupted; // check map are all tiles OK ? corrupted = true; if (corrupted) throw std::runtime_error("Map file is corrupted"); void GameMap::draw() {</pre>
2: Fav						18 19	// display map
	Created Git repository in /Users/bertini/Documents/presentazioni/lezioni/Programmazione 2015-2016/workspace/exception						



Add files

CLion File Edit	View Navigate Code Refactor	Run Tools VCS Window Help		🖂 GameMap.cpp - exc
•	GameMa	o.cpp - exception - [~/Documents/presentazioni/lezioni/Pr	exception A CMakeLists.txt	
exception	txt 〉		Image: Second state of the second	
Project) ≑ 🕸 I← 🛕 CMakeLists.txt × 📑	main.cpp × 🖨 GameMap.cpp × 🗟 GameMap.h × 📑 m	e exception ~/Documents/pre	esen 1 #include "GameMap.h" 2
 exception ~/Doct CMakeLists.txt 	uments/presen 1 #include "Game 2		GameMap.cpp	3 4 bool fileRead; 5 // foren
🚭 GameMap.cpp 🝺 GameMap.h		Read;	main.cpp missing_file_exception.cp	8 throw missing_
🔤 main.cpp 📴 missing_file_ex	S Reload CMake Project	- true; :Read)	V External Libraries	9 10 bool corrupted; 11 // check map.
🝺 missing_file_ex	د 🔏 Cut ೫X	<pre>missing_file_exception("Can not read file", fileN;</pre>		12 corrupted = true;
External Libraries		upted;		13if (corrupted)14throw std::rur
	Copy Paths 企業C	<pre>>ck map are all tiles OK ? = true;</pre>		15 👌
	Copy as Plain Text	ipted)		16 17 ≒ ⊖ void GameMap::draw() {
	Copy References 飞企器C	<pre>std::runtime_error("Map file is corrupted");</pre>		18 // display map
	n Paste #V			19
	Increast Code void GameMap:	draw() {	Version Control: Local Changes Con	nsole Log
	Inspect Code	splay map	Ø € Default 6 files	
	Refactor 20		vcs 🗸 Vnversioned Files 6 fil	
/ersion Control: Local Cha	Clean Python Compiled Files			ents/presentazioni/lezioni/Programn
Default	Add to Favorites		+ D exception.iml ~/	/Documents/presentazioni/lezioni/l /Documents/presentazioni/lezioni/Pr gram
· · · · · · · · · · · · · · · · · · ·		rogrammazione 2015-2016/workspace/exception/.ide lezioni/Programmazione 2015-2016/workspace/excep	— 📕 🔂 modules.xr 🏠 C	Commit Changes Pr Revert て第乙 ran
🕂 🛐 🔂 📴 encod	i os Local History	lezioni/Programmazione 2015-2016/workspace/excep	ລ 💣 💷 🗳 M	Move to Another Changelist
excep		Check In maxime 2015-2016 /workspace /except		Show Diff #D
🔄 🔤 Gamel	🗹 💋 Synchronize selected files	+ Add עצע אלע אלע א		ump to Source
🗹 🗄 📴 Gamel		Annotate		Delete
		Show Current Revision	e: TODO 🔥 CMake 🔲 T	Add to VCS て第A
i ?	Create Gist	Compare with the Same Repository Version	Schedule selected files to be adde	gnore
🔤 missir	ng_file_exception.h ~/Documents/pres	Compare with Latest Repository Version	The ports tree has	Create Patch Shelve Changes
	Terminal <u>9</u> : Version Control	Compare with Branch	[Ripristinato: : 🍊 R	Refresh
Created Git repository in /U	lsers/bertini/Documents/presentazioni/lezio	Show History	Last login: Inu Ji Restored session:	6 Giu 2016 14:06:20 CEST
				ocal History

Stage files adding them to git versioning. Use project view or Version Control tab that shows also invisible files like those of the CLion project.

Commit

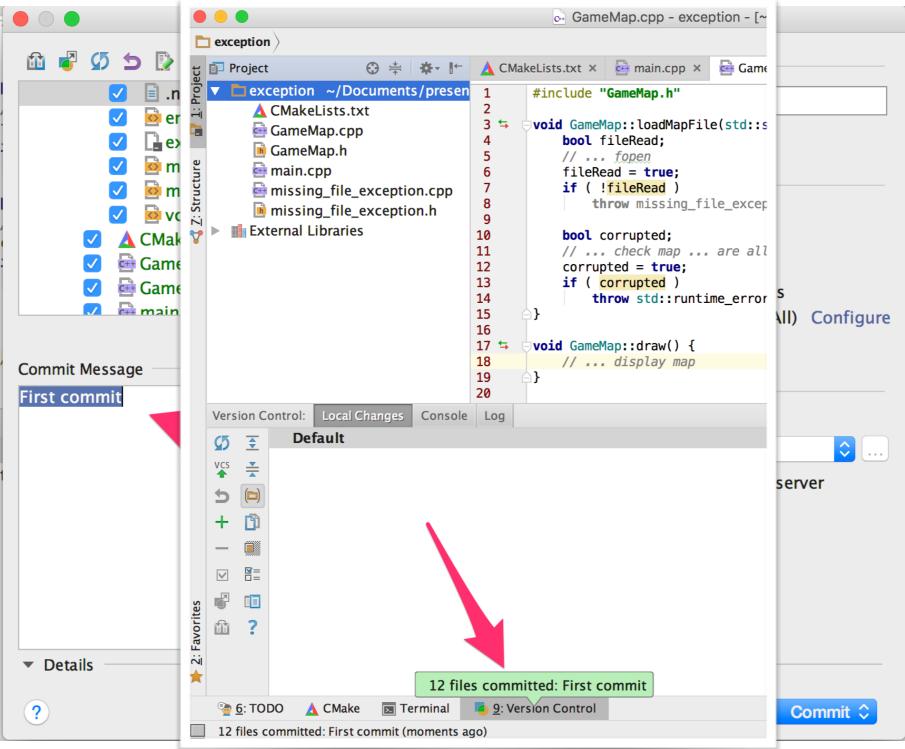
Ś	CLion File Edit	View	Naviga	ate Code	Refac	ctor Rur	n Tool	s VCS	Window He
	• •				⊶ Gam	еМар.срр	- excep	tion - [~/D	ocuments/prese
	exception								
sct	🗗 Project 😯 🕄	+ +	* - ⊪	🛕 CMakeLi	sts.txt ×	📴 main.	cpp ×	📴 GameMa	ap.cpp × 📑 G
<u>1</u> : Project	exception ~/Docu			1 #3	ncludo	ComoMon	6 11		
	CMakeLists.txt	Ne	ew		<pre>apFile(std::string fileName)</pre>				
	🚭 GameMap.cpp 🝺 GameMap.h	🖸 Re	Reload CMake Project					(stu::str	ing fitename)
🕇 Z: Structure	📴 main.cpp	🔏 Cı	ut			ЖХ	e ;		
truc	in missing_file_exe		ору			жc) ha fil	e excenti	.on("Can not r
<u>7</u> : S	missing_file_exe	eption	opy Pat	h		<mark></mark> ት <mark>ዘር</mark>	19_110	e_encoper	
8	External Libraries			Plain Text	bool corrupted:				iloc OK 2
			opy Ref			ፕዕ <mark></mark>	e;	are all l	ILES UK ?
		🗂 Pa				жv)		
				14 15 O}			_runtim	e_error("	Map file is c
			nd Usa	-10		℃F7			
			nd in Pa	1.0		企業F	() { nap		
				n Path		企 <mark></mark> 能R	iap		
		In	spect C	ode			_		
	Version Control: Local Cha	Re	efactor			•			
	☑	L CI	lean Pyt	thon Com	oiled File	es			
	VCS /Users	bertin	i/Docu	ments/pres	entazion	i/lezioni/	rogram	imazione	2015-2016/wo
	↑ ▲ ► □.ide ↓ ▲ CMa	^```		avorites		•			
		51	now Ima	age Thuml	onails	企業T	_		
	Gan Gan		ocal His	tory					
	— 🔳 🖬 🖬 🖬			,			C	ommit Di	rectory
				nize 'excep	tion'		+ A		
SS	🗗 📑 🖬	sing_fi	ile_exce	ption.h			_		
rites	AT 2	Re	eveal in	Finder			A	nnotate	

• Commit whole directory or single files

	Comn	nit
	Commit Changes	
Image: Second secon	Change list: Default 🗘 New: 12	Git Author: Author: Amend commit Before Commit Reformat code Rearrange code Optimize imports Perform code analysis Check TODO (Show All) Configure Cleanup After Commit
First commit First commit • Details		Upload files to: (none) () () () () () () () () () (

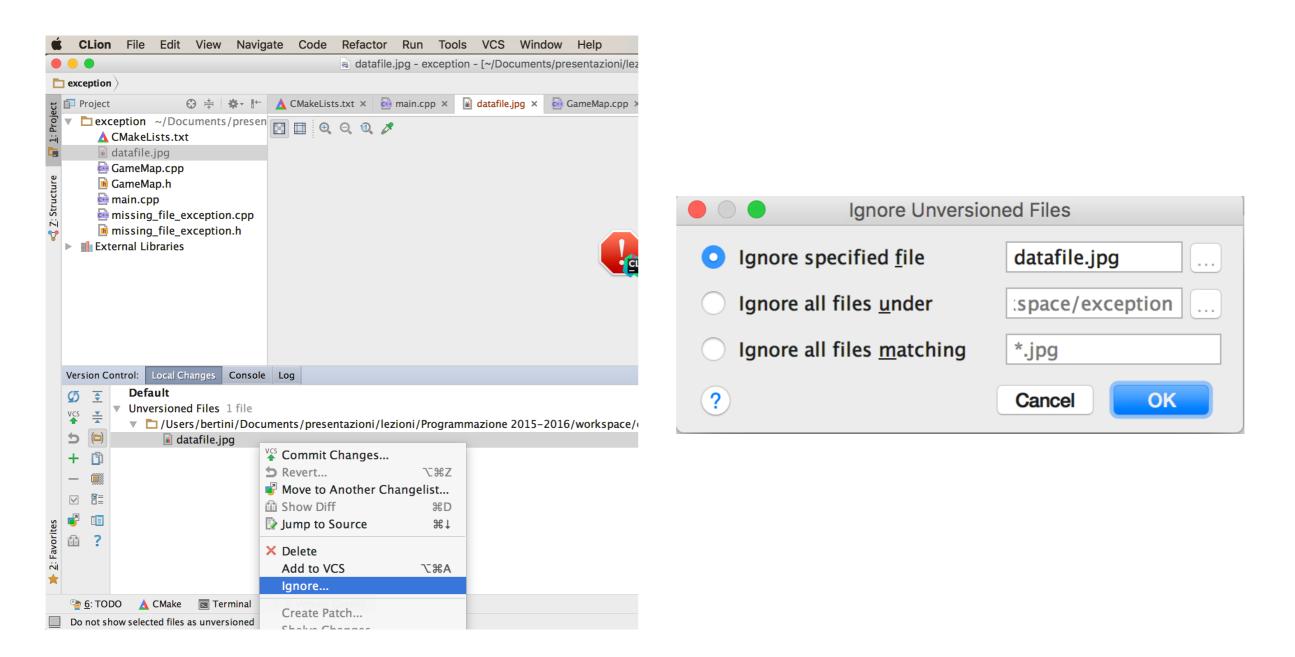
• Commit whole directory or single files

Commit



Commit whole directory or single files

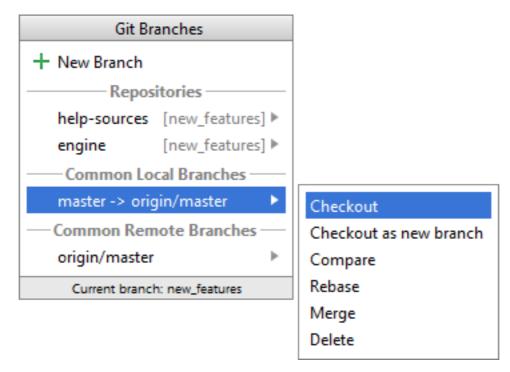
Ignore files



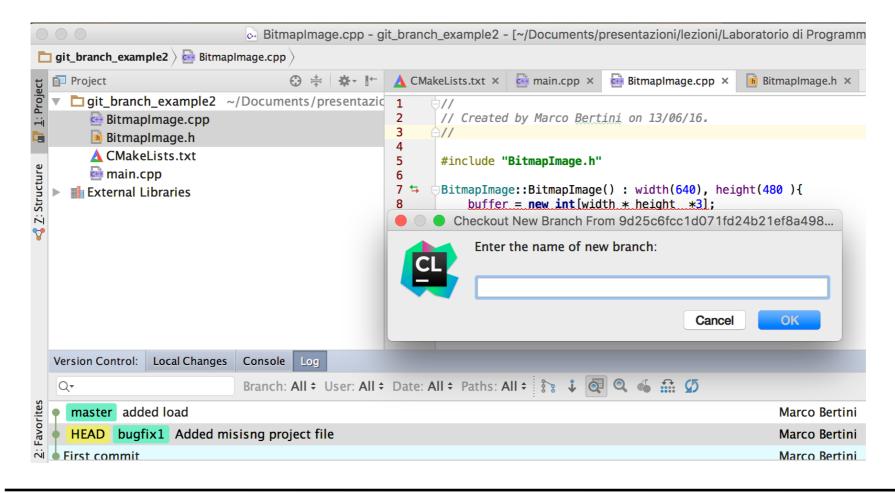
Add to .gitignore with Ignore

Branch

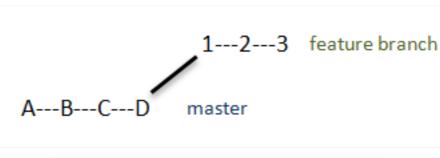
- Use the contextual menu to add new branches, or to checkout them.
- The same menu can be used to merge the current branch with one of the list. The same applies for comparison.



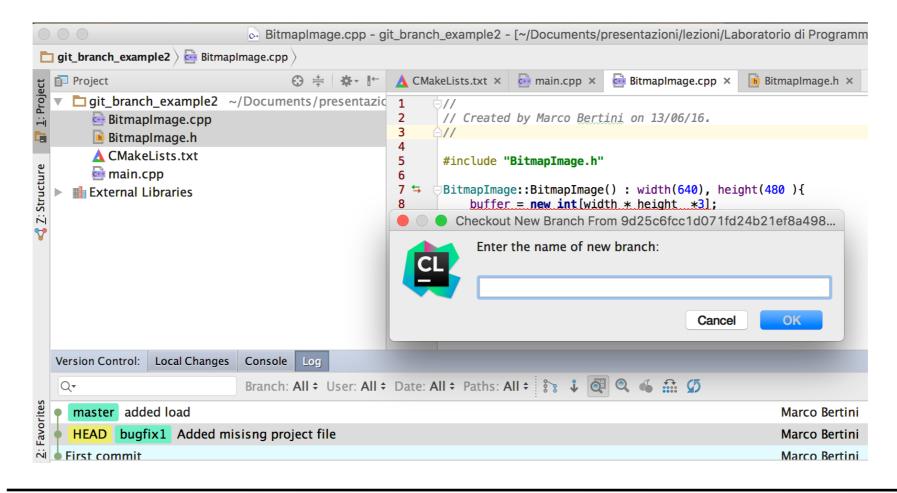
 Create a branch to manage coding related to bug solving on an older commit. Checkout the older commit and start working toward solution.



Branching:



 Create a branch to manage coding related to bug solving on an older commit. Checkout the older commit and start working toward solution.

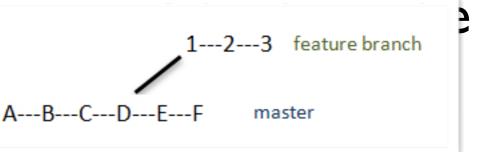


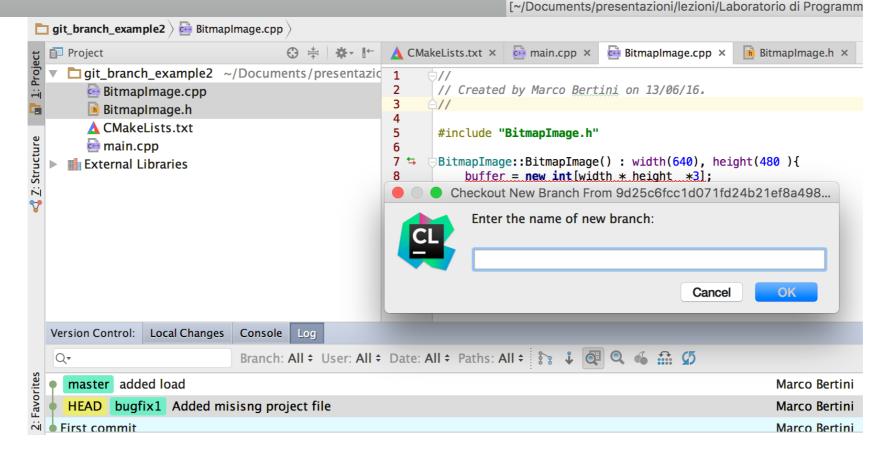




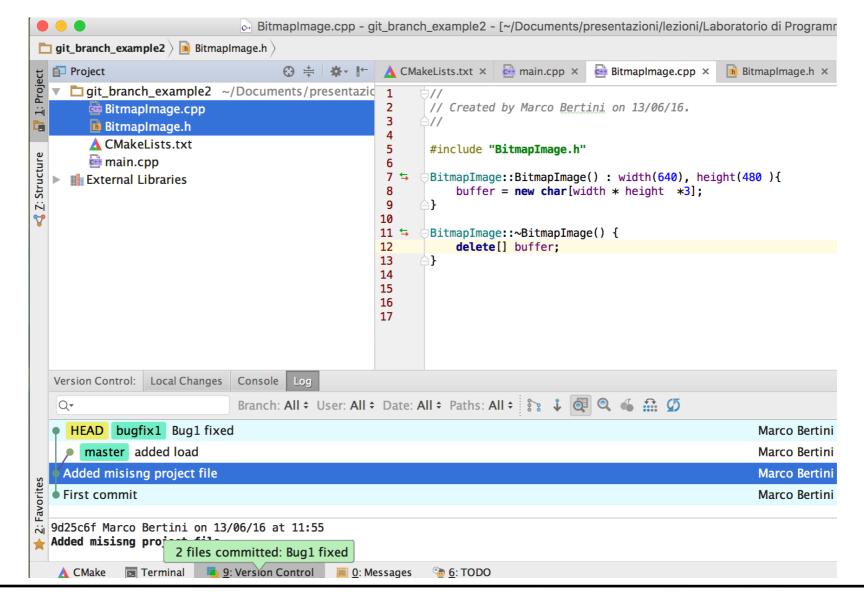
It may happen that while you are working on the branch your colleagues will continue development of master branch:







Version Control: Local Changes Console Log Cleanup		🕒 🕒 BitmapImage.cpp - git_branch_example2 - [~/Documents/presentazioni/lezioni/Laboratorio di Programmazione 2015-2016/workspace/git_branch_example2]				
Bitmapimage.h Bitmapimage.h CMakeLists.txt main.cpp External Libraries Bitmapimage::-Bitmapimage.cpp Bitmapimage::-Bitmapimage.cpp Bitmapimage.cpp Bitmapimage.cpp<	git_branch_example2 BitmapImage.h			Commit Changes		
Version Control: Local Changes Console Log Q- Branch: All + User: All + Date: All + Paths: All + >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>	G	BitmapImage.h CMakeLists.txt main.cpp	<pre>zic 5 #Include "BitmapImage.r 6 7 ≒ BitmapImage::BitmapImag 8</pre>	n 🎲 🇐 🗇 🎲 问 🗄 😤 Change list: Default 🗘 V V Default Documents/presentazioni/lezioni/L V BitmapImage.cpp V BitmapImage.h ag Modified: 2 Commit Message	Author: Amend commit Before Commit Reformat code Rearrange code Optimize imports Perform code analysis Check TODO (Show All) Configure Cleanup	
	🔰 🔲 🔰 <u>2</u> : Favorites	Q+ Branch: All + User: All + Date: All + Paths: All + \$ master added load HEAD bugfix1 Added misisng project file First commit HEAD bugfix1			(none)	



🔴 🕘 💿 Bitma	apImage.cpp - git_branch_example2 - [~/D	Documents/presentazioni/lezioni/Laboratorio di Programmazione
🔁 git_branch_example2 \rangle 🖻 BitmapImage.h \rangle		
Project Image: Comparison of the system	2 // Created by 3 4 5 #include "Bitm 6 7 8 buffer = n 9 10	<pre>Marco Bertini on 13/06/16. mapImage.h" BitmapImage() : width(640), height(480){ new char[width * height *3]; age::load(std::string name) {</pre>
Version Control: Local Changes Console Q- Branch: bugfix1 Bug1 fixed HEAD master added load Added misisng project file First commit bugfix1	Log All ÷ User: All ÷ Date: All ÷ Paths: All ÷ Copy Revision Number Create Patch Cherry-Pick Checkout Revision	Marco Bertini 13/06/ Marco Bertini 13/06/ Marco Bertini 13/06/ Marco Bertini 13/06/
 88a24cf Marco Bertini on 13/06/16 at CMake Terminal 9: Version Co Already up-to-date (moments ago) 	New Tee	Checkout Checkout as New Branch Compare Rebase onto Checkout with Rebase Merge Rename Delete



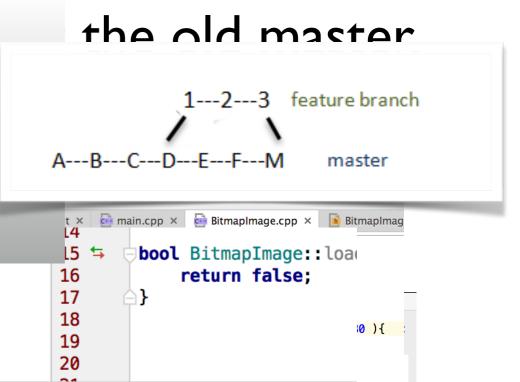
• Commit the solution, checkout the old master, then select the branch and merge with current checkout

mplo2 [.../Documents/presentationi/lationi/l aboratoria di Dr

CHECKOL		BitmapImage.cpp - git_branch_example2 - [~/Documents/presentazion	ii/lezioni/Laboratorio di Pi
	🛅 git_branch_example2 🖉 🖻 Bitm	napImage.h >	
Merge Rev ↑ ↓ Highlight words → 🔡 🎕 Occal Changes (Read-only)	ម្រា Project	Image: Second state Image: Second state	Image::loa
<pre>// Created by Marco Bertini on // #include "BitmapImage.h"</pre>	Version Control: Local Change		
<pre>BitmapImage::BitmapImage() : wi buffer = new char[width * h }</pre>	Q•	Branch: All + User: All + Date: All + Paths: All	
<pre>bool BitmapImage::load(std::str return false; }</pre>	 HEAD Merge branch 'bug Bug1 fixed master added load 	gfix1' into HEAD	
Favorites	Added misisng project file		Marco
	<pre>Birst commit 88a24cf Marco Bertini on 1 Bugl fixed Delet</pre>	.3/06/16 at 12:10 ted branch bugfix1	Marco Marco Marco Marco
	🛕 CMake 🛛 🗖 Terminal 🛛 🗖	9: Version Control 📕 0: Messages 🏻 🤄 6: TODO	
	Deleted branch bugfix1 (momen	ts ago) Jae3c41847761103e566e422e0200d8 // Delete bugfix1 (moments ago)	

When you run merge, the changes from your feature branch are integrated into the HEAD of the target branch:

Merge Rev



🕇 🦊 Highlight words 👻 📳 🎗	18 10){:			
Local Changes (Read-only)	19				
// Created by Marco Bertini on //	20				
<pre>// #include "BitmapImage.h"</pre>	Version Control: Local Changes Console Log				
<pre>BitmapImage::BitmapImage() : wi buffer = new char[width * h }</pre>		-			
<pre>bool BitmapImage::load(std::str return false;</pre>	HEAD Merge branch 'bugfix1' into HEAD				
}	Bug1 fixed				
	master added load				
	Added misisng project file				
		larco I larco I			
-		larco I			
		larco I			
	Deleted branch bugfix1	larco I			
	🛕 CMake 🛛 Terminal 🧧 9: Version Control 📃 0: Messages 🖓 6: TODO				
Deleted branch bugfix1 (moments ago) Merged bugfix1 to 78e3bc891dae3c41847761103e566e422e0200d8 // Delete bugfix1 (moments ago)					

Github

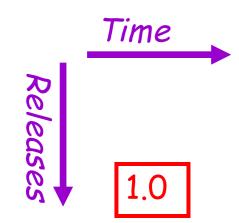
 CLion can use Github as remote server, and allows also to create an account from the options

		Prefere	ences			
Q Search	Version Con	ol > GitHub 🐵 For current project			1	Reset
Appearance & Behavior	Host:	thub.com		Auth Type:	Password	¢
Keymap	Login:					
► Editor	Password:					
Plugins						
 Version Control 	D	not have an account at github.com?	<u>Sign up</u>		Tes	st
Confirmation 🖻						
Background 🖻	Clone gi	repositories using ssh				
Ignored Files 🖷	Connection t	neout: 5.000 İ ms				
Issue Navigation 🖷						
Changelist Conflicts 🛛 🖻						
GitHub 🖻						
CVS @						
Git @						
Mercurial @			N N			
Perforce 🖻			add your account or sign up	for a nou		
Subversion 🖻			add your account or sign up			
TFS @			one			
Build, Execution, Deployment						
Languages & Frameworks						
► Tools						
cppcheck configuration						



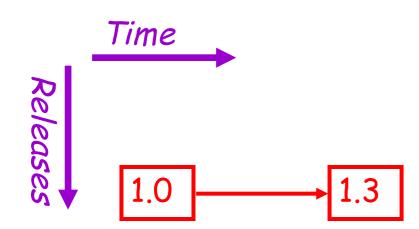
Use scenarios

Scenario I: bug fix



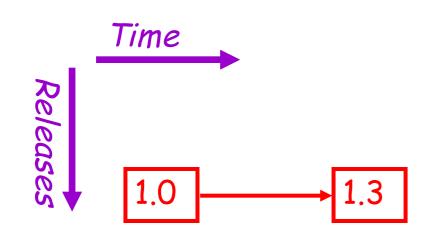
First public release of the hot new product

Scenario I: bug fix



First public release of the hot new product

Scenario I: bug fix



First public release of the hot new product

Internal development continues, progressing to version 1.3

Time

Releases

Scenario I: bug fix

First public release of the hot new product

Internal development continues, progressing to version 1.3



A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

Time

Releases

Note that there are now two lines of development beginning at 1.0. This is branching.

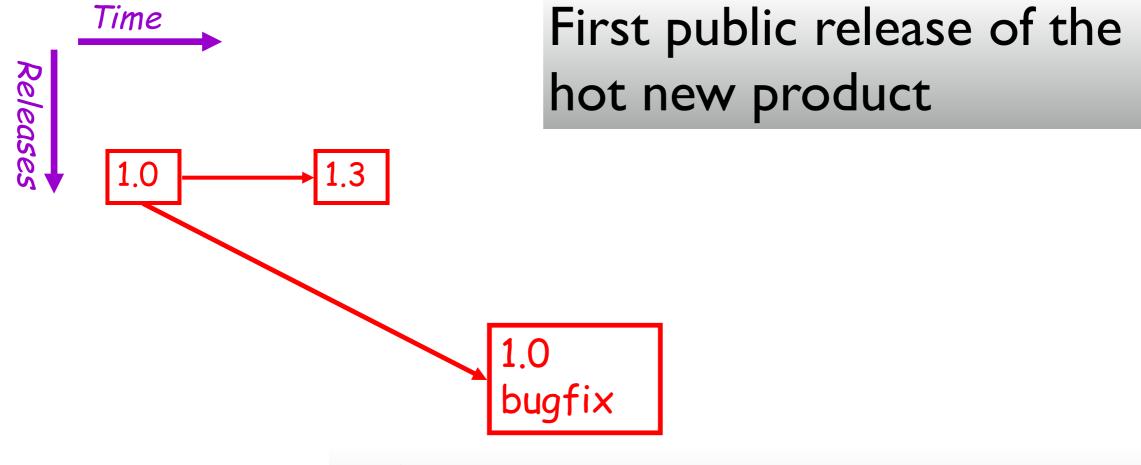
1.0

First public release of the hot new product Internal development continues, progressing to version 1.3

£

A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

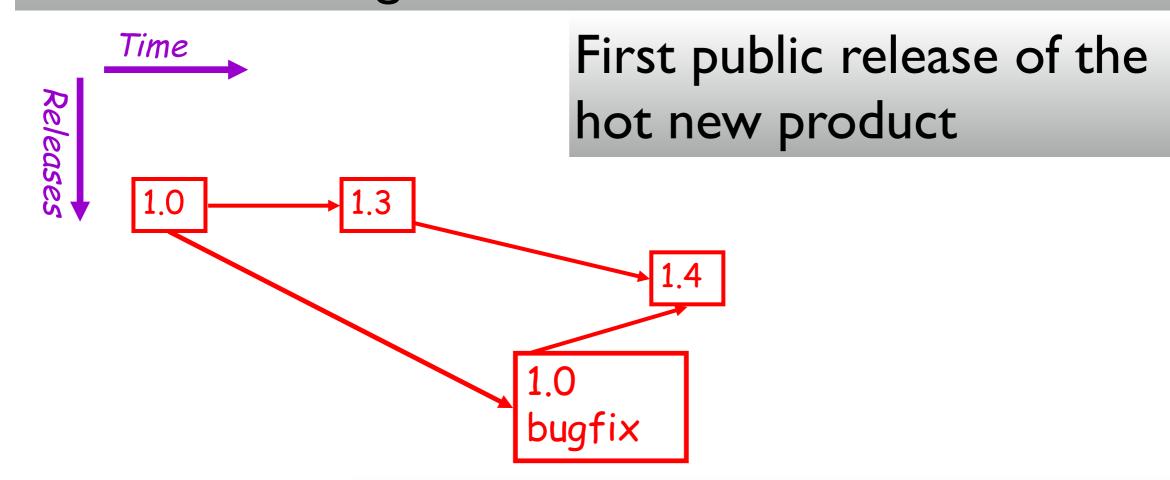
Note that there are now two lines of development beginning at 1.0. This is branching.



A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

a 🦨

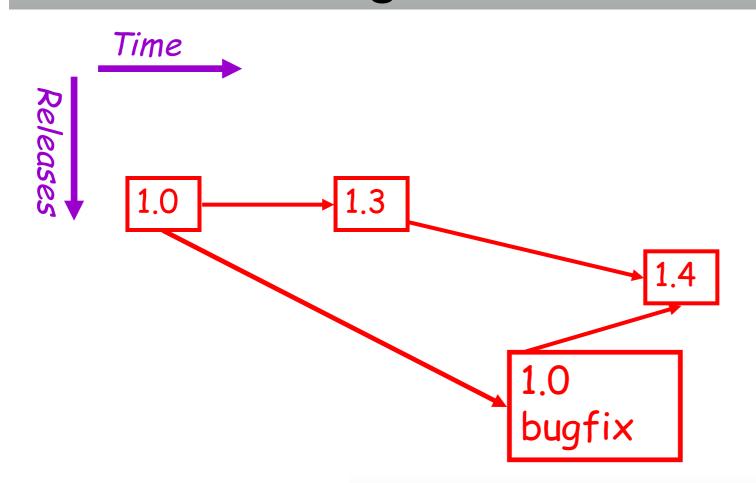
Note that there are now two lines of development beginning at 1.0. This is branching.



A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

a 6

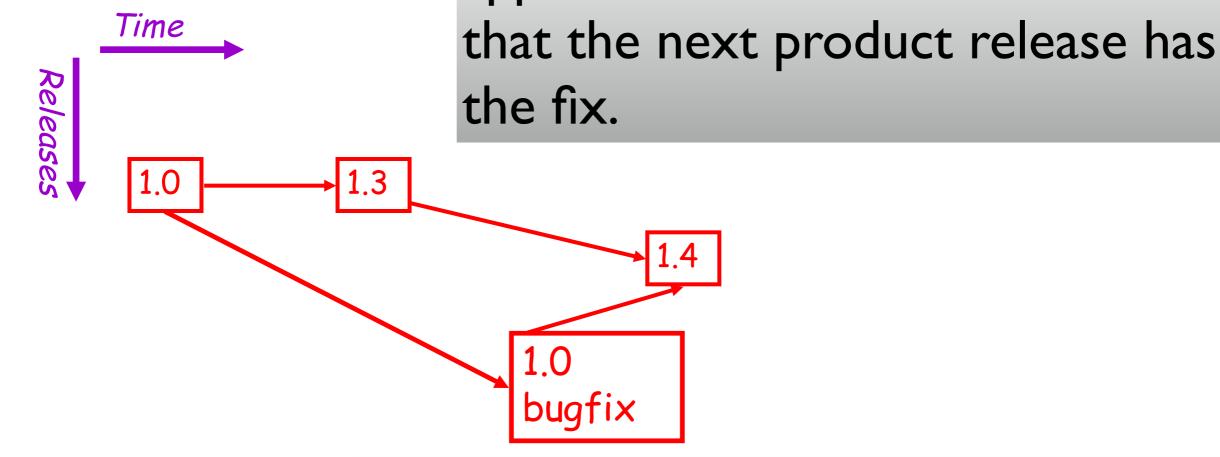
Note that there are now two lines of development beginning at 1.0. This is branching.



A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

n 🏠

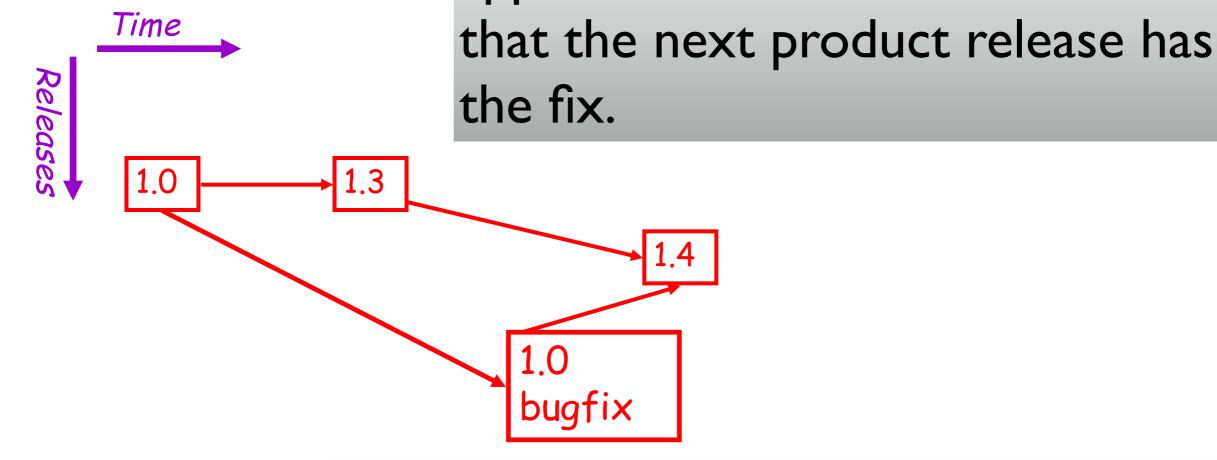
Note that there are now two lines of development beginning at 1.0. This is branching. The bug fix should also be applied to the main code line so



A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release. Solution: Create a version based on 1.0 with the bug fix.

ì Ĝ

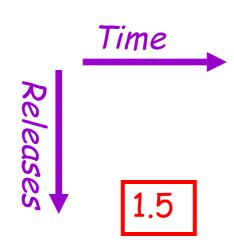
Note that there are now two lines of development beginning at 1.0. This is branching. The bug fix should also be applied to the main code line so



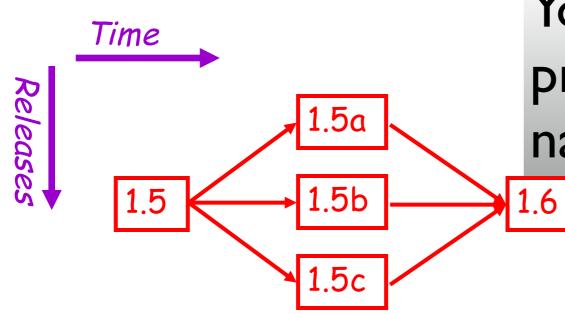
Note that two separate lines of development come back together in 1.4. This is merging.

a 6

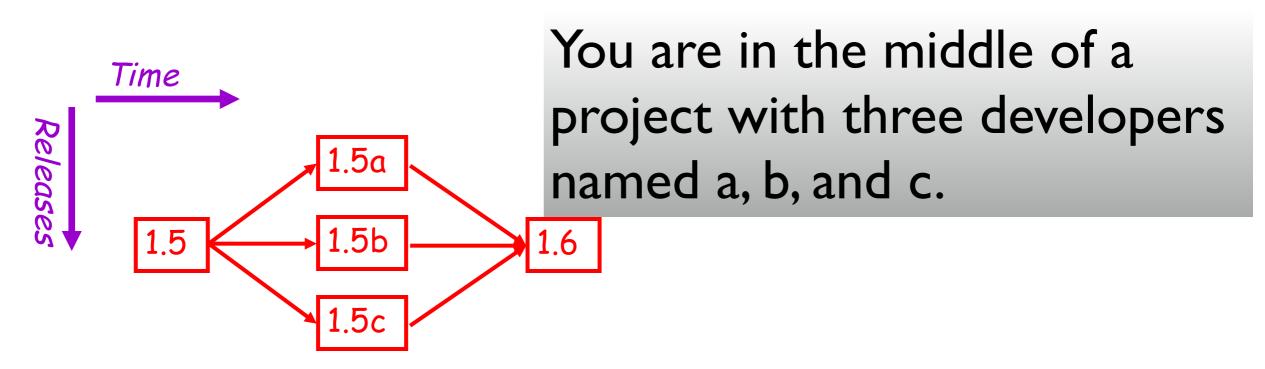
1.6



You are in the middle of a project with three developers named a, b, and c.

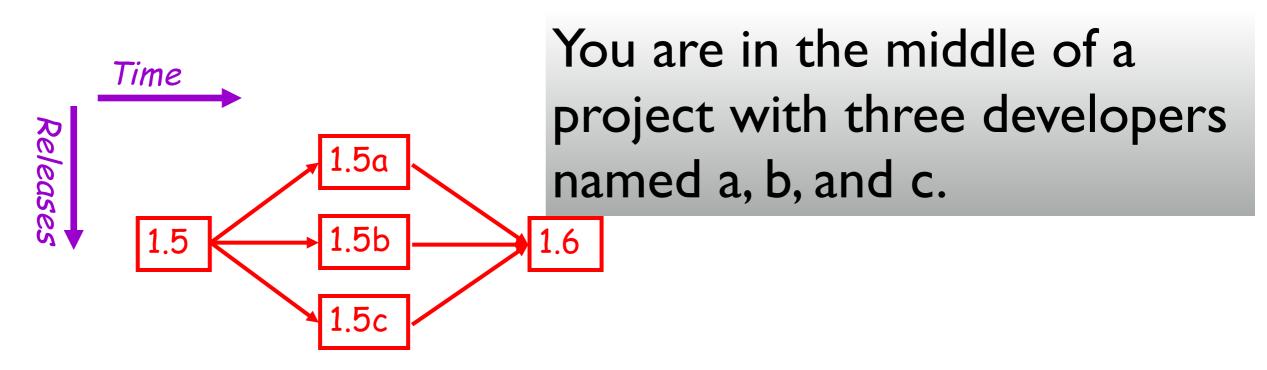


You are in the middle of a project with three developers named a, b, and c.



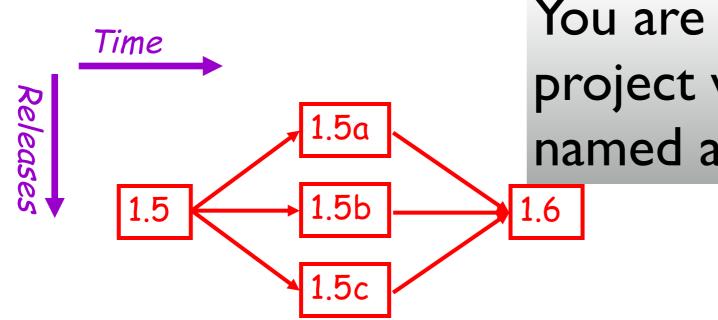
At the beginning of the day everyone checks out a copy of the code.

A check out is a local working copy of a project, outside of the version control system. Logically it is a (special kind of) branch.



The local versions isolate the developers from each other's possibly unstable changes. Each builds on 1.5, the most recent stable version.

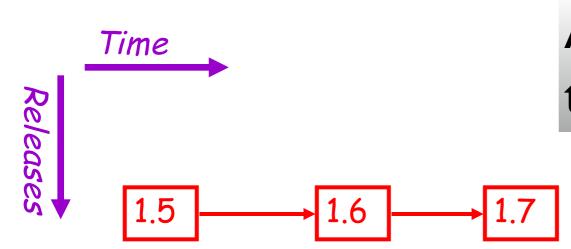
A check out is a local working copy of a project, outside of the version control system. Logically it is a (special kind of) branch.



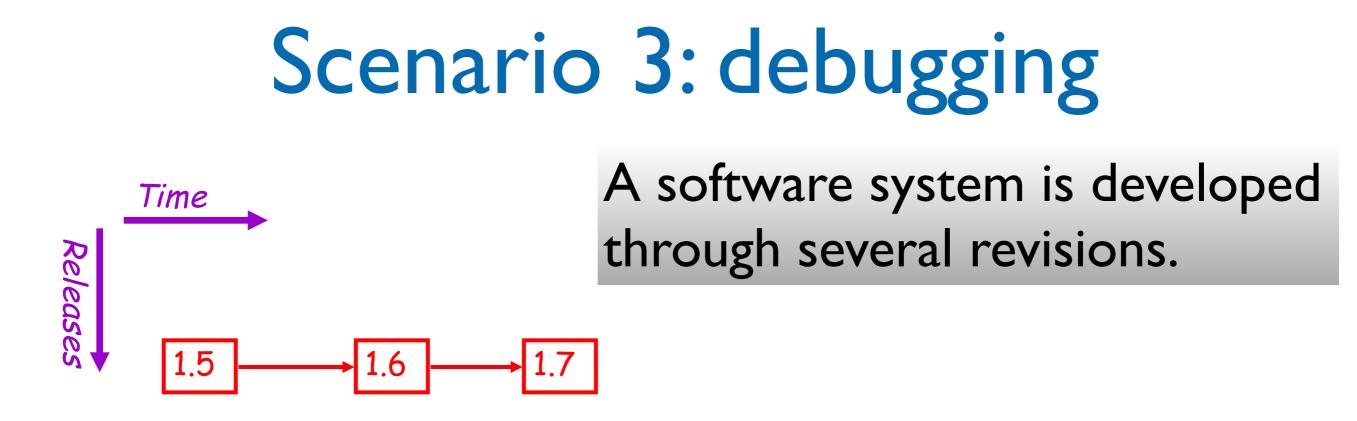
You are in the middle of a project with three developers named a, b, and c.

At the end of the day everyone checks in their tested modifications. A check in is a kind of merge where local versions are copied back into the version control system.

Scenario 3: debugging

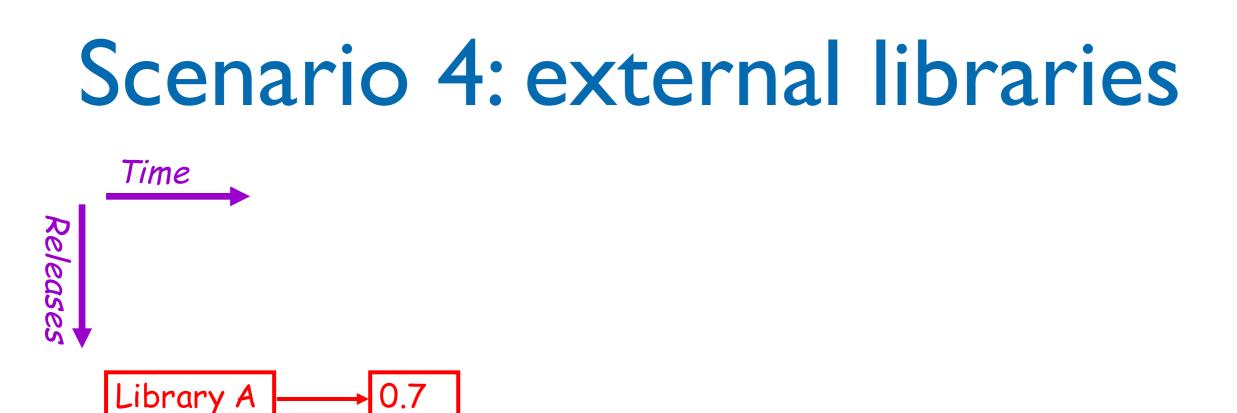


A software system is developed through several revisions.



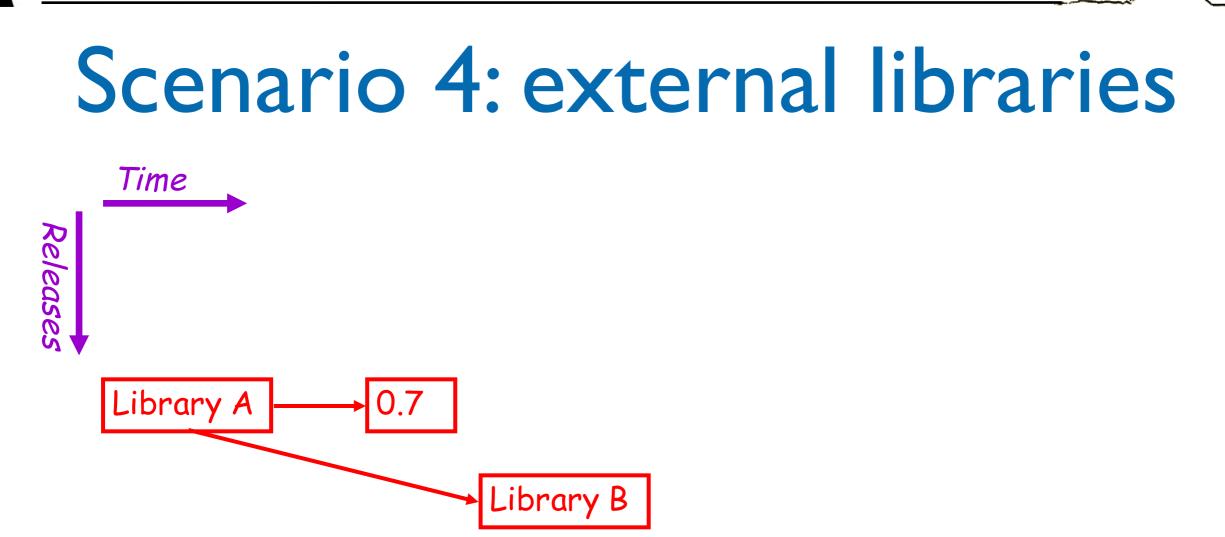
In 1.7 you suddenly discover a bug has crept into the system. When was it introduced?

With version control you can check out old versions of the system and see which revision introduced the bug.



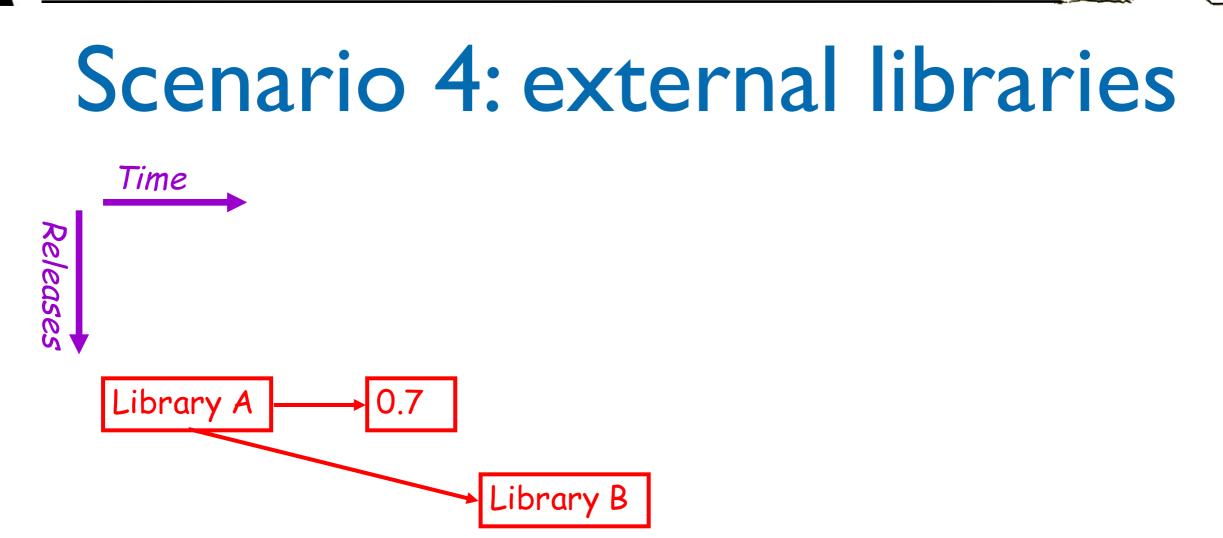
You are building software on top of a thirdparty library, for which you have source.

You begin implementation of your software, including modifications to the library.

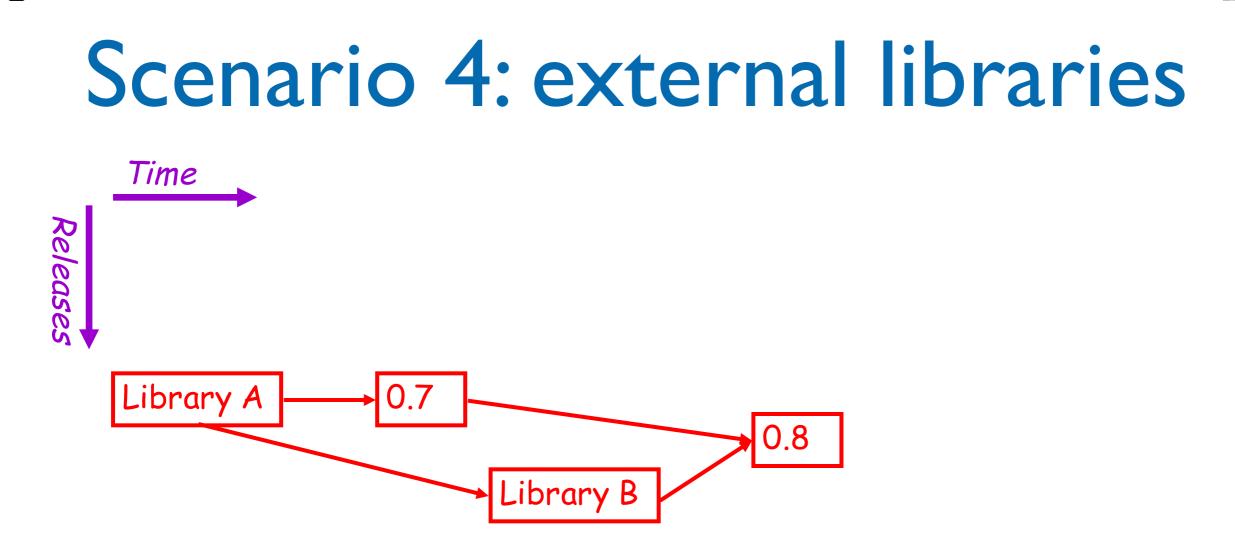


You are building software on top of a thirdparty library, for which you have source.

You begin implementation of your software, including modifications to the library.



A new version of the library is released. Logically this is a branch: library development has proceeded independently of your own development.



You merge the new library into the main code line, thereby applying your modifications to the new library version.



E. Sink, "Version Control by Example" - cap. 2, 4, 8



- These slides are based on the material of:
 - Prof. Aiken,
 - Dr. N. Benatar,
 - Prof. R. Anderson, Univ. Washington
 - P. Chen, Stanford