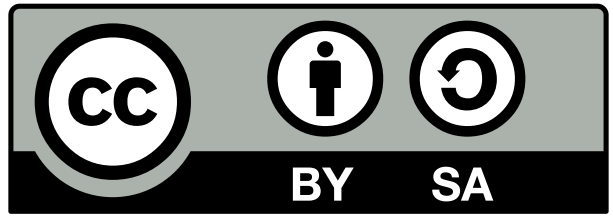


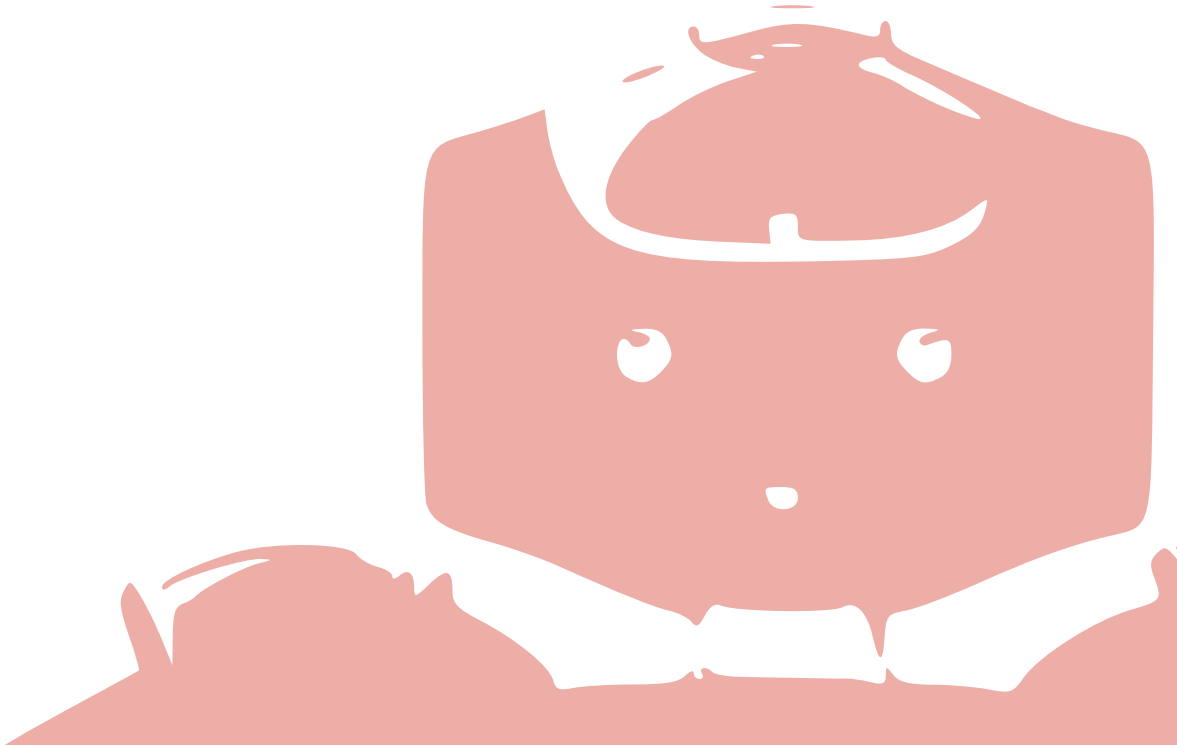
You can download the sources of this presentation here:

github.com/severin-lemaignan/lecture-software-engineering

brl

Software Engineering 101

Getting started with software engineering best practices



Séverin Lemaignan

Bristol Robotics Lab
University of West of England

TODAY'S OBJECTIVES

In 2 hours time, you should know:

- What does "compiling code" ***really*** means
- What **JIT** (Just-In-Time) compilation means
- The difference between a `dll` and a `lib`
- (and what are `dlls`)
- What is **CMake**
- How to organise your code on your hard-drive
- What *Filesystem Standard Hierarchy* means
- What a *ROS package* is
- The differences between the GPL, MIT, BSD, ... licenses
- ...and plenty of things about GIT!

+ intro to ROS (if time permit)

BUILDING CODE

COMPILING CODE IN C++

```cpp
/*
 * Everyone's favourite: "Hello, World!"
 */

#include <iostream>

using namespace std;

int main(void)
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

COMPILING CODE IN C++

```
/*
 * Everyone's favourite: "Hello, World!"
 */

#include <iostream>

using namespace std;

int main(void)
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

```
> g++ hello.cpp -ohello
```

COMPILING CODE IN C++

```cpp
/*
 * Everyone's favourite: "Hello, World!"
 */

#include <iostream>

using namespace std;

int main(void)
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

```
> g++ hello.cpp -ohello
```

```
> ./hello
Hello, World!
```

COMPILING CODE IN C++: THE MAIN STAGES

1. Pre-processing
2. Compilation
3. Assembly
4. Linking

These four steps are transparently performed one after the other by your favourite compiler.

COMPILING CODE IN C++: PRE-PROCESSING

```
/*
 * "Hello, World!": A classic.
 */

#include <iostream>

using namespace std;

int main(void)
{
    cout << "Hello, World!" << endl;
    return 0;
}
```

Pre-processor *directives* start with #

→ `#include <iostream>` is replaced by the content of that file.

COMPILING CODE IN C++: COMPILATION

```
> g++ -S hello.cpp
```

```
main:
.LFB1493:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rsi
        leaq    _ZSt4cout(%rip), %rdi
        call    _ZStlsISt11char_traitsIcEERSt13basic_ost
        movq    %rax, %rdx
        movq    _ZSt4endlIcSt11char_traitsIcEERSt13basic
        movq    %rax, %rsi
        movq    %rdx, %rdi
```

COMPILING CODE IN C++: ASSEMBLY

```
> g++ -s hello.cpp
> hexdump a.out
0000000 457f 464c 0102 0001 0000 0000 0000 0000
0000010 0003 003e 0001 0000 07b0 0000 0000 0000
0000020 0040 0000 0000 0000 1128 0000 0000 0000
0000030 0000 0000 0040 0038 0009 0040 001b 001a
0000040 0006 0000 0005 0000 0040 0000 0000 0000
0000050 0040 0000 0000 0000 0040 0000 0000 0000
0000060 01f8 0000 0000 0000 01f8 0000 0000 0000
0000070 0008 0000 0000 0000 0003 0000 0004 0000
0000080 0238 0000 0000 0000 0238 0000 0000 0000
0000090 0238 0000 0000 0000 001c 0000 0000 0000
00000a0 001c 0000 0000 0000 0001 0000 0000 0000
00000b0 0001 0000 0005 0000 0000 0000 0000 0000
00000c0 0000 0000 0000 0000 0000 0000 0000 0000
00000d0 0b78 0000 0000 0000 0b78 0000 0000 0000
00000e0 0000 0020 0000 0000 0001 0000 0006 0000
```

COMPILING CODE IN C++: LINKING

The linker copies (and re-arrange) the machine code of the static dependencies (***static libraries***) into the executable.

That's what the `-l` flag is used for:

```
> g++ cool_app.cpp -ocool_app -lcv_core -lcv_highgui -lcv_video
```

COMPILED VS NOT COMPILED

Multiple **execution models**:

- Compiled languages (eg C, C++, Ada...)

COMPILED VS NOT COMPILED

Multiple **execution models**:

- Compiled languages (eg C, C++, Ada...)
- Interpreted languages (eg: ...?)

COMPILED VS NOT COMPILED

Multiple **execution models**:

- Compiled languages (eg C, C++, Ada...)
- Interpreted languages (eg: ...?)
- $\rightarrow$ most 'interpreted languages' are actually 'JITed'

Just-In-Time compilation: the interpreter generates an efficient **intermediate representation** (commonly called **bytecode**) that is executed (and often stored).

Very common execution model: Python, C#, Javascript...

MODERN COMPILER INFRASTRUCTURE

For instance, LLVM has a front-end for C/C++ called `clang` and has many backends (like `emscripten` to create the new `wasm` binaries for consumption by the web browsers)

LIBRARIES

A library is a collection of pre-compiled functions that might get called by an executable. *Libraries are not executable* by themselves.

Why libraries?

- to modularise your code
- to make it easier to reuse

LIBRARIES

A library is a collection of pre-compiled functions that might get called by an executable. ***Libraries are not executable*** by themselves.

Why libraries?

- to modularise your code
- to make it easier to reuse

Two main kinds:

- Static libraries, whose code is ***copied*** into the executable by the linker. Extensions: `.a`, `.lib`
- Dynamic libraries, whose code is ***loaded by the operating system*** at runtime. They are also called ***shared libraries***. Extensions: `.so`, `.dll`, `.dylib`

STATIC VS DYNAMIC LIBRARIES

Take 5 min and try to list 2 advantages for the static libraries on one hand, and the dynamic libraries on the other hand.

STATIC VS DYNAMIC LIBRARIES

Advantages of static libraries:

- application can be certain that all its libraries are present
- libraries are the correct version (on Linux, distributions and package managers handle that for dynamic libraries)
- single executable: simpler distribution and installation
- only need to copy (and load into memory) the parts that are needed

STATIC VS DYNAMIC LIBRARIES

Advantages of static libraries:

- application can be certain that all its libraries are present
- libraries are the correct version (on Linux, distributions and package managers handle that for dynamic libraries)
- single executable: simpler distribution and installation
- only need to copy (and load into memory) the parts that are needed

Advantages of dynamic libraries:

- executables smaller because no need to copy the libraries' code
- prevent redundant code in the system
- allows the libraries to be easily updated to fix bugs and security flaws without updating each of the applications

TWO TOOLS TO EXPLORE LIBRARIES

`nm` lists the symbols provided by a shared library:

```
> nm libgazr.so
[...]
000000000002d040 W _ZN4dlib9impl_fhog8init_hogIfNS_33memory_man
000000000027e940 b _ZN4dlibL23OBJECT_PART_NOT_PRESENTE
0000000000010d00 t _ZN9__gnu_cxx12__to_xstringINSt7__cxx1112bas
                 U _ZN9_IplImageC1ERKN2cv3MatE
                 U _Znam@@GLIBCXX_3.4
0000000000011370 T _ZNK18HeadPoseEstimation12drawFeaturesERKSt6
0000000000011b80 T _ZNK18HeadPoseEstimation12intersectionEN2cv6
0000000000011c40 T _ZNK18HeadPoseEstimation4poseEm
0000000000014500 T _ZNK18HeadPoseEstimation5posesEv
0000000000011b30 T _ZNK18HeadPoseEstimation8coordsOfEm14FACIAL_
[...]
```

TWO TOOLS TO EXPLORE LIBRARIES

C++ signatures returned by `nm` are **mangled**. You can demangle them:

```
> nm libgazr.so | c++filt
[...]
000000000027e940 b dlib::OBJECT_PART_NOT_PRESENT
0000000000010d00 t std::__cxx11::basic_string<char, std::char_t
                 U _IplImage::_IplImage(cv::Mat const&)
                 U operator new[](unsigned long)@@GLIBCXX_3.4
0000000000011370 T HeadPoseEstimation::drawFeatures(std::vector
0000000000011b80 T HeadPoseEstimation::intersection(cv::Point_<
0000000000011c40 T HeadPoseEstimation::pose(unsigned long) cons
0000000000014500 T HeadPoseEstimation::poses() const
0000000000011b30 T HeadPoseEstimation::coordsOf(unsigned long,
[...]
```

TWO TOOLS TO EXPLORE LIBRARIES

`ldd` lists the dependencies to shared libraries:

```
> ldd estimate_head_pose
linux-vdso.so.1 (0x00007fff32387000)
libgazr.so (0x00007f3f1822f000)
libopencv_imgcodecs.so.3.2 => /usr/lib/x86_64-linux-gnu/libopen
libopencv_core.so.3.2 => /usr/lib/x86_64-linux-gnu/libopencv_co
libdlib.so.18 => /usr/lib/libdlib.so.18 (0x00007f3f178a8000)
libstdc++.so.6 => /usr/lib/x86_64-linux-gnu/libstdc++.so.6 (0x0
libgcc_s.so.1 => /lib/x86_64-linux-gnu/libgcc_s.so.1 (0x00007f3
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f3f16f1100
libblas.so.3 => /usr/lib/x86_64-linux-gnu/libblas.so.3 (0x00007
liblapack.so.3 => /usr/lib/x86_64-linux-gnu/liblapack.so.3 (0x0
libopencv_calib3d.so.3.2 => /usr/lib/x86_64-linux-gnu/libopencv
libopencv_imgproc.so.3.2 => /usr/lib/x86_64-linux-gnu/libopencv
libm.so.6 => /lib/x86_64-linux-gnu/libm.so.6 (0x00007f3f157c700
[...]
```

HOW TO MAKE & USE LIBRARIES?

Code source of a pathfinding tool for our robots:

```
main.cpp
ui.cpp
pathfinding.cpp
```

```
> g++ main.cpp ui.cpp pathfinding.cpp -opathfinding_ui
```

`pathfinding.cpp` contains the actual pathfinder, and might be useful for many other projects. How to turn it into a library?

HOW TO MAKE & USE LIBRARIES?

First, we need to extract the **API** of our library in a **public header** `pathfinding.hpp`:

```
#ifndef _PATHFINDING_HPP
#define _PATHFINDING_HPP

class Pathfinder {

    Pathfinder(std::shared_ptr<const Map> map);
    Path find(size_t goal_x, size_t goal_y);
}

#endif
```

The header contains the **declarations** of our classes, structures, functions, but not the **definitions** (the definitions are in `pathfinding.cpp`).

HOW TO MAKE & USE LIBRARIES?

Next, compile the library:

```
> g++ -fPIC -shared pathfinding.cpp -olibpathfinding.so
```

HOW TO MAKE & USE LIBRARIES?

Finally, use it:

```
> g++ main.cpp ui.cpp -lpathfinding -opathfinding_ui
```

BUILD SYSTEM

Use and provide a build system!

- Windows-only ⇒ a Visual Studio solution is ok
- MacOS-only ⇒ a XCode project is ok

In all other cases, go for a cross-platform build system like **CMake**.

EXAMPLE OF A CMAKE FILE: CMAKELISTS.TXT

```
cmake_minimum_required(VERSION 2.8.3)
project(cmake_example)

find_package(OpenCV REQUIRED) # one external dependency

# First, the library
add_library(pathfinding pathfinding.cpp)

install(TARGET pathfinding
    LIBRARY DESTINATION lib
)

# then, the executable, which depends on the library's target
include_directories(include)
add_executable(pathfinding_ui main.cpp ui.cpp)
target_link_libraries(pathfinding_ui pathfinding ${OpenCV_LIBRARIES

install(TARGET pathfinding_ui
    RUNTIME DESTINATION bin
)
```

ORGANISING YOUR CODE

principle of least surprise

Make people feel at home when they interact with your project!

REPOSITORY LAYOUT

Try to follow as much as possible the **Filesystem Hierarchy Standard** (FHS). Mainly:

```
src/          # source
include/      # *public* headers
etc/          # configuration files
share/        # data
doc/          # documentation
README
LICENSE
```

NO build artifacts!!

no binaries (except possibly in `share/`)

REPOSITORY LAYOUT

Try to follow as much as possible the **Filesystem Hierarchy Standard** (FHS). Mainly:

```
src/          # source
include/      # *public* headers
etc/          # configuration files
share/        # data
doc/          # documentation
README
LICENSE
```

`README` (or better, use markdown: `README.md`): what is the project about? who is the target audience? how to install? how to get started?

EXAMPLE 1

```
my_proj/
  main.cpp
  ui.cpp
  ui.hpp
  pathfinding.cpp
  pathfinding.hpp
  ui.conf
```

EXAMPLE 1

```
my_proj/
  src/
    main.cpp
    ui.cpp
    ui.hpp
    pathfinding.cpp
  include/
    pathfinding.hpp
  etc/
    ui.conf
  README.md
  LICENSE
  CMakeLists.txt
```

EXAMPLE 1

When compiling the project, create a sub-directory `build` and perform an **out-of-tree** build:

```
> mkdir build && cd build
> cmake ..
> make
```

EXAMPLE 1

```
build/
  ... # lots of compilation artifacts
src/
  main.cpp
  ui.cpp
  ui.hpp
  pathfinding.cpp
include/
  pathfinding.hpp
etc/
  ui.conf
README.md
LICENSE
CMakeLists.txt
```

The `build/` directory can be deleted at any point as it contains only generated files.

README.MD EXAMPLE

```
Better pathfinder
=================

![doc/screenshot.png](Screenshot of the provided UI)

This is a really better pathfinder. Check the
[publication](http://link...).

Pre-requisites
--------------

- dependency 1
- dependency 2

Installation
------------

```
mkdir build && cd build && cmake .. && make install
```
```

EXAMPLE 2: YOU TAKE OVER AN EXISTING PROJECT

```
joe@doe:/usr/robot-planning$ ls -alh
drwxr-xr-x  2 joe  root    4.0K Sep 22 10:40 .
drwxr-xr-x 12 root root    4.0K Sep 22 10:36 ..
-rwxrwxr-x  1 joe  joe     8.8K Sep 22 10:40 pathfinding_ui
-rw-rw-r--  1 joe  joe     2.1K Sep 22 10:39 compile.sh
-rw-rw-r--  1 joe  joe     1.8K Sep 22 10:39 compile.bat
-rw-rw-r--  1 joe  joe      134 Sep 22 10:39 readme-first.txt
-rw-rw-r--  1 joe  joe      895 Sep 21 21:38 main.cpp
-rw-rw-r--  1 joe  joe     1.2K Sep 21 20:27 main.hpp
-rw-rw-r--  1 joe  joe     5.8K Sep 22 10:40 main.o
-rw-rw-r--  1 joe  joe       50 Sep 19 10:36 main.ini
-rw-rw-r--  1 joe  joe     2.3K Sep 20 09:31 pathfinding.cpp
-rw-rw-r--  1 joe  joe      230 Sep 20 10:32 pathfinding.hpp
-rw-rw-r--  1 joe  joe     4.3K Sep 22 10:40 pathfinding.o
-rw-rw-r--  1 joe  joe     7.3K Sep 21 10:40 pathfinding.so
-rw-rw-r--  1 joe  joe     6.7K Sep 20 11:13 core.cpp
-rw-rw-r--  1 joe  joe     7.1K Sep 22 10:40 core.o
-rw-rw-r--  1 joe  joe     6.1K Sep 22 10:40 core.a
-rw-rw-r--  1 joe  joe     6.0K Sep 21 16:22 core.lib
```

EXAMPLE 2: YOU TAKE OVER AN EXISTING PROJECT

Points that can be improved:

- Developping in `usr/` is a bad practice
- Rename files to be more descriptive
- Change the layout to follow the FHS (eg `main.ini` to `etc/main.ini`)
- Perform out-of-tree compilation
- Use a buildsystem (like CMake) instead of relying on ad-hoc scripts
- Add a `README` and a `LICENSE`
- Public headers should be moved to `include/`

ORGANISING YOUR CODE: ROS PACKAGES

When working with ROS, you can use `catkin_create_pkg` to quickly generate a project skeleton + `CMakeLists.txt`:

```
> catkin_create_pkg my_ros_node std_msgs rospy roscpp
> ls -lh my_ros_node
-rw-r--r-- 1 joe joe 6.7K Nov 28 15:46 CMakeLists.txt
drwxr-xr-x 3 joe joe 4.0K Nov 28 15:46 include
-rw-r--r-- 1 joe joe 2.1K Nov 28 15:46 package.xml
drwxr-xr-x 2 joe joe 4.0K Nov 28 15:46 src
```

```
catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

VERSIONING

SEMANTIC VERSIONING

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

- `MAJOR` version when you make incompatible API changes,
- `MINOR` version when you add functionality in a backwards-compatible manner, and
- `PATCH` version when you make backwards-compatible bug fixes.

Additional labels for pre-release and build metadata are available as extensions to the `MAJOR.MINOR.PATCH format`.

Source: `semver` website

SEMANTIC VERSIONING

You are the maintainer of `cool_app`, that depends on OpenCV `2.4.11`.

The OpenCV project releases a new version, what should you do...

- ...if the new version is `2.4.12`?

SEMANTIC VERSIONING

You are the maintainer of `cool_app`, that depends on OpenCV `2.4.11`.

The OpenCV project releases a new version, what should you do...

- ...if the new version is `2.4.12`?
- ...if the new version is `2.5.0`?

SEMANTIC VERSIONING

You are the maintainer of `cool_app`, that depends on OpenCV `2.4.11`.

The OpenCV project releases a new version, what should you do...

- ...if the new version is `2.4.12`?
- ...if the new version is `2.5.0`?
- ...if the new version is `2.9.0`?

SEMANTIC VERSIONING

You are the maintainer of `cool_app`, that depends on OpenCV `2.4.11`.

The OpenCV project releases a new version, what should you do...

- ...if the new version is `2.4.12`?
- ...if the new version is `2.5.0`?
- ...if the new version is `2.9.0`?
- ...if the new version is `3.0.0-beta`?

SEMANTIC VERSIONING

You are the maintainer of `cool_app`, that depends on OpenCV `2.4.11`.

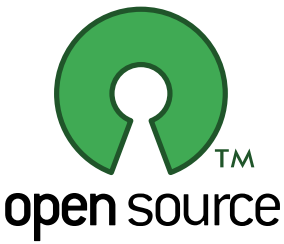
The OpenCV project releases a new version, what should you do...

- ...if the new version is `2.4.12`?
- ...if the new version is `2.5.0`?
- ...if the new version is `2.9.0`?
- ...if the new version is `3.0.0-beta`?
- ...if the new version is `3.0.0`?

SOFTWARE LICENSES, OPEN-SOURCE, FREE SOFTWARE

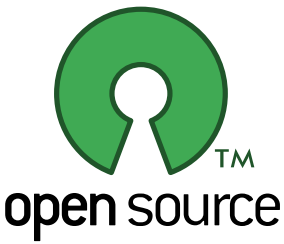
SOFTWARE LICENSES

- **no license** ⇒ default copyright laws apply. You retain all rights to your source code; nobody else may reproduce, distribute, or create derivative works from your work.



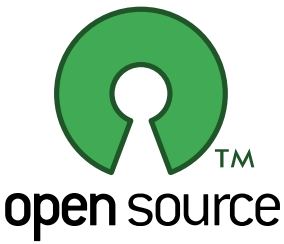
SOFTWARE LICENSES

- **no license** ⇒ default copyright laws apply. You retain all rights to your source code; nobody else may reproduce, distribute, or create derivative works from your work.
- **Permissive licenses**: others do essentially whatever they want with your code, as long as they give your attribution. Examples: MIT, BSD



SOFTWARE LICENSES

- **no license** ⇒ default copyright laws apply. You retain all rights to your source code; nobody else may reproduce, distribute, or create derivative works from your work.
- **Permissive licenses**: others do essentially whatever they want with your code, as long as they give your attribution. Examples: MIT, BSD
- **Copyleft licenses**: Derivative work must be made available under the same terms as the original work (***viral licenses***). Example: GPL



SOFTWARE LICENSES

- **no license** $\Rightarrow$ default copyright laws apply. You retain all rights to your source code; nobody else may reproduce, distribute, or create derivative works from your work.
- **Permissive licenses**: others do essentially whatever they want with your code, as long as they give your attribution. Examples: MIT, BSD
- **Copyleft licenses**: Derivative work must be made available under the same terms as the original work (***viral licenses***). Example: GPL

If you are paid by UWE or UoB, the copyright belongs to the uni.

SOFTWARE LICENSES

- **no license** $\Rightarrow$ default copyright laws apply. You retain all rights to your source code; nobody else may reproduce, distribute, or create derivative works from your work.
- **Permissive licenses**: others do essentially whatever they want with your code, as long as they give your attribution. Examples: MIT, BSD
- **Copyleft licenses**: Derivative work must be made available under the same terms as the original work (***viral licenses***). Example: GPL

Check **http://choosealicense.com/**

WHAT IF YOU WANT TO USE A GPL LIBRARY?

There is a legal dispute to know whether merely ***linking*** with a library result in a ***derivative work*** (which would then have to be licensed as GPL).

WHAT IF YOU WANT TO USE A GPL LIBRARY?

There is a legal dispute to know whether merely ***linking*** with a library result in a ***derivative work*** (which would then have to be licensed as GPL).

The LGPL (***Lesser GPL***) explicitely allows the usage of the library without putting restrictions on the licensing of the resulting executable.

Open-source vs Free software?

"When we call software "free," we mean that it respects the users' essential freedoms: the freedom to run it, to study and change it, and to redistribute copies with or without changes. This is a matter of freedom, not price, so think of "free speech," not "free beer.""

"Open source is a development methodology; free software is a social movement"

Source: GNU website

break!

GIT

These directories are git repositories

They look boringly normal

A few roundtrips later with teammates...

We can do better!

We can do better!

git is essentially about recording the history of files

We can do better!

git is essentially about recording the history of files

(and who did what)

We can do better!

git is essentially about recording the history of files

(and who did what)

(and sharing as well)

CODE VERSIONING

WHY VERSIONING?

- The history of your development/document
- Compare the current code with an older version
- Roll-back to previous versions (think 'undo on steroids')
- Experiment without losing anything
- Trace who did what (at the level of the line of code)
- Annotate your workflow (important milestones, etc)
- Avoid catastrophes!

ATOMIC COMMITS

The single most important concept (because it requires to think about development/writing in terms of **functional units**):

Atomic commit

A (typically small) commit that represent a **single, coherent & complete** functional change.

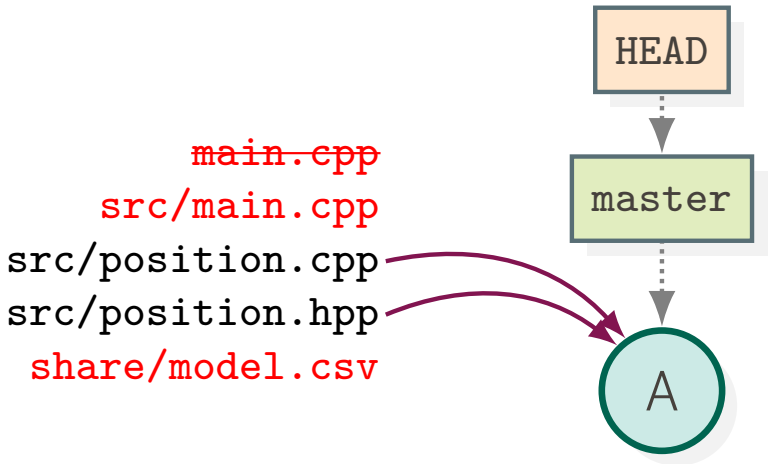
ATOMIC COMMITS

The single most important concept (because it requires to think about development/writing in terms of **functional units**):

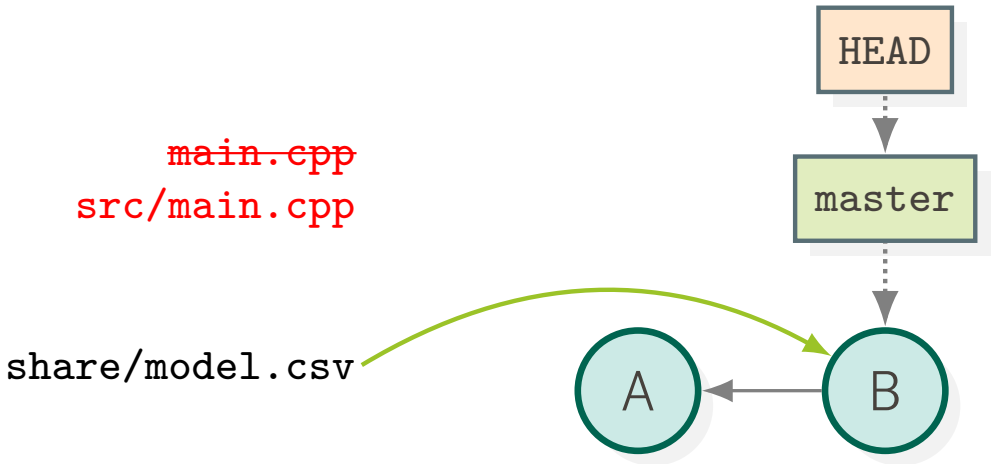
Atomic commit

- Easy to understand the change
- Debugging made easy (`git bisect`)
- Collaboration made easy (less, smaller conflict)
- Easy to write a useful commit message

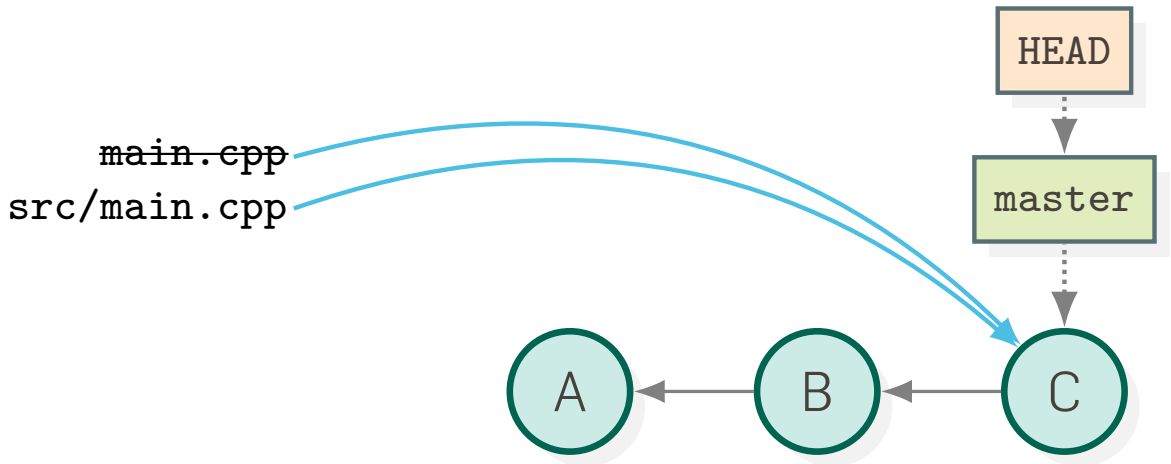
~~main.cpp~~
src/main.cpp
src/position.cpp
src/position.hpp
share/model.csv




```
git add src/position.*
git commit -m"Fix computation of position (float->double)"
```



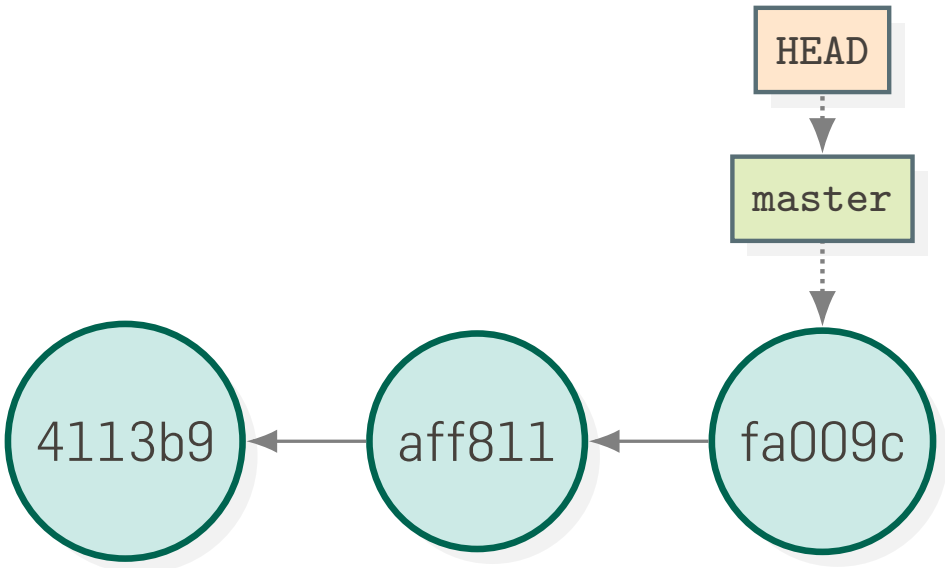

```
git add share/model.csv
git commit -m"Re-trained model with 52 more participants"
```




```
git rm main.cpp
git add src/main.cpp
git commit -m"Move main.cpp to src/"
```

LOG

```
> git log
commit fa009cd7fca05b0b61170b20cf76a5f72b8843c2
Author: Severin Lemaignan <severin.lemaignan@brl.ac.uk>
Date:   Wed Feb 10 16:48:22 2016 +0000

    Move main.cpp to src/

commit aff81119459d9193c09effef1c150c4f7eac08dc
Author: Severin Lemaignan <severin.lemaignan@brl.ac.uk>
Date:   Wed Feb 10 16:48:02 2016 +0000

    Re-trained model with 52 more participants

commit 4113b9b6e6bbc8de532ad90153e0059cb5819de7
Author: Severin Lemaignan <severin.lemaignan@brl.ac.uk>
Date:   Wed Feb 10 16:47:46 2016 +0000

    Fix computation of position (float->double)
```


HEAD
master
4113b9
aff811
fa009c

THE STAGING AREA

But why do we have to manually tell Git what files to add or remove?

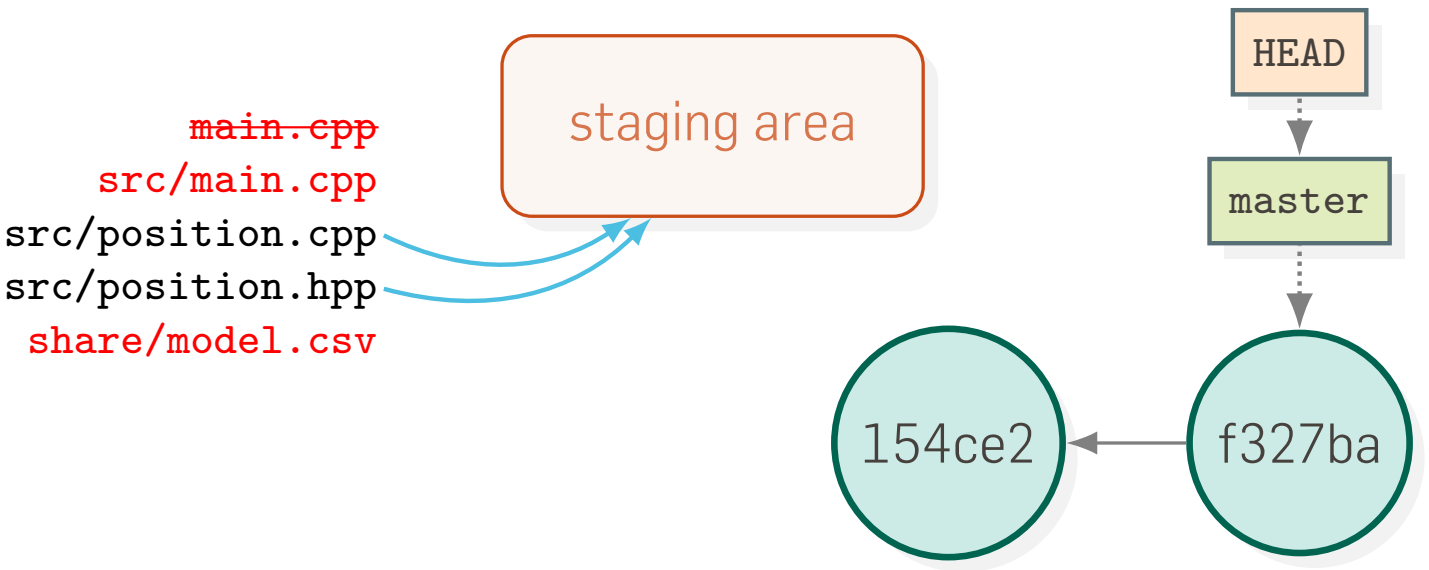
THE STAGING AREA

No “commit all changes” by default (well, you can, actually...)

$\Rightarrow$ Help thinking in terms of atomic commits!

THE STAGING AREA

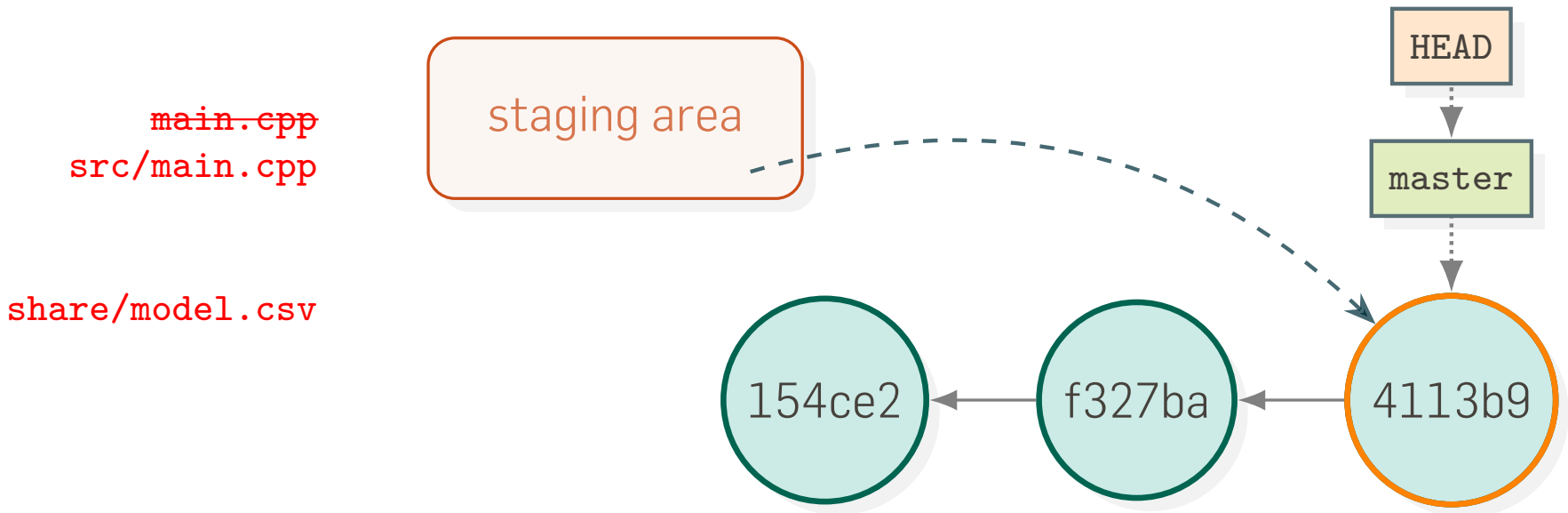
Preparing a commit consists in filling the **staging area** (or **index**) with the list of changes:




```
git add
git rm
git add -p
...
```

THE STAGING AREA

Preparing a commit consists in filling the **staging area** (or **index**) with the list of changes:




```
git commit
```

TO SUMMARIZE...

The first time...

```
> mkdir my_repo && cd my_repo
> git init
```

Then...

```
# make some changes...
> git add <files>
> git commit -m"<commit message>"
# make some changes...
> git add <files>
> git commit -m"<other commit message>"
# That's it!
```

Viewed from a GUI (macOS & Windows)
GitHub Desktop Walkthrough

https://desktop.github.com/

Log in to your GitHub account

Create a (local) repository

GitHub Desktop has already made a first commit on your behalf

Open the repo in Windows Explorer

Add a simple README.md...

The change is listed in the Changes panel

Write a commit message & commit!

The History panel shows the log and a diff of your changes

Viewed from a GUI

Tortoise GIT

https://tortoisegit.org/

Direct interaction in the Windows explorer

Files’ status appear as icons

All the functionalities are available

Commit window

WHAT SHOULD BE TRACKED?

Short answer: **everything you care about in your project**

WHAT SHOULD BE TRACKED?

Short answer: **everything you care about in your project**

(you can left out temporary files, automatically generated files, etc)

WHAT SHOULD BE TRACKED?

Short answer: **everything you care about in your project**

(you can left out temporary files, automatically generated files, etc)

However, versioning is **less useful for binary files**:

- no line-by-line tracking of changes
- every single change creates a whole copy: repo size might grow quickly!

Binary files include images, archives (zip files), **PDF**, **most office document (docx/xlsx/pptx)**

WHAT SHOULD BE TRACKED?

Short answer: **everything you care about in your project**

(you can left out temporary files, automatically generated files, etc)

However, versioning is **less useful for binary files**:

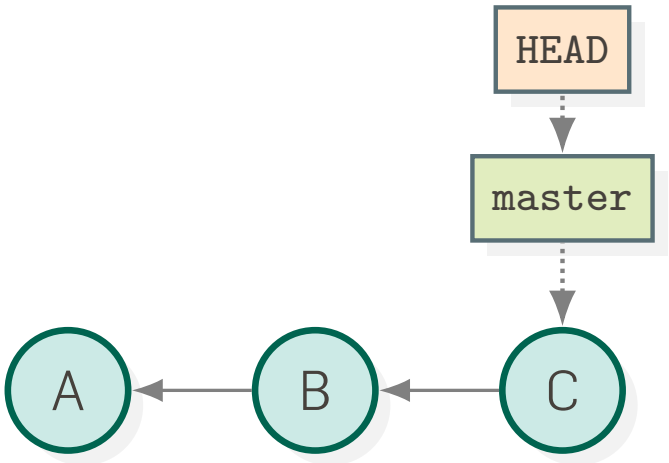
- no line-by-line tracking of changes
- every single change creates a whole copy: repo size might grow quickly!

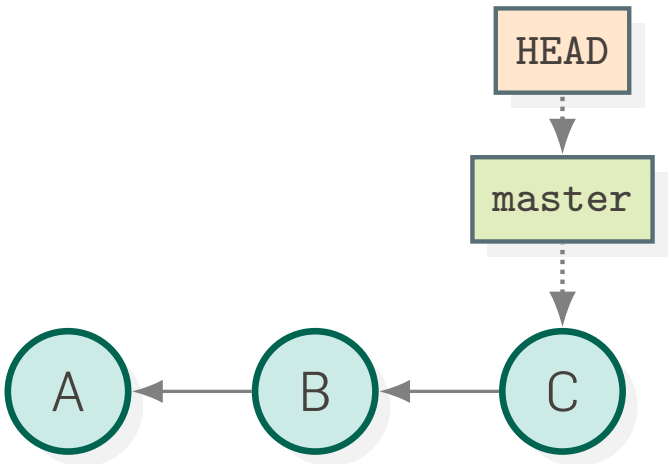
Binary files include images, archives (zip files), **PDF**, **most office document (docx/xlsx/pptx)**

For documents, you might want to consider alternative like markdown.

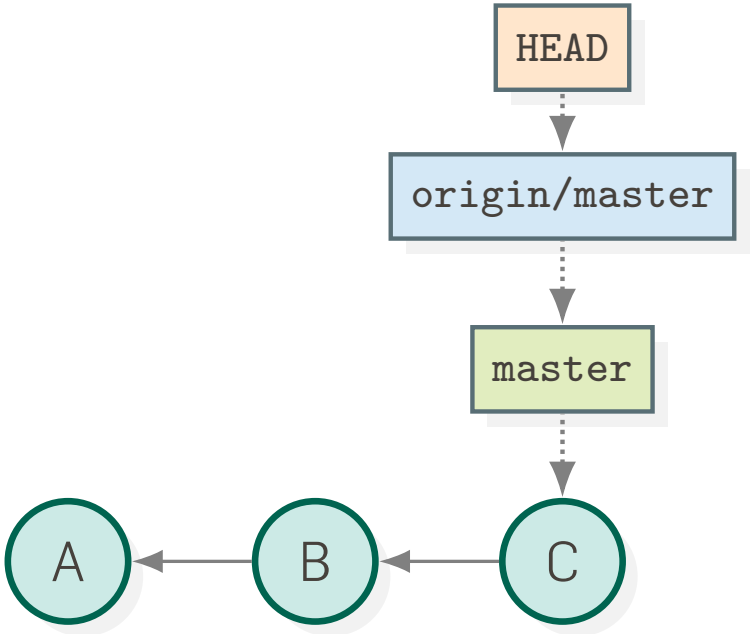
What should I track here?

COLLABORATING


HEAD
master
A
B
C

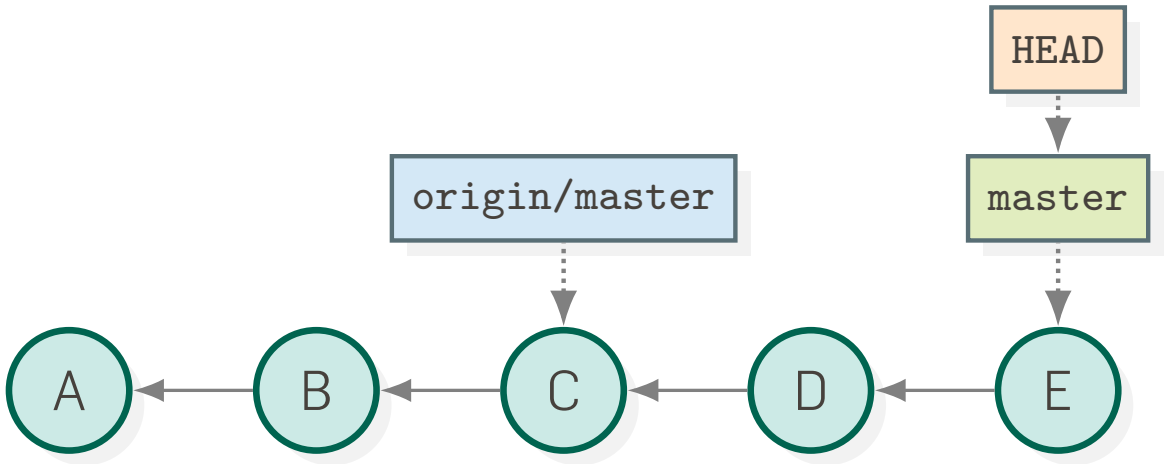


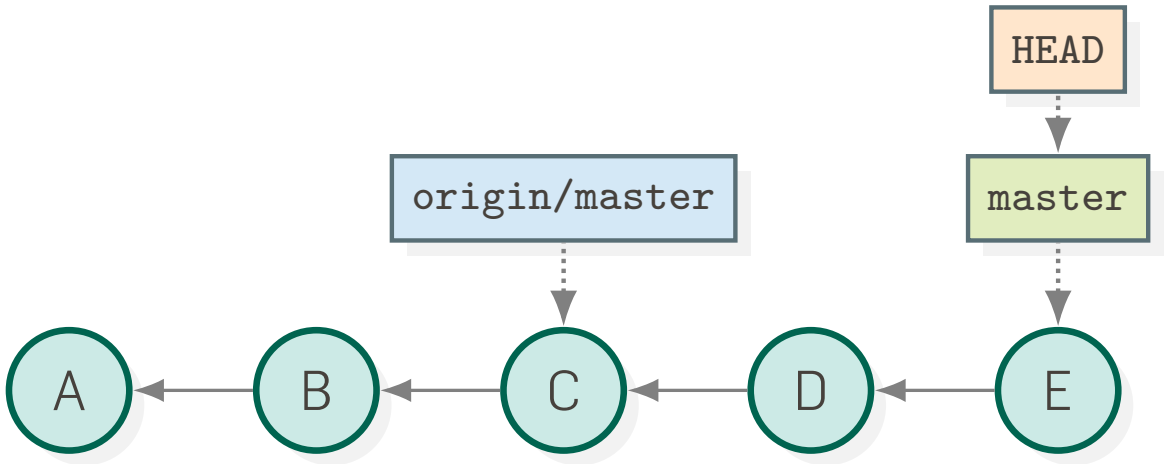

```
git remote add origin git@github.com:user/repo.git
git remote add john-usb E:\john_repo
git remote add ftp-origin ftp://host.xz/path/to/repo.git/
...
```



`git push origin master`

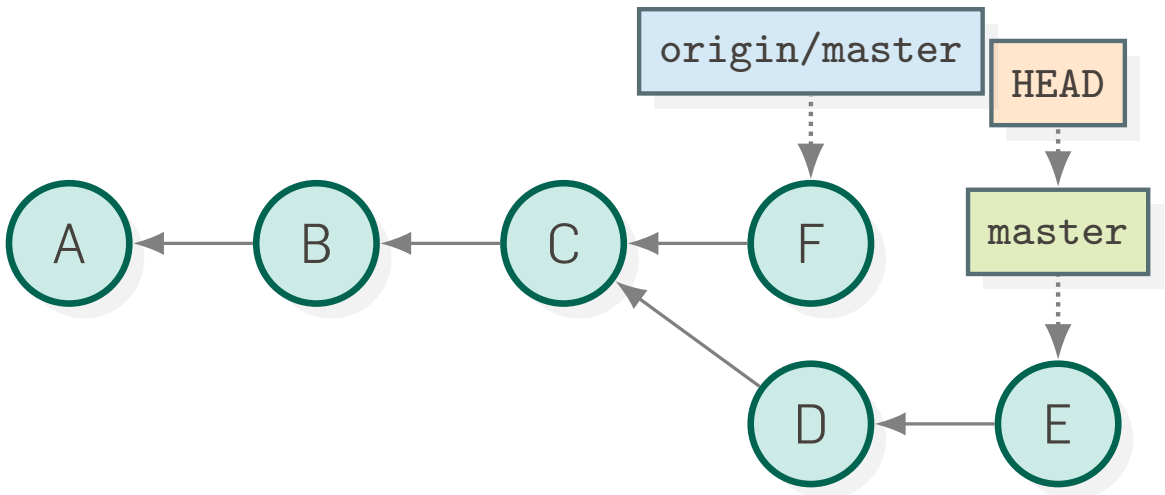
(or simply `git push`)

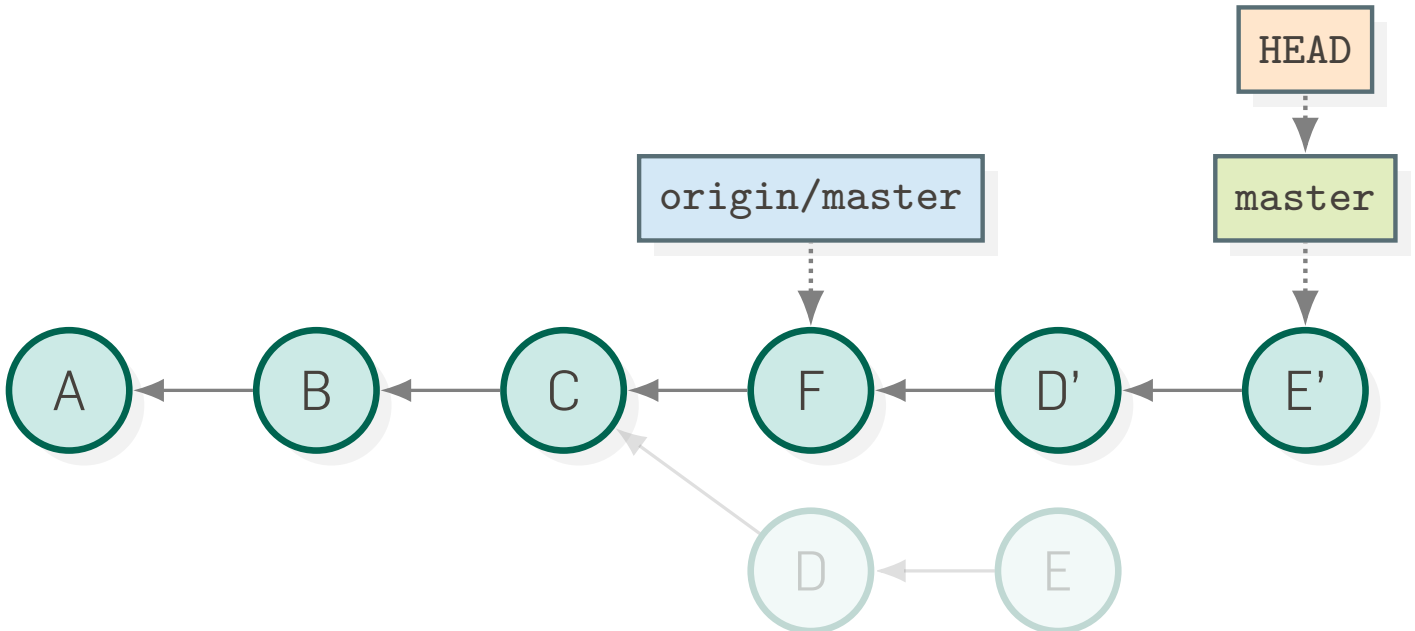

HEAD
origin/master
master
A
B
C
D
E



What happened on our remote? Let's have a look...

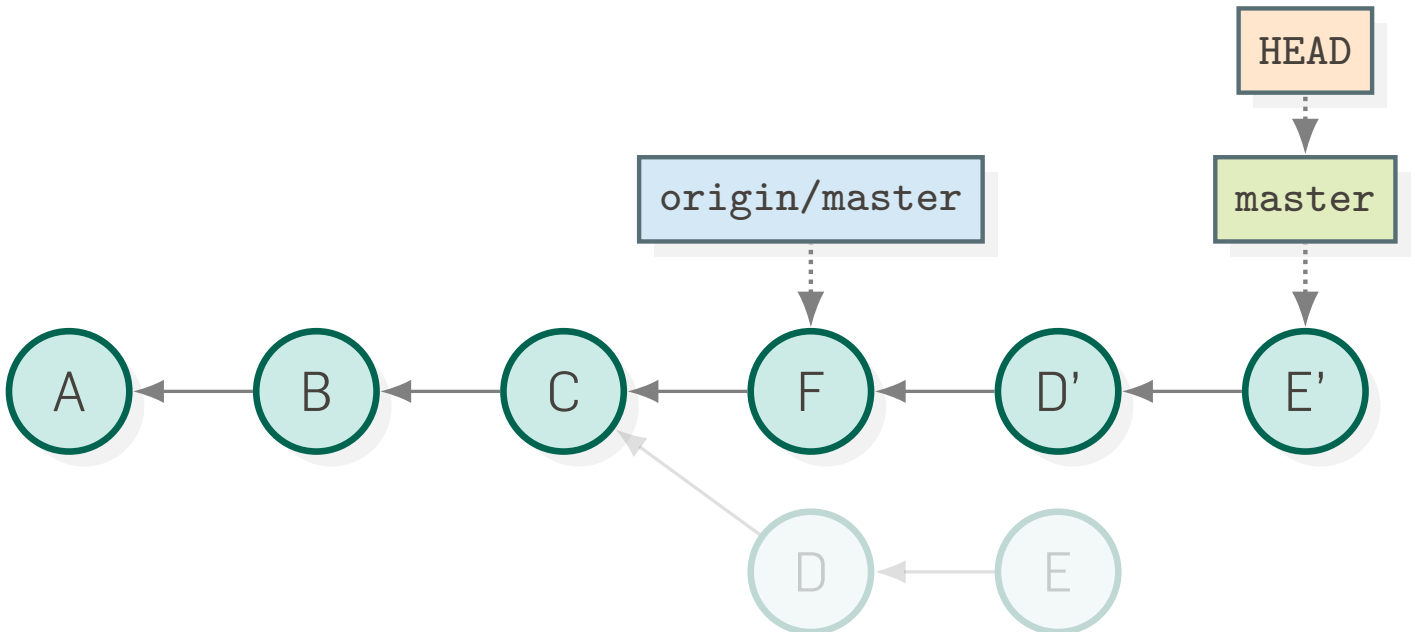
```
git fetch origin
```


origin/master
HEAD
A
B
C
F
master
D
E

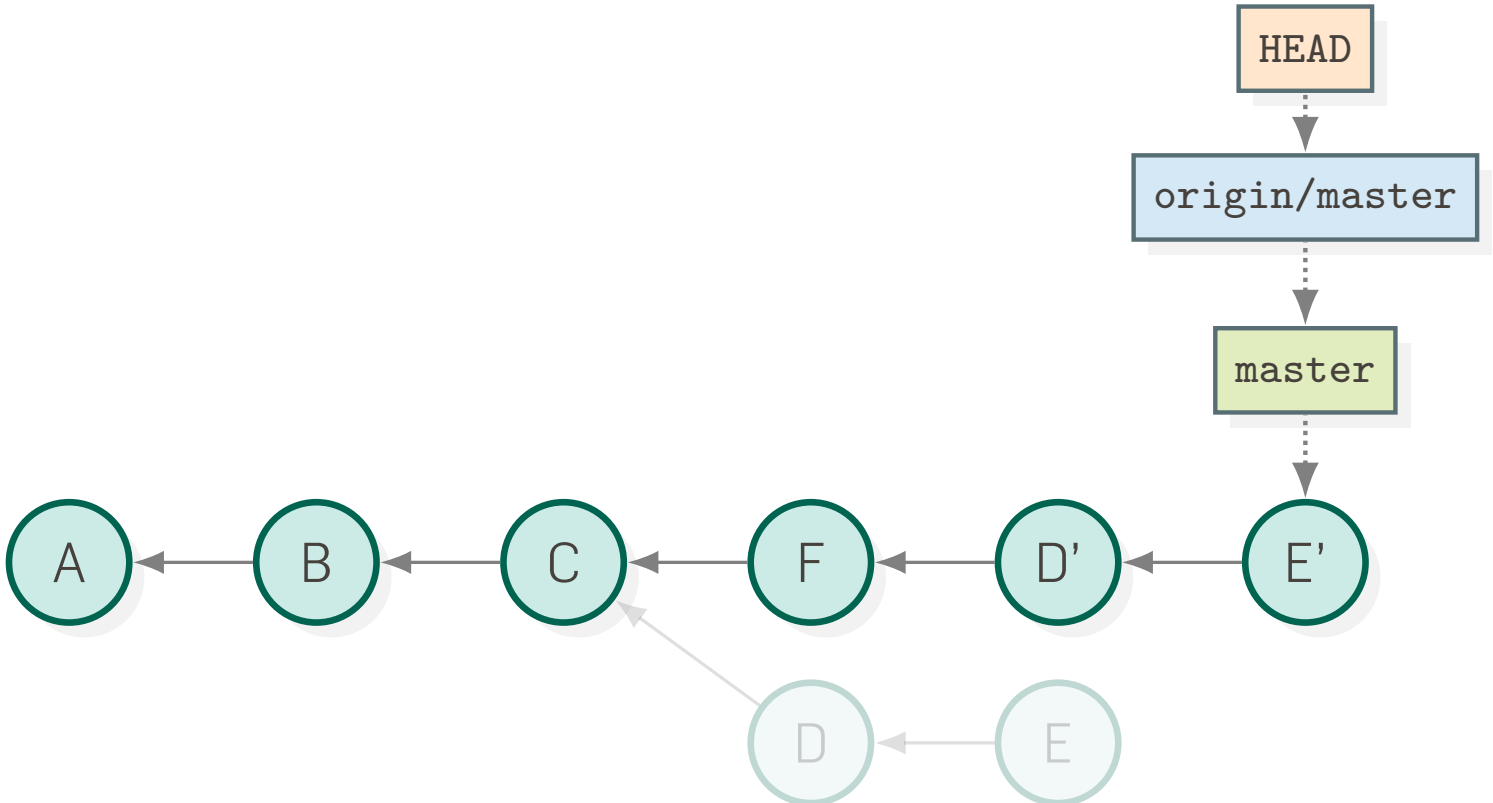


`git rebase origin/master`

(but you don't need it, because...)




```
git pull --rebase
```



`git push`

TO SUMMARIZE...

The first time...

```
> git clone <url>
# for instance,
# git clone https://github.com/user/repo.git
```

Then...

```
> cd <repo>
# make some changes...
> git add <files>
> git commit -m"<commit message>"
# ...
# when you want to share:
$ git pull --rebase # any changes on the remote?
$ git push
```

THE DREADFUL CONFLICTS

THE DREADFUL CONFLICT

While peacefully editing your last (great) report...

```
$ git pull --rebase john master
First, rewinding head to replay your work on top of it...
Applying: Better terminology
Using index info to reconstruct a base tree...
M       main.tex
Falling back to patching base and 3-way merge...
Auto-merging main.tex
CONFLICT (content): Merge conflict in main.tex
error: Failed to merge in the changes.
Patch failed at 0001 Better terminology
The copy of the patch that failed is found in: .git/rebase-appl

When you have resolved this problem, run "git rebase --continue
If you prefer to skip this patch, run "git rebase --skip" inste
To check out the original branch and stop rebasing, run "git re
```

A conflict happens when two modifications of a given file overlap

Two persons can modify the same file at the same time, as long as they do not work on the same region of the file.

```
> git pull --rebase john master
# conflict!
> git mergetool
```

Meld is one of the nice tools to fix conflicts

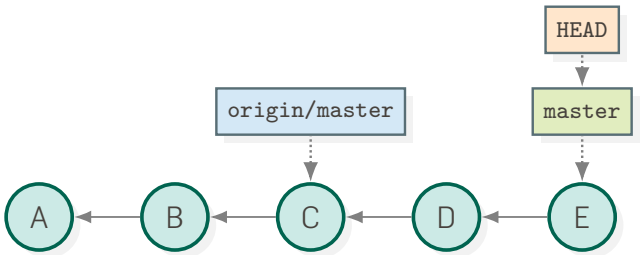
SOCIAL CODING: GITHUB WORKFLOW

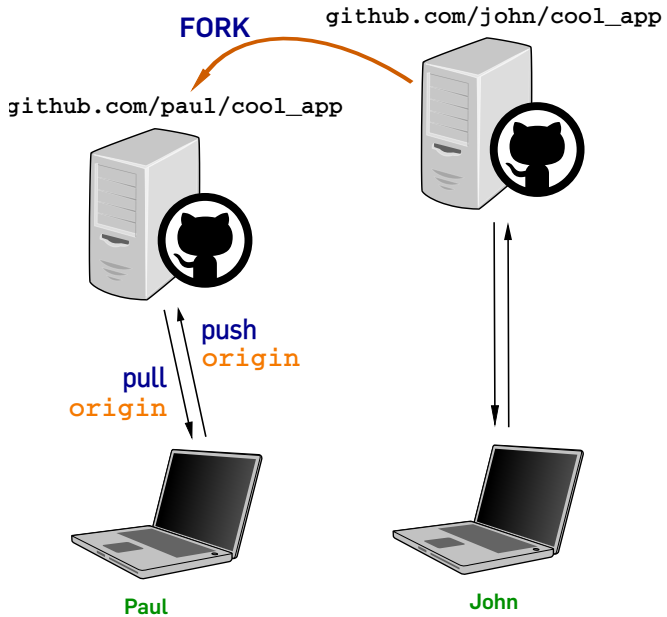
GitHub

BitBucket

GitLab – open-source You can install it on your own server

WHAT HAPPENED EXACTLY?



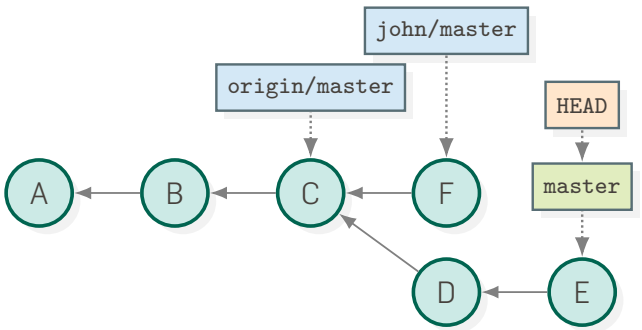


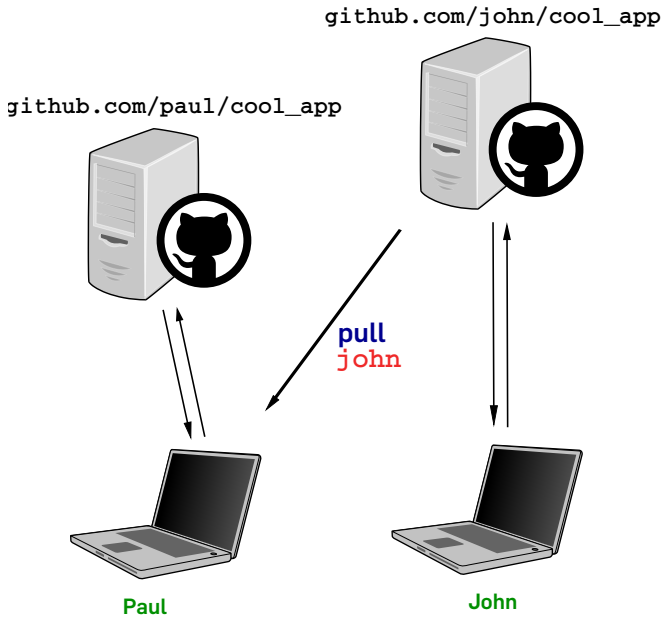
After forking on GitHub, Paul runs

`git clone https://github.com/paul/cool_app.git`

and he adds few local commits

WHAT HAPPENED EXACTLY?



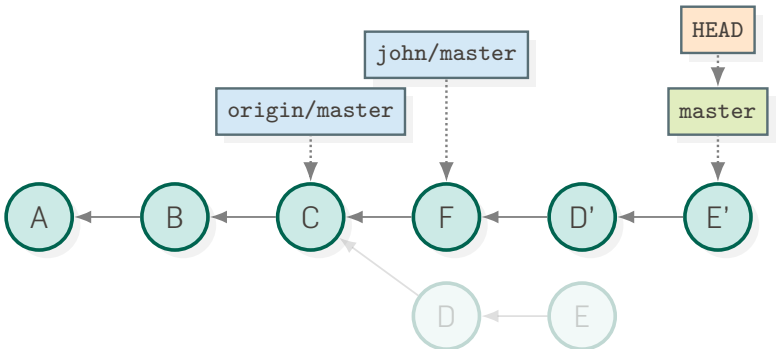


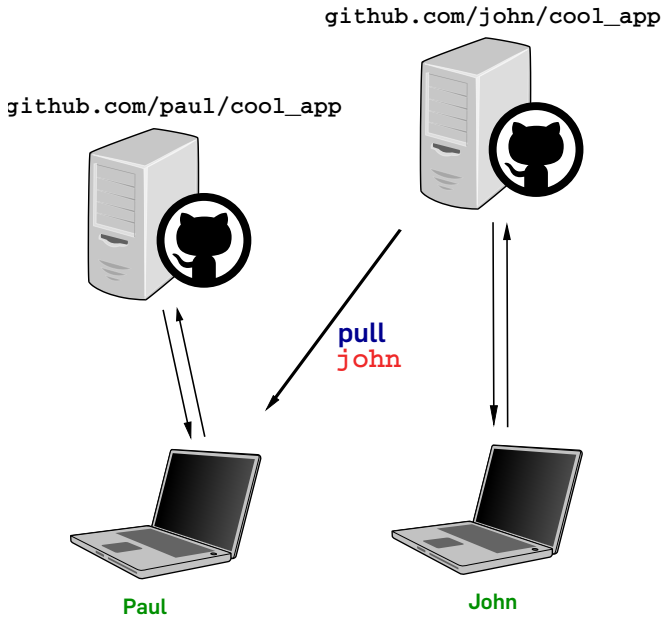
He would like to propose his changes to John

First, he needs to get the latest changes from John:

```
git add remote john https://github.com/john/cool_app.git
git fetch john
```

WHAT HAPPENED EXACTLY?



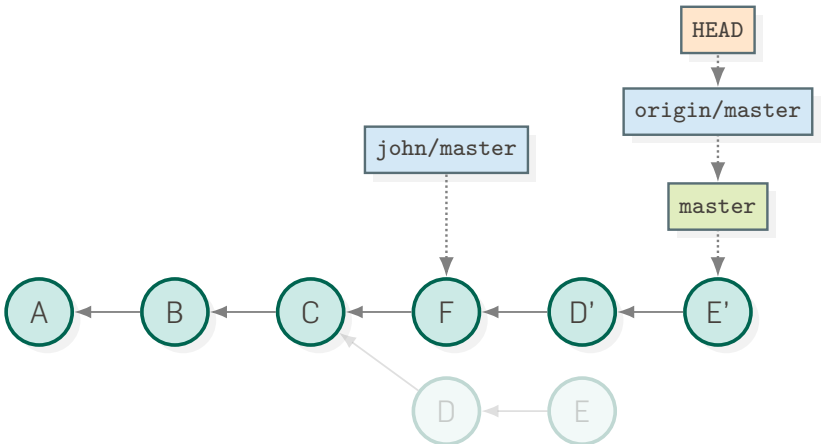


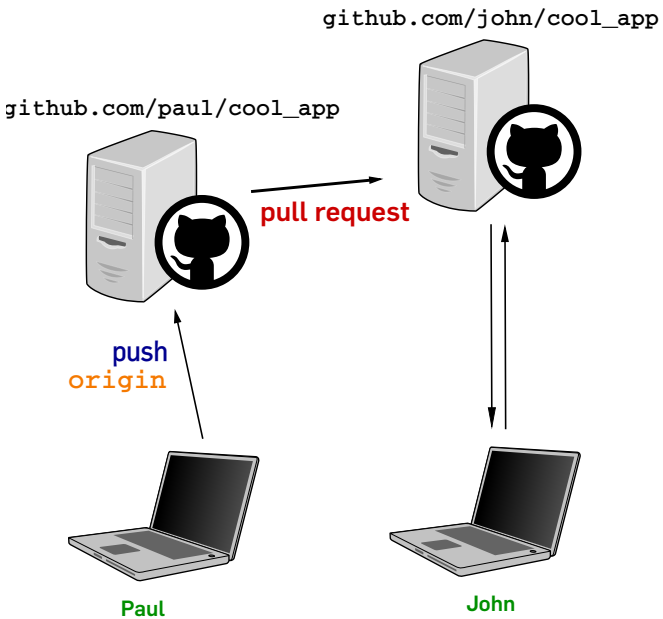
Paul rebases his `master` branch on John's one:

`git rebase john/master`

(actually, Paul would simply run `git pull --rebase john master`)

WHAT HAPPENED EXACTLY?





He pushes his commits to his own GitHub account:

`git push`

...and finally press the “Create a pull request” button in GitHub.

(what happens next on John's side is a story for another day :-)
But to make it short, he can press "Merge pull request" on his GitHub account if he is happy with the pull-request!)

GITLAB@BRL

We have our own 'GitHub': **git.brl.ac.uk**

If you can not yet login, drop an email to **itonline@uwe.ac.uk** and ask for access.

GitLab@BRL

GitLab@BRL

THE ONE SLIDE TO REMEMBER

GIT CHEAT SHEET

To start...
...from scratch: `git init`
...from existing repo: `git clone <url>`

Prepare commits:
`git add`
`git rm`
`git add -p` (partial files)

Commit:
`git commit`

Create branch:
`git checkout -b <branch>`

Jump between branches:
`git checkout <branch>`

“Import” another branch:
`git rebase <other_branch>`

Add a remote source:
`git remote add <name> <url>`

What’s new on a remote?
`git pull <remote> <branch>`
(`git pull` alone ≡ `git pull origin master`)

Share stuff on a remote:
`git push <remote> <branch>`
(`git push` alone ≡ `git push origin master`)

Repo state
`git status`

Repo history
`git log`

Who did what?
`git blame`

I’ve lost everythg!
`git reflog`

COMMIT HYGIENE

“Show me the project history, I’ll tell you what coder you are”

- **Commit often!** Push when needed (or at the end of day)

Because commits are local (ie, private), **do commit often**: **mistakes are ok** as you can fix them before sharing with others.

COMMIT HYGIENE

“Show me the project history, I’ll tell you what coder you are”

- Write useful messages (no “`Fixed bug`” or “`New file`”)
- First line of commit messages < 72 characters

COMMIT HYGIENE

“Show me the project history, I’ll tell you what coder you are”

- Tag important commits!

Notably, GitHub (amongst others) interpret tags as **releases** of your code.

one repo = one thing

make plenty of repos!

A FEW COOL GITHUB STUFF TO FINISH

Besides bugtracking, project homepages and wikis,GitHub integrates with many third-party services & tools:

- **Travis CI** or **AppVeyor** for continuous integration

A FEW COOL STUFF TO FINISH

+ GitHub integrates with many external services & tools:

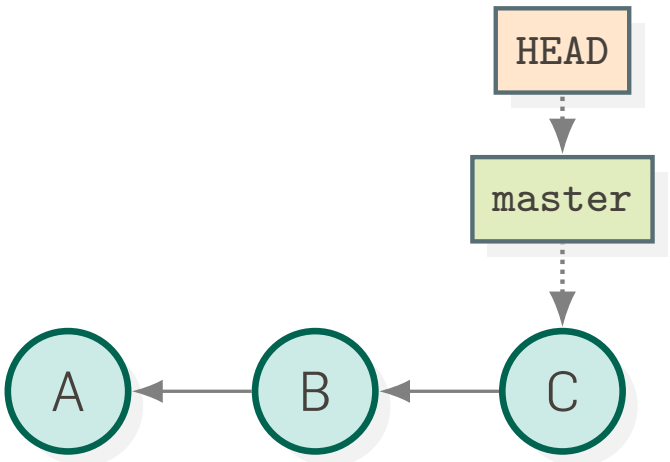
  - **Travis CI** or **AppVeyor** for continuous integration
  - **zenodo**: associate a DOI to your repository
  - **ReadTheDocs**: generate and publish on-line documentation

That's all, folks!

Slides:
github.com/severin-lemaignan/lecture-software-engineering

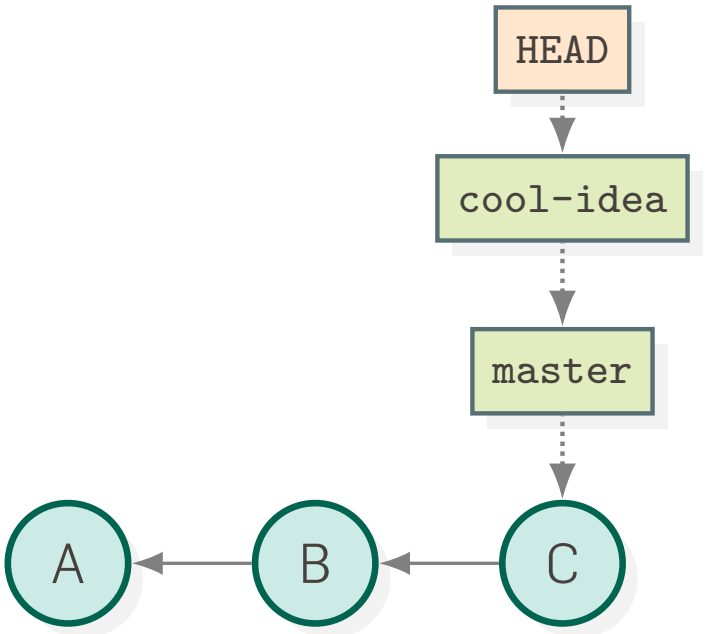
WORKING WITH BRANCHES

BRANCHES



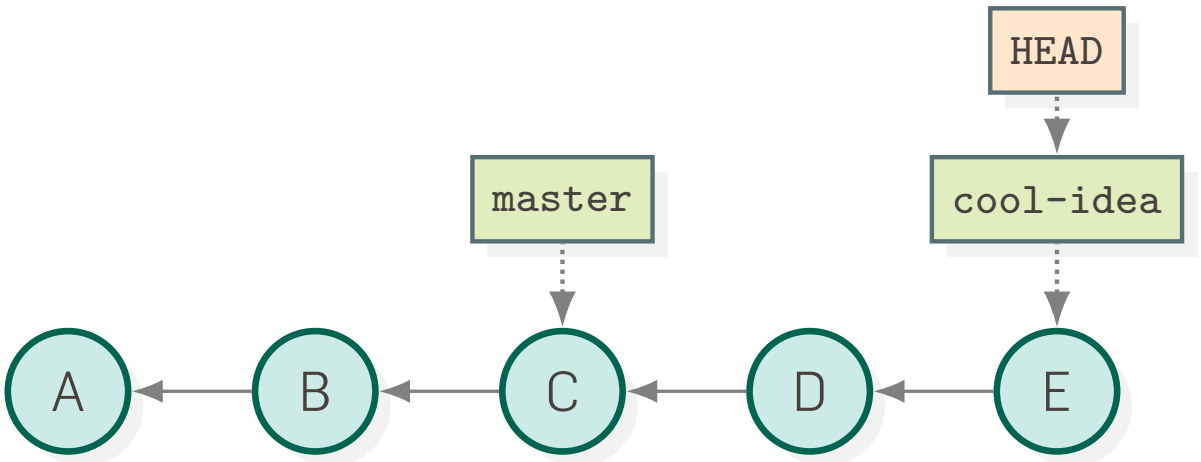
What if...?

BRANCHES

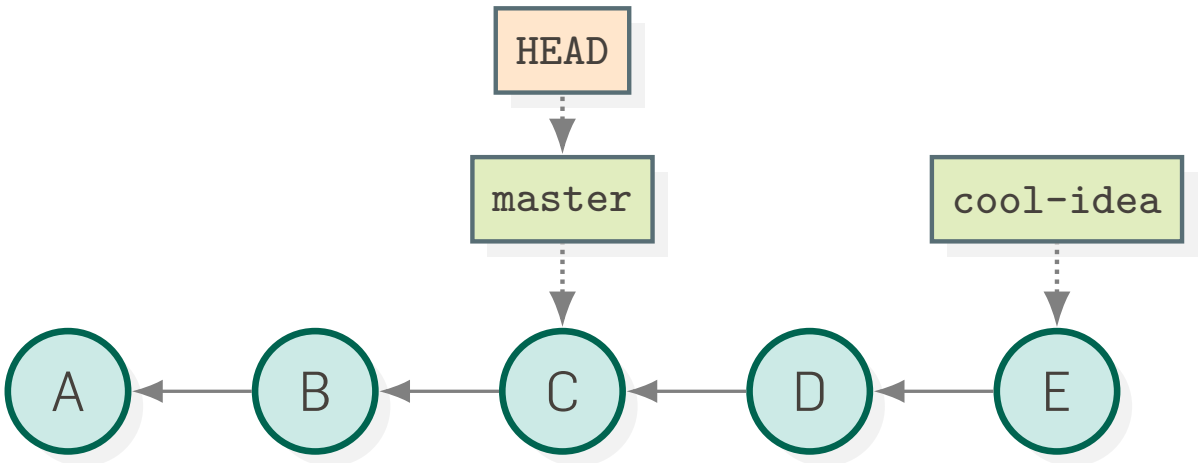



```
git checkout -b cool-idea
```

BRANCHES



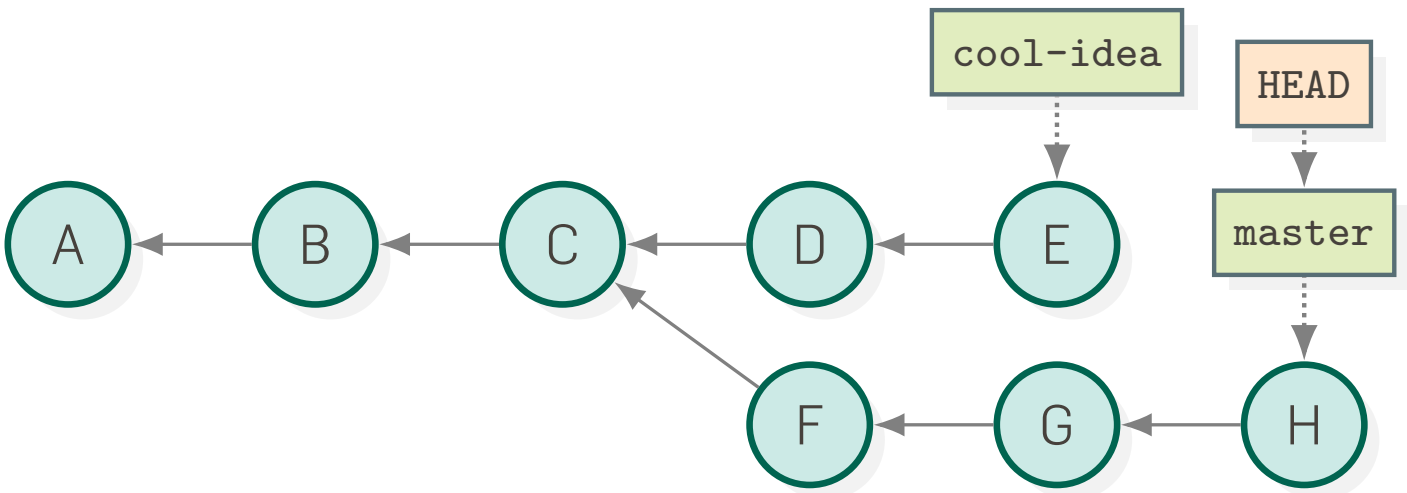
BRANCHES



Let go back to serious stuff!

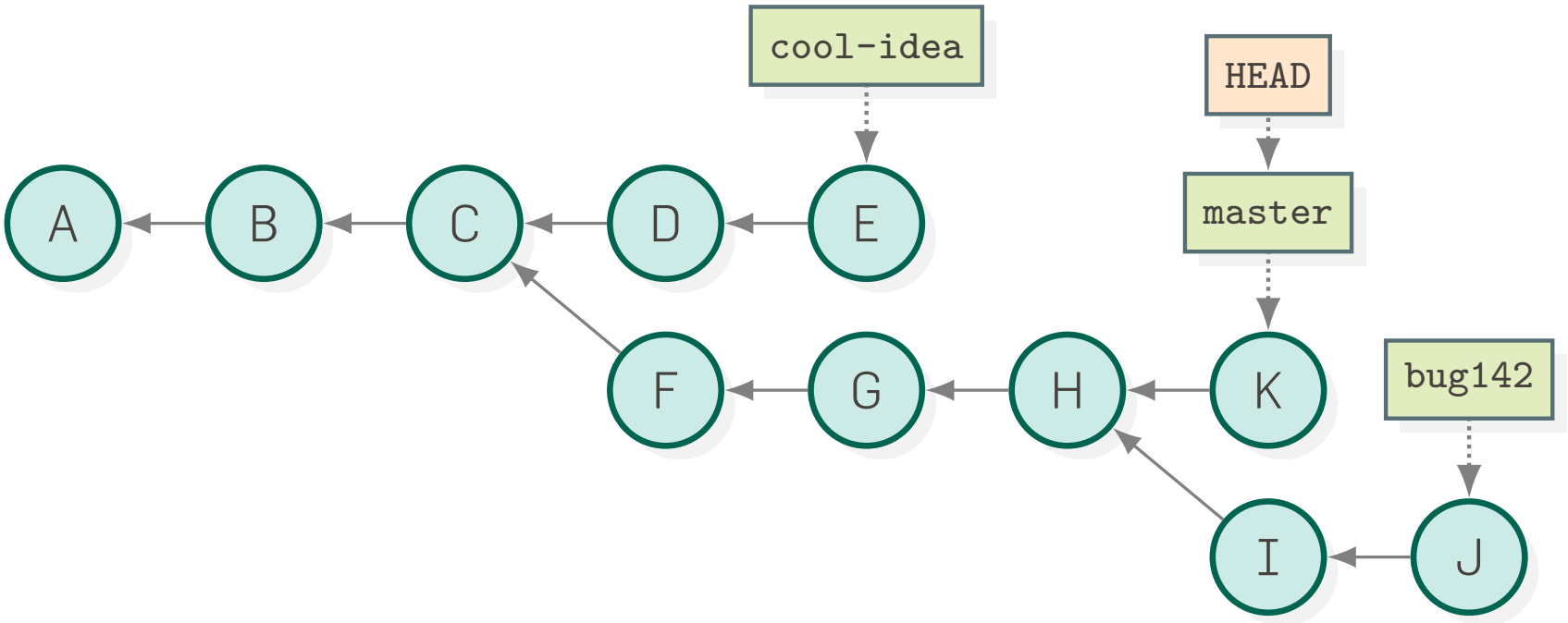
```
git checkout master
```

BRANCHES



The branch name is an alias for the tip of the current branch

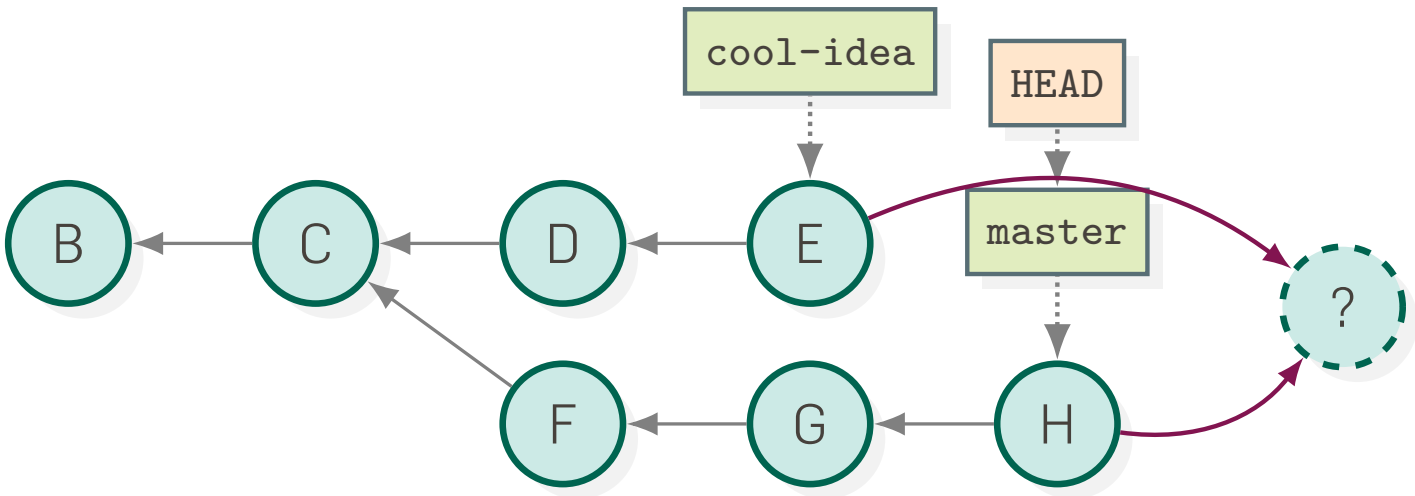
BRANCHES



⇒ branches are very cheap

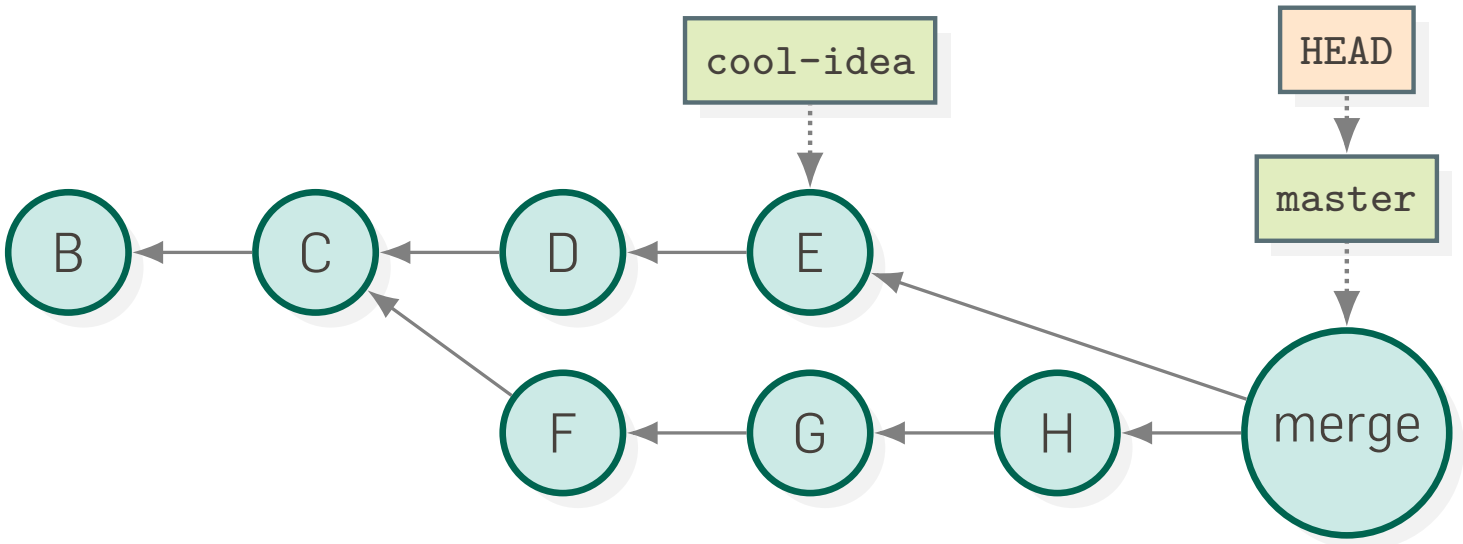
+10 of them at a given time it not uncommon

MERGING BRANCHES



Two options: **merging** and **rebasing**

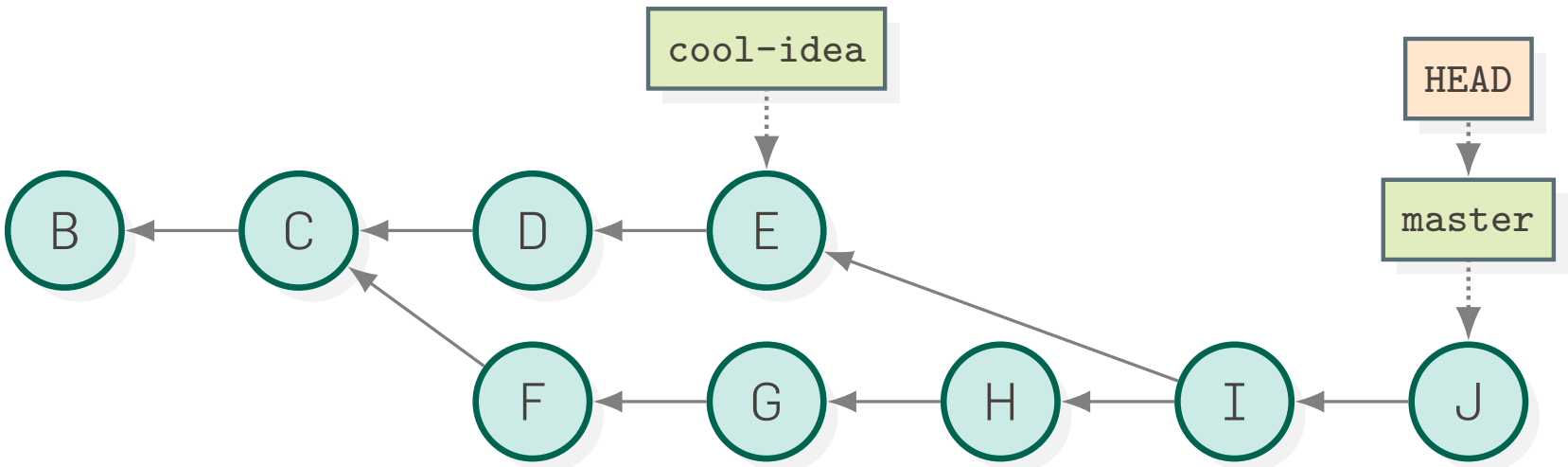
MERGING BRANCHES



Merging

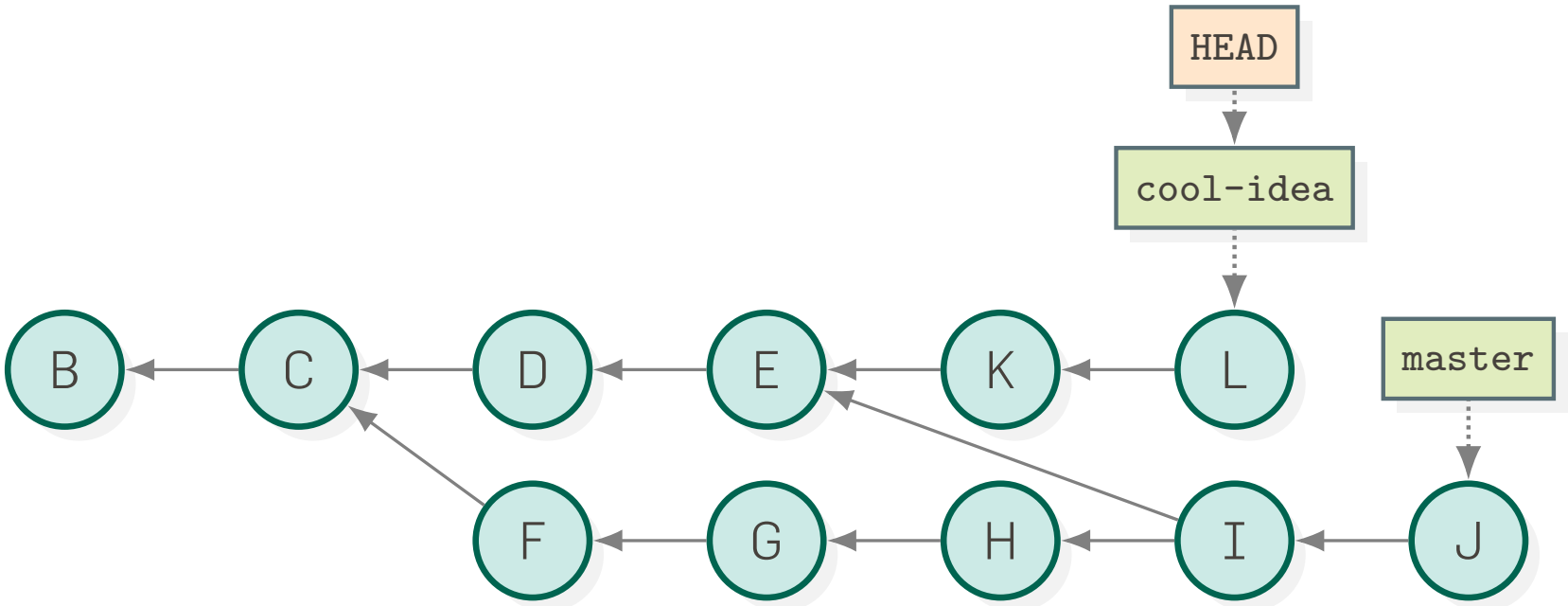
```
git merge cool-idea
```

MERGING BRANCHES




```
git commit
```

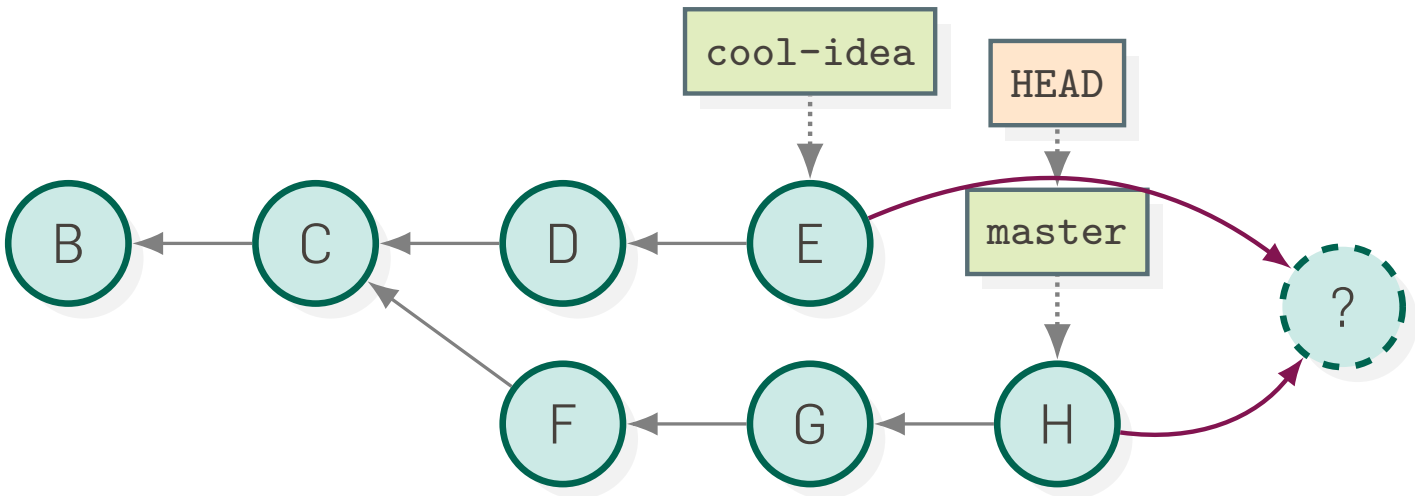
MERGING BRANCHES



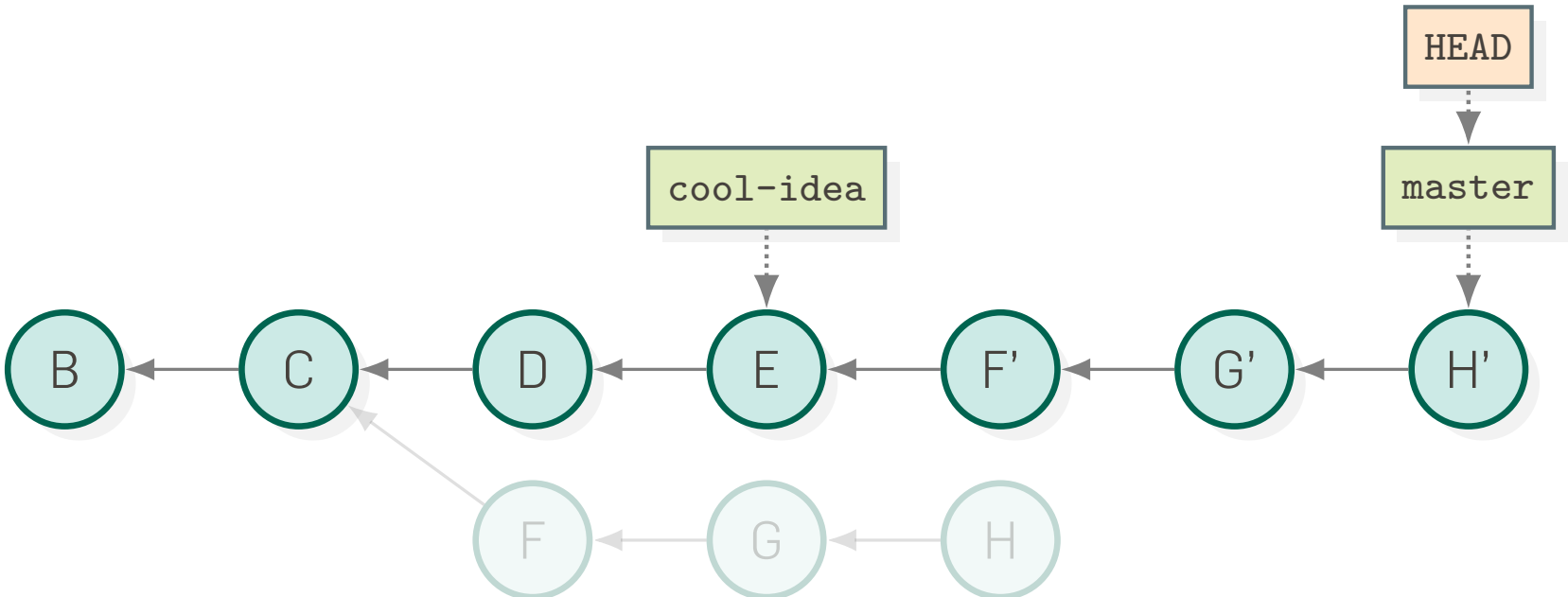

```
git checkout cool-idea
git commit
```

...etc.

REBASING BRANCHES



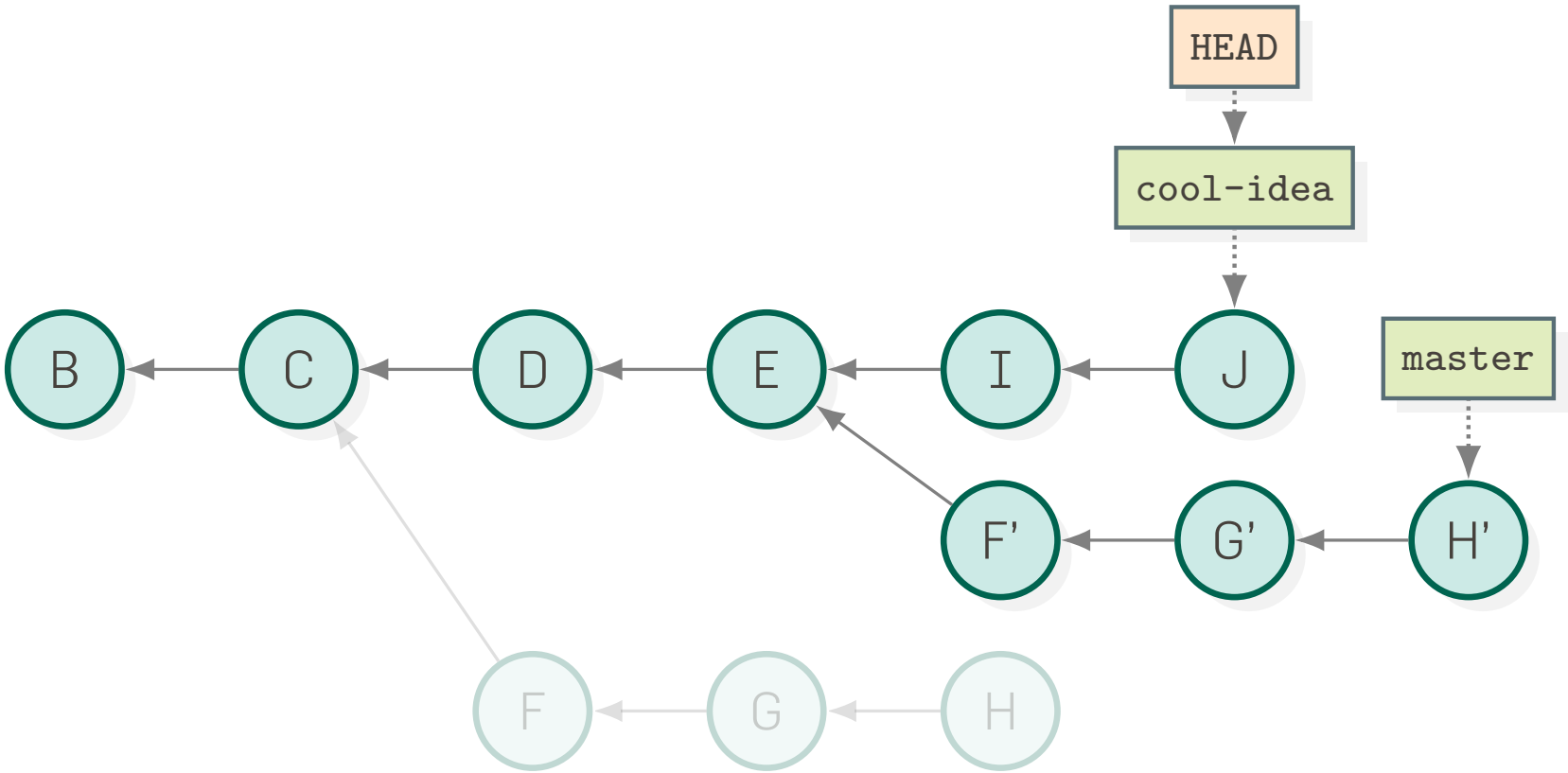
REBASING BRANCHES



Rebasing

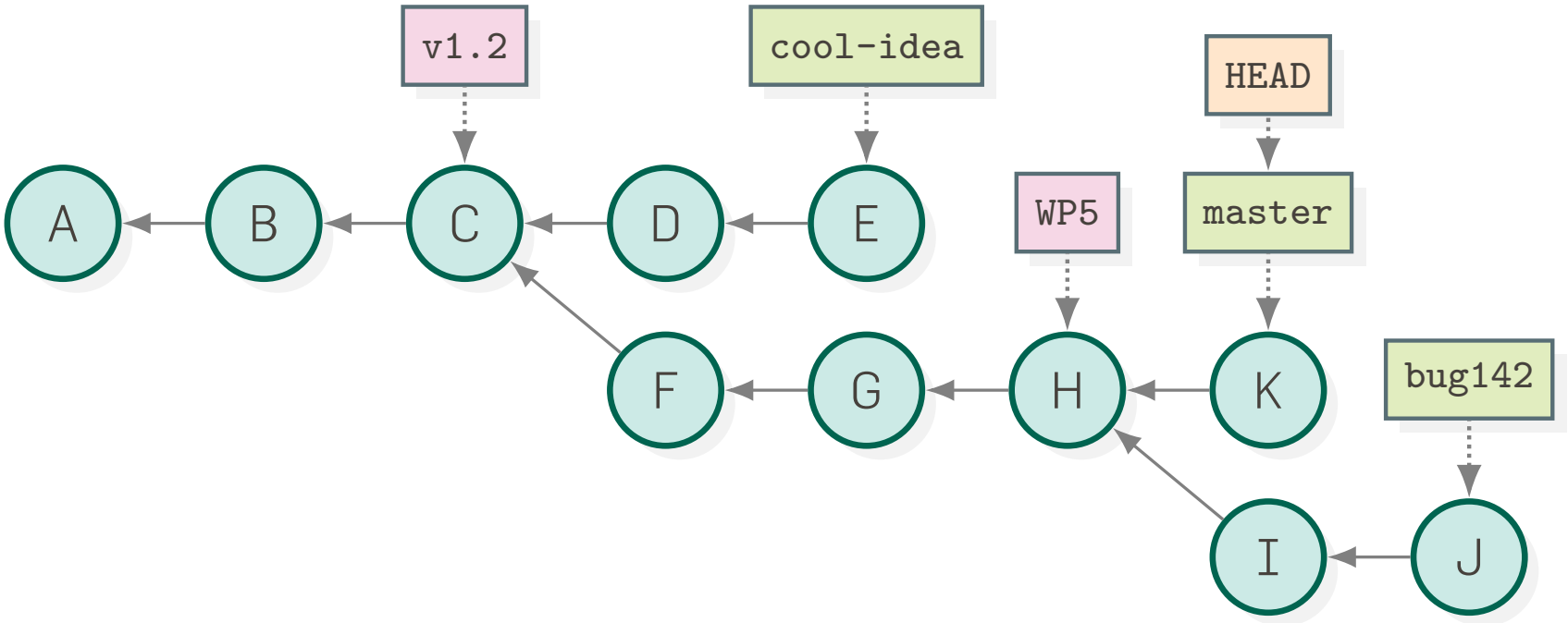
```
git rebase cool-idea
```

REBASING BRANCHES




```
git checkout cool-idea
git commit
```

MORE COMMIT ALIASES: TAGS



Label important commits/milestones

```
git tag v1.2
git tag WP5
```

TO SUMMARIZE...

```
# where are we?
$ git branch
master
# make some changes...
$ git add <files> && git commit -m"<commit message>"
# start working on something new?
$ git checkout -b new-idea
$ git branch
new-idea
# work in that branch for a while
$ git add <files> && git commit -m"<commit message>"
# back to master
$ git checkout master
#...
# rebase master on new-idea: new-idea is now in master
$ git rebase new-idea
```

Viewed from a GUI...

We can easily create a new branch

We can compare numerical_coordinates with master (click on View branch for the full history)

We can jump between branches...

...and watch how they diverge

We switch back to numerical_co-
ordinates and merge in master

The merge commit is reflected in the history of the branch