Tecnologias para a Produção e Publicação de Informação

Licenciatura em Tecnologias da Informação e Comunicação
&
TeSP em Sistemas e Tecnologias de Informação

Pedro Cardoso 

pcardoso@ualg.pt http://w3.ualg.pt/~pcardoso

Departamento de Engenharia Electrotécnica
Instituto Superior de Engenharia – Universidade do Algarve

Última atualização: 22 de Outubro de 2016

Resumo de conteúdos

3ª Aula

Ferramentas colaborativas – Controladores de Versões

Ferramentas Colaborativas

- Quatro grupos de ferramentas:
  - Controlador de Versão (e.g., cvs, svn, git)
  - Escrita Colaborativa (e.g., twiki, pmwiki, etc)
  - Gestão de Projetos (e.g., Microsoft Project, OpenProj)
  - Sincronização de Arquivos (e.g., googledocs, dropbox, etc)

Controlo de versões

- Um sistema de controlo de versão (ou versionamento), VCS (do inglês *version control system*) ou ainda SCM (do inglês *source code management*) na função prática da Ciência da Computação e da Engenharia de Software, é
  - um software com a finalidade de gerir diferentes versões no desenvolvimento de um documento;
  - um sistema de registo de alterações em arquivos ou conjunto de arquivos ao longo do tempo.

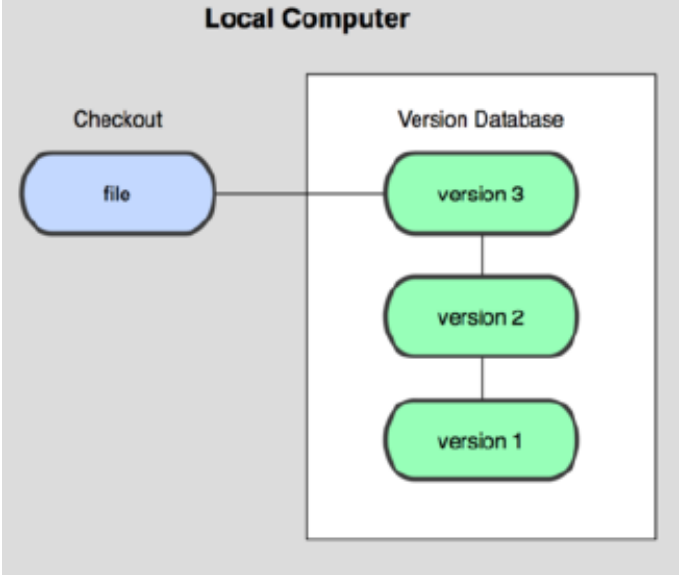
- Esses sistemas são comummente utilizados no desenvolvimento de software para controlar as diferentes versões — histórico e desenvolvimento – dos códigos-fontes e também da documentação.
- Para quem está envolvido em vários projetos ao mesmo tempo com várias equipas diferentes
- Exemplo de controladores de versões:
  - CVS, Mercurial, Git e SVN (livres)
  - SourceSafe, PVCS e ClearCase (comerciais)

Local Version Control Systems

- O controlo de versão de muitas pessoas é copiar os arquivos para outro diretório
  - Fácil esquecer onde está no diretório
  - Escrever sobre o arquivo errado ou copiar arquivos que não quero.
  - ...

Local Version Control Systems

- 1ª solução: programadores desenvolveram VCSs locais que usam uma base de dados para manter as alterações dos ficheiros sobre controlo da revisão – baseados em *patches*



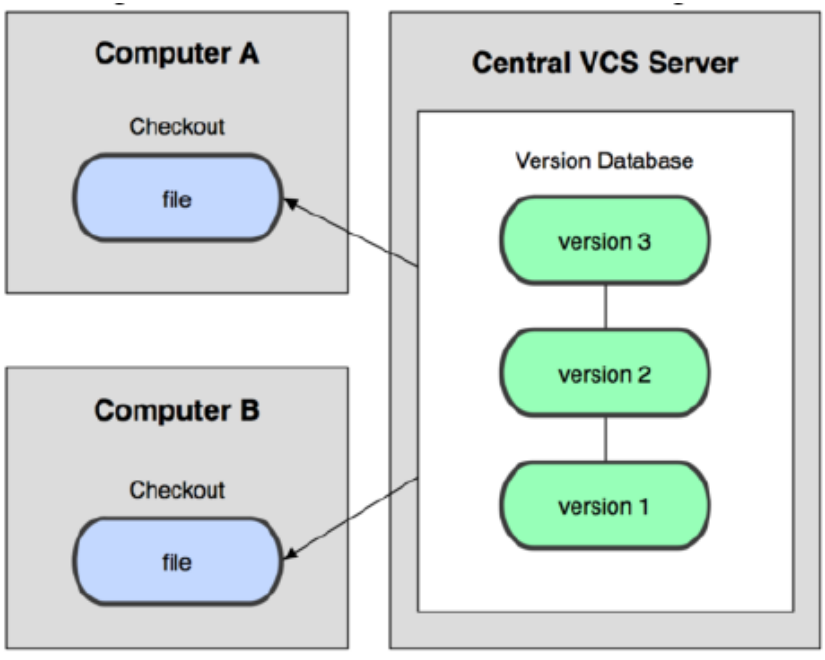
Centralized Version Control Systems I

- Os Sistemas de Controle de Versão Centralizado (Centralized Version Control Systems – CVCS) foram desenvolvidos para as pessoas que precisam de colaborar com desenvolvedores que trabalham noutros sistemas.
- Tem um único servidor (ex: CVS, perforce, subversion) que contém todos os arquivos versionados, e um conjunto de clientes que “utilizam” os ficheiros desse servidor central.
- Padrão para controle de versão durante muitos anos.

Centralized Version Control Systems II

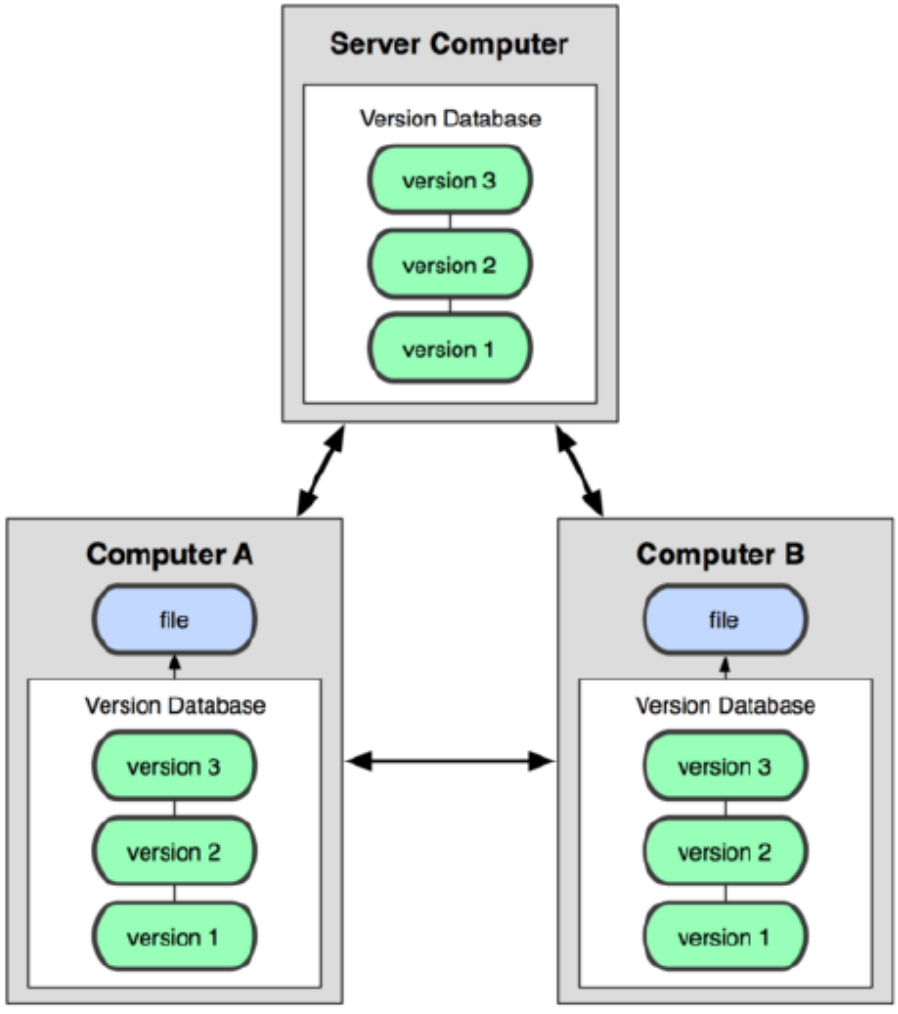
- Vantagens sobre VCSs locais:
  - Todos sabem o que os outros estão a fazer
  - Os administradores têm controle sobre quem pode fazer o quê
  - Mais fácil do que gerir várias bases de dados locais
- Desvantagens sobre VCSs locais:
  - Existe um único ponto de falha (servidor central)
  - Em caso de falha (por exemplo) do disco do servidor perde-se toda a história do projeto

Centralized Version Control Systems III



Distributed Version Control Systems (DVCSs)

- Sistemas Distribuidos de controlo de versões (*Distributed Version Control Systems* – DVCS) dos clientes não fazem o *check* à última "imagem" do repositório mas têm um "imagem" completa do repositório (ou pelo menos de alguns ramos)
- Cada "snapshot" tem um backup completo dos dados
- Se o servidor falhar qualquer cliente pode ser copiado para o servidor
- Exemplo: Git, Mercurial, Bazaar or Darcs


Server Computer
Version Database
version 3
version 2
version 1
Computer A
file
Version Database
version 3
version 2
version 1
Computer B
file
Version Database
version 3
version 2
version 1

Definição

git

Git is an extremely fast, efficient, distributed version control system ideal for the collaborative development of software. [Git, 2012]

Git

- Git é um sistema de controle de revisão popular desenhado para ligar com projectos muito grandes com velocidade e eficiência; é usado por muitos projectos open source de alto perfil, o mais notável é o kernel do Linux.
- O Git encaixa na categoria de ferramentas de gestão de código fonte distribuído. Todos os directórios de trabalho do Git são repositórios totais com capacidades totais de rastreio de revisões, não dependentes de acesso à rede ou de um servidor central.

Git

Vantagens

- Permite fácil edição local do código, seja de programas (Java, C, C++, python, pascal, Basic,...), de LaTeX, de XML, ...
- Permite gravação local de mudanças (importante quando se está a fazer uma "mudança grande")
- Facilita edição simultânea de um mesmo arquivo
- Permite versões concorrentes de um mesmo arquivo/projeto, e.g. versão estável e versão de desenvolvimento

Git

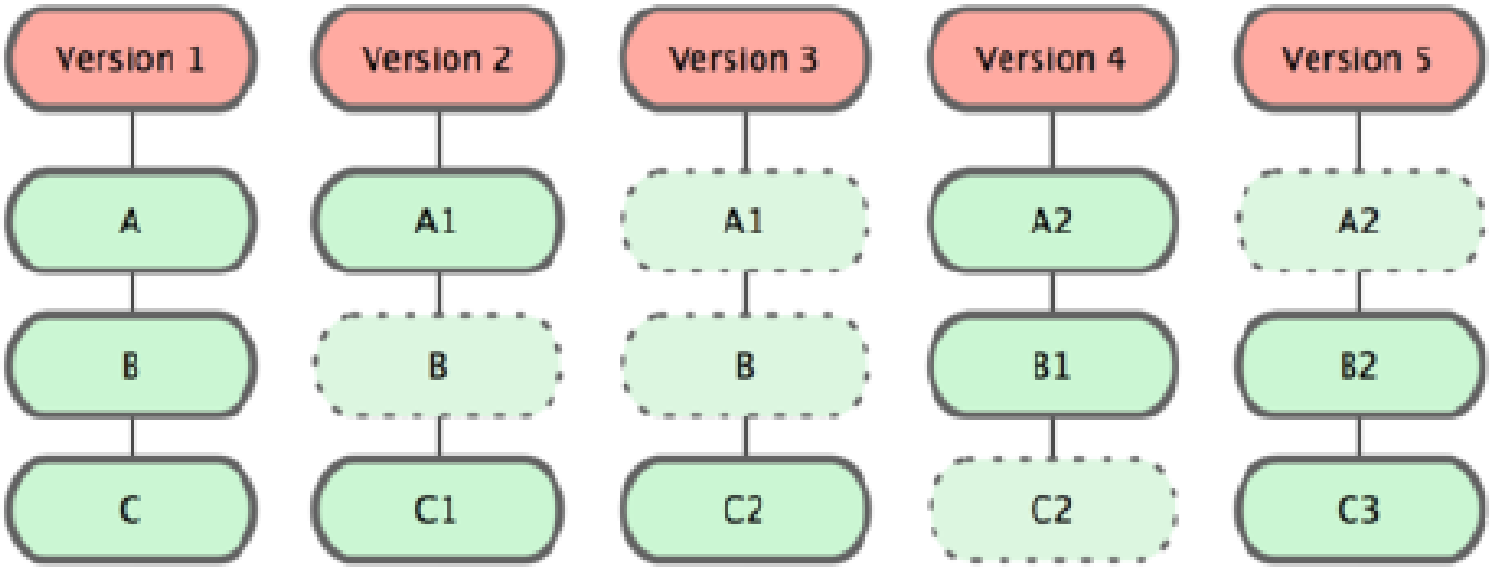
Desvantagens

- Exige algum esforço para começar a usar
- Funciona melhor para documentos de texto, i.e., formatos de arquivo binário podem ser armazenados, mas a funcionalidade do controle de versão para estes arquivos será reduzida, limitando o uso destas ferramentas quando se trabalha com arquivos de imagem ou documentos em formatos binários.

Git

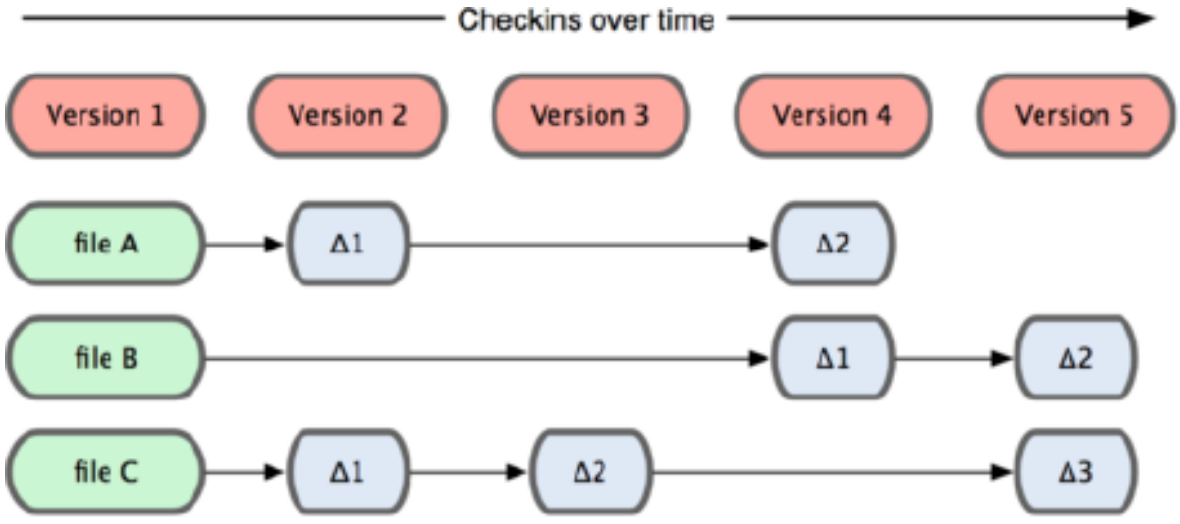
- *Git thinks of its data more like a set of snapshots of a mini filesystem.*
- *Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot.*
- *To be efficient, if files have not changed, Git doesn't store the file again—just a link to the previous identical file it has already stored.*

Git


Version 1
A
B
C
Version 2
A1
B
C1
Version 3
A1
B
C2
Version 4
A2
B1
C2
Version 5
A2
B2
C3

Git

- *Most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time*



Nearly Every Operation Is Local

- *Most operations in Git only need local files and resources to operate generally no information is needed from another computer on your network.*
- *To browse the history of the project, Git doesn't need to go out to the server to get the history and display it for you.*
- *If you want to see the changes introduced between the current version of a file and the file a month ago, Git can look up the file a month ago and do a local difference calculation,*
- *There is very little you can't do if you're offline or off VPN...*

Git Has Integrity

- *Everything in Git is check-summed before it is stored and is then referred to by that checksum – This means it's impossible to change the contents of any file or directory without Git knowing about it.*
- *You can't lose information in transit or get file corruption without Git being able to detect it.*
- *The mechanism that Git uses for this checksumming is called a SHA–1 hash. – is a 40-character string composed of hexadecimal characters (09 and af) and calculated based on the contents of a file or directory structure in Git.*

24b9da6552252987aa493b52f8696cd6d3b00373

The Three States

- *Git has three main states that your files can reside in:*

  committed – *means that the data is safely stored in your local database*

  modified – *means that you changed the file but have not committed it to your database yet*

  staged – *means that you have marked a modified file in its current version to go into your next commit snapshot.*

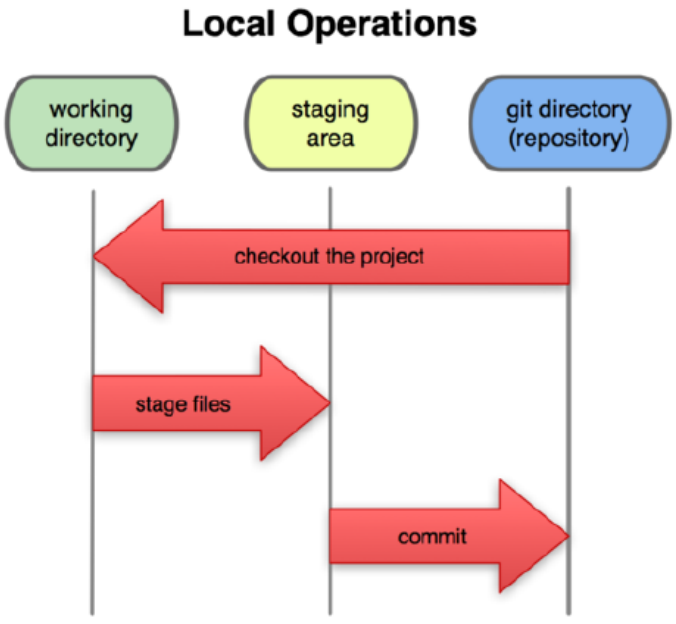
Git Areas

Git directory – *where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.*

Working directory – *is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.*

Staging area – *is a simple file, generally contained in your Git directory, that stores information about what will go into your next commit.*

The basic Git workflow



1. *You modify files in your working directory.*
2. *You stage the files, adding snapshots of them to your staging area.*
3. *You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.*

Instalação do Git

- Linux

```
> apt-get install git
```

- MS windows – instalador em https://git-scm.com/

Configurações iniciais

(start a console in Linux or git-bash in MsWindows)
Done before 1st use

- *Verify the present configurations*

```
> git config --list

color.status=auto
color.branch=auto
user.name=Pedro Cardoso
user.email=pcardoso@ualg.pt
```

Configurações iniciais

- *Sets the default name for Git to use when you commit*

```
> git config --global user.name "Pedro J.S. Cardoso"
```

- *Sets the default email for Git to use when you commit*

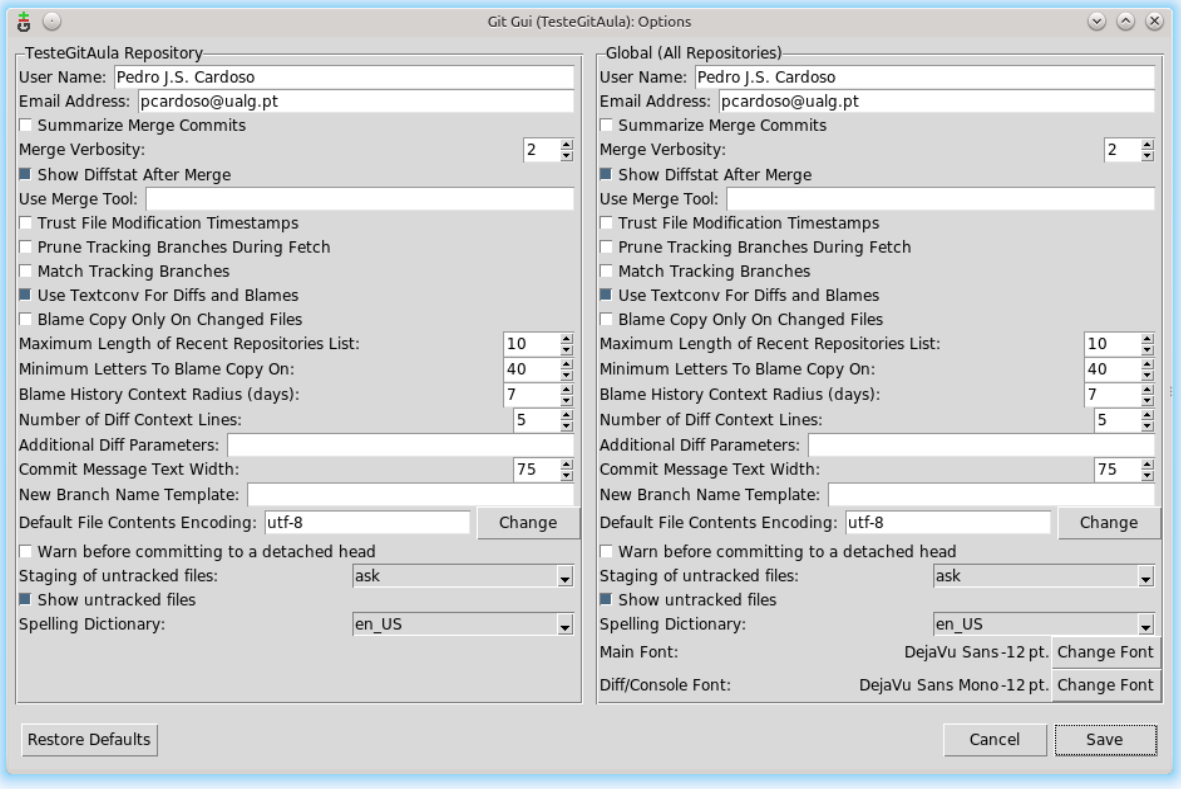
```
> git config --global user.email "pcardoso@ualg.pt"
```

- *Tell Git that you don't want to type your username and password every time you talk to a remote server.*

```
# By default git will cache your password for 15
    minutes
> git config --global credential.helper cache

# Set the cache to timeout after 1 hour
> git config --global credential.helper 'cache --
    timeout=3600'
```

Configurações iniciais - GUI

```
> git gui
```



Novo projeto

```
# Creates a directory for your project
> mkdir TPPI-RULES

# Changes the current working directory
> cd TPPI-RULES

# Sets up the necessary Git files
# Initialized empty Git repository (fazer 'ls -al' antes e
    depois)
> git init
```

Include a file with the filename “README” in your repo

It will automatically be shown on your Bitbucket/GitHub repo’s front page

```
# Create a file called "README" in your TPPI-RULES directory
# podem usar um editor de texto (ex: leafpad, vim, nano,
    ...) para criar/editar o README
> echo 'TPPI RULES!' > README
```

Latter we’ll see how to configure and send to github

Adicionar e fazer o “commit”- `git commit`

```
# Stages your README file, adding it to the list of files to
    be committed
> git add README

# Commits your files, adding the message "Primeiro commit.
    README criado"
> git commit -m "Primeiro commit. README criado"
[master (root-commit) 0ceab41] Primeiro commit. README
    criado
1 file changed, 1 insertion(+)
create mode 100644 README
```

Adicionar ficheiros a serem controlados pelo Git - `git add`

```
# Um unico ficheiro
> git add input.txt

# todos os ficheiros com a extensao txt ...
> git add *.txt

# todos os ficheiros comecados por 'inp'
> git add inp*

# todos os ficheiros na diretoria
> git add .
```

Estado atual - `git status`

```
# Cria ficheiros (vazios)
> touch input1.txt
> touch input2.txt

# Adiciona para commit todos os ficheiros comecados por inp
> git add inp*

# estado atual
> git status
    On branch master
    Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)
  new file:   input1.txt
  new file:   input2.txt

> git commit -m 'adicionados ficheiros de input'
    [master 46287d8] adicionados ficheiros de input
    0 files changed
    create mode 100644 input1.txt
    create mode 100644 input2.txt
```

Estado atual - `git status`

```
# Altera ficheiros
> echo '1 2 3' > input1.txt

> git status
    On branch master
    Your branch is ahead of 'origin/master' by 1 commit.

    Changes not staged for commit:
    (use "git add <file>..." to update what will be
        committed)
    (use "git checkout -- <file>..." to discard changes in
        working directory)

    modified:   input1.txt

    no changes added to commit (use "git add" and/or "git
        commit -a")
```

Estado atual - `git status`

```
> git add .

> git status
  ...

 git commit -m 'alterados os dados no  ficheiros input1.txt'
   [master 9832dfe] alterados os dados no  ficheiros input1.
       txt
   1 file changed, 1 insertion(+)
```

Ignorar ficheiros - `.gitignore`

```
> mkdir TODO
> cd TODO/
> touch lista-atividades.txt
> cd ..

> git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.

Untracked files:
(use "git add <file>..." to include in what will be
    committed)

    TODO/
nothing added to commit but untracked files present (use "
    git add" to track)
```

Ignorar ficheiros - `.gitignore`

Podemos criar uma lista de ficheiros que serão ignorados pelo git. Os ficheiros a serem ignorados são colocados numa lista que é guardada num ficheiro chamado `.gitignore`. Exemplo:

```
# no xxxxx.o files
*.o
# no xxxxx~ files
*~
# but do track lib.o
!lib.o
# ignore all files in the TODO/ directory
TODO/*
# ignore doc/notes.txt, but not doc/server/arch.txt
doc/*.txt
```

Alterações num ficheiro - `git diff`

```
> ls -a
.   ..   .git   .gitignore   input1.txt   input2.txt   README
    TODO   tppi.cpp
> git status
    ...

    nothing to commit (working directory clean)
```

Alterações num ficheiro - `git diff`

Alterando o ficheiro input2.txt para conter os seguintes dados

```
a
b
c
```

Alterações num ficheiro - `git diff`

```
# Qual o estado?
> git status
...
    modified:   input2.txt
...
# Quais as modificacoes?
> git diff
index e69de29..1c943a9 100644
--- a/input2.txt
+++ b/input2.txt
@@ -0,0 +1,3 @@
+a
+b
+c
\ No newline at end of file

#Compares what is in your working directory with what is
#in your staging area.
#The result tells you the changes you've made that you
#haven't yet staged.
```

Alterações num ficheiro - `git diff`

```
> git diff --staged
> git add input2.txt
> git diff
> git diff --staged
    diff --git a/input2.txt b/input2.txt
    index e69de29..1c943a9 100644
    --- a/input2.txt
    +++ b/input2.txt
    @@ -0,0 +1,3 @@
    +a
    +b
    +c
    \ No newline at end of file

   #If you want to see what you've staged that will go into
   #your next commit, you can use git diff --staged
```

Commit sem passar pela staging area - `git commit -a -m ...`

```
> echo 'd' >> input2.txt

# flag -a
> git commit -a -m 'Alteracao dos dados em input2.txt'
    [master f63a3d4] Alteracao dos dados em imput2.xtt
    1 file changed, 1 insertion(+), 1 deletion(-)
```

Remoção de ficheiros - `git rm / mv ...`

```
> rm input2.txt
> git status
    On branch master
    Your branch is ahead of 'origin/master' by 6 commits.

    Changes not staged for commit:
    (use "git add/rm <file>..." to update what will be
        committed)
    (use "git checkout -- <file>..." to discard changes in
        working directory)

  deleted:    input2.txt

    no changes added to commit (use "git add" and/or "git
        commit -a")

> git commit -a -m 'input2.txt foi apagado'
    [master 5bfbe6f] input2.txt foi apagado
    1 file changed, 3 deletions(-)
    delete mode 100644 input2.txt
```

Mover ou renomear ficheiros - `git rm / mv ...`

```
> cp input1.txt input2.txt
> git rm input1.txt
> git add input2.txt

# de modo equivalente
> git mv input1.txt input2.txt


# Git doesn't explicitly track file movement.
# If you rename a file in Git,
# no metadata is stored in Git that tells it you renamed the
    file.


> git commit -a -m 'moved input1.txt para input2.txt'
```

Histórico de Commits - `git log`

```
> git log
    commit a72f0798f20e92c319632fa25da0a6a0b50f1001
    Author: Pedro Cardoso <pcardoso@ualg.pt>
    Date:   Wed Oct 10 19:20:21 2012 +0100

    mv input1.txt para input2.txt
    (...)
    commit fab448fc1a316348cd560d43c1ff0849de09cebf
    Author: Pedro J.S. Cardoso <pcardoso@ualg.pt>
    Date:   Wed Oct 10 10:57:51 2012 +0100

    Criado o codigo para escrever a mensagem TPPI RULES! em
         tppi.cpp

    commit e9aee4f475b75fee0e8cf578a138ecc65bba54ef
    Author: Pedro J.S. Cardoso <pcardoso@ualg.pt>
    Date:   Wed Oct 10 10:50:58 2012 +0100

    Primeiro commit. README criado
```

Histórico de Commits - `git log`

```
# para ver as alteracoes em cada commit
> git log -p

# para ver as alteracoes da ultima entrada
> git log -p -1

# para ver as alteracoes das ultimas  2 entradas
> git log -p -2

# sumario de alteracoes
> git log --stat -1

# Lista de commits -- 1 por linha
> git log --pretty=oneline

# para ver as alteracoes das ultimas  2 horas
#(pode usar weeks, ... -- ver manual)
> git log --pretty=format:"%h - %an, %ar : %s" --since=2.
    hours

#outros limitadores: until, author, after, ...
```

Histórico de Commits - `git log`

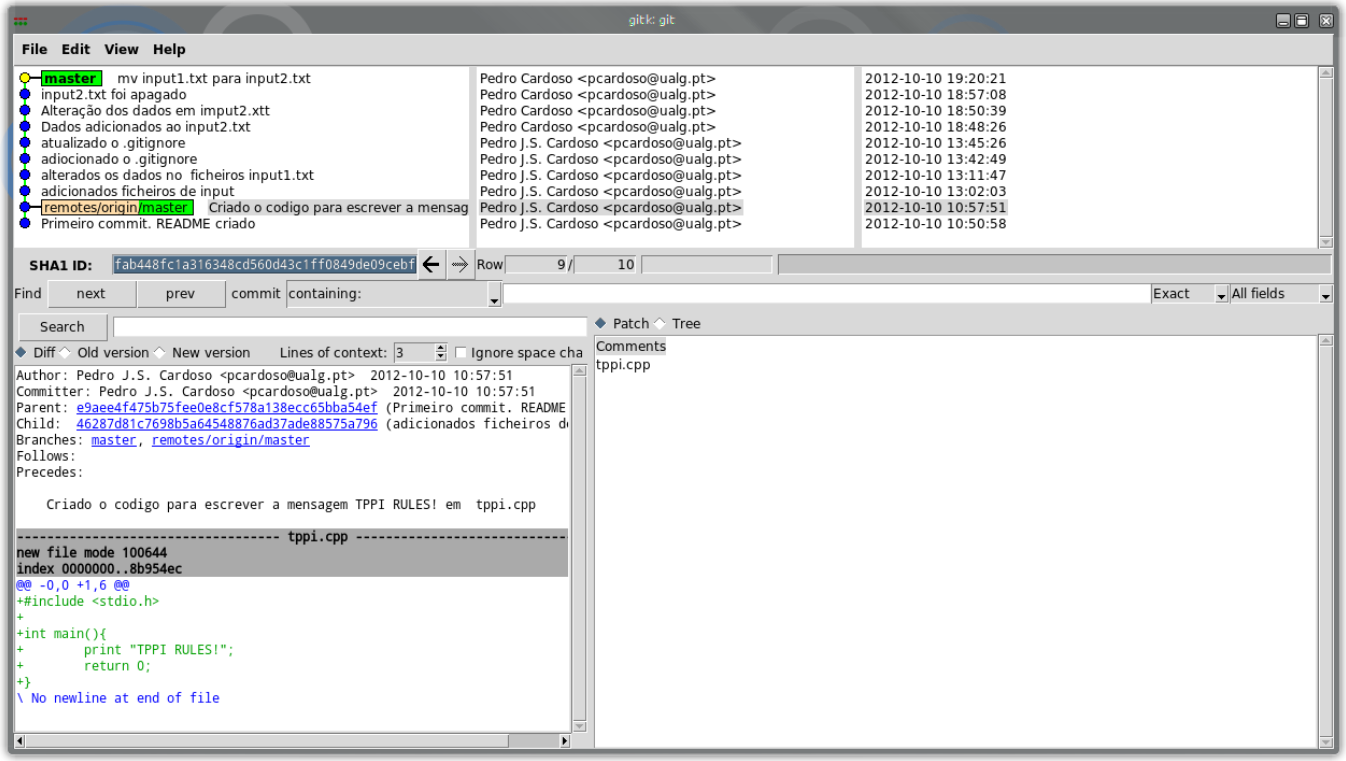
```
> git log --pretty=format:"%h - %an, %ar : %s"
a72f079 - Pedro Cardoso, 11 minutes ago : mv input1.txt para
     input2.txt
5bfbe6f - Pedro Cardoso, 35 minutes ago : input2.txt foi
    apagado
f63a3d4 - Pedro Cardoso, 41 minutes ago : Alteracao dos
    dados em imput2.xtt
77fac36 - Pedro Cardoso, 43 minutes ago : Dados adicionados
    ao input2.txt
(...)
fab448f - Pedro J.S. Cardoso, 9 hours ago : Criado o codigo
    para escrever a mensagem TPPI RULES! em  tppi.cpp
e9aee4f - Pedro J.S. Cardoso, 9 hours ago : Primeiro commit.
     README criado
#ver como formatar no manual
```

Histórico de Commits - `git log`

```
# complicando...
> git log --pretty="%h: %an(%ae): %ar: %s" --author="Pedro
    Cardoso"  --before=1.hour

    a72f079: Pedro Cardoso(pcardoso@ualg.pt): 75 minutes ago
        : mv input1.txt para input2.txt
    5bfbe6f: Pedro Cardoso(pcardoso@ualg.pt): 2 hours ago:
        input2.txt foi apagado
    f63a3d4: Pedro Cardoso(pcardoso@ualg.pt): 2 hours ago:
        Alteracao dos dados em imput2.xtt
    77fac36: Pedro Cardoso(pcardoso@ualg.pt): 2 hours ago:
        Dados adicionados ao input2.txt
```

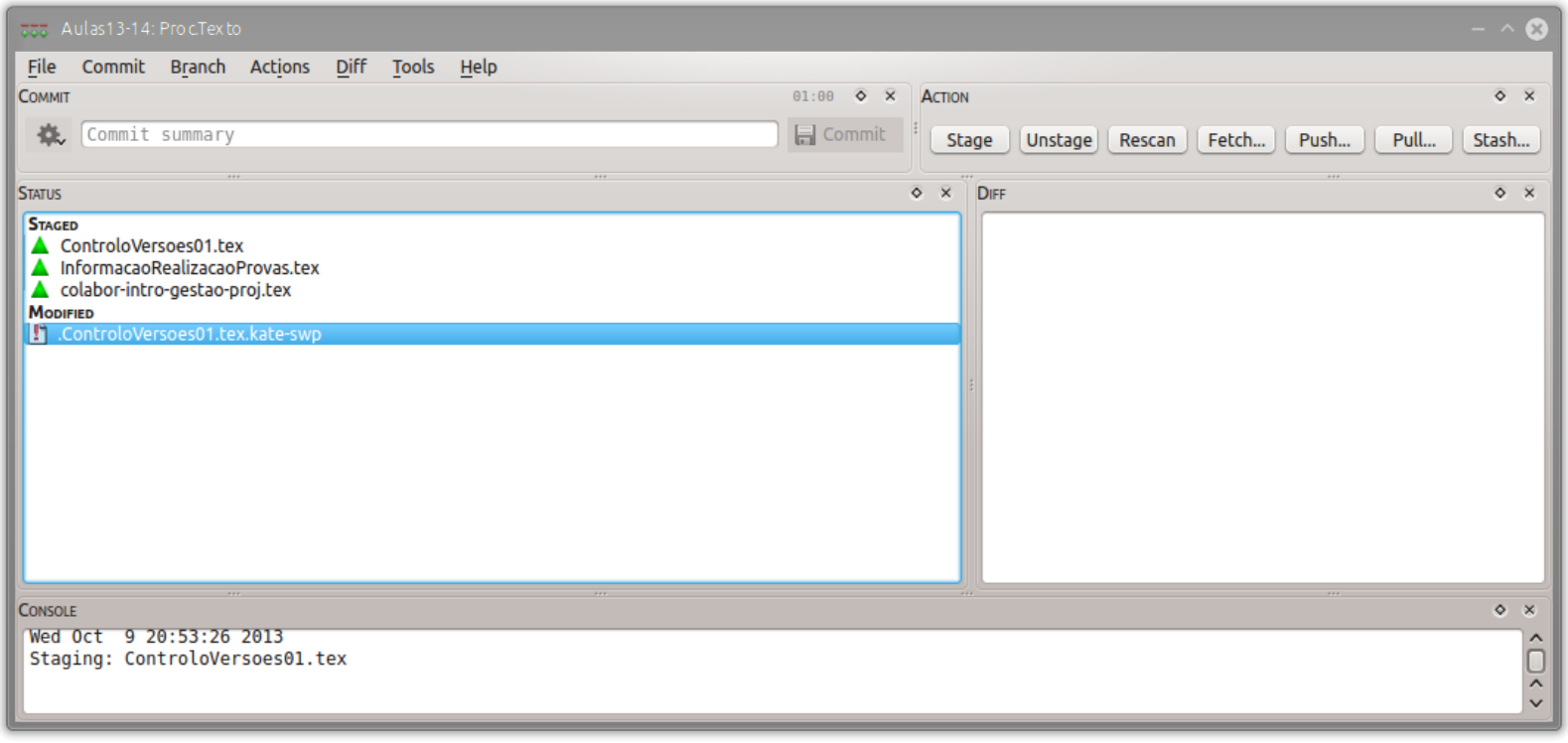
Histórico em modo gráfico - gitk

Podemos usar o gitk para ter um “log” em modo gráfico



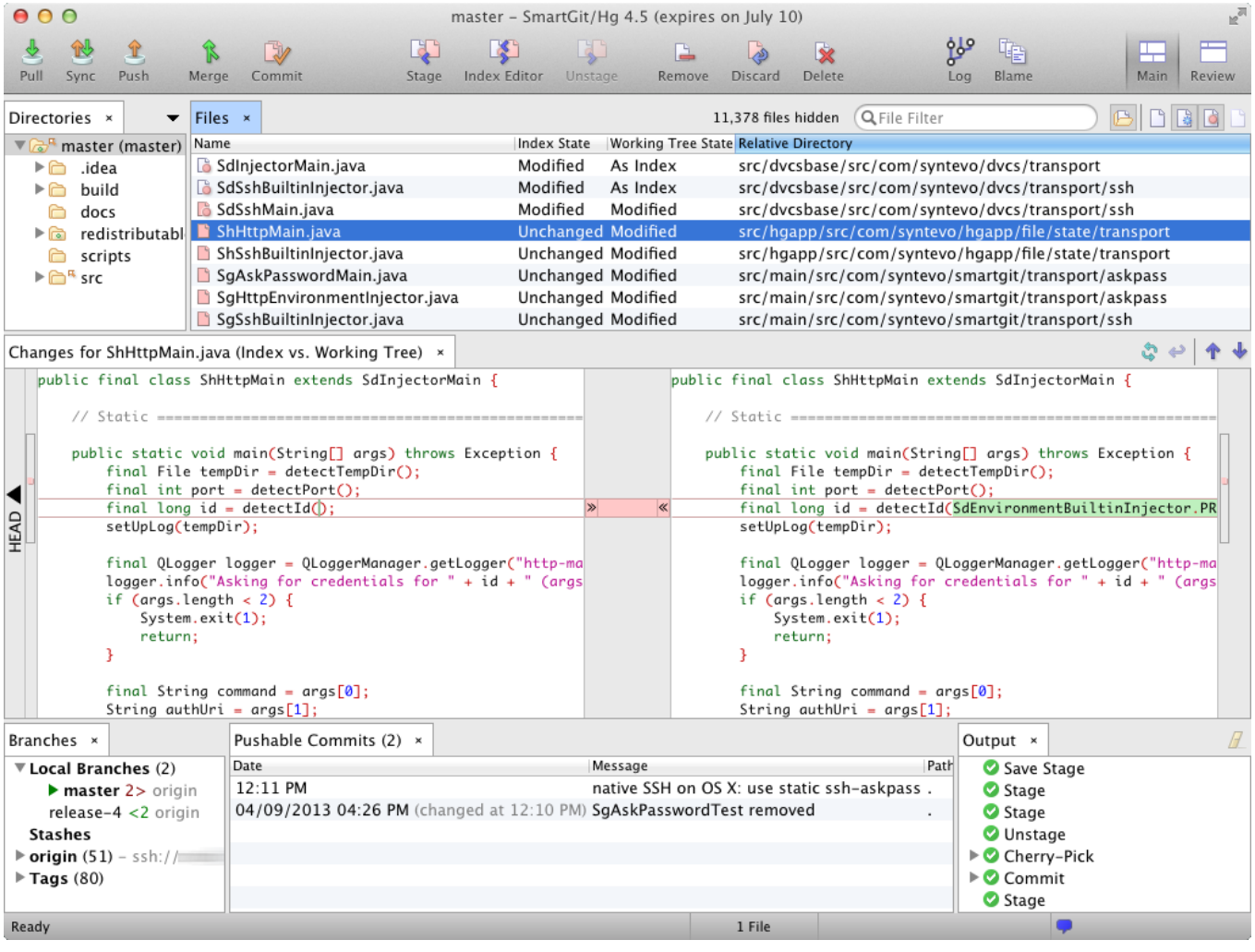
Histórico em modo gráfico - git-cola

Vários interfaces gráficos para os SOs mais comuns



Histórico em modo gráfico - smartgit

Vários interfaces gráficos para os SOs mais comuns



Mudar o último commit

```
# acrescentar um ficheiro ao ultimo commit
> touch input3.txt
> git add input3.txt
> git commit --amend
# editar o texto do commit:
#move input1.txt para input2.txt e adicao do ficheiro input3
    .txt


#podemos tambem simplesmente alterar o texto do ultimo
    commit
> git commit --amend
```

“Unstaging” um ficheiro

```
> touch input4.txt
> git add input4.txt
> git status
    On branch master
    Your branch is ahead of 'origin/master' by 8 commits.
    Changes to be committed:
    (use "git reset HEAD <file>..." to unstage)
    new file:   input4.txt

> git reset HEAD input4.txt
> git status
    On branch master
    Your branch is ahead of 'origin/master' by 8 commits.
    Untracked files:
    (use "git add <file>..." to include in what will be
        committed)
       input4.txt
    nothing added to commit but untracked files present (use
         "git add" to track)
```

4ª e 5ª Aula

Ferramentas colaborativas – Controladores de Versões(cont.)

Branching

- Branching means you diverge from the main line of development and continue to do work without messing with that main line.
- Git encourages a workflow that branches and merges often, even multiple times in a day.

Branching

- When you commit in Git, Git stores a commit object that contains a pointer to the snapshot of the content you staged, the author and message metadata, and zero or more pointers to the commit or commits that were the direct parents of this commit:
  - zero parents for the first commit,
  - one parent for a normal commit,
  - multiple parents for a commit that results from a merge of two or more branches.

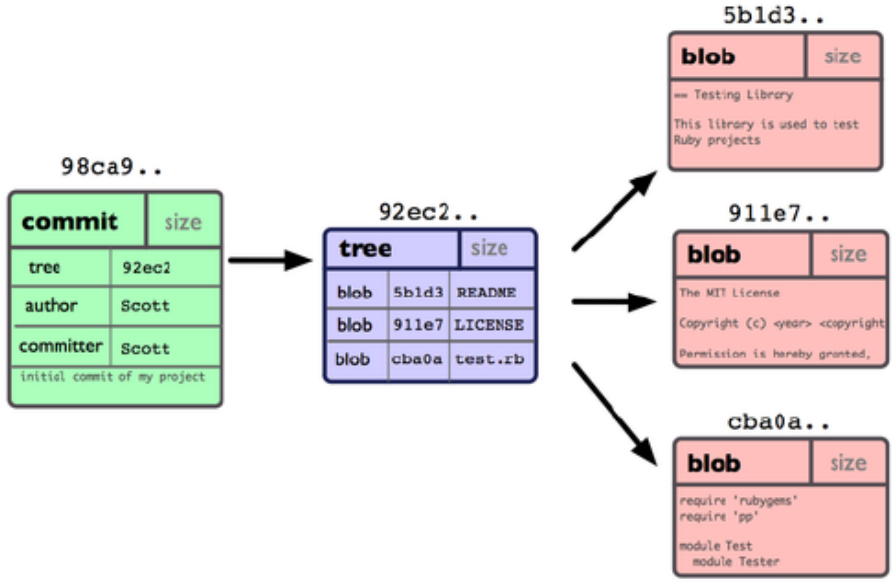
Branching

- Let's assume that you have a directory containing three files, and you stage them all and commit.

```
# create files (empty)
> touch README teste.rb LICENSE2

# Staging the files checksums each one, stores that
    version of the file in the Git repository (Git
    refers to them as blobs), and adds that checksum to
    the staging area
> git add README teste.rb LICENSE2

# When you create the commit (git commit), Git
    checksums each subdirectory and stores those tree
    objects in the Git repository. Git then creates a
    commit object that has the metadata and a pointer to
     the root project tree so it can re-create that
    snapshot when needed.

> git commit -m 'initialization of my project'
```

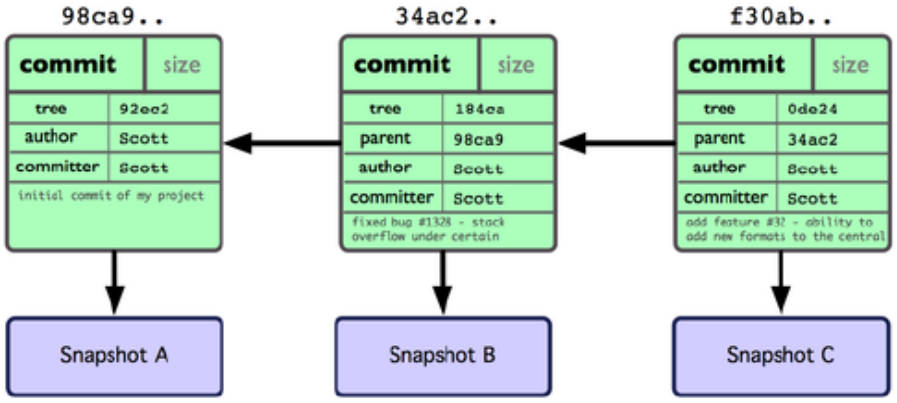
Branching

- Your Git repository now contains five objects: one blob for the contents of each of your three files, one tree that lists the contents of the directory and specifies which file names are stored as which blobs, and one commit with the pointer to that root tree and all the commit metadata.



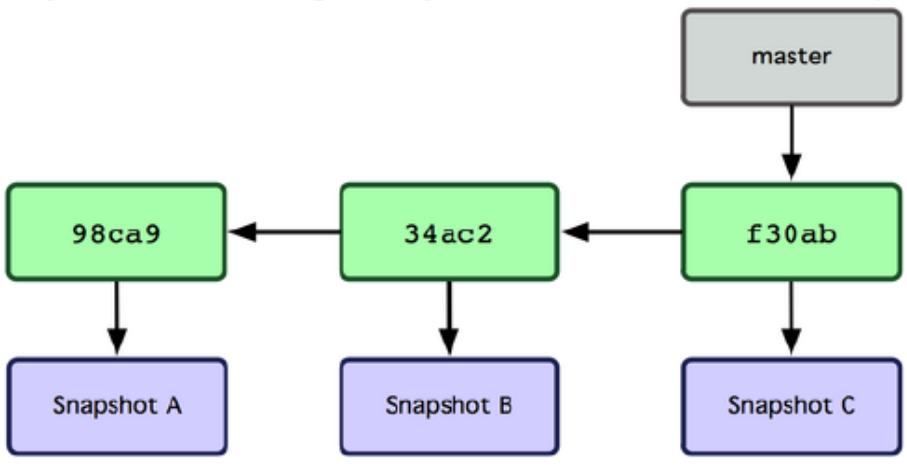
Branching

- If you make some changes and commit again, the next commit stores a pointer to the commit that came immediately before it.



Branching – `git branch`

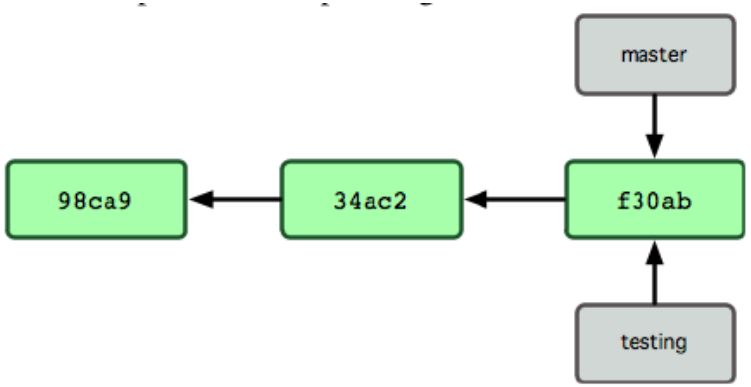
- A branch in Git is simply a lightweight movable pointer to one of these commits.
- The default branch name in Git is master.



Branching – `git branch`

- What happens if you create a new branch? Well, doing so creates a new pointer for you to move around.

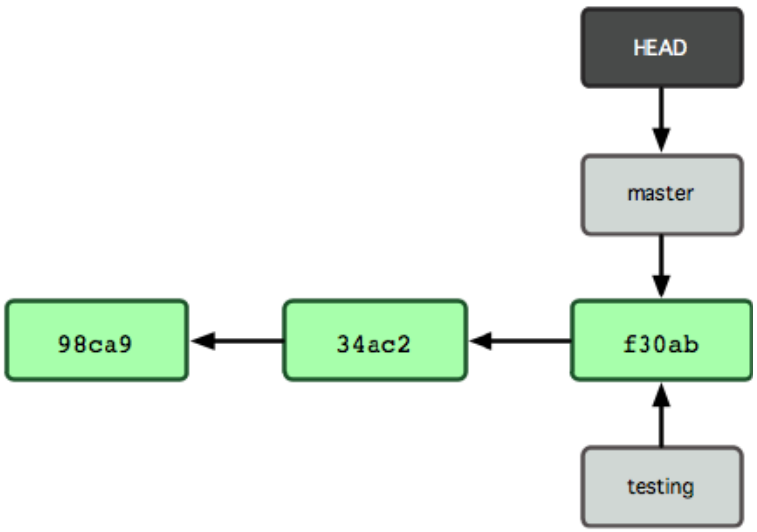
```
#  Let's create a new branch called testing.
> git branch testing

# This creates a new pointer at the same commit you're
    currently
```



Branching – `git branch`

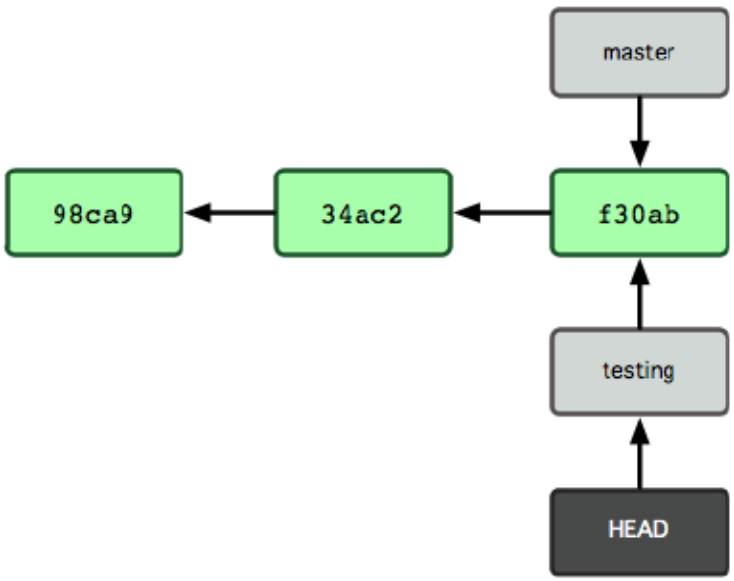
- How does Git know what branch you're currently on? It keeps a special pointer called HEAD.

```
#list branches
> git branch
* master
testing
```



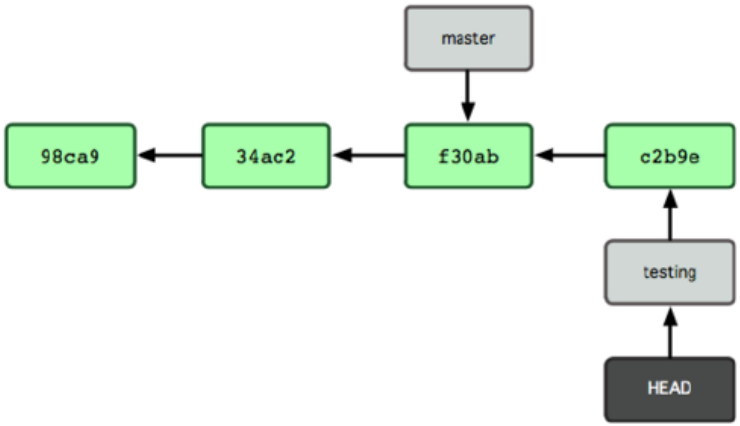
Branching – `git checkout`

```
# To switch to an existing branch, you run the git checkout
    command.
# Let's switch to the new testing branch
> git checkout testing
```



Branching – `git checkout`

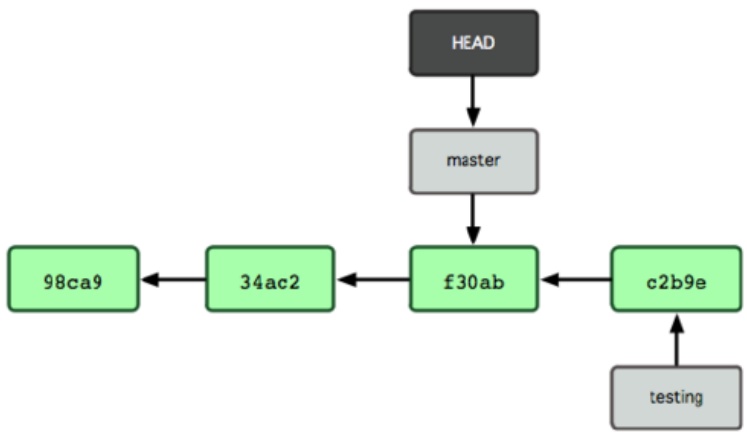
```
# modificar os dados de um dos ficheiros
> nano teste.rb

> git commit -a -m 'made a change on teste.rb'
```



Branching – `git checkout`

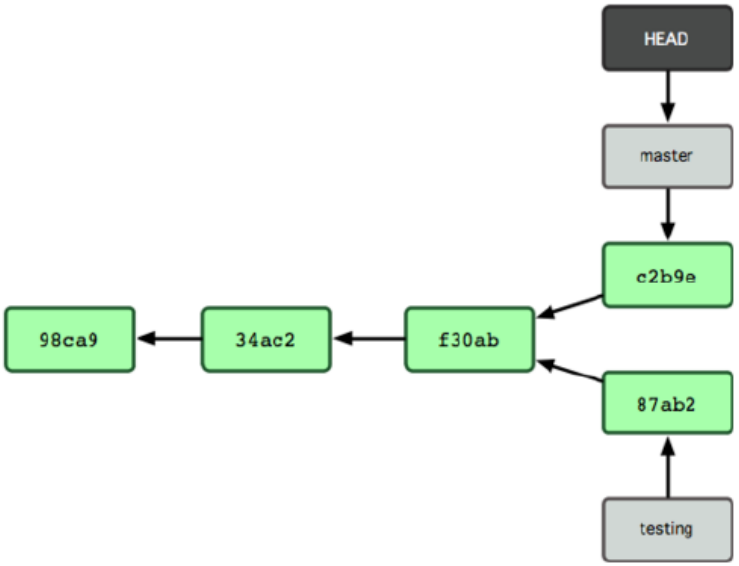
```
#  Let's switch back to the master branch:
> git checkout master
```

- That command did two things:
  - moved the HEAD pointer back to point to the master branch,
  - reverted the files in your working directory back to the snapshot that master points to.



Branching – `git checkout`

```
# modificar os dados do ficheiro teste.rb --- possiveis
    conflitos
> nano teste.rb

> git commit -a -m 'made other changes on teste.rb'
```

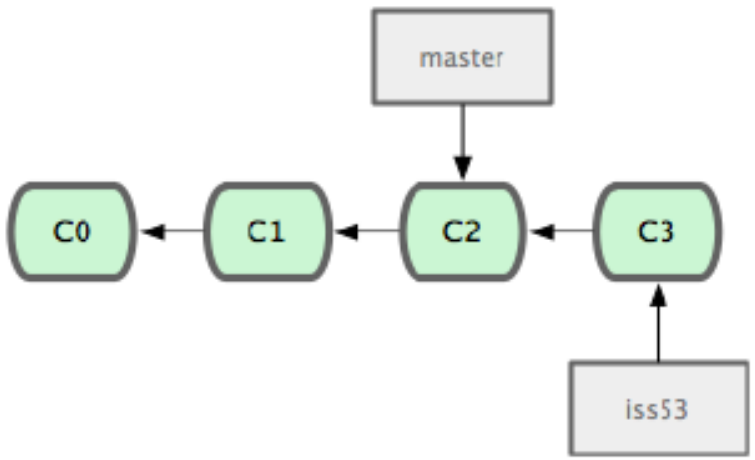


Branching – outro exemplo

- Vejamos outro exemplo:
  - Estamos a trabalhar num website
  - Criar um ramo/“branch” para um assunto/problema/“issue” em que estamos a trabalhar
  - Fazemos algum trabalho nesse ramo
  - Nesta altura recebemos uma chamada para resolver um problema crítico
  - Volto ao ramo de produção
  - Crio um ramo para adicionar o hotfix
  - Após testes faço o merge e envio para produção
  - Volto ao ramo em que estava a trabalhar e contínuo...

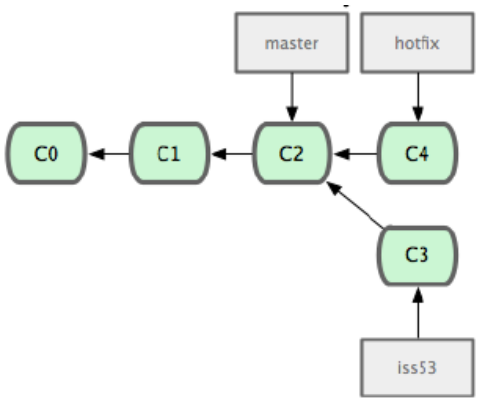
Branching

```
#.... em qualquer altura do projeto ....

# -b: cria e muda para o novo ramo
> git checkout -b iss53
Switched to a new branch "iss53"

#estou a trabalhar num dado ficheiro
> nano index.html
> git commit -a -m 'added a new footer [issue 53]'
```



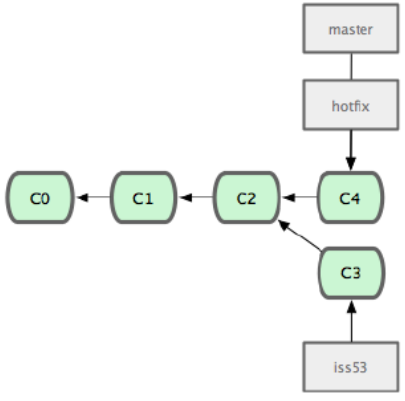
Branching

```
#.... e agora a chamada ....
> git checkout master
Switched to branch "master"

> git checkout -b 'hotfix'
Switched to a new branch "hotfix"

#correcao do erro
> nano index.html
> git commit -a -m 'fixed the broken email address'
[hotfix]: created 3a0874c: "fixed the broken email address"
1 files changed, 0 insertions(+), 1 deletions(-)
```



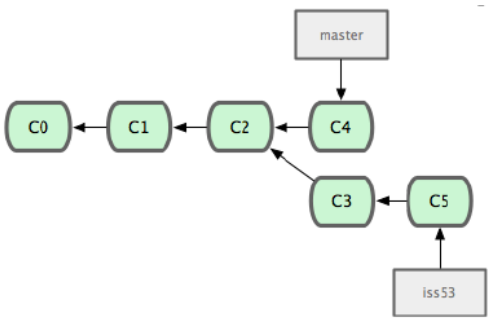
Branching

```
# Feitos os testes podemos fazer o merge para o ramo 'master
    ' e enviar para producao
> git checkout master
> git merge hotfix
Updating f42c576..3a0874c
Fast forward
README |
1 -
1 files changed, 0 insertions(+), 1 deletions(-)

> git push .....
```



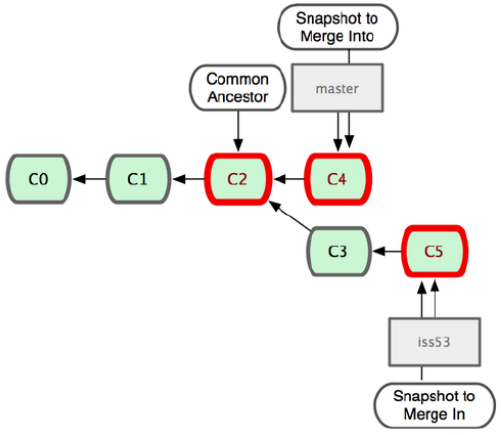
Branching

```
# podemos agora apagar o ramo hotfix
> git branch -d hotfix
Deleted branch hotfix (3a0874c).

#continuar no trabalho anterior
> git checkout iss53
Switched to branch "iss53"

> nano index.html
> git commit -a -m "finished the new footer [issue 53]"
[iss53]: created ad82d7a: "finished the new footer [issue
    53]"
1 files changed, 1 insertions(+), 0 deletions(-)
```



Branching

```
# o trabalho feito no hotfix nao esta no branch iss53
# podiamos ter feito o merge do master com o iss53 ('git
    merge master')
# mas vamos fazer o merge no final de resolver o 'iss53'
> git checkout master
> git merge iss53
Merge made by recursive.
README |
1 +
1 files changed, 1 insertions(+), 0 deletions(-)
```



Branching

```
#  Se alterarmos as mesmas partes dos mesmos ficheiros pode
    nao correr de forma tao 'suave'
> git merge iss53
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the
    result.

[master*]> git status
index.html: needs merge
On branch master
Changed but not updated:

(use "git add <file>..." to update what will be committed)

(use "git checkout -- <file>..." to discard changes in
    working directory)

unmerged:
index.html
```

Branching

```
<<<<<<< HEAD:index.html
<div id="footer">contact : email.support@github.com</div>
=======
<div id="footer">
please contact us at support@github.com
</div>
>>>>>>>iss53:index.html
```

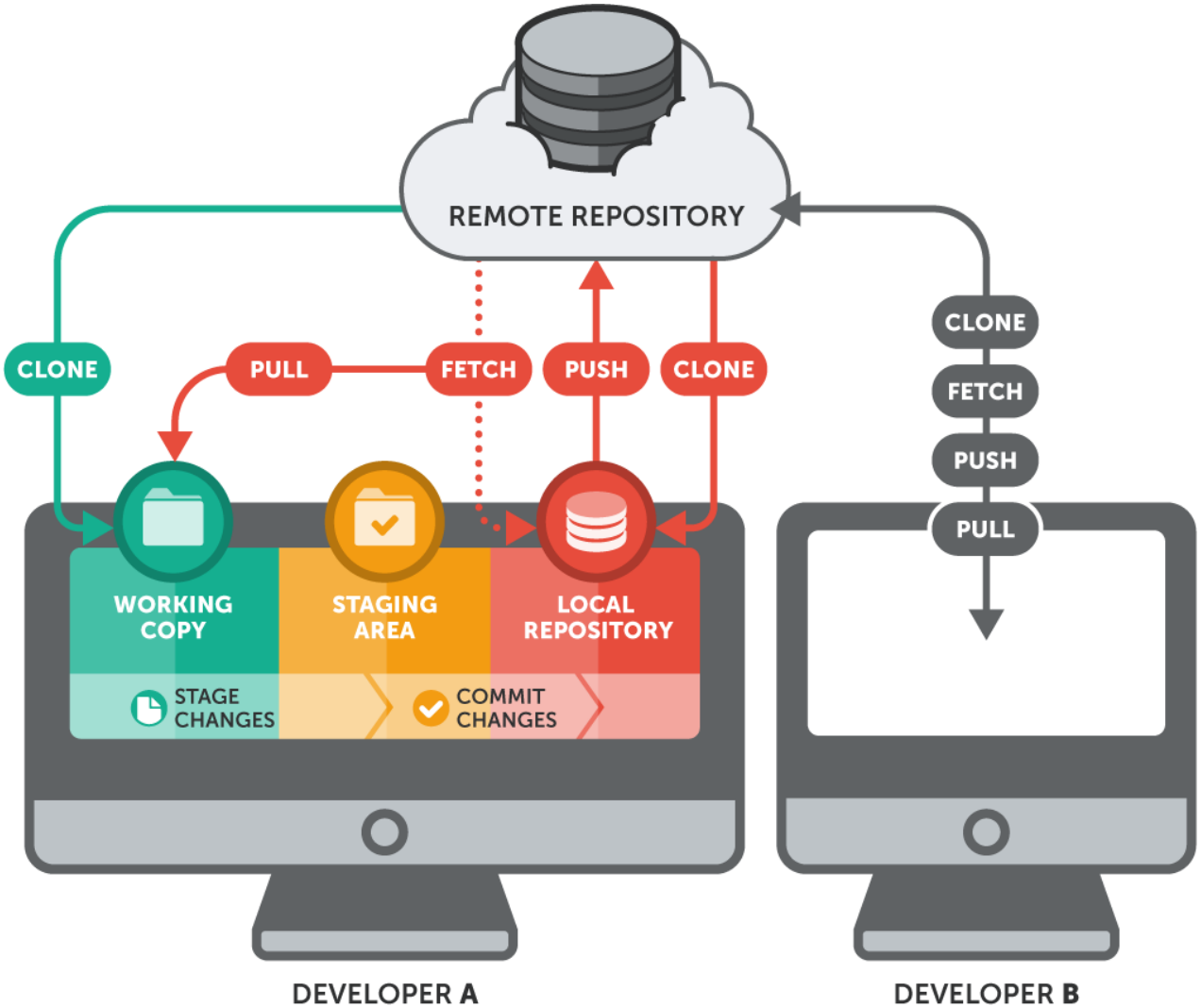
Branching

```
> git mergetool
```

```
merge tool candidates: kdiff3 tkdiff xxdiff meld gvimdiff
    opendiff emerge vimdiff
Merging the files: index.html
Normal merge conflict for 'index.html':
{local}: modified
{remote}: modified
Hit return to start merge resolution tool (opendiff):
```

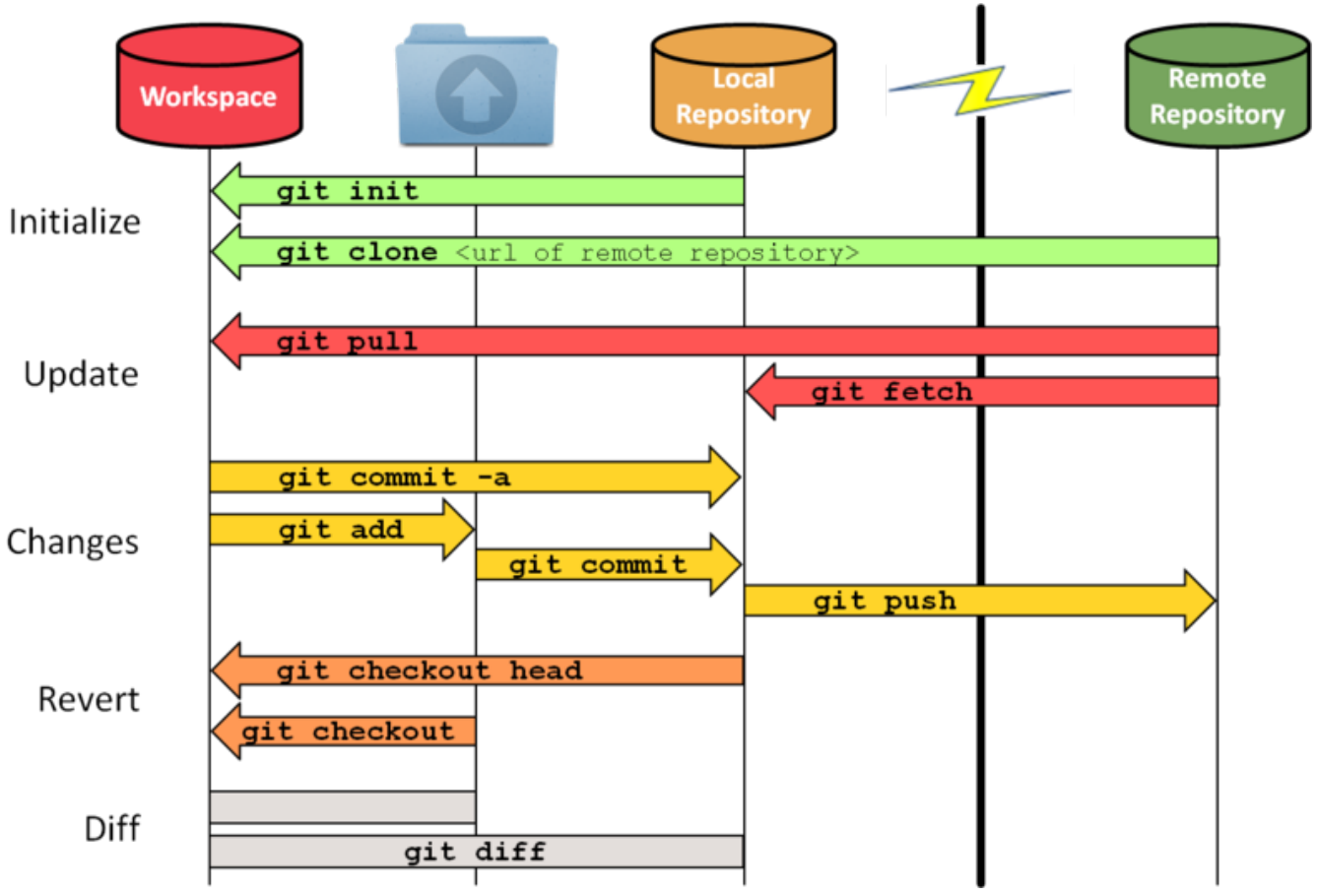
Servidor remoto

- Workflow com servidor



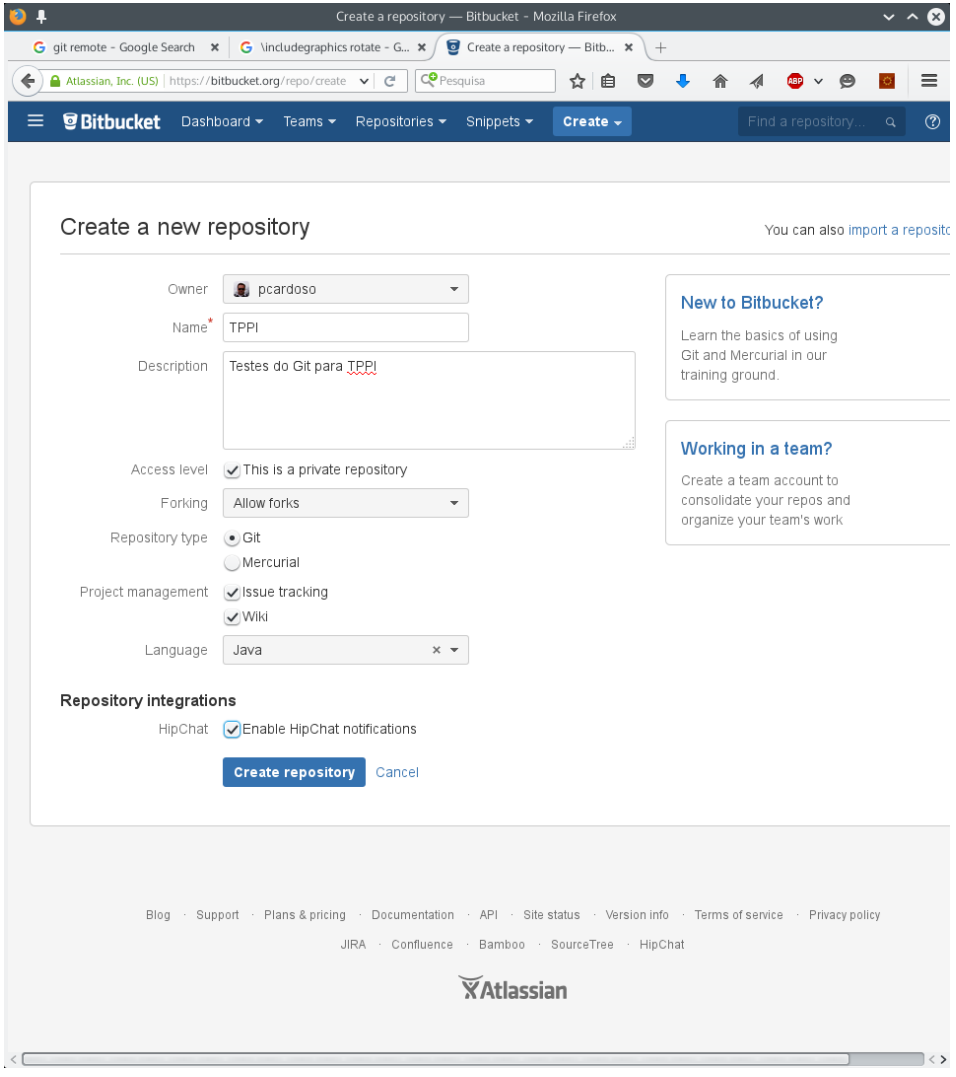
Servidor remoto

- Workflow com servidor



Servidor remoto

- Comece por criar uma conta no bitbucket (https://bitbucket.com/).
- Faça login
- Crie um novo repositório (`Create-Create repository`)



Servidor remoto

Agora é necessário configurar um repositório local para sincronizar com o do servidor remoto. Caso seja um **repositório novo**:

```
# Set up your local directory
# Set up Git on your machine if you haven't already.
> mkdir /path/to/your/project
> cd /path/to/your/project
> git init
> git remote add origin git@bitbucket.org:pcardoso/tppi.git

# Create your first file, commit, and push
> echo "Pedro Cardoso" >> contributors.txt
> git add contributors.txt
> git commit -m 'Initial commit with contributors'
> git push -u origin master
```

Servidor remoto

Agora é necessário configurar um repositório local para sincronizar com o do servidor remoto. Caso seja um **repositório existente localmente**:

```
#Already have a Git repository on your computer? Let's push
    it up to Bitbucket.
> cd /path/to/my/repo
> git remote add origin git@bitbucket.org:pcardoso/tppi.git

# pushes up the repo and its refs for the first time
> git push -u origin --all

# pushes up any tags
> git push -u origin --tags
```

Servidor remoto – `git clone`

Agora é necessário configurar um repositório local para sincronizar com o do servidor remoto. Caso seja um **repositório existente no servidor**:

```
git clone git@bitbucket.org:pcardoso/tppi.git
```

Servidor remoto - configurar SSH para o Git

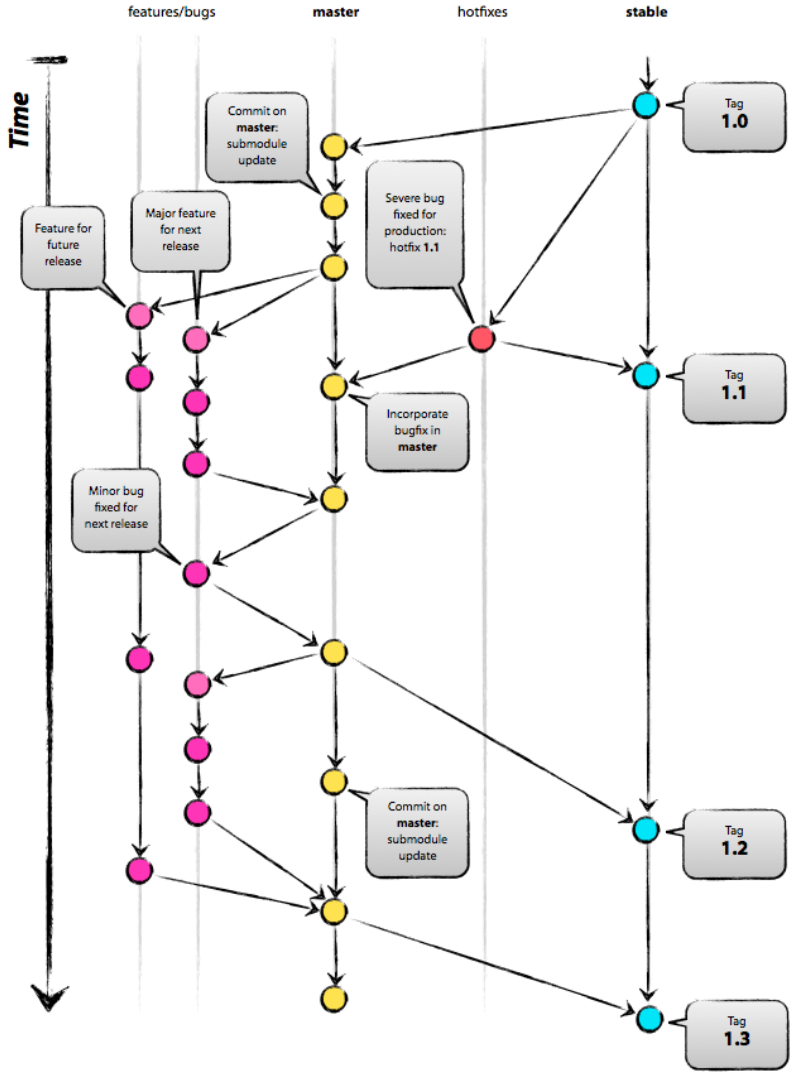
- Se usar HTTPS a autenticação é feita (com username e password) de cada vez que se faz uma ação que requere a ligação ao Bitbucket.
- O uso de *secure shell* (SSH) para comunicar com o servidor do Bitbucket server evita a introdução de credências de cada vez que se quer comunicar com o servidor.
- Para usar SSH
  - Cria-se uma *SSH identity* que consiste numa chave pública e numa chave privada, formando um *key pair*.
  - A chave privada fica no computador local e a pública é carregada para a conta do Bitbucket.
  - Depois de carregada a chave pública, pode-se usar SSH para ligar aos repositórios que o utilizador tem permissões.

Servidor remoto - configurar SSH para o Git

Configurar SSH para o Git:

https://confluence.atlassian.com/bitbucket/set-up-ssh-for-git-728138079.html

Fluxo global recomendado do Git



A estudar

Capítulos 1 a 5 do Manual do Git [Chacon and Straub, 2015]

No mínimo deve ser capaz de responder/fazer... (no git) I

- O que é um “Local Version Control Systems”?
- O que é um “Centralized Version Control Systems”?
- O que é um “Distributed Version Control Systems (DVCSs)”?
- Quais as vantagens e desvantagens de cada uma das soluções anteriores?
- Em qual das categorias anteriores se enquadra o git? porquê?
- Porque é que o git funciona como “snapshots” de um mini sistema de ficheiros?
- Porque é que não pode ocorrer perda ou corrompimento de informação nas transferências com o git?
- Quais são os 3 estados em que um ficheiro se pode encontrar?
- O que é a “git directory”?
- O que é a “working directory”?
- O que é a “staging area”?
- Fazer as configurações iniciais do git.
- Iniciar um repositório em git.
- Adicionar/remover um ficheiro ao repositório,
- Fazer um “commit”.
- Saber para que serve o ficheiro README (github, ...)

No mínimo deve ser capaz de responder/fazer... (no git) II

- Saber adicionar um repositório remoto.
- Fazer o clone de um repositório remoto.
- Enviar as modificações para o repositório remoto.
- Buscar as modificações do repositório remoto.
- Criar e configurar o ficheiro .gitignore.
- Saber qual o estado atual.
- Listar as diferenças entre um ficheiro alterado e a versão que já sofreu commit.
- Fazer um histórico dos commits.
- Fazer um histórico dos commits realizados desde uma dada data.
- Alterar o último commit.
- Criar novos ramos/branchs
- Mudar de ramo/branch
- Fazer o merge de ramos
- Usar o git-cola para fazer as “operações básicas” (começar por instalar no MS Windows e no Linux)
- Usar o gitk

Devo ter ideia de... I

- O que é um controlador de versões?
- Para que serve um controlador de versões?
- Exemplos de controladores de versões.
- Como instalar o git em ambiente Windows e Linux.
- Criar uma conta no github
- Para que serve o ficheiro .gitignore?
- O que é o SHA–1 hash?
- Ter uma ideia para que serve o ficheiro README e quais as formatações possíveis.

Chacon, S. and Straub, B. (2015).

Pro Git.

2nd edition.

Git (2012).

Git.

http://git-scm.com/.

VirtualBox (2012).

Documentação do virtualbox.

https://www.virtualbox.org/manual/.