

Controlo de Versões – Git

Controlo de Versões – o que é?

Consiste em manter o registo de todas as alterações de cada ficheiro, ou conjunto de ficheiros, ao longo do tempo de modo a ser possível recuperar uma qualquer versão pela qual cada ficheiro tenha passado.

Um sistema de controlo de versões – SCV (*version control system* – VCS) permite repor os ficheiros para estados anteriores, fazer revisões de alterações e ver quem alterou o quê e quando.

É uma ferramenta essencial para suportar o trabalho em equipa.

É uma ferramenta essencial para evitar perder trabalho.

Garante que uma versão é sempre “um todo consistente”.

Controlo de Versões – o que **não** é?

Many people's version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they're clever).

This approach is very common because it is so simple.

But it is incredibly error prone.

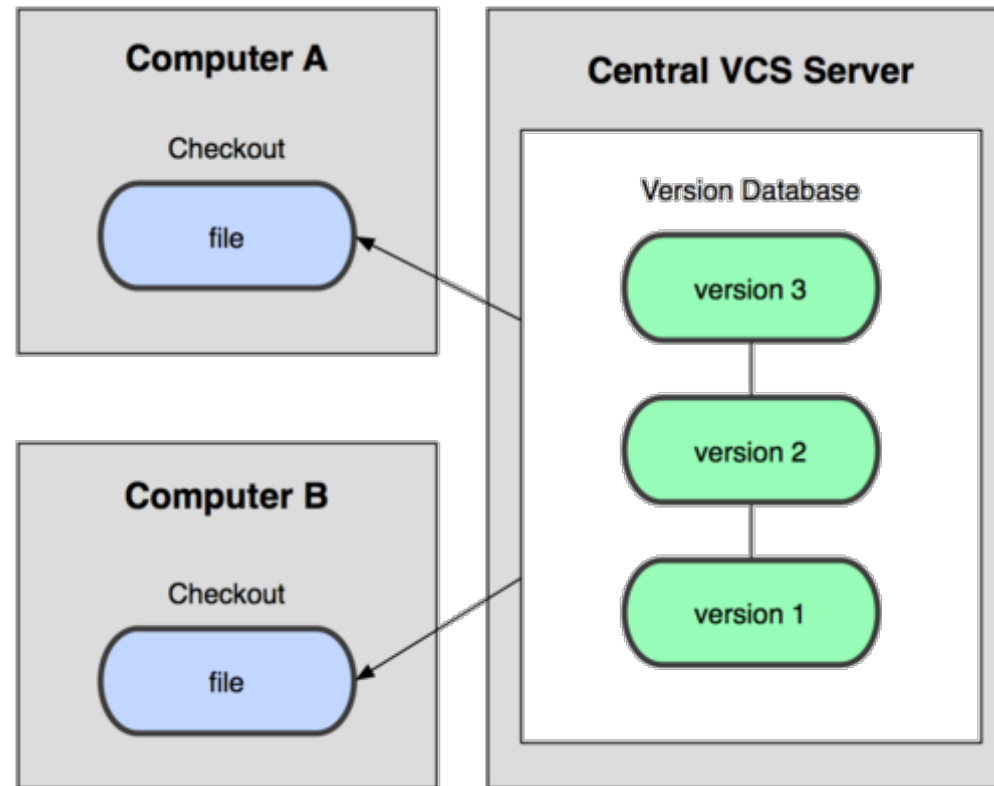
It is easy to forget which directory you're in and accidentally write to the wrong file or copy over files you don't mean to.

Copiar ficheiros para um ambiente comum de armazenamento (e.g., *DropBox*) simplifica a partilha desse ficheiro...

...mas **partilhar ficheiro(s) é diferente de fazer controlo de versões!**

Sistema de Controlo de Versões – modelo centralizado

Um único servidor com todas as versões de todos os ficheiros.
Múltiplos clientes a fazer “*checkout*” e “*checkin*” dos seus ficheiros.



e.g., sistemas com modelo centralizado: CVS, Subversion, Perforce

... modelo centralizado – principais características

- Durante muitos anos foi o modelo mais usado
 - diferentes ferramentas seguem este modelo
 - ... as mais usadas são o CVS e o Subversion
- A limitação mais óbvia é a da dependência do servidor
 - representa um único ponto de falha
 - ... falha do servidor impede equipa de colaborar
 - ... se servidor falha nenhuma alteração poderá ser registada
- Ter todo o projeto está num único repositório é um risco
 - perda desse repositório origina perda de todo o projeto!
 - ... excepto aquilo que estiver em “backup”
 - ... ou que estiver residente em cópia local

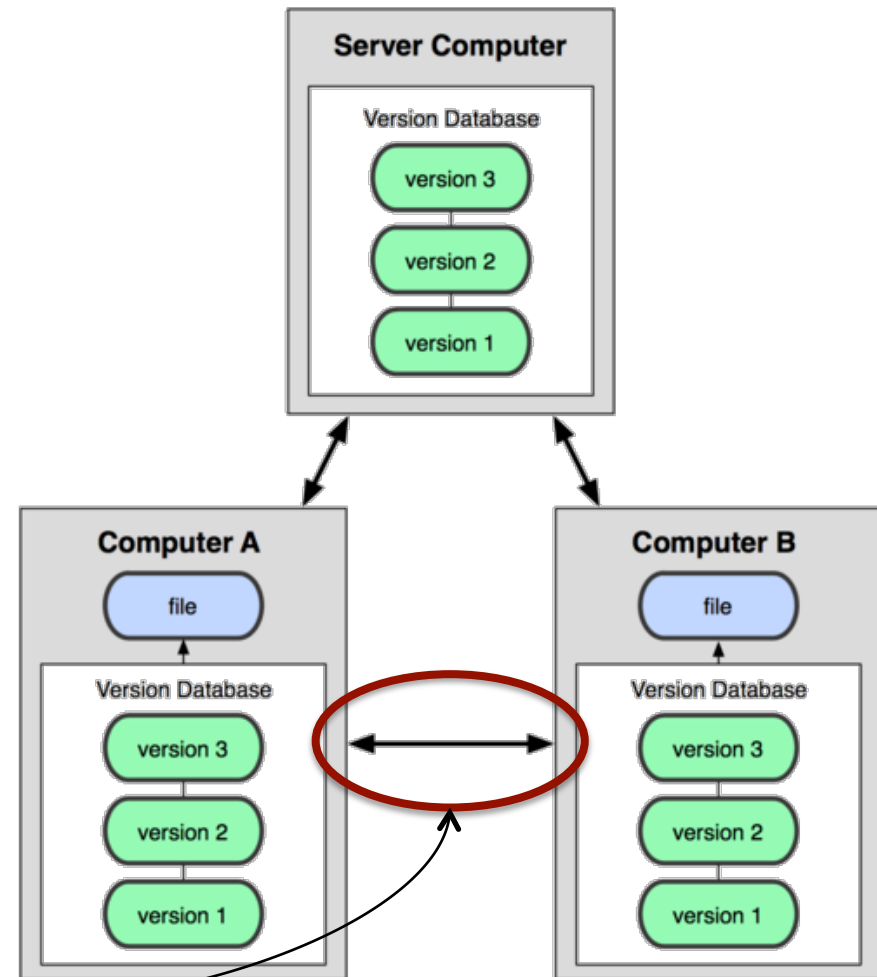
Sistema de Controlo de Versões – modelo distribuído

Cliente não faz apenas “checkout” de determinada versão.

Cliente mantém uma réplica (“mirror”) de todo o repositório.

Cada “checkout” é como que um “backup” total dos dados.

***esta ligação direta
não existe no modelo
centralizado!***



e.g., sistemas com modelo distribuído: Git, Mercurial, Bazaar, Darcs

... modelo distribuído – principais características

- Este modelo tem atualmente grande utilização
 - o Git é talvez a ferramenta mais conhecida
 - ... foi bastante potenciada pelo GitHub
- O aspecto mais óbvio é que reduz a dependência do servidor
 - servidor serve apenas como ponto de apoio à partilha dos dados
 - ... falha do servidor não impede equipa de colaborar diretamente
 - ... comunicam ponto-a-ponto em vez de usarem um ponto comum
- O projeto está replicado em cada repositório
 - perda do repositório no servidor
 - ... é ultrapassada replicando um qualquer outro repositório
 - ... qualquer versão que esteja no repositório comum está também, de certeza, em pelo menos uma cópia local
 - ... pois passou para o repositório comum a partir de um local

Git – contexto

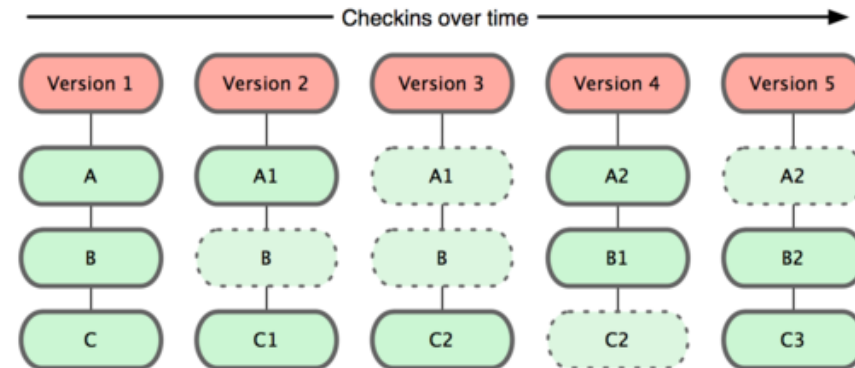
- Git (pode ser) acrónimo de “Global information tracker”
 - ... em gíria (Inglês) é “cabeça dura; pessoa que acha ter sempre razão”...
- Criado por Linus Torvalds
 - ... para gestão do núcleo (“kernel”) Linux
 - surge, com visibilidade pública, em 2005
- Foi construído com o objectivo essencial de ser
 - completamente **distribuído**
- ... outros objetivos incluem
 - desenho simples e elevando desempenho, i.e., resposta rápida
 - suporte a desenvolvimento não linear, i.e., grande número de ramos (“branches”) paralelos
 - tratamento de projetos de grande dimensão (e.g., “kernel” Linux) de modo eficiente, considerando desempenho e dimensão dos dados

Git – o modelo de base

Git – em cada nova versão:

- ficheiros novos ou alterados são copiados na totalidade
 - ficheiros sem alteração mantém uma referência (link)

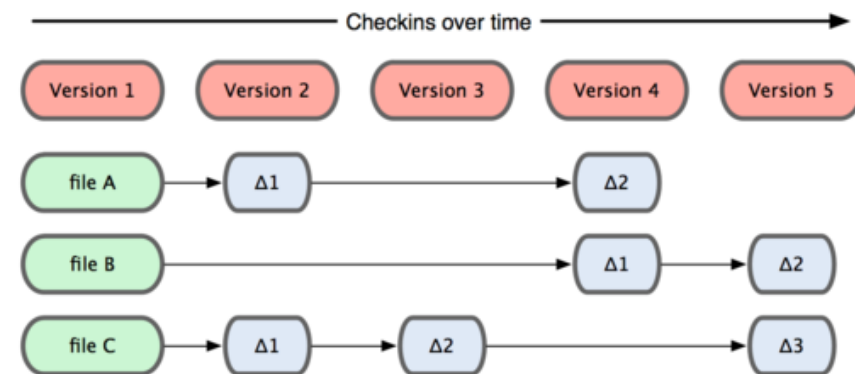
Sistema **Git**



Outros sistemas, e.g., CVS, Subversion

Outros – em cada nova versão:

- são registadas as alterações de cada ficheiro
- considera-se cada ficheiro mais as alterações que teve



Git – manutenção da integridade dos dados

- Cada ficheiro adicionado ao Git fica associado a uma chave única
 - i.e., depois de se fazer `git add nomeFicheiro`
 - é gerada uma chave única para o ficheiro `nomeFicheiro`
- ... a chave é gerada pelo SHA-1 (“Secure Hash Algorithm”)
 - ... 160bits gerados tendo como “input” o conteúdo do próprio ficheiro
- No Git tudo é endereçado por este tipo de chave (“hash value”)
 - os ficheiros não são referidos pelo seu nome mas por esta chave

... algum detalhe sobre a chave usada no Git

- ... a chave é gerada pelo SHA-1 (“Secure Hash Algorithm”)
 - ... 160bits gerados tendo como “input” o conteúdo do próprio ficheiro
- Os 160bits são apresentados como uma “string” com 40 caracteres
 - cada carácter é apresentado como hexadecimal; i.e., varia [0..9] e [a..f]
 - ... 4bits-por-carácter x 40caracteres = 160bits (gerados pelo SHA-1)
- Por exemplo, o texto: “isto é o conteúdo de um documento”
 - gera a chave (“hash value”)
 - 3589b328cc4082f1b262d3f99bc5f2ccdfcefca6
 - ... teste um “SHA1 encoder”, e.g., em <http://sha1-hash-online.waraxe.us/>
- ... resultado teórico sobre a probabilidade de uma colisão no SHA-1
 - uma colisão no SHA-1 exige no mínimo 2^{80} execuções do algoritmo
 - ... Tera = 2^{40} ; Peta = 2^{50} ; Exa = 2^{60} ; Zetta = 2^{70} ; Yotta = 2^{80} execuções!

... ainda sobre a chave usada pelo Git

“If you commit an object that hashes to the same SHA-1 value as a previous object, Git will see the previous object already in database and assume it was already written. If you try to check out that object again at some point, you’ll always get the data of the first object.”

“An example to give an idea of what it takes to get a SHA-1 collision. If all 6.5 billion humans on Earth were programming, and every second, each one was producing code that was the equivalent of the entire Linux kernel history (1 million Git objects) and pushing it into one enormous Git repository, it would take 5 years until that repository contained enough objects to have a 50% probability of a single SHA-1 object collision.

A higher probability exists that every member of your team will be attacked and killed by wolves in unrelated incidents on the same night.”

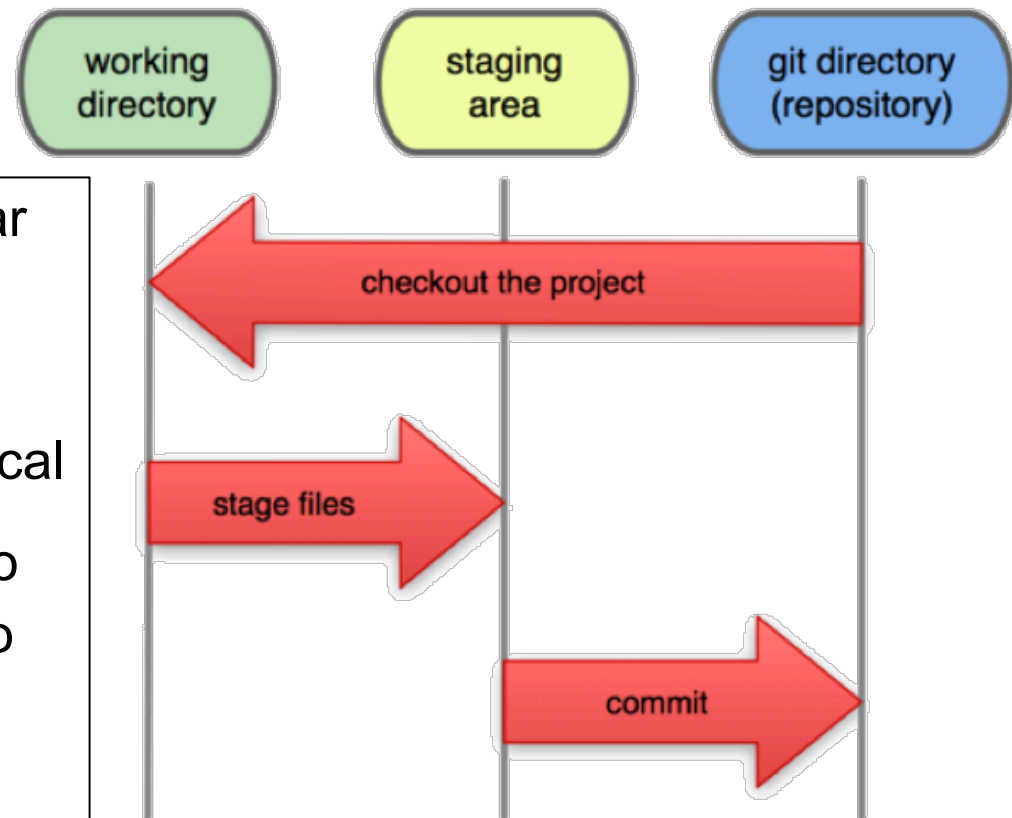
cf., texto em: <http://git-scm.com/book/en/ch6-1.html#A-SHORT-NOTE-ABOUT-SHA-1>

Os 3 estados dos ficheiros no Git

Local Operations

Cada ficheiro apenas pode estar num dos estados:

- ***committed*** – dados estão consolidados no repositório local
- ***modified*** – ficheiro modificado mas ainda não consolidado no repositório local
- ***staged*** – ficheiro modificado marcado para ser “commit”



cf., figura em: <http://git-scm.com/book/en/Git-Basics-Recording-Changes-to-the-Repository>

... as 3 áreas de um repositório Git

- Os 3 estados dos ficheiros originam as 3 áreas do repositório Git
 - “Git directory”, “working directory”, “staging area”
- ... a pasta Git (“**Git directory**”)
 - local onde são registados os meta-dados e a base de dados Git
 - é a área mais importante; é copiada ao replicar (“clone”) o repositório de outro computador
- ... a pasta de trabalho (“**working directory**”)
 - contém todos os ficheiros de uma determinada versão do projeto
 - os ficheiros são descomprimidos, do “Git directory”, e copiados para uso
- ... a área intermédia (“**staging area**”)
 - simples ficheiro, no “Git directory” com indicação de quais os ficheiros que estarão na próxima consolidação (estarão no próximo “commit”)
 - também designado por “index”; usualmente referido como “staging area”

Onde está o Git?

<http://git-scm.com/>

The screenshot shows the Git website homepage. At the top left is the Git logo (an orange diamond with a white branching diagram) followed by the text "git --fast-version-control". To the right is a search bar with the placeholder text "Search entire site...". Below the logo, there are two paragraphs of text describing Git as a free and open source distributed version control system. To the right of the text is a diagram showing several stacks of papers connected by colored lines (red, blue, yellow) representing a branching model. Below the text is a "try Git" button with a GitHub logo. At the bottom, there is a navigation menu with four items: "About", "Documentation", "Downloads", and "Community". A red circle is drawn around this navigation menu. To the right of the circle, the text "testar e conhecer o Git" is written.

git --fast-version-control

Search entire site...

Git is a **free and open source** distributed version control system designed to handle everything from small to very large projects with speed and efficiency.

Git is **easy to learn** and has a **tiny footprint with lightning fast performance**. It outclasses SCM tools like Subversion, CVS, Perforce, and ClearCase with features like **cheap local branching**, convenient **staging areas**, and **multiple workflows**.

Learn Git in your browser for free with Try Git.

About
The advantages of Git compared to other source control systems.

Documentation
Command reference pages, Pro Git book content, videos and other material.

Downloads
GUI clients and binary releases for all major platforms.

Community
Get involved! Mailing list, chat, development and more.

testar e conhecer o Git

Como (começar a) usar o Git?

- Para obter e instalar ir a “<http://git-scm.com/downloads>”
 - escolher o sistema operativo e seguir as instruções de instalação
- Configurar o essencial do Git
 - após instalar fornecer a sua identificação (nome e email)
`git config --global user.name "Paulo Trigo"`
`git config --global user.email paulo.trigo@gmail.com`
 - para ver a configuração corrente fazer: `git config --list`
- **Criar um repositório** Git (numa pasta local)
 - ir para a pasta que deverá ser a raiz do projeto e fazer:
`git init` (este comando cria, na pasta corrente, a subpasta “.git”)
- ... os vários níveis de configuração ficam [no Mac] em :
 - `[/usr/local/git]/etc/gitconfig` (*todos os utilizadores*)
 - `[/Users/ptrigo]/.gitconfig` (*utilizador*) `.git/config` (*projeto*)

Fluxo de trabalho (“Workflow”) típico no Git

- **Editar** ficheiros na “working directory”
 - fazendo, por exemplo (em linha de comando):
`echo "um exemplo" > f1.txt`
`echo "outro exemplo" > f2.txt`
- **Adicionar** ficheiros à “**staging area**”
 - adicionar apenas determinado ficheiro, por exemplo, `f1.txt` fazendo:
`git add f1.txt`
 - ... ou adicionar todos os ficheiros fazendo
`git add *`
- **Consolidar** no “**Git directory**” os ficheiros da “**staging area**”
 - ao consolidar colocar um descritivo (“commit message”) fazendo:
`git commit -m "v00 - o meu primeiro commit"`

... continuar a trabalhar e a consolidar no “Git directory”

- Continuar a trabalhar na “working directory”
 `echo "um exemplo COM MAIS ESTA LINHA" >> f1.txt`
 – i.e., o ficheiro `f1.txt` foi alterado na “working area”
- Consolidar (“commit”) de novo no “Git directory” mas diretamente
 – i.e., sem passar pela “staging area”
 `git commit -a -m "v01 - o meu segundo commit"`
- O que é que está consolidado no repositório (i.e., no “Git directory”)?
 `git log --pretty=oneline`

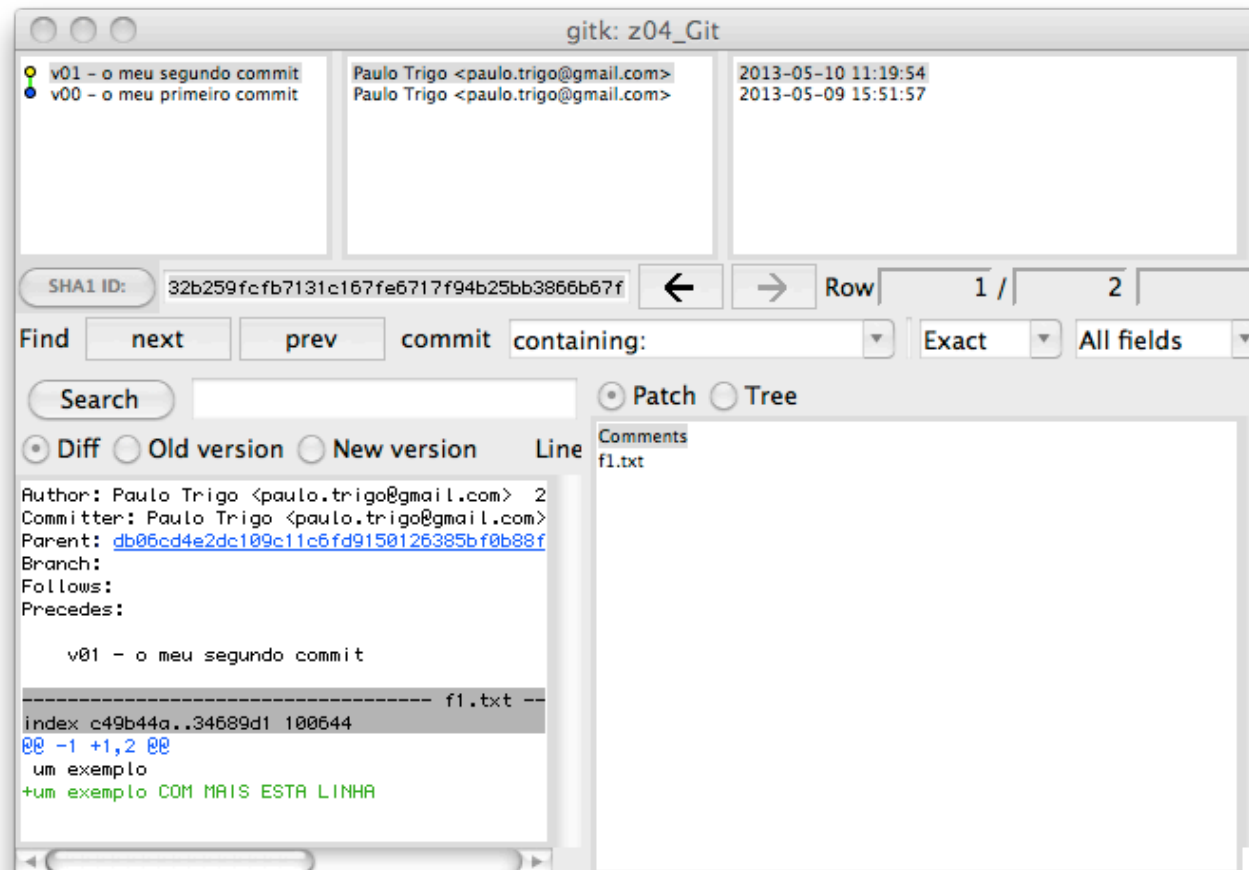
<code>449f7adf3157bf5cd626a7302c748dcf7d51906e</code>	<code>v01 - o meu segundo commit</code>
<code>4d2f1c4276052d633453ad4a3bf49a3ef9f0f825</code>	<code>v00 - o meu primeiro commit</code>

“chave única” (SHA-1) gerada,
pelo Git, no “commit”

comentário (“string”) editado,
pelo utilizador, no “commit”

... uma ferramenta gráfica sobre o “log” do Git

> **gitk** *está incluída na instalação do Git*



mesma informação que a obtida pelo comando
“git log” através das suas várias opções

... e agora recuperar uma outra versão

- Quais as versões consolidadas (“commit”) no repositório?

```
git log --pretty=oneline
```

```
449f7adf3157bf5cd626a7302c748dcf7d51906e v01 - o meu segundo commit
```

```
4d2f1c4276052d633453ad4a3bf49a3ef9f0f825 v00 - o meu primeiro commit
```

- Recuperar** (i.e, pôr na “working directory”) uma daquelas versões

```
git checkout 4d2f1c4276052d633453ad4a3bf49a3ef9f0f825
```

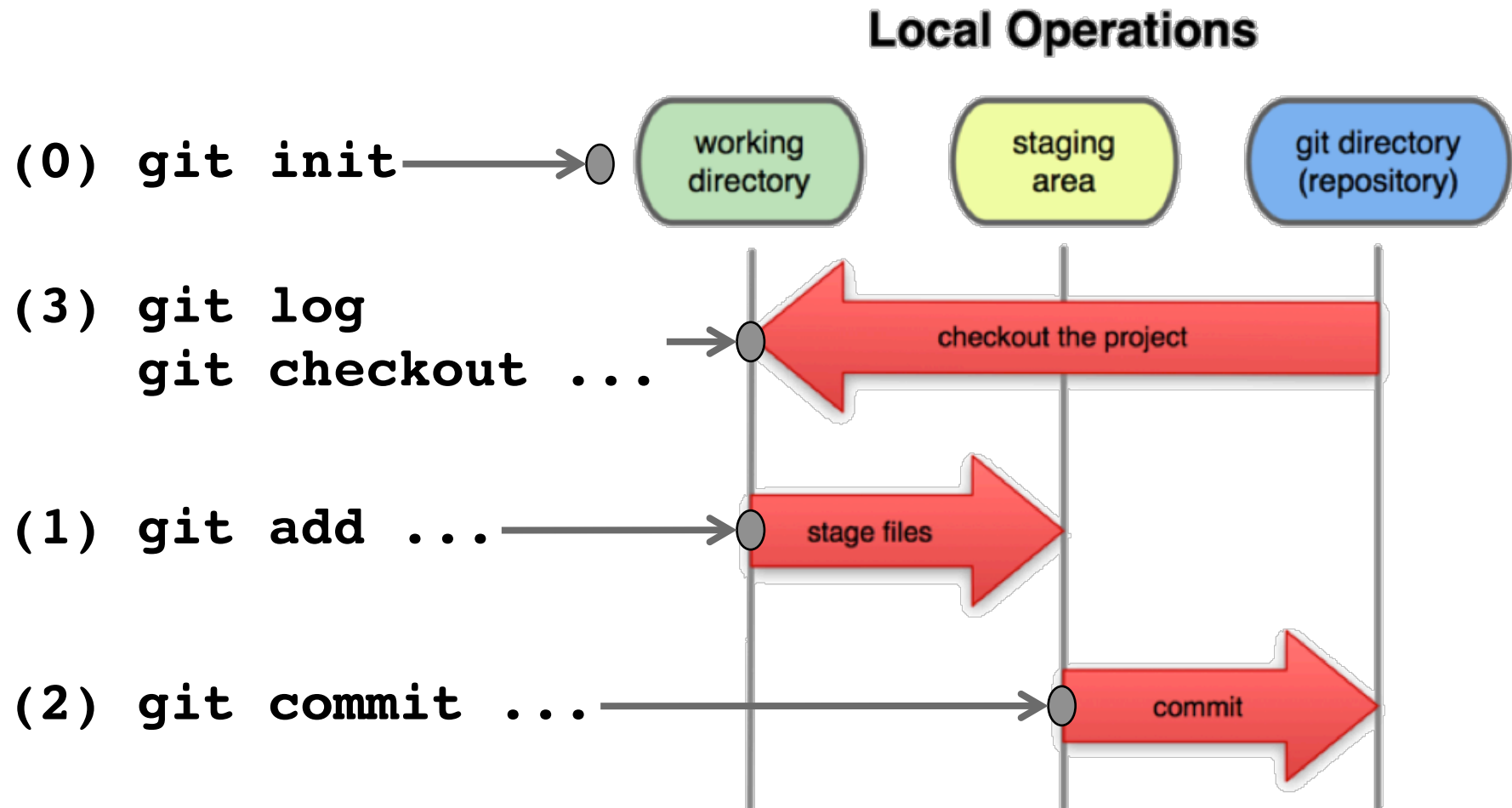
- notar que esta é a chave da versão (e não a de um ficheiro)
- *a chave da versão não é usada para comparar versões pelo que as chaves que obtiver ao fazer este exemplo podem ser diferentes destas*

- ... agora verificar que temos de volta o “f1.txt” da primeira versão

```
> more t1.txt
```

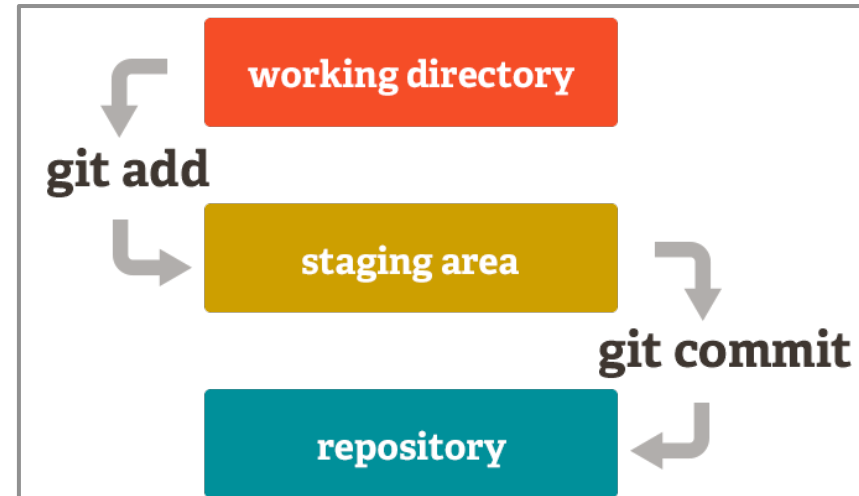
```
> um exemplo
```

Síntese – comandos, estados e áreas do repositório Git

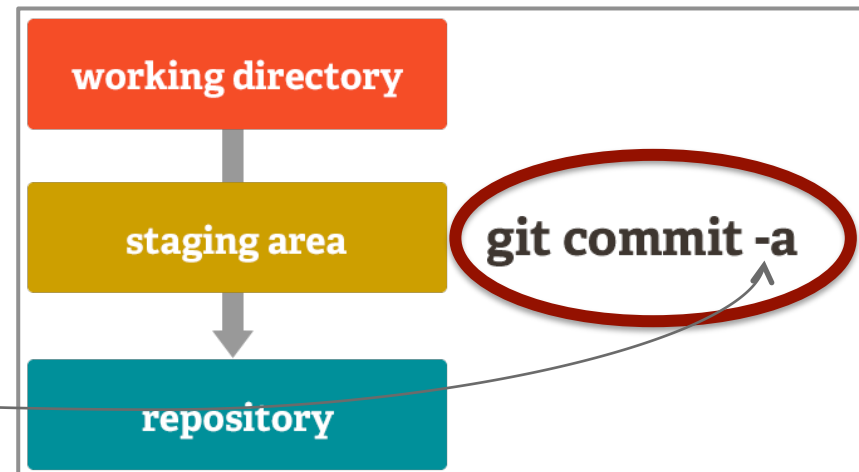


... consolidar “*com e sem*” passar pela “staging area”

- passar à “staging area”
- consolidar no repositório



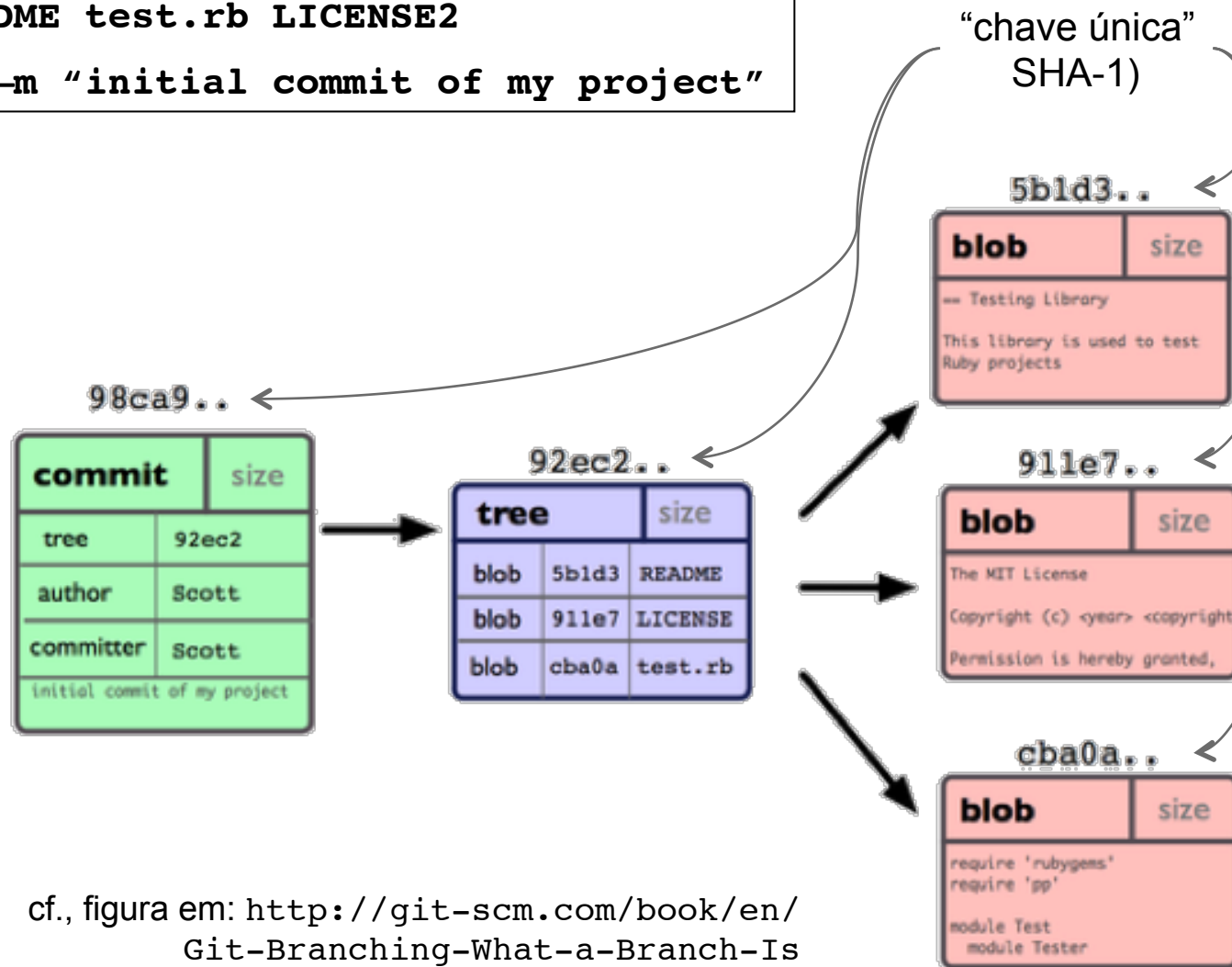
- consolidar no repositório
- não passa na “staging”



cf., figuras em: <http://git-scm.com/about/staging-area>

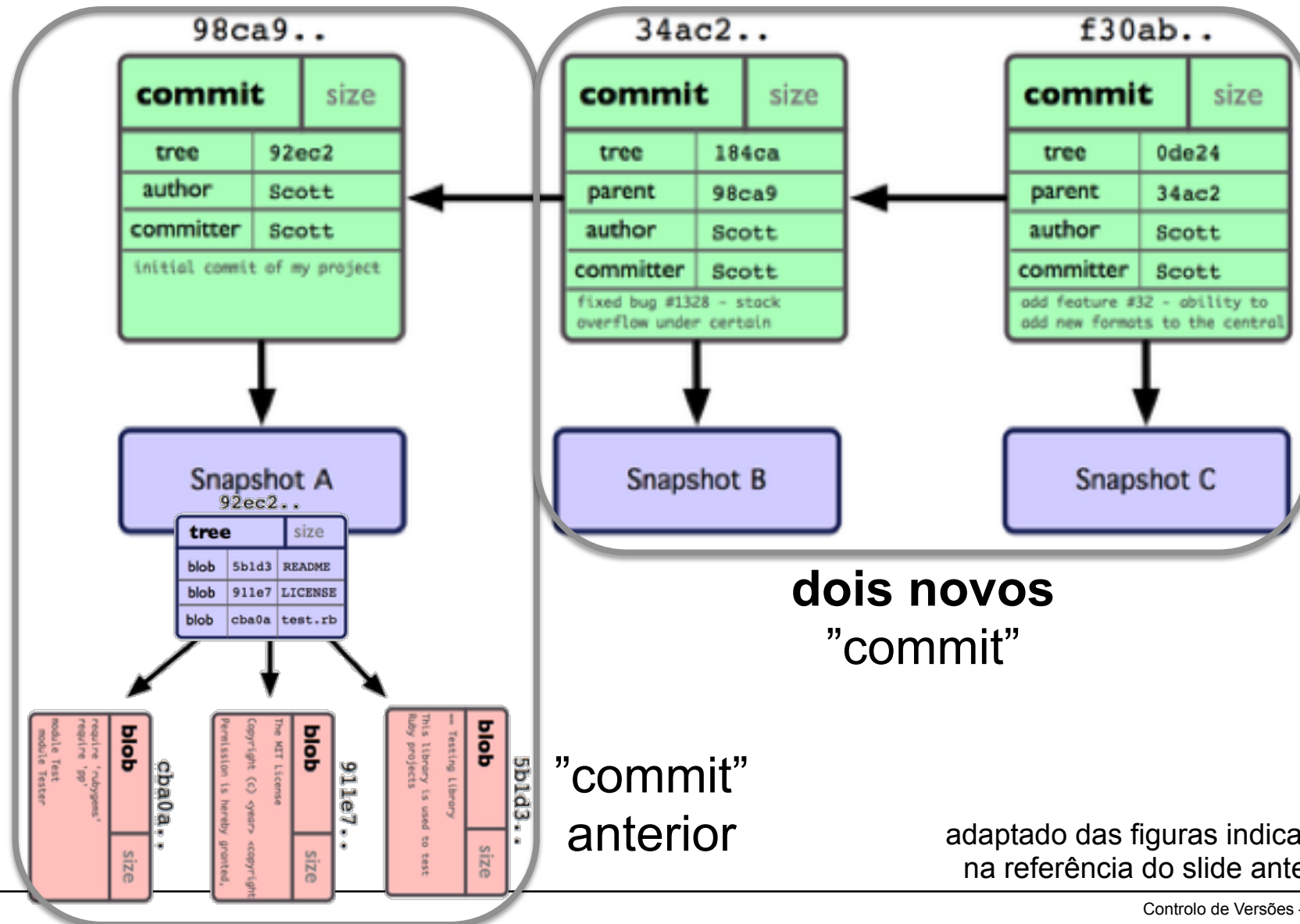
Estrutura interna gerada pelos comandos add e commit

```
git add README test.rb LICENSE2  
git commit -m "initial commit of my project"
```



cf., figura em: <http://git-scm.com/book/en/Git-Branching-What-a-Branch-Is>

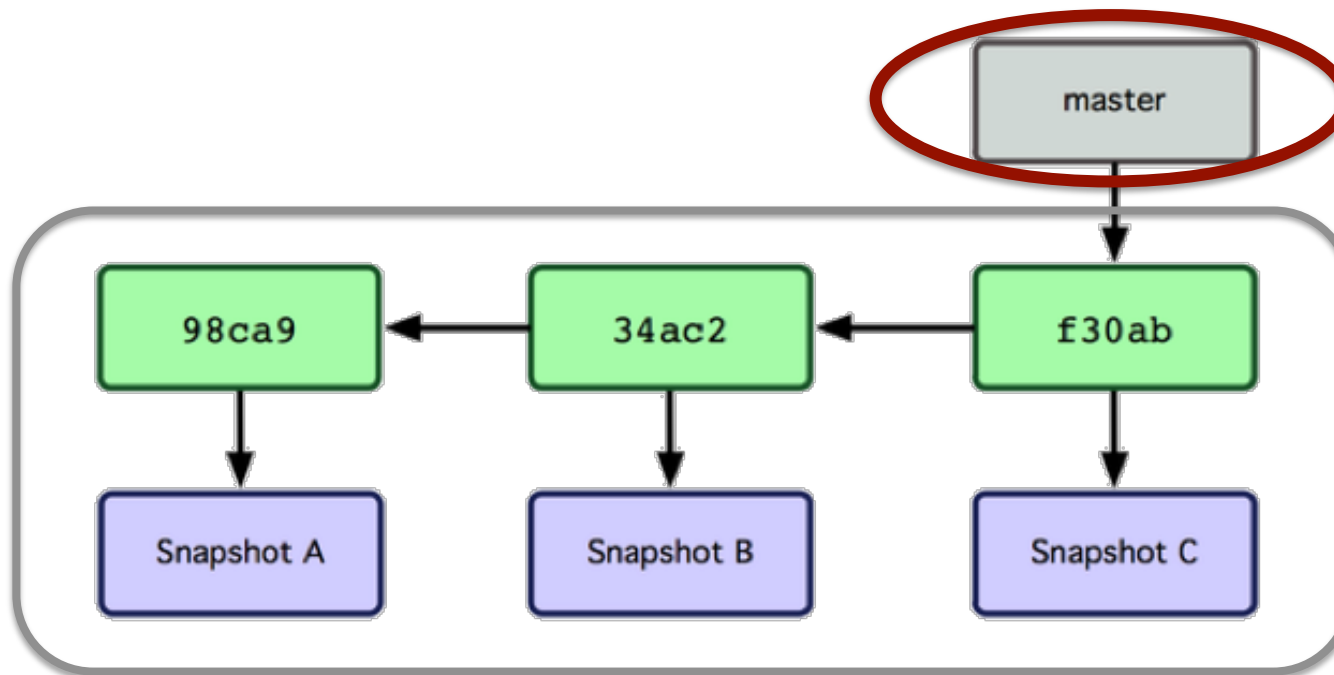
... novos “commit” geram lista encadeada de árvores



adaptado das figuras indicadas na referência do slide anterior

O que é um ramo (“branch”) no Git?

Um ramo é simplesmente um apontador para uma versão.
Cada ramo tem um nome (definido pelo utilizador).
Há um ramo “**default**” designado por “**master**”.

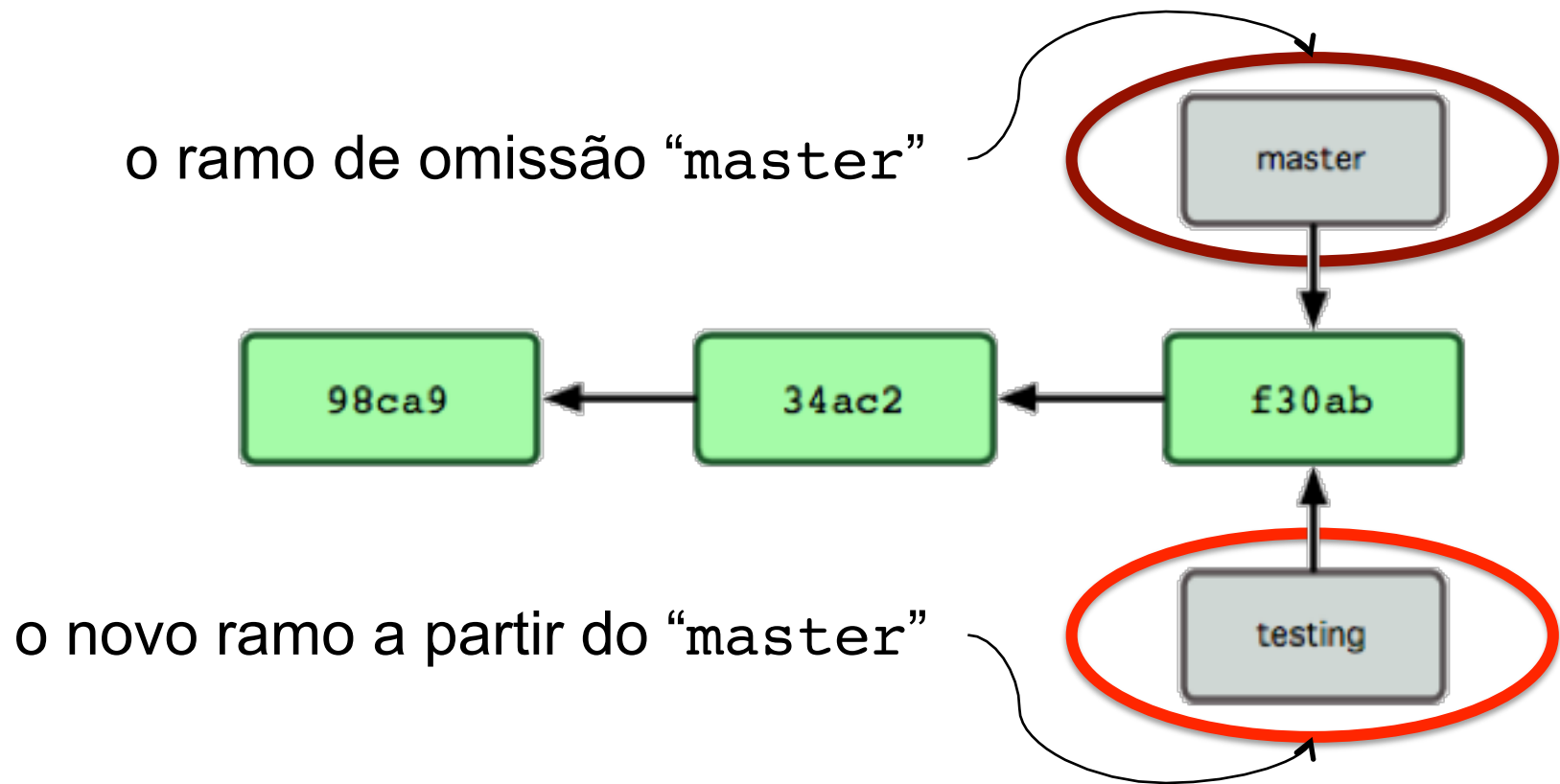


aqueles mesmos 3 “commit” (cf. slide anterior)

... 3 passos para criar um novo ramo (“branch”)

- Garantir que se está posicionado na base do novo ramo
 - i.e., pretende-se criar novo ramo a partir da última versão do “master”
git checkout master
- Verificar quais os ramos existentes e confirmar o ramo corrente
git branch
 - ... para o nosso exemplo (do f1.txt e f2.txt) temos o resultado:
*** master**
 - ... indica que só existe o ramo “master” e o “*” indica que é o corrente
- E agora **criar o novo ramo** a que chamaremos “testesAdHoc”
git branch testesAdHoc
- ... voltando a ver que ramos existem (**git branch**) temos agora
*** master**
testesAdHoc

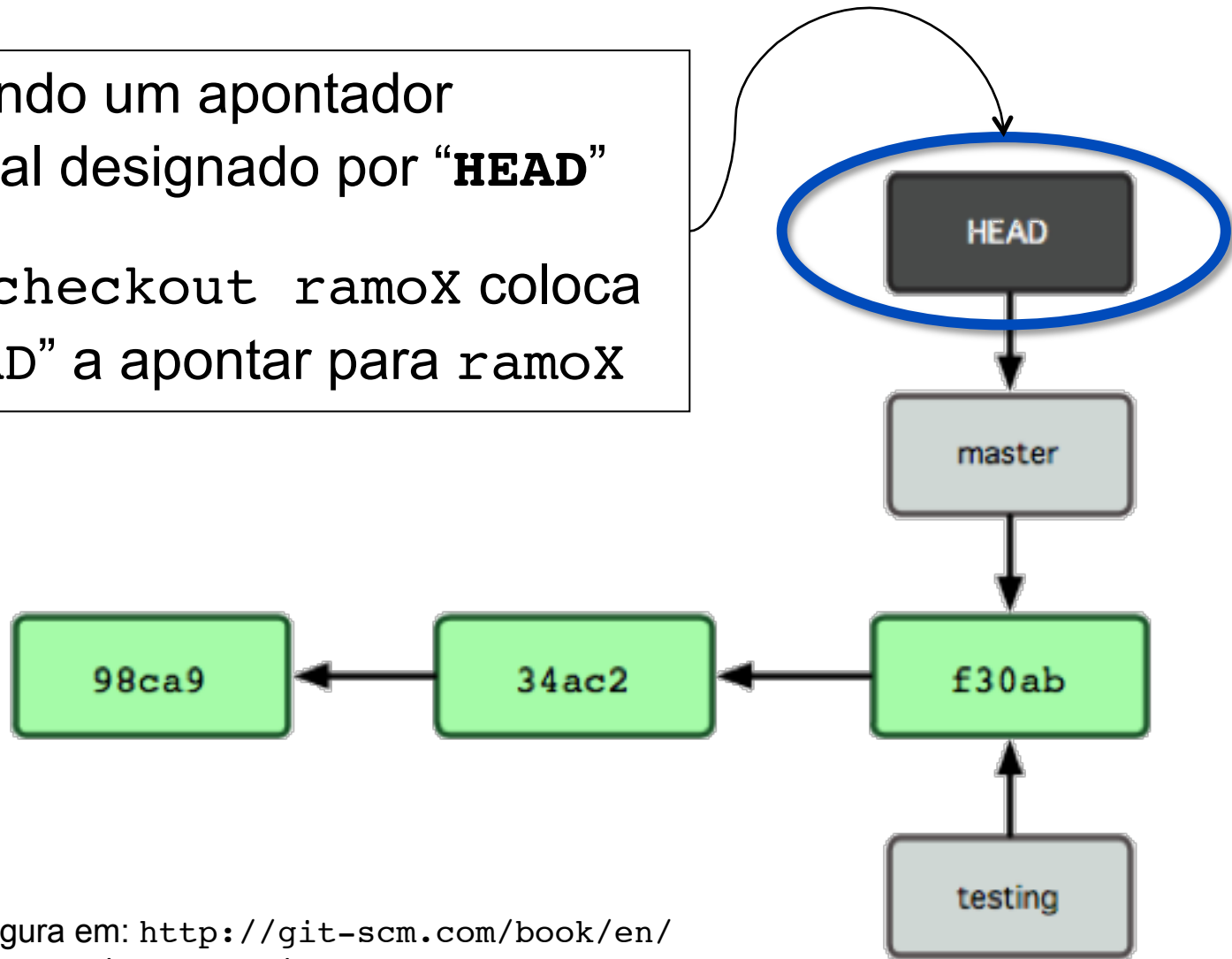
... criar um novo ramo é dar nome ao novo apontador



cf., figura em: <http://git-scm.com/book/en/Git-Branching-What-a-Branch-Is>

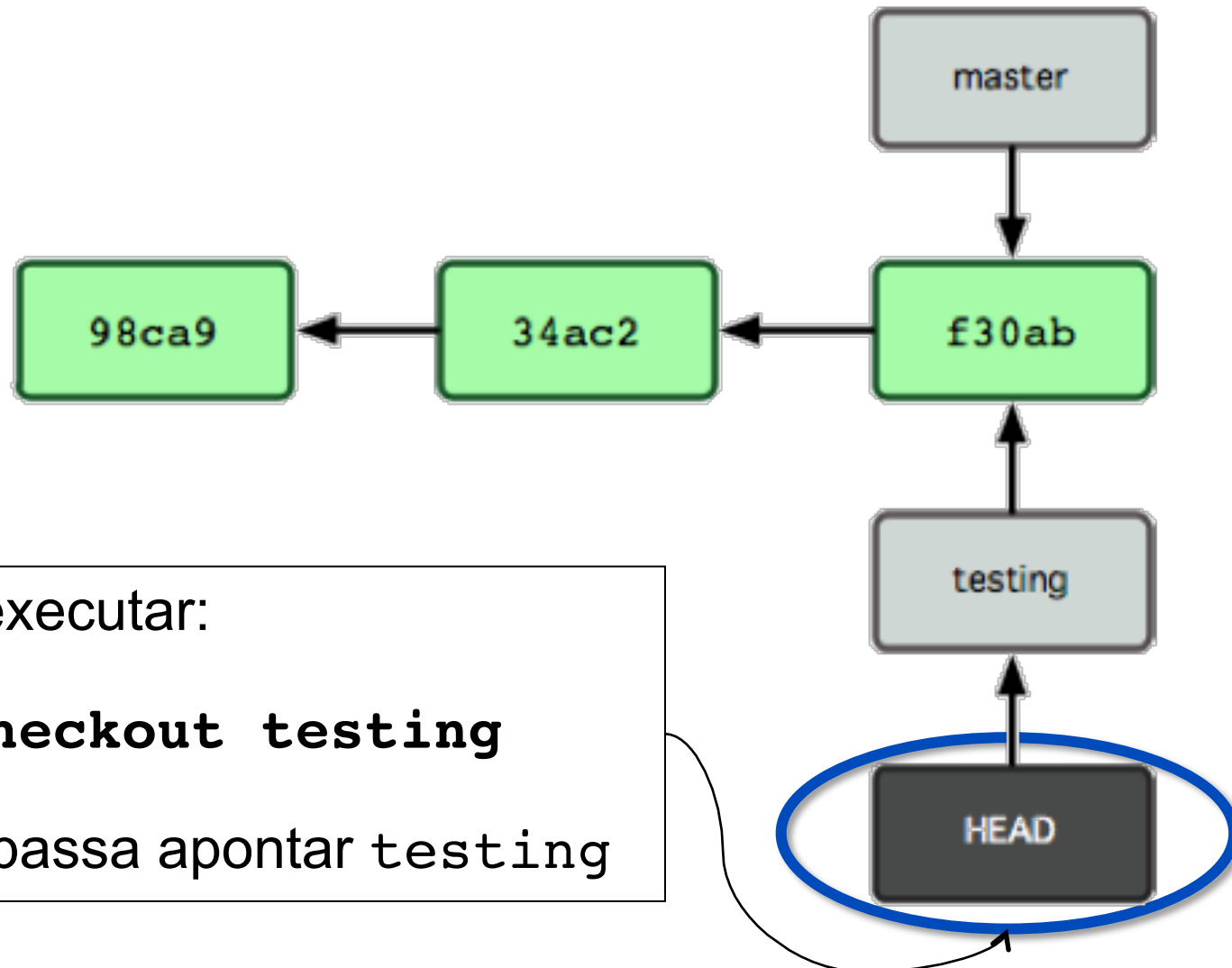
Mas, como é que o Git sabe qual o ramo corrente?

- mantendo um apontador especial designado por “**HEAD**”
- `git checkout ramoX` coloca o “HEAD” a apontar para ramoX



cf., figura em: <http://git-scm.com/book/en/Git-Branching-What-a-Branch-Is>

Alterar o ramo (“branch”) corrente é alterar o “HEAD”



... alterar o ramo corrente (no nosso exemplo)

```
> git checkout testesAdHoc
```

```
> git branch -v
```

```
master          449f7ad    v01 - o meu segundo commit  
* testesAdHoc  449f7ad    v01 - o meu segundo commit
```

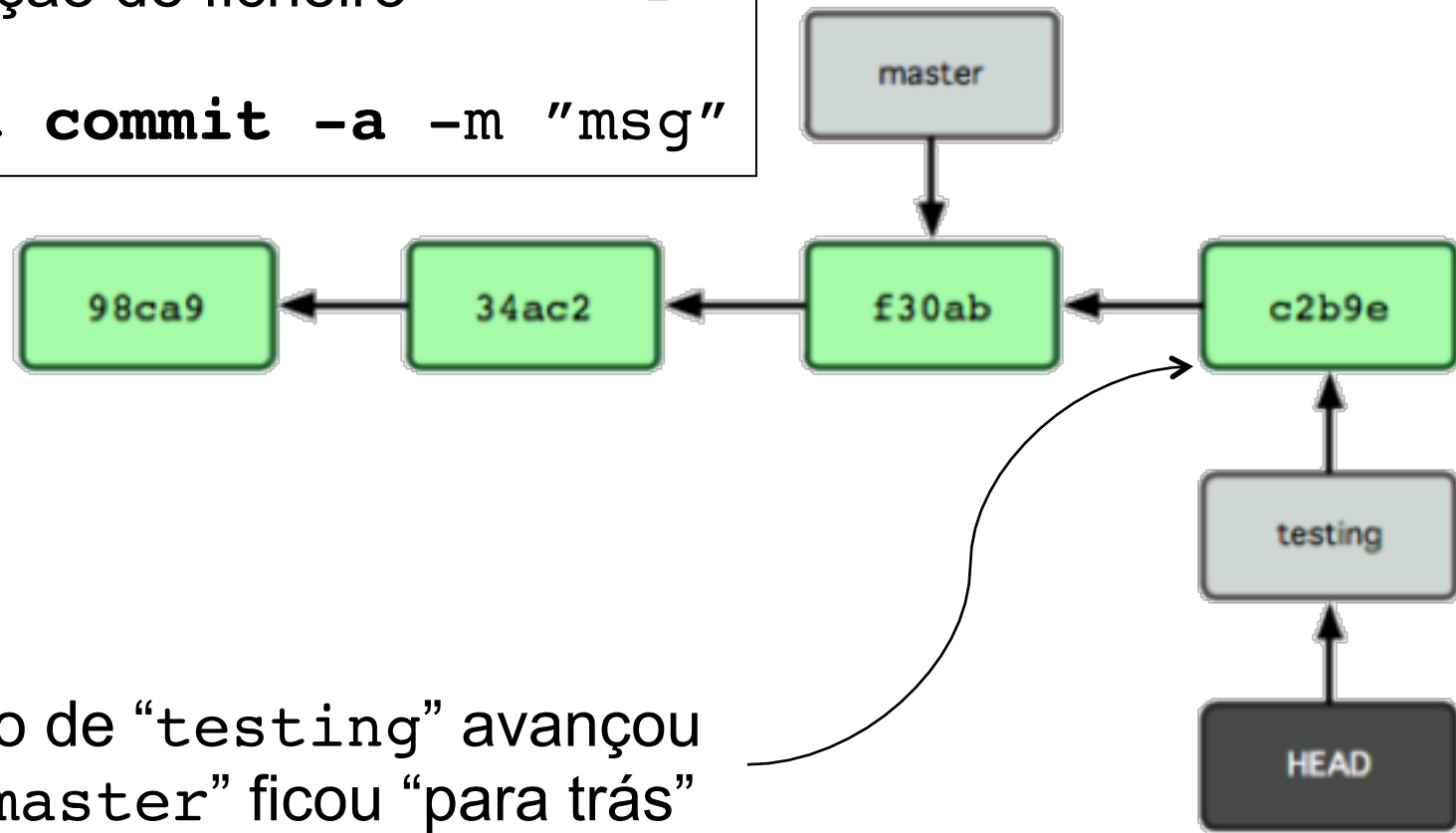
- notar que o “*” está no ramo “**testesAdHoc**”
- ... isso dizer que “**testesAdHoc**” está a ser apontado por “**HEAD**”
- notar que ambos os ramos têm a mesma chave
- ... isso quer dizer que (de momento) apontam a mesma versão

Comando útil para criar um novo ramo e mudar para ele ao mesmo tempo:
`git checkout -b testesAdHoc`

E depois?... trabalho, faço novo “commit”, e como fica?

Efeito de executar:

- edição de ficheiro
- **git commit -a -m “msg”**



o ramo de “testing” avançou
e o “master” ficou “para trás”

.... alterar o ramo corrente (no nosso exemplo)

- Alterar o novo ramo corrente trabalhando na “working directory”
echo "um exemplo NO RAMO testesAdHoc" >> f1.txt
- ver conteúdo de f1.txt fazendo: more f1.txt
um exemplo
um exemplo COM MAIS ESTA LINHA
um exemplo NO RAMO testesAdHoc
- Consolidar alterações neste ramo (recordar que “-a” evita “staging”)
git commit -a -m "v01.t - primeiro commit testesAdHoc"
- ... obter informação sobre os “commit” neste ramo
git log --pretty=oneline

```
4bf112d359dca315fe4e2e1... v01.t - primeiro commit testesAdHoc
449f7adf3157bf5cd626a73... v01 - o meu segundo commit
4d2f1c4276052d633453ad4... v00 - o meu primeiro commit
```


E se quiser fazer ajustes ao meu último “commit”?

- Apenas “emendar” o último “commit” (i.e., sem fazer novo “commit”)
git commit --amend
- ... “--amend” coloca a “staging area” no último “commit” efectuado
 - caso a “staging area” ainda não tenha sido alterada
 - ... então, apenas se altera a mensagem usada nesse último “commit”
git commit --amend -m "v01.t - teste ao input"

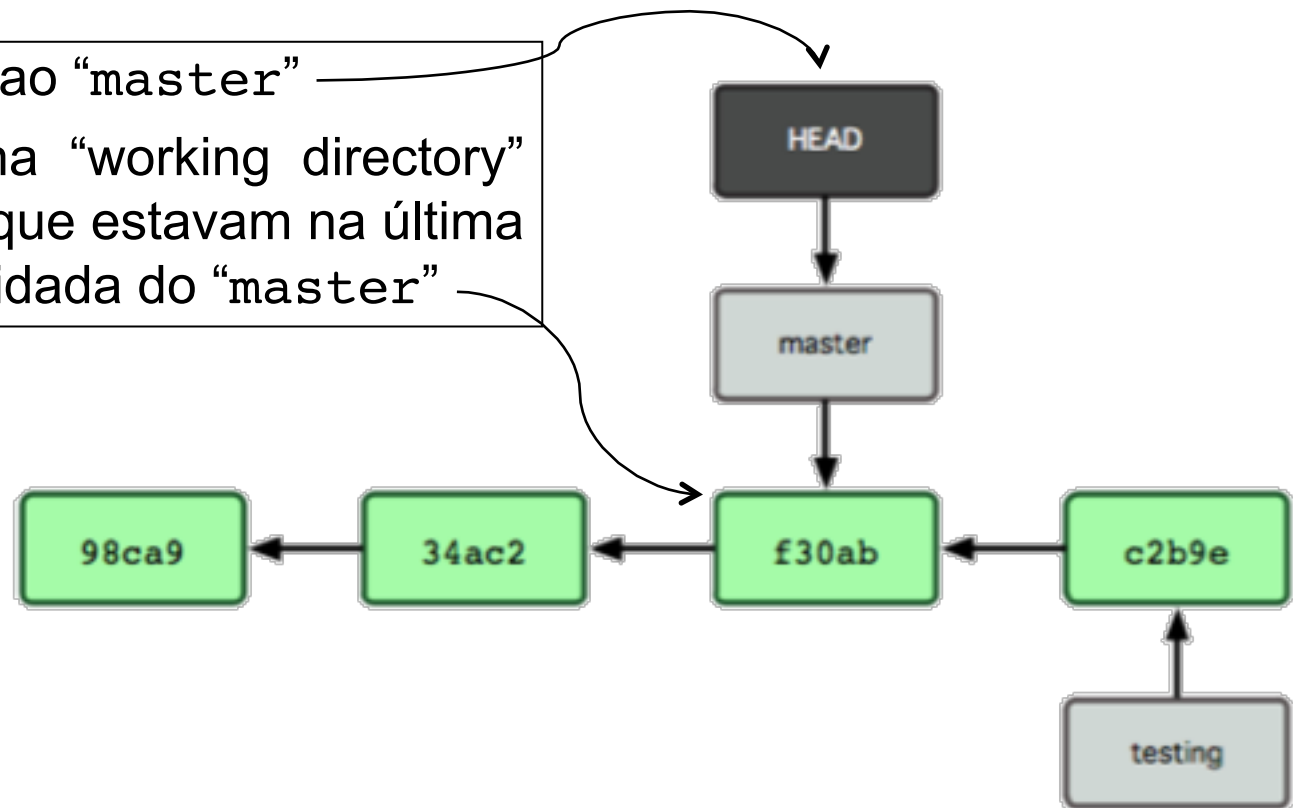
```
git log --pretty=oneline
26a977634ba7e99502b2206... v01.t - teste ao input
449f7adf3157bf5cd626a73... v01 - o meu segundo commit
4d2f1c4276052d633453ad4... v00 - o meu primeiro commit
```

- notar que “**--amend**” não faz novo “commit” mas altera a chave do anterior
- neste exemplo tínhamos a chave **4bf112d359dca315fe4e2e1...** (cf., slide anterior) que passou para **26a977634ba7e99502b2206...**

Agora voltemos ao ramo do “master”

Efeito de executar:

- `git checkout master`
- “HEAD” voltou ao “master”
- os ficheiros na “working directory” são agora os que estavam na última versão consolidada do “master”

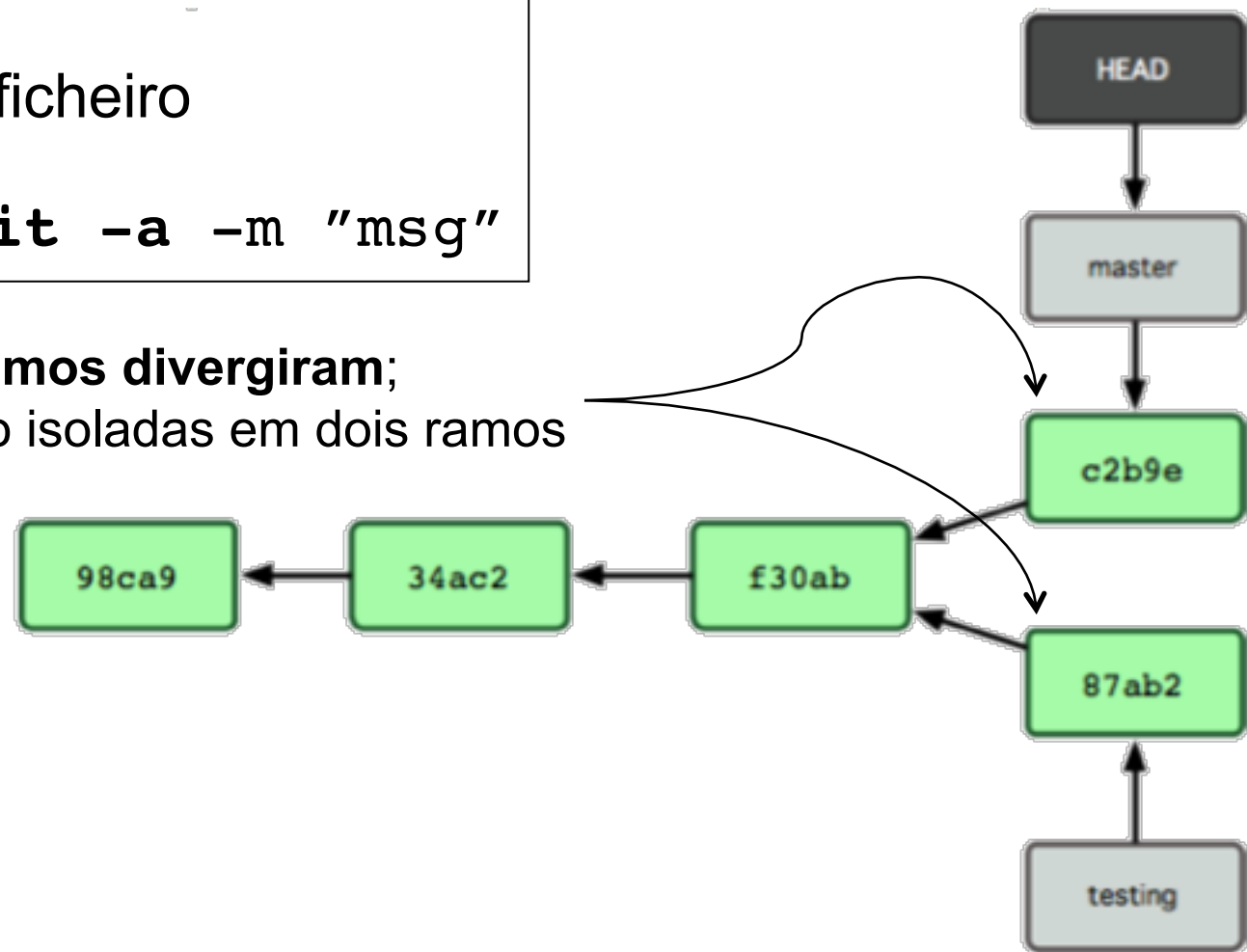


... agora, após novo “commit” no “master” como se fica?

Efeito de executar:

- edição de ficheiro
- `git commit -a -m "msg"`

agora **os ramos divergiram**;
as alterações estão isoladas em dois ramos



.... criar ramos divergentes (no nosso exemplo)

- Voltar ao ramo “master” e trabalhar (i.e., alterar conteúdo ficheiros)

```
git checkout master
```

```
echo "um exemplo que no RAMO MASTER DIVERGE" >> f1.txt
```

– ver conteúdo de f1.txt fazendo: `more f1.txt`

```
um exemplo
```

```
um exemplo COM MAIS ESTA LINHA
```

```
um exemplo que no RAMO MASTER DIVERGE
```

- ... informação sobre os “commit” existentes neste ramo do “master”

```
git log --pretty=oneline
```

```
449f7adf3157bf5cd626a73... v01 - o meu segundo commit
```

```
4d2f1c4276052d633453ad4... v00 - o meu primeiro commit
```

- notar que “**26a977634ba7e99502b2206... v01.t - teste ao input**” é uma versão posterior a esta (do “master”) e portanto não consta desta lista

- recorde que essa versão consta na lista de “log” do ramo **testesAdHoc**

.... criar ramos divergentes (no nosso exemplo)

- Consolidar alterações neste ramo (recordar que “-a” evita “staging”)

```
git commit -a -m "v02 - diverge master"
```

- ... informação sobre os “commit” neste ramo derivado de “master”

```
git log --pretty=oneline
```

```
2776a4e63253e40632021b5... v02 - diverge master ←  
449f7adf3157bf5cd626a73... v01 - o meu segundo commit  
4d2f1c4276052d633453ad4... v00 - o meu primeiro commit
```

- ... e vamos ver que diverge do que está no ramo “testesAdHoc”

```
git checkout testesAdHoc
```

```
git log --pretty=oneline
```

```
26a977634ba7e99502b2206... v01.t - teste ao input ←  
449f7adf3157bf5cd626a73... v01 - o meu segundo commit  
4d2f1c4276052d633453ad4... v00 - o meu primeiro commit
```

Junção (“merge”) de ramos (“branches”)

- Vamos confirmar que estamos no “testesAdHoc” (ramo corrente)

git branch

```
master
* testesAdHoc
```

- Fazer junção do ramo corrente (“testesAdHoc”) com o “master”

git merge master

```
Auto-merging f1.txt
```

```
CONFLICT (content): Merge conflict in f1.txt
```

```
Automatic merge failed; fix conflicts and then commit the result.
```

- ... aquela junção não foi efetuada pois há conflitos
 - este tipo de conflito precisa de ser resolvido pelos utilizadores
 - ... se precisar de saber quais os ficheiros em conflito fazer **git status**

... marcadores de apoio à resolução de conflitos

- Ao conteúdo do(s) ficheiro(s) em conflito são adicionados
 - marcadores de resolução-de-conflito (“conflict-resolution markers”)
 - ... no nosso exemplo, o `f1.txt` tem agora este conteúdo:

```
um exemplo
um exemplo COM MAIS ESTA LINHA
<<<<<<< HEAD
um exemplo NO RAMO testesAdHoc
=====
um exemplo que no RAMO MASTER DIVERGE
>>>>>>> master
```

`f1.txt`

- Os marcadores indicam o(s) conteúdo(s) em conflito nos dois ramos
 - entre “<<<<<<< **HEAD**” e “=====” é o conteúdo no ramo corrente
 - entre “=====” e “>>>>>>> **master**” é o conteúdo no ramo “master”

Resolução de conflitos (para junção de ramos)

- Para resolver o conflito eliminar os marcadores e editar o pretendido
 - e.g., manter conteúdo só de um ramo ou manter conteúdo de ambos
 - ... ou editar uma qualquer outra forma de juntar os conteúdos

```
um exemplo
um exemplo COM MAIS ESTA LINHA
<<<<<< HEAD
um exemplo NO RAMO testesAdHoc
=====
um exemplo que no RAMO MASTER DIVERGE
>>>>>> master
```

f1.txt

neste caso decidimos manter ambos os conteúdos ficando primeiro o que está no ramo “master” e depois o que está no ramo corrente

```
um exemplo
um exemplo COM MAIS ESTA LINHA
um exemplo que no RAMO MASTER DIVERGE
um exemplo NO RAMO testesAdHoc
```

f1.txt

... usando ferramenta de apoio à resolução de conflitos

> **git mergetool** *neste caso está a usar a “opendiff”*

The screenshot displays the opendiff merge tool interface. At the top, two file panes are shown: `f1.txt.LOCAL.13979.txt - /Users/ptrigo/Doc` (left) and `f1.txt.REMOTE.13979.txt - /Users/ptrigo/` (right). Both panes show a conflict in the file `f1.txt`. The left pane (local branch 'testesAdHoc') contains the text: `um exemplo`, `um exemplo COM MAIS ESTA LINHA`, and `um exemplo NO RAMO testesAdHoc`. The right pane (remote branch 'master') contains the text: `um exemplo`, `um exemplo COM MAIS ESTA LINHA`, and `um exemplo que no RAMO MASTER DIVERGE`. A red line highlights the conflicting lines in both panes, and a red arrow with the number '1' points from the left pane to the right pane. Below the panes, the status bar indicates: `status: 1 difference (1 left, 1 right, 1 conflict)`. An 'Actions' menu is open at the bottom right, showing options: `Choose left`, `Choose right`, `Choose both (left first)`, `Choose both (right first)` (highlighted), and `Choose neither`.

*ramo corrente
neste caso é o ramo
“testesAdHoc”*

*ramo “remoto”
neste caso é o ramo
“master”*

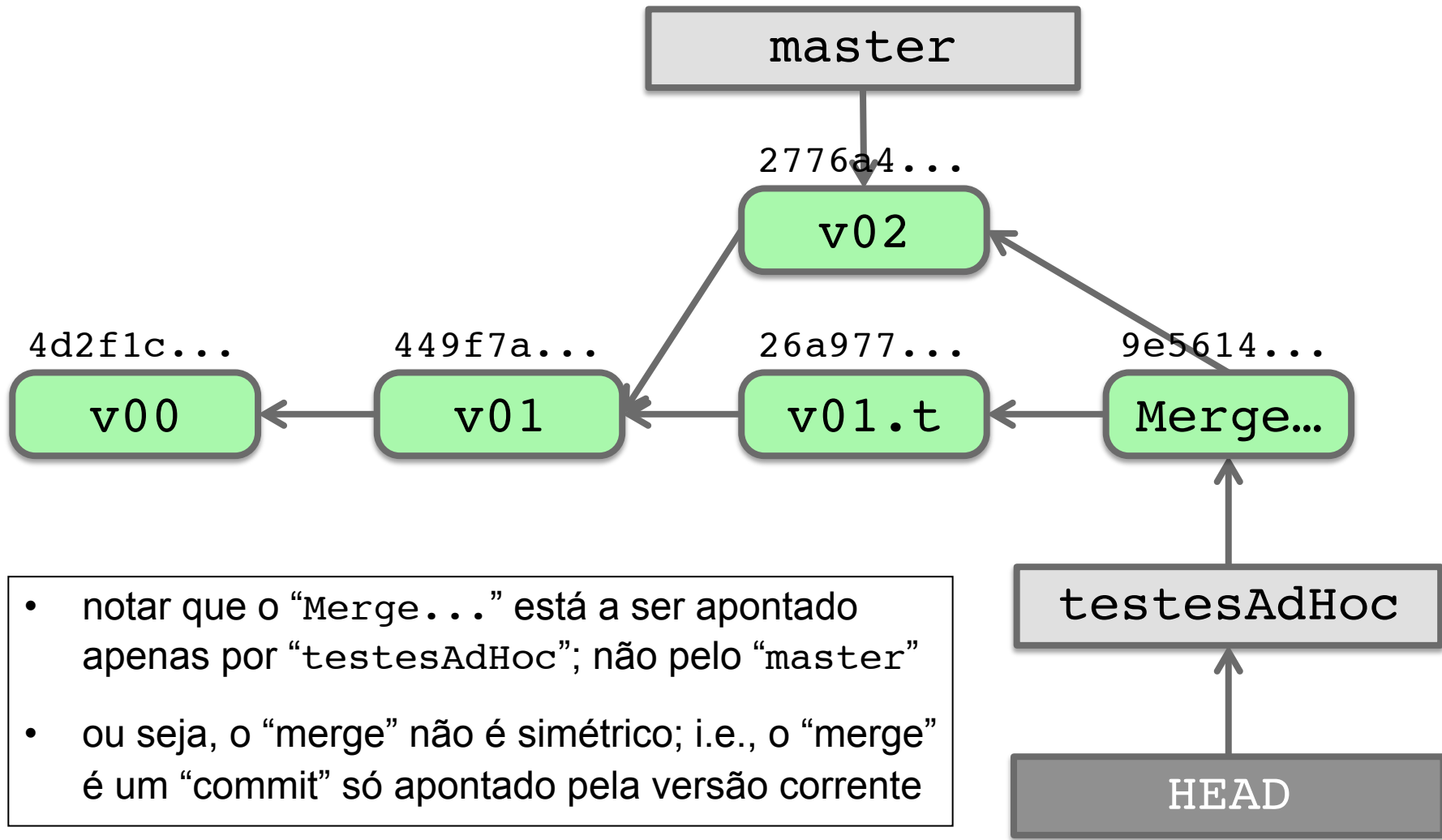
*para mostrar ferramentas suportadas e
disponíveis (na sua máquina) fazer:
git mergetool --tool-help*

... depois de resolvido o conflito fazer junção dos ramos

- Adicionar à “staging area” cada ficheiro cujo conflito foi resolvido
`git add f1.txt` (para o caso do nosso exemplo)
- Consolidar a junção (é mesma coisa que consolidar uma versão)
`git commit`
 - ... e no nosso exemplo vamos manter a mensagem de omissão do Git
- ... ver o efeito deste “merge” na reorganização dos “commit”
`git log --pretty=oneline`

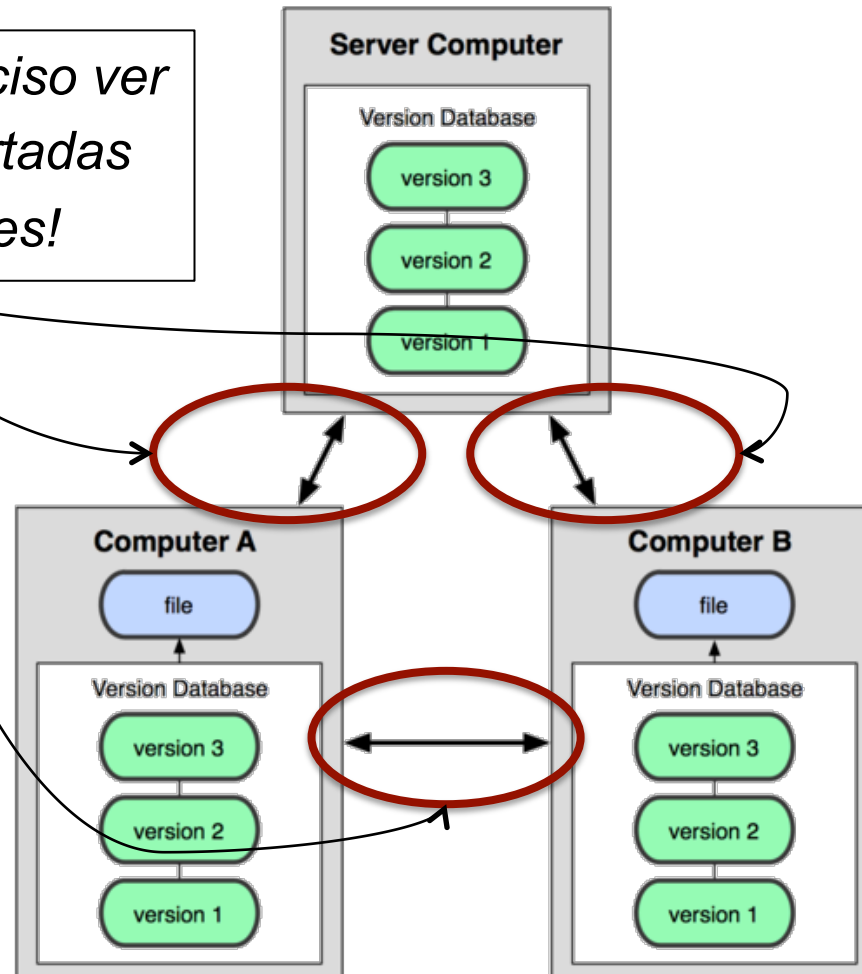
```
9e561408e5991f8b47129d0... Merge branch 'master' into testesAdHoc
2776a4e63253e40632021b5... v02 - diverge master
26a977634ba7e99502b2206... v01.t - teste ao input
449f7adf3157bf5cd626a73... v01 - o meu segundo commit
4d2f1c4276052d633453ad4... v00 - o meu primeiro commit
```

... uma ideia gráfica sobre o que sucedeu após o “merge”



Mas, trabalhar em equipa implica comunicar e partilhar!

*... portanto é preciso ver
como são suportadas
estas ligações!*



... comunicar entre o “Git local” e o “Git remoto”

- Agora já temos uma ideia clara do funcionamento do Git
 - ... mas apenas temos analisado a sua “perspectiva local”
 - ou seja, todas as operações têm sido feitas na “minha máquina”
- Mas, para trabalhar em equipa é preciso comunicar e partilhar
 - as versões precisam de “ser geridas entre os membros” da equipa
- O Git suporta 4 protocolos de comunicação
 - “**local**”, onde o repositório remoto é apenas uma outra pasta no disco; por exemplo o acesso a um sistema de ficheiros partilhado
 - **HTTP**, onde o repositório está disponível via Web Server (e.g., Apache)
 - **Git**, servidor incluído no Git que escuta num porto específico (9418) e que fornece um serviço idêntico ao do “ssh” mas sem autenticação
 - “**secure shell**” (“ssh”), protocolo usua, geralmente disponível em Linux
- ... vamos analisar os protocolos “*local*” e o *HTTP*

Protocolo “*local*”

- Atualmente é simples partilhar via serviço de armazenamento
 - e.g., DropBox, ou GoogleDrive
- Só é necessário ter uma pasta (espaço) remota gerida nesse serviço
 - iniciar o Git nessa pasta remota, e depois,
 - ... fazer “**push**” da pasta local para a pasta remota
 - ... fazer “**fetch**” e “**pull**” da pasta remota para a pasta local
- ... no repositório local é preciso registar o remoto
 - git remote add** [shortname] [url]
 - e para obter mais informação sobre cada repositório remoto
 - git remote show** [shortname]

... o nosso exemplo na DropBox em 5 passos!

... repositório Git remoto
sem “working directory”

1

```
cd ~/Dropbox  
mkdir _Git  
cd _Git  
git init --bare projetoA.git
```

Git local

2

```
git remote add db ~/Dropbox/_Git/projetoA.git
```

3

```
git push db master  
git push db testesAdHoc
```

4

alteração das versões via “push”
por outro(s) membro(s) da equipa

5

```
git fetch db master  
git pull db testesAdHoc
```

copia últimas
atualizações para a
“working area”

faz “fetch” e “merge”
num ramo local

... o nosso exemplo na DropBox (outro processo)

... repositório Git remoto
sem "working directory"

1

```
cd ~/Dropbox  
mkdir _Git  
cd _Git
```

Git local

2

```
git remote add db ~/Dropbox/_Git/projetoA.git
```

3

```
git clone --bare . ~/Dropbox/_Git/projetoA.git
```

4

*alteração das versões via "push"
por outro(s) membro(s) da equipa*

5

```
git fetch db master  
git pull db testesAdHoc
```

***réplica total vai para
repositório remoto***

Protocolo HTTP – o caso do GitHub

`https://github.com/`



`https://github.com/userName?tab=repositories`



... criar novo repositório no GitHub

GitHub (Git remoto) – vamos criar repositório e partilhar com o Git local

The screenshot shows the GitHub 'Create new repository' interface. At the top, there's a 'PUBLIC' label and a repository icon. Below this, the 'Owner' is set to 'paulo-trigo' and the 'Repository name' is 'teste_Git', with a green checkmark indicating it's valid. A red oval highlights these two fields. Below the name field, a message says: 'Great repository names are short and memorable. Need inspiration? How about **drunken-octo-robot**.' The 'Description (optional)' field contains the text 'usado para testar integração com Git'. Underneath, there are two radio button options: 'Public' (selected) and 'Private'. The 'Public' option is described as 'Anyone can see this repository. You choose who can commit.' The 'Private' option is described as 'You choose who can see and commit to this repository.' Below these, there's a checkbox for 'Initialize this repository with a README', which is currently unchecked. A note below it says 'This will allow you to git clone the repository immediately.' At the bottom, there's a dropdown for 'Add .gitignore: None' and a large green 'Create repository' button.

... indicações do GitHub após criar o repositório

GitHub (Git remoto) – indicações para comunicar e partilhar com o Git local

Create a new repository on the command line

```
touch README.md  
git init  
git add README.md  
git commit -m "first commit"
```

*no nosso exemplo
isto já foi feito!*

```
git remote add origin https://github.com/paulo-trigo/teste_Git.git  
git push -u origin master
```

Push an existing repository from the command line

```
git remote add origin https://github.com/paulo-trigo/teste_Git.git  
git push -u origin master
```

***falta fazer
isto!***

... do Git (local) para o GitHub – os ramos após “push”

Git local

```
git remote add gh https://github.com/paulo-trigo/teste_Git.git  
git push gh master  
git push gh testesAdHoc
```

GitHub (Git remoto)

The screenshot shows the GitHub interface for the repository 'paulo-trigo / teste_Git'. The 'Branches' tab is selected, showing a list of branches. The 'master' branch is the base branch, last updated 21 hours ago. The 'testesAdHoc' branch is shown as 2 ahead and 0 behind master, last updated 11 hours ago. A red box highlights the 'testesAdHoc' branch entry. A callout box points to the 'Branches' tab with the text 'os ramos (“Branches”); e o ramo corrente'.

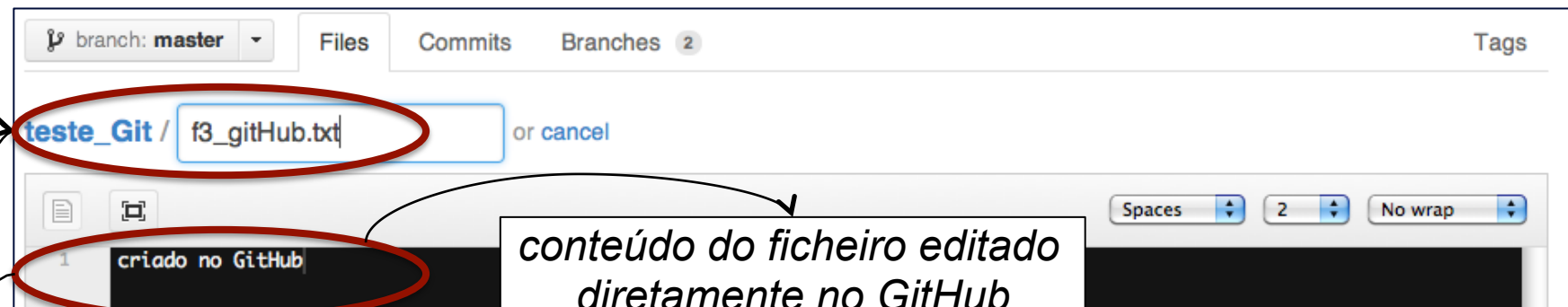
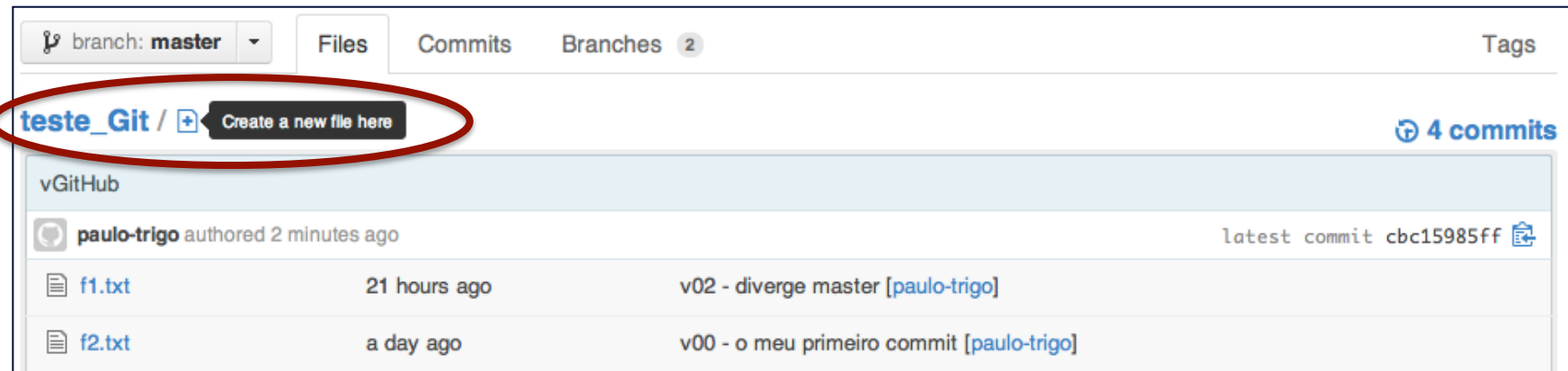
Branches

Showing 1 branch not merged into master. [View merged branches](#)

Branch	Status	Updated	By
master	Base branch	Last updated 21 hours ago	paulo-trigo.
testesAdHoc	2 ahead, 0 behind	Last updated 11 hours ago	paulo-trigo.

[Delete branch](#) [Compare](#)

... no GitHub adicionar um ficheiro (no ramo “master”)



ficheiro, “f3_github.txt” adicionado diretamente no GitHub

... do GitHub para o Git (local)

GitHub (Git remoto)



branch: master Files Commits Branches 2 Tags

teste_Git / + 4 commits

vGitHub

paulo-trigo authored 6 minutes ago latest commit cbc15985ff

f1.txt	21 hours ago	v02 - diverge master [paulo-trigo]
f2.txt	2 days ago	v00 - o meu primeiro commit [paulo-trigo]
f3_github.txt	6 minutes ago	vGitHub [paulo-trigo]

*no Git local “puxar”, do GitHub, o ramo “master” que agora lá está;
ou seja, que agora tem também o ficheiro “f3_github.txt”*

Git local

```
git pull gh master
```

... os “commit” que agora existem (vistos no Git local)

Git local

```
git pull gh master
```

este “commit” foi gerado pelo “pull” pois faz “fetch” e “merge” no ramo local

```
git log --pretty=oneline
```

```
62393cd00635ad71e0... Merge branch 'master' of https://github.co
```

```
cbc15985ff730ee1c9... vGitHub
```

```
9e561408e5991f8b47... Merge branch 'master' into testesAdHoc
```

```
2776a4e63253e40632... v02 - diverge master
```

```
26a977634ba7e99502... v01.t - teste ao input
```

```
449f7adf3157bf5cd6... v01 - o meu segundo commit
```

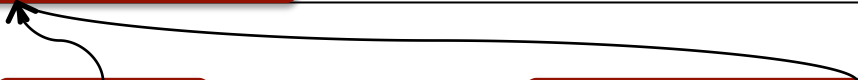
```
4d2f1c4276052d6334... v00 - o meu primeiro commit
```

este “commit” foi gerado no GitHub ao adicionar o ficheiro f3_gitHub.txt

Aspetos adicionais – associar etiquetas (“tagging”)

```
git log --pretty=oneline
```

```
62393cd00635ad71e0... Merge branch 'master' of https://github.co
cbc15985ff730ee1c9... vGitHub
9e561408e5991f8b47... Merge branch 'master' into testesAdHoc
2776a4e63253e40632... v02 - diverge master
26a977634ba7e99502... v01.t - teste ao input
449f7adf3157bf5cd6... v01 - o meu segundo commit
4d2f1c4276052d6334... v00 - o meu primeiro commit
```



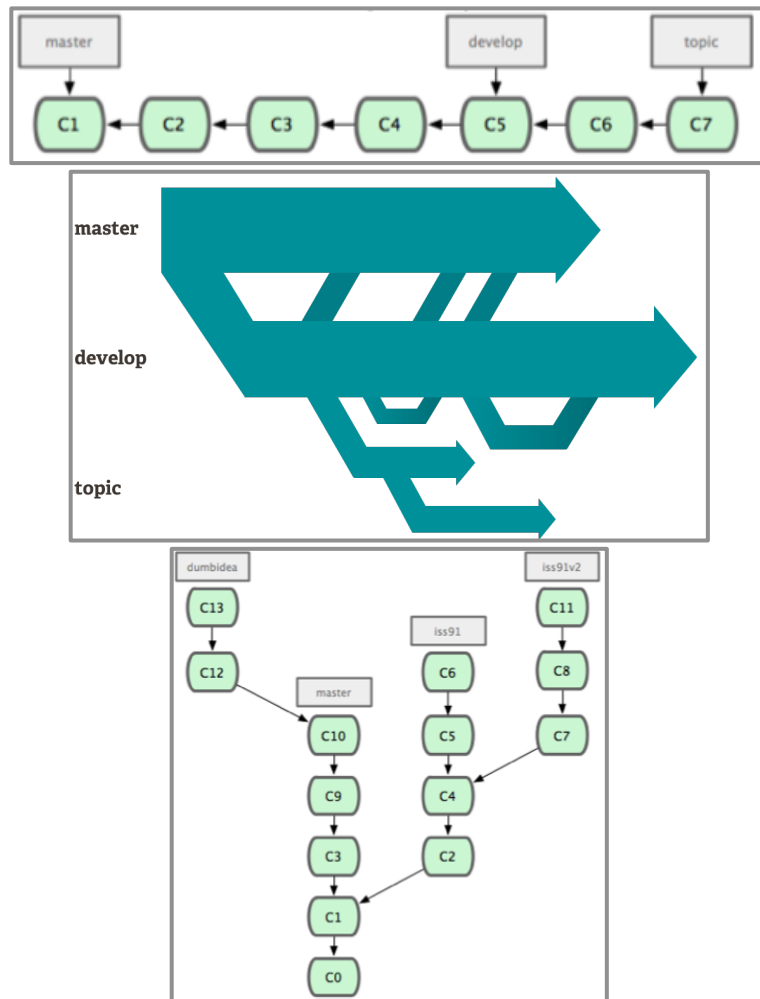
```
git tag -a tag_v00 -m "tagging" 4d2f1c4276052d633453a...
```

***uma etiqueta (“tag”) é como um ramo que nunca se altera;
é um apontador para um “commit” específico***

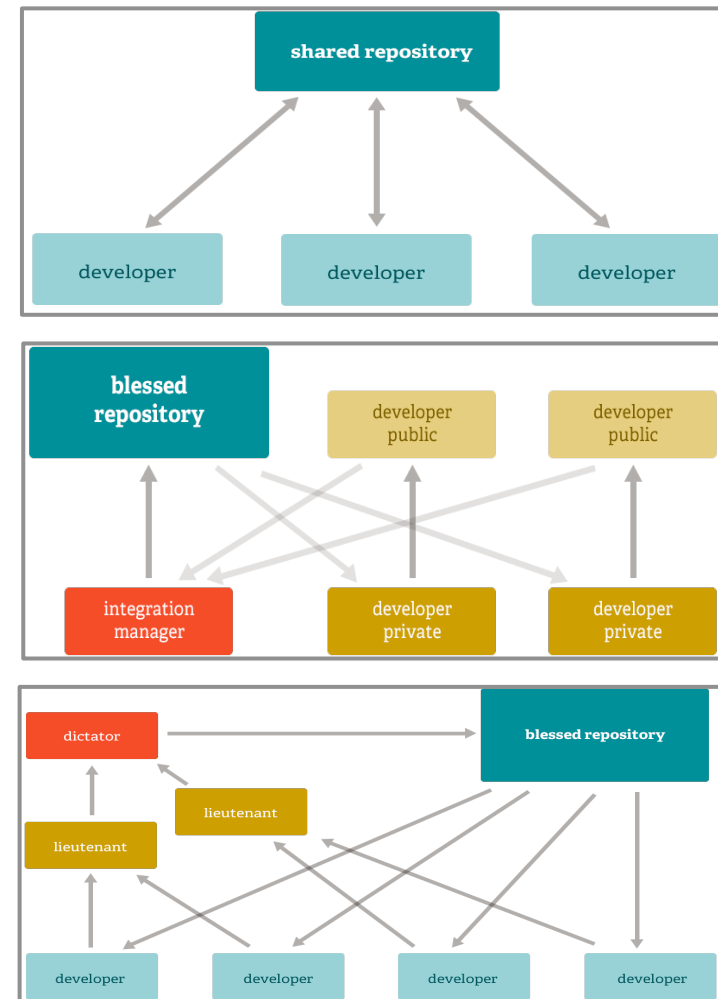
para apresentar a lista de etiquetas fazer: `git tag`
para obter a versão associada a uma etiqueta x fazer: `git checkout x`

Padrões de utilização do Git

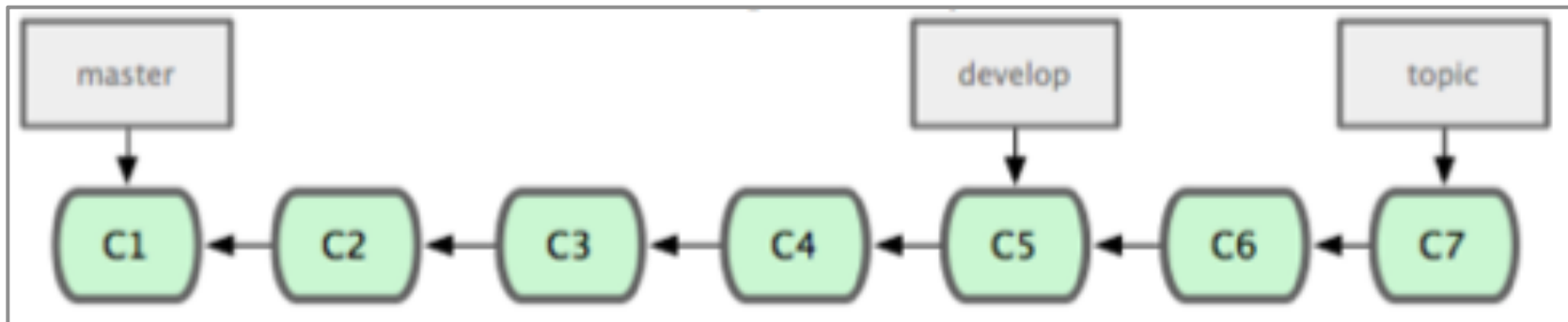
Gestão de ramos (“branches”)



Arquitetura de repositórios



Gestão de ramos (“branches”) – *ramo único*

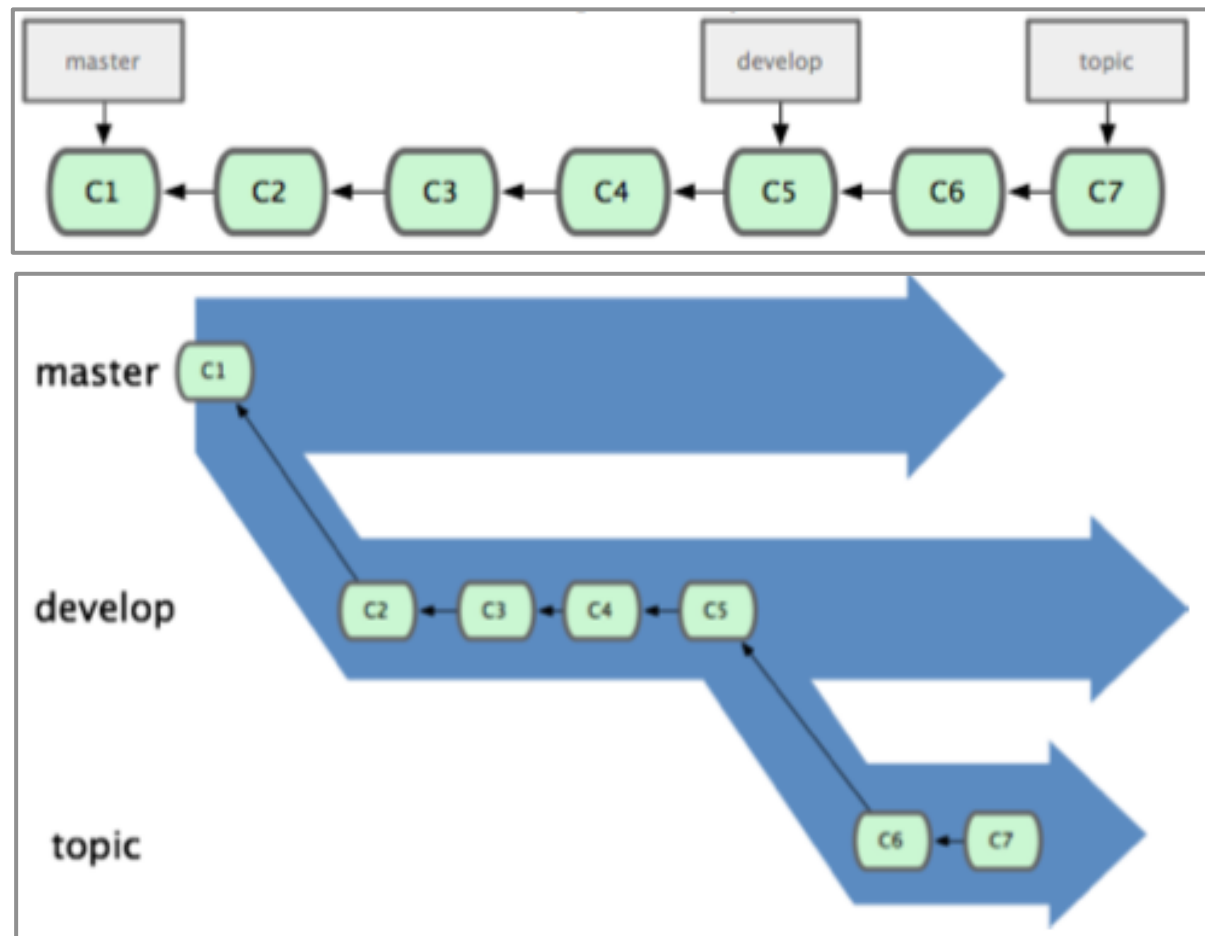


manutenção de 1 único ramo, “master”, que apenas contém versões estáveis; daí deriva outro ramo único, “develop”, onde se testa a estabilidade; e daí deriva outro ramo único, “topic” onde se exploram alternativas.

- “merge” de ramos é simples; consiste apenas em avançar os apontadores
- este “merge” em geral não gera conflitos que exijam ação do utilizador.

... ver aquele ramo como “*múltiplos silos*”

“*multi-silo long-running-branches*”



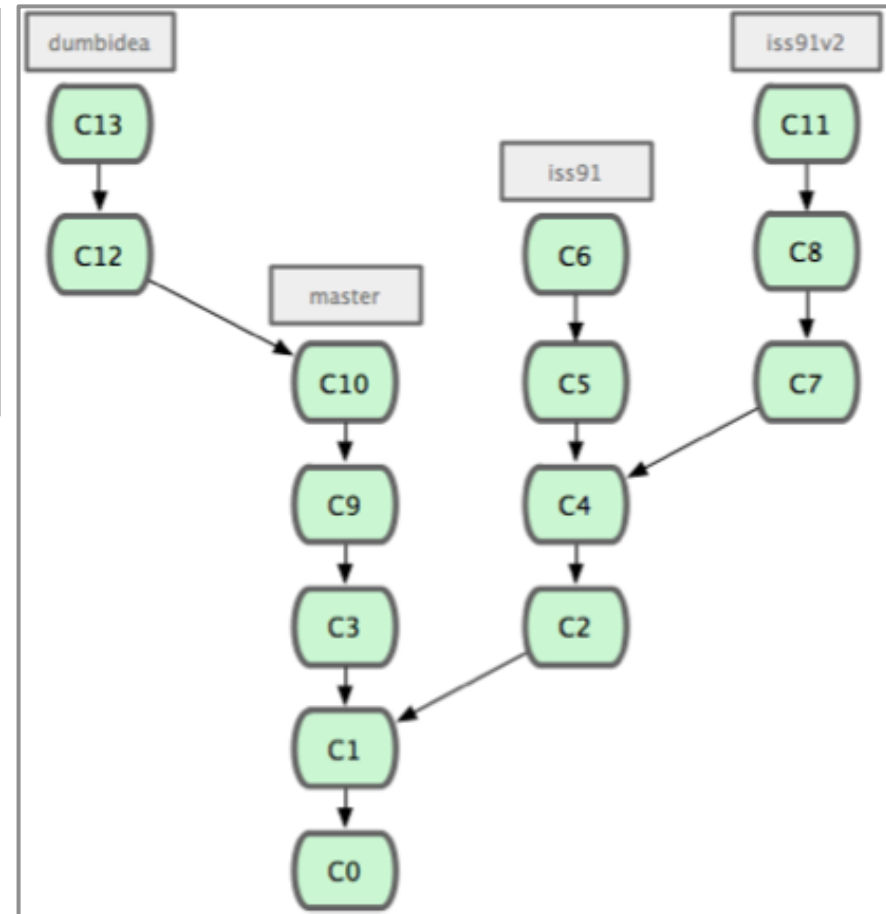
- os ramos representam níveis de estabilidade
- ... *um ramo é um “silo”*
- quando “mais estável” mais perto do “master”

Gestão de ramos (“branches”) – *ramo baseado em tópico*

- um tópico é um ramo de “vida curta”
- ... *tópico desenvolve-se num “silo”*
- é criado para explorar um cenário
- pode haver vários num curto tempo
- ... *e.g., vários no mesmo dia*

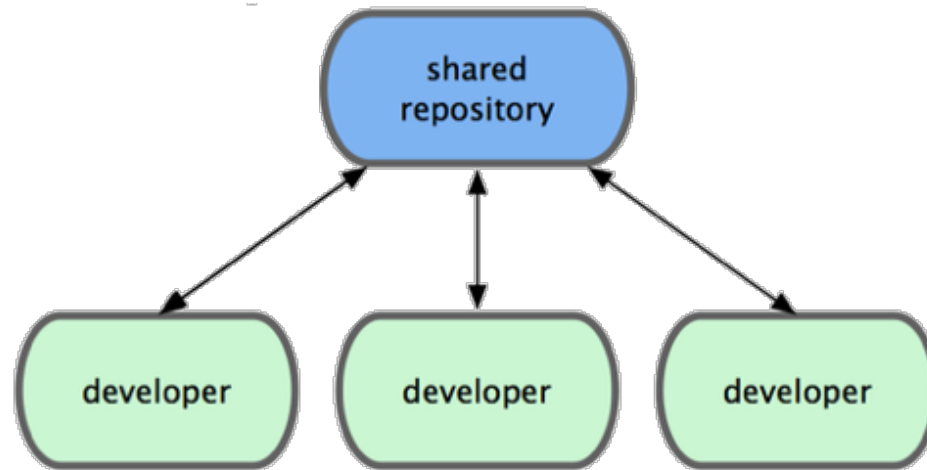
notar que todos estes ramos são criados
(e manipulados) em contexto local;
criar um ramo e fazer junção entre ramos
é feito apenas no repositório Git local; não
exige comunicação com o servidor

“multi-silo short-topic-branches”



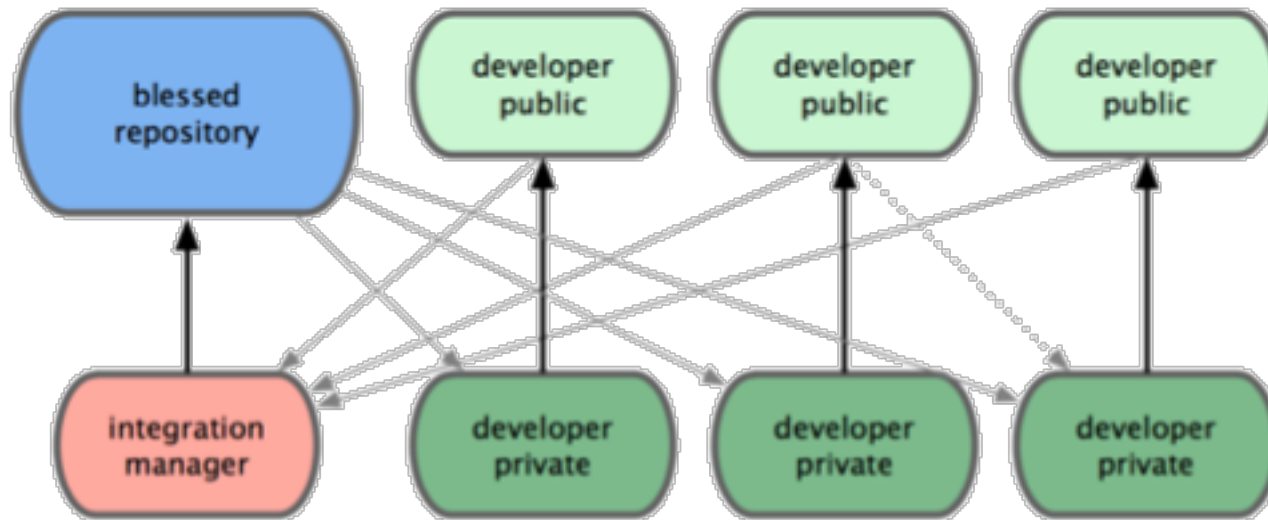
Arquitetura de repositórios – *um repositório comum*

- um único repositório comum, conhecido e compartilhado por todos
- ... a sincronização é sempre feita via um repositório central
- ou seja, o Git pode ser usado para suportar o modelo centralizado
- ... um modelo simples e a que estamos, em geral, habituados
- é o modelo seguido ao usar o GitHub como repositório central
- ... é um modelo adequado, por exemplo, a pequenas equipes
- ... cada membro faz “push” da sua versão e “pull” da versão global



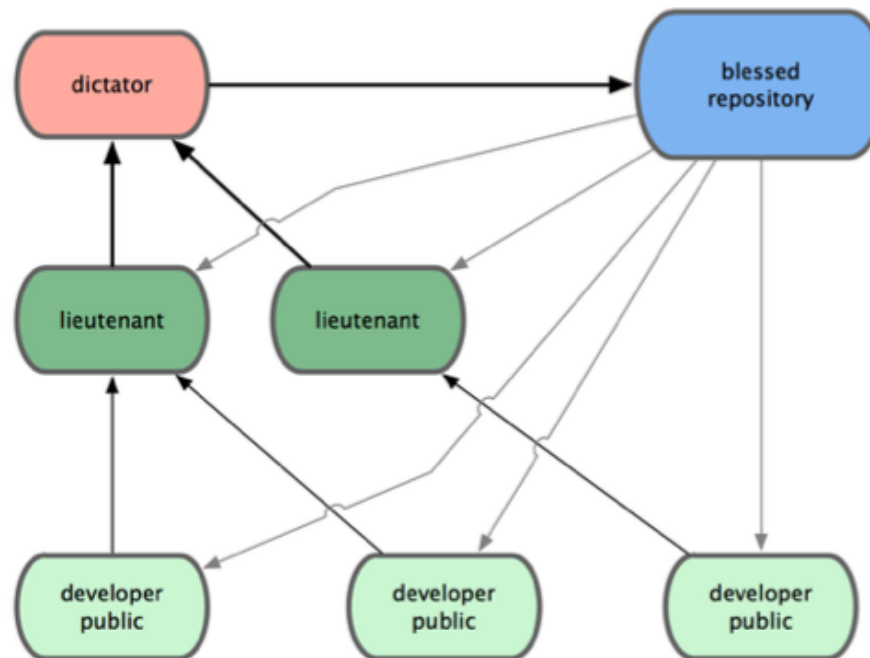
Arquitetura de repositórios – *gestor de integração*

- cada membro tem acesso de escrita ao seu repositório público
- cada membro tem acesso de leitura a todos os outros repositórios
- há um repositório que representa o “projeto oficial”
- ... cada membro faz “push” de versão e avisa o “gestor de integração”
- ... “gestor de integração” faz “pull” da versão e testa as alterações
- ... “gestor de integração” faz “push” da versão no repositório final



Arquitetura de repositórios – *multi-gestor de integração*

- generaliza o modelo (anterior) do “gestor de integração”
- ... aqui existem “vários gestores de integração”
- ... modelo também designado por “*dictator and lieutenants workflow*”
- ... cada “*lieutenant*” tem “gestor de integração” (“*benevolent dictator*”)
- é adequado a projetos de grande dimensão; e.g., “kernel” Linux



Informação adicional sobre Git

- Git está disponível em
 - <http://git-scm.com/>
- Forma simples de testar o Git
 - <http://try.github.io/levels/1/challenges/1>
- Guião simples e interessante
 - <http://rogerdudler.github.io/git-guide/>
- Clientes Git com interação gráfica
 - <http://git-scm.com/downloads/guis>
- Documentação muito completa
 - <http://git-scm.com/book>
 - ... serviu de base a esta apresentação

Algumas outras ferramentas sobre Git (e GitHub)

- Git tem integração nativa com “Aptana Studio 3”
 - <http://www.apтана.com/products/studio3/download>
 - distribuição do Eclipse com GitHub e PyDev já incorporados
- ... mais ferramentas e instruções de instalação
 - <https://wiki.appcelerator.org/display/tis/Git#Git-Windows>
- Uma ferramenta interativa que planeia integrar um “*issue tracker*”
 - <http://rhodecode.org/>
- O GitHub já integra um “*issue tracker*”
 - <https://github.com/blog/411-github-issue-tracker>