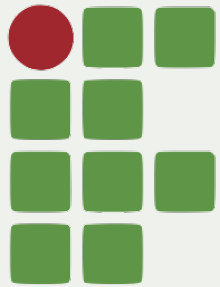
Introdução ao sistema de controle de versão GIT

POO29004 – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

http://docente.ifsc.edu.br/mello/poo

19 de fevereiro de 2020



INSTITUTO FEDERAL
Santa Catarina
Câmpus São José

Preciso de um sistema de controle de versão?

Preciso de um sistema de controle de versão?

Sistema de controle de versão (*Version Control System* – VCS)

- Ferramentas de software que **ajudam o time de desenvolvimento a gerenciar as mudanças de código** ao longo do tempo
- Protege o código fonte de catástrofes e a degradação natural provocada por erros humanos
- **Soluções como Dropbox e Google Drive não seriam adequadas** para cenários onde o VCS é necessário

Sistema de controle de versão (*Version Control System* – VCS)

- Ferramentas de software que **ajudam o time de desenvolvimento a gerenciar as mudanças de código** ao longo do tempo
- Protege o código fonte de catástrofes e a degradação natural provocada por erros humanos
- **Soluções como Dropbox e Google Drive não seriam adequadas** para cenários onde o VCS é necessário

git

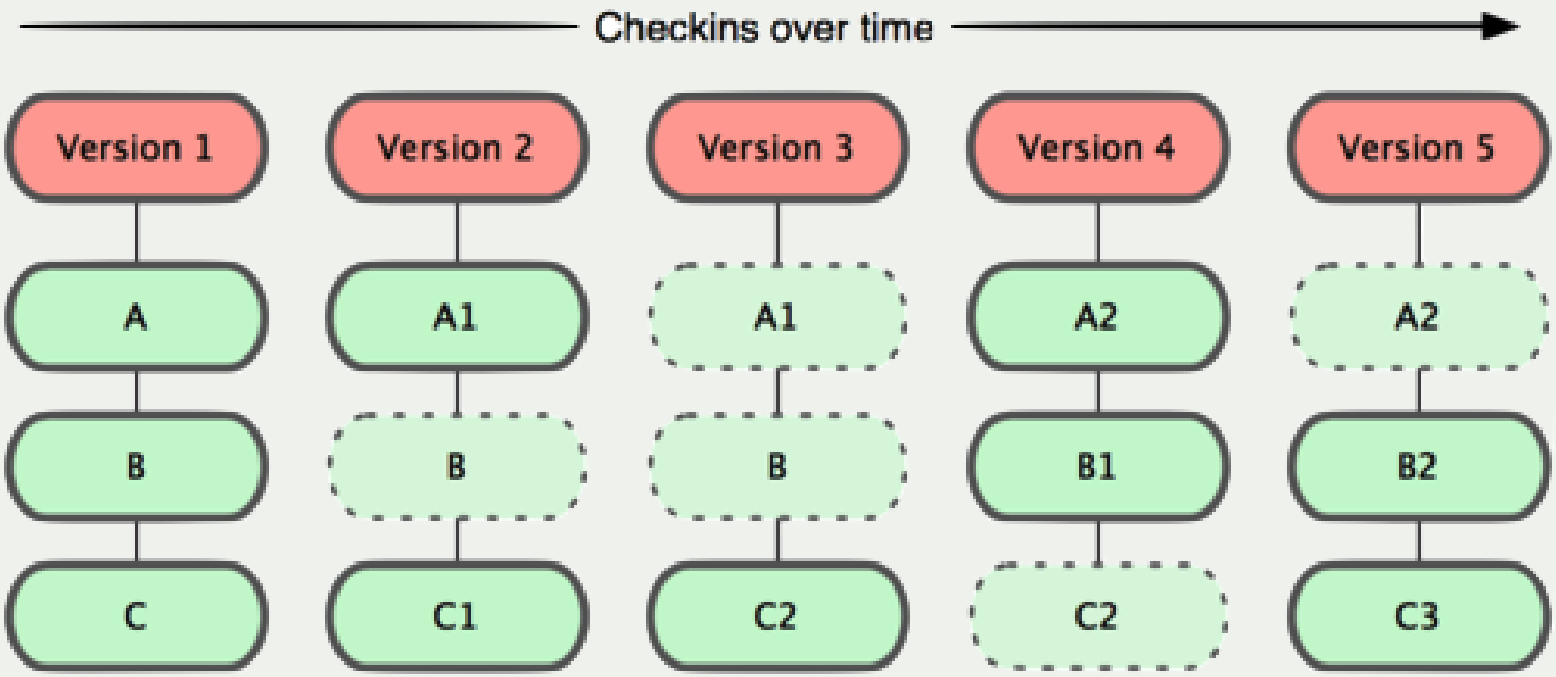
VCS distribuído criado em 2005 por Linus Torvalds para gerenciar o desenvolvimento do kernel Linux

Benefícios de um sistema de controle de versão

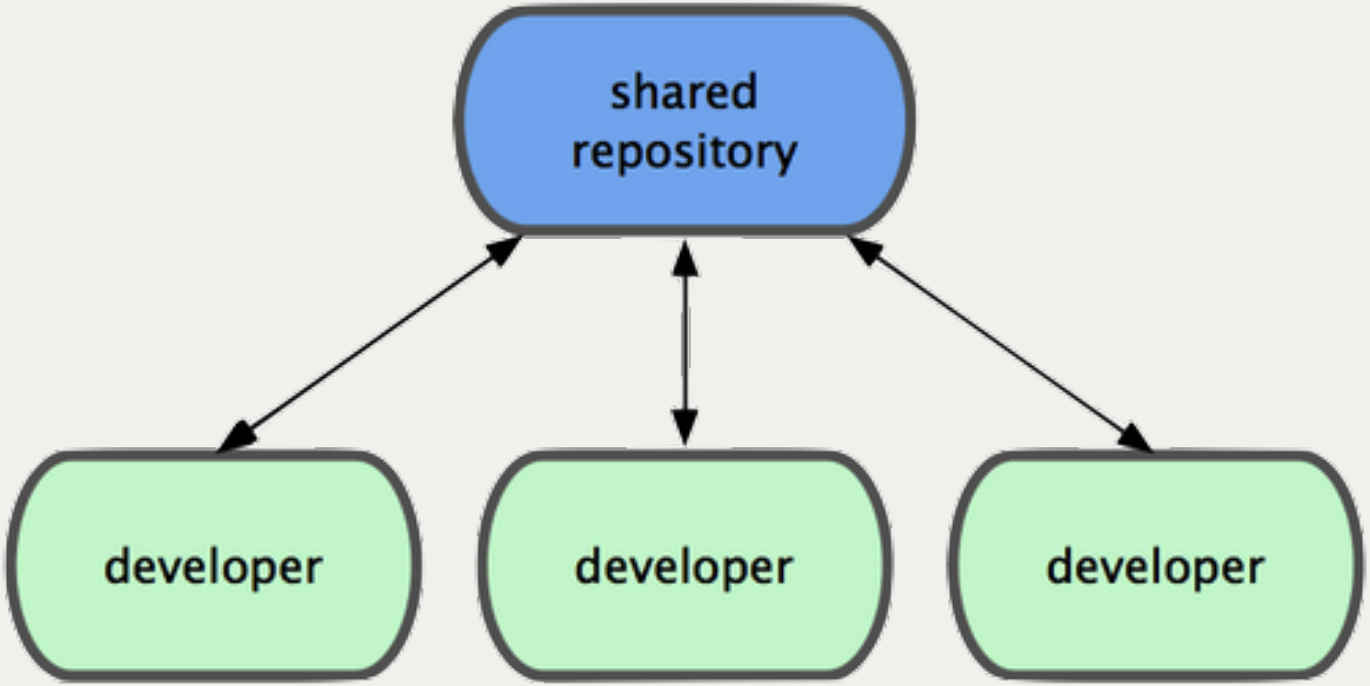
- **Histórico sobre qualquer mudança sobre arquivos do projeto**
  - Mudanças realizadas por cada indivíduo ao longo do tempo
  - Tipos de mudanças: inclusão, alteração e exclusão
- **Rastreabilidade**
  - Possível saber por quem e quando tal arquivo ou trecho do arquivo foi alterado

Snapshots

- **Foto** de todos os arquivos naquele momento
- Faz uso de *links* simbólicos

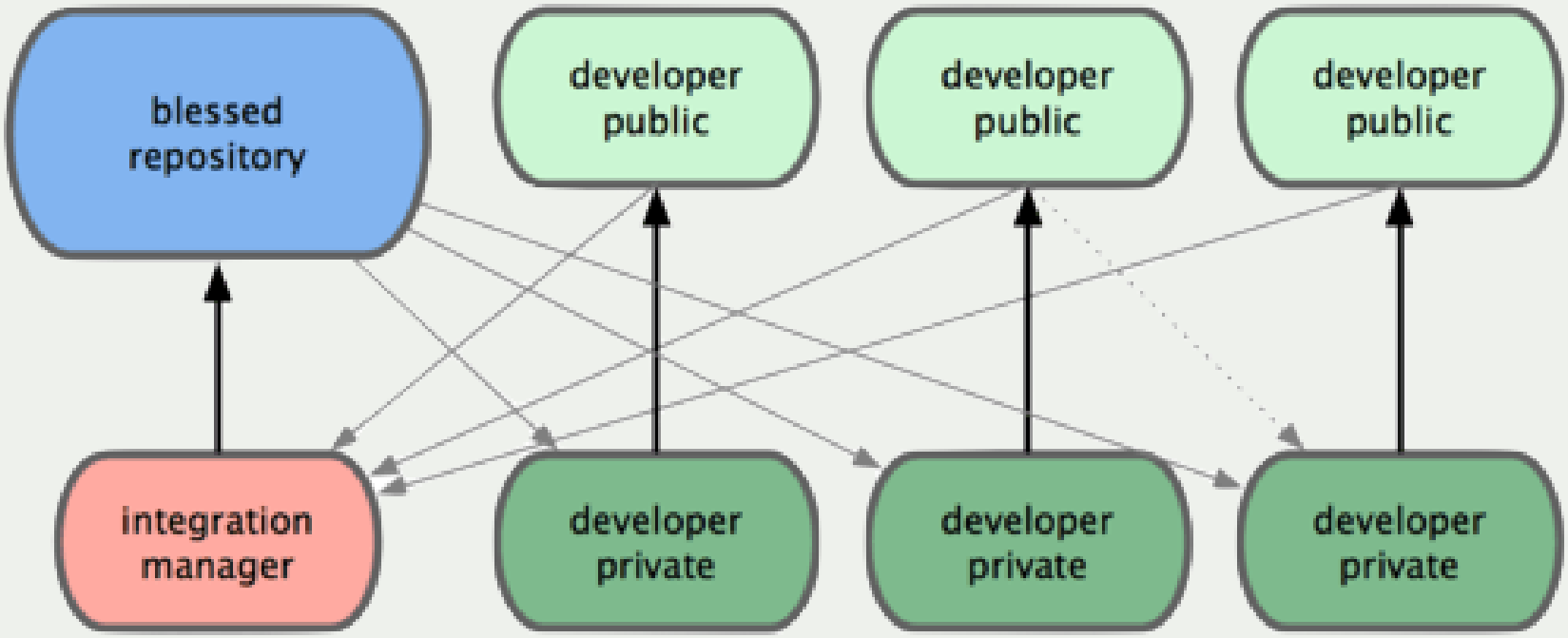


Sistema de controle de versão centralizado



- Exemplos: CVS, Subversion (svn)

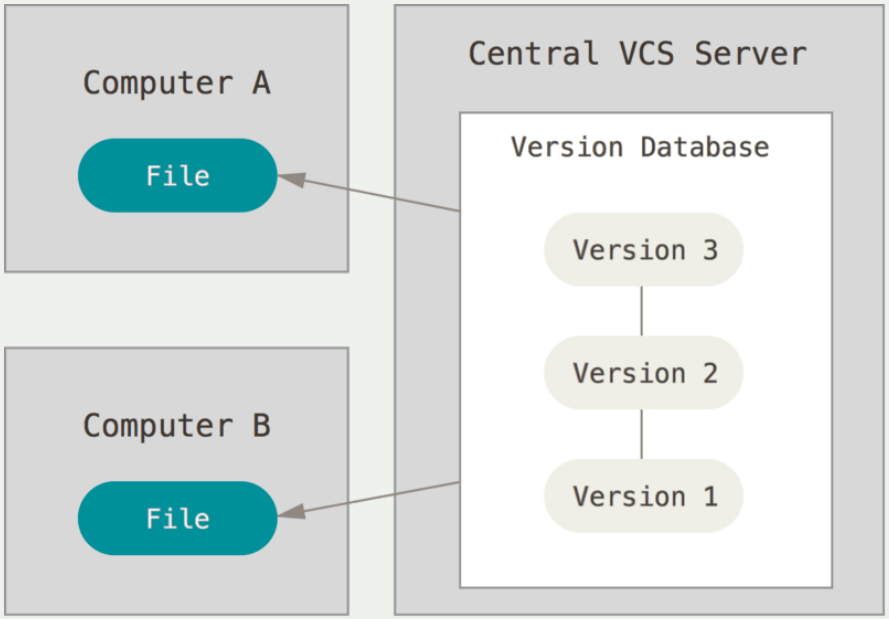
Sistema de controle de versão distribuído



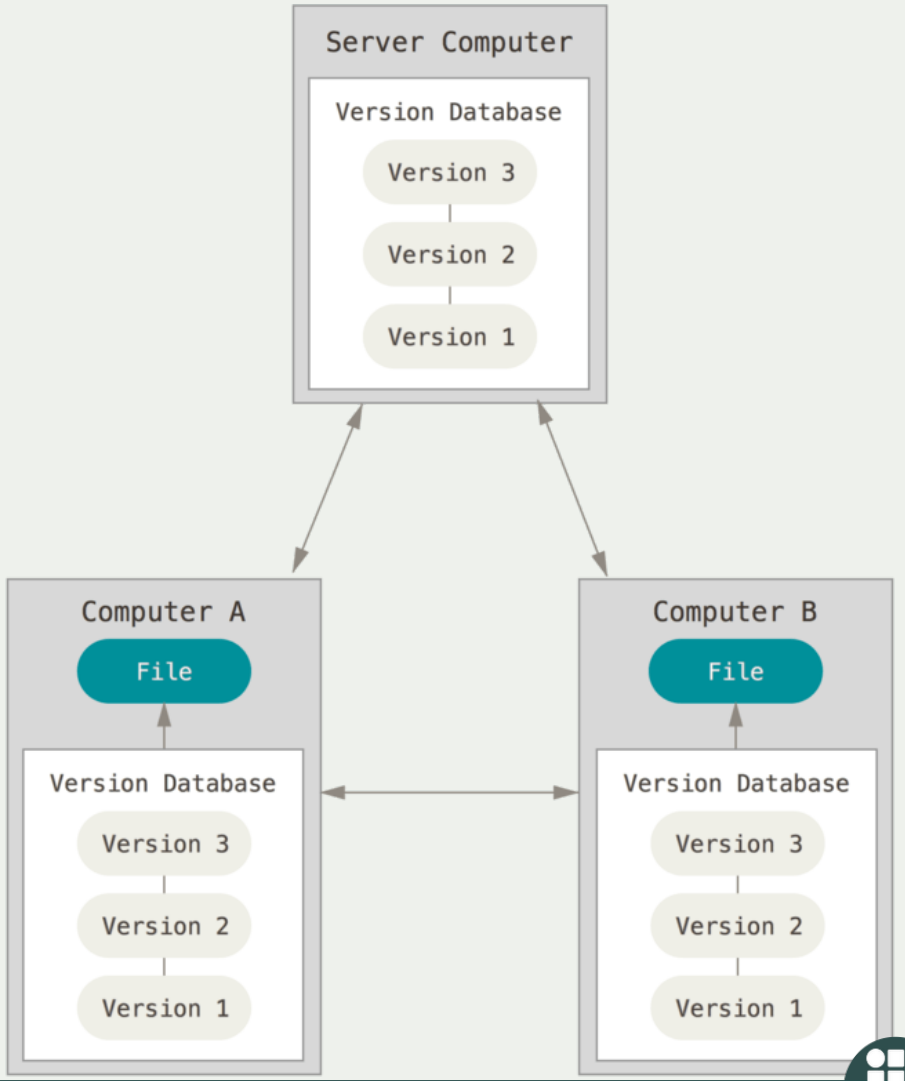
- Exemplos:
  - git, mercurial

Sistemas de controle de versão

Centralizado



Distribuído



Laboratório com git

Configurando sua identidade

- O git usa o **nome e email para rastrear as alterações** feitas por um usuário
- A configuração da identidade só precisa ser feita uma vez em seu computador pessoal
  - No Linux ou macOS as configurações do git ficam no arquivos `$HOME/.gitconfig`
  - As IDEs geralmente usam essa mesma configuração

```
1 git config --global user.name "Nome Sobrenome"
2 git config --global user.email "seu@email.com"
```


Arquivo .gitconfig

- As configurações globais do git ficam armazenadas no arquivo `.gitconfig` do diretório HOME do usuário
  - No Linux: `$HOME/.gitconfig`

- Exemplo de configurações

```
[user]
  name = Nome Sobrenome
  email = nome@email.com

# apelidos para comandos. Útil para comandos com muitos argumentos
[alias]
  tree = log --oneline --graph --decorate --all
```

```
# agora é possível usar o apelido 'tree'
git tree
# seria equivalente a executar a linha abaixo
git log --oneline --graph --decorate --all
```


Iniciando um repositório

- Ao iniciar um repositório git, será criado um subdiretório `.git` e este armazenará os dados e metadados de todas as alterações feitas no repositório

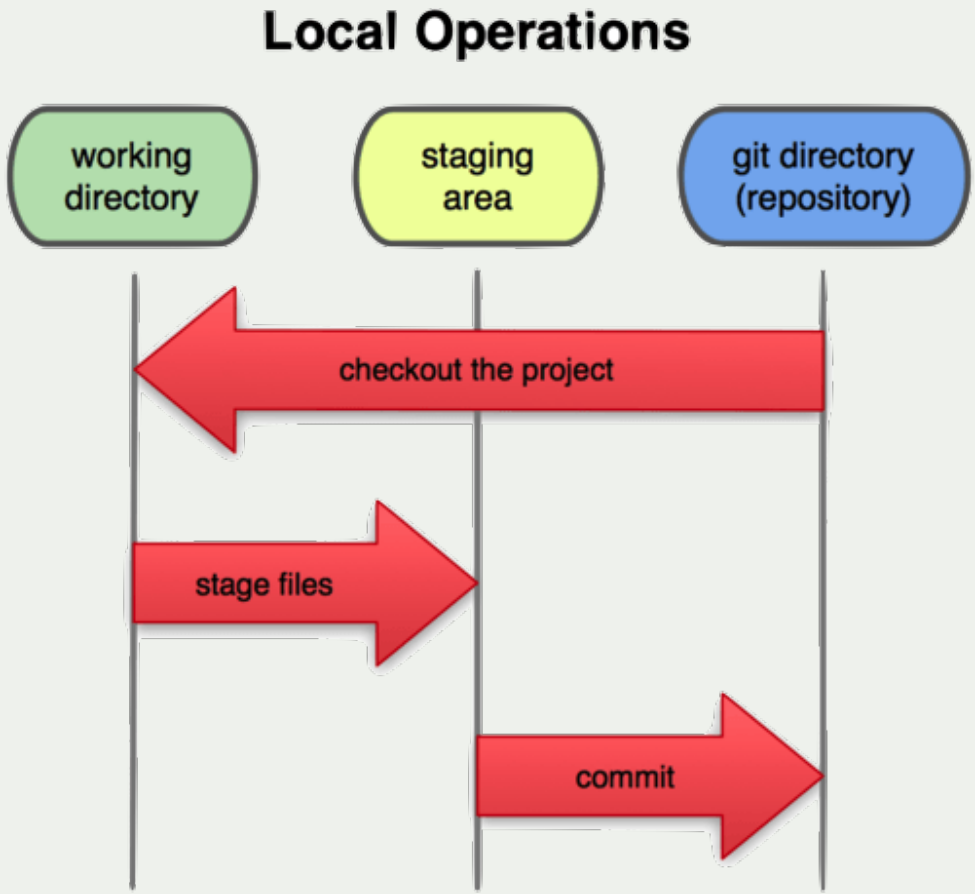
```
14 mkdir projeto
15
16 cd projeto
17
18 git init
```

- Criando um arquivo, adicionando ele no git e persistindo as mudanças

```
19 echo "Olá mundo" >> arquivo.txt
20
21 git add arquivo.txt
22
23 git commit -m "Adicionando arquivo.txt"
```

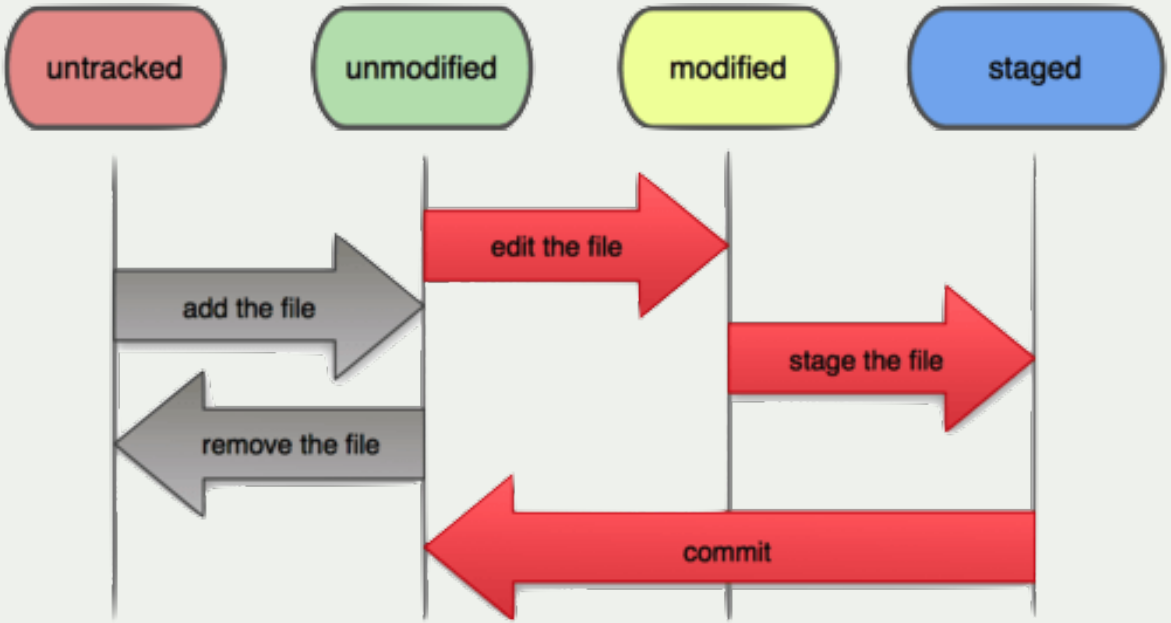

Os três estados que um arquivo pode estar

- **Consolidado** (*committed*)
  - arquivo seguramente armazenado
- **Modificado** (*modified*)
  - arquivo que sofreu mudanças mas que ainda não foi consolidado
- **Preparado** (*staged*)
  - arquivo que fora marcado para ir para o próximo *commit*




```
24 # Verificando a situação atual do repositório
25 git status
```

Ciclo de vida de arquivos em um repositório git




```
26 echo "olá" >> novo.txt # arquivo novo fica como untracked
27 git add novo.txt # arquivo é marcado (staged) para ir para o próximo
       commit
28 git commit -m "adicionando arquivo" # mudanças persistidas
29 echo "mundo" >> novo.txt # arquivo sai do staged para modified
30 git add . # marcando  (staged) todos os arquivos modificados
31 git commit -m "adicionando mundo" # mudanças persistidas
```


Markdown – extensão de arquivo .md

- Linguagem de marcação geralmente usada para documentar projetos no Github
- Guia do Github – https://guides.github.com/features/mastering-markdown/
- Guia https://www.markdownguide.org
- Editor *online* https://dillinger.io

```
32 # Título 1
33 ## Título 2
34
35 - Lista de itens
36 - Segundo item
```

Título 1

Título 2

- Lista de itens
- Segundo item

Comandos de referência

- **Marca arquivo** para ir para o próximo commit

```
37 git add arquivo.md
```

- **Desmarca um arquivo**, mantendo as mudanças no diretório de trabalho (oposto ao `git add`)

```
38 git reset HEAD arquivo.md
```

- Mostra o que **foi alterado** no diretório de trabalho e que **ainda não foi marcado**

```
39 git diff
```

- Mostra o que **foi alterado e marcado**, ou seja, o que vai para o próximo commit

```
40 git diff --staged
```

Comandos de referência

- Visualizando o histórico de commits

```
41 git log --oneline --graph --decorate --all
```

- Desfazendo alterações de um arquivo (voltar para o último *commit*)

```
42 git checkout -- arquivo.md
```

- Restaurando a versão de um arquivo de um *commit* específico

```
43 git checkout [commit] -- arquivo.md
```

- Limpa a área de marcação e reescreve toda a árvore do diretório de trabalho a partir do commit especificado (cuidado!)

```
44 git reset --hard [commit]
```


Comandos de referência

- Excluindo um arquivo do diretório de trabalho e do repositório

```
45 git rm arquivo.md
46 git commit -m "Removendo arquivo"
```

- Excluindo um arquivo do repositório, porém mantendo-o no diretório de trabalho

```
47 git rm --cache arquivo.md
48 git commit -m "Removendo arquivo do índice do git, porém mantendo-
       o no diretório"
```

- Renomeando arquivo e marcando ele para ir para o próximo *commit*

```
49 git mv nome-antigo nome-novo
50 git commit -m "Renomeando arquivo"
```


Ramos – branches

Comandos para trabalhar com ramos (*branch*)

- Listando os ramos existentes

```
51 git branch
```

- Criando um novo ramo

```
52 git branch [nome do ramo]
```

- Alterando para outro ramo

```
53 git checkout [nome do ramo]
```

- Criando um novo ramo a partir de um *commit* específico

```
54 git checkout [commit] -b [nome do ramo]
```

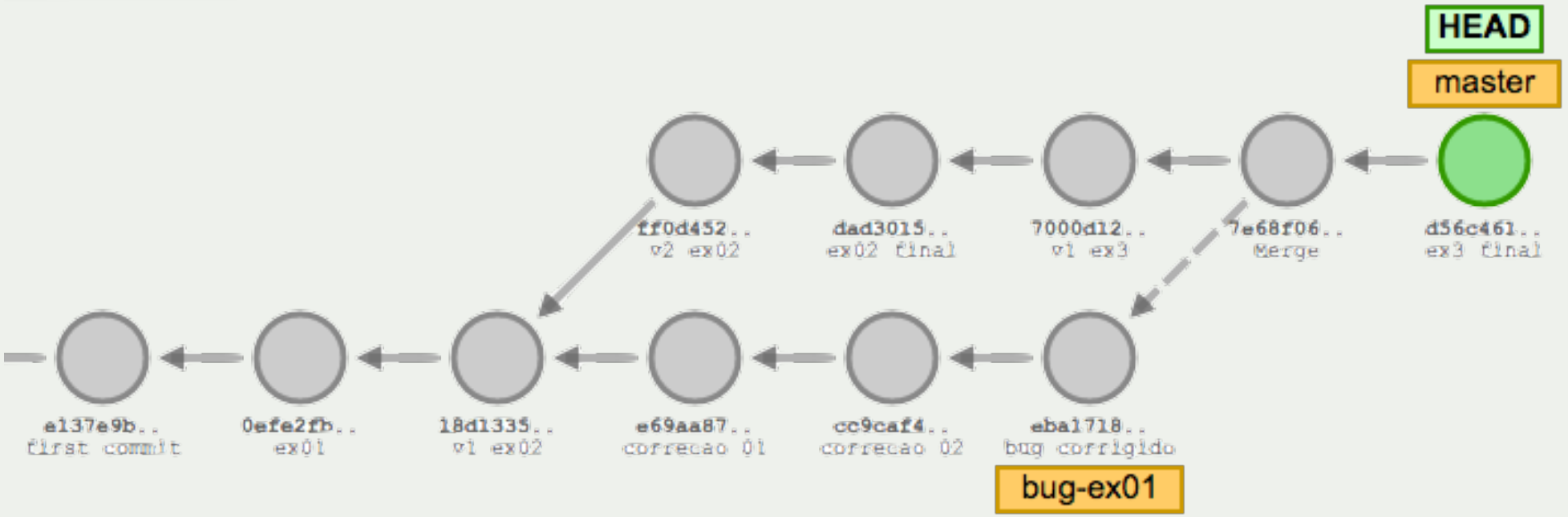
- Mesclando o conteúdo do ramo com o diretório de trabalho atual

```
55 git merge [nome do ramo] # ou git rebase [nome do ramo]
```

Prática com ramos

Prática com ramos

- Faça uso do *site* https://git-school.github.io/visualizing-git/#free e gere a seguinte árvore de *commits*
  - Use somente os comandos: commit, branch, checkout e merge



Repositórios remotos

Trabalhando com repositórios remotos

- Criando repositório local

```
56 git init
57 git add arquivo.md
58 git commit -m "Iniciando repositório"
```

- Adicionando repositório remoto com o apelido “origin”

```
59 git remote add origin https://site-remoto/repositorio.git
```

- Enviando os commits locais para o repositório remoto

```
60 git push -u origin master
```

- Sincronizando repositório local a partir do repositório remoto

```
61 # Obtém os arquivos do repositório remoto, porém não mescla
62 git fetch origin
63 # obtém e mescla qualquer commit de qualquer ramo no repositório
       remoto
64 git pull
```


Clonando um repositório remoto

- É possível clonar um repositório remoto em seu computador pessoal

```
65 git clone https://github.com/emersonmello/modelos-latex
```


Boas práticas para desenvolvimento de software & git

- **Escreva mensagens claras para os *commits***
  - Faça um resumo daquilo que resolveu ou desenvolveu em uma única sentença
- **Faça *commits* frequentemente**
  - Commits ficarão pequenos e facilitará o compartilhamento e resolução de possíveis conflitos durante os (*merges*)
- **Nunca faça *commit* de um trabalho inacabado**
  - Divida seu trabalho em pequenos problemas, resolva-os e teste-os. Somente após isso faça um *commit*.
- **Faça uso de ramos (*branchs*)**
  - No ramo *master* estará somente o código validado e funcional. Novas funcionalidades ou correções de *bugs* devem ser feitas em ramos separados. Isso facilita o trabalho colaborativo

Materias para consulta

1. https://git-school.github.io/visualizing-git
2. https://learngitbranching.js.org/
3. http://rogerdudler.github.io/git-guide/index.pt_BR.html
4. https://try.github.io
5. https://git-scm.com/book/pt-br/
   - Leia capítulos 1, 2 e 3
6. https://www.atlassian.com/git/tutorials
7. https://br.udacity.com/course/how-to-use-git-and-github--ud775/
8. https://www.atlassian.com/git/tutorials/comparing-workflows

Git explicado pelo site xkcd.com

- **A:** Esse é o git. Ele registra o trabalho colaborativo por meio de um modelo de árvore da bela teoria de grafos distribuídos
- **B:** Legal. Como podemos usá-lo?
- **A:** Não sei. Memorize esses comandos e os digite para sincronizar. Se der erro, salve seu trabalho em outro local qualquer, apague o projeto e faça o download novamente do projeto original