CTEDS

Capacitação Tecnológica em Engenharia e Desenvolvimento de Software

Controle de Versão

(com Git)

Professor: Bruno Albertini [PCS/EPUSP] *balbertini@usp.br*

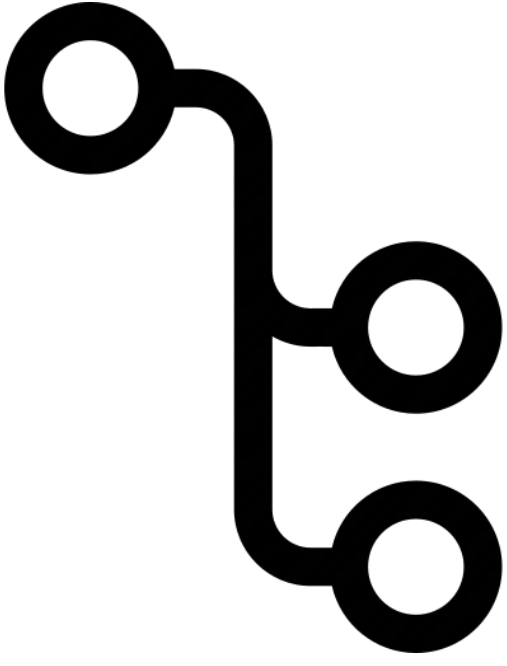
24 de Agosto de 2022

O que é controle de versão?

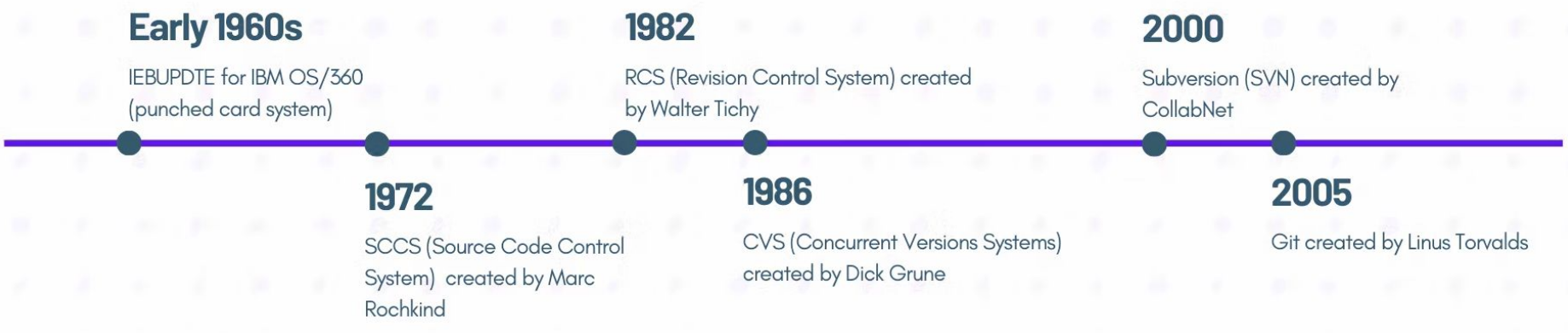
Introdução

O que é o controle de versão?

- AKA controle de (revisão|código(fonte)+) ou VCS
- Consiste em:
  - Rastrear mudanças espaciais e temporais
  - Gerenciar mudanças no software (release, tag, bug fix)
- Parte importante do fluxo DevOps
  - Melhoramento contínuo focado no cliente
  - Integração entre desenvolvimento e operação
  - Desenvolvimento distribuído (colaborativo)
  - Resolução de conflitos
- Implementação
  - Evitar o *locking*
  - Manter informações organizadas (i.e. algum tipo de BD)



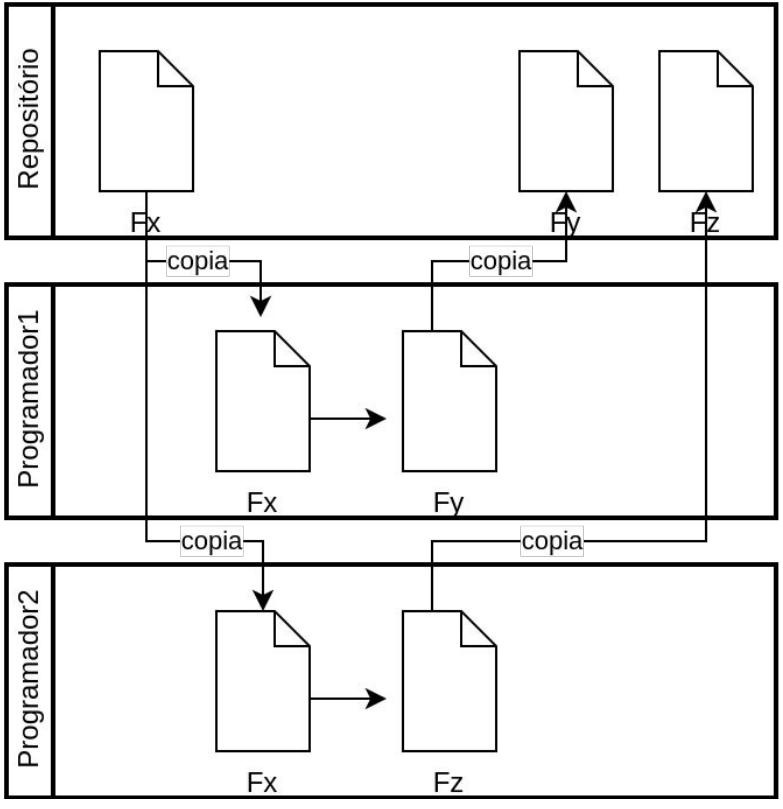
Histórico

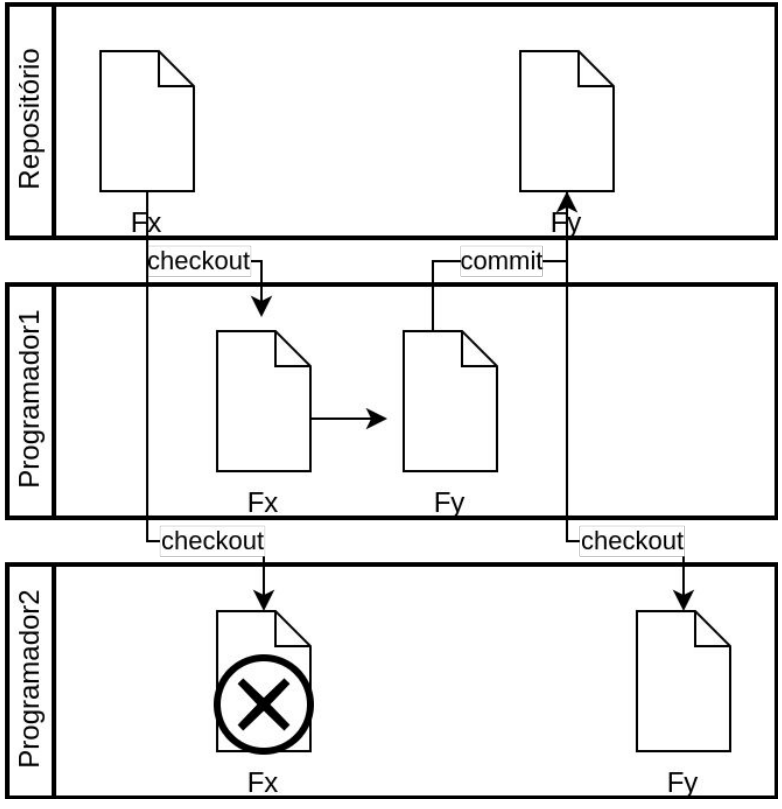


Fonte: [1]

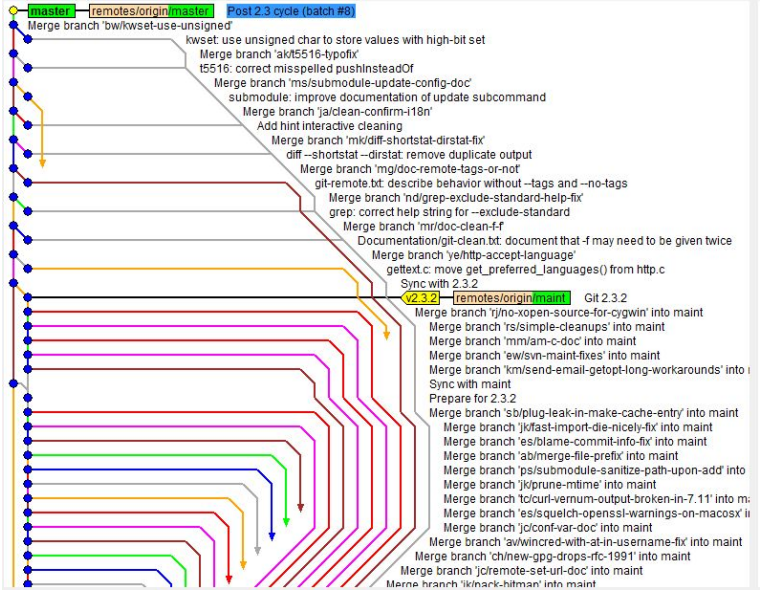
Dá pra desenvolver SW sem VCS?

Controles *naive*





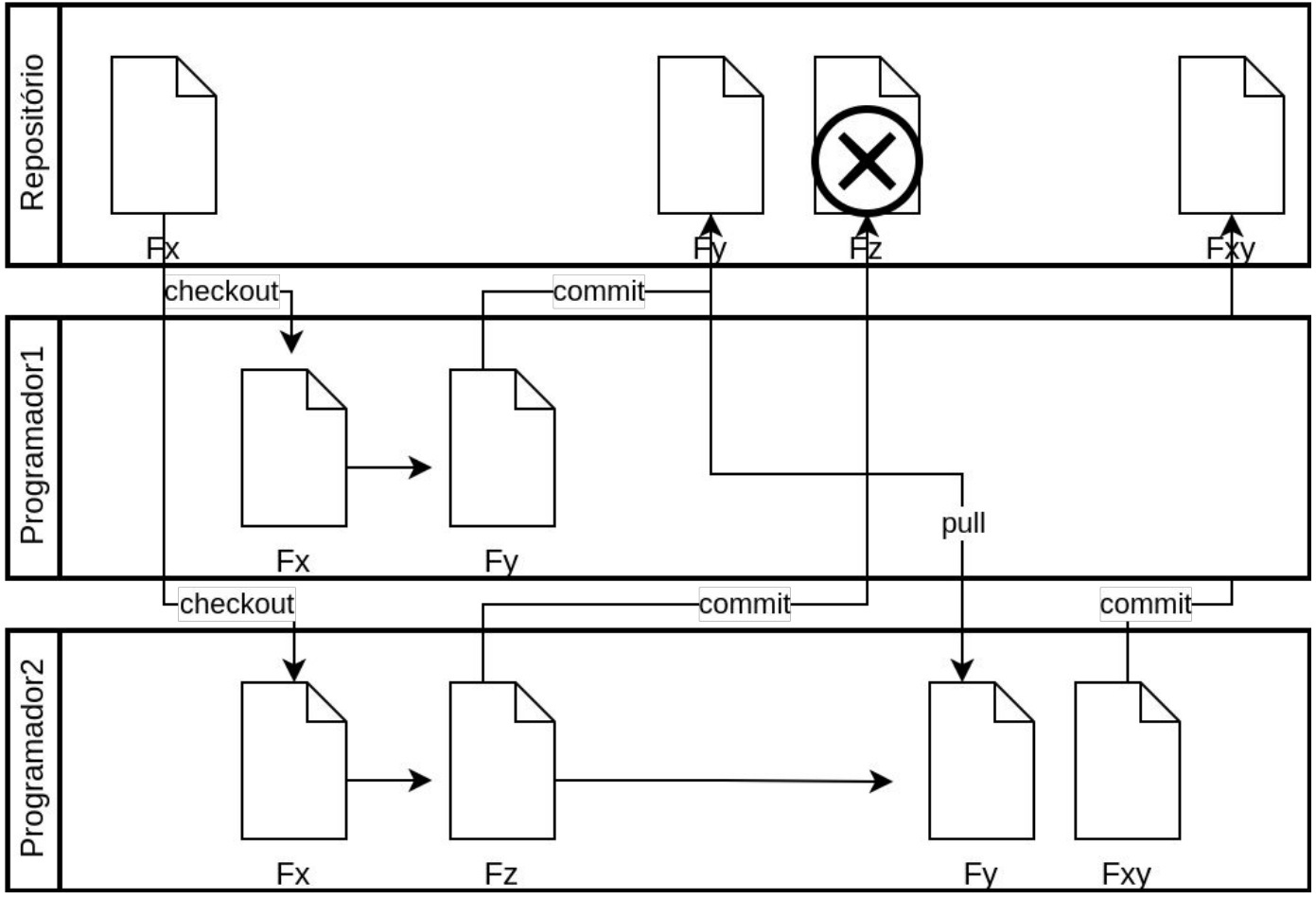
Características de um bom VCS



Fonte: miro.medium.com

1. Histórico completo de cada arquivo desde sua criação
   a. Criação, apagamento, mudanças de nome e de local
   b. Mudanças no arquivo (internamente)
   c. Registro temporal (data) e do usuário
   d. Motivo da modificação
2. Branch e Merge
   a. Fluxos paralelos de desenvolvimento
3. Rastreabilidade
   a. Muito ligado ao 1 acima
   b. Permite ferramentas de gerenciamento de projeto e bug tracking (e.g. Jira)

Características de um bom VCS



SCM - *Source Code Management*

Sistema de Controle de Versão (VCS) focado em software!

Local (monousuário)

- RCS (Revision Control System)
- SCCS (Source Code Control System)

Cliente-servidor

- CVS (Concurrent Versions System)
- SVN (Subversion)
- Team Foundation Version Control
- Visual SourceSafe

Descentralizados

- Git
- Bazaar
- Mercurial
- Fossil
- Monotone
- Code Co-op

SCM - *Source Code Management*

Boas práticas em SCM:

1. Commit, commit, commit

   Os commits são fáceis e baratos em qualquer SCM hoje. Faça quantos achar necessários e mantenha-os curtos.

   Não abuse, um commit significa uma mudança relevante no código. Se necessário, junte vários commits em um único.

SCM - *Source Code Management*

Boas práticas em SCM:

2. Sempre atualize seu repositório

   Comece seu dia atualizando seu repositório local. Reserve um tempo para ver os códigos que outros colocaram no repositório e como eles te afetam

3. Anote!

   Cada commit é acompanhado por uma mensagem. Coloque algo que faça sentido para o resto do time. Regra de ouro: tente incluir o que mudou e por que mudou. Se necessário, faça referência a *tickets/issues*.

SCM - *Source Code Management*

Boas práticas em SCM:

4. Revise seu código

   Use a área de *staging* para colocar somente o que faz sentido para o seu commit. Jamais “aproveite” commit para enviar modificações aleatórias.

5. Use branches

   Também fáceis e baratos em SCMs modernos, então use e abuse. Lembre-se de eventualmente fazer o *merge* e não deixe os fluxos distanciar muito. Restrinja o seu *branch* a um único propósito e descontinue-o (com ou sem *merge*) quando atingido.

SCM - *Source Code Management*

Boas práticas em SCM:

6. Siga o fluxo de trabalho

   Cada time/empresa tem um fluxo de trabalho (*workflow*) consolidado. Um SCM é uma ferramenta, focada em VCS. Acorde com seu time um fluxo de trabalho e siga-o.

   Não esqueça de documentar *branches*, *commits* e obviamente seu código!

   E.g. um *bug fix* será um *branch* ou um *commit*?

Git

The name "git" was given by Linus Torvalds when he wrote the very first version. He described the tool as "the stupid content tracker" and the name as (depending on your mood):

 - random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.

 - stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.

 - "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.

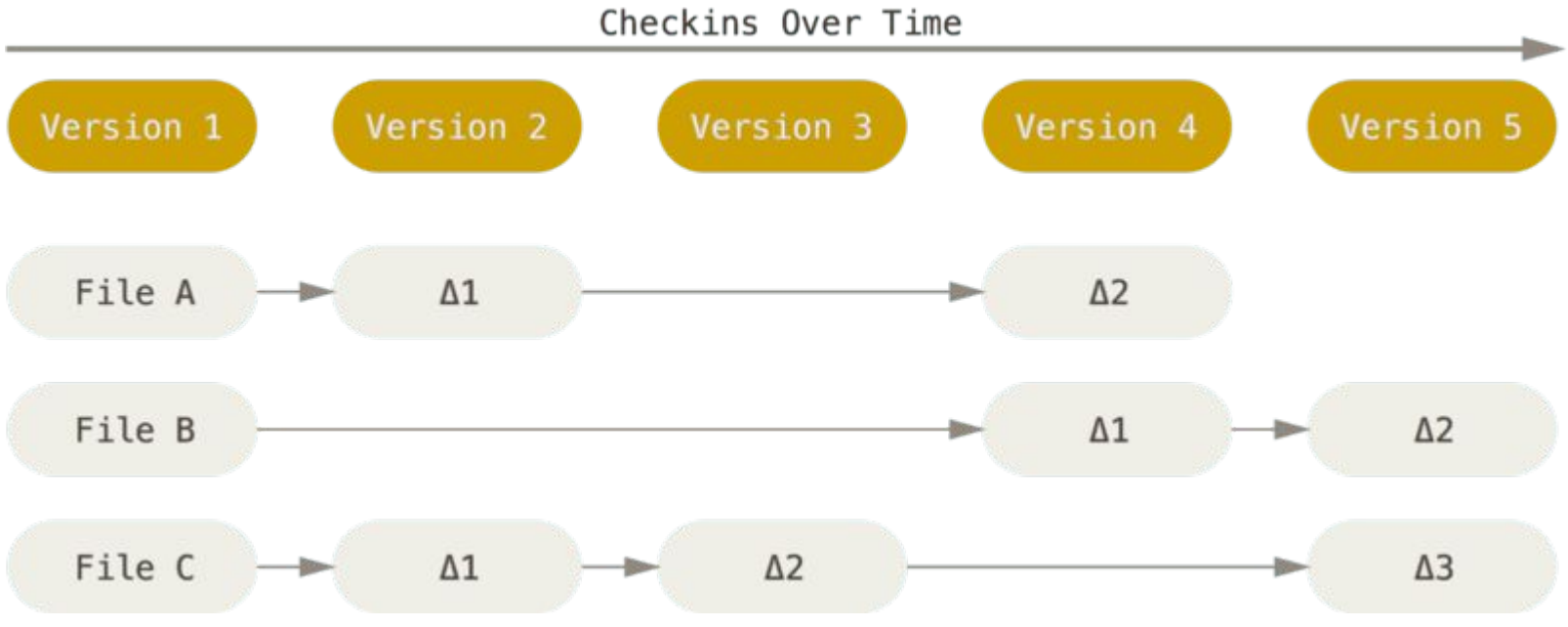
 - "goddamn idiotic truckload of sh*t": when it breaks

Fonte: README.md do projeto Git original

Git

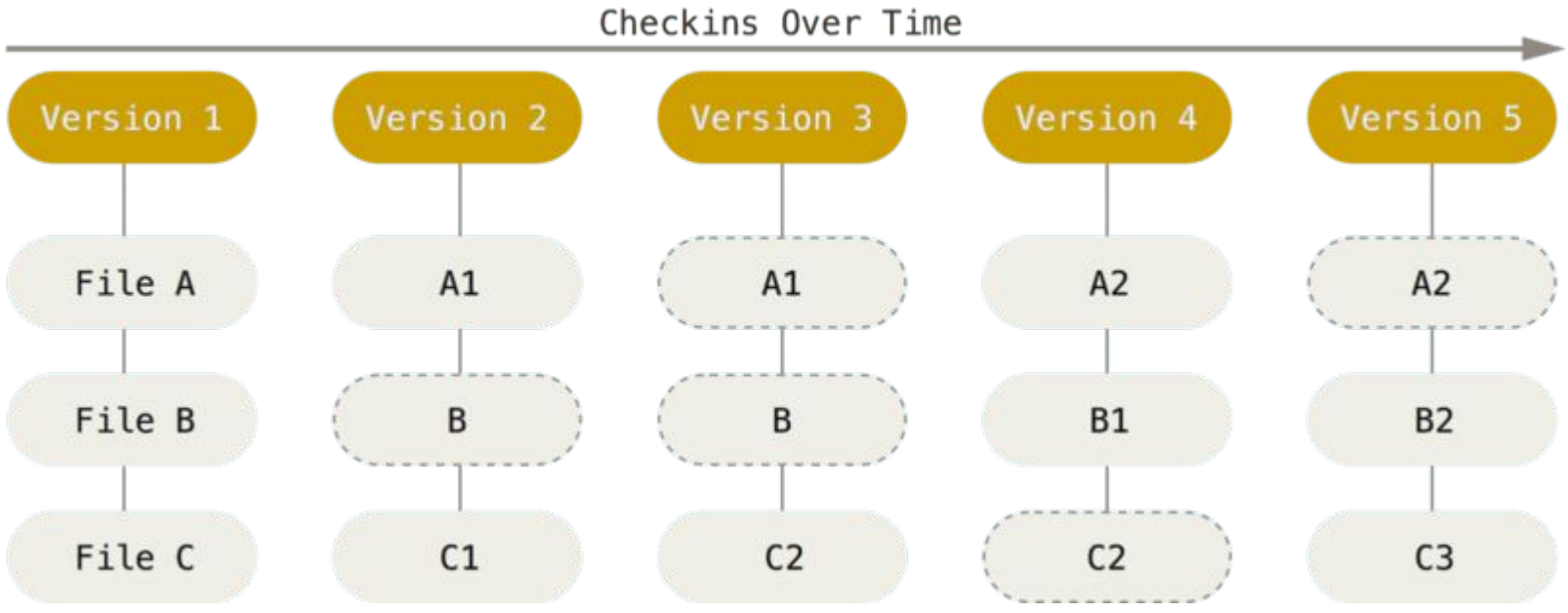
- Rápido
- Projeto simples
- Suporte robusto para desenvolvimento não linear
- Totalmente distribuído
- Eficiente para projetos grandes

Deltas (diferenças)



Fonte: [2]

Snapshots



Fonte: [2]

Suporte *Offline* e armazenamento

Quase tudo no Git opera *offline*

- Todas as cópias (clones) são um repositório completo
  - Todo o histórico de todos os arquivos pode ser calculado localmente
- Você pode fazer *commit* (modificações em um *snapshot*) localmente

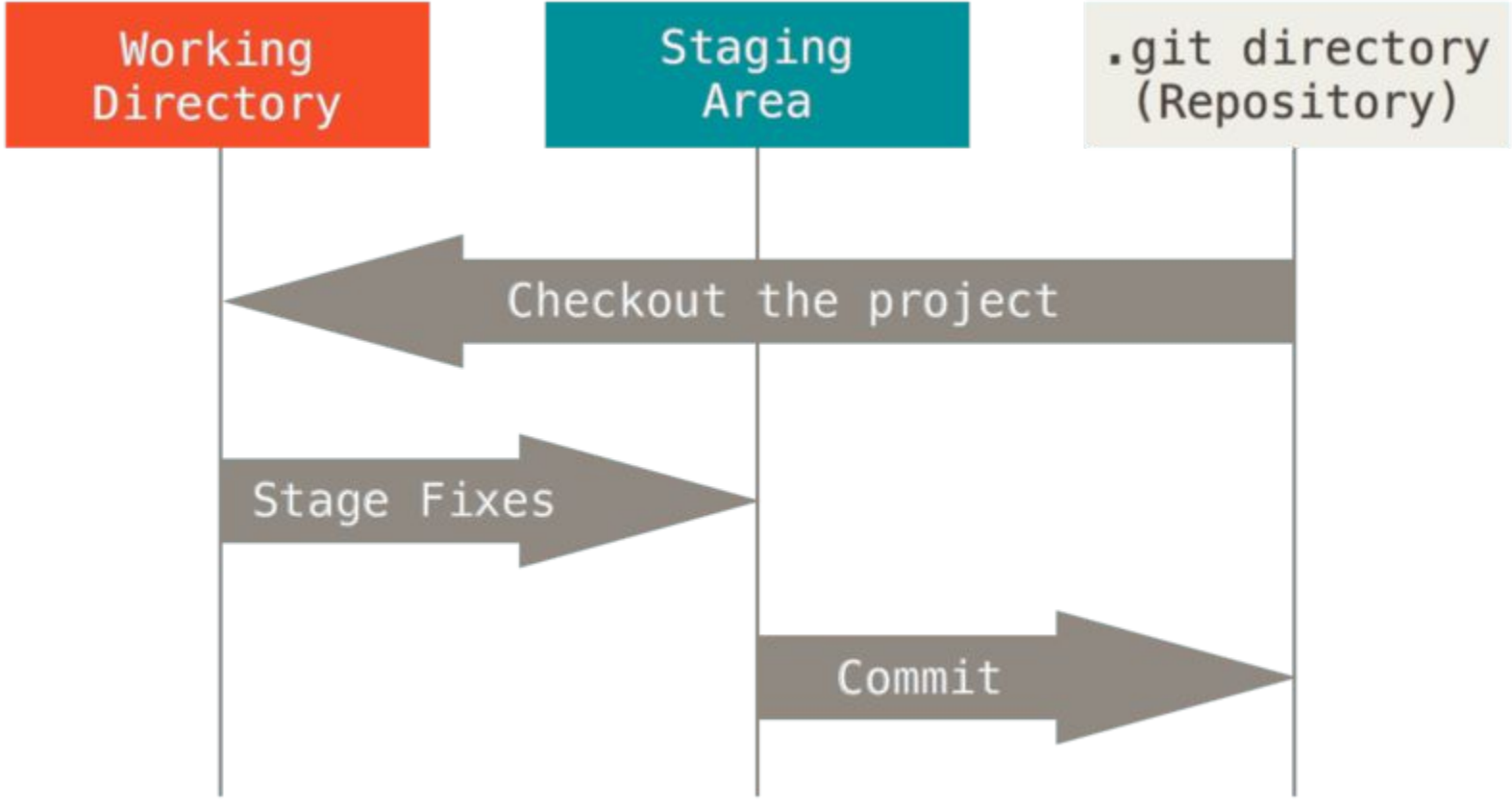
O Git mantém a integridade dos arquivos

- *Hash* SHA-1 (40x4b em hexa)
- O armazenamento é pelo *hash* e não pelo nome do arquivo

O Git foi feito para ser *append only*

- É muito difícil perder algo ou desfazer algo após o *commit*

Modified, Staged e Committed

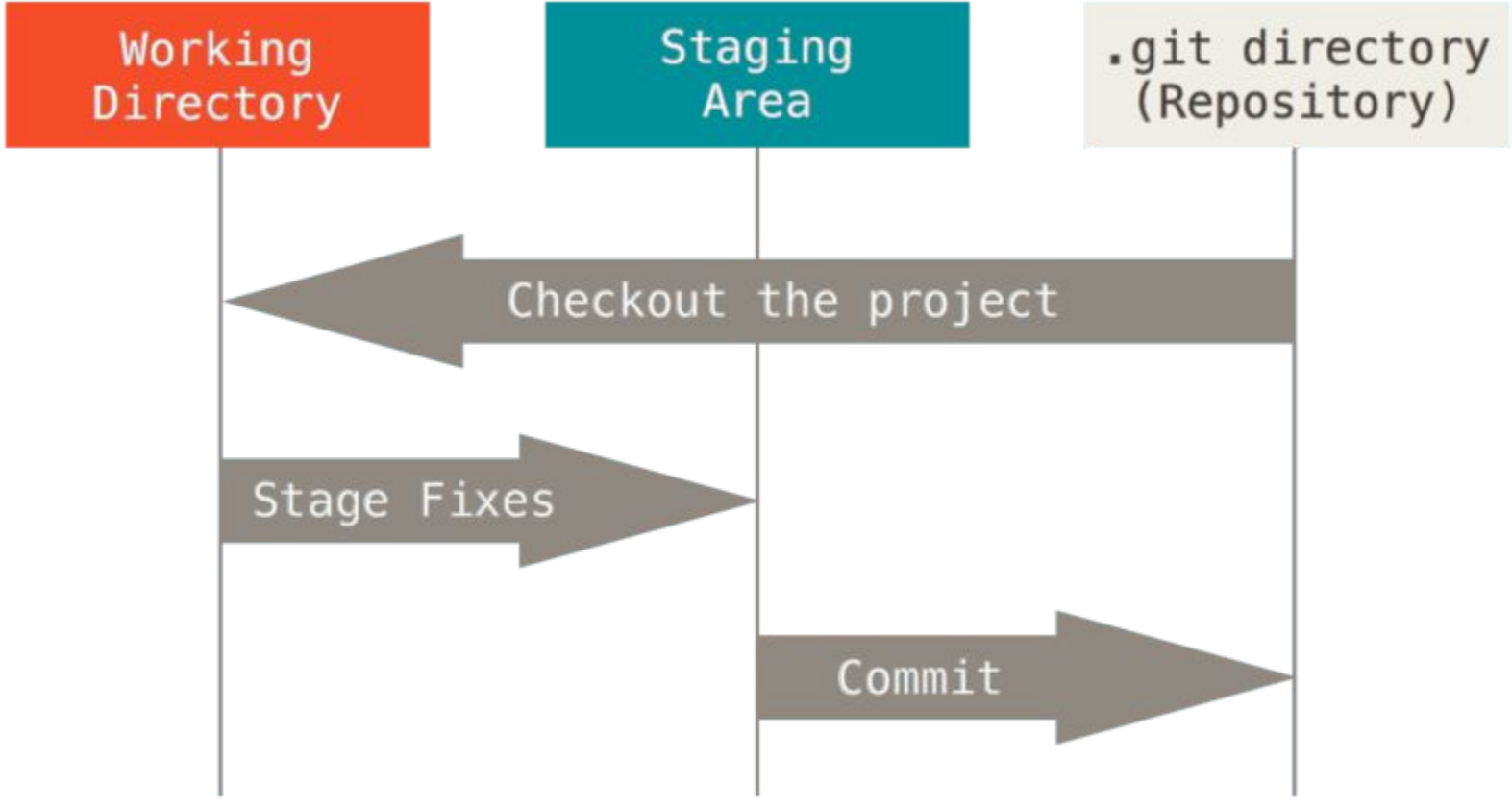


Fonte: [2]

Modified, *Staged* e *Committed*

1. *Modified*: você alterou um arquivo na sua árvore mas não confirmou (*commit*) suas modificações
2. *Staged*: você marcou suas modificações até o momento que você quer que sejam confirmadas (*committed*) no próximo *snapshot*
3. *Committed*: as modificações marcadas foram corretamente armazenadas no seu banco de dados local

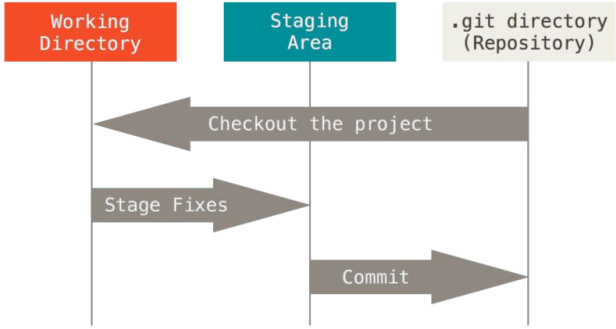
Modified, Staged e Committed



Fonte: [2]

Modified, Staged e *Committed*

- WD ou WT: uma visão particular do projeto, em disco e disponível para trabalho
- SA ou index: oculta dentro do seu armazenamento Git, contém as modificações (ou referências à elas) que serão confirmadas no próximo *commit*
- Diretório .git: é o banco de dados em si, um diretório oculto. A estrutura é o que mantém todo histórico.



Fonte: [2]

Ciclo de Trabalho

1. Modifique os arquivos que desejar no seu diretório (árvore) de trabalho.
2. Faça ***stage*** das modificações que você quer que sejam parte do seu próximo *commit*.
   - As modificações não incluídas continuam locais
3. Faça um ***commit***, que irá criar um *snapshot* das suas área de *staging* e armazenar no seu diretório .git.

Criando um repositório

1. Abra seu editor (recomendado: vscode)
   - Aponte para uma pasta nova da sua máquina, chame como desejar
2. No (painel|menu) Git, escolha criar um repositório
   - Dependendo da sua versão pode ser “init”
3. Liste os arquivos no diretório e veja a pasta .git
   - Pode ser necessário habilitar a visualização de arquivos de sistema (MacOS/Windows)
   - Gaste uns minutos observando os arquivos desta pasta
4. Crie um arquivo chamado `PROFILE.md`
   - Coloque seu nome e usuário do GitHub em algum lugar
   - Descreva sua experiência prévia com Git
   - Salve

Configurando seu Git

5. Abra o painel do Git
    - Verifique se está logado com seu usuário no GitHub
6. Configure seu usuário

    Há três arquivos de configuração:

    - Local (repositório): .git/config
    - Usuário: ~/.gitconfig (C:\Users\$USER\.gitconfig)
    - Global: /etc/gitconfig (C:\ProgramData\Git\config)

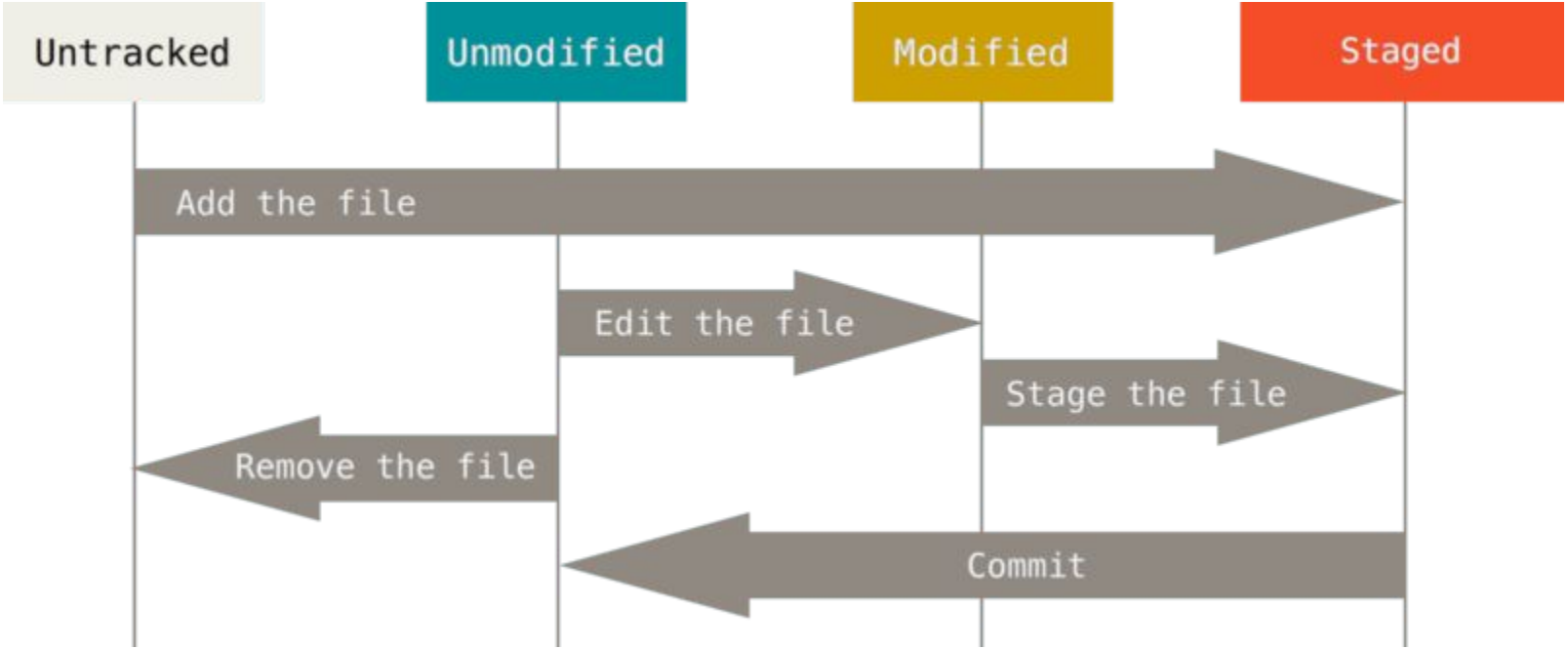
    É possível ter um arquivo por WT!!!

Adicionando arquivos

7. No seu editor (painel Git) encontre a área com os arquivos modificados
8. Adicione o arquivo PROFILE.md para a área de *staging*
9. Faça um *commit*
    - Quando solicitado, adicione uma mensagem de *commit* e salve o arquivo com a mensagem

Você tem até o momento um repositório local com um *commit*.

Arquivos



Fonte: [2]

📝 Exercício 1: *Stage* ou *Modified*?

Faça o seguinte exercício (pode ser no seu repositório):

1. Crie e *commit* um arquivo qualquer (texto)
2. Modifique o arquivo
3. Coloque (add) o arquivo na área de staging
4. Modifique o arquivo novamente
5. Faça commit

Qual versão irá para o *commit*? Grave sua resposta em um arquivo chamado ANSWERS.md (este é o Exercício 1). Não esqueça de fazer *commit* do seu arquivo de respostas!

Lembrete: versões 1 (criação), 2 (primeira modificação) ou 4 (segunda modificação)

Verificando diferenças e apagando

10. Ponto de partida: Exercício 1
11. Visualize as diferenças (editor)
    a. Desfaça as alterações (via menu, não digitando)
12. Faça commit das modificações

13. Apague o arquivo
14. Faça commit das modificações

📝 Exercício 2: Removendo arquivos

Faça o seguinte exercício (pode ser no seu repositório):

1. Crie um arquivo qualquer (texto)
2. Coloque (add) o arquivo na área de staging
3. Apague o arquivo
4. Faça commit

Por qual motivo o Git se nega a apagar o arquivo? Adicione sua resposta no mesmo ANSWERS.md (este é o Exercício 2) e faça *commit*.

Movendo arquivos

15. Ponto de partida: Exercício 1 ou 2
16. Crie e *commit* um arquivo qualquer (texto)
17. Mova o arquivo para uma pasta ou mude o nome do arquivo
18. Faça *commit*

📝 Exercício 3: Renomeando arquivos

Faça o seguinte exercício (pode ser no seu repositório):

1. Veja o estado do seu repositório.
2. Mude o nome do ANSWERS.md para RESPOSTAS.md
3. Veja o estado do seu repositório.
4. Faça commit

Você colocará sua resposta no ANSWERS.md ou no RESPOSTAS.md, a depender de qual sequência escolheu.

A pergunta é: o Git de moveu ou apagou e criou um arquivo novo? Explique o que fez (sequência), coloque a resposta no próprio arquivo e faça *commit*.

https://classroom.github.com/a/s_aopUGe

Clonando repositórios existentes

19. Faça um clone do repositório gerado quando você aceitou o link do slide anterior
    a. O diretório deve ser diferente do que estava trabalhando antes (sugestão:desça um nível e faça o clone)
20. Copie o arquivo PROFILE.md e o RESPOSTAS.md para este repositório
21. Faça *stage* e *commit*

A partir de agora TODOS os exercícios serão feitos nesse repositório!

📝 Exercício 4: Logs

Faça o seguinte exercício (pode ser no seu repositório):

1. Execute git log
    - Em alguns editores (incluindo algumas versões do VSCode) este comando não está disponível. Nesse caso, vá no terminal do editor e digite o comando acima.
2. Identifique o *commit* onde você adicionou os arquivos no exercício anterior
3. Execute git log -p X (veja a explicação do professor sobre o X)

Coloque o *hash* do *commit* e o que você imagina que seja a saída do -p como resposta a esse exercício.

Experimente também:

- git log --stat
- git log --pretty=oneline
- git log --graph
- git log -- <nomearquivo>
- git help log

Enviando modificações para o servidor

22. Adicione um arquivo qualquer (texto) no seu repositório
23. Faça *commit* (lembre-se de adicionar o arquivo ao *staging* antes)
24. Faça um *push*

Se tudo der certo, você terá enviado seu histórico local para o servidor! Abra a página do seu repositório no GitHub e confirme que os arquivos estão lá.

📝 Exercício 5: *Ammendments*

Faça o seguinte exercício (pode ser no seu repositório):

1. Abra o arquivo conhecimento.md
2. Marque apenas um conhecimento que tenha visto nesta aula
3. Faça *commit* (não faça *push*)
4. Marque todos os conhecimentos que você viu até agora
5. Faça outro *commit*, mas com *amendment*
    - git commit --amend ou escolha Commit (Amend) no seu editor
6. Faça o *push*
7. Analise o seu repositório (com git log ou na página do repositório no GitHub)

O que fez o *amend*? Não esqueça de colocar sua resposta no arquivo.

Desfazendo alterações

25. Adicione um arquivo qualquer (texto) no seu repositório e envie para o servidor do GitHub.
26. Modifique o arquivo que criou
27. Apague o seu PROFILE.md pelo seu SO (sem o Git)
28. Faça *staging* de um deles
29. Desfaça as alterações em ambos: git restore com ou sem --staged ou use a interface gráfica do seu editor (desfazer staging)

Se tudo deu certo, você não tem nada a enviar para o servidor.
Verifique com git status.

Desfazendo alterações

30. Modifique um arquivo no seu repositório (só salve)
31. Desfaça as alterações com git checkout <nomearquivo> ou use a opção correspondente no editor (discard changes)

Se tudo deu certo, você não tem nada a enviar para o servidor.
Verifique com git status.

📝 Exercício 6: *reset*, *restore* ou *checkout*?

Pesquise (google ou qualquer outro buscador ou até mesmo git help <comando>) a diferença entre os comandos acima.

Coloque a sua resposta no arquivo de respostas e envie para o servidor.

Nomeando *snapshots*

32. Faça *staging* e *push* de todas as modificações e respostas desta aula.
33. Crie uma tag git tag -a aulaD2_1 -m "meu comentario"
    - Nomeie como aulaD2_1
    - Mensagem é opcional
34. Faça git push origin --tags (não recomendo usar o editor, mas você pode usar a interface web)

Verifique na interface web se você tem um tag.

📝 Exercício 7: .gitignore

1. Crie um arquivo .gitignore no seu repositório (só salve em branco)
2. Copie um PDF qualquer para sua pasta de trabalho
3. Verifique o *status* do git
4. Adicione *.pdf ao arquivo .gitignore, salve e envie para o servidor
5. Verifique o *status* do git

Escreva no seu arquivo de respostas o motivo pelo qual você usaria este arquivo para selecionar o que o Git observa.

Você agora sabe git basics!

Não se esqueça de:

- Enviar seu arquivo RESPOSTAS.md para o servidor;
- Fazer um tag aulaD2_1;
- Eventualmente criar um arquivo de duvidas.md e colocar na raiz;
- A prova será liberada até sexta-feira após a última aula e há todo o final de semana para responder.

Até 22h (ou último aluno na sala):

- Dúvidas sobre todos os comandos de hoje;
- Exercícios extras
- Desafio (5 níveis)

Bruno Albertini [PCS/EPUSP]

balbertini@usp.br

Discord e GitHub: balbertini

Não há tarefa obrigatória fora de aula!

Desafios:

- https://www.w3schools.com/git/exercise.asp
- https://github.com/juanfresia/git-challenge
- https://learngitbranching.js.org/?locale=pt_BR

Bibliografia

(todos os links acessados em julho de 2022)

💻 [1] McMillan, Taryn. A History of Version Control (link).

📚 [2] Chacon, Scott. Pro Git (link).

💻 [3] Introdução com o Git no Azure Repos (link).

💻 [4] Templates para .gitignore (link).