
# WEB11 (Git) —— Aula 3

CURSO DE ESPECIALIZAÇÃO EM DESENVOLVIMENTO WEB COM *FRAMEWORKS* MODERNOS

UTFPR

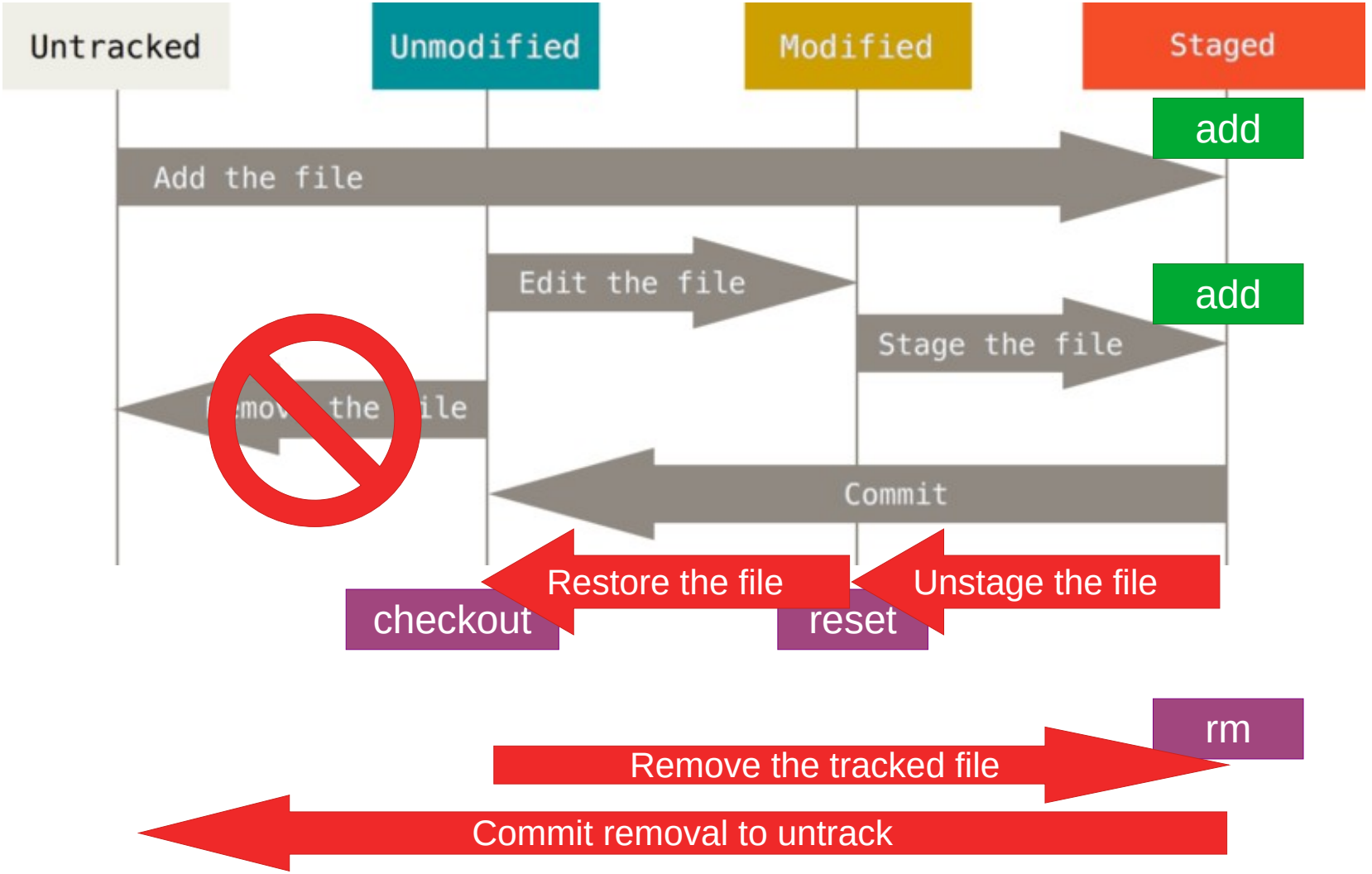


PROF. RUI PIMENTEL LEITE

# FLUXOS DE TRABALHO

- Na aula anterior, vimos:
  - `git init` cria o diretório `.git/`, transformando a pasta em um repositório Git
  - Arquivos iniciam não rastreados
  - Arquivos não rastreados podem ser ignorados
  - Comandos que modificam o arquivo em disco
  - Comandos que alteram o estado do repositório

# FLUXOS DE TRABALHO



# FLUXOS DE TRABALHO

- Mas como fazer backup do repositório?
- Como trabalhar colaborativamente?
- Como mesclar alterações feitas por dois usuários diferentes?
- Como tornar o histórico não linear (e o que isso significa)?

# FLUXOS DE TRABALHO

- Mais sobre *commits*:
  - À exceção do *commit* inicial, todo *commit* possui um ou mais *commits* pais
  - Todo *commit* possui também um ou mais *branches*
  - O *branch* criado com o `git init` chama-se `master` por padrão

# FLUXOS DE TRABALHO

- No projeto da aula anterior, execute o comando `git status` e note o *branch* relacionado

```
rui@gotham:~/Documents/Projetos/CEFWM/temp/novo-repo$ git status
On branch master
nothing to commit, working tree clean
rui@gotham:~/Documents/Projetos/CEFWM/temp/novo-repo$
```

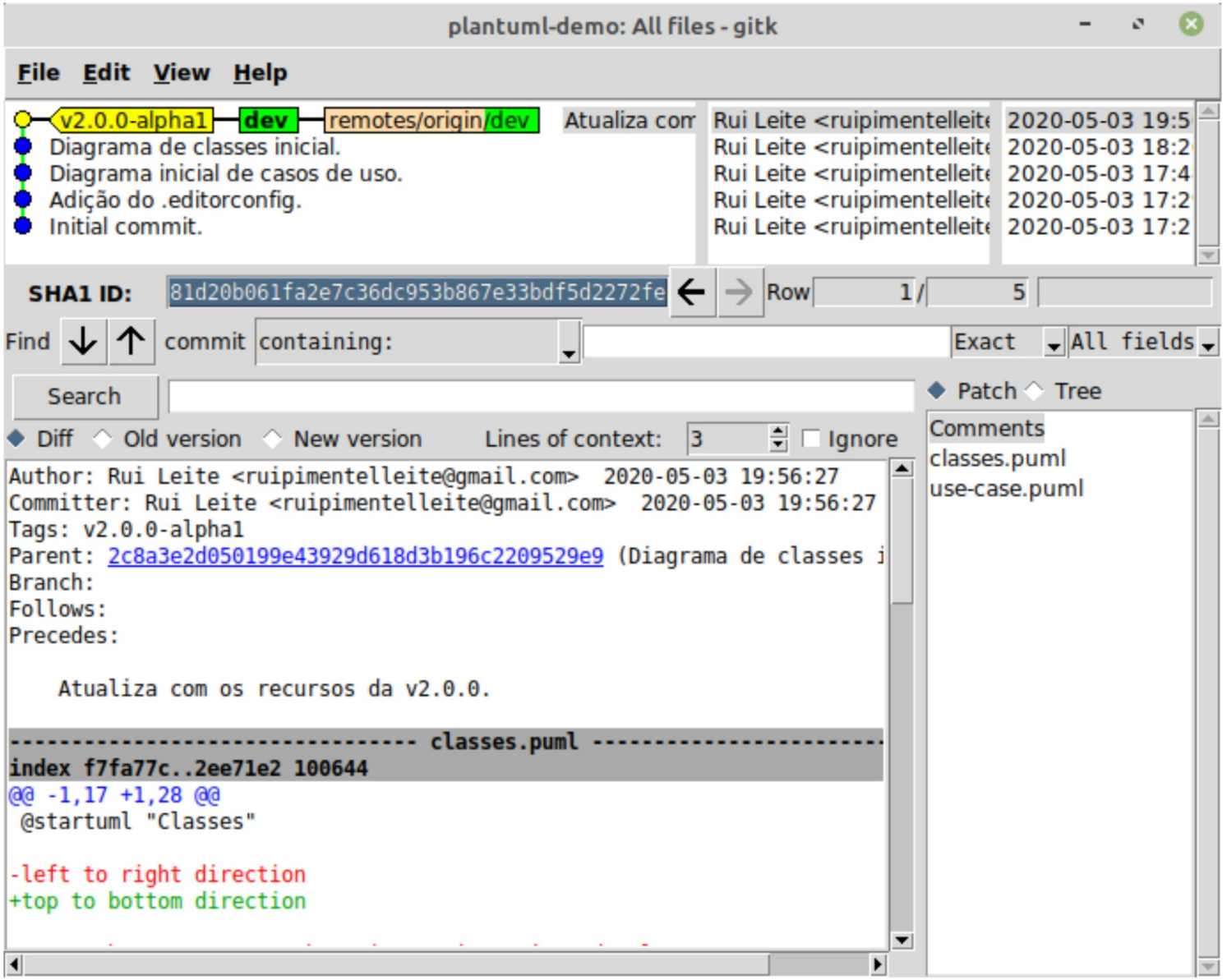
# FLUXOS DE TRABALHO

- Clone o repositório a seguir do GitHub e execute `git status` no diretório do projeto:
  - `ruipimentel/plantuml-demo`

```
rui@gotham:~/Documents/Projetos/CEFWM/temp/plantuml-demo$ git status
On branch dev
Your branch is up to date with 'origin/dev'.

nothing to commit, working tree clean
rui@gotham:~/Documents/Projetos/CEFWM/temp/plantuml-demo$
```

# FLUXOS DE TRABALHO



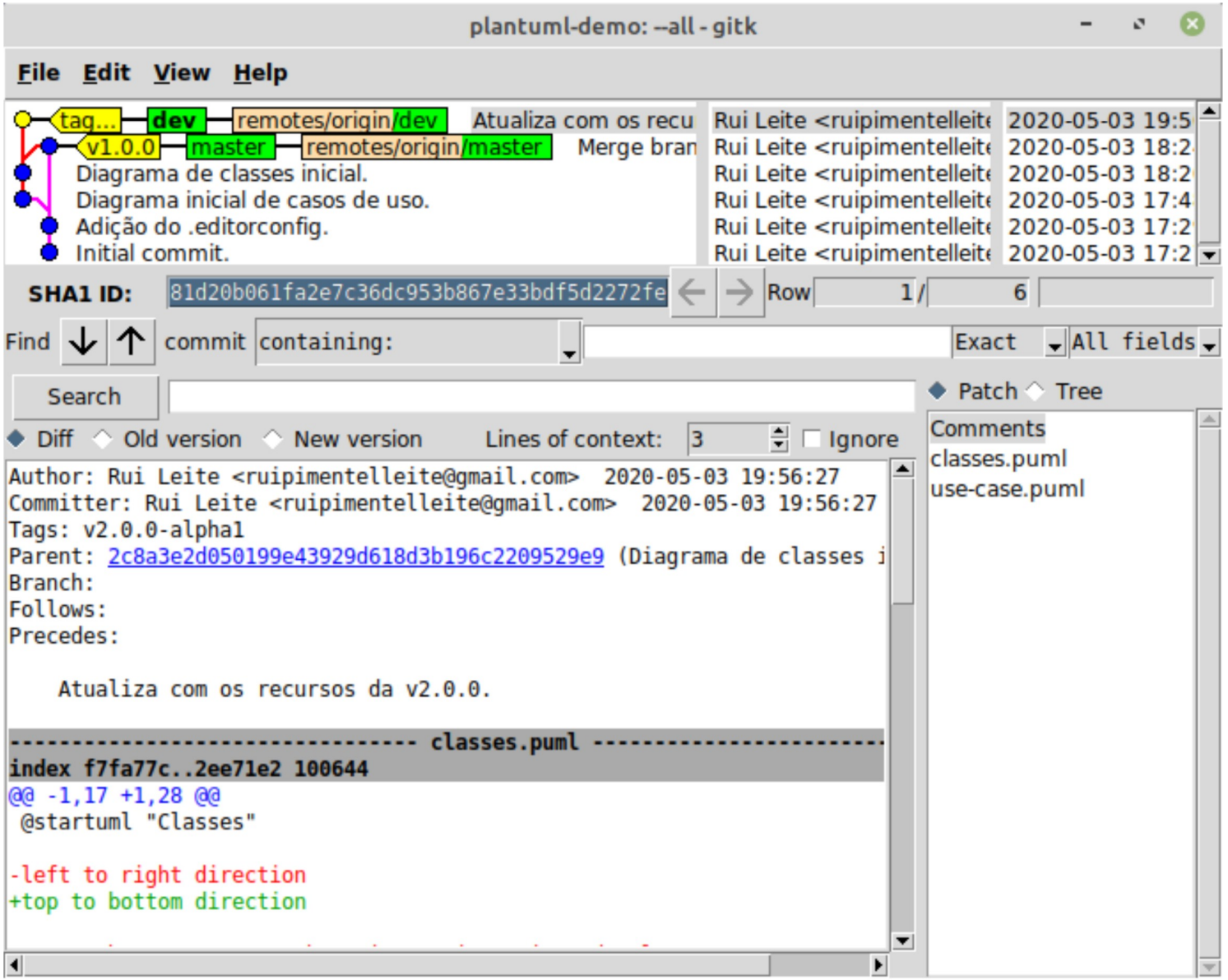
# FLUXOS DE TRABALHO

- Para instalar o projeto:
  - Execute os comandos a seguir no Terminal:
    - `sudo apt install graphviz`
    - `sudo apt install openjdk-11-jdk-headless`
  - Abra o diretório do repositório no Visual Studio Code
  - Instale a extensão `PlantUML` de `jebbs`
  - Abra um dos arquivos .puml e pressione `ALT+D` para abrir o *preview* do diagrama.

# FLUXOS DE TRABALHO

- Execute os comandos a seguir, observando após cada um as mensagens no Terminal, a situação do `gitk --all &` e o conteúdo dos arquivos do projeto:
  - `git checkout dev`
  - `git checkout master`
  - `git checkout v1.0.0`
  - `git checkout v2.0.0-alpha1`

# FLUXOS DE TRABALHO



# FLUXOS DE TRABALHO

- *Tag*: é uma espécie de ponteiro imovível para um *commit* específico
- Criado com comando `git tag nome-da-tag`
- Removido com comando `git tag -d nome-da-tag`
- O nome da *tag* deve ser único
- As *tags* existentes podem ser listadas com `git tag`

# FLUXOS DE TRABALHO

- Execute os comandos a seguir, observando após cada um as mensagens no Terminal e a situação do `gitk --all &`:
  - `git checkout 2c8a3e2`
  - `git tag`
  - `git tag ops`
  - `git tag`
  - `git tag -d ops`
  - `git tag`

# FLUXOS DE TRABALHO

- Faça *checkout* em `81d20b0`
- Repare que já existe uma *tag* nesse *commit*
- Introduza um comentário no arquivo `classes.puml`
- Faça *commit* da alteração
- Observe a situação do `gitk --all &`, verificando também a saída do seguinte comando:
  - `git tag`

# FLUXOS DE TRABALHO

- *Branch*: é uma espécie de ponteiro móvel para o último *commit* de seu ramo
- Criado com comando `git branch nome-do-branch`
- Removido com comando `git branch -d nome-do-branch`
- O nome do *branch* deve ser único
- Os *branches* existentes podem ser listados com `git branch`

# FLUXOS DE TRABALHO

- Execute os comandos a seguir, observando após cada um as mensagens no Terminal e a situação do `gitk --all &`:
  - `git checkout 2c8a3e2`
  - `git branch`
  - `git branch ops`
  - `git branch`
  - `git branch -d ops`
  - `git branch`

# FLUXOS DE TRABALHO

- Faça *checkout* em `81d20b0`
- Crie um novo *branch* e faça *checkout* nele
- Introduza um comentário no arquivo `classes.puml`
- Faça *commit* da alteração
- Observe a situação do `gitk --all &`, verificando também a saída do seguinte comando:
  - `git branch`

# FLUXOS DE TRABALHO

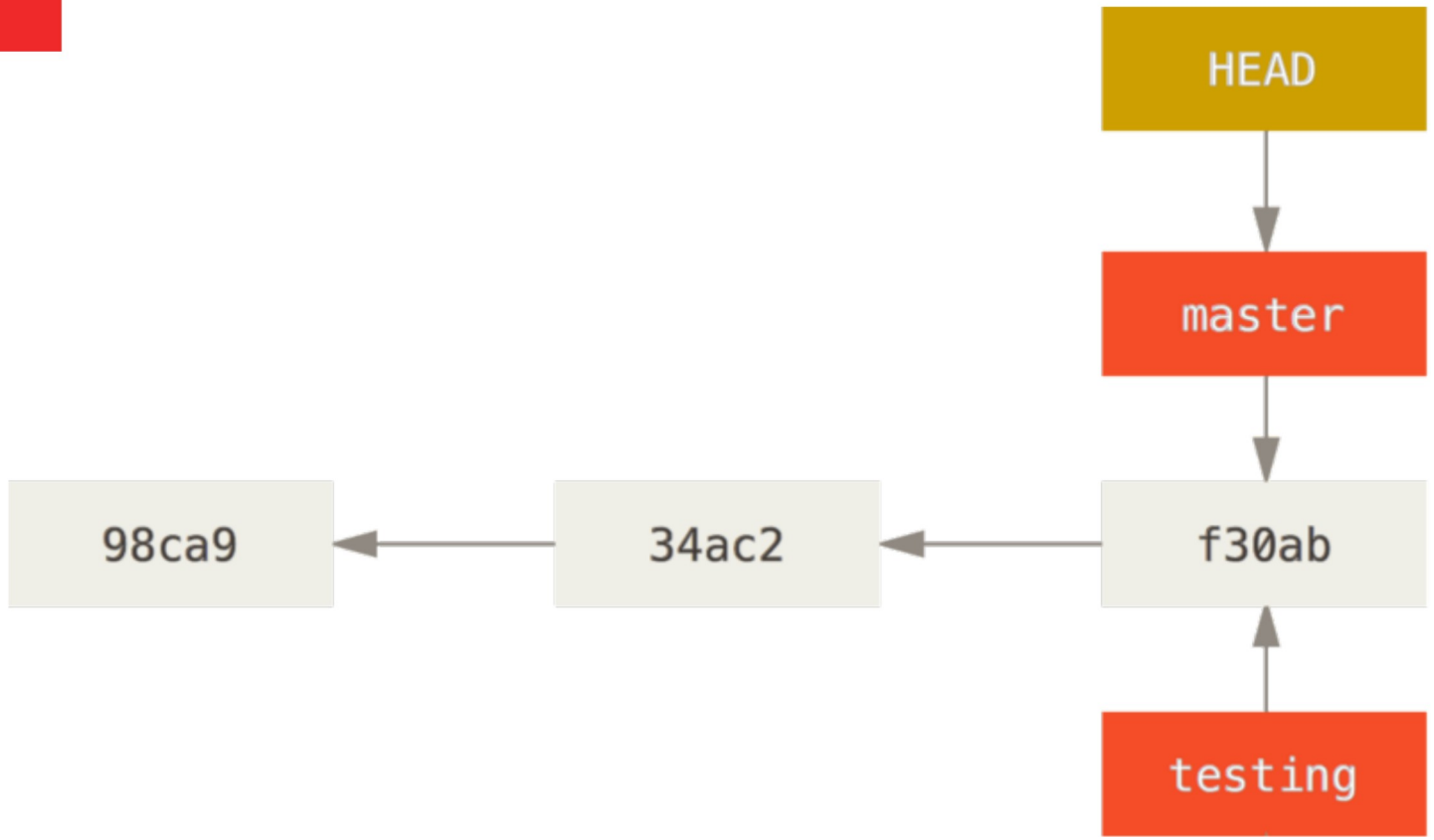
- Qual(is) *branch(es)* são movidos a cada novo *commit*?
- Como o Git sabe em qual *branch* estamos?
- O que é o `HEAD`, exibido por exemplo no comando a seguir?
  - `git log --graph --oneline --all`

# FLUXOS DE TRABALHO

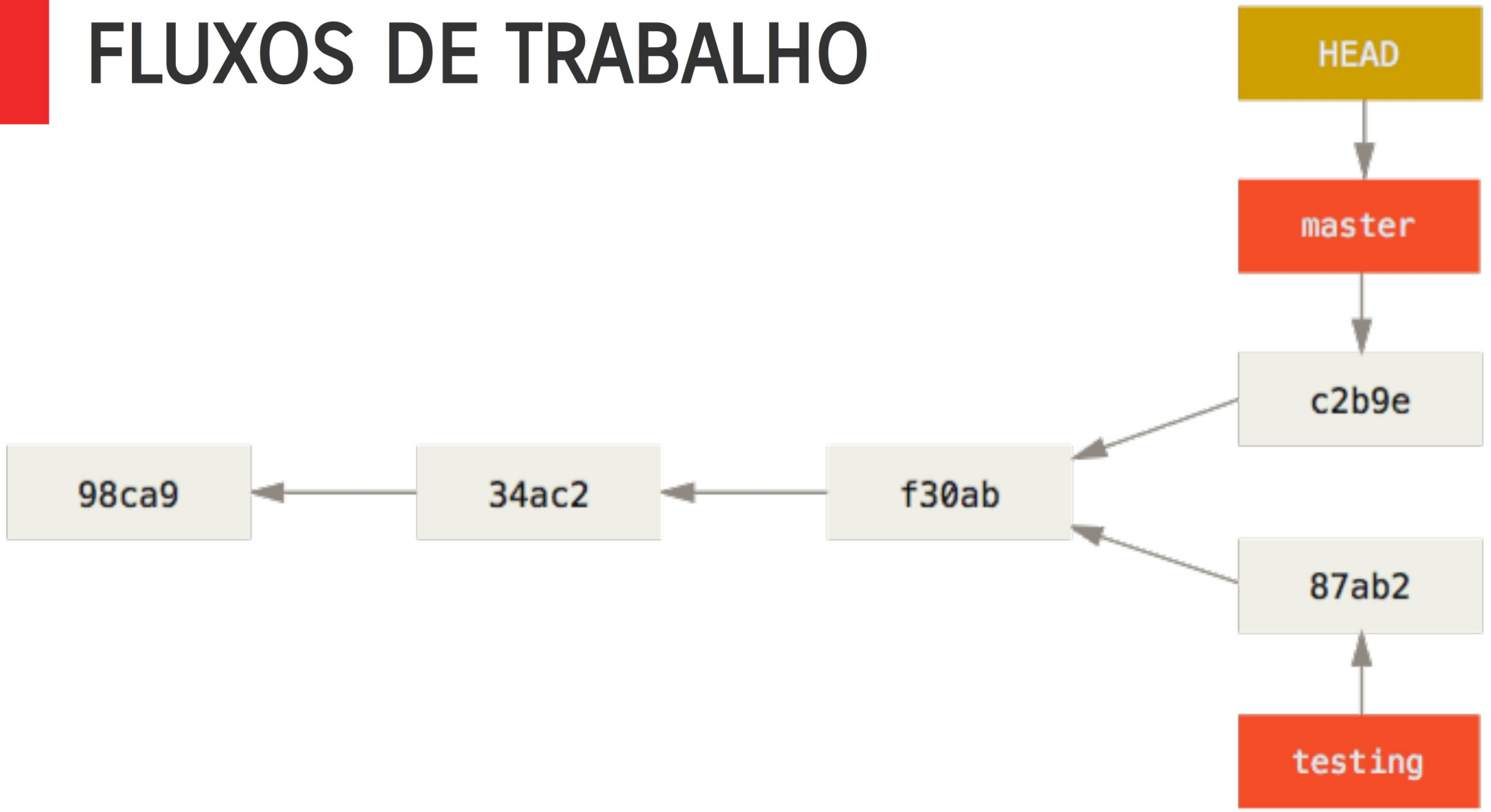
- Execute os comandos a seguir, observando atentamente as saídas no Terminal:
  - `git checkout 2c8a3e2`
  - `git log --graph --oneline --all`
  - `cat .git/HEAD`
  - `git checkout dev`
  - `git log --graph --oneline --all`
  - `cat .git/HEAD`

# FLUXOS DE TRABALHO

# FLUXOS DE TRABALHO



# FLUXOS DE TRABALHO



# FLUXOS DE TRABALHO

- Qual o significado de *branches* divergentes?
  - O código em cada *branch* pode estar em um estado diferente
  - Uma ou mais pessoas podem trabalhar em novos recursos de modo paralelo
  - Pode-se controlar *releases* de software

# FLUXOS DE TRABALHO

- Faça *checkout* em `dev`
- Crie um novo *branch* `dev-alt` e faça *checkout* nele
- Acrescente uma classe no início de `classes.puml`
- Faça *commit* da alteração
- Observe a situação do `gitk --all &`, verificando também a saída do comando:
  - `git log --graph --oneline --all`

# FLUXOS DE TRABALHO

- A cada novo *commit*, o Git atualiza o ponteiro HEAD, o que equivale a dizer que faz "*checkout* automático"
- Caso o HEAD aponte para um *branch* ao invés de um *commit*, no momento da criação de um novo *commit* o Git move o ponteiro desse *branch* juntamente com o HEAD

# FLUXOS DE TRABALHO

- Faça *checkout* em `81d20b0`
- **Não** crie um *branch*
- Acrescente uma classe no fim de `classes.puml`
- Faça *commit* da alteração
- Observe a situação do `gitk --all &`, verificando também a saída do comando:
  - `git log --graph --oneline --all`

# FLUXOS DE TRABALHO

- *"detached HEAD state"* é o nome do estado no qual o HEAD aponta diretamente para um *commit* ao invés de um *branch* local
  - Entra-se nesse estado ao fazer *checkout* através de uma *tag*, de um *hash* de *commit* ou de um *branch* remoto
  - Não é recomendado trabalhar neste modo, devido à propensão a "perder" *commits* por confusão

# FLUXOS DE TRABALHO

- Para sair do *Detached HEAD State* (DHS) e, assim, manter o novo *commit* criado:
  - Crie um *branch* com `git checkout -b sem-dhs`
  - Observe a situação do `gitk --all &`

# FLUXOS DE TRABALHO

- Tão útil quanto divergir *branches* é uni–los:
  - Para agrupar as alterações de uma ou mais pessoas para um *release* ou teste
  - Para copiar melhorias efetuadas sobre uma revisão para outra
- A mesclagem de *branches* resulta na soma das alterações em cada *branch*, desde que eles divergiram

# FLUXOS DE TRABALHO

- Faça *checkout* em `dev-alt`
- Digite o comando `git merge sem-dhs`
  - Deixe a mensagem padrão de *commit*
- Quais classes há no arquivo `classes.puml`?
- Observe a situação do `gitk dev-alt sem-dhs &`

# FLUXOS DE TRABALHO

- *Merge* automático: tende a manter tudo o que foi alterado em relação ao *commit* em comum, em qualquer um dos *branches* divergentes
- Exceção: alterações diferentes, feitas sobre um mesmo segmento de código, em mais que um *branch*
  - A isso se dá o nome de ***merge conflict*** ("conflito de mesclagem")

# FLUXOS DE TRABALHO

- Mantenha o *checkout* em `dev-alt`
- Mova os métodos de `AlertaAbsoluto` para Alerta, e então delete `AlertaAbsoluto`
- Faça *commit* da alteração

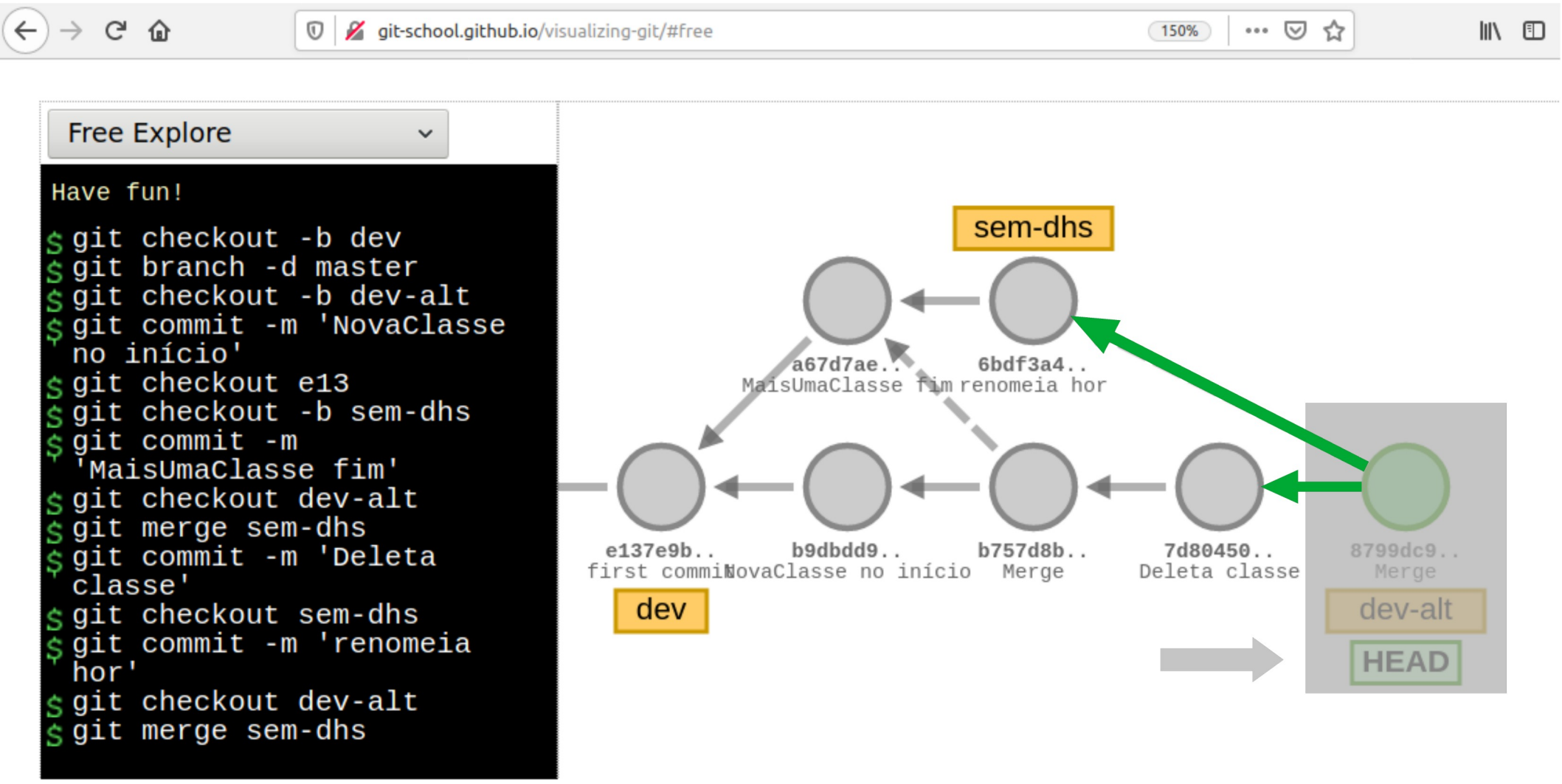
# FLUXOS DE TRABALHO

- Faça *checkout* em `sem-dhs`
- Renomeie parâmetros dos métodos de `AlertaAbsoluto`
- Faça *commit* da alteração

# FLUXOS DE TRABALHO

- Faça *checkout* novamente em `dev-alt`
- Digite o comando `git merge sem-dhs`
- Qual o conteúdo do arquivo `classes.puml`?
- Observe a situação do `gitk dev-alt sem-dhs &`
- Observe a situação também através do Meld

# FLUXOS DE TRABALHO



# FLUXOS DE TRABALHO

- *Merge conflict*: cada arquivo conflitante ficará de fora do *index*, e com *conflict markers* ("marcadores de conflito") em seu conteúdo.
  - O modo padrão de desistir de um conflito é `git merge --abort`
  - Ao abortar uma mesclagem, todas as alterações não "comitadas" serão desfeitas

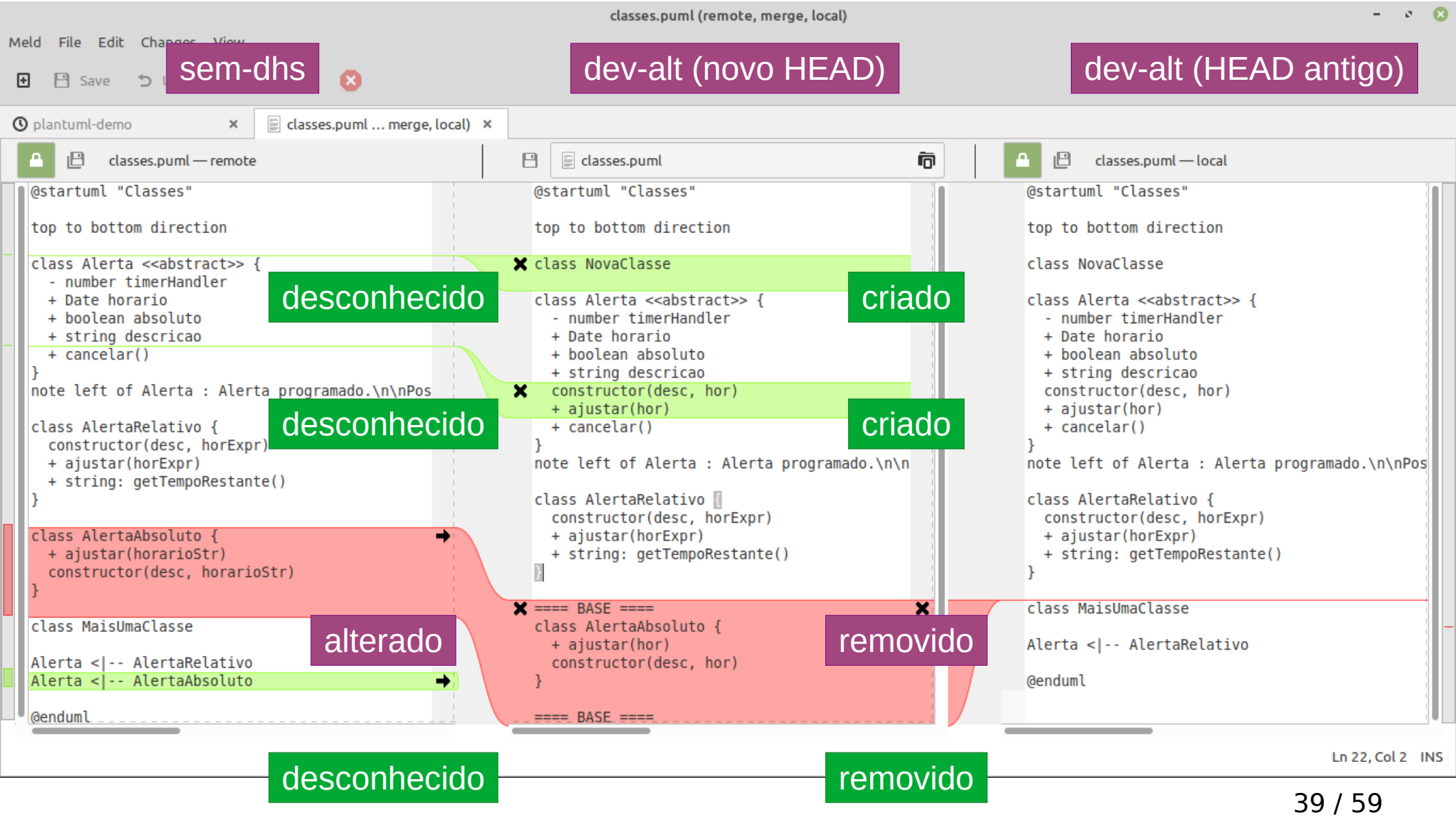
# FLUXOS DE TRABALHO

- É importante saber quando se está em modo de *merge*
- Ao "comitar", o Git considerará os *branches* unidos, mesmo que o desenvolvedor tenha, na verdade, apenas tentado limpar seu *working directory* para fazer outra atividade
  - Fora `git status`, as IDEs também possuem indicadores gráficos do modo de *merge*

# FLUXOS DE TRABALHO

- *Merge conflict*:
  - No arquivo com conflito, os marcadores indicarão o estado final de cada *branch* antes da mesclagem
  - O objetivo é resolvermos o conflito, e então adicionar o arquivo ao índice com `git add [path]`
  - Ao adicionar todos os arquivos (após resolver seus conflitos), finaliza-se o *merge* com `git commit`

# FLUXOS DE TRABALHO



# FLUXOS DE TRABALHO

- Estratégias ao efetuar uma mesclagem:
  - Mexer somente no que está estritamente relacionado ao conflito
  - Testar novamente o código no final
  - Em casos extremos, aceitar todas as alterações do *branch* sendo puxado (também chamado de *remote*, *theirs* ou *incoming*), e refazer nossas alterações sobre ele.

# FLUXOS DE TRABALHO

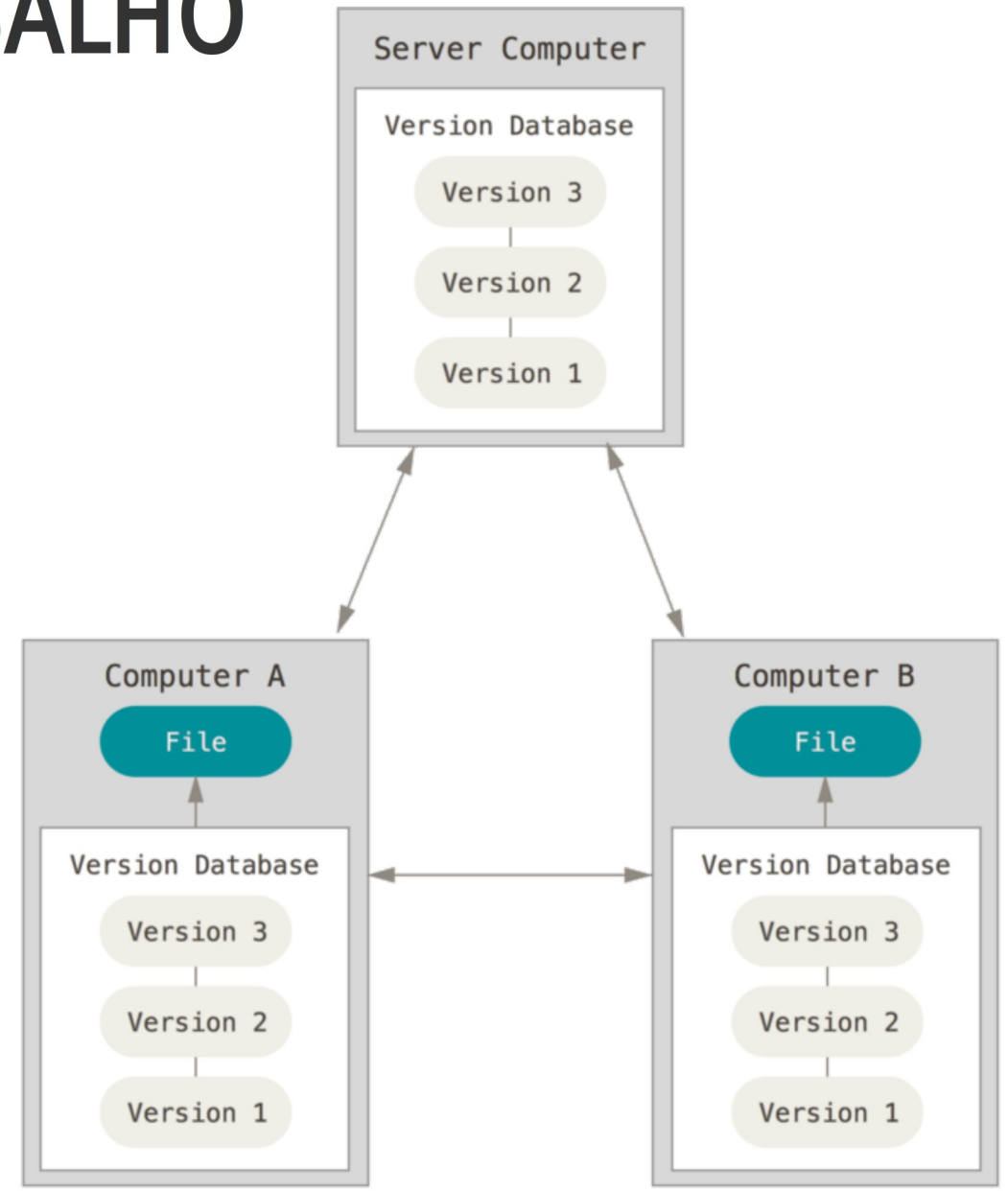
- Estratégias de prevenção de conflito:
  - Atualizar os *branches* constantemente (*pull*)
  - Modularizar o projeto sempre que possível
  - Divisão de tarefas (evitar que duas pessoas mexam no mesmo segmento simultaneamente)
  - Ferramentas de padronização de código (EditorConfig, *lint*)

# FLUXOS DE TRABALHO

- Estratégias de redução de conflitos:
  - Deixar o diretório de trabalho limpo antes de iniciar mesclagens
  - Vários *commits* pequenos ao invés de um grande
  - Indicação do objetivo das alterações na mensagem de *commit*
  - Evitar mexer no que não é necessário

# FLUXOS DE TRABALHO

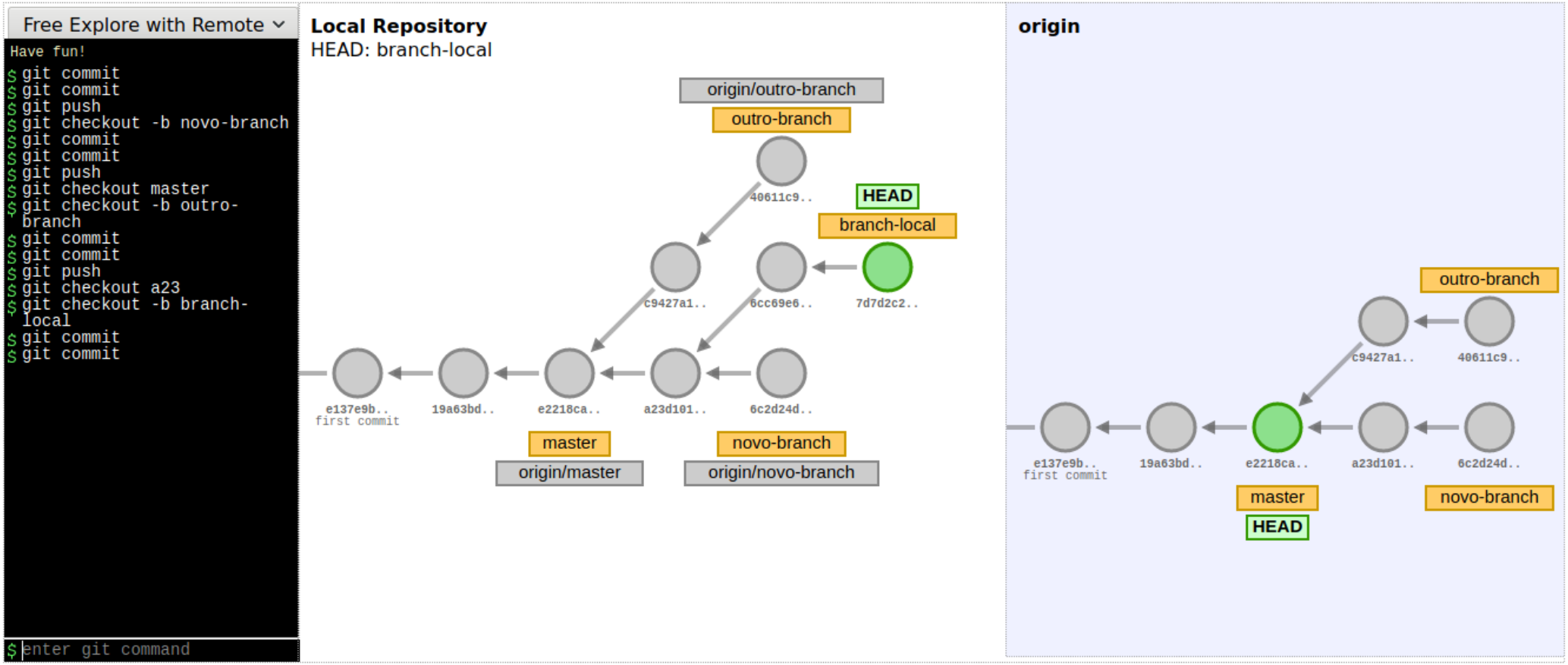
- Repositório remoto
  - Na filosofia do Git, um cliente não é tão diferente de um servidor



# FLUXOS DE TRABALHO

- Todo cliente e todo servidor possui todos os *commits* (ou ao menos todos os que já conseguiu sincronizar com os demais nós)
  - Todos os *branches* e todas as *tags* também são sincronizados entre nós
  - O servidor fica incumbido dos aspectos administrativos (proteção de gravação no repositório ou em certos *branches*)

# FLUXOS DE TRABALHO



# FLUXOS DE TRABALHO

- Para enviar o estado atual de um *branch* para um repositório remoto, utiliza-se o comando `git push`
- Caso o *branch* local ainda não esteja configurado para rastrear um *branch* remoto, o próprio Git sugerirá o comando `git push --set-upstream [remote] [branch]`

# FLUXOS DE TRABALHO

- Para visualizar os dados do(s) repositório(s) remoto(s) atrelado(s) ao nosso repositório local, utiliza-se o comando `git remote -v`
- Remova a origem atual com `git remote remove origin`
- Faça *fork* do projeto no GitHub, e então adicione sua cópia como *remote* com `git remote add origin https://github.com/cefwm-git/plantuml-demo.git`

# FLUXOS DE TRABALHO

- Utilize `git push --set-upstream origin dev-alt` para enviar seu *branch* local para o *remote*
- Faça *checkout* no *branch* `dev` e então mescle-o com `dev-alt`
- Em seguida, faça *push* desse novo *branch* para o *remote*

# FLUXOS DE TRABALHO

- O comando `git fetch` baixa do *remote* os "ponteiros" remotos, mas não altera os locais.
- Já o comando `git pull` faz o equivalente a um `git fetch` seguido do `git merge` do *branch* atual com seu *branch* remoto, ou seja, busca avançar o *branch* local para junto do remoto

# FLUXOS DE TRABALHO

- Para acrescentar desenvolvedores à equipe, basta garantir seu acesso ao repositório central (GitHub ou outro), através das ferramentas administrativas presentes no servidor
- A partir de então, cada desenvolvedor criará seus *branches* locais, podendo criar "backups" deles no repositório central

# FLUXOS DE TRABALHO

- A estratégia mais simples de *branching* para colaboração envolve um ramo `master` estável (isto é, um ramo permanente considerado o principal do repositório)
  - Nessa estratégia, para cada atividade é criado um novo *branch* que diverge do `master`
  - Ao fim de cada atividade, o próprio desenvolvedor ou um responsável realiza a mescla ao `master`

# FLUXOS DE TRABALHO

- Ao se trabalhar em repositórios colaborativos, é comum que o administrador do repositório central crie restrições de *push* para alguns *branches*
  - Nesse caso, um desenvolvedor que não possui acesso receberá uma mensagem de erro no ato do `git push`

# FLUXOS DE TRABALHO

- No GitHub, chama-se de *Pull Request* (PR) o mecanismo que permite que um *branch* seja enviado para aprovação em um *branch*/repositório ao qual o **desenvolvedor não possui acesso**
  - O pedido pode ser aceito ou recusado
  - Normalmente cria-se um PR na página do *fork*
  - Um usuário privilegiado pode, então, fazer *pull* do *branch* enviado e mesclá-lo ao *branch* alvo

# FLUXOS DE TRABALHO

- Ao se trabalhar com repositórios remotos, principalmente após efetuar `git fetch`, é comum receber avisos sobre *commits* à frente (*ahead*) ou atrás (*behind*) em um certo *branch*
  - Trata-se apenas da quantidade de *commits* conhecidos apenas pelo *branch* local ou apenas pelo seu *branch* remoto, respectivamente

# FLUXOS DE TRABALHO

- Assim como o comando `git branch` lista os *branches* locais, temos também comandos para listar *branches* remotos:
  - `git branch -r` lista somente os *branches* remotos
  - `git branch -a` lista os *branches* locais e remotos

# FLUXOS DE TRABALHO

- Vale lembrar que, por padrão, o `git push` não envia *tags*
  - Para isso, deve-se usar `git push --tags`

# FLUXOS DE TRABALHO

- Resumo até agora:
  - *Commits* possuem um ou mais pais
  - *Commits* podem ser referenciados por *hash* SHA-1, por *tag* ou por *branch*
  - *Tags* podem ser criadas e removidas facilmente, e enviadas ao *remote* através de `git push --tags`

# FLUXOS DE TRABALHO

- Resumo até agora:
  - *Branches* podem ser criados facilmente com `git checkout -b nome-do-branch`
  - Há várias técnicas para resolver conflitos, e também há diretrizes para evitá–los ou reduzir sua complexidade quando forem inevitáveis
  - A manipulação de *branches* é a base da colaboração entre desenvolvedores no Git

# FLUXOS DE TRABALHO

- Já somos capazes de:
  - Desenvolver repositórios complexos
  - Operar *branches* de diversos modos
  - Subir nosso código para um servidor remoto
  - Introduzir novos membros na equipe de desenvolvimento
- Mas como se trabalha no mercado, na prática?