WEB11 (GIT) — AULA 5

CURSO DE ESPECIALIZAÇÃO EM DESENVOLVIMENTO WEB COM *FRAMEWORKS* MODERNOS

UTFPR

PROF. RUI PIMENTEL LEITE

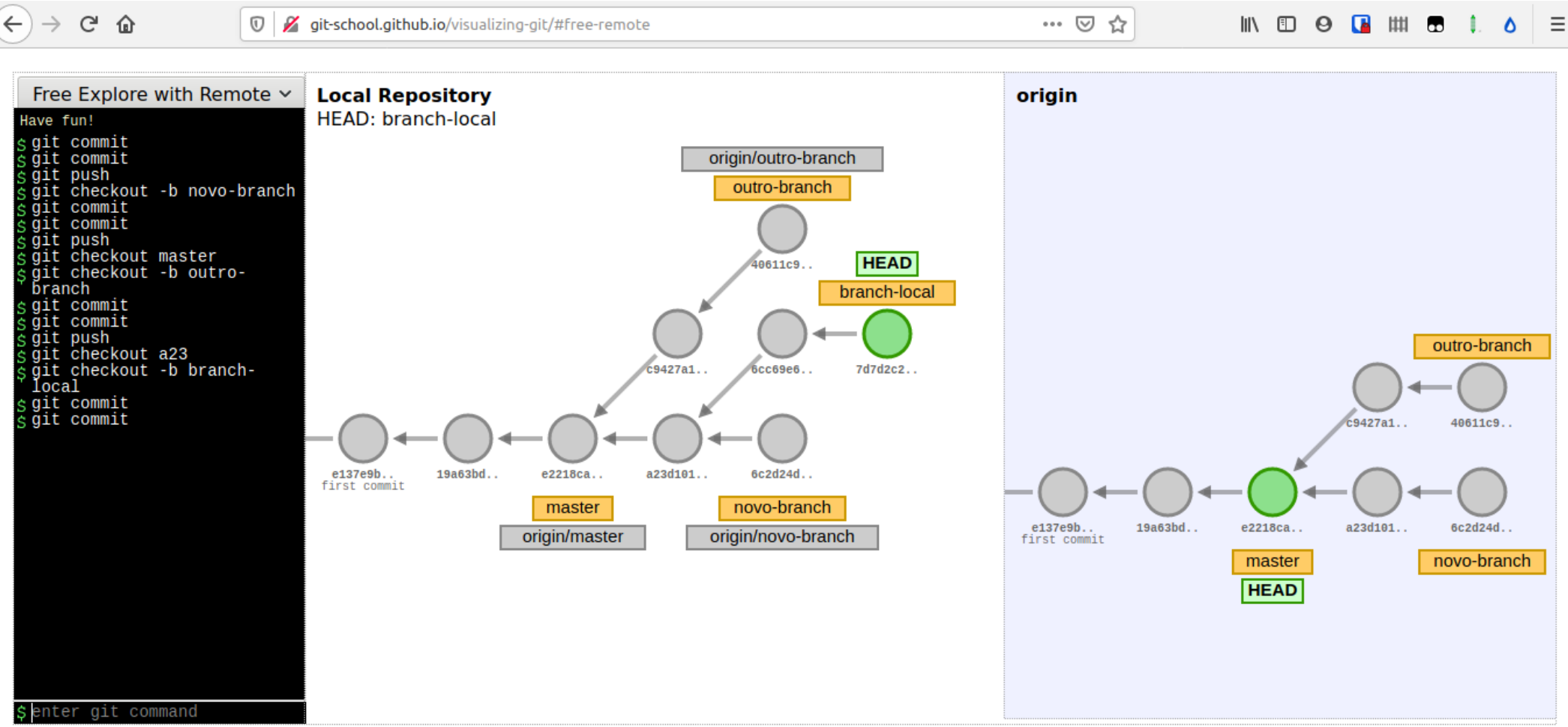
COTIDIANO E PRODUTIVIDADE

- Na aula anterior, vimos:
  - Como são os fluxos de trabalho profissionais mais populares com o Git
  - O melhor fluxo de trabalho vai depender da natureza da organização; o GitHub *Flow* é o mais recomendado para entrega contínua, e o Git *Flow* é mais recomendado para entrega em *releases*
  - O SemVer é um padrão popular para *tagging*

COTIDIANO E PRODUTIVIDADE

- Mas o que mais o Git pode fazer pelo desenvolvedor?
- Que técnicas de produtividade são viabilizadas por uso de Git?

COTIDIANO E PRODUTIVIDADE



COTIDIANO E PRODUTIVIDADE

- Listar o(s) *remote*(s) de nosso repositório:
  `git remote -v`
- Remover o *remote* `remote-x`:
  `git remote remove remote-x`
- Adicionar *remote*:
  `git remote add remote-y https://githu....git`

COTIDIANO E PRODUTIVIDADE

- Configurar *branch* atual para rastrear `brch-x` em `rmt`:
  ```
  git branch --set-upstream-to=rmt-y/brch-x
  git branch                    -u rmt-y/brch-x
  ```
- Fazer *push* do *branch* atual para `branch-x` em `remote-y`, configurando o *branch* local para rastrear este *branch* remoto:
  ```
  git push --set-upstream remote-y branch-x
  ```
- Listar detalhes dos *branches* atuais:
  ```
  git branch -a -vv
  ```

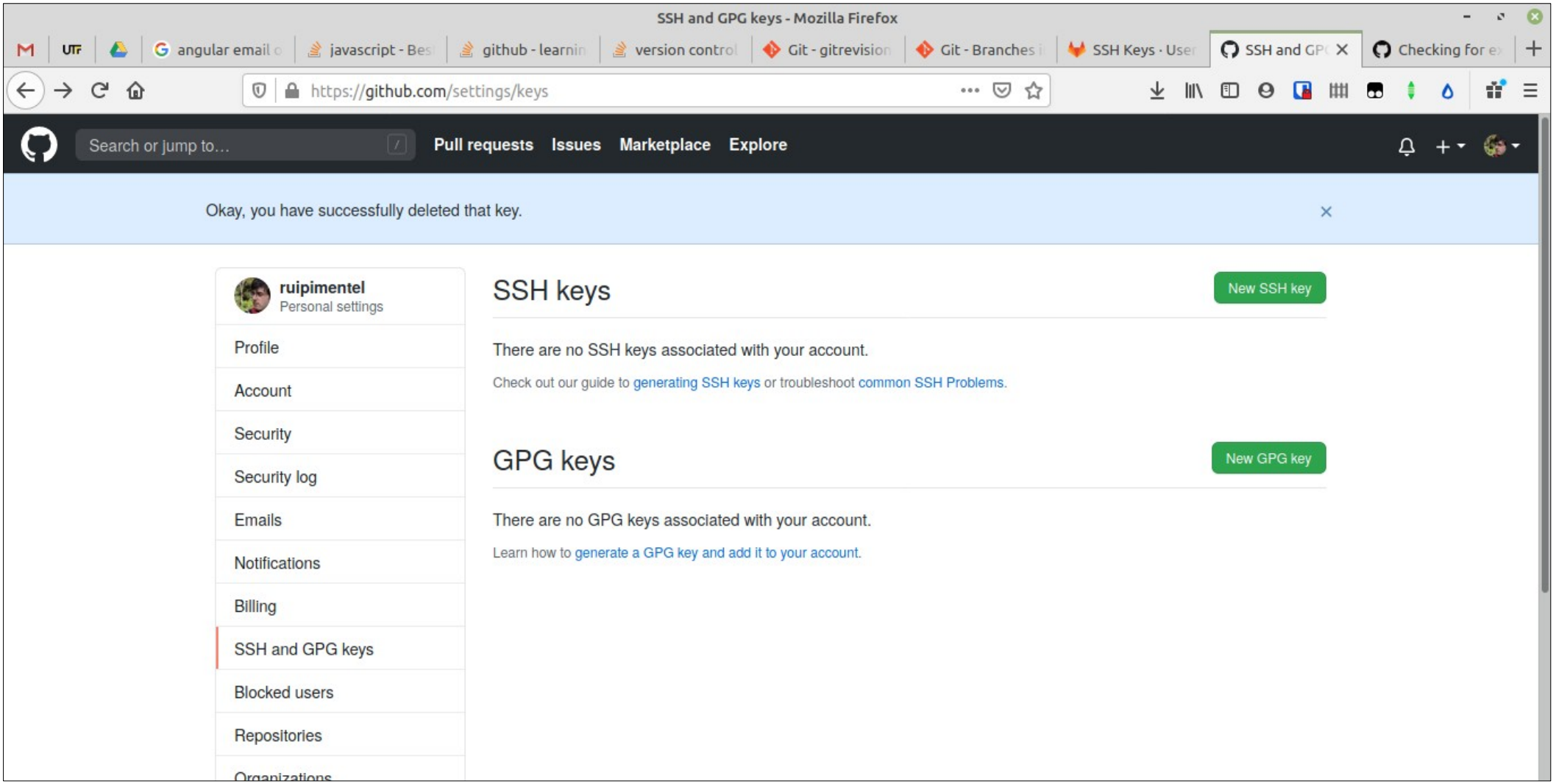
COTIDIANO E PRODUTIVIDADE

- Criar *branch* local `brch-x` e já fazer *check–out* nele:
  `git checkout -b brch-x`
- Criar *branch* local copiando o `brch-x` remoto (atenção: é preciso que `brch-x` só exista em um único *remote*):
  `git checkout brch-x`
- Excluir `branch-x` de `remote-y`:
  `git push --delete remote-y branch-x`

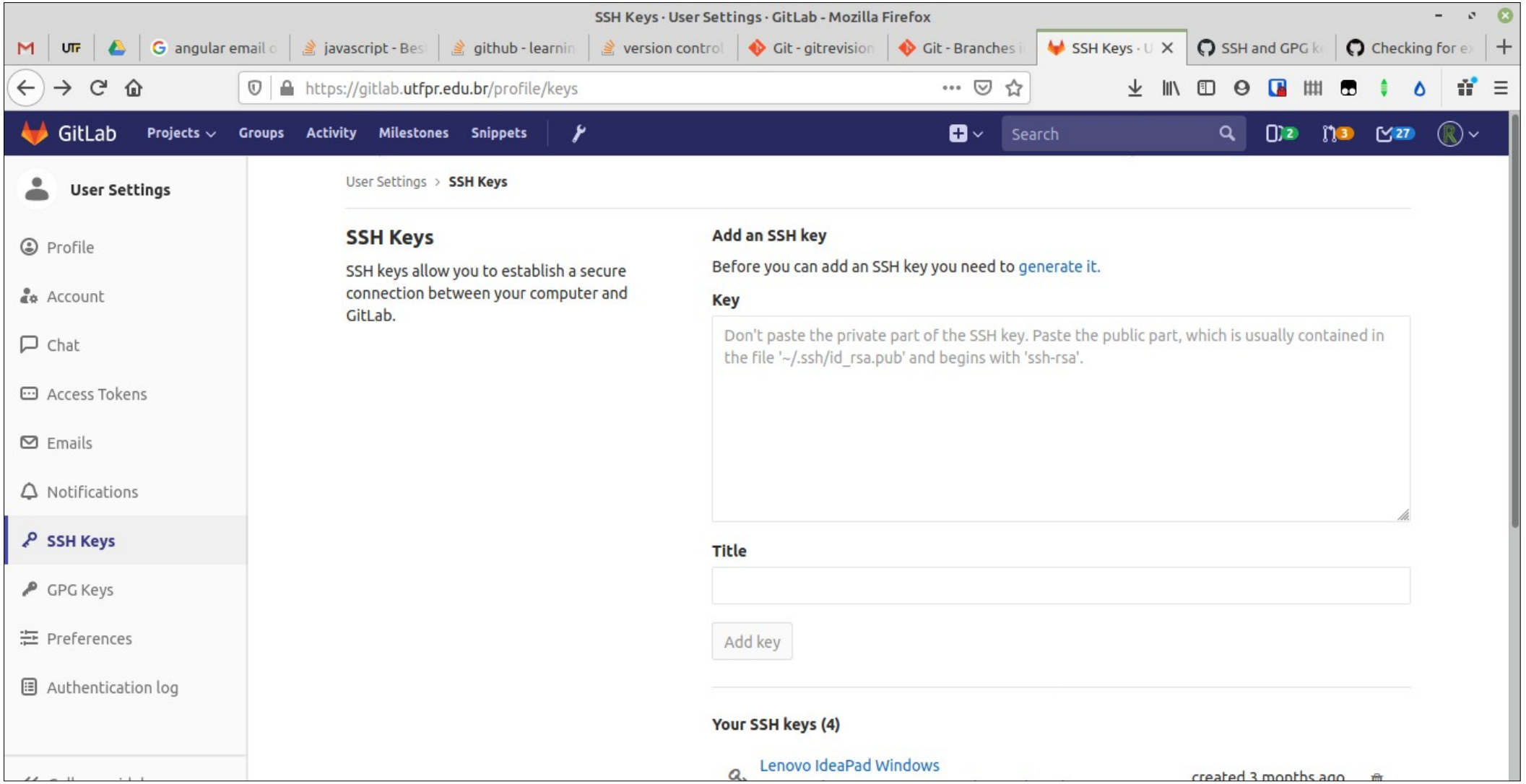
COTIDIANO E PRODUTIVIDADE

- Quando passamos a interagir frequentemente com repositórios remotos, torna-se conveniente configurar uma chave SSH de acesso remoto
  - Na realidade, trata-se de um **par de chaves pública** (compartilhada com o servidor) e **privada** (mantida somente na máquina do desenvolvedor)
  - Dessa forma, ao fazer *fetch* / *pull* / *push*, o servidor já identifica automaticamente o perfil do usuário baseado na máquina conectada

COTIDIANO E PRODUTIVIDADE



COTIDIANO E PRODUTIVIDADE



COTIDIANO E PRODUTIVIDADE

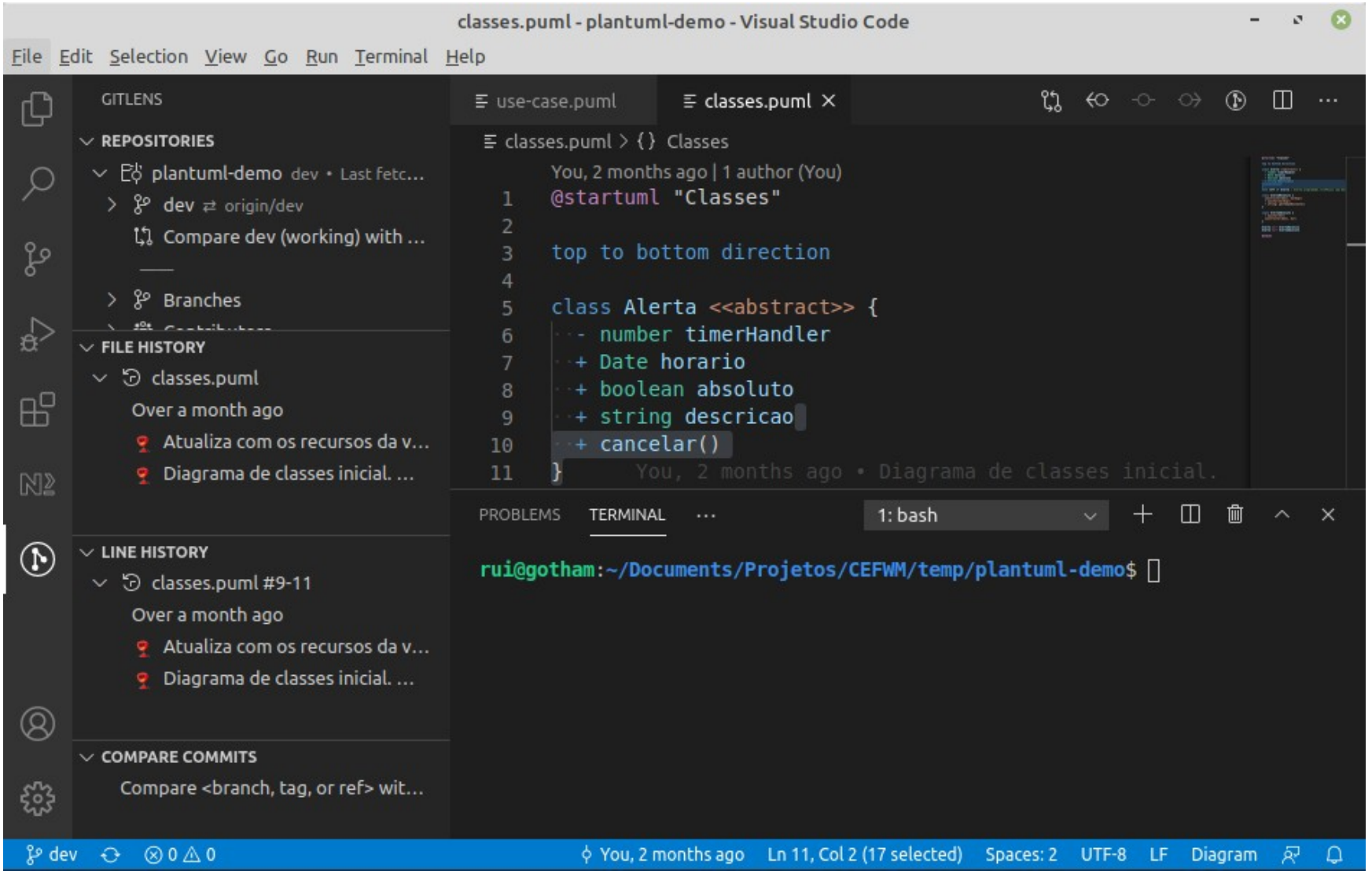
- Dois excelentes guias de como configurar chaves SSH são o do GitHub e o do GitLab, mas em síntese os passos são os seguintes:
  - Verificar se já há um par de chaves gerado
  - Caso não haja, gerar um novo par de chaves
  - Acrescentar a chave **privada** ao agente SSH local
  - Copiar a chave **pública** e colar no servidor

https://docs.github.com/en/github/authenticating-to-github/checking-for-existing-ssh-keys
https://gitlab.utfpr.edu.br/help/ssh/README#locating-an-existing-ssh-key-pair

COTIDIANO E PRODUTIVIDADE

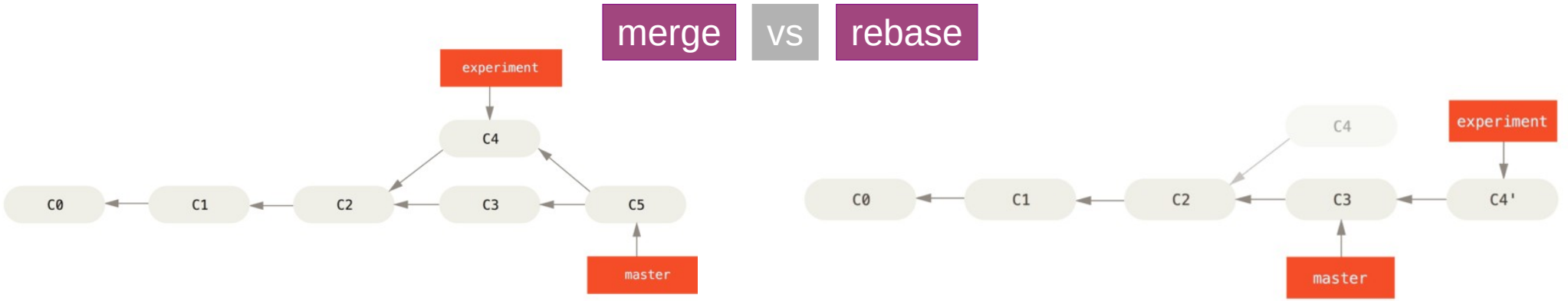
- Outra dica de produtividade é a adoção de ferramentas gráficas (GitHub Desktop) ou IDEs com Git integrado (como o Visual Studio Code)
  https://git-scm.com/downloads/guis
  - No Visual Studio Code, uma extensão particularmente útil é o GitLens (de Eric Amodio)
- Com a integração e alguns *plugins*, torna-se trivial verificar o status do diretório de trabalho e do índice, fazer commits, visualizar o histórico, entre outras operações

COTIDIANO E PRODUTIVIDADE



COTIDIANO E PRODUTIVIDADE

- *Rebase*: operação similar ao *merge*, produzindo um *snapshot* idêntico, porém com histórico linear
- Realizado com o comando:
  `git rebase nome-do-branch`



Adaptado de https://git-scm.com/book/en/v2/Git-Branching-Rebasing

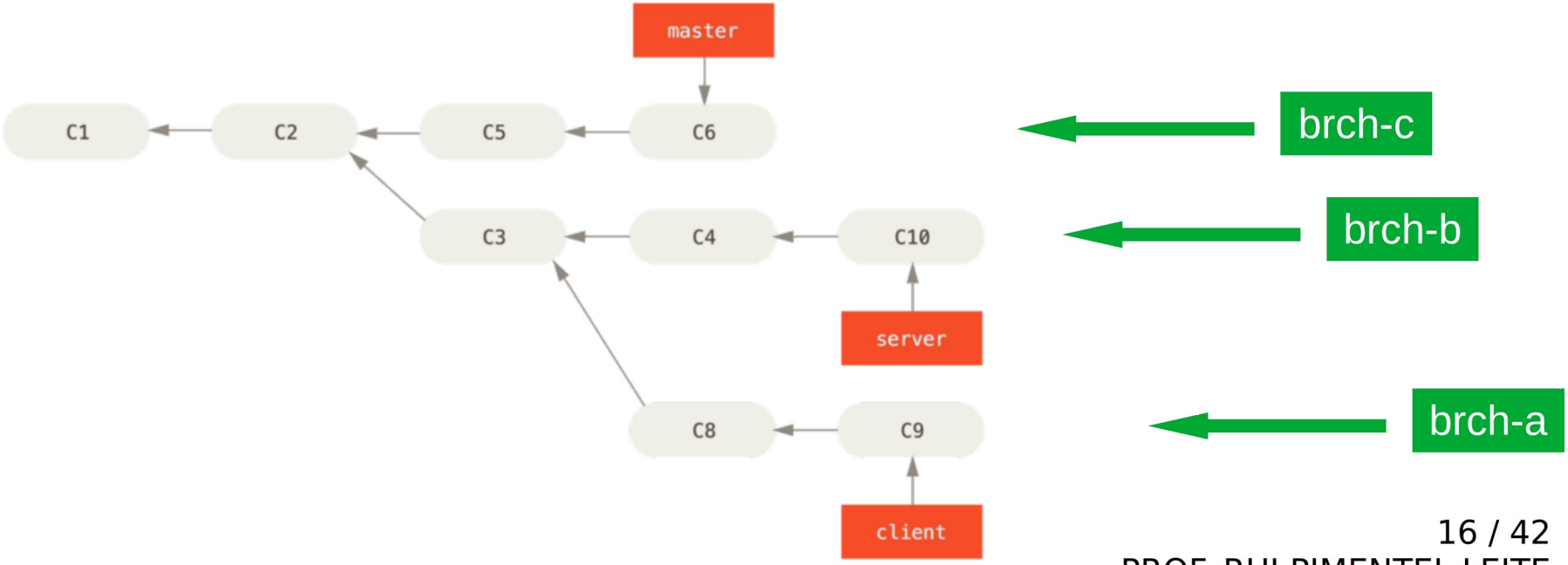
COTIDIANO E PRODUTIVIDADE

- Durante um `rebase`, também podem acontecer conflitos, que devem ser solucionados através de técnicas similares às dos conflitos de `merge`
  - Para prosseguir (após `add` dos arquivos resolvidos), utiliza-se `git rebase --continue`
  - Para pular *commits*, usa-se `git rebase --skip`
  - Para abortar, usa-se `git rebase --abort`
- Pode-se executar o `rebase` de modo interativo através da *flag* `--interactive` (`-i`)

COTIDIANO E PRODUTIVIDADE

- Com o `rebase`, algumas opções mais interessantes se abrem, como transportar divergências de um *branch* `brch-a` em relação a um `brch-b` para um `brch-c`:

```
git rebase --onto brch-c brch-b brch-a
```



Adaptado de https://git-scm.com/book/en/v2/Git-Branching-Rebasing

COTIDIANO E PRODUTIVIDADE

- Uma operação similar é o `cherry-pick`, que surgiu originalmente para "copiar" um *commit* para o *branch* atual (mas evoluiu com o tempo para suportar a cópia de diversos *commits* simultaneamente)
- Ou seja, enquanto o `rebase` transporta nosso *branch* atual para outra base, o `cherry-pick` transporta *commits* de outro *branch* utilizando nosso *branch* atual como base

COTIDIANO E PRODUTIVIDADE

- Para copiar desde o *commit* `rev1` (**exclusive**) até o *commit* `rev2` (inclusive), utiliza-se a sintaxe:
  `git cherry-pick rev1..rev2`
- Para copiar desde o *commit* `rev1` (**inclusive**) até o *commit* `rev2` (inclusive), utiliza-se a sintaxe:
  `git cherry-pick rev1^..rev2`
- O manual `gitrevisions(7)` especifica a notação de revisões e ranges no Git

COTIDIANO E PRODUTIVIDADE

- Uma dúvida de muitas pessoas que iniciam no Git é como desfazer *commits*
  - Isso pode ser feito de modo destrutivo (ou seja, reescrevendo o histórico) através do `reset` (que veremos adiante), ou então de modo aditivo com o `revert`
- O `git revert rev` é uma operação bastante similar ao `cherry-pick`, mas que aplica o exato contrário (inverso) do *diff* da revisão `rev` especificada

COTIDIANO E PRODUTIVIDADE

- `git reset`:
  - Move o *branch* para o qual `HEAD` aponta (diferentemente de `checkout`, que move apenas o `HEAD` e não o que ele aponta)
  - A *flag* padrão (`--mixed`) desfaz o *commit* e devolve as alterações dele ao *working directory*
  - A *flag* `--soft` apenas desfaz o *commit*, mantendo as alterações no índice (prontas para novo *commit*)

COTIDIANO E PRODUTIVIDADE

- `git reset`:
  - Já a *flag* `--hard` desfaz o *commit*, aplicando à força o *snapshot* anterior ao *working directory*
    - As alterações "comitadas" podem ser encontradas no `git reflog`
    - As alterações não "comitadas" são **destruídas definitivamente**

COTIDIANO E PRODUTIVIDADE

- `git reset`:
  - Podemos utilizar o `reset` para desfazer *commit*(s) do *branch* atual:
    `git reset --hard HEAD^`
  - O *commit* anterior permanece disponível caso seja acessível a partir de uma referência nomeada
  - *Commits* não referenciados possuem expiração padrão de 3 meses, quando podem ser deletados a qualquer momento pelo Git

COTIDIANO E PRODUTIVIDADE

- Resumo (`reset` versus `checkout`)

	Move branch/ HEAD?	Desfaz índice?	Sobrescreve arquivos da pasta? (WD)	Preserva novos arquivos?
Commit Level				
`reset --soft [commit]`	Branch	Não	Não	Sim
`reset [commit]`	Branch	Sim	Não	Sim
`reset --hard [commit]`	Branch	Sim	Sim	NÃO
`checkout <commit>`	HEAD	Sim	Sim	Sim
File Level				
`reset [commit] <paths>`	Não	Sim	Não	Sim
`checkout [commit] <paths>`	Não	Sim	Sim	NÃO

Adaptado de https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified

COTIDIANO E PRODUTIVIDADE

- Como vimos, pode-se utilizar o `reset` para desfazer um *commit*
- Caso a intenção seja apenas incluir nesse *commit* alterações esquecidas, podemos adicionar essas alterações ao índice e usar `git commit --amend`
- Caso a intenção seja apenas alterar a mensagem de *commit*, basta ter o índice vazio ao fazer o `--amend`

COTIDIANO E PRODUTIVIDADE

- Caso a intenção seja incluir em um *commit* antigo alterações complementares, podemos adicionar essas alterações ao índice e usar `git commit --fixup=[hash do commit]` seguido de `git rebase --interactive autosquash [hash do commit]~`
- Caso a intenção seja apenas alterar a mensagem desse *commit*, pode-se utilizar a instrução `reword` do `git rebase --interactive [hash do commit]~`

COTIDIANO E PRODUTIVIDADE

- Rebasear *branches* por `rebase` ou `reset` pode causar problemas caso o *branch* já tenha sido distribuído (`push`), portanto a boa prática é apenas reescrever o histórico local, e se necessário

COTIDIANO E PRODUTIVIDADE

- Algumas vezes, possuímos alterações não "comitadas" (*dirty working directory*) que ainda não estão prontas para *commit*, no entanto necessitamos alternar imediatamente para outro *branch*
- Após resolvermos esse outro *branch*, queremos continuar as alterações de onde paramos
- Este é exatamente o caso de uso perfeito para o comando `git stash`

COTIDIANO E PRODUTIVIDADE

- `git stash`:
  - Utilizado para operar uma pilha (*stack*) de *stashes*, sendo cada *stash* um conjunto de alterações salvo
  - A operação `push` é a *default*, e cria uma nova *stash* contendo as alterações em arquivos não rastreados; sua *flag* `--include-untracked` (`-u`) é usada para salvar na *stash* inclusive arquivos não rastreados

COTIDIANO E PRODUTIVIDADE

- `git stash`:
  - A operação `list` lista as *stashes* salvas
  - A operação `apply stash@{N}` restaura as alterações contidas na *stash* `N`. Caso o argumento seja omitido, o *default* é a `stash@{0}` (a mais recente)
    - A *flag* `--index` do `apply` torna a restauração mais precisa, ao tomar o cuidado de devolver ao índice o que já tinha sido adicionado a ele

COTIDIANO E PRODUTIVIDADE

- `git stash`:
  - A operação `show stash@{N}` exibe o *diff* da *stash* `N`
  - A operação `drop stash@{N}` **deleta permanentemente** a *stash* `N`
  - A operação `pop stash@{N}` é equivalente a um `apply` da *stash* `N` **seguido de seu** `drop`

COTIDIANO E PRODUTIVIDADE

- É bastante comum precisarmos saber quem modificou uma certa linha, e/ou em qual *commit* isso aconteceu
  - Exemplo: um conflito foi gerado sobre uma linha de código e precisamos saber qual o propósito dela
- O comando `git blame arquivo1` é usado para exibir cada linha de `arquivo1` juntamente com informações sobre o *commit* que a deixou naquele estado
  - A *flag* `-L 5,15` pode ser usada para mostrar apenas as linhas de 5 a 15

COTIDIANO E PRODUTIVIDADE

- O Git também possui outras ferramentas de busca, como o `git grep termo-de-busca`
  - Com ele, podemos buscar ocorrências de `termo-de-busca` em todos os arquivos da revisão atual
  - Pode-se restringir a busca a um grupo de arquivos, e mais interessantemente, a uma revisão diferente da atual sem precisar fazer *check-out* nela
  - Há várias outras opções interessantes, com especial destaque à variedade de mecanismos de RegEx

COTIDIANO E PRODUTIVIDADE

- O `git log` também possui algumas *flags* que o transformam em uma excelente ferramenta de busca:
  - A *flag* `-S termo-de-busca` permite localizar todas as revisões em que `termo-de-busca` foi "inserido" ou "excluído" (ou uma aproximação disso)
  - A *flag* `git log -L :nome-funcao:arquivo` localiza (e exibe os segmentos dos *diffs*) dos *commits* onde houveram alterações na função `nome-funcao`

COTIDIANO E PRODUTIVIDADE

- Outra ferramenta extremamente útil, que geralmente é utilizada para encontrar *bugs*, é o comando `bisect`
  - A linha de raciocínio é: procuramos um *commit* cujo conteúdo, autor ou data não conhecemos, mas conseguimos detectar indiretamente seus efeitos
  - Exemplo 1: o código tem problemas para compilar sob certas condições, o que não acontecia antes
  - Exemplo 2: o layout está quebrado há algum tempo

COTIDIANO E PRODUTIVIDADE

- No modo de busca binária, o Git apresentará um *commit* por vez para que o desenvolvedor os revise
  - O revisor deve julgar cada *commit* apresentado como `good` (bom) ou `bad` (ruim)
- Inicia-se o modo de busca com `git bisect start`
- Deve-se em seguida buscar manualmente (através de *check-out*) um *commit* `good` e outro `bad`, marcando-os com os respectivos comandos:

  `git bisect good` OU `git bisect bad`

COTIDIANO E PRODUTIVIDADE

- Assim que o Git perceber que ao menos um *commit* bom e um ruim foram marcados, ele providenciará os próximos *check-outs* sozinho, na sequência ótima para reduzir a quantidade de testes necessários
- Assim que o Git tiver certeza de qual *commit* foi o primeiro a passar de bom para ruim, ele apresentará seu resumo e interromperá a busca
- Para encerrar a busca, utiliza-se `git bisect reset`

COTIDIANO E PRODUTIVIDADE

- Outros recursos interessantes do Git:
  - Submódulos
  - *Aliasing* de comandos (do Git e do bash)
  - *Squashing*
  - Assinatura de *commits*
  - `rerere`
  - Substituição de histórico

COTIDIANO E PRODUTIVIDADE

- Resumo até agora:
  - O Git é uma ferramenta bastante complexa, no entanto possui muitos recursos para apoiar a edição de código (e que vão muito além de versionar)
  - Uma das dicas mais importantes, que tem forte impacto em diversos dos comandos, é a de manter nossos *commits* pequenos e bem documentados

COTIDIANO E PRODUTIVIDADE

- Resumo até agora:
  - Cada recurso do Git possui uma finalidade bem específica, e ao conhecermos bem cada um deles, desenvolvemos a capacidade de escolher a melhor ferramenta para a tarefa em mãos
  - Ao conhecermos o funcionamento interno do Git, fica mais fácil reconhecermos a razão de cada funcionalidade e agruparmos em nosso modelo mental funções que pareciam desconexas

COTIDIANO E PRODUTIVIDADE

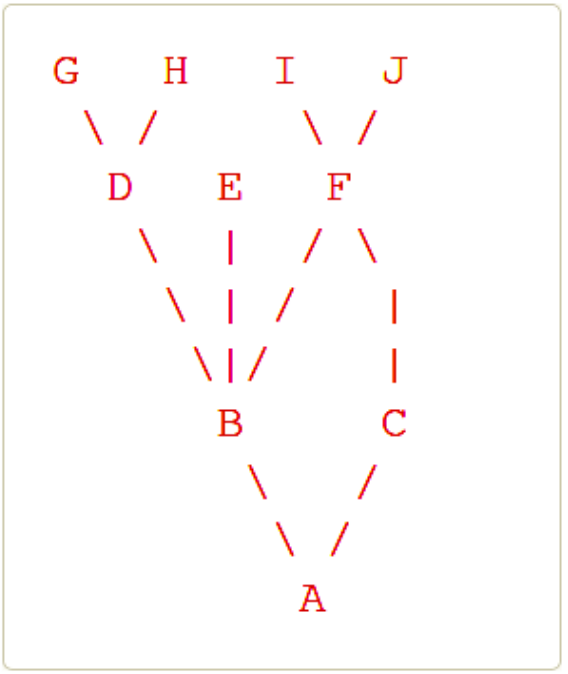
- Já somos capazes de:
  - Realizar operações complexas de divergência e mesclagem de código
  - Produzir e desfazer alterações em nosso repositório
  - Sincronizar as alterações com repositórios remotos
  - Buscar linhas de código e *commits* ao longo de todo o histórico
  - Aumentar nossa produtividade com IDEs, ferramentas gráficas e alguns comandos úteis

COTIDIANO E PRODUTIVIDADE

- Exercícios:
  - Questões de concursos públicos:
    http://jkolb.com.br/questoes-git/
  - Jogo educativo de Git (em inglês):
    https://learngitbranching.js.org/?demo
  - Mini-simulador de Git (em inglês):
    https://git-school.github.io/visualizing-git/

COTIDIANO E PRODUTIVIDADE

- Desafio de notação de revisões:




```
A =       = A^0
B = A^    = A^1      = A~1
C         = A^2
D = A^^   = A^1^1    = A~2
E = B^2   = [redacted]^2
F = [redacted]  = A^^3
G = A^^^  = A^1^1^1  = A~3
H = [redacted]  = B^^2     = [redacted]^^^2  = [redacted]^2
I = F^    = B^3^     = A^^3^
J = [redacted]  = [redacted]^3^2   = [redacted]^^3^2
```


Adaptado de https://git-scm.com/docs/gitrevisions