REVISÕES DE PROJETO, REVISÕES DE CÓDIGO E SISTEMAS DE CONTROLE DE VERSÕES

ACH2006 – ENGENHARIA DE SISTEMAS DE INFORMAÇÃO
SIN5005 – TÓPICOS EM ENGENHARIA DE SOFTWARE

Daniel Cordeiro

Escola de Artes, Ciências e Humanidades | EACH | USP

Revisões de projeto/código

- Revisão de projeto (*design review*): encontro onde os autores apresentam o projeto
  - aproveita-se o conhecimento dos participantes da reunião
- Revisão de código (*code review*) realizada depois da implementação do projeto

AGENDA DE UMA REVISÃO

- Prepare uma lista de perguntas/problemas que você gostaria de discutir
- Comece com uma descrição de alto nível do que o cliente quer
- Mostre a arquitetura do software, mostrando as APIs e deixando claro os padrões de projeto usados em cada nível de abstração
- Percorra **o código e a documentação**: planejamento do projeto, cronograma, plano de testes, ... mostre a *Verificação & Validação* (V & V) do projeto

RECEITA PARA UMA BOA REUNIÃO: SAMOSAS

- *S*tart (comece) e pare uma reunião no momento certo
- Agende os tópicos a serem tratados antes da reunião; se não houver uma agenda, cancele a reunião
- *M*inutes (atas) devem ser registradas para que, posteriormente, os resultados sejam lembrados por todos; o primeiro item da agenda é encontrar um escrivão
- *O*ne (um) falante por vez; não interrompa quando o outro estiverem falando
- *S*end (envie) o material antes da reunião, já pessoas leem mais rápido do que falam
- *A*ction items (liste o que deve ser feito), no final do encontro, para que as pessoas saibam o que fazer como resultado da reunião
- *S*et (estabeleça) a data e o horário do próximo encontro

A lista do que deve ser feito resultante da última reunião deve ser a primeira coisa a ser apresentada no início da próxima reunião.

COMO MELHORAR AS REVISÕES?

- Alan Shalloway[1]: o projeto formal e revisões de código são feitos muito tarde no processo para terem um grande impacto
- Recomenda fazer isso mais cedo, com reuniões menores chamadas de “revisões de abordagem” (*approach reviews*)
  - alguns desenvolvedores mais experientes ajudam o time a preparar uma primeiro abordagem para resolver o problema
  - *brainstorms* com o grupo para identificar abordagens diferentes
- Se for fazer uma revisão formal do projeto, sugere que se faça primeiro uma *mini revisão do projeto* para se preparar

[1] Agile Design and Code Reviews, 2002, `http://www.netobjectives.com/download/designreviews.pdf`

MÉTRICAS QUANTITATIVAS VS. REVISÕES

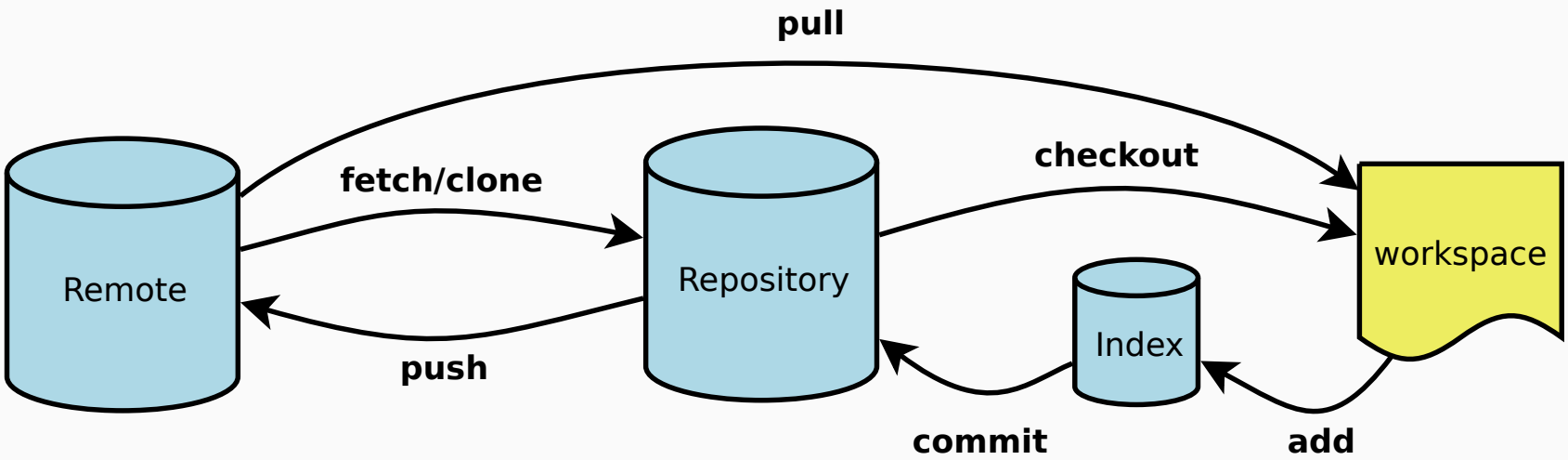
- Estude muitos projetos e guarde as médias, defina como base para novos projetos e compare:
  - tamanho do código (KLOC), esforço (meses)
  - marcas (*milestones*) cumpridas, casos de testes realizados
  - defeitos descobertos, taxa de reparo / mês
- Será que esses valores são correlatos e poderiam substituir as revisões?

  "Entretanto, nós ainda estamos muito longe dessa situação ideal e não há sinais de que um método de avaliação automática de qualidade se torne realidade em um futuro próximo" — Sommerville, 2010

- Pivotal Labs — programação pareada implica em revisões contínuas ⇒ não fazem revisões especiais
- GitHub — Pull Requests[2] ao invés de revisões
  - um desenvolvedor pede para que seu código seja integrado à base de código
  - todos os desenvolvedores veem cada pedido e determinam como ele pode afetar seu próprio código
  - se houver problema, uma discussão online se inicia na página com o pedido
  - essas "mini revisões" acontecem diariamente ⇒ não fazem revisões especiais

[2] https://help.github.com/articles/about-pull-requests/

Usando ramificações de forma eficaz: branches de funcionalidade

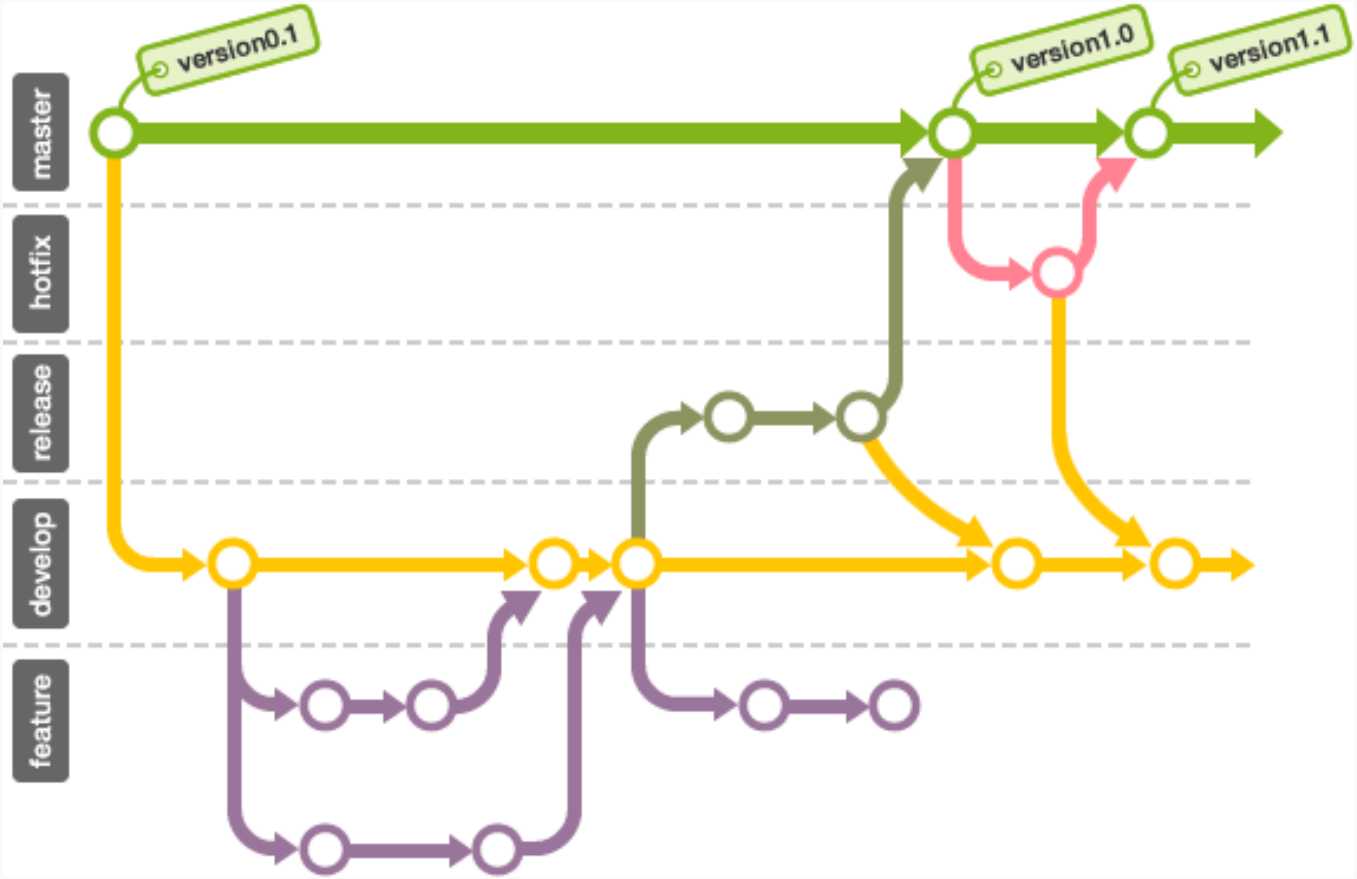


Veja uma boa introdução em:
`https://github.com/HackBerkeley/intro-git`

RAMOS (BRANCHES)

- Desenvolvimento no ramo principal (*master*) vs. *branches*
  - criar um *branch* é barato!
  - para escolher outro *branch*: `checkout`
- Separa o histórico de *commits* por *branch*
- Faça o *merge* do *branch* de volta ao *master*
  - ... ou faça o *push* das mudanças feitas no *branch*
  - a maior parte dos *branches* eventualmente morrem
- A melhor aplicação da ideia de *branches* em apps SaaS Ágeis: *branch* de funcionalidade

RAMOS (BRANCHES)



Criando novas funcionalidades sem atrapalhar o código que já funciona

1. Para trabalhar em uma nova funcionalidade, crie uma nova *branch* **apenas para aquela funcionalidade**
   - muitas funcionalidades podem estar em desenvolvimento simultaneamente
2. Use o *branch* **apenas** para as mudanças necessárias para **essa funcionalidade**, depois faça o *merge* no *master*
3. Remover essa funcionalidade $\iff$ desfazer o *merge*

Em um app bem fatorado
Uma funcionalidade não toca muitas partes de uma aplicação.

MECÂNICA

- Para criar um novo *branch* e usá-lo

```
git branch funcionalidade-nova
git checkout funcionalidade-nova
```

- Edite, adicione, faça *commits*, etc. no *branch*
- Faça um *push* das mudanças para o repositório original (opcional)

```
git push origin funcionalidade-nova
```

  - cria um *tracking branch* no repositório remoto

- Volte para o *branch master* e faça o *merge*:

```
git checkout master
git merge funcionalidade-nova
```

REBASE & PULL REQUEST

- Fazer o *rebase* de um *branch* em x significa fingir que o ramo foi criado a partir de x
- Pra quê?
- Os conflitos de *merge* devem ser resolvidos manualmente, assim como um *merge* comum
  - opcional: *squash* (“esprema”) múltiplos *commits* em um, para simplificar um *merge* futuro
  - *pull request* pode ser do mesmo repositório ou de uma cópia *forked* do repositório
- Só então faça o *pull request* no *master*
- Por quê usar isso ao invés de *merge*?
- Veja:
  `https://developer.atlassian.com/blog/2014/12/pull-request-merge-strategies-the-great-debate/`

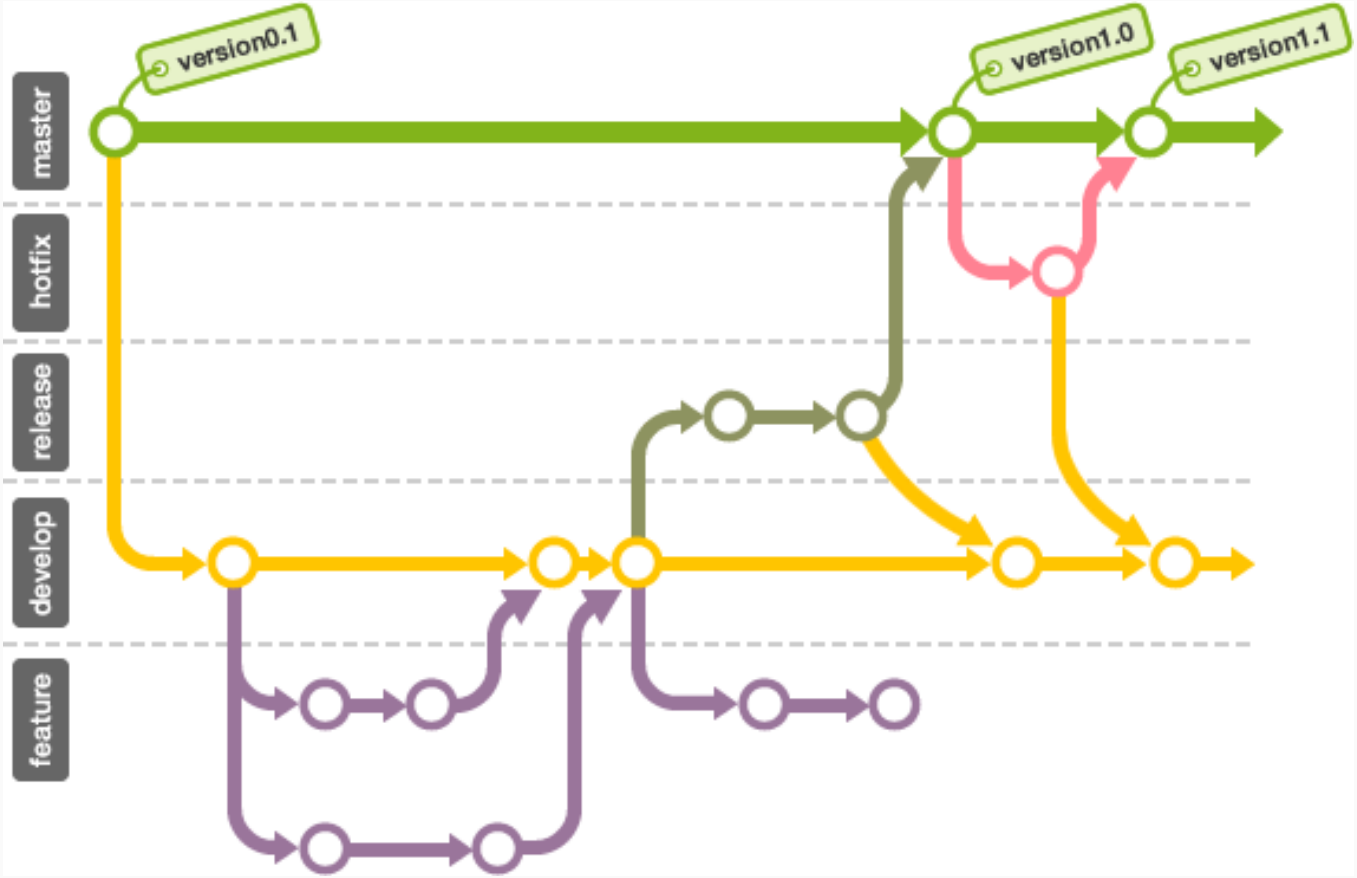
PULL REQUESTS PODEM SER USADOS COMO REVIÕES DE CÓDIGO

- Abrir um Pull Request significa esperar que outros membros do time façam uma revisão do código e comentem as modificações
  - no Google, nenhum código é incorporado à base de código sem que ao menos 1 pessoa revise o código, mesmo que seja só para dizer *"Looks good to me"* (LGTM)
- Dependendo do resultado da revisão, o PR pode ser fechado (retirado) ou revisto antes do *merge*
- Os desenvolvedores do GitHub trabalham exclusivamente desse jeito

Branches & implantações

- *Branches* de funcionalidade devem ter vida curta
  - senão, o ramo vai ficando dessincronizado e vai ficando cada vez mais difícil reconciliar
  - `git rebase` pode ser usado para fazer um *merge* "incremental"
  - *git cherry-pick* pode ser usado para fazer o *merge* de *commits* específicos
- "Implantar do *master*" é o mais comum
- Usar "um *branch* por lançamento" é uma estratégia alternativa

RAMOS DE LANÇAMENTO/CORREÇÕES E CHERRY-PICKING



Justificativa
O ramo de lançamento é um pedaço estável do código onde correções podem ser incrementais.

BRANCH VS. FORK

- Git permite dois modelos de colaboração: *fork* & *pull*
- Se você tem permissão para fazer *push* no repositório:
  - *branch*: crie um *branch* neste repositório
  - *merge*: combine as mudanças do *branch* no *master* (ou em outro *branch*)
- Se você não tiver:
  - *fork*: faça o clone do repositório que está no GitHub em um lugar onde você possa criar *branches*, fazer *push*, etc.
  - Termine seu trabalho no seu próprio *branch*
  - Cortesia: faça o *rebase* do seu *branch* com *commit squash*
  - Crie um *pull request* para que o responsável faça *pull* do seu *commit*

ARMADILHAS

- “Atropele” suas mudanças depois de fazer um *merge* ou uma troca de *branches*
- Faça mudanças “simples” diretamente no *branch master*

DESFAZER

- Para desfazer as modificações locais:

```
git reset --hard ORIG_HEAD
git reset --hard HEAD
git checkout commit-id  -- arquivos …
```

- Para comparar ou entender o que aconteceu:

```
git diff commit-id-or-branch -- arquivos …
git diff "master@{01-Sep-12}" -- arquivos …
git diff "master@{2 days ago}" -- arquivos …
git show mydevbranch:myfile.rb
git blame arquivos
git log arquivos
```

Corrigindo erros de programação

NÃO CORRIJA ERROS SEM UM TESTE!

1. **R**elate
2. **R**eproduza o problema e/ou **R**eclassifique-o
3. Teste por **R**egressões
4. **R**epare
5. **R**elance o código reparado (*commit* ou implante)

- Faça isso mesmo se sua empresa não segue métodos Ágeis
- Processos ágeis podem ser adaptados para a correção de erros de programação

- Pivotal Tracker
  - erros de programação (*bugs*) = 0 pontos de história (o que não significa zero esforço)
  - automação: os *service hooks* do GitHub podem ser configurados para marcar a história do Tracker como “entregue” quando for feito o *push* com um *commit* devidamente anotado
- GitHub *issues*
- Sistemas de controle de bugs (ex: Bugzilla)
- Use a ferramenta mais simples que funcionar para a sua equipe & escopo de projeto

RECLASSIFICAR? OU REPRODUZIR + REPARAR?

- Reclassifique como “não é um erro” (*not a bug*) ou “não será corrigido” (*won’t be fixed*)
- Reproduza o erro com o *teste mais simples possível* e adicione-o ao conjunto de testes de regressão
  - minimize as pré-condições (ex: blocos `before` no RSpec ou `Given` e `Background` no Cucumber)
- Reparar == teste falha na presença do erro, mas passa na ausência dele
- Lançamento: pode significar tanto fazer o *push* no repositório como fazer uma nova implantação

Falácias & Armadilhas, Comentários finais sobre o Capítulo 10

- Dividir o trabalho baseado na pilha do software ao invés de dividir baseado em funcionalidades
  - Ex: especialista em front-end/back-end, em relacionamento com cliente (*customer liaison*), etc.
- Métodos Ágeis: resultados melhores se cada membro da equipe entrega todos os aspectos de uma história
  - cenários Cucumber, testes RSpec, visões, ações de controlador, lógica de modelo, etc.
  - Todo mundo na equipe tem uma visão de “pilha inteira” do produto

ARMADILHA

- “Atropelar” acidentalmente as mudanças depois de fazer um *merge* ou uma troca de *branches*
  - no *branch* errado, sobrescrever as mudanças incorporadas com um *merge* ao gravar uma versão velha aberta no editor, etc.
- Antes de fazer *pull* ou *merge*, faça o *commit* de todas as modificações
- Depois de fazer *pull* ou *merge*, recarregue todos os arquivos no editor
  - ou fechar o editor antes de fazer o *commit*

ARMADILHA

- Deixar sua cópia do repositório ficar muito defasada em relação à versão de origem
  - o que significa que fazer o *merge* pode ficar bem complicado
- Faça um `git pull` antes de começar e um `git push` assim que as mudanças que você fez *commit* localmente ficarem estáveis o suficiente
- Se essa *branch* tiver longa duração, faça `git rebase` periodicamente

FALÁCIA

- Tudo bem fazer pequenas mudanças no *branch master*
  - você primeiro acha que é uma mudança de 1 linha, mas depois vira 5, mexe com outro arquivo, aí precisa mexer nos testes, ...
- Sempre crie um *branch* de funcionalidade antes de começar um novo trabalho
  - criar um *branch* é quase que instantâneo no Git
  - se a mudança *for pequena*, você pode apagar o *branch* logo depois de fazer o *merge* para evitar que o seu espaço de nomes de *branches* fique bagunçado

COMENTÁRIOS FINAIS

- “Equipes de 2-pizzas” reduzem o problema de gerenciamento se comparados com “equipes de banquetes”, mas não acabam com o problema
  - Scrum é um modo informal de organização que casa bem com Desenvolvimento Ágil
- Pontos, Velocidade, Tracker $\Rightarrow$ mais previsíveis
- P-e-D: Gerente de projeto é o chefe; Revisões são formas de aprender com os outros
- Quando o projeto estiver pronto, separe um tempo para pensar no que vocês aprenderam com ele antes de pular para o próximo
  - o que funcionou bem, o que não funcionou, o que podemos fazer diferente

Os 10 mandamentos para ser um mau jogador em uma equipe de software

```
git commit -m 'se vira' &&
git push --force origin master
```

os 10 mandamentos (e sugestões de alternativas)

1.	Essas falhas não são importantes	Nunca faça *push* no vermelho
2.	Meu *branch*, meu santuário	Mantenha *branches* de curta duração
3.	É uma mudança simples	*Mount a scratch monkey*[3]
4.	Eu sou especial	1 projeto, 1 estilo de código
5.	Tabs salvam bytes valiosos	Não use tabs[4]
6.	Inteligência é impressionante	Transparência é humildade
7.	Faça a mudança rapidinho no servidor de produção	Faça com que toda mudança seja automatizável
8.	Tempo gasto procurando pelas coisas = tempo perdido não codificando	Gaste 5 minutos procurando por código menor/melhor
9.	"Febre verde": pegue você também	Mais testes ≠ mais qualidade
10.	Semanas programando pode salvá-lo de horas pensando/planejamento	Reflita sobre o *design*

[3] Veja: `http://www.catb.org/jargon/html/S/scratch-monkey.html`
[4] Silicon Valley S03E06 — Tabs vs. Spaces: `https://youtu.be/SsoOG6ZeyUI`

MANDAMENTOS EXTRAS PARA UM PROJETO GERENCIÁVEL

- Qualquer método com flog > 10 é rejeitado
- Qualquer *branch* com duração > 3 dias é destruído
- Qualquer *merge* que quebre a compilação/testes é destruído e o responsável **deve** fazer o *rebase* no *master*
- Qualquer correção de erro ou código novo submetido com menos de 90% de cobertura é rejeitado