GESTÃO DE PROJETOS: SCRUM, PROGRAMAÇÃO PAREADA E SISTEMAS DE CONTROLE DE VERSÃO

ENGENHARIA DE SISTEMAS DE INFORMAÇÃO

Daniel Cordeiro

14 de novembro de 2017

Escola de Artes, Ciências e Humanidades | EACH | USP

ENGENHARIA DE SOFTWARE É UM ESPORTE COLETIVO

- Estamos na era pós-desenvolvedor-super-herói :)
- Expectativas sobre funcionalidades/qualidade mais altas
  - um único programador brilhante não é mais capaz de construir sozinho software completamente inovador
- Carreira de sucesso em SW ⇒ excelente programador && trabalha bem com outras pessoas && consegue ajudar uma equipe a ser bem sucedida
- *"Não há vencedores em uma equipe mal sucedida, nem perdedores em uma equipe bem sucedida."*
  –Fred Brooks Jr.

EXEMPLO: VIDEOGAMES

Jogo	Ano	Plataforma	Equipe de Devs.
Space Invaders	1981	Arcade	1

Jogo	Ano	Plataforma	Equipe de Devs.
Space Invaders	1981	Arcade	1
Super Mario Bros.	1985	NES (Nintendo)	8

EXEMPLO: VIDEOGAMES

Jogo	Ano	Plataforma	Equipe de Devs.
Space Invaders	1981	Arcade	1
Super Mario Bros.	1985	NES (Nintendo)	8
Sonic the Hedgehog	1999	Sega Dreamcast	30

Jogo	Ano	Plataforma	Equipe de Devs.
Space Invaders	1981	Arcade	1
Super Mario Bros.	1985	NES (Nintendo)	8
Sonic the Hedgehog	1999	Sega Dreamcast	30
Resident Evil 6	2012	PC, PS3, Xbox 360	600

ORGANIZAÇÃO ALTERNATIVA DA EQUIPE?

- Planeje-e-Documente requer documentação e planejamento extensivos e depende de um gerente experiente
- Como deveríamos organizar uma equipe Ágil?
- Há alguma alternativa a organização hierárquica, com um gerente que executa o projeto?

SCRUM: ORGANIZAÇÃO DA EQUIPE

- Equipe de “2 pizzas” (4 a 9 pessoas)
- “Scrum” inspirado em reuniões curtas e frequentes:
  - reuniões de 15 minutos sempre na mesma hora e lugar
  - para aprender mais: *Agile Software Development with Scrum*, de Schwaber & Beedle

REUNIÃO DIÁRIA DE SCRUM

- Todos devem responder a três perguntas:
    1. O que você fez desde a reunião de ontem?
    2. O que você planeja fazer hoje?
    3. Existe algum impedimento ou obstáculo?
- Ajuda cada membro da equipe a identificar o que ele precisa

PAPÉIS NO SCRUM

- Equipe: uma equipe de tamanho “2 pizzas” que entrega o software
- ScrumMaster: o membro da equipe que:
  - protege a equipe de distrações externas
  - mantém a equipe focada no trabalho em questão
  - impõe regras (ex: padrões de código)
  - remove os obstáculos que impedem a equipe de progredir
- Proprietário do produto (*product owner*): um membro da equipe (que não o ScrumMaster) que representa a voz do cliente e prioriza as histórias de usuário

O scrum baseia-se na auto-organização e os membros da equipe frequentemente se revezam entre funções diferentes.

RESOLUÇÃO DE CONFLITOS I

Conflitos podem ocorrer (como em qualquer trabalho em equipe).
Ex: visões contrárias sobre a direção técnica que deverá ser seguida.

1. 1º liste os itens nos quais todos concordam:
   - não comece listando as divergências
   - muitas vezes percebe-se que eles estão mais de acordo do que imaginavam
2. cada lado tenta articular os argumentos do outro, mesmo que não concorde com tudo
   - evita confusão sobre os termos ou hipóteses, o que muitas vezes é a causa do conflito

RESOLUÇÃO DE CONFLITOS II

3. Confronto construtivo (Intel) — se você discorda veemente de uma proposta, você é obrigado a contestá-la (mesmo para os seus chefes)
4. Discordar, mas se comprometer (Intel)
    - uma vez que a decisão for tomada, você deve abraçá-la e seguir em frente
    - “Eu discordo, mas vou ajudar a fazer mesmo que eu discorde”.

Resolução de conflitos também pode ser útil na vida pessoal!

- Basicamente, uma pequena equipe auto-organizada com uma reunião curta (e de pé) todos os dias
- Trabalho dividido em “sprints” de 2–4 semanas
- Sugere que os membros troquem os papéis entre si (especialmente o proprietário do produto) a cada iteração

Créditos: https://commons.wikimedia.org/wiki/File:ST_vs_Gloucester_-_Match_-_23.JPG

DICA PARA USO DE SCRUM

- “As reuniões em pé de 5 minutos realmente nos ajudaram a ficar na linha e a compartilhar conhecimento quando estávamos empacados”

- “O maior desafio para nós era a comunicação/coordenação do time”
- “Tenha um líder scrum por vez, rotacione o cargo”
- “1 reunião por semana não era suficiente”

Qual afirmação relacionada a equipes é verdadeira?

1. Se compararmos P-e-D com Scrum, gerentes de projetos P-e-D agem tanto como o ScrumMaster como quanto o proprietário do produto
2. Equipes devem evitar conflitos entre seus membros a todo custo
3. P-e-D pode ter equipes muito maiores do que as equipes Scrum, com cada equipe reportando diretamente para o gerente de projeto
4. Como mostram os estudos, 84–90% dos projetos terminam no prazo e no orçamento. Gerentes P-e-D podem prometer um conjunto de funcionalidades a seus clientes com toda confiança de chegarão a um acordo sobre o custo e o prazo

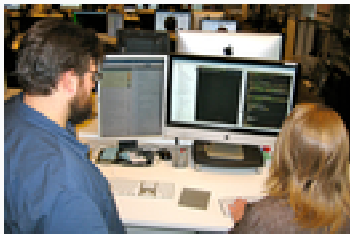
Programação pareada

DUAS CABEÇAS PENSAM MELHOR DO QUE UMA?

- Estereótipo: um “lobo solitário” trabalhando a noite toda à base de energéticos
- Será que há uma maneira mais sociável de programar?
  - Quais seriam os benefícios de termos várias pessoas programando juntas?
- Como você faria para evitar que uma pessoa fizesse todo o trabalho, enquanto a outra pega um café e checa seu Facebook?

Objetivo
Melhorar a qualidade do software e reduzir o tempo para fazer uma tarefa ao manter 2 pessoas desenvolvendo o mesmo código.

- Algumas pessoas (e empresas) adoram
- Alguns alunos também gostavam e aplicavam isso em seus projetos



- Sente lado a lado colocando os monitores voltados para si
- Os computadores não são “pessoais”; são para uso dos pares
- Para evitar distrações, nada de ler e-mails ou navegar na web

PROGRAMAÇÃO PAREADA

- O piloto digita o código e pensa taticamente em como completar a tarefa atual, explicando seus pensamentos em voz alta enquanto digita
- O navegador revê cada linha de código assim que ela é digitada, agindo como uma rede de segurança para o piloto
- O navegador pensa estrategicamente sobre os problemas futuros, fazendo sugestões para o piloto
- Requer bastante conversa e concentração
- **Os pares alternam os papéis**

AVALIAÇÃO DE PROGRAMAÇÃO PAREADA

- PP é mais rápida quando a tarefa é mais simples
- PP alcança maior qualidade quando a tarefa é mais difícil
  - ... e código mais legível também
- Mas requer mais esforço de concentração do que programação solo
- Além disso, funciona como ferramenta de transferência de conhecimento
  - expressões idiomáticas de linguagens de programação, truques de ferramentas, processos da empresa, tecnologias mais novas, etc.
  - algumas equipes forçam a troca dos pares a cada nova tarefa; eventualmente todo mundo é pareado com todo mundo ("pareamento promíscuo")

PROGRAMAÇÃO PAREADA: O QUE (NÃO) FAZER

- Não fique mexendo no seu *smartphone* enquanto estiver de navegador
- Considere formar um par com alguém que não tenha o mesmo nível de experiência que você — vocês dois irão aprender com isso!
  - tentar explicar é uma ótima forma de entender algo
- Troque o papel frequentemente — cada papel exercita um conjunto de habilidades diferentes e você aprenderá algo com ambos
  - o navegador aprimora sua capacidade de explicar suas ideias ao piloto

Qual afirmação relacionada à Programação Pareada é verdadeira?

1. Programação pareada é mais rápida, produz melhor qualidade, é mais barata e requer menos esforço do que programação solo
2. O Piloto trabalha na tarefa atual, o navegador pensa estrategicamente nas próximas tarefas
3. Um par eventualmente perceberá quem é o melhor como piloto e como navegador, e, então, ficarão sempre nesses papéis
4. Programação promíscua é uma solução de longo prazo para o problema de falta de programadores

- “Nos ajudou a evitar erros bobos que poderiam ter nos custado bastante tempo para depurar”
- “A troca de pares frequentemente acabou deixando o time mais coeso”

Revisões de Projeto, Revisões de Código e a Perspectiva Planeje-e-Documente sobre a gerência de projetos

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

- *Start* (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
- *Minutes* (atas) devem ser registradas para que, posteriormente, os resultados sejam lembrados por todos; o primeiro item da agenda é encontrar um escrivão
- *One* (um) falante por vez; não interrompa quando o outro estiverem falando
- *Send* (envie) o material antes da reunião, já pessoas leem mais rápido do que falam
- *Action items* (liste o que deve ser feito), no final do encontro, para que as pessoas saibam o que fazer como resultado da reunião
- *Set* (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.

- 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

MÉTODOS ÁGEIS E REVISÕES

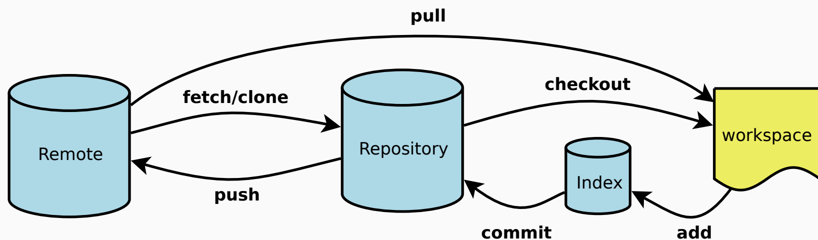
- Pivotal Labs — programação pareada implica em revisões contínuas $\Rightarrow$ não fazem revisões especiais
- GitHub — Pull Requests[2] ao invés de revisões
  - um desenvolvedor pede para que seu código seja integrado à base de código
  - todos os desenvolvedores veem cada pedido e determinam como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

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

Qual afirmação sobre Revisões e Reuniões é falsa?

1. Tentam melhorar a qualidade do software usando o conhecimento dos participantes
2. Elas resultam em troca de informações técnicas e são de grande valia como ferramenta educacional para o pessoal júnior
3. Podem ser benéficos tanto para os que apresentam como para os que assistem
4. O 'A' em SAMOSA significam *Agenda* e `Action itens`, que são características opcionais de boas reuniões

Usando ramificações de forma eficaz: branches de funcionalidade

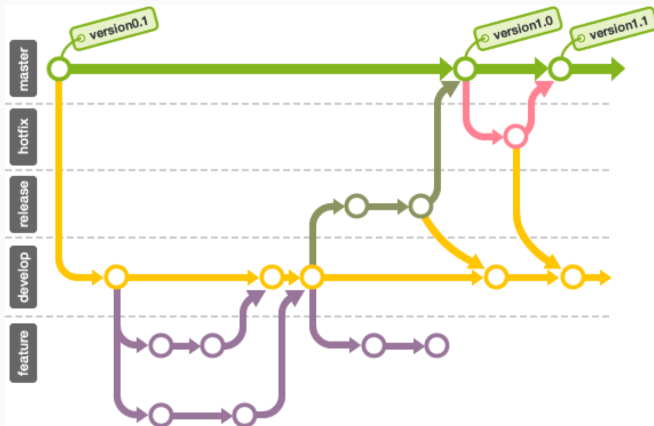


Veja uma boa introdução em:
`https://github.com/HackBerkeley/intro-git`

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.

- 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
```

- 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

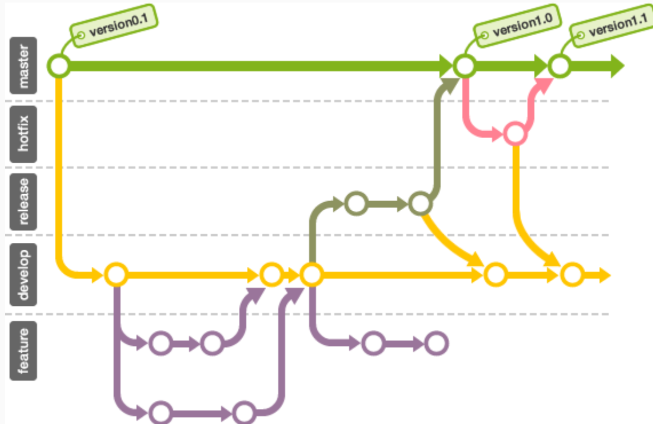
- Abrir um Pull Request significa esperar que outros membros do time façam uma revisão do código e comentem as modificações
    - no Google, nenhum código é incorporado à base de código sem que ao menos 1 pessoa revise o código, mesmo que seja só para dizer *"Looks good to me"* (LGTM)
- Dependendo do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

Se você tentar fazer um *push* e receber o erro `non-fast-forward (error): failed to push some refs`, qual afirmação é falsa?

1. Alguns *commits* feitos no repositório remoto não estão presentes no seu repositório local
2. Você precisa fazer um *merge*/*pull* antes de poder completar o *push*
3. Você precisa manualmente corrigir os conflitos no *merge* em um ou mais arquivos
4. Seu repositório local está desatualizado em relação ao remoto

Usando ramificações de forma eficaz: branches & implantação

- *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



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

- 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
```

PERGUNTA

Se subequipes diferentes forem designadas para trabalhar em *correções de uma versão já lançada* (*release bug fixes*) e em *novas funcionalidades*, você precisará usar:

1. Um *branch* por lançamento
2. Um *branch* por funcionalidade
3. Um *branch* por lançamento + um *branch* por funcionalidade
4. Qualquer um desses funcionará