CORRIGINDO ERROS DE PROGRAMAÇÃO

ENGENHARIA DE SISTEMAS DE INFORMAÇÃO

Daniel Cordeiro

16 de novembro de 2017

Escola de Artes, Ciências e Humanidades | EACH | USP

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

PERGUNTA

Suponha que você descobriu que a versão mais recente do programa contém um erro, e que para escrever o teste de regressão você irá precisar usar um monte de *mocks* e *stubs* porque o código onde o erro aparece é bem complicado. Qual ação não é apropriada?

1. Faça a refatoração, usando TDD, no *branch* de lançamento e faça um *push* com o código que corrige o erro e com novos testes
2. Faça a refatoração, usando TDD, em *branch* diferente, faça um *push* com o código que corrige o erro e com novos testes e então faça o *cherry-pick* da correção no *branch* de lançamento
3. Crie um teste de regressão com os *mocks* e *stubs* necessários (por mais difícil que fazer isso seja) e faça o *push* da correção e dos testes no *branch* de lançamento
4. Dependendo das prioridades do projeto e do gerenciamento do projeto, qualquer ação acima pode ser apropriada

Falácias & Armadilhas, Comentários finais sobre o Capítulo 10

ARMADILHA

- 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*[1]
4.	Eu sou especial	1 projeto, 1 estilo de código
5.	Tabs salvam bytes valiosos	Não use tabs[2]
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*

[1] Veja: `http://www.catb.org/jargon/html/S/scratch-monkey.html`
[2] 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