Código Limpo:

Habilidades Práticas do Agile Software

Capítulo 9: Testes de Unidade

Este material foi desenvolvido pelo grupo PET/ADS do IFSP São Carlos

Testes de Unidade - Introdução

São pequenos trechos de código utilizados para testar isoladamente uma parte do sistema de maneira rápida.

Se implementados corretamente trazem diversas vantagens:

- Tornam um sistema *quase* à prova de falhas .
- Facilitam muito a manutenção de um projeto.
- Direcionam o desenvolvimento, evitando más práticas.

As Três Leis do TDD

Primeira Lei: Escreva códigos de produção apenas para fazer testes unitários que estão falhando passarem.

Segunda Lei: Escreva não mais do que um teste unitário que falhe (erro de compilação também é uma falha).

Terceira Lei: Escreva apenas o necessário de código de produção para fazer o teste de unidade falhando passar.

Como manter os testes limpos

Ao fazer testes, quantidade não é sinônimo de qualidade. Ter testes malfeitos é equivalente ou até mais danoso do que não ter teste nenhum!

Quanto pior o teste mais difícil será mudá-lo; e existe maior chance dele se depreciar e bagunçar a manutenção do código de produção.

> *"Os códigos de testes são tão importantes quanto o código de produção. Ele não é componente secundário. Ele requer raciocínio, planejamento e cuidado. É preciso mantê-lo tão limpo quanto o código de produção."*

Os testes habilitam as “-idades”

São os testes que mantêm a *flexibilidade*, *manutenibilidade* e *reusabilidade* em um projeto e permitem alterações ou expansões do mesmo.

Sem eles cada modificação pode gerar um bug, mas com os testes você pode fazer mudanças quase sem penalidade ao código de produção!

Quando não se tem testes, há medo de alterar o código e ele degrada. Com testes, você sabe que pode refatorar e melhorar o sistema, preservando o comportamento anterior.

Testes Limpos

O que torna um teste limpo é a **LEGIBILIDADE**.

E as coisas que produzem um teste legível são as mesmas que produzem códigos legíveis:

- **Clareza**
- **Simplicidade**
- **Consistência**

> *"Num teste você quer dizer muito com o mínimo de expressões possíveis."*

Testes Limpos

Veja o seguinte código de teste:

```
public void testGetPageHierachyAsXML() throws Exception {

  crawler.addPage(root, PathParser.Parse("PageOne"));
  crawler.addPage(root, PathParser.Parse("PageOne.ChildOne"));
  crawler.addPage(root, PathParser.Parse("PageTwo"));

  request.setResource("Root");
  request.addInput("type", "pages");
  Responder responder = new SerializedPageResponder();
  SimpleResponse response = (SimpleResponse) responder.makeResponse(new FitnessContext(root), request);
  String xml = response.getContent();

  assertEquals("text/xml", response.getContentType());
  assertSubstring("<name>PageOne</name>", xml);
  assertSubstring("<name>PageTwo</name>", xml);
  assertSubstring("<name>ChildOne</name>", xml);
}
```

Testes Limpos

Coisas que atrapalham na leitura de testes:

- Duplicação de código.
- Necessitar saber muitos detalhes sobre o código para entender o testes.
- Linhas não relevantes ao teste em si:
  - Detalhes da criação de objetos.
  - Montagem de requisições.

Testes Limpos

Para estruturar testes limpos utilize o padrão **Construir**, **Operar**, **Verificar**:

1. Produzir dados de teste.
2. Operar em cima dos dados.
3. Verificar se os dados gerados foram os esperados.

Ao dividir o teste nestas três partes, o código maçante é eliminado e fica muito legível!

Testes Limpos

Agora veja o código refatorado:

```
public void testGetPageHierachyAsXML() throws Exception {
  makePages("PageOne", "PageOne.ChildOne", "PageTwo");

  submitRequest("root", "type:pages");

  assertResponseIsXML();
  assertResponseContains(
     "<name>PageOne</name>",
     "<name>PageTwo</name>",
     "<name>PageOne.ChildOne</name>"
  );
} // Bem mais entendível, não?
```

Linguagem de testes específica ao domínio

Ao invés de usar API's diretamente nos testes, o ideal é criar funções e utilitários ligados ao domínio da aplicação, que se utilizam delas para testar o software.

Exemplos de uso:

- Funções que criam e populam automaticamente um banco de dados.
- Montagem de requisições comumente usadas.
- Resumir qualquer trecho que tenha outra função senão testar o código.

Um padrão duplo

Um código dentro da API de teste tem um conjunto diferente de padrões de engenharia em relação ao código de produção.

Em certos sistemas há muita preocupação com espaços de memória e desempenho, há certos códigos que são inviáveis em ambientes de produção reais.

Porém, no ambiente de testes a perspectiva é outra: não tenha medo de sacrificar um pouco de recurso do sistema por mais legibilidade e manutenibilidade dos testes!

Uma confirmação por teste

Diversas asserções em uma função não são um grande problema, mas testes com apenas uma asserção chegam mais rápido em conclusões e são mais fáceis de entender:

- Testes com múltiplas asserções podem ser divididos em vários com uma em cada.
- Utilize meios de inicializar código repetível para não duplicá-lo (@Before).
- Utilize padrão given-when-then para organizar os testes (construir-operar-verificar).

Um único conceito por teste

Funções longas que testam múltiplas requisições do domínio são ruins para o código e terríveis para os olhos!

Juntar coisas distintas em uma mesma função obriga o leitor a entender o contexto de todas elas para compreender o teste completo.

Provavelmente, a melhor regra seja minimizar o número de asserções por conceito e testar apenas um conceito por função de teste.

F.I.R.S.T.

Testes devem seguir cinco regras para serem considerados testes limpos:

- **Rapidez**: os testes devem ser rápidos e executar com rapidez.
- **Independência**: testes devem ser independentes uns dos outros.
- **Repetitividade:** deve ser possível reproduzir os testes em qualquer ambiente.
- **Autoavaliação**: testes devem ter uma saída booleana, sucesso ou falha.
- **Pontualidade**: testes devem ser criados ANTES do código de produção.

Conclusão

Testes são extremamente importantes em um projeto, pois preservam e aumentam a *flexibilidade*, *capacidade de manutenção* e *reutilização do código*.

- Trabalhe para torná-los curtos, mas expressivos.
- Crie funções utilitárias baseados no domínio para ajudá-lo a fazer testes.

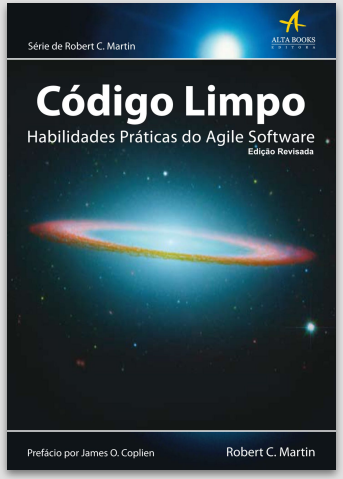
"Se deixar os testes se degradarem, seu código também irá."

Sobre o autor

Robert C. “Uncle Bob” Martin é desenvolvedor e consultor de software desde 1990. Ele é o fundador e o presidente da Object Mentor, Inc., uma equipe de consultores experientes que orientam seus clientes no mundo todo em C++, Java, C#, Ruby, OO, Padrões de Projeto, UML, Metodologias Agile e eXtreme Programming.

Este conjunto de slides foi elaborado a partir da obra:

MARTIN, Robert. **Código Limpo:**
Habilidades Práticas do Agile Software. 1. ed.
Rio de Janeiro: Alta Books, 2009.



A equipe

Vitor Bonelli, Autor

Aluno e bolsista do PET/ADS desde março de 2021
LinkedIn

Lucas Oliveira, Revisor

Professor de Computação, é tutor do PET/ADS desde janeiro de 2023.
LinkedIn