Notas sobre o capitulo 9 de Clean Code – by Lucas Trevizan

Testes de unidade

Tio Bob diz que testes não eram comuns lá na era deles, os testes eram pensados simplesmente como "código que valida o que eu realmente desenvolvi". Dessa maneira era comum escreve-los, mostrar e descartar.

Claro que com o passar do tempo os testes foram ganhando mais peso na aplicação e começou a ser levado de uma maneira mais profissional, não atoa se fala muito em TDD, mas o ponto do tiozão é nos lembrar de coisas simples que são esquecidas.

As três leis do TDD

Primeira lei: *Não se escreve o código de implementação antes de ter escrito o de teste.*

Segunda lei: *Não se escreve mais de um teste do que o necessário para FALHAR, e não compilar é falhar (Aqui a ideia é "pra falhar nesse cenário eu tenho que usar UM teste de falha).*

Terceira lei: *Não se escreve mais código de implementação do que você precisa para falhar no teste*

Se reparar bem nas leis, dá pra ver que elas visam manter uma coesão dos testes de maneira que até mesmo eles fiquem limpos e o código de implementação seja simples. Outro ponto que ele levanta é que usar essa prática frequente tende a ter um código de testes imenso e intimidador.

Como manter os testes limpos

Algumas pessoas fazem o teste de "qualquer jeito", esse qualquer jeito está ligado a más práticas que basicamente são "melhor isso do que nada". Esse tipo de pensamento é péssimo em todos os quesitos possíveis, uma funcionalidade tende a ser modificada o tempo todo e consequentemente isso vai fazer os testes quebrarem.

Agora imagina olhar um monte de testes com nomes nada descritivos e feitos só pra passar de qualquer jeito e garantir o mínimo de cobertura para ir em produção. Onde trabalhei me deparei com vários testes que seguiam esse preceito, realmente eles eram sujos e facilmente mal interpretados na leitura, o ponto é que o **código de teste TAMBÉM É UM CÓDIGO** e, portanto, deve se manter limpo.

Testes habilitam as "-idades"

São os testes que permitem modificações seguras no código implementado, afinal, com uma boa cobertura você perde o medo de refatorar um código esquisito que pode quebrar outra parte do programa sem que você saiba.

Testes limpos

O que torna um teste limpo é a mesma coisa que também torna um código limpo, LEGIBILIDADE. Quanto mais legível o teste for, mais fácil é entende-lo e aqui ainda podemos ver uma estrutura que é um dos maiores padrões para um teste: **CONSTRUIR-OPERAR-VERIFICAR (*cenário-execução-assertiva).***

Esse estilo de escrita de código de teste é um padrão universal que facilita tanto a leitura quanto a escrita.

Linguagem de testes específicas ao domínio

Talvez seja mais eficiente para o software se os testes usarem uma API própria, isso me lembra do meu trabalho onde usamos mocks próprios adptados para o framework do cliente.

O ganho aqui talvez seja simplificar ainda mais as leituras de testes e ter uma "integração" melhor com o que é desenvolvido.

Um padrão duplo

O padrão duplo nos fala sobre coisas que você pode fazer em um ambiente de teste que não vai fazer no ambiente de produção/implementação e isso está ligado com performance.

O que faz sentido, quando eu trabalhava em projetos batch muitas vezes eu precisava de um cache de memória ou trabalhar bem com os recursos, já em testes eu só preciso me assegurar de que meu processo esteja funcionando.

Uma confirmação por teste

Alguns "puritanos" acreditam que cada teste deve ter apenas uma confirmação (Assertiva), o tio Bob diz que tudo bem se você usar o mínimo possível de confirmação já que ele não tem receio de fazer mais de uma confirmação por teste.

Eu sou mais da primeira linha de pensamento, cada teste deve ser diretamente sobre um cenário, claro que isso pode gerar códigos repetidos mas o JUnit oferece alternativas como o @BeforeEach/@BeforeAll para lidar com isso, ou até mesmo podemos adaptar a classe para abrigar as variáveis repetidas.

Um único conceito por teste

Talvez o melhor meio termo seja aplicar um conceito por teste, dessa maneira não ficaríamos limitados a vários testes em diferentes cenários e nem a um teste que faz tudo, mas, teríamos um CONCEITO que mantém a coesão do que vai ser testado.

F.I.R.S.T

First é um acrônimo formado por princípios que devem ser aplicados no teste.

Fast/Rápido: Auto explicativo, um teste automatizado precisa ser rápido a ponto de que não tenhamos preguiça de executa-lo e dessa maneira ganhamos a disposição de refatorar.

Independent/Independente: Um teste não pode depender de outro, eles têm que ter a capacidade de serem executados de maneira independente.

Repeatable/Repetitivo: Os testes devem ser executados em qualquer ambiente, eles devem ser executados no trampo, repetidos em casa no computador pessoal, não importa. Eu tenho que poder executar eles em qualquer lugar de maneira repetitiva o quanto eu quiser.

Self-Validating/Auto validação: Aqui estamos falando explicitamente de uma saiba booleana, então o meu teste tem que ser capaz de se validar sem que eu precise fazer comparativos entre arquivos/registros e blabla. Eu tenho que validar meu teste com uma saída booleana e já era (***os asserts da vida já nos garantem isso).***

Timely/Pontualidade: Timing, seguindo a prática do TDD o teste deve ser criado no momento certo, que é antes de escrever o código de implementação. Se você perder esse timming então ocorre a tendência do teste "ficar pra depois".

Conclusão

Testes são tão importantes quanto a própria aplicação, talvez até mais. São eles que nos permitem abusar de refatoração para limpar um código.