1

Por que devo me importar com o desenvolvimento orientado a testes?

Este livro foi escrito por desenvolvedores para desenvolvedores. Como tal, a maior parte do aprendizado será por meio de código. Cada capítulo apresentará uma ou mais práticas **de desenvolvimento orientado a testes (TDD)** e tentaremos dominá-las resolvendo **katas.** No karatê, um kata é um exercício em que você repete uma forma muitas vezes, fazendo pequenas melhorias em cada uma. Seguindo a mesma filosofia, faremos pequenas, mas significativas melhorias de um capítulo para o outro. Você aprenderá a projetar e codificar melhor, reduzir o **tempo de lançamento no mercado (TTM),** produzir documentação sempre atualizada, obter alta cobertura de código por meio de testes de qualidade e escrever código limpo que funcione.

Toda jornada tem um começo e essa não é exceção. Nosso destino é um desenvolvedor Java com faixa preta em TDD.

Para saber para onde vamos, teremos que discutir e encontrar respostas para algumas perguntas que definirão nossa viagem. O que é TDD? É uma técnica de teste ou algo mais? Quais são os benefícios da aplicação do TDD?

O objetivo deste capítulo é obter uma visão geral do TDD, entender o que é e compreender os benefícios que ele proporciona para seus praticantes.

Os seguintes tópicos serão abordados neste capítulo:

- Entendendo o TDD O
- que é TDD?
- Teste

- Zombando
- Documentação executável
- Sem depuração

Por que TDD?

Você pode estar trabalhando em um ambiente ágil ou em cascata. Talvez você tenha procedimentos bem definidos que foram testados ao longo de anos de trabalho duro, ou talvez você tenha acabado de iniciar sua própria empresa. Não importa qual era a situação, você provavelmente enfrentou pelo menos uma, se não mais, das seguintes dores, problemas ou causas para uma entrega malsucedida:

- Parte de sua equipe é mantida fora do circuito durante a criação de requisitos, especificações ou histórias de usuários
- A maioria, se não todos, dos seus testes são manuais ou você não tem testes
- Mesmo que você tenha testes automatizados, eles não detectam problemas reais
- Testes automatizados são escritos e executados quando é tarde demais para fornecer algum valor real ao projeto
- Há sempre algo mais urgente do que dedicar tempo para testar
- As equipes são divididas entre os departamentos de teste, desenvolvimento e análise funcional, e muitas vezes estão fora de sincronia
- Há uma incapacidade de refatorar o código por causa do medo de que algo seja quebrado
- O custo de manutenção é muito alto
- O tempo de colocação no mercado é muito grande
- Os clientes não sentem que o que foi entregue é o que eles pediram
- A documentação nunca está atualizada
- Você tem medo de implantar na produção porque o resultado é desconhecido
- Muitas vezes, você não consegue implantar na produção porque os testes de regressão demoram muito para serem executados
- A equipe está gastando muito tempo tentando descobrir o que algum método ou classe faz

O TDD não resolve magicamente todos esses problemas. Em vez disso, coloca-nos no caminho da solução. Não existe bala de prata, mas se existe uma prática de desenvolvimento que pode fazer a diferença em tantos níveis, essa prática é o TDD.

O TDD acelera o tempo de lançamento no mercado, facilita a refatoração, ajuda a criar um design melhor e promove um acoplamento mais flexível.

Além dos benefícios diretos, o TDD é um pré-requisito para muitas outras práticas (entrega contínua é uma delas). Melhor design, código bem escrito, TTM mais rápido, documentação atualizada e cobertura de teste sólida são alguns dos resultados que você obterá aplicando o TDD.

Não é fácil dominar o TDD. Mesmo depois de aprender toda a teoria e passar pelas melhores práticas e antipadrões, a jornada está apenas começando. TDD requer tempo e muita prática. É uma longa viagem que não termina com este livro. Na verdade, nunca termina verdadeiramente. Há sempre novas maneiras de se tornar mais proficiente e mais rápido. No entanto, embora o custo seja alto, os benefícios são ainda maiores. As pessoas que passaram bastante tempo com TDD afirmam que não há outra maneira de desenvolver um software. Nós somos um deles e temos certeza que você também será.

Acreditamos firmemente que a melhor maneira de aprender alguma técnica de codificação é codificando. Você não poderá terminar este livro lendo-o no metrô a caminho do trabalho. Não é um livro que se pode ler na cama. Você vai ter que sujar as mãos e codificar.

Neste capítulo, veremos o básico; a partir do próximo, você aprenderá lendo, escrevendo e executando código. Gostaríamos de dizer que quando você terminar de ler este livro, você será um programador de TDD experiente, mas isso não é verdade. Ao final deste livro, você estará confortável com o TDD e terá uma base sólida tanto na teoria quanto na prática.
O resto depende de você e da experiência que você estará construindo ao aplicá-la em seu trabalho diário.

Entendendo o TDD

Neste momento, você provavelmente está dizendo para si mesmo, *OK, eu entendo que o TDD me trará alguns benefícios, mas o que exatamente é o TDD?* TDD é um procedimento simples de escrever testes antes da implementação real. É uma inversão de uma abordagem tradicional em que o teste é realizado depois que o código é escrito.

Red-Green-Refactor

TDD é um processo que se baseia na repetição de um ciclo de desenvolvimento muito curto. Ele é baseado no conceito test-first de **Extreme Programming (XP)** que incentiva o design simples com um alto nível de confiança. O procedimento que conduz este ciclo é chamado **Red-Green-Refactor.**

O procedimento em si é simples e consiste em algumas etapas que são repetidas várias vezes:

1. Escreva um teste
2. Execute todos os testes
3. Escreva o código de implementação
4. Execute todos os testes
5. Refatorar
6. Execute todos os testes

Como um teste é escrito antes da implementação real, supõe-se que ele falhe. Se isso não acontecer, o teste está errado. Descreve algo que já existe ou foi escrito incorretamente.
Estar no estado verde ao escrever testes é um sinal de um falso positivo. Testes como esses devem ser removidos ou refatorados.

Ao escrever testes, estamos no estado vermelho. Quando a implementação de um teste for finalizada, todos os testes deverão passar e então estaremos no estado verde.

Se o último teste falhou, a implementação está errada e deve ser corrigida. Ou o teste que acabamos de terminar está incorreto ou a implementação desse teste não atendeu à especificação que havíamos definido. Se algum teste falhar, exceto o último, quebramos algo e as alterações devem ser revertidas.

Quando isso acontece, a reação natural é gastar o tempo necessário para corrigir o código para que todos os testes sejam aprovados. No entanto, isso está errado. Se uma correção não for feita em questão de minutos, o melhor a fazer é reverter as alterações. Afinal, tudo funcionou há pouco tempo. Uma implementação que quebrou alguma coisa está obviamente errada, então por que não voltar para onde começamos e pensar novamente sobre a maneira correta de implementar o teste? Dessa forma, perdemos minutos em uma implementação errada em vez de perder muito mais tempo para corrigir algo que não foi feito corretamente em primeiro lugar. A cobertura de teste existente (excluindo a implementação do último teste) deve ser sagrada. Mudamos o código existente por meio de refatoração intencional, não como uma forma de corrigir o código escrito recentemente.

Não torne a implementação do último teste final, mas forneça apenas código suficiente para este teste passar.

Escreva o código da maneira que quiser, mas faça isso rápido. Quando tudo estiver verde, temos confiança de que existe uma rede de segurança na forma de testes. A partir deste momento, podemos proceder à refatoração do código. Isso significa que estamos tornando o código melhor e mais otimizado sem introduzir novos recursos. Enquanto a refatoração está em vigor, todos os testes devem passar o tempo todo.

Se, durante a refatoração, um dos testes falhou, a refatoração quebrou uma funcionalidade existente e, como antes, as alterações devem ser revertidas. Além disso, nesta fase não estamos alterando nenhum recurso, mas também não estamos introduzindo novos testes. Tudo o que estamos fazendo é melhorar o código enquanto executamos continuamente todos os testes para garantir que nada seja quebrado. Ao mesmo tempo, estamos provando a correção do código e reduzindo os custos de manutenção futuros.

Uma vez que a refatoração é concluída, o processo é repetido. É um loop infinito de um ciclo muito curto.

Velocidade é a chave

Imagine um jogo de pingue-pongue (ou tênis de mesa). O jogo é muito rápido; às vezes é difícil até mesmo seguir a bola quando os profissionais jogam o jogo. TDD é muito semelhante. Os veteranos do TDD tendem a não passar mais de um minuto em cada lado da mesa (teste e implementação). Escreva um teste curto e execute todos os testes (ping), escreva a implementação e execute todos os testes (pong), escreva outro teste (ping), escreva a implementação desse teste (pong), refatore e confirme se todos os testes estão passando (pontuação), e depois repita — ping, pong, ping, pong, ping, pong, pontuação, saque novamente. Não tente fazer o código perfeito. Em vez disso, tente manter a bola rolando até achar que é a hora certa de marcar (refatorar).

O tempo entre a mudança dos testes para a implementação (e vice-versa) deve ser medido em minutos (se não segundos).

Não é sobre testar

T em **TDD** é muitas vezes mal interpretado. TDD é a forma como abordamos o design. É a maneira de nos forçar a pensar sobre a implementação e o que o código precisa fazer antes de escrevê-lo.
É a maneira de focar nos requisitos e na implementação de apenas uma coisa de cada vez – organizar seus pensamentos e estruturar melhor o código. Isso não significa que os testes resultantes do TDD sejam inúteis – eles estão longe disso. Eles são muito úteis e nos permitem desenvolver com grande velocidade sem medo de que algo seja quebrado. Isso é especialmente verdadeiro quando ocorre a refatoração. Ser capaz de reorganizar o código e ter a confiança de que nenhuma funcionalidade está quebrada é um grande impulso para sua qualidade.

O principal objetivo do TDD é o design de código testável com testes como um produto secundário muito útil.

Teste

Embora o objetivo principal do TDD seja a abordagem ao design de código, os testes ainda são um aspecto muito importante do TDD e devemos ter uma compreensão clara de dois grandes grupos de técnicas, como segue:

- Teste de caixa preta
- Teste de caixa branca

Teste de caixa preta

O teste de caixa preta (também conhecido como **teste funcional)** trata o software em teste como uma caixa preta sem conhecer seus componentes internos. Os testes usam interfaces de software e tentam garantir que funcionem conforme o esperado. Contanto que a funcionalidade das interfaces permaneça inalterada, os testes devem ser aprovados mesmo que os internos sejam alterados. O testador está ciente do que o programa deve fazer, mas não tem conhecimento de como ele o faz. O teste de caixa preta é o tipo de teste mais comumente usado em organizações tradicionais que têm testadores como um departamento separado, especialmente quando não são proficientes em codificação e têm dificuldade em entendê-lo.
Essa técnica fornece uma perspectiva externa sobre o software em teste.

Algumas das vantagens do teste de caixa preta são as seguintes:

- É eficiente para grandes segmentos de código
- Acesso ao código, compreensão do código e capacidade de codificar não são necessários
- Oferece separação entre as perspectivas de usuários e desenvolvedores

Algumas das desvantagens do teste de caixa preta são as seguintes:

- Ele fornece cobertura limitada, pois apenas uma fração dos cenários de teste é executada
- Pode resultar em testes ineficientes devido à falta de conhecimento do testador sobre os componentes internos do software
- Pode levar a uma cobertura cega, pois os testadores têm conhecimento limitado sobre o aplicativo

Se os testes estão conduzindo o desenvolvimento, eles geralmente são feitos na forma de critérios de aceitação que são usados posteriormente como uma definição do que deve ser desenvolvido.

O teste automatizado de caixa preta depende de alguma forma de automação, como **desenvolvimento orientado a comportamento (BDD).**

Teste de caixa branca

O teste de caixa branca (também conhecido como **teste** de caixa transparente , **teste de caixa de vidro, teste de caixa transparente** e **teste estrutural)** examina o software que está sendo testado e usa esse conhecimento como parte do processo de teste. Se, por exemplo, uma exceção deve ser lançada sob certas condições, um teste pode querer reproduzir essas condições. O teste de caixa branca requer conhecimento interno do sistema e habilidades de programação. Ele fornece uma perspectiva interna sobre o software em teste.

Algumas das vantagens do teste de caixa branca são as seguintes:

- É eficiente em encontrar erros e problemas
- O conhecimento necessário dos componentes internos do software em teste é benéfico para testes completos
- Permite encontrar erros ocultos

- Incentiva a introspecção do programador
- Ajuda a otimizar o código
- Devido ao conhecimento interno necessário do software, a cobertura máxima é obtida

Algumas das desvantagens do teste de caixa branca são as seguintes:

- Pode não encontrar recursos não implementados ou ausentes
- Requer conhecimento de alto nível dos componentes internos do software em teste
- Requer acesso ao código
- Os testes geralmente são fortemente acoplados aos detalhes de implementação do código de produção, causando falhas de teste indesejadas quando o código é refatorado

O teste de caixa branca é quase sempre automatizado e, na maioria dos casos, assume a forma de testes unitários.

Quando o teste de caixa branca é feito antes da implementação, ele assume a forma de TDD.

A diferença entre verificação de qualidade e garantia de qualidade

A abordagem de teste também pode ser distinguida observando os objetivos que eles estão tentando alcançar. Esses objetivos geralmente são divididos entre **verificação de qualidade (QC)** e **garantia de qualidade (QA).** Enquanto o QC está focado na identificação de defeitos, o QA tenta evitá-los. O QC é orientado para o produto e pretende garantir que os resultados sejam os esperados. Por outro lado, o QA está mais focado em processos que garantem que a qualidade seja incorporada. Ele tenta garantir que as coisas corretas sejam feitas da maneira correta.

Embora o QC tivesse um papel mais importante no passado, com o surgimento do TDD, **do desenvolvimento orientado a testes de aceitação (ATDD)** e, posteriormente, do BDD, o foco mudou para o QA.

Testes melhores

Não importa se está usando caixa preta, caixa branca ou ambos os tipos de teste, a ordem em que eles são escritos é muito importante.

Os requisitos (especificações e histórias de usuários) são escritos antes do código que os implementa. Eles vêm primeiro para definir o código, e não o contrário. O mesmo pode ser dito para os testes. Se eles são escritos depois que o código é feito, de certa forma, esse código (e as funcionalidades que ele implementa) está definindo testes. Testes definidos por um aplicativo já existente são tendenciosos. Eles tendem a confirmar o que o código faz e não a testar se as expectativas de um cliente são atendidas ou se o código está se comportando conforme o esperado. Com o teste manual, esse é menos o caso, pois geralmente é feito por um departamento de controle de qualidade em silos (mesmo que seja chamado de controle de qualidade). Eles tendem a trabalhar na definição de teste isoladamente dos desenvolvedores. Isso por si só leva a problemas maiores causados por uma comunicação inevitavelmente ruim e a **síndrome da polícia,** onde os testadores não estão tentando ajudar a equipe a escrever aplicativos com qualidade incorporada, mas para encontrar falhas no final do processo. Quanto mais cedo encontrarmos problemas, mais barato será corrigi-los.

Testes escritos no estilo TDD (incluindo seus sabores como ATDD e BDD) são uma tentativa de desenvolver aplicativos com qualidade incorporada desde o início. É uma tentativa de evitar problemas em primeiro lugar.

Zombando

Para que os testes sejam executados rapidamente e forneçam feedback constante, o código precisa ser organizado de forma que os métodos, funções e classes possam ser facilmente substituídos por mocks e stubs. Uma palavra comum para esse tipo de substituição do código real é **test double.**
A velocidade de execução pode ser severamente afetada com dependências externas; por exemplo, nosso código pode precisar se comunicar com o banco de dados. Ao zombar de dependências externas, podemos aumentar essa velocidade drasticamente. A execução de todo o conjunto de testes de unidade deve ser medida em minutos, se não em segundos. Projetar o código de uma maneira que possa ser facilmente zombado e stub nos força a estruturar melhor esse código aplicando uma separação de interesses.

Mais importante que a velocidade é o benefício da remoção de fatores externos. A configuração de bancos de dados, servidores web, APIs externas e outras dependências que nosso código pode precisar é demorado e não confiável. Em muitos casos, essas dependências podem nem estar disponíveis. Por exemplo, podemos precisar criar um código que se comunique com um banco de dados e fazer com que outra pessoa crie um esquema. Sem mocks, precisaríamos esperar até que o esquema seja definido.

Com ou sem mocks, o código deve ser escrito de forma que possamos substituir facilmente uma dependência por outra.

Documentação executável

Outro aspecto muito útil do TDD (e testes bem estruturados em geral) é a documentação.
Na maioria dos casos, é muito mais fácil descobrir o que o código faz observando os testes do que a própria implementação. Qual é o objetivo de alguns métodos? Veja os testes associados a ele. Qual é a funcionalidade desejada de alguma parte da interface do usuário do aplicativo? Veja os testes associados a ele. A documentação escrita na forma de testes é um dos pilares do TDD e merece maiores explicações.

O principal problema com a documentação de software (tradicional) é que ela não está atualizada na maioria das vezes. Assim que alguma parte do código muda, a documentação deixa de refletir a situação real. Esta declaração se aplica a quase qualquer tipo de documentação, com requisitos e casos de teste sendo os mais afetados.

A necessidade de documentar o código é muitas vezes um sinal de que o próprio código não está bem escrito. Além disso, não importa o quanto tentemos, a documentação inevitavelmente fica desatualizada.

Os desenvolvedores não devem confiar na documentação do sistema porque quase nunca está atualizada.
Além disso, nenhuma documentação pode fornecer uma descrição do código tão detalhada e atualizada quanto o próprio código.

Usar código como documentação não exclui outros tipos de documentos. A chave é evitar a duplicação. Se os detalhes do sistema puderem ser obtidos lendo o código, outros tipos de documentação podem fornecer orientações rápidas e uma visão geral de alto nível. A documentação não codificada deve responder a perguntas como qual é o propósito geral do sistema e quais tecnologias são usadas pelo sistema. Em muitos casos, um simples README é suficiente para fornecer o início rápido que os desenvolvedores precisam. Seções como descrição do projeto, configuração do ambiente, instalação e instruções de compilação e empacotamento são muito úteis para os recém-chegados. A partir daí, o código é a bíblia.

O código de implementação fornece todos os detalhes necessários, enquanto o código de teste atua como a descrição da intenção por trás do código de produção.

Testes são documentações executáveis com TDD sendo a forma mais comum de criá-los e mantê-los.

Supondo que alguma forma de **integração contínua (CI)** esteja em uso, se alguma parte da documentação do teste estiver incorreta, ela falhará e será corrigida logo em seguida. A CI resolve o problema de documentação de teste incorreta, mas não garante que todas as funcionalidades sejam documentadas. Por esse motivo (entre muitos outros), a documentação de teste deve ser criada no estilo TDD. Se todas as funcionalidades forem definidas como testes antes que o código de implementação seja escrito e a execução de todos os testes for bem-sucedida, os testes atuarão como uma fonte de informações completa e atualizada que pode ser usada pelos desenvolvedores.

O que devemos fazer com o resto da equipe? Testadores, clientes, gerentes e outros não codificadores podem não conseguir obter as informações necessárias do código de produção e teste.

Como vimos anteriormente, dois dos tipos mais comuns de teste são o teste de caixa preta e o teste de caixa branca. Essa divisão é importante, pois também divide os testadores entre aqueles que sabem escrever ou pelo menos ler código (teste de caixa branca) e aqueles que não sabem (teste de caixa preta). Em alguns casos, os testadores podem fazer os dois tipos. No entanto, na maioria das vezes, eles não sabem como codificar, então a documentação que pode ser usada pelos desenvolvedores não pode ser usada por eles. Se a documentação precisar ser desacoplada do código, os testes de unidade não são uma boa combinação. Essa é uma das razões pelas quais o BDD surgiu.

O BDD pode fornecer a documentação necessária para não codificadores, mantendo as vantagens do TDD e da automação.

Os clientes precisam ser capazes de definir novas funcionalidades do sistema, bem como obter informações sobre todos os aspectos importantes do sistema atual. Essa documentação não deve ser muito técnica (código não é uma opção), mas ainda deve estar sempre atualizada.
As narrativas e cenários de BDD são uma das melhores formas de fornecer esse tipo de documentação. A capacidade de atuar como critério de aceitação (escrito antes do código), ser executado com frequência (de preferência em cada commit) e ser escrito em uma linguagem natural torna as histórias de BDD não apenas sempre atualizadas, mas também utilizáveis por aqueles que não deseja inspecionar o código.

A documentação é parte integrante do software. Como acontece com qualquer outra parte do código, ele precisa ser testado com frequência para que tenhamos certeza de que é preciso e atualizado.

A única maneira econômica de ter informações precisas e atualizadas é ter documentação executável que possa ser integrada ao seu sistema de CI.

O TDD como metodologia é uma boa maneira de avançar nessa direção. Em um nível baixo, os testes de unidade são os mais adequados. Por outro lado, o BDD fornece uma boa maneira de trabalhar em um nível funcional, mantendo a compreensão que é alcançada usando linguagem natural.

Sem depuração

Nós (autores deste livro) quase nunca depuramos os aplicativos em que estamos trabalhando!

Esta afirmação pode soar pomposa, mas é verdade. Quase nunca depuramos porque raramente há uma razão para depurar um aplicativo. Quando os testes são escritos antes do código e a cobertura do código é alta, podemos ter alta confiança de que o aplicativo funciona conforme o esperado. Isso não significa que aplicativos escritos usando TDD não tenham bugs — eles têm. Todos os aplicativos fazem. No entanto, quando isso acontece, é fácil isolá-los simplesmente procurando o código que não é coberto pelos testes.

Os próprios testes podem não incluir alguns casos. Nessas situações, a ação é escrever testes adicionais.

Com alta cobertura de código, encontrar a causa de algum bug é muito mais rápido por meio de testes do que gastar tempo depurando linha por linha até que o culpado seja encontrado.

Resumo

Neste capítulo, você obteve o entendimento geral da prática de TDD e insights sobre o que é e o que não é TDD. Você aprendeu que é uma maneira de projetar código por meio de um ciclo curto e repetível chamado Red-Green-Refactor. A falha é um estado esperado que deve não apenas ser adotado, mas aplicado em todo o processo de TDD. Este ciclo é tão curto que passamos de uma fase para outra com grande velocidade.

Embora o design de código seja o objetivo principal, os testes criados ao longo do processo de TDD são um ativo valioso que deve ser utilizado e impactar severamente nossa visão das práticas de teste tradicionais. Passamos pelas práticas mais comuns, como testes de caixa branca e caixa preta, tentamos colocá-las na perspectiva do TDD e mostramos os benefícios que elas podem trazer umas para as outras.

Você descobriu que os mocks são ferramentas muito importantes que geralmente são obrigatórias ao escrever testes. Por fim, discutimos como os testes podem e devem ser utilizados como documentação executável e como o TDD pode tornar a depuração muito menos necessária.

Agora que temos conhecimento teórico, é hora de configurar o ambiente de desenvolvimento e obter uma visão geral e comparação de diferentes estruturas e ferramentas de teste.