O TDD é o desenvolvimento orientado/guiado por testes, nele a ideia é bem simples: escreva seus testes antes mesmo de escrever o código de produção. Ao escrever os testes antes, o desenvolvedor garante que boa parte (ou talvez todo) do seu sistema tem um teste que garante o seu funcionamento. Além disso, muitos desenvolvedores também afirmam que os testes os guiam no projeto de classes do sistema.

A ideia do TDD é um dos pilares (práticas) do XP (Extreme Programming), mas também é utilizado em outras metodologias, além de poder ser utilizada livremente. Onde os testes são utilizados para facilitar no entendimento do projeto, dando ideia do que se deseja em relação ao código.

Basicamente o TDD se baseia em pequenos ciclos de repetições, onde para cada funcionalidade do sistema um teste é criado antes. Este novo teste criado inicialmente falha, já que ainda não temos a implementação da funcionalidade em questão e, em seguida, implementamos a funcionalidade para fazer o teste passar.

A criação de teste unitários ou de componentes é parte crucial para o TDD.

Notemos aqui que o teste visa auxiliar a codificação, reduzindo consideravelmente os problemas na fase de desenvolvimento. No TDD é indicado que o projeto de teste unitário ocorra antes da fase de codificação/implementação.

O teste antes da codificação ou test-first, define implicitamente tanto uma interface como uma especificação do comportamento para a funcionalidade que está sendo desenvolvida.

Note que ao criar o teste antes de implementar a unidade, são reduzidos problemas como mal entendimento de requisitos ou interfaces, pois como criar um teste se eu não sei o que devo testar.

Neste caso o desenvolvedor, para implementar os testes iniciais, deve compreender com detalhes a especificação do sistema e as regras de negócio, só assim, será possível escrever testes para o sistema. Imagine o caso de querer testar um pneu criado para o carro, se não entendi que o pneu é redondo, por exemplo, criarei um teste para um pneu quadrado, não podendo ser realizado o teste. Desta forma, é de extrema importância, para o desenvolvedor, o entendimento dos requisitos do cliente. Além disso, não adianta criar testes que não validem o código como um todo para reduzir o tempo, é necessário criar testes para o conjunto completo de unidades, só assim o TDD vai funcionar como deve, devendo fornecer uma cobertura completa aos testes.

Além disso, os testes devem seguir o modelo F.I.R.S.T.:

F (Fast) - Rápidos: devem ser rápidos, pois testam apenas uma unidade;

I (Isolated) - Testes unitários são isolados, testando individualmente as unidades e não sua integração;

R (Repeateble) - Repetição nos testes, com resultados de comportamento constante;

S (Self-verifying) - A auto verificação deve verificar se passou ou se deu como falha o teste;

T (Timely) - O teste deve ser oportuno, sendo um teste por unidade.

Temos de lembrar que qualidade não é uma etapa apenas, uma fase do desenvolvimento, é parte de todo o ciclo de vida de desenvolvimento do software. A qualidade aplicada tardiamente não gera um produto com qualidade, a maioria das organizações aplicam em seus processos de desenvolvimento testes tardios, o que não garante a qualidade do produto, e, desta forma, o custo do produto aumenta consideravelmente.

Segundo Presmann, “a motivação que está por trás do teste de programas é a confirmação da qualidade de software com métodos que podem ser economicamente e efetivamente aplicados a todos os sistemas, de grande e pequena escala”. Desta forma, utilizar uma metodologia que permita testes no processo de desenvolvimento, auxilia na garantia de qualidade.

Se pararmos para analisar, o grande responsável pelo aumento da qualidade interna e externa não é o TDD, mas sim o teste automatizado, produzido a partir do uso da prática. A pergunta comum é justamente então: Qual a diferença entre fazer TDD e escrever o teste depois?

Quanto mais cedo o desenvolvedor receber feedback, melhor. Quando se tem muito código já escrito, mudanças podem ser trabalhosas e custar caro. Ao contrário, quanto menos código escrito, menor será o custo de mudança. E é justamente isso que acontece com praticantes de TDD: eles recebem feedback no momento em que mudar ainda é barato. Veja pela ilustração da Figura 1 que os testes são distribuídos no TDD.

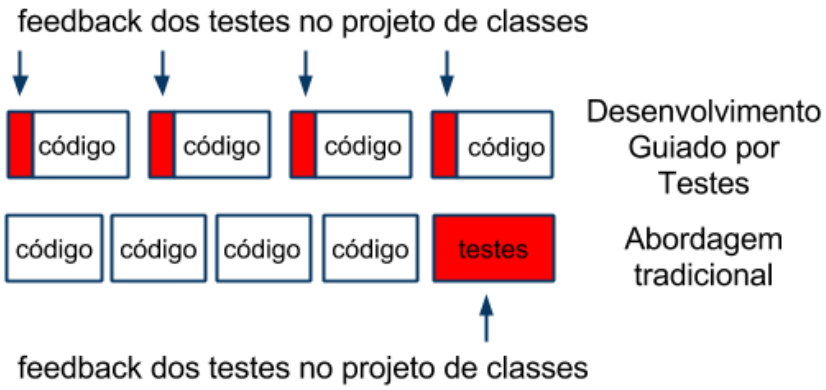

Figura 1 – Diferença entre TDD e abordagem tradicional.

A mecânica da prática é simples (Figura 2): escreva um teste que falha, faça-o passar da maneira mais simples possível e, por fim, refatore o código. Esse ciclo é conhecido como Ciclo Vermelho-Verde-Refatora.

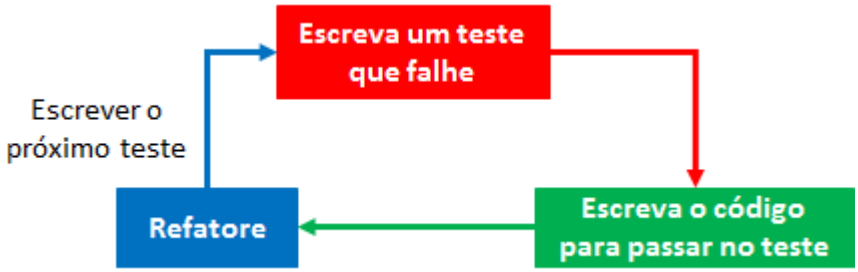

Figura 2 – Ciclo do TDD.

O Processo do TDD (exemplo com um método simples)

1 - Criar uma implementação vazia do método (“casca”);
2 - Criar um teste para mapear um ou mais requisitos (é recomendável começar simples);
3 - Rodar o novo teste, para confirmar que este mapeia o(s) requisito(s) (vermelho);
4 - Implementar a solução mais simples que faça todos os testes rodarem;
5 - Rodar os testes para verificar a implementação (verde)
6 - Simplificar a implementação, se possível (refatorar)
7 - Repetir os passos 2 a 5 até atender a todos os requisitos.

Refatoração:

- Refatoração é o processo de alterar e otimizar o código de maneira que seu comportamento externo não seja alterado;
- Em código que está funcionando não se mexe? O TDD permite que este medo deixe de ter fundamento, pois ele atua como uma rede de segurança, capturando qualquer bug que seja inserido durante a refatoração.

Sempre que um desenvolvedor pega uma funcionalidade para fazer, ele a quebra (muitas vezes mentalmente) em pequenas tarefas. Tarefas essas que exigem a escrita de código. Classes são criadas, outras são modificadas, mas sempre com algum propósito.

Ao praticar TDD, o desenvolvedor antes de começar a fazer essas modificações explicita esses objetivos. Só que ele faz isso por meio de testes automatizados. O teste em código nada mais é do que um trecho de código que deixa claro o que determinado trecho de código deve fazer.

Ao formalizar esse objetivo na forma de um teste automatizado, esse teste falha, claro; afinal, a funcionalidade ainda não foi implementada. O desenvolvedor então trabalha para fazer esse teste passar. Como? Implementando a funcionalidade.

Pontos negativos do TDD:

- Desenvolvimento é mais lento;
- Mudança de hábito (inicialmente);
- Quem cria os testes é o desenvolvedor, portanto ambos os testes e o código de produção podem ter os mesmos erros conceituais;
- A barra verde (do JUnit) pode levar a uma falsa sensação de segurança e fazer com que a equipe relaxe nos testes de integração e funcionais.

Pontos positivos:

- O desenvolvedor pode resolver o problema aos poucos, aspecto a aspecto;
- Testes facilitam o entendimento/documentam dos requisitos;
- Bugs são percebidos mais cedo;
- É mais fácil identificar a causa;
- A correção é menos custosa;
- O aprendizado é melhor;
- Garante uma boa base de testes;
- A arquitetura tende a apresentar baixo nível de acoplamento;
- O código é, naturalmente, facilmente testável;
- Refatorações são menos arriscadas.