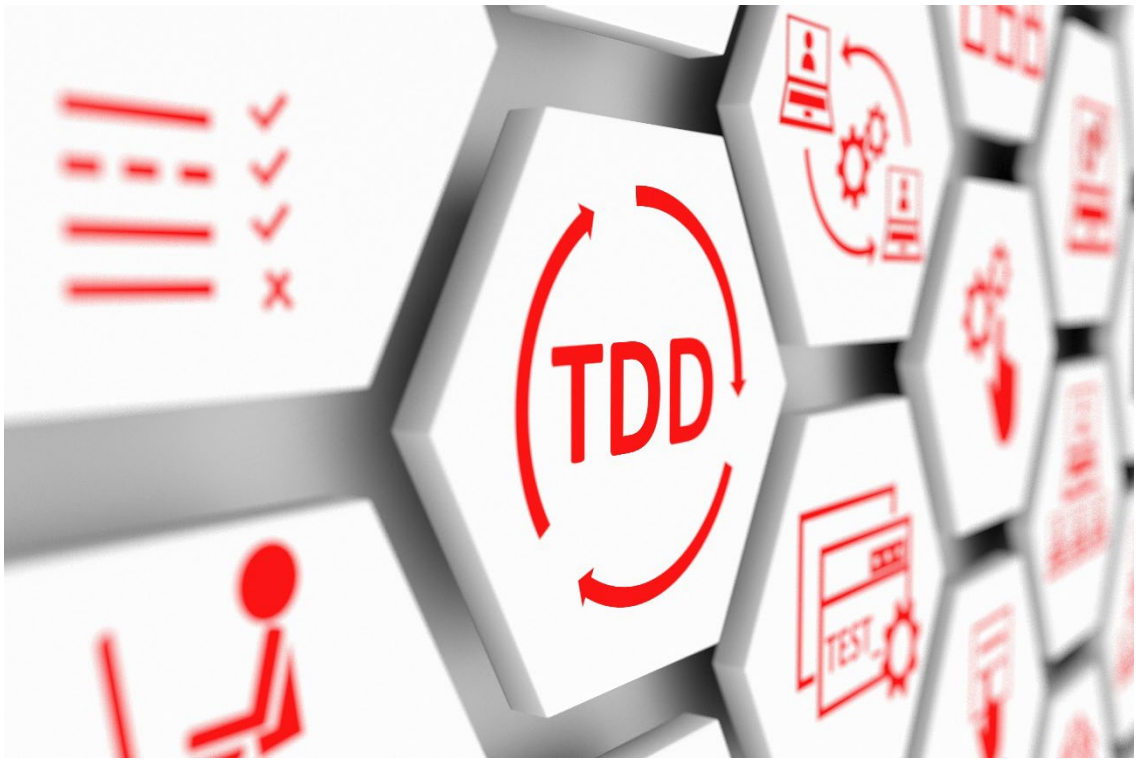
METODOLOGIAS ÁGEIS

AULA 5

Profª Mariana Lucia Kalluf Dakkache Leal

CONVERSA INICIAL

Neste capítulo, exploraremos a aplicabilidade do TDD (*Test Driven Development* ou desenvolvimento orientado a testes) e como pode ser usado para desenvolver *softwares* de maneira eficiente e confiável. Abordaremos os principais conceitos e técnicas relacionados ao TDD, incluindo a importância de criar testes antes da implementação do código, os ciclos de refinamento contínuo, a criação de testes e a abordagem *red green refactory*. Além disso, discutiremos o conceito de complexidade incremental e seu papel no *design* incremental de *software*.



Crédito: Profit_Image/Shutterstock.

TEMA 1 – POR QUE CRIAR UM TESTE ANTES DA IMPLEMENTAÇÃO DO CÓDIGO

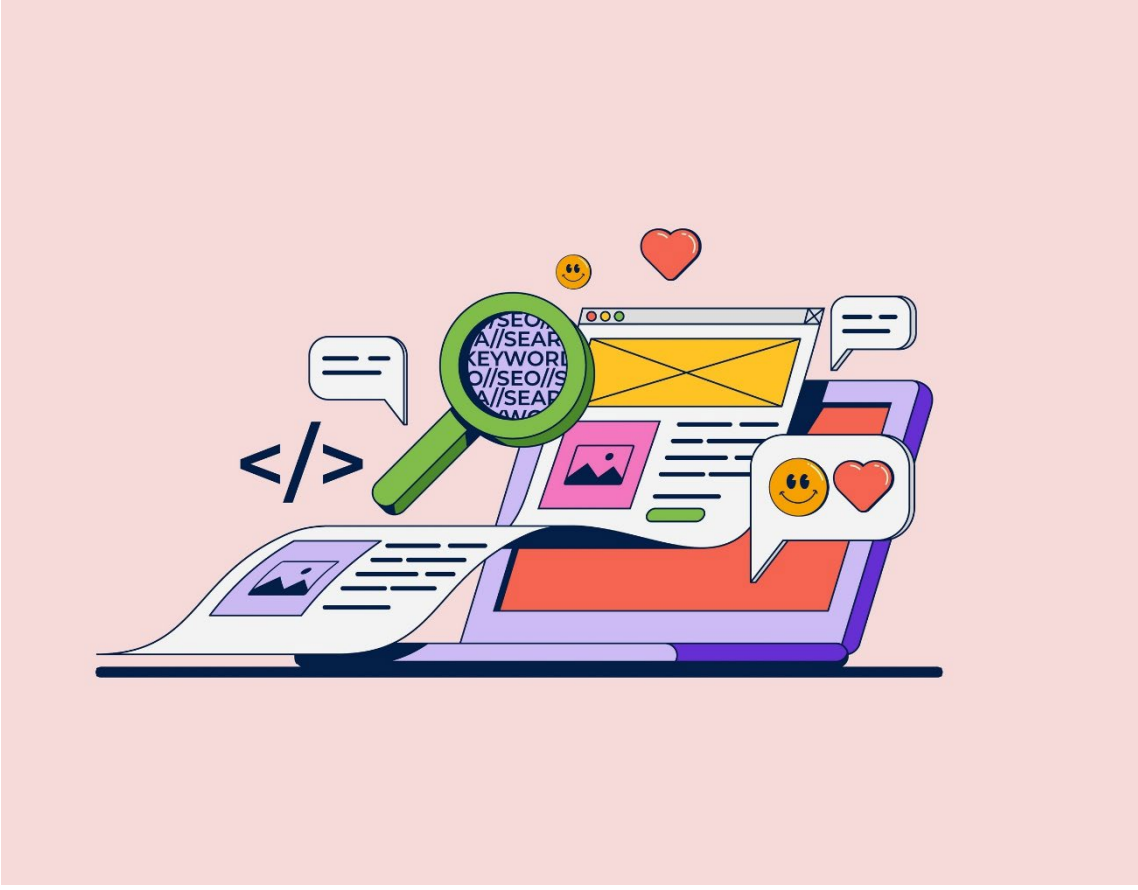
Em etapa anterior, vimos o conceito de TDD, suas características e seu funcionamento. Nesta etapa, aprofundaremos nosso conhecimento sobre essa prática.

De acordo com Massol e Husted (2003, p. 5), "*Test Driven Development* é uma prática de programação que instrui desenvolvedores a escrever código

novo apenas se um teste automatizado estiver falhado, e a eliminar duplicação. O TDD tem por objetivo escrever código claro que funcione".

A prática do TDD traz inúmeras vantagens para o processo de desenvolvimento, sendo aprimorar a qualidade do produto o benefício mais significativo. Muitas vezes, nos deparamos com novas versões do *software* que introduzem novas funcionalidades, mas acabam afetando o funcionamento das características existentes. Por meio da criação de testes automatizados, é possível obter maior segurança durante as mudanças realizadas.

Os testes automatizados podem ser executados repetidamente pelo desenvolvedor ao longo do dia, levando apenas alguns segundos para serem concluídos. Isso é extremamente ágil se comparado aos testes manuais, que seriam inviáveis de serem realizados com essa frequência. Se algum problema surgir durante a execução dos testes automatizados, o desenvolvedor é imediatamente notificado e pode corrigir o erro antes de disponibilizar a versão para o cliente. Evitar a entrega de um *software* com *bugs* é uma prática altamente benéfica para a satisfação do usuário final.

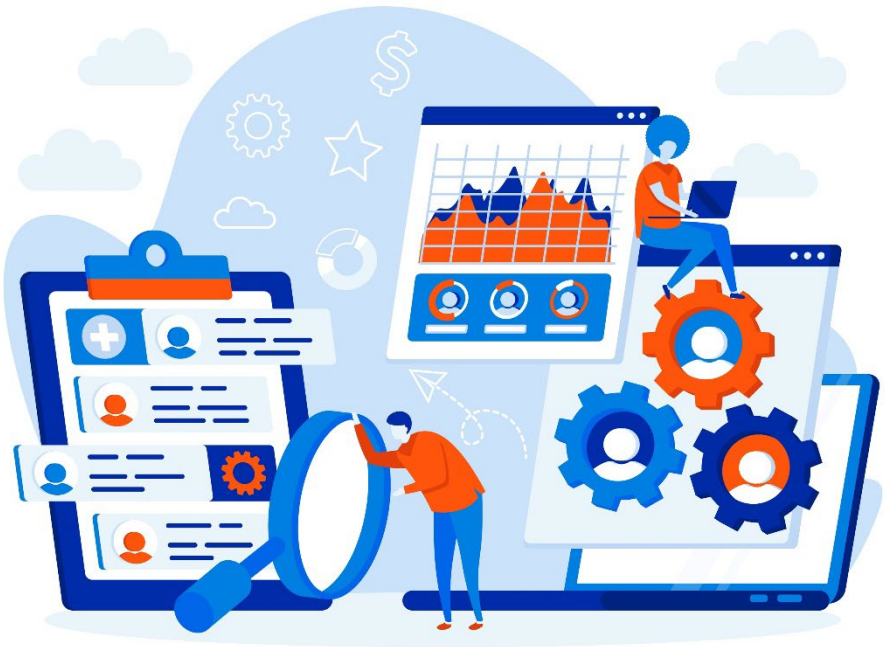


Crédito: Paul Craft/Shutterstock.

A leitura atenta do código dos testes pode fornecer informações valiosas sobre a qualidade do código em desenvolvimento. O TDD auxilia o desenvolvedor a escrever um código de produção com excelência. Ou seja, se a criação de um teste automatizado se torna uma tarefa complexa, é um indicativo de que o código de produção também pode estar complexo. Esse é um ponto valioso para o desenvolvedor.

Dessa forma, fica evidente que a adoção do TDD auxilia a equipe a garantir que os requisitos funcionem conforme o esperado, ao mesmo tempo em que auxilia o desenvolvedor na identificação de problemas de código durante as implementações.

A diferença fundamental entre a prática do TDD e escrever o teste depois está no momento que o desenvolvedor obtém *feedback* do teste. Enquanto no método tradicional de escrever o teste após a implementação do código, há um longo período sem *feedback*, o TDD proporciona um retorno constante durante o processo de desenvolvimento. O desenvolvedor que adota o TDD divide seu trabalho em pequenas etapas, escrevendo um teste e implementando uma parte da funcionalidade de cada vez, repetindo esse ciclo. A cada teste escrito, o desenvolvedor recebe um *feedback* imediato.



Crédito: Alexdndz/Shutterstock.

Receber *feedback* o mais cedo possível é vantajoso. Quando há uma grande quantidade de código já escrito, fazer mudanças pode ser trabalhoso e custoso. Por outro lado, quanto menos código foi escrito, menor será o custo das alterações necessárias. É exatamente isso que ocorre com os praticantes de TDD que recebem *feedback* quando as mudanças ainda são de baixo custo. Ao adotar o TDD, o desenvolvedor pode realizar ajustes e melhorias rapidamente, sem a preocupação de ter que refatorar grandes porções do código, o que resulta em um processo mais ágil e eficiente.

1.1 TDD *versus* abordagem tradicional de testes

O TDD (*Test Driven Development*) e a abordagem tradicional de testes são duas formas distintas de abordar a qualidade de *software*. Vamos compará-las em diferentes aspectos:

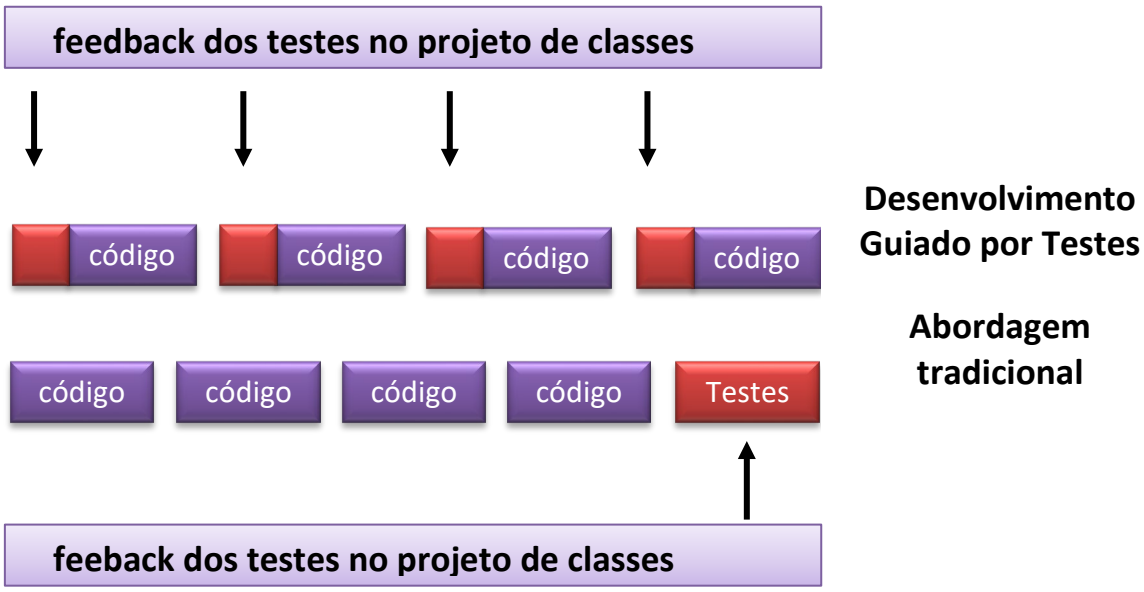
Sequência de desenvolvimento	
TDD	Testes são escritos antes da implementação do código. O desenvolvedor começa definindo um teste que descreve a funcionalidade desejada e, em seguida, implementa o código necessário para fazer o teste passar.
Abordagem tradicional	Código é implementado primeiro e, em seguida, os testes são escritos posteriormente para validar o funcionamento do código.

Foco no resultado	
TDD	Enfatiza a criação de testes que representam os requisitos e comportamentos esperados do *software*. O objetivo é alcançar um conjunto de testes automatizados abrangente que valide todas as funcionalidades desejadas.
Abordagem tradicional	Maior enfoque na correção de erros específicos ou para a validação de partes específicas do código.

Ciclo de *feedback*	
TDD	Desenvolvedor obtém *feedback* instantâneo sobre a funcionalidade implementada por meio dos testes automatizados. permitindo correções rápidas e identificação antecipada de problemas.
Abordagem tradicional	*Feedback* dos testes pode ser obtido posteriormente, uma vez que os testes são escritos após a implementação do código. Isso pode levar a um tempo de *feedback* mais longo e a possíveis atrasos na detecção e correção de erros (Figura 1).

Fonte: Leal, 2023.

Figura 1 – TDD *versus* Abordagem tradicional



Orientação ao usuário	
TDD	Incentiva a criação de testes que representam a perspectiva do usuário final. Ao definir os testes primeiro, os desenvolvedores são guiados pelas necessidades e expectativas dos usuários, resultando em um software mais alinhado com suas demandas.
Abordagem tradicional	Perspectiva do usuário pode ser abordada posteriormente, quando os testes são escritos. Isso pode resultar em uma menor ênfase na usabilidade e na experiência do usuário durante a implementação.

Colaboração e comunicação	
TDD	Promove a colaboração e a comunicação entre os membros da equipe de desenvolvimento. A definição conjunta dos testes ajuda a alinhar as expectativas e a garantir uma compreensão compartilhada dos requisitos.
Abordagem tradicional	Comunicação pode ser menos intensa durante o processo de desenvolvimento dos testes, que são escritos posteriormente à implementação do código.

Fonte: Leal, 2023.

Pesquisas confirmam que sistemas criados com o uso de TDD geram um código de melhor qualidade e com menos falhas do que sistemas que empregaram apenas testes convencionais (Jones, 2004).

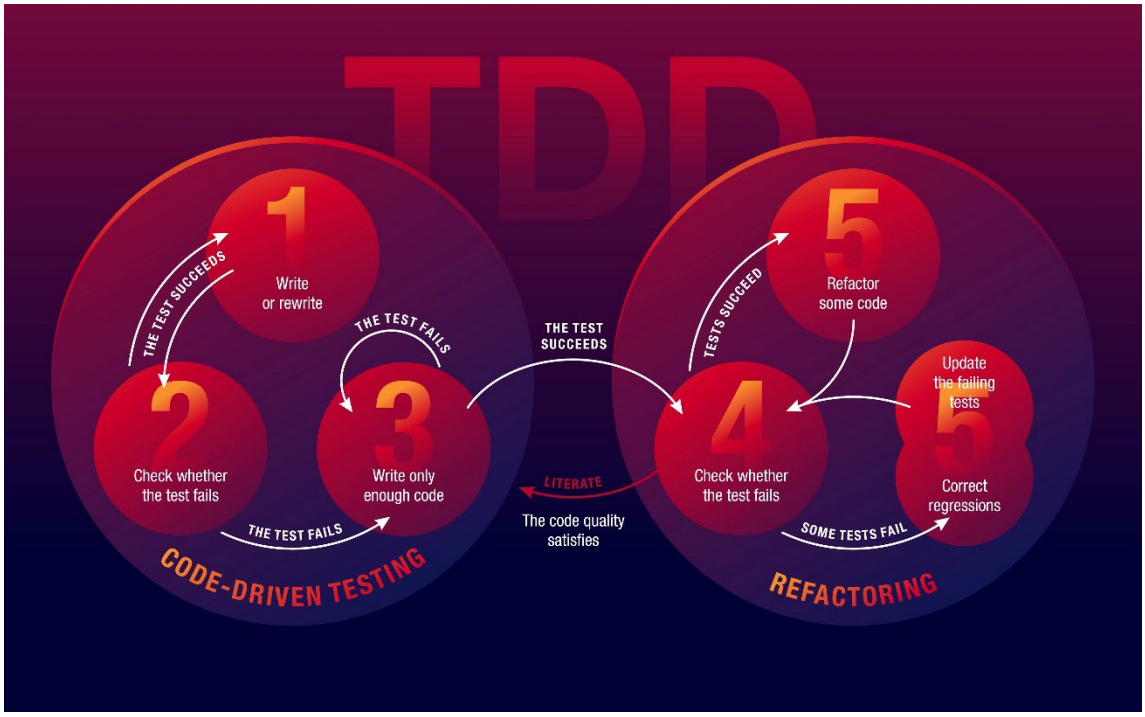
Criar um teste antes da implementação do código traz benefícios significativos, como garantir a qualidade do *software*, direcionar o foco para a perspectiva do usuário final, detectar problemas mais cedo e promover a

colaboração na equipe de desenvolvimento. Essa prática é uma estratégia poderosa para desenvolver *softwares* de alta qualidade, confiável e orientado às necessidades dos usuários.

TEMA 2 – CICLOS DE REFINAMENTO CONTÍNUO

Para Feitosa (2007), a metodologia TDD envolve ciclos curtos nos quais são elaborados novos conjuntos de testes que abrangem uma nova funcionalidade ou aprimoramento e somente em seguida é implementado o código necessário e suficiente para atender a esses testes. Após essa etapa, o *software* passa por uma reestruturação para incorporar as alterações de modo que os testes permaneçam bem-sucedidos

Os ciclos de refinamento contínuo do TDD (*Test Driven Development*) têm como objetivo principal melhorar a qualidade do código e do *software* de forma iterativa. São compostos por etapas sequenciais e repetitivas, onde cada ciclo consiste na criação de um teste automatizado antes da implementação do código correspondente.



Crédito: Dmitry Kovalchuk/Shutterstock.

A seguir, apresentaremos os propósitos e benefícios de cada etapa do ciclo de refinamento contínuo do TDD:

1. **Criação do teste**

- Nessa etapa, um teste automatizado é elaborado com base nos requisitos ou funcionalidades desejadas.
- O teste é desenvolvido de forma a validar uma pequena parte da funcionalidade em questão.
- O teste deve ser simples, focado e representar um caso específico que se deseja verificar.

2. **Execução do teste**

- Depois de criado, o teste é executado e espera-se que inicialmente falhe, pois o código correspondente ainda não foi implementado.
- A falha do teste é esperada e serve como uma medida de validação do seu propósito.

3. **Implementação do código**

- Nesta etapa, o código de produção é desenvolvido com o objetivo de fazer o teste previamente criado passar.
- O código é escrito de forma a cumprir os requisitos e assegurar o funcionamento esperado.

4. **Nova execução do teste**

- Depois da implementação do código, o teste previamente criado é executado novamente.
- Espera-se que, desta vez, o teste passe, validando assim a correta implementação do código.

5. **Refatoração**

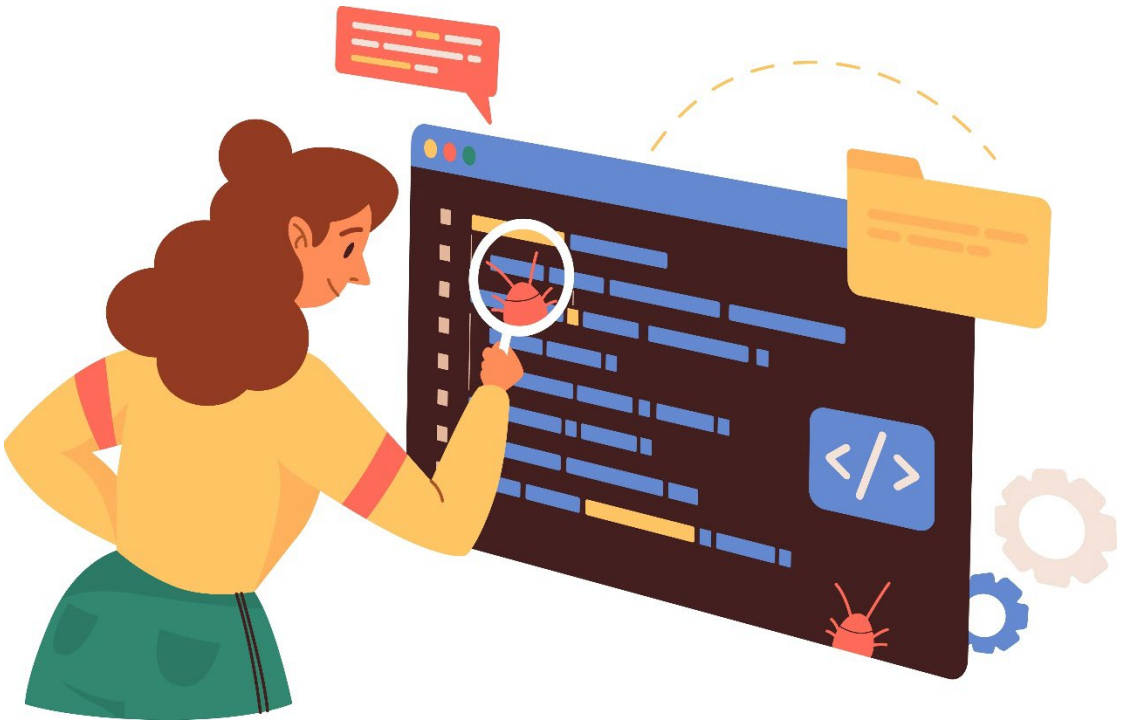
- Depois do teste passar, é realizada uma etapa de refatoração do código.
- A refatoração visa melhorar a estrutura, a legibilidade e a eficiência do código, sem alterar o seu comportamento externo.

6. **Repetição dos ciclos**

- Todo o processo é repetido para cada nova funcionalidade ou requisito.
- Os ciclos de criação de teste, implementação, execução e refatoração são realizados continuamente, garantindo a melhoria contínua do código.

Os ciclos de refinamento contínuo do TDD servem para promover a qualidade do código e do *software* de várias maneiras:

- Os testes automatizados garantem a integridade do código ao verificar se as funcionalidades estão sendo implementadas corretamente;
- A implementação do código é orientada pelos testes, o que resulta em um código mais seguro e confiável;
- A refatoração contínua ajuda a eliminar duplicações, melhorar a legibilidade e a manutenibilidade do código;
- A repetição dos ciclos permite uma evolução iterativa do *software*, onde cada ciclo adiciona novas funcionalidades ou corrige problemas;
- O TDD ajuda a identificar problemas e falhas precocemente, facilitando a correção antes que se tornem mais complexos e onerosos.



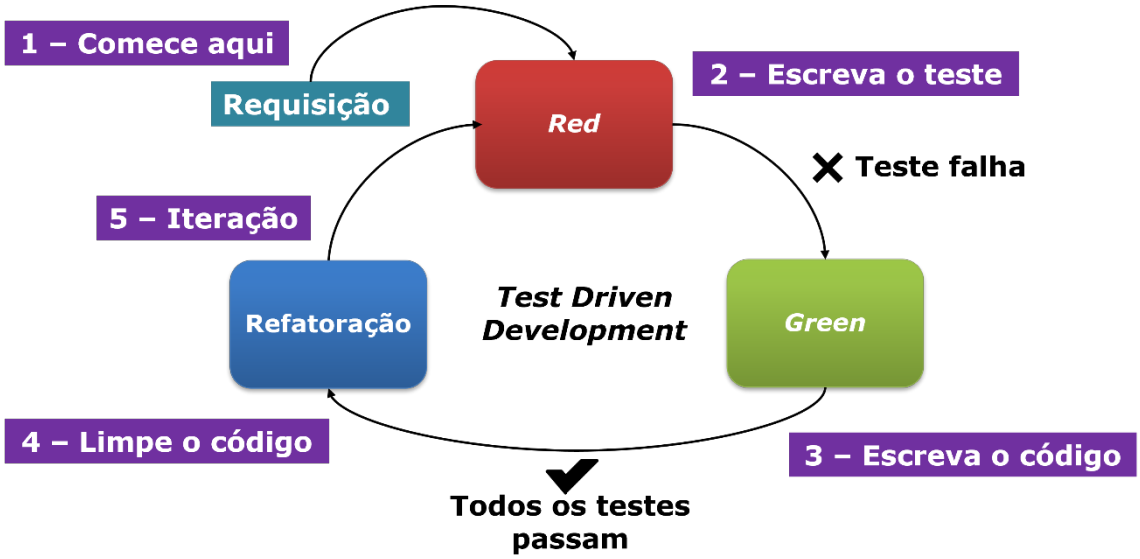
Crédito: Julia Lazebnaya/Shutterstock.

Os ciclos de refinamento contínuo do TDD servem para promover a qualidade do código e do *software* por meio da criação de testes automatizados, implementação do código, execução dos testes, refatoração e repetição contínua dos ciclos. Essa abordagem resulta em um desenvolvimento mais seguro, confiável e iterativo, melhorando a qualidade do *software* de forma gradual e contínua.

O ciclo de trabalho do TDD é chamado de *Red-Green-Refactoring* (Vermelho-Verde-Refatoração). Conforme apresentado na figura 02, o ciclo de trabalho é composto por três fases principais:

- Vermelho (*Red*);
- Verde (*Green*);
- Refatoração (*Refactoring*).

Figura 2 – Ciclo de trabalho do TDD



Na fase Vermelho, o objetivo é escrever um teste automatizado que irá falhar, representando o estado vermelho. Essa etapa é onde identificamos a funcionalidade que desejamos implementar e escrevemos um teste que se espera que falhe, indicando que o código ainda não possui esse recurso.

Uma vez que o teste tenha sido escrito e esteja falhando corretamente, passamos para a fase verde. Nessa etapa, o desenvolvedor deve escrever a quantidade mínima de código necessária para fazer o teste passar, ou seja, tornar o estado verde. O objetivo é fazer com que o teste automatizado seja executado com sucesso, indicando que a funcionalidade foi implementada corretamente.

Após alcançar o estado verde, temos a fase de *refatoração*. Nesse momento, o desenvolvedor revisa o código que foi escrito e o melhora sem alterar seu comportamento externo. É uma oportunidade para aprimorar a legibilidade, a eficiência e a organização do código, tornando-o mais limpo e sustentável a longo prazo. O objetivo é garantir que o código permaneça

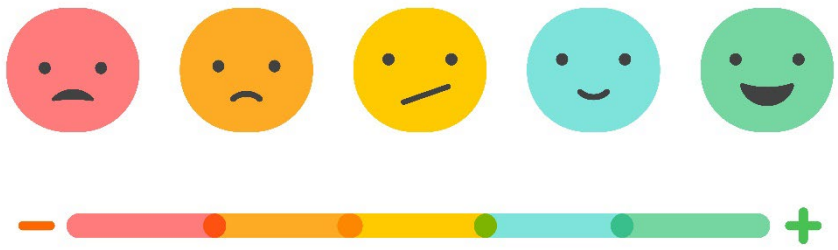
funcionando corretamente após as modificações realizadas durante a refatoração.

Esse ciclo *Red-Green-Refactoring* é repetido continuamente durante o processo de desenvolvimento com TDD. Cada funcionalidade a ser implementada passa por esse ciclo, o que permite um desenvolvimento iterativo e incremental, com testes confiáveis e um código mais robusto.

2.1 Importância da iteração rápida e contínua

A iteração rápida e contínua no contexto do TDD (*Test Driven Development*) desempenha um papel crucial no processo de desenvolvimento de *software*. Ao adotar ciclos curtos de iteração, as equipes podem obter diversos benefícios que impactam diretamente na qualidade do código e no sucesso do projeto.

Em primeiro lugar, a iteração rápida permite que as equipes obtenham um *feedback* contínuo sobre o código que estão produzindo. A cada ciclo, um teste automatizado é criado antes da implementação do código, e sua execução imediata fornece informações valiosas sobre a precisão das funcionalidades desenvolvidas. Esse *feedback* imediato possibilita a identificação precoce de erros e falhas, permitindo correções rápidas e eficientes. Dessa forma, a iteração rápida garante a melhoria contínua do código ao longo do tempo.



Crédito: Marish/Shutterstock.

Além disso, a iteração contínua no TDD contribui para a manutenção de um código legível e bem estruturado. Com a criação de testes automatizados

antes da implementação do código, a equipe é incentivada a escrever um código mais claro e conciso, uma vez que o teste serve como um guia para o desenvolvimento. Isso resulta em um código mais compreensível e de fácil manutenção, facilitando o trabalho colaborativo e a detecção de possíveis problemas.

Outro benefício da iteração rápida é a possibilidade de responder rapidamente a mudanças nos requisitos ou necessidades do cliente. Como o TDD valoriza a criação de testes antes da implementação, a equipe está preparada para realizar ajustes e adaptações de forma ágil, garantindo que o *software* atenda aos requisitos em constante evolução. Essa flexibilidade é essencial em um mercado dinâmico, em que as demandas e expectativas dos usuários podem mudar rapidamente.

Por fim, a iteração rápida e contínua do TDD promove a confiança e a transparência entre os membros da equipe e os *stakeholders*. A entrega frequente de versões incrementais do *software*, acompanhada de testes automatizados, demonstra o progresso e a qualidade do trabalho realizado. Isso permite que as partes interessadas acompanhem de perto o desenvolvimento do projeto, sintam-se envolvidas no processo e tenham maior confiança no produto.

TEMA 3 – COMO DESENVOLVER UM TESTE

Desenvolver um teste utilizando a metodologia *Test Driven Development* (TDD) é um processo sistemático e iterativo que envolve a criação de testes antes mesmo da implementação do código. O objetivo principal do TDD é garantir a qualidade do *software* desde o início do desenvolvimento e promover uma abordagem mais assertiva na criação de funcionalidades.

Para iniciar o desenvolvimento de um teste utilizando TDD, é necessário compreender a funcionalidade que será implementada. Isso envolve entender os requisitos do sistema e definir claramente o comportamento esperado. Com base nessa compreensão, o próximo passo é escrever o teste inicial.

De acordo com Beck (2003), o teste inicial deve ser simples e focado em um aspecto específico da funcionalidade. É importante ter em mente que esse teste inicial provavelmente irá falhar, pois ainda não existe código implementado para atender ao comportamento esperado. Essa falha é esperada e faz parte do processo.

Após escrever o teste inicial, é hora de executá-lo e verificar a falha. Em seguida, o desenvolvedor deve implementar a funcionalidade mínima necessária para fazer o teste passar. Nesse momento, a simplicidade é fundamental. O código deve ser escrito apenas para fazer o teste passar, sem se preocupar com otimizações ou detalhes complexos.

Após a implementação inicial, o próximo passo é executar novamente o teste. Se o teste passar, significa que o código foi implementado corretamente e atende ao comportamento esperado. No entanto, se o teste falhar, é necessário refinar o código e corrigir os erros identificados.

Após a correção, é importante repetir o processo de execução do teste para verificar se todas as falhas foram corrigidas

Um ponto muito importante é que os testes desenvolvidos devem ser automatizados e executados frequentemente durante o processo de desenvolvimento.

Crédito: ArtemisDiana/Shutterstock.

Testes eficazes no TDD utilizam um conjunto de princípios que norteiam sua criação, chamado de modelo F.I.R.S.T. Cada letra do acrônimo F.I.R.S.T. representa um princípio específico. Vamos explorar cada um deles:

- **F – *Fast* (rápido):** os testes devem ser rápidos, ou seja, executarem de forma rápida. Isso é importante para permitir uma execução frequente dos testes durante o ciclo de desenvolvimento. Testes rápidos também

incentivam a criação de testes unitários que verificam unidades pequenas de código, em vez de testar tudo no sistema. Essa abordagem ajuda a identificar problemas com mais facilidade;

- **I – *Isolated* (isolado):** cada teste deve ser independente e isolado de outros testes. Isso significa que um teste não deve depender do estado deixado por outros testes executados anteriormente. Isolamento garante que os testes não produzam resultados inconsistentes ou afetem uns aos outros. Para alcançar esse isolamento, é comum usar técnicas como a criação de novos objetos ou redefinir o estado entre os testes;
- **R - *Repeatable* (repetível):** os testes devem ser repetíveis e produzir os mesmos resultados sempre que forem executados. Isso garante que os testes sejam confiáveis e consistentes. Se um teste falhar, deve falhar repetidamente até que o problema seja resolvido. Repetibilidade é importante para que os desenvolvedores possam confiar nos resultados dos testes e identificar falhas de forma consistente;
- **S – *Self-validating* (autovalidação):** os testes devem ser auto validáveis, ou seja, o resultado do teste deve ser automaticamente verificado. Os testes devem conter asserções ou verificações que determinam se o resultado esperado é alcançado. Isso torna o processo de teste mais claro e ajuda a identificar rapidamente se um teste passou ou falhou;
- **T – *Timely* (oportuno):** os testes devem ser escritos em tempo hábil, ou seja, antes do código de produção correspondente ser implementado. No TDD, o teste é escrito primeiro, antes de escrever o código de produção. Isso ajuda a definir claramente o comportamento esperado do código e a orientar o desenvolvimento. Os testes oportunos também ajudam a evitar a acumulação de código não testado, melhorando a qualidade geral do sistema.

Seguir esses princípios do modelo F.I.R.S.T. no TDD ajuda a criar uma suíte de testes robusta, confiável e fácil de manter. Esses princípios garantem que os testes sejam rápidos, independentes, repetíveis, autovalidáveis e escritos no momento adequado, tornando o processo de desenvolvimento mais ágil e eficaz.

3.1 Definição de casos de teste e cenários

Na abordagem do TDD, a definição de casos de teste e cenários é uma parte essencial do processo. Isso envolve identificar e descrever os diferentes casos de uso, comportamentos e condições que o sistema deve ser capaz de lidar.

A definição de casos de teste começa com a compreensão dos requisitos do sistema. Com base nessa compreensão, é possível identificar os diferentes cenários que precisam ser testados. Cada cenário é uma situação específica em que o sistema é solicitado a executar uma ação e produzir um resultado esperado.

Ao definir os casos de teste, é importante considerar as diversas situações que o sistema pode enfrentar. Isso inclui casos normais, onde o sistema deve funcionar conforme o esperado, bem como casos excepcionais, onde o sistema precisa lidar com condições de erro, entradas inválidas ou situações inesperadas.

Os casos de teste devem ser específicos e abrangentes, cobrindo diferentes funcionalidades e comportamentos do sistema. Cada caso de teste deve ser claro e descrito de forma apropriada, indicando a ação a ser realizada, as condições iniciais e o resultado esperado.

Ao escrever os casos de teste, você pode utilizar diferentes técnicas, como a especificação por exemplo GWT (*given-when-then*), para estruturar e comunicar claramente o cenário de teste. Por exemplo:

- ***Given* (dado):** descrever as condições iniciais ou pré-requisitos necessários para configurar o cenário de teste. Definir o estado inicial do sistema, valores para variáveis ou objetos relevantes e realiza qualquer preparação necessária. O objetivo é estabelecer o contexto no qual o teste será executado;
- ***When* (quando):** descrever a ação ou evento específico que está sendo testado. Nesta seção, é executada a ação que deseja testar, geralmente chamando um método ou função específica do sistema. Essa etapa representa a interação do sistema que está sendo avaliada;
- ***Then* (então):** especificar o resultado esperado após a ação ou evento ter ocorrido. Será verificado se o comportamento do sistema corresponde ao que era esperado. Normalmente, isso é feito por meio de asserções ou

verificações que comparam o resultado real com o resultado esperado. Se o resultado não atender às expectativas, o teste falhará.

Essa estrutura GWT ajuda a comunicar claramente o objetivo do caso de teste e permite que todas as partes interessadas entendam facilmente o que está sendo testado, sem a necessidade de ler todo o corpo do teste. Além disso, a estrutura GWT promove uma divisão clara entre a configuração do cenário, a execução da ação e a verificação do resultado, facilitando a manutenção e o entendimento dos testes.

Saiba mais

Aqui está um exemplo simples para ilustrar a técnica GWT:

- Dado que o usuário está logado no sistema;
- Quando clica no botão *Enviar*;
- Então o sistema deve exibir uma mensagem de confirmação;
- E o sistema deve salvar os dados enviados no banco de dados.

Nesse exemplo, a seção Given descreve a condição inicial em que o usuário está logado no sistema. A seção *When* descreve a ação de clicar no botão *Enviar*. E a seção *Then* especifica o resultado esperado: a exibição de uma mensagem de confirmação e a persistência dos dados no banco de dados.

Essa abordagem ajuda a criar casos de teste claros e compreensíveis, permitindo que todos os envolvidos tenham uma visão clara do que está sendo testado e quais resultados esperar.

Os casos de teste escritos são a base para a criação dos testes automatizados. No TDD, são escritos os testes antes de escrever o código de produção correspondente. Cada caso de teste se torna um teste automatizado que verifica se o comportamento esperado é atendido pelo código.

Conforme o ciclo de desenvolvimento evolui, adicionando mais funcionalidades e requisitos, é importante revisar e atualizar os casos de teste existentes para garantir que reflitam corretamente o comportamento esperado. Também é comum adicionar novos casos de teste à medida que novos requisitos são identificados ou mudanças são solicitadas.

3.2 Escrevendo testes automatizados

Os testes automatizados são escritos utilizando-se de estruturas e *frameworks* específicos para a linguagem de programação escolhida. Geralmente, seguem uma estrutura similar, independentemente da linguagem ou *framework* utilizado.

Aqui estão os passos básicos para escrever testes automatizados:

3.2.1 Configuração

Inicie criando um ambiente de teste adequado. Isso pode incluir a importação das bibliotecas ou *frameworks* de teste necessários e a configuração de quaisquer dependências ou recursos externos que o teste possa exigir.

3.2.2 Preparação

Defina as condições iniciais do teste, como a criação de objetos ou a configuração de estados específicos do sistema que serão testados. Essa etapa é geralmente realizada na seção *Given* (Dado) da técnica GWT.

3.2.3 Execução

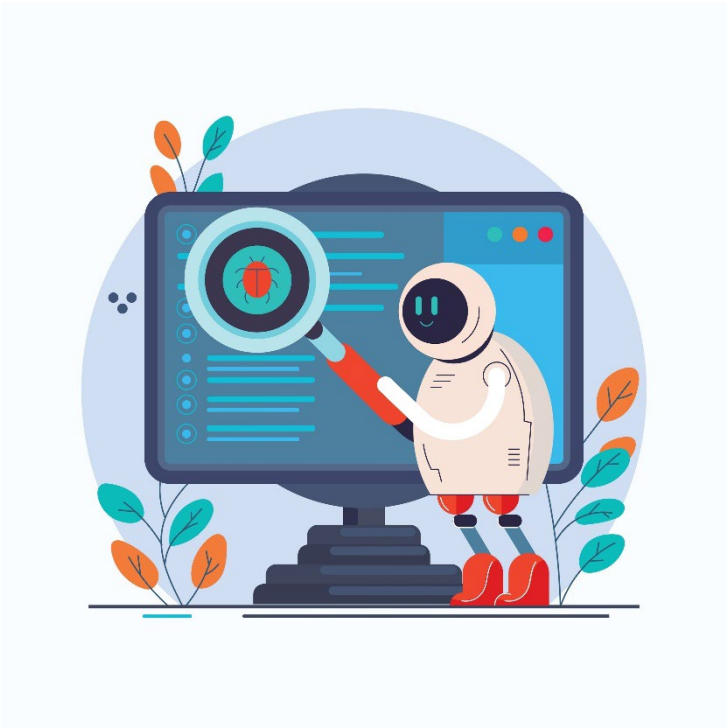
Execute a ação ou o comportamento que você deseja testar. Isso pode envolver a chamada de um método, a interação com a interface do usuário ou a simulação de uma entrada externa. Essa etapa é geralmente realizada na seção *When* (Quando) da técnica GWT.

3.2.4 Verificação

Verifique se o resultado da ação executada corresponde ao resultado esperado. Isso é feito por meio de asserções ou verificações que comparam o resultado real com o resultado esperado. Se houver discrepâncias, o teste falhará. Essa etapa é geralmente realizada na seção *Then* (Então) da técnica GWT.

3.2.5 Limpeza

Realize qualquer limpeza necessária após a conclusão do teste. Isso pode envolver a liberação de recursos, a restauração de estados anteriores ou a eliminação de dados de teste. Essa etapa é importante para garantir que os testes subsequentes não sejam afetados por ações anteriores.



Crédito: CkyBe/Shutterstock.

Recomenda-se uma cobertura de testes abrangente, ou seja, garantir que diferentes cenários e comportamentos do sistema sejam testados para minimizar a possibilidade de erros.

Frameworks populares para teste automatizado, como JUnit (Java), NUnit (.NET), pytest (Python) e Jasmine (JavaScript), fornecem uma estrutura e recursos adicionais para facilitar a escrita, execução e relatório de testes automatizados.

TEMA 4 – TESTE LEGÍVEL, ISOLADO, MINUCIOSO E EXPLÍCITO

No TDD, o teste legível, isolado, minucioso e explícito refere-se a uma abordagem específica para escrever testes unitários. Esses quatro atributos são considerados essenciais para garantir a eficácia e a qualidade dos testes.

4.1 Teste legível

Um teste legível é aquele que pode ser facilmente compreendido por qualquer pessoa que o leia, incluindo desenvolvedores, colegas de equipe ou outros *stakeholders*. A legibilidade é importante para garantir que todos entendam o propósito do teste e possam identificar rapidamente possíveis problemas ou falhas. Os nomes de métodos e variáveis devem ser descritivos e claros, e o código do teste deve ser organizado de maneira lógica e fácil de seguir.

Exemplo:

```
def test_calculate_total_amount():
    # Configuração
    cart = ShoppingCart()
    cart.add_item(Item("Product 1", 10.99))
    cart.add_item(Item("Product 2", 15.99))

    # Execução
    total_amount = cart.calculate_total_amount()

    # Verificação
    assert total_amount == 26.98
```

Nesse exemplo de um teste legível em Python, fica claro o propósito do teste: calcular o valor total de um carrinho de compras. Os nomes de métodos e variáveis são descritivos, tornando o teste fácil de entender. A estrutura do teste é organizada, seguindo a configuração, execução e verificação.

4.2 Teste isolado

Um teste isolado é aquele que não depende de nenhuma outra parte do sistema além da unidade específica que está sendo testada. Isolar o teste significa evitar dependências externas, como acesso a bancos de dados, chamadas de API ou interações com outros componentes do sistema. Isso é alcançado usando substitutos, como *mocks* ou *stubs*, para simular o comportamento das dependências externas. A vantagem de testes isolados é que são mais rápidos, confiáveis e fáceis de depurar.

Exemplo:

```
@Test
public void testCalculateDiscountedPrice() {
    // Configuração
    Product product = new Product("ABC123", "Product 1", 50.0);
    DiscountCalculator calculator = new DiscountCalculator();

    // Execução
    double discountedPrice = calculator.calculateDiscountedPrice(product, 0.1);

    // Verificação
    assertEquals(45.0, discountedPrice, 0.001);
}
```

Neste exemplo de teste isolado em Java, a unidade de código testada é o método calculateDiscountedPrice de um DiscountCalculator. O teste não depende de nenhuma outra parte do sistema além do produto e do próprio DiscountCalculator. Isso é alcançado fornecendo valores diretamente para a configuração e não acessando recursos externos.

4.3 Teste minucioso

Um teste minucioso significa que todas as funcionalidades e comportamentos relevantes da unidade de código estão sendo testados. O objetivo é garantir que todos os caminhos possíveis no código sejam executados e que todas as regras de negócio sejam adequadamente validadas. Um teste minucioso busca cobrir diferentes casos, como entradas válidas e inválidas, limites de valores, exceções e situações de erro. Isso é crucial para detectar possíveis bugs e garantir que a unidade de código se comporte corretamente em todas as circunstâncias.

Exemplo

```
def test_is_valid_email():
    # Caso de sucesso
    assert is_valid_email("example@email.com") == True
```

```
    # Casos de falha
    assert is_valid_email("exampleemail.com") == False
    assert is_valid_email("example@.com") == False
    assert is_valid_email("@email.com") == False
    assert is_valid_email("example@") == False
```

Nesse exemplo, o teste minucioso para a função is_valid_email valida diferentes casos para verificar se um email é válido ou não. Além do caso de sucesso, são testados casos de falha, como *emails* sem o símbolo @ ou sem o domínio adequado. Dessa forma, diferentes cenários são cobertos para garantir a validação correta.

4.4 Teste explícito

Um teste explícito é aquele que descreve claramente o que está sendo testado e qual é o resultado esperado. Define as pré-condições necessárias para a execução do teste, executa a operação que está sendo testada e, em seguida, verifica se o resultado é o esperado. Isso é feito por meio de declarações assertivas que comparam o resultado real com o esperado. Testes explícitos são importantes para fornecer *feedback* rápido e preciso sobre o estado da unidade de código. Se um teste falhar, deve fornecer informações claras sobre o motivo da falha.

Exemplo

```
test("calculateTax should correctly calculate the tax amount", () => {
    // Configuração
    const price = 100;
    const taxRate = 0.1;

    // Execução
    const taxAmount = calculateTax(price, taxRate);

    // Verificação
    expect(taxAmount).toEqual(10);
});
```

Neste exemplo em JavaScript, o teste é explícito em seu propósito. Testa a função calculateTax para garantir que ela calcule corretamente o valor do imposto. O teste configura um preço e uma taxa de imposto, executa o cálculo e verifica se o resultado é igual ao esperado (10).

TEMA 5 – COMPLEXIDADE INCREMENTAL – *INCREMENTAL DESIGN*

O TDD tem como objetivo principal melhorar a qualidade do código e facilitar a manutenção contínua, por meio de um processo iterativo de desenvolvimento. Uma das características fundamentais do TDD é a abordagem incremental, na qual a complexidade é adicionada gradualmente ao longo do tempo. Neste tópico, exploraremos a complexidade incremental e o *incremental design* no contexto do TDD.

Enquanto a complexidade incremental é uma característica mais geral que pode surgir durante o desenvolvimento incremental ou em outras situações em que a complexidade aumenta ao longo do tempo, o *incremental design* é uma abordagem específica para o desenvolvimento de sistemas em etapas,

5.1 O que é complexidade incremental?

A complexidade incremental é a prática de adicionar novos recursos e funcionalidades de forma gradual e controlada ao longo do processo de desenvolvimento. Em vez de tentar implementar todos os requisitos de uma só vez, a complexidade incremental propõe uma abordagem passo a passo, onde cada incremento é testado e validado antes de avançar para o próximo. Essa abordagem permite que os desenvolvedores identifiquem e resolvam problemas de forma mais eficaz, além de manter um código mais limpo e organizado.

5.2 Complexidade incremental no TDD

No contexto do TDD, a complexidade incremental é uma abordagem essencial. O ciclo básico do TDD consiste em três etapas: escrever um teste, implementar o código mínimo para que o teste passe e refatorar o código para melhorar sua qualidade. Essa sequência de atividades iterativas permite que a complexidade seja adicionada gradualmente ao código à medida que novos testes são escritos e implementados.

Vamos explorar os detalhes da complexidade incremental no TDD:

5.2.1 Escrita de testes como requisitos

Os testes são escritos antes da implementação do código. Cada teste representa um requisito funcional específico que o *software* deve atender. Esses testes são baseados nas expectativas e comportamentos desejados do *software*. Ao escrever testes como requisitos, os desenvolvedores têm uma visão clara do que é esperado antes mesmo de começar a escrever o código. Esses testes funcionam como uma especificação viva do *software*, orientando o desenvolvimento incremental.

Exemplo:

Suponha que você esteja desenvolvendo uma calculadora simples. Um exemplo de teste como requisito poderia ser:

```
def test_soma():
    calculadora = Calculadora()
    resultado = calculadora.soma(2, 3)
    assert resultado == 5
```

Nesse caso, o teste "test_soma" representa o requisito de que a função soma() da calculadora deve retornar corretamente a soma de dois números.

5.2.2 Implementação mínima

Após escrever os testes, o próximo passo é implementar o código mínimo necessário para que o teste passe. Nessa fase, o foco principal é atender aos requisitos do teste atual, sem se preocupar com funcionalidades futuras. Isso significa que o código pode ser simples e direcionado, sem adicionar complexidade desnecessária desde o início. A implementação mínima permite que os desenvolvedores avancem gradualmente, adicionando novas funcionalidades de forma controlada.

Exemplo:

Ao implementar a função soma() da calculadora para que o teste passe, a implementação mínima seria:

```
class Calculadora:
    def soma(self, a, b):
        return a + b
```

Nesse exemplo, a implementação mínima apenas realiza a soma dos dois números passados como argumento.

5.2.3 Teste e validação contínuos

Após a implementação mínima, os testes são executados para verificar se o código está funcionando corretamente. Essa validação contínua é uma parte essencial do TDD. Os testes funcionam como um mecanismo de *feedback* imediato, identificando problemas e erros. Se um teste falhar, isso indica que algo não está funcionando corretamente e requer uma correção imediata. Os desenvolvedores podem depurar e resolver o problema antes de prosseguir.

Exemplo:

Após a implementação, os testes são executados para verificar se o código está funcionando corretamente. No exemplo anterior, o teste "test_soma" seria executado para validar a função soma():

```
def test_soma():
    calculadora = Calculadora()
    resultado = calculadora.soma(2, 3)
    assert resultado == 5
```

Se o teste passar, significa que a implementação está correta. Caso contrário, o teste falhará, indicando que algo precisa ser corrigido.

5.2.4 Refatoração

Após um teste passar, o código é refatorado. A refatoração é o processo de melhorar a estrutura interna do código sem alterar seu comportamento externo. Essa etapa é fundamental para manter o código limpo, legível e organizado. Durante a refatoração, as melhorias são feitas para eliminar duplicações, melhorar a clareza e a eficiência do código, reduzindo a complexidade acumulada e facilita a manutenção futura.

Exemplo:

Suponha que você queira refatorar a função soma() para melhorar sua legibilidade. A refatoração poderia ser feita da seguinte maneira:

```
class Calculadora:
    def soma(self, a, b):
        resultado = a + b
        return resultado
```

Nesse exemplo, a refatoração consiste em armazenar o resultado da soma em uma variável antes de retorná-lo.

5.2.5 Repetição do ciclo

Após a refatoração, o ciclo se repete com a adição de novos requisitos. Novos testes são escritos para as funcionalidades adicionadas, e a implementação mínima é realizada para que os testes passem. Em seguida, ocorre a validação contínua e a refatoração, garantindo a qualidade contínua do código.

Exemplo:

Após a refatoração, você pode adicionar novos requisitos e repetir o ciclo do TDD. Por exemplo, você pode adicionar um novo teste para verificar a função de subtração da calculadora:

```
def test_subtracao():
    calculadora = Calculadora()
    resultado = calculadora.subtracao(5, 2)
    assert resultado == 3
```

Em seguida, você implementa a função subtracao() com a implementação mínima, executa os testes, faz a validação contínua e, se necessário, realiza a refatoração.

A complexidade incremental no TDD traz uma série de benefícios. Ela permite que os desenvolvedores construam software de alta qualidade, melhorem a detecção precoce de problemas, sejam ágeis na resposta a mudanças de requisitos e facilitem o trabalho em equipe por meio de testes como requisitos claros e comuns.

5.3 Benefícios da complexidade incremental no TDD

A abordagem de complexidade incremental no TDD traz vários benefícios significativos:

5.3.1 Melhoria da qualidade do código

Ao adicionar complexidade de forma gradual e controlada, os desenvolvedores podem focar na criação de um código mais limpo, modular e bem estruturado. Isso resulta em *software* de maior qualidade, mais fácil de entender, dar manutenção e evoluir.

5.3.2 Detecção precoce de problemas

A escrita de testes antes da implementação permite que os desenvolvedores identifiquem e corrijam problemas de forma mais rápida. Os testes atuam como um mecanismo de *feedback* imediato, alertando sobre possíveis falhas no código. Isso contribui para a detecção precoce de erros e melhora a robustez do *software*.

5.3.3 Agilidade e flexibilidade

A abordagem incremental permite que a equipe de desenvolvimento responda de forma ágil a mudanças de requisitos e novas necessidades do negócio. Como o *software* é construído em pequenos incrementos, é mais fácil adaptar e ajustar o código à medida que as demandas evoluem.

5.3.4 Facilita o trabalho em equipe

A complexidade incremental no TDD promove a colaboração e a comunicação efetiva entre os membros da equipe. Os testes funcionam como uma linguagem comum, facilitando o entendimento das funcionalidades e dos requisitos do *software*.

5.4 O que é *incremental design* (*design* incremental)?

O *incremental design* é uma abordagem de *design* de *software* em que o sistema é desenvolvido em pequenos incrementos, com cada incremento

adicionando novas funcionalidades ou melhorias ao sistema existente. O *design* incremental se concentra na evolução gradual do sistema, permitindo que cresça e se adapte às necessidades em constante mudança. Ao adotar o Incremental Design, a arquitetura e o *design* do sistema são construídos e expandidos gradualmente, evitando decisões de *design* em grande escala antecipadamente.

A abordagem incremental de *design* geralmente está relacionada ao processo de desenvolvimento de *software*, onde as funcionalidades são adicionadas iterativamente, sendo aprimoradas e refinadas à medida que o sistema é construído. Esse processo permite a validação contínua das decisões de design, a correção de erros e a adaptação às mudanças de requisitos.

5.5 *Incremental design* no TDD

O *incremental design* no contexto do *Test Driven Development* refere-se à abordagem de projetar e construir *software* de forma incremental, adicionando novas funcionalidades ou melhorias em pequenos incrementos ao longo do tempo. É uma prática que permite que o *design* do sistema evolua gradualmente, em resposta aos requisitos em constante mudança, enquanto mantém um foco contínuo na validação e teste do *software*.

Apresentaremos a seguir os principais pontos a serem considerados ao aplicar o *incremental design* no TDD:

- **Pequenos incrementos:** o desenvolvimento incremental no TDD envolve a criação de incrementos pequenos e significativos de funcionalidades. Cada incremento adiciona uma nova peça de valor ao *software*. Esses incrementos são planejados e implementados em uma sequência lógica, construindo sobre o que já foi implementado anteriormente;
- **Testes como requisitos:** cada teste descreve um requisito funcional específico. Ao projetar incrementalmente no TDD, os testes são utilizados como um guia para a construção incremental do *software*. Cada incremento deve ser projetado para atender aos requisitos dos testes relacionados;
- **Refatoração contínua:** à medida que novos incrementos são adicionados, é essencial revisar e melhorar continuamente o *design* do software. A refatoração envolve a reestruturação do código existente, sem alterar seu comportamento externo. Isso é feito para melhorar a

legibilidade, a manutenibilidade e a qualidade geral do código, permitindo que o *design* evolua com maior eficiência;

- **Validação contínua:** à medida que novas funcionalidades são adicionadas. Cada incremento é testado individualmente para garantir que atenda aos requisitos definidos pelos testes escritos anteriormente. A validação contínua assegura que o *software* esteja funcionando corretamente e que os novos incrementos não introduzam erros ou problemas no sistema existente;
- ***Feedback* e aprendizado:** o desenvolvimento incremental no TDD também promove um ciclo rápido de *feedback* e aprendizado. À medida que os incrementos são adicionados e validados, os desenvolvedores recebem *feedback* imediato sobre a funcionalidade implementada. Esse *feedback* permite ajustes e melhorias contínuas, além de facilitar a identificação precoce de problemas ou requisitos não atendidos.

5.6 Benefícios do *incremental design* no TDD

Existem vários benefícios do *incremental design* no TDD. Apresentaremos alguns deles:

5.6.1 Código mais limpo e modular

Ao projetar e implementar o código em pequenos incrementos, é possível manter o código mais limpo, coeso e modular. Cada incremento representa uma funcionalidade específica, o que ajuda a evitar a criação de código desnecessário e a manter a estrutura geral do sistema mais organizada.

5.6.2 Melhor compreensão dos requisitos

O desenvolvedor se concentra em desenvolver apenas a funcionalidade necessária para passar nos testes escritos. Isso ajuda a entender melhor os requisitos do sistema, por estar constantemente iterando sobre pequenas partes do código e validando-as com testes.

5.6.3 *Feedback* rápido

Ocorre o *feedback* rápido sobre o sucesso ou falha dos testes à medida que são escritos e executados, permitindo a pronta detecção de erros e com isso a imediata correção, facilitando a depuração do código.

5.6.4 Maior confiança na qualidade do código

Os testes são constantemente adicionados e atualizados à medida que novas funcionalidades são implementadas, auxiliando no aumento da confiança na qualidade do código.

5.6.5 Flexibilidade para mudanças

Cada incremento é pequeno e independente, fica mais fácil fazer alterações no código quando novos requisitos surgem ou quando há a necessidade de refatoração. Isso reduz o risco de introduzir erros em áreas não relacionadas do sistema durante as modificações.

FINALIZANDO

Neste capítulo, exploramos a aplicabilidade do TDD e destacamos os benefícios de criar testes antes da implementação do código. Ao seguir essa abordagem, podemos garantir que nosso código atenda aos requisitos e funcione corretamente desde o início, evitando erros e problemas futuros.

Discutimos os ciclos de refinamento contínuo do TDD, que nos permitem melhorar gradualmente nosso código por meio de iterações constantes. Ao escrever um teste, implementar o código necessário e, em seguida, refinar e aprimorar, podemos alcançar soluções mais eficientes e de maior qualidade.

Exploramos também como desenvolver testes eficazes, enfatizando a importância de torná-los legíveis, isolados, minuciosos e explícitos. Essas características garantem que nossos testes sejam compreensíveis, abranjam todas as situações relevantes e especifiquem claramente o comportamento esperado do código.

Por fim, abordamos o conceito de complexidade incremental e *design* incremental. Ao adotar uma abordagem gradual na construção do nosso código,

adicionando funcionalidades de forma incremental, podemos evitar a complexidade excessiva e facilitar a manutenção e evolução futura.

REFERÊNCIAS

BECK, K. **Test-Driven development**: by example. Boston: Addison-Wesley Professional, 2003.

FEITOSA, D. S. **Um estudo sobre o impacto do uso de desenvolvimento orientado por testes na melhoria da qualidade de *software***. Salvador: Universidade Federal da Bahia, Instituto de Matemática, 2007.

FREEMAN, S.; PRYCE, N. **Desenvolvimento de *software* orientado a objetos, guiado por testes**. Rio de Janeiro: Alta Books, 2012.

JONES, C. G. Test-Driven development goes to school. **ACM Digital Library**, 2004. Disponível em: <https://shre.ink/9LZc>. Acesso em: 20 jul. 2023.

MASSOL, V.; HUSTED, T. **JUnit in Action**. 2. ed. New York: Manning Publications, 2003.