PROGRAMANDO MELHOR

TEST DRIVEN DESIGN

TDD & C#

DESENVOLVIMENTO,
PRODUTIVIDADE,
QUALIDADE.

ALEXANDRE MONTANHA

Sobre o Autor

Sobre o Autor - Professor Montanha

Biografia:

Professor Montanha é um especialista em programação e desenvolvimento de software. Com vasta experiência no campo da tecnologia da informação, ele tem se dedicado a compartilhar seu conhecimento e habilidades por meio do ensino e da pesquisa.

Formação Acadêmica:

- Tec. Processamento de Dados
- Historiador pela Universidade Federal de Minas Gerais
- Especialista em Gamification
- Especialista em Educação
- Mestrado em Educação e Gestão Social (Ensino e Games)

Experiência Profissional:

Professor Montanha possui uma carreira sólida como professor universitário, consultor e palestrante em importantes eventos e conferências da área. Seus principais campos de especialização incluem TDD, desenvolvimento ágil de software, testes de software e programação orientada a objetos.

Ele também é conhecido por sua paixão em transmitir conhecimentos de forma clara e acessível, tornando conceitos complexos compreensíveis para estudantes e profissionais de todos os níveis. Seus métodos pedagógicos inovadores têm sido amplamente reconhecidos e elogiados por seus alunos.

LinkedIn:

Para saber mais sobre o Professor Montanha e se manter atualizado sobre seus projetos e publicações, você pode visitar o seu perfil no LinkedIn: https://www.linkedin.com/in/professor-montanha

A importância do teste

A importância dos testes na programação é indiscutível. Os testes desempenham um papel crucial no desenvolvimento de software, garantindo a qualidade, confiabilidade e funcionalidade do código. Eles permitem que os programadores identifiquem erros e problemas antes que o software seja implantado, o que é fundamental para evitar falhas e melhorar a experiência do usuário.

Existem diferentes tipos de testes que podem ser aplicados durante o processo de desenvolvimento de software. Os testes unitários são responsáveis por verificar se cada unidade individual do código funciona corretamente. Eles ajudam a identificar erros em partes específicas do software e permitem que os programadores corrijam esses problemas antes que eles se espalhem por todo o sistema.

Além dos testes unitários, existem os testes de integração, que verificam se diferentes componentes do software funcionam corretamente quando combinados. Esses testes são essenciais para garantir que as partes do sistema se comuniquem adequadamente e que não ocorram problemas de compatibilidade.

Os testes de aceitação ou testes funcionais são voltados para verificar se o software atende aos requisitos e às expectativas do usuário. Esses testes são realizados sob a perspectiva do usuário final, simulando cenários reais de uso. Eles garantem que o software seja intuitivo, fácil de usar e cumpra seu propósito principal.

Além desses testes, também é importante realizar testes de desempenho, segurança e usabilidade, dependendo das necessidades específicas do software em desenvolvimento.

A implementação de testes adequados traz vários benefícios para os desenvolvedores e para as empresas. Em primeiro lugar, os testes ajudam a encontrar e corrigir erros mais cedo no processo de desenvolvimento, o que reduz o tempo e os custos associados às correções tardias. Isso também contribui para um processo de desenvolvimento mais ágil e eficiente.

Além disso, os testes fornecem uma camada de segurança e confiança ao software. Eles permitem que os programadores validem a funcionalidade do código em diferentes cenários e condições, o que reduz o risco de falhas inesperadas quando o software é implantado em produção.

Os testes também facilitam a manutenção do código ao longo do tempo. Com uma suíte de testes bem elaborada, os desenvolvedores podem realizar alterações e adições de funcionalidades com maior segurança, pois podem executar os testes para garantir que as mudanças não afetem negativamente o restante do sistema.

Por fim, os testes promovem uma cultura de qualidade e excelência na equipe de desenvolvimento. Ao incorporar os testes como uma prática fundamental, os desenvolvedores se tornam mais conscientes da importância da qualidade do código e se esforçam para entregar um produto mais sólido.

Em resumo, os testes desempenham um papel vital no desenvolvimento de software. Eles garantem a qualidade, confiabilidade e funcionalidade do código, reduzem o risco de falhas, facilitam a manutenção e promovem uma cultura de qualidade na equipe de desenvolvimento. Portanto, investir tempo e recursos em testes é fundamental para o sucesso de um projeto de programação.

TDD

TDD, ou Test-Driven Development (Desenvolvimento Orientado a Testes), é uma abordagem de desenvolvimento de software que coloca os testes no centro do processo. Com o TDD, os testes são escritos antes mesmo da implementação do código. Essa abordagem segue um ciclo iterativo e incremental, onde os testes orientam o desenvolvimento do software e fornecem feedback imediato sobre a qualidade do código.

O processo do TDD é geralmente composto por três etapas: "Red-Green-Refactor" (Vermelho-Verde-Refatorar). Na primeira etapa, chamada de "Red" (Vermelho), o programador escreve um teste automatizado que descreve a funcionalidade que será implementada. Nesse ponto, o teste falhará, pois a funcionalidade ainda não existe.

Em seguida, na etapa "Green" (Verde), o programador escreve a implementação mínima necessária para fazer o teste passar. O foco é apenas fazer o teste ser bem-sucedido, sem se preocupar com otimizações ou estruturas complexas. O objetivo é atender aos requisitos mínimos para que o teste seja aprovado.

Após a etapa "Green", o próximo passo é a etapa "Refactor" (Refatorar). Nessa fase, o programador melhora a estrutura do código, realiza otimizações, remove duplicações e torna o código mais legível e sustentável. É importante ressaltar que todos esses refinamentos são feitos sem alterar o comportamento do software, pois o teste deve continuar passando.

O processo "Red-Green-Refactor" é repetido continuamente para cada nova funcionalidade ou alteração no código existente. Dessa forma, o código é desenvolvido de forma incremental, com testes sendo adicionados e aperfeiçoados à medida que o

software evolui. Essa abordagem garante que cada parte do código seja testada de forma isolada e que o software permaneça funcional mesmo após adições e modificações.

O TDD traz vários benefícios para o desenvolvimento de software. Em primeiro lugar, ele ajuda a melhorar a qualidade do código, uma vez que os testes garantem que o software funcione corretamente. Além disso, a abordagem do TDD encoraja a modularidade e o baixo acoplamento, pois os testes favorecem a escrita de código em pequenos módulos independentes.

O TDD também oferece uma maior segurança ao fazer alterações no código existente. Como todos os testes são executados automaticamente, qualquer problema ou regressão será identificado imediatamente. Isso dá aos desenvolvedores a confiança necessária para realizar refatorações e melhorias no código sem medo de introduzir erros.

Outro benefício do TDD é a documentação viva do software. Os testes servem como exemplos claros de como o código deve ser utilizado e quais resultados esperar. Isso facilita a compreensão do sistema por parte de outros desenvolvedores e ajuda na manutenção e evolução do software.

Em resumo, o TDD é uma abordagem de desenvolvimento de software que coloca os testes como uma parte central do processo. Essa abordagem traz benefícios como melhoria da qualidade do código, modularidade, segurança ao realizar alterações e documentação clara do software. Ao adotar o TDD, os desenvolvedores podem criar software mais confiável, sustentável e de alta qualidade.

TDD e o C#

O C# é uma linguagem de programação versátil e poderosa que suporta a implementação da metodologia TDD (Test-Driven Development). Com recursos e ferramentas específicas, o C# fornece aos desenvolvedores um ambiente propício para a prática do TDD.

Uma das principais características do C# que facilita a implementação do TDD é o suporte nativo a testes unitários por meio do framework de testes chamado MSTest. O MSTest é uma biblioteca fornecida pelo Microsoft Visual Studio que permite a criação e execução de testes de unidade de forma simples e eficiente.

Para implementar o TDD no C#, você pode criar classes de teste separadas para cada unidade do seu código. Essas classes de teste contêm métodos de teste individuais que verificam o comportamento esperado de cada parte do código. Ao utilizar as asserções do MSTest, você pode verificar se os resultados obtidos estão de acordo com o esperado.

Além do MSTest, existem outros frameworks de teste populares para C#, como NUnit e xUnit, que também podem ser usados para implementar o TDD. Esses frameworks oferecem recursos semelhantes ao MSTest, permitindo a criação de testes de unidade e a verificação dos resultados esperados.

Além dos frameworks de teste, o C# possui recursos que auxiliam na implementação do TDD, como a capacidade de criar classes e métodos com modificadores de acesso restrito (como private e internal). Isso permite que você se concentre no teste das unidades individuais do código, mantendo o encapsulamento adequado.

Outra funcionalidade importante do C# que facilita o TDD é o recurso de reflexão. Através da reflexão, é possível obter informações sobre as classes e seus membros em tempo de execução, o que pode ser útil para criar testes dinâmicos e explorar a estrutura do código.

Além disso, o ecossistema do C# é rico em ferramentas de desenvolvimento que apoiam o TDD, como o Visual Studio e o Visual Studio Code. Essas IDEs oferecem recursos de depuração, execução de testes automatizados e integração com frameworks de testes, tornando o processo de implementação e execução dos testes mais eficiente e produtivo.

Em resumo, o C# é uma linguagem que oferece suporte nativo e ferramentas específicas para a implementação do TDD. Com o MSTest, NUnit, xUnit e outras bibliotecas de teste disponíveis, os desenvolvedores podem escrever testes unitários eficazes e verificar se o código atende aos requisitos esperados. A combinação dessas ferramentas com recursos como modificadores de acesso restrito e reflexão torna o processo de implementação do TDD no C# uma tarefa mais fácil e eficiente.

Mão na Massa: Planejando bons Testes

Ao planejar bons testes no TDD (Test-Driven Development), é essencial seguir algumas diretrizes para obter resultados eficazes. Aqui estão algumas etapas para ajudá-lo a planejar testes sólidos no TDD:

1. Compreender os requisitos: Antes de começar a escrever qualquer código, é crucial entender claramente os requisitos e as funcionalidades que devem ser implementadas. Isso permitirá que você planeje seus testes de forma adequada e abrangente.

2. Identificar as unidades de teste: Analise o código e identifique as unidades ou partes individuais que serão testadas. As unidades podem ser classes, métodos ou até mesmo pequenos trechos de código. É importante ter uma visão clara das unidades a serem testadas para que você possa planejar os testes de maneira mais precisa.

3. Escrever testes de unidade: No TDD, os testes são escritos antes da implementação do código. Comece escrevendo testes de unidade para cada uma das unidades identificadas. Concentre-se em testar o comportamento esperado das unidades, fornecendo diferentes cenários e casos de teste.

4. Definir casos de teste: Ao escrever os testes, defina casos de teste específicos para cada funcionalidade ou comportamento que você deseja testar. Considere diferentes entradas, valores limites e situações excepcionais para garantir uma cobertura abrangente.

5. Testes automatizados: Os testes no TDD devem ser automatizados, o que significa que podem ser executados repetidamente sem intervenção manual. Certifique-se de utilizar

um framework de teste adequado, como MSTest, NUnit ou xUnit, para automatizar seus testes e obter relatórios de execução precisos.

6. Executar os testes: Após escrever os testes, execute-os para verificar se todos falham, como esperado. Essa é a etapa "Red" do ciclo "Red-Green-Refactor" do TDD. Se todos os testes falharem, isso indica que a implementação do código ainda está pendente.

7. Implementar o código mínimo necessário: Agora, implemente a funcionalidade mínima necessária para fazer os testes passarem. Concentre-se apenas em fazer os testes "verdes" sem se preocupar com otimizações ou melhorias adicionais. Mantenha a simplicidade e a clareza do código.

8. Refatorar o código: Após fazer os testes passarem, é hora de refatorar o código. Melhore a estrutura, remova duplicações, aplique boas práticas de programação e torne o código mais legível e sustentável. Certifique-se de que todos os testes ainda estejam passando após a refatoração.

9. Iterar: Repita essas etapas para cada nova funcionalidade ou alteração no código. O TDD é um processo iterativo, onde você adiciona testes, implementa o código mínimo e refatora continuamente. Cada iteração aumenta a qualidade e a robustez do software.

Ao planejar seus testes no TDD, lembre-se de manter os testes simples, focados e independentes uns dos outros. Isso ajudará a garantir que seus testes sejam confiáveis, fáceis de manter e forneçam uma cobertura abrangente do código.

A calculadora

Vamos supor que estamos desenvolvendo uma calculadora simples no TDD. Seguindo os passos mencionados, vamos planejar um caso de teste para a funcionalidade de soma. Aqui está um exemplo de como isso poderia ser feito:

Passo 1: Compreender os requisitos

Requisito: A calculadora deve ser capaz de somar dois números inteiros.

Passo 2: Identificar as unidades de teste

Unidade de teste: Função de soma da calculadora.

Passo 3: Escrever testes de unidade

Escrevemos um teste de unidade para a função de soma da calculadora.

```
[TestClass]
0 referências
public class CalculatorTests
{
    [TestMethod]
    0 referências
    public void Add_TwoIntegers_ReturnsSum()
    {
        // Arrange
        Calculator calculator = new Calculator();

        // Act
        int result = calculator.Add(2, 3);

        // Assert
        Assert.AreEqual(expected: 5, actual: result);
    }
}
```

Passo 4: Definir casos de teste

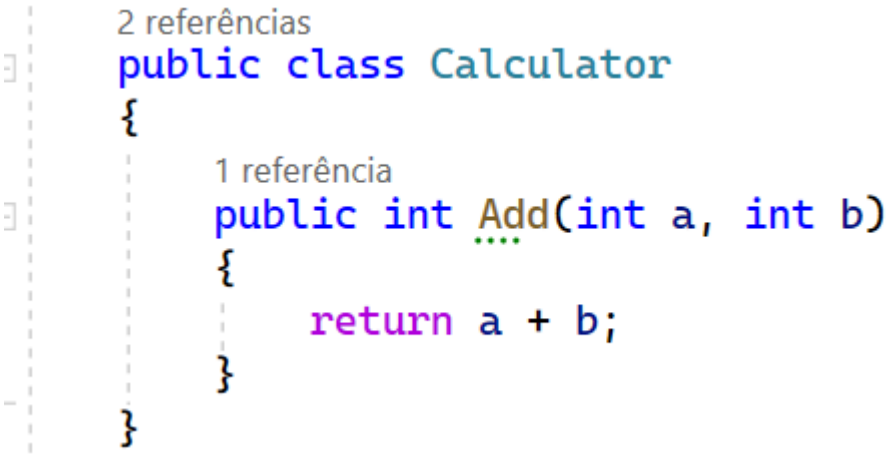
Definimos um caso de teste específico para a função de soma, onde somamos os números 2 e 3.

Passo 5: Testes automatizados

Os testes são escritos usando um framework de teste, como MSTest. O exemplo acima usa a sintaxe do MSTest.

Passo 6: Executar os testes

Ao executar o teste, esperamos que ele falhe, já que ainda não implementamos a função de soma na calculadora.



Passo 7: Implementar o código mínimo necessário

Agora, implementamos a função de soma na calculadora:

Passo 8: Refatorar o código

Após a implementação, podemos refatorar o código para melhorar a legibilidade, estrutura e aderência às boas práticas de programação. Neste exemplo simples, não há muito espaço para refatoração.

Passo 9: Iterar

Podemos repetir esse processo para adicionar mais funcionalidades à calculadora, como subtração, multiplicação e divisão, seguindo os mesmos passos.

Ao seguir esses passos, criamos um caso de teste usando o TDD para a funcionalidade de soma da calculadora. Esse teste nos permite validar se a função de soma está funcionando corretamente e fornecerá um feedback imediato se algo der errado ao adicionar mais funcionalidades à calculadora.

Exercício

Você foi contratado para desenvolver um programa de cálculo de aposentadoria. O programa deve solicitar ao usuário o seu nome, sexo e tempo de contribuição. Com base nessas informações, o programa deve calcular o tempo que falta para a pessoa se aposentar.

Fórmula de aposentadoria:

A fórmula de aposentadoria será diferente para homens e mulheres:

- Homens: Aposentadoria por tempo de contribuição. Para se aposentar, é necessário ter pelo menos 35 anos de contribuição.

- Mulheres: Aposentadoria por idade e tempo de contribuição. Para se aposentar, é necessário ter pelo menos 30 anos de contribuição e ter no mínimo 60 anos de idade.

Pontos para testes unitários:

1. Teste para um homem com tempo de contribuição de 30 anos: Espera-se que o programa retorne o tempo restante até a aposentadoria como 5 anos.

2. Teste para uma mulher com tempo de contribuição de 25 anos e idade de 55 anos: Espera-se que o programa retorne o tempo restante até a aposentadoria como 5 anos.

3. Teste para um homem com tempo de contribuição de 40 anos: Espera-se que o programa retorne o tempo restante até a aposentadoria como 0 anos.

4. Teste para uma mulher com tempo de contribuição de 20 anos e idade de 50 anos: Espera-se que o programa retorne o tempo restante até a aposentadoria como 10 anos.

Implementação do cálculo e testes unitários:

Implemente uma função chamada "calcularTempoAposentadoria" que recebe como parâmetros o sexo (uma string contendo "masculino" ou "feminino") e o tempo de contribuição (um número inteiro representando a quantidade de anos de contribuição). A função deve retornar o tempo restante até a aposentadoria, em anos.

Em seguida, implemente os testes unitários descritos acima para verificar se a função está calculando corretamente o tempo restante até a aposentadoria.

Exemplo de implementação em Python:

```python
def calcularTempoAposentadoria(sexo, tempo_contribuicao):
    if sexo == "masculino":
        tempo_restante = 35 - tempo_contribuicao
    elif sexo == "feminino":
        if tempo_contribuicao >= 30:
            tempo_restante = 0
        else:
            tempo_restante = 30 - tempo_contribuicao
    else:
        raise ValueError("Sexo inválido!")

    return tempo_restante
```

Lembre-se de implementar os testes unitários correspondentes e executá-los para verificar se a função está retornando os resultados esperados.

Referências Bibliográficas

Aqui estão algumas referências bibliográficas no formato da ABNT que podem fornecer suporte ao conteúdo de TDD, TDD com C#, testes de software em geral, C# e código limpo:

1. Livros sobre TDD e Testes de Software:

Autor: Kent Beck

Título: Test-Driven Development: By Example

Ano: 2003

Editora: Addison-Wesley Professional

ISBN: 978-0321146533

Autor: Robert C. Martin

Título: Clean Code: A Handbook of Agile Software Craftsmanship

Ano: 2008

Editora: Prentice Hall

ISBN: 978-0132350884

2. Livros sobre TDD com C#:

Autor: James W. Newkirk, Alexei A. Vorontsov

Título: Test-Driven Development in Microsoft .NET

Ano: 2004

Editora: Microsoft Press

ISBN: 978-0735619487

3. Livros sobre Testes de Software em Geral:

Autor: Cem Kaner, Jack Falk, Hung Q. Nguyen

Título: Testing Computer Software

Ano: 1999

Editora: Wiley

ISBN: 978-0471358466

Autor: Rex Black

Título: Foundations of Software Testing: ISTQB Certification

Ano: 2011

Editora: Cengage Learning

ISBN: 978-1844809899

4. Livros sobre C#:

Autor: Andrew Troelsen

Título: Pro C# 7: With .NET and .NET Core

Ano: 2017

Editora: Apress

ISBN: 978-1484230176

Autor: Jon Skeet

Título: C# in Depth

Ano: 2019

Editora: Manning Publications

ISBN: 978-1617294532

5. Livros sobre Código Limpo:

Autor: Robert C. Martin

Título: Clean Code: A Handbook of Agile Software Craftsmanship

Ano: 2008

Editora: Prentice Hall

ISBN: 978-0132350884

Autor: Steve McConnell

Título: Code Complete: A Practical Handbook of Software Construction

Ano: 2004

Editora: Microsoft Press

ISBN: 978-0735619678