TDD

Test Driven Development

Test-Driven Development (TDD) é uma prática de desenvolvimento de software onde testes unitários automatizados são escritos de forma incremental antes mesmo do código de produção ser desenvolvido (BECK, 2003). O TDD ganhou popularidade quando foi definido por Kent Beck como uma parte essencial da **Extreme Programming (XP)** (BECK, 2010), que é uma metodologia ágil de desenvolvimento de software focada na aplicação de técnicas de programação, comunicação clara e trabalho em equipe. Essa abordagem, também chamada de Test-First, traz uma visão completamente oposta à abordagem tradicional (Test-Last), onde o **código de produção é escrito de acordo com as especificações do projeto** e, normalmente, somente depois de grande parte do código de produção ser escrito, os casos de teste começam a ser desenvolvidos.

Os resultados que o TDD proporciona, o que se encontra são análises e resultados divergentes em relação ao seu custo-benefício, sendo que em muitos casos são considerados inconclusivos, mesmo que sejam executados experimentos em ambientes diversos. Portanto, está revisão tem o objetivo de entender o atual cenário do TDD, considerando ambientes industriais e acadêmicos, com enfoque em seu custo-benefício, ou seja, busca-se analisar a qualidade do software desenvolvido com TDD e a produtividade de equipes que fizeram uso do TDD, para, assim, obter um comparativo entre o custo-benefício do desenvolvimento de software que faz uso do TDD e do TLD.

1 METODOLOGIA

O método de pesquisa utilizado neste estudo foi uma revisão bibliográfica sistemática (RBS), que consiste em um estudo empírico no qual uma hipótese ou pergunta de pesquisa é abordada para agrupar indicadores de estudos primários por meio de um processo sistemático de pesquisa e extração de dados.

- **2.1 – Processo de Pesquisa**

O processo de pesquisa teve seu início a partir da definição do protocolo de pesquisa. Toda a pesquisa foi estruturada e documentada no software Parsifal (PARSIFAL. . . , s.d.), uma ferramenta online projetada para auxiliar a realização de revisões sistemáticas da literatura no contexto da engenharia de software Após a definição do protocolo, foram elaboradas as perguntas de pesquisa, as quais exercem papel fundamental em uma revisão sistemática, visto que as mesmas guiam todo o processo. Abaixo estão as perguntas de pesquisa do presente estudo:

P1 - Os custos envolvidos na aplicação do Test-Driven Development compensam seus benefícios?

P2 - Quando o Test-Driven Development for aplicado, a quantidade de defeitos encontrados no software nas fases mais avançadas do desenvolvimento será menor?

P3 - Quando o Test-Driven Development for aplicado, muitos problemas serão detectados e removidos ainda nas fases iniciais, muito provavelmente, pelos próprios desenvolvedores?

Logo após a elaboração das perguntas de pesquisa, os estudos primários foram filtrados em repositórios online. Após a busca nos repositórios digitais, os artigos foram selecionados e classificados nas seguintes categorias

1 - Experimento sobre TDD: aqui se encaixam artigos que relatam experimentos conduzidos e analisados pelo(s) próprio(s) autor(es);

2 - Experimento de terceiros e experimento próprio: engloba artigos que analisam experimentos de outros autores e, também, descrevem experimento(s) do(s) próprio(s) autor(es);

3 - Revisão Bibliográfica Sistemática sobre TDD: consiste em estudos que são revisões sistemáticas, gerando grande embasamento para o presente estudo;

4 - Test-Driven Development: estudos que são descritivos e detalham a prática do TDD durante o processo de desenvolvimento de software.

2 ANÁLISE DOS RESULTADOS

Para cada estudo foram analisados dois pontos principais, que serviram como base para responder às perguntas de pesquisa: a qualidade do software e a produtividade dos desenvolvedores.

- **3.1 – Qualidade**

Em relação à variável qualidade, nenhum estudo apontou que o TDD diminua a qualidade do software quando comparado com o desenvolvimento TLD. Entre os estudos analisados, 42,86% apontaram um aumento na qualidade (seja ela interna ou externa); 28,57% não apontaram nenhuma melhoria na qualidade; e 28,57% são inconclusivos em relação à variável qualidade. Com o uso do TDD, a preocupação com a escrita de testes se torna maior, visto que é requerido a criação de testes unitários que desenvolvem gradualmente pequenas partes da funcionalidade até que um recurso seja totalmente implementado A maioria dos estudos selecionados, apontam que o

TDD aumenta a qualidade de modo geral, conforme previa Kent Beck quando apresentou a metodologia XP, que faz uso do TDD.

Pode-se observar que o desenvolvimento de software com TDD, aparentemente, mesmo considerando as limitações do estudo, produz código com qualidade superior quando comparado com o código desenvolvido com uma prática mais tradicional, por exemplo, o TLD, utilizando um conjunto de casos de teste de caixa preta.

- **3.2 – Produtividade**

Analisando os estudos selecionados, levando em consideração a variável produtividade, pode-senotar que o TDD, de modo geral, não apresenta aumento na produtividade dos desenvolvedores de software onde o grupo Test-Last, segundo os autores do experimento, parece ser mais produtivo que o Test-First (TDD). Nesta revisão, somente um estudo apontou que quando o TDD foi aplicado a produtividade aumenta (GUPTA; JALOTE, 2007), enquanto que 21,43% foram classificados como "Sem Aumento". Seguindo a análise, 28,57% dos estudos apontaram que quando o TDD foi utilizado a produtividade diminui e, por fim, 42,86% dos estudos são inconclusivos.

3 CONCLUSÃO

Esta revisão bibliográfica sistemática foi realizada com o objetivo de identificar, selecionar e analisar estudos empíricos que abordam o uso da prática TDD no desenvolvimento de software, sempre tendo como foco avaliar os efeitos originados na qualidade do software desenvolvido e na produtividade dos desenvolvedores. A maior parte dos estudos presentes nesta revisão (42,86%) indicam uma melhoria na qualidade do software quando o TDD foi utilizado, sendo que nenhum estudo apontou uma diminuição na qualidade. É importante ressaltar que os resultados da análise desses artigos mostraram que, em quase metade deles, não houve um resultado claro sobre a relação entre custo e benefício do TDD, sendo assim, seus resultados foram inconclusivos. Apenas 7% dos estudos apontaram o uso do TDD como positivo de modo geral, ou seja, neste estudo a qualidade aumentou e a produtividade também, não havendo necessidade de empenhar mais tempo durante o desenvolvimento do experimento. Estes mesmos 7%, representam, também, os estudos que descrevem o TDD como uma prática negativa, ou seja, não aumenta a qualidade e eleva o tempo gasto durante o desenvolvimento do software. Esta revisão buscou entender a relação de custo-benefício do TDD, mas não chegou a ser conclusiva, sendo assim, este estudo não pode afirmar ou negar que o uso da prática realmente traz melhorias compensando seus custos. Todavia, pode-se afirmar que o TDD traz mais qualidade para o software e, principalmente, para os testes unitários que são desenvolvidos pelo próprio programador.

4 NA PRÁTICA

- **4.1 – Tudo que se precisa saber do TDD**

TDD é a sigla para Test Driven Development, que em português significa Desenvolvimento Orientado por Testes. Esse é um método de desenvolvimento muito comum atualmente. Ele se baseia na aplicação de pequenos ciclos de repetições. Em cada um deles, um teste é aplicado**. O objetivo é desenvolver uma função que permita que esse teste tenha um resultado positivo, uu seja, a função está pronta para ser implementada.**

- **4.2 – Como é um TDD?**

Um TDD é aplicado da seguinte forma. Primeiramente, os desenvolvedores criam um teste que irá falhar de qualquer forma. Afinal de contas, ainda não existe um recurso para ele. Em seguida o time desenvolve a função que deve fazer o teste passar e então reaplicar ele. Se o resultado é positivo, os profissionais implantam o novo recurso no código, e então partem para o desenvolvimento de um novo teste. **Esse ciclo é repetido até o final do projeto**, quando o programa ou aplicativo é finalizado.

- **4.3 – Vantagens de usar o TDD no dia a dia do desenvolvimento**

O TDD é uma excelente alternativa para quem atua de forma autônoma, mas também para quem trabalha em equipes de desenvolvimento. Isso porque, você consegue ter um feedback rápido. Se feita a função, mas não obteve um resultado positivo no teste, ela pode ser modificada logo em seguida. Mas além disso, esse método também traz uma série de outras vantagens, tais como:

  - é possível focar em problemas específicos de desenvolvimento;
  - Crie códigos mais limpos e simples, e que também são fáceis de refatorar;
  - Mais facilidade para corrigir bugs por meio dos ciclos de desenvolvimento;
  - Flexibilidade no código;
  - Mais produtividade;
  - Foco na resolução de problemas;
  - Índice menor de retrabalho.

Por meio do TDD, você poderá se tornar mais produtivo e criar códigos mais limpos (clean code) para o cliente.

- **4.4 – Porque usar o método TDD?**

O método TDD é amplamente utilizado por diversos profissionais e empresas de desenvolvimento. Isso porque, ele permite que você faça testes rápidos e construa códigos em pedaços, o que melhora o resultado final. Além disso, existem outros motivos do porquê esse tipo de desenvolvimento é tão utilizado, tais como:

  - Veja de forma rápida o que está comprometendo o seu código e quais funções já estão funcionando plenamente;
  - Tenha mais segurança no seu desenvolvimento;
  - Não desperdice tempo programando o código inteiro para só então descobrir uma falha;
  - Consiga entregar um feedback periódico para o cliente, uma vez que é possível apresentar os resultados de cada ciclo;
  - Desenvolva funções mais completas;
  - Facilite a manutenção no futuro, uma vez que o código será divido em pedaços;
  - Ofereça soluções mais completas e de qualidade, mas gastando menos tempo no desenvolvimento;
  - Desenvolva funções mais completas;
  - Mais organização em todo o processo de desenvolvimento.

O TDD pode ser um método muito útil para os desenvolvedores. Isso porque, é através dele que você poderá testar cada uma das funções, evitando programar a solução inteira, para só então descobrir que determinadas funções não estão funcionando corretamente. Por isso, vale à pena implementá-lo no seu dia a dia.

- **4.5 - Como Escrever Testes Unitários (Usando as Melhores Práticas)**

Criar testes unitários é o mesmo que desenvolver qualquer código, mas há uma diferença. Você cria um código de aplicação funcional para resolver um problema para seus clientes. Você cria testes unitários para resolver os problemas que surgem no desenvolvimento do código da aplicação, adivinhe o que isso significa? Sim, você é seu próprio cliente! E, claro, você quer tornar sua vida o mais fácil possível. Melhores práticas:

  - Dê nomes de seus métodos de teste que o ajudem a entender os requisitos do código que você está testando sem ter que procurar

em outro lugar. Escolha um dos vários modelos experimentados e testados que existem para isso e mantenha-se fiel a ele.

- Certifique-se de que um teste só tenha sucesso porque o código que ele testa está correto. Da mesma forma, assegure-se de que um teste só falha porque o código que ele testa está incorreto. Qualquer outra razão para o sucesso ou fracasso é enganar a si mesmo ou perseguir um buraco de coelho.

- Faça um esforço para criar mensagens curtas e significativas de falha que incluam parâmetros de teste relevantes. Há poucas coisas mais frustrantes do que “Esperado 5, mas encontrado 7” e ter que caçar o valor dos parâmetros para o método em teste.
- Certifique-se de que cada teste possa produzir os resultados corretos (sucesso ou falha) mesmo quando for o único teste que você executar.

Quando um teste depende de como outro teste mudou o ambiente (valores de variáveis, conteúdo de coleções, etc.), você terá dificuldade em acompanhar as condições iniciais para cada teste. Mais importante ainda, quando você obtiver resultados inesperados, você sempre se perguntará se as condições do teste ou o código de produção os causaram.

- Não otimizar a configuração do teste além do arquivo de origem de uma classe de teste. Mantenha uma classe de teste como um mundo para si mesma, independente das outras.

Use métodos de configuração e classes de utilidade aninhadas, se desejar, mas evite hierarquias de classes de teste e classes de utilidade geral. Isso evitará que você tenha que caçar através de diversas classes base ou unidades de classe de utilidade para encontrar os valores que um teste utiliza.

- Siga o padrão Arrange, Act, Assert. Marque cada parte com um comentário. A razão para seguir este padrão está no próximo ponto.

- Cada teste deve lidar apenas com:
  - Um arrange — Um cenário a ser testado (um “dado”);
  - Uma action — Um método para testar (um “quando”);
  - Um assert — Uma chamada para um método de verificação (um “então”).

Quando uma ação deve ter mais de um efeito, teste cada um deles em um método diferente. Quando você se sentir tentado a usar mais de uma afirmação, pergunte-se se você está afirmando o fato mais significativo.

- Quando você fica frustrado com o quão difícil é organizar um teste, ouça essa dor. Tome a dica de que seu design de código precisa ser melhorado.

Resistir ao impulso de usar gambiarras e “automágica”. As gambiarras só abordam os sintomas, e a automágica reduz a transparência necessária para descobrir por que um teste que deveria ser bem-sucedido é reprovado ou, pior ainda, por que um teste que deveria ser reprovado é bem-sucedido.

Enfrente o que está causando a dor usando os meios mais diretos possíveis. Com bastante frequência, você pode fazer mudanças simples para tornar pelo menos parte de uma classe mais testável. “Working effectively with Legacy Code” de Michael Feathers é um excelente recurso para isso.

***Armadilhas Comuns em Testes Unitários**

Erros simples podem te fazer tropeçar seriamente. Pior ainda, eles podem te acalmar com uma falsa sensação de segurança. Estes são os erros que você quer evitar.

- Escrever testes sem afirmações.

Tal teste nunca falhará! Se não houver problema porque você está meramente verificando se não há exceções, faça isso explicitamente com uma afirmação e uma mensagem apropriada.

- Escrever mais de um teste de cada vez.

É uma indicação de que você está testando após o fato (em vez de uma abordagem de test-first), e está criando dores de cabeça para você mesmo ao acompanhar quais testes estão completos e devem ser bem sucedidos e quais testes falham por estarem incompletos.

- Não começar com um teste reprovado.

Quando você não começa com um teste reprovado, você não saberá se ele é bem sucedido porque você tem um erro em seu teste ou porque o código funcional está correto.

- Não executar testes frequentemente.

Idealmente, você deseja executar todos os testes unitários em cada etapa de um ciclo vermelho-verde-refatorar, quer você use isso com ou sem uma abordagem de test-first, como TDD e BDD.

- Escrever testes lentos.

Os testes lentos interrompem seu fluxo. A propósito, não há problema em usar um framework de testes unitários (veja abaixo) para escrever testes (mais) lentos, mas eles não são testes unitários, e você quer mantê-los em um conjunto de testes separado.

- Tornar os testes dependentes de seu ambiente de testes.

- **4.6 – Ferramentas e Técnicas de Testes Unitários**

Frameworks de Testes Unitários

Os frameworks de testes unitários fornecem tudo o que você precisa para criar testes unitários

- Atributos de código para que um test runner (veja abaixo) possa descobrir e executar seus testes.
- Atributos de código para fornecer aos seus testes os dados do teste.
- Classes e métodos para verificar os efeitos da ação que você deseja testar.

Você pode encontrar um ou mais frameworks de teste unitário para quase todas as linguagens de programação por aí.

- Teste Unitário Runners

Teste unitário runners descobrem os testes em seu código de teste automaticamente, executam todos os testes ou uma seleção específica, e depois reportam os resultados do teste. Eles vêm como extensões de IDE e como utilitários autônomos de linha de comando. Você pode usar estes últimos em scripts de compilação, de modo que as builds de integração falhem quando um merge quebra o código existente. Os frameworks de teste unitário têm seus próprios runners, mas você também pode encontrar runners dedicados que podem descobrir e executar testes escritos usando múltiplos frameworks.

- Bibliotecas de Mock

Bibliotecas de mock permitem facilmente criar dublês e imitações para testes. Você os utiliza para fornecer uma classe em teste com instâncias de suas classes colaboradoras. Desta forma, você pode facilmente personalizar o comportamento dos colaboradores de acordo com as necessidades de seu teste.

- Injeção de Dependência / Inversão de Controle

Inversão de Controle, ou Injeção de Dependência, é um padrão de projeto que você usa para romper laços fortes entre as classes. Em vez da classe A instanciar uma classe B5, você a fornece com uma instância do ancestral mais abstrato do B5 que dá à A os métodos e propriedades que ela precisa. Facilita o teste unitário de uma classe com colaboradores, pois é muito mais fácil lhe fornecer imitações.

- Recursos de Testes Unitários
    - Growing Object-Oriented Software, Guided by Tests by Steve Freeman
    - The Art of Unit Testing: With Examples in c# by Roy Osherove
    - Working effectively with Legacy Code by Michael C. Feathers
    - Refactoring: Improving the Design of Existing Code by Martin Fowler
    - Clean Code: A Handbook of Agile Software Craftsmanship by Robert C. Martin (Uncle Bob)