Universidade Católica do Salvador
Bacharelado em Engenharia de Software

Gabryela Santana Barros
Lielson R. Pereira Junior

Análise experimental entre as técnicas TDD e Test-Last no processo de manutenção corretiva de software

Salvador
2019

Gabryela Santana Barros
Lielson R. Pereira Junior

Análise experimental entre as técnicas TDD e Test-Last no processo de manutenção corretiva de software

Trabalho de Conclusão de Curso apresentado à Universidade Católica do Salvador como parte dos requisitos necessários para a obtenção do Título de Bacharel em Engenharia de Software.
Orientador: Prof. Me. André Wyzykowski

Universidade Católica do Salvador
Bacharelado em Engenharia de Software

Salvador
2019

Gabryela Santana Barros
Lielson R. Pereira Junior

Análise experimental entre as técnicas TDD e Test-Last no processo de manutenção corretiva de software

Trabalho de Conclusão de Curso apresentado à Universidade Católica do Salvador como requisito parcial para a obtenção do título de Bacharel em Engenharia de Software.

Comissão Examinadora

Prof. Me. André Wyzykowski
Universidade Católica do Salvador
Orientador

Prof. Me. Marcelo Indio dos Reis
Universidade Católica do Salvador

Prof. Antônio Cláudio Pedreira Neiva
Universidade Católica do Salvador

Salvador, 9 de julho de 2019

Dedicamos este trabalho a nós mesmos e a nossa resiliência, pois sem ela,
não saberíamos se ainda estaríamos juntos como um casal.

Agradecimentos

Em primeiro lugar, agradecemos a nossa família, que sempre esteve presente e nos apoiou em todos os momentos, não medindo esforços para que chegássemos a esta etapa de nossas vidas.

Agradecemos também ao professor e coordenador do curso, Osvaldo Requião Melo, pelo convívio, apoio, compreensão e amizade.

À todos os professores do curso de Engenharia de Software da Universidade Católica do Salvador, que foram tão importantes em todo o nosso processo de graduação, nossa sincera gratidão. Em especial ao Prof. Me. Marcelo Indio e ao Prof. Mário Jorge Pereira, responsáveis por impulsionar a execução deste trabalho.

E por último, mas não menos importante, agradecemos também ao nosso orientador, Prof. Me. André Wyzykowiski, que por muitas vezes puxou nossas orelhas e quintuplicou o nosso trabalho. No entanto, mesmo com todas as adversidades da vida, sempre esteve disponível e deu o melhor de si para a conclusão deste trabalho.

"Do the simplest thing that could possibly work"
(Kent Beck)

Resumo

No contexto de desenvolvimento de software, é possível encontrar várias abordagens. A técnica teste depois do desenvolvimento (TLD) é o processo mais comum na prática de construção de um software. O desenvolvimento orientado a testes (TDD) é uma técnica em que os testes são escritos antes da implementação de uma funcionalidade.

Devido ao aprimoramento dos processos de software e com a influência do movimento ágil, o TDD ganhou notoriedade e atualmente é bastante difundido na comunidade. Muito se sabe sobre seus benefícios para o processo de desenvolvimento de software, mas há poucas evidências de seu domínio em relação a manutenção corretiva de software. Dito isto, o objetivo desta pesquisa é, através de experimentos com estudantes de níveis variados da área de computação, testificar a eficiência do TDD na manutenção corretiva de software em comparação com o TLD.

O trabalho foi dividido em três etapas. A primeira fase teve como objetivo a criação de um *dataset* contendo códigos desenvolvidos com as técnicas de TDD e TLD. Na segunda fase foram selecionados um total de dez códigos válidos para os experimentos seguintes: cinco códigos utilizando TDD e cinco códigos utilizando TLD. Na terceira fase, os códigos selecionados na fase anterior passaram por um processo de manutenção, onde deveriam ser detectados possíveis erros de implementação.

Ao final deste estudo é possível analisar o resultado de duas perspectivas: desenvolvimento e manutenção. No desenvolvimento, percebeu-se que os códigos construídos com TDD possuem menor tamanho e maior qualidade quando comparado ao TLD. Na manutenção, percebeu-se que os códigos construídos com TDD, possuíam um número maior de bugs solucionados do que códigos construídos com TLD. No entanto, os resultados na economia de tempo para correção de bugs foram inconclusivos.

Palavras-Chave: 1. TDD. 2. Test-Last. 3. Manutenção de Software. 4. Experimento.

Abstract

In the context of software development, It is possible to find any approaches. The test last development technique (TLD) is the most common process in software building practice. The test-driven development (TDD) is a technique in which tests are applied before the implementation of a feature. Due to software process improvement and with the influence of the agile movement, the TDD gained notoriety and nowadays is very widespread in the community. It is well known about benefits of software development, but there are few sources of information about corrective maintenance.

This work was done in three steps. The first step had the objective of creating a dataset of codes made using TDD and TLD aprroach. For the second step a total of ten valid codes were selected during this phase, five codebases using TLD approach and five codebases using TDD approach. The third step using code that were create on previous phase, were submitted for maintenance, where possible implmentation errors should be detected.

The objective of this research is, through experiments with students of various levels in the area of computing, is to check the efficiency of TDD in software maintenance compared to TLD. During development session, It is possible to notice that code built with TDD has less size and more quality comparing to TLD. In maintenance, it was observed that the codes built with TDD, had a larger number of bugs solved than the codes built with TLD. However, results of the time-saving for fixing bugs were inconclusive.

Keywords: 1. TDD. 2. Test-Last. 3. Software Maintenance. 4. Experiment.

Lista de figuras

Figura 1 – Fases do Ciclo de Vida do Software 21
Figura 2 – Opinião de Programadores Iniciantes por Preferência Entre as Abordagens TDD e TLD (JANZEN; SAIEDIAN, 2007) 28
Figura 3 – Distribuição do número de pessoas por semestre durante a fase de desenvolvimento 30
Figura 4 – Distribuição do número de pessoas por semestre durante a fase de manutenção 37
Figura 5 – Análise comparativa TDD x TLD no processo de manutenção corretiva de software 41
Figura 6 – Análise comparativa TDD x TLD dos itens corrigidos 42

Lista de tabelas

Tabela 1 – Currículo e Noção Conceitual de Dificuldade (BUTLER; MORGAN, 2007) 31
Tabela 2 – Análise dos códigos desenvolvidos com a técnica TLD 33
Tabela 3 – Análise dos códigos desenvolvidos com a técnica TDD 34
Tabela 4 – Itens entregues para os códigos que utilizaram TDD 35
Tabela 5 – Itens entregues para os códigos que utilizaram TLD 35
Tabela 6 – Quantidade de erros encontrados em cada projeto 36
Tabela 7 – Média de Erros Encontrados por Tipo de Código 36
Tabela 8 – Tempo médio de reconhecimento, análise e correção de erros utilizando a metodologia TLD 40
Tabela 9 – Tempo médio de reconhecimento, análise e correção de erros utilizando a metodologia TDD 41

Lista de Siglas e Abreviaturas

UCSAL	*Universidade Católica do Salvador*
TDD	*Test Driven Development*
TLD	*Test Last Development*
BES	*Test Bacharelado em Engenharia de Software*
ADS	*Tecnólogo em Análise e Desenvolvimento de Sistemas*

Sumário

1 INTRODUÇÃO 16
1.1 Aplicabilidade e Motivação 17
1.2 Objetivos 17
1.3 Estruturação do Documento 18

2 FUNDAMENTAÇÃO TEÓRICA 19
2.1 Ciclo de Vida do Software 19
2.1.1 Testes de Software 21
2.1.1.1 Testes de Unidade 23
2.1.1.2 Test Last Development (TLD) 23
2.1.1.3 Test Driven Development (TDD) 23
2.1.2 Trabalhos Relacionados 24
2.2 Manutenção de Software 25
2.2.1 Manutenção Corretiva 26
2.2.1.1 Métricas de qualidade 26

3 DESENVOLVIMENTO 27
3.1 Planejamento e Execução do Estudo 28
3.2 Fase de Desenvolvimento 29
3.2.1 Participantes da Fase de Desenvolvimento 29
3.2.2 Problemas Propostos na Fase de Desenvolvimento 30
3.2.3 Execução da Fase de Desenvolvimento 32
3.3 Fase de Análise e Seleção 32
3.3.1 Análise 33
3.3.2 Seleção 34
3.3.2.1 Contabilização dos Defeitos Encontrados 35
3.4 Fase de Manutenção 36
3.4.1 Participantes da Fase de Manutenção 36
3.4.2 Problemas Propostos na Fase de Manutenção 37
3.4.3 Execução da Fase de Manutenção 37

4 RESULTADOS 39
4.1 Inspeção dos Códigos-Fonte 39
4.2 Métricas 40
4.3 Discussões 42

5 CONCLUSÕES 44
5.1 Sugestão Para Trabalhos Futuros 44

REFERÊNCIAS BIBLIOGRÁFICAS 46

A FORMULÁRIO PRÉ EXPERIMENTO 49

B FORMULÁRIO PÓS DESENVOLVIMENTO (TDD) 50

C FORMULÁRIO PÓS DESENVOLVIMENTO (TLD) 51

D FORMULÁRIO PÓS DESENVOLVIMENTO (TLD) 52

E PROBLEMAS PROPOSTOS 53

1 Introdução

No contexto de desenvolvimento de software, uma das premissas básicas é que o produto, ou serviço, atenda todos os objetivos que levaram à sua construção, não deixando de lado aspectos como a qualidade do resultado final. Diante disso, os desenvolvedores de software buscam adotar tecnologias e práticas que prometem maior produção utilizando menos recursos.

No entanto, por mais que os desenvolvedores de software busquem alcançar a qualidade do que está sendo produzido, existe a possibilidade de conter erros no código implementado oriundos de equívocos humanos, como por exemplo a falha de comunicação , erros na análise e regras de negócio, problemas com a codificação, dentre outros.

A fim de reduzir os erros, são previstos os testes de software, com a finalidade de verificar se o comportamento de um sistema está de acordo com o que foi planejado e, a partir disso, identificar possíveis falhas em seu funcionamento.

O livro (PRESSMAN, 2010) aborda sobre as estratégias de testes mais utilizadas no cotidiano das equipes de software. De um lado, a execução dos testes ocorre em todo o sistema logo após a finalização da construção do software para identificar possíveis defeitos. De outro lado, os testes são executados constantemente, sempre que uma parte do sistema for construída. E, em meio a isso tudo, a estratégia de teste escolhida por muitas equipes de software está entre os dois extremos. Os testes são encarados de forma incremental e evolutivos, começando pelas camadas mais básicas do software até o seu fluxo completo.

No modelo tradicional, os testes de software acontecem após a fase de implementação e, geralmente, são realizados por alguém que não teve contato com os artefatos gerados no processo de desenvolvimento. Os testes percorrem o código com o objetivo de encontrar possíveis erros existentes entre a codificação e os resultados esperados nas documentações de requisitos. Essa metodologia de trabalho é defina como *Test Last Development* (TLD).

Test-Last é a ação de escrever testes depois da implementação ser realizada, com o propósito de verificar a funcionalidade do código desenvolvido. Desta forma, é possível detectar falhas ou até mesmo previni-las (GELPERIN; HETZEL, 1988). Estes testes são usados pelo desenvolvedor para corrigir o item codificado que não está funcionando até que nenhum outro erro seja apontado pelos testes (THOMAS, 2014).

É possível que o processo de desenvolvimento inicie a partir dos testes. Os testes são escritos antes mesmo da implementação do software na abordagem de TDD - *Test Driven Development*; em português brasileiro, Desenvolvimento Orientado a Testes (BECK, 2003). De modo geral, a estratégia do TDD exige que os testes automatizados sejam escritos antes mesmo do desenvolvimento do código, em iterações pequenas e rápidas (Janzen;

Saiedian, 2005).

1.1 Aplicabilidade e Motivação

Verificar a relação da utilização do TDD no meio acadêmico analisando seu processo de aceitação pelos alunos de graduação no curso de Bacharelado em Engenharia de Software (BES) e no curso Tecnólogo em Análise e Desenvolvimento de Sistemas (ADS) da Universidade Católica do Salvador.

A crescente popularidade do TDD no processo de aprendizagem, com base na metodologia de ensino de professores de graduação. A influência do TDD no processo de manutenção corretiva é mencionado na literatura, mas pouco explorado em termos quantitativos. O grande desafio dos desenvolvedores de software é produzir códigos com a menor quantidade de erros possíveis. Quaisquer técnicas que prometam alcançar esse objetivo precisam ser analisadas pela comunidade de software.

Diante desse cenário, o TDD é frequentemente mencionado como uma possível técnica que auxilia no processo de construção de software à produzir mais artefatos de software com maior qualidade.

Segundo (TOSUN et al., 2018) conclui que os códigos produzidos com a abordagem TDD possuem maior cobertura de testes e é suscetível a gerar um número menor de defeitos. Além do mais, em seus estudos, os autores afirmam que o TDD aumenta a produtividade das pessoas.

Já (GEORGE; WILLIAMS, 2003) afirma que apesar do TDD impulsionar códigos com maior qualidade, a produtividade dos desenvolvedores pode cair cerca de 16% quando confrontado com metodologias de desenvolvimento mais tradicionais.

Verificar qual a influência do TDD no processo de manutenção de software, a saber se o código produzido com essa técnica tem real relevância no processo de manutenção corretiva. Obter a informação de se os erros são mais fáceis de serem localizados, corrigidos e, consequentemente, validar a produtividade desta técnica perante esses efeitos.

1.2 Objetivos

O objetivo geral deste trabalho é comparar as diferenças entre as técnicas de desenvolvimento orientado a testes e testes após o desenvolvimento, no âmbito da qualidade e produtividade de software, por meio da análise dos resultados obtidos nos experimentos realizados em turmas de diferentes níveis de conhecimento dos cursos de BES e ADS.

Objetivos específicos:

1. Definir critérios e métricas para comparação dessas duas técnicas.

2. Realizar experimentos visando a obtenção de códigos construídos com as técnicas TDD e TLD para serem utilizados na manutenção corretiva.

3. Realizar experimentos visando obter uma análise comparativa com ambas as técnicas no processo de manutenção corretiva.

1.3 Estruturação do Documento

No Capítulo 2, são apresentados os fundamentos teóricos da pesquisa, que são conceitos relevantes ter em mente para o completo entendimento dos termos que serão abordados durante o capitulo de desenvolvimento. No Capítulo 3, é descrito o experimento aplicado, bem como a premissa dos problemas, os participantes e o que se deseja buscar através do experimento. Os resultados encontrados estão detalhados no Capítulo 4. No Capítulo 5 se encontra as conclusões como um todo do trabalho feito.

2 Fundamentação Teórica

Neste capítulo serão abordados alguns conceitos importantes para o entendimento completo deste trabalho. Para isso, foi realizada uma revisão da literatura dos conceitos correlacionados com o objeto de estudo deste trabalho. Nas seções seguintes são apresentadas as definições e características sobre o ciclo de vida de um software, tal como as especificações de suas fases, principalmente a fase de testes de software. Também são descritas as técnicas *Test Last Development* e *Test Driven Development*. Por fim, explana-se sobre o processo de manutenção de software, manutenção corretiva e as métricas de qualidade na manutenção de software.

2.1 Ciclo de Vida do Software

Um processo é um conjunto de passos ordenados logicamente, formados por atividades, métodos, práticas e transformações, com a finalidade de atingir uma meta (FILHO, 2008). Na engenharia de Software, o ciclo de vida está relacionado ao processo de como um software é planejado, construído e mantido.

Segundo (ISO/IEC/IEEE..., 2017) o ciclo de vida de um software é a estrutura de procedimentos, atividades e tarefas envolvidos no desenvolvimento, operação e manutenção de um produto de software, que abrange a vida do sistema desde a definição dos seus requisitos até a sua descontinuidade ou substituição.

Segundo (PRESSMAN, 2010), o ciclo de vida de um software é definido como *arcabouço de um processo*, onde um pequeno número de *atividades de arcabouço* são aplicáveis a todos os projetos de software, independente do seu tamanho ou complexidade. O resultado final do processo será muito diferente em cada caso, mas as fases permanecem as mesmas.

Semelhante a isso, (WAZLAWICK, 2013) afirma que embora a nomenclatura possa variar de um modelo para outro, o processo é dividido em fases. As fases contêm determinadas atividades com objetivos bem específicos, que são realizadas em um período de tempo pré-definido. Então, as fases são os segmentos que compõem um processo.

Desta forma, por se tratar de um modelo genérico, as fases do ciclo de vida de um software são estruturas previamente organizadas, nas quais se encaixam a maioria dos projetos de software. No entanto, é possível que haja diferença entre os modelos ciclo de vida do software. Isso está em concordância com (SOMMERVILLE, 2011) quando afirma que as atividades básicas do processo são organizadas de forma diferente a depender do modelo de processo de software. Isso provém das necessidades tanto do cliente quanto da equipe, e implica na mudança da ordem em que as fases ocorrem, a nomenclatura, a importância, o tempo, as atividades realizadas e os artefatos entregues em cada fase do

ciclo.

O ciclo de vida do software é composto por cinco fases. São elas: definição de requisitos, análise de projetos, modelagem, construção e implantação. Em cada fase do processos de software podem ocorrer pequenos ciclos. Ou seja, em um processo macro (a fase do ciclo de vida do software, por exemplo) pode haver subprocessos (uma etapa a ser cumprida dentro de uma fase do ciclo de vida do software, por exemplo). Ao final de cada fase do ciclo de vida podem ser gerados artefatos, que compõem uma parte do produto final (WAZLAWICK, 2013).

1. Primeira fase: A definição dos requisitos é a primeira fase a ser contemplada quando se produz algum elemento de software. Envolve alta comunicação e colaboração com os interessados. Sendo assim, se obtém o conhecimento necessário da situação atual e a identificação do problema para que possam elaborar uma proposta de solução. Desta maneira, deve ser documentado todo o comportamento do software a ser desenvolvido (BASSIL, 2012). Para dar continuidade às fases seguintes, é realizado o estudo de viabilidade e a análise dos custos-benefício.

2. Segunda fase: É definida como análise de projeto, onde é delimitada a estratégia de trabalho, bem como todo o planejamento do projeto. Contém o guia das tarefas técnicas, os prováveis riscos, os recursos necessários, o que será produzido e o cronograma de entregas (PRESSMAN, 2010).

3. Terceira fase: A modelagem especifica e prepara tudo que for relacionado a documentação, ferramentas e serviços, que serão utilizados tanto pelos desenvolvedores quanto pelo cliente. Inclui a criação de modelos que permitem compreender melhor os requisitos do software e o projeto que irá satisfazê-los (PRESSMAN, 2010).

4. Quarta fase: A fase de construção é quando o projeto é traduzido em um sistema. Essa atividade combina, geração de código e os testes necessários para revelar possíveis erros (PRESSMAN, 2010). Para isso, é necessário a utilização de tecnologias, ferramentas, bibliotecas, dentre outros recursos de software. O desenvolvimento da solução deve refletir a estrutura, regras e o comportamento descrito nas fases de definição de requisitos e análise de projeto (GOVARDHAN, 2010). Também é nessa fase do ciclo de vida do software que ocorrem os testes para verificação do que foi produzido na etapa de desenvolvimento. Os testes devem ser planejados e analisados para que todos os comportamentos do software sejam cobertos (VLIET, 2008). Caso isso não aconteça, pode não ter a efetividade almejada que a quarta fase do ciclo de vida do software necessita.

5. Quinta fase: A implantação é a ultima fase do ciclo de vida do software, também conhecida como a fase de homologação. É realizada a aceitação e validação por

meio dos *stakeholders* (partes interessadas). O software é entregue ao cliente, que o avalia e fornece as considerações ao produto com base na avaliação (PRESSMAN, 2010).

Figura 1 – Fases do Ciclo de Vida do Software

A figura Figura 1 demonstra visualmente as fases do clico de vida do software. Todas as fases do ciclo de vida do software precisam ser realizadas com o maior zelo possível, pois qualquer erro não resolvido em uma das fases pode causar prejuízos ao projeto e/ou ao cliente.

2.1.1 Testes de Software

Como visto na seção anterior, os testes de software é uma das fases que está contida no processo de ciclo de vida do software e, para chegar nessa etapa, uma série de atividades precisam ser cumpridas anteriormente.

De acordo com (BERTOLINO, 2007), os testes de software é um termo abrangente que engloba uma série de atividades um tanto que subjetivas. Os testes vão desde a avaliação do código realizados pelo próprio desenvolvedor à validação do cliente de um

sistema, ou até mesmo o monitoramento em execução. Nesses vários estágios, os casos de testes podem ser planejados com objetivos diferentes, como verificação dos requisitos do usuário, análise da conformidade das especificações, avaliação da robustez e performance do software, ou medir determinados atributos como desempenho ou usabilidade, estimar confiabilidade e assim por diante.

O teste de software é um processo, ou uma série de processos, projetado para garantir que um sistema faça o que foi projetado. Em outras palavras, o software não pode fazer nada que seja não intencional. O software deve ser previsível e consistente, sem surpresas para os usuários (MYERS COREY SANDLER, 2012).

Frequentemente, os testes são mencionados como verificação e validação. A verificação está relacionada ao fato de garantir que o software implementa uma função corretamente, Já a validação se refere a garantia que o produto de software está de acordo aos requisitos das partes interessadas (PRESSMAN, 2010).

A fase de teste é um subprocesso contido no processo de software (ciclo de vida). O processo de teste pode ser subdividido em etapas separadas: planejamento de teste, desenvolvimento de caso de teste, execução de casos de teste e avaliação dos resultados do teste (JORGENSEN, 2014).

No planejamento de testes é realizada a análise dos requisitos para verificar quais os cenários serão testados. São definidos os critérios de aceitação de uma funcionalidade e a delimitação do que será testado no momento. Feito isso, determina-se a estratégia de testes, visando cobrir todos os fluxos de uma ou mais funcionalidades do sistema.

No desenvolvimento de caso de testes realiza-se o levantamento técnico para a escrita dos testes. São definidas as ferramentas, bibliotecas e componentes, de modo geral. Além disso, é necessário delimitar o escopo a ser testado. Sendo assim, tudo o que estiver relacionado à configuração para execução dos casos de testes são realizados nessa etapa.

O plano estratégico, criado na etapa de planejamento, é posto em prática na etapa de execução. Essa atividade obtém como resultado a escrita, de fato, dos testes para a funcionalidade em questão. Nesse processo, há a possibilidade da utilização de ferramentas que automatizam a execução destes testes.

O resultado destas três etapas devem garantir o comportamento esperado da funcionalidade testada, no caso, deve avaliar se todas as ações do software estão amparadas por testes. Além de verificar os códigos, os testes servem como uma forma de documentação viva do software. Pois, a cada modificação feita ao longo do ciclo de desenvolvimento, os testes podem indicar quais itens sofreram alteração em sua conduta. Consequentemente, é possível adequar-se a estas mudanças.

Há várias maneiras de testar um sistema. A escolha do tipo de teste vai depender do objetivo a ser alcançado. Cada tipo de teste tem uma finalidade e para isso é necessário identificar quais níveis da aplicação devem e precisam ser testados. Conforme (LUO, 2001), os espectros de testes são divididos em quatro níveis, começando da camada mais

baixa até a mais alta: testes unitários, teste de integração, teste de sistema e teste de aceitação.

2.1.1.1 Testes de Unidade

Teste de unidade é um dos tipos de testes utilizados para averiguar se a codificação atende aos requisitos do sistema. Ele se trata da camada mais baixa e testa a unidade mais básica do software, que é a menor parte testável, por esse motivo é chamado de unidade.

Os testes de unidade, geralmente, concentram-se na lógica do programa dentro de um componente de software e na implementação correta da interface do componente (BERNER ROLAND WERBER, 2005).

Através dos testes unitários é possível garantir e averiguar todas as características do sistema, de forma mais isolada possível. E para isso, é necessário testar todas as operações associadas ao programa, definir e verificar o valor de todos os atribuídos a ele associado, bem como testar todos os estados possíveis simulando os eventos que causam as mudanças de estado, tal qual dito por (SOMMERVILLE, 2011).

2.1.1.2 Test Last Development (TLD)

Nos processos de construção de software, o modelo mais tradicional, que surgiu na década de 1970, é chamado de cascata. Nele o software é desenvolvido em uma série de fases sequenciais, ou seja , ao finalizar uma fase, inicia-se a próxima, sem a possibilidade de retornar a fase anterior (SOMMERVILLE, 2011). Elas foram configuradas desta maneira para que as fases anteriores permitissem que as fases posteriores fossem mais simples de serem concluídas (THOMAS, 2014).

O *Test Last Development* (TLD) é uma técnica que surgiu juntamente com o modelo cascata. Isso se deu pelo fato dos testes serem realizados apenas ao final desenvolvimento, e até então, este era o único método de teste que fazia sentido usar (THOMAS, 2014).

A prática do TLD consiste em criar testes somente após a etapa de desenvolvimento de uma funcionalidade ou de um módulo de um sistema. Na maioria dos casos, para testar um software, são utilizadas ferramentas de apoio; as bibliotecas de testes automatizados por exemplo. Uma vez escritos, os testes poderão ser executados quantas vezes for preciso.

2.1.1.3 Test Driven Development (TDD)

Com o surgimento das metodologias ágeis na década de 1990, houve a necessidade do aperfeiçoamento das técnicas de testar software. Já não fazia mais sentido os testes no final, pois as entregas eram constantes. Houve a adoção da abordagem de iterações para os testes também, não esperando para testar quando o produto ficasse completamente

pronto. Sendo assim, o TLD foi deixado de lado em favor de um estilo diferente conhecido como *test-frist*, traduzido para teste inicial. (FUCCI et al., 2016)

A ideia de testar primeiro foi implementada no desenvolvimento orientado a testes, tradução do *Test Driven Development* (TDD). Essa abordagem foi sugerida e difundida pelo *Extreme Programming* (XP) (BECK, 2005)

Test-first é a prática de escrever testes antes que o código tenha sido escrito e, em seguida, escrever o código para que os testes sejam aprovados. Deve-se pontuar que, como os testes são escritos antes do código de produção, *test-first* tende a ser fortemente vinculado a métodos de desenvolvimento e, portanto, os modelos mais comuns desta abordagem também incluem elementos de desenvolvimento.

O modelo mais conhecido da abordagem *test-first* é o TDD. Nele, os testes unitários são escritos antes dos códigos de produção, fazendo com que o desenvolvedor esteja completamente focado no comportamento correto do recurso pretendido desde o estágio inicial. Em outras palavras, é necessário escrever apenas o código suficiente para fazer o teste de unidade passar e, depois, refatorá-lo posteriormente (BECK, 2005).

2.1.2 Trabalhos Relacionados

O TDD é uma prática usada independentemente do contexto organizacional, seja acadêmico ou empresarial. Nos estudos realizados ao longo dos anos são demonstrados os efeitos positivos, ou não, da prática do TDD no processo de desenvolvimento de software mediante comparação à técnica TLD (BISSI; NETO; EMER, 2016).

Durante as pesquisas, os benefícios mais comuns que serviram como medida para comparação foram: a quantidade dos testes implementados e se previam todas as possibilidades de comportamento e a qualidade do produto final e a produtividade do desenvolvedor.

De acordo com Hakan Erdogmus, Maurizio Morisio, Marco Torchiano conduziram um experimento controlado com estudantes de graduação comparando o TDD com uma metodologia de desenvolvimento tradicional, onde a implementação dos testes acontece depois da etapa de desenvolvimento. No estudo, os participantes foram divididos em dois grupos, um utilizando TDD e outro utilizando *test-last*, onde foram propostas atividades a serem desenvolvidas de maneira incremental de modo que ao final de cada iteração o grupo que utilizou-se da técnica *test-last* deveria codificar os testes unitários. Os dados obtidos foram que o grupo que praticou o TDD codificaram mais casos de teste e tiveram mais produtividade, dado que implementaram mais itens em um tempo menor do que o grupo que praticou o *test-last*. Além disso, ambos os grupos obtiveram um aumento na qualidade já que testes unitários foram produzidos a fim de verificar o comportamento do software. (ERDOGMUS; MORISIO; TORCHIANO, 2005).

Segundo Yahya Rafique e Vojislav Misic em sua pesquisa (RAFIQUE; MISIC, 2013) realiza uma revisão de literatura de vinte e sete trabalhos que examinam a prática do

TDD, levando em consideração aspectos de qualidade do produto desenvolvido e da produtividade. Esses trabalhos foram divididos em dois grandes grupos, estudos primários que foram realizados na indústria e na academia, comparando os trabalhos destes dois grupos entre si. Por haver informações divergentes sobre estes assuntos, foi comparado esses trabalhos a fim de ter uma resposta mais concreta em respeito da produtividade e qualidade. Diante disso, percebeu-se um aumento da qualidade mas não era algo significativo. Já a produtividade notou-se que TDD possuiu uma queda. No entanto, a melhoria da qualidade do produto e a redução da produtividade do desenvolvedor é maior nos estudos industriais em comparação com os estudos realizados na academia. Também foi relatado que quanto maior for a experiência do desenvolvedor, tarefas maiores poderiam ser resolvidas utilizando TDD.

Conforme Matthias M. e Oliver Hagner, em seu trabalho (MüLLER; HAGNER, 2002) realizam experimentos comparando a técnica de desenvolvimento de *test-first* e técnica de programação mais tradicional (*test-last*). Como resultado obtém a informação que o *test-first* não acelera o processo de implementação, ou seja, irrelevante no quesito de aumento na produtividade. Porém, é possível que o *test-first* dê ao desenvolvedor maior clareza sobre o problema a ser resolvido.

Boby George e Laurie Williams em sua pesquisa (GEORGE; WILLIAMS, 2003) produziram uma série de experimentos comparando o TDD com processo de desenvolvimento cascata, que em outras palavras possui as mesmas características do TLD. Nesse estudo houve a participação de vinte e quatro pessoas, sendo elas profissionais da área de desenvolvimento. Os autores concluíram que TDD produziram código com mais qualidade, pois tiveram 18% a mais de assertividade em seus testes. Contudo as pessoas que utilizaram a técnica TDD foram menos produtivas pois levaram mais tempo para desenvolver a mesmas atividades das pessoas que utilizaram o TLD.

2.2 Manutenção de Software

O processo de manutenção de software ocorre após a finalização de todas as etapas do ciclo de vida do software. Ela consiste em manter, conservar, gerenciar e administrar um produto ou serviço de software. Em outras palavras, a modificação de um produto de software após a entrega com o objetivo de corrigir falhas, aprimorar o desempenho ou outras características ou adaptação à um cenário modificado (MAMONE, 1994). Desta forma, a manutenção está diretamente ligada à qualidade do software ao decorrer do tempo.

Apesar da manutenção ser considerada como uma continuação do desenvolvimento do software, há uma grande diferença que permite distinguir essas duas atividades. No processo de desenvolvimento, há mais alternativas de resoluções de problemas, isso claro, dentro das limitações do software. Já no processo de manutenção, além dessas limitações,

existem também as restrições ocasionadas por estar trabalhando dentro de um sistema existente. (GRUBB, 2003) Isso implica que a manutenção de software pode ser prejudicada devido ao rigor dado durante a sua construção.

A atividade de manutenção de software tem como objetivo a modificação de um produto existente, tornando-o mais aderente às necessidades em que foi designado. Como por exemplo, atualização de seus comportamentos para estarem condizentes com a atualidade, ou até mesmo na resolução de erros que não foram identificados na fase de desenvolvimento.

De acordo com (LIENTZ; SWANSON; TOMPKINS, 1978) a atividade de Manutenção de Software se desdobra em três grandes grupos: Manutenção Perfeccionista, Adaptativa e Corretiva.

A manutenção perfeccionista tem como foco as mudanças que foram originadas através de solicitações externas, melhorias de segurança, performance, dentre outros (MALL, 2018). São mudanças que não são diretamente ligadas a defeitos da aplicação, somente do seu amadurecimento.

Em alguns momentos durante a vida de um software será necessário adaptá-lo ao ambiente em que ele está inserido. Seja pela inserção de uma nova tecnologia, um novo sistema operacional, ou quando é preciso haver conexão com novas ferramentas(REZENDE, 2005). Além disso, às vezes se faz necessário adequar o software à proposta de negócio, e esta modificação é feita justamente para evitar futuros problemas.

Corretiva: Inclui todas as modificações necessárias para resolução de problemas que foram encontrados, que estão impedindo o funcionamento correto do software.

2.2.1 Manutenção Corretiva

A manutenção corretiva está relacionada às modificações realizadas em um software devido a defeitos oriundos da implementação. Um defeito pode resultar de erros de projeto, erros de entendimento de requisitos e erros de codificação (GRUBB, 2003).

Este tipo de manutenção se limita a correção de falhas já existentes no sistema que por algum motivo não satisfazem os requisitos necessários, e impedem que o software estejam de acordo com as especificações levantadas no inicio do projeto. E estes erros vão desde problemas de interpretação à falhas de codificação.

2.2.1.1 Métricas de qualidade

Nos trabalhos relacionados foi percebido que dois aspectos são levados em consideração para avaliação dos códigos-fonte. São eles: a produtividade e a qualidade.

De modo geral, a produtividade é medida a partir de dado um número de proposições a serem implementadas, o tempo que foi levado para realizá-las. Já a qualidade é medida com número de defeitos encontrados após a entrega do produto ao cliente.

3 Desenvolvimento

É possível notar que tanto no meio acadêmico quanto no meio industrial há a utilização das técnicas de TDD e TLD, mesmo este último fazendo parte de um modelo de desenvolvimento de software mais tradicional.

Realizar testes durante o desenvolvimento, ou ao final dele, pode ser uma escolha pessoal e/ou uma consequência da escolha do modelo de ciclo de vida utilizado, podendo ser feita pelo time ou empresa.

Há análises que comprovam as vantagens da prática do TDD sobre a prática do TLD, principalmente no que diz respeito à qualidade interna do software, que está diretamente relacionado ao código em si. (BISSI; NETO; EMER, 2016)

No entanto, quando na literatura há a comparação dessas técnicas de desenvolvimento, há uma face que causa discussões: qual metodologia utilizar para se obter mais produtividade?

De acordo com (BISSI; NETO; EMER, 2016), em seu trabalho realizado revisando os experimentos e estudos de caso produzidos até aquele ano, notou-se que a produtividade em cenário diferentes apresentam resultados diferentes. No meio acadêmico, utilizando a técnica de TDD a produtividade aumentou ou ficou constante. Porém, utilizando a técnica de TDD no meio industrial nenhum estudo mostrou aumento de produtividade, no máximo permaneceu contínua.

Para a realização do TDD, o pensamento de como codificar uma solução passa por um processo de mudança e necessita de maior disciplina do desenvolvedor. Não é atoa que desenvolvedores menos experientes optam por testar suas aplicações ao final do ciclo de desenvolvimento (JANZEN; SAIEDIAN, 2007), conforme a Figura 2.

Associado ao objetivo deste trabalho, foi proposto um experimento a fim de comparar as diferenças entre o TDD e o TLD no âmbito da qualidade e produtividade.

Nesta seção foram abordados quais os métodos utilizados para cumprir com os objetivos deste trabalho, tanto o principal quanto os específicos.

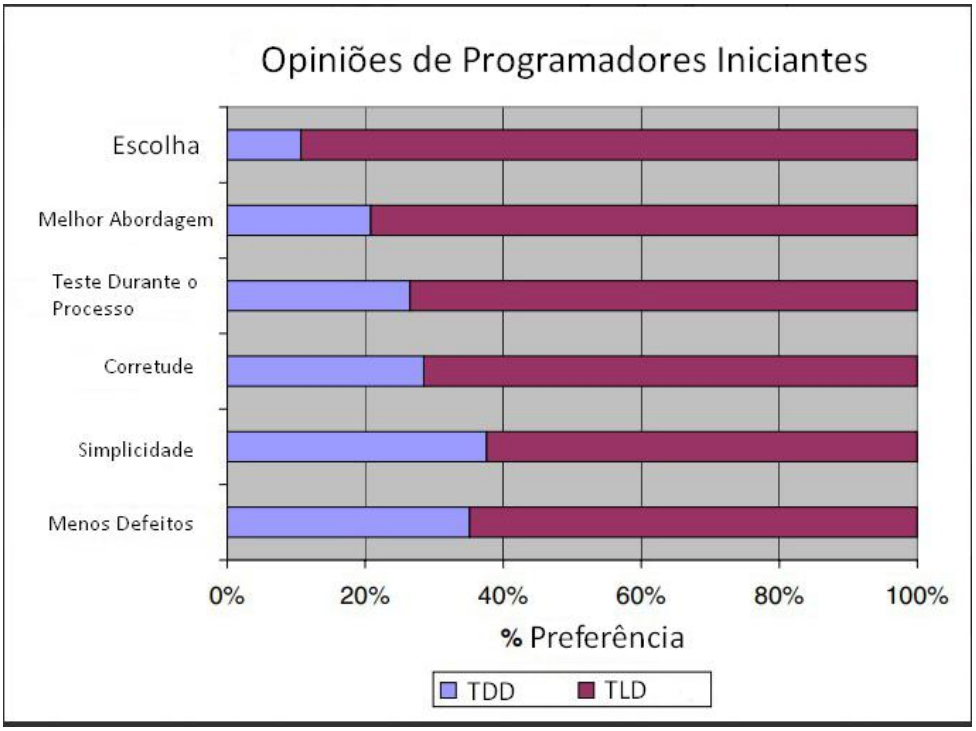

Figura 2 – Opinião de Programadores Iniciantes por Preferência Entre as Abordagens TDD e TLD (JANZEN; SAIEDIAN, 2007)

3.1 Planejamento e Execução do Estudo

A Engenharia de Software experimental é um ramo da informática que, com o uso da abordagem científica, visa apoiar o planejamento, execução e investigação de estudos relacionados à engenharia de software. De acordo com (TRAVASSOS DMYTRO GUROV, 2002), um experimento de software, dentre outras atividades, tem o objetivo de caracterizar, avaliar, prever, controlar e melhorar quaisquer atributos relacionados ao processo de desenvolvimento de software. Para este trabalho, a metodologia escolhida foi o experimento.

Para auxiliar e apoiar o processo de experimentações, houve a necessidade da elaboração de um *workshop* com os participantes do experimento para que todos tivessem proficiência dos mesmos conceitos. O *workshop*, com duração de aproximadamente uma hora, foi o contato inicial com os possíveis participantes do experimento. Nesse momento, por meio de um formulário pré-experimento, houve a possibilidade de conhecer os membros que decidiram fazer parte do trabalho.

Os assuntos abordados no *workshop* deram aos participantes uma visão geral sobre testes e testes automatizados, explicando como funciona todo o seu ciclo de vida e como algumas ferramentas de software podem auxiliar a execução dessa atividade. A exemplo disso, em dado momento, para por em prática os conceitos de testes, fez-se necessário a utilização do JUnit - *framework* que auxilia no processo de criação de testes automatizados

na linguagem de programação Java.

O intuito do *workshop* foi obter maior compreensão sobre as demais etapas do experimento, a fim de minimizar os ruídos ocasionados ao longo do processo. Além disso, com o embasamento técnico e teórico passado, os participantes puderam, de maneira prática, entender o funcionamento dos testes correlacionados às técnicas TLD e TDD.

Para este trabalho foi planejada a execução de um experimento dividido em três etapas: a fase de desenvolvimento, a de análise e seleção dos dados e a fase de manutenção.

Na fase de desenvolvimento foram elaborados experimentos com o propósito de, a partir da resolução de um conjunto de problemas propostos, coletar códigos desenvolvidos com a utilização da metodologia TDD e códigos desenvolvidos com a metodologia TLD.

Na fase de análise e seleção dos dados, os códigos previamente coletados passaram por uma inspeção e, de acordo com alguns critérios que foram definidos na seção de análise e seleção, posteriormente escolhidos para serem utilizados na etapa seguinte.

A última fase do experimento é focada na manutenção dos códigos selecionados. Nela, os participantes do experimento deveriam analisar a aplicação a procura de erros e, se houver a possibilidade, corrigi-los.

3.2 Fase de Desenvolvimento

A manutenção corretiva de um software está diretamente relacionada ao seu desenvolvimento. Isso porque os defeitos causados na implementação de um sistema, assim como as práticas utilizadas, que podem influenciar tanto positivamente quanto negativamente, são propagados na sua manutenção (GRUBB, 2003).

As pessoas têm percepções diferentes sobre um mesmo problema, isso implica diretamente em como será implementado uma determinada funcionalidade de um sistema, por exemplo. Por esse motivo, foi acoplado ao experimento a criação de uma base de dados contendo implementações do problema utilizando os testes após o desenvolvimento e implementações utilizado o desenvolvimento guiado a testes.

Nessa etapa do experimento, algumas pessoas foram selecionadas a participar desta fase e, além disso, desafiadas a implementar um programa de locadora de veículos. Durante a execução do experimento, uma série de restrições foram estabelecidas para que todos os envolvidos no processo tivessem as mesmas condições. O decurso desta fase do experimento está descrito nas seções seguintes.

3.2.1 Participantes da Fase de Desenvolvimento

O experimento foi elaborado com alunos do curso de BES e do curso de ADS, ambos pertencentes à Universidade Católica do Salvador (UCSal).

Optou-se por trabalhar com a maior quantidade possível de turmas de ambos os cursos, de modo que todos os níveis de conhecimento fossem contemplados, desde os mais iniciais até os mais avançados. O curso de BES é composto por oito semestres, já o curso de ADS é composto por quatro semestres. Havia, pelo menos, um aluno pertencente a turma de cada semestre de ambos os cursos.

Dada a dificuldade de encontrar pessoas disponíveis e interessadas em participar de estudos científicos, todos àqueles que se dispuseram, foram designados ao estudo. Ao todo, nesta fase do experimento, trinta e duas pessoas participaram. Elas foram divididas em dois grandes grupos: implementação com viés TDD e implementação com viés TLD.

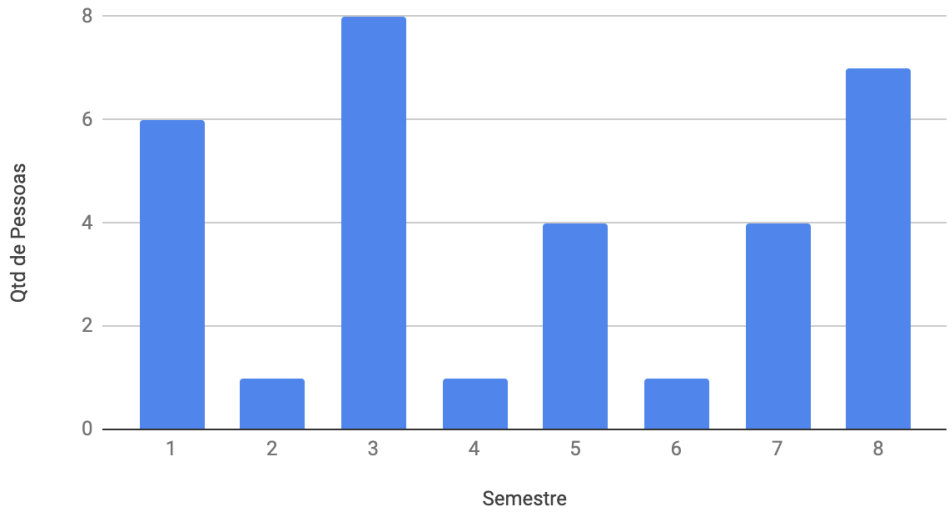

Figura 3 – Distribuição do número de pessoas por semestre durante a fase de desenvolvimento

A Figura 3 representa a distribuição dos participantes de acordo com seus respectivos semestres, contabilizando apenas as pessoas que participaram da fase de desenvolvimento.

Em cada turma que a fase de desenvolvimento do experimento foi aplicado, fez-se necessário a realização de sorteios para definir com qual abordagem (TDD ou TLD) cada pessoa iria trabalhar, de modo que proporcionasse balanceamento entre as metodologias. Devido à diversidade dos níveis de conhecimento entre os participantes, o sorteio das técnicas foi o meio encontrado para não enviesar o objeto de estudo. Além disso, serviu também como artifício para que os participantes não escolhessem qual técnica utilizar e, de alguma forma, influenciar o resultado da pesquisa.

3.2.2 Problemas Propostos na Fase de Desenvolvimento

Os participantes da fase de desenvolvimento do experimento foram desafiados a resolver uma lista de exercícios, que continham informações a respeito de uma locadora de veículos. O principal objetivo da tarefa era implementar o cálculo do valor final da

reserva de um veículo feito por um cliente. Ao todo, a lista de problemas continham treze proposições a serem codificadas.

Para que o sistema funcionasse de forma correta, uma série de regras deveriam ser atendidas. Dado uma determinada entrada, o sistema deveria processar e validar as informações com o proposito de retornar o valor esperado ao usuário, que no caso seria o valor total da reserva.

Os problemas foram pensados de forma que todos os participantes pudessem resolvê-los, independentemente da técnica utilizada e dos seus níveis de conhecimento. Isso porque, as questões possuíam níveis de dificuldades variados.

As regras da atividade eram interdependentes, de forma que todos os itens influenciavam no comportamento final da funcionalidade. No entanto, na maioria dos casos, não se fazia necessário seguir uma ordem de codificação dos itens apresentados.

Em relação a complexidade da atividade, os itens propostos exigiam dos participantes o conhecimento baixo-médio a respeito de programação, pois tratavam-se de problemas simples de lógica e operações numéricas. De acordo com (BUTLER; MORGAN, 2007), alunos da área de computação tendem a não possuir dificuldade com atividades de codificação que envolvem estruturas de repetição e de decisão.

A Tabela 1 expressa a percepção de alunos do curso da área de computação nos estudos realizados por (BUTLER; MORGAN, 2007). A primeira coluna indica qual o assunto abordado e a segunda coluna o seu respectivo o nível de dificuldade.

Currículo	Dificuldade Conceitual
Algoritmo	Alto
Sintaxe	Baixo
Variáveis	Médio
Decisões e Loops	Médio
Vetores	Médio
Métodos	Alto
Conceitos O.O	Alto
Programa no Geral e Projeto em OO	Alto
Teste e Debug	Médio

Tabela 1 – Currículo e Noção Conceitual de Dificuldade (BUTLER; MORGAN, 2007)

No entanto, algumas das proposições eram correlacionadas, o que implicava no aumento da dificuldade de sua resolução. Nesses casos, era preciso do desenvolvedor a análise mais crítica de como as funcionalidades se comportariam em conjunto, do mesmo modo que novos problemas poderiam surgir.

3.2.3 Execução da Fase de Desenvolvimento

Como requisito para o desenvolvimento do experimento, os problemas deveriam ser codificados usando a linguagem de programação Java na versão 8, e os testes unitários deveriam ser implementados utilizando o *framework* JUnit na versão 4.12. Antes de iniciar o processo, os participantes deveriam configurar o ambiente de desenvolvimento para que todos tivessem as mesmas condições.

A lista de atividades, proposta para implementação, foi disponibilizada aos participantes por meio de um arquivo bloqueado para alterações. Neste arquivo continha os requisitos para a resolução do problema. Foi esclarecido que não havia a obrigatoriedade da resolução de todos os itens propostos, muito menos que a implementação deveria seguir a mesma ordem em que as tarefas estavam dispostas no documento.

Os participantes tinham a possibilidade tirar dúvidas sobre o documento com os aplicadores do experimento, além de buscar informações na internet que, de certa forma, ajudassem na resolução dos problemas. Porém, não era permitido haver comunicação entre as pessoas que faziam parte da população do experimento.

As pessoas possuíam uma hora para resolver o máximo de tarefas possíveis, mas havia diferença de como utilizar o tempo a depender da metodologia de desenvolvimento utilizada. O grupo que utilizou o desenvolvimento guiado a testes dispôs de uma hora ininterrupta focado somente na resolução das atividade propostas.Já o grupo com a metodologia testes depois do desenvolvimento, dispôs de quarenta minutos focados somente na implementação, e os vinte minutos restantes apenas para a criação de testes. Os aplicadores do experimento geriram o tempo de execução do dois grupos, de modo que os horários, inicial e final, foram informados aos participantes. Foram criados marcos no tempo para que as pessoas pudessem se organizar melhor. Os marcos de tempo foram avisados a todos os participantes ao decorrer do experimento: o primeiro foi avisado após trinta minutos do inicio do experimento, o segundo com quarenta minutos de duração e o terceiro e ultimo faltando apenas cinco minutos para a entrega do projeto.

Ao final do tempo de desenvolvimento, os participantes deveriam submeter os códigos produzidos em uma plataforma disponibilizada pelos aplicadores do experimento.

3.3 Fase de Análise e Seleção

Depois de recolher os códigos elaborados na fase de desenvolvimento, algumas investigações foram realizadas e, em seguida, apenas foram selecionados os códigos que se enquadravam nos requisitos necessários para a fase de manutenção.

Esta seção do documento descreve todo o passo a passo para análise e seleção dos códigos-fontes.

3.3.1 Análise

Os códigos gerados na etapa de desenvolvimento do experimento foram submetidos à análise para avaliar alguns itens relacionados à qualidade de implementação. Inicialmente, cada código passou por uma inspeção manual. Posteriormente, como auxilio para essa atividade, fez-se necessário a utilização do SonarQube - plataforma que inspeciona, de forma automática e continua, a qualidade de código (SONARSOURCE, 2008).

O SonarQube provê uma série de métricas que indicam se a implementação do código foi feita da melhor maneira possível, ou seja, através da análise de *bugs*, *code smells* e quantidade de linhas codificadas.

LOC, lines of code, é uma métrica utilizada para avaliar o tamanho de um software em relação à quantidade de linhas. Ao analisar os *bugs* (JORGENSEN, 2014) do sistema, obtém-se numericamente a quantidade de defeitos de lógica na implementação de uma tarefa. Já o termo *Code Smells*, traduzindo para mal cheiro de código, representa um conjunto de práticas que divergem das convenções de qualidade de software, como por exemplo: um numero muito grande de LOC em classes e métodos, acoplamento excessivo entre atributos de um sistema, não limitando-se apenas a isso (Khomh; Di Penta; Gueheneuc, 2009).

Para certificar de que o participante aplicou a técnica de TDD ou TLD foi utilizado a métrica *Coverage*. Essa métrica indica o quão o código foi coberto por testes em suas classes, método e linhas. Para isso, fez-se necessário o uso do ambiente de desenvolvimento integrado IntelliJ.

A Tabela 2 representa os dados obtidos após a análise dos códigos desenvolvidos com a técnica TLD.

	Bugs	Code Smells	Linhas de Código	Coverage Classe (%)	Coverage Método (%)	Coverage Linhas (%)
	3	8	143	25	5	12
	1	2	114	100	54	50
	0	19	107	80	61	47
	1	8	129	100	50	45
	0	7	66	100	50	28
	0	6	88	0	0	0
	0	3	72	33	6	15
	3	4	108	75	90	87
	1	16	132	0	0	0
	2	15	51	0	0	0
	2	4	89	50	25	25
	1	15	64	0	0	0
Média	1,16	8,91	96,91	46,91%	28,41%	25,75%

Tabela 2 – Análise dos códigos desenvolvidos com a técnica TLD

Já a Tabela 3 representa os dados obtidos após análise dos códigos que utilizaram

TDD.

	Bugs	Code Smells	Linhas de Código	Coverage Classe (%)	Coverage Método (%)	Coverage Linhas (%)
	2	12	40	100	66	71
	1	1	82	100	47	59
	0	13	39	100	80	76
	0	1	28	100	66	40
	2	13	28	100	66	36
	0	6	31	100	100	93
	3	2	28	100	100	75
	9	22	70	100	100	47
	0	2	32	100	50	68
	0	11	84	0	0	0
	0	7	35	100	75	56
	4	2	135	0	0	0
	4	5	69	0	0	0
Média	1,92	7,46	54,7	76,92%	57,7%	47,76%

Tabela 3 – Análise dos códigos desenvolvidos com a técnica TDD

3.3.2 Seleção

A fase de desenvolvimento serviu como apoio para a criação de códigos que serão utilizados para dar manutenção de possíveis defeitos encontrados. Por esse motivo , códigos que mais seguiram o padrão foram selecionados para a próxima fase do experimento. Há dois tipos de descarte de código: descarte total e descarte parcial. A seguir serão descritos os critérios de descarte e seleção das soluções desenvolvidas.

Para um código ser completamente descartado, ele teria que infringir uma das duas regras:

1. O primeiro, e mais importante, critério de descarte foi o fato de alguns códigos não possuírem testes unitários em sua implementação. Isso implica que nenhuma das técnicas foram aplicadas para a resolução dos problemas propostos, TDD ou TLD.

2. O segundo critério de descarte foi a não resolução de pelo menos um dos problemas propostos. Caso o código não contemplasse em sua implementação a resolução de algum item da lista de atividade, ele seria descartado automaticamente.

Códigos parcialmente descartados tiveram os seus trechos removidos quando infringirem as seguintes regras:

1. Trechos de códigos que provoquem erros de compilação.

2. Quando o item proposto não foi entregue completamente.

Ao final desta etapa, foram obtidos dez códigos válidos para manutenção, sendo metade deles construídos com a metodologia TDD e a outra metade com a metodologia TLD.

3.3.2.1 Contabilização dos Defeitos Encontrados

Para estruturação das seleções dos códigos, os dados aptos à participar da fase de manutenção foram categorizados em uma tabela. Os códigos selecionados passaram por uma alteração em sua nomenclatura para que não houvesse indícios de sua autoria.

Os aplicadores do experimento investigaram, de forma manual e unitariamente, os códigos selecionados em busca de defeitos lógicos relacionados às regras de negócio exigidos na lista de problemas propostos. A busca se deu tanto nos algoritmos quanto nos testes.

Como os códigos tinham suas especificidades, foi analisado o número de itens entregues em relação a cada técnica utilizada para o desenvolvimento. A Tabela 4 representa o número de questões resolvidas, ou parcialmente resolvidas, pelas pessoas que utilizaram TDD para a construção do código. A primeira coluna representa o nome de identificação do projeto e a segunda coluna o número de proposições entregues. Lembrando que foram propostos treze itens para a implementação.

Numero do Código	Itens Entregues
TCC05	2
TCC06	5
TCC07	7
TCC08	5
TCC09	3
MÉDIA	4,4

Tabela 4 – Itens entregues para os códigos que utilizaram TDD

A Tabela 5 representa o número de questões resolvidas, ou parcialmente resolvidas, pelas pessoas que utilizaram TLD para a construção do código. A primeira coluna representa o nome de identificação do projeto e a segunda coluna o número de proposições entregues. Lembrando que foram propostos treze itens para a implementação.

Numero do Código	Itens Entregues
TCC10	8
TCC04	8
TCC03	8
TCC01	8
TCC02	9
MÉDIA	8,2

Tabela 5 – Itens entregues para os códigos que utilizaram TLD

Em virtude da particularidade de cada código, os *bugs* não eram iguais mesmo comparado códigos que utilizaram a mesma metodologia de desenvolvimento. Os *bugs* mais

comuns tratavam-se de análise incorreta dos limites que a aplicação deveria ter e dos cálculos matemáticos relacionados à porcentagem.

Na Tabela 6, a primeira coluna indica o tipo da metodologia utilizada no desenvolvimento do código. A segunda coluna representa a nomenclatura de identificação dos projetos, e na terceira e última coluna a quantidade de *bugs* encontrados com base nos requisitos entregues.

Tipo de Código	Código	Erros Encontrados
TLD	TCC01	3
TLD	TCC02	3
TLD	TCC03	8
TLD	TCC04	5
TDD	TCC05	2
TDD	TCC06	2
TDD	TCC07	3
TDD	TCC08	2
TDD	TCC09	4
TLD	TCC10	3

Tabela 6 – Quantidade de erros encontrados em cada projeto

Com o intuito de facilitar a visualização dos dados a Tabela 7 foi criada. Ela é a representação da média de defeitos encontrados nos códigos, levando em consideração a metodologia de desenvolvimento utilizada.

Tipo de Código	Média Erros Encontrados
TDD	2,6
TLD	4,4

Tabela 7 – Média de Erros Encontrados por Tipo de Código

3.4 Fase de Manutenção

Os códigos selecionados e a identificação dos *bugs* neles contidos serviram como base para dar início a fase de manutenção do experimento.

Nesta etapa da pesquisa, novas experimentações foram realizadas. Agora, os códigos passariam pelo processo de manutenção corretiva.

3.4.1 Participantes da Fase de Manutenção

A escolha dos participantes foi realizada da mesma maneira que na etapa de desenvolvimento. No entanto, o número de participantes que decidiram participar do estudo

foi maior. Ao todo, sessenta e oito pessoas participaram do processo de manutenção dos códigos. Verificar Figura 4.

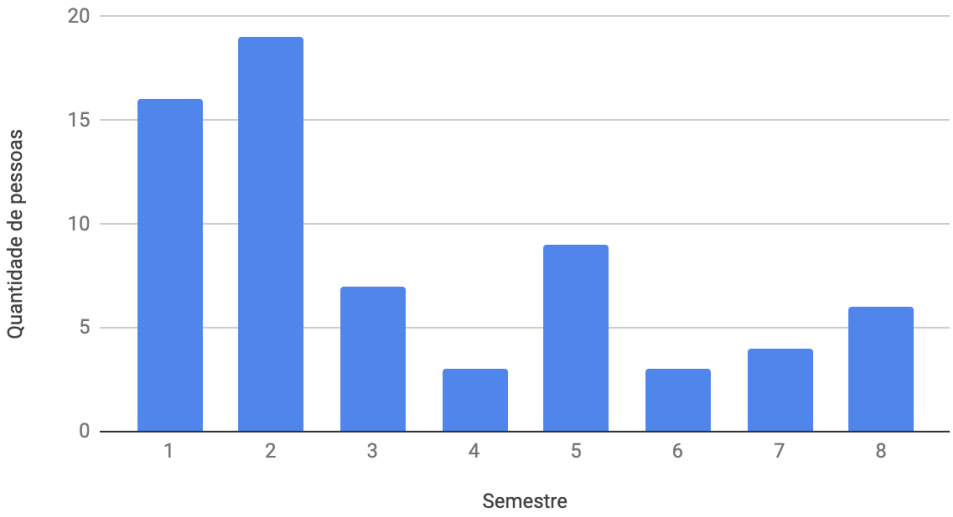

Figura 4 – Distribuição do número de pessoas por semestre durante a fase de manutenção

3.4.2 Problemas Propostos na Fase de Manutenção

Ao analisar os códigos, foi percebida a necessidade de criar, para cada projeto, um novo arquivo contendo todos os itens que foram entregues pelos participantes na fase de desenvolvimento. Desta forma, o documento possuía apenas as tarefas e regras de negócio que foram implementados na etapa de desenvolvimento.

Como os códigos eram diferentes entre si, os documentos também seguiam essa linha, pois eles atendiam somente as necessidades e particularidades do projeto a qual estava relacionado. Os participantes deveriam se basear apenas nos requisitos daquele documento para dar manutenção ao código disponibilizado.

3.4.3 Execução da Fase de Manutenção

Antes de dar início a fase de manutenção do experimento, os participantes foram instruídos de como proceder durante o processo de experimentação. Os participantes poderiam pedir ajuda aos aplicadores, de modo que as perguntas fossem relacionadas ao processo de experimentação. Não era permitido a comunicação entre os participantes, porém quaisquer incertezas técnicas poderiam ser consultadas na internet.

Os materiais utilizados nesta etapa da experimentação foram dois arquivos. O primeiro continha os problemas solucionados, ou parcialmente solucionados, derivados da etapa de desenvolvimento. Já o segundo tratava-se de um *template* para apoiar no registro das informações relacionadas as mudanças realizadas no código.

De posse do documento que continha os problemas propostos, o participante deveria analisar o código a fim de procurar por possíveis *bugs*. Ao encontrar um erro no sistema,

seria necessário fazer uma breve descrição e registra-lo em uma tabela, juntamente com o horário em que foi localizado este erro. Havia a possibilidade de solucionar os *bugs* detectados. Caso o participante realizasse a correção, deveria ser indicado na mesma tabela como foi realizado e qual horário da sua resolução.

Os projetos disponíveis para manutenção foram sorteados entres os participantes, desta forma nenhuma das pessoas tinham ciência de qual metodologia foi utilizada em seu desenvolvimento. As instruções de como configurar o ambiente foram dadas antes de iniciar a contabilização do tempo de execução do experimento.

Os participantes tinham no máximo uma hora para reconhecer, analisar e, se fosse o caso, corrigir os *bugs* encontrados. Porém, caso a pessoa julgasse que o código não possuía mais defeitos, o experimento poderia ser finalizado.

Faltando dez minutos do tempo disponível para finalizar a fase de manutenção do experimento, os participantes foram avisados para que pudessem se organizar para o processo de entrega do projeto. Ao término desta fase, os códigos, que passaram pelo processo de manutenção, foram entregues em uma plataforma disponibilizada pelos aplicadores do experimento.

4 Resultados

Na tentativa de verificar qual técnica, TDD ou TLD, possui mais eficiência no processo de manutenção corretiva, algumas métricas foram selecionadas para medir o tempo desde o reconhecimento do problema até o seu reparo. Os quesitos de análise fazem parte de um conjunto de métricas elaboradas para medir a manutenibilidade de um sistema e foram criadas por Tom Gilb.

1. Reconhecimento do problema: É medido através do tempo que um defeito ou qualquer outro tipo de problema no sistema é identificado. É contabilizado o horário de inicio e de término para saber o tempo resultante (GILB, 2008).

2. Análise do problema: Após o reconhecimento do problema é preciso analisar qual a causa raiz oriunda de uma possível falha. É medido através do tempo em que uma pessoa leva para examinar o código e identificar quais pontos devem ser dados atenção para a etapa de correção (GILB, 2008).

3. Modificação de Implementação: Após o planejamento de uma possível solução para a falha encontrada, é realizado a atividade de correção. Possui a finalidade de buscar a qualidade do sistema. É medido pela quantidade de tempo em que leva para realizar as devidas alterações (GILB, 2008).

4.1 Inspeção dos Códigos-Fonte

Cada código e a sua respectiva tabela de registros, ambos gerados na fase de manutenção do experimento, foram analisados manualmente. A tabela deveria conter em seus registros qual o *bug* e o horário em que foi encontrado. Caso o participante tivesse resolvido o erro, era necessário mencionar como resolveu e o horário da sua correção.

Feito isso, o algoritmo precisaria ser analisado para certificar que o que foi registrado na tabela estava em conformidade com a realidade do código-fonte. As entregas que não possuíam algoritmos ou a tabela de registros não puderam ser analisados. Por esse motivo, foram descartados de imediato.

Em uma grande parte das entregas não houveram modificações na estrutura do código. Isso implica que algumas pessoas só reconheceram/analisaram os erros, mas não corrigiram e outras pessoas não reconheceram/analisaram e nem corrigiram. Esses dados não foram descartados.

Mesmo com a tentativa de balancear a quantidade de código por metodologia, ao final do processo, devido às perdas de dados não foi possível obter um número similar entre as metodologias.

4.2 Métricas

As Tabelas 8 e 9 representam os códigos produzidos com TLD e TDD, respectivamente. Os dados de ambas tabelas representam códigos que em sua manutenção foi analisado e corrigido pelo menos um erro. É possível verificar a média de tempo que uma pessoa levou para analisar os *bugs* existentes no código, o tempo médio que levou para corrigi-los e o número de erros corrigidos e a representação do seu percentual de acordo com a quantidade existente.

	Tempo Médio Reconhecimento + Análise de Erros	Tempo Médio Correção de Erros	Correção de Erros	Percentual Erros Corrigidos
	00:06:35	00:02:40	3 de 8	37.5%
	00:11:35	00:03:35	4 de 5	80%
	00:22:30	00:12:00	2 de 3	66.6%
	00:08:30	00:16:30	2 de 8	25%
	00:08:00	00:14:00	1 de 3	33.3%
	00:07:00	00:03:30	1 de 3	33.3%
	00:17:00	00:07:00	1 de 3	33.3%
	00:14:18	00:01:20	3 de 5	60%
	00:21:00	00:01:00	1 de 3	33.3%
	00:07:33	00:03:00	3 de 8	37.5%
	00:14:00	00:01:00	1 de 3	33.3%
	00:19:00	00:02:30	2 de 3	66.6%
	00:40:00	00:00:45	1 de 3	33.3%
	00:11:40	00:01:00	3 de 8	37.5%
	00:43:00	00:02:00	1 de 8	12.5%
MÉDIA	00:16:47	00:04:47	-	41.55%

Tabela 8 – Tempo médio de reconhecimento, análise e correção de erros utilizando a metodologia TLD

	Tempo Médio Reconhecimento + Análise de Erros	Tempo Médio Correção de Erros	Correção de Erros	Percentual Erros Corrigidos
	00:17:30	00:02:00	1 de 4	25%
	00:11:00	00:08:00	1 de 2	50%
	00:10:40	00:05:40	4 de 4	100%
	00:18:00	00:01:00	1 de 2	50%
	00:15:00	00:09:00	1 de 2	50%
	00:25:00	00:08:00	1 de 2	50%
	00:04:00	00:20:00	1 de 3	33.3%
	00:12:00	00:04:30	2 de 3	66.6%
	00:39:00	00:05:00	1 de 2	50%
MÉDIA	00:16:54	00:07:00	-	52,78%

Tabela 9 – Tempo médio de reconhecimento, análise e correção de erros utilizando a metodologia TDD

De acordo com a Figura 5 percebe-se que em relação ao tempo gasto para descobrir um erro no código a diferença entre TDD e TLD é praticamente imperceptível. Já o tempo que leva para corrigir um erro é maior no TDD quando comparado ao TLD.

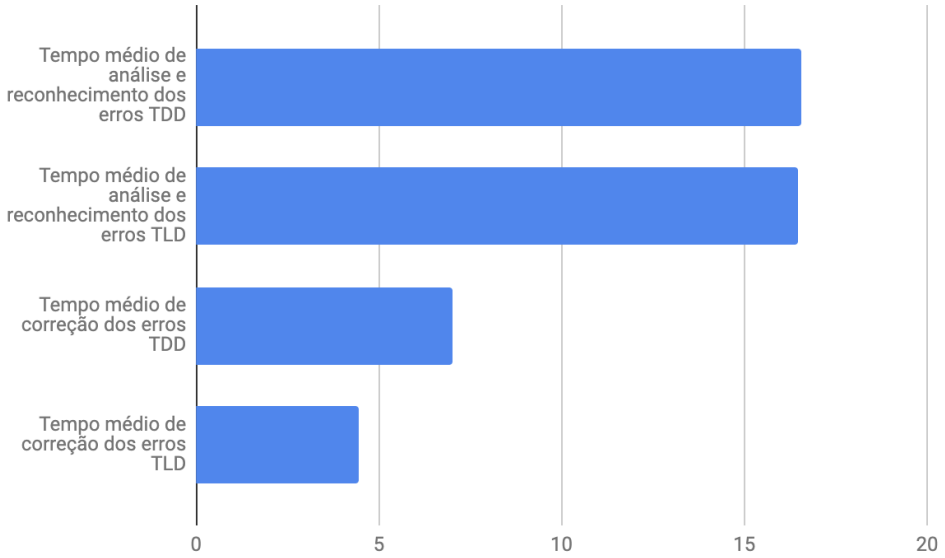

Figura 5 – Análise comparativa TDD x TLD no processo de manutenção corretiva de software

No entanto, conforme a Figura 6, a quantidade de erros corrigidos no TDD é superior à quantidade de erros corrigidos no TLD.

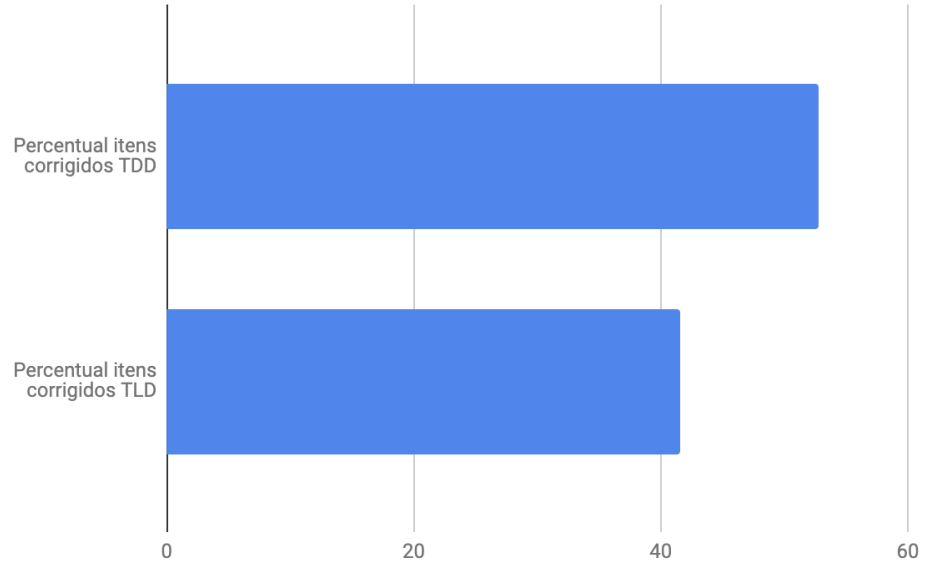

Figura 6 – Análise comparativa TDD x TLD dos itens corrigidos

4.3 Discussões

No presente trabalho, como houveram duas etapas de experimentações, é possível discutir aspectos sobre o desenvolvimento e a manutenção do software.

Todos os códigos gerados na etapa de desenvolvimento foram submetidos a uma análise em busca de mapear o que foi entregue e consultar possíveis defeitos lógicos na implementação. Nesta perspectiva, foi notado que os códigos que utilizam TDD, tiveram uma cobertura maior de teste em todo o programa. Em termos numéricos, códigos que utilizaram a metodologia TDD em seu desenvolvimento possuem aproximadamente 45% de *coverage* maior que códigos desenvolvidos com a metodologia TLD. Uma série de códigos foram descartados pois não atendiam as especificações necessárias para a etapa de manutenção.

Um dos aspectos que influenciou negativamente grande número de descarte de códigos foi a performance dos participantes no desenvolvimento, utilizando as metodologias apresentadas. É possível que isso tenho ocorrido porque as pessoas possuíam pouca ou nenhuma experiência com testes unitários. Essa situação foi relatada pelos próprios participantes dos experimentos.

Dos códigos aptos a participarem da manutenção, foi percebido que entre as duas metodologias analisadas nos experimentos a TLD, no geral, teve mais itens entregues. Consequentemente, devido a um número maior de números de LOC e problemas resolvidos, mais erros foram gerados. Proporcionalmente falando, as metodologias ficaram

equiparadas, pois, em média, o TLD teve o dobro de erros gerados com o dobro de itens entregues, comparado com TDD. A Tabela 7 representa o que foi dito.

Na fase de manutenção, os participantes deveriam localizar e corrigir possíveis erros existentes nos códigos a eles apresentados.

Pessoas que deram manutenção em códigos que foram construídos utilizando TLD levaram em média 17 minutos para localizar o possível defeito. Já para corrigir os erros, esses participantes levaram em média 5 minutos, e tiveram 41,5% de efetividade na correção.

Pessoas que deram manutenção em códigos que foram construídos utilizando TDD levaram em média 17 minutos para localizar o possível defeito. Já para corrigir os erros, esses participantes levaram em média 7 minutos, e tiveram 52,8% de efetividade na correção.

Os participantes dos experimentos relataram que alguns erros foram perceptíveis mesmo sem o uso de nenhuma das técnicas propostas, mas, de modo geral, os testes auxiliaram na detecção e posteriormente na correção dos erros existentes nas aplicações.

Comparando TDD com TLD, percebe-se que a diferença do tempo para detecção do problema é irrelevante, nesse aspecto a produtividade é irrisória. Já quando comparado o tempo de correção de defeitos entre as duas metodologias, o TLD mostrou-se mais produtivo, pois a média de tempo para a realização desta atividade é menor. No aspecto da qualidade, foi analisados a quantidade de defeitos corrigidos, nessa perspectiva, o TDD mostrou-se ligeiramente mais eficiente.

5 Conclusões

Neste trabalho foram realizados experimentos a fim de comparar as técnicas de TDD e TLD, buscando encontrar qual delas possui melhor desempenho em virtude da manutenção de software. Para isso, métricas foram definidas para avaliar a qualidade do software gerado e a produtividade. Elas foram definidas como a quantidade de erros encontrados após a conclusão de uma funcionalidade de software e o tempo resultante de reconhecimento, análise e correção destes erros.

Inicialmente foram elaborados experimentos para criação de um *dataset* contendo códigos construídos com ambas metodologias de desenvolvimento. Depois, com a utilização deste *dataset* criado anteriormente, novos experimentos foram elaborados com o propósito de atestar maior eficiência do TDD em relação a manutenção quando comparado com TLD.

Para o quesito de desenvolvimento de software, os resultados do primeiro experimento mostraram que pessoas que utilizam o desenvolvimento guiado a testes produzem em média metade do volume de código quando comparados com pessoas que utilizam testes após desenvolvimento. Porém, código com TDD possuem mais indícios de qualidade do que o códigos com TLD.

Para o quesito manutenção corretiva de software, os resultados do segundo experimento mostraram que as técnicas não influenciaram nem positivamente e negativamente em relação ao tempo de análise e o tempo de correção de um erro. Mas, os códigos com TDD possuíram mais itens corrigidos no mesmo intervalo de tempo.

5.1 Sugestão Para Trabalhos Futuros

Um possível trabalho futuro seria padronizar o nível e a quantidade dos *bugs* oriundos de códigos gerados durante os experimentos realizados na etapa de desenvolvimento. Pois do jeito em que se encontra o presente trabalho, a comparação dos códigos foi dificultada devido as suas estruturas internas serem completamente distintas umas das outras. Isso impactou diretamente nas experimentações realizadas na fase de manutenção.

Diante disso, para os próximos trabalhos será melhorado os seguintes quesitos:

1. Definir novas métricas para complementar as já existentes.
2. Delimitar o escopo da resolução dos problemas proposto.
3. Pré-selecionar os participantes a fim elaborar melhor os exercícios de acordo com seus perfis.

A partir dessas mudanças pretende-se encontrar resultados mais fiéis e com o objetivo de detectar qual metodologia é mais eficiente no processo de manutenção corretiva.

Referências Bibliográficas

BASSIL, Y. A simulation model for the waterfall software development life cycle. 05 2012. 20

BECK, C. A. K. *Extreme Programming Embrace Change*. 2. ed. [S.l.]: John Wait, 2005. 24

BECK, K. *Test Driven Development: By Example*. 2. ed. [S.l.]: Person Education, Inc. and Kent Beck, 2003. 16

BERNER ROLAND WERBER, R. K. K. S. Observations and lessons learned from automated testing. Proceedings of the 27th international conference on Software engineerin, p. 571–579, 2005. 23

BERTOLINO, A. Software testing research: Achievements, challenges, dreams. Future of Software Engineering, 2007. 21

BISSI, W.; NETO, A.; EMER, M. C. F. P. The effects of test driven development on internal quality, external quality and productivity: A systematic review. *Information and Software Technology*, v. 74, 02 2016. 24, 27

BUTLER, M.; MORGAN, M. Learning challenges faced by novice programming students studying high level and low feedback concepts. 01 2007. 12, 31

ERDOGMUS, H.; MORISIO, M.; TORCHIANO, M. On the effectiveness of the test-first approach to programming. *Software Engineering, IEEE Transactions on*, v. 31, p. 226–237, 04 2005. 24

FILHO, W. de P. P. *Engenharia de software: fundamentos, métodos e padrões*. 3. ed. [S.l.]: LTC, 2008. ISBN 8521616503. 19

FUCCI, D. et al. A dissection of the test-driven development process: Does it really matter to test-first or to test-last? *CoRR*, abs/1611.05994, 2016. Disponível em: <http://arxiv.org/abs/1611.05994>. 24

GELPERIN, D.; HETZEL, B. The growth of software testing. *Communications of the ACM*, v. 31, p. 687–695, 06 1988. 16

GEORGE, B.; WILLIAMS, L. An initial investigation of test driven development in industry. In: *Proceedings of the 2003 ACM Symposium on Applied Computing*. New York, NY, USA: ACM, 2003. (SAC '03), p. 1135–1139. ISBN 1-58113-624-2. Disponível em: <http://doi.acm.org/10.1145/952532.952753>. 17, 25

GILB, T. Designing maintainability in software engineering: a quantified approach. *INCOSE*, 2008. 39

GOVARDHAN, D. A comparison between five models of software engineering. *IJCSI International Journal of Computer Science Issues 1694-0814*, v. 7, p. 94–101, 09 2010. 20

GRUBB, A. A. T. P. *Software Maintenance: Concepts And Practice*. 2. ed. [S.l.]: World Scientific, 2003. 26, 29

ISO/IEC/IEEE International Standard - Systems and software engineering – Software life cycle processes. *ISO/IEC/IEEE 12207:2017(E) First edition 2017-11*, p. 1–157, Nov 2017. 19

Janzen, D.; Saiedian, H. Test-driven development concepts, taxonomy, and future direction. *Computer*, v. 38, n. 9, p. 43–50, Sep. 2005. ISSN 0018-9162. 17

JANZEN, D.; SAIEDIAN, H. A leveled examination of test-driven development acceptance. *29th International Conference on Software Engineering (ICSE'07)*, p. 719–722, 2007. 11, 27, 28

JORGENSEN, P. C. *Software Testing Fourth Edition Software Testing A Craftsman's Approach*. [S.l.]: CRC Press Taylor & Francis Group, 2014. 22, 33

Khomh, F.; Di Penta, M.; Gueheneuc, Y. An exploratory study of the impact of code smells on software change-proneness. In: *2009 16th Working Conference on Reverse Engineering*. [S.l.: s.n.], 2009. p. 75–84. ISSN 1095-1350. 33

LIENTZ, B. P.; SWANSON, E. B.; TOMPKINS, G. E. Characteristics of application software maintenance. *Commun. ACM*, ACM, New York, NY, USA, v. 21, n. 6, p. 466–471, jun. 1978. ISSN 0001-0782. Disponível em: <http://doi.acm.org/10.1145/359511.359522>. 26

LUO, L. Software testing techniques. Institute for Software Research International Carnegie Mellon University, 2001. 22

MALL, R. *Fundamentals of Software Engineering*. 5. ed. [S.l.]: PHI Learning Pvt. Ltd., 2018. 26

MAMONE, S. The ieee standard for software maintenance. *SIGSOFT Softw. Eng. Notes*, ACM, New York, NY, USA, v. 19, n. 1, p. 75–76, jan. 1994. ISSN 0163-5948. Disponível em: <http://doi.acm.org/10.1145/181610.181623>. 25

MYERS COREY SANDLER, T. B. Glenford j. *The Art of Software Testing*. 3. ed. [S.l.]: John Wiley & Sons, Inc, 2012. 22

MüLLER, M. M.; HAGNER, O. Experiment about test-first programming. *IEE Proceedings - Software*, v. 149, p. 131–136, 01 2002. 25

PRESSMAN, R. *Engenharia de software*. 6. ed. [S.l.]: LTC, 2010. 16, 19, 20, 21, 22

RAFIQUE, Y.; MISIC, V. The effects of test-driven development on external quality and productivity: A meta-analysis. *Software Engineering, IEEE Transactions on*, v. 39, p. 835–856, 06 2013. 24

REZENDE, D. A. *Engenharia de Software e Sistemas de Informação*. 3. ed. [S.l.]: Brasport Livros e Multimídia Ltda., 2005. 26

SOMMERVILLE, I. *Engenharia de software*. 9. ed. [S.l.]: Pearson Education, 2011. 19, 23

SONARSOURCE. *SonarQube*. 2008. Disponível em: <https://www.sonarqube.org>. 33

THOMAS, C. M. An overview of the current state of the test-first vs. test-last debate. Division of Science and Mathematics University of Minnesota, Morris Morris, Minnesota, 2014. 16, 23

TOSUN, A. et al. On the effectiveness of unit tests in test-driven development. In: *Proceedings of the 2018 International Conference on Software and System Process*. New York, NY, USA: ACM, 2018. (ICSSP '18), p. 113–122. ISBN 978-1-4503-6459-1. Disponível em: <http://doi.acm.org/10.1145/3202710.3203153>. 17

TRAVASSOS DMYTRO GUROV, E. A. G. d. A. G. H. Introdução à engenharia de software experimental. 2002. 28

VLIET, H. v. *Software Engineering: Principles and Practice*. 3rd. ed. [S.l.]: Wiley Publishing, 2008. ISBN 0470031468. 20

WAZLAWICK, R. *Engenharia de Software: Conceitos e práticas*. [S.l.: s.n.], 2013. ISBN 978-85-352-6084-7. 19, 20

A Formulário Pré Experimento

1. Qual seu Nome?
2. Qual seu Semestre?
3. Qual seu E-mail?
4. Experiência com Desenvolvimento de Software. (Não possui experiência, De 1 à 6 Meses, de 6 meses à 1 ano, de 1 ano à 2 anos, de 2 anos à 3 anos, mais que 3 anos, outros).
5. Qual a linguagem de programação do seu conhecimento?
6. Experiência com desenvolvimento com o paradigma orientado a objeto. (Não possui experiência, De 1 à 6 Meses, de 6 meses à 1 ano, de 1 ano à 2 anos, de 2 anos à 3 anos, mais que 3 anos, outros).
7. Possui conhecimento em casos de testes?
8. Experiência com testes automatizados (Junit). (Não possui experiência, De 1 à 6 Meses, de 6 meses à 1 ano, de 1 ano à 2 anos, de 2 anos à 3 anos, mais que 3 anos, outros).
9. Experiência com desenvolvimento guiado a testes (TDD). (Não possui experiência, De 1 à 6 Meses, de 6 meses à 1 ano, de 1 ano à 2 anos, de 2 anos à 3 anos, mais que 3 anos, outros).
10. Possui conhecimento em manutenção de software?

B Formulário Pós Desenvolvimento (TDD)

1. Qual seu Nome?
2. Qual seu Semestre?
3. Qual seu E-mail?
4. Complexidade dos problemas. (Fácil (1 - 2 - 3 - 4) Difícil).
5. Dificuldade para implementar testes. (Fácil (1 - 2 - 3 - 4) Difícil).
6. Tempo para resolver o problema. (Insuficiente (1 - 2 - 3 - 4) Mais que suficiente).
7. Dificuldade para implementar TDD em algum exercício? Se sim, qual?
8. TDD auxiliou na organização da estrutura da solução?
9. Testes auxiliaram a confiança no código que foi desenvolvido?

C Formulário Pós Desenvolvimento (TLD)

1. Qual seu Nome?
2. Qual seu Semestre?
3. Qual seu E-mail?
4. Complexidade dos problemas (Fácil (1 - 2 - 3 - 4) Difícil).
5. De modo geral, indique a dificuldade para implementar testes (Fácil (1 - 2 - 3 - 4) Difícil). para implementar testes em algum exercício? Se sim, qual?
6. Tempo para resolver o problema (Insuficiente (1 - 2 - 3 - 4) Mais que suficiente).
7. Testes auxiliaram a confiança no código que foi desenvolvido?

D Formulário Pós Desenvolvimento (TLD)

1. Qual seu Nome?
2. Qual seu Semestre?
3. Qual seu E-mail?
4. Dificuldade para encontrar o problema (Fácil (1 - 2 - 3 - 4) Difícil).
5. Dificuldade para corrigir o problema (Fácil (1 - 2 - 3 - 4) Difícil). Legibilidade do código; Facilidade de compreensão (Fácil (1 - 2 - 3 - 4) Difícil).
6. Testes ajudaram, atrapalharam ou foram indiferentes no processo de manutenção? Justifique sua resposta.
7. Visão geral do experimento: Explique como foi a sua experiência com o experimento.

E Problemas propostos

Calculadora Reserva GaJu Veículos

O participante deve implementar uma calculadora de aluguel da GaJu Veículos. Um cliente possui idade, clube fidelidade, tempo em dias do aluguel, grupo do carro e forma de pagamento: que pode ser parcelado ou à vista. Os dados devem ser inseridos na respectiva ordem. De acordo com as condições é estipulado o valor da diária do carro.

Requisitos: As diárias dos carros na Gaju Veículos são separados por preço: Grupo S: 220 Reais; Grupo A: 150 Reais; Grupo B: 90 Reais

1. Pessoas com idade menor que 18 anos não podem fazer reservas.
2. Pessoas menores de 20 anos terão um acréscimo de 7% no valor total da reserva.
3. Reservas do Grupo B não são elegíveis de desconto.
4. A partir de 10 dias de reserva é aplicado um desconto de 5% no valor total da reserva.*
5. Clube Fidelidade é aplicado um desconto de 5% no valor total da reserva.*
6. Clientes Acima de 60 anos possui um desconto de 3,5%.
7. O pagamento da reserva pode ser à vista ou parcelado em 2 vezes. Caso seja parcelado, guarde os valores das parcelas utilizando um vetor.
8. Somente os descontos para pessoas acima de 60 anos não são acumulativos. Ou seja, se algum dos descontos existentes for aplicado, este não deverá ser computado no cálculo do valor final da reserva.

* Os descontos são progressivos, ou seja, se o cliente tiver direito aos dois descontos, deve-se aplicar um e, em seguida, aplicar o outro.

* O desconto que prevalece é sempre o que possui maior maior valor.