UNIVERSIDADE DO SUL DE SANTA CATARINA

LINDOMAR PEIXINHO REITZ

QUALIDADE DE SOFTWARE COM TESTES AUTOMATIZADOS: UM ESTUDO DE CASO SOBRE TESTES DE UNIDADE, INTEGRAÇÃO E DE SISTEMA

Florianópolis

2014

LINDOMAR PEIXINHO REITZ

QUALIDADE DE SOFTWARE COM TESTES AUTOMATIZADOS: UM ESTUDO DE CASO SOBRE TESTES DE UNIDADE, INTEGRAÇÃO E DE SISTEMA

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Sistemas de Informação da Universidade do Sul de Santa Catarina, como requisito parcial à obtenção do título de Bacharel em Sistemas de Informação.

Orientador: Profa. Maria Inés Castiñeira, Dra.

Florianópolis

2014

LINDOMAR PEIXINHO REITZ

QUALIDADE DE SOFTWARE COM TESTES AUTOMATIZADOS: UM ESTUDO DE CASO SOBRE TESTES DE UNIDADE, INTEGRAÇÃO E DE SISTEMA

Este Trabalho de Conclusão de Curso foi julgado adequado à obtenção do título de Bacharel em Sistemas de Informação e aprovado em sua forma final pelo Curso de Graduação em Sistemas de Informação da Universidade do Sul de Santa Catarina.

Florianópolis, 16 de Junho de 2014.

Professora e orientadora Maria Inés Castiñeira, Dra.
Universidade do Sul de Santa Catarina

Prof. Flávio Ceci, Msc.
Universidade do Sul de Santa Catarina

Alessandra Mohr, Bel. em Sistemas de Informação.
Dígitro

Agradeço a todas as pessoas que me ajudaram direta ou indiretamente, em especial a professora Maria Inés por toda ajuda e paciência nesse tempo.

AGRADECIMENTOS

Primeiramente agradeço a professora Maria Inés, sendo que meses antes de começar este trabalho já se prontificou em aceitar o convite de ser minha orientadora, assim como me auxiliou em refinar as ideias e escopo deste trabalho. Também agradeço por todos os conselhos e as orientações que me davam uma nova visão para o trabalho e para o meu futuro em si, assim como os conselhos para pegar “mais leve” e descansar em alguns dias.

A coordenadora Vera, que sempre me ajudou em tudo que precisei nesse período da graduação e por ter me indicado a Maria Inés como orientadora.

Ao professor Ivo, pelas suas correções de português e me auxiliando na melhora da escrita no decorrer deste trabalho.

A todos os professores da UNISUL com quem eu tive oportunidade de obter novos conhecimentos e mudar a minha visão sobre o mercado de TI e computação em geral.

A todo o pessoal que estudou comigo nesses últimos tempos, principalmente pelos momentos de diversão quando estava mais estressado. Existem várias pessoas, mas preciso agradecer a Viviane, Paulo César, Jonathan Alves, Jhonatan Martarello e Valter pelo tempo dedicado nos trabalhos, assim como a amizade gerada disso.

Por fim, agradeço a minha família que me deu subsídios para estar aqui. Mesmo sabendo que eles não me incentivaram diretamente, sei que estarão felizes por esse momento.

“Foco é uma questão de decidir as coisas que você não irá fazer.” (John Carmack).

RESUMO

A qualidade de *software* está ganhando cada vez mais importância, pois sem ela, alguns defeitos não são identificados a tempo, trazendo prejuízos financeiros à empresa e diminuindo a confiança dos clientes e a produtividade dos usuários. Vários tipos de testes são propostos para aferir a qualidade de um sistema, sendo que, em alguns casos, é pertinente a sua automação, devido a sua reprodutibilidade e confiabilidade, já que se feito por seres humanos estes podem esquecer-se de passos importantes. Neste trabalho foi realizado um estudo de caso sobre testes automatizados nos níveis de unidade, integração e sistema, utilizando como base um sistema *web*. Os testes automatizados ganharam grande importância nos últimos tempos, principalmente com o advento das metodologias ágeis, que procuram entregar *software* com qualidade de forma incremental e iterativa, a partir de curtos períodos de tempo. Além disso, também existe a promessa de serem rápidos e com pouca ou nenhuma intervenção humana, aumentando o grau de confiança e repetição na execução dos testes. O estudo de caso é uma pesquisa aplicada, usando dados qualitativos sobre a elaboração e implementação dos casos de teste de forma automatizada, além dos seus resultados e recomendações em geral. Os testes foram realizados em um sistema web que realiza gerenciamento de receitas, despesas e contas bancárias, desenvolvido na linguagem Java. As ferramentas JUnit, DBUnit, Selenium WebDriver e JaCoCo foram utilizadas para dar apoio a essa automação, de acordo com o nível de teste e necessidade no projeto. Como resultados, 39 casos de teste foram executados em 53,87 segundos, atingindo um grau satisfatório de cobertura de código (acima de 70%) para os testes de unidade e integração, assim como uma boa cobertura para os testes de sistema, levando em consideração seus requisitos e casos de uso.

Palavras-chave: Testes automatizados. Qualidade de *software*. Engenharia de *software*.

ABSTRACT

The software quality is gaining more and more importance, because without it, some defects are not identified at time, bringing financial losses to the company and decreasing the trust of the customers and the users productivity. Many test types are proposed to assess the quality of a system, being that in some cases is pertinent your automation, because your reproducibility and reliability, already that if made by humans they can forget important steps. In this work was realized a study case about automated tests in the levels of unit, integration and system, using as base a web system. The automated tests gained big importance on the last times, mainly with the advent of agile methodologies that found deliver software with quality of iterative and incremental form, from short periods of time. Furthermore, also exists the promise of be fast and with a little or no human intervention, increasing the degree of confidence and repetition in execution of the tests. The study case is an applied research, using qualitative data about the elaboration and implementation of test cases in automated way, beyond your results and recommendations in general. The tests were realized in a web system that performs the management of revenues, deposits and bank accounts, developed in the Java language. The tools JUnit, DBUnit, Selenium WebDriver and JaCoCo were used to give support to this automation, according with the test level and needs of the project. As results, 39 test cases are executed at 53,87 seconds, reaching a satisfactory degree of code coverage (above 70%) to the unit and integration tests, as well as a good coverage for the system tests, taking in consideration your requirements and use cases.

Keywords: Automated tests. Software Quality. Software Engineering.

LISTA DE ILUSTRAÇÕES

LISTA DE QUADROS

LISTA DE SIGLAS

ABNT - Associação Brasileira de Normas Técnicas
API - *Application Programming Interface*
CMMI - *Capacity Maturity Model Integrated*
HTML - *Hypertext Markup Language*
IDE - *Integrated Development Environment*
IEEE - *Institute of Eletrical and Eletronic Engineers*
IEC - *Commision Eletrotechnique Internatiole*
ISO - *International Organization for Standardization*
JPA - *Java Persistence API*
MPS.BR - Melhoria dos Processos de *Software* Brasileiro
OO - Orientado a Objeto
ORM - *Object-relational mapping*
TDD - *Test-Driven Development*
UML - *Unified Modeling Language*
URL - *Uniform Resource Locator*
XML - *eXtensible Markup Language*
XP - *Extreme Programming*

SUMÁRIO

1 INTRODUÇÃO

A qualidade de *software* tem cada vez mais importância para as empresas que desenvolvem software, uma vez que o mercado é cada vez mais competitivo e exigente perante todos os sistemas criados e utilizados. Assim, qualidade deixa de ser uma opção para se tornar um requisito essencial para o sucesso do produto desenvolvido. (BARTIÉ, 2002).

Através de processos e técnicas, a engenharia de *software* possui várias formas de avaliar a qualidade de um *software*, sendo uma delas os testes de *software*. As principais vantagens de se testar um *software* podem ser tanto por garantir que um *software* funciona de forma correta quanto descobrir problemas que poderiam custar muito caro caso fossem descobertos em produção, diminuindo a confiança dos clientes e manchando o nome da empresa.

Para que um sistema seja construído conforme os requisitos e atenda critérios de qualidade, a engenharia de *software* utiliza processos de desenvolvimento, nesse sentido, conforme Pressman (2011, pag. 40): "Processo é um conjunto de atividades, ações e tarefas realizadas na criação de algum produto de trabalho (*work product*).".

No processo de desenvolver *software*, existem alguns modelos ou metodologias de processo que podem ser aplicadas. Segundo Pressman (2011, pag. 40):

> Uma metodologia (*framework*) de processo estabelece o alicerce para um processo de engenharia de *software* completo, por meio da identificação de um pequeno número de atividades estruturais aplicáveis a todos os projetos, independentemente de tamanho ou complexidade.

Atualmente, essas metodologias são classificadas como tradicionais e ágeis, sendo que a segunda dá ênfase na utilização de testes automatizados no processo de desenvolvimento, como, por exemplo, na metodologia *Extreme Programming* (XP). (PRESSMAN, 2011, pag. 90).

1.1 PROBLEMÁTICA

Como já apresentado, os testes auxiliam a obter um produto de software com menor quantidade de erros, mas a sua execução deve ser bem planejada. Para isso, é necessário entender qual tipo de teste pode ser aplicado em cada etapa do desenvolvimento, como aplicar o teste, quem deve aplicar o teste, se é possível utilizar ferramentas para auxiliar nesse processo, entre outras. Ou seja, essas atividades exigem um conhecimento especializado.

Apesar de que os testes auxiliam na parte de garantia de qualidade, também podem ser caros no sentido de serem executados de forma manual, dificultando a sua repetição e a garantia de que foram executados corretamente, uma vez que passos importantes podem ser esquecidos em uma execução feita por seres humanos.

Diante desse cenário, chega-se à seguinte pergunta de pesquisa: É possível realizar testes de *software* de forma automatizada, de tal forma que os mesmos possam ser executados diversas vezes, sejam rápidos e sofram intervenção humana no menor grau possível?

1.2 OBJETIVOS

São apresentados, a seguir, os objetivos deste trabalho.

1.2.1 Objetivo geral

Este projeto tem como objetivo principal realizar um estudo de caso de como um *software* pode ser testado de forma automatizada, aplicando esses testes em um sistema *web*.

1.2.2 Objetivos específicos

Como objetivos específicos, são apresentados os seguintes:

- pesquisar processo de testes de *software* que podem ser usados dentro de metodologias tradicionais e ágeis;
- levantar os tipos de testes, como e quando podem ser aplicados;
- pesquisar na literatura sobre automação de testes, seus procedimentos e ferramentas, entre outros;
- exemplificar a utilização de testes automatizados, utilizando testes de unidade, integração e testes de sistema em um aplicativo *web*;
- fazer recomendações sobre automação de testes.

1.3 JUSTIFICATIVA

Nos dias atuais, as empresas de *software* estão interessadas em melhorar cada vez mais a qualidade dos seus sistemas, já que a qualidade de *software* detecta erros que são identificados e corrigidos antes de chegar aos seus clientes, agregando valor ao produto desenvolvido.

Na perspectiva da qualidade de *software*, os testes automatizados possuem uma grande importância, uma vez que em longo prazo os mesmos diminuem os custos com a execução e melhoram o próprio processo de qualidade. Dessa forma, o *software* pode ser testado de forma automatizada, mudando o foco da execução dos testes para a melhoria da análise e avaliação do *software* a ser testado. Os testes realizados neste trabalho serão aplicados em um sistema *web*, uma vez que, atualmente, esse tipo de *software* é muito utilizado no mercado.

Mesmo havendo um custo e esforço maior em um primeiro momento, as vantagens de automação dos testes podem ser justificadas. Segundo Bartié (2002, pag. 63-64):

> A automação de testes é altamente desejada por diversos fatores, inclusive em termos de custos finais. Como esse processo requer um investimento inicial, a automação passa a ser encarada como mais um trabalho a ser realizado. À medida que reexecutamos os testes, o ganho de tempo, controle, confiabilidade e as diversas possibilidades existentes com essa tecnologia, fica clara a vantagem inerente a esse processo.

Essa preocupação com a automação de testes não é recente, além do que a sua utilização ficou mais evidente com as metodologias ágeis, que focam em construir sistemas de forma iterativa e incremental, utilizando testes de unidade, integração e sistema como garantia da qualidade do *software* construído. (PRESSMAN, 2011, pag. 90). Isso também muda um pouco a forma em que um profissional de testes atua, sendo uma necessidade atual do mercado de testes e *software* em geral.

1.4 ESTRUTURA DA MONOGRAFIA

Este trabalho está dividido em cinco capítulos. A seguir, será apresentada a sua estrutura:

- Capítulo 1 – Introdução: apresenta a introdução, problema, os objetivos e a justificativa do trabalho.
- Capítulo 2 – Revisão bibliográfica: este capítulo apresenta temas relacionados à Qualidade, Processos, Testes de *software* e automação de testes.
- Capítulo 3 – Metodologia: apresentação da metodologia de pesquisa, a proposta da solução e as delimitações do projeto.
- Capítulo 4 – Desenvolvimento: este capítulo apresenta o desenvolvimento de um estudo de caso, assim como a definição das ferramentas utilizadas, entre outros.
- Capítulo 5 – Conclusões finais: este capítulo apresenta as conclusões deste trabalho e sugestões para trabalhos futuros.

2 REVISÃO DA LITERATURA

Este capítulo apresenta a revisão da literatura, abordando assuntos de qualidade testes e processos de *software*, entre outros.

2.1 O QUE É QUALIDADE DE *SOFTWARE*?

Atualmente, a qualidade é cada vez mais levada em consideração quando ocorre a aquisição de um produto. Essa qualidade geralmente é definida por alguns critérios que são subjetivos, porém podem ser fatores fundamentais para a utilização/escolha entre um produto e outro.

Com os produtos de *software*, isso não é diferente, uma vez que um sistema considerado de "má qualidade" poderá trazer prejuízos financeiros a quem utiliza, assim como abrir brechas para que a concorrência obtenha vantagem competitiva, prejudicando diretamente a empresa que o desenvolveu. (RIOS; MOREIRA, 2006). Para melhorar a qualidade dos produtos de *software*, entra em questão o entendimento do conceito qualidade de *software* e de como ela pode ajudar a evitar esses problemas.

Segundo Bartié (2002, pag. 16), a qualidade de *software* pode ser definida como: "[...] um processo sistemático que focaliza todas as etapas e artefatos produzidos com o objetivo de garantir a conformidade de processos e produtos, prevenindo e eliminando defeitos.". Essa necessidade de utilizar um processo de qualidade de *software* surgiu pelos problemas recorrentes que as empresas sofrem com produtos de baixa qualidade.

> O clamor por maior qualidade de *software* começou realmente quando o *software* passou a se tornar cada vez mais integrado em todas as atividades de nossas vidas. Na década de 1990, as principais empresas reconheciam que bilhões de dólares por ano estavam sendo desperdiçados em *software* que não apresentava as características e as funcionalidades prometidas. (PRESSMAN, 2011, pag. 358).

Segundo Bartié (2002), a qualidade de *software* deve estabelecer duas dimensões para garantia da qualidade, sendo estas as dimensões da qualidade do processo e da qualidade do produto. A primeira dimensão foca-se em adotar uma cultura de não tolerância a erros,

garantindo que todos os artefatos que serão gerados durante o ciclo de desenvolvimento possam ter procedimentos que avaliem a sua qualidade. Para Schuhmacher (2006, pag. 23):

> Quando é avaliado o processo de *software*, avaliam-se todas as etapas do processo de produção do produto, da análise do problema a manutenção do mesmo no cliente. Se a empresa possui um processo de desenvolvimento imaturo, seu projeto não é rigorosamente cumprido, o controle de qualidade e as funcionalidades do produto podem ficar comprometidos pois os prazos devem ser cumpridos, os custos de manutenção podem ficar elevados. Todo o projeto pode ficar comprometido.

Já na qualidade do produto, o foco deixa de ser o processo para "[...] garantir a qualidade do produto tecnológico gerado durante o ciclo de desenvolvimento." (BARTIÉ, 2002, pag. 18). Para Pressman (2011, pag. 371), essa dimensão pode ser definida como:

> O objetivo da garantia da qualidade é fornecer ao pessoal técnico e administrativo os dados necessários para ser informados sobre a qualidade do produto, ganhando, portanto, entendimento e confiança de que as ações para atingir a qualidade do produto estão funcionando.

Para que essas atividades da qualidade de *software* tenham diretrizes que possam ser seguidas, foram criadas normas e padrões para avaliar tanto o processo utilizado quanto o produto de *software*. (SCHUHMACHER, 2006, pag. 28).

Para Sommerville (2011), os padrões são importantes pelo fato de que:

- são baseados em boas práticas, ajudando também a reusar conhecimento que foi obtido anteriormente com tentativa e erro;
- fornecem uma estrutura padrão que ajuda a empresa a definir o significado de qualidade, e assim a empresa possui uma base para decidir se a mesma foi atingida ou não;
- asseguram que todos os envolvidos adotem as mesmas práticas, diminuindo o esforço de aprendizado.

Esses padrões foram criados por instituições internacionais como ISO (*International Organization for Standardization*), IEEE (*Institute of Eletrical and Eletronic Engineers*) e IEC (*Commision Eletrotechnique Internationale*). (SOMMERVILLE, 2011, pag. 459-460).

No Brasil, a ABNT (Associação Brasileira de Normas Técnicas) também possui normas do tipo. (SCHUHMACHER, 2006, pag. 30). Além disso, padrões de qualidade

também são definidos em modelos de maturidade organizacional, como CMMI (*Capacity Maturity Model Integrated*). (SOMMERVILLE, 2011, pag. 504). No Brasil, o MPS.BR (Melhoria dos Processos de *Software* Brasileiro) também segue uma linha parecida com CMMI, porém voltada ao mercado brasileiro de *software*. (SOFTEX, 2012).

Segundo Pressman (2011, pag. 685) esses modelos de maturidade organizacional podem servir para:

> [...] proporcionar uma indicação geral da "maturidade do processo" exibida por uma organização de *software*. Por exemplo, a indicação da qualidade do processo de *software*, o grau segundo o qual os profissionais entendem e aplicam o processo, e o estado geral da prática de engenharia de *software*.

Complementando essa definição, Bartié (2002, pag. 8-9) diz que "[...] tem como foco o processo de *software* na proposta de melhoria contínua, trazendo disciplina e controle no desenvolvimento e manutenção do *software*.".

Uma das formas de medir a qualidade de *software* é através dos testes, pois "[...] o processo de qualidade de *software* utiliza-se dos testes em todo o ciclo de desenvolvimento, de forma a garantir tanto o processo de engenharia quanto o produto de *software* desenvolvido." (BARTIÉ, 2002, pag. 18-19). Dessa forma, os testes de *software* serão definidos a seguir.

2.2 O QUE SÃO TESTES DE *SOFTWARE*?

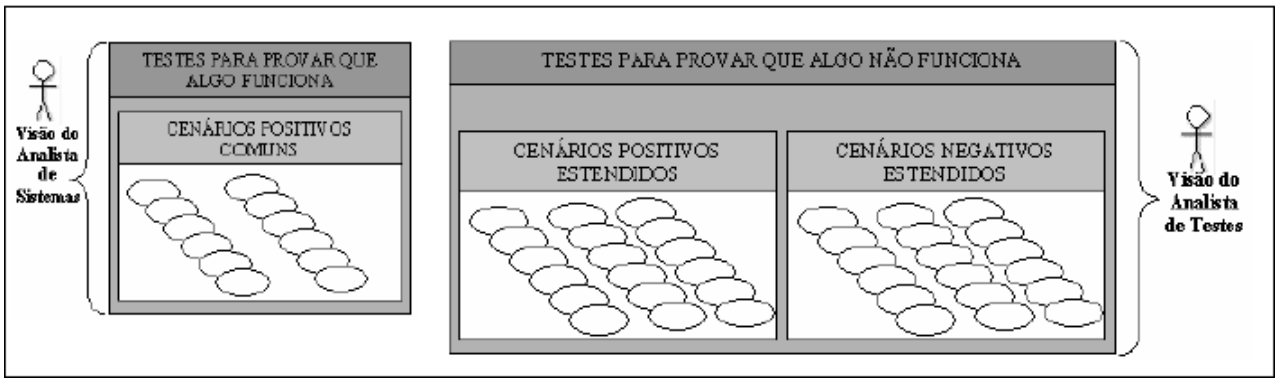
Os testes de *software* são uma etapa muito importante para detectar erros nos sistemas. Segundo Sommerville (2011, pag. 144):

> O teste é destinado a mostrar que um programa faz o que é proposto e para descobrir os defeitos do programa antes do uso. Quando se testa o *software*, o programa é executado usando dados fictícios. Os resultados do teste são verificados à procura de erros, anomalias ou informações sobre os atributos não funcionais do programa.

Para Pressman (2011, pag. 402): "Teste é um conjunto de atividades que podem ser planejadas com antecedência e executadas sistematicamente". Seguindo essa mesma linha, Bartié (2002, pag. 22) define que: "Teste é um processo sistemático e planejado que tem por finalidade única a identificação de erros".

Os testes podem ser vistos por perspectivas diferentes, como é representado na figura 1:

Figura 1 - Visões dos testes por perspectivas diferentes



Fonte: BARTIÉ (2002, pag. 21).

Testes de *software* existem não somente para provar que algo está funcionando, mas também para provar que algo não funciona, sendo, às vezes, o segundo caso mais difícil de aplicar do que o primeiro. Por exemplo, um Analista de Sistemas irá focar mais em cenários positivos, sendo que um Analista de Testes, por não estar envolvido emocionalmente no projeto, conseguirá não só avaliar e melhorar os cenários positivos, quanto criar os cenários negativos. Dessa forma, é realizado um esforço adicional, porém com uma chance maior de encontrar problemas que não foram vistos anteriormente. (BARTIÉ, 2002).

2.3 PROCESSO DE TESTES DE *SOFTWARE*

É importante ressaltar a relevância das empresas adotarem um processo de desenvolvimento de *software*, como também se faz necessário elas adotarem e definirem um processo de testes. Sobre esse processo de testes pode-se afirmar que: "[...] deve-se basear-se em uma metodologia aderente ao processo de desenvolvimento, em pessoal técnico qualificado, em ambiente e ferramentas adequadas." (RIOS; MOREIRA, 2006, pag. 8).

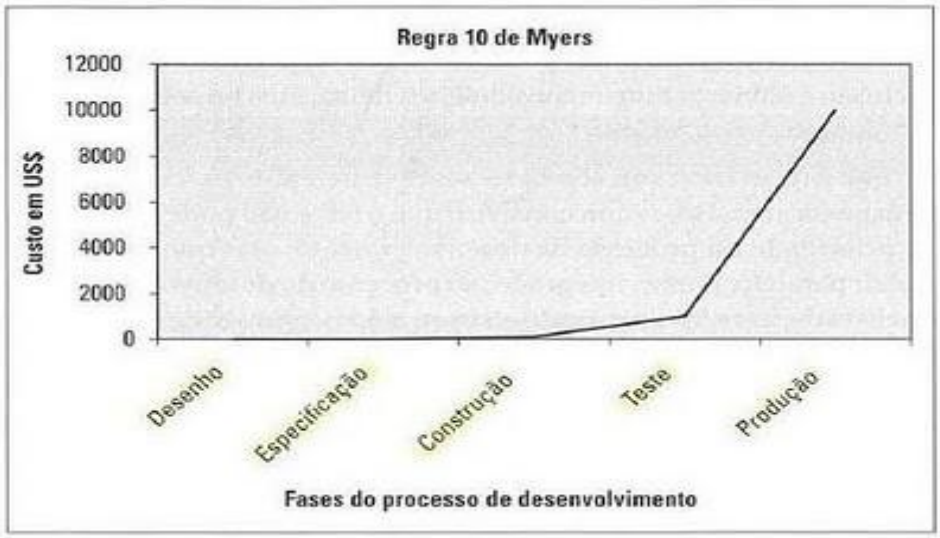
> Como o processo de desenvolvimento cria produtos com defeito, necessário se faz descobrir esses defeitos. Num modelo de garantia de qualidade de *software*, isso é insuficiente. Quem poderia garantir que um *software* testado pelos próprios desenvolvedores está corretamente testado? Com toda certeza, existem exceções,

> mas a melhor maneira de testar um *software* é ter um processo de teste claramente definido. (BASTOS et al., 2007, pag. 17).

Para Bastos et al. (2007), o processo de testes deve andar junto com o processo de desenvolvimento, já que em fases iniciais, como o levantamento de requisitos e modelagem, os erros encontrados nessa fase podem diminuir os custos de correção, otimizando a atividade de testes como um todo. Além disso, devido aos prazos serem curtos e, muitas vezes, pré-definidos, as funcionalidades que trazem maiores riscos ao negócio devem ter prioridade de testes, e caso não seja possível testar tudo, ao menos as partes mais críticas devem ser testadas.

Erros nos *softwares* produzidos, independentemente se são grandes ou pequenos, geram custos, sendo que o custo não é somente "[...] da alocação de tempo e recursos para produzir algo defeituoso, mas também nos danos provocados pelo erro, bem como a sua identificação, correção, teste e implantação da correção [...]" (BARTIÉ, 2002, pag. 30). Considerando essas questões, em 1979 Myers enunciou a, atualmente, chamada regra 10 de Myers. Essa regra pode ser visualizada na figura 2, a seguir:

Figura 2 - Custo da correção dos defeitos



Fonte: BASTOS et al. (2007, pag. 19).

Essa formulação explica que os erros que não são encontrados em cada fase do projeto tendem a ser 10 vezes mais caros para cada fase em que o erro é migrado, sendo que o custo de um erro encontrado em fases iniciais é muito mais baixo que um erro encontrado em produção. (MYERS, 1979, apud BASTOS et al., 2007, pag. 18).

2.3.1 Verificação e validação no processo de testes

Dentro de um processo de testes de *software*, dependendo da sua etapa, os testes podem ser tanto para verificação quanto para validação.

A **verificação,** segundo Bastos et al. (2007, pag. 30), tem como objetivo "[...] realizar inspeções/revisões sobre os produtos gerados pelas diversas etapas do processo de testes", sendo que, para Pressman (2011, pag. 402), a verificação "[...] refere-se ao conjunto de tarefas que garantem que o *software* implementa corretamente uma função específica.". São consideradas revisões de requisitos, modelos, inspeções de código fonte e outras revisões e inspeções como atividades de verificação. (Bastos et al., 2007, pag. 30).

> Compreensiva verificação garante que a performance do *software* e qualidade dos requerimentos são adequadamente testadas e os resultados dos testes possam ser repetidos, mesmo depois de qualquer mudança no *software*. Verificação é um "processo de melhoria contínua" que não tem fim. Deve ser usado para garantir a operação e manutenção do sistema. (MOLINARI, 2003, pag. 24).

Segundo Pressman (2011, pag. 402), "*Validação* refere-se a um conjunto de tarefas que asseguram que o *software* foi criado e pode ser rastreado segundo os requisitos do cliente.". Complementando essa definição, Bastos et al. (2007, pag. 30) diz que testes unitários, de integração, de sistema, de aceitação, homologação e testes de regressão são considerados como atividades de **validação**. Para Molinari (2003, pag. 23):

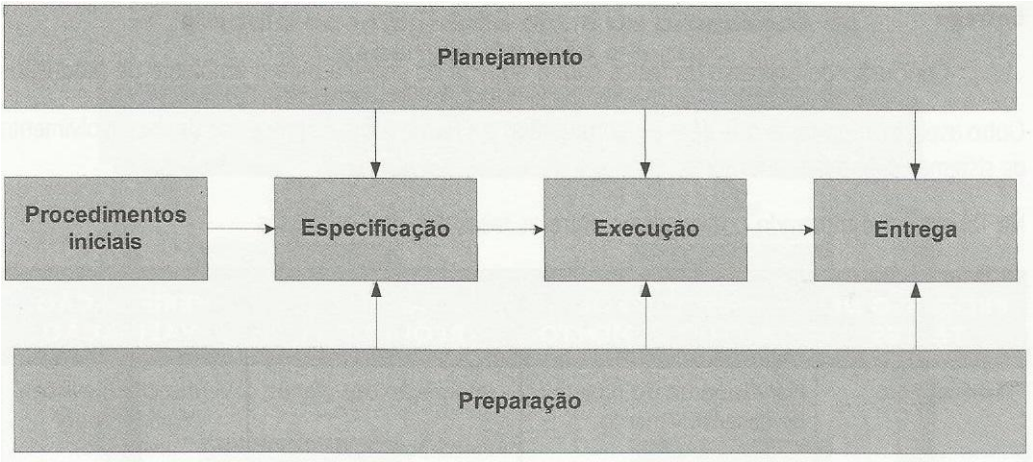
> Tradicionalmente teste de *software* é considerado um processo de validação, isto é, uma fase do ciclo de desenvolvimento do produto. Depois que o programa é terminado, o sistema é validado ou testado para determinar sua funcional e operacional performance.

Além de todas essas definições, essas atividades podem ser respondidas através de duas perguntas: "Estamos criando o produto corretamente?" para a **verificação** e "Estamos criando o produto certo?" para a **validação**. (BOEHM, 1981, apud PRESSMAN, 2011, pag. 402).

2.3.2 Fases do processo de testes

Dentro do processo de testes, as fases são correspondentes a todas as atividades, produtos e documentos gerados. (BASTOS et al., 2007, pag. 44). A figura 3, a seguir, representa as fases do processo de testes:

Figura 3 - Processo de testes 3P x 3E



Fonte: RIOS; MOREIRA (2006, pag. 9).

Dentro de cada fase existem diversas atividades, cada uma delas será brevemente descrita a seguir.

2.3.2.1 Procedimentos iniciais

Nesta fase é "[...] aprofundado um estudo dos requisitos do negócio que dará origem ao sistema de informação a ser desenvolvido, de modo que o mesmo esteja completo e sem nenhuma ambiguidade." (BASTOS et al., 2007, pag. 45). Para Rios e Moreira (2006, pag. 57), opcionalmente poderá ser gerado um Guia Operacional de Testes (GOT), sendo que o mesmo trata de "[...] um acordo entre as partes envolvidas (desenvolvedores, usuários e testadores) visando formalizar o inicio do projeto de testes.".

2.3.2.2 Planejamento

Dentro dessa fase, o objetivo é de "[...] elaborar a Estratégia de Teste e o Plano de Teste a ser utilizados de modo a minimizar os principais riscos do negócio e fornecer os caminhos para as próximas etapas." (BASTOS et al., 2007, pag. 46). Ainda, segundo Bastos et al. (2007, pag. 46), "A atividade de planejamento tem de permanecer ativa até que o projeto seja concluído, visto que se fará necessário avaliar constantemente se os rumos do projeto estão dentro do que foi previsto e planejado.".

2.3.2.3 Preparação

Na fase de preparação, o principal objetivo é de "[...] preparar o ambiente de teste (equipamentos, pessoal, ferramentas de automação, hardware e *software*)" para que os testes sejam executados corretamente. Cada ambiente de teste (desenvolvimento, teste e produção) terá seus tipos de teste, de tal forma que na sua execução se atinja os melhores resultados. (RIOS; MOREIRA, 2006, pag. 117).

2.3.2.4 Especificação

Na fase de especificação, os objetivos básicos serão de tanto elaborar quanto revisar todos os casos de teste e roteiro de teste, uma vez que os mesmos serão elaborados de forma dinâmica durante todo o projeto, na medida em que os módulos ou partes dos sistemas desenvolvidos sejam liberados para tais atividades. (BASTOS et al., 2007, pag. 47).

2.3.2.5 Execução

A fase de execução tem como principal objetivo a execução de testes, que devem estar de acordo com os casos/roteiros de teste, e caso sejam testes automatizados, também

devem ser usados os *scripts* de teste. (BASTOS et al., 2007, pag. 47). Além disso, Bastos et al. (2007, pag. 47) citam que:

> Os testes deverão ser executados integralmente, por regressão ou parcialmente, sempre que surgir alguma mudança de versão dos programas em teste e nos ambientes preparados (desenvolvimento, testes, homologação, produção), conforme previsto na Estratégia e nos Planos de Teste.

Estes testes são mais bem detalhados no decorrer deste trabalho.

2.3.2.6 Entrega

Nesta última fase, tudo o que foi realizado durante o processo será documentado/arquivado, relatando também todas as ocorrências de conformidades e não conformidades encontradas no *software* testado. (BASTOS et al., 2007, pag. 47-48).

2.3.3 Ciclo de vida do processo de testes

Para Bastos et al. (2007, pag. 40-41) devem-se realizar testes ao longo de todo o processo de desenvolvimento, ou seja, o ciclo de vida de testes começa junto com o levantamento de requisitos. Para isso, ambas as equipes de desenvolvimento e de testes devem começar do mesmo ponto, sendo que o primeiro irá capturar e documentar todos os requisitos do sistema e o segundo utilizará esses mesmos requisitos com o objetivo de realizar os casos de teste de *software*. Esse procedimento é chamado de conceito “V” de teste, tem como meta que o “fazer” e “conferir” sejam aplicados do inicio ao fim do projeto. Desse modo, quem “faz” trabalha com o objetivo principal de implementar o sistema e quem “confere” deve de forma simultânea ter a missão de executar testes, tendo o objetivo de minimizar ou eliminar os riscos do *software*. Ainda, conforme Bastos et al. (2007, pag. 41), “Com esse enfoque, se os grupos trabalharem juntos e de maneira integrada, o alto nível de riscos que caracteriza os projetos de desenvolvimento de *software* irá decrescer a um patamar

aceitável que permita a conclusão bem-sucedida do projeto.”. Esse conceito é representado na figura 4, a seguir:

Figura 4 - Conceito "V" de teste de *software*

Fonte: BASTOS et al. (2007, pag. 41).

O ciclo de vida no processo de testes é desmembrado em 11 partes, sendo esses passos apresentados no quadro 1, a seguir. (BASTOS et al., 2007, pag. 43-44):

Quadro 1 - Ciclo de vida do processo de testes

(continua)

Passo	Objetivo	Descrição
1	**Acesso ao Plano de Desenvolvimento**	Serve como pré-requisito para a construção do Plano de Teste. Também tem como objetivo estimar os recursos necessários para a realização dos testes.
2	**Desenvolvimento do Plano de Teste**	Tem como objetivo a preparação do Plano de Teste, mudando o conteúdo em relação ao plano de desenvolvimento.
3	**Inspeção ou teste dos requisitos de *software***	Avalia a qualidade dos requisitos de *software*, usando técnicas de verificação.
4	**Inspeção ou teste do desenho do *software***	Avaliação do desenho interno e externo do *software*, usando técnicas de verificação.
5	**Inspeção ou teste da construção do *software***	Determinação dos tipos de testes e suas extensões a serem usados, conforme método utilizado.

Fonte: Baseado em BASTOS et al. (2007, pag. 43-44).

Quadro 1 - Ciclo de vida do processo de testes

(conclusão)

Passo	Objetivo	Descrição
6	**Execução dos testes**	Testar o código em estado dinâmico, usando as ferramentas e os métodos especificados no Plano de Teste.
7	**Teste de aceitação**	Avaliação do sistema por parte dos usuários.
8	**Informações dos resultados dos testes**	Coletar e registrar todas as informações importantes sobre os testes executados.
9	**Teste da instalação do *software***	Tem como objetivo determinar se o *software* poderá ser implantado em produção sem maiores problemas.
10	**Teste das mudanças no *software***	Avaliar as mudanças que ocorrem no *software* após a sua implantação.
11	**Avaliação da eficácia dos testes**	Realizar a avaliação dos testes executados, tendo a participação de todos os envolvidos no projeto.

Fonte: Baseado em BASTOS et al. (2007, pag. 43-44).

Os cincos primeiros passos utilizam a verificação como principal meio de avaliação do *software* a ser construído, enquanto o restante utiliza a validação para aferir a qualidade durante a construção até a implantação do sistema. (BASTOS et al., 2007, pag. 42).

2.4 TERMINOLOGIA UTILIZADA NESTE TRABALHO SOBRE TESTES

Além de ter um processo de testes de *software* definido, também é muito importante ter um conhecimento básico dos tipos de testes que podem ser aplicados no sistema a ser testado. A esse respeito, os principais autores de artigos e livros possuem definições diferentes sobre os termos de tipos de testes, sendo que alguns os citam como técnicas de testes de caixa branca e caixa preta e outros como estratégias de testes de caixa branca e caixa preta, não havendo um consenso em comum nesse sentido. (MOLINARI, 2003, pag. 157-158).

O autor deste trabalho utiliza as mesmas definições utilizadas por Masuda, dividindo as estratégias de testes em caixa branca, caixa preta e caixa cinza, assim como são definidos os tipos de testes que podem ser utilizados em cada uma dessas estratégias. Além das estratégias e tipos de testes, os testes serão divididos em quatro estágios ou níveis de teste: testes de unidade, integração, sistema e aceitação. (MASUDA, 2009, pag. 38).

2.5 ESTRATÉGIAS DE TESTES DE *SOFTWARE*

Conforme foi apresentado na seção 2.3.1, os testes de **verificação** e **validação** servem para garantir a qualidade de *software*, sendo que o segundo possui diferentes estratégias de testes de *software* para atingir esses objetivos. Segundo Bartié (2002, pag. 104), "A estratégia escolhida determina o modo como iremos estabelecer o problema e como serão conduzidos os procedimentos de testes visando minimizar esforços e ampliar as chances de detecção de erros que estão inseridos no *software"*.

> Uma estratégia de teste de *software* deve acomodar testes de baixo nível, necessários para verificar se um pequeno segmento de código fonte foi implementado corretamente, bem como testes de alto nível, que validam as funções principais do sistema de acordo com os requisitos do cliente. (PRESSMAN, 2011, pag. 402).

Essas estratégias são divididas em três categorias, que são os testes de caixa branca, caixa preta e caixa cinza, A esse respeito, Bartié (2002, pag. 104) diz que:

> A aplicação de testes de caixa preta não exclui a necessidade de aplicarmos os testes de caixa branca e vice-versa. Na verdade, essas estratégias são complementares e não exclusivas, o que significa que teremos um produto de maior qualidade se ambos os processos foram aplicados nas etapas de validação do *software*.

Nas seções, a seguir, são apresentados esses conceitos sobre as estratégias de testes de *software*.

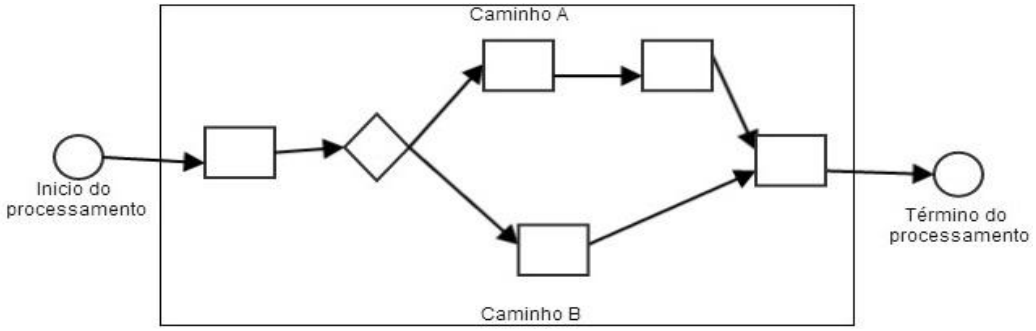
2.5.1 Testes de caixa branca (*White Box*)

Testes de caixa branca, também conhecidos com testes estruturais, são "[...] baseados na arquitetura interna do *software*. Esses testes empregam técnicas que objetivam identificar defeitos nas estruturas internas dos programas através de situações que exercitem adequadamente todas as estruturas utilizadas na codificação." (BARTIÉ, 2002, pag. 42). Seguindo essa definição, Pressman (2011, pag. 431) cita que:

> [...] Usando métodos de caixa-branca, o engenheiro de *software* pode criar casos de teste que (1) garantam que todos os caminhos independentes de um módulo foram exercitados pelo menos uma vez, (2) exercitam todas as decisões lógicas nos seus estados verdadeiro e falso, (3) executem todos os ciclos em seus limites e dentro de suas fronteiras operacionais, e (4) exercitam estruturas de dados internas para assegurar a sua validade.

Molinari (2003, pag. 165) resume que os testes de caixa branca devem "[...] garantir que todas as linhas foram executadas pelo menos uma vez, e estejam corretas.". Essa estratégia de testes é importante pelo fato de que "A estrutura do *software* em si é uma fonte valiosa de informação para selecionar casos de teste e determinar se um conjunto de casos de teste é suficientemente rigoroso." (PEZZÈ; YOUNG, 2008, pag. 230). A figura 5, a seguir, representa a visão sobre essa estratégia de testes:

Figura 5 - Visão sobre testes de caixa branca



Fonte: Baseado em BARTIÉ (2002).

Essa estratégia de testes é bastante valorizada pelo fato de ser eficiente, uma vez que consegue detectar muitos erros, diminuindo assim os custos das correções dos mesmos. No entanto, a sua implementação é mais difícil de realizar do que outras estratégias, como, por exemplo, os testes de caixa preta. Além disso, geralmente esses testes são criados pela

equipe de desenvolvimento, mas nada impede a um profissional da área de testes auxiliar nessa tarefa. Para, isso o mesmo deve ter conhecimento da tecnologia utilizada, além de acesso ao código fonte, estruturas dos bancos de dados e realização dos testes de validação nos componentes de *software*. (BARTIÉ, 2002, pag. 104-105).

2.5.2 Testes de caixa preta (*Black Box*)

Testes de caixa preta ou testes funcionais utilizam-se de "[...] técnicas para garantir que os requisitos do sistema são plenamente atendidos pelo software que foi construído. Não é seu objetivo verificar como ocorrem internamente os processamentos no *software*, mas se o algoritmo inserido no software produz os resultados esperados." (BARTIÉ, 2002, pag. 105). Pressman (2011, pag. 439) complementa essa definição, sendo que "[...] O teste caixa-preta não é uma alternativa aos testes de caixa-branca. Em vez disso, é uma abordagem complementar, com possibilidade de descobrir uma classe de erros diferente daquela obtida com métodos de caixa-branca.".

Segundo Molinari (2003, pag. 167), "O teste de caixa-preta tende a ser aplicado durante as últimas etapas da atividade de teste.", uma vez que é necessário que o *software* esteja construído o suficiente para a realização desses testes. A visão da estratégia de testes de caixa preta pode ser representada na figura 6, a seguir:

Figura 6 - Visão sobre os testes de caixa preta

Fonte: Baseado em BARTIÉ (2002).

A principal vantagem da utilização dessa estratégia é que não necessita de alto conhecimento tecnológico e da arquitetura do sistema a ser testado, facilitando também a

contratação de um profissional de testes, já que a criação dos casos de teste será baseada em requisitos ao invés da estrutura do *software*. Isso não quer dizer que deixa de ser complexo e não exija um grande esforço para a sua utilização, se comparado com os testes de caixa branca. Além disso, geralmente é um ótimo candidato para substituir o seu processo manual por um processo automatizado. (BARTIÉ, 2002, pag. 105-106).

2.5.3 Testes de caixa cinza (*Gray Box*)

Os testes de caixa cinza são utilizados em casos que seria interessante ter conhecimentos tanto da estrutura interna do *software*, não precisando ser necessariamente do código fonte, quanto da estrutura externa, utilizando ambas as técnicas de caixa preta e caixa branca.

Conforme Moreira (2008), essa estratégia é utilizada quando é necessário utilizar tanto o conhecimento da estrutura interna quanto as saídas esperadas para validar o *software* a ser testado. O conhecimento interno pode ser obtido por meio de diagramas ou modelos da arquitetura de *software*, juntando os conhecimentos e aplicando o teste, como, por exemplo, em testes de *WebService* e consultas no banco de dados.

2.6 TIPOS DE TESTES DE *SOFTWARE*

Nesta seção, são apresentados alguns tipos de testes que são utilizados dentro das estratégias de testes de *software*, citadas anteriormente.

2.6.1 Teste de Funcionalidade

Segundo Bartié (2002, pag. 113), os testes de funcionalidade:

> [...] tem por objetivo simular todos os cenários de negócio e garantir que todos os requisitos funcionais estejam implementados. Os testes funcionais exigem profundo conhecimento das regras de negócio de uma aplicação para que todas as variações possíveis sejam simuladas, obtendo o máximo de cobertura dos cenários de negócio.

Esse tipo de teste é utilizado para garantir que o sistema não possua diferenças entre os requisitos em relação ao software construído, de tal forma que se possa representar a aderência do software em relação às regras de negócio. (BARTIÉ, 2002, pag. 131).

2.6.2 Teste de Usabilidade

Os testes de usabilidade têm como objetivo fazer a simulação da utilização do *software* com a visão do usuário final, focando em validar a facilidade de navegação entre telas, utilização de atalhos, como, também, verificar se os textos e mensagens estão fáceis e claros de entender, medindo, assim, o nível de facilidade que um usuário pode utilizar o sistema sem maiores problemas. Também pode avaliar se o *software* auxilia o usuário em algumas operações, como, por exemplo, de desfazer e cancelar operações, impactando diretamente na usabilidade do sistema, que deve tentar sempre ser simples e intuitiva. (BARTIÉ, 2002, pag. 114).

2.6.3 Teste de Carga (*Stress*)

Testes de carga ou testes de estresse têm como objetivo "[...] simular condições atípicas de utilização do *software*, provocando aumentos e reduções sucessivas de transações que superem os volumes máximos previstos para o *software*, gerando contínuas situações de pico e avaliando como o *software* e toda a infraestrutura estão se comportando." (BARTIÉ, 2002, pag. 114).

> Nos testes de estresse, o sistema deve ser executado como seria de fato executado no ambiente de produção. Os testadores usam a documentação-padrão do sistema, e as

> pessoas que entram as transações ou que utilizam o *software* devem ser as mesmas que normalmente o fariam no ambiente de produção. *Softwares on-line* precisam ser testados por um período de tempo prolongado, e os *softwares batch*, testados com o uso de mais de um lote de transações. (BASTOS et al., 2007, pag. 50).

Para Bastos et al. (2007, pag. 49), esse tipo de teste irá indicar que "[...] Se o sistema funcionar adequadamente durante os testes, presume-se que também funcionará adequadamente com os volumes normais de trabalho.".

2.6.4 Teste de Volume

Esse tipo de teste tem como objetivo de verificar quais são os limites de processamento do *software*, a partir da infraestrutura que será utilizada no ambiente de produção. Esse tipo de teste difere-se dos testes de carga pelo motivo de que, ao invés de aplicar oscilações de processamento, o mesmo irá aumentar gradativamente para ter noção do limite que o *software* suporta no momento. Isso é útil para comparar se o volume suportado está próximo ou distante dos requisitos levantados. (BARTIÉ, 2002, pag. 115).

2.6.5 Teste de Configuração (Ambiente)

Os testes de configuração servem para "[...] executar o *software* sobre diversas configurações de *softwares* e *hardwares*. A ideia é garantir que a solução tecnológica "rode" adequadamente sobre os mais variados ambientes de produção previstos na fase de levantamento de requisitos." (BARTIÉ, 2002, pag. 115).

Ainda, conforme Bartié (2002, pag. 115-116), esses testes são geralmente recomendados para sistemas de missão crítica, devendo variar os sistemas operacionais utilizados, navegadores *web* (*web browsers*), *hardwares* e, também, combinando todos esses elementos.

2.6.6 Teste de Compatibilidade (Versionamento)

Segundo Bartié (2002, pag. 116), os testes de compatibilidade têm como meta a execução do *software* em versões diferentes de outros sistemas ou *hardware*, de tal forma que se possa identificar qualquer interface que possa ter sido modificada e cause algum tipo de incompatibilidade com as rotinas que as utilizavam. O objetivo é de garantir que novas versões consigam se comunicar e suportem versões antigas de outros protolocos, aumentando a qualidade do sistema como um todo.

2.6.7 Teste de Segurança

Conforme Bastos et al. (2007, pag. 56), esses testes são necessários "[...] para garantir a confidencialidade das informações e a proteção dos dados contra o acesso indevido de terceiros. A quantidade de segurança fornecida depende dos riscos associados.". Para Bartié (2002, pag. 116), "[...] Esses ataques à segurança do *software* podem ter origens internas ou externas, provenientes de *hackers*, quadrilhas especializadas, profissionais descontentes ou mesmo pessoas com intenções de ganhos ilícitos.", aumentando ainda mais o grau de atenção que se deve dar para esses tipos de testes.

Apesar da importância, os defeitos encontrados nem sempre são simples de serem encontrados, fora que em muitos casos necessita-se de profissionais especializados na área com técnicas avançadas. Com isso, a organização deve avaliar os riscos associados com a segurança e possíveis prejuízos com a falha da mesma, e caso o risco seja muito alto, os testes devem ser encarregados a pessoas especializadas em segurança da informação. (BASTOS et al., 2007, pag. 56-57).

2.6.8 Teste de Performance (Desempenho)

Segundo Bartié (2002, pag. 117), os testes de performance têm como objetivo "[...] determinar se o desempenho, nas situações previstas de pico máximo de acesso e concorrência, está consistente com os requisitos definidos.". Pressman (2011, pag. 420) cita que:

> O teste de desempenho é feito em todas as etapas no processo de teste. Até mesmo em nível de unidade, o desempenho de um módulo individual pode ser avaliado durante o teste. No entanto, o verdadeiro desempenho de um sistema só pode ser avaliado depois que todos os elementos do sistema estiverem totalmente integrados.

Assim, os testes de desempenho devem ser especificados a partir de cenários de testes claros, estabelecendo tempos de respostas factíveis, que vão desde a simulação de vários usuários, utilizando o sistema de forma simultânea, como requisitos de desempenho, tráfego de rede ou a combinação de todos esses elementos. (BARTIÉ, 2002, pag. 117).

2.6.9 Teste de Instalação

Testes de instalação têm como o objetivo validar todos os procedimentos de instalação de uma aplicação, assim como avaliar se possibilita as diversas alternativas identificadas nos requisitos. É recomendado que esse teste seja realizado pelo próprio usuário e que o mesmo verifique que em instalações tanto normais quanto alternativas, o *software* se comporte da maneira adequada e não apresente maiores problemas nesse sentido. (BARTIÉ, 2002, pag. 118).

2.6.10 Teste de Confiabilidade e Disponibilidade

Esse tipo de teste tem como objetivo avaliar e monitorar o *software* durante um determinado período de tempo, de tal forma que essas informações sejam coletadas durante a execução dos testes de sistema e verifiquem se os problemas de interrupção são causados por falhas de infraestrutura (confiabilidade) e o tempo necessário para resolução desse problema (disponibilidade). Geralmente, essas monitorações devem ser realizadas nos testes de aceitação e a chance de incidência de problemas na infraestrutura ficarem maiores devido à utilização dos usuários no sistema, realizando a monitoração contínua e identificação dessas falhas. (BARTIÉ, 2002, pag. 118).

2.6.11 Teste de Recuperação

Conforme Bartié (2002, pag. 119), os testes de recuperação têm como objetivo de:

> [...] avaliar o comportamento do software após a ocorrência de um erro ou de determinadas condições anormais. [...] Os Testes de Recuperação devem também contemplar os procedimentos de recuperação do estado inicial da transação interrompida, impedindo que determinados processamentos sejam realizados pela metade e sejam futuramente interpretados como completos.

Esses testes podem ser realizados com interrupções de acesso à rede, processamento dos servidores e computadores, além da verificação de arquivos que eram para ser gerados e foram cancelados, de tal forma que possa avaliar se os mesmos existem. (BARTIÉ, 2002, pag. 119).

2.6.12 Teste de Contingência

Esse tipo de teste tem o objetivo de avaliar os procedimentos de contingência em uma determinada situação de emergência, avaliando a precisão desses procedimentos. A equipe que fica de plantão é quem deve realizar esses testes, de tal forma que o tempo para realizar o processo será avaliado, garantindo que em situações dessa categoria o processo seja satisfatório e evite maiores problemas. (BARTIÉ, 2002, pag. 119).

2.6.13 Teste de Regressão

Segundo Bartié (2002, pag. 108), os testes de regressão têm como objetivo:

> [...] reexecutar um subconjunto (total ou parcial) de testes previamente executados. Seu objetivo é assegurar que as alterações ou inserções de determinados segmentos do produto (durante uma manutenção ou melhorias implementadas) não afetaram outras partes do produto. Toda nova versão do produto deveria ser submetida a uma nova sessão de testes para detectar eventuais impactos em outras funcionalidades.

Pressman (2011, pag. 404) cita que "O teste de regressão pode ser executado manualmente, reexecutando um subconjunto de todos os casos de teste ou usando ferramentas automáticas de captura/reexecução.", sendo assim um forte candidato e até uns dos principais motivos de se automatizar os testes, o que é o foco deste trabalho.

2.7 ESTÁGIO OU NÍVEL DE TESTES

Conforme foram definidas as estratégias de testes de *software* e os tipos de testes, o estágio ou nível de testes é muito importante para separar e organizar melhor todo o planejamento, construção e execução dos testes. (RIOS; MOREIRA, 2006, pag. 33). Esses estágios são necessários, uma vez que se "[...] assume uma visão incremental do teste,

começando com o teste das unidades individuais de programa, passando para os testes destinados a facilitar a integração de unidades e culminando com testes que usam o sistema concluído." (PRESSMAN, 2011, pag. 407).

A seguir, são apresentados esses conceitos sobre os estágios ou níveis de testes de *software*.

2.7.1 Testes de unidade

Segundo Pressman (2011, pag. 407), esse estágio de teste, também chamado de testes unitários é definido como:

> O *teste de unidade* focaliza o esforço da verificação na menor unidade de projeto do *software* – o componente ou módulo do *software*. Usando como guia a descrição de projeto nível de componente, caminhos de controle importantes são testados para descobrir erros dentro dos limites do módulo.

Molinari (2003, pag. 160) resume esse conceito como "[...] Teste em nível e componente ou classe. É o teste cujo objetivo é um 'pedaço de código'.". Para Sommerville (2011, pag. 148), os testes unitários devem ser projetados visando à cobertura de código nos objetos, realizando testes em todas as operações do mesmo, além de verificar seus atributos e simular os mais diversos eventos que causem alguma mudança nesse objeto, utilizando diferentes entradas para atingir tal objetivo.

Apesar de que os testes unitários possam ser executados de forma manual, Sommerville (2011, pag. 149) recomenda que:

> Sempre que possível, você deve automatizar os testes unitários. Em testes unitários automatizados, pode-se utilizar um *framework* de automação de teste (como JUnit) para escrever e executar os testes de seu programa. *Frameworks* de testes unitários fornecem classes de teste genéricas que você pode estender para criar casos de teste específicos. Eles podem, então executar todos os testes que você implementou e informar, muitas vezes por meio de alguma interface gráfica, sobre o sucesso ou o fracasso dos testes.

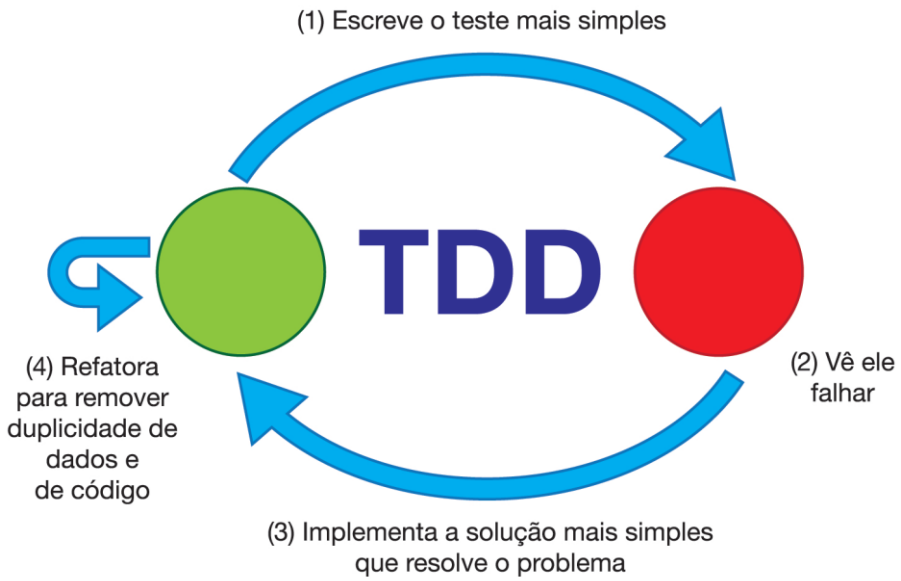
Assim, Pressman (2011, pag. 408) diz que "O teste de unidade normalmente é considerado um auxiliar para a etapa de codificação. O projeto dos testes de unidade pode ocorrer antes de começar a codificação ou depois que o código-fonte tiver sido gerado.".

Conforme foi citado acima, a prática de criar os testes unitários antes do código é conhecida como *Test-Driven Development* (TDD) ou no português, “desenvolvimento dirigido a testes”. Essa prática auxilia o desenvolvedor a criar a implementação do código através dos testes de forma incremental, sendo que essa prática é utilizada na metodologia ágil XP. (SOMMERVILLE, 2011, pag. 155). Conforme Aniche (2012, pag. 25), o ciclo de TDD utiliza os seguintes passos:

- um teste de unidade é escrito para uma nova funcionalidade;
- o teste falha, já que não existe implementação qualquer;
- o código mais simples é implementado para essa funcionalidade;
- o teste agora passa;
- o código sofre refatoração assim que for necessário.

O ciclo do TDD, também conhecido como ciclo *red-green-refactor* (vermelho-verde-refatora), é apresentado na figura 7, a seguir:

Figura 7 - Ciclo do TDD



Fonte: ANICHE (2012, pag. 25).

Conforme Priolo (2009, pag. 248-249), Sommerville (2011, pag. 156) e Aniche (2012, pag. 26), o ciclo de TDD apresenta as seguintes vantagens:

- oferece certa cobertura de código, já que cada segmento do código foi executado pelo menos uma vez nos testes, além de garantir que os defeitos serão encontrados mais cedo;
- o código já nasce testado, desde que o desenvolvedor siga o ciclo de forma correta, isso implica que a cada classe de produção haverá um teste de unidade que valida as suas funções corretamente;
- melhora o *design* das classes, uma vez que não é gerada muita complexidade no código, já que o objetivo é buscar um código simples e com qualidade. Isso também melhora o entendimento tanto da classe a ser testada quanto das classes que interagem com a mesma;
- servem como documentação, já que os testes unitários representam o que a classe deveria fazer e em quais casos a mesma deve funcionar. Essa documentação é muito útil para o entendimento dos desenvolvedores e testadores sobre todo o código criado.

Segundo Sommerville (2011, pag. 149), os testes unitários podem ter dependências de outros objetos, o que pode dificultar os testes, uma vez que esses objetos ainda não estão escritos e/ou podem atrasar a criação dos testes. Para isso, são utilizados *mock objects* para simular o comportamento das classes externas, como, por exemplo, classes que interagem com bancos de dados. A utilização de *mock objects* garante que os testes unitários sejam rápidos e possam ser rodados frequentemente, já que não realizam conexões ou acessos ao disco, além de facilitar os testes em situações anormais ou eventos raros.

A seguir, são apresentados os testes de integração, sendo este um estágio posterior ao dos testes unitários.

2.7.2 Testes de integração

Testes de integração ou também chamados de testes de componentes, “[...] é uma técnica sistemática para construir a arquitetura de *software* ao mesmo tempo que conduz testes para descobrir erros associados com as interfaces.” (PRESSMAN, 2011, pag. 409). Além dessa definição, Molinari (2003, pag. 160) cita que o teste de integração “Garante que um ou mais componentes combinados (ou unidades) funcionam corretamente.”. Para

Delamaro, Maldonado e Jino (2007, pag. 129), os testes de integração são importantes pelo fato de que:

> [...] deve-se ressaltar que os tipos de erro em geral revelados com o teste de integração elevam em muito o custo da atividade de teste se forem detectados nos estágios mais avançados, em especial se a correção do erro forçar modificações em unidades previamente testadas. Desse modo, a realização de testes de integração é de fundamental importância para assegurar uma melhor qualidade do *software* que está sendo construído e reduzir os custos associados.

Dentro do contexto de aplicações que usam o paradigma de OO (Orientado a Objeto), Pressman (2011, pag. 415-416) diz que os testes de integração possuem duas abordagens: sequência de execução e baseado no uso. A primeira utiliza todas as classes que respondem a uma entrada ou evento do *software*, sendo que cada uma dessas classes precisam ser integradas e testadas de forma individual. Após isso, testes de regressão são executados para verificar se não ocorreram problemas. Já, os testes baseados em uso começam pelas classes que podem possuir poucas classes dependentes ou nenhuma, sendo que ao final desses testes, as classes que as utilizam irão sendo testadas até que todo o sistema esteja construído, sendo isso realizado de forma incremental.

O próximo passo, após que os testes de integração forem realizados é a execução dos testes de sistema, sendo este estágio definido a seguir.

2.7.3 Testes de sistema

Segundo Pressman (2011, pag. 421), os testes de sistema caracterizam-se em "[...] uma série de diferentes testes cuja finalidade primária é exercitar totalmente o sistema.". Pezzè e Young (2008, pag. 443) complementam essa afirmação, sendo que "[...] Da mesma forma que o teste unitário e o de integração, o teste de sistema é focado principalmente na descoberta de falhas, mas ao contrário das atividades de teste de nível de granularidade mais fino, se concentra nas propriedades do nível do sistema.".

Alguns dos tipos de testes que são realizados nesse estágio foram definidos anteriormente, como, por exemplo, os testes de funcionalidade (2.6.1), os testes de recuperação (2.6.11), testes de segurança (2.6.7), testes de desempenho (2.6.8) e testes de disponibilidade (2.6.10).

Apesar dos diversos tipos de testes que são executados no sistema, Sommerville (2011, pag. 154) cita que:

> Para a maioria dos sistemas, é difícil saber o quanto o teste de sistema é essencial e quando você deve parar de testar. É impossível fazer testes exaustivos, em que cada sequência possível de execução do programa seja testada. Assim, os testes precisam ser baseados em um subconjunto possível de casos de teste. [...] Como alternativa, podem basear-se na experiência de uso do sistema e em testes de características do sistema operacional.

Ainda, conforme Sommerville (2011, pag. 155), os testes de sistema são mais difíceis de automatizar do que testes de unidade e integração, uma vez que a verificação pode ser dificultada pela necessidade da geração de grandes volumes de dados ou de dados que não possuam certa precisão, dificultando os casos de teste que precisam comparar a saída do sistema com a saída ideal do teste.

A seguir, é apresentado o último estágio dos testes, que são os testes de aceitação.

2.7.4 Testes de aceitação

Neste último estágio, Sommerville (2011, pag. 159) define que os testes de aceitação têm como objetivo:

> [...] que os usuários ou clientes fornecem entradas e conselhos sobre o teste de sistema. Isso pode envolver o teste formal de um sistema que foi aprovado por um fornecedor externo ou processo informal em que os usuários experimentam um produto de *software* novo para ver se gostam e verificar se faz o que eles precisam.

Para Molinari (2003, pag. 161), esse estágio de testes se caracteriza por um "[...] teste exploratório voltado para validar aquilo que o usuário deseja, tendo um objetivo claro: dar o aceite ou não.". Dentro dos testes de aceitação, os mesmos podem ser divididos em dois estágios, sendo estes o teste alfa e o teste beta. (BARTIÉ, 2002, pag. 158-159):

Teste alfa: É caracterizado pelos testes realizados pelos usuários em um ambiente simulado, podendo este estar localizado dentro da empresa criadora do *software*. Esse estágio de teste tem como objetivo observar a naturalidade com que o sistema é utilizado pelos usuários finais, não sendo muito rigoroso em relação aos

procedimentos mais críticos a serem validados. Apesar disso, ele deve ter um conjunto reduzido desses procedimentos, garantindo, assim, um mínimo de cobertura dos testes de aceitação.

Teste beta: Tem como característica a execução do sistema na própria infraestrutura da empresa que vai utilizar o *software* e, assim, será utilizado nas mesmas proporções que são utilizadas diariamente. Esse estágio de teste serve para que os usuários possam ter segurança em utilizar essa versão em produção, contando com o apoio da equipe de desenvolvimento para correção de qualquer problema que possa prejudicar o seu o uso.

2.8 MÉTRICAS DE TESTES DE *SOFTWARE*

Segundo Sommerville (2011, pag. 466), "Uma métrica de *software* é uma característica de um sistema de *software*, documentação de sistema ou processo de desenvolvimento que pode ser objetivamente medido.". Essas métricas, no contexto de testes de *software*, permitem que se observe "[...] o nível de eficiência da detecção de defeitos, medindo o quanto efetivo foi o processo de garantia da qualidade aplicado no projeto." (BARTIÉ, 2002, pag. 220).

Conforme Bartié (2002, pag. 220), as métricas de testes de *software* podem ser divididas em dois indicadores de cobertura dos testes, sendo eles cobertura baseada nos requisitos e na cobertura baseada na estrutura interna (linhas de código) do sistema testado. Esses indicadores auxiliam em estabelecer um nível de risco gerado pela não cobertura dos testes, sejam eles pelos requisitos ou pelas linhas de código, de tal forma que se possam direcionar esforços para o aumento dessa cobertura e consequente melhoria da qualidade do *software*.

Além dessas definições, Rios e Moreira (2006, pag. 111-112) sugerem alguns tipos de métricas e como elas podem ser medidas, sendo essas métricas apresentadas no quadro 2, a seguir:

Quadro 2 - Métricas para testes de *software*

Tipos de métricas	**Métricas do teste de desenvolvimento (testes unitários e de integração)**	**Métricas do teste de execução (testes de sistema e de aceitação)**
Métricas funcionais	- Número de requisitos alocados por teste - % de requisitos por etapa de teste de desenvolvimento	- Número de requisitos verificados - % dos requisitos testados por versão do *software*
Métricas de código	- % do código coberto pelo teste - % do código coberto no maior componente do *software*	- % de código cobertos pelos testes para cada versão do *software*
Métricas de planejamento	- % de término dos testes funcionais de requisito por etapa de teste - % geral de execução dos testes de requisito	- Testes terminados por versão - Etapas de teste já cumpridas ou % de execução - Número estimado de dias para terminar em função da estimativa inicial - Tempo de execução do ciclo completo de teste (tempo total do projeto de teste) - Tempo de término do teste por etapa
Métricas de problemas	- Número de problemas encontrados pelos testes de regressão - Condições extremas testadas para cada teste funcional	- Problemas encontrados por versão testada - Problemas encontrados por componente do *software* - Números de problemas críticos encontrados por versão

Fonte: RIOS; MOREIRA (2006, pag. 111-112).

Apesar de que essas métricas sejam muitos úteis para medir a qualidade de um processo de testes, as mesmas podem ser difíceis de serem coletadas. Sommerville (2011, pag. 467) exemplifica esse problema, já que essas métricas podem ser prejudicadas:

> [...] por fatores subjetivos, como a experiência e a educação do usuário e, portanto, não podem ser medidos objetivamente. Para fazer um julgamento sobre esses atributos, você deve medir alguns atributos internos do *software* (como tamanho, complexidade, etc.) e assumir que estão relacionados com as características de qualidade com as quais você se preocupa.

Para Molinari (2003, pag. 66): "O fator humano influi radicalmente na coleta dos dados da métrica. Ao criar uma métrica devem-se levar em conta quais comportamentos positivos serão encorajados.".

2.9 CASOS DE TESTE

Os casos de teste são muito importantes para realizar a validação de um *software*, sendo que Sommerville (2011, pag. 147) define como "Os casos de teste são especificações das entradas para o teste e da saída esperada do sistema (os resultados do teste), além de uma declaração do que está sendo testado.". Complementando essa definição, Inthurn (2001, pag. 78) cita que:

> Um caso de teste é um documento que descreve uma entrada, ação ou evento e uma resposta esperada, a fim de determinar se uma característica da aplicação está executando corretamente ou não. Um caso de teste deve conter identificação, objetivos, condições de entrada, sequência de passos e resultados esperados.

Segundo Bartié (2002, pag. 122-123), uma das vantagens da utilização de casos de teste seria que:

> É através dos casos de testes que poderemos monitorar os avanços da qualidade de um *software*, avaliando os históricos de cobertura dos testes no decorrer de sucessivos ciclos de interações do desenvolvimento do *software*. Dessa forma, os casos de testes se transformam no elemento mais essencial de um processo de teste de *software*.

Além disso, a norma IEE 829 trata especificamente da documentação desses casos de teste e outros documentos, o que será abordado no decorrer deste trabalho.

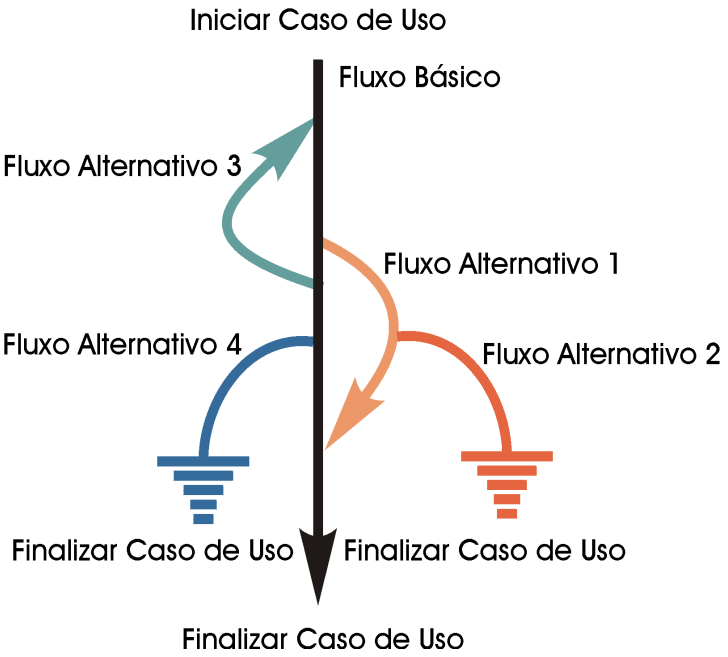
Dadas as definições sobre os casos de teste, também é muito importante conhecer as formas de obtenção desses casos de teste, sendo estes explicados a seguir.

2.9.1 Obtenção dos casos de teste

Para realizar a obtenção dos casos de teste, Bastos et al. (2007, pag. 154) citam que esses casos de teste "[...] costumam derivar de uma especificação formal (caso de uso etc.). É necessário desenvolver casos de teste para cada cenário de caso de uso.". Esses casos de uso são derivados dos requisitos dos usuários e, geralmente, pertencem a um dos tipos de diagramas do padrão UML (*Unified Modeling Language*). (GUEDES, 2011).

Com essas informações do caso de uso, os casos de teste podem ser gerados através do seu fluxo principal e alternativos, além das exceções e regras de negócio descritas nesse diagrama. Assim, haverá condições para elaborar os cenários de testes a partir dos cenários dos casos de uso. (BASTOS et al., 2007, pag. 154-155). Uma representação desse fluxo é apresentada na figura 8, a seguir:

Figura 8 - Fluxo do caso de uso



Fonte: BASTOS et al. (2007, pag. 155).

Outra forma de se elaborar os casos de teste é aquela utilizada nas metodologias ágeis, com o uso de histórias de usuário (*User Story*). Elas são descritas pelo cliente e os testadores possuem o papel de não somente levantar os casos de teste positivos, quanto os negativos, auxiliando a equipe também na elaboração desses casos de teste no desenvolvimento de *software*. (CAETANO, 2012, pag. 14).

Além dos requisitos funcionais, existem também os requisitos não funcionais, como, por exemplo, requisitos de desempenho e segurança que podem não ser extraídos diretamente pelo caso de uso, necessitando que o mesmo seja obtido por meio de especificações suplementares. Esse caso também se enquadra para testes de interface e usabilidade. Caso não seja possível obter nenhum documento de referência, deve-se relatar isso ao cliente, de tal forma que se busque alguma forma de validar os testes criados perante a implementação realizada. (BASTOS et al., 2007, pág. 156).

A seguir, são mostradas algumas das formas que se podem obter os casos de teste para as estratégias de testes de caixa branca e caixa preta.

2.9.1.1 Métodos de elaboração de casos de teste de caixa branca

Esses métodos têm como principal objetivo exercitar da melhor forma a estrutura interna do *software*. Segundo Bartié (2002, pag. 126), “Esses cenários de testes devem ser modelados para que atendam ao maior número de situações, exigindo o menor esforço possível para executá-los.”. Esses métodos são representados no quadro 3, a seguir.

Quadro 3 - Métodos de elaboração de casos de teste de caixa branca

Método	**Descrição do método**
Cobertura de linhas de código	É o método mais tradicional utilizado para cobertura de testes de caixa branca, realizando a medição de linhas que foram executadas dentro de um caso de teste. Por exemplo, se um código fonte possui 100 linhas e 87 delas foram executadas, a cobertura de código foi de 87% nos testes.
Cobertura de caminhos	Complementa a visão simplificada da cobertura por linhas de código, exercitando todos os possíveis caminhos de execução que podem encontrar erros de inicialização de variáveis, fluxos não previstos, etc.
Cobertura de desvios condicionais	Tem como objetivo executar todas as condições lógicas de um código fonte, como, por exemplo, estruturas de *if*, *then*, *else*, *while*, *for*, entre outros.

Fonte: Baseado em BARTIÉ (2002, pag. 126-130).

Conforme o quadro mostrado anteriormente, esses métodos para elaborar os testes de caixa branca são algumas das formas de exercitar o maior número possível de condições, o que visa buscar o maior número de erros, sendo esse um dos objetivos mais importantes dos testes. (BARTIÉ, 2002, pag. 126).

2.9.1.2 Métodos de elaboração de casos de teste de caixa preta

Conforme Bartié (2002, pag. 123), esses métodos servem para identificar que "[...] a maior variedade de cenários permitirá o maior conjunto de simulações que serão avaliadas e comparadas com os requisitos contratados [...]". Esses métodos são representados no quadro 4, a seguir.

Quadro 4 - Métodos de elaboração de casos de teste de caixa preta

Método	**Descrição do método**
Decomposição dos requisitos	Essa decomposição é baseada nos cenários primário (situação básica), alternativo (variações possíveis do cenário primário) e cenários de exceção (possíveis problemas e inconsistências).
Análise de documentos	Utiliza documentos para detalhar os comportamentos e regras de negócios do sistema, além de poder utilizar diagramas UML, como diagramas de atividades e de estados para gerar os casos de teste.

Fonte: Baseado em BARTIÉ (2002, pag. 123-126).

Como mostra o quadro anterior, esses métodos são algumas das formas de garantir que um *software* se comporte como o esperado, utilizando diversas fontes para elaborar os casos de teste, com o principal objetivo de encontrar defeitos. (BARTIÉ, 2002, pag. 123).

2.9.2 Técnicas de refinamento para casos de teste

As técnicas de refinamento servem para identificar o menor número de cenários de testes que consigam cobrir a maior parte das situações. Isso é muito importante pelo fato de aumentar as probabilidades de encontrar defeitos com o menor esforço possível, aumentando, assim, a eficiência do processo, uma vez que podem ser eliminados casos de teste que seriam redundantes em uma situação em particular. (BARTIÉ, 2002, pag. 134).

Algumas dessas técnicas que podem ser utilizadas para a diminuição dos casos de teste são mostradas a seguir.

2.9.2.1 Partição de equivalência

Segundo Bartié (2002, pag. 134), essa técnica pode ser definida como:

> [...] um método que divide o domínio de entrada de dados em classes (grupos de valores). Cada classe representa um possível erro a ser identificado, permitindo que os casos de testes redundantes de cada classe sejam eliminados sem que a cobertura dos cenários existentes seja prejudicada.

Para Pressman (2011, pag. 442), essa técnica é importante no sentido de que "[...] podem ser desenvolvidos e executados casos de teste para o domínio de entrada em cada item de dado. Os casos de teste são selecionados de maneira que o máximo de atributos sejam exercitados ao mesmo tempo.". Um exemplo dessa técnica pode ser visto no quadro 5, a seguir:

Quadro 5 - Exemplo de refinamento por partição de equivalência

Entrada	Valores Permitidos	Classes	Casos de Teste
Idade (esse valor é obtido através da digitação da data de aniversário)	Número entre 18 e 120	18 a 120	Idade = 20
		< 18	Idade = 10
		> 120	Idade = 150

Fonte: BARTIÉ (2002, pag. 135).

Conforme o quadro anterior, a partição por equivalência utiliza um conjunto de valores que, de forma potencial, tem a mesma capacidade de detectar erros, o que dispensa a execução de mais testes com valores distintos que pertencem a uma mesma classe de equivalência. (BARTIÉ, 2002, pag. 134).

2.9.2.2 Análise de valor limite

A análise de valor limite é uma técnica que complementa a participação de equivalência, já que os casos de teste são selecionados nos limites de suas fronteiras de

domínio, podendo, assim, aumentar as chances de que se encontrem erros. Além disso, a diferença é que essa técnica não somente foca a análise das condições de entrada como também das condições de saída. (BARTIÉ, 2002, pag. 135). O quadro 6, a seguir, exemplifica o uso dessa técnica.

Quadro 6 - Exemplo de refinamento por análise de valor limite

Entrada	Valores Permitidos	Classes	Casos de Teste
Idade (esse valor é obtido através da digitação da data de aniversário)	Número entre 18 e 120	18 a 120	Idade = 18 Idade = 120
		< 18	Idade = 17
		> 120	Idade = 121
		Negativa	Idade = - 18 (data futura)

Fonte: BARTIÉ (2002, pag. 135).

Como mostra o quadro anterior, o refinamento por análise de valor limite complementa a partição de equivalência, uma vez que amplia o número de classes, o que por consequência aumenta as chances de encontrar erros. (BARTIÉ, 2002, pag. 136).

2.9.2.3 Probabilidade de erro

Essa técnica é utilizada para identificar situações que causem erros no *software* e, assim, havendo um histórico desses erros mais recorrentes, os casos de teste possam ser enriquecidos com essas informações importantes. Esses erros podem surgir das mais diversas maneiras, como, por exemplo, arquivos e tabelas de banco de dados que estão vazias ou nulas, situações de erro que ocorrem quando não existem informações a serem processadas, erros gerados na primeira execução, valores que estão em branco ou nulos, valores que estão inválidos e/ou negativos, entre outros. (BARTIÉ, 2002, pag. 136-137). A utilização dessa técnica é exemplificada no quadro 7, a seguir.

Quadro 7 - Exemplo de refinamento por probabilidade de erro

Entrada	Valores Permitidos	Classes	Casos de Teste
Idade (esse valor é obtido através da digitação da data de aniversário)	Número entre 18 e 120	18 a 120	Idade = 18 Idade = 120
		< 18	Idade = 17
		> 120	Idade = 121
		Zero	Idade = Zero
		Negativa	Idade = - 18 (data futura)
		Branco	Idade = Nula (data não digitada)
		Inválida	Idade = Inválida (data incorreta)

Fonte: BARTIÉ (2002, pag. 137).

Dadas as técnicas de refinamento dos casos de teste, a seguir serão apresentados os conceitos referentes a automação de testes de *software*.

2.10 AUTOMAÇÃO DE TESTES DE *SOFTWARE*

Segundo Bartié (2002, pag. 196), os testes automatizados são definidos como "[...] a utilização de ferramentas de testes que possibilitem simular usuários ou atividades humanas de forma a não requerer procedimentos manuais no processo de execução dos testes.". Segundo Molinari (2003, pag. 104), a aplicação de testes automatizados "[...] permitirá aumentar a profundidade e abrangência dos casos de testes envolvidos.".

A automação de testes é importante pelo fato de que podem ser executados de forma frequente, além de se executarem mais rapidamente do que os testes manuais. Ainda, serão executados testes de regressão que irão confirmar se novos *bugs* foram introduzidos no sistema. (SOMMERVILLE, 2011, pag. 147).

> Nos últimos tempos, a automação de testes tem se tornado uma atividade vital em projetos de teste de *software*. A maturidade das ferramentas e a vasta quantidade de opções comerciais e *Open Source* são algumas das razões motivadoras para essa mudança de enfoque. Somado a isto, a promessa de testes de regressão repetitíveis com baixo custo e o aumento da cobertura dos testes, reforçam esse súbito interesse na automação de testes. (CAETANO, 2007, pag. 67).

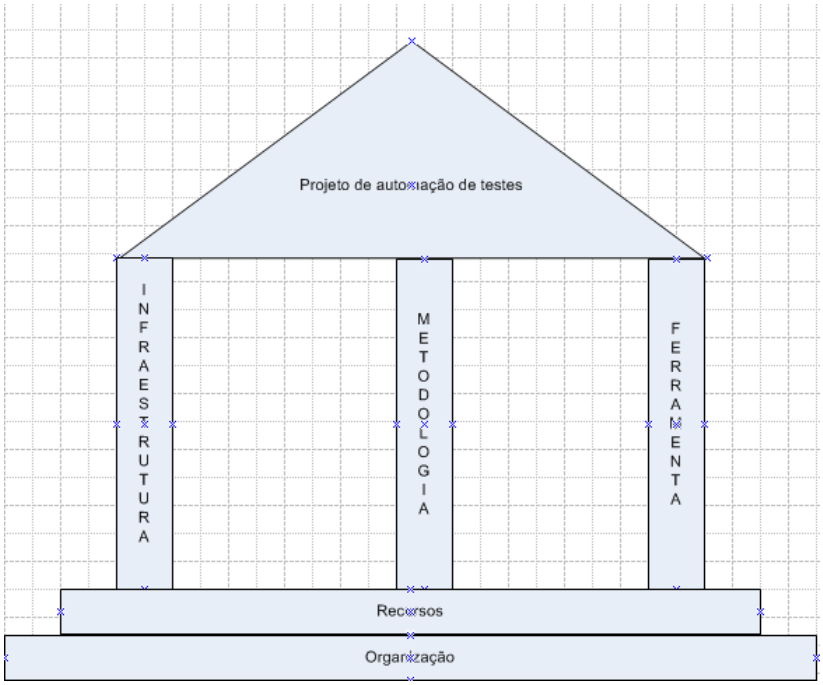
Além dos benefícios da automação de testes de *software*, a necessidade surgiu também pela alta necessidade de utilizar ferramentas que facilitassem os testes em situações que, de forma manual, seriam muito difíceis ou impossíveis de serem aplicadas, como testes de performance e estresse em sistemas cliente/servidor ou sistemas *web*. (RIOS; MOREIRA, 2006, pag. 153). Conforme Bartié (2002, pag. 188), "As ferramentas de automação dos testes possibilitam o desenvolvimento de *scripts* automatizados, de forma a viabilizar um processo de teste com as atividades de entrada e conferência de valores totalmente automatizados.".

Outro motivo para sua utilização é que "A automação exige um esforço inicial de criação, porém possibilita uma incomparável eficiência e confiabilidade, impossível de ser atingida com procedimentos manuais." (BARTIÉ, 2002, pag. 181).

2.10.1 Pilares da automação de testes

Rios e Moreira (2006, pag. 156) esclarecem que a automação de testes está fundamentada em três pilares que são as ferramentas, a metodologia e a infraestrutura. A representação desses pilares da automação de testes é mostrada na figura 9, a seguir:

Figura 9 - Pilares da automação de testes



Fonte: RIOS; MOREIRA (2006, pag. 156).

Ainda conforme Rios e Moreira (2006, pag. 156), a automação de testes acaba sendo um processo de longo prazo e alto custo, então existe uma alta expectativa sobre os seus resultados e benefícios pelas pessoas que dirigem a organização. Os pilares para utilizar a automação de testes dentro de uma organização são:

Ferramentas: Devem ser selecionadas de forma correta, de tal forma que sejam adequadas a tecnologia utilizada e se adaptem aos processos utilizados tanto para o desenvolvimento quanto para os testes.

Metodologia: Devem existir metodologias que são bem consolidadas e utilizadas, seja no desenvolvimento ou testes, sendo que essas ferramentas consigam se adequar a tal metodologia.

Infraestrutura: A mesma deve estar sempre disponível e, se possível, de forma dedicada, garantindo que os projetos de desenvolvimento consigam aplicar os testes automatizados e quaisquer outras necessidades para atingir esse objetivo.

2.10.2 Testes manuais x Testes automatizados

Segundo Molinari (2003, pag. 104), os testes automatizados são muito interessantes de serem aplicados, porém:

> Projetos de automação de testes devem ser implementados com muito cuidado. A seleção da ferramenta de automação de testes, bem como o que deverá ser automatizado ou não, deve ser analisada de forma clara, porque dependendo das técnicas de teste utilizadas, veremos que o retorno de investimento pode ser positivo ou negativo.

Devido a isso, algumas das questões que seriam interessantes de se fazer, quando deve-se focar em testes automatizados ou em testes manuais, são mostradas a seguir. (MOLINARI, 2003, pag. 105).

Para testes automatizados, deve-se focar nos casos em que:

- Testes que são repetidos constantemente, necessitando de técnicas como as de regressão e confirmação.
- Uso de testes aleatórios, que utilizam caminhos aleatórios e necessitam de uma grande quantidade de dados de testes.
- Testes de carga, volume e capacidade, que necessitam simular uma grande quantidade usuários simultâneos no sistema.
- Testes de performance e confiabilidade, sendo que o primeiro é cada vez mais utilizado para sistemas *web*.
- Quais testes de componentes, como, por exemplo, testes unitários, seriam interessantes de automatizar, já que precisam ser executados frequentemente.

Para os testes manuais, pode-se dar foco nos casos em que:

- Quando se faz uso de testes de instalação ou operações que lidam diretamente com *hardware*.
- Quando se faz uso de testes de configuração e de compatibilidade, requerendo, assim, um alto grau de intervenção humana. Outros casos podem ser testes que trabalhem

com tratamento de erros e usabilidade, além de verificações em documentos e opções de *help* do sistema.

Além dessas questões, Lages (2010, pag. 23-24) cita que alguns cuidados devem ser tomados, quando uma organização pretende automatizar os seus testes, já que os mesmos possuem um alto custo na sua construção. Além disso, necessitam de profissionais qualificados e que possuam conhecimentos em programação, dependendo da ferramenta que será utilizada. Outro cuidado seria sobre a automatização prematura, ou seja, tentar automatizar os testes em que os requisitos não estejam bem definidos, aumentando a manutenção de *scripts* de testes e, consequentemente, aumentando os seus custos.

Também se deve evitar automatizar testes que serão executados uma ou poucas vezes, já que o custo não será pago pelo número de execuções do mesmo. Apesar de que os testes automatizados possam parecer a solução ideal para todos os problemas, os mesmos não irão substituir os testes manuais, e, sim, servir como um complemento, já que os testes manuais utilizam a criatividade e percepção humana para a sua execução, algo que ainda não pode ser substituído pela automatização dos testes.

2.11 FERRAMENTAS DE TESTES

Conforme Bartié (2002, pag. 4), o interesse pelo uso de ferramentas de testes começou na década de 90, quando abriu a possibilidade para que determinados tipos de testes, que não eram possíveis de serem executados, pudessem começar a ser feitos graças às ferramentas desenhadas para esses objetivos.

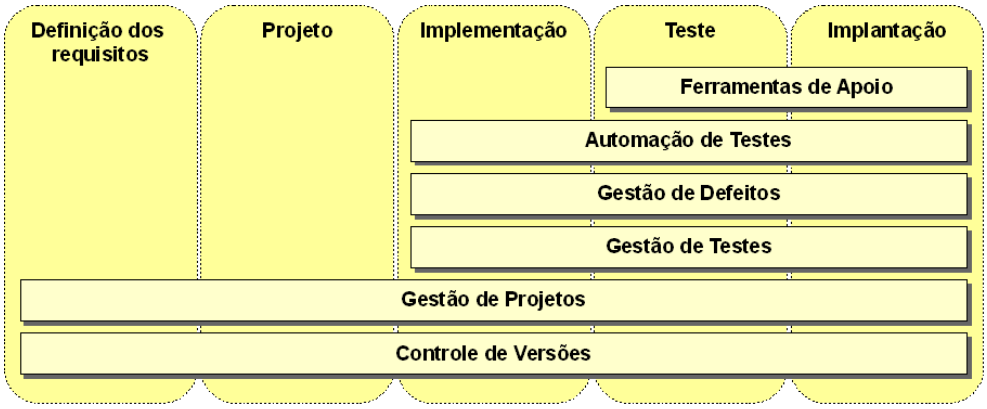
Sem essas ferramentas, muitas atividades eram feitas de forma manual, e não eram eficientes para as organizações. Além disso, tanto os setores de desenvolvimento quanto de testes acabaram gerando muitas informações e diversas atividades exigiam um alto grau de repetição, de tal forma que aumentavam mais ainda o trabalho da execução dos testes, geração de informações e comunicação sobre o resultado dessas execuções. (BASTOS et al., 2007, pag. 86-87).

Com isso, Bartié (2002, pag. 4) afirma que "[...] a aquisição de ferramentas é vital ao sucesso e viabilização de um trabalho desse porte – a implantação de um processo de garantia da qualidade de *software*.". Segundo Bastos et al. (2007, pag. 87):

> A escolha da ferramenta apropriada é um aspecto importante do processo de teste. As técnicas são poucas e muito abrangentes, ao passo que as ferramentas são numerosas e mais restritas. Cada uma delas oferece diferentes opções, projetada para alcançar um objetivo de teste específico.

Devido ao grande número de opções de ferramentas disponíveis no mercado, sejam elas comerciais ou *open source*, essas ferramentas são dividas em algumas categorias, sendo que essas categorias estão representadas na figura 10, a seguir.

Figura 10 - Ferramentas utilizadas ao longo do processo de desenvolvimento



Fonte: CAETANO (2007, pag. 12).

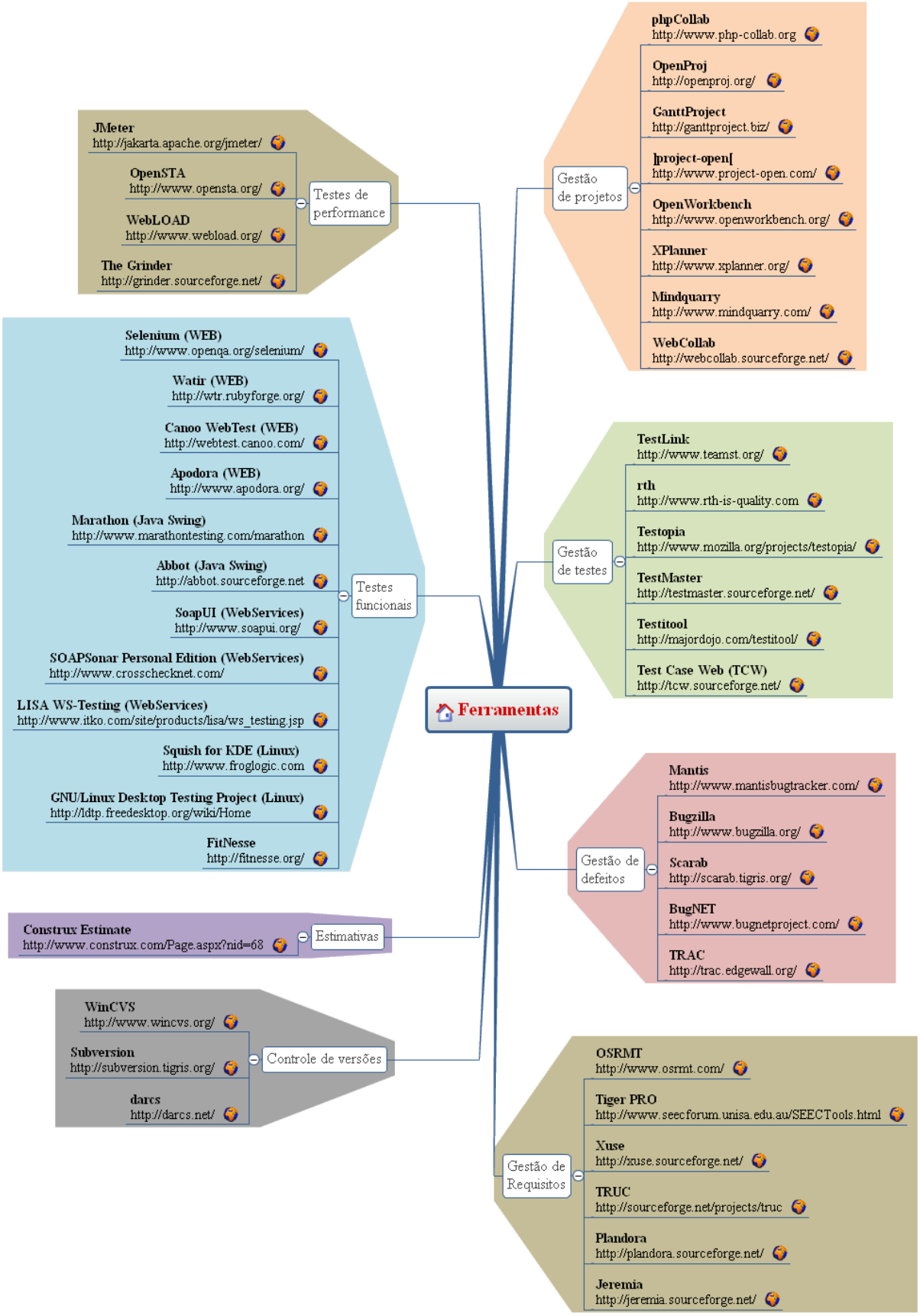
Segundo Molinari (2003, pag. 109-111), as escolhas das ferramentas passam por alguns critérios de escolha, tais como:

- a compatibilidade da ferramenta com o sistema operacional, ambiente da aplicação e sistemas de terceiros que são utilizados na empresa;
- a investigação da ferramenta, ou seja, devem-se verificar quais funcionalidades são oferecidas pela ferramenta, quem irá utilizá-la no dia a dia, quanto tempo será destinado a treinamentos, além da questão de facilidade de uso da mesma;
- os custos que essa ferramenta pode trazer à organização, verificando o quanto e quando o retorno do investimento será alcançado;

- se os fornecedores atendem os requisitos de negócio, utilizando isso como peso para a avaliação e consequente escolha da ferramenta, desqualificando os fornecedores que menos atendem esses requisitos de forma antecipada.

Algumas das opções de ferramentas de *software* livre que podem ser utilizadas nos processo de desenvolvimento e/ou testes são mostradas na figura 11, a seguir.

Figura 11 - Ferramentas utilizadas no processo de testes e desenvolvimento



Fonte: CAETANO (2007, pag. 171).

Como mostra a figura anterior, essas ferramentas *open source* abrangem diversas partes do processo de desenvolvimento, o que inclui a parte de testes, tendo assim o apoio de uma grande comunidade e com liberações frequentes de novas funcionalidades e melhorias. (CAETANO, pag. 168).

2.12 AMBIENTE PARA OS TESTES

O ambiente para os testes deve ser preparado para cada estratégia de teste, já que o ambiente não é somente formado pelo *hardware*, mas também pela massa de dados, sistemas operacionais, ferramentas, navegadores *web* e qualquer outro recurso que dê condições de executar os testes. Esse ambiente deve estar mais próximo de um ambiente de produção, quando se chega aos mais altos estágios ou níveis de testes, como os testes de sistema e aceitação. (BASTOS et al., 2007, pag. 77-78).

Conforme Bartié (2002, pag. 194), existem dois tipos de ambientes de testes, sendo estes o ambiente sob demanda e ativo. O primeiro necessita que todo o ambiente seja criado a cada execução dos testes, sendo que, assim, que esteja terminada essa execução, o ambiente será desfeito, necessitando repetir esse processo toda vez que precisa executar os testes. Já o ambiente de testes ativo é instalado apenas uma vez, de tal forma que o mesmo fique disponível para executar os testes diariamente. Esse ambiente é muito útil para sistemas críticos, que necessitam de um ambiente pronto para validar os testes e mudanças emergenciais no *software*, além da necessidade de ser altamente automatizado.

> A criação, pela equipe de teste, de um ambiente isolado, organizado, representativo e mensurável garante a descoberta de erros reais – ou seja, aqueles que realmente ocorreriam na produção e que porventura não foram descobertos em tempo de desenvolvimento – e, o mais importante, oferece a garantia de que não houve influência externa. (BASTOS et al., 2007, pag. 83).

No quadro 8, a seguir, são mostrados os diferentes atributos que compõem um ambiente de testes, dependendo do seu estágio ou nível de teste.

Quadro 8 - Distribuição do ambiente por atributos

Atributos	Nível			
	Unitário	Integração	Sistema	Aceitação
Escopo	Unidade individual	Unidades integradas	Sistema inteiro	Simulação da aplicação
Equipe	Desenvolvedores	Desenvolvedores e testadores	Testadores	Testadores e usuários
Volume dos dados	Pequeno	Pequeno	Grande	Grande
Origem dos dados	Criação manual	Criação manual	Criação por processos	Dados reais
Interfaces	Não existem	Não existem	Simuladas	Reais/simuladas
Ambiente	Desenvolvimento	Desenvolvimento	Testes	Produção

Fonte: Bastos et al. (2007, pag. 79).

Como mostra o quadro anterior, esse ambiente de testes deve ser discutido já no inicio do projeto, pois ele contempla diversos componentes, como, por exemplo, quais equipamentos, *softwares* e *browsers* serão utilizados. Isso facilita as atividades das fases posteriores, já que será necessário criar a massa de testes e executar os testes nesse ambiente. (BASTOS et al., 2007, pag. 83).

2.13 DOCUMENTAÇÃO DOS TESTES

Segundo Bartié (2002, pag. 210), a documentação dos testes tem como objetivo "[...] registrar todas as atividades relacionadas ao processo de qualidade do *software*. São utilizados como instrumentos de medição e análise, permitindo compreender como o *software* comportou-se durante as fases de verificação e validação.".

Dentre esses documentos, existe o modelo de documentação da norma IEE 829 que estabelece o formato e as informações que devem ser preenchidas, servindo como um modelo para as atividades de verificação e validação do *software* na organização. (KOSCIANSKI; SOARES, 2007, pag. 360).

De forma resumida, a norma abrange os seguintes documentos. (KOSCIANSKI; SOARES, 2007, pag. 360):

1. plano de testes: quem o fará, como, quando, etc.;
2. especificação de projeto de teste: pré e pós-condições do teste;
3. especificação do caso de teste: os dados a serem utilizados;
4. especificação de procedimento: como o teste deve ser realizado;
5. relatório de transmissão de item, quando um componente testado passa da fase de teste a outra fase de desenvolvimento;
6. log de teste: uma narrativa (log) de como foi executado o teste;
7. relatório de incidentes: detalhando resultados esperados e resultados obtidos para testes que falharam;
8. relatório final: um relatório de caráter gerencial, resumindo toda a atividade de testes e efetuando um balanço dos resultados.

Para Bastos et al. (2007, pag. 224-225), os documentos são importantes para um processo de testes, porém nenhum deles será efetivo caso não exista um plano de comunicação bem definido, de tal forma que indique para quem encaminhar e quando esses documentos serão utilizados.

2.14 CONSIDERAÇÕES FINAIS

Neste capítulo foram abordados os seguintes assuntos: qualidade e testes de *software*, processo de testes, estratégias e tipos de testes. Também foram definidos os estágios ou níveis de testes, métricas de testes de *software*, casos de teste, automação de testes, ferramentas e ambiente de testes e, por fim, a documentação para os testes.

Esse levantamento da teoria existente no domínio de testes foi da maior importância para fundamentar o estudo de caso que será realizado. À continuação, é apresentado o método deste trabalho.

3 MÉTODO

Conforme Cervo e Bervian (2002, pag. 23), o método se caracteriza como "[...] a ordem que se deve impor aos diferentes processos necessários para atingir um certo fim ou um resultado desejado. Nas ciências, entende-se por método o conjunto de processos empregados na investigação e na demonstração da verdade.". Marconi e Lakatos (2003, pag. 83) complementam essa afirmação enfatizando que "[...] a utilização de métodos científicos não é da alçada exclusiva da ciência, mas *não há ciência sem o emprego de métodos científicos.*".

> É muito provável que a sabedoria humana não resolva realmente todos os problemas de modo sistemático. Mas depois que o problema é resolvido, o método cientifico é utilizado para explicá-lo e expor a sua solução de um modo ordenado para poder ser compreendido por todos aqueles que estão no processo da produção científica nas universidades, e precisam compreender que a ciência possui um plano formal de desenvolvimento. (OLIVEIRA, 1999, pag. 57).

Com essas definições, a seguir, são apresentadas a caracterização da pesquisa deste trabalho, além das etapas metodológicas e delimitações do mesmo.

3.1 CARACTERIZAÇÃO DO TIPO DE PESQUISA

Segundo Silva e Menezes (2005, pag. 20), a pesquisa pode ser definida como um "[...] conjunto de ações, propostas para encontrar a solução para um problema, que têm por base procedimentos racionais e sistemáticos. A pesquisa é realizada quando se tem um problema e não se têm informações para solucioná-lo.". Além disso, a pesquisa trata da "[...] reunião de informações sobre um tema, sua compreensão, análise, interpretação, comparação, e aplicação a casos semelhantes. Exige reflexão e elaboração nova e pessoal do assunto." (CERVO; BERVIAN, 2002, pag. 186).

Existem várias formas de se classificar uma pesquisa, quanto aos objetivos, ela pode ser definida como pesquisa básica ou aplicada. Este trabalho irá desenvolver uma **pesquisa aplicada**, já que a mesma "[...] objetiva gerar conhecimentos para aplicação prática e dirigidos à solução de problemas específicos. Envolve verdades e interesses locais."

(SILVA; MENEZES, 2005, pag. 20). No presente trabalho, pretende-se resolver o problema de automatizar os testes de *software* de tal forma que os mesmos sejam executados de forma frequente e sofram pouca ou nenhuma intervenção humana.

A forma de abordagem da pesquisa deste trabalho será **qualitativa**.

A abordagem qualitativa é aquela que, segundo Oliveira (1999, pag. 116):

> [...] difere do *quantitativo* pelo fato de não empregar métodos estatísticos como centro do processo de análise de um problema. A diferença está no fato de que o *método qualitativo* não tem a pretensão de numerar ou medir unidades ou categorias homogêneas. [...] Uma das maneiras que os pesquisadores utilizam para transformar dados qualitativos em quantitativos consiste em empregar como parâmetro o uso de critérios, categorias, escalas de atitudes ou, ainda, identificar com que intensidade, ou grau, um determinado conceito, uma opinião, um comportamento se manifesta.

Dessa forma, os critérios para escolha das ferramentas e dos casos de teste, assim como a avaliação da aplicação dos casos de teste depende de alguns critérios subjetivos dependentes do conhecimento e experiência do pesquisador.

Para dar suporte a esta pesquisa aplicada, a pesquisa bibliográfica é utilizada para "[...] recolher informações e conhecimentos prévios acerca de um problema para o qual se procura resposta ou acerca de uma hipótese que se quer experimentar." (CERVO; BERVIAN, 2002, pag. 66). Oliveira (1999, pag. 119) complementa essa definição como "A pesquisa bibliográfica tem por finalidade conhecer as diferentes formas de contribuição científica que se realizaram sobre determinado assunto ou fenômeno.". Assim, essas fontes podem ser obtidas por meio de "[...] publicações avulsas, boletins, jornais, revistas, livros, pesquisas, monografias, teses, material cartográfico etc." (MARCONI; LAKATOS, 2003, pag. 183).

Segundo Miguel (2007, pag. 219), um estudo de caso pode ser definido como "[...] um estudo de natureza que investiga um determinado fenômeno, geralmente contemporâneo, dentro de um contexto real de vida, quando as fronteiras entre o fenômeno e o contexto em que ele se insere não são claramente definidas.".

Além disso, o estudo de caso pode aprofundar o conhecimento por meio de um problema, com o objetivo de estimular a sua compreensão, sugerir hipóteses ou aplicar o conhecimento teórico. (MATTAR, 1996, apud MIGUEL, 2007, pag. 219).

Apesar de que um estudo de caso possa ser flexível, isso não quer dizer que não precise ter algum planejamento prévio, sendo esse planejamento dividido em algumas etapas, as quais são cruciais para o seu sucesso. Dessas etapas, pode-se incluir a modelagem do estudo de caso, a coleta dos dados, a análise desses dados coletados e o relatório desse estudo

de caso. (RUNESON; HÖST, 2009). Cada uma dessas etapas será brevemente descrita a seguir.

Para a modelagem de um estudo de caso, o mesmo deve ter como base todo o referencial bibliográfico, já que o mesmo irá influenciar a abrangência do assunto e também as suas fronteiras, indicando a familiaridade e conhecimento do pesquisador sobre o assunto proposto. (MIGUEL, 2007, pag. 221-222). Além disso, deve-se escolher a(s) unidade(s) que farão parte desse estudo, além dos objetivos e abordagens do estudo. (RUNESON; HÖST, 2009).

Na coleta de dados, o pesquisador pode utilizar diferentes fontes de informação, sendo que o principal motivo disso é de evitar que a interpretação seja limitada em uma única fonte, podendo analisar as diferenças e semelhanças entre elas. (RUNESON; HÖST, 2009, pag. 144). Para Miguel (2007, pag. 224) "A coleta dos dados deve ser dada como concluída quando a quantidade de dados e informações reduzir e/ou quando se considera dados suficientes para endereçar a questão da pesquisa.".

Na etapa de análise de dados, Miguel (2007, pag. 224) afirma que:

> [...] o pesquisador deve então produzir uma espécie de narrativa geral do caso. [...] Geralmente será necessário fazer uma redução dos dados (*data reduction*) de tal forma que seja incluído na análise somente aquilo que é essencial e que tem estreita ligação com os objetivos e constructos da pesquisa.

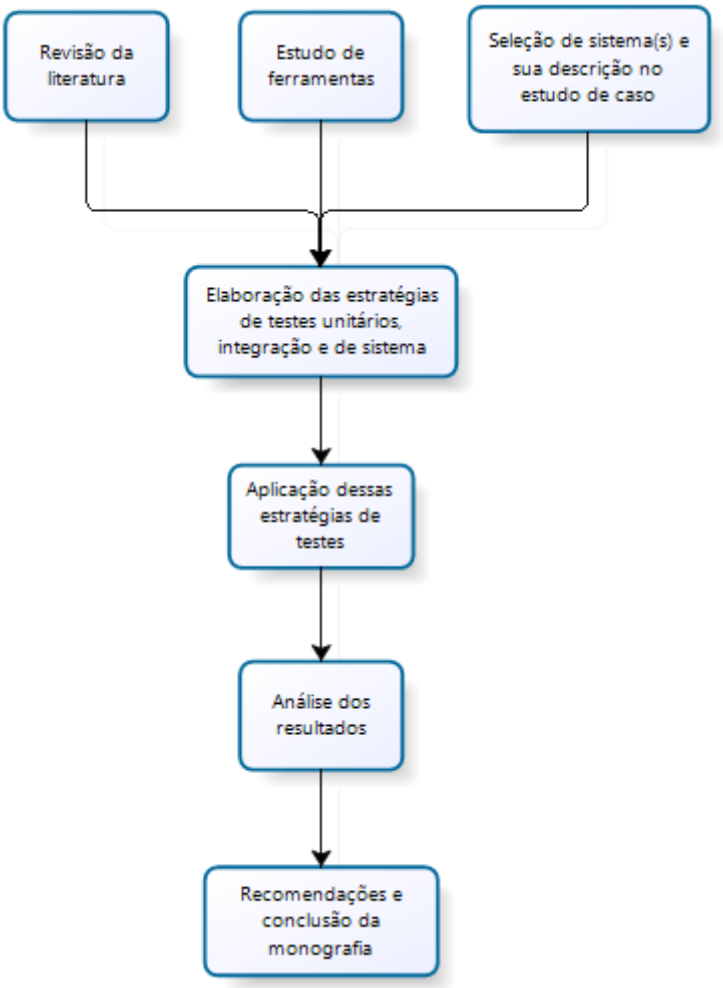
Por último, a elaboração do relatório consiste em comunicar os resultados encontrados no estudo de caso, servindo como fonte principal para avaliar a sua qualidade. Esse mesmo relatório pode ser diferente para cada público, seja ele no meio acadêmico ou na própria indústria, colocando as informações que mais fazem sentido para cada um desses públicos, permitindo também a sua comparação em relação a outros estudos de caso. (RUNESON; HÖST, 2009, pag. 154-155).

Com essa caracterização do tipo de pesquisa, um estudo de caso será aplicado sobre o assunto de testes automatizados, sendo que outros trabalhos são relacionados com esse tipo de estudo, tais como o de Masuda (2010) e mais recentemente dos autores Macedo, Moraes e Catini (2012). As suas etapas metodológicas são descritas a seguir.

3.2 ETAPAS METODOLÓGICAS

As etapas metodológicas são as fases realizadas na pesquisa, sendo estas etapas representadas na figura 12, a seguir:

Figura 12 - Etapas metodológicas do trabalho



Fonte: Elaboração do autor (2014).

Definidas as etapas na imagem apresentada anteriormente, as quatro primeiras são relacionadas com a etapa de modelagem do estudo de caso, sendo que a etapa de aplicação das estratégias de testes é relacionada à etapa de coleta de dados e, por fim, as duas últimas etapas são relacionadas com a etapa de análise de dados. As descrições dessas etapas serão brevemente apresentadas a seguir:

- **Revisão da literatura:** compreende o estudo da teoria de testes de *software*, incluindo as definições de qualidade e testes de *software*, processo de testes, estratégias e tipos de testes, além dos estágios ou níveis de testes, métricas de testes de *software*, casos de teste, automação de testes, ferramentas, ambiente de testes e documentação, conforme foi apresentado no capítulo 2.
- **Estudo de ferramentas:** nesta etapa serão pesquisadas as diversas ferramentas necessárias para a automatização de testes nos níveis de unidade, integração e sistema, sendo assim escolhidas e definidas para o estudo de caso deste trabalho.
- **Seleção de sistema(s) para o estudo de caso:** nesta etapa será definido o sistema que fará parte do estudo de caso, utilizando os seus requisitos e funcionalidades como base para a aplicação de testes automatizados.
- **Elaboração das estratégias de testes:** nesta etapa são definidas as estratégias aplicadas aos testes unitários, de integração e de sistema, assim como a elaboração dos casos de testes e refinamento dos mesmos. Estes serão posteriormente automatizados.
- **Aplicação das estratégias de testes:** nesta etapa são automatizados os casos de teste previamente preparados na etapa anterior.
- **Análise dos resultados:** esta etapa será caracterizada por analisar todos os resultados após a automatização e execução dos testes, como a cobertura de código em testes de caixa branca e o número dos casos de teste de caixa preta que foram executados.
- **Recomendações e conclusão da monografia:** nesta última etapa serão descritas as recomendações do processo de automação de testes de *software*, contendo também as observações do autor deste trabalho. Após isso, será feita a conclusão deste trabalho, sendo este conhecimento obtido da revisão da literatura e também da aplicação do estudo de caso.

3.3 DELIMITAÇÕES

Dentro das limitações deste trabalho, os testes automatizados não são aplicados em uma empresa, sendo que não necessariamente o estudo de caso é aplicado em único aplicativo, além de não serem criados casos de teste para todas as funcionalidades desse

sistema. O(s) sistema(s) nos quais são aplicados os testes podem ser protótipos, não necessariamente sistemas com objetivo comercial. Logo, isso permite a exibição do código fonte, facilitando, assim, o entendimento das ferramentas utilizadas e dos casos de teste criados. Por fim, este trabalho não tem como objetivo em fazer um comparativo na detecção de falhas se comparado com testes manuais.

4 ESTUDO DE CASO

Este capítulo apresenta o desenvolvimento do estudo de caso, iniciando pela descrição das ferramentas utilizadas, a seguir são descritos os casos de teste, assim como a automatização nos níveis de unidade, integração e sistema, entre outros assuntos.

4.1 FERRAMENTAS UTILIZADAS

A seguir, são apresentadas as ferramentas que o autor deste trabalho utilizou para o estudo de caso com testes automatizados. Todas essas ferramentas são *open source*, sendo conhecidas e utilizadas pela comunidade de desenvolvimento de *software*.

Como o estudo de caso é aplicado em um sistema desenvolvido na linguagem de programação Java, foram utilizadas algumas ferramentas que são específicas para essa linguagem.

4.1.1 JUnit[1]

O JUnit é utilizado para a criação de testes de unidade automatizados na linguagem de programação Java. O JUnit possui um mecanismo de comparação (*asserts*) de valores, de tal forma que é possível comparar se uma saída está igual ao resultado esperado de um certo processamento, por exemplo.

Algumas das funcionalidades do JUnit são descritas no quadro 9, a seguir:

[1] JUnit: http://junit.org/ - Versão 4.10

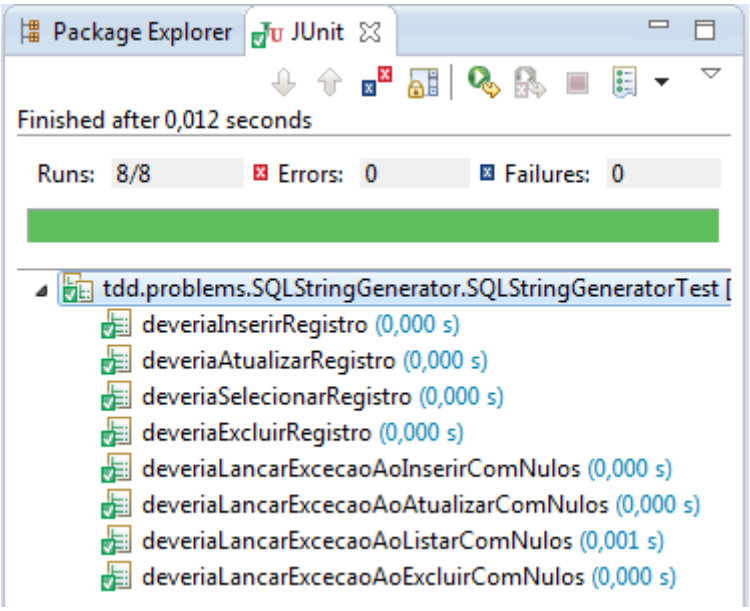
Quadro 9 - Funcionalidades do JUnit

Funcionalidade	Descrição
assertEquals(objetoEsperado, objetoRecebido)	Compara dois objetos quanto a sua igualdade.
assertTrue(condicaoBooleana)	Verifica se um valor está com a condição de verdadeiro (*true*).
assertFalse(condicaoBooleana)	Verifica se um valor está com a condição de falso (*false*).
assertNull(objetoRecebido)	Verifica se o objeto recebido está nulo.
assertNotNull(objetoRecebido)	Verifica se o objeto recebido não está nulo.
fail(mensagem)	O teste falha, exibindo a mensagem definida.

Fonte: Elaboração do autor (2014).

O JUnit é uma das ferramentas mais conhecidas na linguagem de programação Java para a escrita de testes automatizados. Inclusive, ela é acoplada a diversos ambientes de desenvolvimento integrado, assim, conhecida como uma ferramenta IDE (*Integrated Development Environment*). Um exemplo de testes executados com sucesso na IDE Eclipse é demonstrado na figura 13, a seguir:

Figura 13 - Casos de teste executados com sucesso no JUnit

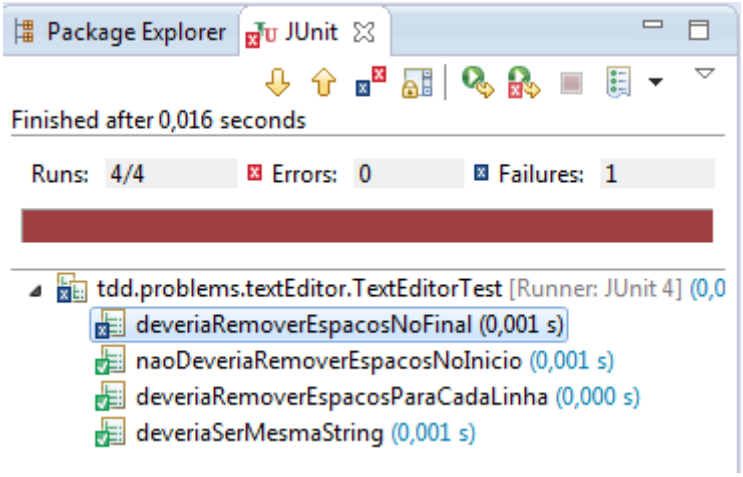


Fonte: Elaboração do autor (2014).

Como mostra a figura acima, a classe SQLStringGeneratorTest possui 8 casos de teste, sendo que todos eles foram executados com sucesso em 0,012 segundos.

Um exemplo de um teste executado com falha no JUnit é mostrado na figura 14, a seguir:

Figura 14 - Caso de teste executado com falha no JUnit



Fonte: Elaboração do autor (2014).

Nessa figura, o método deveriaRemoverEspacosNoFinal acabou falhando, sendo que o JUnit lança uma exceção explicando o motivo da falha. Esse *feedback* é muito importante para entender o porque que esse teste não passou, assim como facilitar a correção do mesmo.

4.1.2 DBUnit[2]

O DBUnit foi criado como uma extensão do JUnit para a criação de testes que envolvam banco de dados relacionais, com várias opções para facilitar a inserção e remoção dos dados de forma segura e que não causem falhas em outros testes devido a dados remanescentes no banco de dados.

[2] DBUnit: http://www.dbunit.org/ - Versão 2.4.8

Essa ferramenta facilita a inserção de dados através de arquivos XML (*eXtensible Markup Language*) que são chamados de *datasets*, podendo ser tanto criados manualmente quanto extraídos de dados existentes nas tabelas do banco de dados, sendo úteis, principalmente, em tabelas que possuam muitos campos e relacionamentos. Isso facilita os testes de integração que precisam interagir com os bancos de dados relacionais, já que eliminam boa parte do trabalho de criar a massa de dados que os casos de teste irão utilizar.

4.1.3 Selenium WebDriver[3]

O Selenium WebDriver tem como principal objetivo automatizar os testes que envolvam um navegador web (*browser*), simulando a interação real de um usuário dentro de um sistema, preenchendo formulários, clicando em botões, etc.

O Selenium WebDriver oferece uma API simples de ser utilizada, além de oferecer suporte a diversos navegadores *web* como Google Chrome, Mozilla Firefox, Internet Explorer e Opera, além de navegadores *web* nativos para sistemas móveis, como Android e IOS. Além disso, essa ferramenta pode ser utilizada em várias linguagens de programação, como, por exemplo, Java, C#, Python, Ruby, PHP, Perl e Javascript.

4.1.4 JaCoCo (*Java Code Coverage*)[4]

O JaCoCo é utilizado para medir a cobertura de código em testes de caixa branca, como, por exemplo, testes unitários e de integração. Com isso, é possível saber se determinados trechos de código que não foram cobertos precisam ser testados, além de servir como uma métrica para medir o percentual de cobertura dos testes.

---

[3] Selenium WebDriver: http://docs.seleniumhq.org/projects/webdriver/ - Versão 2.22.0

[4] JaCoCo: http://www.eclemma.org/jacoco/ - Versão 0.6.5

Essa cobertura é indicada em três cores, sendo que a cor verde indica cobertura total, a cor amarela indica cobertura parcial e, por fim, a cor vermelha indica que aquele trecho de código não foi executado em nenhum caso de teste.

Na figura 15, a seguir, é mostrado um exemplo de trecho de código em que foi verificada a sua cobertura pelo JaCoCo, na IDE Eclipse:

Figura 15 - Exemplo de cobertura de código com JaCoCo

```
    public String atualizar(String tabela, String coluna) {
        if (tabela == null || coluna == null)
            throw new NullPointerException(
                    "Valores nulos não são aceitos em uma atualização de registros!");

        StringBuilder buffer = new StringBuilder();
        buffer.append("update ");
        buffer.append(tabela);
        buffer.append(" set ");
        buffer.append(coluna);

        return buffer.toString();
    }

    public String listar(String tabela, String coluna) {
        if (tabela == null || coluna == null)
            throw new NullPointerException(
                    "Valores nulos não são aceitos em uma listagem de registros!");

        StringBuilder buffer = new StringBuilder();
        buffer.append("select * from ");
        buffer.append(tabela);
        buffer.append(" where ");
        buffer.append(coluna);

        return buffer.toString();
    }
```

Fonte: Elaboração do Autor (2014).

Na figura 16, a seguir, é exibido um relatório gerado na IDE Eclipse sobre a cobertura dos testes que foram executados no projeto:

Figura 16 - Relatório de cobertura gerado pelo JaCoCo

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
TestesUnitarios	82,1 %	656	143	799
src	82,1 %	656	143	799
tdd.problems.SQLStringGenerator	69,5 %	182	80	262
tdd.problems.reloadCountdown	76,8 %	76	23	99
tdd.problems.URLsplitting	86,8 %	132	20	152
tdd.problems.wordWrapping	92,8 %	192	15	207
tdd.problems.textEditor	93,7 %	74	5	79

Fonte: Elaboração do autor (2014).

Conforme mostra a figura anterior, o projeto TestesUnitarios apresenta uma cobertura de 82,1%, assim como também são mostradas as coberturas do *source folder* src e dos pacotes. Nesse caso, também são exibidas a cobertura de cada classe dentro dos pacotes, o que ajuda na visualização do projeto como um todo.

4.2 SISTEMA UTILIZADO PARA O ESTUDO DE CASO

Para o estudo de caso deste trabalho, é utilizado, como exemplo, um sistema *web* que tem como principal objetivo o gerenciamento de receitas, despesas e contas bancárias, podendo realizar operações, como saldo, depósito, transferência entre contas correntes, além de outras funcionalidades. Os requisitos desse sistema são, brevemente, descritos a seguir.

4.2.1 Requisitos do sistema

Os requisitos funcionais levantados para esse sistema de gerenciamento de contas bancárias são os seguintes:

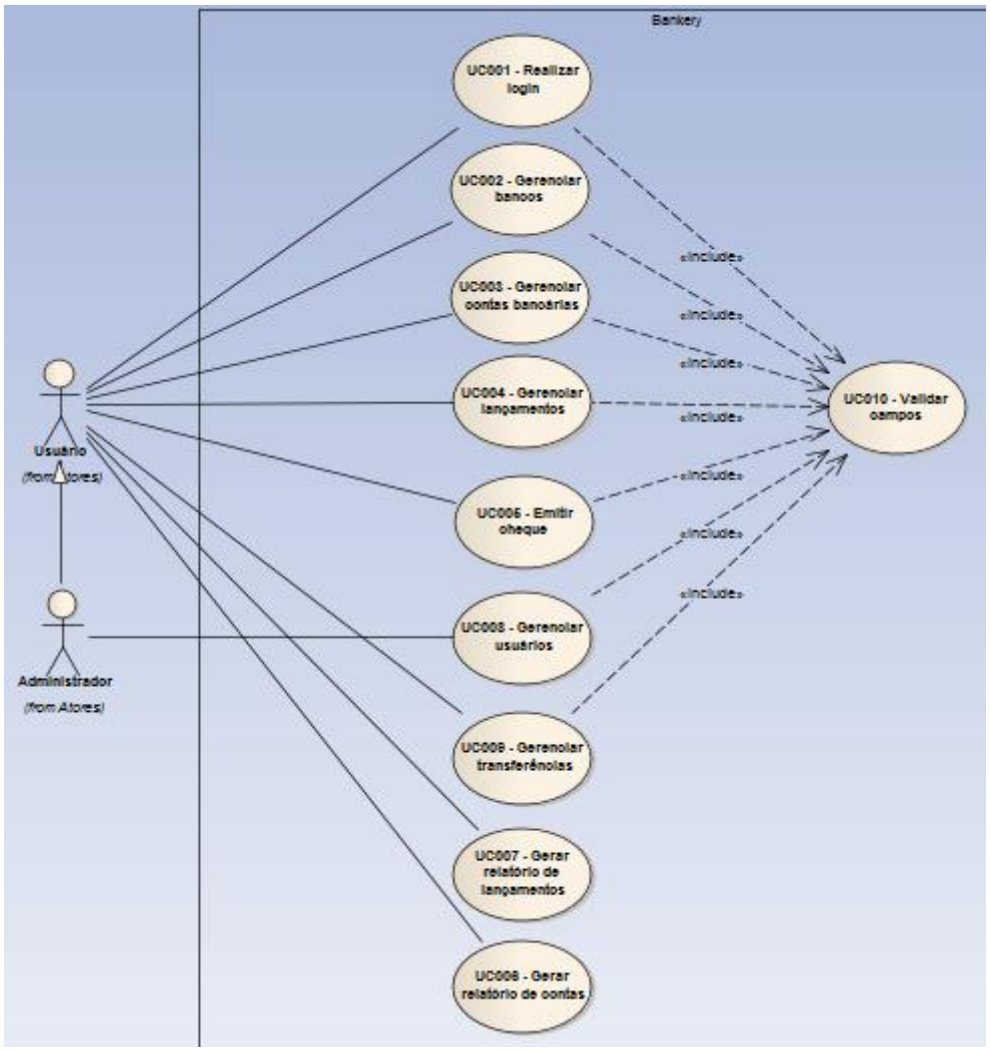
- R1: O sistema irá permitir o gerenciamento das contas correntes que o usuário possui em diferentes bancos.
- R2: O sistema deve permitir lançamentos dos tipos saque e depósito. Devem constar também a descrição desses lançamentos, data, valor da movimentação, tipo e uma observação (motivo do lançamento).
- R3: O sistema deve permitir o controle de transferências realizadas entre contas correntes.
- R4: O sistema deverá controlar a emissão e compensação de cheques referentes a uma conta corrente.
- R5: O sistema permitirá emitir um relatório do saldo atual de todas as contas correntes e um relatório com os lançamentos de uma conta corrente específica.

Com os requisitos definidos, a seguir, são mostrados os casos de uso que foram criados para atender esses requisitos funcionais do sistema.

4.2.2 Casos de uso do sistema

Os casos de uso originados pelos requisitos, citados acima, são demonstrados na figura 17, a seguir:

Figura 17 - Casos de uso do sistema



Fonte: Elaboração do autor (2014).

A figura apresentou todos os casos de uso do sistema. A seguir, a seleção dos casos de uso para o estudo de caso.

4.2.3 Seleção dos casos de uso para o estudo de caso

Para o estudo de caso deste trabalho, foram selecionados dois casos de uso, que são os casos de uso UC004 e UC009, sendo esses referentes às partes de lançamentos e de transferências entre contas, respectivamente.

Esses casos de uso são melhor demonstrados nos quadros 10 e 11, a seguir:

Quadro 10 - Especificação do caso de uso UC004

Nome	Gerenciar lançamentos
Identificador	UC004
Descrição	Gerenciamento dos lançamentos de uma conta
Pré-condição	Ter uma conta cadastrada
Fluxo Principal	1. Usuário clica em cadastrar 2. Usuário preenche os campos 3. Sistema valida os campos (UC010) 4. Sistema grava os dados no banco 5. Sistema exibe mensagem de sucesso e retorna para a tela de listagem
Fluxo alternativo 1a[5]: Editar lançamento	1. Usuário seleciona o lançamento a ser editado 2. Usuário edita os campos 3. Sistema valida os campos (UC010) 4. Sistema edita os dados no banco 5. Sistema exibe mensagem de sucesso e retorna para a tela de listagem
Fluxo alternativo 1b: Excluir lançamento	1. Usuário clica em excluir
Fluxo alternativo 1b 1a: Usuário clica na opção “OK”	1. Sistema exclui os dados no banco 2. Sistema exibe mensagem de sucesso e retorna para a tela de listagem
Fluxo alternativo 1b 1b: Usuário clica na opção “Cancelar”	1. Sistema retorna para a tela de listagem
Fluxo de exceção	1. Sistema exibe mensagem de erro
Fluxo de exceção 1b 4a: Erro ao editar dados	1. Sistema exibe mensagem de erro
Pós-condição	Lançamento cadastrado/atualizado/excluído no sistema
Regras de negócio	RN001, RN002, RN003

Fonte: Elaboração do autor (2014).

[5] Os fluxos alternativos e de exceção estão numerados para indicar em qual passo esse fluxo pode ocorrer.

Quadro 11 - Especificação do caso de uso UC009

Nome	Gerenciar transferências
Identificador	UC009
Descrição	Gerenciamento das transferências de uma conta
Pré-condição	Ter uma conta origem e destino cadastradas
Fluxo Principal	1. Usuário clica em cadastrar 2. Usuário preenche os campos 3. Sistema valida os campos (UC010) 4. Sistema grava os dados no banco 5. Sistema exibe mensagem de sucesso e retorna para a tela de listagem
Fluxo alternativo 1a: Editar lançamento	1. Usuário seleciona a transferência para editar 2. Usuário edita os campos 3. Sistema valida os campos (UC010) 4. Sistema edita os dados no banco 5. Sistema exibe mensagem de sucesso e retorna para a tela de listagem
Fluxo alternativo 1b: Excluir transferência	1. Usuário clica em excluir
Fluxo alternativo 1b 1a: Usuário clica na opção “OK”	1. Sistema exclui os dados no banco 2. Sistema exibe mensagem de sucesso e retorna para a tela de listagem
Fluxo alternativo 1b 1b: Usuário clica na opção “Cancelar”	1. Sistema retorna para a tela de listagem
Fluxo de exceção	1. Sistema exibe mensagem de erro
Fluxo de exceção 1b 4a: Erro ao editar dados	1. Sistema exibe mensagem de erro
Pós-condição	Transferência cadastrada/atualizada/excluída no sistema
Regras de negócio	RN004, RN005, RN006

Fonte: Elaboração do autor (2014).

Além dos casos de uso, também foram definidas seis regras de negócio para atender o sistema desse estudo de caso, sendo isso mostrado no quadro 12, a seguir:

Quadro 12 - Regras de negócio do sistema

Regra de negócio	Descrição
RN001 – Crédito dos lançamentos	Todos os lançamentos do tipo saque não podem ter valor zero, negativo ou acima do saldo atual da conta.
RN002 – Débito dos lançamentos	Todos os lançamentos do tipo depósito não podem ter valor zero ou negativo.
RN003 – Data dos lançamentos	Lançamentos não podem ter data no futuro.
RN004 – Transferência entre contas	Ambas as contas de origem e destino devem estar cadastradas no sistema. Não se pode realizar transferência com a mesma conta para origem e destino ou sem alguma delas.
RN005 – Valores da transferência	Os valores seguem as mesmas regras de crédito e débito dos lançamentos (RN001 e RN002).
RN006 – Data das transferências	Transferências não podem ter data no futuro.

Fonte: Elaboração do autor (2014).

4.2.4 Elaboração dos casos de teste

A seguir, é apresentado de que forma os casos de teste nos níveis de unidade, integração e sistema foram elaborados, considerando também as semelhanças e diferenças para cada nível de teste. Todos os casos de teste criados neste trabalho estão listados no apêndice B.

4.2.4.1 Casos de teste no nível de unidade

Conforme foi explicado na seção 2.7.1, os testes de unidade têm como objetivo testar uma classe de forma isolada, garantindo que essa unidade esteja funcionando conforme o esperado, sendo que isso pode ser especificado pelos requisitos, casos de uso ou pelas regras de negócio.

Com isso, como os casos de uso para lançamentos e transferências são utilizados para esse estudo de caso, foram elaborados 18 casos de teste, sendo nove casos de teste para cada caso de uso.

Para a definição dos casos de teste para os lançamentos, foi levado em consideração que um lançamento altera o saldo atual de uma conta, podendo adicionar valor ao saldo (lançamentos do tipo depósito) ou subtrair valor desse saldo (lançamentos do tipo saque). As transferências seguem a mesma linha, porém com duas contas, uma de origem e outra de destino, sendo que a primeira retira o valor do seu saldo e deposita esse mesmo valor na conta de destino.

Devido ao fato de que se podem ter dois tipos de lançamentos e o número de entradas pode ser muito grande, seria inviável e até desnecessário criar casos de teste cobrindo cada valor de entrada. Para isso, conforme foi explicado na seção 2.9.2.1, a técnica de partição de equivalência foi utilizada, levando em consideração somente as partições com os valores menores que zero, de zero até o valor total do saldo e os valores acima do saldo, diminuindo assim de forma considerável o número de casos de teste.

Para isso, foram criados os casos de teste CT001, CT002 e CT004 para os lançamentos do tipo saque e os casos de teste CT006 e CT007 para os lançamentos do tipo depósito, uma vez que esse tipo de lançamento pode aceitar valores acima do saldo, já que irá somar com o valor do saldo atual. Para as transferências, os casos de testes CT010, CT011 e CT012 também foram definidos com a partição de equivalência, com a diferença que as partições são para a conta de origem. Além desses, outros dois casos de teste criados e que não estão explícitos nem nos requisitos, casos de uso nem nas regras de negócio é que se pode realizar um saque com o valor total da conta, tendo o caso de teste CT005 para lançamentos e CT013 para transferências que cobrem essa situação.

Assim, os outros casos de teste foram criados de acordo com as restrições dadas pelas regras de negócio, como, por exemplo, que o valor do lançamento não pode ser zero (CT003 e CT014), que as duas contas de origem e destino devem existir, não pode utilizar a mesma conta ou sem uma delas para transferir (CT015, CT016, CT017) e que a data de qualquer lançamento não pode ser em uma data no futuro (CT009 e CT018).

4.2.4.2 Casos de teste no nível de integração

Assim como foi explicado no seção 2.7.2, os testes de integração têm como objetivo de testar as interações entre o sistema alvo e a comunicação com sistemas externos, como, por exemplo, bancos de dados e *Web Services.*

Nesse estudo de caso, o sistema externo utilizado é um banco de dados relacional, sendo criados apenas casos de teste que dizem a respeito sobre os lançamentos.

Para isso, foram criados nove casos de teste com objetivo de validar as principais funções do sistema, como, por exemplo, salvar (CT019), atualizar (CT021), excluir (CT023) e listar todos os lançamentos cadastrados (CT024).

Além desses casos, também foram definidos casos de teste para buscar todos os lançamentos de um tipo específico, seja ele saque ou depósito (CT025 e CT026), assim como buscar um lançamento pelo seu ID (CT027).

Por fim, também foram elaborados casos de teste para situações que não devem acontecer, como salvar um lançamento inválido (CT020) e atualizar um lançamento inválido (CT022).

4.2.4.3 Casos de teste no nível de sistema

Neste nível, os casos de teste foram elaborados com base nos casos de uso, conforme foi explicado na seção 2.9.1.2, sendo elaborados 14 casos de testes.

Embora a maioria destes testes seja muito parecida com os casos de teste de unidade, conforme apresentado na seção 2.7.3, os casos de teste funcionais não validam apenas uma unidade ou integração, mas, sim, todo o sistema, incluindo nesse caso a interface gráfica, que, neste estudo de caso, são as páginas *web* de cadastro, edição e listagem dos lançamentos.

Utilizando também a técnica de partição de equivalência, os casos de teste CT028, CT029, CT032, CT034, foram criados, porém com a adição dessas validações na parte de edição dos lançamentos, dando origem aos casos de teste CT038 e CT040. Um caso de teste para exclusão também é considerado para esse nível de teste (CT041), além dos outros casos

para validações em geral, como, por exemplo, campos em branco (CT030 e CT036), datas no futuro (CT031 e CT037) e lançamentos com valor zero (CT033 e CT039).

4.2.5 Automatização dos casos de teste

Com os casos de uso e casos de teste definidos, a automatização desses casos de teste são implementados no estudo de caso, sendo este composto de um projeto em Java *Web*.

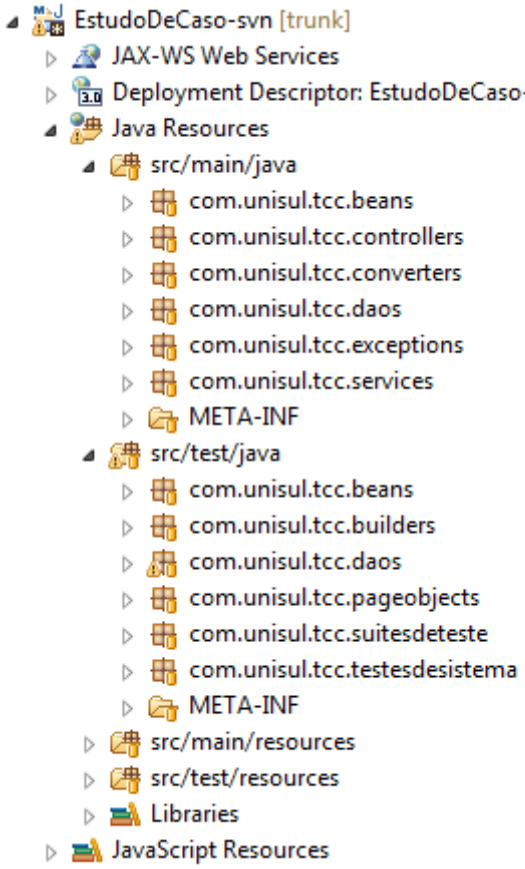
Esse projeto utiliza *frameworks* como JPA (*Java Persistence API*) com Hibernate e *Spring* MVC, utilizando o banco de dados PostgreSQL. Além disso, todos os *frameworks* citados em 4.1 estão inclusos para a criação dos testes.

4.2.5.1 Casos de teste no nível de unidade

Por se tratar do menor nível de teste, todos os casos de teste são criados através da menor unidade do sistema que, no caso da linguagem Java, são as suas classes. Com isso, as classes utilizadas são as classes de Lancamento e Transferencia, que se encontram no pacote com.unisul.tcc.beans e que ficam no diretório src/main/java, lugar onde ficam todas as classes de “produção” do projeto.

Essa estrutura do projeto pode ser visualiza na figura 18, a seguir:

Figura 18 - Estrutura do projeto do estudo de caso



Fonte: Elaboração do autor (2014).

Assim, essas duas classes são responsáveis por fazer um lançamento e uma transferência, respectivamente. Dentro desse contexto, no diretório src/test/java foram criadas duas classes de testes: LancamentoTest e TransferenciaTest. Essa nomenclatura pode ser diferente de projeto para projeto, não tendo um padrão pré-definido. Por questões de padrão, o pacote dessas classes de testes é o mesmo da classe de produção, ficando também em com.unisul.tcc.beans, sendo o principal motivo disso o de facilitar a localização das classes que estão sendo testadas.

A estrutura inicial da classe LancamentoTest é mostrada na figura 19, a seguir:

Figura 19 - Estrutura inicial da classe LancamentoTest

```
18 public class LancamentoTest {
19     private Conta conta;
20
21     @Before
22     public void setUp() {
23         conta = new Conta();
24         conta.setNome("Conta corrente");
25         conta.setNumeroAgencia(1234);
26         conta.setNumeroConta(13456698);
27
28         Banco banco = new Banco();
29         banco.setNome("Santander");
30
31         conta.setBanco(banco);
32     }
```

Fonte: Elaboração do autor (2014).

A linha 21 da classe LancamentoTest contem uma anotação *@Before* no método setUp(), sendo que essa anotação é responsável por criar uma conta a cada caso de teste executado, com o principal objetivo de facilitar a criação das pré-condições para os casos de teste, além de diminuir a repetição de código.

Definido isso, o próximo passo é a automatização do caso de teste em si, criando um método anotado com *@Test*, sendo que o JUnit entende que cada método com essa anotação é um caso de teste e que o mesmo deve ser executado. O caso de teste CT001 automatizado com JUnit é apresentado na figura 20, a seguir:

Figura 20 - Automatização do caso de teste CT001

```
34     @Test
35     public void deveRealizarUmSaque() {
36         criarDepositoInicial();
37
38         Lancamento saque = new CriadorDeLancamento()
39                 .paraAConta(conta)
40                 .comADescricao("Pagamento de conta de luz")
41                 .noValorDe(150d)
42                 .naDataDeHoje()
43                 .doTipo(SAQUE)
44                 .construir();
45
46         saque.lancar();
47
48         double saldoAtualEsperado = 4850d;
49
50         assertEquals(saldoAtualEsperado, conta.getSaldoAtual());
51     }
```

Fonte: Elaboração do autor (2014).

A criação de um caso de teste automatizado não é muito diferente de um caso de teste comum, pois é necessário definir uma pré-condição, que já foi definido pelo método setUp(), um cenário, sendo esse composto pelo método criarDepositoInicial(), que consiste em um método auxiliar para criar um depósito inicial na conta no valor de R$ 5.000,00. Após isso, um saque no valor de R$ 150,00 é criado e logo, em seguida, é invocado o método lancar(), podendo esse ser entendido como a ação necessária para criar um lançamento e alterar o valor do saldo atual da conta. Como foi lançado um saque de R$ 150,00 com o saldo de R$ 5.000,00, o resultado esperado seria de R$ 4.850,00, sendo isso representado pela variável saldoAtualEsperado na linha 48.

Por fim, na linha 50 é chamado o método assertEquals() que compara se o saldo atual da conta é igual ao valor dessa variável e, caso seja, o teste passa com sucesso, do contrário o JUnit irá informar que alguma coisa aconteceu de errado na execução desse teste.

Para facilitar a criação dos cenários de testes, foi utilizado o padrão *Test Data Builder*[6] que funciona de forma parecida com o padrão de projeto *Builder*, porém sendo esse mais específico para facilitar a construção dos cenários para os casos de teste, além de utilizar o conceito de *fluent interfaces* que facilitam a leitura pelas chamadas dos métodos feitas de forma encadeada, conforme é mostrado nas linhas 38 até 44, que utiliza a classe CriadorDeLancamento. Isso também facilita a manutenção, já que apenas uma classe fica com a responsabilidade de construir os cenários, diminuindo o número de mudanças nas classes de teste.

Outro exemplo dessa classe de teste pode ser visto no teste que representa o caso de teste CT002, como mostra a figura 21, a seguir:

Figura 21 - Automatização do caso de teste CT002

```
53      @Test(expected = SaqueInvalidoException.class)
54      public void naoDeveRealizarUmSaqueComValorNegativo() {
55          Lancamento saque = new CriadorDeLancamento()
56                  .paraAConta(conta)
57                  .comADescricao("Lancamento com valor negativo")
58                  .noValorDe(-15d)
59                  .naDataDeHoje()
60                  .doTipo(SAQUE)
61                  .construir();
62
63          saque.lancar();
64      }
```

Fonte: Elaboração do autor (2014).

Nesse caso de teste, o JUnit espera uma exceção da classe SaqueInvalidoException no atributo *expected*, sendo que o teste passa, quando essa exceção é lançada em algum lugar na classe de Lancamento. Essa exceção é lançada quando o valor do lançamento é negativo, tendo, nesse caso, um valor de -15. O ponto de onde isso é chamado na classe de Lancamento é mostrado na figura 22, a seguir:

Figura 22 - Código que lança a exceção SaqueInvalidoException

```
150     private void sacar() {
151         if (valor <= 0) {
152             throw new SaqueInvalidoException(
153                     "Saque não pode ser feito com valores menores ou iguais a zero!");
154         }
```

Fonte: Elaboração do autor (2014).

O caso de teste espera que essa exceção seja lançada, quebrando o fluxo do sistema e evitando que se tenha um lançamento com valor negativo, não sendo necessário fazer mais algum tipo de validação posteriormente.

Na classe de TransferenciaTest, o caso de teste CT010 é mostrado na figura 23, a seguir:

Figura 23 - Automatização do caso de teste CT010

```
38      @Test
39      public void deveRealizarTransferencia() {
40          criarDepositoInicialDaContaOrigem(2500d);
41          criarDepositoInicialDaContaDestino(3500d);
42
43          Transferencia transferencia = new CriadorDeTransferencia()
44                  .daConta(contaOrigem)
45                  .paraAConta(contaDestino)
46                  .comValorDe(500d)
47                  .naDataDeHoje()
48                  .construir();
49
50          double saldoEsperadoContaOrigem = 2000d;
51          double saldoEsperadoContaDestino = 4000d;
52
53          transferencia.transferir();
54
55          assertEquals(saldoEsperadoContaOrigem, contaOrigem.getSaldoAtual());
56          assertEquals(saldoEsperadoContaDestino, contaDestino.getSaldoAtual());
57      }
```

Fonte: Elaboração do autor (2014).

---

[6] *Test Data Builder*: http://www.c2.com/cgi/wiki?TestDataBuilder

Nesse caso de teste, inicialmente são criados os depósitos iniciais para as contas de origem e destino nos valores de R$ 2.500,00 e R$ 3.500,00, sendo que essas contas são criadas previamente para cada caso de teste no método setUp() anotado com *@Before*. Em seguida, uma transferência é criada no valor de R$ 500,00 e o método transferir() da classe Transferencia é chamado e, para que o teste passe, o saldo atual da conta origem deve ser igual a R$ 2.000,00 e na conta destino o saldo atual deve ser igual a R$ 4.000,00, sendo isso validado nas linhas 55 e 56, respectivamente.

Outro caso de teste nessa classe seria o CT017, como mostra na figura 24, a seguir:

Figura 24 - Automatização do caso de teste CT017

```
149     @Test(expected = TransferenciaInvalidaException.class)
150     public void naoDeveRealizarTransferenciaComMesmaContaDeOrigemEDestino() {
151         Transferencia transferencia = new CriadorDeTransferencia()
152                 .daConta(contaOrigem)
153                 .paraAConta(contaOrigem)
154                 .comValorDe(250d)
155                 .naDataDeHoje()
156                 .construir();
157
158         transferencia.transferir();
159     }
```

Fonte: Elaboração do autor (2014).

Esse caso de teste espera que seja lançada a exceção TransferenciaInvalidaException ao realizar uma transferência com uma mesma conta para origem e destino, sendo que essa exceção é lançada no seguinte trecho de código, como mostra a figura 25, a seguir:

Figura 25 - Código que lança a exceção TransferenciaInvalidaException

```
103         if (contaOrigem.equals(contaDestino)) {
104             throw new TransferenciaInvalidaException(
105                     "Conta origem e destino não podem ser iguais!");
106         }
```

Fonte: Elaboração do autor (2014).

Todos os casos de teste elaborados que foram citados, na seção 4.2.4.1, foram automatizados com o auxílio do JUnit no projeto, sendo que todos eles podem ser vistos nos códigos fonte disponibilizados pelo autor.

4.2.5.2 Casos de teste no nível de integração

A automação dos casos de teste de integração sofrem algumas mudanças em relação aos casos de teste de unidade, sendo a principal delas é a necessidade de se ter implementações completas das classes que precisam integrar com outros sistemas. Neste estudo de caso, a integração é realizada com o banco de dados PostgreSQL. Como era necessário que a classe de Lancamento fosse implementada antes, os testes automatizados, neste estudo de caso, são da classe LancamentoDAO, sendo essa responsável unicamente por se comunicar com o banco de dados. Devido a essa responsabilidade única de se comunicar com um sistema externo, não faz sentido ter testes unitários dessa classe, é muito mais útil e conveniente ter um teste de integração que garanta o funcionamento dessa classe.

Com isso, a classe LancamentoDAOTest foi criada no mesmo padrão das classes de teste de unidade, que se encontra no pacote com.unisul.tcc.daos e também no diretório src/test/java.

Além dessa diferença de implementação das classes, na classe de teste, é utilizado um método setUp() anotado com *@Before* e um método tearDown() anotado com *@After*. Esse segundo método tem como responsabilidade de informar ao JUnit que o mesmo deve ser executado após terminar de executar cada caso de teste. A estrutura desses dois métodos é mostrada na figura 26, a seguir:

Figura 26 - Estrutura inicial da classe LancamentoDAOTest

```
33 public class LancamentoDAOTest {
34     private LancamentoDAO dao;
35     private IDatabaseConnection dbConn;
36     private IDataSet contasDataSet;
37     private IDataSet lancamentosDataSet;
38     private EntityManagerFactory factory;
39
40     @Before
41     public void setUp() throws Exception {
42         factory = Persistence.createEntityManagerFactory("test");
43         EntityManager entityManager = factory.createEntityManager();
44
45         dao = new LancamentoDAO(factory);
46
47         Session session = entityManager.unwrap(Session.class);
48         SessionFactoryImplementor sfi = (SessionFactoryImplementor) session.getSessionFactory();
49         ConnectionProvider cp = sfi.getConnectionProvider();
50         Connection conn = cp.getConnection();
51
52         dbConn = new DatabaseConnection(conn);
53         contasDataSet = new FlatXmlDataSetBuilder().build(new File("contas-dataset.xml"));
54         lancamentosDataSet = new FlatXmlDataSetBuilder().build(new File("lancamentos-dataset.xml"));
55
56         DatabaseOperation.CLEAN_INSERT.execute(dbConn, contasDataSet);
57         DatabaseOperation.CLEAN_INSERT.execute(dbConn, lancamentosDataSet);
58     }
59
60     @After
61     public void tearDown() throws Exception {
62         DatabaseOperation.DELETE_ALL.execute(dbConn, lancamentosDataSet);
63         DatabaseOperation.DELETE_ALL.execute(dbConn, contasDataSet);
64
65         dbConn.close();
66         factory.close();
67     }
```

Fonte: Elaboração do autor (2014).

Como mostrado na figura acima, o método setUp() carrega um arquivo específico do JPA com as configurações da base de testes e uma classe que representa uma conexão com o banco de dados é passado a classe DataBaseOperation do DBUnit. Após isso, são carregados dois arquivos, que são os *datasets* de contas e lançamentos utilizados para os testes, terminando o método com a chamada da classe DataBaseOperation que irá limpar os dados nas tabelas de contas e lançamentos e inserir os dados de cada *dataset* antes da execução de cada caso de teste.

Já, o método tearDown() tem como objetivo de excluir todas as tabelas e dados, utilizando os *datasets* como referência para essa operação, assim como fechar as conexões da base de dados e da própria conexão do DBUnit. Como o JPA utiliza o recurso de ORM (*Object-relational mapping*) ou mapeamento objeto-relacional, a manipulação da classe de Lancamento é espelhada com a tabela lancamentos que se encontra no banco de dados, assim como a classe Conta está mapeada com a tabela contas no PostgreSQL e assim por diante. A utilização desses dois métodos para testes de integração são muito importantes para evitar que

os dados estejam válidos para um caso de teste fiquem inválidos para outros, assim como deixar a base em um estado sempre disponível para executar os testes sem maiores problemas.

Os arquivos de *dataset* são arquivos XML que representam as tabelas como *tags* e as colunas e seus valores como atributos dessas *tags*. Os dois datasets utilizados nessa classe de teste são mostrados nas figuras 27 e 28, a seguir:

Figura 27 - Conteúdo do arquivo contas-dataset.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <dataset>
3      <bancos id="1" nome="Santander"/>
4      <bancos id="2" nome="Bradesco"/>
5      <bancos id="3" nome="Banco do Brasil"/>
6      <bancos id="4" nome="Caixa Econômica"/>
7      <bancos id="5" nome="HSBC"/>
8      <usuarios id="1" nome="João da Silva" data_nascimento="1992-05-12" login="admin" senha="teste"/>
9      <contas id="1" nome="Conta corrente PF" numero_agencia="123" numero_conta="456" saldo_atual="0" id_usuario="1" id_banco="1"/>
10 </dataset>
```

Fonte: Elaboração do autor (2014).

Figura 28 - Conteúdo do arquivo lancamentos-dataset.xml

```
1  <?xml version='1.0' encoding='UTF-8'?>
2  <dataset>
3      <lancamentos id="1" descricao="Mercado" data_lancamento="2013-11-14" valor="200" tipo_lancamento="SAQUE" id_conta="1"/>
4      <lancamentos id="2" descricao="Salário" data_lancamento="2013-12-14" valor="1000" tipo_lancamento="DEPOSITO" id_conta="1"/>
5      <lancamentos id="3" descricao="Mercado" data_lancamento="2013-12-14" valor="500" tipo_lancamento="SAQUE" id_conta="1"/>
6      <lancamentos id="4" descricao="Conta de luz" data_lancamento="2013-12-14" valor="90.25" tipo_lancamento="DEPOSITO" id_conta="1"/>
7  </dataset>
```

Fonte: Elaboração do autor (2014).

Nesse estudo de caso, a massa de dados foi dividida em dois arquivos por questões de organização, mas não existe um padrão pré-definido, podendo criar um arquivo de *dataset* por caso de teste, caso seja necessário. Assim, o DBUnit irá inserir os dados linha por linha, sendo, que nesse, caso irá criar a tabela bancos e inserir os dados primeiro e assim por diante, até que toda a massa de dados necessária para os testes esteja pronta, podendo, assim, seguir para a execução dos casos de teste.

Uma funcionalidade muito importante para o sistema seria o de salvar um lançamento, que é validado pelo caso de teste CT019, sendo que a sua implementação na classe LancamentoDAOTest é mostrada na figura 29, a seguir:

Figura 29 - Automatização do caso de teste CT019

```
69      @Test
70      public void deveSalvarUmLancamento() {
71          Lancamento lancamento = new CriadorDeLancamento()
72                  .comADescricao("Compra no mercado")
73                  .noValorDe(200d)
74                  .doTipo(SAQUE)
75                  .naDataDeHoje()
76                  .comAObservacao("teste")
77                  .construir();
78
79          dao.salvar(lancamento);
80
81          List<Lancamento> lancamentos = dao.listarTodos();
82
83          Lancamento ultimoLancamentoSalvo = lancamentos.get(lancamentos.size() - 1);
84
85          assertEquals("Compra no mercado", ultimoLancamentoSalvo.getDescricao());
86          assertEquals(200d, ultimoLancamentoSalvo.getValor());
87      }
```

Fonte: Elaboração do autor (2014).

Nesse caso de teste, o cenário é de basicamente criar um lançamento e chamar o método salvar() da classe LancamentoDAO, fazendo com que esse lançamento fique salvo no banco de dados. Porém, para ter certeza que esse lançamento foi salvo, é necessário chamar o método listarTodos() que irá buscar todos os lançamentos salvos no banco e verificar o último lançamento, como é feito na variável ultimoLancamentoSalvo, sendo necessário validar se esse último lançamento é o mesmo que foi salvo anteriormente, o que foi feito nas linhas 85 e 86, respectivamente.

Outra funcionalidade importante para o sistema é a exclusão de um lançamento, sendo esse validado pelo caso de teste CT023 e a sua implementação, mostrada na figura 30, a seguir:

Figura 30 - Automatização do caso de teste CT023

```
119     @Test
120     public void deveExcluirUmLancamento() {
121         dao.excluir(2L);
122
123         Lancamento lancamento = dao.buscarPeloId(2L);
124         assertNull(lancamento);
125     }
```

Fonte: Elaboração do autor (2014).

Esse caso de teste é bem simples, já que, na linha 121, é invocado o método excluir() passando o ID 2 como parâmetro, que é um dos lançamentos que estavam definidos no *dataset* de lançamentos. Assim, para garantir que esse lançamento não existe mais no

banco, o método buscarPeloId() é chamado com o ID 2 passado por parâmetro e a validação é realizada na linha 124 pelo método assertNull(), que deve garantir que não veio nenhum lançamento nessa consulta.

O restante dos casos de teste elaborados, na seção 4.2.4.2, foram implementados nessa classe de teste, estando disponíveis para consulta nos códigos fonte do projeto de estudo de caso.

4.2.5.3 Casos de teste no nível de sistema

Nesse último nível de automatização dos casos de teste, o objetivo principal é de validar as funcionalidades em seu todo, ou seja, não somente a sua unidade ou integração com um sistema externo, mas sim com o sistema em seu funcionamento pleno, contando agora com as páginas *web*.

Como os testes são do tipo caixa preta, ou seja, não é necessário conhecer a estrutura interna do código, o *framework* Selenium WebDriver foi utilizado para conseguir interagir com os elementos das páginas e efetuar algumas ações, como, por exemplo, preencher os campos das telas de cadastro e edição dos lançamentos, clicar nos botões, verificar os dados nas tabelas de listagem, entre outros.

Para isso, os testes foram executados no navegador Google Chrome, utilizando um *driver* específico para esse navegador, sendo que a estrutura inicial da classe LancamentosTest é mostrada na figura 31, a seguir:

Figura 31 - Estrutura interna da classe LancamentosTest

```
20 public class LancamentosTest {
21     private static WebDriver driver;
22     private static String URL_BASE = "http://localhost:8083/EstudoDeCaso";
23     private LancamentosPage lancamentos;
24     private ContasPage contas;
25
26     @BeforeClass
27     public static void setUpClass() {
28         System.setProperty("webdriver.chrome.driver", "C:\\chromedriver\\chromedriver.exe");
29
30         driver = new ChromeDriver();
31         driver.get(URL_BASE);
32     }
33
34     @Before
35     public void setUp()  throws Exception {
36         lancamentos = new LancamentosPage(driver);
37         contas = new ContasPage(driver);
38
39         lancamentos.visitar();
40     }
41
42     @After
43     public void tearDown() {
44         driver.get(URL_BASE + "/excluirTodosLancamentos");
45     }
46
47     @AfterClass
48     public static void tearDownClass() {
49         driver.quit();
50     }
```

Fonte: Elaboração do autor (2014).

Essa classe está no pacote com.unisul.tcc.testesdesistema e não tem um padrão parecido com as outras classes de teste devido ao fato de que não existe nenhuma classe em produção que a represente, sendo que o teste é relacionado aos lançamentos no sistema *web*.

Assim, como nos testes de integração, essa classe também possui os métodos setUp() e tearDown() anotados com *@Before* e *@After*, com a adição dos métodos setUpClass() e tearDownClass() anotados com *@BeforeClass* e *@AfterClass*. Esses últimos dois métodos têm um comportamento parecido com os métodos setUp() e tearDown(), com a exceção de que eles são somente executados antes e depois da classe de teste ter sido executada como um todo, servindo, nesses casos, para abrir o navegador uma vez e fechar o mesmo quando todos os casos de teste forem executados nessa classe.

A diferença é que ao invés de criar uma conexão com o banco e depois fechá-la, a interface WebDriver recebe um *driver* para o navegador Google Chrome, conforme é mostrado na linha 30. Para que esse navegador seja utilizado, uma configuração precisou ser feita na linha 28.

Após isso, o método get() recebe uma URL (*Uniform Resource Locator*) definida na linha 31 pela constante URL_BASE, que representa a URL do ambiente de testes que são

executados todos os casos de teste dessa classe. Em seguida, esse *driver* é passado por parâmetro para os construtores das classes LancamentosPage e ContasPage no método setUp(), com a variável lançamentos chamando o método visitar() na linha 39, que redireciona para a página com a listagem de lançamentos.

Essas duas classes foram criadas a partir de um padrão de projeto chamado de *Page Objects*[7], tendo como principal objetivo de representar cada página do sistema como uma classe, isolando em um único ponto a lógica de como se encontram os elementos na página e sua manipulação, facilitando a escrita dos testes e também a sua manutenção, pois, se um elemento for modificado, apenas um lugar precisa ser alterado, dando maior estabilidade de mudanças para todos os casos de teste que utilizam essas páginas.

Um exemplo dessas classes que utilizam o padrão de *Page Objects* pode ser encontrado na classe CadastroLancamentoPage, como mostra a figura 32, a seguir:

Figura 32 - Métodos da classe CadastroLancamentoPage

```
29      public void comValor(String valor) {
30          WebElement campoValor = driver.findElement(By.id("campoValor"));
31          campoValor.sendKeys(valor);
32      }
33
34      public void comObservacao(String observacao) {
35          WebElement campoObservacao = driver.findElement(By.id("campoObservacao"));
36          campoObservacao.sendKeys(observacao);
37      }
38
39      public void selecionarConta(String nomeConta) {
40          Select comboConta = new Select(driver.findElement(By.id("comboConta")));
41          comboConta.selectByVisibleText(nomeConta);
42      }
43
44      public void selecionarTipoLancamento(String tipoLancamento) {
45          Select comboTipoLancamento = new Select(driver.findElement(By.id("comboTipoLancamento")));
46          comboTipoLancamento.selectByVisibleText(tipoLancamento);
47      }
48
49      public void clicarEmSalvar() {
50          WebElement botaoSalvar = driver.findElement(By.id("botaoSalvar"));
51          botaoSalvar.click();
52
53          new WebDriverWait(driver, 10).until(
54                  ExpectedConditions.alertIsPresent());
55
56          alerta = driver.switchTo().alert();
57          alerta.accept();
58      }
59
60      public String getMensagemAlerta() {
61          return alerta.getText();
62      }
63  }
```

Fonte: Elaboração do autor (2014).

[7] *Page Objects*: https://code.google.com/p/selenium/wiki/PageObjects

Nessa classe, alguns métodos foram criados para facilitar o preenchimento dos campos e, consequentemente, realizar o cadastro, como, por exemplo, o método comValor() que recebe por parâmetro uma *String* que representa o valor de um lançamento e o método getMensagemAlerta(), que retorna uma *String* com a mensagem do alerta na página, podendo ser uma mensagem que o lançamento foi cadastrado com sucesso ou se aconteceu algum erro nesse cadastro.

No primeiro método, o Selenium representa cada elemento na página como um objeto da classe WebElement, sendo que na linha 30 a variável campoValor recebe o elemento encontrado pelo ID campoValor, que está no HTML (*HyperText Markup Language*) da página de cadastros de lançamentos. Após isso, esse campo é preenchido com o valor da variável, valor utilizando o método sendKeys(), assim como os outros campos também são preenchidos dessa forma nos métodos comDescricao(), comDataDeLancamento() e comObservacao(). Na linha 40, ao invés de ser utilizado um WebElement, o Selenium possui a classe Select para manipular um componente do tipo *combo box*, que, nesse caso, seria o *combo box* para selecionar uma conta para essa lançamento, selecionando a opção pelo método selectByVisibleText(), recebendo a variável nomeConta como parâmetro. A mesma situação acontece no método selecionarTipoLancamento() nas linhas 45 e 46. Por fim, na linha 50, a variável botaoSalvar é criado com o elemento encontrado com o ID de botaoCadastrar, sendo que na linha 51 é invocado o método click() que indica ao Selenium efetuar um evento de clique no botão e fazer o cadastro. Na linha 53, a classe WebDriverWait é utilizada para aguardar por até 10 segundos um alerta do Javascript aparecer na tela e logo, em seguida, as linhas 56 e 57 acessam esse alerta e invocam o método accept() para clicar no botão “OK” e fechar esse componente.

Já, o método getMensagemAlerta() apenas retorna o texto do alerta chamando o método getText(), que, posteriormente, será chamado nos casos de teste para verificar as mensagens disparadas pelo sistema.

Com toda essa estrutura inicial definida, um dos casos de teste que foram automatizados é o caso de teste CT028, que consiste no cadastro de um lançamento do tipo depósito, como é mostrado na figura 33, a seguir:

Figura 33 - Automatização do caso de teste CT028

```
52    @Test
53    public void deveCadastrarUmDeposito() {
54        String descricao = "Salário de teste";
55        String dataLancamento = "13/12/2013";
56        String valor = "5.000,00";
57        String observacao = "Salário para testes";
58        String tipoLancamento = "Depósito";
59        String nomeConta = "Conta corrente PF";
60
61        CadastroLancamentoPage paginaCadastro = lancamentos.cadastrar();
62        paginaCadastro.comDescricao(descricao);
63        paginaCadastro.comDataDeLancamento(dataLancamento);
64        paginaCadastro.comValor(valor);
65        paginaCadastro.comObservacao(observacao);
66        paginaCadastro.selecionarTipoLancamento(tipoLancamento);
67        paginaCadastro.selecionarConta(nomeConta);
68        paginaCadastro.clicarEmSalvar();
69
70        assertEquals("Lançamento salvo com sucesso!", paginaCadastro.getMensagemAlerta());
71        assertTrue(lancamentos.existeNaListagem(descricao, dataLancamento, valor, observacao, nomeConta));
72
73        contas.visitar();
74        assertEquals("5.000,00", contas.getValor());
75    }
```

Fonte: Elaboração do autor (2014).

Esse caso de teste não é muito diferente dos outros apresentados anteriormente, porém, ao invés de atribuir variáveis para as classes do projeto, são criadas as variáveis que são os dados que devem ser preenchidos/selecionados no formulário da tela de cadastro, sendo que, na linhas 62 até 67, são chamados os métodos por preencher/selecionar os campos na tela de cadastro e, na linha 68, é chamado o método clicarEmSalvar(). O cenário desse caso de teste se trata de um lançamento do tipo depósito com o valor de R$ 5.000,00 para a conta com o nome de Conta Corrente PF, sendo que essa conta está pré-cadastrada no banco de dados do ambiente de testes.

Após isso, nas linhas 70 e 71, os métodos assertEquals() e assertTrue() são chamados, com o primeiro método, verificando se a mensagem do alerta está igual a “Lançamento cadastrado com sucesso!” e o segundo verifica se realmente os dados cadastrados estão aparecendo na listagem, que nesse caso é considerada a última linha da tabela de lançamentos. Ao final desse caso de teste, outra validação precisa ser feita, que seria a de acessar a listagem de contas e validar com o método assertEquals() o valor da coluna com o saldo atual da conta, que deve estar em R$ 5.000,00, sendo isso realizado nas linhas 73 e 74, respectivamente.

Outro caso de teste que valida se o sistema não ficará em um estado inconsistente, é o caso de teste CT033 que realiza um cadastro com o valor do lançamento igual a zero, como mostra a figura 34, a seguir:

Figura 34 - Automatização do caso de teste CT033

```
149     @Test
150     public void naoDeveCadastrarUmLancamentoComValorZero() {
151         String valor = "0,00";
152
153         CadastroLancamentoPage paginaCadastro = lancamentos.cadastrar();
154         paginaCadastro.comValor(valor);
155         paginaCadastro.clicarEmSalvar();
156
157         assertTrue(paginaCadastro.getMensagemAlerta().contains("O campo valor não pode ser zero!"));
158     }
```

Fonte: Elaboração do autor (2014).

Nesse caso de teste, também é feito um cadastro, porém passando o valor como R$ 0,00, sendo que, na linha 157, o método assertTrue() verifica se a mensagem do alerta contem a mensagem “O campo valor não pode ser zero!”, evitando, assim, que o lançamento seja cadastrado.

Os outros casos de testes automatizados estão disponíveis para consulta nos códigos fonte do projeto.

4.2.6 Execução e avaliação dos resultados

Devido ao fato de que cada classe de teste deve ser executada de forma separada, foram criadas quatro suítes de teste para agrupar as classes de teste por nível, sendo que essas suítes de teste se encontram no pacote com.unisul.tcc.suitesdeteste.

O principal motivo de se utilizar uma suíte de testes é que um projeto pode ter várias classes de teste para serem executadas e há necessidade de, muitas vezes, estabelecer uma ordem de execução, como, por exemplo, os testes de unidade devem ser executados antes dos testes de integração. Assim, foram criadas as classes TestesDeUnidade, TestesDeIntegracao, TestesDeSistema e AllTests, sendo que essa última tem como objetivo executar todas as outras suítes de teste em uma ordem pré determinada. A figura 35, a seguir, mostra essa sequência definida pela classe AllTests:

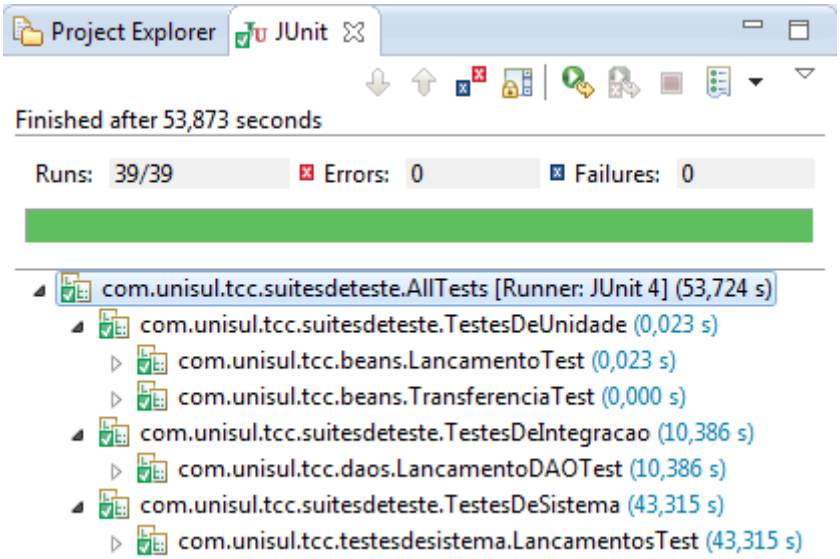
Figura 35 - Definição da ordem de execução dos testes na classe AllTests

```
7  @RunWith(Suite.class)
8  @SuiteClasses({TestesDeUnidade.class, TestesDeIntegracao.class, TestesDeSistema.class })
9  public class AllTests {
10
11 }
```

Fonte: Elaboração do autor (2014).

Assim, para executar todos os testes desse estudo de caso, é necessário somente utilizar essa classe, e o resultado dessa execução é mostrado na figura 36, a seguir:

Figura 36 - Execução de todos os casos de teste do estudo de caso



Fonte: Elaboração do autor (2014).

Levando em consideração a quantidade de casos de teste executados, o tempo total de execução foi baixo, de tal forma que executar toda a suíte de testes levou um pouco menos de 1 minuto. Além disso, a diferença de tempo entre os casos de teste é justificável, devido ao fato de que os testes de unidade têm pouca ou nenhuma dependência externa, assim como os testes no nível de integração executaram muito mais rápido do que os testes de sistema, já que a sua dependência é o banco de dados, ao invés de toda a infra-estrutura de utilizar um navegador *web*, efetuar requisições e receber respostas do servidor e também do acesso ao banco de dados que é feito pelo mesmo.

Por isso, o tempo de *feedback* é mais rápido para os testes de unidade e integração, sendo que os testes de sistema são executados por último, geralmente, pelos testadores ou automatizando essa execução por um sistema de integração contínua, como, por exemplo, o Jenkins.

Apesar do baixo tempo de execução dos testes, isso não pode ser comparado ao seu tempo de desenvolvimento dos casos de teste. No total do projeto, quatorze horas de desenvolvimento foram aplicadas nessa atividade, sendo quatro horas para os testes no nível de unidade, três horas para os testes no nível de integração e mais seis horas nos testes no nível de sistema. Porém, mesmo com esse tempo gasto, não haverá problemas com a repetição dos mesmos testes quantas vezes forem necessárias. Se todo o processo de teste fosse realizado de forma manual, certamente poderia ultrapassar esse tempo de desenvolvimento.

Como os níveis de unidade e integração são testes de caixa branca e conforme foi explicado na seção 4.1.4, a ferramenta JaCoCo foi utilizada para medir a cobertura dos casos de teste, conforme mostra a figura 37, a seguir:

Figura 37 - Cobertura de código no projeto com JaCoCo

Markers | Servers | Properties | Data Source Explorer | Snippets | Console | Progress | Search | Coverage

Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
EstudoDeCaso	37,4 %	635	1.065	1.700
src/main/java	37,4 %	635	1.065	1.700
com.unisul.tcc.daos	25,9 %	188	539	727
ContaDAO.java	0,0 %	0	208	208
TransferenciaDAO.java	0,0 %	0	202	202
LancamentoDAO.java	79,7 %	188	48	236
PersistenceManager.java	0,0 %	0	40	40
BancoDAO.java	0,0 %	0	31	31
UsuarioDAO.java	0,0 %	0	10	10
com.unisul.tcc.beans	69,1 %	427	191	618
Conta.java	57,1 %	84	63	147
Usuario.java	5,4 %	3	53	56
Transferencia.java	79,9 %	139	35	174
Banco.java	26,9 %	7	19	26
Lancamento.java	90,3 %	149	16	165
TipoLancamento.java	90,0 %	45	5	50
com.unisul.tcc.controllers	0,0 %	0	182	182
com.unisul.tcc.services	0,0 %	0	114	114
com.unisul.tcc.converters	0,0 %	0	24	24
com.unisul.tcc.exceptions	57,1 %	20	15	35

Fonte: Elaboração do autor (2014).

Se for levar em consideração somente o percentual do projeto, a cobertura de código está muito baixa, ficando com 37,4%. Porém, se forem consideradas apenas as classes que possuem testes, o pacote com.unisul.tcc.daos que contem 6 classes, somente uma delas tem uma classe de teste, que seria a classe LancamentoDAO, ficando, assim, com a cobertura de 79,7%, o que é um percentual razoável.

As classes do pacote com.unisul.tcc.beans tiveram apenas duas classes sob teste, que foram as classes Lancamento e Transferencia, com a primeira classe contendo 90,3% de

cobertura e a segunda, com 79,9% de cobertura. As outras classes desse pacote tiveram algum percentual de cobertura por terem sido utilizadas indiretamente por essas duas classes. O mesmo aconteceu com as classes do pacote com.unisul.tcc.exceptions, que foram utilizadas para criar os casos de teste que precisavam lançar alguma exceção em um estado inválido, como, por exemplo, que o valor de um lançamento não pode ser zero.

Conforme foi explicado na seção 2.8, a cobertura de código é utilizada para indicar o grau de risco da não cobertura de testes, e, caso seja baixo, deve-se direcionar esforços para melhorar essa cobertura. No entanto, muitas das classes do projeto não fazem sentido de serem testadas em trechos específicos ou no seu todo, como, por exemplo, os métodos get() e set() das classes, já que elas apenas colocam e retornam um valor de uma variável, o que inclusive não é recomendado como item de teste devido ao fato de que não agregam valor nos testes. Outros exemplos seriam os das classes de exceções, uma vez que elas não possuem uma validação em específico e sua maior cobertura irá influenciar pouco ou nada em todo do sistema, já que não se pode considerar como uma funcionalidade do mesmo.

Por último, a cobertura dos casos de teste de sistema, que são do tipo caixa preta, podem ser medidos pelo número de casos de teste executados, pelos requisitos e pelos casos de uso.

Nesse sentido, pelo número de requisitos, os testes cobrem os dois primeiros citados na seção 4.2.1, sendo o primeiro coberto parcialmente, pois envolve as contas bancárias do usuário e o segundo, completamente, já que é referente aos lançamentos. Como foram elaborados 14 casos de teste, e desses 12 foram automatizados e executados, o percentual de execução, nesse caso, foi de aproximadamente 95,12% dos mesmos, o que é um bom percentual, mais ainda pelo fato de que os testes de regressão podem ser executados em poucos minutos, diminuindo os testes manuais como um todo. Já pelos casos de uso, nesse nível, foi executado somente um dos dois casos de uso utilizados nesse estudo de caso, sendo esse o caso de uso UC004, o que representa 50% do total.

A seguir, são apresentadas algumas recomendações e considerações finais sobre o desenvolvimento deste estudo de caso.

4.2.7 Recomendações sobre automação de testes

Terminada a elaboração, automatização e análise dos casos de teste, algumas recomendações são interessantes de se pontuar, não só incluindo esse estudo de caso, mas qualquer outro projeto com testes automatizados. As recomendações são as seguintes:

- automatizar os testes somente em classes que façam sentido para o projeto, como as funcionalidades principais, foco do negócio. Como foi apresentado, anteriormente, criar testes para aumentar a cobertura sem um ganho real é desnecessário e não irá justificar o custo da automação. O mesmo caso vale para os testes de sistema, pelo fato de serem mais caros de serem criados/executados em termos de tempo, motivo pelo que se deve ter um maior cuidado com a sua automação;
- utilizar padrões de projeto para os testes, como *Test Data Builders* e *Page Objects*, pois assim como os padrões de projeto são interessantes para resolver problemas em classes de produção, o mesmo caso vale para as classes de teste, melhorando a sua manutenção e legibilidade na criação/validação dos cenários de testes;
- independente do nível de teste, cada caso de teste precisa ser o mais autônomo possível, pois um caso de teste pode ser executado independente da ordem que foi chamado em uma classe. Para isso, recursos do JUnit como *@BeforeClass*, *@Before*, *@After* e *@AfterClass* são muito úteis para inicializar/finalizar algumas pré-condições e pós-condições que os casos de teste possam ter em comum;
- utilizar classes/métodos auxiliares para facilitar e manter a integridade dos testes, como aconteceu com os testes para lançamentos que tinham uma URL disponível para excluir todos os lançamentos. Isso facilita que os próximos casos de testes não tenham problemas com esses dados, já que o foco é validar comportamentos e não dados anteriores de outros casos de teste;
- subdividir as classes de teste em suítes de teste seja elas por nível, módulo do sistema ou subprojeto. Isso facilita toda a execução dos testes de forma bem definida, além da possibilidade de priorizar quais classes de testes precisam ser executadas antes, melhorando o *feedback* dos testes que validam as principais funcionalidades do sistema.

4.3 CONSIDERAÇÕES FINAIS

Este capítulo teve como objetivo apresentar o estudo de caso proposto para este trabalho, descrevendo as ferramentas utilizadas, bem como o projeto em si, desde os seus casos de uso, regras de negócio, elaboração dos casos de teste e a sua automatização. Ao final, também foram apresentados os resultados de execução e análise dos testes nos níveis de unidade, integração e sistema, assim como algumas recomendações sobre a automação de testes que podem ser utilizadas em qualquer outro projeto de *software*.

A seguir, são apresentadas as conclusões sobre este estudo e propostas de trabalhos futuros que poderiam dar continuidade a esta proposta.

5 CONCLUSÕES E TRABALHOS FUTUROS

A qualidade de *software* tem cada vez mais importância para as empresas, uma vez que a sua falta pode acarretar em prejuízos financeiros. Se o usuário utiliza um sistema em que as funcionalidades não respondem como esperado, ou que possui erros, então, a imagem da empresa desenvolvedora pode ficar seriamente comprometida. Com isso, os testes de *software* são utilizados para aferir a qualidade desses sistemas, sejam eles garantindo que estão funcionando como esperado ou com a descoberta de erros, para poder solucioná-los.

Neste trabalho, a revisão da literatura foi de suma importância para responder aos objetivos específicos, como, por exemplo, mostrar as etapas de um processo de testes de *software*, mostrar quais tipos de testes existem e em quais casos eles são utilizados, apresentar vantagens e desvantagens de aplicar testes automatizados, além de outros tópicos que abrangem a qualidade de *software* como um todo.

Baseado nisso, um estudo de caso foi realizado para demonstrar como um sistema pode ter testes automatizados, sendo que os testes foram aplicados em um sistema *web*. Neste projeto, que consiste em um sistema não comercial, foi demonstrado como os testes nos níveis de unidade, integração e de sistema podem ser utilizados, com as suas diferenças e semelhanças entre cada nível de teste. As ferramentas JUnit, DBUnit, Selenium WebDriver e JaCoCo foram escolhidas pelo fato de serem ferramentas conhecidas pela comunidade, além de serem *open source*. Cada uma delas foi utilizada em um determinado nível de teste ou em todos eles, de acordo com o escopo para o qual foram projetadas e, também, pela necessidade do estudo de caso em si.

Com baixo tempo de execução, os testes automatizados podem ser rodados frequentemente, o que garante testes de regressão que ajudam a validar as funcionalidades que já estavam funcionando anteriormente, trazendo mais segurança para a evolução/correção do *software*. Isso também melhora o próprio processo de testes, já que o tempo que era gasto em testes manuais pode ser revertido em um melhor planejamento/revisão dos casos de teste, com o intuito de encontrar mais falhas e aumentar a qualidade do sistema como um todo.

Outra preocupação que ocorre, quando se pensa em automação dos testes, é em relação a sua manutenção, pois, se a construção dos casos de teste for mal implementada, poderá trazer custos mais altos que o custo da própria automação em si. Por isso, foram utilizados dois padrões de projetos de testes para amenizar e facilitar essa manutenção,

evitando um maior gasto de tempo, levando em consideração que um sistema não é estático, pois sempre está em evolução com novas funcionalidades, correções e melhorias.

No entanto, automatizar por si só não é garantia de que os problemas com testes de *software* são resolvidos e muito menos são a solução para todos os casos, já que os testes manuais ainda são muito importantes e ambos se complementam, devendo avaliar em quais casos são interessantes de aplicar um ou outro.

Como sugestões para trabalhos futuros, seria interessante implementar as outras funcionalidades e por consequência criar e/ou melhorar os casos de teste, assim como utilizar ferramentas para gerenciar todos esses artefatos de testes, garantindo uma maior rastreabilidade com os requisitos/casos de uso. Também seria interessante abordar outros tipos de testes, como, por exemplo, testes de performance e testes de carga.

Além disso, também se pode pensar na alternativa de executar todos os testes, pelo menos no nível de sistema, em um ambiente em nuvem, tirando toda a dependência de uma infraestrutura local de testes, diminuindo principalmente os custos com configurações e manutenções. Por fim, o tempo de execução pode ser reduzido com uma estratégia de rodar os testes em paralelo, principalmente em casos em que os mesmos precisam garantir o funcionamento em diferentes *browsers* ou em um mesmo *browser* com versões diferentes, aumentando mais ainda a abrangência desses testes, algo que seria muito custoso caso fossem executados de forma manual. Também poderia ser realizado um comparativo de tempo entre os testes manuais e automatizados.

Também seria possível estudar a automação de testes acontecendo em um projeto de desenvolvimento de *software* de uma organização específica, ou o uso da automação de testes com um particular modelo de desenvolvimento, como, por exemplo, na metodologia Scrum, ou XP.

REFERÊNCIAS

ANICHE, Mauricio. **Test-Driven Development:** Teste e *Design* no Mundo Real. São Paulo: Casa do Código, 2012.

BARTIÉ, Alexandre. **Garantia da qualidade de *software:*** adquirindo maturidade organizacional. Rio de Janeiro: Campus, 2002.

BASTOS, Aderson; RIOS, Emerson; CRISTALLI, Ricardo; MOREIRA, Trayahú. **Base de conhecimento de teste de *software*.** 2ª edição. São Paulo: Martins, 2007.

CAETANO, Cristiano. **Aumentando a Produtividade com as Principais Soluções *Open Source* e Gratuitas.** 1ª Edição. 2007.

CAETANO, Cristiano. Testes ágeis. **Engenharia de *Software Magazine***, n. 48, p. 10-15, 2012.

CERVO, Luiz Amado; BERVIAN, Pedro Alcino. **Metodologia científica.** 5ª edição. São Paulo: Prentice Hall, 2002.

DELAMARO, Márcio Eduardo; MALDONADO, José Carlos; JINO, Mario. **Introdução ao teste de *software*.** Rio de Janeiro: Elsevier, 2007.

GUEDES, Guileanes Thorwald Araujo. UML 2 : uma abordagem prática. São Paulo: Novatec Editora, 2011.

INTHURN, Cândida. **Qualidade e teste de *software*.** Florianópolis: Visual Books, 2001.

KOSCIANSKI, André; SOARES, Michel dos Santos. **Qualidade de *software*- Aprenda as metodologias e técnicas mais modernas para o desenvolvimento de *software*.** 2ª edição. São Paulo: Novatec Editora, 2007.

LAGES, Scaldaferri Daniel. Automação dos Testes: um lobo na pele de cordeiro?. **Engenharia de *Software Magazine***, n. 29, p. 20-25, 2010.

MACEDO, Paulo Cesar; MORAES, Marcos Roberto de; CATINI, Rita de Cássia. **Uso de testes automatizados em projetos de *software*: Estudo da aplicação em *software* comercial. Universitas**, Ano 5, n. 9, p. 107-130, 2012. Disponível em <http://revistauniversitas.inf.br/index.php/UNIVERSITAS/article/view/33>. Acesso em: 01 mai. 2014.

MARCONI, Marina de Andrade; LAKATOS, Eva Maria. **Fundamentos da metodologia científica.** 5ª edição. São Paulo: Atlas, 2003.

MASUDA, Cristiana Yukie. **Processo de automação de testes de *software* com ferramentas *open source:*** Um estudo de caso com integração contínua, do curso de Sistemas de Informação da UNISUL – Campus de Palhoça. 2009. 152 f. Monografia (Graduação de Sistemas de Informação)-Universidade do Sul de Santa Catarina, Palhoça, 2009. Disponível

em: <http://aplicacoes.unisul.br/pergamum/pdf/99959_Cristiana.pdf>. Acesso em: 17 ago. 2013.

MIGUEL, Paulo Augusto Cauchick. Estudo de Caso na engenharia de produção: estruturação e recomendações para sua condução, em: **Produção**, v.17, n. 1, p. 216-229, Jan/Abr. 2007.

MOLINARI, Leonardo. **Testes de *Software*:** Produzindo Sistemas Melhores e Mais Confiáveis. São Paulo: Érica, 2003.

NOGUEIRA, Elias. **Testes de caixa cinza (Gray Box).** Disponível em: <http://www.testexpert.com.br/?q=node/838>. Aceso em: 31 ago. 2013.

OLIVEIRA, Silvio Luiz de. **Tratado de Metodologia Científica:** projetos de pesquisa, TGI, TCC, monografias, dissertações e teses. 2ª edição. São Paulo: Pioneira Thomson Learning, 1999.

PEZZÈ, Mauro; YOUNG, Michael. **Teste e análise de *software:*** processo, princípios e técnicas. Porto Alegre: Bookman, 2008.

PRESSMAN, Roger S.. **Engenharia de *software*:** Uma abordagem profissional. 7ª edição. Porto Alegre: AMGH, 2011.

PRIOLO, Sebastian. **Métodos ágiles:** Uma alternativa real y competitiva a los procesos tradiocionales de desarrolo. Banfield – Lomas de Zamora: Gradi, 2009.

RIOS, Emerson; MOREIRA, Trayahú. **Testes de *Software*.** 2ª edição. Rio de Janeiro: Alta Books, 2006.

RUNESON, Per; HÖST, Martin. *Guidelines for conducting and reporting case study research in software engineering*, em: Empir *Software* Eng, n. 14, p. 131-164, Dez. 2009.

SCHUHMACHER, Vera Rejane Niedersberg. **Qualidade de *software*.** 2ª edição. Palhoça: UnisulVirtual, 2006.

SILVA, Edna Lúcia da; MENEZES, Estera Muszkat. **Metodologia da pesquisa e elaboração de dissertação.** 4ª Edição. Florianópolis: UFSC, 2005.

SOFTEX, **Guia Geral MPS de *Software***, publicado em 2012. Disponível em: <www.softex.br/wpcontent/uploads/2013/07/MPS.BR_Guia_Geral_Software_2012.pdf> Acesso em: 24 ago. 2013.

SOMMERVILLE, Ian. **Engenharia de *software*.** 9ª edição. São Paulo: Pearson Prentice Hall, 2011.

APÊNDICES

APÊNDICE A – CASOS DE TESTE

CT001 – Saque com valor válido	
Resumo	Deve ser realizado um lançamento do tipo saque com valor válido (Dentro dos limites de valor do saldo atual da conta).
Pré-condições	Ter uma conta associada e saldo maior que zero.
Passos	1. Criar um objeto de lançamento com um valor dentro do saldo atual e do tipo saque. 2. Invocar o método que realiza os lançamentos.
Resultados esperados	Lançamento criado, sendo que o saldo atual da conta deve estar igual ao valor anterior menos o valor do lançamento.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT002 – Saque com valor negativo	
Resumo	Não deve ser permitido um saque com valor negativo.
Pré-condições	Ter uma conta associada.
Passos	1. Criar um objeto de lançamento com valor negativo e do tipo saque. 2. Invocar o método que realiza os lançamentos.
Resultados esperados	Lançar uma exceção, impedindo que a conta tenha esse lançamento com valor negativo e seu saldo atual seja alterado.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT003 – Saque com valor zero	
Resumo	Não deve ser permitido um saque com valor igual à zero.
Pré-condições	Ter uma conta associada.
Passos	1. Criar um objeto de lançamento com valor zero e do tipo saque. 2. Invocar o método que realiza os lançamentos.
Resultados esperados	Lançar uma exceção, impedindo que a conta tenha esse lançamento com valor zero e seu saldo atual seja alterado.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT004 – Saque com valor acima do saldo atual da conta	
Resumo	Não deve ser permitido um saque com o valor acima do saldo atual da conta.
Pré-condições	Ter uma conta associada com o saldo maior que zero.
Passos	1. Criar um objeto de lançamento com o valor acima do saldo atual da conta e do tipo saque. 2. Invocar o método que realiza os lançamentos.
Resultados esperados	Lançar uma exceção, impedindo que a conta tenha esse lançamento com valor acima do saldo atual, ficando com o seu saldo atual com valor negativo.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT005 – Saque com valor total da conta	
Resumo	Deve ser permitido um saque que tenha o valor igual ao saldo atual da conta.
Pré-condições	Ter uma conta associada com o saldo maior que zero.
Passos	1. Criar um objeto de lançamento com o valor igual ao saldo atual da conta e do tipo saque. 2. Invocar o método que realiza os lançamentos.
Resultados esperados	Lançamento criado, sendo que o saldo atual da conta deve estar igual à zero.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT006 – Depósito com valor válido	
Resumo	Deve ser realizado um lançamento do tipo depósito com valor válido (maior que zero).
Pré-condições	Ter uma conta associada.
Passos	1. Criar um objeto de lançamento com valor maior que zero e do tipo depósito. 2. Invocar o método que realiza os lançamentos.
Resultados esperados	Lançamento criado, sendo que o saldo atual da conta deve estar igual ao saldo anterior mais o valor do depósito.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT007 – Depósito com valor negativo	
Resumo	Não deve ser permitido um depósito com valor negativo.
Pré-condições	Ter uma conta associada.
Passos	1. Criar um objeto de lançamento com valor negativo e do tipo depósito. 2. Invocar o método que realiza os lançamentos.
Resultados esperados	Lançar uma exceção, impedindo que a conta tenha esse lançamento com valor negativo.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT008 – Depósito com valor zero	
Resumo	Não deve ser permitido um depósito com valor igual a zero.
Pré-condições	Ter uma conta associada.
Passos	1. Criar um objeto de lançamento com valor zero e do tipo depósito. 2. Invocar o método que realiza os lançamentos.
Resultados esperados	Lançar uma exceção, impedindo que a conta tenha esse lançamento com valor zero.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT009 – Lançamento com data no futuro	
Resumo	Não deve ser permitido um lançamento (qualquer tipo) com data no futuro.
Pré-condições	Ter uma conta associada.
Passos	1. Criar um objeto de lançamento com data no futuro. Ex.: com ano 3000. 2. Invocar o método que realiza os lançamentos.
Resultados esperados	Lançar uma exceção, impedindo que a conta tenha esse lançamento com data no futuro.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT010 – Transferência com valor válido	
Resumo	Deve ser realizada uma transferência com valor válido, levando em consideração o valor limite do saldo atual da conta de origem.
Pré-condições	Ter duas contas, sendo uma para origem e outra para a conta de destino.
Passos	1. Criar um objeto de transferência com valor válido. 2. Invocar o método que realiza as transferências.
Resultados esperados	Transferência criada, sendo que o saldo atual da conta origem deve ter o valor da transferência subtraído por esse valor e o saldo atual da conta destino com esse valor somado.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT011 – Transferência com valor acima do saldo	
Resumo	Não deve ser permitido que uma transferência tenha valor acima do saldo da conta origem, já que esse valor é um saque para essa conta.
Pré-condições	Ter duas contas, sendo uma para origem e outra para a conta de destino.
Passos	1. Criar um objeto de transferência com valor acima do saldo atual da conta origem. 2. Invocar o método que realiza as transferências.
Resultados esperados	Lançar exceção, impedindo que as contas origem e destino tenham os seus saldos atuais alterados.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT012 – Transferência com valor negativo	
Resumo	Não deve ser permitido que uma transferência tenha valor negativo.
Pré-condições	Ter duas contas, sendo uma para origem e outra para a conta de destino.
Passos	1. Criar um objeto de transferência com valor negativo. 2. Invocar o método que realiza as transferências.
Resultados esperados	Lançar exceção, impedindo que as contas origem e destino tenham os seus saldos atuais alterados.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT013 – Transferência com valor total do saldo	
Resumo	Deve ser realizada uma transferência com valor igual ao saldo atual da conta origem.
Pré-condições	Ter duas contas, sendo uma para origem e outra para a conta de destino.
Passos	1. Criar um objeto de transferência com valor igual ao saldo atual da conta origem. 2. Invocar o método que realiza as transferências.
Resultados esperados	Transferência criada, sendo que a conta origem deve estar com o saldo atual igual a zero e o saldo atual da conta destino com esse valor somado.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT014 – Transferência com valor zero	
Resumo	Não deve ser permitido ter uma transferência com valor zero.
Pré-condições	Ter duas contas, sendo uma para origem e outra para a conta de destino.
Passos	1. Criar um objeto de transferência com valor zero. 2. Invocar o método que realiza as transferências.
Resultados esperados	Lançar exceção, impedindo que as contas origem e destino tenham os seus saldos atuais alterados.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT015 – Transferência sem conta origem	
Resumo	Não deve ser permitido ter uma transferência sem conta origem associada à mesma.
Pré-condições	Ter duas contas, sendo uma para origem e outra para a conta de destino.
Passos	1. Criar um objeto de transferência sem conta origem associada. 2. Invocar o método que realiza as transferências.
Resultados esperados	Lançar exceção, impedindo que as contas origem e destino tenham os seus saldos atuais alterados.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT016 – Transferência sem conta destino	
Resumo	Não deve ser permitido ter uma transferência sem conta destino associada à mesma.
Pré-condições	Ter duas contas, sendo uma para origem e outra para a conta de destino.
Passos	1. Criar um objeto de transferência sem conta destino associada. 2. Invocar o método que realiza as transferências.
Resultados esperados	Lançar exceção, impedindo que as contas origem e destino tenham os seus saldos atuais alterados.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT017 – Transferência com mesma conta origem e destino	
Resumo	Não deve ser permitido ter uma transferência com duas contas tanto para origem quanto para destino iguais.
Pré-condições	Ter duas contas, sendo uma para origem e outra para a conta de destino.
Passos	1. Criar um objeto de transferência no qual as contas origem e destino sejam as mesmas. 2. Invocar o método que realiza as transferências.
Resultados esperados	Lançar exceção, impedindo que a conta origem tenha o seu saldo atual alterado.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT018 – Transferência com data no futuro	
Resumo	Não deve ser permitido ter uma transferência com data no futuro.
Pré-condições	Ter duas contas, sendo uma para origem e outra para a conta de destino.
Passos	1. Criar um objeto de transferência com data no futuro. Ex.: ano 3000. 2. Invocar o método que realiza as transferências.
Resultados esperados	Lançar exceção, impedindo que as contas origem e destino tenham os seus saldos atuais alterados.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT019 – Salvar um lançamento	
Resumo	Deve salvar um lançamento no banco de dados.
Pré-condições	Estar conectado na base de testes com as tabelas criadas, assim como os dados necessários.
Passos	1. Criar um objeto de lançamento com dados obrigatórios. 2. Invocar o método que salva esse lançamento no banco. 3. Invocar o método que lista todos os lançamentos salvos. 4. Verificar o último lançamento salvo.
Resultados esperados	Lançamento deve estar salvo no banco de dados, comparando se alguns dados estão iguais em comparação a esse último lançamento.
Pós-condições	Excluir todos os dados e tabelas utilizados no caso de teste.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT020 – Salvar um lançamento inválido	
Resumo	Não deve ser permitido salvar um lançamento inválido no banco de dados.
Pré-condições	Estar conectado na base de testes com as tabelas criadas, assim como os dados necessários.
Passos	1. Criar um objeto de lançamento sem algum dado obrigatório, como, por exemplo, a descrição. 2. Invocar o método que salva esse lançamento no banco.
Resultados esperados	Lançar exceção, evitando que um lançamento em estado inconsistente seja salvo no banco de dados.
Pós-condições	Excluir todos os dados e tabelas utilizados no caso de teste.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT021 – Atualizar um lançamento	
Resumo	Deve atualizar um lançamento no banco de dados.
Pré-condições	Estar conectado na base de testes com as tabelas criadas, assim como os dados necessários.
Passos	1. Invocar o método que busca um lançamento pelo seu ID, como, por exemplo, pelo ID igual a 1. 2. Alterar algum valor desse lançamento. 3. Invocar o método que atualiza esse lançamento. 4. Buscar novamente esse lançamento pelo mesmo ID.
Resultados esperados	Lançamento deve estar atualizado, comparando se o valor alterado é igual ao que está no banco de dados.
Pós-condições	Excluir todos os dados e tabelas utilizados no caso de teste.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT022 – Atualizar um lançamento inválido	
Resumo	Não deve ser permitido atualizar um lançamento inválido no banco de dados.
Pré-condições	Estar conectado na base de testes com as tabelas criadas, assim como os dados necessários.
Passos	1. Invocar o método que busca um lançamento pelo seu ID, como, por exemplo, pelo ID igual a 1. 2. Alterar algum valor desse lançamento para um estado inválido, como, por exemplo, deixar a descrição vazia. 3. Invocar o método que atualiza esse lançamento.
Resultados esperados	Lançar exceção, evitando que um lançamento em estado inconsistente seja atualizado no banco de dados.
Pós-condições	Excluir todos os dados e tabelas utilizados no caso de teste.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT023 – Excluir um lançamento	
Resumo	Deve excluir um lançamento no banco de dados.
Pré-condições	Estar conectado na base de testes com as tabelas criadas, assim como os dados necessários.
Passos	1. Invocar o método que busca um lançamento pelo seu ID, como, por exemplo, pelo ID igual a 1. 2. Invocar o método que excluí esse lançamento. 3. Buscar esse mesmo lançamento invocando o método do passo 1.
Resultados esperados	Lançamento excluído, não sendo encontrado esse lançamento na busca feita posteriormente pelo mesmo ID.
Pós-condições	Excluir todos os dados e tabelas utilizados no caso de teste.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT024 – Listar todos os lançamentos	
Resumo	Deve listar todos os lançamentos salvos no banco de dados.
Pré-condições	Estar conectado na base de testes com as tabelas criadas, assim como os dados necessários.
Passos	1. Invocar o método que lista todos os lançamentos salvos.
Resultados esperados	Verificar que essa lista não esteja vazia, assim como verificar os dados de alguns lançamentos salvos no banco, conferindo se estão iguais aos lançamentos salvos nas pré-condições.
Pós-condições	Excluir todos os dados e tabelas utilizados no caso de teste.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT025 – Listar todos os saques	
Resumo	Deve listar todos os saques salvos no banco de dados.
Pré-condições	Estar conectado na base de testes com as tabelas criadas, assim como os dados necessários.
Passos	1. Invocar o método que lista todos os lançamentos do tipo saque.
Resultados esperados	Verificar que essa lista não esteja vazia, assim como verificar os dados de alguns saques salvos no banco, conferindo se estão iguais aos saques salvos nas pré-condições.
Pós-condições	Excluir todos os dados e tabelas utilizados no caso de teste.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT026 – Listar todos os depósitos	
Resumo	Deve listar todos os depósitos salvos no banco de dados.
Pré-condições	Estar conectado na base de testes com as tabelas criadas, assim como os dados necessários.
Passos	1. Invocar o método que lista todos os lançamentos do tipo depósito.
Resultados esperados	Verificar que essa lista não esteja vazia, assim como verificar os dados de alguns depósitos salvos no banco, conferindo se estão iguais aos depósitos salvos nas pré-condições.
Pós-condições	Excluir todos os dados e tabelas utilizados no caso de teste.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT027 – Buscar um lançamento pelo seu ID	
Resumo	Deve buscar um lançamento de acordo com o seu ID.
Pré-condições	Estar conectado na base de testes com as tabelas criadas, assim como os dados necessários.
Passos	1. Invocar o método que busca um lançamento pelo seu ID, como, por exemplo, pelo ID igual a 1.
Resultados esperados	Verificar se esse lançamento não está nulo e se os dados são os mesmos que foram salvos nas pré-condições.
Pós-condições	Excluir todos os dados e tabelas utilizados no caso de teste.
Tipo de implementação	Automatizada
Iteração	1ª Iteração

CT028 - Cadastrar um depósito	
Resumo	Deve cadastrar um depósito no sistema.
Pré-condições	Estar logado no sistema e ter uma conta cadastrada.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Cadastrar. 3. Preencher todos os campos obrigatórios, selecionando o tipo do lançamento com a opção de depósito. 4. Clicar no botão Salvar.
Resultados esperados	Lançamento cadastrado com sucesso. O saldo da conta deve estar igual ao valor do lançamento.
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT029 - Cadastrar um saque	
Resumo	Deve cadastrar um saque no sistema.
Pré-condições	Estar logado no sistema, ter uma conta cadastrada e um depósito cadastrado.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Cadastrar. 3. Preencher todos os campos obrigatórios, com um valor dentro do limite do saldo e selecionando o tipo do lançamento com a opção de saque. 4. Clicar no botão Salvar.
Resultados esperados	Lançamento cadastrado com sucesso. O saldo da conta deve estar igual ao saldo anterior menos o valor desse lançamento.
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT030 - Cadastrar um lançamento com campos em branco	
Resumo	Sistema não deve permitir que um lançamento seja cadastrado com campos obrigatórios em branco.
Pré-condições	Estar logado no sistema e ter uma conta cadastrada.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Cadastrar. 3. Deixar algum campo obrigatório em branco, como, por exemplo, o campo de descrição. 4. Clicar no botão Salvar.
Resultados esperados	Lançamento não deve ser cadastrado, aparecendo a seguinte mensagem “Campo [nome do campo] não pode estar em branco!”.
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT031 - Cadastrar um lançamento com data no futuro	
Resumo	Sistema não deve permitir que um lançamento seja cadastrado com data no futuro.
Pré-condições	Estar logado no sistema e ter uma conta cadastrada.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Cadastrar. 3. Preencher o campo de data com uma data no futuro. Ex.: 10/01/3000. 4. Clicar no botão Salvar.
Resultados esperados	Lançamento não deve ser cadastrado, aparecendo a seguinte mensagem “Data não pode ser no futuro!”.
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT032 - Cadastrar um saque com valor acima do saldo atual	
Resumo	Sistema não deve permitir que um saque seja cadastrado com valor acima do saldo atual da conta.
Pré-condições	Estar logado no sistema e ter uma conta cadastrada.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Cadastrar. 3. Preencher o campo valor com um número acima do saldo atual da conta. 4. Clicar no botão Salvar.
Resultados esperados	Lançamento não deve ser cadastrado, aparecendo a seguinte mensagem "Não se pode efetuar um saque com o valor acima do saldo!".
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT033 - Cadastrar um lançamento com valor zero	
Resumo	Sistema não deve permitir que seja cadastrado um lançamento com valor zero.
Pré-condições	Estar logado no sistema e ter uma conta cadastrada.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Cadastrar. 3. Deixar o campo valor com zero. 4. Clicar no botão Salvar.
Resultados esperados	Lançamento não deve ser cadastrado, aparecendo a seguinte mensagem "O campo valor não pode ser zero!".
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT034 - Cadastrar um lançamento com valor negativo	
Resumo	Sistema não deve permitir que seja cadastrado um lançamento com valor negativo.
Pré-condições	Estar logado no sistema e ter uma conta cadastrada.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Cadastrar. 3. Deixar o campo valor com algum valor negativo. 4. Clicar no botão Salvar.
Resultados esperados	Lançamento não deve ser cadastrado, aparecendo a seguinte mensagem "O campo valor não pode ser negativo".
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT035 - Editar um lançamento	
Resumo	Deve editar um lançamento no sistema.
Pré-condições	Estar logado no sistema, ter uma conta cadastrada e um lançamento cadastrado.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Editar. 3. Editar os campos, principalmente o valor. 4. Clicar no botão Atualizar.
Resultados esperados	Lançamento deve ser editado, aparecendo a seguinte mensagem "O lançamento foi editado com sucesso!". Verificar as informações desse lançamento aparece na tabela de lançamentos e se o saldo foi alterado corretamente com esse valor.
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT036 - Editar um lançamento com campos em branco	
Resumo	Sistema não deve permitir editar um lançamento com os campos em branco.
Pré-condições	Estar logado no sistema, ter uma conta cadastrada e um lançamento cadastrado.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Editar em algum lançamento cadastrado. 3. Deixar algum campo obrigatório em branco, como, por exemplo, o campo de descrição. 4. Clicar no botão Atualizar.
Resultados esperados	Lançamento não deve ser editado, aparecendo a seguinte mensagem “Campo [nome do campo] não pode estar em branco!”.
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT037 - Editar um lançamento com data no futuro	
Resumo	Sistema não deve permitir editar um lançamento com data no futuro.
Pré-condições	Estar logado no sistema, ter uma conta cadastrada e um lançamento cadastrado.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Editar em algum lançamento cadastrado. 3. Preencher o campo de data com uma data no futuro. Ex.: 10/01/3000. 4. Clicar no botão Atualizar.
Resultados esperados	Lançamento não deve ser editado, aparecendo a seguinte mensagem “Data não pode ser no futuro!”.
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT038 - Editar um saque com valor acima do saldo	
Resumo	Sistema não deve editar um saque com valor acima do saldo atual da conta.
Pré-condições	Estar logado no sistema, ter uma conta cadastrada e um lançamento do tipo saque cadastrado.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Editar em algum saque. 3. Alterar o campo valor com um valor acima do saldo atual da conta. 4. Clicar no botão Atualizar.
Resultados esperados	Lançamento não deve ser editado, aparecendo a seguinte mensagem "Não se pode efetuar um saque com o valor acima do saldo!".
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT039 - Editar um lançamento com valor zero	
Resumo	Sistema não deve permitir editar um lançamento com valor igual à zero.
Pré-condições	Estar logado no sistema, ter uma conta cadastrada e um lançamento cadastrado.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Editar em algum lançamento. 3. Deixar o campo valor com zero. 4. Clicar no botão Atualizar.
Resultados esperados	Lançamento não deve ser editado, aparecendo a seguinte mensagem "O campo valor não pode ser zero!".
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT040 - Editar um lançamento com valor negativo	
Resumo	Sistema não deve permitir editar um lançamento com valor negativo.
Pré-condições	Estar logado no sistema, ter uma conta cadastrada e um lançamento cadastrado.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Editar em algum lançamento. 3. Deixar o campo valor com algum valor negativo. 4. Clicar no botão Atualizar.
Resultados esperados	Lançamento não deve ser editado, aparecendo a seguinte mensagem "O campo valor não pode ser negativo!".
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração

CT041 - Excluir um lançamento	
Resumo	Deve excluir um lançamento.
Pré-condições	Estar logado no sistema, ter uma conta cadastrada e um lançamento cadastrado.
Passos	1. Clicar no menu de lançamentos. 2. Clicar no botão Excluir em algum lançamento. 3. Confirmar a exclusão.
Resultados esperados	Lançamento deve ser excluído, aparecendo a seguinte mensagem "O lançamento foi excluído com sucesso!".
Pós-condições	Excluir todos os lançamentos da base de dados.
Tipo de Implementação	Automatizada
Iteração	1ª Iteração