UNIVERSIDADE FEDERAL DE ALFENAS

INSTITUTO DE CIÊNCIAS EXATAS

Bacharelado em Ciência da Computação

Douglas Miranda da Silva

Renan Domingues Siqueira

FRAMEWORKS PARA AUTOMAÇÃO DE TESTES

Alfenas, 27 de Março de 2013

UNIVERSIDADE FEDERAL DE ALFENAS

INSTITUTO DE CIÊNCIAS EXATAS

Bacharelado em Ciência da Computação

FRAMEWORKS PARA AUTOMAÇÃO DE TESTES

Douglas Miranda da Silva

Renan Domingues Siqueira

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Alfenas como requisito parcial para obtenção do Título de Bacharel em Ciência da Computação.

Orientador: Prof. Rodrigo Martins Pagliares

Alfenas, 27 de Março de 2013

Douglas Miranda da Silva

Renan Domingues Siqueira

FRAMEWORKS PARA AUTOMAÇÃO DE TESTES

A Banca examinadora abaixo-assinada aprova a monografia apresentada como parte dos requisitos para obtenção do título de Bacharel em Ciência da Computação pela Universidade Federal de Alfenas.

Prof. Humberto César Brandão de Oliveira
Universidade Federal de Alfenas

Prof. Douglas Donizeti De Castilho Braz
Universidade Federal de Alfenas

Prof. Rodrigo Martins Pagliares (Orientador)
Universidade Federal de Alfenas

Alfenas, 27 de Março de 2013

RESUMO

Alguma das principais preocupações no desenvolvimento de um software é garantir que ele funcione de maneira correta cumprindo todas as regras pelo qual ele foi desenvolvido. Uma vez que requisitos mudam consideravelmente durante o desenvolvimento, é necessário que o desenvolvedor seja capaz de fazer as alterações necessárias sem medo de que elas possam causar erros inesperados. Com isso testes são inseridos no processo de desenvolvimento para a garantia da sua qualidade. Eles ajudam a equipe a ganhar confiança para prosseguir o desenvolvimento não sendo prejudicada pela dúvida. A queda da confiabilidade do software e perda de dinheiro das empresas muitas vezes são causadas pela não descoberta das falhas de software.

Os testes ajudam a encontrar as falhas no decorrer do desenvolvimento. Eles podem ser aplicados de forma manual ou automática, porém testes manuais são muito repetitivos e propensos a erros. Com isso *frameworks* para testes foram desenvolvidos em várias linguagens de programação para automatizar o processo que era feito de forma manual através de depuradores. Estes *frameworks* permitem que os testes sejam escritos apenas uma vez e executados inúmeras vezes.

Neste sentido este trabalho tem como objetivo estudar técnicas e ferramentas que auxiliam no desenvolvimento de testes automatizados e aplicá-los em um estudo de caso, mostrando como implementar esses testes, as vantagens de utilizá-los e as dificuldades encontradas no decorrer da implementação.

Muitos conceitos fundamentais de como testar em um sistema orientado a objetos utilizando testes unitários serão apresentados. Os testes unitários focam em uma única classe. Existem para certificar de que o código funciona. Eles controlam todos os aspectos no contexto em que a classe é testada, através de substituição de colabores reais por dublês.

Uma outra preocupação deste trabalho é testar de acordo com os requisitos do estudo de caso proposto. Para isso foi necessário o entendimento dos diagramas, cenários e contratos descridos pelo autor do estudo de caso.

Palavras-Chave: ponto de venda, testes unitários, dublês de teste, Mockito.

ABSTRACT

Some of the main concerns in the development of software is to ensure that he works correctly in compliance with all rules by which it was developed. As requirements change considerably during development, it is necessary that the developer is able to make the necessary changes without fear that they may cause unexpected errors. With this testing are included in the development process to ensure quality. They help the team to gain reliance to continue the development isn't affected by doubt. The fall of the software reliability and loss of corporate money are often not caused by discovery of software failures.

The tests help to find bugs and fails in the course of development. They can be applied manually or automatically, however manual tests are very repetitive and error-prone. This way testing frameworks have been developed in some programming languages to automate the process that was done manually by debuggers. These frameworks allow the tests to be written only once and executed many times that you want.

In this way, this work have the objective to study techniques and tools that assist in the development of automated tests and apply them in a case study, showing how to implement these tests, the advantages of using them and the difficulties encountered in the course of implementation.

Many fundamental concepts of how to test in an object-oriented system using unit tests will be presented. The unit tests focus on a single class. There are for sure the code works. They control every aspect in the context in which the class is tested, through replacement real collaborators for stunts.

Another concern of this study is to test according to the requirements of the proposed case study. This required an understanding of diagrams, scenarios and contracts described the author of the case study.

Keywords: point of sale, unit testes, double tests, Mockito.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIAÇÕES

DOC	Depend On Component
JAR	Java Archive
JDK	Java Development Kit
SUT	System Under Test
PDV	Ponto de Venda
UML	Unified Modeling Language

SUMÁRIO

1

Introdução

Este capítulo apresenta alguns detalhes pela razão o qual este trabalho foi desenvolvido, bem como seus objetivos, justificativa e motivação para a realização deste.

Certificar que sistemas de software estão cumprindo todas as regras e o propósito pelo o qual ele foi desenvolvido é um grande desafio devido à alta complexidade dos sistemas e às inúmeras dificuldades relacionadas ao processo de desenvolvimento. Tais dificuldades em que envolve questões humanas, técnicas e de regras de negócio.

Devido às dificuldades encontradas, empresas perdem muito dinheiro todos os anos com os chamados bugs de software. Muitos destes bugs só são descobertos depois que os softwares já estão lançados no mercado e trazem prejuízos imensuráveis (WESTLAND, 2000). Para diminuir este tipo de problema, testes tem sido adotados no desenvolvimento de software.

Testes ajudam a equipe de desenvolvimento ganhar confiança para prosseguir o desenvolvimento sem ser prejudicada pela dúvida, além disto, encontram os bugs mais cedo e consequentemente, possibilitam o reparo com mais antecedência. Através deles é possível que o software chegue ao cliente com maiores chances de sucesso (FOWLER, 1999). Nos testes encontramos duas vertentes: os manuais, realizados pelos desenvolvedores e testadores e os automatizados, que consistem em realizar os mesmos através de ferramentas e frameworks.

Os testes não automatizados são normalmente dispendiosos por serem repetitivos. Assim, essa é uma tarefa propensa a muitos erros. Além disso, os planos de testes são muito longos, e à medida que o software avança, é necessário que os testes passados sejam refeitos. Tudo isso pode acarretar na queda de produtividade, e ainda não garantir a qualidade do sistema. Com a automatização dos testes, todo esse trabalho não é necessário, podendo ainda aumentar a produtividade dos desenvolvedores, além de ser um mecanismo que garante uma maior qualidade (BECK; ANDRES, 2004).

Neste sentido este trabalho tem como objetivo estudar técnicas e ferramentas que auxiliam no desenvolvimento de testes automatizados e aplicá-los em um estudo de caso, mostrando como implementar esses testes, as vantagens de utilizá-los e as dificuldades encontradas no decorrer da implementação.

1.1. Justificativa e Motivação

Algumas das principais preocupações no desenvolvimento de um software são aumentar a garantia da sua qualidade e confiabilidade com menores custos. Uma das maneiras de aumentar as garantias de que um software funciona corretamente é através de testes. Testes não automatizados são normalmente dispendiosos por serem repetitivos, é necessário que o desenvolvedor ou testador execute todo o processo de teste passo a passo para certificar que o software estará funcionando corretamente. Assim, essa é uma tarefa muito propensa a erros.

Outro problema em testes manuais é que os mesmos requerem planos de testes muito longos, e à medida que o software sofre mudanças, é necessário que os testes passados sejam refeitos. Tudo isso pode acarretar na queda de produtividade, e ainda, não garantir a qualidade do sistema.

Frameworks para testes foram desenvolvidos em várias linguagens de programação para automatizar o processo que era feito de forma manual através de depuradores. Estes *frameworks* permitem que os testes sejam escritos apenas uma vez e executados inúmeras vezes. Como os testes automatizados são executados pelo computador então os resultados são verificados sem falha, e como principal vantagem, executam milhares deles em um curto período de tempo.

Segundo Michael Feathers (FEATHERS, 2004), códigos sem testes automatizados é um código ruim. Não importa o quão bem ele foi escrito ou o quanto bonito é a sua orientação a objetos e o encapsulamento. Com testes podemos mudar o comportamento do software de forma verificável e sem eles não podemos realmente saber se o código está cada vez melhor.

Testes automatizados aplicados em todo o processo de desenvolvimento tem demonstrado uma grande melhora na qualidade do produto final entregue ao cliente, reduzindo os custos e prazos no decorrer do desenvolvimento (FOWLER, 1999).

1.2. Problematização

Segundo Christopher Westland (WESTLAND, 2000) é cada vez maior o custo que as empresas e grandes corporações têm para corrigir erros em seus sistemas, erros esses que geralmente são descobertos tardiamente, causando grandes prejuízos financeiros além de reduzir a confiabilidade de um sistema.

O estudo de caso que será testado neste trabalho é baseado na descrição do sistema PDV ProxGER (LARMAN, 2007). Este é um sistema de vendas descrito pelo livro passando por todo o processo de desenvolvimento, desde a coleta dos requisitos até a sua codificação. No entanto nenhuma técnica de testes foi considerada no seu desenvolvimento.

Neste contexto, deseja-se saber:

- Como testar o estudo de caso utilizando das melhores técnicas e *frameworks* de teste para observar o comportamento realizado pelo mesmo?

1.3. Objetivos

1.3.1. Gerais

Recentemente começaram a surgir novos métodos sugerindo uma abordagem de desenvolvimento ágil onde os processos adotados tentam se adaptar às mudanças, apoiando a equipe de desenvolvimento no seu trabalho (BECK, et. al., 2001). Essa prática tem sido adotada por diversas equipes de desenvolvimento e testes automatizados são meios rápidos para garantir a qualidade do produto final sem extensos planos de testes e documentações.

Neste sentido este trabalho tem como o objetivo geral estudar técnicas e ferramentas disponíveis no mercado para a automação de testes, e posteriormente escolher uma das ferramentas e aplicar da melhor maneira no contexto do estudo de caso proposto.

Os testes que serão aplicados no contexto do desenvolvimento vão ser inseridos no sistema de acordo com os seus requisitos para garantir que este funciona de maneira correta. Exemplos e dificuldades encontradas em todo o processo de desenvolvimento dos testes serão apresentados.

1.3.2. Específicos

A fim de se alcançar os objetivos gerais deste trabalho, alguns objetivos específicos foram determinados:

- Estudo de técnicas de teste.
- Estudo da especificação de requisitos do estudo de caso proposto.
- Aplicação das técnicas de testes estudadas.
- Demonstrar que é possível aplicar testes mesmo em um sistema construído sem a utilização destas técnicas.

1.4. Método de Pesquisa

O método de pesquisa utilizado é a experimentação, no qual serão estudadas técnicas de testes e suas formas de implementações, além de *frameworks* e ferramentas de teste. Também serão estudados o sistema de vendas e suas regras de negócio apresentadas no livro discutido. Então essas técnicas aprendidas serão aplicadas no sistema e no final espera-se chegar a um sistema testado.

2

Revisão Bibliográfica

Este capítulo apresenta uma revisão dos conceitos necessários para um bom entendimento deste trabalho.

Testes de software podem ser definidos como o processo de avaliar um sistema ou um componente de um sistema por meios manuais ou automáticos para verificar se ele satisfaz os requisitos especificados ou identificar diferenças entre resultados esperados e obtidos (IEEE STANDARD 729, 1987). O objetivo é identificar as falhas de um projeto de software, possibilitando as equipes de desenvolvimento corrigirem as não conformidades antes da entrega do produto final e poder estimar a confiabilidade e a qualidade do produto.

Entender o modelo de um sistema orientado a objetos é de fundamental importância para compreender o contexto de atuação dos testes e como lidar com as dependências existentes entre os componentes.

2.1. Um Sistema Orientado a Objetos

Um sistema orientado a objetos geralmente é formado por camadas onde cada uma delas realiza um conjunto de tarefas relacionadas em diferentes níveis do sistema. As camadas são compostas por objetos que trocam mensagens entre si para que se tenha a colaboração necessária dos mesmos e assim cumprir os requisitos pelo qual software foi proposto. Um único objeto tem poucas responsabilidades, ou seja, cada um deles por si só não pode fazer muita coisa. Assim eles são obrigados a cooperar entre si para conseguir realizar algo útil no sistema (KACZANOWSKI, 2012). As classes dos objetos podem ser classificadas de duas maneiras segundo Tomek Kaczanowski (KACZANOWSKI, 2012): classes trabalhadoras e classes gerentes.

As classes trabalhadoras são as classes que fazem o trabalho real do sistema, são nelas em que as regras são implementadas e representam a abstração do mundo real, enquanto as classes gerentes fazem apenas a amarração entre as classes trabalhadoras, passando as mensagens entre as mesmas para que elas colaborem uma com as outras. Resumindo, o trabalho delas é orquestrar os trabalhadores e testar esse tipo de classe gera um impacto sério nos testes.

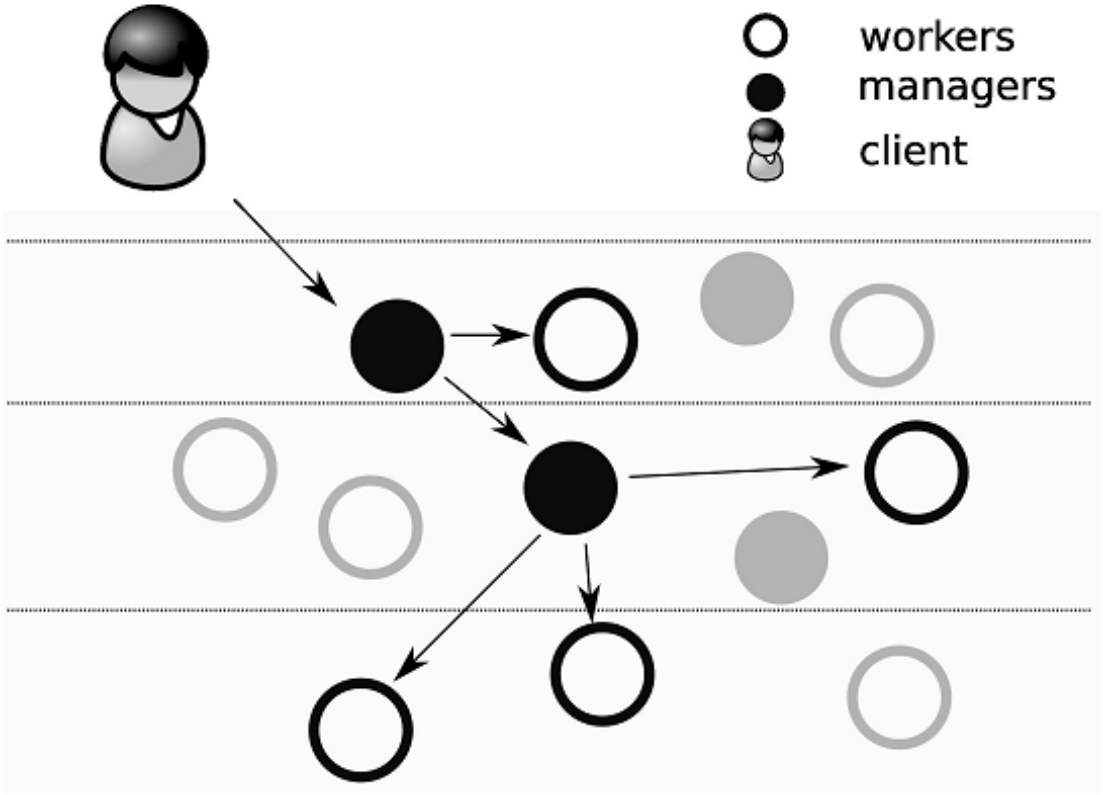

Figura 1 - Sistema Orientado a Objetos**. Fonte:** (KACZANOWSKI, 2012).

Na Figura 1, as linhas são as camadas do sistema, os círculos são objetos, ou seja, instâncias das classes e o grupo de flechas são as mensagens transmitidas entre eles. Observe que existe um cliente nesta figura e que a sua ação dá inicio a uma grande quantidade de atividades no sistema. Veja que os objetos preenchidos de preto são os gerentes e os demais objetos são os trabalhadores. Os objetos gerentes fazem ligações diretas com vários trabalhadores. Eles são responsáveis pela troca de mensagens entre os trabalhadores, que são obrigados a cooperar entre si para conseguir realizar algo útil no sistema.

2.2. Introdução aos Testes

Depois de ter a figura do sistema orientado a objetos em mente, é possível visualizar como a colaboração entre os objetos acarretará em dependências entre os mesmos. As dependências vão afetar diretamente no modo como os testes serão desenvolvidos e para isto, dois conceitos serão apresentados segundo Gerard Meszaros (MESZAROS, 2007):

- SUT (*System Under Test*): É a parte do sistema que está sendo testada(Sistema sobre Teste). Dependendo do tipo de teste, eles podem ser de diferentes granularidades, podendo partir de uma única classe até uma aplicação completa.

- DOC (*Depend On Component*): É qualquer componente exigido pelo SUT para cumprir as suas funções, ou seja, as suas dependências. Usualmente um DOC é da mesma granularidade de um SUT. Um exemplo da sua granularidade em relação ao SUT: Se o SUT é uma classe, ele colabora com outras classes, do mesmo modo, se ele é um módulo ele colabora com outros módulos.

O entendimento destes conceitos é de fundamental importância para a escrita dos testes. O isolamento das dependências é necessário para testar somente o que é relevante em um determinado contexto.

2.3. Testes Unitários

Segundo Tomek Kaczanowski (KACZANOWSKI, 2012) "*Testes unitários focam em uma única classe. Existem para certificar de que o código funciona. Eles controlam todos os aspectos no contexto em que a classe é testada, através de substituição de colaboradores reais por dublês*". O teste unitário não tem conhecimento de nenhum sistema externo ou outras camadas do próprio sistema. Cada parte deve ser testada de maneira isolada.

Como mostrado na Figura 2, apenas o SUT é claramente visível neste tipo de teste, todos os outros objetos e componentes não são tocados pelo teste, eles devem ser substituídos por dublês de teste, eles serão posteriormente descritos neste trabalho.

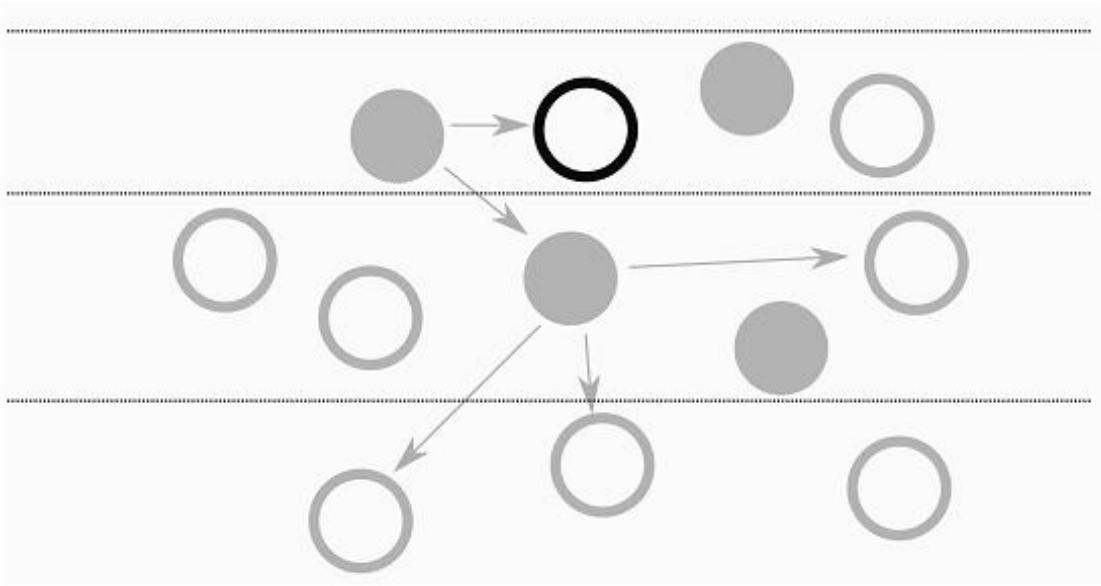

Figura 2 - Escopo Teste Unitário. **Fonte**: (KACZANOWSKI, 2012).

2.4. Dublês de Testes

A maneira correta de garantir que uma classe está apresentando um comportamento correto é testar seus métodos de maneira isolada. Porém, em sistemas construídos em linguagens Orientadas a Objetos, diversas classes se relacionam e seus métodos dependem de serviços realizados por métodos de outras classes, o que dificulta o isolamento de uma classe de influências externas. Essas influências são as chamadas entradas e saídas indiretas.

O grande problema com as entradas e saídas indiretas é não ter controle sobre os dados que entram e saem da classe a ser testada, pois são gerados por outras classes. Assim, quando um método que utiliza serviços feitos por colaboradores é testado, mesmo que ele passe no teste, não é possível ter certeza de sua eficácia, pois não é possível saber se a tarefa feita pelo colaborador foi realizada corretamente, ou, em caso de uma falha, não se sabe se o erro foi causado pelo comportamento errado do(s) método(s) do colaborador ou pelo comportamento errado do método testado.

Os dublês de testes são adicionados no código pelo programador para aumentar o controle sobre as entradas e saídas indiretas. Eles fingem ser os colaboradores e seu comportamento é determinado pelo programador de maneira mais conveniente para a realização dos testes. Assim é mais fácil garantir a eficácia dos testes e a confiabilidade do sistema. Segundo Gerard Meszaros (Meszaros,2007) existem quatro tipos de dublês de testes:

- Dummy: É usado para preparar o ambiente para o teste. Ele não é usado para verificação. É empregado geralmente para preencher uma lista de parâmetros do SUT. Resumindo, é um objeto necessário apenas para a execução do teste devido à existência de alguma regra de negócio, mas o SUT não utiliza seus métodos.
- Stub: É um objeto que fornece respostas pré-definidas para chamadas feitas durante o teste, responde apenas ao que foi programado para ele responder. É usado para passar os dados para o SUT, esses dados são as entradas indiretas.
- Spy e Mock: São utilizados para verificar se o SUT chamou métodos específicos dos colaboradores. Existem algumas diferenças de como eles são usados nos códigos de teste e esta é a razão de terem nomes distintos. Estas diferenças não serão abordadas neste trabalho.

2.5. Frameworks de Testes Unitários

Existem no mercado muitos *frameworks* gratuitos de teste unitários para várias linguagens de programação que podem ser usados para o desenvolvimento das atividades de testes. A maioria deles vem integrada ao ambiente de desenvolvimento como um *plugin*, diferindo em algumas particularidades como usabilidade e interface com usuário. Quase todos trabalham com testes dublês e cada um, obviamente, usa a sintaxe própria de cada linguagem de programação na descrição dos códigos de testes. Abaixo será apresentado uma breve discussão dos principais *frameworks* de teste presentes no mercado. Como o trabalho será desenvolvido na linguagem JAVA, serão abordados alguns *framewoks* para essa linguagem.

2.5.1. JUnit

JUnit (JUnit, 1998) é o *framework* de teste mais conhecido entre os desenvolvedores JAVA. Ele possui integração com vários ambientes de desenvolvimento e é usado por equipes de desenvolvimento ágil.

Dentre as funcionalidades, possui um conjunto com uma boa quantidade de asserções para as comparações entre os valores esperados e os valores obtidos pelos métodos testados. Tem opções para construção de suítes de testes, verificação de lançamento de exceções, teste multithreads para métodos que necessitem ser testados neste contexto e outras opções.

O JUnit é um *framework* de alta qualidade, tem suporte suficiente para testes unitários e pode ser utilizado em qualquer tipo de projeto, seja complexo ou simples, grande ou pequeno. É mais indicado para desenvolvedores que não querem passar muito tempo testando o código, pois não possui muitas opções de configurações de suas funcionalidades.

2.5.2. TestNG

O TestNG (TestNG, 2004) é um *framework* de testes baseado no JUnit. Surgiu para suprir algumas deficiências do mesmo. Ele possui todas as funcionalidades presentes no JUnit além ter outras mais, o que o torna uma ferramenta mais poderosa e mais fácil de usar (KACZANOWSKI, 2012).

Algumas das novas funcionalidades são: notações diferenciadas; rodar os testes arbitrariamente com multithread com várias políticas; configuração flexível dos testes; suporte a testes dirigidos por dados; suporte para parâmetros; modelo de execução poderosa; suportada por uma variedade de ferramentas e *plugins*; utiliza funções padrões do JDK (KACZANOWSKI, 2012).

Uma das principais vantagens do TestNG em relação ao JUnit é que ele possui uma maior legibilidade dos códigos de teste, com notações de testes mais claras e sem ambiguidade. Com isso apresenta uma maior taxa de aprendizado. Outra vantagem é a

maior possibilidade de configurações das funcionalidades tais como a criação de grupos de teste e mais opções para parametrização de testes.

Devido à sua melhor legibilidade e maior quantidade de funcionalidades, neste trabalho será utilizado o TestNG. Muitas das suas funcionalidades serão demostradas nos próximos capítulos.

2.5.3. Mockito

O Mockito (Mockito, 2008) é um *framework* que permite a criação de dublês de teste, chamados também de objetos mocks, em testes automatizados. Ele é *open-source* com foco em testes unitários para testar o comportamento de uma classe isolando-a das dependências dos colaboradores. O Mockito vem para resolver o problema dessas dependências, substituindo-as pelos dublês.

Os dublês de teste fingem serem os reais objetos colaboradores, onde os comportamentos desses objetos são pré-definidos pelo programador da maneira que o mesmo julga mais conveniente para construir o teste. Assim é possível isolar as dependências e testar apenas o comportamento da classe que interessa.

O Mockito funciona de forma integrada com o JUnit, TestNG e outros *frameworks*, formando uma ferramenta poderosa para as atividades de teste. Ele também permite verificar se métodos foram invocados, a quantidade dessas invocações, capturar argumentos, além de testar lançamento de exceções e outras funcionalidades que serão apresentadas nos próximos capítulos.

3

Aplicando os Testes no Estudo de Caso

Neste capítulo são apresentadas as ferramentas utilizadas nos testes e a aplicação dos mesmos no estudo de caso.

Para a aplicação das técnicas de testes que serão estudadas neste trabalho, foi adotado como estudo de caso um sistema de ponto de venda. Este sistema é o PDV ProxyGER descrito por Craig Larman (LARMAN, 2007), o qual não apresenta nenhuma técnica de teste em seu projeto. Todos os requisitos deste sistema foram levantados pelo autor e descritos no livro. O entendimento dos requisitos e a modelagem do sistema serão de fundamental importância para desenvolver os critérios de testes.

3.1. Estudo de Caso

Segundo Craig Larman (LARMAN, 2007) "*Um sistema PDV é uma aplicação computadorizada usada (em parte) para registrar vendas e cuidar de pagamentos; é tipicamente usada por lojas de varejo*". O sistema PDV é formado por vários casos de uso, que segundo Ian Sommerville (SOMMERVILLE, 2007), caso de uso constitui de uma técnica baseada em cenários para levantamento de requisitos. Entender esses requisitos detalhadamente é de fundamental importância para aplicar os testes de maneira correta.

No livro são descritos vários casos de uso, porém com maiores detalhes somente o cenário do caso de uso processar venda para pagamento em dinheiro. Este cenário pode ser descrito como:

> "*um cliente chega em um ponto de pagamento com itens que deseja adquirir. O caixa usa o sistema PDV para registrar cada item comprado. O sistema apresenta um total parcial e detalhes de linha de item. O cliente*

> *entra os dados sobre o pagamento, que são validados e, em seguida, registrados pelo sistema. O sistema atualiza o estoque. O cliente recebe um recibo do sistema e sai com os itens comprados*"(LARMAN, 2007).

Casos de uso e outras características do sistema são maneiras de descrever o seu comportamento. Em testes uma descrição mais detalhada e precisa do comportamento do sistema tem valor. Os contratos de operações podem ajudar, segundo Craig Larman (LARMAN, 2007) eles usam uma forma pré e pós-condições para descrever modificações detalhadas em objetos em um modelo de domínio, como o resultado de uma operação do sistema.

Em cada um dos contratos é apresentado as pré e pós-condições para aquela operação determinada, é um elemento crucial utilizado para fundamentar o começo e o fim dos testes. As pré-condições declaram o que deve sempre ser verdadeiro antes de iniciar um cenário da operação, elas não serão testadas, ao contrário, são as condições que serão assumidas verdadeiras, em testes pode ser considerado como o estado em que o sistema deverá estar antes que o teste que seja executado. As pós-condições são as garantias de sucesso, ou seja, declaram o que deve ser verdadeiro quando bem sucedida a conclusão da operação.

Uma outra maneira de representar visualmente as classes do sistema, ou objetos do mundo real segundo Martin Fowller (FOWLER *apud* Larman 2007) é através do modelo de domínio. O modelo de domínio do sistema, contratos e outros diagramas referentes ao sistema PDV foram colocados como anexo deste trabalho.

Primeiramente é necessário entender um pouco mais sobre o modelo de domínio do PDV ProxGer antes de adentrar nos testes. O modelo foi implementado de acordo com as especificações do livro "Utilizando UML e Padrões" c. A Figura 3 mostra o diagrama de classes do modelo de domínio do sistema PDV usando a notação UML.

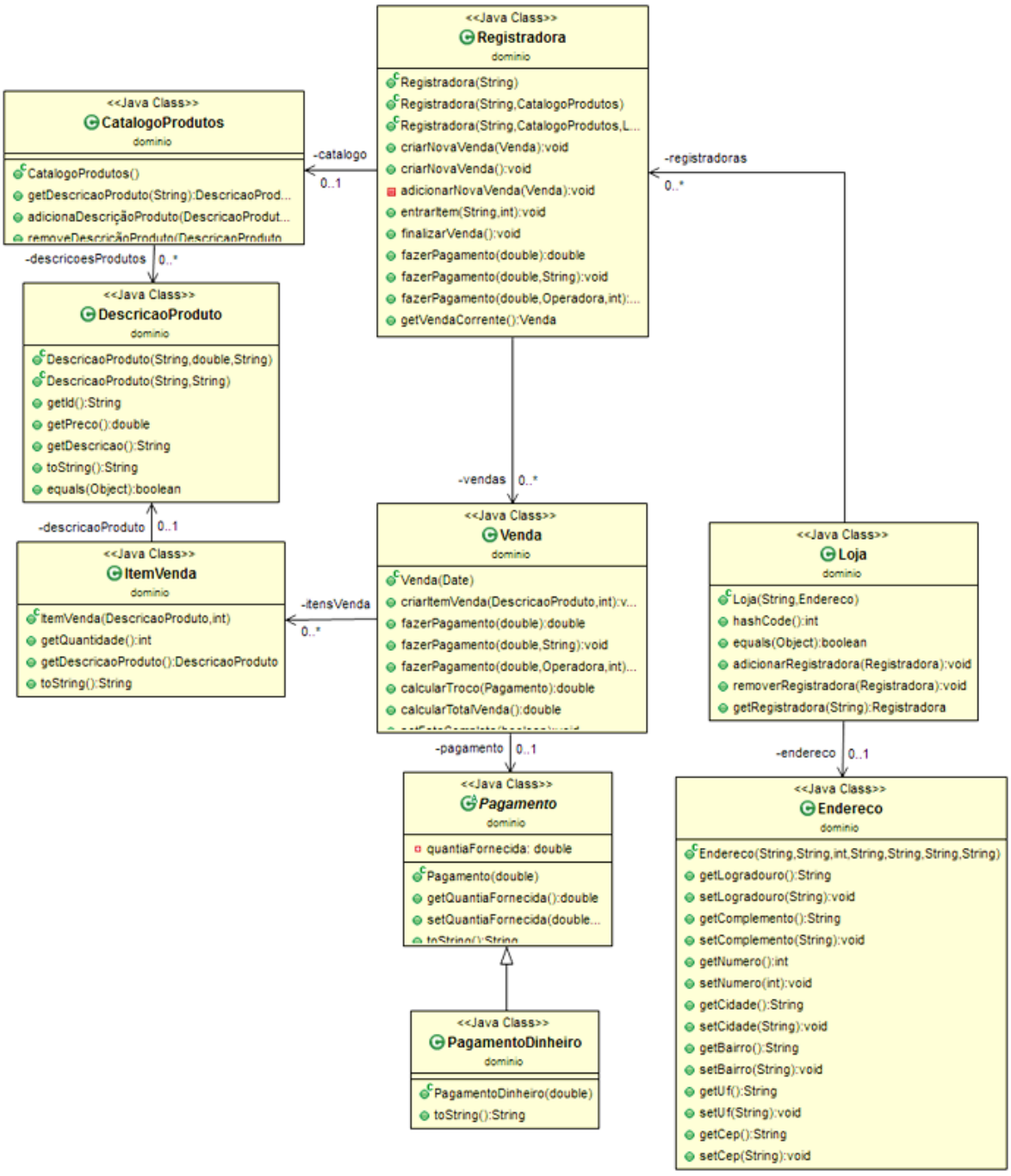

Figura 3 - Modelo de domínio do sistema PDV usando a notação UML. **Fonte:** (LARMAN, 2007).

Observe na Figura 3 todas as classes relevantes para o sistema de vendas são exibidas. O sistema é composto de uma loja na qual possui o seu endereço e um conjunto de registradoras; por sua vez a registradora possui um catálogo de produtos e um conjunto de vendas criadas; o catálogo de produtos possui uma lista com várias descrições de produtos; a descrição do produto contém informações sobre o produto como nome, gênero, preço e um identificador único. A venda possui um conjunto de itens de venda e é responsável pelo pagamento das vendas correntes; cada item de venda possui uma

descrição de um produto com a sua respectiva quantidade; o pagamento é uma classe abstrata que pode ser herdada por vários tipos de pagamento, porém neste trabalho apenas foi implementado o pagamento em dinheiro.

Com o modelo de domínio em mente é possível ver as dependências entre as classes do sistema. As dependências refletem no modo como os testes serão criados. As classes sem dependências se encaixam somente em testes unitários e são mais simples de testar, pois dependem somente de seus dados e operações. As classes com dependências os testes devem ser feitos com dublês. A seguir é apresentado essas duas vertentes de testes.

3.2. Testes Unitários

Por simplicidade, começar pelas classes sem dependências podem ajudar na facilidade de entendimento, tanto nas técnicas de teste, quanto na compreensão do modelo. A primeira classe sem dependência que será analisada é a classe *DescricaoProduto*, cuja a listagem pode ser encontrada na Figura 4. Como próprio nome já diz, essa classe é responsável por fornecer a descrição de um produto. Os atributos necessários para essa tarefa são: *descricao*: um campo texto que contém informações sobre o produto como nome e gênero; *id*: um campo texto utilizado para identificação de um objeto da classe; *preco*: um campo de precisão dupla que representa o preço de um produto.

Neste caso o SUT é a classe *DescricaoProduto*, ou seja, é a classe testada. Como esta classe não possui dependência não haverá o DOC. Este cenário apesar de não acontecer na maioria das vezes em um sistema real, permitirá entender alguns conceitos importantes nos testes. Com esse conhecimento em mente, é hora de começar a escrever os testes para essa classe. A seguir, a Figura 4 mostra o código de domínio da classe *DescricaoProduto*.

```java
package dominio;


public class DescricaoProduto {
    private String id;
    private double preco;
    private String descricao;

    public DescricaoProduto(String id, double preco, String descricao) {
        this.id = id;
        this.preco = preco;
        this.descricao = descricao;
    }

    public DescricaoProduto(String id, String descricao) {
        this(id, 0.0, descricao);
    }

    public String getId() {
        return id;
    }

    public double getPreco() {
        return preco;
    }

    public String getDescricao(){
        return this.descricao;
    }

    @Override
    public String toString() {
        return descricao + "\t\t" + preco + "\t";
    }

    public boolean equals(Object obj) {
        DescricaoProduto desc = null;
        if ((obj instanceof DescricaoProduto) && (obj != null)) {
           desc = (DescricaoProduto) obj;
           return this.id.equals(desc.getId());
        }
        return false;
    }
}
```

Figura 4 - Classe *DescricaoProduto*.

Segundo Tomek Kaczanowski (KACZANOWSKI, 2012), antes de começar a testar, é necessário ter uma lista de casos de teste pronta. Olhando o código da classe, o único método que tem alguma lógica é o método *equals()* que retorna verdadeiro se um objeto é igual ao outro, e falso caso contrário. É um método que merece ser testado. Outro método a ser testado é o construtor. Portanto esses dois métodos serão adicionados à lista de testes.

É mais fácil começar a testar pelo método mais simples. O construtor se encaixa melhor nessa ideia, pois não há lógica de implementação. A primeira coisa a ser feita antes de começar a codificar os testes é criar um pacote específico no projeto para eles, isto permite que os códigos do domínio fiquem separados dos códigos de testes. A idéia de colocar em diferentes pacotes é que os JARs gerados pelo código de produção não serão poluídos pelos códigos de teste. Por convenção, pacote é nomeado como "test". Neste trabalho os testes foram separados em dois pacotes de testes, um chamado de "testesunitarios" e outro chamado de "testescomdubles". Esta divisão foi necessária, pois duas vertentes separadas de testes serão feitas.

Depois de criado o pacote de testes é necessário criar a classe de teste. Por convenção a classe de teste sempre terá o nome da classe de domínio testada pós-fixada da palavra "Test", assim facilmente é identificada de qual classe de domínio são os métodos testados. Neste caso o nome da classe de teste é *DescricaoProdutoTest*. O código da classe de teste criada para *DescricaoProduto* é ilustrado na Figura 5.

```
package testesunitarios;

import org.testng.annotations.Test;

@Test
public class DescricaoProdutoTest {


}
```

Figura 5 - Classe de teste gerada pelo TestNG para a classe *DescricaoProduto*.

Observe na Figura 5 que o TestNG cria a classe de teste com a notação "@Test". Esta notação é necessária para que ele reconheça esta classe como uma classe de teste, em outras palavras, significa que todos os métodos públicos desta classe são considerados métodos de teste. Ela também pode ser utilizada em nível de método, dizendo que aquele método é um método de teste.

Segundo Tomek Kaczanowski (KACZANOWSKI, 2012), o nome do método de teste deve sempre refletir o que o método deve testar. O primeiro método a ser testado será o construtor. Neste método, o que deve ser testado é se o objeto foi criado corretamente, ou seja, com os atributos com os mesmos valores passados no momento da sua construção. O primeiro código de teste é mostrado a seguir na Figura 6.

```java
package testesunitarios;

import org.testng.annotations.Test;
import static org.testng.Assert.*;

import dominio.DescricaoProduto;

@Test
public class DescricaoProdutoTest {

  public void construtorDeveSetarIdPrecoEDescricao(){
      DescricaoProduto descricaoProduto = new DescricaoProduto("01", 3.75, "Chocolate Talento");

      assertEquals(descricaoProduto.getId(), "01");
      assertEquals(descricaoProduto.getPreco(), 3.75);
      assertEquals(descricaoProduto.getDescricao(), "Chocolate Talento");
  }
}
```

Figura 6 - Teste *construtorDeveSetarIdPrecoEDescricao*().

Observe na Figura 6 que o nome do teste criado reflete exatamente o que será testado. Nesse caso o construtor deve atribuir o *id*, *preco* e a *descricao*. Dentro do teste foi criado uma instância da classe *DescricaoProduto* com alguns parâmetros em seu construtor e depois foi utilizado o método *assertEquals()*. O *assertEquals()* é um método do TestNG para a comparação entre valores iguais. Este método possui sobrecarga para permitir a comparação de diversos tipos de valores. Neste caso esse método recebe dois parâmetros, o valor atual e o valor esperado respectivamente. O teste só passa se todas as comparações forem verdadeiras. A Figura 7 demonstra o resultado da execução deste teste.

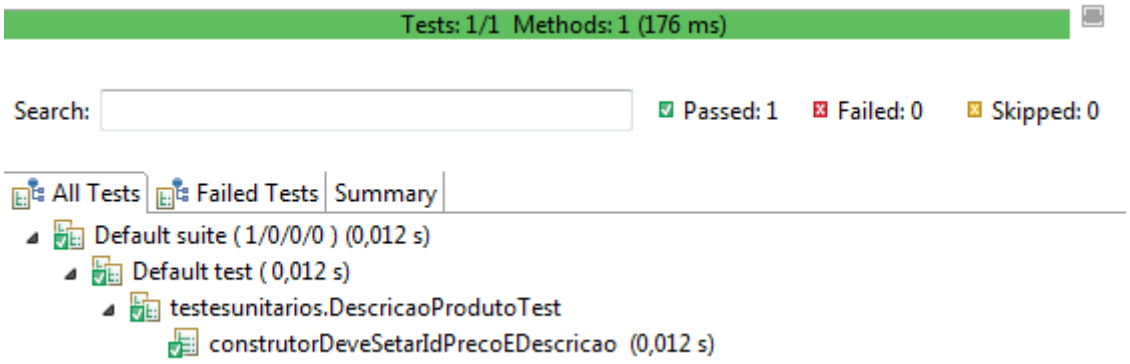

Figura 7 - Execução do Teste *construtorDeveSetarIdPrecoEDescricao*().

A barra verde indica que o teste passou com sucesso. Caso o método tivesse falhado, uma barra vermelha seria mostrada e uma mensagem indicaria onde ocorreu o erro. O TestNG também apresenta a descrição com o motivo da falha. Na Figura 8 foi feito um exemplo para forçar a quebra do teste.

```java
package dominio;


public class DescricaoProduto {
    private String id;
    private double preco;
    private String descricao;

    public DescricaoProduto(String id, double preco, String descricao) {
        this.id = id;
        //this.preco = preco;
        this.descricao = descricao;
    }
```

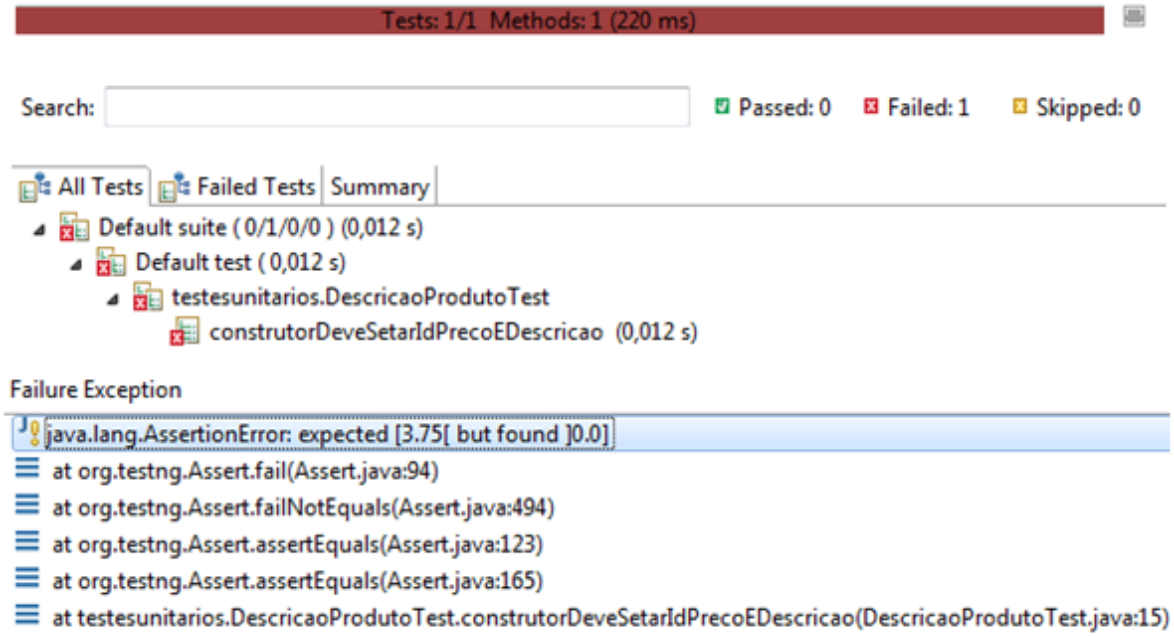

Figura 8 - Quebra do Teste *construtorDeveSetarIdPrecoEDescricao*().

Note que o erro foi causado pelo comentário adicionado no construtor da classe *DescricaoProduto*. Este comentário fez com que valor 3.75 do preço do produto não fosse atribuído no atributo *preco*, assim quando o TestNG fez a comparação do valor esperado do preço com o método *getPreco()* o teste falhou. A Figura 8 também apresenta uma linha destacada no qual o TestNG apresenta uma descrição da causa da falha do teste.

Prosseguindo nos testes, ainda falta testar o método *equals()* da classe *DescricaoProduto*. Lembrando que o método *equals()* desta classe faz a comparação entre

dois objetos, retornado um booleano verdadeiro se os mesmos forem iguais ou falso caso contrário. A seguir é apresentado, Figura 9, o código de teste para este caso.

```
public void deveVerficarSeDuasDescrioesProdutosSaoIguais() {
    DescricaoProduto descricaoDoProduto = new DescricaoProduto("01", 5.00, "Chiclete Trident");
    DescricaoProduto descricaoDeOutroProduto = new DescricaoProduto("01", 5.00, "Chiclete Trident");

    assertTrue(descricaoDoProduto.equals(descricaoDeOutroProduto));
}
```

Figura 9 - Teste *deveVerificarSeDuasDescricoesProdutosSaoIguais*().

Com base no código acima, foram criados duas instâncias de *DescricaoProduto* com os mesmos valores passados para os construtores, ou seja, duas instâncias com os atributos iguais. A Figura 10 apresenta o resultado a execução da classe de teste *DescricaoProdutoTest*.

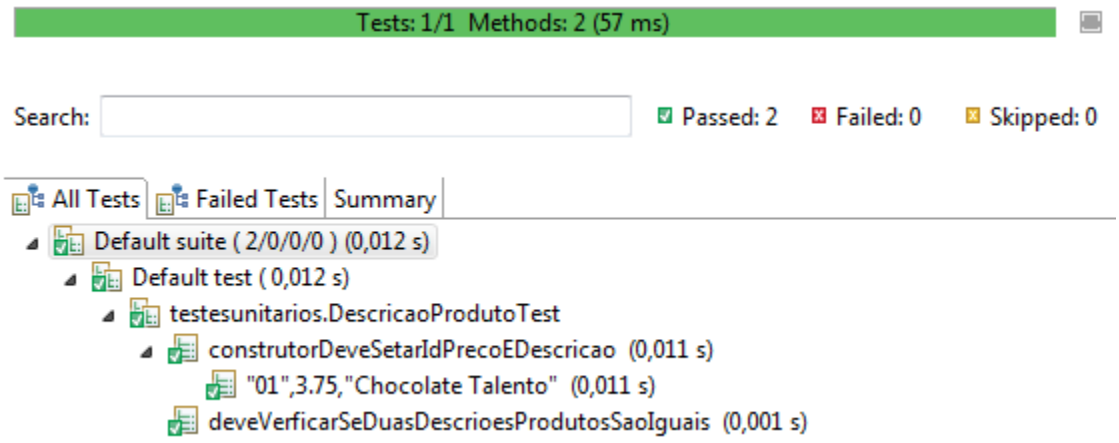

Figura 10 - Execução doTeste *deveVerificarSeDuasDescricoesProdutosSaoIguais*().

O teste também passou. No código deste teste foi utilizada uma nova asserção. Trata-se do método *assertTrue()* do TestNG. Ele verifica se o parâmetro passado a ele é um valor booleano verdadeiro. Se esta condição for satisfeita, o teste continua a execução; caso contrário o teste falha. Como o parâmetro passado foi o retorno da execução do método *equals()* da classe de domínio, significa que este método retornou verdadeiro na comparação dos objetos e funcionou como esperado.

Melhorias nos testes ainda podem ser feitas. É possível fazer com que o teste do construtor execute para vários produtos. Para isto podemos adicionar vários objetos da classe *DescricaoProduto* e realizar todas as comparações anteriores. A Figura 11 apresenta

o teste com dois produtos. Porém note que o código para este teste está ruim, pois existe uma replicação de código.

```
package testesunitarios;

import org.testng.annotations.Test;
import static org.testng.Assert.*;

import dominio.DescricaoProduto;

@Test
public class DescricaoProdutoTest {

    public void construtorDeveSetarIdPrecoEDescricao(){
      DescricaoProduto descricaoProduto = new DescricaoProduto("01", 3.75, "Chocolate Talento");

      assertEquals(descricaoProduto.getId(), "01");
      assertEquals(descricaoProduto.getPreco(), 3.75);
      assertEquals(descricaoProduto.getDescricao(), "Chocolate Talento");

      DescricaoProduto descricaoProduto2 = new DescricaoProduto("02", 1.50, "Chiclete Trident");

      assertEquals(descricaoProduto2.getId(), "02");
      assertEquals(descricaoProduto2.getPreco(), 1.50);
      assertEquals(descricaoProduto2.getDescricao(), "Chiclete Trident");

  }
}
```

Figura 11 - Teste *construtorDeveSetarIdPrecoEDescricao*() para dois produtos.

O TestNG fornece uma solução para este código de teste ruim. Ele permite executar um teste para entradas distintas. Este tipo de teste é chamado de teste parametrizado. As entradas podem ser definidas usando a notação "@DataProvider" em um método estático que retorna o conjunto de dados a serem inseridos no teste. Logo após são colocados os parâmetros correspondentes no método de teste e então é adicionado o *dataprovider* na notação do teste. A Figura 12 exemplifica este tipo de modificação.

```
@Test
public class DescricaoProdutoTest {

    @DataProvider
    private static final Object[][] getDescricaoProdutoComOPreco(){
        return new Object[][] {
                {"01", 3.75, "Chocolate Talento"},
                {"02", 1.50, "Chiclete Trident"}};
    }

    @Test(dataProvider="getDescricaoProdutoComOPreco")
    public void construtorDeveSetarIdPrecoEDescricao(String id, double preco, String descricao){
        DescricaoProduto descricaoProduto = new DescricaoProduto(id, preco, descricao);

        assertEquals(descricaoProduto.getId(), id);
        assertEquals(descricaoProduto.getPreco(), preco);
        assertEquals(descricaoProduto.getDescricao(), descricao);
    }
```

Figura 12 - Teste *construtorDeveSetarIdPrecoEDescricao*() com @DataProvider.

Repare que o código ficou limpo, fez com que os dados ficassem separados da implementação do método de teste. Isto traz como vantagem a flexibilidade de inserção de dados em qualquer momento no teste. Observe na Figura 13 que agora o teste foi executado para os dois conjuntos de dados.

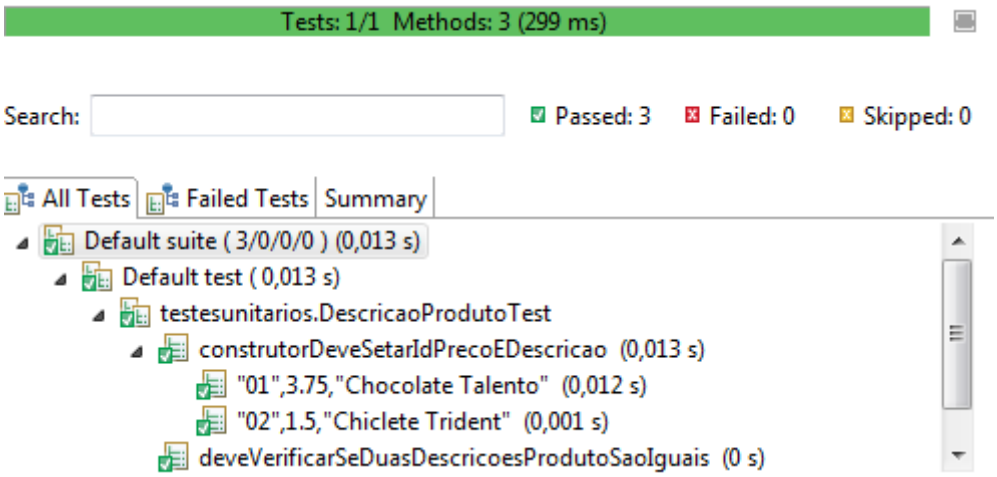

Figura 13 - Execução do Teste *construtorDeveSetarIdPrecoEDescricao*() com @DataProvider.

Depois de todas essas modificações, a Figura 14 ilustra o código atualizado da classe *DescricaoProdutoTest*.

```
package testesunitarios;

import org.testng.annotations.Test;

@Test
public class DescricaoProdutoTest {

    @DataProvider
    private static final Object[][] getDescricaoProdutoComOPreco(){
        return new Object[][] {
                {"01", 3.75, "Chocolate Talento"},
                {"02", 1.50, "Chiclete Trident"}};
    }

    @Test(dataProvider="getDescricaoProdutoComOPreco")
    public void construtorDeveSetarIdPrecoEDescricao(String id, double preco, String descricao){
        DescricaoProduto descricaoProduto = new DescricaoProduto(id, preco, descricao);
        assertEquals(descricaoProduto.getId(), id);
        assertEquals(descricaoProduto.getPreco(), preco);
        assertEquals(descricaoProduto.getDescricao(), descricao);
    }

     public void deveVerficarSeDuasDescrioesProdutosSaoIguais() {
         DescricaoProduto descricaoDoProduto = new DescricaoProduto("01", 5.00, "Chiclete Trident");
         DescricaoProduto descricaoDeOutroProduto = new DescricaoProduto("01", 5.00, "Chiclete Trident");

         assertTrue(descricaoDoProduto.equals(descricaoDeOutroProduto));
    }
}
```

Figura 14 - Classe *DescricaoProdutoTest* atualizada

Com o código atualizado, todos os testes relevantes foram escritos e a classe *DescricaoProduto* funciona como esperado. Portanto esta é a primeira classe de domínio do PDV ProxGer que está certificada contra falhas pelos testes.

Prosseguindo nos testes outra classe de domínio sem dependência que será testada é a classe *Endereco*. Esta classe tem o objetivo de fornecer o endereço para a classe *Loja*. O código de domínio classe *Endereco* é apresentado, Figura 15.

```
package dominio;
public class Endereco {
    private String logradouro;
    private String complemento;
    private int numero;
    private String cidade;
    private String bairro;
    private String uf;
    private String cep;

    public Endereco(String logradouro, String complemento,
            int numero, String cidade, String bairro, String uf, String cep) {
        this.logradouro = logradouro;
        this.complemento = complemento;
        this.numero = numero;
        this.cidade = cidade;
        this.bairro = bairro;
        this.uf = uf;
        this.cep = cep; }

    public String getLogradouro() { return logradouro; }

    public void setLogradouro(String logradouro) { this.logradouro = logradouro; }

    public String getComplemento() { return complemento; }

    public void setComplemento(String complemento) { this.complemento = complemento; }

    public int getNumero() { return numero; }

    public void setNumero(int numero) {this.numero = numero; }

    public String getCidade() { return cidade;  }

    public void setCidade(String cidade) { this.cidade = cidade; }

    public String getBairro() { return bairro; }

    public void setBairro(String bairro) { this.bairro = bairro; }

    public String getUf(){ return uf; }

    public void setUf(String uf) { this.uf = uf; }

    public String getCep() { return cep; }

    public void setCep(String cep) { this.cep = cep; }
}
```

Figura 15 - Classe de domínio *Endereco*.

Observe que no código acima, o único método que merece ser testado é o construtor. Neste caso, as mesmas técnicas aplicadas para testar a classe *DescricaoProduto* serão suficientes para testar classe *Endereco*. O código do teste escrito, Figura 16, é mostrado a seguir.

```
package testesunitarios;

import org.testng.annotations.Test;

public class EnderecoTest {

    @Test
    public void construtorDeveSetarAsInformacoesDeEndereco() {
        Endereco endereco = new Endereco("Rua Gabriel Monteiro da Silva", "Universidade", 700,
                "Alfenas", "Centro", "MG", "37130000");

        assertEquals(endereco.getLogradouro(), "Rua Gabriel Monteiro da Silva");
        assertEquals(endereco.getComplemento(), "Universidade");
        assertEquals(endereco.getNumero(), 700);
        assertEquals(endereco.getCidade(), "Alfenas");
        assertEquals(endereco.getBairro(), "Centro");
        assertEquals(endereco.getUf(), "MG");
        assertEquals(endereco.getCep(), "37130000");
    }

}
```

Figura 16 - Teste *construtorDeveSetarAsInformacoesDeEndereco*().

Observe na Figura 16 que novamente foi usado o método *assertEquals()* para a comparação dos valores esperados e os atuais. Também foi mantido o padrão para o nome do teste. O resultado da execução do teste é mostrado abaixo.

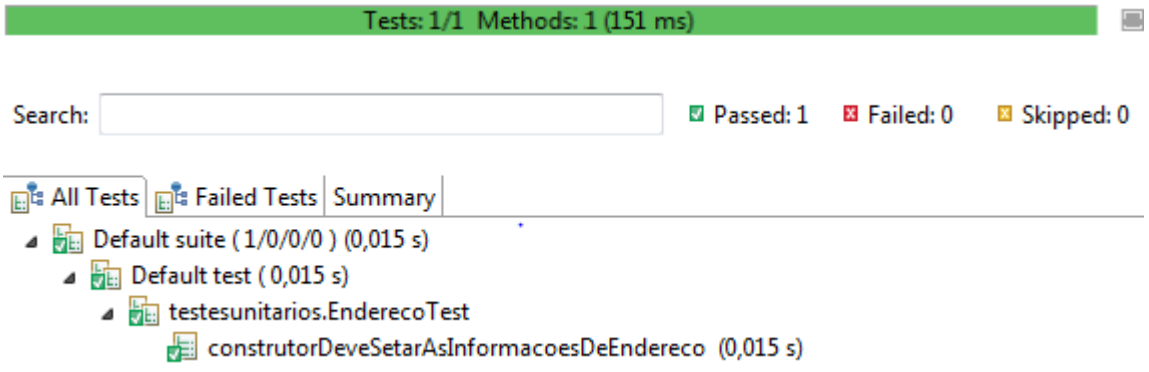

Figura 17 - Execução do Teste *construtorDeveSetarAsInformacoesDeEndereco*().

O teste passou com sucesso e a classe *Endereco* funcionou como esperado. Até agora os testes escritos foram bastante simples devido ao isolamento natural das classes

testadas. Na próxima seção serão abordadas as classes com dependências e os testes ficarão mais interessantes.

3.3. Testes Unitários com Dublês

Neste momento as classes que necessitam ser testadas se relacionam com outras e seus métodos dependem de serviços realizados por métodos de outras classes. Note pela Figura 3 que a maior parte do modelo de domínio se encaixa nesta situação. Para realizar este tipo de teste, os dublês de teste são necessários. Um dublê é apenas um objeto java comum. Ele apenas finge ser um objeto de um tipo específico. Seus métodos possuem comportamentos que são pré-definidos pelo programador, assim os dublês são utilizados para substituir os DOCs, ou seja, substituir as dependências.

A ferramenta que é utilizada para realizar esta tarefa de substituir os objetos é o Mockito (Mockito, 2008). O Mockito usa o método mock() para criar os de dublês de teste, podendo simular objetos de qualquer classe. O ponto principal do dublê de teste é criá-lo e definir como ele irá se comportar.

Neste contexto, a primeira classe que será testada é a *CatalogoProdutos*. O papel desta classe no modelo de domínio é fornecer à classe Registradora a descrição de um produto específico, portanto *CatalogoProdutos* possui um conjunto de descrições de produto. Essa interação será detalhada mais tarde. No momento é necessário isolar essa classe da dependência da classe *DescricaoProduto*. A Figura 18 mostra o código de domínio da classe *CatalogoProdutos*.

```java
package dominio;
import excecoes.DescricaoProdutoInexistente;

public class CatalogoProdutos {
    private List<DescricaoProduto> descricoesProdutos;

    public CatalogoProdutos() {
        descricoesProdutos = new ArrayList<DescricaoProduto>();
        DescricaoProduto d1 = new DescricaoProduto("01", 3.75, "Chocolate Talento");
        DescricaoProduto d2 = new DescricaoProduto("02", 1.50, "Chiclete Trident");
        DescricaoProduto d3 = new DescricaoProduto("03", 2.50, "Lata de Coca-cola");
        DescricaoProduto d4 = new DescricaoProduto("04", 2.00, "Água Mineral");
        descricoesProdutos.add(d1);
        descricoesProdutos.add(d2);
        descricoesProdutos.add(d3);
        descricoesProdutos.add(d4);
    }

    public DescricaoProduto getDescricaoProduto(String id) throws DescricaoProdutoInexistente {
        for (DescricaoProduto desc : descricoesProdutos) {
            if (id.equals(desc.getId()))
                return desc;
        }
        throw new DescricaoProdutoInexistente("Descrição Inexistente para o produto ", id);
    }

    public void adicionaDescricaoProduto(DescricaoProduto descricao){
        descricoesProdutos.add(descricao);
    }

    public void removeDescricaoProduto(DescricaoProduto descricao){
        descricoesProdutos.remove(descricao);
    }
}
```

Figura 18 - Classe de domínio *CatalogoProdutos*.

Seguindo a definição de tipos de dublês de testes de Tomek Kaczanowski (KACZANOWSKI, 2012), o tipo de dublê que será necessário para testar a classe CatalogoProdutos é o Stub. Observe que o DOC, ou seja, a descrição de produtos é sempre passada como um parâmetro para o SUT, portanto, é uma entrada indireta. Repare também que será necessário definir o comportamento do dublê, pois o SUT utiliza um de seus métodos, mais precisamente o método *getId()*. Esta movimentação define o dublê como um Stub.

O primeiro método a ser testado é *adicionaDescricaoProduto()*. Uma forma de saber se uma descrição do produto foi adicionada é adicionar uma descrição, buscá-la e comparar se o objeto retornado da busca é o mesmo que foi adicionado.

Para realizar este teste, a primeira coisa a ser feita é criar um dublê de teste para uma descrição do produto. Isso é feito utilizando o método *mock()* do Mockito. Todos os tipos de dublês são criados da mesma maneira. Eles vão se diferenciar na maneira em

como serão utilizados durante o teste. A criação do dublê é demonstrada na linha abaixo do comentário na Figura 19. Após criar o dublê, o ponto principal é definir o seu comportamento. Isso é possível graças ao método *when()* do Mockito.

Com o método *when()* é possível dizer ao dublê o que ele deve fazer quando algum de seus métodos for invocado. Na Figura 19 quando foi descrito o método *when()* , quer dizer que quando o método *getId()* do dublê for invocado, então ele retornará "05". Neste caso, é neste ponto que este dublê de teste se torna um Stub.

Depois de definido o comportamento do dublê, ele é adicionado ao catálogo de produtos. Logo após ele é buscado utilizando o método *getDescricaoProduto()* passando o mesmo parâmetro que foi configurado para o dublê retornar se o seu método *getId()* fosse invocado. O método *getDescricaoProduto()* percorre sua lista de descrição do produto e utiliza o método *getId()* de cada descrição para retornar a descrição com mesmo *id* que o parâmetro passado. Caso não encontre, a exceção de descrição de produto inexistente será lançada. Este teste vai testar a interação entre as classes *CatalogoProdutos* e *DescricaoProduto*. Assim o dublê será retornado, pois o seu *id* é igual ao parâmetro. Para confirmar, o método *assertEquals()* é invocado para a comparação dos objetos.

```
@Test
public void deveVerificarSeADescricaoProdutoFoiAdicionada() throws DescricaoProdutoInexistente{
    CatalogoProdutos catalogoProdutos=new CatalogoProdutos();

    //Criando Stub De DescricaoProduto
    DescricaoProduto descricaoProduto =  mock(DescricaoProduto.class);
    when(descricaoProduto.getId()).thenReturn("05");

    catalogoProdutos.adicionaDescriçãoProduto(descricaoProduto);

    DescricaoProduto descricaoBuscadaDoCatalogo=catalogoProdutos.getDescricaoProduto("05");
    assertEquals(descricaoProduto, descricaoBuscadaDoCatalogo);
}
```

Figura 19 - Teste *deveVerificarSeADescricaoProdutoFoiAdicionada*().

A seguir, Figura 20, a execução do teste anterior. O teste passou como esperado. Observe que quando os objetos são comparados com o método *assertEquals()* os objetos são comparados pela a sua referência confirmando que era o mesmo objeto que foi adicionado.

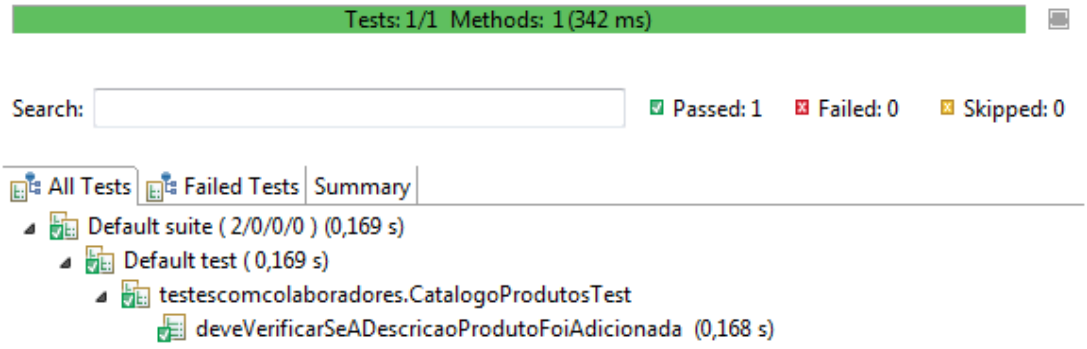

Figura 20 - Execução do Teste *deveVerificarSeADescricaoProdutoFoiAdicionada*().

O segundo método a ser testado é o *removeDescricaoProduto()*. Uma maneira de saber se uma descrição do produto foi removida é adicionar uma descrição, removê-la com próprio método *removeDescricaoProduto()* e depois buscá-la no catálogo. Obviamente, uma exceção de descrição de produto inexistente deverá ser lançada, pois a descrição não estará mais contida na lista do catálogo.

Observe que o código deste novo teste será igual ao anterior até a parte da adição da descrição no catálogo. Depois, o método *removeDescricaoProduto()* será invocado passando como parâmetro o objeto dublê da descrição para ser removido. Para finalizar, será invocado o método *getDescricaoProduto()* passando como parâmetro o id do objeto dublê. Isso deverá gerar o lançamento da exceção descrição de produto inexiste, confirmando que o objeto dublê foi removido do catálogo.

Observe que o teste só passará se a exceção for lançada. Este é outro recurso do TestNG. Para utilizar este recurso é necessário adicionar a notação @Test (expectedExceptions = {classe(s) de exceção a ser lançada}) no método de teste. A notação pode receber vários tipos de classes de exceções separadas por vírgula, assim é possível testar várias exceções em um único teste. Caso nenhuma das exceções seja lançada, o teste falhará. A Figura 21 apresenta o código deste teste.

```
@Test(expectedExceptions = DescricaoProdutoInexistente.class)
public void deveVerificarSeADescricaoProdutoFoiRemovida() throws DescricaoProdutoInexistente{
    CatalogoProdutos catalogoProdutos=new CatalogoProdutos();

    //Criando Stub De DescricaoProduto
    DescricaoProduto descricaoProduto=  mock(DescricaoProduto.class);
    when(descricaoProduto.getId()).thenReturn("05");

    catalogoProdutos.adicionaDescriçãoProduto(descricaoProduto);
    catalogoProdutos.removeDescriçãoProduto(descricaoProduto);

    catalogoProdutos.getDescricaoProduto("05");
}
```

Figura 21 - Teste *deveVerificarSeADescricaoProdutoFoiRemovida*().

Observe que neste teste não é necessário fazer comparações pois ele está esperando o lançamento da exceção *DescricaoProdutoInexistente*. Esta classe de exceção herda da classe *Exception* e seu código é mostrado abaixo na Figura 22. Por questões de organização essa classe foi colocada em um pacote separado chamado "excecoes".

```
package excecoes;

public class DescricaoProdutoInexistente extends Exception {

    private String id;

    public DescricaoProdutoInexistente(String mensagem, String id) {
        super(mensagem);
        this.id = id;
    }

    public DescricaoProdutoInexistente(String string, Throwable throwable) {
        super(string, throwable);
    }

    public DescricaoProdutoInexistente(String string) {
        super(string);
    }

    public DescricaoProdutoInexistente() {
        super();
    }


    public String toString() {
        return super.toString() + "\n" +
                "ID....: " + this.id;
    }
}
```

Figura 22 - Classe de exceção *DescricaoProdutoInexistente*.

Após todas essas considerações, a Figura 23 mostra a execução do teste escrito.

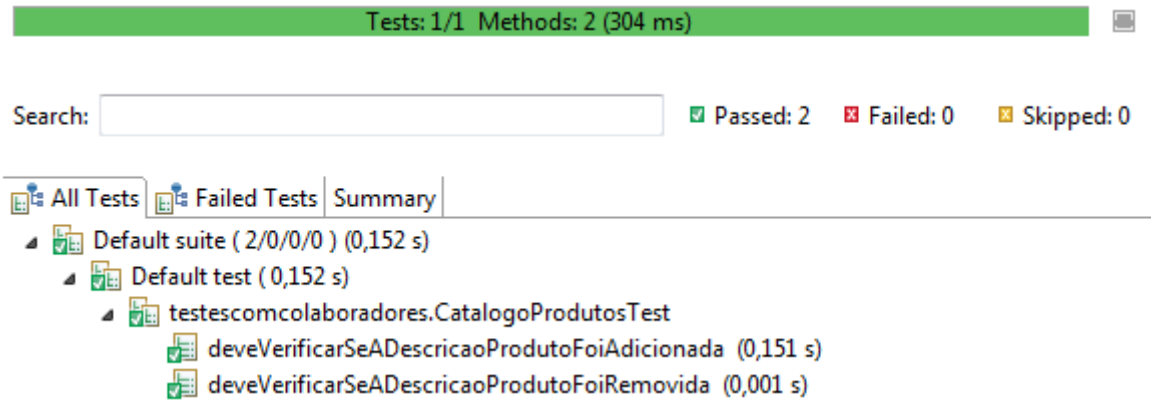

Figura 23 - Execução do Teste *deveVerificarSeADescricaoProdutoFoiRemovida*().

O teste passou como esperado. A exceção foi lançada confirmando que a descrição adicionada ao catálogo foi realmente removida. Apesar de parecer uma inversão de conceitos quando um teste passa quando uma "falha" ocorre, este é um importante recurso quando é preciso saber como o código se comporta em caso de uma exceção.

O último método a ser testado é *getDescricaoProduto()*. Observe que com os testes anteriores este método já é automaticamente testado. No primeiro teste, ele retornou a descrição do produto que era esperada. No segundo, ele lançou a exceção de descrição do produto inexistente quando não encontrou a descrição procurada, como era esperado. Assim seu comportamento já está funcionando de forma correta.

Observe que existe uma replicação de código nos dois últimos testes na primeira linha. O SUT, ou seja, o objeto *catalogoProdutos* está sempre sendo instanciado dentro dos dois testes. Para tentar evitar essa replicação, o primeiro pensamento que pode vir em mente é de que esse objeto poderia ser instanciado como um atributo da classe de teste. Mas atenção, se isso tivesse sido feito e o método de teste *deveVerificarSeADescricaoProdutoFoiAdicionada()* for executado primeiro, o segundo teste falharia pois ainda existiria uma descrição do produto com *id* igual a "05" contida no atributo global e a exceção não seria lançada. Isto criaria uma dependência entre os testes, o que seria muito ruim. Portanto o objeto deve sempre estar sendo instanciado novamente a cada execução de um método teste.

O TestNG fornece este recurso utilizando as notações @BeforeMethod e @AfterMethod. Com elas é possível configurar métodos que serão invocados antes e depois de cada teste, respectivamente. Assim os testes ficam com código mais limpo focando apenas em suas lógicas de implementação. Como exemplos de códigos presentes nestes métodos podem-se destacar: códigos de inicialização de objetos; abertura e fechamentos de arquivos no caso de testes com arquivos; além de quaisquer outras configurações que devem ser feitas antes e depois da execução de cada teste. A Figura 24, com todo o código da classe CatalogoProdutosTest, exemplifica o uso deste artifício.

Por convenção, o nome do método que receberá a notação @BeforeMethod é "setup". Quanto ao @AfterMethod, o nome do método que recebe esta notação deve seguir uma também uma padronização.

```
package testescomcolaboradores;

import static org.testng.Assert.*;

public class CatalogoProdutosTest {
    CatalogoProdutos catalogoProdutos;

    @BeforeMethod
     public void setUp() {
        catalogoProdutos = new CatalogoProdutos();
     }

    @Test
    public void deveVerificarSeADescricaoProdutoFoiAdicionada() throws DescricaoProdutoInexistente{
        //Criando Stub De DescricaoProduto
        DescricaoProduto descricaoProduto = mock(DescricaoProduto.class);
        when(descricaoProduto.getId()).thenReturn("05");

        catalogoProdutos.adicionaDescriçãoProduto(descricaoProduto);

        DescricaoProduto descricaoBuscadaDoCatalogo = catalogoProdutos.getDescricaoProduto("05");
        assertTrue(descricaoProduto.equals(descricaoBuscadaDoCatalogo));
    }

    @Test(expectedExceptions = DescricaoProdutoInexistente.class)
    public void deveVerificarSeADescricaoProdutoFoiRemovida() throws DescricaoProdutoInexistente{
        //Criando Stub De DescricaoProduto
        DescricaoProduto descricaoProduto = mock(DescricaoProduto.class);
        when(descricaoProduto.getId()).thenReturn("05");

        catalogoProdutos.adicionaDescriçãoProduto(descricaoProduto);
        catalogoProdutos.removeDescriçãoProduto(descricaoProduto);

        catalogoProdutos.getDescricaoProduto("05");
    }
}
```

Figura 24 - Classe CatalogoProdutosTest atualizada com @Before.

Repare que neste caso, apenas o método setup() foi necessário. O SUT é instanciado antes de cada teste e nenhuma ação é necessária para após a sua execução.

3.4. Testes Baseados em Contratos de Operações

Nesta sessão serão apresentados os testes de acordo com os contratos de operações presentes no anexo deste trabalho.

Como demonstrado no modelo de domínio, a classe Registradora é a classe que engloba as operações do sistema previstas no caso de uso Processar venda com pagamento em dinheiro. Observe que ela é uma classe gerente na qual faz a troca das mensagens entre *CatalogoProdutos* e *Venda*. É principalmente nesta classe que os testes devem ser focados para ter certeza de que tudo funciona de acordo com os requisitos especificados. Para facilitar o entendimento a Figura 25 apresenta o código desta classe.

```java
public class Registradora {
    private String id;
    private List<Venda> vendas;
    private CatalogoProdutos catalogo;


    public Registradora(String id){
        catalogo = new CatalogoProdutos();
        vendas = new ArrayList();
        this.id = id;
    }

    public void criarNovaVenda() {
        vendas.add(new Venda(new Date()));
    }

    public void entrarItem(String id, int quantidade) throws DescricaoProdutoInexistente {
        Venda venda = null;
        DescricaoProduto descricaoProduto = getCatalogo().getDescricaoProduto(id);
        venda = this.getVendaCorrente();
        venda.criarItemVenda(descricaoProduto, quantidade);
    }

    public void finalizarVenda() {
        this.getVendaCorrente().setEstaCompleta(true);
    }

    public double fazerPagamento(double quantiaFornecida) {
        return this.getVendaCorrente().fazerPagamento(quantiaFornecida); // retorna o troco
    }

    public Venda getVendaCorrente() {
        Venda venda = null;

        if (!vendas.isEmpty()) {
            venda = vendas.get(vendas.size() - 1);
        }
        return venda;
    }

    public CatalogoProdutos getCatalogo() {
        return catalogo;
    }

    public void setCatalogo(CatalogoProdutos catalogo) {
        this.catalogo = catalogo;
    }

    public String getId() {
        return id;
    }
}
```

Figura 25 - Classe de domínio *Registradora*.

A classe *Registradora* possui um catálogo com a descrição dos produtos; a descrição de um produto catalogado é usado pela venda para criar um item de venda. Esta classe também possui uma lista das vendas que são criadas pela *Registradora.* Note que os métodos desta classe são as operações-chave do sistema de vendas.

Para testar esta classe, as técnicas com dublês de testes apresentadas até aqui serão necessárias, porém não serão suficientes. Um novo tipo de dublê de teste será necessário e sua aplicação será demonstrada a seguir.

Primeiramente para testar a interação entre os objetos desta classe com os objetos da classe *Venda* e *CatalogoProdutos*, foi necessária a criação de alguns métodos sobrecarregados (overload). Isto será necessário para que seja possível injetar os dublês de testes na classe *Registradora*. A Figura 26 mostra esta modificação.

```
public Registradora(String id){
    catalogo = new CatalogoProdutos();
    vendas = new ArrayList<Venda>();
    this.id = id;
}

public Registradora(String id, CatalogoProdutos catalogo){
    this.catalogo = catalogo;
    vendas = new ArrayList<Venda>();
    this.id = id;
}

public Registradora(String id, CatalogoProdutos catalogo, Loja loja){
    this.catalogo = catalogo;
    vendas = new ArrayList<Venda>();
    this.id = id;
}

public void criarNovaVenda(Venda v){
    adicionarNovaVenda(v);
}

public void criarNovaVenda() {
    adicionarNovaVenda(new Venda(new Date()));
}

private void adicionarNovaVenda(Venda v){
    vendas.add(v);
}
```

Figura 26 - Sobrecarga de Métodos da Classe *Registradora*.

Note que dois métodos foram sobrecarregados, o método construtor da classe *Registradora* e o método *criarNovaVenda()*. O construtor *Registradora()* foi sobrecarregado para que um catalogo dublê seja injetado nesta classe no qual será utilizado nos testes. Outro método sobrecarregado é o método *criarNovaVenda()*. Esta modificação também foi necessária para que um objeto de dublê de venda seja inserido na registradora.

Uma observação importante é que um método privado foi adicionado nesta classe para evitar a replicação de código, pois se fosse necessário realizar manutenção, as alterações devem refletir nos dois métodos sobrecarregados.

Com estas modificações feitas os testes podem ser iniciados. De acordo com o primeiro contrato, o Contrato C01, o primeiro método a ser testado nesta classe é o *criarNovaVenda()*. Para esse teste não existe nenhuma pré-condição, apenas três pós-condições. Elas são ações que devem ser verificadas para que esta operação seja validada. As pós-condições escritas no Contrato C01 são:

- Foi criada uma instância v de Venda (criação de instância).
- v foi associada com Registradora (associação formada).
- Os atributos de v foram iniciados.

Analisando novamente o SUT, note que ao criar uma nova venda a registradora insere esta venda criada em uma lista de vendas, e o único meio pelo qual a venda pode ser recuperada é pelo método *getVendaCorrente()*. Note se o teste realizar este fluxo, duas das três pós-condições serão satisfeitas. O fluxo descrito é mostrado na Figura 27.

```
@BeforeMethod
public void setUp(){
    registradora = new Registradora("101");
}

@Test
public void deveCriarUmaNovaVenda(){
    Venda venda = mock(Venda.class);
    registradora.criarNovaVenda(venda);

    Venda vendaCorrente = registradora.getVendaCorrente();
    assertEquals(venda, vendaCorrente);
}
```

Figura 27 - Teste *deveCriarUmaNovaVenda*().

Observe que neste teste, o dublê de venda se torna um Dummy. Nenhum método ou atributo do dublê interessa para o teste, apenas a sua presença é necessária. A seguir a execução do teste, Figura 28.

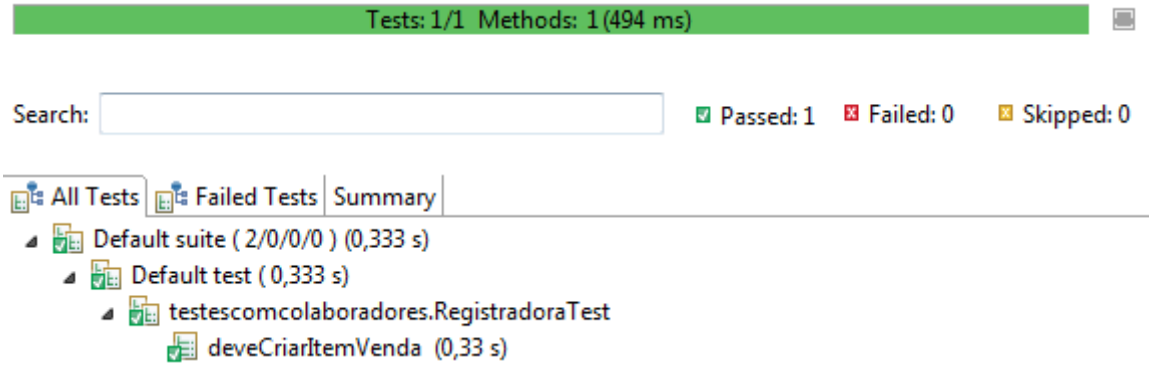

Figura 28 - Execução do Teste *deveCriarUmaNovaVenda*().

A terceira condição diz que os atributos de venda devem ser inicializados. A classe *Venda* possui somente um atributo, a *dataVenda*. Note que para a condição de inicialização dos atributos, somente um teste unitário na classe *Venda* é pertinente, pois tudo o que o teste precisa verificar está somente na classe *Venda*, não existe dependência. A Figura 29 apresenta esse teste.

```
@Test
public  void construtorDeveSetarADataVenda(){
    Date data = new Date();
    Venda venda = new Venda(data);

    assertEquals(data, venda.getData());
}
```

Figura 29 - Teste *construtorDeveSetarADataVenda*().

O segundo contrato que será testado é o Contrato C02. Ele é referente à operação *entrarItem()*. Para está operação existe uma pré-condição e quatro pós-condições. As pós-condições devem ser feitas e verificadas do mesmo modo que o teste anterior, porém para isso deve ser feito a pré-condição. A pré-condição nos testes são o que inicialmente é necessário para prosseguir com a verificação das pós-condições. A pré-condição deste contrato é:

- Existe uma venda em andamento

Para este teste é necessário que haja uma venda em andamento na *Registradora*. Para isso as mesmas operações do teste anterior foram definidas no começo deste segundo teste. Logo após as pós-condições deverão ser analisadas. Para este contrato são elas:

- Foi criada uma instância liv de LinhaDeItemDeVenda (criação de instância).

- liv foi associada com a Venda corrente (associação formada).

- liv.qtdade tornou-se quantidade (modificação de atributo).

- liv foi associada a uma DescricaoDeProduto, com base na correspondência de idItem (associação formada).

O teste para esse contrato é apresentado na Figura 30. Uma nova técnica foi utilizada neste teste. Trata-se do método *verify()* do Mockito. O método *verify()* foi utilizado devido ao encapsulamento. Ele age como um espião verificando se um determinado método de um objeto dublê com seus respectivos parâmetros foram chamados. Segundo Tomek Kaxzanowski (KACZANOWSKI, 2012) é neste ponto que os dublês de teste são classificados como Spy. Para facilitar o entendimento, ele será explicado melhor no contexto das pós-condições do contrato.

```
@Test
public void deveCriarItemVenda() throws DescricaoProdutoInexistente {
    CatalogoProdutos catalogo = mock(CatalogoProdutos.class);
    Registradora registradora = new Registradora("101", catalogo);

    //pré condição
    Venda venda = mock(Venda.class);
    registradora.criarNovaVenda(venda);

    DescricaoProduto descricaoProduto = mock(DescricaoProduto.class);
    when(catalogo.getDescricaoProduto("01")).thenReturn(descricaoProduto);

    registradora.entrarItem("01", 2);

    verify(venda).criarItemVenda(descricaoProduto, 2);
    verify(catalogo).getDescricaoProduto("01");
}
```

Figura 30 - Teste *deveCriarItemVenda*().

Observe no teste que a primeira coisa feita é um dublê de catalogo e logo após ele é inserido no construtor da registradora criada na linha abaixo. Esta declaração é necessária pois nem todos os testes irão utilizar o dublê do catalogo inserido na registradora, então é

necessário definir esta declaração dentro do próprio teste. Mais abaixo é criado um dublê de venda que é associado com a registradora. Isto quer dizer que a registradora agora tem uma venda em andamento. Com isso a pré-condição foi satisfeita para começar o teste.

O próximo passo é fazer o teste e verificar as pós-condições. A primeira pós-condição é verificar se foi criada uma instância de *ItemVenda*. Para criar uma Item venda é preciso de um objeto da classe *DescricaoProduto* e a quantidade desse produto. Note que na classe *Registradora* quando o método *entrarItem()* for invocado internamente, existe uma consulta interna em seu catálogo. Por isso foi criado um dublê da classe *DescricaoProduto* e o seu retorno foi programado no dublê do catálogo.

O método *entrarItem()* é então invocado passando o mesmo código que foi passado para o dublê do catalogo para que o dublê de descrição do produto fosse retornado, além da quantidade com o valor 2. Para as duas pós-condições, sendo que a primeira é que *ItemVenda* foi associada a venda e a segunda, que *DescricaoProduto* foi associada ao Item venda só é possível testar através do método *verify()*. Neste caso o *verify()*, esta verificando se a venda criada chamou o método *CriarItemVenda()* passando o dublê de descrição do produto com a quantidade com valor 2. Isto garante as duas pós-condições: se a classe de *ItemVenda* estiver realizando todas as operações corretamente, então esta classe também funcionará corretamente.

É importante ressaltar que utilizando o *verify()*, o teste só passará se o método descrito for invocado com seus respectivos parâmetros definidos. Falta verificar a pós-condição restante, ela diz que a quantidade deve tornar atributo da classe *ItemVenda*. De forma análoga foi implementado o teste unitário no construtor como anteriormente. A execução dos testes escritos até aqui da classe *RegistradoraTest* é ilustrada na Figura 31.

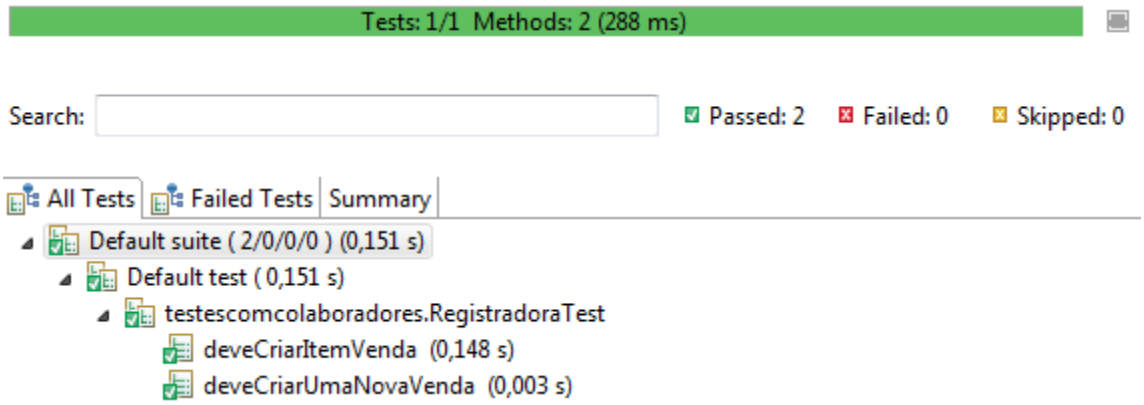

Figura 31 - Execução do Teste *deveCriarUmaNovaVenda*().

O próximo contrato testado é o Contrato 03, ele é referente ao método finalizarVenda(). Neste contrato existe apenas uma pré-condição e uma pós-condição que respectivamente são:

- Existe uma venda em andamento.
- Venda.EstaCompleta tornou-se verdadeira (modificação de atributo).

Existir uma venda em andamento será feito de forma análoga ao teste anterior. O único caso diferente que deve ser tratado é se o atributo booleano *estaCompleta* da classe Venda foi atribuído como verdadeiro. Para isto foi utilizando o método *verify()* novamente para verificar se o método *setEstaCompleta()* da classe *Venda* foi chamado com o argumento verdadeiro. O teste para este contrato é bem simples. Nada de diferente do que foi usado anteriormente. A Figura 32 apresenta o teste para este contrato.

```
@Test
public void deveFinalizarVendaSetandoEstaComoCompleta(){
    Venda venda = mock(Venda.class);
    registradora.criarNovaVenda(venda);

    registradora.finalizarVenda();
    verify(venda).setEstaCompleta(true);
}
```

Figura 32 - Teste *deveFinalizarVendaSetandoEstaComoCompleta*().

A execução dos testes, Figura 33.

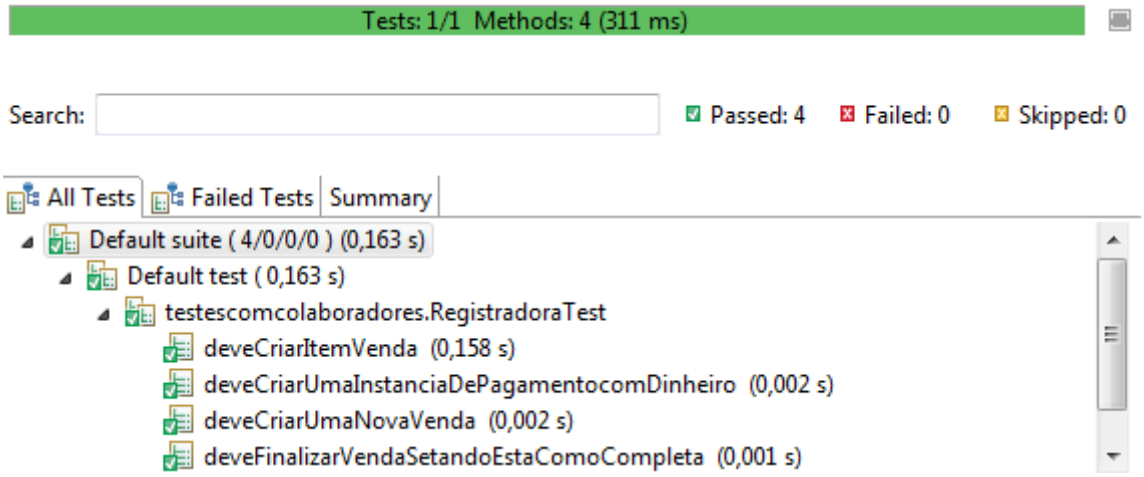

Figura 33 - Execução Teste *deveFinalizarVendaSetandoEstaComoCompleta*().

O último contrato que será testado é o Contrato C04. Neste contrato é testado o método *fazerPagamento()*. Nele existe apenas uma pré-condição e três pós-condições. A pré-condição é apresentada a seguir:

- Uma venda deve estar em andamento.

Essa é uma pré-condição feita anteriormente. Para isso as mesmas operações do teste anterior foram definidas no começo deste segundo teste. Logo após as pós-condições deverão ser analisadas. Para este contrato são elas:

- Foi criada uma instância p de pagamento (criação de instância).
- p.quantiaFornecida tornou-se quantia (modificação de atributo).
- p foi associada com a venda corrente (associação formada).

Observe que a classe *Registradora* possui o método *fazerPagamento()*, mas esta classe apenas delega para a classe *Venda* fazer o pagamento. Tudo que se deve ser testado principalmente é relacionado *Venda* e, posteriormente, testar se a *Registradora* está chamando os métodos de *Venda* corretamente. Por ser uma classe gerente, *Registradora* apenas delega a função de fazer pagamento para *Venda*, pois o que importa para *Registradora* é apenas obter o troco. Por esse motivo, este último teste será feito considerando a classe *Venda*.

O código da classe venda é apresentado na Figura 34. Note que esta classe apresenta um atributo da classe *Pagamento* no qual é utilizado em duas das quatro pós-condições.

```
public class Venda {
    private List<ItemVenda> itensVenda;
    private boolean estaCompleta;
    private Date data;
    private Pagamento pagamento;

    public Venda(Date data) {
        itensVenda = new ArrayList<ItemVenda>();
        this.data = data;
    }

    public void criarItemVenda(DescricaoProduto desc, int quantidade) {
        ItemVenda liv = new ItemVenda(desc, quantidade);
        itensVenda.add(liv);
    }

    public double fazerPagamento(double quantiaFornecida) {
        pagamento = new PagamentoDinheiro(quantiaFornecida);
        return calcularTroco();
    }

    private double calcularTroco() {
        return pagamento.getQuantiaFornecida() - calcularTotalVenda();
    }

    public double calcularTotalVenda() {
        double totalVenda = 0.0;
        for (ItemVenda itemVenda : itensVenda) {
            totalVenda += itemVenda.getDescricaoProduto().getPreco()
                          * itemVenda.getQuantidade();
        }
        return totalVenda;
    }

    public void setEstaCompleta(boolean estaCompleta) {
        this.estaCompleta = estaCompleta;
    }

    private double calcularTroco() {
        return pagamento.getQuantiaFornecida() - calcularTotalVenda();
    }

    public double calcularTotalVenda() {
        double totalVenda = 0.0;
        for (ItemVenda itemVenda : itensVenda) {
            totalVenda += itemVenda.getDescricaoProduto().getPreco()
                          * itemVenda.getQuantidade();
        }
        return totalVenda;
    }

    public void setEstaCompleta(boolean estaCompleta) {
        this.estaCompleta = estaCompleta;
    }

    public Pagamento getPagamento(){
        return this.pagamento;
    }
```

Figura 34 - Classe de domínio *Venda*.

Para testar a primeira e a terceira condição apenas um teste unitário foi necessário. Como o pagamento é criado dentro da própria venda então é possível certificar que a instância foi criada e que este pagamento está associado a venda. A seguir, a Figura 35 com este teste.

```
@Test
public void deveCriarUmaInstanciaDePagamentocomDinheiro(){
    Venda venda = new Venda(new Date());
    venda.fazerPagamento(20.0);
    Pagamento p = venda.getPagamento();

    assertNotNull(p);
    assertTrue(p instanceof PagamentoDinheiro);
}
```

Figura 35 - Teste *deveCriarUmaInstanciaDePagamentoComDinheiro*().

Note que este teste utiliza o método *assertNotNull()* do TestNG. Este método verifica se o parâmetro passado a ele não é nulo para não falhar o teste. A próxima comparação é para certificar que o *pagamento* é referente ao *PagamentoDinheiro*, isto foi necessário, pois *Pagamento* é uma classe abstrata. Com este teste foi criado um pagamento associado a uma venda. A execução dos testes, Figura 36, é mostrada a seguir.

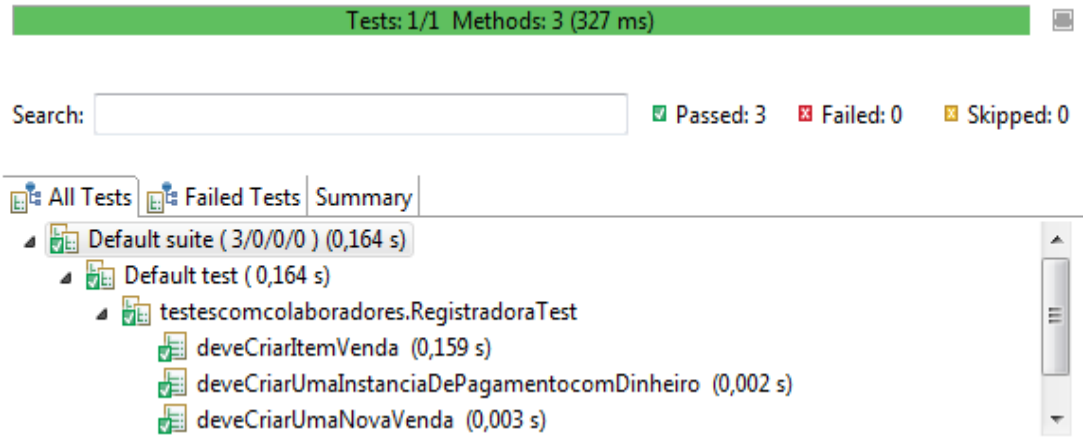

Figura 36 - Execução do teste *deveCriarUmaInstanciaDePagamentoComDinheiro*().

A segunda pós-condição é referente aos atributos da classe *Pagamento*. Para certificar esta condição é necessário apenas um teste unitário simples de construtor. Um teste relacionado à classe *PagamentoDinheiro* será feito. A seguir, a Figura 37 apresenta o teste e a Figura 38, a sua execução.

```
@Test
public void construtorDeveSetarQuantiaEmPagamento(){
    PagamentoDinheiro pagamento = new PagamentoDinheiro(2.55);

    assertEquals(pagamento.getQuantiaFornecida(),2.55);
}
```

Figura 37 - Teste *construtorDeveSetarQuantiaEmPagamento*().

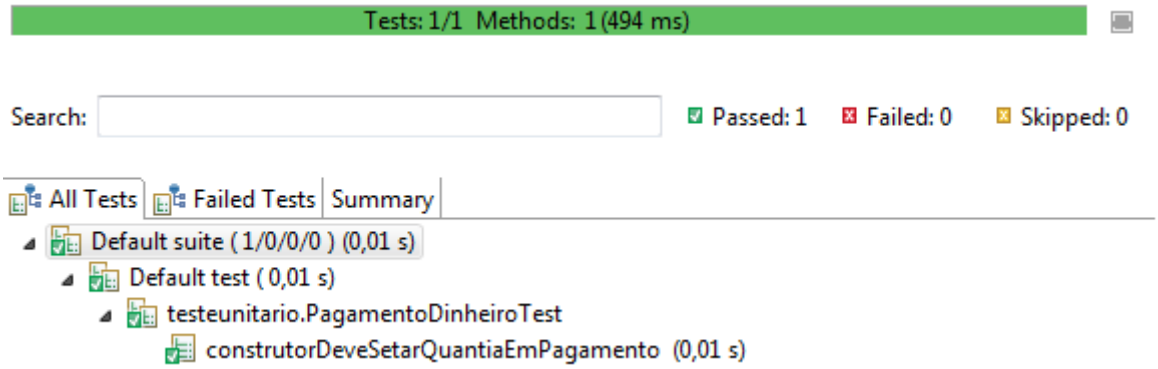

Figura 38 - Execução do Teste *construtorDeveSetarQuantiaEmPagamento().*

4

Conclusões e Trabalhos Futuros

Este capítulo apresenta as conclusões relacionadas com este trabalho.

4.1. Conclusões

Este trabalho possibilitou o aprendizado de técnicas de testes aplicadas em um estudo de caso. Muitas dificuldade foram encontradas desde o entendimento do estudo de caso, implementação do sistema e adequação dos testes de acordo com os requisitos. Os testes aplicados no sistema PDV certificaram de que todos os seus métodos funcionam de acordo com os seus requisitos.

Os testes também possibilitaram que o sistema PDV ProxGer se tornasse flexível, mais confiável e com mais segurança para receber mudanças, pois é possível checar rapidamente se qualquer alteração causou alguma falha no sistema já existente. Os testes garantem uma maior qualidade do sistema, porém existem pontos nos quais é difícil de testar. Dificuldades foram encontradas, principalmente em testar métodos e atributos privados. O TestNG nem o Mockito não são capazes de acessar estes métodos, seria necessário o uso da Api Reflection do JAVA.

Devido a vasta quantidade de funcionalidades tanto do TestNG quanto do Mockito e a razão de simplificar este trabalho ao máximo possível para facilitar o entendimento em um âmbito geral, não foi possível demostrar todas as funcionalidades interessantes que agregariam mais valor aos códigos de testes apresentados. Porém os principais objetivos traçados foram atingidos e o sistema PDV ProxGer foi testado utilizando as técnicas de testes estudadas.

Vale destacar que não existe apenas uma única maneira correta de se escrever testes e eles foram escritos na tentativa de facilitar o entendimento durante sua aplicação.

Existem outras técnicas de testes que atuam em diferentes níveis do sistema que, se combinadas com os dublês de testes, tornam um sistema ainda mais protegido e com maior qualidade.

Concluindo, os dublês de teste são um bom recurso para testar as classes de um sistema em nível de granularidade fina. Com eles é possível testar e garantir que os códigos de uma classe funcionam como o esperado. Porém, em um ambiente de produção, um sistema precisa ser testado em níveis de maior granularidade utilizando outras técnicas de teste para aumentar a sua eficácia.

Por fim, não é possível afirmar com 100% de certeza que um sistema estará livre que qualquer falha se estiver testado, pois não é possível prever todos os pontos de possíveis falhas. Porém as chances de erros são bastante reduzidas, diminuindo os custos para as empresas na realização de reparos, além de agregar valor ao produto pois o mesmo inspirará mais confiança entre os usuários.

4.2. Trabalhos Futuros

Sugestões para trabalhos futuros que os autores deste trabalho propõe são:

- Aplicar as técnicas e os *frameworks* estudados em uma abordagem de desenvolvimento ágil, inserindo algumas das técnicas presentes neste trabalho em um processo de desenvolvimento.

- Comprovar a melhoria do processo de desenvolvimento de um sistema utilizando testes em relação ao mesmo sem testes. Critérios que podem ser considerados: falhas encontradas, tempo de desenvolvimento das equipes, impacto da inserção de novos requisitos, dentre outros.

- Estudar outras técnicas e tipos de testes como: testes de integração, teste de estresse, testes ponta a ponta, testes de desempenho, testes de aceitação, dentre outros.

5
Anexos

Este capítulo apresenta alguns anexos referentes ao estudo de caso do livro.

5.1. Caso de Uso Processar Venda

Escopo: Aplicação PDV ProxGer
Nível: objetivo do usuário
Ator Principal: Caixa
Interessados e Interesses:
– Caixa: deseja entrada de pagamento rápida, precisa e sem erros, pois a falta de dinheiro na gaveta do caixa será deduzida do seu salário.
– Vendedor: deseja comissões sobre vendas atualizadas.
– Cliente: deseja comprar, receber um serviço rápido e com o mínimo de esforço. Deseja a exibição, facilmente visível, dos itens e preço inseridos. Deseja um comprovante da compra, necessário no caso de devoluções de mercadorias.
– Empresa: deseja registrar precisamente as transações e satisfazer os interesses do cliente. Quer garantir que os pagamentos a receber do Serviço de Autorização de Pagamentos sejam registrados. Deseja algum tipo de proteção contra falhas para permitir que as vendas sejam capturadas mesmo se os componentes do servidor (por exemplo, validação remota de crédito) se encontrarem indisponíveis. Deseja uma atualização automática e rápida da contabilidade e do estoque.
– Gerente: deseja poder realizar rapidamente operações de correção e facilmente corrigir problemas do caixa.
– Órgãos fiscais governamentais: desejam cobrar os impostos de cada venda. Podem estar envolvidos vários órgãos, como, por exemplo, federais, estaduais e municipais.
– Serviços de autorização de pagamentos: deseja receber solicitações de autorização digital no formato e protocolo corretos. Deseja contabilizar com precisão seus débitos e pagar para a loja.
Pré-Condições: Caixa está identificado e autenticado.
Garantia de Sucesso (ou Pós-Condições): Venda foi salva. Impostos foram corretamente calculados. Contabilidade e Estoque foram atualizados. Comissões

foram registradas. Recibo foi gerado. Autorizações de pagamento foram registradas.

Cenário de Sucesso Principal (ou Fluxo Básico):

1. Cliente chega a saída do PDV com seus bens e serviços a adquirir.
2. Caixa começa uma nova venda.
3. Caixa insere o identificador do item.
4. Sistema registra a linha de item de venda e apresenta uma descrição do item, seu preço e total parcial da venda. Preço calculado segundo um conjunto de regras de preços.

Caixa repete os passos 3 e 4 até que indique ter terminado.

5. Sistema apresenta o total com impostos calculados.
6. Caixa informa o total ao Cliente e solicita pagamento.
7. Cliente paga e Sistema trata pagamento.
8. Sistema registra venda completa e envia informações de venda e pagamento para Sistema externo de contabilidade (para contabilidade e comissões) e para Sistema de Estoque (para atualizar o estoque).
9. Sistema apresenta recibo.
10. Cliente vai embora com recibo e mercadorias (se houver).

Extensões (ou Fluxos Alternativos):

*a. A qualquer momento, Gerente solicita uma operação de correção:

1. Sistema entra no modo autorizado pelo Gerente.
2. Gerente ou Caixa realiza uma das operações do modo Gerente, por exemplo, modificação do saldo em dinheiro, retoma uma venda suspensa em outro registrador, anula uma venda, etc.
3. Sistema reverte para o modo Autorizado pelo Caixa.

*b. A qualquer momento, Sistema falha:

Para fornecer suporte à recuperação e à correta contabilidade, garanta que todos os estados e os eventos sensíveis das transações possam ser recuperados a partir de qualquer passo do cenário.

1. Caixa reinicia Sistema, registra-se e solicita a recuperação do estado anterior.
2. Sistema restaura estado anterior.

2a. Sistema detecta anomalias que impedem a restauração:

1. Sistema avisa Caixa sobre erro, registra o erro e, então, entra em um novo estado consistente.
2. Caixa começa uma nova venda.

1a. Cliente ou Gerente indica a retomada de uma venda suspensa:

1. Caixa realiza a operação retomada e insere a Identidade para recuperar a venda.
2. Sistema mostra o estado da venda retomada, com subtotal.

2a. Venda não encontrada

1. Sistema avisa Caixa sobre erro.

2. Caixa, provavelmente, começa uma nova venda e re-insere todos os itens.

3. Caixa continua a venda (provavelmente inserindo mais itens ou tratando o pagamento).

2-4a. Cliente diz ao Caixa que tem uma condição de isenção de imposto (por exemplo, idoso, cidadão local):

1. Caixa verifica e depois insere o código de condição de isenção de imposto.

2. Sistema registra a condição (que vai usar durante os cálculos de imposto).

3a. ID do item inválido (não encontrado no sistema):

1. Sistema avisa o erro e rejeita a entrada.

2. Caixa responde ao erro.

2a. Existe um ID do item legível a uma pessoa humana (por exemplo, CUP numérico):

1. Caixa insere manualmente o ID do item.

2. Sistema mostra descrição e preço.

2a. ID do item inválido: sistema avisa erro. Caixa tenta método alternativo.

2b. Não existe identificador de item, mas existe um preço na etiqueta:

1. Caixa solicita ao Gerente executar operação de correção.

2. Gerente realiza correção.

3. Caixa indica entrada manual de preço, insere preço e solicita imposto padrão para essa quantia (como não existe informação de produto, o calculador de imposto não pode inferir como calculá-lo).

2c. Caixa invoca Procurar Ajuda de Produto a fim de obter o ID e preço reais do item.

2d. Caso contrário, Caixa pergunta a um empregado da empresa o preço e identificador reais do item e executa a introdução manual do identificador ou a introdução manual do preço (ver acima).

3b. Existem vários itens do mesmo tipo e rastrear um item físico individual não é importante (por exemplo, 5 pacotes de sanduíches naturais):

1. Caixa pode inserir o identificador do tipo do item e quantidade.

3c. Item exige a introdução manual do tipo e preço (como por exemplo, flores ou cartões com etiqueta de preço):

1. Caixa insere código de tipo manual especial, mais preço.

3-6a. Cliente pede ao Caixa para remover (isto é, anular) um item da compra: isso só é possível se o valor do item é menor que o limite de anulação do Caixa, caso contrário é necessário correção do Gerente:

1. Caixa insere o identificador do item a ser removido da venda.

2. Sistema remove o item e exibe o total parcial atualizado.

2a. Preço do item excede o limite de anulação do Caixa:

1. Sistema avisa que houve erro e sugere correção do Gerente.

2. Caixa solicita correção do Gerente e, após obtê-la, repete a operação.

3-6b. Cliente diz ao Caixa para cancelar a venda:

1. Caixa cancela a venda no sistema.

3-6c. Caixa suspende a venda:

1. Sistema registra a venda de forma que ela fique disponível para acesso a partir de qualquer terminal PDV.
2. Sistema apresenta "recibo de suspensão" que inclui a linha de item e uma identificação da venda usada para recuperar e restaurar a venda.

4a. Preço do item gerado pelo Sistema não é desejado (por exemplo, Cliente se queixa de que algo está sendo oferecido a um preço mais baixo):

1. Caixa solicita a aprovação do gerente.
2 Gerente realiza operação de correção.
3. Caixa insere a correção manual do preço.
4. Sistema apresenta novo preço.

5a. Sistema detecta uma falha na comunicação com o serviço externo de cálculo de impostos:

1. Sistema reinicia esse serviço no nó do PDV e continua.

1a. Sistema detecta que o serviço não reinicia.

1. Sistema avisa o erro
2. Caixa pode calcular e inserir manualmente o imposto ou cancelar a venda.

5b. Cliente diz que tem direito a um desconto (por exemplo, empregado ou cliente preferencial):

1. Caixa avisa sobre uma solicitação de desconto.
2. Caixa insere a identificação do Cliente.
3. Sistema apresenta o total do desconto, com base nas regras para descontos.

5c. Cliente diz que tem um crédito na sua conta que pode ser usado para pagar a compra:

1. Caixa avisa sobre uma solicitação de crédito.
2. Caixa insere a identificação do Cliente.
3. Sistema aplica o crédito até que preço = 0 e reduz o crédito remanescente.

6a. Cliente diz que pretende pagar com dinheiro, mas não tem dinheiro suficiente:

1. Caixa solicita um método alternativo para pagamento.

1a. Cliente diz ao Caixa para cancelar a venda. Caixa cancela a venda no Sistema.

7a. Pagamento em dinheiro:

1. Caixa insere quantia de dinheiro fornecida.
2. Sistema apresenta valor do troco e libera gaveta de dinheiro.
3. Caixa deposita dinheiro fornecido e entrega troco ao cliente.
4. Sistema registra pagamento em dinheiro.

7b. Pagamento a crédito.

1. Cliente insere as informações de sua conta de crédito.

2. Sistema mostra seu pagamento para verificação.
3. Caixa confirma.

3a. Caixa cancela o passo de pagamento.

1. Sistema reverte para o modo “introdução do item”.

4. Sistema envia a solicitação de autorização de pagamento para um sistema externo de serviço de Autorização de Pagamento e solicita sua aprovação.

4a. Sistema detecta uma falha ao tentar colaborar com o sistema externo.

1. Sistema avisa erro ao Caixa.

2. Caixa pede ao Cliente uma forma de pagamento Alternativa.

5. Sistema recebe aprovação do pagamento, Sistema avisa ao Caixa da aprovação e libera a gaveta de dinheiro (para a inserção do recibo de pagamento a crédito e assinado).

5a. Sistema recebe rejeição do pagamento.

1. Sistema avisa rejeição ao Caixa.

2. Caixa solicita ao Cliente uma forma alternativa de pagamento.

5b. Esgotado o tempo de espera da resposta.

1. Sistema avisa ao Caixa o esgotamento.

2. Caixa pode tentar novamente ou solicitar ao cliente uma forma de pagamento alternativo.

6. Sistema registra o pagamento a crédito, que inclui a aprovação do pagamento.

7. Sistema Apresenta o mecanismo para entrada de assinatura do pagamento a crédito.

8. Caixa solicita ao Cliente uma assinatura para pagamento a crédito. Cliente fornece a assinatura.

9. Se assinatura em recibo de papel, Caixa coloca recibo na gaveta de dinheiro e a fecha.

7c. Pagamento em cheque...

7d. Pagamento com débito em conta...

7e. Caixa cancela o passo de pagamento.

1. Sistema reverte ao modo “introdução de itens”.

7f. Cliente apresenta cupons:

1. Antes de tratar o pagamento, Caixa registra cada cupom e sistema reduz o preço conforme estabelecido. Sistema registra os cupons usados por razões contábeis.

1a. Cupom inserido não serve para quaisquer dos itens comprados.

1. Sistema avisa erro ao Caixa.

9a. Existe descontos de preços específicos para certos produtos:

1. Sistema apresenta os formulários de descontos e os recibos de descontos para cada item ao qual se aplica o desconto.

9b. Cliente solicita o recibo especial para presente (os campos de preços não ficam visíveis):

1. Caixa solicita o recibo para presente e Sistema o apresenta.

9c. Impressora não tem papel.

1. Se Sistema detecta a falha, avisará sobre o problema.
2. Caixa repõe papel.
3. Caixa solicita outro recibo.

Requisitos Especiais:

– Interface de Usuário (IU) por tela sensível ao toque em um monitor de tela plana grande. O texto deve ser visível à distância de um metro.

– Resposta de autorização de crédito dentro de 30 segundos em 90% do tempo.

– De alguma forma, queremos uma recuperação robusta quando o acesso ao serviços remotos, tal como o sistema de estoque, estiver falhando.

– Internacionalização de linguagem no texto exibido.

– …

Lista de Variantes Tecnológicas e de Dados:

*a. Correção de Gerente inserida pela passagem de um catão de correção numa leitora de cartão ou inserindo um código de autorização através do teclado.

3a. Identificador de item inserido por leitora a laser de código de barras (quando o item tiver código de barra) ou pelo teclado.

3b. Identificador de item pode ser qualquer um dos seguintes esquemas de codificação: CUP (UCP), EAN, JAN ou SKU.

7a. Informação sobre a conta de crédito inserida por leitura de cartão ou pelo teclado.

7b. Assinatura de pagamento a crédito captada em recibo de papel. No entanto, prevemos que dentro de dois anos muitos clientes vão desejar a captação digital de assinatura.

Frequência de Ocorrência: poderia ser quase contínuo.

Problemas em Aberto:

– Quais são as variações das leis de impostos?

– Deve-se explorar o problema de recuperação do serviço remoto.

– Qual personalização é necessária para diferentes negócios?

– Um Caixa deve levar seu porta-notas da gaveta de dinheiro quando ele sai do sistema?

– O Cliente pode usar diretamente o leitor de cartão ou é o Caixa que deve fazê-lo?

5.1.1 Diagrama de Sequência do Sistema

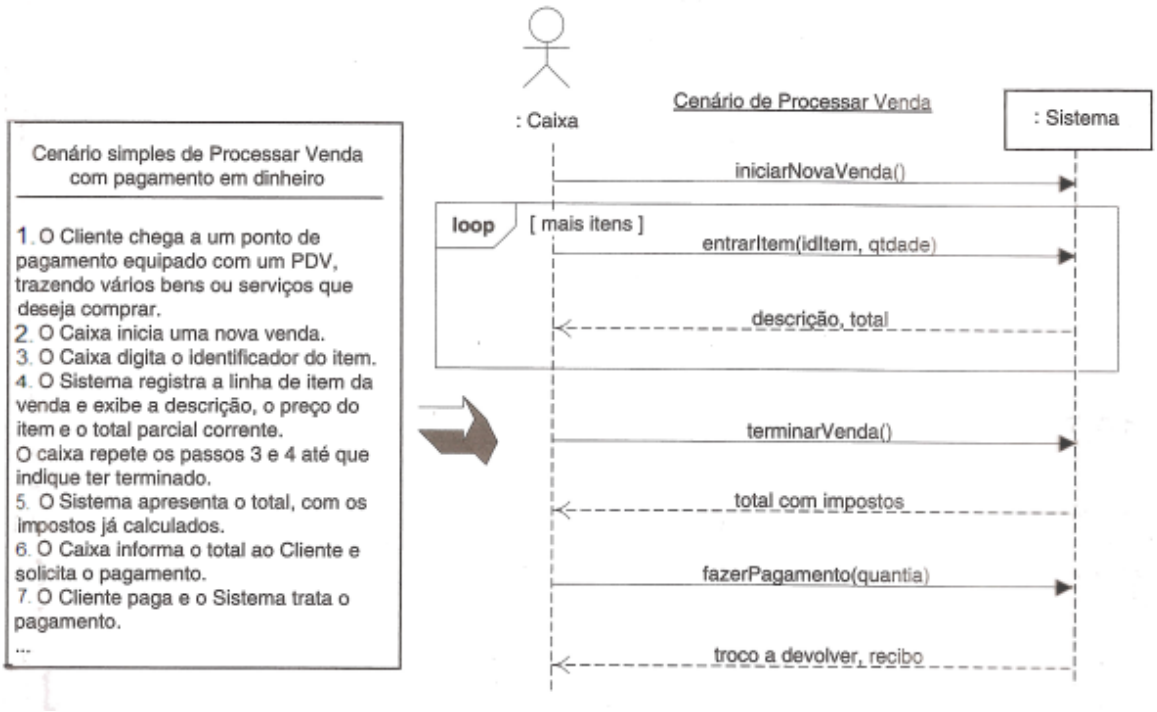

Figura 39 - Diagrama de Sequência do Sistema.

5.1.2. Modelo de domínio

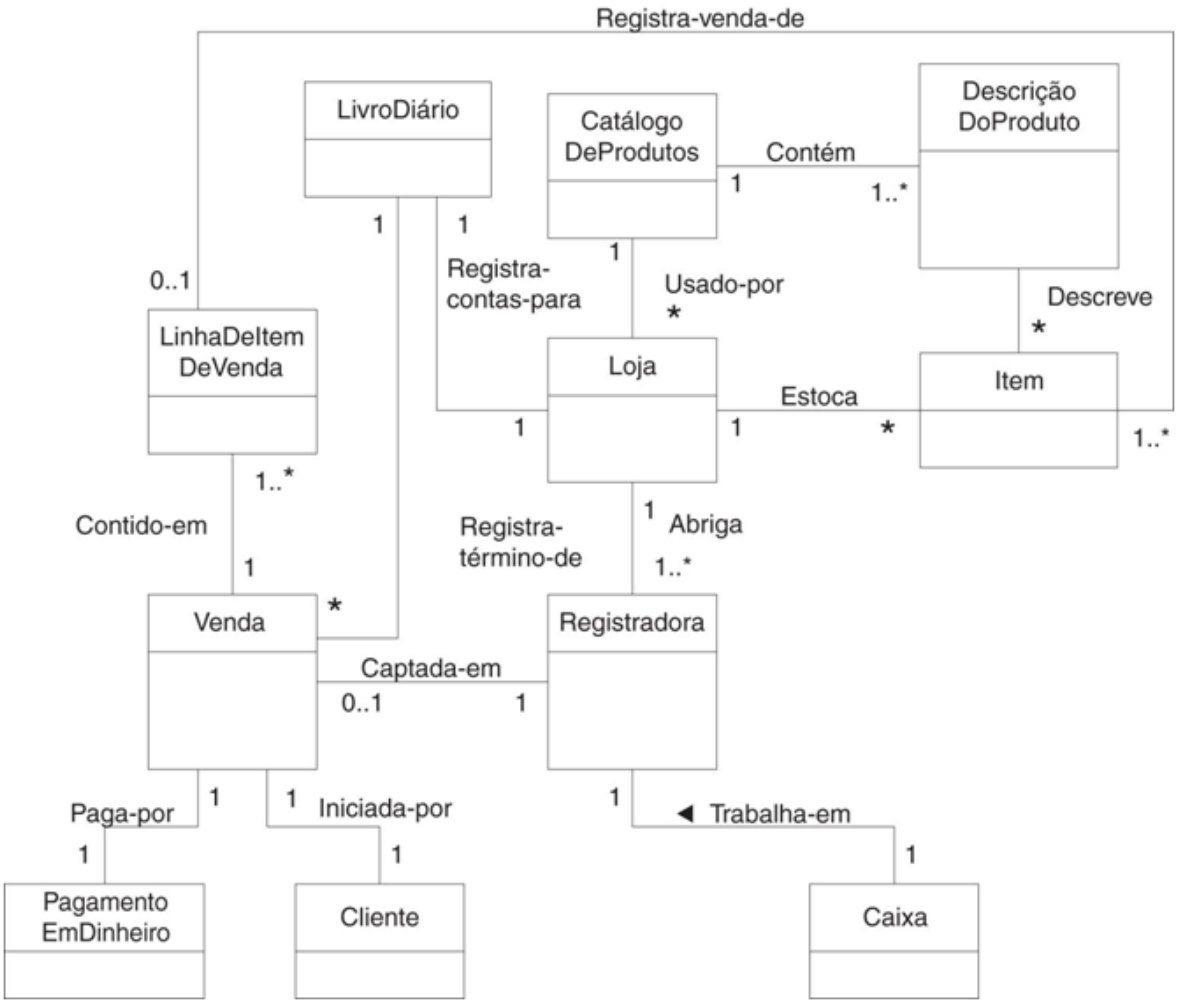

Figura 40 - Um modelo parcial do domínio PDV ProxGer.

5.2. Contratos de Operações

Tabela 1 – Contrato C01: criarNovaVenda.

Operação:	criarNovaVenda()
Referências Cruzadas	Casos de Uso: Processar Venda
Pré-Condições:	Nenhuma.
Pós-Condições:	- Foi criada uma instância v de Venda (criação de instância). - v foi associada com Registradora (associação formada). - Os atributos de v foram iniciados.

Tabela 2 – Contrato C02: entrarItem.

Operação:	entrarItem(idItem: IdItem, qtdade: inteiro)
Referências Cruzadas	Casos de Uso: Processar Venda
Pré-Condições:	Existe uma Venda em andamento
Pós-Condições:	-Foi criada uma istância liv de LinhaDeItemDeVenda (criação de instância). -liv foi associada com a Venda corrente (associação formada). -liv.qtdade tornou-se quantidade (modificação de atributo). - liv foi associada a uma EspecificaçãoDeProduto, com base na correspondência de idItem (associação formada).

Tabela 3 – Contrato C03: finalizarVenda.

Operação:	finalizarVenda()
Referências Cruzadas	Casos de Uso: Processar Venda
Pré-Condições:	Existe uma venda em andamento.
Pós-Condições:	- Venda.EstaCompleta tornou-se verdadeira (modificação de atributo).

Tabela 4 – Contrato C04: fazerPagamento.

Operação:	fazerPagamento(quantia: Moeda)
Referências Cruzadas	Casos de Uso: Processar Venda
Pré-Condições:	Existe uma venda em andamento.
Pós-Condições:	- Foi criada uma instância p de pagamento (criação de instância). - p.quantiaFornecida tornou-se quantia (modificação de atributo). - p foi associada com a venda corrente (associação formada).

REFERÊNCIAS

BECK, K. et al.Agile Manifesto. 2001. Disponível em: <http://www.agilemanifesto.org>. Acessado em: 27 Fev. 2013.

BECK, K.; ANDRES, C. Extreme Programming Explained: Embrace Change. Segunda edição. Ed. Addison-Wesley, 2004

BECK, K. *Test-Driven Development by Example*. Addison Wesley, 2003.

FEATHERS, Michael. Working Effectively With Legacy Code, Prentice Hall, 2004.

FOWLER, M.; BECK, K.; BRANT, J; OPDYKE, W.; ROBERTS, D, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley Professional, 1999.

IEEE Standards Board, "IEEE Standard for Software Unit Testing: An American National Standard, ANSI/IEEE Std 1008-1987" in IEEE Standards: Software Engineering, Volume Two.

KACZANOWSKI, *Tomek. Pratical Unit Testing With TestNG and Mockito*. Amazon, 2012.

JUNIT 1998, Disponível em: <http://junit.org/>. Acessado em: 27 Fev. 2013.

LARMAN, Craing. *Utilizando UML E Pradrões. 3 ed.* Porto Alegre: Editora Bookman, 2007.

MESZAROS, Gerard, *xUnit Test Patterns, Refactoring Test Code*. Addison Wesley, 2007.

Mockito, 2008. Disponível em: <http://mockito.org/>. Acessado em: 27 Fev. 2013.

SOMMERVILLE, I. Engenharia de Software. 8.ed. São Paulo. Pearson Addison-Wesley,2007.

TestNG, 2004, Disponível em: <http://testng.org/>. Acessado em: 27 Fev. 2013.

WESTELAND, J. Christopher. *The cost of erros in software development: evidence from industry*, Departament of information and system Management, The Hong Kong Universaty of Science and Technology, 2000. Disponível em: <http://in953.kelon.org/archives/in953/papers/TheCostOfErrorsInSoftwareDevelopmentEvidenceFromIndustry.pdf>. Acessado em: 09 Fev. 2013.