DISSERTAÇÃO DE MESTRADO

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO

"Integração da abordagem Domain-Driven Design e da técnica Behaviour-Driven Development no Desenvolvimento de Aplicações web"

ALUNA: Eloisa Cristina Silva Santos
ORIENTADOR: Prof. Dr. Delano Medeiros Beder

São Carlos
Junho/2015

CAIXA POSTAL 676
FONE/FAX: (16) 3351-8233
13565-905 - SÃO CARLOS - SP
BRASIL

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INTEGRAÇÃO DA ABORDAGEM DOMAIN-DRIVEN DESIGN E DA TÉCNICA BEHAVIOUR-DRIVEN DEVELOPMENT NO DESENVOLVIMENTO DE APLICAÇÕES WEB

ELOISA CRISTINA SILVA SANTOS

São Carlos - SP
Junho/2015

UNIVERSIDADE FEDERAL DE SÃO CARLOS

CENTRO DE CIÊNCIAS EXATAS E DE TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

INTEGRAÇÃO DA ABORDAGEM DOMAIN-DRIVEN DESIGN E DA TÉCNICA BEHAVIOUR-DRIVEN DEVELOPMENT NO DESENVOLVIMENTO DE APLICAÇÕES WEB

ELOISA CRISTINA SILVA SANTOS

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de São Carlos, como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação, área de concentração: Engenharia de Software.
Orientador: Prof. Dr. Delano Medeiros Beder

São Carlos - SP
Junho/2015

Ficha catalográfica elaborada pelo DePT da Biblioteca Comunitária da UFSCar

S237ia

Santos, Eloisa Cristina Silva.
Integração da abordagem Domain-Driven Design e de técnica Behaviour-Driven Development no desenvolvimento de aplicações web / Eloisa Cristina Silva Santos. -- São Carlos : UFSCar, 2015.
94 f.

Dissertação (Mestrado) -- Universidade Federal de São Carlos, 2015.

1. Engenharia de software. 2. Métodos ágeis. 3. Computação ubíqua. I. Título.

CDD: 005.1 (20ª)

UNIVERSIDADE FEDERAL DE SÃO CARLOS

Centro de Ciências Exatas e de Tecnologia
Programa de Pós-Graduação em Ciência da Computação

Folha de Aprovação

Assinaturas dos membros da comissão examinadora que avaliou e aprovou a Defesa de Dissertação de Mestrado da candidata Eloisa Cristina Silva Santos, realizada em 01/06/2015:

Prof. Dr. Delano Medeiros Beder
UFSCar

Prof. Dr. Auri Marcelo Rizzo Vincenzi
UFG

Prof. Dr. Marcos Lordello Chaim
USP

ESTE TRABALHO É DEDICADO A TODA MINHA FAMÍLIA E AO GABRIEL

AGRADECIMENTOS

Ao Prof. Dr. Delano Medeiros Beder, por ter me concedido a honra de ser sua aluna. Por toda a atenção, amizade, pela excelente orientação e principalmente por ter me ensinado a evoluir como pesquisadora.

À Profa. Dra. Rosângela Aparecida Dellosso Penteado, por em conjunto com o professor Delano, ter me apoiado em diversos momentos e por toda a sua atenção.

A toda minha família, que sempre esteve ao meu lado nessa jornada, me apoiando e acreditando no meu potencial. Em especial, à minha mãe Lourdes, por toda a dedicação.

Ao Gabriel Malimpensa, por toda a paciência, por me aturar nos momentos de estresse, por me ajudar inúmeras vezes, por sempre, sempre estar comigo. Por toda a sua dedicação, meu sincero reconhecimento.

Aos meus colegas de mestrado, foi uma honra ter conhecido cada um e poder ter dividido momentos de dificuldades e de alegrias também. Não serão esquecidos jamais.

Aos meus amigos (as), primos (as), por entenderem toda a minha ausência e ainda assim, não se esquecerem de mim.

Ao CNPq pelo apoio financeiro ao meu trabalho.

RESUMO

Contexto: Os métodos ágeis de desenvolvimento de software surgiram como uma alternativa às abordagens tradicionais, com o intuito de despender menos tempo com documentação e mais com a resolução de problemas de forma interativa e iterativa. Neste contexto, a abordagem de desenvolvimento *Domain-Driven Design* (DDD) representa uma forma de desenvolver software em que o processo de *design* de uma aplicação é guiado pelo modelo de domínio. Em conjunto com a abordagem de desenvolvimento DDD, a existência de testes durante a implementação de uma aplicação é necessária para garantir a qualidade. **Objetivo:** Este trabalho teve como objetivo o estudo da abordagem ágil de desenvolvimento *Domain-Driven Design* (DDD) e as técnicas de teste de software *Test-Driven Development* (TDD) e *Behaviour-Driven Development* (BDD). Um estudo de caso foi construído para exemplificar a integração de cada técnica de teste com o DDD. O estudo de caso foi desenvolvido por meio do *framework Apache Isis*, a partir de um modelo de domínio bem definido. **Metodologia:** Com base na literatura foram extraídos os conceitos da abordagem de desenvolvimento DDD e das técnicas de testes TDD e BDD que proporcionaram a comparação entre as técnicas e posterior implementação para exemplificar a pesquisa. O *framework Apache Isis* foi utilizado neste trabalho porque permite desenvolver aplicações com DDD de forma rápida. No decorrer do desenvolvimento do sistema exemplo deste trabalho, notou-se a possibilidade da criação de um gerador automático de testes e cenários para BDD. **Resultados:** Foi criado um estudo de caso empregando os conceitos de DDD e testes com o TDD e com o BDD. Além disso, foi desenvolvido um protótipo de gerador de testes e cenários para projetos de software que empregam DDD, por meio do *framework Apache Isis* e testes utilizando BDD. **Conclusões:** A junção das técnicas de testes citadas com o DDD visa potencializar o desenvolvimento de aplicações, uma vez que o DDD não aborda nenhuma técnica de teste. A escrita de cenários em linguagem ubíqua é um grande diferencial ao integrar DDD e BDD, pois permite um claro entendimento a todos os envolvidos do projeto. Ademais, a criação do gerador automático agiliza a fase de testes, sendo possível detectar erros que poderiam não ser notados ou apenas serem encontrados com a evolução do projeto.

Palavras-chave: Métodos Ágeis, *Domain-Driven Design* (DDD), *Test-Driven Development* (TDD), *Behaviour-Driven Development* (BDD), Modelo de domínio, Linguagem Ubíqua.

ABSTRACT

Context: The agile methods of software development have emerged as an alternative to traditional approaches, to spend less time on documentation and more time with solving problems interactively and iteratively. In this context, the development approach Domain Driven Design (DDD) is a way of developing software in which the application design process is guided by a domain model. In conjunction with the DDD development approach, the existence of tests during the implementation of an application is required to ensure quality. **Objective:** This work aims to study the agile development approach Domain-Driven Design (DDD) and the software test techniques Test-Driven Development (TDD) and Behaviour-Driven Development (BDD). A case study was built to exemplify the integration of each test technique with DDD. The case study was developed with the support of the Apache Isis framework, from a well-defined domain model. **Methodology:** Based on the literature, the concepts of the DDD development approach and of the TDD and BDD test techniques, that provided the comparison between the techniques, and later, the implementation to illustrate this research, were extracted. The Apache Isis framework was used in this work because it allows develop DDD applications quickly. During the development of the sample system of this work, the possibility of creating an automatic generator tests and scenarios for BDD was noted. **Results:** A case study was created using the concepts of DDD and tests, with TDD and BDD. Furthermore, a prototype of tests and scenarios generator for software projects that use DDD, through the Apache Isis framework and tests using BDD, was developed. **Conclusions:** The combination of test techniques mentioned with DDD aim to boosting the development of applications, since DDD is not associate with any test technique. Writing scenarios with ubiquitous language is a great advantage to integrate DDD and BDD, because it allows a clear understanding for all involved in the project. Moreover, the creation of automatic generator speeds up the testing phase, and can detect errors that might go unnoticed or only be found as the project evolved.

Keywords: Agile Methods, Domain-Driven Design (DDD), Test-Driven Development (TDD), Behaviour-Driven Development (BDD), Domain model, Ubiquitous Language.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ATDD	*Acceptance Test Driven Development*
BDD	*Behaviour-Driven Development*
DDD	*Domain-Driven Design*
HTML	*Hyper Text Markup Language*
MDD	*Model Driven Design*
MVC	*Model View Controller*
TDD	*Test-Driven Development*
XML	*Extensible Markup Language*

SUMÁRIO

Capítulo 1

INTRODUÇÃO

Este capítulo apresenta o contexto da proposta de integração das técnicas de teste de software, TDD e BDD, em conjunto com a abordagem DDD, expondo a motivação, os objetivos, a metodologia empregada, trabalhos relacionados e a organização desta dissertação.

1.1 Contexto e Motivação

Os métodos ágeis para desenvolver sistemas de software são tratados como uma resposta aos métodos classificados como tradicionais que consumem longo tempo com documentação excessiva. Apesar da notória evolução das técnicas e ferramentas ao longo dos anos, a produção de software confiável que estará de acordo com os requisitos e será entregue dentro do tempo estipulado, sem acréscimo de custos, ainda é algo muito difícil (SOARES, 2004).

Métodos ágeis requerem agilidade de processo e ciclos rápidos de desenvolvimento. Todo ciclo origina a implantação de um incremento de uma aplicação (BEDER, 2012). Pressman & Lowe (2009) fundamentam que engenheiros de software necessitam compreender que as empresas requerem adaptações, estratégias e regras de negócio que são alteradas constantemente. Dessa forma, é preciso que as respostas sejam apresentadas de forma imediata e que os interessados (*stakeholders*) não hesitem em fazer alterações mesmo com entregas frequentes por parte dos desenvolvedores. Por essa razão é essencial que a equipe enfatize a agilidade durante o desenvolvimento do produto.

Todo desenvolvimento de sistema deve ser realizado seguindo alguns passos. Dessa forma, deve-se previamente entender o problema, arquitetar uma solução funcional, para posteriormente ser desenvolvida e testada. Também é de suma importância saber administrar as mudanças enquanto se trabalha na solução do problema e possuir mecanismos com o intuito de garantir a qualidade do resultado final. Essas premissas devem ser aplicadas não somente em sistemas convencionais de engenharia de software, mas também pelos desenvolvedores Web (PRESSMAN, 1998).

A abordagem *Domain-Driven Design* (DDD) conduz, de modo ágil, o processo de projetar a aplicação pelo domínio. Essa premissa pode parecer evidente, porém existem diversos sistemas de software que não são implementados em conformidade com o domínio em que atuam. Esse problema pode ser observado no código de várias aplicações atuais, em que as entidades não estão de acordo com a realidade dos usuários e também são difíceis de serem compreendidas (SILVEIRA *et al*., 2012).

Encontrar uma solução adequada para resolver o problema no domínio do cliente sem o entendimento completo desse domínio é tão difícil no DDD quanto em qualquer outra abordagem de desenvolvimento. Por este motivo, uma das principais características em DDD é a Linguagem Ubíqua. Essa linguagem comum entre todos os membros envolvidos (*stakeholders*) no projeto tem o objetivo de associar todas as atividades relacionadas ao desenvolvimento de software, de modo a facilitar a comunicação e o entendimento em relação ao domínio. Os procedimentos existentes no DDD também se aplicam para casos em que os analistas e desenvolvedores possuem maior conhecimento sobre o domínio e refatoram o código e suas entidades conforme a aplicação é implementada e, com isso, práticas de *Test-Driven Development* e refatoração são complementares às práticas propostas pelo DDD (SILVEIRA *et al*., 2012).

O TDD permite ao engenheiro de software verificar, por meio de testes, se o projeto desenvolvido está de acordo com o pretendido mesmo antes de sua total implementação, proporcionando maior segurança e confiabilidade a todos os membros de um projeto. Uma das extensões ao TDD é o *Behaviour-Driven Development* que amplifica o cerne do TDD para o comportamento da aplicação. Dessa forma, o desenvolvimento é guiado pelas especificações que descrevem as funcionalidades do software (REZENDE, 2011).

Atualmente, percebe-se que no mercado de desenvolvimento de software atender aos requisitos do cliente e às constantes mudanças desses requistos é essencial. Por isso se faz cada vez mais necessária a utilização de práticas ágeis de desenvolvimento, como o DDD e de testes como o TDD e o BDD.

1.2 Objetivo

Diante do contexto apresentado, o objetivo deste trabalho é estudar a abordagem ágil de desenvolvimento DDD e as técnicas de teste de software, TDD e BDD. Um estudo de caso foi construído para exemplificar a integração de cada técnica de teste com o DDD. Pretende-se fazer uma comparação e mostrar as vantagens e desvantagens das técnicas de teste abordadas. A proposta de integração visa suprir a lacuna presente no DDD em relação a testes.

Também objetiva-se com esse trabalho, por meio do gerador automático de testes, descrito no Capítulo 5, agilizar a fase de criação de testes e cenários da técnica BDD. Essa fase do ciclo de desenvolvimento é muito custosa. Visto que, o gerador é capaz de criar testes iniciais de validação de atributos das entidades do modelo de domínio, erros mais simples são rapidamente detectados e podem ser corrigidos, antes do projeto alcançar um estágio de desenvolvimento mais avançado.

1.3 Metodologia de desenvolvimento do trabalho

Inicialmente, foram pesquisados os conceitos da abordagem e de cada técnica estudada neste trabalho: DDD, TDD e BDD. Para exemplificar a proposta de integração de cada técnica de teste com o DDD, iniciou-se a definição de um modelo de domínio a partir de um documento de requisitos. Para implementar o sistema exemplo, foi utilizado o *framework Apache Isis* (APACHE ISIS, 2010) de prototipação rápida para projetos em linguagem Java (JAVA, 1995), que são guiados pelo modelo de domínio (DDD). Por meio deste *framework*, foi possível gerar automaticamente a camada de visualização de dados a partir do modelo criado - a camada V do modelo

MVC (*Model View Controller*). Também foram feitos testes com as técnicas de TDD e BDD. Com base nos conceitos estudados e da experiência obtida com a implementação do exemplo, foi possível elaborar uma comparação entre as técnicas de teste, incluindo o ciclo de execução e as vantagens e desvantagens de cada uma.

Posteriormente, notou-se a possibilidade de criar um gerador automático de testes e cenários para projetos que fazem uso do *framework Apache Isis* e da técnica de teste BDD. De forma mais específica, entidades do modelo de domínio podem ter métodos de validação de seus atributos. Esses métodos são usados pelo *framework* nas interfaces geradas automaticamente. Baseado nessa característica do *framework Apache Isis*, o gerador cria testes que invocam os métodos de validação descritos nas entidades. Os testes e os cenários são elaborados no formato do *framework Cucumber* (CUCUMBER, 2014), que implementa a técnica BDD.

1.4 Trabalhos relacionados

Foram encontrados na literatura alguns trabalhos que evidenciam a importância da criação de testes ao desenvolver aplicações utilizando os conceitos de DDD. Mesmo no livro de Evans (2003), é mencionada a necessidade dos testes, porém nenhuma técnica específica foi abordada. Além disso, nenhum trabalho que relacione o DDD com a técnica BDD foi encontrado.

Landre, Wesenberg e Olmheim (2007) citam que antes da escrita de determinada funcionalidade, são feitos testes automatizados em JUnit (JUNIT, 2014). Tais testes cobrem mais de 80% do código, garantindo que as funcionalidades mesmo após refatorações e melhorias, continuarão funcionando corretamente sem nenhuma quebra de integridade.

Sitefane (2007) propõe que para diminuir a sobrecarga de testes manuais e aprimorar o tempo gasto na implementação e no teste de software de um produto, a solução seria a escolha de um método que possibilitasse testar automaticamente cada alteração feita em nível de código. Ainda no trabalho de Sitefane (2007) é evidenciado que os testes não foram implementados apenas com o intuito de

identificar defeitos, e sim com a intenção de validar se o comportamento estava condizente com as regras do domínio.

O trabalho proposto se diferencia por integrar técnicas de teste específicas, sendo elas TDD e BDD, com a abordagem de desenvolvimento DDD, exemplificando tais integrações por meio de estudo de caso: um sistema web de Vídeo Locadora.

1.5 Organização do trabalho

Esta dissertação está organizada em 6 capítulos e 2 Apêndices. O Capítulo 1 apresentou o contexto e motivação do trabalho, os objetivos pretendidos, a metodologia empregada e os trabalhos relacionados.

No Capítulo 2 são apresentados os conceitos relacionados a Métodos Ágeis e *Domain-Driven Design*, tema central deste trabalho.

O Capítulo 3 contém o resultado da revisão bibliográfica relativa aos conceitos de teste de software, no qual são abordados, em especial, o *Test-Driven Development* e o *Behaviour-Driven Development*.

O Capítulo 4 apresenta como foi definido o modelo de domínio e como foi feita a integração de cada técnica de teste TDD e BDD, separadamente, com a abordagem DDD. Além disso, também é apresentada a arquitetura de um projeto *Apache Isis* relacionando-a com a arquitetura em camadas do DDD e como foi feito o desenvolvimento utilizando os conceitos do DDD. Por fim, neste capítulo é apresentada a camada de visualização dos dados gerada automaticamente pelo *framework Apache Isis*.

No Capítulo 5 é apresentado como foi desenvolvido e como é o funcionamento do protótipo de gerador de testes e cenários para projetos de software que empregam DDD e BDD, por meio do *framework Apache Isis*.

O Capítulo 6 apresenta as conclusões obtidas na pesquisa, as limitações detectadas e sugestões de trabalhos futuros.

O Apêndice A apresenta o documento de requisitos do modelo de domínio do estudo de caso em sua versão final e o Apêndice B é um tutorial de utilização do gerador automático de testes definido nesse trabalho.

Capítulo 2

MÉTODOS ÁGEIS & DOMAIN-DRIVEN DESIGN

Este capítulo apresenta uma visão geral sobre Métodos Ágeis no desenvolvimento de software, os conceitos relacionados a abordagem de desenvolvimento DDD e o framework Apache Isis empregado no desenvolvimento do estudo de caso definido neste trabalho.

2.1 Considerações Iniciais

A palavra agilidade nos dias atuais é comumente lembrada ao descrever um moderno processo de software. Uma equipe pode ser vista como ágil se estiver preparada para responder apropriadamente a determinadas mudanças. Mudanças estão presentes durante o desenvolvimento de software, sejam relacionadas à troca de integrantes da equipe, sejam relacionadas a novas tecnologias ou mudanças que ocorrem durante o desenvolvimento de software. Tais mudanças podem impactar no produto que está sendo criado ou até mesmo refletir alterações no projeto do produto em construção (JACOBSON, 2002).

O manifesto ágil (BECK, 2012) foi o marco para a mudança da forma de desenvolvimento. A recomendação é para que os desenvolvedores observem, principalmente, a parte inicial (à esquerda) das quatro premissas, para a construção de um software de qualidade:

Indivíduos e interações acima de **processos e ferramentas**.

Software operacional acima de **documentação completa**.

Colaboração dos clientes acima de **negociação contratual**.
Respostas a mudanças acima de **seguir um plano**.

Nas décadas de 1980 e 1990, na visão dos engenheiros de software a melhor forma de alcançar software de boa qualidade era por meio de um planejamento atento de projeto, garantia na qualidade formalizada, aplicação de métodos de análise e projeto apoiados por ferramentas CASE sendo comandado por um rígido processo de desenvolvimento de software (BEDER, 2012).

Pressman (2006) afirma que o desenvolvimento ágil, apesar de fornecer inúmeros benefícios, não é apropriado para todos os projetos, produtos, pessoas e situações. Assim como, também não é a contradição da prática da engenharia de software consistente e pode ser empregado como uma filosofia geral para todos os trabalhos de software.

Neste capítulo serão apresentados os principais conceitos para estabelecer a fundamentação teórica, por meio de uma revisão bibliográfica de suas definições e estado da arte, visando a compreensão do trabalho aqui apresentado. Na Seção 2.2 são apresentados os conceitos sobre Métodos Ágeis. Em seguida na Seção 2.3 são apresentados os conceitos sobre *Domain-Driven Design* mencionando-se brevemente o *framework Apache Isis*. Na Seção 2.4 são tecidas as considerações finais deste capítulo.

2.2 Métodos Ágeis

Os métodos ágeis descrevem uma abordagem iterativa e incremental para a especificação, o desenvolvimento e a entrega de software. Foram concebidos sobretudo para apoiar o desenvolvimento de aplicações de negócios nas quais os requisitos de sistema mudam rapidamente durante o processo de desenvolvimento. (SOMMERVILLE, 2007).

Com o uso dos métodos ágeis os desenvolvedores, além de entregarem o software rapidamente para o cliente, possuem interações frequentes com eles, e também priorizam a comunicação entre os membros das equipes em vez de se ter atenção voltada para documentação. Métodos ágeis propõem o desenvolvimento de

software por meio de pequenas e constantes iterações para analisar, projetar, implementar, testar e também entregar o produto. Atividades propostas pelos métodos ágeis requerem comunicações orais entre os membros da equipe, de modo que eles possam aceitar mudanças nos requisitos por meio de interações diretas com os clientes. Essas são algumas das características que diferenciam esses métodos de outras abordagens de desenvolvimento. Na prática, quaisquer atividades com características semelhantes podem ser facilmente integradas aos métodos ágeis (KERAMATI; MIRIAM-HOSSEINABADI, 2008).

2.2.1 Desenvolvimento tradicional e desenvolvimento ágil de software

O desenvolvimento tradicional consiste em projetos de software baseados no modelo cascata. A equipe deve construir o sistema procedendo gradativamente nas etapas de desenvolvimento, o que admite que ela lide com níveis de dificuldades cada vez mais elevados, sendo recomendado que o sistema seja elaborado linearmente acompanhando uma sequência de fases (TELES, 2004):

1. Análise - os membros da equipe fazem o levantamento dos requisitos e procuram entendê-los detalhadamente.
2. Projeto - depois de feita uma análise a equipe planeja a arquitetura do sistema e cria artefatos que auxiliarão na fase de implementação.
3. Implementação - para implementar todas as partes do software a equipe tem como base a arquitetura e a análise discutidas nas fases anteriores.
4. Teste - a equipe testa as funcionalidades do software e executa as correções necessárias. Nesse ciclo de teste é analisado se o software atende todas as especificações e necessidades que o cliente relatou.
5. Implantação - nesta fase o sistema é colocado em produção e então os usuários finais começam a utilizá-lo.
6. Manutenção - ocorre pelo fato de o software sofrer mudanças por inúmeros motivos, desde correções até adição de novas funcionalidades.

As práticas utilizadas pelo desenvolvimento ágil de software estimulam a melhoria do projeto e da arquitetura do software de forma regular e não somente no início do desenvolvimento. Propõem que os testes devem ocorrer regularmente e

não exclusivamente no final. Nessa abordagem de desenvolvimento também é possível encontrar e eliminar as duplicações para que se permita simplificar as mudanças de projeto que possam ocorrer no futuro. A automatização de testes reduz o custo dos testes que são feitos regularmente, entretanto não elimina a necessidade de testes manuais. Durante o desenvolvimento é muito difícil prever quais mudanças poderão ser necessárias em longo prazo. Por esta razão deve-se sempre escutar o *feedback* do cliente e adaptar o software para atender às suas necessidades (KAJKO-MATTSSON *et al*., 2006).

O desenvolvimento ágil consiste na premissa de que o envolvimento do cliente minimiza possíveis desentendimentos por parte do desenvolvedor. O cliente ao utilizar o sistema, pode compreender os seus detalhes, perceber as dificuldades técnicas da implementação, enxergar novas possibilidades e, sendo assim, solicitar as mudanças para que o software se aproxime ao máximo do que ele necessita. Na prática ágil, o contato entre cliente e desenvolvedor se faz presente no decorrer de todo o processo de desenvolvimento e desempenha um papel fundamental, pois gera aprendizado para todas as partes envolvidas (TELES, 2004).

2.3 Domain-Driven Design

O desenvolvimento de software, como citado anteriormente, não é tarefa simples. Isso porque o desenvolvedor e o cliente não têm completo conhecimento do domínio do sistema a ser desenvolvido. Deve-se obter um bom modelo de domínio para controlar a complexidade do software a ser desenvolvido, modelo este que deve conter informações relevantes para a construção do domínio, de modo que proporcione aos desenvolvedores de software um aproveitamento satisfatório, o que é considerado algo difícil de fazer (EVANS, 2003).

O ponto principal da elaboração de um modelo conceitual é não desconsiderar as questões de implementação. Conceito e implementação devem sempre estar juntos porque o maior valor de um modelo de domínio se concentra no fato de que ele oferece uma linguagem onipresente (linguagem ubíqua). A Linguagem Ubíqua (Figura 2.1) proporciona a união entre os *stakeholders*:

especialistas, desenvolvedores, analistas, tecnólogos e qualquer outro membro inserido no domínio do projeto.

Acreditar que modelos são feitos primeiro e em seguida implementados é um engano. Na metodologia ágil, o pensamento de projetar e depois construir tornou-se obsoleto. Os modelos que evoluem com o tempo são os considerados verdadeiramente bons, e também os modeladores com maior experiência acreditam que conquistam melhores ideias depois do lançamento inicial do sistema (EVANS, 2003).

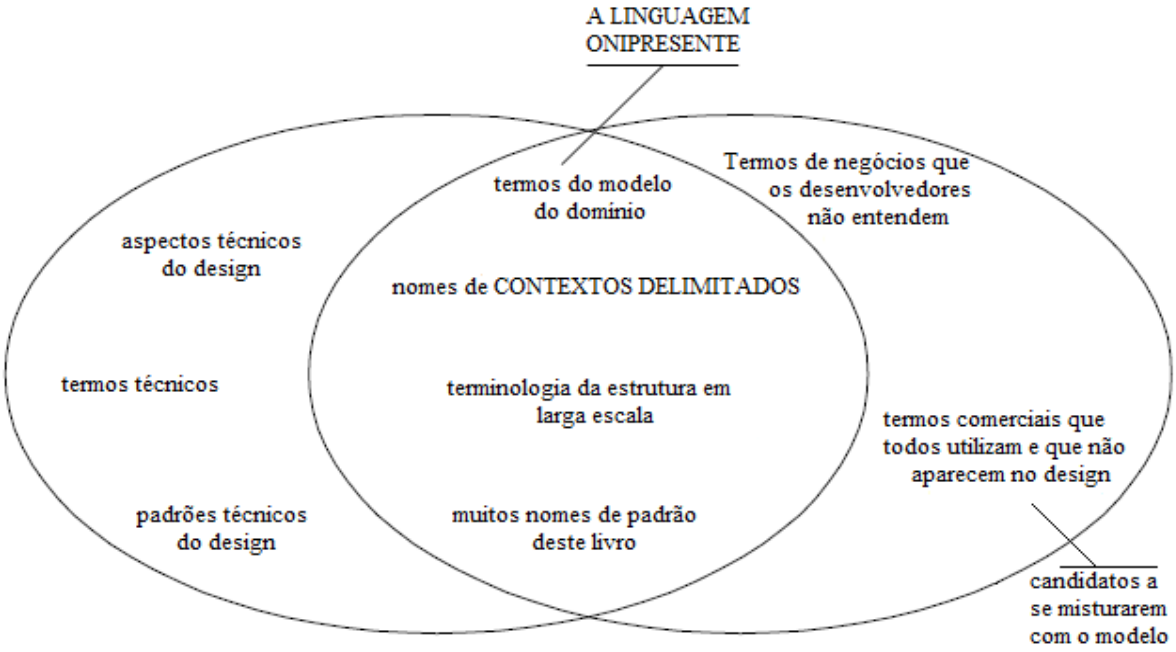


Figura 2.1: Linguagem Onipresente é cultivada na intersecção dos jargões - (Traduzida EVANS, 2003)

O DDD é uma forma de pensar em um conjunto de prioridades, ou seja, o principal enfoque deve estar na complexidade da atividade ou negócio do usuário, de modo que quando essa complexidade de domínio não é tratada no projeto, não importa se a tecnologia de infraestrutura foi bem concebida. São duas as premissas que impulsionaram Evans (2003) a criar esta técnica. São elas:

1. Na maior parte dos projetos de software o foco central deve ser o domínio e a lógica do domínio.

2. Projetos de domínios complexos devem se fundamentar em um modelo.

O DDD é empregado como uma maneira de orientar processos de desenvolvimento de modo ágil, e para isso são abordados dois princípios essenciais: o desenvolvimento iterativo, o qual é defendido e aplicado há décadas; desenvolvedores e especialistas em domínio ter uma relação próxima. A linguagem utilizada antes e durante o desenvolvimento do projeto deve ser comum para todas as partes, e concordar com a sua evolução, não permitindo que exista a degradação do *design* em relação à implementação (EVANS, 2003).

Para que o uso da linguagem ubíqua tenha efeito no desenvolvimento do software é necessário que ela reflita as necessidades de especialistas e desenvolvedores de domínio. Como tal, sua utilização proporciona a concepção do domínio do modelo que está de acordo com os padrões empregados no projeto de software, diminui a necessidade de análise e modelos de projeto, e permite que a equipe de desenvolvimento se concentre na realização das principais tarefas durante o desenvolvimento de determinada funcionalidade, mesmo quando o código é revisto após oito ou dez meses, ele deve estar claro, e a lógica de negócio não precisa de nenhuma documentação adicional (LANDRE; WESENBERG; OLMHEIM, 2007).

2.3.1 Modelo: utilidade no Domain-Driven Design

Os programas de software estão associados com alguma atividade ou área de interesse do usuário final. A área relacionada a um programa, as entidades que a constituem e as ligações existentes entre elas estabelecem o domínio do programa. Tem-se uma abstração do domínio como o modelo do domínio, elaborado com a intenção de solucionar problemas relacionados a esse domínio (SITEFANE, 2007).

Alguns domínios envolvem o mundo físico, como por exemplo: o domínio de reservas de passagens aéreas. Outros são intangíveis: uma aplicação de contabilidade, a qual envolve finanças. A carga de informações pode ser enorme. No entanto os modelos são tidos como ferramentas para reduzir essa sobrecarga, sendo uma forma de conhecimento rigidamente simplificada e conscientemente estruturada, fazendo com que as informações tenham sentido direcionando o foco para um problema. O modelo também não é um diagrama específico, e sim a ideia

que o diagrama quer transmitir, uma abstração organizada e seletiva de um determinado conhecimento (EVANS, 2003).

No contexto do DDD, Evans (2003), destaca três finalidades básicas que estabelecem a escolha de um modelo:

1. O modelo e a implementação dão forma um ao outro. Acontece uma ligação próxima entre modelo e implementação que reverte a um modelo importante, o qual garante que se aplique a um produto final. Concede auxílio durante a manutenção, pois o código é interpretado com embasamento na compreensão do modelo.

2. O modelo é o núcleo de uma linguagem utilizada por todos os indivíduos da equipe. Todos os membros da equipe podem conversar acerca do programa nessa linguagem, pois existe a ligação entre modelo e implementação. A comunicação com os especialistas de domínio acontece sem a necessidade de tradução.

3. O modelo é o conhecimento destilado. É a maneira pela qual a equipe estrutura o conhecimento do domínio e diferencia os elementos de maior interesse. A conexão entre modelo e implementação faz com que exista um *feedback* para o processo de modelagem.

A forma de um modelo deve ser uma entidade empregada durante toda a fase de desenvolvimento e não apenas um artefato usado para análise nas fases iniciais do desenvolvimento. Para relacionar de forma eficaz o modelo e a implementação, deve haver uma correspondência entre eles. Tal correspondência pode ser alcançada por meio de um paradigma de modelagem, o qual admite a criação de coerência entre conceitos existentes entre eles. Um modelo equivalente ao conjunto de conceitos, termos e ligações entre eles, caracterizados pelos componentes de uma equipe, e que refletem o conteúdo do domínio, concebem a semântica necessária para criar uma linguagem em torno do domínio. A principal forma de comunicação empregada na linguagem não deve se restringir a diagramas, como UML. Pelo contrário, deve-se difundir de várias formas, como documentos textuais, diagramas, código e até conversas casuais entre membros de uma equipe (SITEFANE, 2007).

O DDD é classificado como a consolidação da orientação a objetos, pois é um chamado às boas práticas de programação que existem desde a época do SmallTalk

(SMALLTALK, 1999). Ao mencionar a orientação a objetos remete-se a pensar em classes, herança, polimorfismo e encapsulamento. Porém a natureza da orientação a objetos também inclui (CUKIER, 2010):

1. Alinhamento do código com o negócio: A comunicação dos desenvolvedores com os especialistas do domínio é fundamental quando se pratica DDD (conhecido em métodos ágeis).
2. Isolamento entre domínios: A ideia de DDD é criar isolamento entre os domínios, a saber, qual é o contexto de cada sistema, qual regra pertence a cada domínio.
3. Favorecer reutilização: Por meio de blocos de construção que favorecem a reutilização de um mesmo conceito de domínio ou mesmo código em vários lugares, pois tem-se um sistema com funções bem definidas.
4. Mínimo de acoplamento: Desde que se tenha um modelo bem elaborado, organizado, todas as partes de uma aplicação comunicam-se sem que exista muita dependência entre módulos ou classes de objetos de conceitos distintos.
5. Independente de tecnologia: O foco é em torno de compreender as regras de negócio e de qual a maneira elas devem estar reproduzidas no código e no modelo de domínio. Enfatiza-se que não se deve considerar que a tecnologia não seja importante, porém essa não é uma preocupação do DDD.

Ao utilizar a linguagem ubíqua cria-se um modelo de domínio por meio do *Model Driven Design*[1] (MDD). O MDD desconsidera a divisão entre o modelo da análise e do *design* e busca por um modelo único que atenda as duas finalidades. Questões puramente técnicas são descartadas, cada objeto do *design* exerce um papel conceitual descrito no modelo, parte do código é uma expressão do modelo, quando o código é alterado, o modelo também é alterado. Assim é preciso que exista maior exigência em relação ao modelo escolhido, visto que ele desempenha dois objetivos bastante distintos (EVANS, 2003).

[1] O Model Driven Design é diferente do Model Driven Development, o qual tem como objetivo reconhecer a importância dos modelos no processo de software, não sendo somente um "guia" para tarefas de desenvolvimento e manutenção, mas como parte integrante do software (LUCRÉDIO, 2009).

Em um processo ágil empregado pelo MDD a elaboração do modelo abstrato deve ser feita em grupo. É possível obter a criação de um modelo que não é implementável ou que empregará uma tecnologia inadequada, caso arquitetos e analistas de negócio criem o modelo sem a presença dos programadores. Assim como se não houver base de um modelo consistente durante a codificação pelos programadores, certamente irá existir um software que não serve para o domínio. O processo de maturidade ao desenvolver uma aplicação empregando-se MDD é contínuo. O modelo serve como um direcionamento para a elaboração do código, e ao mesmo tempo, o código auxilia a aperfeiçoar o modelo (CUKIER, 2010).

2.3.2 Isolando o domínio

Para preservar a codificação do software combinado com o modelo de domínio é preciso aplicar as melhores práticas de modelagem. Quando estas práticas são empregadas torna-se fácil alcançar um modelo no qual os conceitos do domínio estejam definidos de forma natural e intuitiva (SITEFANE, 2007).

Ao criar um modelo empregando o MDD, primeiramente é necessário isolar o modelo de domínio das outras partes que compõem a aplicação. Tal separação pode ser feita pela arquitetura em camadas, como pode ser visto na Figura 2.2. A importância das camadas é que cada uma possui uma determinada tarefa em um programa de computador, permitindo dessa forma projetos mais coesos. Mesmo havendo variações, a maior parte das arquiteturas contém essas quatro camadas (EVANS, 2003):

1. Interface de usuário (ou camada de apresentação): Responsável por exibir informações ao usuário e por interpretar comandos do usuário, são os elementos visuais, ou seja, elementos de contato com o usuário final.

2. Camada de aplicação: As funções que estão sob responsabilidade desta camada possuem grande importância para o negócio. Mesmo não possuindo lógica de negócio, é necessária para a comunicação com camadas de aplicação de demais sistemas. É também responsável por conectar a Interface de usuário às camadas que estão abaixo dela.

3. Camada do domínio (ou do modelo): Representa conceitos de negócio. Particularidades técnicas de armazenamento são transferidas para a infraestrutura. Deve ficar o mais isolado possível de todas as outras camadas.

4. Camada da infraestrutura: Os recursos técnicos que serão concedidos para prover suporte às camadas superiores são fornecidos por esta camada. São partes de uma aplicação responsável por conexão com banco de dados, gravação e leitura de dados, persistência de dados, envio de mensagens por redes e outras funções.

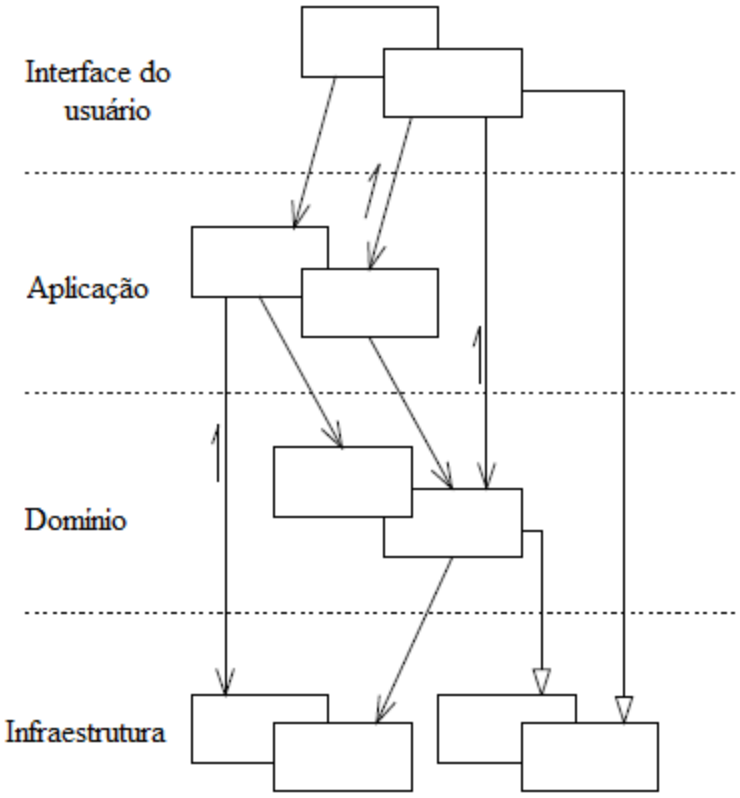


Figura 2.2: Arquitetura em camadas - (Traduzida EVANS, 2003)

Após separar o sistema em camadas, o foco é apenas na camada de domínio. Utiliza-se padrões em DDD representados na Figura 2.3 para criar a modelagem da camada de domínio. Tais padrões são conhecidos por blocos de construção (EVANS, 2003). Sendo eles:

1. Entidades: Um objeto caracterizado principalmente por meio de sua identidade é denominado entidade. Entidades contêm considerações especiais de

modelagem e de *design*. Devem ser mantidas em uma linha de continuidade, apesar de seu ciclo de vida poder ser alterado tanto em sua forma como conteúdo. Em outras palavras, é tudo na aplicação que é importante no domínio e possua uma identidade. Para exemplificar: um Cliente realiza o cadastro no sistema, efetua compras, torna-se inativo, é excluído, etc.

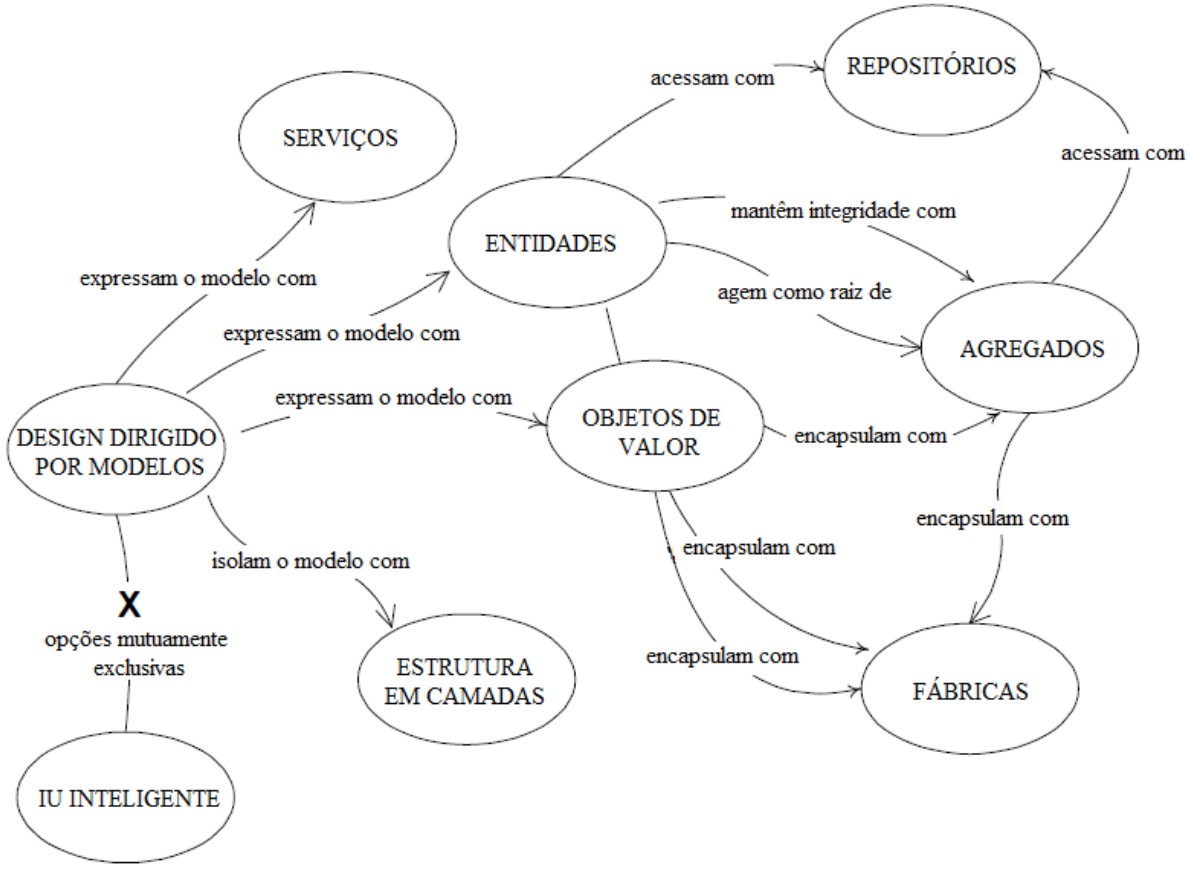


Figura 2.3: Mapa de navegação da linguagem do Design Dirigido por Modelos - (Traduzida EVANS, 2003)

2. Objetos de Valor: É um aspecto descritivo do domínio, não possui nenhuma identidade conceitual ou descreve característica de alguma coisa, apenas carregam valores, porém não possuem distinção de identidade. São instanciados para representar partes do *design*. As instâncias dos Objetos de Valor são imutáveis. A partir do momento de sua criação, seus atributos não podem ser alterados. Também não precisa ser persistido em um banco de dados. É usado localmente em um código e a sua utilização acaba sendo mais rápida. Como exemplo pode-se dizer que somar dois valores e inserir o seu resultado em outra variável, valor um e dois, não devem ser modificados durante a operação.

3. Serviços: Operações fornecidas como interface que se encontram isoladas do modelo. As classes possuem lógica de negócio que contém operações específicas, porém não pertencem a nenhuma Entidade ou Objetos de Valor. Destaca-se o fato de que não guardam estados. Em outras palavras, toda chamada a um serviço, fornecida uma mesma pré-condição, deve retornar sempre o mesmo resultado.

4. Módulos: Os módulos também são entendidos como pacotes, são componentes de *design* antigos e estabelecidos. Propiciam às pessoas terem duas visões do modelo: eles podem verificar as particularidades do modelo sem que sejam ofuscados pelo todo, ou então examinar as ligações entre os módulos em visões que desprezam particularidades internas. Eles devem aparecer como sendo uma fração expressiva do modelo na camada de domínio, pela qual descrevem a história do domínio em uma escala maior. A maior parte das linguagens de programação fornece suporte a módulos, por exemplo, pacotes em Java, namespaces em .NET ou módulos em Ruby. Como um anti-padrão pode-se citar a elaboração de módulos, os quais reúnem as classes de acordo com conceitos de infraestrutura. Os Módulos devem progredir de acordo com o resto do modelo, ou seja, é necessário refatorar os Módulos juntamente com o modelo e o código.

5. Agregados: Os Agregados delimitam o próprio modelo determinando uma propriedade e limites claros, impedindo um emaranhado caótico de objetos. Este padrão é essencial para preservar a integridade em todas as fases do ciclo de vida. Ele é tido como um conjunto de objetos associados que são considerados como uma unidade a fim de efetuar alterações de dados. Cada Agregado contém uma raiz e um limite. O que está no interior do Agregado é o limite. A raiz trata-se de uma Entidade única e específica contida no Agregado. A raiz é o único elemento pelo qual objetos externos são autorizados a fazer referências, entretanto os objetos que pertencem àquele limite podem efetuar referências uns aos outros.

6. Fábricas: Da mesma forma que a interface de um objeto deve encapsular sua implementação, deixando que o cliente use o comportamento do objeto sem ter conhecimento da maneira que ele funciona, uma Fábrica encapsula o conhecimento para construir um objeto complexo ou um Agregado. Gera uma interface que reflete as intenções do cliente e uma visão abstrata do objeto construído. A Fábrica quando utilizada de forma correta auxilia a manter o MDD em seu curso, e, a saber, a Fábrica não faz parte do modelo, porém ela pertence ao domínio.

7. Repositórios: Trata-se de classes responsáveis pelo gerenciamento do ciclo de vida dos objetos, sendo Entidades, Objetos de Valor e Agregados. Eles concentram operações de criação, alteração e remoção dos objetos. Objetos são requisitados pelos clientes por meio dos Repositórios empregando-se métodos de consulta que selecionam objetos de acordo com critérios descritos pelo cliente, sendo na maioria das vezes o valor de certos atributos.

2.3.3 Design flexível: refatorar para compreender o modelo

Design flexível é um complemento e torna claro o potencial da modelagem a partir do momento em que se retiram os conceitos implícitos tornando-os explícitos, como resultado é gerada a matéria-prima. Por meio de um ciclo comunicativo é feita a modelagem deste material fornecendo-lhe uma forma útil, elaborando um modelo que domine de maneira simples as principais questões, permitindo que o desenvolvedor modele um projeto e coloque esse modelo em ação. A medida em que evolui, o modelo necessita ser refatorado de acordo com a compreensão e conhecimento do domínio. Padrões que podem ajudar a compreender o modelo são descritos abaixo (EVANS, 2003):

1. Interface reveladora de intenção: É um dos padrões, no qual os nomes em métodos ou classes são usados para expressar o que eles executam, porém não como executam. Para isso existe uma interface, sendo que somente nome e os parâmetros dessas classes ou métodos são necessários.

2. Funções isentas de efeitos colaterais: São códigos com métodos que não modificam o estado dos objetos. A lógica complexa de mudança de estado deve ser deslocada para o Objeto de Valor quando acontecer um conceito adequado à responsabilidade.

3. Asserções: Declaram-se pós-condições das operações e as invariantes das classes e dos Agregados. Caso as Asserções não possam ser codificadas de modo direto na sua linguagem de programação, é preciso escrever testes de unidade automatizados para elas. Estes testes escritos devem ser mantidos na documentação do projeto.

2.3.4 O Núcleo do Domínio

A principal parte no DDD é o método chamado de Destilação do Domínio. Chega-se à plenitude ao extrair o que havia de mais importante no Núcleo do Domínio (*Core Domain*). No começo tem-se blocos que contêm código de diversas camadas e classes com funcionalidades divergentes. É necessário centralizar esforços para desagregar módulos, refatorando-se, extraindo-se métodos, classes e conceitos, até que se alcance os conceitos centrais do negócio da aplicação, sendo este o Núcleo do Domínio. Conceitos que não estão inseridos no núcleo devem ser retirados e divididos em subdomínios genéricos, pois mesmo sendo componentes que constituem o domínio, estes não são centrais (CUKIER, 2010).

De acordo com EVANS (2003), para entender melhor o que é o Núcleo do Domínio escreve-se um documento pequeno, nomeado de Declaração da Visão do Domínio, este documento descreve o domínio principal em termos amplos. De forma que, apesar de serem essenciais, informações como: tecnologias que serão utilizadas, como será oferecido o acesso ao software ou até mesmo como a interface será implementada para usuários experientes e acessível a usuários inexperientes, não são detalhadas nesse documento.

Caso não seja satisfatório documentar o Núcleo dessa forma, outra possível solução pode ser destacar os elementos que estão inseridos no domínio, para isso emprega-se o padrão Núcleo Destacado, no qual é elaborado um documento mais amplo, que contemple, porém, não ultrapasse sete páginas, declarando todos os componentes do núcleo e a maneira como tais componentes interagem (CUKIER, 2010).

Uma equipe relatou ter empregado o DDD e com isso ampliou sua arquitetura empresarial com técnicas de nível estratégico. Como a simplicidade de exploração do domínio por meio do uso da linguagem ubíqua, na qual foi discutido o domínio, com o uso de termos e conceitos nitidamente definidos, admitiu que os diferentes membros do projeto compreenderam e contribuíram para a análise do domínio (WESENBERG; LANDRE; RONNRBERG, 2006).

Com este levantamento acerca da técnica empregada pelo DDD é importante saber exatamente o que faz parte do domínio, e extrair o que contém maior relevância para o sistema, em conjunto com o uso de seus padrões, que estimulam

uma comunicação eficiente, sendo este um valor considerado importante pelos adeptos das metodologias ágeis.

2.3.5 Apache Isis: framework de implementação Domain-Driven Design

O *Apache Isis* é um *framework* desenvolvido e mantido pela *Apache Software Foundation* sob a *Apache License* versão 2.0 (APACHE, 1999), ou seja, é permitido o uso e a distribuição de seu código fonte tanto para um software *open source* como para um software proprietário.

O objetivo do *framework Apache Isis* (APACHE ISIS, 2010) é auxiliar e principalmente agilizar o desenvolvimento de aplicações Java guiadas pelo domínio, ou seja, usando a abordagem DDD. Pode ser utilizado para prototipação de novas aplicações ou até mesmo na criação de software para produção. O desenvolvedor de um sistema que utiliza este *framework* pode focar seus esforços na criação de seu domínio, com suas entidades, serviços e regras de negócios, pois o *Apache Isis* será capaz de gerar dinamicamente a representação desse domínio como uma aplicação web ou uma *API RESTful*.

Ao optar-se pela geração de uma aplicação web, o *Apache Isis* integra outros *frameworks* e padrões de desenvolvimento para determinados módulos do sistema. Para gerenciar a autenticação e autorização de acesso, é utilizado o *Apache Shiro* (APACHE SHIRO, 2008), um *framework* da *Apache Software Foundation* especializado em soluções de segurança. Para garantir a qualidade da aplicação, o *Apache Isis* suporta a integração de *frameworks* de testes unitários como o JUnit (JUNIT, 2014), e de testes de integração. A interface, gerada dinamicamente, utiliza o *Apache Wicket* (APACHE WICKET, 2014), que permite renderizar os objetos do domínio em calendários, gráficos ou até mesmo em mapas.

Caso seja feita a escolha por gerar uma aplicação *RESTful* (RESTFUL OBJECTS, 2013), todas as funcionalidades do domínio criado serão expostas de acordo com as especificações desse padrão. Dessa forma, outras aplicações podem ter acesso aos dados e às operações do domínio, inclusive aplicações escritas em diferentes linguagens. É possível criar uma camada de visão própria e a aplicação *RESTful* atuaria como o *back-end* do sistema. Isso torna a aplicação *RESTful* muito dinâmica e amplamente utilizada.

Na Figura 2.4 é possível ver um desenho da arquitetura do *framework Apache Isis*, com os vários módulos que o compõe. Para cada módulo, o *Apache Isis* oferece opções de utilização, que são facilmente configuráveis. Como citado anteriormente, a forma como o domínio vai ser exposto pode ser feita por meio do *Wicket*, *RESTful* e também HTML (*Hyper Text Markup Language*). A persistência pode ser feita por arquivos XML (*Extensible Markup Language*), usando um banco de dados relacional ou até mesmo um não-relacional.

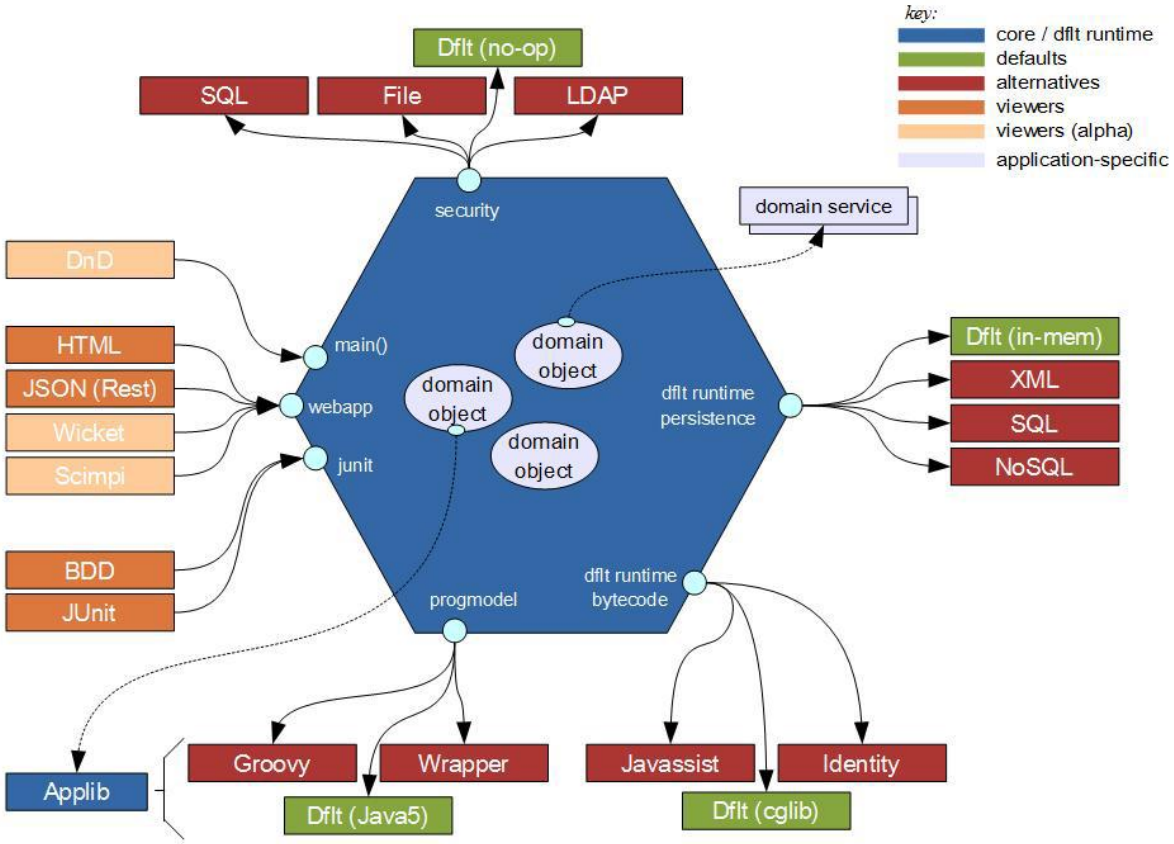


Figura 2.4: Arquitetura do framework Apache Isis (APACHE ISIS, 2010)

Além dessas formas de apresentação do domínio, que são geradas dinamicamente, não é necessária a criação de uma camada de controladores, comumente encontrada em uma arquitetura MVC. Essas são as características que fazem o *framework Apache Isis* ser muito utilizado entre os desenvolvedores e também o difere de tantos outros *frameworks* para desenvolvimento rápido de aplicações (HAYWOOD, 2013).

Outras ferramentas que dão suporte ao desenvolvimento com a abordagem DDD são: Actifsource (ACTIFSOURCE, 2013), um plugin para a IDE Eclipse, que

permite a criação de aplicações combinando DDD com geração de código; OpenXava (OPENXAVA, 2005), um *framework* que possibilita a criação de aplicações Ajax a partir do modelo de domínio; CubicWeb (CUBICWEB, 2008), um *framework* web inteiramente guiado pelo domínio que também gera uma aplicação web funcional.

2.4 Considerações Finais

Neste capítulo, foram apresentados os conceitos sobre Métodos Ágeis que são utilizados com o intuito de apoiar o desenvolvimento de aplicações em que as alterações de requisitos solicitadas pelos clientes acontecem rapidamente. Também foi visto que seu uso prioriza a comunicação entre os integrantes das equipes. Foi feito um levantamento sobre o desenvolvimento tradicional e acerca do desenvolvimento ágil, a fim de analisar as diferenças entre essas duas metodologias.

Em especial, maior atenção foi destinada à abordagem de desenvolvimento DDD, que é o principal enfoque deste trabalho, a qual destaca a prioridade de se conhecer bem as regras do domínio antes de iniciar o desenvolvimento. Também observou-se a importância da linguagem ubíqua e do MDD. Além disso, foi apresentado o *framework Apache Isis*, o qual utiliza a abordagem DDD para desenvolvimento de aplicações web.

Após o levantamento dos conceitos do DDD, notou-se a ausência de uma abordagem específica acerca de testes, uma vez que o DDD apenas menciona a necessidade de testes mas não elucida como esses testes devem ser feitos.

Diante deste cenário, no próximo capítulo serão apresentados os fundamentos dos conceitos sobre teste de software necessários ao entendimento do trabalho aqui apresentado.

Capítulo 3

TESTE DE SOFTWARE

Este capítulo apresenta conceitos sobre Teste de Software, assim como a importância da estratégia do teste de software para o ciclo de desenvolvimento. São apresentados conceitos acerca do Teste de Unidade, de Integração e o Automatizado. As técnicas de teste TDD e BDD também são apresentadas, evidenciando as vantagens e desvantagens de cada uma delas e o ciclo de teste.

3.1 Considerações Iniciais

O teste de software é definido como a execução de um sistema com o propósito de identificar falhas. Sendo também um método de controle de qualidade que verifica o comportamento do sistema em conformidade a um conjunto de requisitos que são expressos como testes. Um conjunto de atividades é planejado com antecedência e executado constantemente durante o ciclo de teste (PRESSMAN, 2006). Para que o desenvolvimento de software e a atividade de teste sejam bem sucedidos é necessário que exista um bom planejamento e qualidade das métricas empregadas. O processo de teste, assim como o processo de desenvolver software requer fases, procedimentos e passos bem elaborados (BURNSTEIN, 2003).

Neste Capítulo são apresentados os conceitos relacionados à atividade de teste de software para estabelecer a fundamentação teórica visando auxiliar o entendimento do trabalho aqui apresentado. Na Seção 3.2 serão apresentados os conceitos sobre estratégia de teste de software, sendo de total importância para o ciclo de desenvolvimento de software, identificando e colaborando para a redução

de falhas e contribuindo para o aumento da qualidade dos produtos finais de software, assim como as fases necessárias para a execução do processo de teste. Na Seção 3.3 são apresentados os conceitos relacionados ao teste de unidade. Em seguida, na Seção 3.4 são apresentados os conceitos sobre teste de integração. Na Seção 3.5 são apresentados os conceitos relacionados ao teste automatizado, comparando-o brevemente com o teste manual. Na Seção 3.6 são apresentados os conceitos fundamentais da técnica *Test-Driven Development*. Na Seção 3.7 são apresentados os conceitos sobre *Behaviour-Driven Development*. Por fim, na Seção 3.8 são apresentadas as considerações finais deste capítulo.

3.2 Fundamentação Teórica

Uma estratégia de teste bem definida é fundamental para comprovar se o software se comporta como o esperado. A partir da estratégia de teste, é definida a abordagem que será utilizada e os objetivos da atividade de teste (MYERS *et al.*, 2004).

As estratégias de teste de software possuem características comuns e fornecem um modelo para o teste de software (PRESSMAN, 2006):

1. Para que o teste seja executado de forma eficaz é necessário inserir revisões técnicas. Estas, por sua vez, têm a função de encontrar defeitos precocemente. Dessa forma diversos defeitos poderão ser extintos antes do início do teste.

2. O início do teste é feito em nível de componente e avança para a integração do sistema.

3. As técnicas de teste de software devem ser aplicadas de acordo com as abordagens que serão utilizadas, ou seja, para cada abordagem de engenharia de software existe uma técnica adequada.

4. Para projetos grandes e complexos os testes podem requerer um grupo que trabalhe somente na elaboração de testes. Caso contrário eles são feitos e executados pelos próprios desenvolvedores.

5. Existe distinção entre o teste e a depuração, entretanto a depuração deve estar associada com uma estratégia de teste.

Após a definição da estratégia de teste adotada, algumas fases são necessárias para a execução do processo de teste. Elas são definidas como (MYERS *et al.*, 2004):

1. Planejamento de teste: Esta fase é fundamental para o processo de teste, pois é nesse momento que é definido como será executado todo o processo, desde a forma em que os testes serão aplicados, a finalidade, métodos e ferramentas empregadas para a realização dos testes.

2. Projeto de casos de teste: Tem como entrada o plano de teste, algumas definições são de suma importância como análise de risco, qual funcionalidade será testada e qual a abordagem utilizada, a fim de atingir o nível de satisfação dos critérios estabelecidos.

3. Execução do teste: Nessa fase pode ou não ser empregada alguma ferramenta para executar o teste, independentemente é analisado se o resultados do teste condiz com o esperado. Se ocorrer alguma falha esta necessita ser documentada.

4. Análise do resultado do teste: Visa garantir que a finalidade do teste foi alcançada e este deve ser apresentado em forma de relatório.

A fase considerada mais importante é o projeto de casos de teste. Testar a aplicação por completo pode tornar-se impraticável por ser custoso computacionalmente e financeiramente. Por isso sugere-se a elaboração dos casos de teste como forma de descobrir a maioria das falhas com pequeno esforço e tempo (MYERS *et al.*, 2004).

O teste de software nunca chega ao fim, na realidade ele desloca-se do engenheiro de software para o usuário final, ou seja, conforme o usuário faz uso de uma aplicação (PRESSMAN, 2006).

3.2.1 Técnicas de Teste de Software: Funcional e Estrutural

Atualmente existem diversas formas para se testar um software. Técnicas que sempre foram muito utilizadas em sistemas desenvolvidos em linguagens estruturadas ainda hoje têm grande valia para sistemas orientados a objetos. Mesmo se tratando de paradigmas de desenvolvimento diferentes, o objetivo destas técnicas levam em consideração o mesmo princípio: encontrar falhas no software para que sejam corrigidas. As técnicas de teste fornecem diretrizes sistemáticas para projetar testes que exercitam a lógica interna, as interfaces de cada componente, e os domínios de entrada e saída do programa a fim de encontrar defeitos na função, no desempenho e comportamento do programa (PRESSMAN, 2006).

Existem duas principais técnicas de teste: funcional e estrutural. Cada uma dessas técnicas tem critérios específicos que auxiliam na seleção e avaliação dos casos de teste, e o que difere cada uma é a fonte utilizada para estabelecer os requisitos de teste (FABBRI *et al*., 2007).

A técnica funcional, também conhecida como teste caixa-preta, é uma técnica de teste em que o componente de software, preocupa-se com entradas e saídas do programa, sem se importar com detalhes internos de implementação (FABBRI *et al*., 2007). O componente de software que irá ser testado pode ser um método, uma função interna, um programa, um conjunto de programas e componentes ou até mesmo uma funcionalidade. Essa técnica de teste pode ser utilizada em todas as fases de teste (PRESMAN, 2006).

Para a derivação de casos de teste, a técnica estrutural, conhecida também como caixa-branca, avalia o comportamento interno do componente de software. Tais casos de teste podem avaliar: teste de condição, teste de fluxo de dados, teste de ciclos e teste de caminhos lógicos (PRESMAN, 2006). Uma representação do programa para o teste estrutural é o grafo de fluxo de controle, o qual representa a estrutura interna de módulos do programa em termos de seus comandos sequenciais e condicionais (MYERS *et al*., 2004).

3.3 Teste Automatizado

Testes automatizados são programas ou *scripts* simples, os quais fazem verificações de forma automática nas funcionalidades do sistema que está sendo testado. Um dos principais benefícios desta abordagem é que os casos de teste que demoram muito tempo para serem executados manualmente podem ser rapidamente e facilmente refeitos utilizando-se pouco esforço (BERNARDO; KON, 2008). Diferentemente dos testes manuais, essa abordagem não somente automatiza a execução dos casos de teste, como efetua a concepção e a verificação do resultado do teste (LEITNER *et al.*, 2007).

Para Pressman (2006), o uso de algumas ferramentas tende a diminuir o tempo despendido para o teste, por isso são muito valiosas. É pertinente aplicar-se uma ferramenta de teste a partir do momento em que surgirem fortes pressões para garantir a qualidade, quando houver situações no projeto que não podem ser testadas apropriadamente pelos métodos tradicionais (BASTOS *et al*, 2012). Aplicações de critérios de teste que não contém ferramentas automatizadas podem acarretar em atividades propensas a falhas e restringidas a programas simples (DELAMARO; MALDONADO; JINO, 2007).

Conforme estes testes são executados é possível refazer diversas vezes determinadas situações, assegurando-se que alguns passos essenciais sejam executados e que não serão esquecidos por algum engano humano. Também são fundamentais para apontar qualquer comportamento que não deveria acontecer (BERNARDO; KON, 2008).

Esses casos de verificação são descritos por meio de um código interpretado de computador, proporcionando a elaboração de testes mais complexos em vista dos testes criados manualmente, permitindo diversas combinações de comandos e operações. Outro ponto que pode ser analisado como sendo uma vantagem de teste automatizado é que a magnitude dos testes também pode ser modificada, por exemplo, de maneira simples pode-se simular uma grande quantidade de usuários acessando um sistema ou efetuar milhares de registros em uma base de dados, verificando se o sistema é apto a processar grande quantidade de dados, conforme as especificações de tempo e processamento, determinando dessa forma a escalabilidade do sistema (BERNARDO; KON, 2008).

Por possuírem conhecimento no domínio os desenvolvedores são mais capacitados para elaborar a criação de dados de entrada complexos e desenvolver casos de testes interessantes (neste caso 'interessantes' seriam casos de testes propícios a detectar falhas). Porém, eventualmente o desenvolvedor também não está imune a interpretar de forma errônea o domínio real de uma aplicação (LEITNER *et al.*, 2007).

Comparado com o teste manual, o teste automatizado reduz a quantidade de defeitos e eleva a qualidade do software. Na maior parte dos casos, os testes automatizados são escritos programaticamente, logo se torna necessário obter conhecimento de programação (BERNARDO; KON, 2008).

Um conjunto robusto de ferramentas *open source* que fundamenta o desenvolvimento ágil de automação de teste é o Selenium (SELENIUM HQ, 2012), o qual propicia um vasto conjunto de métodos de testes de aplicações com base na Web. Tais operações são bem flexíveis, assentindo-se que diversas operações sejam elaboradas para a localização de elementos na interface do usuário e permite que os resultados que são esperados para o teste sejam comparados ao comportamento atingido. Além disso, o Selenium possui um recurso de gravação (*record*), que registra as ações do usuário exatamente da forma como são executadas. Em seguida, é permitido exportar todas as ações executadas como um *script* que podem ser reexecutadas (*playback*) posteriormente em uma das muitas linguagens de programação.

3.4 Teste de Unidade

O teste de unidade é a fase de teste na qual cada unidade do sistema é testada individualmente. Diversos métodos e recursos com o objetivo de auxiliar o desenvolvedor na elaboração de testes de unidade têm sido propostos. Esse tipo de teste possui o código escrito, e assim como qualquer outra parte do código, ele precisa estar coerente com os requisitos. O código tem que ser legível e compreensível para possíveis manutenções e refatorações de atividades (RAMLER; KASPAR, 2012). Contudo a qualidade dos testes de unidade está relacionada com a qualidade do engenheiro responsável pela sua escrita (QUSEF *et al.*, 2011).

O foco da utilização do teste de unidade é manter esforços na menor unidade de uma aplicação (componente, módulo ou classe). A dificuldade relativa desse tipo de teste e defeitos que são por ele encontrados limita-se pelo escopo restrito organizado para o teste de unidade, também pode ser conduzido simultaneamente com outros componentes (PRESSMAN, 2006).

Casos de teste são elaborados para a implementação dos testes, sendo que cada teste de unidade deve conter quatro elementos fundamentais. São eles (SHRIVASTAVA; JAIN, 2011):

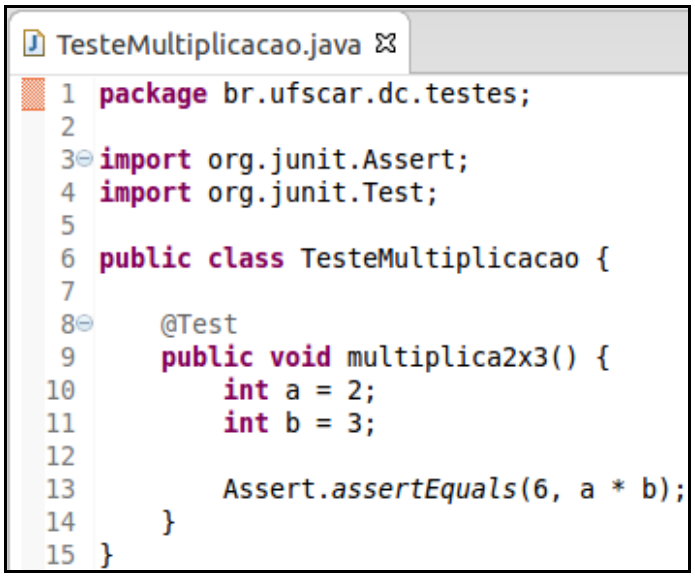
1. Uma declaração para o teste de unidade.
2. A entrada para o teste de unidade, em que são incluídos quaisquer valores de dados externos.
3. O que efetivamente o caso de teste irá testar, no que se diz respeito às funcionalidades de uma unidade e a análise empregada no *design* do caso de teste.
4. E por fim, o resultado previsto para o caso de teste.

Se apenas alguns membros apoiarem e observarem os benefícios que o teste de unidade pode favorecer em termos de qualidade e redução de falhas, provavelmente os esforços desses membros não irão ser suficientes. O tempo despendido para os testes de unidade precisa ser definido no cronograma de desenvolvimento. Toda a equipe deve levar em consideração que o desenvolvimento pode acarretar um tempo maior, porém a qualidade final do produto será melhor. O teste de unidade precisa progredir juntamente com outras partes do código e a sua qualidade precisa ser a mesma que a qualidade do código do produto. A capacidade do sistema ser testado deve ser considerada como parte da arquitetura e *design* para entender como isso impacta nos testes, no custo dos testes, na efetividade e no processo de teste. Esse tipo de teste deve ser medido de alguma forma, porém não pela sua quantidade e sim pela sua qualidade, de forma que tais medidas necessitam ser transmitida a toda equipe de forma clara (WILLIAMS; KUDRJAVETS; NAGAPPAN, 2009).

O JUnit (JUNIT, 2014) é um *framework* para testes de unidade caixa-preta para a linguagem de programação Java (JAVA, 1995), e consiste em uma forma simples para a escrita de testes repetíveis. Ele é utilizado em conjunto com o desenvolvimento de software que emprega o TDD. A vasta comunidade de usuários

que utiliza o JUnit acredita que este *framework* é a base de muitas ideias novas para o desenvolvimento de tecnologias para testes de unidade.

Um simples exemplo de teste de unidade utilizando o JUnit é mostrado abaixo na Figura 3.1. Para testar a multiplicação entre dois números pode-se pensar no seguinte caso de teste: Dado o número inteiro 2 e o número inteiro 3, a multiplicação entre eles deve resultar o número inteiro 6.


TesteMultiplicacao.java

```
1  package br.ufscar.dc.testes;
2
3  import org.junit.Assert;
4  import org.junit.Test;
5
6  public class TesteMultiplicacao {
7
8      @Test
9      public void multiplica2x3() {
10         int a = 2;
11         int b = 3;
12
13         Assert.assertEquals(6, a * b);
14     }
15 }
```

Figura 3.1: Exemplo de teste de unidade com JUnit

Para criar um teste de unidade com a ferramenta JUnit é necessário que sua biblioteca esteja importada no projeto. Então, basta escrever um método com a anotação `@Test` que ele será reconhecido como um teste de unidade.

3.5 Teste de Integração

O teste de integração é feito após a fase do teste de unidade com destaque para a construção da estrutura do sistema. Conforme as partes do sistema necessitam ser posicionadas para trabalharem em conjunto é preciso que exista uma interação entre essas partes de forma pertinente. E para isso deve haver um entendimento das estruturas internas e da relação existente entre as partes do sistema. É por este motivo que o teste de integração na maioria das vezes é

efetuado pela própria equipe de desenvolvimento (DELAMARO; MALDONADO; JINO, 2007).

Para que se tenha um bom preparo no planejamento dos testes de integração é fundamental atender alguns pontos essenciais, como por exemplo:

1. O responsável pela execução dos testes: Podem ser executados tanto por uma equipe específica de testes ou por projetistas responsáveis pela integração. A escolha de quem executará os testes está associada à complexidade do sistema, caso o sistema seja menos complexo poderá ser testado pelos próprios projetistas, porém se o sistema contiver um grau elevado de dificuldade, uma equipe de teste para testar a aplicação é fundamental.

2. Quais partes são testadas: Interações entre componentes, geradas por chamadas a métodos de interface ou passagem de parâmetros. Preferencialmente, são selecionadas as interações que expressam as funcionalidades principais da aplicação.

3. Momento de execução dos testes: Ocorre conforme a montagem do sistema. Consequentemente a sequência dos testes pode induzir a sequência de integração do sistema.

4. A forma de realização dos testes: Deve-se averiguar a integração de um componente de cada vez com o intuito de simplificar a localização de defeitos, e então é importante que uma ordem de integração seja determinada.

5. Quanto se deve testar: Não existe um valor pré-determinado e sim uma relação com a cobertura dos cenários da aplicação e a quantidade de interações contidas na arquitetura de componentes.

A maneira de interagir e elaborar a montagem dos componentes pode induzir na forma em que os testes serão realizados, por esta razão é de suma importância considerar o processo de teste combinado com o processo de desenvolvimento (GOUVEIA, 2004).

3.6 Test-Driven Development

Test-Driven Development faz parte dos princípios ágeis de desenvolvimento. Tal técnica foi criada por Kent Beck que impulsionou a ideia de escrever testes automatizados antes da implementação do código (BECK, 2012).

TDD é um processo de desenvolvimento iterativo, no qual cada iteração é feita com a escrita de um teste que faz parte da especificação do que será implementado. Todas as iterações são caracterizadas por não serem longas e propiciarem um rápido *feedback* sobre o código em desenvolvimento. O início do desenvolvimento com TDD é feito definindo-se uma meta, ao contrário de definir o código que será implementado para resolver determinado problema (JOHANSEN, 2011).

Esta técnica ágil é composta por três fases, que são: vermelha, verde e refatoração. Na fase vermelha um caso de teste deve ser criado para falhar e algumas vezes ele pode não compilar (BECK, 2012). Deve-se pensar em sua funcionalidade e escrever o teste como se o código a ser testado já estivesse implementado, porém não será possível compilar, pois o método ainda não foi codificado, então é necessário escrever um trecho de código para que o teste seja executado. E então ele deverá falhar. Este passo é feito para assegurar que o teste faz referência ao trecho de código correto, e que ele não é executado com sucesso propositalmente (PALERMO, 2006).

Fornecidos alguns requisitos que serão implementados no sistema, os desenvolvedores irão desenvolver código por meio de iterações rápidas. Este ciclo altamente iterativo é utilizado pelos analistas para desenvolver uma parte de uma nova funcionalidade. Primeiramente é necessário que todos os casos de testes sejam executados com sucesso para que depois o novo código seja incluído no código base, deve-se garantir que o novo código não tenha falhas que serão replicadas no código base, ou até mesmo oculte alguma falha do código base (WILLIAMS; MAXIMILIEN, 2003).

Na fase verde é conveniente que o teste seja executado com sucesso, independente das técnicas de programação utilizadas (BECK, 2012). Na fase de refatoração, são removidas todas as duplicações de código existentes em sua aplicação, porém é preciso garantir que todos os testes continuem funcionando.

Essa exclusão de código duplicado é feita, pois foram adicionadas novas funcionalidades, deve-se fazer modificações de *design* para aperfeiçoar a solução global. Depois de efetuada cada refatoração, todos os testes são validados com o intuito de verificar que eles continuem funcionando. Todo o ciclo do TDD deve ser ágil de forma que durante uma hora ele possa ser repetido diversas vezes, como visto na Figura 3.2 (PALERMO, 2006).

De maneira abrangente o ciclo do TDD pode ser descrito como segue (LAPOLLI *et al*., 2010; BECK, 2012 *apud* USSAMI, 2013):

1. Escrita de teste(s) de unidade automatizado visando qual o comportamento da funcionalidade que será implementada e qual deve ser a interface da aplicação;
2. Efetuada a fase vermelha é necessário executar o teste e comprovar que ele será falho;
3. Alterações são aplicadas buscando obter a realização do teste com sucesso, mesmo que não sejam empregadas boas práticas de programação;
4. Executar outra vez o teste e constatar seu sucesso;
5. Refatoração do código com o objetivo de excluir as possíveis duplicações e adicionar boas práticas de programação, para que resulte em um bom desenvolvimento de código.
6. O ciclo deve ser reiniciado e um novo caso de teste deve ser escrito para passar por todas as fases com uma nova função a ser testada.

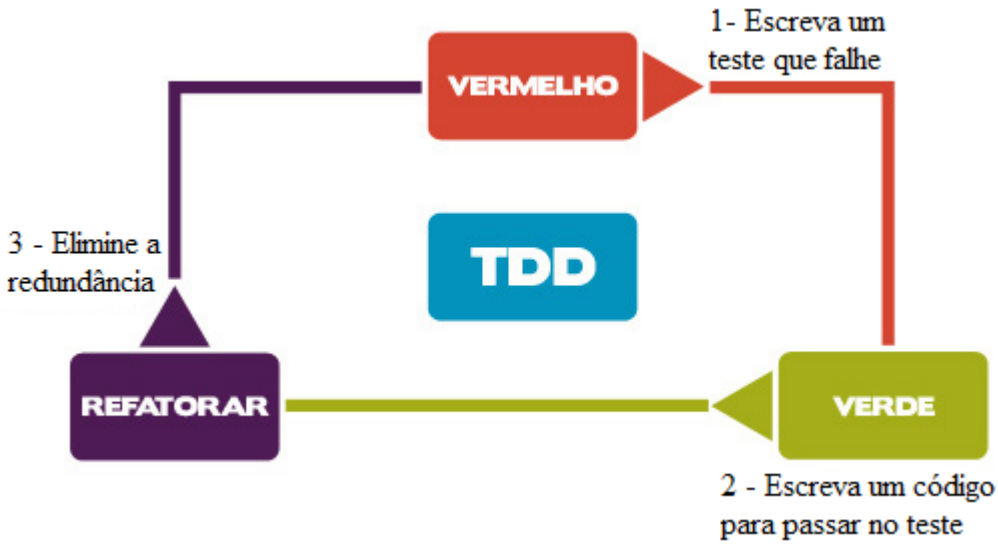


Figura 3.2: Ciclo de desenvolvimento do Test-Driven Development (Traduzida BECK, 2012)

Como visto na Seção 3.3, a ferramenta JUnit apoia a criação de testes de unidade, que são muito utilizados na técnica TDD. Além do JUnit, existem outras ferramentas como o NUnit (NUNIT, 2002) para linguagem C#, o MinUnit (MINUNIT, 1998) para linguagem C e o PHPUnit (PHPUNIT, 2001) para linguagem PHP, que também podem ser usadas em conjunto com o TDD para a criação de testes.

O objetivo do TDD não é somente testar e também não há garantia de que ele trata todos os casos de testes possíveis. Ao adicionar um novo recurso, para escrever o teste necessita-se de um exercício mental – se deve descrever o problema a ser solucionado, então após essa etapa é possível iniciar a implementação do problema descrito. Em outras palavras, o TDD impulsiona que se deve pensar na codificação antes de começar a desenvolver a solução (JOHANSEN, 2011).

3.6.1 Comparação entre testar a aplicação primeiro ou depois do desenvolvimento do código

Seguindo-se as diretrizes do TDD, o teste deve efetivado primeiro e depois desenvolvido o código para determinar qual a funcionalidade será implementada. Ao contrário do processo de desenvolvimento tradicional que desenvolve primeiramente para depois efetivar testes no código, como demonstrado na Figura 3.3 (VU *et al*., 2009).

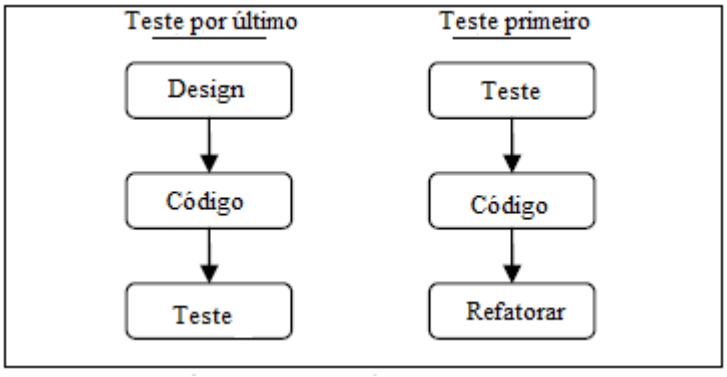


Figura 3.3: Comparação da realização de testes entre TDD e o Desenvolvimento tradicional - (Traduzida VU *et al*., 2009)

Na Figura 3.3 está descrito a fase de *design*, código e de testes que comumente são feitas no ciclo de desenvolvimento de um software. Tanto o teste executado durante o TDD como no desenvolvimento tradicional assumem que a fase de requisitos e arquitetura de alto nível os precedem, e que os testes também são acompanhados pela garantia de qualidade. No desenvolvimento tradicional, a refatoração é feita no momento em que os desenvolvedores encontram algumas falhas no software (VU *et al*., 2009).

A seguir estão os passos de forma resumida tanto do TDD como do desenvolvimento tradicional, ilustrados na Figura 3.4 (ERDOGMUS; MORISIO; TORCHIANO, 2005):

Passos para teste no TDD:

1. Escolha um requisito que o cliente forneceu.

2. Escreva o teste para uma tarefa ou parte da funcionalidade que deve ser implementada, nesse passo o teste terá que falhar.

3. Implemente o código para que o teste que foi escrito previamente seja executado com sucesso.

4. Devem-se executar todos os testes.

5. Refatore o código de teste de forma que ele esteja claro e simples, assumindo que ele será bem sucedido.

6. Refaça do passo 2 ao passo 5 até a funcionalidade ser implementada.

Passos para teste do desenvolvimento tradicional:

1. Escolha um requisito que o cliente forneceu.

2. Implemente o código de acordo com a funcionalidade que atenda ao requisito do cliente.

3. Escreva testes para validar a funcionalidade que foi implementada.

4. Devem-se executar todos os testes.

5. Refatore, caso seja preciso.

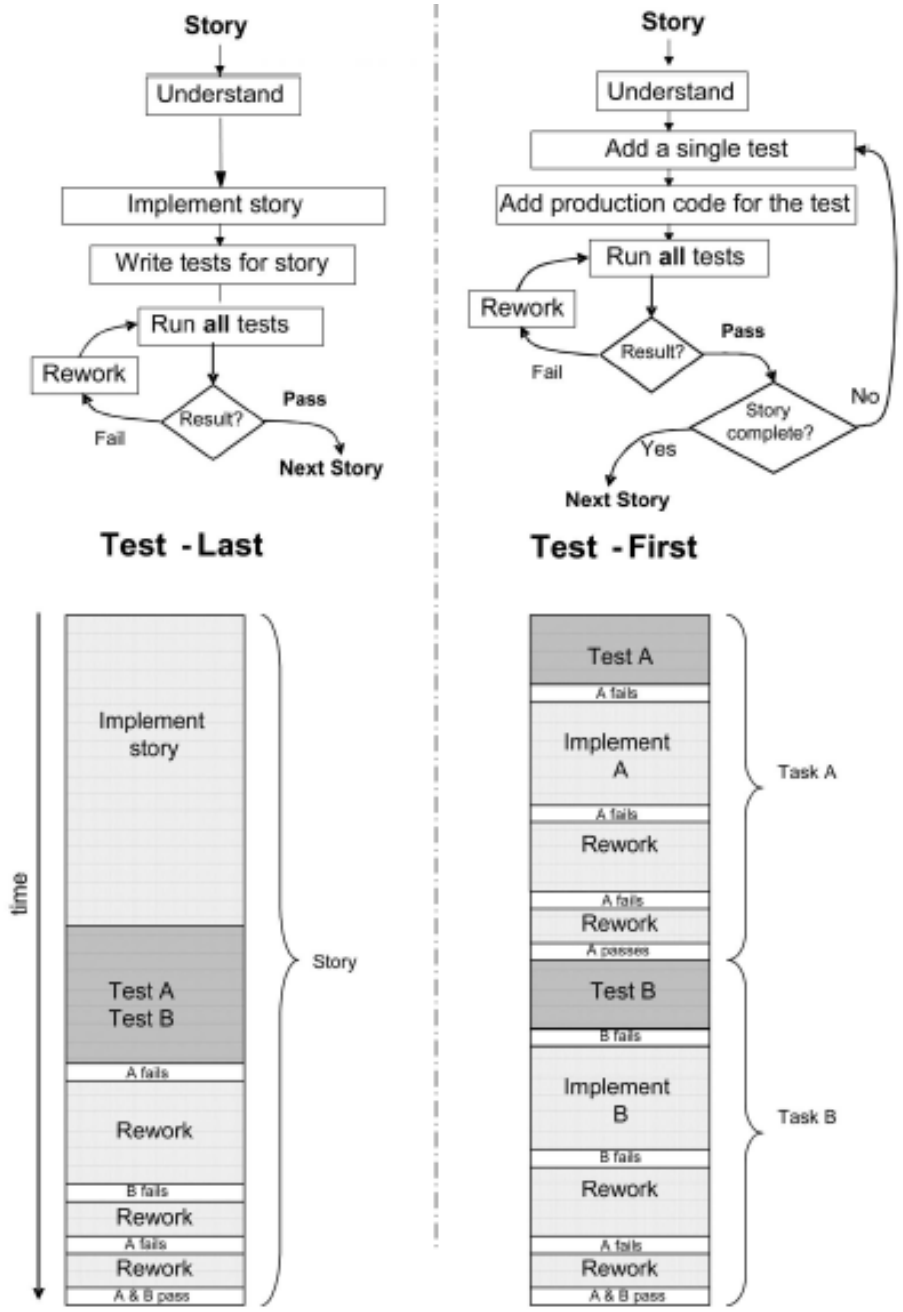


Figura 3.4: Dinâmica do TDD versus Desenvolvimento tradicional - (ERDOGMUS; MORISIO; TORCHIANO, 2005)

O desenvolvedor que faz uso do TDD faz a criação de testes de maneira gradual, implementando partes de uma funcionalidade para um recurso ser inteiramente desenvolvido. Ao contrário do desenvolvimento tradicional em que primeiramente é implementado o código de um requisito que o cliente forneceu e após essa fase é feito o desenvolvimento e a execução dos testes (VU *et al*., 2009).

3.6.2 Vantagens e Desvantagens do Test-Driven Development

Foram estudadas algumas possíveis vantagens da adoção da técnica TDD, são elas (WILLIAMS; MAXIMILIEN, 2003):

1. A sua eficiência, pois o ciclo de testar e depois implementar permite que o desenvolvedor obtenha um *feedback* constante. São descobertas falhas ou defeitos precocemente sendo que de forma rápida um novo código é adicionado na aplicação para solucioná-los. Como as falhas e os defeitos são identificados e removidos rapidamente, então se tem que o tempo gasto para escrever os casos de teste será compensado.

2. Asserções de teste em TDD têm a atenção dos programadores para escrever código que são automaticamente testáveis, seria como possuir funções e métodos, nas quais os resultados podem ser conferidos com os resultados esperados. Além disso, o TDD também permite a construção de uma aplicação mais confiável; o aperfeiçoamento da qualidade do trabalho de testes; a diminuição do esforço despendido para o teste. A prática de desenvolver o teste e depois efetuar a implementação do código são características para um projeto com a abordagem do TDD. Tais testes são também utilizados para o teste de regressão, nos quais ocorre a reexecução dos mesmos testes que já foram executados, visando garantir que quaisquer mudanças não tenham afetado negativamente demais partes do código.

3. A depuração e manutenção de uma aplicação muitas vezes são vistas como atividades de menor custo em que o código é corrigido pelos desenvolvedores, para modificar suas especificações e propriedades. Tais correções e certas modificações podem produzir um código quase 40 vezes mais favorável a conter defeitos (HUMPHREY, 1989). Para solucionar tal problema o TDD possui a simplicidade de executar casos de testes automatizados após algumas modificações feitas de forma que novas funcionalidades sejam integradas para o código base, com isso diminui a hipótese de correções ou manutenções implicarem em falhas permanentes.

Em contraste com as vantagens apresentadas, também se pode analisar algumas desvantagens, como (SLYNGSTAD *et al*., 2008):

1. *Design* - TDD geralmente contém pouco ou nenhum *design* do código, o que pode acarretar em ausência de documentação ao ocorrer algum defeito na aplicação, e então é notada a falta de informações que indique o estado anterior.

2. Contexto - Deve existir um contexto para a escrita dos casos de teste e muitas vezes o tempo requerido para desenvolver o caso de teste pode tornar-se considerável. Além disso, não há como garantir que riscos relacionados com a falta de requisitos ou com requisitos erroneamente definidos serão eliminados.

3. Rafatoração - É extensamente empregado para atender a complexidade ao utilizar o TDD.

4. Nível de habilidade - Necessita-se possuir um bom nível de experiência e conhecimento para desenvolver e conservar as asserções de teste em TDD.

De acordo com as vantagens e desvantagens apresentadas pode-se analisar que com a utilização de TDD é possível alterar um comportamento de determinada parte da aplicação de modo a garantir que não ocorrerão efeitos colaterais em outras partes. Logo, quando se tem um conjunto de testes que são elaborados de forma eficiente, eleva-se a capacidade de criar projetos flexíveis. Outro ponto a ser mencionado é que o TDD não assegura que a aplicação será entregue na data estimada, porém propõe que com a realização dos testes é possível diminuir o tempo de depuração do software. Também existe uma grande facilidade para saber qual o andamento de uma tarefa ao ter a possibilidade de ver que os testes estão sendo executados com sucesso (MARTIN, 2007).

3.7 Behaviour-Driven Development

Behaviour-Driven Development (BDD) é uma técnica ágil (NORTH, 2006), e uma evolução do *Test-Driven Development* (TDD), como uma junção de outras técnicas ágeis existentes que são úteis para o desenvolvimento de software, cujo destaque para sua utilidade se deve à redução dos custos com modificações no software e funções do comportamento. O BDD como uma técnica ágil incentiva a participação e a colaboração de todos os membros de um projeto (LAZAR;

MOTOGNA; PÂRV, 2010). Ao longo dos anos, BDD progrediu para um processo que abrange análise de requisitos e desenvolvimento do código (LAPOLLI *et al*., 2010).

Tal prática de teste faz uso da linguagem narrativa em junção com a linguagem ubíqua, para a escrita de casos de testes e favorece a definição do comportamento do sistema, a linguagem que se aplica ao BDD é retirada dos cenários criados durante a fase de análise ou levantamento de requisitos e permite uma comunicação entre todos os membros da equipe (SOARES, 2011).

Na narrativa da Figura 3.5, observa-se que se trata de um texto de fácil entendimento empregando-se uma linguagem comum, na qual qualquer membro de uma equipe que esteja inserida no desenvolvimento do projeto é capaz de compreender (ou seja, um *stakeholder*). Tal narrativa é denominada de *user stories,* utilizada para descrever os requisitos do software e de cenários para relatar os contextos característicos. Esses cenários são úteis para auxiliar a especificação em relação ao comportamento esperado do sistema para um contexto específico, tais cenários necessitam ser automatizados e executáveis para validar que o comportamento esperado do sistema esteja correto (USSAMI, 2013).

```
Título (uma linha descrevendo a história)

Narrativa:
Como [o papel]
Eu quero [recurso]
Assim que [benefício]

Critérios de Aceitação: (apresentado como Cenários)

Cenário 1: Título
Dado contexto []
  E [um pouco mais de contexto] ...
Quando [eventos]
Então [resultado]
  E [outro resultado ...]

Cenário 2: ...
```

Figura 3.5: Modelo de narrativa em linguagem ubíqua - (SOARES, 2011)

O ponto central está na especificação do comportamento do sistema que contém partes suficientes para que possam ser automatizadas (SOLÍS; WANG, 2011). A utilização de BDD proporciona proximidade entre desenvolvedores e testadores. Praticar o que está descrito nas metodologias ágeis deve ser aplicado a todos os membros que trabalham em corporações que adotam tais práticas, entretanto a equipe de teste às vezes não é bem aceita pela equipe de

desenvolvimento, quando na realidade a equipe de teste trabalha para entregar um produto de qualidade para os clientes (SOARES, 2011). A dificuldade que se pode notar é a adaptação para pessoas que não possuem experiência no desenvolvimento guiado ao comportamento. O passo inicial de BDD é a escrita de testes antes de desenvolver o código funcional (SOARES, 2011).

O BDD contém uma gama de atividades de desenvolvimento incluindo o levantamento de requisitos, análise, projeto e implementação. De acordo com SÓLIS e WANG (2011), foram identificadas seis características principais de BDD com base na literatura e na análise de ferramentas, são elas:

1. Linguagem Ubíqua: Como já visto na Seção 2.3, a utilidade da linguagem ubíqua dentre os conceitos estudados em relação ao DDD, acrescenta-se que ela também é empregada em projetos que contenham a técnica ágil BDD, de forma que o conceito de linguagem ubíqua encontra-se no cerne de BDD. É usada para estruturar as *user stories* e cenários. Existe uma linguagem baseada no modelo de domínio de negócios, na qual desenvolvedores, analistas, testadores e pessoas de negócio conseguem comunicar-se sem ambiguidade, o reflexo disso está em comportamentos bem especificados do sistema (NORTH, 2006).

2. Processo Iterativo de Decomposição: Às vezes torna-se um processo difícil o levantamento de requisitos durante a comunicação com os clientes. Portanto, no BDD a análise se inicia com a identificação dos comportamentos esperados de uma aplicação que são mais simples para serem identificados. Estes comportamentos são derivados dos resultados de negócios esperados que se pretende construir, eles são especificados e separados em conjuntos de recursos (SOLÍS; WANG, 2011).

A criação de *User stories* concebe o contexto das funcionalidades entregues por uma aplicação, é possível verificar qual a regra do usuário, qual o requisito desejado e quais são as vantagens que a funcionalidade propicia ao sistema. Existem três questões que devem ser elucidadas por uma *user story*:

- Qual é o papel do usuário na *user story*?
- Qual o recurso que o usuário solicitou?
- Qual a vantagem que o usuário pode ganhar se o sistema conceber tal recurso?

Para uma *user story*, podem existir versões distintas em contextos distintos. As instâncias específicas de uma *user story* são denominadas cenários, estes

são empregados como modo de aceitação, pois são fornecidos pelos clientes. Todo esse processo deve ser feito de maneira iterativa (SOLÍS; WANG, 2011).

3. Descrição de textos simples por meio de *templates* de *user stories* e cenário: Para as descrições de *user stories* e cenários existe um padrão, modelos pré-definidos são usados para especificá-los.

User stories podem possuir mais de um cenário para a descrição da forma em que a funcionalidade será implementada e quais as consequências para a aplicação no momento em que um estado específico for atingido e um evento for executado (SOLÍS; WANG, 2011). O modelo para a escrita de cenários pode ser visto conforme a Figura 3.6:

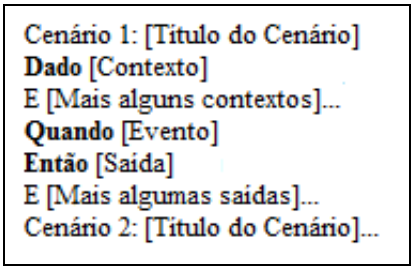


Figura 3.6: Template para cenários - (Traduzida NORTH, 2006)

O resultado do cenário é uma ação que modifica o estado de uma aplicação ou ocasiona a saída de uma aplicação. *Templates* tanto das *user stories* como dos cenários são descritos com o uso da linguagem ubíqua e mapeados diretamente em testes, o que quer dizer que os métodos e as classes também precisam ser escritos na linguagem onipresente (SOLÍS; WANG, 2011).

4. Testes de aceitação automatizados com regras de mapeamento: BDD herdou uma característica de desenvolvimento dirigido a testes de aceitação (ATDD), que são descritos em uma linguagem comum a todos os membros da equipe, essa atividade de teste representa as expectativas acerca do comportamento do sistema, ou seja, quais são as ações permitidas ou as restrições que devem estar presentes na aplicação (FILHO; SILVA, 2012). Desenvolvedores começam a partir de cenários produzidos em um processo de decomposição iterativa. Cenários são traduzidos para a execução de testes que irão coordenar a

implementação durante o desenvolvimento. Um cenário é um conjunto de vários elementos: contexto, eventos e consequências ou ações, sendo que cada elemento é mapeado em um método de teste. Com o intuito de passar pelo cenário descrito é preciso antes passar por todas as etapas, cada passo executado segue o processo de TDD, sendo vermelho, verde e refatoração, para obtenção de sucesso durante a implementação de um cenário, todos os métodos necessitam passar por testes bem sucedidos (SOLÍS; WANG, 2011).

5. Código legível de especificação orientado a comportamento: BDD propõe que o código deve ser parte da documentação de uma aplicação, seguindo os valores dos princípios ágeis. O código deve ser compreendido e a especificação deve ser parte do código, os nomes dos métodos devem ser intuitivos de forma que indiquem qual o comportamento dos métodos, refletindo suas funções, além de que todos devem estar na linguagem ubíqua determinada em um projeto (SOLÍS; WANG, 2011).

6. Comportamento dirigido em diferentes fases: É essencial que o comportamento esperado do sistema reflita os requisitos do cliente e seja praticado nas diferentes fases do sistema. Durante a fase de análise, os resultados de negócios discutidos previamente devem ser decompostos em um conjunto de funcionalidades que representam o comportamento esperado do sistema, na fase de implementação o comportamento tem que ser contemplado pelos testes de aceitação automatizados e pelos nomes de métodos e classes que são intuitivos para saber qual o seu comportamento (SOLÍS; WANG, 2011).

Um *framework* que apoia a técnica BDD é o *Cucumber* (CUCUMBER, 2014), que será usado no estudo de caso definido neste trabalho. Além do *Cucumber*, existem outros que se destacam, são eles: JBehave (JBEHAVE, 2003), RSpec (RSPEC, 2005), SpecFlow (SPECFLOW, 2013) e StoryQ (STORYQ, 2006). Todos os *frameworks* citados, inclusive o *Cucumber*, não dão suporte a todas as etapas da técnica BDD. Eles apoiam basicamente a etapa de implementação, porém, a etapa de projeto depende exclusivamente do desenvolvedor.

A seguir é mostrado um exemplo de teste com o *framework Cucumber*. O caso de teste é o mesmo apresentado na Seção 3.3, relativo à multiplicação de dois números. A Figura 3.7 mostra um cenário de teste, escrito em linguagem narrativa. Este cenário será executado em uma classe de teste, mostrada na Figura 3.8.

```
Feature: Testar multiplicação entre dois números

    @integration
    Scenario: Multiplicar 2 por 3 deve resultar em 6
        Given Eu tenho o numero 2 e o numero 3
        When  Eu multiplico os dois numeros
        Then  Eu obtenho o numero 6
```

Figura 3.7: Exemplo de cenário de teste com Cucumber

```
public class TesteMultiplicacao {

    int a, b, res;

    @Given("^Eu tenho o numero (\\d+) e o numero (\\d+)$")
    public void Eu_tenho_o_numero_e_o_numero(int arg1, int arg2) {
        a = arg1;
        b = arg2;
    }

    @When("^Eu multiplico os dois numeros$")
    public void Eu_multiplico_os_dois_numeros() {
        res = a * b;
    }

    @Then("^Eu obtenho o numero (\\d+)$")
    public void Eu_obtenho_o_numero(int arg1) {
        Assert.assertEquals(arg1, res);
    }
}
```

Figura 3.8: Exemplo de classe de teste com Cucumber

Como visto acima, para criar um teste com o *Cucumber* é necessário criar um cenário em linguagem narrativa e uma classe. A classe é responsável por executar todo cenário que corresponda com as respectivas frases nas anotações `@Given`, `@When`, `@Then`. Essa correspondência é feita pelo *framework* por meio de expressão regular.

3.7.1 O ciclo do Behaviour-Driven Development

O ciclo de execução do BDD é fundamentado na fase vermelha, verde e de refatoração proposta inicialmente pelo TDD, como exposto na Seção 3.6. Contém

passos curtos e iterativos, sendo ao todo sete passos, conforme pode ser analisado na Figura 3.9 (CHELIMSKY *et al.*, 2009):

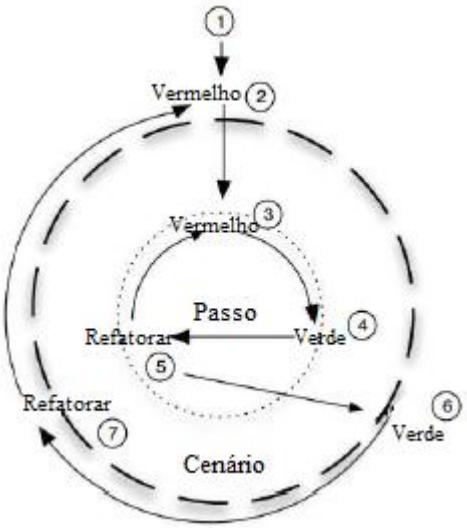


Figura 3.9: Ciclo do Behaviour-Driven Development - (Traduzida CHELIMSKY *et al*., 2009)

O primeiro passo que deve ser feito ao utilizar BDD para o desenvolvimento de uma aplicação é descrever um cenário específico de uma funcionalidade. O segundo passo é a escrita de um cenário que irá falhar. Para atingir um cenário de sucesso é preciso definir passos de acordo com a metodologia empregada pelo TDD, são eles: escrita de uma etapa falha, terceiro passo do ciclo, a correção da mesma, quarto passo, e refatoração do código, quinto passo. As etapas *Given*, *When* e *Then*, fazem parte de um cenário denominado simples no BDD, dessa forma o terceiro, quarto e quinto passos devem ser repetidos até atingir o estado de um cenário de sucesso, sendo este o sexto passo, e por fim acontece a refatoração do ciclo, sétimo passo. Terminada a execução de todos os passos contidos em um ciclo do BDD, todos eles são executados novamente, porém com a descrição de um novo cenário (CHELIMSKY *et al*., 2009).

Diversas abordagens são combinadas para fazer os passos propostos pelo BDD, incluindo a linguagem ubíqua, TDD e teste automatizado de aceitação. O BDD aprimora todas essas abordagens para extrair o maior proveito de todas elas.

3.8 Comparação entre Test-Driven Development e Behaviour-Driven Development

Essas duas técnicas de teste de software têm seus testes escritos como uma forma de garantir que a maior parte do código será testada, e não como uma maneira de pensar no código em si. O TDD e o BDD são muito parecidos do ponto de vista do ciclo de execução, ou seja, dos passos que devem ser feitos para que a técnica seja usada de maneira correta. Isso se deve ao fato de que o BDD se originou do TDD (CHELIMSKY, 2009). As fases do ciclo de execução de cada técnica podem ser vistas na Tabela 1.

Tabela 3.1: TDD e BDD: Ciclo de Execução

Fases	**TDD**	**BDD**
1	Escrever um teste	Escrever um cenário de teste
2	Executar o teste e o ver falhar	Executar o cenário e o ver falhar
3	Escrever código suficiente para que o teste passe	Escrever código suficiente para implementar o comportamento esperado
4	Executar o teste e o ver passar	Executar o cenário e o ver passar
5	Refatorar o código	Refatorar o código

Apesar de o ciclo de execução das técnicas ser semelhante, elas são abordagens diferentes para solucionar o mesmo tipo de problema. A diferença entre as duas técnicas é que o TDD lida com testes unitários, enquanto que o BDD lida com o comportamento de toda uma estória de usuário.

Além disso, em BDD, a escrita de cenários em uma linguagem narrativa proporciona maior entendimento por todos os envolvidos em um projeto, tanto da área técnica quanto da área de negócios. Esses cenários podem ser utilizados como parte da documentação do projeto. Portanto, essas características fazem do BDD uma abordagem que facilita a comunicação, o que é essencial para projetos ágeis.

3.9 Considerações Finais

Neste Capítulo foram discutidos os conceitos relacionados à atividade de teste e a importância de sua aplicação para a obtenção de um produto final de maior qualidade. A realização dos testes nunca se encerra, visto que essa atividade desloca-se do engenheiro de software para o usuário final.

Por meio deste levantamento bibliográfico, foram discutidos três abordagens de teste de software sendo elas: o teste de unidade, teste de integração e o teste automatizado, com o intuito de salientar o quanto esses conceitos são relevantes e estão fortemente relacionados às técnicas TDD e BDD.

O TDD e o BDD são práticas para a elaboração de testes que visam à qualidade e à redução de falhas de uma aplicação, almejando-se maior satisfação para o cliente. Dessa forma, espera-se aplicá-las, separadamente, com a abordagem de desenvolvimento DDD que foi introduzida no Capítulo anterior. O próximo Capítulo apresenta como foi feita a integração de cada técnica de teste TDD e BDD com a abordagem de desenvolvimento DDD. Além disso, é mostrada a arquitetura de um projeto *Apache Isis* e como é gerada a camada de visualização de dados por meio do *framework Apache Isis*.

Capítulo 4

PROPOSTA DE INTEGRAÇÃO DDD, TDD E BDD

Este capítulo apresenta a proposta de integração das técnicas TDD e BDD com a abordagem de desenvolvimento DDD. A integração é apresentada por meio da definição de um estudo de caso: um sistema web de Vídeo Locadora. O estudo de caso está descrito em duas etapas: definição do modelo de domínio e a posterior implementação por meio do framework Apache Isis.

4.1 Considerações Iniciais

Considerando que no cenário atual as mudanças nos requisitos de software são frequentes, o processo de desenvolvimento requer agilidade e respostas rápidas a tais mudanças. Além disso, é de grande importância que todos os envolvidos tenham bom entendimento do domínio do projeto. Essas são, entre outras, características da abordagem ágil de desenvolvimento *Domain-Driven Design*, que foi descrita no Capítulo 2.

Como visto no Capítulo 3, a utilização de técnicas de teste são essenciais para garantir a qualidade de um software e por isso o objetivo deste trabalho é integrar cada técnica de teste estudada (*Test-Driven Development* e *Behaviour-Driven Development*) com o DDD, uma vez que o foco do DDD é no *design* do software. De acordo com a Seção 1.4, pouco é dito sobre as estratégias de teste em DDD.

Para exemplificar a proposta de integração das técnicas ágeis apresentadas, optou-se por desenvolver um estudo de caso: uma aplicação web de vídeo locadora. O desenvolvimento foi apoiado pelo *framework Apache Isis*, o qual emprega os conceitos de DDD e é indicado para prototipação rápida de sistemas.

Assim, este capítulo está dividido da seguinte forma: A Seção 4.2 apresenta como foi definido o modelo de domínio do exemplo de vídeo locadora proposto neste trabalho. Na Seção 4.3 é apresentada a integração de cada técnica de teste TDD e BDD, separadamente, com a abordagem DDD. Além disso, também é apresentada a arquitetura de um projeto *Apache Isis* relacionando-a com a arquitetura em camadas proposta pelo DDD e como foi feito o desenvolvimento utilizando os conceitos do DDD. Na Seção 4.4 é mostrada a camada de visualização dos dados gerada automaticamente pelo *framework Apache Isis*. As considerações finais são feitas na Seção 4.5.

4.2 Modelo de Domínio

A definição de um modelo de domínio é um processo contínuo na construção de um software. Por ser uma abordagem ágil, o DDD incentiva a evolução constante do modelo de domínio, o qual não é apenas um documento formal ou um diagrama, mas sim um conjunto de artefatos. Fazem parte do modelo de domínio: requisitos do cliente, evidências de conversas entre envolvidos no projeto, a linguagem ubíqua, diagramas de modelagem e até o próprio código desenvolvido.

No caso deste trabalho, não houve uma equipe para elaboração do modelo de domínio e posterior implementação. Porém, para exemplificar a integração proposta, foi escolhido um domínio de vídeo locadora que já havia sido explorado na tese de doutorado intitulada "Desenvolvimento e o reúso de frameworks com base nas características do domínio" (VIANA, 2014). Criou-se então um documento com alguns requisitos iniciais para esse domínio. São eles:

1. A locadora realiza o aluguel de DVDs de filmes que podem ter uma ou mais cópias. Cada filme possui um identificador, título e ano.

2. Cada DVD possui um identificador, um status de disponibilidade e o filme nele contido.

3. Os filmes são classificados por categorias, que indicam o valor da locação.

4. Os filmes também são classificados por gêneros, que definem uma idade mínima para que possa ser locado por um cliente.

5. Os DVDs são locados para os clientes cadastrados da locadora. São cadastrados um identificador, nome, CPF, idade e telefone para um cliente.

6. As informações de locação são: identificador único da locação, data de locação, data de devolução prevista, data de devolução efetiva, cliente que realizou a locação, a lista de DVDs locados e o valor total da locação.

7. Para que a locação de um filme seja feita por um cliente, este deve ter a idade mínima indicada no gênero do filme.

8. Para calcular a data prevista de devolução dos filmes considera-se a quantidade de filmes locados. Para 1 ou 2 filmes: 2 dias de prazo. Para 3 ou 4 filmes: 3 dias de prazo. Para 5 filmes ou mais: 4 dias de prazo.

9. Caso a locação ocorra durante os seguintes dias da semana (Segunda, Terça, Quarta, Quinta), o cliente ganha mais 1 dia de prazo para devolução.

10. Caso o cliente ultrapasse a data de devolução ele será sujeito a multa. Para cálculo de multa será considerado 10% do valor da locação para cada dia de atraso.

O documento de requisitos foi evoluído ao longo do processo de análise e desenvolvimento do projeto. Tal evolução refletiu em alterações no modelo de domínio, como prega o DDD. A partir do documento de requisitos e de algumas conversas casuais com o Prof. Dr. Matheus Carvalho Viana e com o orientador desta pesquisa, foi possível criar o modelo do domínio de vídeo locadora. O documento de requisitos em sua versão final é apresentado no Apêndice A. O diagrama de classes, também em versão final, elaborado para auxiliar na implementação é mostrado na Figura 4.1:

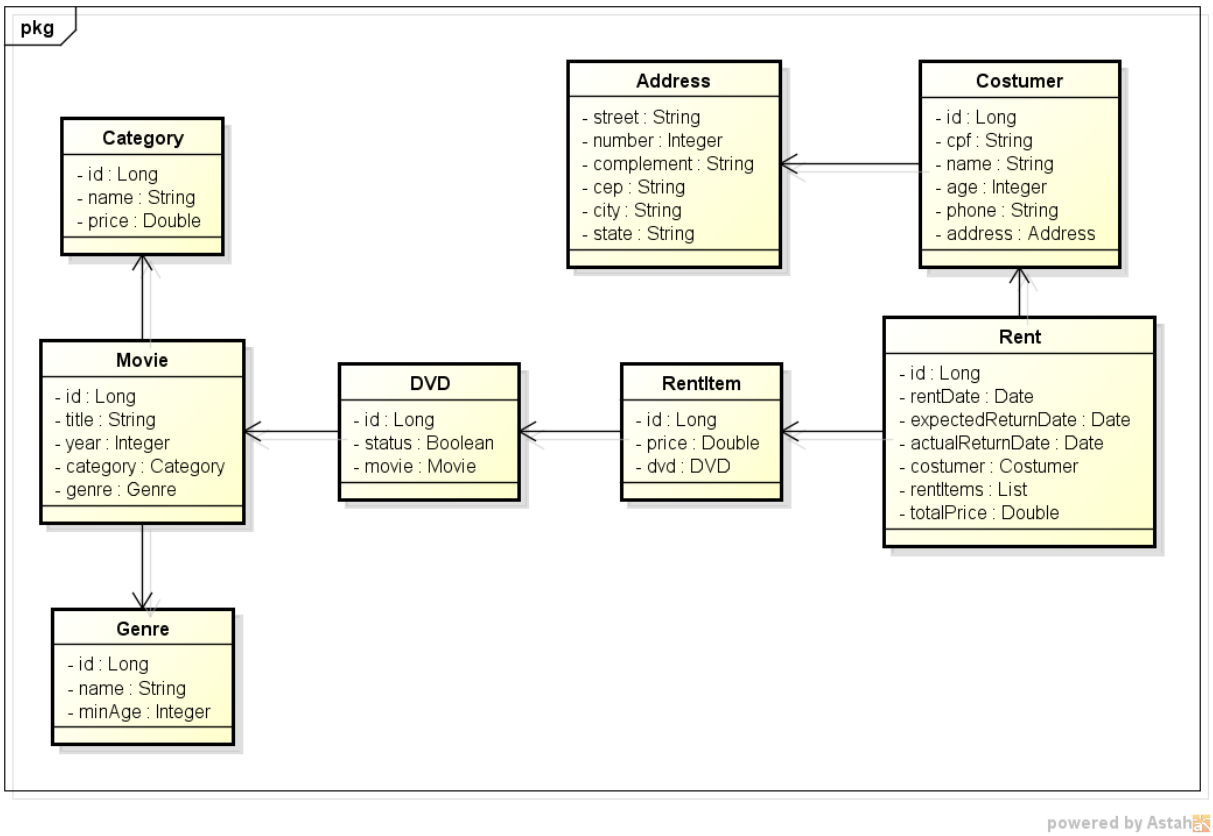


Figura 4.1: Diagrama de Classes do domínio de vídeo locadora

Como pode ser visto na descrição dos requisitos e no diagrama de classes, os termos utilizados são comuns aos envolvidos com a área de negócios e aos envolvidos com o desenvolvimento. Essa é a principal característica do DDD: a linguagem ubíqua. Como mencionado na Seção 2.3, a linguagem ubíqua é de grande importância para que haja uma boa comunicação e um bom entendimento acerca do domínio. Os blocos de construção presentes no DDD que foram utilizados para a implementação do modelo de domínio deste trabalho, serão mostrados na Seção 4.3.3.

4.3 Integração de técnicas de teste com Domain-Driven Design

A abordagem de desenvolvimento *Domain-Driven Design* não explica como devem ser feitos os testes de uma aplicação, apesar de enfatizar que testes são essenciais. Então, a seguir será mostrado como foi feita a integração da abordagem

DDD com cada técnica de teste, *Test-Driven Development* e *Behaviour-Driven Development* na implementação do exemplo proposto de vídeo locadora.

4.3.1 Integrando Domain-Driven Design e Test-Driven Development

Como visto na Seção 3.6, o TDD parte do princípio de pensar primeiramente sobre como testar uma funcionalidade que será implementada. Portanto, antes de iniciar, efetivamente, a codificação do modelo de domínio, foram implementados alguns testes seguindo os conceitos de TDD. Para a implementação dos testes de unidade, o *Apache Isis* incorpora o JUnit (JUNIT, 2014), que é um *framework* para criação de testes de unidade automatizados. Os testes foram criados seguindo as fases vermelha, verde e refatoração, de acordo com a literatura.

Para exemplificar essa integração foi escolhido o requisito número 12 do documento de requisitos do domínio de vídeo locadora:

12. Para que a locação de um filme seja efetivada, deve haver pelo menos um DVD disponível daquele filme.

No domínio escolhido, um filme pode ter um ou mais DVD's. Logo, para que esse requisito seja validado, o sistema de vídeo locadora deve checar se há pelo menos um DVD disponível daquele filme. A seguir serão mostrados os passos para a criação do teste dessa funcionalidade, seguindo as fases do TDD.

Após a escolha do requisito a ser testado, foi feita a criação de uma classe de teste e de um método responsável por validar tal requisito. Nesse momento, o método de teste contém apenas a chamada do método da própria entidade `Movie`, entidade que representa um filme, que verifica a disponibilidade de um filme. Porém, a entidade `Movie` ainda não foi criada, o que causará a falha desse método de teste. Esta é denominada a fase vermelha do TDD. O código de teste inicial pode ser visto na Figura 4.2:

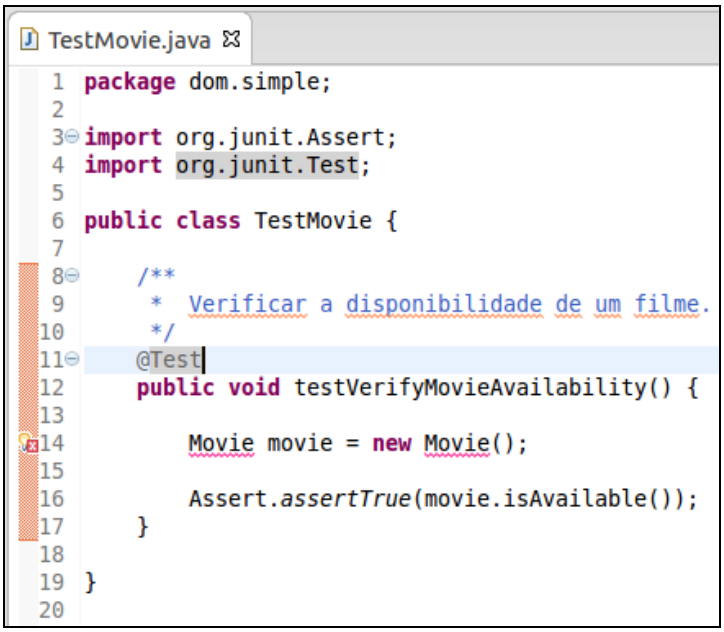


Figura 4.2: Código inicial de teste

Ao executar o teste, como já mencionado acima, uma falha irá ocorrer. O resultado da execução do teste é mostrado na Figura 4.3:

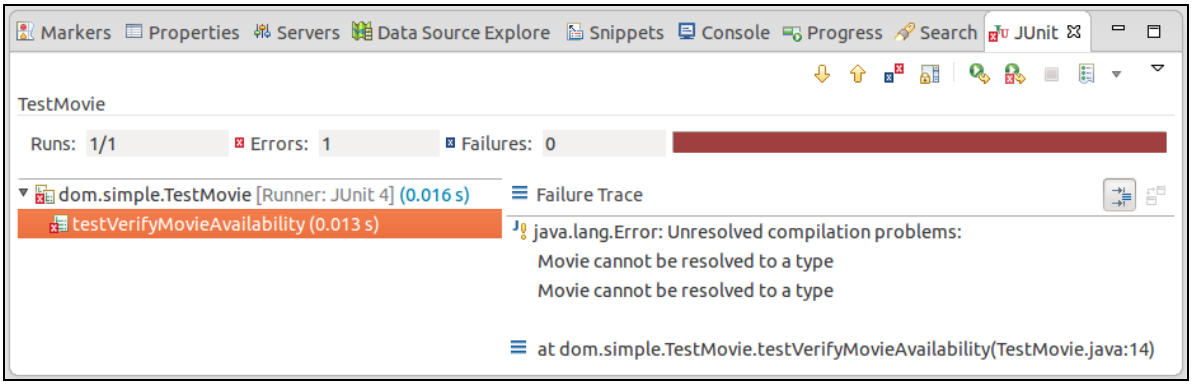


Figura 4.3: Resultado de falha da execução do teste

O próximo passo deve ser a escrita do código necessário para que o teste seja executado com sucesso. No caso do domínio exemplificado, foram criadas as entidades `Movie` e `DVD`. Esta última representa um DVD e tem um relacionamento com `Movie`, porque um filme possui uma referência para uma lista de DVD's. Na classe `Movie` também foi criado o método que percorre a lista de DVD's e verifica se há um disponível. A seguir, as Figuras 4.4 e 4.5 mostram o código da classe `Movie` e da classe `DVD`, respectivamente:

```java
public class Movie {

    private Long id;
    public Set<DVD> dvds = new HashSet<DVD>();

    public Long getId() {
        return this.id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Set<DVD> getDvds() {
        return this.dvds;
    }

    public void setDvds(Set<DVD> dvds) {
        this.dvds = dvds;
    }

    public boolean isAvailable() {

        for (DVD dvd : dvds) {
            if (dvd.getStatus()) {
                return true;
            }
        }

        return false;
    }
}
```

Figura 4.4: Código da entidade Movie

```java
package dom.simple;

public class DVD {

    private Long id;
    private Boolean status;
    private Movie movie;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public Boolean getStatus() {
        return status;
    }

    public void setStatus(Boolean status) {
        this.status = status;
    }

    public Movie getMovie() {
        return movie;
    }

    public void setMovie(Movie movie) {
        this.movie = movie;
    }
}
```

Figura 4.5: Código da entidade DVD

Também foi necessário fazer adequações no código do método de teste. Nele foram criados dois objetos `DVD`, um com status `true`, que significa disponível e um com status `false`, que significa indisponível. Eles foram adicionados na lista de DVD's do mesmo filme. Esse código pode ser visto na Figura 4.6:

```
6  public class TestMovie {
7
8      /**
9       *  Verifica a disponibilidade de um filme.
10      *
11      *  Deve retornar que o filme está disponível
12      *  quando há um DVD indisponível e um DVD
13      *  disponível para o mesmo filme.
14      */
15     @Test
16     public void testVerifyMovieAvailability_MustReturnAvailable() {
17
18         Movie movie = new Movie();
19
20         DVD dvd1 = new DVD();
21         dvd1.setStatus(false);
22         dvd1.setMovie(movie);
23
24         DVD dvd2 = new DVD();
25         dvd2.setStatus(true);
26         dvd2.setMovie(movie);
27
28         movie.getDvds().add(dvd1);
29         movie.getDvds().add(dvd2);
30
31         Assert.assertTrue(movie.isAvailable());
32     }
```

Figura 4.6: Código final de teste

Após a realização das alterações necessárias no código, o teste foi executado novamente. Desta vez, o teste foi executado com sucesso, porém nem sempre este resultado é alcançado. O desenvolvedor pode realizar quantas alterações forem necessárias até que o teste passe com sucesso. Esta é a fase verde do TDD. A Figura 4.7 exibe o resultado positivo da execução do teste:

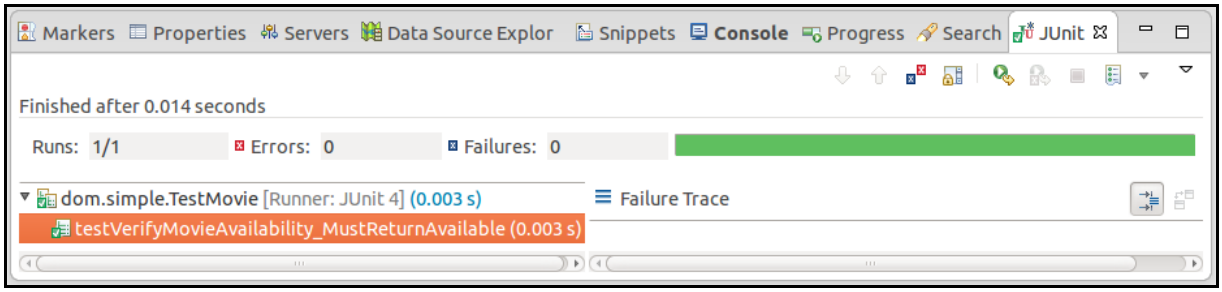


Figura 4.7: Resultado de sucesso da execução do teste

Após o teste ser executado com sucesso, o código pode ser refatorado, caso necessário, para melhorar sua legibilidade e remover duplicações. Essa é a fase de refatoração do TDD. Todas essas fases compõem um ciclo. O TDD prega que devem ser executados vários ciclos para uma mesma funcionalidade ou para novas funcionalidades.

4.3.2 Integrando Domain-Driven Design e Behaviour-Driven Development

Como visto na Seção 3.7, o BDD é uma evolução do TDD, então muitas características presentes no BDD vieram do TDD, como por exemplo o ciclo de execução, composto pelas fases vermelha, verde e refatoração. Porém, com o BDD, é possível criar cenários de teste em linguagem narrativa. Estes cenários são favorecidos com o uso da linguagem ubíqua, o que, consequentemente, facilita o entendimento de todas as partes envolvidas no projeto. Para a criação dos testes seguindo a técnica do BDD, o *Apache Isis* incorpora o *framework Cucumber* (CUCUMBER, 2014).

Os requisitos de número 8 e 9 do documento de requisitos de vídeo locadora foram escolhidos para exemplificar a integração da abordagem DDD com a técnica de teste BDD:

8. Para calcular a data prevista de devolução dos filmes considera-se a quantidade de filmes locados. Para 1 ou 2 filmes: 2 dias de prazo. Para 3 ou 4 filmes: 3 dias de prazo. Para 5 filmes ou mais: 4 dias de prazo.

9. Caso a locação ocorra durante os seguintes dias da semana (Segunda, Terça, Quarta, Quinta), o cliente ganha mais 1 dia de prazo para devolução.

De acordo com a descrição do requisito, um cenário de teste foi especificado como pode ser visto na Figura 4.8. Este cenário visa validar se são dados dois dias de prazo para devolução quando são locados um ou dois filmes e a locação não ocorre em uma segunda, terça, quarta ou quinta-feira. O cenário descrito é apenas um dos possíveis casos de teste desta funcionalidade. Várias outras combinações entre quantidade de filmes locados e datas de locação poderiam ser feitas para validar o requisito. Isso pode ser feito com a criação de vários cenários de teste, escritos em linguagem narrativa.

ExpectedReturnDateCalculationSpec.feature

```
1  Feature: Valida o cálculo de data prevista de devolução da locação
2
3      @integration
4      Scenario: Para 1 ou 2 filmes, deve ser dado 2 dias de prazo na locação
5          Given Eu desejo locar um filme
6          And   Eu desejo locar um filme
7          And   A locação ocorre em "05/09/2014"
8          When  Eu calculo a data de devolução da locação
9          Then  Eu me certifico que a data de devolução é "07/09/2014"
10
```

Figura 4.8: Cenário de teste com BDD

Caso a classe de teste responsável por executar os cenários não tenha sido criada o *framework Cucumber* exibirá um aviso de que a classe está indefinida. Esse aviso é mostrado na Figura 4.9:

```
7 Scenarios (1 undefined, 6 passed)
32 Steps (5 undefined, 27 passed)
0m0.205s
```

Figura 4.9: Aviso de classe de teste indefinida

Então uma classe de teste inicial deve ser escrita para que os cenários possam ser executados. Ao executar o cenário de teste, o *framework Cucumber* relaciona cada frase do cenário com um método correspondente, implementado na classe de teste. Esse relacionamento ocorre por meio da comparação da frase do cenário com a frase que está dentro das anotações `@Given`, `@When`, `@Then`. A Figura 4.10 mostra o código inicial da classe de teste.

Ao executar o cenário novamente, após a criação da classe inicial, o teste irá falhar, como previsto na fase vermelha. Como próximo passo, deve-se escrever todo o código necessário para que o cenário seja executado com sucesso. Para o domínio de vídeo locadora, a implementação do método de cálculo da data prevista de devolução é mostrado na Figura 4.11. Este método verifica a quantidade de filmes associados a uma locação para definir quantos dias serão dados de prazo. Após essa contagem, também é necessário checar em qual dia da semana a locação está sendo feita. Para isso foi utilizada a classe `Calendar` do Java. Por fim, o método retorna a data prevista de devolução para a locação.

ExpectedReturnDateCalculationGlue.java

```
1  package integration.glue.simple;
2
3  import org.apache.isis.core.specsupport.specs.CukeGlueAbstract;
8
9  public class ExpectedReturnDateCalculationGlue extends CukeGlueAbstract {
10
11     @Given("^Eu desejo locar um filme$")
12     public void Eu_desejo_locar_um_filme() throws Throwable {
13
14     }
15
16     @Given("^A locação ocorre em \"([^\"]*)\"$")
17     public void A_locação_ocorre_em(String arg1) throws Throwable {
18
19     }
20
21     @When("^Eu calculo a data de devolução da locação$")
22     public void Eu_calculo_a_data_de_devolução_da_locação() throws Throwable {
23
24     }
25
26     @Then("^Eu me certifico que a data de devolução é \"([^\"]*)\"$")
27     public void Eu_me_certifico_que_a_data_de_devolução_é(String arg1) throws Throwable {
28
29     }
```

Figura 4.10: Código inicial da classe de teste

Rent.java

```
249    public Date calculateExpectedReturnDate(Date rentDate) {
250
251        int days = 0;
252
253        if (this.rentItems.size() > 4) {
254            days = 4;
255        } else if (this.rentItems.size() > 2) {
256            days = 3;
257        } else if (this.rentItems.size() > 0) {
258            days = 2;
259        } else {
260            return null;
261        }
262
263        Calendar cal = Calendar.getInstance();
264        cal.setTime(rentDate);
265
266        if (cal.get(Calendar.DAY_OF_WEEK) == Calendar.MONDAY ||
267            cal.get(Calendar.DAY_OF_WEEK) == Calendar.TUESDAY ||
268            cal.get(Calendar.DAY_OF_WEEK) == Calendar.WEDNESDAY ||
269            cal.get(Calendar.DAY_OF_WEEK) == Calendar.THURSDAY) {
270            days++;
271        }
272
273        cal.add(Calendar.DAY_OF_MONTH, days);
274
275        return cal.getTime();
276    }
```

Figura 4.11: Método para calcular a data prevista de devolução de uma locação

O código da classe de teste também foi implementado para realizar a validação da funcionalidade. Nele é feita a chamada para o método de cálculo descrito acima, que se encontra na entidade `Rent`, entidade que representa locação. Este código pode ser visto nas Figuras 4.12 e 4.13.

```
@Given("^Eu desejo locar um filme$")
public void Eu_desejo_locar_um_filme() throws Throwable {
    Movie movie = new Movie();

    DVD dvd = new DVD();
    dvd.setMovie(movie);

    movie.getDvds().add(dvd);

    RentItem rentItem = new RentItem();
    rentItem.setDvd(dvd);

    Rent rent = new Rent();
    rent.getRentItems().add(rentItem);

    putVar("Rent", "rent", rent);
}

@Given("^A locação ocorre em \"([^\"]*)\"$")
public void A_locação_ocorre_em(String strRentDate) throws Throwable {
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
    Date rentDate = sdf.parse(strRentDate);
    putVar("Date", "rentDate", rentDate);
}
```

Figura 4.12: Métodos com a anotação @Given

```
@When("^Eu calculo a data de devolução da locação$")
public void Eu_calculo_a_data_de_devolução_da_locação() throws Throwable {
    Rent rent = getVar("Rent", "rent", Rent.class);
    Date result = rent.calculateExpectedReturnDate(getVar("Date", "rentDate", Date.class));
    putVar("Date", "result", result);
}

@Then("^Eu me certifico que a data de devolução é \"([^\"]*)\"$")
public void Eu_me_certifico_que_a_data_de_devolução_é(String strResult) throws Throwable {
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
    Date result = sdf.parse(strResult);
    Assert.assertEquals(getVar("Date", "result", Date.class), result);
}
```

Figura 4.13: Métodos com as anotações @When e @Then

Depois que todas as alterações necessárias no código foram feitas, o cenário de teste foi executado novamente. O resultado da execução foi de sucesso, o que representa a fase verde. A Figura 4.14 exibe o resultado da execução:

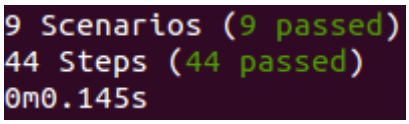


Figura 4.14: Cenário de teste executado com sucesso

Como mencionado anteriormente para o TDD, após o cenário executar com sucesso pode-se refatorar o código implementado, caso seja necessário, sendo esta a fase de refatoração. Também como no TDD, os ciclos de teste devem ser repetidos ao longo do desenvolvimento.

4.3.3 Implementação do modelo de domínio com o framework Apache Isis

Para desenvolver um projeto com o *framework Apache Isis* utiliza-se, necessariamente, a linguagem de programação Java (JAVA, 1995). Como ferramenta para auxiliar a construção, o gerenciamento de dependências, os testes e a implantação, é utilizado o *Apache Maven* (MAVEN, 2002). Esta é uma ferramenta amplamente usada no desenvolvimento de aplicações Java e por isso, por padrão, projetos feitos com o *Apache Isis* estão configurados para usá-la.

Um projeto *Apache Isis* tem algumas características particulares, como por exemplo, sua arquitetura. A arquitetura básica, usada no exemplo proposto, divide o projeto em quatro módulos: `dom`, `fixture`, `integtests` e `web-app`. O módulo `dom` contém toda a definição do modelo de domínio, portanto, é o módulo principal. No módulo `fixture` está todo o código ou script para o *setup* inicial do projeto. O `integtests` armazena os testes de integração, sendo este, o local padrão para a criação dos testes com BDD, utilizando o *framework Cucumber*. Por fim, o módulo `web-app` é o responsável por gerar automaticamente, à partir de alguns arquivos de configuração e do modelo de domínio, toda a camada de visualização de dados.

A separação em módulos é uma convenção do *Apache Isis* que permite o isolamento do domínio. Isolar o domínio é de suma importância para obter um modelo bem definido e coeso. Como citado na Seção 2.3.2 por Evans (2003), as

arquiteturas, de forma geral, possuem quatro camadas: camada de apresentação, camada de aplicação, camada de domínio e camada de infraestrutura. No exemplo proposto é possível fazer uma correspondência com tais camadas. A camada de infraestrutura é composta pelo servidor de aplicações responsável por disponibilizar a aplicação construída e pelo banco de dados em memória do *framework Apache Isis*. A camada de domínio é constituída inteiramente pelo módulo `dom` e a camada de apresentação pelo módulo `web-app`, que a gera de forma automática. A camada de aplicação não está presente na arquitetura do exemplo desenvolvido porque o *framework Apache Isis* dispensa a sua criação. Toda regra de negócio está presente no núcleo do modelo de domínio.

As Seções 4.3.1 e 4.3.2 mostraram, respectivamente, a integração da abordagem *Domain-Driven Design* com as técnicas de teste *Test-Driven Development* e *Behaviour-Driven Development*. Em ambas técnicas, a implementação do teste é feita antes mesmo da implementação do código da funcionalidade. Por isso, no caso do domínio de vídeo locadora proposto por este trabalho, grande parte do modelo de domínio, principalmente Entidades, foi implementado em conjunto com os testes.

A seguir serão mostrados trechos de código do modelo de domínio implementado. Esses trechos representam os blocos de construção presentes no DDD, conforme também explicado na Seção 2.3.2.

A Figura 4.15 mostra a Entidade `Movie`. Nela é possível notar que há um atributo identificador, ou seja, cada filme é único no sistema. Isso é o que caracteriza uma Entidade no modelo de domínio.

Movie.java

```
1  package dom.simple;
2
3  import java.util.HashSet;
17
18 public class Movie implements Comparable<Movie> {
19
20     private Long id;
21     private String title;
22     private Integer year;
23     private Category category;
24     private Genre genre;
25     public Set<DVD> dvds = new HashSet<DVD>();
```

Figura 4.15: Entidade Movie

Na Figura 4.16 é exibido um Objeto de Valor. A diferença entre uma Entidade e um Objeto de Valor é que o último não possui uma identidade, ou seja, tem apenas um aspecto descritivo. Neste caso, o Objeto de Valor chamado `Address`, que representa endereço, carrega os valores do endereço de um cliente.

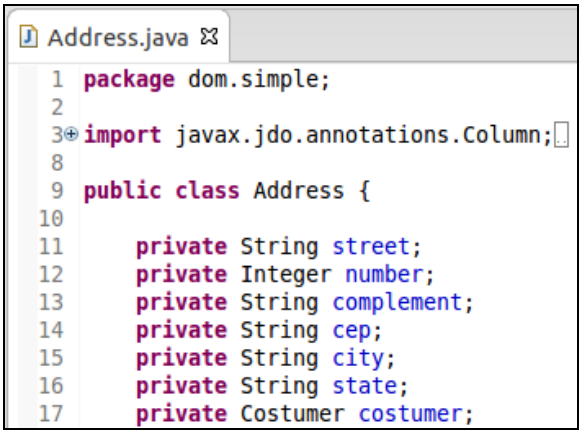


Figura 4.16: Objeto de Valor Address

Na Figura 4.17 pode-se visualizar um Serviço para geração de relatórios. No exemplo proposto é possível gerar três tipos de relatórios: de filmes por categoria, de filmes por gênero e de locações por cliente. Cada método realiza uma operação bem definida e não guarda estados.

A Figura 4.18 mostra a Entidade `Rent`. Esta Entidade também é considerada um Agregado pois contém outras Entidades do modelo de domínio. Neste caso, são armazenadas as Entidades `Costumer`, que representa um cliente, e uma lista de `RentItem`, que representa itens de locação. Criar esse conjunto de objetos associados auxilia na preservação da integridade do modelo de domínio pois qualquer alteração que se faça no `Costumer` ou no `RentItem` refletirá para o Agregado `Rent`.

Reports.java

```java
package dom.simple;

import java.util.List;

@DomainService(menuOrder = "70")
public class Reports {

    @Inject
    DomainObjectContainer container;

    @MemberOrder(sequence = "1")
    public List<Movie> moviesByCategory(final @Named("Category") Category category) {
        return container.allMatches(Movie.class, Movie.allByCategory(category));
    }

    @MemberOrder(sequence = "2")
    public List<Movie> moviesByGenre(final @Named("Genre") Genre genre) {
        return container.allMatches(Movie.class, Movie.allByGenre(genre));
    }

    @MemberOrder(sequence = "3")
    public List<Rent> rentsByCostumer(final @Named("Costumer") Costumer costumer) {
        return container.allMatches(Rent.class, Rent.allByCostumer(costumer));
    }
}
```

Figura 4.17: Serviço de geração de relatórios

Rent.java

```java
package dom.simple;

import java.util.Calendar;

public class Rent implements Comparable<Rent> {

    private Long id;
    private Date rentDate;
    private Date expectedReturnDate;
    private Date actualReturnDate;
    private Costumer costumer;
    private Set<RentItem> rentItems = new HashSet<RentItem>();
    private Double totalPrice;
```

Figura 4.18: Agregado Rent

Pode ser visto na Figura 4.19 um Repositório para Entidades `DVD`. Repositórios são responsáveis por gerenciar o ciclo de vida dos objetos do modelo de domínio. Este Repositório contém os métodos de criação e listagem dos DVD's. Por padrão, os objetos são armazenados em um banco de dados em memória, mas isso pode ser configurado no *framework Apache Isis*.

DVDs.java

```
1  package dom.simple;
2
3  import java.util.List;
13
14 @DomainService(menuOrder = "40", repositoryFor = DVD.class)
15 public class DVDs {
16
17     @Inject
18     DomainObjectContainer container;
19
20     @ActionSemantics(Of.SAFE)
21     @MemberOrder(sequence = "1")
22     public List<DVD> listAll() {
23         return container.allInstances(DVD.class);
24     }
25
26     @MemberOrder(sequence = "2")
27     public DVD create(final @Named("Movie") Movie movie) {
28         final DVD dvd = container.newTransientInstance(DVD.class);
29         dvd.setStatus(true);
30         dvd.setMovie(movie);
31         container.persistIfNotAlready(dvd);
32         container.flush();
33         return dvd;
34     }
35 }
```

Figura 4.19: Repositório de Entidades DVD

Todo o modelo de domínio, com seus blocos de construção, estão agrupados em um mesmo módulo, ou seja, estão em um mesmo pacote Java. Os testes unitários estão em outro pacote. Esse conceito de módulos do DDD permite que sejam criadas diferentes visões que façam sentido, o que auxilia na organização do projeto.

4.4 Camada de visualização dos dados

O foco do *framework Apache Isis* é o modelo de domínio, então toda a complexidade da aplicação está no módulo `dom`, como explicado anteriormente. Este módulo contém entidades que podem ser assinaladas com anotações. Tais anotações estão relacionadas à persistência e à interface de usuário. Para persistência, o *Apache Isis* usa, por padrão, o *Java Data Objetcs API* (JAVA DATA OBJECTS, 2014) e para interface de usuário utiliza suas próprias anotações, que serão explicadas com mais detalhes a seguir.

Uma anotação importante é `@MemberOrder`. Essa anotação define a ordem dos membros de uma classe em um formulário. Também define a ordem das ações, caso seja usada em um Serviço. O título do formulário pode ser especificado usando a anotação `@Title`. O uso dessas duas anotações é mostrado na Figura 4.20, e o resultado pode ser visto na Figura 4.21. As figuras que serão mostradas estão relacionadas com a Entidade `Category`, que representa a categoria de um filme. Essa entidade contém um identificador, o nome da categoria e o preço do filme pertencente a esta categoria.

```
@MemberOrder(sequence="1")
public Long getId() {
    return this.id;
}

@MemberOrder(sequence="2")
@Title(sequence="1")
public String getName() {
    return this.name;
}

@MemberOrder(sequence="3")
public Double getPrice() {
    return this.price;
}
```

Figura 4.20: Anotações @MemberOrder e @Title na Entidade Category

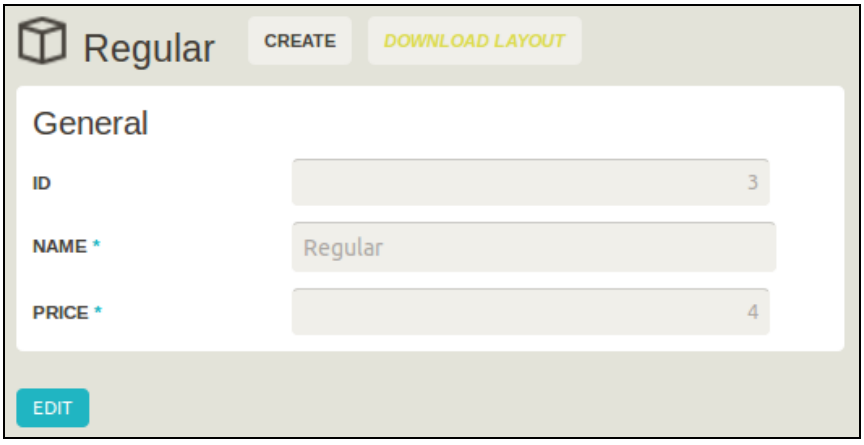


Figura 4.21: Resultado do uso das anotações @MemberOrder e @Title

Outra anotação bastante importante é `@Hidden`. Ela é usada para esconder membros ou ações da interface de usuário. Para mudar o nome de membros ou parâmetros na interface é possível usar a anotação `@Named`. Caso esta última não

seja usada, o nome dos membros e das ações serão mostradas como descritas nas classes.

Por fim, a anotação `@Bounded` é usada em uma classe para definir que esta pode ser um campo de seleção de objetos em formulários de outras classes, ou seja, uma lista com valores já definidos para seleção. Essa anotação é muito usada para fazer o relacionamento entre entidades, como é o caso das Entidades `Category`, `Genre` (que representa o gênero do filme) e `Movie` no exemplo proposto. Essa relação será mostrada abaixo. Na Figura 4.22 e 4.23 são mostrados exemplos da utilização da anotação `@Bounded` e a Figura 4.24 exibe o resultado na interface de usuário.

```
@Bounded
public class Category implements Comparable<Category> {

    private Long id;
    private String name;
    private Double price;
```

Figura 4.22: Entidade Category com a anotação @Bounded

```
@Bounded
public class Genre implements Comparable<Genre> {

    private Long id;
    private String name;
    private Integer minAge;
```

Figura 4.23: Entidade Genre com a anotação @Bounded

Figura 4.24: Formulário com os campos Categoria e Gênero como listas de seleção

O *framework Apache Isis* possui muitas outras anotações para personalizar a interface gráfica das aplicações, porém foram apresentadas apenas as anotações utilizadas no desenvolvimento do exemplo proposto.

Para gerar automaticamente a camada de visualização dos dados, o *framework* também utiliza algumas definições do modelo de domínio. Será mostrado o Serviço relativo aos filmes a fim de exemplificar como o *Apache Isis* realiza esse procedimento. Para definir que uma classe será um Serviço utiliza-se a anotação `@DomainService`. Essa anotação tem dois atributos: `menuOrder` e `repositoryFor`. O primeiro define a ordem que essa ação é mostrada no menu da página e o segundo define que este Serviço é responsável pelo repositório das Entidades `Movie`. O Serviço definido para os filmes tem dois métodos. O método `listAll` retorna uma lista contendo todos os filmes cadastrados. Por consequência, o *framework* exibe uma tabela com todos os filmes. O método `create` é responsável por cadastrar novos filmes e por isso possui parâmetros para o cadastro. Dessa forma o *framework* cria um formulário com um campo para cada parâmetro do método, como pode ser visto acima, na Figura 4.24. O código do Serviço descrito pode ser visto na Figura 4.25:

Movies.java

```java
package dom.simple;

import java.util.List;

@DomainService(menuOrder = "30", repositoryFor = Movie.class)
public class Movies {

    @Inject
    DomainObjectContainer container;

    @ActionSemantics(Of.SAFE)
    @MemberOrder(sequence = "1")
    public List<Movie> listAll() {
        return container.allInstances(Movie.class);
    }

    @MemberOrder(sequence = "2")
    public Movie create(final @Named("Title") String title,
                        final @Named("Year") Integer year,
                        final @Named("Category") Category category,
                        final @Named("Genre") Genre genre) {
        final Movie movie = container.newTransientInstance(Movie.class);
        movie.setTitle(title);
        movie.setYear(year);
        movie.setCategory(category);
        movie.setGenre(genre);
        container.persistIfNotAlready(movie);
        container.flush();
        return movie;
    }
}
```

Figura 4.25: Serviço para criação e listagem de filmes

A Figura 4.26 mostra o menu criado automaticamente pelo *framework Apache Isis*. Pode-se notar que cada item do menu segue a ordem definida pelo atributo `menuOrder` de cada Serviço. Além disso, é possível visualizar que cada método criado dentro da classe de Serviço se torna um subitem no menu e é tratado como uma ação que o usuário pode invocar.

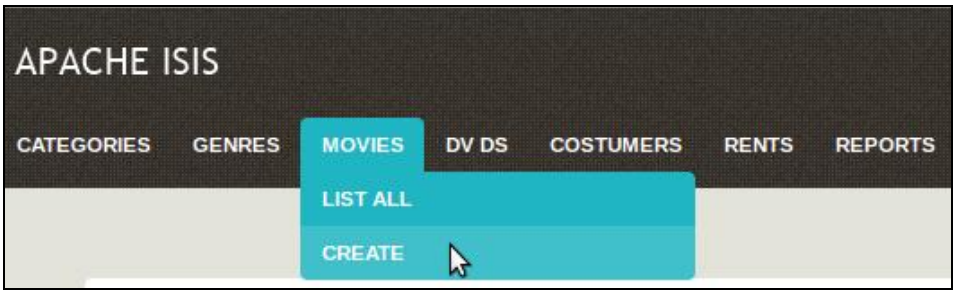


Figura 4.26: Menu criado pelo framework Apache Isis

4.5 Considerações Finais

Este capítulo mostrou como aplicar a abordagem ágil *Domain-Driven Design* na criação de aplicações Java, por meio do *framework Apache Isis*. Em conjunto ao desenvolvimento, também foram mostradas a utilização de técnicas de teste, a fim de garantir qualidade ao produto final.

Para iniciar o desenvolvimento, observou-se a importância da definição e da elaboração do modelo de domínio, que é a base para toda a implementação da aplicação. Em particular, notou-se que é essencial empregar a Linguagem Ubíqua para que se tenha uma boa comunicação entre todos os envolvidos no projeto e um claro entendimento de cada um.

Também foram apresentados os testes criados com cada técnica de teste pesquisada. Para cada uma, foi exibido um passo a passo de como os testes devem ser implementados. É evidente que com o uso de TDD é possível melhorar a qualidade de um software, garantindo que funcionalidades estejam implementadas de acordo com o esperado. Porém, com o uso do BDD em conjunto com o DDD para validar comportamentos esperados no modelo de domínio, notou-se que a integração é favorecida pela criação de cenários de teste utilizando a Linguagem Ubíqua.

Por fim, constatou-se a eficácia ao usar o *Apache Isis*, em relação à geração automática da camada de visualização de dados. Foi mostrado que com o uso das anotações providas pelo *framework*, é possível customizar a interface de usuário da aplicação criada.

No próximo Capítulo será apresentado o protótipo de gerador de testes e cenários para projetos de software que empregam DDD, por meio do *framework Apache Isis* e testes utilizando BDD.

Capítulo 5

GERADOR AUTOMÁTICO DE TESTES

Este capítulo apresenta o desenvolvimento e o funcionamento do protótipo do gerador de testes e cenários para projetos de software que utilizam DDD, por meio do framework *Apache Isis e testes com BDD.*

5.1 Considerações Iniciais

Com a implementação do exemplo proposto e a utilização da técnica *Behaviour-Driven Development* em conjunto com a abordagem *Domain-Driven Design*, por meio do *framework Apache Isis*, notou-se a possibilidade de criar, devido a similaridade da estrutura das classes e cenários de teste do BDD, uma ferramenta capaz de gerar testes específicos de maneira automática.

Para que os testes possam ser gerados, o desenvolvedor precisa implementar métodos de validação dos atributos das entidades. Esses métodos são invocados pelo *framework Apache Isis* para a validação da entrada de dados nas interfaces de usuário. Dessa forma, o gerador de testes se beneficia dessa característica do *Apache Isis*, para criar de forma genérica e automática, testes de validação dos atributos das entidades. Esses testes são criados utilizando os conceitos de BDD.

Dessa forma, este capítulo apresenta na Seção 5.2 as ideias e conceitos que impulsionaram a criação do gerador automático de testes e cenários. Na Seção 5.3 é mostrada parte da implementação do gerador. Também são apresentados os trechos de código mais relevantes. A Seção 5.4 ilustra um exemplo de geração automática de testes e cenários com o uso do gerador, tendo por base o modelo de

domínio do estudo de caso definido: aplicação web de Vídeo Locadora. Por fim, na Seção 5.5 são tecidas as considerações finais deste Capítulo.

5.2 Conceituando o Gerador Automático de Testes

Considerando que um teste feito com BDD sempre tem a mesma estrutura, ou seja, sempre há uma entrada de dados (*Given*), um processamento daquela entrada (*When*) e por fim, uma comparação entre o comportamento esperado e o resultado obtido (*Then*), foi observado que era possível gerar automaticamente um cenário padrão e uma classe de teste para validações específicas no modelo de domínio. Dessa forma, uma aplicação desenvolvida com o *framework Apache Isis* poderia ter validações de cada atributo de suas entidades.

A geração dos testes é baseada em um método que o *Apache Isis* permite que seja criado para cada atributo de uma entidade. Esse método é chamado `validate<atributo>`. Por exemplo, caso exista um atributo `nome` em uma entidade, é possível criar um método chamado `validateNome` que contém a lógica que o desenvolvedor julga apropriada para validar aquele nome. Esse método é invocado automaticamente pelo *Apache Isis* na aplicação web, gerada para exibir o modelo de domínio. Tal método deve sempre retornar uma `String`, que representa uma mensagem de erro, caso o atributo não passe na validação. Nenhuma configuração adicional é necessária para que o *framework* use esse método, apenas seguir o padrão de nomenclatura e retornar uma `String` é suficiente.

Por isso, o gerador automático proposto por este trabalho utiliza essa característica do *framework Apache Isis* para permitir que o desenvolvedor crie métodos de validação de dados, que serão usados para gerar automaticamente testes com a abordagem BDD.

O protótipo desenvolvido funciona de maneira simples. Ele possui um arquivo de configuração que contém definições de caminhos de entrada e saída. O caminho de entrada define onde está o pacote do modelo de domínio que será usado. Os caminhos de saída definem onde serão geradas as classes e os cenários de teste. A partir desses dados, o gerador pode analisar as classes do modelo de domínio e obter informações sobre elas, como por exemplo, o nome da classe, atributos e

métodos. Assim, o gerador é capaz de criar um cenário padrão e uma classe de teste para validar cada atributo. A estrutura do gerador automático pode ser vista na Figura 5.1:

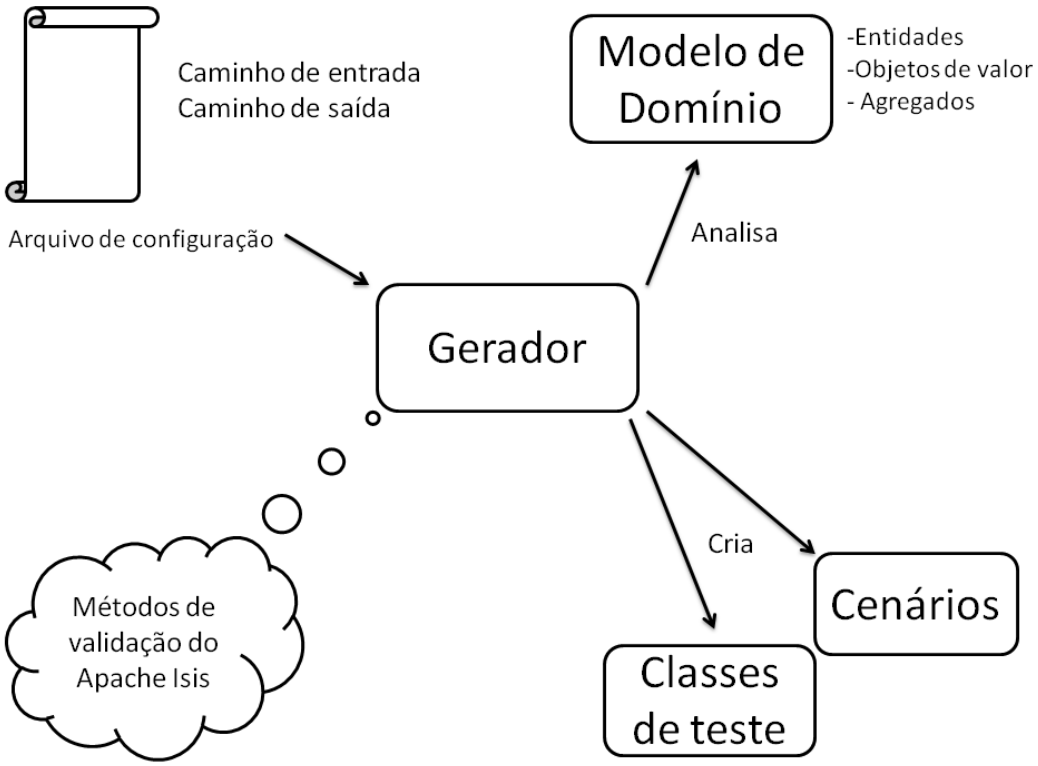


Figura 5.1: Estrutura do gerador de testes

5.3 Implementação do Gerador

O protótipo do gerador automático de testes e cenários foi implementado em linguagem Java em conjunto com a ferramenta Maven, que auxilia a construção e o gerenciamento de dependências. Para facilitar o entendimento do código e futuras manutenções, o projeto foi dividido em pacotes, cada qual com uma responsabilidade bem definida. Foram criados, então, seis pacotes. O pacote `main` é o principal, nele está a classe `Main` que é a responsável por executar o gerador. O pacote `config` contém a classe que armazena todas as configurações do gerador. Essas configurações são mantidas em um arquivo texto de propriedades que pode ser alterado manualmente pelo usuário do gerador. O pacote `exception` possui

exceções personalizadas para a execução do gerador. Caso ocorra algum erro, mensagens intuitivas serão apresentadas ao usuário. No pacote `model` está a classe de modelo para a geração automática dos testes e cenários. Ela armazena as informações obtidas das entidades do modelo de domínio alvo. Por fim, os pacotes `parser` e `builder` possuem de fato a lógica para obter informações do modelo de domínio e criar as classes de teste e os cenários, respectivamente. A Figura 5.2 mostra a estrutura do projeto descrita acima.

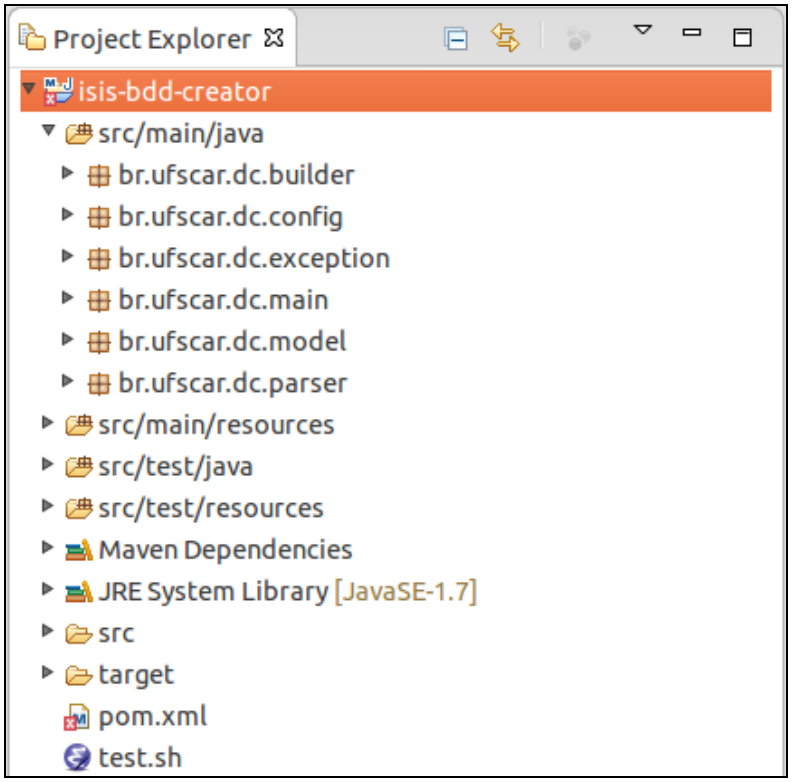


Figura 5.2: Estrutura do projeto do gerador automático de testes

O pacote `parser` tem grande importância para todo o funcionamento do gerador porque ele é o responsável por obter as informações necessárias do modelo de domínio para a geração dos testes. Para obter essas informações, é usada uma biblioteca da Google, chamada javaparser (JAVAPARSER, 2007). Essa biblioteca é capaz de analisar um arquivo de código Java, ou seja, uma classe e extrair informações como nome da classe, os atributos e métodos que ela contém e até comentários e documentação de código. A partir dessas informações coletadas é que o pacote `builder` consegue gerar os arquivos de teste e cenário automaticamente.

A Figura 5.3 ilustra um trecho de código da classe `ClassParser`. Nela é possível ver a utilização simplificada da biblioteca javaparser.

ClassParser.java

```
49    public List<ClassModel> parseClasses()
50    {
51        logger.info("Iniciando parser das classes...");
52
53        File[] files = this.domInputDirectory.listFiles(new ClassFilenameFilter());
54
55        ClassModel model = null;
56        List<ClassModel> models = new ArrayList<>();
57
58        FileInputStream in = null;
59        CompilationUnit cu = null;
60
61        for (File file : files)
62        {
63            logger.debug("Fazendo parser do arquivo: " + file.getName());
64
65            try
66            {
67                in = new FileInputStream(file.getAbsoluteFile());
68                cu = JavaParser.parse(in);
69
70                model = parseClassModel(cu);
71
72                if (!model.getMapMembers().isEmpty()) {
73                    models.add(model);
74                }
75            }
```

Figura 5.3: Trecho de código que realiza o parser das entidades

O trecho de código mostrado acima é bem simples. A partir de uma lista de arquivos, que são as entidades do modelo de domínio, percorre-se cada um realizando a operação de *parser* da biblioteca javaparser (linha 68). As informações coletadas são, finalmente, armazenadas em uma lista para que possam ser usadas posteriormente pelo `builder`.

O pacote `builder`, também de suma importância para o projeto, é o que de fato cria os arquivos de testes e cenários, a partir das informações obtidas pelo `parser`. Para criar os arquivos, não é necessário nenhuma biblioteca externa, ao contrário do pacote `parser`. A Figura 5.4 mostra um trecho de código da classe `ClassBuilder`.

ClassBuilder.java

```
91          for (Entry<String, String> entry : model.getMapMembers().entrySet())
92          {
93              member = entry.getKey();
94              type = entry.getValue();
95
96              sb.append("\t@Given(\"^Eu tenho o valor \\\"([^\\\"]*)\\\" para " + member + "$\")\n");
97              sb.append("\tpublic void euTenhoOValorNPara" + ClassModel.toCapital(member)
98                      + "(String value) throws Throwable {\n");
99              sb.append("\t\t" + type + " " + member + " = " + getConverter(type) + ";\n");
100             sb.append("\t\tputVar(\"" + type + "\", \"" + member + "\", " + member + ");\n");
101             sb.append("\t}\n\n");
102
103             sb.append("\t@When(\"^Eu valido " + member + " de " + model.getClassName() + "$\")\n");
104             sb.append("\tpublic void euValido" + ClassModel.toCapital(member) + "De"
105                     + model.getClassName() + "() throws Throwable {\n");
106             sb.append("\t\t" + model.getClassName() + " " + model.getClassName().toLowerCase()
107                     + " = new " + model.getClassName() + "();\n");
108             sb.append("\t\tString mensagemErro = " + model.getClassName().toLowerCase()
109                     + ".validate" + ClassModel.toCapital(member) + "(getVar(\"" + type + "\", "
110                     + "\"" + member + "\", " + type + ".class));\n");
111             sb.append("\t\tputVar(\"String\", \"mensagemErro\", mensagemErro);\n");
112
113             sb.append("\t}\n\n");
114         }
115
116         sb.append("\t@Then(\"^Eu recebo a mensagem de erro \\\"([^\\\"]*)\\\"$\")\n");
117         sb.append("\tpublic void euReceboAMensagemDeErro(String mensagemErro) throws "
118                 + "Throwable {\n");
119         sb.append("\t\tAssert.assertEquals(mensagemErro, getVar(\"string\", \"mensagemErro\", "
120                 + "String.class));\n");
121         sb.append("\t}\n");
122         sb.append("}\n");
```

Figura 5.4: Trecho de código que cria a classe de teste

Como é possível ver no trecho de código apresentado acima, a classe `ClassBuilder` percorre a lista obtida pelo `parser` e concatena todo o texto necessário para a criação da classe de teste, com suas anotações `@Given`, `@When` e `@Then`.

5.4 Exemplo de Geração Automática de Testes no Domínio de Vídeo Locadora

O funcionamento do gerador automático de testes e cenários será apresentado nesta seção por meio de um exemplo: a entidade `Categoria` possui um atributo `preço`, que determina o preço de todos os filmes pertencentes àquela categoria. Dessa forma, há uma validação neste atributo, que o impede de receber valores menores que zero. Consequentemente, ao executar o gerador neste modelo de domínio, ele irá criar uma classe de teste e um cenário padrão para a validação

do atributo `preço`. Na Figura 5.5 pode ser visto o método de validação do atributo `preço`, conforme explicado acima:

```
public String validatePrice(Double price) {
    if (price < 0) {
        return "O preço não pode ser negativo!";
    } else {
        return null;
    }
}
```

Figura 5.5: Método de validação do atributo preço

A Figura 5.6 mostra a estrutura do projeto proposto de vídeo locadora, que será usado para exemplificar a geração automática de testes.

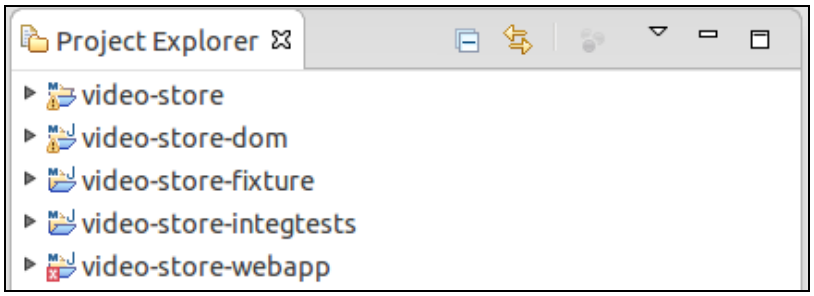


Figura 5.6: Estrutura do projeto de vídeo locadora

A estrutura do projeto é importante pois, para que o gerador execute corretamente, é preciso definir o caminho de entrada do modelo de domínio e os caminhos de saída para a classe de teste e para o cenário de teste. A Figura 5.7 exibe o arquivo de configuração com os caminhos de entrada e saída para o projeto de vídeo locadora.

config.properties

```
# Configuration file
dom-input-directory =
/home/eloisa/Projects/video-store/dom/src/main/java/dom/simple/

class-output-directory =
/home/eloisa/Projects/video-store/integtests/src/test/java/integration/glue/simple/

scenario-output-directory =
/home/eloisa/Projects/video-store/integtests/src/test/java/integration/specs/simple/
```

Figura 5.7: Arquivo de configuração do gerador automático de testes

O gerador não possui interface gráfica, portanto deve ser executado por meio do comando `java -jar isis-bdd-creator-0.0.1-SNAPSHOT.jar` em um terminal. Também é necessário que o Java esteja instalado.

Após executar o gerador de testes, a classe e o cenário são criados em seus respectivos caminhos, como determinado no arquivo de configuração. A Figura 5.8 mostra os arquivos gerados na estrutura do projeto de vídeo locadora.

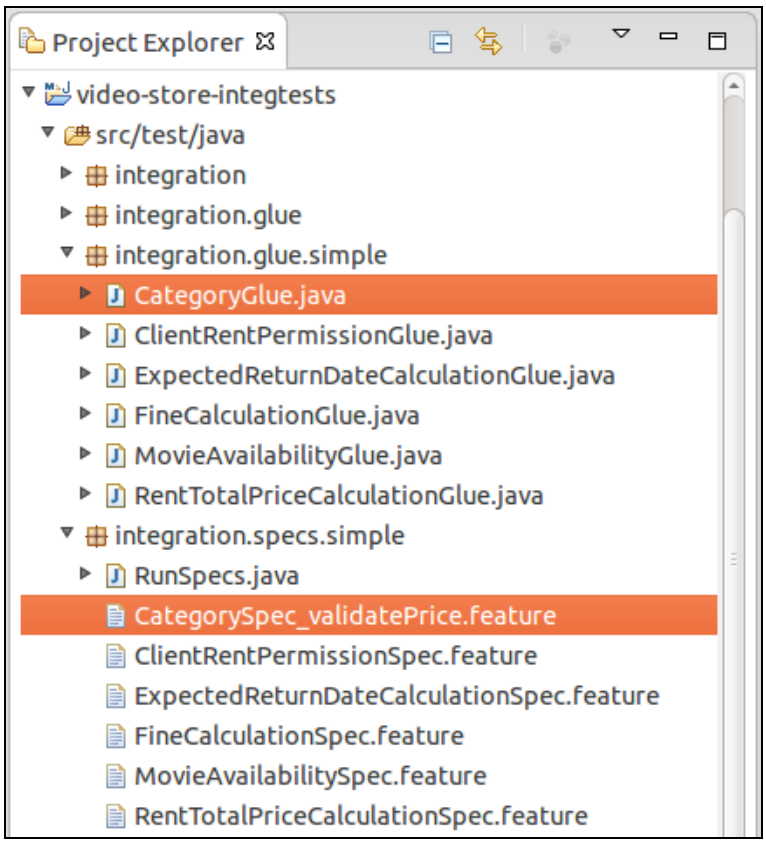


Figura 5.8: Classe e cenário de teste gerados automaticamente

A classe de teste gerada está completa e funcional para executar qualquer cenário de validação do atributo `preço`. Vários cenários podem ser executados na mesma classe, desde que sigam o padrão de nomenclatura dos métodos. O cenário gerado não está completo, ele é apenas um *template*, portanto deve ser completado pelo desenvolvedor, de acordo com as regras de negócio. Além disso, outros cenários podem ser criados manualmente caso o desenvolvedor julgue necessário. A classe de teste e o cenário padrão gerados para a validação do atributo `preço` da entidade `Categoria` podem ser vistos nas Figuras 5.9 e 5.10, respectivamente:

CategoryGlue.java

```java
package integration.glue.simple;

import org.apache.isis.core.specsupport.specs.CukeGlueAbstract;

public class CategoryGlue extends CukeGlueAbstract {

    @Given("^Eu tenho o valor \"([^\"]*)\" para price$")
    public void euTenhoOValorNParaPrice(String value) throws Throwable {
        Double price = Double.parseDouble(value);
        putVar("Double", "price", price);
    }

    @When("^Eu valido price de Category$")
    public void euValidoPriceDeCategory() throws Throwable {
        Category category = new Category();
        String mensagemErro = category.validatePrice(getVar("Double", "price", Double.class));
        putVar("String", "mensagemErro", mensagemErro);
    }

    @Then("^Eu recebo a mensagem de erro \"([^\"]*)\"$")
    public void euReceboAMensagemDeErro(String mensagemErro) throws Throwable {
        Assert.assertEquals(mensagemErro, getVar("string", "mensagemErro", String.class));
    }
}
```

Figura 5.9: Classe de teste

CategorySpec_validatePrice.feature

```
Feature: Validar price de Category

    @integration
    Scenario:
        Given Eu tenho o valor "" para price
        When  Eu valido price de Category
        Then  Eu recebo a mensagem de erro ""
```

Figura 5.10: Cenário de teste padrão

Por fim, para completar o cenário padrão, foram adicionados o nome do cenário e os valores necessários de acordo com a regra de negócio em questão. Assim, o cenário foi denominado "Categoria com preço negativo". Como entrada de dados foi utilizado o valor "-1" e como resultado esperado a mensagem de erro "O preço não pode ser negativo!". Este cenário completo pode ser visto na Figura 5.11:

CategorySpec_validatePrice.feature

```
Feature: Validar price de Category

    @integration
    Scenario: Categoria com preço negativo
        Given Eu tenho o valor "-1" para price
        When  Eu valido price de Category
        Then  Eu recebo a mensagem de erro "O preço não pode ser negativo!"
```

Figura 5.11: Exemplo de cenário de teste completo

De acordo com o método de validação do preço da categoria, este cenário será executado com sucesso porque ao validar um valor negativo para o atributo `preço`, a mensagem de erro "O preço não pode ser negativo!" será retornada.

Como explicado anteriormente, novos cenários de teste podem ser criados manualmente pelo desenvolvedor. Essa prática visa cobrir o maior número possível de fluxos da funcionalidade que está sendo testada. A seguir, a Figura 5.12 mostra outro cenário de teste.

CategorySpec_validatePrice.feature

```
1  Feature: Validar price de Category
2
3      @integration
4      Scenario: Categoria com preço negativo
5          Given Eu tenho o valor "-1" para price
6          When  Eu valido price de Category
7          Then  Eu recebo a mensagem de erro "O preço não pode ser negativo!"
8
9      @integration
10     Scenario: Categoria com preço positivo
11         Given Eu tenho o valor "2" para price
12         When  Eu valido price de Category
13         Then  Eu recebo a mensagem de erro ""
```

Figura 5.12: Exemplo com mais de um cenário de teste

O novo cenário criado verifica se não há mensagem de erro quando o preço de uma categoria for positivo. Também é esperado que esse cenário seja executado com sucesso, de acordo com a regra de negócio da validação de preço.

5.5 Considerações Finais

Este capítulo apresentou os conceitos e características dos *frameworks Apache Isis* e *Cucumber*, utilizados na integração da abordagem *Domain-Driven Design* e da técnica de teste *Behaviour-Driven Development*, que possibilitaram a criação do gerador automático de testes e cenários. Também foram mostrados alguns trechos de código do gerador, com o objetivo de explicar como ele foi implementado, assim como um exemplo de utilização no modelo de domínio do estudo de caso deste trabalho.

Notou-se que o gerador automático de testes e cenários é capaz de elaborar corretamente testes de validação dos atributos das entidades de qualquer modelo de domínio. Dessa forma, erros simples são encontrados e corrigidos rapidamente. A criação automática dos testes agiliza essa etapa do desenvolvimento do software, portanto é um fator muito importante no cenário atual.

Em conjunto com o *Apache Isis*, que é indicado para a prototipação rápida de aplicações pois gera automaticamente a camada de visualização de dados de um modelo de domínio bem definido, o uso do gerador automático de testes potencializa a criação desses protótipos, porque adiciona um grau de validação dos atributos das entidades, garantindo um aumento na qualidade.

O próximo Capítulo apresenta as conclusões do trabalho realizado, as contribuições obtidas, as respectivas limitações e possíveis trabalhos futuros.

Capítulo 6

CONCLUSÕES

Este capítulo apresenta as conclusões da pesquisa realizada, evidenciando as contribuições, as limitações e possíveis trabalhos futuros.

6.1 Considerações Finais

Esta pesquisa apresentou um estudo de abordagens e técnicas ágeis de desenvolvimento e teste de software, essenciais para atender o mercado nos dias atuais. Também propôs a integração do *Domain-Driven Design*, separadamente, com o *Test-Driven Development* e com o *Behaviour-Driven Development*, a fim de suprir a lacuna relativa a testes, existente no DDD. Essa integração foi realizada com a implementação de um estudo de caso, por meio do *framework Apache Isis*. Além disso, foi criado um gerador automático de testes e cenários com o intuito de agilizar a fase de testes de projetos que empregam o *Apache Isis* em conjunto com o BDD.

Por meio deste trabalho, pode-se observar que é possível integrar técnicas de teste como *Test-Driven Development* e *Behaviour-Driven Development* com o *Domain-Driven Design* para aumentar a qualidade de um software. Cada técnica de teste tem suas vantagens e desvantagens, porém ao integrar o BDD com o DDD é possível utilizar a linguagem ubíqua na criação dos cenários. Essa característica presente no BDD é um grande diferencial em relação ao TDD, pois permite que qualquer membro envolvido no projeto seja capaz de entender as funcionalidades do sistema e consequentemente os testes, uma vez que estes são escritos em linguagem narrativa e fazem parte da documentação do projeto.

Como resultado da pesquisa realizada e da implementação do estudo de caso proposto, foi criado o gerador automático de testes. Este agiliza a fase de testes de um projeto *Apache Isis*, por meio da validação de atributos das entidades do modelo de domínio. Com seu uso no próprio estudo de caso, foi possível rapidamente detectar erros que poderiam não serem notados ou apenas serem encontrados com a evolução do projeto.

A seguir são apresentadas as contribuições e limitações deste trabalho, assim como os possíveis trabalhos futuros.

6.2 Contribuições e Limitações

As principais contribuições deste trabalho são:

- O levantamento da carência de técnicas de teste na abordagem de desenvolvimento *Domain-Driven Design*;
- A comparação entre as técnicas de teste TDD e BDD;
- A implementação de um estudo de caso para exemplificar a integração de cada técnica de teste pesquisada com a abordagem DDD;
- A exemplificação do uso do *framework Apache Isis* para dar suporte à implementação do estudo de caso;
- A criação do gerador automático de testes e cenários para a técnica *Behaviour-Driven Development*.

As principais limitações deste trabalho são:

- A implementação do estudo de caso proposto não foi realizado por uma equipe de desenvolvimento, por motivo de viabilidade da pesquisa;
- A implementação do estudo de caso proposto ocorreu com o uso de *frameworks* específicos: o *Apache Isis* e o *Cucumber*;
- A utilização do gerador automático de testes está condicionada ao uso dos *frameworks Apache Isis* e *Cucumber*.

6.3 Trabalhos Futuros

A seguir são listadas as atividades que poderiam dar continuidade a este trabalho:

- Aplicar um estudo de caso, similar ao realizado nesta pesquisa, porém com uma equipe de desenvolvimento;
- Utilizar outros *frameworks* para a abordagem DDD e para a técnica BDD;
- Ampliar a capacidade do gerador automático para que ele possa criar testes a partir de outros *frameworks*.

6.4 Artigo Publicado

Durante este projeto de mestrado foi publicado o seguinte artigo científico:

- Eloisa Santos, Delano Beder, Rosângela Penteado. **A study of test techniques for integration with Domain-Driven Design**. The International Conference on Information Technology: New Generations (ITNG). Las Vegas, Nevada, USA. 12-15 de abril de 2015. Qualis: B1.

REFERÊNCIAS

ACTIFSOURCE. *Actifsource: Build your domain specific development tool that turns your software specification into running code*. 2013. Disponível em: <http://www.actifsource.com/>. Acessado em: Março, 2015.

APACHE. *The Apache Software Foundation*. 1999. Disponível em: <http://www.apache.org/>. Acessado em: Janeiro, 2014.

APACHE ISIS. *Apache Isis is a framework for rapidly developing domain-driven apps in Java*. 2010. Disponível em: <http://isis.apache.org/>. Acessado em: Janeiro, 2014.

APACHE SHIRO. *Apache Shiro: Welcome to Apache Shiro*. 2008. Disponível em: < http://shiro.apache.org/>. Acessado em: Janeiro, 2014.

APACHE WICKET. *Apache Wicket: Welcome to Apache Wicket*. 2014. Disponível em: <http://wicket.apache.org/>. Acessado em: Dezembro, 2014.

BASTOS, A.; RIOS, E.; CRISTALLI, R.; MOREIRA, T. Base de conhecimento em teste de software. São Paulo: Martins, 2012.

BECK, K. *Test Driven Development: By Example*. Ed. Addilson-Wesley, p. 240. 2012.

BEDER, D. M., *Engenharia Web: uma abordagem sistemática para o desenvolvimento de aplicações web*. p. 118, 2012. ISBN 978-85-7600-2901-1.

BERNARDO, P. C.; KON, F. *A Importância dos Testes Automatizados: Controle ágil, rápido e confiável de qualidade*. pp. 54-57, 2008. Disponível em <http://www.ime.usp.br/~kon/papers/EngSoftMagazine-IntroducaoTestes.pdf>. Acessado em: Fevereiro, 2014.

BURNSTEIN, I. *Pratical software testing: a process-oriented approach* [S.1.]: Springer, 2003. ISBN 9780387951317.

CHELIMSKY, D.; ASTELS, D.; DENNIS, Z.; HELLESOY, A.; HELMKAMP, B.; NORTH, D. *The RSpec Book Beta: Behaviour Driven Development with RSpec, Cucumber, and Friends*. Ed The Pragmatic Programmers, p. 305, 2009.

CUBICWEB. *CubicWeb: The Semantic Web is a construction game*. 2008. Disponível em: <http://www.cubicweb.org/>. Acessado em: Março, 2015.

CUCUMBER. *Cucumber behaviour driven development with elegance and joy*. 2014. Disponível em: <http://cukes.info/>. Acessado em: Janeiro, 2014.

CUKIER, D. *Introdução a Domain Driven Design*. 2010. Disponível em: < http://www.agileandart.com/2010/07/16/ddd-introducao-a-domain-driven-design/>. Acessado em: Janeiro, 2014.

DELAMARO, M. E.; MALDONADO, J. C.; JINO, M. *Introdução ao Teste de Software*. [S.l.: s.n.], 2007. ISBN 9788535226348. ERDOGMUS, H.; MORISIO, M.; TORCHIANO, M. On the effectiveness of the test-first approach to programming. *Software Engineering, IEEE Transactions on*, vol.31, no.3, pp.226,237, March 2005.

EVANS, E. *Domain Driven Design: Tackling Complexity in the Heart of Software*. Ed. Addilson-Wesley, 2003.

FABBRI, S; VICENZI, A. M. R; MALDONADO, J. C. Teste de software In: Introdução ao Teste de software Delamaro, M.E; Maldonado, J.C; Jino, M. 2007.

FILHO, J. I. F; SILVA, O. A. *Desenvolvimento Orientado a Testes de Aceitação*. Pontifícia Universidade Católica de Goiás. Goiânia, GO. 2012.

GOUVEIA, C. C. *Teste de Integração para Sistemas Baseados em Componentes*. Dissertação (Mestrado em Engenharia de software). Universidade Federal da Paraíba. 2004.

HAYWOOD, D. *Apache Isis: Developing Domain-driven Java Apps*. 2013. Disponível em: <http://www.methodsandtools.com/PDF/mt201302.pdf>. Acessado em Janeiro, 2014.

HUMPHREY, W. S. *Managing the Software Process*. Ed. Adilson-Wesley, 1989.

JACOBSON, I. *A Resounding 'Yes' to Agile Process - But Also More*. Cutter IT Journal, vol. 5 n. 1, p. 18, 2002.

JAVA. *Java*. 1995. Disponível em: <http://www.java.com>. Acessado em: Março, 2014.

JAVAPARSER. *Java Parser and Abstract Syntax Tree*. 2007. Disponível em: <http://javaparser.github.io/javaparser/>. Acessado em: Março, 2015

JBEHAVE. *JBehave is a framework for Behaviour-Driven Development*. 2003. Disponível em: <http://jbehave.org/>. Acessado em: Março, 2015.

JOHANSEN, C. *Test-Driven JavaScript Development*. Ed. Addilson- Wesley. p. 21-30. 2011.

JUNIT. *A programmer-oriented testing framework for Java*. 2014. Disponível em: < http://junit.org/>. Acessado em: Janeiro, 2014.

KAJKO-MATTSSON, M.; LEWIS, G.A.; SIRACUSA, D.; NELSON, T.; CHAPIN, N.; HEYDT, M.; NOCKS, J.; SNEED, H. Long-term Life Cycle Impact of Agile Methodologies. *Software Maintenance*, *2006. ICSM '06. 22nd IEEE International Conference on*, vol., no., pp.422,425, Sept. 2006.

KERAMATI, H.; MIRIAM-HOSSEINABADI S. 2008. Integrating software development security activities with agile methodologies. *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*, vol., no., pp.749,754, March 31 2008-April 4 2008.

LANDRE, E.; WESENBERG, H.; OLMHEIM, J. 2007. Agile enterprise software development using domain-driven design and test first. *In* Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion *(OOPSLA '07)*. ACM, New York, NY, USA, 983-993.

LAPOLLI, F.; MOTTA, C. L. R.; CRUZ, C. M.; OLIVEIRA, C. E. T. *Modelo de desenvolvimento de objetos de aprendizagem baseados em metodologias ágeis e scaffoldings*. Revista de informática na Educação. v.18, n.2, 2010.

LAZAR, I.; MOTOGNA, S.; PÂRV, B. *Behavior Driven Development of Foundational UML Components. Eletronic Notes in Theoretical Computer Science***,** v.264, n.1, p.91-105, August 2010.

LEITNER, A.; CIUPA, I.; MEYER, B.; HOWARD, M. Reconciling Manual and Automated Testing: The AutoTest Experience. *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, vol., no., pp.261a,261a, Jan. 2007.

LUCRÉDIO, D. *Uma Abordagem Orientada a Modelos para Reutilização de Software*. 287 f. Tese (Doutorado em Ciência de Computação e Matemática Computacional). Instituto de Ciências Matemáticas e de Computação Universidade de São Paulo, São Carlos, SP. 2009.

MARTIN, R. C. Professionalism and Test-Driven Development. *Software. IEEE*, vol.24, no.3, pp.32,36, May-June 2007.

MAVEN. *Apache Maven Project*. 2002. Disponível em: <http://maven.apache.org/>. Acessado em: Março, 2014.

MINUNIT. *MinUnit: a minimal unit testing framework for C*. 1998. Disponível em: <http://www.jera.com/techinfo/jtns/jtn002.html>. Acessado em: Março, 2015.

MYERS, G. J. *et al*. *The Art of Software Testing*. 2nd. ed. Hoboken/NJ - USA: John Wiley & Sons, 2004.

NORTH, D. *Introducing Behaviour-Driven Development*. Better Software Magazine. 2006. Disponível em: <http://dannorth.net/introducing-bdd/>. Acessado em: Janeiro, 2014.

NUNIT. *NUnit is a unit-testing framework for all*. 2002. Disponível em: <http://www.nuint.org/>. Acessado em: Março, 2015.

OPENXAVA. *OpenXava: OpenXava is an AJAX Java Framework for Rapid Development of Enterprise Web Applications*. 2005. Disponível em: <http://www.openxava.org/>. Acessado em: Março, 2015.

PALERMO, J. *Guidelines for Test-Driven Development*. May, 2006. Disponível em: <http://msdn.microsoft.com/en-us/library/aa730844%28v=vs.80%29.aspx>. Acessado em: Janeiro, 2014.

PHPUNIT. *PHPUnit is a programmer-oriented testing framework for PHP*. 2001. Disponível em: <https://phpunit.de/>. Acessado em: Março, 2015.

PRESSMAN, R. S. Can Internet-Based Applications Be Engineered? IEEE Software, v. 15, n. 5, p. 104-110, set. 1998.

PRESSMAN, R. S. *Engenharia de Software: uma abordagem profissional*. 6. ed. São Paulo: McGraw-Hill, 2006.

PRESSMAN, R. S.; LOWE, D. *Engenharia Web*. Rio de Janeiro: LTC, 2009.

QUSEF, A.; BAVOTA, G.; OLIVETO, R.; DE LUCIA, A.; BINKLEY, D. SCOTCH: Test-to-code traceability using slicing and conceptual coupling*. Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, vol., no., pp.63,72, 25-30 Sept. 2011.

RAMLER, R.; KASPAR, T. Applicability and benefits of mutation analysis as an aid for unit testing. *Computing and Convergence Technology (ICCCT), 2012 7th International Conference on*, vol., no., pp.920,925, 3-5 Dec. 2012.

RESTFUL OBJECTS. *Restful Objects Specification*. 2013. Disponível em: < http://restfulobjects.org/>. Acessado em: Janeiro, 2014.

REZENDE, B. A. C. *Utilização de TDD em Projetos de Software: Estudo de Caso Acadêmico*. Monografia (Graduação em Engenharia de Software). Pontifícia Universidade Católica de Minas Gerais. 2011.

RSPEC. *Behaviour-Driven Development for Ruby*. 2005. Disponível em: <http://rspec.info/>. Acessado em: Março, 2015.

SELENIUM HQ. SeleniumHQ Browser Automation. 2012. Disponível em: <http://docs.seleniumhq.org/>. Acessado em: Fevereiro, 2014.

SHRIVASTAVA, D. P.; JAIN, R. C. Unit test case design metrics in test driven development*. Communications, Computing and Control Applications (CCCA), 2011 International Conference on*, vol., no., pp.1,6, 3-5 March 2011.

SILVEIRA, P.; SILVEIRA, G.; LOPES, S.; MOREIRA. G.; STEPPAT, N.; KUNG, F. *Introdução à Arquitetura e Design de Software: Uma visão sobre a plataforma Java*. Rio de Janeiro : Elsevier, 2012. ISBN: 978-85-352-5030-5.

SITEFANE, J. P. M. *Desenvolvimento de Software Centrado no Domínio*. Dissertação (Mestrado em Engenharia Informática e de Computadores). Instituto Superior Técnico - Universidade Técnica de Lisboa. 2007.

SLYNGSTAD, O. P. N.; LI, J.; CONRADI, R.; RØNNEBERG, H.; LANDRE, E.; WESENBERG, H. The impact of test driven development on the evolution of reusable framework of components – and industrial case study. In *The Third International Conference on Software Engineering Advances*, IEEE, pp. 214-223, 2008.

SMALLTALK. *Community and industry meet inventing the future*. 1999. Disponível em: <http://www.smalltalk.org/main/>. Acessado em: Março, 2014.

SOARES, I. *Desenvolvimento orientado por comportamento (BDD): Um novo olhar sobre TDD*. Java Magazine, Rio de Janeiro, v. 1, n. 91, 2011. ISSN 1676836-1.

SOARES, M. S. *Metodologias Ágeis Extreme Programming e Scrum para o Desenvolvimento de Software*. Universidade Presidente Antônio Carlos. Conselheiro Lafaiete-SP, 2004.

SOLÍS, C.; WANG, X. A Study of the Characteristics of Behaviour Driven Development. *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on* , vol., no., pp.383,387, Aug. 30 2011-Sept. 2 2011.

SOMMERVILLE, I. *Engenharia de Software*. 8 ed. São Paulo: Pearson Addilson-Wesley, 2007.

SPECFLOW. *Binding Business requirements to .NET code*. 2013. Disponível em: <http://www.specflow.org/>. Acessado em: Março, 2015.

STORYQ. *StoryQ is a portable (single dll), embedded BDD framework for .NET*. 2006. Disponível em: <http://storyq.codeplex.com/>. Acessado em: Março, 2015.

TELES, V. M. *Extreme Programming: Aprenda como encantar seus usuários desenvolvendo software com agilidade e alta qualidade*. São Paulo: Novatec, 2004. ISBN: 85-7522-047-0.

USSAMI, T. H. *Um estudo sobre BDD e as ferramentas Cucumber e Jbehave*. Monografia (Graduação em Engenharia de Computação). Universidade Federal de São Carlos, São Carlos, SP. 2013.

VIANA, M. C. *Desenvolvimento e o reúso de frameworks com base nas características do domínio*. (Doutorado em Ciência da Computação) - Universidade Federal de São Carlos - 2014.

VU, J.H.; FROJD, N.; SHENKEL-THEROLF, C.; JANZEN, D.S. Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project. *Information Technology: New Generations, 2009. ITNG '09. Sixth International Conference on*, vol., no., pp.229,234, 27-29 April 2009.

WESENBERG, H.; LANDRE, E.; RONNRBERG, H. 2006. Using domain-driven design to evaluate commercial off-the-shelf software. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. ACM, New York, NY, USA, 824-829.

WILLIAMS, L.; KUDRJAVETS, G.; NAGAPPAN, N. On the Effectiveness of Unit Test Automation at Microsoft. *Software Reliability Engineering, 2009. ISSRE '09. 20th International Symposium on*, vol., no., pp.81,89, 16-19 Nov. 2009.

WILLIAMS, L.; MAXIMILIEN, E. M.; VOUK, M. Test-Driven development as a defect reduction practice. In *14th International Symposium on Software Reliability Engineering*, IEEE, pp.1-12, 2003.

APÊNDICE A

DOCUMENTO DE REQUISITOS

Tabela de Versionamento

Tabela A.1: Tabela de versionamento do documento de requisitos

Versão	Data	Comentários
1.0	02/09/2014	Criação da primeira versão do documento de requisitos do sistema de locadora de filmes.
1.1	13/09/2014	Correções no diagrama de classes do sistema: adição de identificador para a classe `RentItem`, correções de nomes e tipos de alguns atributos.
1.2	20/09/2014	Novo requisito de endereço para o cadastro de clientes adicionado ao modelo.
1.3	30/09/2014	Novo requisito de verificação de disponibilidade de um DVD para locação adicionado ao modelo.

Requisitos do Sistema

1. A locadora realiza o aluguel de DVDs de filmes que podem ter uma ou mais cópias. Cada filme possui um identificador, título e ano.

2. Cada DVD possui um identificador, um status de disponibilidade e o filme nele contido.

3. Os filmes são classificados por categorias, que indicam o valor da locação.

4. Os filmes também são classificados por gêneros, que definem uma idade mínima para que possa ser locado por um cliente.

5. Os DVDs são locados para os clientes cadastrados da locadora. São cadastrados um identificador, nome, CPF, idade e telefone para um cliente.

6. As informações de locação são: identificador único da locação, data de locação, data de devolução prevista, data de devolução efetiva, cliente que realizou a locação, a lista de DVDs locados e o valor total da locação.

7. Para que a locação de um filme seja feita por um cliente, este deve ter a idade mínima indicada no gênero do filme.

8. Para calcular a data prevista de devolução dos filmes considera-se a quantidade de filmes locados. Para 1 ou 2 filmes: 2 dias de prazo. Para 3 ou 4 filmes: 3 dias de prazo. Para 5 filmes ou mais: 4 dias de prazo.

9. Caso a locação ocorra durante os seguintes dias da semana (Segunda, Terça, Quarta, Quinta), o cliente ganha mais 1 dia de prazo para devolução.

10. Caso o cliente ultrapasse a data de devolução ele será sujeito a multa. Para cálculo de multa será considerado 10% do valor da locação para cada dia de atraso.

11. O cadastro do cliente também deve conter seu endereço, com as seguintes informações: nome da rua, número, complemento, CEP, cidade e estado.

12. Para que a locação de um filme seja efetivada, deve haver pelo menos um DVD disponível daquele filme.

APÊNDICE B

TUTORIAL DE UTILIZAÇÃO DO GERADOR AUTOMÁTICO DE TESTES

Este é um tutorial de utilização da biblioteca de geração de testes, disponível em http://www.dc.ufscar.br/~delano/isis-bdd-creator.zip, para projetos desenvolvidos com o *framework Apache Isis*. O *framework* auxilia a criação de projetos baseados em DDD (*Domain-Driven Design*), no qual é definido um modelo de domínio que representa as entidades e regras de negócios da aplicação. A partir desse modelo de domínio, o *framework Apache Isis* é capaz de criar uma interface web para manipulação e visualização dos dados das entidades envolvidas.

Para que os testes possam ser gerados automaticamente, o desenvolvedor precisa implementar métodos de validação dos atributos das entidades. Esses métodos são invocados pelo *framework* para a validação da entrada de dados nas interfaces criadas por ele. Dessa forma, o gerador de testes se beneficia dessa característica do *Apache Isis*, para criar de forma genérica e automática, testes de validação dos atributos das entidades. Esses testes são criados utilizando os conceitos de BDD (*Behaviour-Driven Development*).

Para que o gerador funcione de maneira correta, é necessário que o projeto alvo seja criado de acordo com a estrutura definida pelo *framework Apache Isis*, como pode ser visto na Figura B.1:

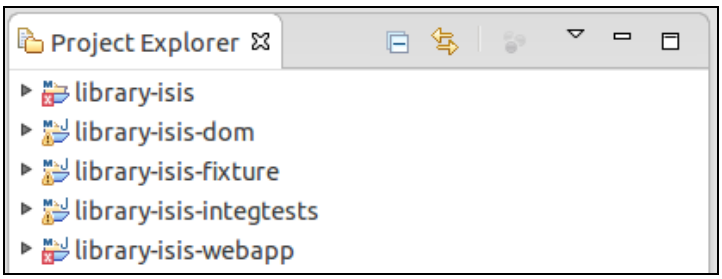


Figura B.1: Estrutura de projeto utilizando framework Apache Isis

Também como premissa de funcionamento do gerador, o desenvolvedor necessita declarar e implementar o método `validate<atributo>` para os atributos que ele deseja que sejam testados. Exemplo: a entidade `Aluno` tem um atributo RA (registro acadêmico) do tipo `Long`, e esse atributo não pode ser negativo. Na Figura B.2 é possível ver como fica a declaração do atributo e do seu método `validate`:

```
public class Aluno implements Comparable<Aluno> {

    private Long ra;

    public Long getRa() {
        return ra;
    }

    public void setRa(Long ra) {
        this.ra = ra;
    }

    public String validateRa(Long ra) {
        if (ra > 0) {
            return null;
        } else {
            return "O RA não pode ser negativo!";
        }
    }
```

Figura B.2: Entidade Aluno com atributo ra e método validateRa

No diretório disponibilizado por *download* está o gerador, o arquivo de configuração com os caminhos de entrada e saída, o arquivo de configuração de *log* do gerador e a pasta de dependências do gerador, como pode ser visto na Figura B.3:

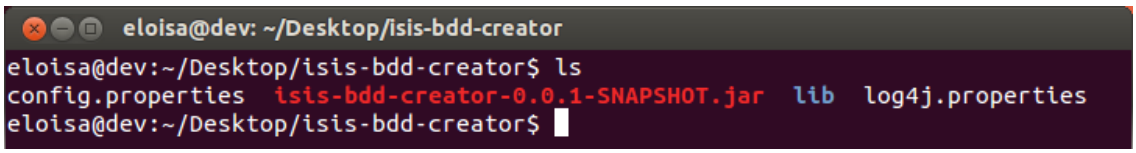


Figura B.3: Arquivos disponibilizados para geração dos testes

Antes de executar o gerador de testes, é preciso definir os caminhos de entrada e saída. O caminho de entrada é a pasta que contém as entidades do projeto alvo. Os caminhos de saída são as pastas nas quais o desenvolvedor deseja

que sejam gerados a classe de teste e o arquivo de especificação dos cenários de teste. Um exemplo é mostrado na Figura B.4:

Figura B.4: Arquivo de configuração do gerador de testes

Com todas essas etapas concluídas, basta executar o gerador de testes para que as classes de teste e os cenários sejam gerados nas pastas determinadas. Na Figura B.5 é exibido como se deve executar o gerador:

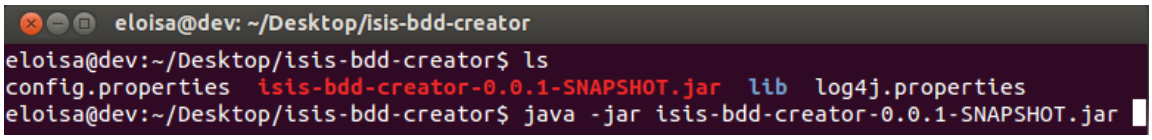


Figura B.5: Execução do gerador de testes

Por fim, é possível visualizar que a classe de teste `AlunoGlue.java` e os arquivos de especificação dos cenários `AlunoSpec_validate*.feature` foram gerados nas pastas definidas no arquivo de configuração do gerador, como é apresentado na Figura B.6:

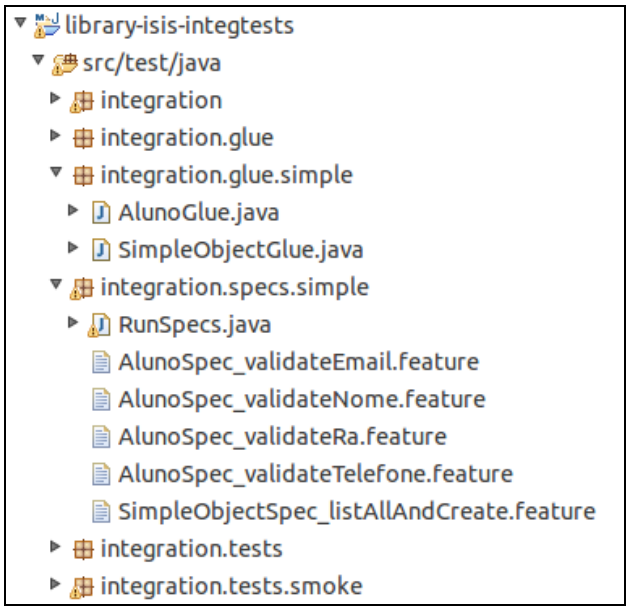


Figura B.6: Classe de teste e arquivos de cenários gerados

Na Figura B.7 pode ser visto o conteúdo da classe de teste e na Figura B.8 o conteúdo da especificação dos cenários de teste:

AlunoGlue.java

```java
package integration.glue.simple;

import org.apache.isis.core.specsupport.specs.CukeGlueAbstract;

public class AlunoGlue extends CukeGlueAbstract {

    @Given("^Eu tenho o valor \"([^\"]*)\" para um ra$")
    public void euTenhoOValorNParaUmRa(String value) throws Throwable {
        Long ra = Long.parseLong(value);
        putVar("Long", "ra", ra);
    }

    @When("^Eu valido o ra do Aluno$")
    public void euValidoORaDoAluno() throws Throwable {
        Aluno aluno = new Aluno();
        String mensagemErro = aluno.validateRa(getVar("Long", "ra", Long.class));
        putVar("String", "mensagemErro", mensagemErro);
    }

    @Then("^Eu recebo a mensagem de erro \"([^\"]*)\"$")
    public void euReceboAMensagemDeErro(String mensagemErro) throws Throwable {
        Assert.assertEquals(getVar("string", "mensagemErro", String.class), mensagemErro);
    }
```

Figura B.7: Classe de teste

AlunoSpec_validateRa.feature

```
Feature: Validar o ra de Aluno

    @integration
    Scenario:
        Given Eu tenho o valor "" para um ra
        When  Eu valido o ra do Aluno
        Then  Eu recebo a mensagem de erro ""

```

Figura B.8: Cenário de teste