

**UNIVERSIDADE FEDERAL FLUMINENSE**  
**INSTITUTO DE COMPUTAÇÃO**  
**CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO**

**DOMAIN DRIVEN DESIGN:**

*Análise das práticas preconizadas pelo DDD no desenvolvimento de software*

**Ícaro Geraldo Magnago**  
**Débora de Sousa Ribeiro Magnago**

**Niterói**  
**2016**

**Ícaro Geraldo Magnago**  
**Débora de Sousa Ribeiro Magnago**

**DOMAIN-DRIVEN DESIGN:**

***Análise das práticas preconizadas pelo DDD no desenvolvimento de software***

Trabalho de Conclusão de Curso  
apresentado ao Curso de Graduação em  
Ciência da Computação da Universidade  
Federal Fluminense, como requisito  
parcial para obtenção do Grau de  
Bacharel em Ciência da Computação.

**Orientador: Rodrigo Salvador**

**Niterói**  
**2016**

Ficha Catalográfica elaborada pela Biblioteca da Escola de Engenharia e Instituto de Computação da UFF

M196 Magnago, Ícaro Geraldo

*Domain-driven design* : análise das práticas preconizadas pelo  
DDD no desenvolvimento de software / Ícaro Geraldo Magnago,  
Débora de Sousa Ribeiro Magnago. – Niterói, RJ : [s.n.], 2016.  
59 f.

Trabalho (Conclusão de Curso) – Departamento de Computação,  
Universidade Federal Fluminense, 2016.

Orientador: Rodrigo Salvador.

1. Desenvolvimento de software. 2. Projeto orientado a domínio.  
3. Arquitetura de software. I. Magnago, Débora de Sousa. II.  
Título.

CDD 005.1

Ícaro Geraldo Magnago  
Débora de Sousa Ribeiro Magnago

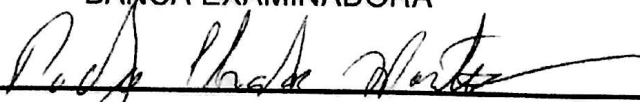
**DOMAIN-DRIVEN DESIGN:**

***Análise das práticas preconizadas pelo DDD no desenvolvimento de software***

Trabalho de Conclusão de Curso  
apresentado ao Curso de Graduação em  
Ciência da Computação da Universidade  
Federal Fluminense, como requisito  
parcial para obtenção do Grau de  
Bacharel em Ciência da Computação.

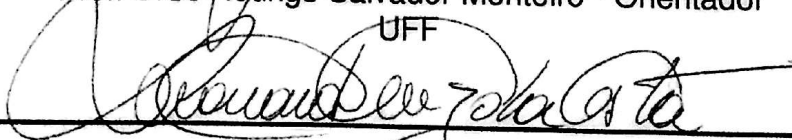
Aprovado em 28 / JULHO / 2016

**BANCA EXAMINADORA**



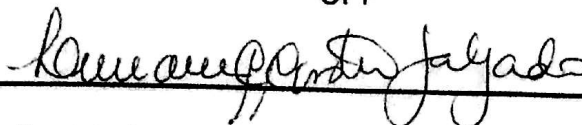
Prof. D.Sc. Rodrigo Salvador Monteiro - Orientador

UFF



Prof. D.Sc. Leonardo Cruz da Costa

UFF



Prof. D.Sc. Luciana Cardoso de Castro Salgado

UFF

## AGRADECIMENTOS

---

*Por Ícaro Geraldo Magnago*

Agradeço a Deus por ter me concedido a graça da vida e por ter me dado uma família tão maravilhosa que, com seus esforços, me ajudaram a chegar até aqui.

Aos meus pais, José e Merivan, e às minhas irmãs, Kelly e Carinne, que sempre estiveram ao meu lado, cuidando de mim com muita dedicação e amor.

À minha avó Elza e à minha tia Neida, que acompanharam o meu desenvolvimento desde criança e contribuíram para a formação do meu caráter.

À minha amiga e esposa Débora, pelos momentos juntos e por sua dedicação neste trabalho.

Ao professor e orientador Rodrigo Salvador pela paciência, tempo e dedicação para com este trabalho.

À Universidade Federal Fluminense e seus docentes que tanto me ensinaram e educaram para a vida profissional e pessoal.

## AGRADECIMENTOS

---

***Por Débora de Sousa Ribeiro Magnago***

À Deus por tudo que Ele fez por mim durante esses anos de estudos e por todas as experiências que pude viver ao Seu lado. Posso dizer que se cheguei ao final do curso foi por ajuda dEle e somente com a força dEle eu não desisti. Todas as honras, glórias e louvores sejam dadas ao Senhor Deus criador dos céus e da Terra, o único conhecedor de toda a ciência e que é e sempre será o motivo da minha alegria.

Aos meus familiares que em todos os momentos me ajudaram, mesmo sem saber do esforço que eu fazia. A vitória é de vocês.

Ao meu lindo esposo, Ícaro, que em todo o tempo me apoiou e participou de todos os momentos da minha vida acadêmica.

Aos meus amigos de estudos que fizeram os momentos de aulas e estudos mais divertidos e de grande crescimento. Um agradecimento especial aos meus amigos e amigas da minha linda e amada Primeira Igreja Batista de Irajá, que em todo o tempo torceram por mim e só tinham palavras positivas para me animar. Muito obrigada por entenderem a minha ausência nos ensaios, Chega Junto, aniversários e encontros em geral. Vocês moram no meu coração.

À Universidade Federal Fluminense e todos os professores que colaboram para a minha formação. Agradeço ao incentivo que tive do Instituto de Computação no tempo em que estive trabalhando no Projeto Incluir.

Ao nosso professor orientador Rodrigo Salvador por colaborar com a nossa formação e pelas palavras de incentivo.

Dedico a minha formação ao meu avô Manoel.

# SUMÁRIO

---

<b>CAPÍTULO 1:</b>	<b>INTRODUÇÃO .....</b>	<b>14</b>
<b>CAPÍTULO 2:</b>	<b>PRINCIPAIS CONCEITOS .....</b>	<b>16</b>
2.1	UM MODELO DE DOMÍNIO .....	16
2.2	DESIGN DE CÓDIGO .....	17
2.3	DOMAIN DRIVEN DESIGN .....	18
2.3.1	<i>Ubiquitous Language</i> .....	19
2.3.2	<i>Model Driven Design</i> .....	19
2.3.2.1	Arquitetura em camadas .....	20
2.3.2.2	Entidades .....	22
2.3.2.3	Objetos de Valor .....	23
2.3.2.4	Serviços de Domínio .....	25
2.3.2.5	Agregados .....	27
2.3.2.6	Fábricas .....	28
2.3.2.7	Repositórios .....	29
2.3.2.8	Módulos .....	30
2.3.3	<i>Design flexível</i> .....	31
2.3.3.1	Interfaces reveladoras de intenções .....	31
2.3.3.2	Funções isentas de efeitos colaterais .....	32
2.3.3.3	Asserções .....	32
2.4	RESUMO DO CAPÍTULO .....	33
<b>CAPÍTULO 3:</b>	<b>APLICAÇÃO EXEMPLO PARA AVALIAÇÃO DAS PRÁTICAS DO DDD .....</b>	<b>34</b>
3.1	ARQUITETURA EM CAMADAS .....	36
3.2	ENTIDADES .....	37
3.3	OBJETOS DE VALOR .....	39
3.4	SERVIÇOS DE DOMÍNIO .....	41
3.5	MÓDULOS .....	42
3.6	AGREGADOS .....	42
3.7	FÁBRICAS .....	45
3.8	REPOSITÓRIOS .....	45
<b>CAPÍTULO 4:</b>	<b>ANÁLISE DAS PRÁTICAS DO DDD .....</b>	<b>47</b>
4.1	BENEFÍCIOS .....	47
4.2	DESVANTAGENS .....	52
4.3	SUGESTÕES .....	53
4.3.1	<i>Uso de bibliotecas de código</i> .....	54

4.3.2	<i>Uso da metodologia Test Driven Development</i> .....	54
4.3.3	<i>Uso da metodologia Behavior Driven Development</i> .....	54
4.3.4	<i>Adoção de um Checklist</i> .....	54
<b>CAPÍTULO 5:</b>	<b>CONCLUSÃO</b> .....	<b>56</b>
	<b>REFERÊNCIAS BIBLIOGRÁFICAS</b> .....	<b>58</b>



## LISTA DE FIGURAS

---

Figura 1 - Visão de design.....	18
Figura 2 - Padrões do design dirigido por modelo.....	20
Figura 3 - Arquitetura em camadas .....	21
Figura 4 - Exemplo Entidade Pessoa .....	23
Figura 5 - Value Object Dinheiro .....	24
Figura 6 - Exemplo Domain Service .....	26
Figura 8 - Exemplo de Factory.....	29
Figura 9 - Exemplo Repository .....	30
Figura 7 - Exemplo Módulo Conta Bancária.....	31
Figura 10 - Diagrama de classe da PoC .....	35
Figura 11 - Arquitetura em Camadas PoC .....	36
Figura 12 - Classe abstrata Entidade .....	38
Figura 13 - Entidade Membro .....	39
Figura 14 - Objeto de Valor Estimativa .....	40
Figura 15 - Implementação do equals do VO Estimativa .....	41
Figura 16 - Domain Service AdicaoMembroEquipeService.....	41
Figura 17 - Módulo Equipe .....	42
Figura 18 - Diagrama agregado .....	43
Figura 19 – BacklogItem, raíz do agregado .....	44
Figura 20 - Criação de uma nova tarefa .....	44
Figura 21 - Factory para criação de defeito.....	45
Figura 22 - Exemplo de uso da factory .....	45
Figura 23 - Repositório Genérico .....	46
Figura 24 - Repositório da Entidade Membro .....	46
Figura 25 - Serviço para incluir um novo produto utilizando um repository em hibernate .....	47
Figura 26 - Implementação do Incluir em Hibernate .....	48
Figura 27 - Serviço para incluir um novo produto utilizando um repository em memória.....	48
Figura 28 – Implementação do Incluir repositório em memória .....	48
Figura 29 - Fábrica e Interface reveladora de intenção .....	49

Figura 30 - Mantendo a integridade dos dados com agregado.....	50
Figura 31 - Entidade Tarefa antes do refactoring.....	51
Figura 32 - Entidade Tarefa após a refatoração .....	51
Figura 33 - Value Object Estimativa .....	52

## LISTA DE SIGLAS E ABREVIATURAS

---

API	Application Programming Interface
CSS	Cascading Style Sheets
CPF	Cadastro de Pessoa Física
DDD	Domain-Driven Design
DTO	Data Transfer Object
HTML	HyperText Markup Language
ID	Identificador
JSF	Java Server Faces
PoC	Proof of Concept (Prova de Conceito)
SMS	Short Message Service
UML	Unified Modeling Language

## RESUMO

---

Softwares geralmente são desenvolvidos para automatizar processos existentes do mundo real ou solucionar problemas de negócios reais. Estes processos ou problemas denominam-se domínio do software. As características do domínio da aplicação e a atenção dada ao seu design determinarão o quão complexo será o desenvolvimento de um sistema. A prática do *Domain-Driven Design* (DDD) consiste em focar no design do domínio para manter a complexidade sob controle. Este estudo foi desenvolvido com o objetivo de apresentar os principais conceitos do DDD e de analisar os benefícios e desvantagens impostas por esta abordagem no desenvolvimento de software. Trata-se de um estudo de caso, para o qual foi desenvolvida uma aplicação – *Proof of Concept* (PoC) - para apoiar o gerenciamento ágil de projetos que utilizem o *Scrum*, o qual foi escolhido por se tratar de um domínio comum para a maioria dos profissionais envolvidos no desenvolvimento de um software. A aplicação foi arquitetada em camadas e por meio dela foi possível identificar e analisar os conceitos de entidades, objetos de valor, serviços de domínio, módulos, agregados, fábricas e repositórios presentes na prática do DDD. A utilização do DDD permite o isolamento dos aspectos do domínio, a separação de responsabilidades de cada componente, e a geração de um código de fácil leitura, entendimento e utilização. Por outro lado, por não ser de fácil implementação, requer mais esforço e tempo inicial de desenvolvimento, e que todos os membros da equipe estejam alinhados quanto ao entendimento de seus conceitos.

**Palavras-chave:** Domain-Driven Design; Scrum; Proof of Concept; Design de Código.

## ABSTRACT

---

Softwares are usually developed to automate real world existing process or to solve real business issues. Those process or business issues are named software domain. The domain application features and the attention given to its design will determine how complex will be the development of a system. The Domain-Driven Design (DDD) approach consists in focusing on the domain design to keep the complexity under control. This paper was written with the purpose to present the main concepts of DDD and to analyse its benefits and disadvantages in software development. In addition, this paper is a study case, which was developed an application – *Proof of Concept* (PoC) – to support the agile management of scrum projects, this domain was chosen because it's most common to all professionals involved in software development. The application was layered and through it was possible to identify and to analyse entity, value objects, domain services, modules, aggregates, factories and repositories concepts, which are part of DDD. The usage of DDD allow the isolation of domain features, the segregation of responsibilities of each component, and a code that is easy to read, to understand and to use. On the other hand, it is not easy to implement it, thus it requires more effort and time on the earlier stages of the development, and it requires that all team members are aligned to the understanding of its concepts.

**Key words:** Domain-Driven Design; Scrum; Proof of Concept; Code Design.

## CAPÍTULO 1: INTRODUÇÃO

---

Softwares geralmente são desenvolvidos para automatizar processos existentes do mundo real ou solucionar problemas de negócios reais. Estes processos ou problemas denominam-se domínio do software (AVRAM e MARINESCU, 2006). As características do domínio da aplicação e a atenção dada ao seu design determinarão o quão complexo será o desenvolvimento de um sistema. Se cada desenvolvedor implementar sua tarefa com o seu próprio entendimento a respeito do domínio sem se preocupar com o design, o resultado será um código de difícil entendimento, manutenção e extensão.

Alguns aspectos de design estão relacionados à tecnologia, e muito esforço tem sido aplicado pela comunidade para resolvê-los, por exemplo, a construção e evolução de frameworks de mapeamento objeto-relacional que visam facilitar a persistência em bancos de dados relacionais. Contudo, a maior complexidade encontra-se no próprio domínio do software. E este aspecto não pode ser ignorado.

A prática do *Domain-Driven Design* (DDD) consiste em focar no design do domínio – este que é a parte principal do software numa visão focada no negócio – para manter a complexidade sob controle. É importante que os envolvidos no projeto utilizem a mesma linguagem em todos os artefatos gerados no projeto, eliminando, assim, a necessidade de tradução e mau entendimento entre termos, e elevando e uniformizando o entendimento do domínio entre a equipe.

O DDD não se trata de uma tecnologia ou uma metodologia. Ele provê um conjunto de práticas e terminologias para a tomada de decisões de design, visando acelerar o desenvolvimento de softwares que lidam com domínios complicados (DDD COMMUNITY, 2007).

Os objetivos deste trabalho são: apresentar os principais conceitos do DDD e analisar os benefícios e desvantagens impostas por esta abordagem no desenvolvimento de software.

A análise deste trabalho se baseia na experiência dos autores no desenvolvimento de uma prova de conceito. O sistema que será utilizado é uma ferramenta para apoiar o gerenciamento ágil de projetos que utilizem o *Scrum*, o

qual foi escolhido por se tratar de um domínio comum para a maioria dos profissionais envolvidos no desenvolvimento de um software. Não se pretendeu desenvolver um sistema completo e totalmente funcional, mas o necessário para se aplicar os conceitos do DDD e servir de base para a análise proposta.

Este trabalho encontra-se organizado em quatro capítulos para além deste que apresenta a introdução. O capítulo 2 discute os principais conceitos relacionados ao DDD. No capítulo 3 se expõe a aplicação construída utilizando-se das práticas do DDD, evidenciando os padrões adotados. No capítulo 4 faz-se uma análise da abordagem do DDD a partir da experiência dos autores com o desenvolvimento do sistema de exemplo. O capítulo 5 finaliza esta monografia por meio das considerações finais.

## **CAPÍTULO 2: PRINCIPAIS CONCEITOS**

---

Este capítulo define e apresenta os conceitos que estão diretamente relacionados a este trabalho.

### **2.1 Um modelo de domínio**

No processo de desenvolvimento de um software se lida com grandes quantidades de informações a respeito do negócio do usuário, desta forma, precisa-se encontrar maneiras de representar esse conhecimento sucintamente. Faz-se necessário criar uma abstração desse conhecimento para lidar com as complexidades e focar nas partes mais importantes do domínio. Isso pode ser feito por meio dos modelos de domínio.

Um modelo de domínio é uma simplificação do conhecimento da atividade do usuário que visa destacar os aspectos mais relevantes para resolver tal problema. Não se trata de um diagrama, mas sim da ideia que o diagrama pretende transmitir. Ele é uma abstração rigorosamente organizada e seletiva daquele conhecimento (EVANS, 2004). Ele pode ser representado de várias maneiras: diagramas Unified Modeling Language (UML), casos de usos, documentos auxiliares e até mesmo por código escrito.

Ter-se um modelo de domínio na abordagem do DDD é de fundamental importância, pois:

1. O modelo e a implementação estão intimamente ligadas, isto é, a implementação deve refletir o modelo, assim como o modelo deve refletir a implementação;
2. O modelo é a base para se ter uma linguagem comum que será utilizada pelos membros da equipe, possibilitando o trabalho em conjunto entre os desenvolvedores e os especialistas do domínio;
3. O modelo é a forma como se escolhe representar o conhecimento destilado e distinguir os elementos mais importantes do negócio do usuário.



## 2.2 Design de código

Para se compreender o DDD, antes é preciso entender o termo “design” que será utilizado no decorrer do texto.

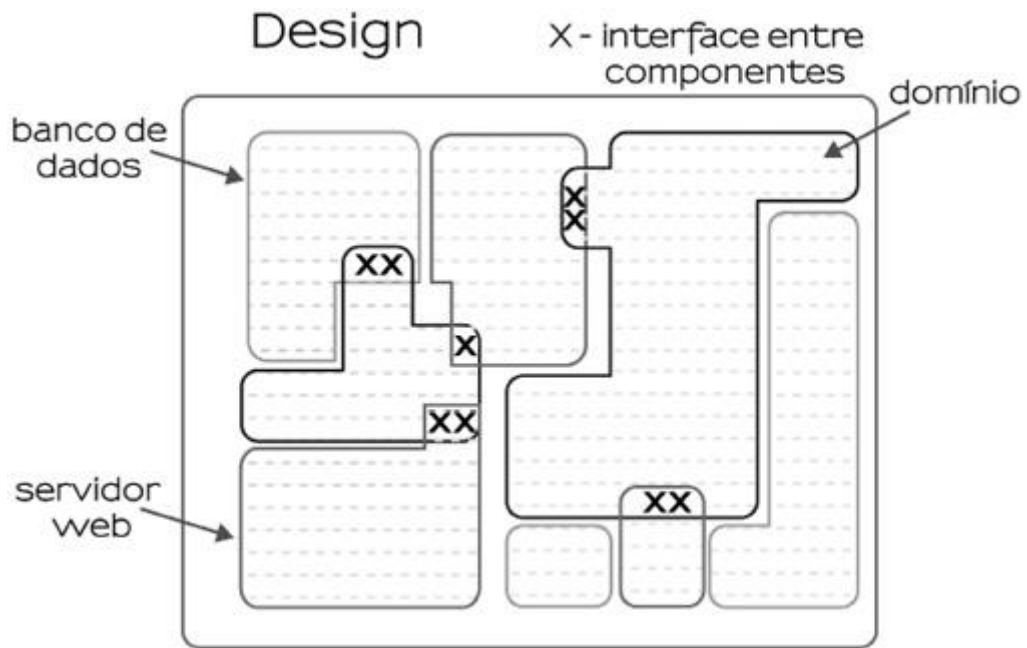
Evans (2004) não define o termo design e, na verdade, há uma linha tênue entre o que é design e o que é arquitetura. Martin Fowler, em seu livro “Patterns of Enterprise Application Architecture”, diz:

Alguns dos padrões neste livro podem ser chamados arquiteturais, já que representam decisões importantes sobre essas partes; outros são mais sobre design e o ajudam a implementar essa arquitetura. Não faço nenhuma forte tentativa de separar esses dois, uma vez que aquilo que é arquitetural ou não é subjetivo (FOWLER, 2002, p. 2).

Aqui também não se tentará distinguir arquitetura de design, porém utilizar-se-á a definição de design encontrada em “Introdução à arquitetura e design de software: uma visão sobre a plataforma java”:

Design é uma forma de interpretar a implementação, analisando e compreendendo as interações entre determinadas partes do sistema, como possíveis impactos que uma mudança em um ponto causa em outro componente que o acessa direta ou indiretamente. No ponto de vista do design, importam características das interfaces de comunicação entre partes do sistema, seus componentes, em todos os níveis de abstração, desde entre classes até dois softwares distintos (SILVEIRA et al., 2012, p. 4).

Um exemplo de aspecto de design é o modo como a aplicação se comunicará com o banco de dados.



**Figura 1 - Visão de design**

Fonte: Extraído de Silveira et al., 2012

Um bom design possibilita um software mais flexível para mudanças e consequentemente acelera o desenvolvimento e diminui custos.

### 2.3 Domain Driven Design

O desenvolvimento de um software pode parecer simples no início de um projeto, porém muitos fatores, como a falta de compromisso dos envolvidos, equipe pouco capacitada e questões jurídicas/contratuais, podem influenciar negativamente tornando-o complexo. Porém, a maior complexidade no desenvolvimento de um sistema está no domínio em que o software será empregado, ou seja, o negócio do usuário e o nível de atenção dada ao seu design (EVANS, 2004).

Um software muito complexo e com um design ruim impede que os desenvolvedores alterem o código de forma simples e segura. Isto contribui ainda mais para o aumento da complexidade da aplicação.

DDD ou Design Orientado a Domínio, segundo Evans (2004), é uma abordagem para desenvolver softwares complexos, concentrando-se principalmente no design do domínio a partir de uma conexão direta e mútua entre o modelo de domínio e a implementação do sistema.

Pode-se entender o DDD como um conjunto de princípios de design, padrões de projeto e melhores práticas, com o objetivo de alcançar um desenvolvimento rápido de software que lide com domínios de negócios complexos, e uma base de código que seja fácil de manter e estender (HOMANN, 2009).

### **2.3.1 Ubiquitous Language**

Dentre todas as causas que podem fazer um projeto falhar, a que mais tem influência sobre as outras é a comunicação (JONES, *et. al*, 2006). Os especialistas do domínio possuem seus próprios jargões e os utiliza para se comunicarem com a equipe técnica, que nem sempre os compreende. Em meio a essa divisão linguística, os especialistas do domínio descrevem vagamente o que querem. Os desenvolvedores, por sua vez, lutando por entender um domínio novo para eles, entendem tudo vagamente (EVANS, 2004). Portanto, há importância de uma linguagem compartilhada, comum a todos os envolvidos no projeto.

Ubiquitous Language ou Linguagem Ubíqua, segundo Evans (2004), refere-se a uma linguagem formada por termos que representam o domínio do negócio e que é comum entre todos os membros da equipe - técnica e usuária. Esta linguagem deve aparecer em todos os artefatos gerados no desenvolvimento de software: documentações, diagramas, código etc, eliminando a necessidade de tradução de termos entre desenvolvedores e usuários. Isto porque a tradução enfraquece a linguagem e torna deficiente a assimilação do conhecimento acerca do domínio.

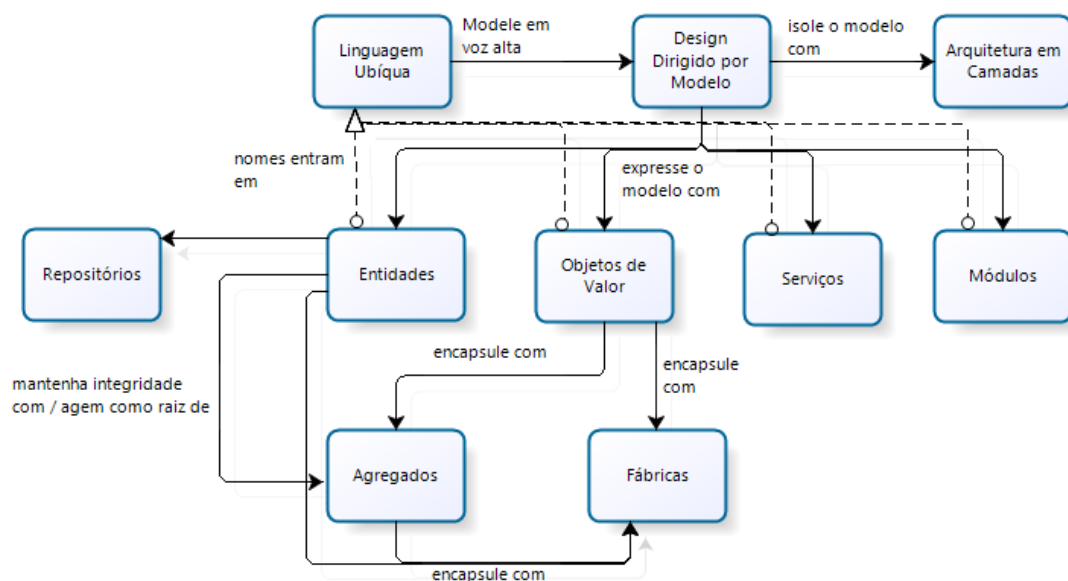
### **2.3.2 Model Driven Design**

Assim como a Linguagem Ubíqua é uma forma de representação do modelo, o código também o é; trata-se da forma mais detalhada de representação do modelo. Hoffmann (2009) argumenta que diagramas e documentos podem ser usados para esclarecer as decisões e intenções sobre o design, porém estes artefatos devem representar esboços de ideias e não ser excessivamente detalhados. Isto é de responsabilidade do código.

Para que se tenha um código que seja reflexo do modelo, os desenvolvedores necessitam conhecê-lo muito bem e devem sentir-se responsáveis por ele. De tal

modo, devem perceber que uma alteração no código pode representar uma mudança também no modelo.

Para que o código expresse o modelo de forma eficiente, técnicas de design e implementação devem ser empregadas ao desenvolver os componentes pertencentes ao domínio. Evans (2004) chama este processo de codificação de componentes de Model Driven Design ou Design Dirigido por Modelo, cujos conceitos e suas relações estão apresentados Figura 1.



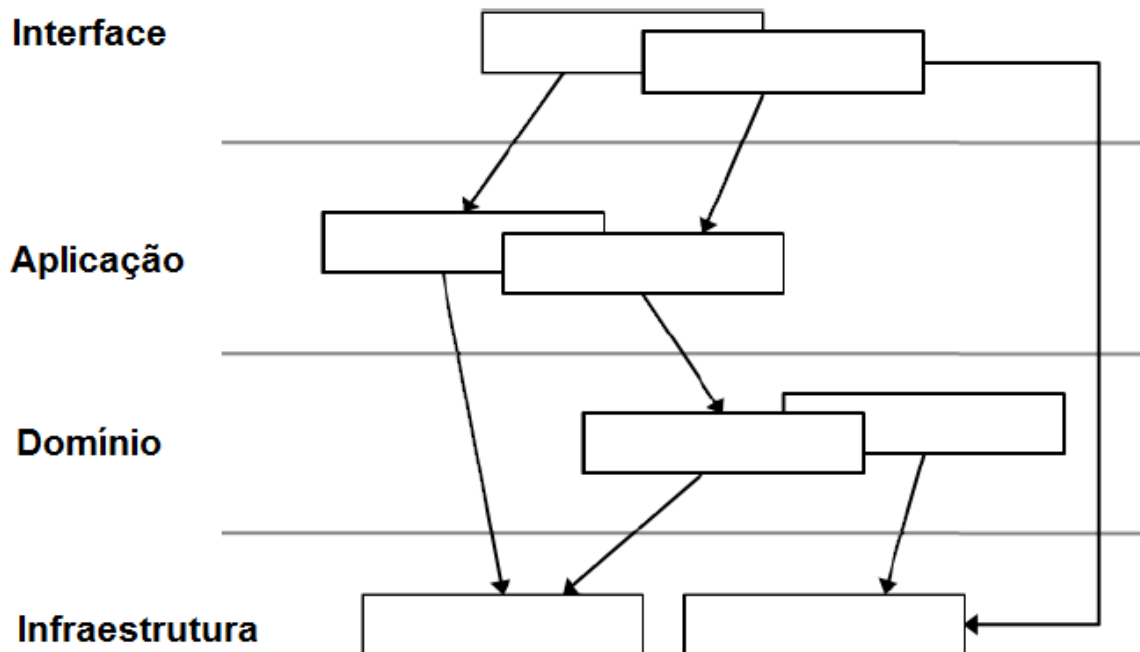
**Figura 2 - Padrões do design dirigido por modelo**

Fonte: Adaptado de Evans, 2004

### 2.3.2.1 Arquitetura em camadas

A divisão lógica do software em camadas é uma técnica bastante conhecida e utilizada em projetos de software e visa a separação de responsabilidades de cada componente, promovendo, assim, a reutilização e manutenibilidade do código. Evans (2004) propõe uma estrutura em camadas, podendo ter variações, que visa desacoplar os objetos de domínio de outras funções do sistema, evitando confundir os conceitos do domínio com outros conceitos relacionados somente à tecnologia do software ou perdendo de vista o domínio como um todo na massa formada pelo sistema.

A Figura 2 mostra a divisão de camadas proposta.



**Figura 3 - Arquitetura em camadas**

Fonte: Adaptado de Evans, 2004

A **camada de apresentação (User interface)** é onde encontra-se o código responsável pela exibição de informações e interpretação de comandos do usuário. O usuário pode ser um outro sistema de computador ao invés de um usuário humano. A **camada de aplicação (Application)** possui a responsabilidade de coordenar as atividades da aplicação. É uma fina camada e não contém lógica de negócio e nem armazena estado dos objetos de negócio, mas pode armazenar o estado das diversas tarefas em progresso da aplicação. A **camada de domínio (Domain)** é responsável por representar os conceitos do negócio e deve conter toda a lógica do domínio. Esta camada é o coração do software e o foco do DDD. Por fim, a **camada de infraestrutura (Infrastructure)** é responsável por suportar as outras camadas, fornecendo recursos técnicos como: acesso a banco de dados, envio de mensagens, bibliotecas de suporte etc.

Na arquitetura em camadas, cada elemento de uma camada deve depender somente de elementos da mesma camada ou das camadas inferiores, promovendo, desta forma, o baixo acoplamento com as camadas acima, isto é, um baixo grau de dependência entre as camadas.

O valor das camadas é que cada uma se especializa em um determinado aspecto. Essa especialização permite designs mais coesos de cada aspecto e facilita a interpretação desses designs (EVANS, 2004).

#### 2.3.2.2 Entidades

Uma entidade é um objeto do domínio que possui uma identidade única. Este objeto pode mudar constantemente durante seu ciclo de vida, porém continua sendo o mesmo objeto. Por exemplo, um objeto que representa uma pessoa física e é identificada unicamente pelo seu Cadastro de Pessoa Física (CPF) poderia mudar de nome e endereço, não obstante continuaria sendo a mesma pessoa (objeto), desde que seu identificador não mude.

É muito comum a utilização de identificadores (ID) que são gerados automaticamente, seja por meio de algum framework ou pelo banco de dados. Geralmente esses IDs gerados automaticamente não tem importância para o usuário, servem unicamente para identificar um objeto no sistema.

A construção da entidade deve estar focada na continuidade do ciclo de vida e na sua identidade. Nem todos os objetos possuem uma identidade e estes não devem ser entidades.

```

01 public class Pessoa {
02
03     private String id;
04     private String cpf;
05     private String nome;
06     private Date dataNascimento;
07
08     public Pessoa(id, cpf,
09                 nome, dataNascimento) {
10         this.id = id;
11         this.cpf = cpf;
12         this.nome = nome;
13         this.dataNascimento = dataNascimento;
14     }
15
16     //setters e getters omitidos
17
18     @Override
19     public boolean equals(Object outro) {
20         if(outro != null) {
21             Pessoa outraPessoa = (Pessoa) outro;
22             return this.id == outraPessoa.id;
23         }
24
25         return false;
26     }
27 }

```

**Figura 4 - Exemplo Entidade Pessoa**

Fonte: Elaboração dos autores, 2016

Para as entidades é importante implementar a função de igualdade baseada no seu ID.

### 2.3.2.3 Objetos de Valor

Evans (2004) descreve um objeto de valor como sendo um objeto que representa um aspecto descritivo do domínio que não possui uma identidade conceitual. Objetos de valor são utilizados para descrever objetos sobre os quais interessa **o que** eles são e não **quem** eles são.

Vernon (2013) argumenta que se deve, sempre que possível, modelar usando objetos de valor ao invés de entidades, pois tipos de valores que medem, quantificam ou descrevem alguma coisa são mais fáceis de criar, testar, usar, otimizar e manter. É recomendável que um objeto de valor seja imutável, ou seja, após ser criado ele não deve ser alterado durante seu ciclo de vida. Quando for

necessário alterá-lo é necessário substituí-lo por um novo objeto. Um exemplo de objeto de valor seria um objeto para representar o dinheiro em uma aplicação.

```
01 public class Dinheiro {
02
03     private Currency moeda;
04     private double valor;
05
06     public Dinheiro(double valor, Currency moeda) {
07         this.valor = valor;
08         this.moeda = moeda;
09     }
10
11     public Dinheiro somar(Dinheiro dinheiro) {
12         return new Dinheiro(
13             dinheiro.valor + this.valor,
14             dinheiro.moeda);
15     }
16
17     public Currency getMoeda() {
18         return this.moeda;
19     }
20
21     public double getValor() {
22         return this.valor;
23     }
24
25     @Override
26     public boolean equals(Object outro) {
27         if(outro != null) {
28             if(outro instanceof Dinheiro) {
29                 Dinheiro outroDinheiro = (Dinheiro) outro;
30                 return this.valor == outroDinheiro.valor
31                     &&
32                 this.moeda.equals(outroDinheiro.moeda);
33             }
34
35             return false;
36         }
37     }
```

**Figura 5 - Value Object Dinheiro**

Fonte: Elaboração dos autores, 2016

No exemplo da Figura 5, garantimos a imutabilidade do objeto de valor com suas propriedades privadas sem *setters* públicos. Percebe-se, ainda, que no método *somar* retornamos um novo objeto do tipo *dinheiro* com os valores somados, garantindo a imutabilidade do objeto de valor. No exemplo, implementamos o método *equals*, comparando as propriedades do objeto *dinheiro*. Isto quer dizer que



dois objetos do tipo dinheiro serão iguais se os seus valores foram iguais e as suas moedas forem iguais. Exemplo: “dez reais” é igual a “dez reais”.

Os objetos de valor devem ser pequenos, simples e representar um papel conceitual, formado a partir da união de atributos relacionados semanticamente entre si. Estes objetos podem conter outros objetos de valor e até mesmo entidades. Porém, o uso de entidades dentro de objetos de valor deve ser moderado e cauteloso, já que, diferentemente de um objeto de valor, uma entidade pode ser alterada.

Segundo Vernon (2013), um objeto de valor possui as seguintes características:

- Mede, quantifica, ou descreve alguma coisa no domínio;
- Pode ser mantido imutável após a sua criação;
- Representa um papel conceitual, formado a partir da união de atributos relacionados semanticamente;
- Pode ser substituído por completo quando a medida ou a descrição muda;
- Pode ser comparado com outro usando igualdade de valor.

Se um objeto se encaixa nas maiorias das características descritas acima, isto é um grande indicador de objeto de valor.

#### 2.3.2.4 Serviços de Domínio

Algumas operações parecem não pertencer a um objeto específico; elas representam um importante comportamento do domínio e não podem ser incorporados diretamente a uma entidade ou um objeto de valor. Estas operações não são “uma coisa”, mas sim alguma atividade ou ação (EVANS, 2004). Um serviço de domínio encapsula uma operação do domínio que semanticamente não pertence a nenhum outro objeto.

Segundo Evans (2004), o nome do serviço tende a remeter a uma atividade, um verbo ao invés de um substantivo. Além disso, os nomes das operações devem vir da linguagem ubíqua ou serem incluídos nela. Os parâmetros e os resultados produzidos devem ser objetos do domínio.

O conceito de serviço é amplamente usado na ciência da computação, por isso não se deve confundir serviços de domínio com outros serviços que são comumente utilizados em outras camadas e com outros propósitos, como web services.

Um bom serviço de domínio tem três características:

- A operação está relacionada com um conceito do domínio que não é naturalmente parte de uma entidade ou objeto de valor;
- A interface é definida em termos de outros elementos do modelo de domínio;
- A operação é *stateless* (sem estado), ou seja, o serviço não armazena estado que afeta seu próprio comportamento.

```
01 public class TransferenciaBancariaService {
02
03     private SMSService smsService;
04     private ContaBancariaRepository contaBancariaRepository;
05
06     public boolean transferir(double valor,
07                             ContaBancaria contaOrigem,
08                             ContaBancaria contaDestino) {
09
10         contaOrigem.debitar(valor);
11         contaDestino.creditar(valor);
12
13         contaBancariaRepository.adicionar(contaOrigem);
14         contaBancariaRepository.adicionar(contaDestino);
15
16         smsService
17             .enviarSMS(contaOrigem
18                       .getCorrentista().getTelefone());
19
20         return true;
21     }
22 }
```

**Figura 6 - Exemplo Serviço de Domínio**

Fonte: Elaboração dos autores, 2016

No exemplo da Figura 6, temos um serviço de domínio responsável por realizar uma transferência bancária entre duas contas. Decidiu-se utilizar um serviço de domínio, pois: i) transferência bancária faz parte do domínio em uma aplicação cujo domínio seja bancário; ii) colocar o comportamento *transferir* no objeto ContaBancaria parece esquisito, pois lida com duas contas bancárias; iii) o serviço

de domínio utiliza o repositório ContaBancariaRepository e o serviço SMSService para atualizar as contas bancárias e enviar uma Short Message Service (SMS) para o correntista, respectivamente. Utilizar serviços e repositórios em entidades e objetos de valores não é apropriado.

#### 2.3.2.5 Agregados

Um agregado é um grupo de objetos associados que são tratados como um único elemento no que se refere à alteração de dados (EVANS, 2004). A motivação por trás de agregados é garantir que objetos relacionados mantenham-se consistentes após uma alteração.

Todo agregado possui uma raiz que é uma entidade específica contida no agregado e um delimitador que define o que está dentro do agregado. A raiz é o único membro do agregado acessível para objetos externos, ou seja, objetos fora do agregado. Objetos dentro do agregado podem referenciar-se uns aos outros. Outras entidades dentro de um agregado possuem identidade local, ou seja, estas entidades só fazem sentido dentro do agregado, isto porque objetos externos não tem acesso a elas fora do contexto da raiz.

Objetos externos podem alterar a raiz do agregado ou pedi-lo para realizar alguma operação, mas nunca alterar um objeto dentro de um agregado diretamente. Desta forma, agregados mantêm a integridade dos dados e suas invariantes (condições para manter a consistência dos dados).

As regras gerais para agregados são:

- A entidade raiz possui identidade global, isto é, ela é única dentro do sistema como um todo; e é responsável por checar as invariantes;
- Entidades raízes possuem identidade global, enquanto que entidades dentro dos limites do agregado possuem identidade local, isto é, única somente dentro do agregado;
- Objetos externos podem conter referências somente para a raiz do agregado. A raiz pode entregar referências das entidades internas para objetos externos, porém esses objetos podem usá-las somente de forma transiente, sem manter referência para elas. Uma maneira de se fazer isso é passando cópias de objetos de valor;

- Somente raízes de agregados podem ser obtidas diretamente através de consultas de banco de dados. Os outros objetos devem ser obtidos através das associações transversais;
- Objetos dentro do agregado podem conter referências para outras raízes de agregados;
- Uma operação de exclusão deve remover tudo dentro de um agregado em uma operação única;
- Quando ocorrer uma alteração em qualquer objeto dentro de um agregado, todas as suas invariantes devem ser satisfeitas.

#### 2.3.2.6 Fábricas

O padrão fábrica já é bastante usado no desenvolvimento de software, sendo o elemento responsável pela criação de outros objetos sem expor sua lógica de criação. Fábrica encapsula o conhecimento necessário para a criação de objetos complexos ou agregados (EVANS, 2004). Um importante benefício na utilização de fábricas ao invés de construtores para a criação de objetos é que podemos devolver, em seus métodos, um objeto abstrato ou uma implementação de uma interface. Desta forma, o usuário que utiliza a fábrica não precisa referenciar o tipo concreto do objeto sendo criado. Isto torna a mudança ou a substituição desses objetos transparente para quem o usa.

Uma característica que as fábricas devem possuir é que cada método de criação deve ser atômico, isto é, todos os elementos necessários para a criação do objeto devem ser passados para a fábrica de maneira que este possa criá-lo em uma única operação. Além disso, uma fábrica deve garantir que todas as invariantes do objeto criado sejam satisfeitas.

Vale ressaltar que uma fábrica não deve ser utilizada para criar todos os objetos. Objetos simples podem ser facilmente criados através de um construtor. Em linhas gerais, não se utiliza uma fábrica quando:

- O construtor do objeto não é muito complicado;
- A criação do objeto não envolve a criação de outros objetos e todos os atributos necessários à construção são passados por parâmetro.

Na Figura 7 está apresentado um exemplo simples de uma fábrica que é utilizada para criar pedidos. Os pedidos poderiam ser facilmente criados através de seus construtores, porém perderíamos um grande benefício que uma fábrica também nos proporciona: o nome do método que revela a sua intenção. Tal benefício não é alcançado através dos construtores.

Muitas vezes não é necessário ter-se uma classe separada para a fábrica, pode-se ter os métodos de fábrica na raiz de um agregado, por exemplo.

```
01 public abstract class PedidoFactory {
02
03     public static Pedido criaPedido(Cliente cliente,
04                                     List<ItemPedido> itensPedido) {
05
06         if(cliente.temSaldo()) {
07             return new Pedido(cliente,
08                               itensPedido,
09                               false);
10         }
11
12         return null;
13     }
14
15     public static Pedido criaPedidoComEnvioUrgente(
16         Cliente cliente,
17         List<ItemPedido> itensPedido) {
18
19         if(cliente.temSaldo() && cliente.preferencial()) {
20             return new Pedido(cliente,
21                               itensPedido,
22                               true);
23         }
24
25         return null;
26     }
27 }
```

**Figura 7 - Exemplo de Fábrica**

Fonte: Elaboração dos autores, 2016

### 2.3.2.7 Repositórios

De acordo com Fowler (2002), um repositório encapsula o conjunto de objetos persistidos em um mecanismo de armazenamento de dados e as operações que podem ser exercidas sobre eles, provendo uma visão mais orientada a objetos da camada de persistência. Desta forma, um repositório esconde todo o código intrínseco à persistência dando a impressão de que todos os objetos estão em memória.

Um repositório deve ter uma interface global bem conhecida, com métodos para adicionar e remover objetos e métodos que retornam um objeto ou uma coleção desses objetos baseado em algum critério de pesquisa. A interface pode ter métodos complementares que retornam algum tipo de cálculo, como o número total de um determinado tipo de objeto.

Evans (2004) salienta que apenas as raízes de agregados que precisam ser acessadas diretamente devam possuir repositório, isto evita o acesso indevido aos objetos internos do agregado, os quais devem ser obtidos a partir da raiz do agregado mediante associações transversais.

```
01 public interface ContaBancariaRepository {  
02  
03     void adicionar(ContaBancaria contaBancaria);  
04  
05     void desativar(ContaBancaria contaBancaria);  
06  
07     ContaBancaria contaBancariaDeNumero(String numeroContaBancaria);  
08  
09     Collection<ContaBancaria>  
10         todasContasBancariasDaAgencia(String agenciaBancaria);  
11 }
```

### Figura 8 - Exemplo Repositório

Fonte: Elaboração dos autores, 2016

#### 2.3.2.8 Módulos

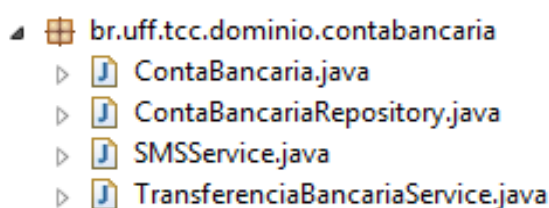
O padrão de design módulo já é bastante conhecido e utilizado no desenvolvimento de software. Na linguagem de programação Java ele é conhecido como *packages*, enquanto na linguagem C# ele chama-se *namespaces*. Módulos são uma forma de organizar o código, dividi-lo em grupos que, por algum critério, estão relacionados (HAVERBEKE, 2014).

No contexto do DDD, os módulos na camada de domínio devem emergir como uma parte significativa do modelo, contando uma história do domínio em larga escala (EVANS, 2004). Módulos não são apenas um mecanismo para divisão do código, mas também conceitos.

Sendo assim, módulos em DDD são um mecanismo de comunicação. O seu nome transmite seu significado e deve ser parte da linguagem ubíqua. Além disso, os módulos devem ter baixo acoplamento e alta coesão entre si.

Na Figura 9 é mostrado um exemplo de módulo referente à conta bancária. Toda a lógica relativa à conta bancaria está no mesmo *package* (módulo), cujos artefatos são:

- ContaBancaria: Entidade referente à uma conta;
- ContaBancariaRepository: Interface do repositório de conta bancária;
- SMSService: Interface para o serviço de envio de SMS;
- TransferenciaBancariaService: Serviço de domínio referente ao processo de transferência bancária.



**Figura 9 - Exemplo Módulo Conta Bancária**

Fonte: Elaboração dos autores, 2016

### **2.3.3 Design flexível**

Geralmente, quando se inicia o desenvolvimento de um software, não se sabe de antemão todos os seus requisitos, e mesmo que fossem conhecidos, eles tendem a mudar durante o desenvolvimento. As mudanças de requisitos ou adição de novas funcionalidades leva a refatoração do código existente a fim de adequá-lo aos novos requisitos ou as novas funcionalidades.

Evans (2004) diz que, para um software ser aberto a mudanças, seu design deve ser flexível. Esta seção aborda três padrões que podem ser utilizados para que se obtenha um design flexível.

#### **2.3.3.1 Interfaces reveladoras de intenções**

Interfaces reveladoras de intenções são como devem ser nomeados os métodos e classes do sistema que reflitam suas intenções. Nas palavras de Evans (2004): dê nomes significativos às classes, operações e seus argumentos para

descrever o seu efeito e propósito sem fazer referência a como eles fazem o que prometem.

Desta forma, mantém-se o encapsulamento e facilita o uso de tal código por desenvolvedores, pois elimina-se a necessidade do desenvolvedor de conhecer detalhes internos de implementação.

#### 2.3.3.2 Funções isentas de efeitos colaterais

O termo efeito colateral geralmente refere-se a uma consequência não intencional. Contudo, em nosso contexto, refere-se a qualquer alteração do estado do sistema decorrente de uma operação.

As operações em um sistema podem ser divididas em duas categorias: operações que obtêm informações do sistema e operações que alteram o estado do sistema. Operações ou funções que retornam valor sem produzir efeitos colaterais são mais fáceis de serem testadas e utilizadas. Em contrapartida, as operações que produzem efeitos colaterais podem mascarar comportamentos inesperados pelo desenvolvedor, tornando-as, conseqüentemente, mais difíceis de utilizar e testar (EVANS, 2004).

Evidentemente, os efeitos colaterais não podem e nem devem ser evitados no sistema. Neste sentido, para mitigar os problemas decorrentes dos efeitos colaterais, o padrão de funções isentas de efeitos colaterais diz que se devem separar os dois tipos de comportamento: métodos que retornam algum resultado não devem alterar o estado do sistema; e métodos que causam modificações não devem retornar valor.

#### 2.3.3.3 Asserções

Vimos na seção anterior que existe um tipo de função que causa efeitos colaterais e que o mesmo não pode ser evitado. Porém, aqueles que utilizarem tais funções devem estar cientes do resultado que elas produzem. O padrão de asserções juntamente com o padrão de interfaces reveladoras de intenção torna explícito os efeitos colaterais de tais funções, tornando-as mais fáceis de serem compreendidas e utilizadas.

As asserções são um conjunto de pré-condições e pós-condições que devem ser satisfeitas antes e depois, respectivamente, da chamada do método. Asserções



podem ser alcançadas através de *frameworks de programação* por contrato, através da escrita de testes unitários e até mesmo em documentos e diagramas do projeto.

## **2.4 Resumo do capítulo**

O presente capítulo teve por objetivo introduzir ao leitor os conceitos fundamentais para o entendimento da abordagem do DDD. Foram apresentados aqui os principais padrões do design dirigido por modelo cujo objetivo é fazer com que o código produzido expresse o modelo de forma mais eficiente. O capítulo ainda apresentou três padrões para se ter um design flexível cujo objetivo é facilitar a refatoração e manutenção do código existente.

## CAPÍTULO 3: APLICAÇÃO EXEMPLO PARA AVALIAÇÃO DAS PRÁTICAS DO DDD

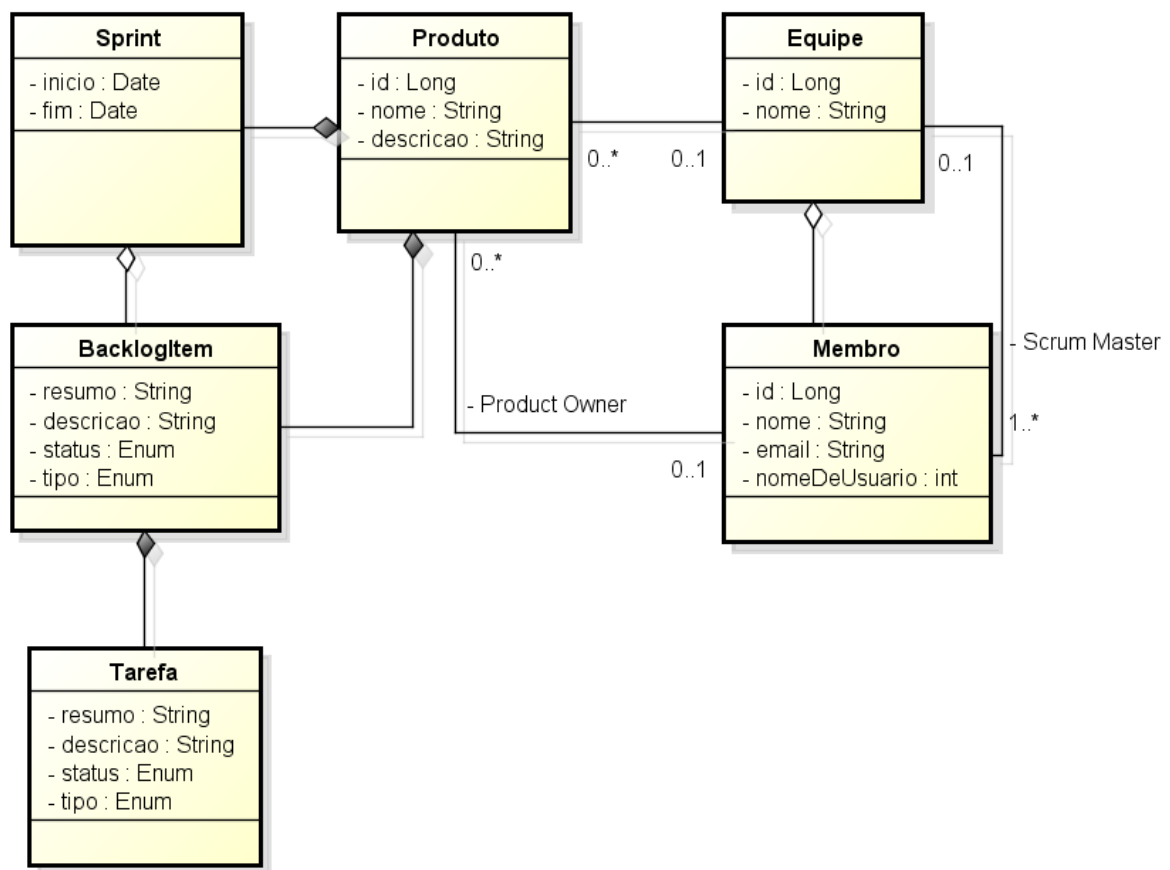
---

Este capítulo apresenta, de modo prático, a empregabilidade de alguns padrões discutidos nos capítulos anteriores. Para isso foi desenvolvida uma ferramenta para apoiar o gerenciamento ágil de projetos que utilizem o *Scrum*. O Scrum é um framework para organizar e gerenciar trabalhos complexos (Denisson Vieira, 2014). Ressalta-se, porém, que serão discutidos apenas os elementos necessários ao objetivo deste estudo. Significa dizer que nem todos os elementos presentes na metodologia *Scrum* serão vistos no modelo desenvolvido. Escolheu-se este domínio em razão de seus conceitos serem comuns à maioria dos desenvolvedores.

O software é uma aplicação web que utiliza os conceitos da referida metodologia alvo da avaliação deste trabalho. Algumas funcionalidades deste sistema são: cadastro de produtos, cadastro de membros e sua associação com uma equipe, definição do *scrum master* da equipe, definição do *product owner* e gerenciamento de *sprints*.

O sistema desenvolvido não é uma aplicação completa, sendo implementadas somente funcionalidades que serviram para demonstrar os conceitos aqui apresentados e suficientes para a análise proposta.

Para simplificação, o sistema elaborado será referido como PoC (*Proof of Concept* – Prova de Conceito). Seu diagrama de classes pode ser visto na Figura 10.



**Figura 10 - Diagrama de classe da PoC**

Fonte: Elaboração dos autores, 2016

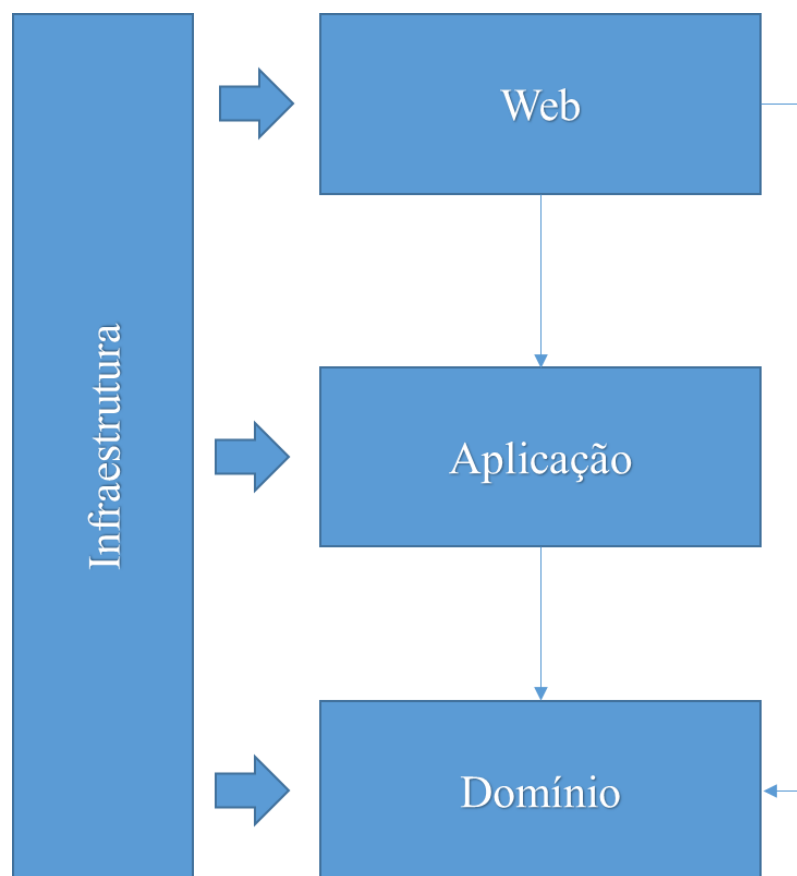
Para um melhor entendimento do domínio, os conceitos de cada entidade do modelo estão descritos abaixo.

- **Produto:** é o que se quer construir. Pode ser um software, uma casa, um carro etc.;
- **Equipe:** conjunto de pessoas que trabalha para o desenvolvimento do projeto ou produto;
- **Membro:** representa uma pessoa que está envolvida de alguma forma na construção do produto;
- **Sprint:** um período de tempo, geralmente de duas a quatro semanas, dentro do qual um conjunto de atividades deve ser executado;
- **BacklogItem:** representa o trabalho que precisa ser feito a fim de construir, manter ou sustentar o produto;

- **Tarefa:** item bem detalhado do que precisa ser feito em um período de tempo.

### 3.1 Arquitetura em camadas

O aspecto analisado nesta seção é a separação lógica em camadas do código e a relação de dependência entre elas. O objetivo dessa arquitetura é o isolamento dos aspectos do domínio e a separação de responsabilidades de cada componente. A Figura 11 mostra como a PoC está dividida e a relação de dependência entre as camadas.



**Figura 11 - Arquitetura em Camadas PoC**

Fonte: Elaboração dos autores, 2016

- **Web:** é onde encontram-se os artefatos como páginas em *HyperText Markup Language* (HTML), *Cascading Style Sheets* (CSS), javascripts, etc. É através dessa camada que se dá a interação entre o sistema e o usuário.

- Aplicação: esta camada tem a finalidade de coordenar as tarefas da aplicação e gerenciar as transações. Esta é uma camada fina e não contém lógica de negócio. A nomenclatura de suas classes geralmente contém a palavra *Service*.
- Domínio: é o coração da aplicação. Contém as classes e a implementação de suas regras de negócio, assim como interfaces para seus repositórios e os serviços de domínio.
- Infraestrutura: suporta as outras camadas oferecendo recursos técnicos, como bibliotecas de suporte e acesso a banco de dados.

Na arquitetura adotada, a relação de dependência entre as camadas se dá somente em um sentido, isto é, elementos de camadas inferiores não dependem das camadas superiores. Desta forma, conseguimos um baixo acoplamento entre as mesmas.

### 3.2 Entidades

Nesta seção se vê um exemplo de implementação de uma entidade da PoC. O diagrama da Figura 10 mostra as entidades do sistema, entre as quais iremos utilizar a entidade “Membro”. No sistema confeccionado todas as entidades herdam da classe “Entidade”, que possui a propriedade “id” que servirá para identificar um objeto no sistema, este identificador será gerado automaticamente e não possui valor para o usuário.

Futuramente, esta classe poderá implementar lógicas comuns a todas as entidades. A Figura 12 mostra a classe Entidade, das linhas 4 a 6 são os códigos utilizados para a geração automática deste identificador através da *api javax.persistence* do java.

```

01 public abstract class Entidade implements Serializable {
02     ...
03
04     @Id
05     @GeneratedValue(strategy = GenerationType.IDENTITY)
06     @Column(name = "id", unique = true)
07     protected Long id;
08
09     public Long getId() {
10         return id;
11     }
12
13     public void setId(Long id) {
14         this.id = id;
15     }
16
17     ...
18 }

```

**Figura 12 - Classe abstrata Entidade**

Fonte: Elaboração dos autores, 2016

No que se refere à entidade “Membro”, destacam-se alguns aspectos. O primeiro é o uso da linguagem ubíqua, como pode ser visto na Figura 13. Tanto a classe, como as propriedades e métodos possuem nomes significativos ao domínio e de fácil compreensão. O segundo aspecto é o uso de interfaces reveladoras de intenção, ou seja, damos nomes significativos aos métodos de uma forma que descreve o que o método faz e não como ele faz.

O terceiro aspecto a ser destacado é o conceito de funções isentas de efeitos colaterais. O método que se inicia na linha 20 somente retorna um valor sem alterar nenhum estado do objeto. Em contrapartida, o método que se inicia na linha 24 apenas altera o estado do objeto sem retornar valor. É importante seguir esse padrão para evitar efeitos colaterais inesperados e garantir que determinado método realize somente a operação que ele se propõe a realizar. Por fim, o último aspecto é a implementação do método *equals*, no qual consideramos que dois objetos são iguais se eles possuírem o mesmo valor de id.

```

01 public class Membro extends Entidade {
02     ...
03
04     //Nome do membro
05     private String nome;
06
07     //Email do membro
08     private String email;
09
10     //Nome de usuário do membro. Deve ser único
11     private String nomeDeUsuario;
12
13     //Indica se o membro está ativo no sistema
14     private boolean ativo;
15
16     //Equipe na qual ele é membro
17     private Equipe equipe;
18
19
20     public boolean ehMembroDeAlgumaOutraEquipe(Long idEquipe) {
21         return getEquipe() != null && getEquipe().getId() !=
idEquipe;
22     }
23
24     public void vincularAEquipe(Equipe equipe) {
25         setEquipe(equipe);
26     }
27
28     //Equals simplificado
29     public boolean equals(Object obj) {
30         if(obj != null) {
31             if(obj instanceof Membro) {
32                 return this.id == ((Membro)obj).id;
33             }
34         }
35
36         return false;
37     }
38     ...
39 }

```

**Figura 13 - Entidade Membro**

Fonte: Elaboração dos autores, 2016

### 3.3 Objetos de Valor

Nesta seção exemplificaremos o conceito de objetos de valor. Em nossa PoC utilizaremos o objeto de valor “Estimativa”, que tem por finalidade descrever a quantidade de horas estimadas e gastas em uma determinada tarefa (Figura 14).

```

01 public class Estimativa implements Serializable {
02     private int horasEstimadas;
03     private int horasApontadas;
04
05     public Estimativa(int horasEstimadas) {
06         this.setHorasEstimadas(horasEstimadas);
07     }
08
09     public Estimativa(int horasEstimadas, int horasApontadas) {
10         this.setHorasEstimadas(horasEstimadas);
11         this.setHorasApontadas(horasApontadas);
12     }
13
14     public Estimativa apontarHoras(int novoApontamento) {
15         return new Estimativa(this.horasEstimadas,
16 novoApontamento);
17     }
18
19     public int horasEstimadas() {
20         return this.horasEstimadas;
21     }
22
23     public int horasApontadas() {
24         return this.horasApontadas;
25     }
26
27     public int horasRestantes() {
28         return this.horasEstimadas - this.horasApontadas;
29     }
30
31     private void setHorasEstimadas(int horasEstimadas) {
32         this.horasEstimadas = horasEstimadas;
33     }
34
35     private void setHorasApontadas(int horasApontadas) {
36         this.horasApontadas = horasApontadas;
37     }

```

**Figura 14 - Objeto de Valor Estimativa**

Fonte: Elaboração dos autores, 2016

Um objeto de valor deve ser imutável. Isto é, após ser criado ele não deve ser alterado durante seu ciclo de vida. Por esta razão o objeto de valor Estimativa foi implementado de uma forma que não seja possível alterá-lo externamente. Caso haja a necessidade de um novo valor, deve-se utilizar os métodos que retornam uma nova instância do objeto.

No caso dos objetos de valor, importa-se o que eles são e não quem eles são, por isso eles não possuem identidade. A implementação do método *equals* considera duas estimativas iguais se suas propriedades tiverem os mesmos valores, vide Figura 15.



```

01 @Override
02 public boolean equals(Object obj) {
03     if(obj != null) {
04         if(obj instanceof Estimativa) {
05             Estimativa outro = (Estimativa) obj;
06             return this.horasApontadas == outro.horasApontadas
07                    && this.horasEstimadas ==
08                    outro.horasEstimadas;
09         }
10         return false;
11     }

```

**Figura 15 - Implementação do equals do VO Estimativa**

Fonte: Elaboração dos autores, 2016

### 3.4 Serviços de Domínio

Nesta seção veremos um exemplo de serviço de domínio implementado na PoC. Um serviço de domínio encapsula uma operação do domínio que semanticamente não pertence a nenhum outro objeto. Na nossa PoC criamos o serviço de domínio “AdicaoMembroEquipeService”, que possui apenas o método “membrosEmOutraEquipe” o qual retornará uma lista contendo todos os membros que já fazem parte de alguma outra equipe. Por consequência, estes não poderão ser associados a outra equipe.

```

01 public class AdicaoMembroEquipeService {
02
03     public List<Membro> membrosEmOutraEquipe(
04         Set<Membro> membros,
05         Long idEquipe) {
06         List<Membro> membrosEmEquipe = Collections.emptyList();
07         if (membros != null && !membros.isEmpty()) {
08             membrosEmEquipe = membros
09                 .stream()
10                 .filter(membro ->
11                     membro.ehMembroDeAlgumaOutraEquipe(idEquipe))
12                 .collect(Collectors.toList());
13         }
14         return membrosEmEquipe;
15     }
16 }

```

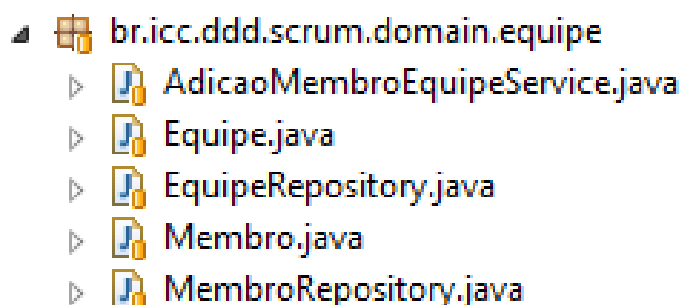
**Figura 16 – Serviço de Domínio AdicaoMembroEquipeService**

Fonte: Elaboração dos autores, 2016

Decidimos criar este serviço de domínio, pois entendemos que a operação “membrosEmOutraEquipe” não era de responsabilidade nem da entidade “Equipe” e nem da entidade “Membro”.

### 3.5 Módulos

Os módulos são uma forma de organizar o código por conceitos que se relacionam, facilitando a manutenção e objetivando o baixo acoplamento e a alta coesão. Na Figura 17 vê-se o módulo ou *package* referente ao conceito de equipe da nossa PoC.



**Figura 17 - Módulo Equipe**

Fonte: Elaboração dos autores, 2016

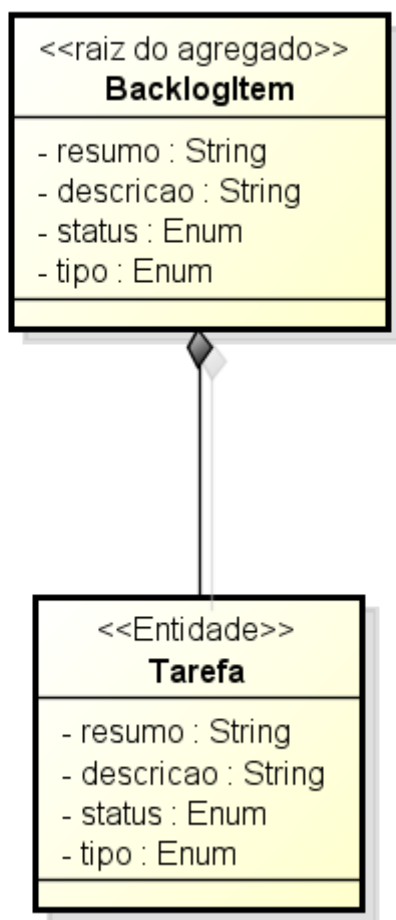
Equipe e Membro são entidades. EquipeRepository e MembroRepository são as interfaces de seus repositórios. AdicaoMembroEquipeService é um serviço de domínio cujo conceito está relacionado tanto com a equipe quanto com o membro.

### 3.6 Agregados

No modelo da Figura 18 há um exemplo de agregado cuja raiz é a entidade BacklogItem; no diagrama temos também a entidade Tarefa que está contida no agregado. Neste caso, estes objetos são tratados como um único elemento no que se refere à alteração dos dados. Por exemplo, se removermos um item de *backlog* do sistema todas as suas tarefas relacionadas também serão removidas.

Objetos – exceto a raiz – dentro de um agregado não devem ser acessados por outros que estejam fora do agregado. A responsabilidade de alterar um objeto dentro de um agregado é da raiz. Por isso, qualquer alteração a ser realizada na

entidade Tarefa deve ser de responsabilidade da sua raiz, neste caso a entidade BacklogItem, e é também por esta razão que não temos um repositório para a entidade “Tarefa”.



**Figura 18 - Diagrama com um exemplo de agregado**

Fonte: Elaboração dos autores, 2016

Em nosso exemplo, a responsabilidade da criação e inclusão de uma nova tarefa relacionada com o “BacklogItem” é da raiz do agregado, a saber, o próprio “BacklogItem”. Veja na Figura 19 que, ao fazermos isso, garantimos as seguintes invariantes: i) a tarefa será associada corretamente ao item de *backlog* ao qual pertence; ii) que toda nova tarefa se iniciará no status **NAO\_INICIADA** e; iii) ao expormos o conjunto de tarefas para objetos externos retornarmos um conjunto que não pode ser modificado por eles, garantindo, assim, a integridade dos dados e suas invariantes.

```

01 public class BacklogItem extends Entidade {
02     ...
03
04     Set<Tarefa> tarefas = Sets.newHashSet();
05
06     public void novaTarefa(String resumo, String descricao, Estimativa
estimativa) {
07         Tarefa novaTarefa = new Tarefa(
08             resumo, descricao,
09             StatusTarefa.NAO_INICIADA,
10             estimativa, this);
11
12         this.tarefas.add(novaTarefa);
13     }
14
15     public Set<Tarefa> getTarefas() {
16         return Collections.unmodifiableSet(tarefas);
17     }
18
19     ...
20 }

```

**Figura 19 – BacklogItem, raiz do agregado**

Fonte: Elaboração dos autores, 2016

O código que um cliente utilizaria para criar uma nova tarefa pode ser visto na Figura 20.

```

01 public class BacklogItemServiceImpl implements BacklogItemService {
02     ...
03
04
05     public void novaTarefa(Long idBacklogItem,
06         String resumo, String descricao,
07         Estimativa estimativa) {
08
09         BacklogItem backlogItem =
backlogItemRepository.obter(idBacklogItem);
10         backlogItem.novaTarefa(resumo, descricao, estimativa);
11
12         backlogItemRepository.atualizar(backlogItem);
13     }
14
15     ...
16 }

```

**Figura 20 - Criação de uma nova tarefa**

Fonte: Elaboração dos autores, 2016

### 3.7 Fábricas

Uma das funções do padrão fábrica é a criação de objetos sem expor sua lógica de criação. Em nossa aplicação utilizamos o conceito de fábrica para criarmos um novo item de *backlog* do tipo defeito (Figura 21 e Figura 22).

```
01 public class BacklogItem extends Entidade {
02     ...
03     private BacklogItem(String resumo, String descricao,
04         TipoBacklogItem tipo, StatusBacklogItem status) {
05         this.resumo = resumo;
06         this.descricao = descricao;
07         this.tipoBacklogItem = tipo;
08         this.statusBacklogItem = status;
09     }
10
11     public static BacklogItem novoItemDefeito(String resumo,
12         String descricao) {
13         return new BacklogItem(
14             resumo, descricao,
15             TipoBacklogItem.DEFEITO,
16             StatusBacklogItem.PLANEJADO_BACKLOG);
17     }
18 }
```

**Figura 21 - Fábrica para criação de defeito**

Fonte: Elaboração dos autores, 2016

```
01 ...
02 BacklogItem defeito = BacklogItem.novoItemDefeito(
03     resumo, descricao);
04 ...
```

**Figura 22 - Exemplo de uso da fábrica**

Fonte: Elaboração dos autores, 2016

Às vezes não é necessário criar uma classe separada para a fábrica. Aqui, por exemplo, colocamos o método na entidade *BacklogItem*.

### 3.8 Repositórios

Nesta seção falaremos sobre a implementação de um repositório da nossa PoC. Todas as *interfaces* de repositório do nosso sistema irão herdar da interface genérica *Repository*, que possui alguns métodos em comum, como incluir, excluir e alterar. Na Figura 23 tem-se a *interface* do repositório genérico.

Vale ressaltar que apenas as raízes de agregados que precisam ser acessadas diretamente devem possuir repositório, isto evita o acesso indevido aos objetos internos do agregado.

```
01 public interface Repository<E extends Entidade> extends Serializable {
02     E obter(Long id);
03     List<E> listarTodos();
04     E incluir(E entidade);
05     void incluir(List<E> entidades);
06     E atualizar(E entidade);
07     void atualizar(List<E> entidades);
08     void excluir(E entidade);
09     void excluir(Long id);
10 }
```

### Figura 23 - Repositório Genérico

Fonte: Elaboração dos autores, 2016

```
01 public interface MembroRepository extends Repository<Membro> {
02
03     List<Membro> todosOsMembrosDaEquipe(Long idEquipe);
04     List<Membro> membrosPorNome(String nome);
05 }
```

### Figura 24 - Repositório da Entidade Membro

Fonte: Elaboração dos autores, 2016

Na Figura 24 temos a interface do repositório referente à entidade “Membro”. Neste repositório, além dos métodos herdados da interface *Repository*, temos mais dois métodos: o primeiro obtém todos os membros de uma determinada equipe, e o segundo obtém todos os membros que possuem um dado nome.

## CAPÍTULO 4: ANÁLISE DAS PRÁTICAS DO DDD

---

Neste capítulo discute-se sobre as práticas do DDD exemplificadas através da construção da PoC apresentada no capítulo anterior.

### 4.1 Benefícios

Evans (2004) cita que uma das maiores vantagens da abordagem do DDD é o isolamento dos conceitos do domínio de detalhes mais técnicos. A partir do momento em que conseguimos isolar o domínio das outras camadas, temos um alto ganho no reuso do código do domínio. Em nossa aplicação, por exemplo, se houvesse a necessidade de substituir por completo o mecanismo de persistência da camada de infraestrutura, isso seria uma tarefa simples, pois a camada de domínio é independente dos detalhes técnicos da infraestrutura.

Para fins de demonstração, substituímos o nosso mecanismo de persistência em *hibernate* para um mecanismo em memória. A implementação para incluir um novo produto com *hibernate* é vista na Figura 25 e Figura 26. Enquanto que na Figura 27 e Figura 28 mostra-se como ficaria com a substituição pelo mecanismo em memória.

```
01 public class ProdutoServiceImpl implements ProdutoService {
02
03     @Inject
04     private ProdutoRepository produtoRepository;
05
06     @Override
07     public Long novoProduto(Produto produto) {
08         validarEntidade(produto);
09         Produto novoProduto = produtoRepository.incluir(produto);
10
11         return novoProduto.getId();
12     }
13 }
```

**Figura 25 - Serviço para incluir um novo produto utilizando um repositório em hibernate**

Fonte: Elaboração dos autores, 2016

```

01 public class ProdutoHibernateRepository
02     extends AbstractHibernateRepository<Produto> implements
ProdutoRepository {
03
04     @Override
05     public Produto incluir(Produto produto) {
06         em.persist(produto);
07         em.flush();
08
09         return produto;
10     }
11 }

```

**Figura 26 - Implementação do Incluir em Hibernate**

Fonte: Elaboração dos autores, 2016

```

01 public class ProdutoServiceImpl implements ProdutoService {
02
03     @Inject
04     private ProdutoRepository produtoRepository;
05
06     @Override
07     public Long novoProduto(Produto produto) {
08         validarEntidade(produto);
09         Produto novoProduto = produtoRepository.incluir(produto);
10
11         return novoProduto.getId();
12     }
13 }

```

**Figura 27 - Serviço para incluir um novo produto utilizando um repository em memória**

Fonte: Elaboração dos autores, 2016

```

01 public class ProdutoMemoriaRepository
02     implements ProdutoRepository {
03     private static List<Produto> produtos = Lists.newArrayList();
04
05     @Override
06     public Produto incluir(Produto produto) {
07         produtos.add(produto);
08
09         return produto;
10     }
11 }

```

**Figura 28 – Implementação do Incluir repositório em memória**

Fonte: Elaboração dos autores, 2016

Repare que o código cliente *ProdutoServiceImpl* não sofreu nenhuma alteração, pois a forma como é feita a persistência sempre foi transparente para ele.



Este isolamento do domínio nos permite obter um código menos acoplado, pois diminui-se a dependência de código de outras camadas do nosso sistema, e mais coeso, pois garantimos que no domínio teremos somente código relacionado ao negócio.

Outro benefício do DDD advém ao utilizarmos conceitos como o de fábricas e interfaces reveladoras de intenção. O código resultante é um código de fácil leitura, entendimento e utilização.

```
01 public class BacklogItem extends Entidade {
02     ...
03     private BacklogItem(String resumo, String descricao,
04         TipoBacklogItem tipo, StatusBacklogItem status) {
05         this.resumo = resumo;
06         this.descricao = descricao;
07         this.tipoBacklogItem = tipo;
08         this.statusBacklogItem = status;
09     }
10
11     public static BacklogItem novoItemDefeito(String resumo,
12         String descricao) {
13         return new BacklogItem(
14             resumo, descricao,
15             TipoBacklogItem.DEFEITO,
16             StatusBacklogItem.PLANEJADO_BACKLOG);
17     }
18 }
```

**Figura 29 - Fábrica e Interface reveladora de intenção**

Fonte: Elaboração dos autores, 2016

Pode-se notar que poderíamos ter optado por utilizar um construtor ao invés do método para criar um novo defeito. Porém, preferimos delegar essa responsabilidade a fábrica, pois: i) desta forma evitamos expor a lógica de criação de um novo item de *backlog* do tipo defeito e; ii) desta forma temos no nome do método a sua intenção revelada. É fácil para qualquer desenvolvedor entender que aquele método cria um novo item do tipo defeito, mesmo sem conhecer como está implementado.

Um dos desafios recorrentes no desenvolvimento de software segundo Evans (2003) é manter a integridade dos dados após uma alteração em um determinado objeto; a fim de garantir a integridade dos dados e suas invariantes o DDD oferece o padrão Agregados. Em nossa PoC temos uma relação de agregado entre *BacklogItem* e Tarefa, como visto na Figura 18. Imagine a seguinte situação como

uma regra de negócio, situação na qual gostaríamos de assegurar que, no momento em que todas as tarefas de um item de *backlog* forem resolvidas, o seu item de *backlog* também seja resolvido. Desta forma, poderíamos ter o método **alterarStatusDaTarefa** na entidade *BacklogItem* que verificaria esta restrição (invariante) e se encarregaria de alterar o status do *backlogitem* (integridade do dado). Esta implementação pode ser vista na Figura 30.

```
01 public class BacklogItem extends Entidade {
02     ...
03     public void alterarStatusDaTarefa(Long idTarefa,
04         StatusTarefa novoStatus) {
05         tarefas.forEach(tarefa -> {
06             if(tarefa.getId() == idTarefa) {
07                 tarefa.alterarStatus(novoStatus);
08             }
09         });
10
11         boolean todasAsTarefasResolvidas = tarefas
12             .stream()
13             .allMatch(tarefa ->
14 tarefa.getStatus().isResolvida());
15
16         if(todasAsTarefasResolvidas) {
17             setStatusBacklogItem(StatusBacklogItem.FEITO);
18         }
19         ...
20 }
```

**Figura 30 - Mantendo a integridade dos dados com agregado**

Fonte: Elaboração dos autores, 2016

Discutimos em capítulos anteriores o fato de que a implementação deve refletir o modelo e vice-versa. Assim, se o modelo é alterado, esta alteração deve ser refletida também no código e, portanto, o código precisa ser refatorado. O *refactoring* é um outro aspecto importante do DDD. Na medida em que o desenvolvedor realiza constantes refatorações, seu entendimento sobre o domínio aumenta e consequentemente a qualidade do seu código também. Para algumas empresas e desenvolvedores, o *refactoring* pode ser visto com maus olhos, uma vez que uma alteração mal feita pode gerar efeitos colaterais indesejáveis. Contudo, o *refactoring* é uma parte inerente ao DDD (EVANS, 2004).

Na Figura 31 mostramos a primeira implementação feita para a entidade **Tarefa**. Neste primeiro momento mapeamos os atributos *horasEstimadas* e

horasApontadas e os métodos como responsabilidade desta entidade. Porém, posteriormente entendemos que estes comportamentos não pertenciam a esta entidade e refatoramos o código. A nova implementação pode ser vista na Figura 32 e Figura 33. Com a refatoração criamos o objeto de valor Estimativa, responsável pelos comportamentos relacionados ao conceito de estimativas e, com isso, a entidade Tarefa passou a referenciar este objeto de valor. Desta forma tornamos o código mais semântico e coeso, pois conseguimos isolar as responsabilidades de cada classe.

```
01 public class Tarefa extends Entidade {
02     private String resumo;
03
04     private String descricao;
05
06     private StatusTarefa status;
07
08     private BacklogItem backlogItem;
09
10     private int horasEstimadas;
11
12     private int horasApontadas;
13
14     public void estimarHoras(int horasEstimadas) {
15         this.horasEstimadas = horasEstimadas;
16     }
17
18     public void apontarHoras(int novoApontamento) {
19         this.horasApontadas += novoApontamento;
20     }
21
22     ...
23 }
```

**Figura 31 - Entidade Tarefa antes do refactoring**

Fonte: Elaboração dos autores, 2016

```
01 public class Tarefa extends Entidade {
02     private String resumo;
03
04     private String descricao;
05
06     private StatusTarefa status;
07
08     private BacklogItem backlogItem;
09
10     private Estimativa estimativa;
11 }
```

**Figura 32 - Entidade Tarefa após a refatoração**

Fonte: Elaboração dos autores, 2016

```

01 public class Estimativa implements Serializable {
02     private int horasEstimadas;
03     private int horasApontadas;
04
05     public Estimativa() {
06         super();
07         this.horasApontadas = 0;
08         this.horasEstimadas = 0;
09     }
10
11     public Estimativa(int horasEstimadas) {
12         this.setHorasEstimadas(horasEstimadas);
13     }
14
15     public Estimativa(int horasEstimadas, int horasApontadas) {
16         this.setHorasEstimadas(horasEstimadas);
17         this.setHorasApontadas(horasApontadas);
18     }
19
20     public Estimativa apontarHoras(int novoApontamento) {
21         return new Estimativa(this.horasEstimadas, novoApontamento);
22     }
23 }

```

**Figura 33 – Objeto de Valor Estimativa**

Fonte: Elaboração dos autores, 2016

## 4.2 Desvantagens

Evans (2004) argumenta que seguir um desenvolvimento utilizando as práticas do DDD não é fácil; é preciso que todos os membros da equipe estejam alinhados quanto ao entendimento de seus conceitos. É necessário ainda que cada membro entenda cada padrão e saiba como aplicá-los de forma correta. Desenvolver um código alinhado com o domínio do negócio requer mais esforço e tempo, deve-se pensar em cada terminologia de classes e métodos de forma que estes reflitam os termos do negócio.

Além do esforço e tempo gastos em terminologias, outra área que requer grande esforço é o design do comportamento dos objetos. Identificar as responsabilidades que cada objeto terá em termos de negócio também não é uma tarefa simples, é um esforço que vai além da criação de classes com atributos e sua exposição através de *getters* e *setters* públicos. Deve-se pensar cuidadosamente nas responsabilidades de cada objeto. As vezes não é possível identificar de antemão todas as responsabilidades de um objeto e, noutras vezes, atribuímos um comportamento a um objeto quando, na verdade, deveria pertencer a outro objeto.

Em razão disso é necessário dispendir tempo para se refatorar o código e torná-lo coerente novamente.

Para desenvolvermos a nossa PoC, utilizamos para a camada web o *Java Server Faces* (JSF). A tecnologia do *JSF* permite que nossos objetos sejam automaticamente populados com os valores do formulário web quando submetidos ao servidor. Para isso, o *JSF* utiliza o padrão *JavaBean* que, em poucas palavras, significa ter *getters* e *setters* públicos para os atributos da nossa classe. Este padrão da tecnologia pode levar os desenvolvedores a fazerem o design de suas classes de forma a tirar o melhor proveito dela, ferindo alguns princípios do DDD. Quando seguimos o padrão *JavaBeans*, deixamos de dar nomes significativos aos métodos e temos a tendência de criar *getters* e *setters* públicos para todos os atributos, mesmo para aqueles que não são necessários. Com isso fica mais difícil manter as invariantes e a integridade dos dados.

Se fossemos implementar um objeto de valor para ser utilizado em uma tela JSF, infringiríamos totalmente o seu conceito, pois eles devem ser imutáveis após criados. A Figura 14 mostra como implementamos um objeto de valor. Nela, repare que ele não possui *setters* públicos e, portanto, não conseguiríamos utilizar este objeto em nossa tela JSF. O DDD trará este trabalho extra ao desenvolvimento. Devemos ter objetos que assumam esta função de armazenar e transferir os dados da interface sem comprometer o design dos objetos de domínio. Poderíamos usar para essa finalidade, por exemplo, o padrão *Data Transfer Object (DTO)*.

### 4.3 Sugestões

Muitos dos conceitos proposto pelo DDD não são padrões que possam ser assegurados pela tecnologia, portanto é preciso que os membros da equipe estejam comprometidos em utilizá-lo desde o início até o fim do desenvolvimento. Porém, manter-se no caminho nem sempre será uma tarefa simples, pois, muitos fatores podem levar a equipe a se desviar desta abordagem. Por esta razão elencamos algumas sugestões.

#### **4.3.1 *Uso de bibliotecas de código***

Uma biblioteca de código nada mais é do que um conjunto de código reutilizável. A fim de agilizar o desenvolvimento e eliminar o retrabalho que pode decorrer do uso do DDD é importante que a equipe crie sua própria biblioteca de código que os ajude a não reescrever código repetitivo e que os ajude a seguir o DDD. Por exemplo, construir uma biblioteca que facilite a transformação de objetos de domínio em DTO.

#### **4.3.2 *Uso da metodologia Test Driven Development***

A prática do *Test Driven Development* (TDD) consiste em escrever os testes unitários antes de desenvolver-se a funcionalidade em si (ANICHE, 2012). Esta prática garante a segurança no *refactoring*, e constitui-se no meio pelo qual se implementa o padrão de asserções adotado pelo DDD e ajuda os desenvolvedores no entendimento da funcionalidade sendo desenvolvida.

#### **4.3.3 *Uso da metodologia Behavior Driven Development***

No contexto desse trabalho, as funções do *Behavior Driven Development* (BDD) se assemelham aos do TDD. O que os distingue é que, com o uso do BDD, consegue-se escrever testes utilizando a linguagem natural, isto é, a ubíqua. Assim, qualquer pessoa envolvida no projeto que tenha conhecimento da funcionalidade poderia escrever estes testes.

#### **4.3.4 *Adoção de um Checklist***

Com o intuito de ajudar a equipe a se manter na direção certa no uso do DDD pode-se utilizar um *checklist*. Um *checklist* é um instrumento de controle, composto por um conjunto de perguntas e/ou afirmações sobre determinada circunstância. Os desenvolvedores poderiam se utilizar deste recurso para verificar se o código produzido está aderente às práticas do DDD, um exemplo de *checklist* seria:

- O nome da classe que eu criei faz parte da linguagem ubíqua?
- O nome do método que eu criei faz parte da linguagem ubíqua e diz o que ele faz sem dizer como ele faz?

- O método que eu criei respeita o padrão “funções isentas de efeitos colaterais”?
- A variável que criei tem um nome significativo?

## CAPÍTULO 5: CONCLUSÃO

---

O presente trabalho teve como foco apresentar os conceitos do DDD e a realização de uma análise sobre suas práticas no desenvolvimento de um sistema. A análise se deu através da experiência dos autores no desenvolvimento de uma aplicação de exemplo, proposto por eles. Devido a este fato e de que os únicos envolvidos nos projetos foram os próprios autores, a análise ficou restrita a este cenário.

Através da análise do uso do DDD no domínio proposto foi possível identificar alguns benefícios que esta abordagem proporciona ao desenvolvimento de software: i) isolamento da camada de domínio, o que proporciona o seu reuso; ii) alta coesão e baixo acoplamento da camada de domínio; iii) o código expressa o modelo e se torna mais legível através da aplicação da linguagem ubíqua, fábricas e interfaces reveladoras de intenção; iv) oferece um mecanismo para manter a consistência dos dados através do conceito de agregados; v) obtém-se um código alinhado com o modelo através das constantes refatorações.

Foi possível ainda a verificação de alguns pontos negativos na adoção do DDD: i) requer grande esforço e tempo para que o código esteja alinhado com os conceitos do domínio, isto inclui esforço e tempo para a criação das classes e métodos com nomes da linguagem ubíqua, associação correta de cada comportamento aos seus objetos e refatoração; ii) os membros da equipe devem conhecer e saber aplicar cada padrão do DDD de maneira correta; iii) algumas tecnologias usada no projeto podem levar os desenvolvedores a ferir alguns padrões do DDD, portanto é necessário atenção na utilização da tecnologia a fim de que os padrões sejam respeitados.

Após a identificação dos pontos citados acima, pôde-se sugerir algumas práticas com o objetivo de facilitar a adoção dos padrões do DDD. Tais práticas são: uso de bibliotecas de código, uso do TDD, uso do BDD e a adoção de *checklists*. O uso deles não é obrigatório e pode-se utilizar uma ou mais práticas ao mesmo tempo.



Haja vista que a análise se baseou em um cenário minimalista e considerando que um dos aspectos mais importante do DDD é a linguagem ubíqua, sugere-se como trabalho futuro uma análise utilizando-se um projeto real para verificar, por exemplo, como é o processo de formação da linguagem ubíqua e o quanto ela ajuda na comunicação entre equipe técnica e usuária. Além disso, sugere-se ainda a criação de *checklists* que possam ser utilizados em projeto reais para que seja possível verificar a utilização correta dos padrões do DDD. Por último, pode-se sugerir a criação de um *framework* ou biblioteca de código que ajude os desenvolvedores a tirar o melhor proveito do DDD.

Por fim, concluímos que o DDD traz um conjunto de boas práticas já existentes no desenvolvimento de software, porém tendo como foco principal o domínio da aplicação.

## REFERÊNCIAS BIBLIOGRÁFICAS

---

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

AVRAM, Abel; MARINESCU, Floyd. **Domain-Driven Design Quickly**. C4Media, 2007.

DAN NORTH & ASSOCIATES. **Introducing BDD**. Disponível em: <<https://dannorth.net/introducing-bdd/>>. Acesso em 05 de julho de 2016.

DOMAIN-DRIVEN DESIGN COMMUNITY. **What is Domain-Driven Design?** DDD Community [Internet], 2007. Disponível em: <[http://dddcommunity.org/learning-ddd/what\\_is\\_ddd/](http://dddcommunity.org/learning-ddd/what_is_ddd/)> Acesso em 01 de julho de 2016.

EVANS, Eric. **Domain-Driven Design**: Tackling Complexity in the Heart of Software. Indianapolis: Addison-Wesley Professional, 2004.

FIRMINO, Giovanni Luis Baroni; ZANELATO, Ana Paula Ambrósio. **Desenvolvimento de camadas utilizando DDD**. In: ETIC 2014 - Encontro Toledo de Iniciação Científica, ISSN 2176-8498., 2014, Presidente Prudente, SP. *Anais...* São Paulo: Revista Eletrônica (Anais), Toledo Prudente Centro Universitário, 2014. p. 1 - 11.

FOWLER, Martin. **Patterns of Enterprise Application Architecture**. Indianapolis: Addison-Wesley Professional, 2002.

HAVERBEKE, Marijin. **Eloquent Javascript**: A Modern Introduction to Programming. 2. ed. San Francisco, CA: No Starch Press, 2014.

HOMANN, Klaus Byskov. **Domain-driven design in action**: Designing an identity provider. 2009. 155 f. Dissertação (Mestrado) - Curso de Computer Science, Department of Computer Science, University of Copenhagen, 2009.

JONES, Capers; JOST, Alan C; PERKINS, Timothy K, et al. **What we've got here is: Failure to communicate**. **CrossTalk – The Journal of Defense Software Engineering**, v. 19, n. 6, p.10-12, 2006.

MORAES, Taciano Messias; SOUZA, Adriana Silveira de; OLIVEIRA, Juliano Lopes de. **Revisão sistemática sobre a comunicação dentro do processo de desenvolvimento de software**. Instituto de Informática - Universidade Federal de Goiás, 2011.

SCRUM.ORG. **Improving the Profession of Software Development** [Internet]. Disponível em: <<http://www.scrum.org/>>. Acesso em 01 de julho de 2016.

SILVEIRA, Paulo; SILVEIRA Guilherme; LOPES, Sérgio; et al. **Introdução à Arquitetura e Design de Software**: uma visão sobre a Plataforma Java. Rio de Janeiro: Elsevier, 2012.

VERNON, Vaughn. **Implementing Domain-Driven Design**. Westford: Addison-wesley Professional, 2013.

VIEIRA, Dennison. **Scrum**: A Metodologia Ágil Explicada de forma Definitiva, 2014. Disponível em:<<http://www.mindmaster.com.br/scrum/>>. Acesso em: 01 de agosto de 2016.