braspag
UMA EMPRESA CIELO

Domain-Driven Design: Parte I

Johnathan Fercher

Sumário

Introdução

DDD não é uma biblioteca/framework

- C#: Install-Package DomainDrivenDesign
- Python: pip install domain_driven_design
- JavaScript: npm install –save domain_driven_design
- Rust: ddd = "1.0.1"
- C++:
  - git clone https://github.com/eric-evans/ddd-devcpp
  - cd ddd-devcpp
  - mkdir build
  - cd build
  - cmake ..
  - make
  - sudo make install

O que é DDD?

- O que é um Domínio?
- O que é um Modelo?
- O que é um Design?

O que é um Domínio?

- Uma esfera de conhecimento;
- Uma área de interesse ao qual um programa é criado para lidar;
- Regras de negócio;

Ex: Uma **pessoa** possui **cartões de créditos**. **Cartões de créditos** podem ser utilizados em **transações**. Quando uma **transação** é realizada com um **cartão de crédito** sem permissão, a mesma se caracteriza como **fraude**. Uma **fraude** pode acarretar em um **chargeback**. Quando um **chargeback** ocorre, o **cartão de crédito** vinculado aquela **transação** é cancelado.

O que é um Modelo?

- Um sistema de abstrações que descreve os aspectos mais importantes de um domínio;

O que é um Modelo?

- **Pessoa**
  - Id;
  - Nome;
  - Cartões;
- **Cartão**
  - HashNumber;
  - HashHolderName;
  - ExpirationDate;
  - Brand;
  - IsActive;
- **Transação**
  - Id;
  - Data;
  - Valor;
  - Cartão;
  - Pessoa;
- **Chargeback**
  - Id;
  - Transação;
  - IsFraud;

O que é um Design?

- Implementação do software;
- Aspectos técnicos: performance, reusabilidade, segurança, tolerância a falhas, escalabilidade e etc;

Vamos armazenar as **transações** em um **Redis**. Vamos utilizar um **facade** que tente obter os valores no **cache**, caso não ache ele vai no **banco**.

O que é DDD?

- Uma abordagem de desenvolvimento de software iterativo para sistemas complexos que busca ligar o design ao modelo.

Por que é importante ligar o design ao modelo?

- Um software útil não pode ser desacoplado da realidade do domínio;
- Um domínio pode ser abstraído em um modelo de formas diferentes.
- Um modelo pode ser implementado de formas diferentes;
- Um modelo construído somente visando as regras de negócio pode ser péssimo de se implementar e manter;
- Um design construído somente visando aspectos técnicos pode não resolver o problema do domínio de forma satisfatória.

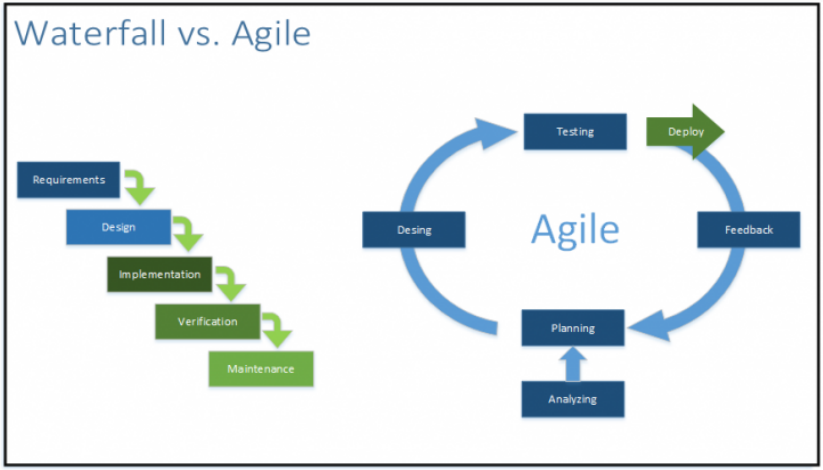
O que é necessário para realizar essa ligação?

- Os especialistas de domínio devem construir o modelo em conjunto com os desenvolvedores; (Planning)
- Mudanças no modelo devem refletir no código e mudanças no código devem refletir no modelo. (Refactoring)
- Especialistas de domínio e desenvolvedores devem utilizar a **linguagem ubíqua**.

O que é necessário para realizar essa ligação?

- É interessante que o software seja construído usando algum **padrão arquitetural** que isole o domínio e mantenha um baixo acoplamento e uma alta coesão entre camadas.
- O modelo deve ser construído utilizando os **blocos de construção**.

Ligação de DDD com Desenvolvimento Ágil



Devo ou não devo usar DDD?

- Usar uma biblioteca de IA para aprender e gerar um artefato;
- Escrever o controle embarcado de um robô;
- Escrever um *console application* para tratar algo pontual;
- Coisas simples;

- Construir uma API de OAuth;
- Construir um sistema de pagamentos;
- Criar um jogo;
- Qualquer coisa complexa, onde o domínio precisa ser muito bem compreendido;

O que custa mais caro?

- Errar uma abstração de um modelo.
  - Ex: A tabela X deveria ser na verdade duas tabelas. E essas tabelas possuem casos de uso e responsabilidades totalmente diferentes.
- Errar a implementação de um aspecto de um modelo.
  - Ex: Não contava que teriamos que paralelizar esse processamento, não vai ser possível usar um **decorator**. Podemos utilizar um **composite**.

Linguagem Ubíqua

Linguagem Ubíqua ou Linguagem Onipresente

- Formas de expressar um modelo: diagramas, casos de uso, desenhos e etc;
- Tudo isso precisa ser atualizado constantemente;
- A fala/escrita é usada de forma constante. Se algo soa errado, provavelmente está errado.
- Por que não fazer a fala/escrita nossa representação do modelo?

Linguagem Ubíqua ou Linguagem Onipresente

Ex: Um **cartão de crédito** realiza uma **transação**. Uma **transação** pode ser uma **fraude**.

Linguagem Ubíqua ou Linguagem Onipresente

Ex: Um **cartão de crédito** realiza uma **transação**. Uma **transação** pode ser uma **fraude**. Um **chargeback** é uma **fraude**.

Linguagem Ubíqua ou Linguagem Onipresente

Ex: Um **cartão de crédito** realiza uma **transação**. Uma **transação** pode ser uma **fraude**. Um **chargeback** pode acarretar em uma **fraude**.

Linguagem Ubíqua ou Linguagem Onipresente

Ex: Uma **pessoa** possui **cartões de créditos**. **Cartões de créditos** podem ser utilizados em **transações**. Uma **transação** pode ser uma **fraude**. Um **chargeback** pode acarretar em uma **fraude**.

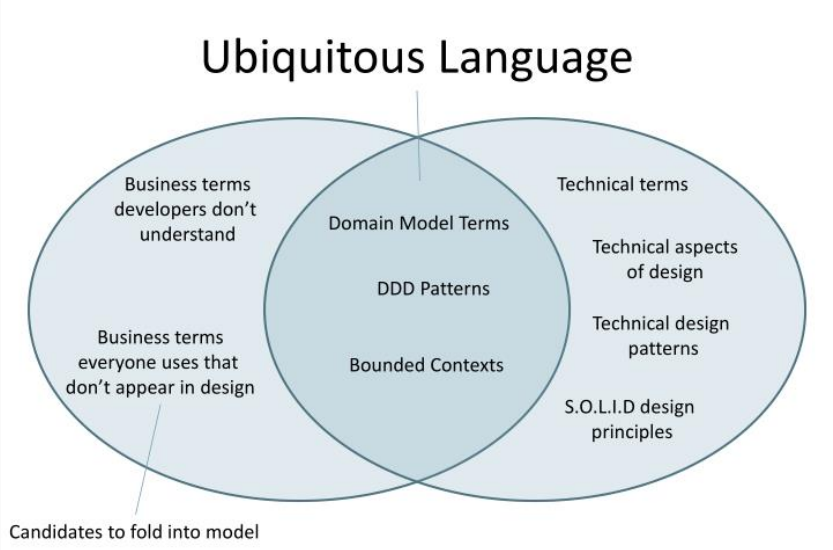
Linguagem Ubíqua ou Linguagem Onipresente

Ex: Uma **pessoa** possui **cartões de créditos**. **Cartões de créditos** podem ser utilizados em **transações**. Quando uma **transação** é realizada com um **cartão de crédito** sem permissão, a mesma se caracteriza como **fraude**. Um **chargeback** pode acarretar em uma **fraude**.

Linguagem Ubíqua ou Linguagem Onipresente

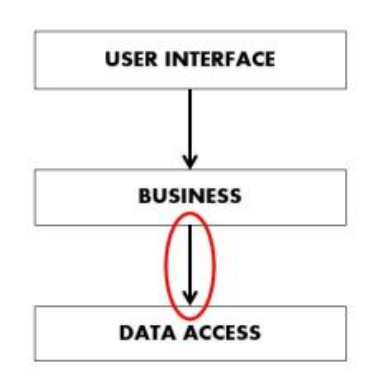
Ex: Uma **pessoa** possui **cartões de créditos**. **Cartões de créditos** podem ser utilizados em **transações**. Quando uma **transação** é realizada com um **cartão de crédito** sem permissão, a mesma se caracteriza como **fraude**. Uma **fraude** pode acarretar em um **chargeback**. Quando um **chargeback** ocorre, o **cartão de crédito** vinculado aquela **transação** é cancelado.

Linguagem Ubíqua ou Linguagem Onipresente

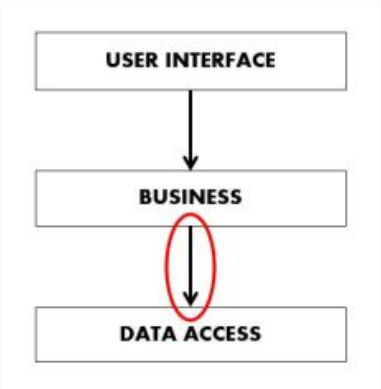


Padrões Arquiteturais

Arquitetura em Camadas (Clássica)

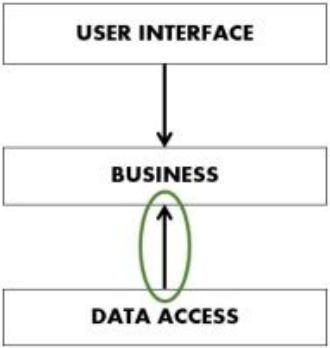


Arquitetura em Camadas (Clássica)

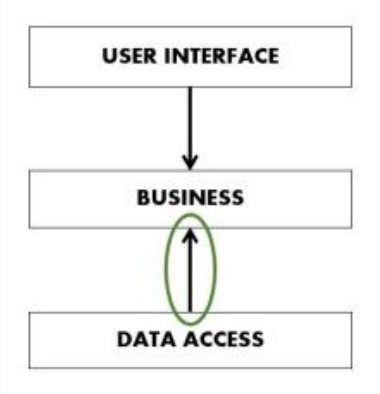


- **Business** carrega as regras de negócio e é consequentemente o domínio;
- **User Interface** depende de **Business** que depende de **Data Access**;
- **Business** pode quebrar caso **Data Access** mude;
- **Business** deveria ser a camada com menores chances de quebrar, ela não deveria depender de ninguém.

Arquitetura em Camadas (Com foco no Negócio)

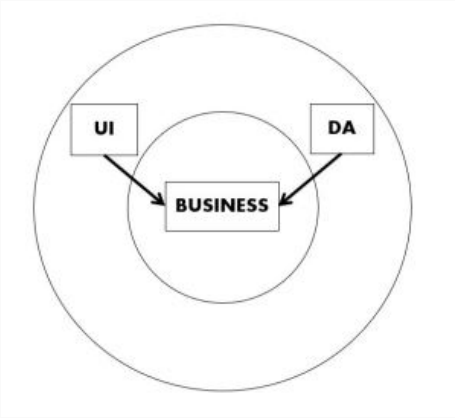


Arquitetura em Camadas (Com foco no Negócio)

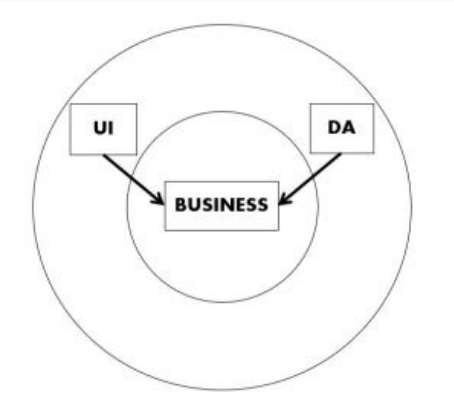


- Foi aplicado o *Dependency Inversion Principle* de **Data Access** para **Business**;
- **Business** agora não depende de ninguém e somente quebra com mudanças de negócio;

Arquitetura de Cebola

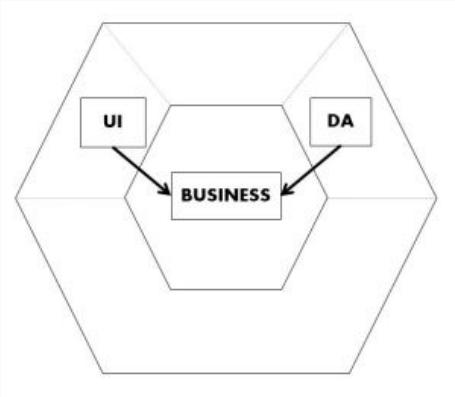


Arquitetura de Cebola



- Também isola a camada **Business**;
- O *Release Equivalence Principle* é violado, pois alterações na camada **UI** afetam a versão de **DA**, e vice-versa.
- O *Common Closure Principle* é violado, pois a camada muda por razões de **UI** e **DA**.
- O *Common Reuse Principle* é violado, pois **DA** não poderia ser reutilizado sem que dependências de **UI** sejam levadas juntas, e vice-versa.

Arquitetura Hexagonal



Arquitetura Hexagonal

- Também isola a camada **Business**;
- O *Release Equivalence Principle* é violado, pois alterações na camada **UI** afetam a versão de **DA**, e vice-versa.
- O *Common Closure Principle* é violado, pois a camada muda por razões de **UI** e **DA**.
- O *Common Reuse Principle* é violado, pois **DA** não poderia ser reutilizado sem que dependências de **UI** sejam levadas juntas, e vice-versa.

Por que separar em camadas além de isolar o domínio?

- Desacoplar soluções;
- Aumentar coesão dentro dos pacotes;

Blocos de Construção

Blocos de Construção

- Entidades;
- Objetos de Valor;
- Serviços;
- Módulos;

Entidades

...

Objetos de Valor

...

Serviços

...

Módulos

...

Ciclo de Vida de um Objeto de Domínio

Blocos de Construção

- Agregados;
- Fábricas;
- Repositórios;

Agregados

...

Fábricas

...

Repositórios

...

Referências

Referências

- https://github.com/johnfercher/software-literature-review

Obrigado