

# Domain Driven Design

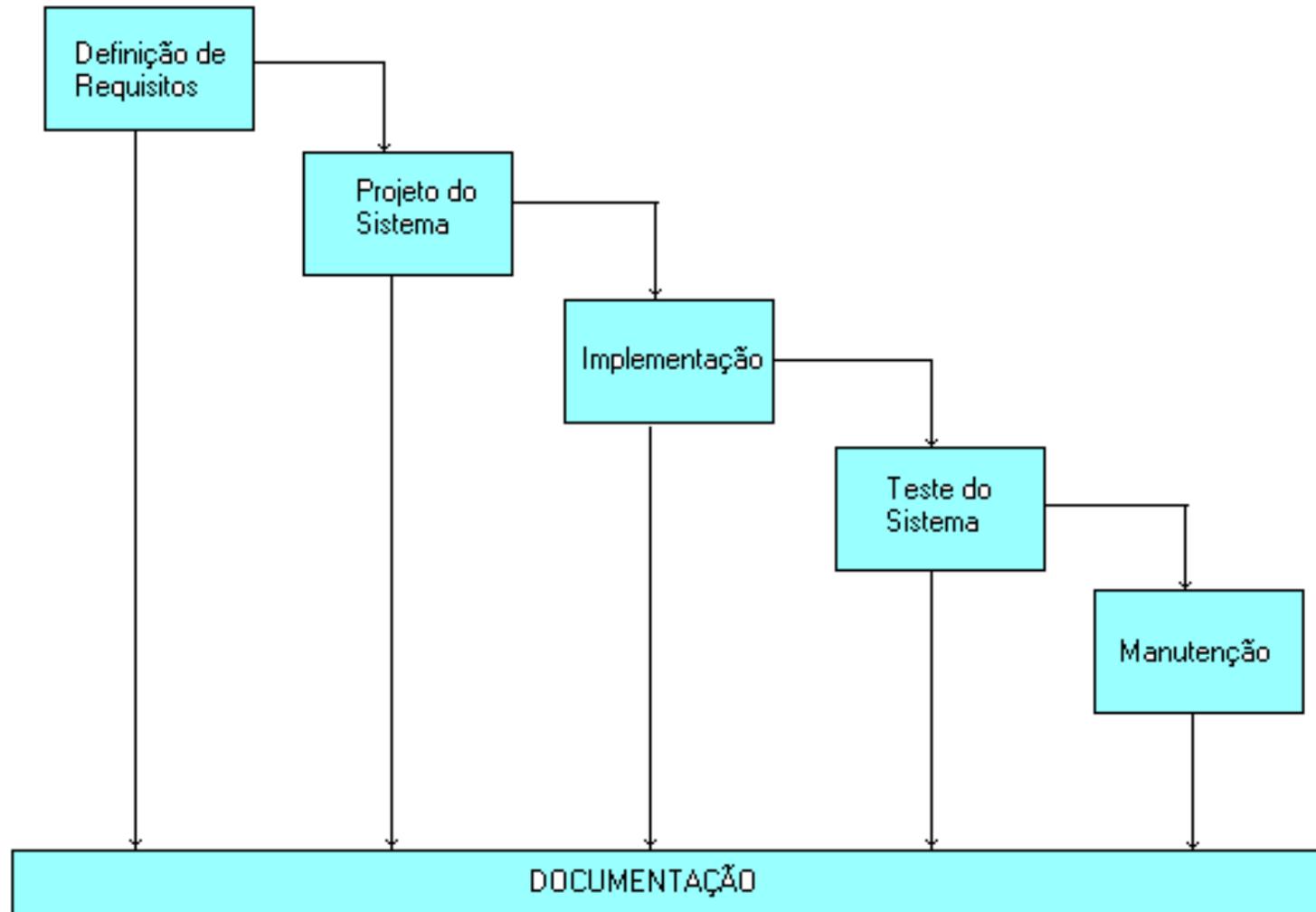
Prof. Rodrigo Macedo

# Escopo do Curso

- Conceituação Geral: processo de desenvolvimento, ddd.
- Domain, Domain Model, Domain Expert.
- Linguagem e Comunicação.
- Arquitetura
- Questões de concursos



# Processo de Software



# Processo de Software

Clientes e Desenvolvedores não conseguem se comunicar bem.



# Conceitos Gerais

“Embora nunca tenha sido fornecida claramente, uma **filosofia** emergiu como recorrente na comunidade de orientação a objetos, uma filosofia que eu chamo de domain-driven design” - Eric Evans.

- Uma abordagem de desenvolvimento de software.
- Design (Projeto) orientado ao domínio.
- Software orientado a objeto da maneira correta.
- Inclui alguns valores e princípios descritos no Manifesto Ágil.

# Terminologias do Domain - Driven

Para um completo entendimento dos conceitos que compõem o domain-driven design, é preciso entender suas peças fundamentais e como elas se inter-relacionam.

1. Domain.
2. Domain Model.
3. Domain Experts.



# Domain

De todas as peças, o domain é a mais importante, em virtude de que é baseado nele que orientaremos nosso design.

Domain - “O campo de ação, conhecimento. Ex: O domínio da ciência”.

- Campo de ação, conhecimento e influência do software.
- Não é um diagrama ou algo do tipo.
- Ele é abstrato e muitas vezes intangível que deve ser percebido ao longo do tempo em um esforço de descoberta e aprendizado, desvendando os limites de uma operação ou suas influências exercidas ou recebidas de fatores externos.

# Domain



# Foco

- O foco é no **domínio**, são as regras de negócio, os comportamentos.
- O foco não está na tecnologia.
- Software de negócios com regras complexas, que justifiquem a adesão da filosofia Domain Driven Design.

*“Domínio é a área de conhecimento do software”*



# Domain Model

- Estrutura do conhecimento adquirido até o presente momento que irá evoluir a um entendimento mais profundo.
- Não é um diagrama, mas uma ideias que o diagrama tenta representar.
- O domain é a situação a ser resolvida, enquanto o domain model é uma representação estruturada do conhecimento.

Segundo Eric Evans, há algumas considerações importantes:

1. O modelo e o coração do design se encaixam.
2. O modelo é a base da linguagem utilizada pelos membros do time.
3. O modelo é conhecimento destilado.

# Domain x Domain Model



Representa o modelo do domínio.



Representa o domínio.

# Domain Expert

- Domain Experts são as pessoas que conhecem profundamente os detalhes de um domains, que sabem por quê, quando e como um problema deve ser resolvido pelo software.
- O sucesso de um bom software depende da comunicação enntre os desenvolvedores e os domain experts.
- Muitas vezes, os domain experts não conhecem o domain inteiro, pois existem vários domain experts para um mesmo domain, cada um com expertise em uma parte do sistema.
- Isso diferencia um domain expert de um cliente e de um Product Owner.

# Premissas

- O foco deve ser o domínio e a lógica do domínio.
- Projetos complexos devem ser baseados em um **modelo**.
- Iniciar a colaboração criativa entre a área técnica e os domain experts para iterativamente estarem mais perto do coração conceitual do problema.



# O Design do Software

- O objetivo do domain-driven é facilitar o controle da complexidade de um sistema, impedindo que o design fique demasiadamente complexo e incompreensível, pois quando isso acontece, os desenvolvedores evitam realizar manutenções, com receio de comprometer o seu funcionamento.
- Para evitar esse cenário, os desenvolvedores devem estar familiarizados com o domain e os domain experts deve estar disponível e dispostos a compartilhar seu conhecimento.
- Isso só é possível se ambos falarem a mesma linguagem de negócio.

# O Design do Software



**Quantas vezes, ao iniciar um novo projeto nos deparamos com um domain desconhecido?**

# O Design do Software



Download from  
Dreamstime.com  
The watermark can be removed for purchasing purposes only.

id 97840709  
© Md. Delwar Hossain | Dreamstime.com

**Desenvolvedores x Domain Experts**

# Processo de Aprendizagem do Domain

- Uma das propostas do domain-driven design é que a disseminação do conhecimento comece pelo domain expert.
- Mesmo que eles estejam saturado de informações e detalhes sobre o domain, inicialmente, o ideal é que a disseminação comece de forma mais simplificada, abstraindo alguns detalhes.
- O aprofundamento em detalhes do domain é uma decisão a ser feita de acordo com o andamento do projeto.

# Processo de Aprendizagem do Domain

Desenvolver um software que simule a alimentação de um gavião.

**Escopo:** “O software deve simular o sistema de caça do gavião para presas pequenas e presas grandes”.

Quais seriam os possíveis alvos do gavião??



# Processo de Aprendizagem do Domain

Desenvolver um software que simule a alimentação de um gavião.

**Escopo:** “O software deve simular o sistema de caça do gavião para presas pequenas e presas grandes”.

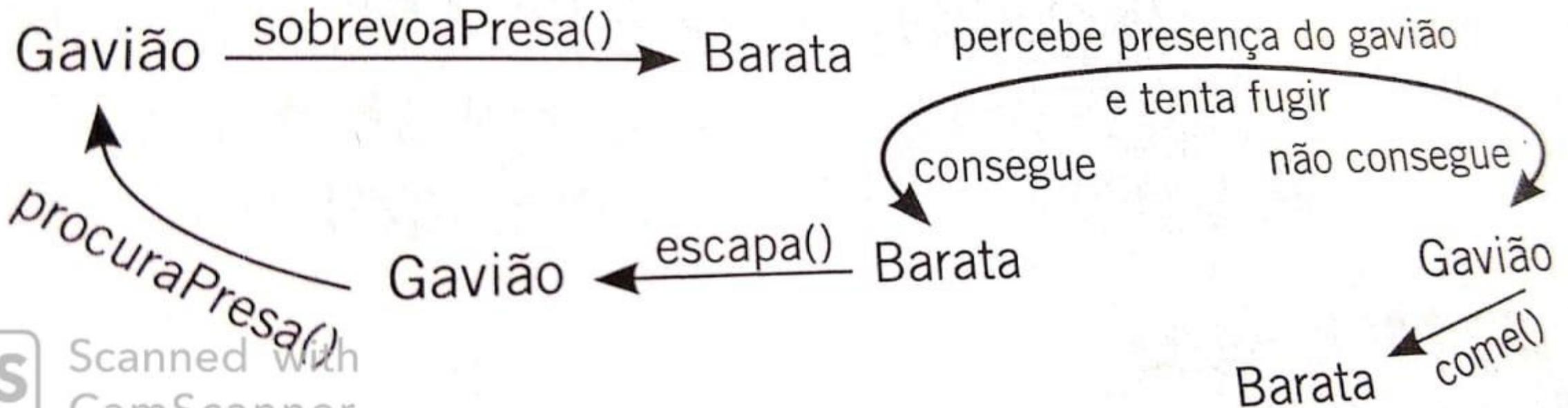


Lê-se: “Gavião é uma Ave que é um Vertebrado que é um Animal”. “Barata é um Inseto que é um Invertebrado que é um Animal”.

# Processo de Aprendizagem do Domain

Desenvolver um software que simule a alimentação de um gavião.

**Escopo:** “O software deve simular o sistema de caça do gavião para presas pequenas e presas grandes”.



# Processo de Aprendizagem do Domain

Desenvolver um software que simule a alimentação de um gavião.

**Escopo:** “O software deve simular o sistema de caça do gavião para presas pequenas e presas grandes”.

Após a finalização da compreensão do domain, validada pelo domain expert, o time de desenvolvimento pode dá início a implementação da simulação da alimentação do gavião.

```
gaviao.cacar()
do{
    presa = gaviao.procuraPresas();
    gaviao.sobrevoaPresas(presa)
    if(presa.escapa()){
        continue;
    }
    gaviao.come()
} while(gaviao.comeu?())
```



# Linguagem e Comunicação

- Os domain experts têm dificuldade para entender a linguagem técnica dos desenvolvedores, e é comum que os times de desenvolvimento tentem impor seus termos a eles.
- Em contrapartida, os domain experts podem tentar sobrepor-se aos desenvolvedores, definindo como fazer tecnicamente, em vez de focar em explicar o domain e deixar a parte técnica aos desenvolvedores.
- O ideal, é que nós, os desenvolvedores, façamos uma ponte inicial, abstraindo a parte técnica para os domain experts.
- O **domain model** pode ser a base da linguagem comum em um projeto de software.

# Linguagem Ubíqua

- A linguagem ubíqua é uma linguagem presente em todas as facetas do desenvolvimento, sendo utilizadas por todos os envolvidos com o objetivo de eliminar a necessidade de traduções.
- Traduções ajudam a comunicação, mas não a deixam perfeita, já que termos e detalhes são perdidos.
- Para uma comunicação perfeita, é preciso que se utilize a mesma linguagem, o que, em muitos casos, requer um aprendizado.

Quando isso acontece:

1. Mudanças na linguagem são reconhecidas como mudanças no domain model.
2. Os desenvolvedores são capazes de apontar imprecisão e pontos de contradição no domain.

# Linguagem Ubíqua



TRADUÇÃO: O banco de dados foi danificado porque o servidor ficou sem espaço em disco.

# Modelando em voz alta

- Uma vez que todos estejam utilizando uma linguagem comum, uma das melhores formas de explorar um design é por meio da fala.
- Tentar expressar o design em frases coerentes resulta em um refinamento dos objetos e um melhor entendimento nos detalhes.

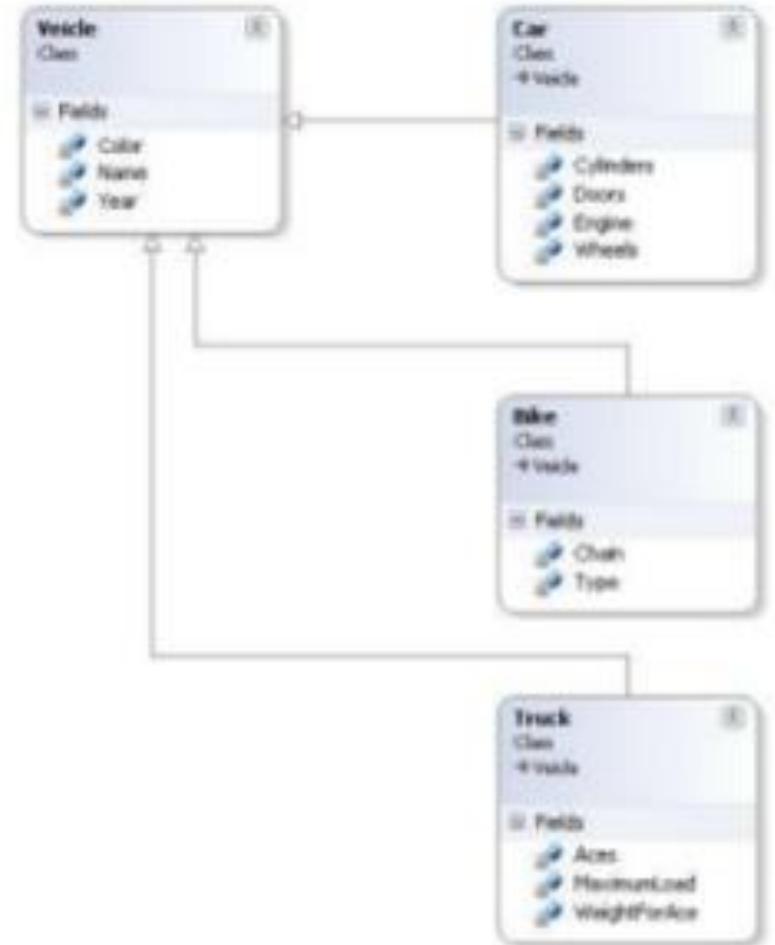
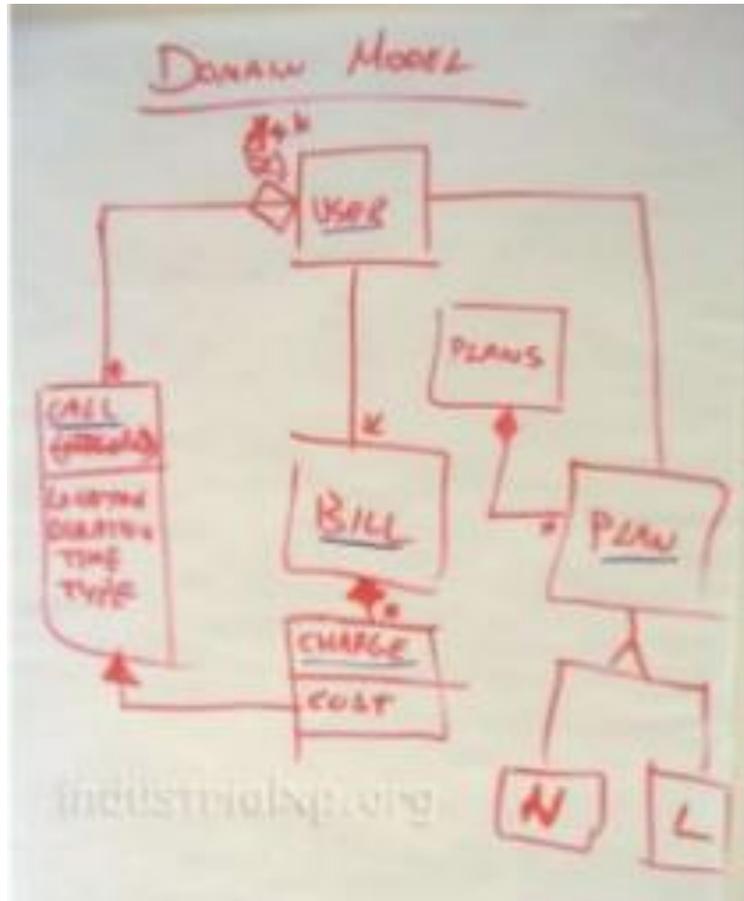
Alguns exemplos extraídos do livro de Evans (2004):

1. “Um Serviço de Rotas (RoutingService) encontra um Itinerário que satisfaz uma Especificação de Rota (RouteSpecification)”.
2. “A origem, destino e assim por diante ... tudo isso se encaixano Serviço de Rotas (RoutingService) e obtemos, então, um Itinerário que tem tudo que nós precisamos”.

# Diagramas

- Os diagramas podem ser utilizados para auxiliar na obtenção de feedback e confirmação do entendimento obtido.
- Ao desenharmos em frente a um domain expert, ativamos partes cognitivas do cérebro, fazendo todos terem outro ponto de vista em relação ao que está sendo explicado, o que facilita muito a comunicação.
- O domain-drive design não sugere uma forma ou sintaxe específica para os diagramas. Na maioria dos casos, um diagrama simples que represente a interação entre os objetos é suficiente: alguns retângulos que representem objetos, linhas que os conectam, comentários ligados por uma seta, etc.

# Diagramas



O mais importante é a mensagem que um diagrama quer passar.

# Documentação

- O código escrito e a comunicação falada representam o domain e complementam um ao outro.
- Para times grandes, pode ser que seja necessário documentos escritos. Um grande desafio, nesse caso, é manter os documentos atualizados.

Algumas diretrizes para escrever documentos:

1. Documentos devem complementar o código e a comunicação falada.
2. Um documento não deve tentar fazer o que um código fez bem.
3. Documentos devem ser sempre atualizados.
4. Um documento deve estar envolvido nas atividades do projeto.

# Documentação



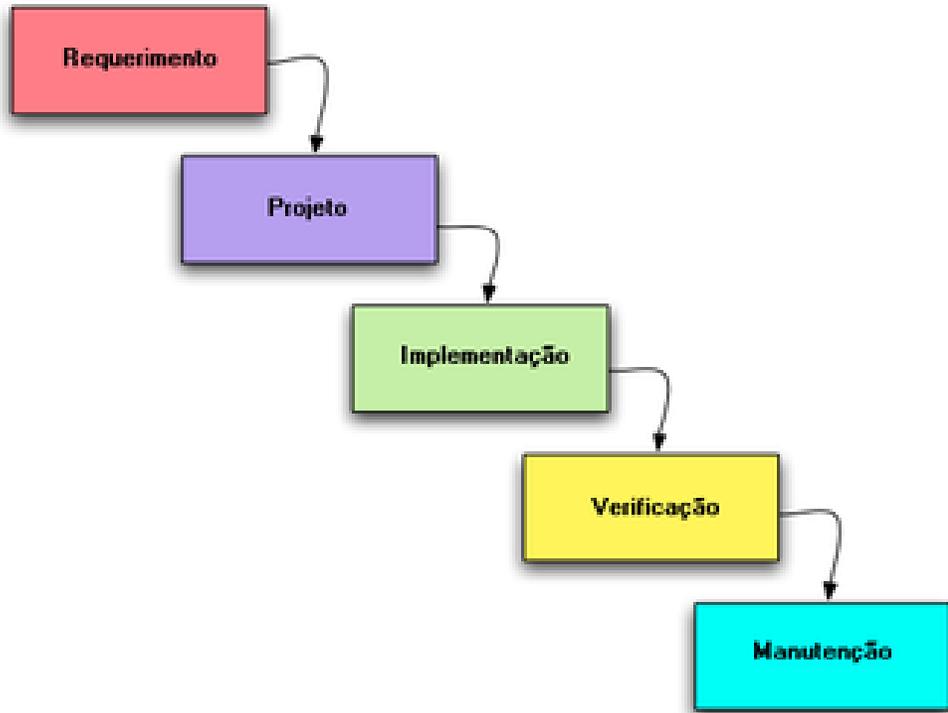
# Implementação

- É importante que o código expresse os detalhes do domain de forma clara e objetiva.
- Dá se origem a uma nova sigla: **MDD** (Model-Driven Design)
- O MDD consiste em uma filosofia de modelagem que sugere que foquemos o domain ao modelar um software, de forma a expressá-lo tanto nos diagramas quanto no código.
- O resultado esperado do MDD não é um diagrama ou documento, mas a organização e o design dos componentes do código.
- O design é a forma como o código é disposto e, por isso, é necessário um **paradigma** de programação, por exemplo, o paradigma orientado a objetos. Nesse caso, teremos objetos representando os conceitos do domain.

# Modelagem e Desenvolvedores

- Os desenvolvedores devem ser os responsáveis pela modelagem do software, pois precisam sentir-se responsáveis pelo modelo.
- Se as pessoas que escrevem o código não assumirem a responsabilidade pelo design e pelo domain modelou mesmo não entenderem como fazer o model funcionar, então o model não terá nada a ver com o software.
- Ao mudarmos o código, mudamos o domain model e seu design.
- Ao dividirmos o trabalho em fases com pessoas diferentes realizando-o, estamos tornando impossível alcançar um domain model coerente. Pois a compreensão de um domain, é um processo que leva tempo.

# Modelagem e Desenvolvedores



“... A verdade está no código”

# Os blocos do MDD

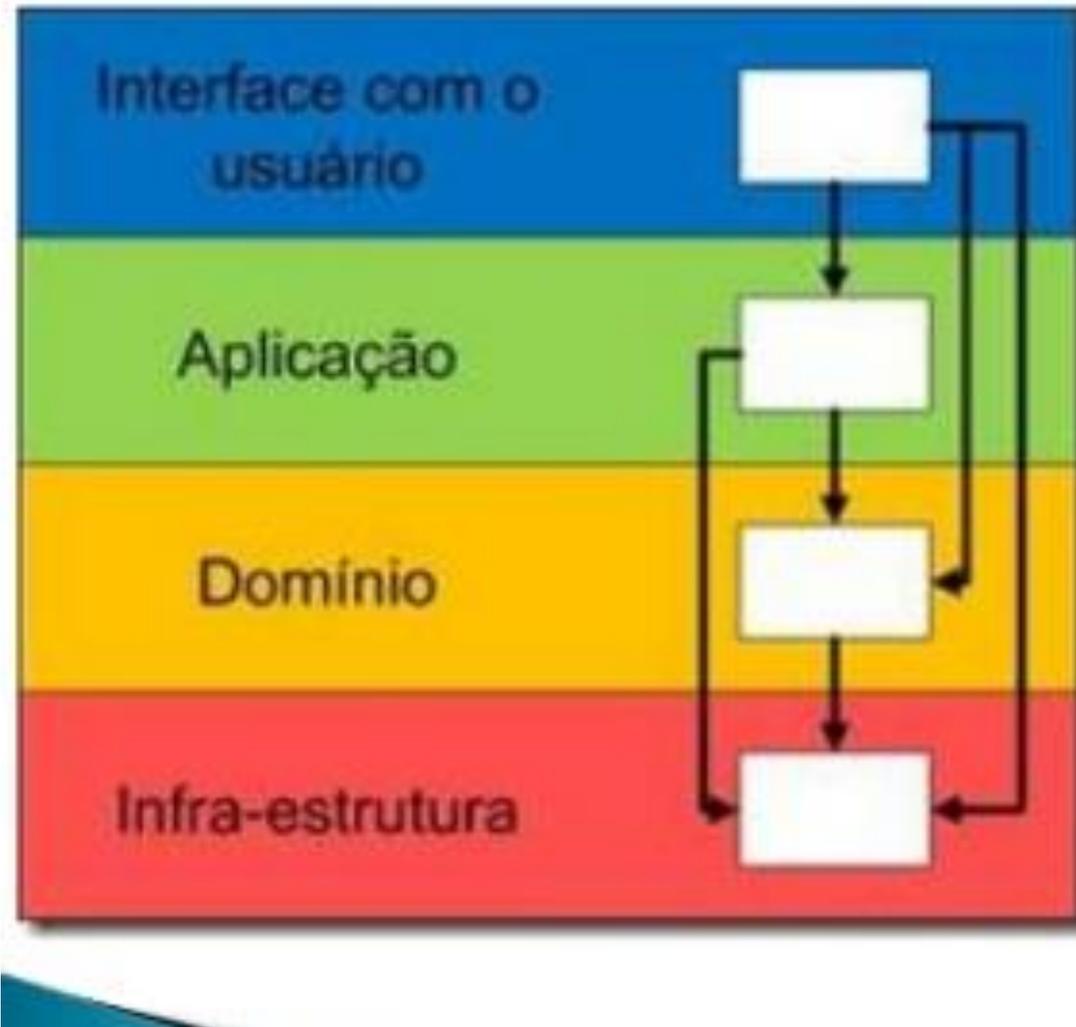
- Para facilitar o design, o MDD nos fornece estereótipos de objetos e artefatos que nos ajudam a representar, via código, os conceitos do domain.
- Alguns desses artefatos mapeiam diretamente os artefatos do domain, enquanto outros, cuidam dos ciclos de vida de artefatos do domain.
- Todo o projeto de software terá detalhes técnicos de níveis variados, como conexões com banco de dados e protocolos de rede, abstrações causadas por mistura de paradigmas (como no caso de ORMs).
- Esses detalhes continuarão a existir, porém devem ser separados da lógica de negócios.

# Os blocos do MDD

- O objetivo, é que o domain esteja isolado com o intuito de manter sua coerência, simplicidade e clareza.
- Camadas devem estar separadas.
- Separação de responsabilidades.
- Arquitetura flexível.



# Os blocos do MDD



Usuário pode ser outro sistema.  
Camada de **apresentação**

Coordena as atividades (workflow) da aplicação. Transformação de objetos (DTOs). Transação, auditoria e segurança.

“A camada de domínio é o coração do software”

Persistência de dados.  
Conceitos transversais: autenticação, autorização, cache, gerenciamento de exceções, log, validação.

O princípio essencial é que qualquer elemento de uma camada dependa apenas de outros elementos da mesma camada ou de camadas inferiores.

# Interface de Usuário

- Camada mais alta em abstração.
- Responsável por mostrar ao usuário a informação necessária e por interpretar as ações e os comandos emitidos pelo usuário.
- Nem sempre o usuário será uma pessoa, pode ser um sistema externo.



# Aplicação

- Tem a responsabilidade de adaptar as ações da camada inferior aos diversos tipos de UI.
- Nesta camada são tratadas as conversões e tradução de dados, além de detalhes de segurança nas operações.



# Aplicação

- Tem a responsabilidade de adaptar as ações da camada inferior aos diversos tipos de UI.
- Nesta camada são tratadas as conversões e tradução de dados, além de detalhes de segurança nas operações.



# Domínio

- É a camada em que os artefatos que expressam o domain model estão.
- É responsável por representar os conceitos de negócio, as informações de negócio e as regras do negócio.
- É o coração do software.



# Infraestrutura

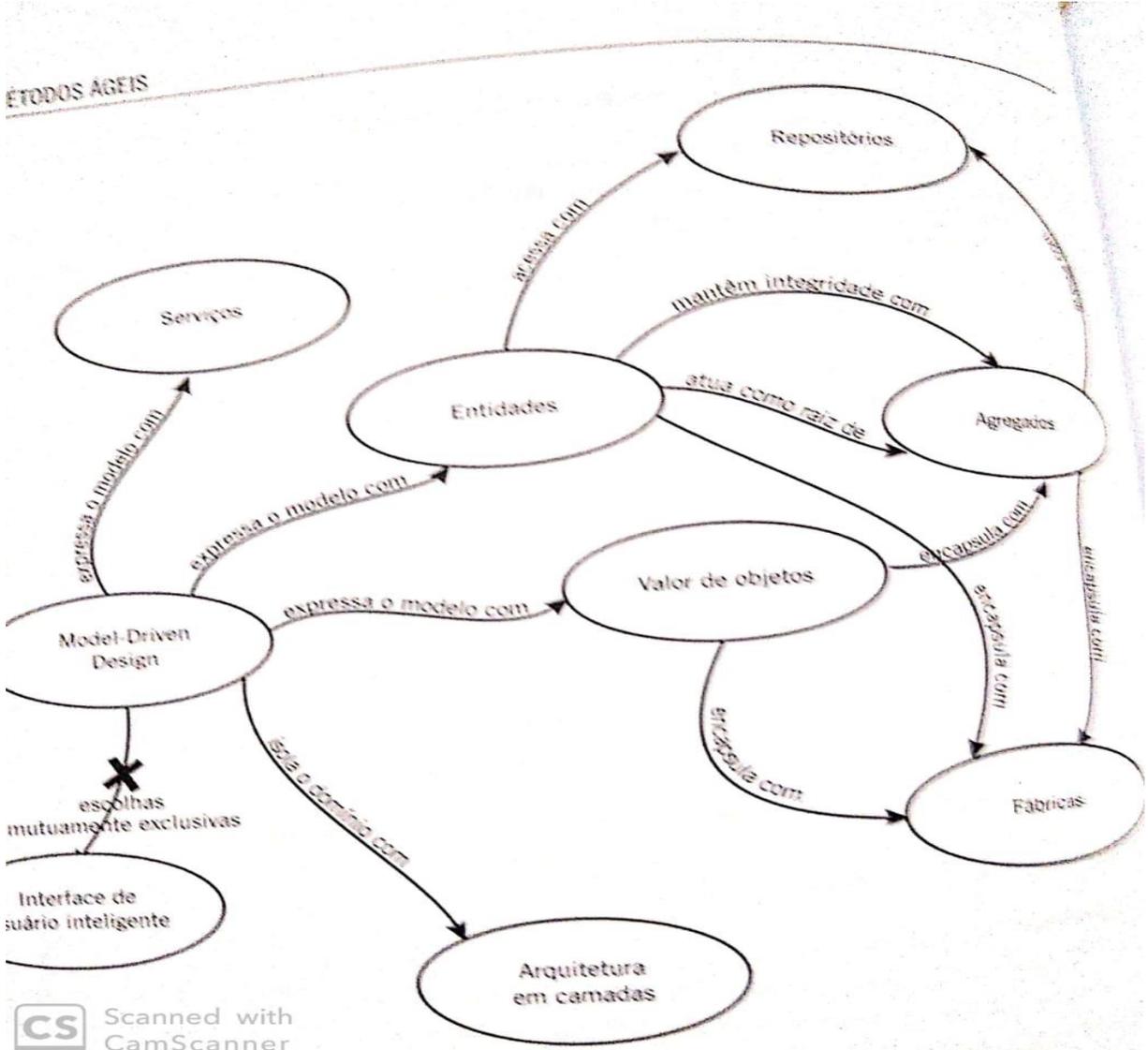
- Camada com mais baixo nível de abstração.
- Fornece suporte técnico e capacidades de mais baixo nível técnico às outras camadas, como tratamento de armazenamento, log e auditoria, etc.



# Domain Objects

- O MDD sugere alguns estereótipos de objetos para trabalharmos com linguagens orientada a objetos.
- O model-driven design é isolado através de uma arquitetura em camadas, sua camada correspondente é a camada de Domain, na qual os objetos referentes ao negócio estão.
- O MDD sugere alguns estereótipos para utilizarmos em nosso design. Podemos utilizar os conceitos de Entities, Value Objects e Services.

# Domain Objects



# Entities

- Semelhantemente a conotação orientação a objetos (Classe e Objetos).
- Entidades tem significado em um domínio.

Exemplos:

1. Cliente.
2. Carro.
3. Produto.

# Exemplo Entities

```
Cliente.cs + X
WorkshopDDD.Dominio.Cliente
namespace WorkshopDDD.Dominio
{
    public class Cliente
    {
        public int Id { get; set; }

        public string Nome { get; set; }

        public Endereco Endereco { get; set; }
    }
}
```

Entidade simples

```
Abstração para entidades
Entidade.cs + X
WorkshopDDD.Dominio.Entidade
using System;
namespace WorkshopDDD.Dominio
{
    public abstract class Entidade
    {
        public int Id { get; protected set; }

        public override bool Equals(object obj) {...}

        public override int GetHashCode() {...}

        public static bool operator ==(Entidade left, Entidade right) {...}

        public static bool operator !=(Entidade left, Entidade right) {...}
    }
}
```

# Value Objects

- Não tem identidade para o domínio, como tem uma entidade.
- São reconhecidos pelos seus atributos.
- Equivale a uma instância de uma classe.

Exemplos:

1. Cores.
2. Coordenadas.
3. Endereços.

# Exemplo Value Objects

Objeto de valor simples

```
WorkshopDDD.Dominio.Endereco
namespace WorkshopDDD.Dominio
{
    public class Endereco
    {
        public string Cidade { get; set; }
        public string Logradouro { get; set; }
        public int Numero { get; set; }
        public string Complemento { get; set; }
    }
}
```

```
WorkshopDDD.Dominio.ObjetoDeValor<T>
using System;
namespace WorkshopDDD.Dominio
{
    public class ObjetoDeValor<T> : IEquatable<T>
        where T : ObjetoDeValor<T>
    {
        public bool Equals(T other) {...}
        public override bool Equals(object obj) {...}
        public override int GetHashCode() {...}
        public static bool operator ==(ObjetoDeValor<T> left, ObjetoDeValor<T> right) {...}
        public static bool operator !=(ObjetoDeValor<T> left, ObjetoDeValor<T> right) {...}
    }
}
```

Abstração para  
objetos de valor

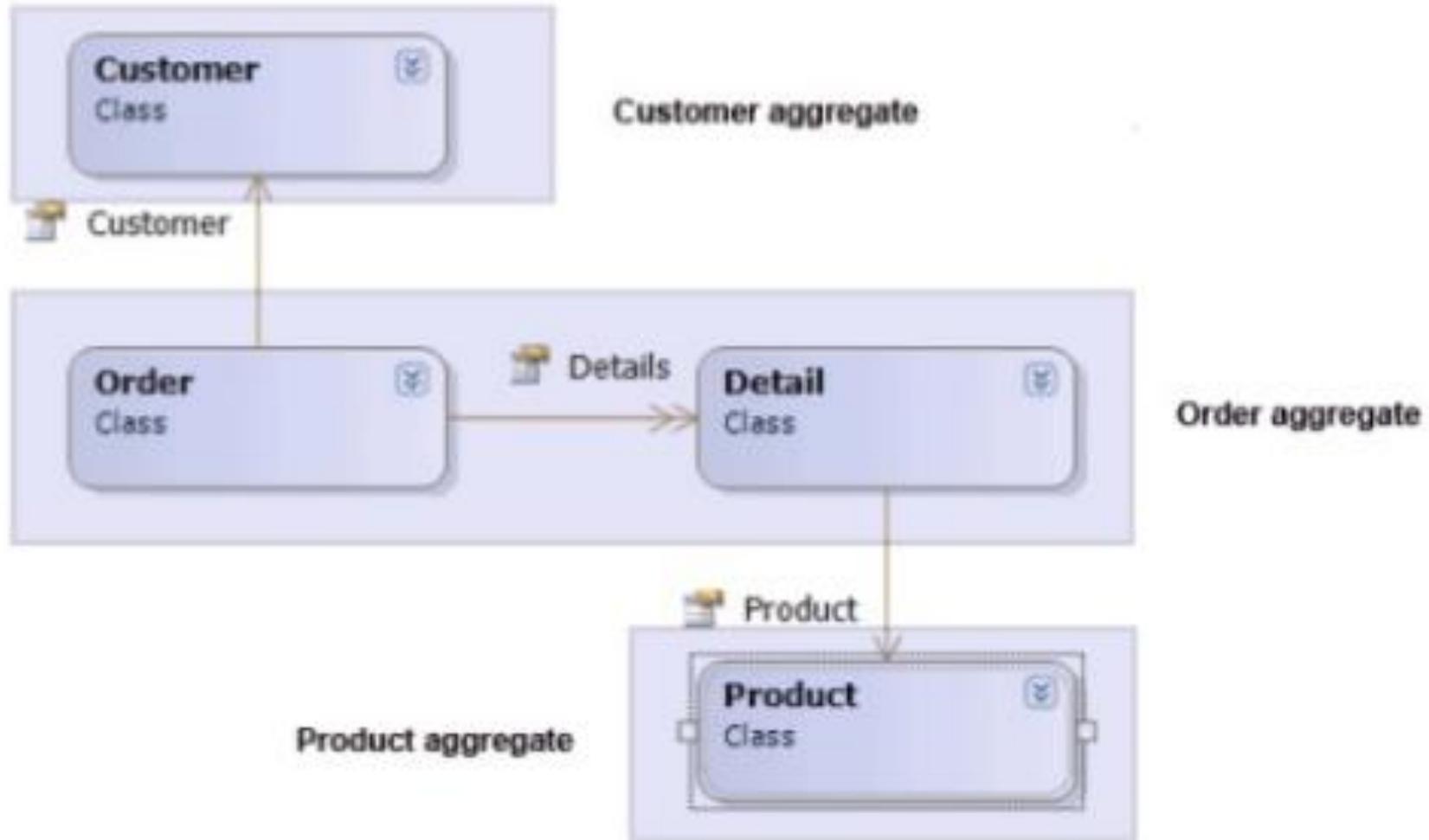
# Aggregates

- Reúne entidades e objetos de valor que fazem sentido no domínio.
- São delineadas pela raiz da agregação.

Regras:

1. Todas as atualizações passam pela raiz.
2. Todas as referências obtidas para os objetos recupera-se pela raiz.
3. Exclusão apaga todos os filhos da agregação.
4. Regras de negócio são garantidas pela raiz.

# Aggregates



# Serviços

- Resolvem problemas de negócio, mas não são entidades ou objetos de valor.

Um bom serviço tem três características:

1. A operação é relacionada a um conceito do domínio que não é naturalmente parte de uma entidade ou objeto de valor.
2. A interface é baseada em outros objetos do modelo do domínio.
3. A operação é Stateless.

# Serviços

```
WorkshopDDD.Dominio.ServicoDeDescontos
using System.Linq;

namespace WorkshopDDD.Dominio
{
    public class ServicoDeDescontos : IServicoDeDescontos
    {
        public decimal CalcularDesconto(Venda venda)
        {
            if (venda.Cliente.Compras.Where(c => c.ValorTotal >= 1000).Count() > 10)
                return 0.20m; // 20%

            if (venda.ValorTotal >= 2000)
                return 0.15m; // 15%

            if (venda.ValorTotal >= 1000)
                return 0.1m; // 10%

            return 0;
        }
    }
}
```

# Fábricas

- A criação de um domain object pode ser simples ou complexa. O MDD sugere a aplicação do padrão Factory para encapsular essa operação.
- Criam objetos de negócio.
- É responsabilidade de um objeto construir a si mesmo?
- Regras complexas para construção.
- Fábrica cria objetos consistentes.

# Fábricas

```
FabricaDeClientes.cs [X]
WorkshopDDD.Dominio.FabricaDeClientes
namespace WorkshopDDD.Dominio
{
    public static class FabricaDeClientes
    {
        public static Cliente CriarCliente(string nome, Endereco endereco)
        {
            var cliente = new Cliente();

            cliente.Nome = nome;
            cliente.Endereco = endereco;

            return cliente;
        }
    }
}
```

# Repositório

- Tem como objetivo encapsular e esconder a complexidade dos detalhes técnicos da persistência de dados.
- Representam uma lista de itens em memória.
- Não tem lógica de negócio. No máximo valida uma consistência do objeto.
- Geralmente, são especificados para Agregações e não para Entidades.

# Repositório

```
RepositorioDeClientes.cs # X
• WorkshopDDD.Dominio.IRepositorioDeClientes
using System.Collections.Generic;

namespace WorkshopDDD.Dominio
{
    public interface IRepositorioDeClientes
    {
        IEnumerable<Cliente> ObterTodos();

        Cliente ObterPeloId(int id);

        void Armazenar(Cliente cliente);

        void Remover(int id);
    }
}
```

# Repositório - Entity Framework

```
RepositorioDeClientesEF.cs x
WorkshopDDD.Infraestrutura.Dados.RepositorioDeClientesEF

using System;
using System.Collections.Generic;
using System.Linq;
using WorkshopDDD.Dominio;

namespace WorkshopDDD.Infraestrutura.Dados
{
    public class RepositorioDeClientesEF : IRepositoryDeClientes
    {
        public IEnumerable<Cliente> ObterTodos()
        {
            using (var context = ConnectionFactory.CreateContext())
            {
                return context.Set<Cliente>().AsEnumerable();
            }
        }

        public Cliente ObterPeloId(int id) {...}

        public void Armazenar(Cliente cliente) {...}

        public void Remover(int id) {...}
    }
}
```

**Q1) [CESPE STJ 2015]** Acerca de arquitetura de software e Domain-Driven Design, julgue o seguinte item.

Domain-Driven Design pode ser aplicada ao processo de concepção arquitetural de um sistema de software, sendo que domain, em um software, designa o campo de ação, conhecimento e influência.

**Q2) [CESPE SLU DF 2019]** Julgue o próximo item, a respeito de domain-driven design, design patterns, emergent design, enterprise content management e REST.

No desenvolvimento embasado em domain-driven design, a definição da tecnologia a ser utilizada tem importância secundária no projeto.

**Q1) [CESPE STJ 2015]** Acerca de arquitetura de software e Domain-Driven Design, julgue o seguinte item.

Domain-Driven Design pode ser aplicada ao processo de concepção arquitetural de um sistema de software, sendo que domain, em um software, designa o campo de ação, conhecimento e influência. CERTO.

**Q2) [CESPE SLU DF 2019]** Julgue o próximo item, a respeito de domain-driven design, design patterns, emergent design, enterprise content management e REST.

No desenvolvimento embasado em domain-driven design, a definição da tecnologia a ser utilizada tem importância secundária no projeto. CERTO.

**Q3) [CESPE MPC PA 2019]** No Domain-Driven Design, a Ubiquitous Language é considerada

a) uma linguagem de programação utilizada pelos desenvolvedores para escrever os códigos.

b) uma linguagem de modelagem utilizada pelos analistas para representar os processos de negócio.

c) uma linguagem do projeto de desenvolvimento de software utilizada para comunicação de todos os envolvidos no projeto.

d) uma linguagem de notação utilizada pelos arquitetos para representar as funcionalidades do software.

e) uma linguagem de semântica utilizada pelos especialistas para definir as especificações de negócio.

**Q3) [CESPE MPC PA 2019]** No Domain-Driven Design, a Ubiquitous Language é considerada

a) uma linguagem de programação utilizada pelos desenvolvedores para escrever os códigos.

b) uma linguagem de modelagem utilizada pelos analistas para representar os processos de negócio.

c) uma linguagem do projeto de desenvolvimento de software utilizada para comunicação de todos os envolvidos no projeto.

d) uma linguagem de notação utilizada pelos arquitetos para representar as funcionalidades do software.

e) uma linguagem de semântica utilizada pelos especialistas para definir as especificações de negócio.

**Q4) [CESPE STJ 2015]** Julgue o próximo item, relativo a Domain-Driven Design e design patterns.

Um dos princípios-chave do Domain-Driven Design é o uso de uma linguagem ubíqua com termos bem definidos, que integram o domínio do negócio e que são utilizados entre desenvolvedores especialistas de negócio.

**Q5) [CESPE MPE PI 2018]** Julgue o item subsequente, referente a Domain Driven Design e a Design Patterns.

No Domain Driven Design, o projeto de software baseia sua reação em eventos externos e internos, tendo como premissa uma quantidade finita de estados que enfatizam a separação entre os modelos abstratos independentes de implementação e os específicos de implementação.

**Q4) [CESPE STJ 2015]** Julgue o próximo item, relativo a Domain-Driven Design e design patterns.

Um dos princípios-chave do Domain-Driven Design é o uso de uma linguagem ubíqua com termos bem definidos, que integram o domínio do negócio e que são utilizados entre desenvolvedores especialistas de negócio. CERTO.

**Q5) [CESPE MPE PI 2018]** Julgue o item subsequente, referente a Domain Driven Design e a Design Patterns.

No Domain Driven Design, o projeto de software baseia sua reação em eventos externos e internos, tendo como premissa uma quantidade finita de estados que enfatizam a separação entre os modelos abstratos independentes de implementação e os específicos de implementação. ERRADO.

**Q6) [CESPE TRE PE 2017]** O DDD (domain-driven design)

- a) consiste em uma técnica que trata os elementos de domínio e que garante segurança à aplicação em uma programação orientada a objetos na medida em que esconde as propriedades desses objetos.
- b) não tem como foco principal a tecnologia, mas o entendimento das regras de negócio e de como elas devem estar refletidas no código e no modelo de domínio.
- c) prioriza a simplicidade do código, sendo descartados quaisquer usos de linguagem ubíqua que fujam ao domínio da solução.
- d) constitui-se de vários tratadores e(ou) programas que processam os eventos para produzir respostas e de um disparador que invoca os pequenos tratadores.
- e) define-se como uma interface de domínio normalmente especificada e um conjunto de operações que permite acesso a uma funcionalidade da aplicação.

**Q6) [CESPE TRE PE 2017]** O DDD (domain-driven design)

a) consiste em uma técnica que trata os elementos de domínio e que garante segurança à aplicação em uma programação orientada a objetos na medida em que esconde as propriedades desses objetos.

b) não tem como foco principal a tecnologia, mas o entendimento das regras de negócio e de como elas devem estar refletidas no código e no modelo de domínio.

c) prioriza a simplicidade do código, sendo descartados quaisquer usos de linguagem ubíqua que fujam ao domínio da solução.

d) constitui-se de vários tratadores e(ou) programas que processam os eventos para produzir respostas e de um disparador que invoca os pequenos tratadores.

e) define-se como uma interface de domínio normalmente especificada e um conjunto de operações que permite acesso a uma funcionalidade da aplicação.

**Q7) [CESPE TRT 8]** Assinale a opção correta com relação à modelagem orientada a domínio (DDD – domain driven design).

a) Fábricas são classes que contêm a lógica do negócio, mas que não pertencem a nenhuma entidade ou objeto de valores.

b) O uso de DDD será aplicável quando o software atender uma área de negócio muito específica e complexa.

c) DDD oferece uma série de conceitos e padrões que auxiliam o desenvolvedor no projeto da solução, exclusivamente no nível estratégico.

d) Capacidade de desenvolvimento iterativo e regras de negócio simples são requisitos básicos para a aplicação efetiva da modelagem DDD.

e) DDD utiliza os mesmos conceitos e premissas do processo de análise e projeto em orientação a objetos.

**Q7) [CESPE TRT 8]** Assinale a opção correta com relação à modelagem orientada a domínio (DDD – domain driven design).

a) Fábricas são classes que contêm a lógica do negócio, mas que não pertencem a nenhuma entidade ou objeto de valores.

b) O uso de DDD será aplicável quando o software atender uma área de negócio muito específica e complexa.

c) DDD oferece uma série de conceitos e padrões que auxiliam o desenvolvedor no projeto da solução, exclusivamente no nível estratégico.

d) Capacidade de desenvolvimento iterativo e regras de negócio simples são requisitos básicos para a aplicação efetiva da modelagem DDD.

e) DDD utiliza os mesmos conceitos e premissas do processo de análise e projeto em orientação a objetos.

**Q8) [ESAF 2015]** O Domain-Driven Design – DDD utiliza o conceito de divisão do sistema em camadas. As camadas desse modelo são:

a) aplicação, apresentação, sessão, transporte, rede, enlace e física.

b) de apresentação, de negócio e de dados.

c) do modelo, da visualização e de controle..

d) domínio de usuário, domínio de negócio e domínio de dados.

e) Interface com usuário, aplicação, domínio e infraestrutura.

**Q8) [ESAF 2015]** O Domain-Driven Design – DDD utiliza o conceito de divisão do sistema em camadas. As camadas desse modelo são:

a) aplicação, apresentação, sessão, transporte, rede, enlace e física.

b) de apresentação, de negócio e de dados.

c) do modelo, da visualização e de controle..

d) domínio de usuário, domínio de negócio e domínio de dados.

e) Interface com usuário, aplicação, domínio e infraestrutura.

# GABARITO

**Q1 – CERTO.**

**Q2 - CERTO.**

**Q3 – LETRA C.**

**Q4 – CERTO.**

**Q5 - ERRADO.**

**Q6 - LETRA B.**

**Q7 – LETRA B.**

**Q8 – LETRA E.**