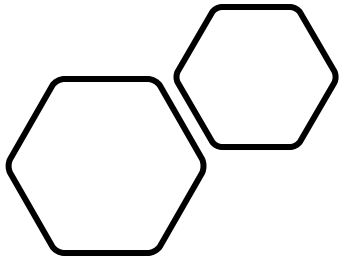
Projeto baseado em domínio

Prof. Bernardo Copstein

Entendendo DDD (Domain Driven Design)



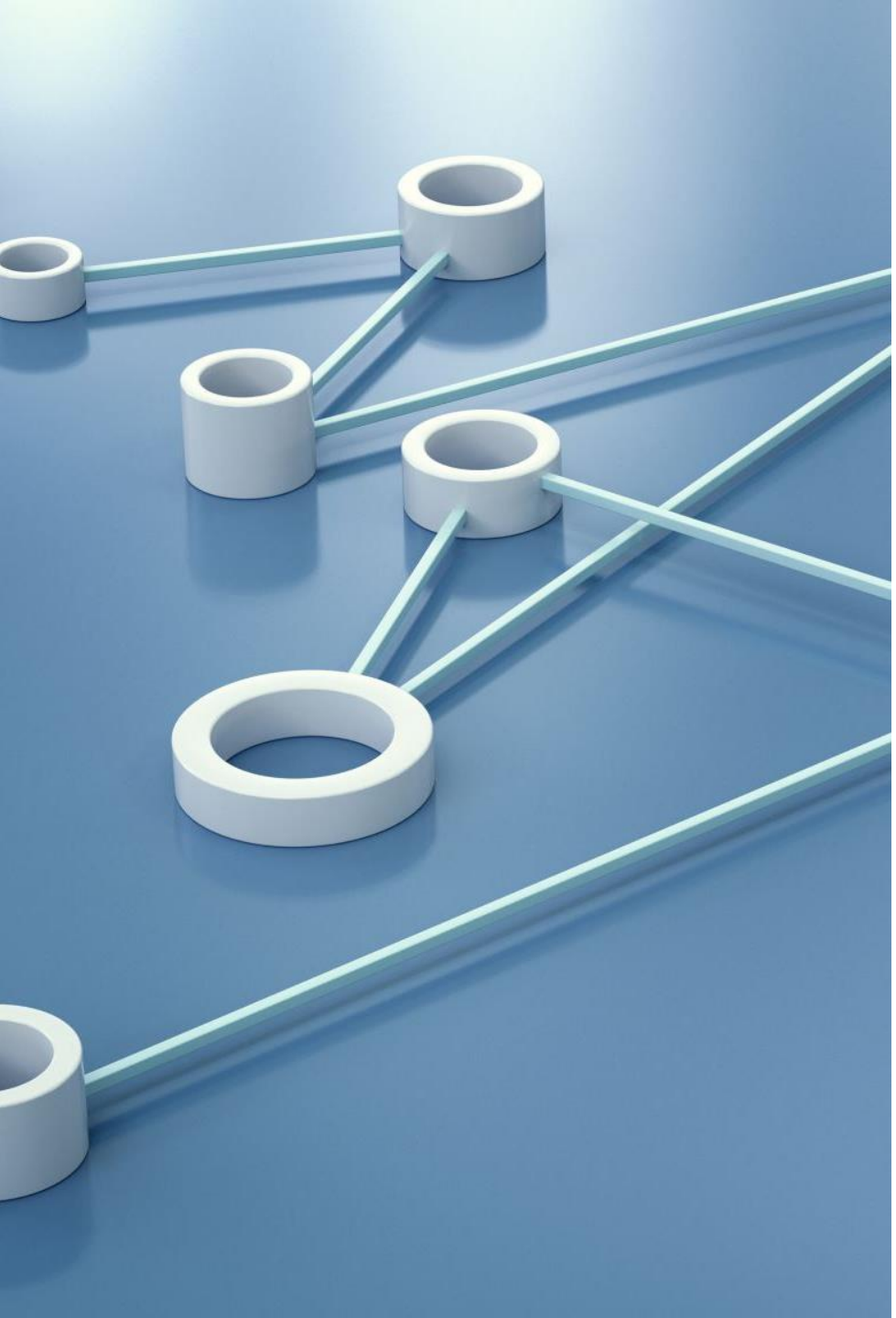
Leituras recomendadas:

- Domain Driven Design Quickly

  By: Abel Avram & Floyd Marinescu

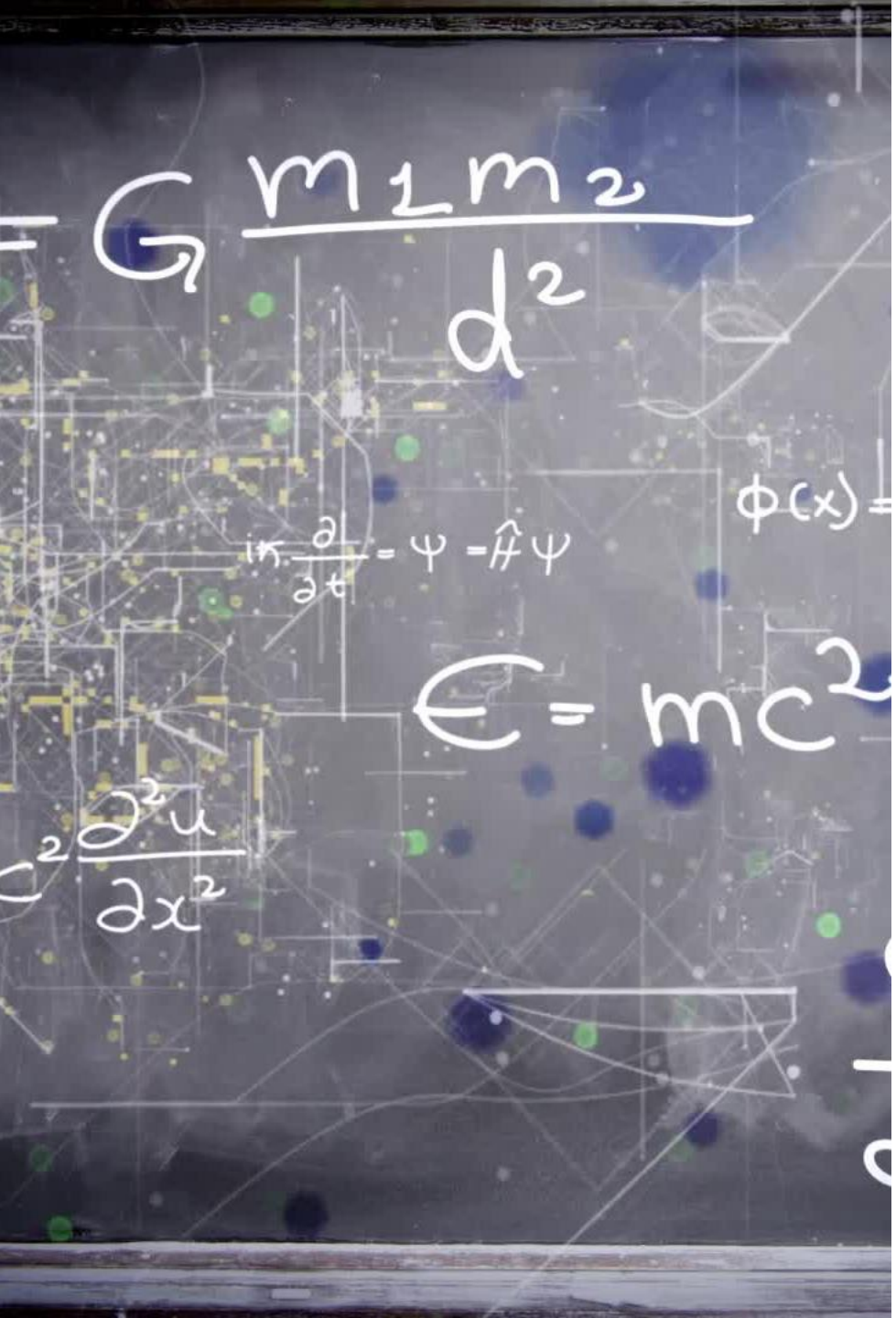
O modelo de camadas clássico

- O modelo de camadas promove projeto orientado pelo BD
- A estrutura de dependências do modelo nos leva a desenvolver primeiro o banco de dados já que tudo é construído a partir dele.
- Isso pode fazer sentido em função do fluxo de dependências, mas não faz nenhum sentido do ponto de vista da lógica de negócios



Passo 1: conhecer o domínio

- No contexto de DDD, o primeiro passo para se construir software é entender o domínio da aplicação
  - Não é possível desenvolver um sistema de compra e venda de ações sem entender um pouco do mercado financeiro
  - Não é possível desenvolver um sistema de controle de tráfego aéreo sem conhecer o jargão da área
  - Desenvolver uma “linguagem de domínio” facilita a compreensão dos artefatos por parte de todos os membros da equipe



Exemplo: sistema acadêmico da PUCRS

A “linguagem do domínio” certamente irá incluir as seguintes expressões:

- Sistema semestral seriado
- Sistema de créditos
- PPC: Projeto Pedagógico de Curso
- Grade curricular
- Atividades curriculares (disciplinas, estágios, TCC etc)
- Turma
- Grau G1
- Grau G2
- Tecnólogos de nível superior, bacharelados, mestrados, doutorados
- etc

Criando abstrações

- Para converter o domínio em código é necessário criar abstrações
  - Abstrações que representem os elementos do domínio no contexto das necessidades do sistema sendo desenvolvido
  - Abstrações são modelos
  - Modelo é uma representação do domínio alvo
- Diagramas de classe são uma boa forma de começar a representar e organizar estas abstrações
- Devem ser completados por outros documentos capazes de expressar o comportamento dessas entidades

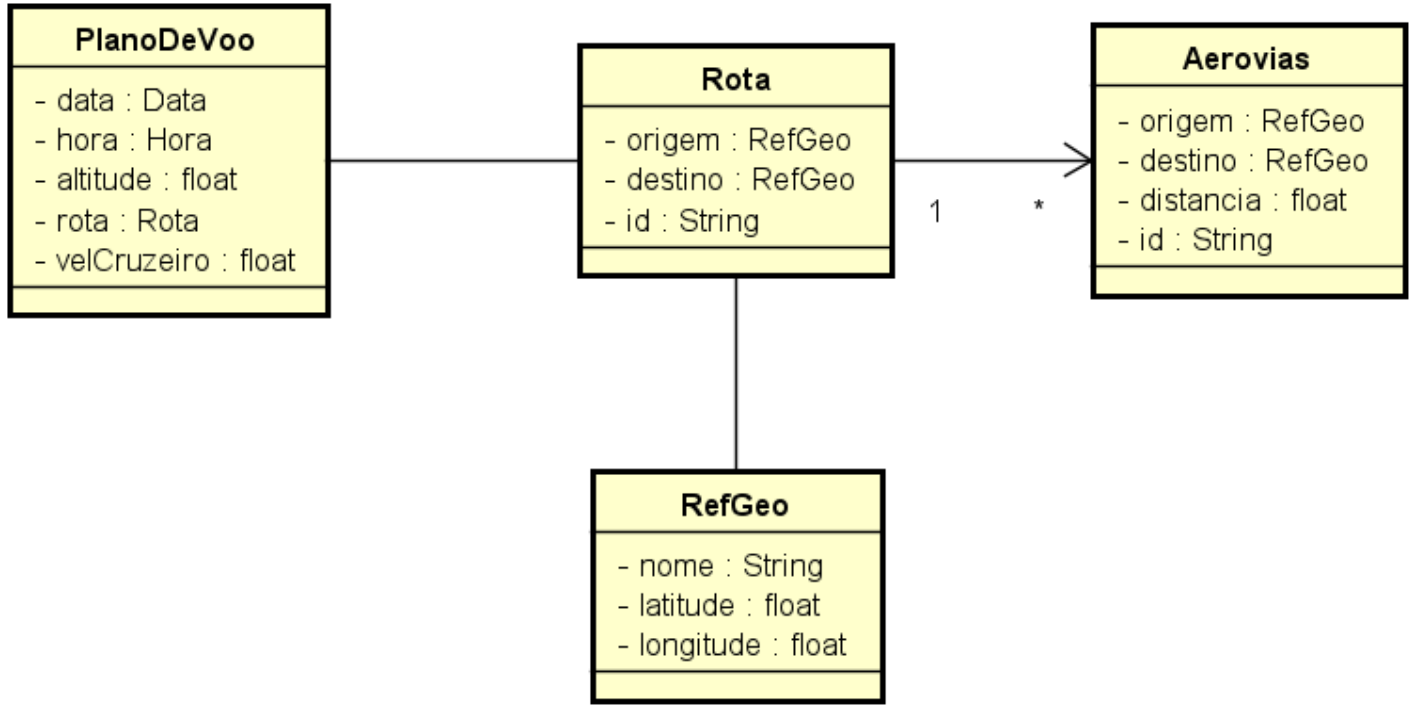
Exemplo: sistema de seleção de rotas de tráfego aéreo

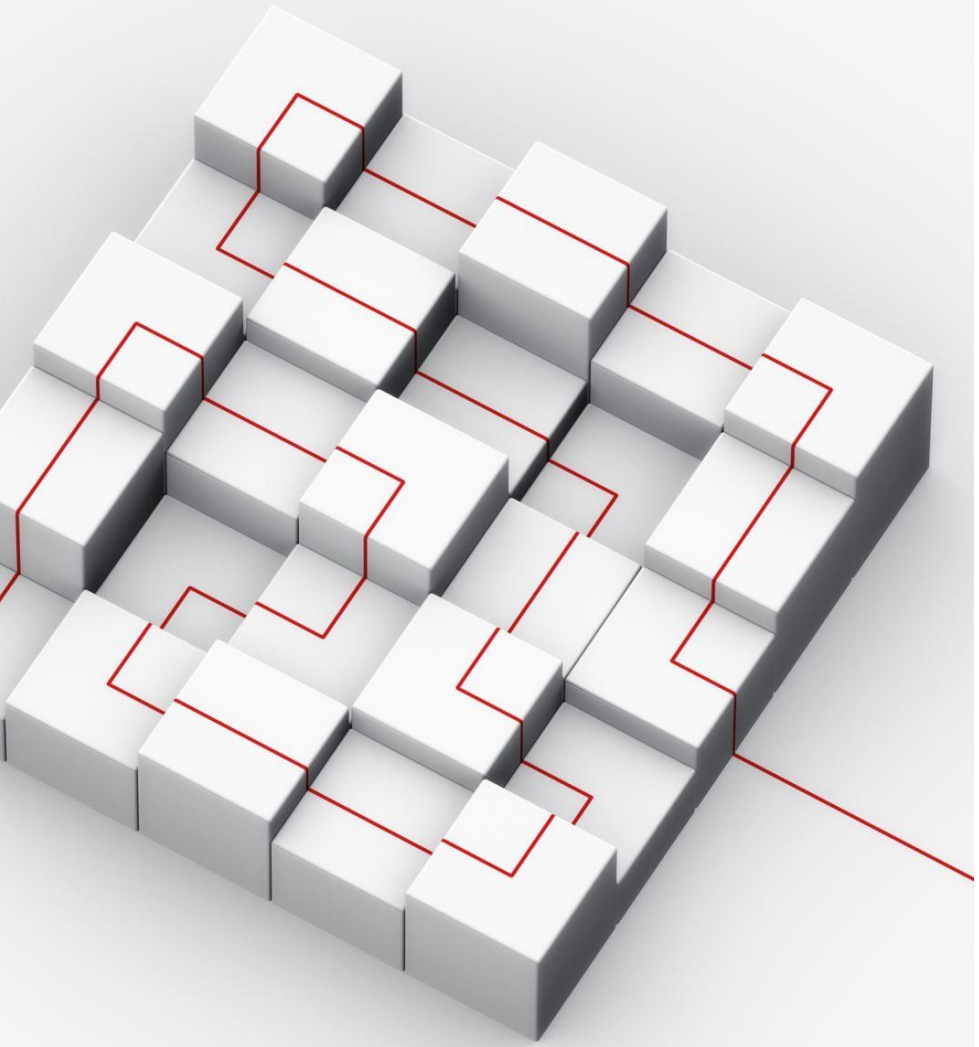
- Os aviões não andam livres pelos céus. Eles se mantém em **aerovias**
- Uma aerovia é um tipo de corredor virtual que delimitam a trajetória e a altitude que as aeronaves devem seguir num terminado espaço aéreo (ver figura). As diferentes altitudes permitem que mais de um avião utilize a mesma aerovia ao mesmo tempo.
- Aerovias ligam **coordenadas geográficas**. Uma ou mais aerovias compõem uma **rota**. Aeroportos são ligados por uma ou mais rotas distintas.
- Um **plano de voo** define a data e o horário de partida do voo, a rota que será usada, e as altitudes usada em cada aerovia que compõe a rota.

Os conceitos apresentados neste exemplo são apenas inspirados nos conceitos reais

Usando diagramas de classe para modelar o domínio

- Diagramas de classe são um bom ponto de partida para representar as abstrações do domínio
- Muitas vezes precisam ser complementados por outros artefatos capazes de descrever o comportamento das entidades





Domain Driven Design (1)

- Uma aplicação pode ser modelada de várias formas e um modelo pode ser implementado de diferentes maneiras
- DDD propõe uma forma de se alcançar este objetivo:
  - O primeiro passo é construir um modelo do domínio sem se preocupar com o sistema sendo desenvolvido
  - O objetivo deste modelo é entender o domínio
    - Naturalmente algum conhecimento do sistema pretendido é importante para estabelecer limites no modelo
  - Este modelo deve ser construído em conjunto com a equipe de desenvolvimento de maneira que certas decisões já irão considerar aspectos de implementação
  - **A ideia é construir um modelo que possa ser adequadamente expresso em software**

Domain Driven Design (2)

O sistema a ser desenvolvido pode ser modelado a partir dos casos de uso tendo como estrutura básica o modelo do domínio

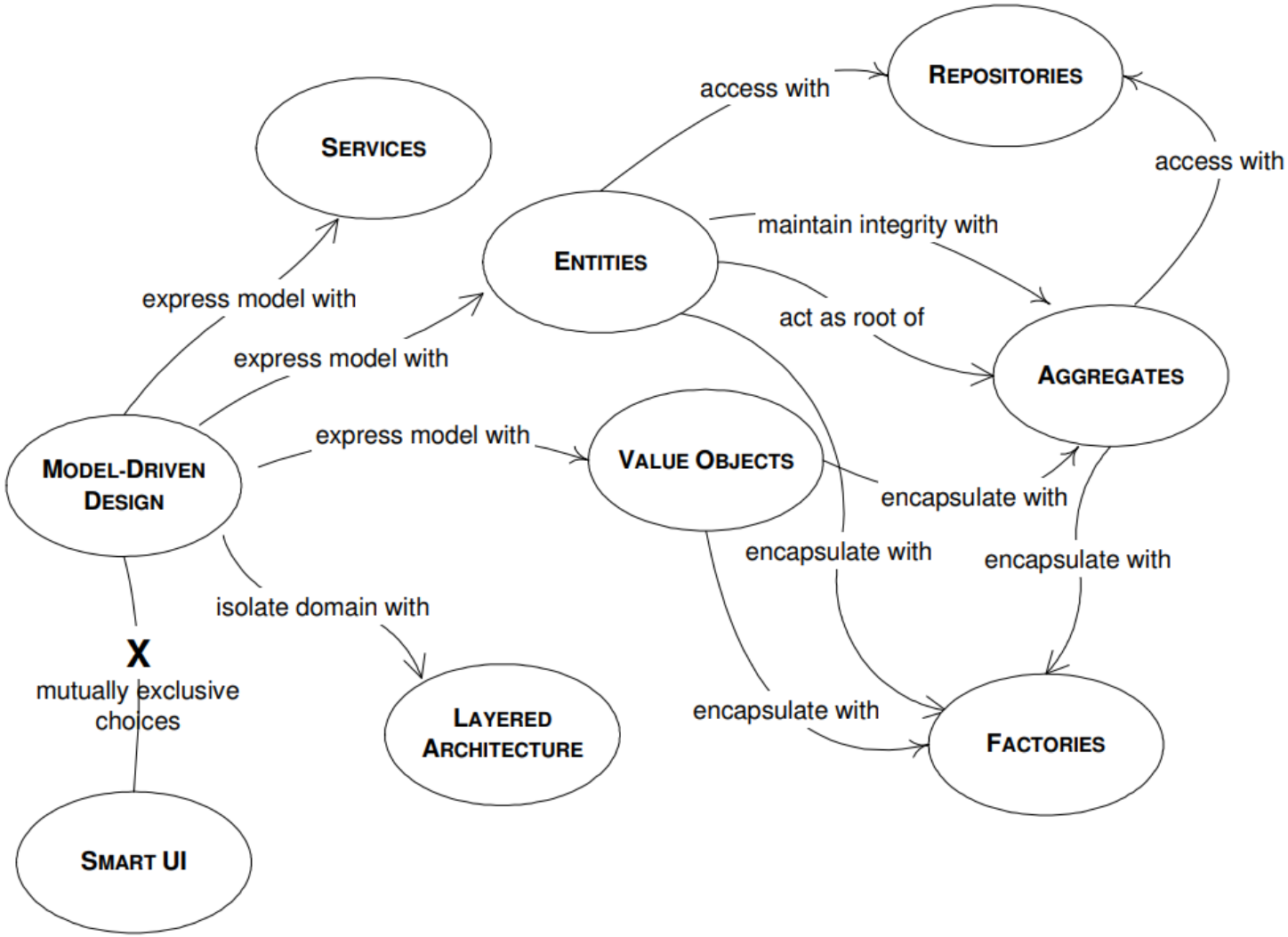
O modelo do sistema é construído “ao redor” do modelo do domínio

A “linguagem” do domínio deve ser usada durante todo o tempo

Linguagens orientadas a objetos são as mais adequadas para a implementação de tais modelos visto que ambos se valem dos mesmos conceitos

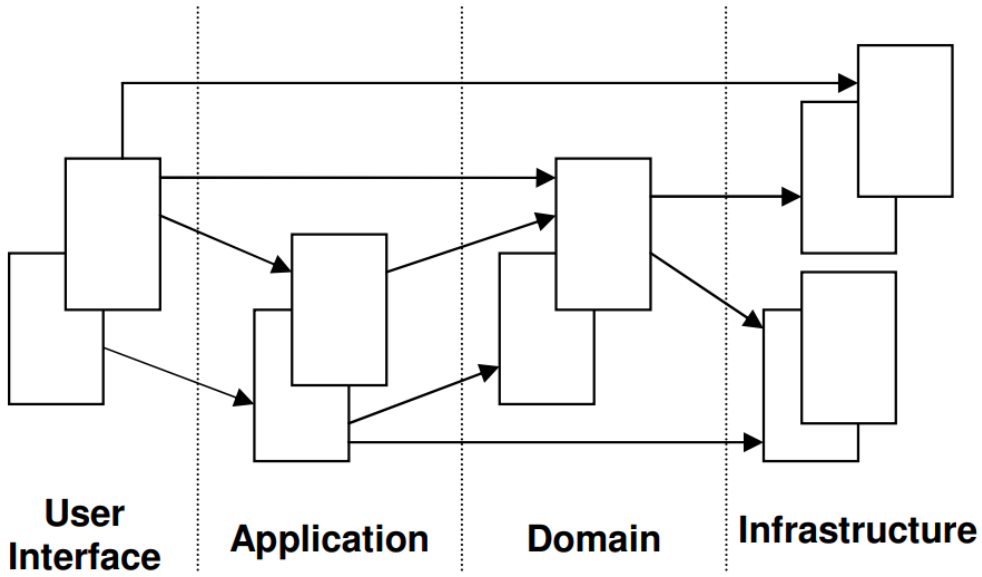
Os “blocos de montar” do DDD

- Estes são os principais padrões a serem usados em “model driven design”
- O propósito destes padrões é apresentar os elementos chaves da modelagem de software orientada a objetos pela perspectiva de DDD



Arquitetura em camadas

- Quando se cria software é normal que grande parte dele não tenha relação com a modelagem do domínio
- Em função disso é normal misturar o código de acesso ao BD ou o código da GUI com os elementos do domínio
- Isso torna o código difícil de testar e manter
- DDD sugere que:
  - O software seja organizado em camadas coesas
  - Cada camada deve depender apenas das camadas inferiores
  - Usam-se os padrões arquiteturais sugeridos para manter o acoplamento entre as classes o menor possível



Organização das camadas

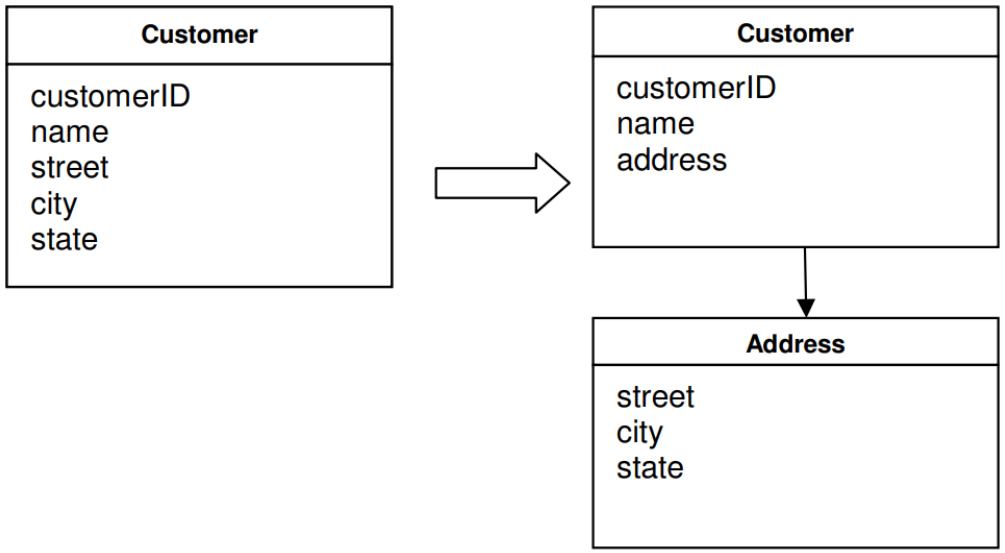
Camada	Responsabilidades
Apresentação	Apresentar a informação para o usuário e interpretar os comandos do usuário • Como fica isso em um backend?
Aplicação	Coordenar a aplicação. Não contém a lógica de negócio e não mantem o estado dos objetos de negócio, mas pode manter o estado de progresso das tarefas da aplicação
Domínio	Contém à informação sobre o domínio. Mantém o estado dos objetos de domínio. Conhece as regras de negócio do domínio. A persistência de seus objetos e do seu estado é delegada para a camada de infraestrutura.
Infraestrutura	Esta camada age como suporte para todas as demais. Provê a comunicação entre as camadas, implementa a persistência dos objetos de negócio, contém bibliotecas de suporte para a interface com o usuário etc.

Entidades

- Modelam os elementos fundamentais do domínio
- Normalmente são as primeiras a serem modeladas
- Entidades tem uma identidade, um identificador próprio
  - Pessoa → CPF
  - Aeroporto → sigla de identificação única
  - Automóvel → RENAVAM
  - Conta bancária → número

Value Objects

- Entidades tem uma identidade única e isso tem um custo:
  - Garantir que cada identificador seja único
- Nem todos os elementos de um modelo necessitam ter uma identidade única:
  - Ponto no plano
  - Data
  - Placa de carro
  - Endereço
- Na verdade toda a abstração que não tem uma identidade única pode ser modelada como um “value object” (VO).
- Muitas vezes VOs são usados para caracterizar conceitos dentro de entidades sem que tenham de ser modelados eles mesmos como entidades



Serviços

- As ações relacionadas a uma entidade compõe o comportamento daquele tipo de entidade
- Algumas ações do domínio, entretanto, não estão relacionadas com uma entidade única. Exemplos:
  - Transferência entre contas correntes
  - Seleção da melhor rota para uma encomenda
  - Recomendação de um filme para certo tipo de cliente
- Estas ações são melhor organizadas em “serviços”. Tais objetos não tem estado interno, e seu propósito é simplesmente prover funcionalidades para o domínio. Um serviço pode agrupar funcionalidades úteis para entidades e “value objects”. É sempre uma vantagem declarar os serviços explicitamente porque eles modelam um conceito.

Características dos serviços

- Características inerentes:
  1. As operações executadas por um serviço referem-se a um conceito do domínio que naturalmente não pertence a uma entidade ou VO
  2. A operação executada envolve os objetos do domínio
  3. A operação não tem estado
- Serviços podem aparecer em todas as camadas, mas é importante garantir o isolamento do domínio. Nem sempre é fácil decidir a que camada pertence um serviço

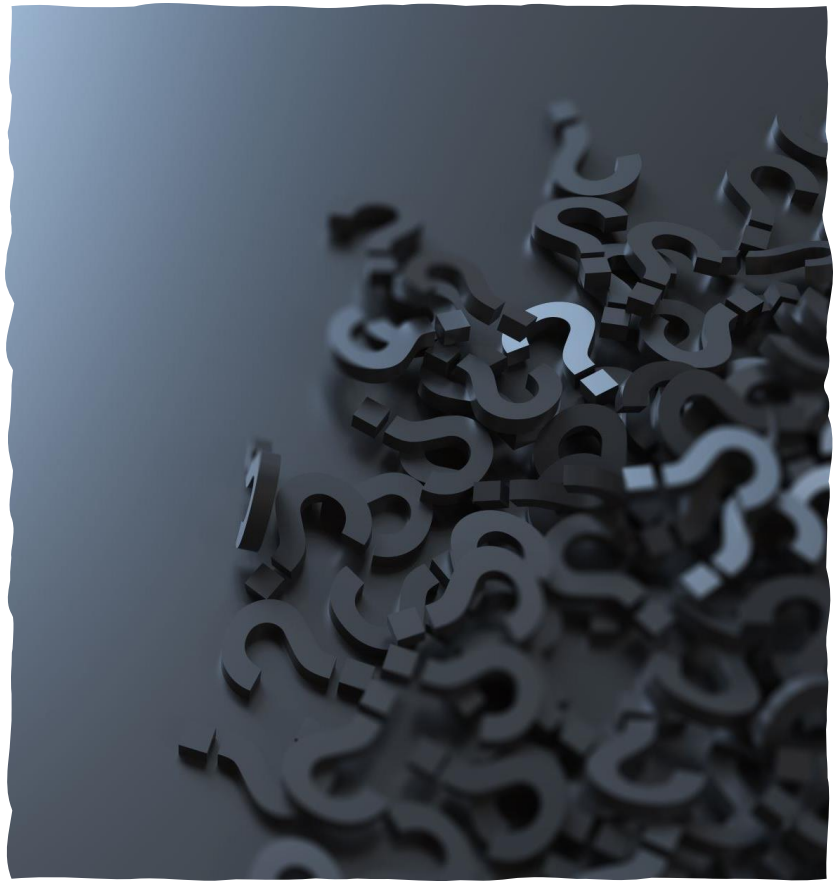
Exemplo:

- Em uma aplicação bancária as entidades principais são o cliente e a conta corrente.
- Informações de interesse para a gestão das contas tais como saldo médio da conta corrente, gasto médio mensal no cartão de crédito e pontualidade são agrupadas em um serviço de estatísticas na camada de domínio. Este serviço pertence ao domínio porque modela estatísticas típicas deste.
- Já a seleção de clientes com base em um conjunto de características pode ser parte de um serviço da camada de aplicação, considerando uma aplicação de gestão de clientes. A forma de seleção não é característica do domínio, sendo específica desta aplicação.

Módulos

- Quando a aplicação fica muito grande para que se possa falar dela como um todo está na hora de organizar ela em módulos
- Módulos podem agrupar classes com grande acoplamento tornando essas regiões fáceis de identificar. Dessa forma pode-se aumentar a coesão dos módulos e diminuir o acoplamento entre eles, facilitando a manutenção do sistema.
- O nome dos módulos também deve usar a linguagem do domínio

Padrões relacionados a Entidades



- Objetos de domínio passam por uma série de estados durante sua existência. São criados, alocados na memória, usados em computações diversas e então destruídos. Em alguns casos são armazenadas em repositórios permanentes tais como um banco de dados de onde podem ser recuperados mais tarde. Em algum momento podem ser completamente apagados do sistema, incluindo dos repositórios permanentes.
- Gerenciar o ciclo de vida de um objeto de domínio constitui um desafio em si mesmo, e se não for feito da maneira adequada pode ter um impacto negativo no modelo do domínio. Na sequencia serão apresentados 3 padrões que auxiliam neste desafio:
  - Aggregate é um padrão usado para ajudar a definir a propriedade do objeto e seus limites
  - Factory é um padrão usado na criação de objetos
  - Repository é um padrão que nos auxilia a lidar com o armazenamento dos objetos

Aggregate (1)

- Os relacionamentos entre os objetos de domínio podem ser simples ou complexos, mas um ou mais mecanismos de implementação terão de suporta-los e esse suporte pode variar ao longo do ciclo de vida:
  - Relacionamentos “um para um” podem ser representados por uma referência enquanto o objeto está em memória, mas terá de ser traduzido em chaves estrangeiras no banco de dados no momento da persistência.
  - Da mesma forma relacionamentos “um para muitos” ou “muitos para muitos” irão implicar no uso de coleções na memória e, eventualmente, tabelas adicionais no banco de dados.
- De forma a garantir que essa complexidade seja gerenciável é importante manter o modelo o mais simples possível.

Aggregate (2)

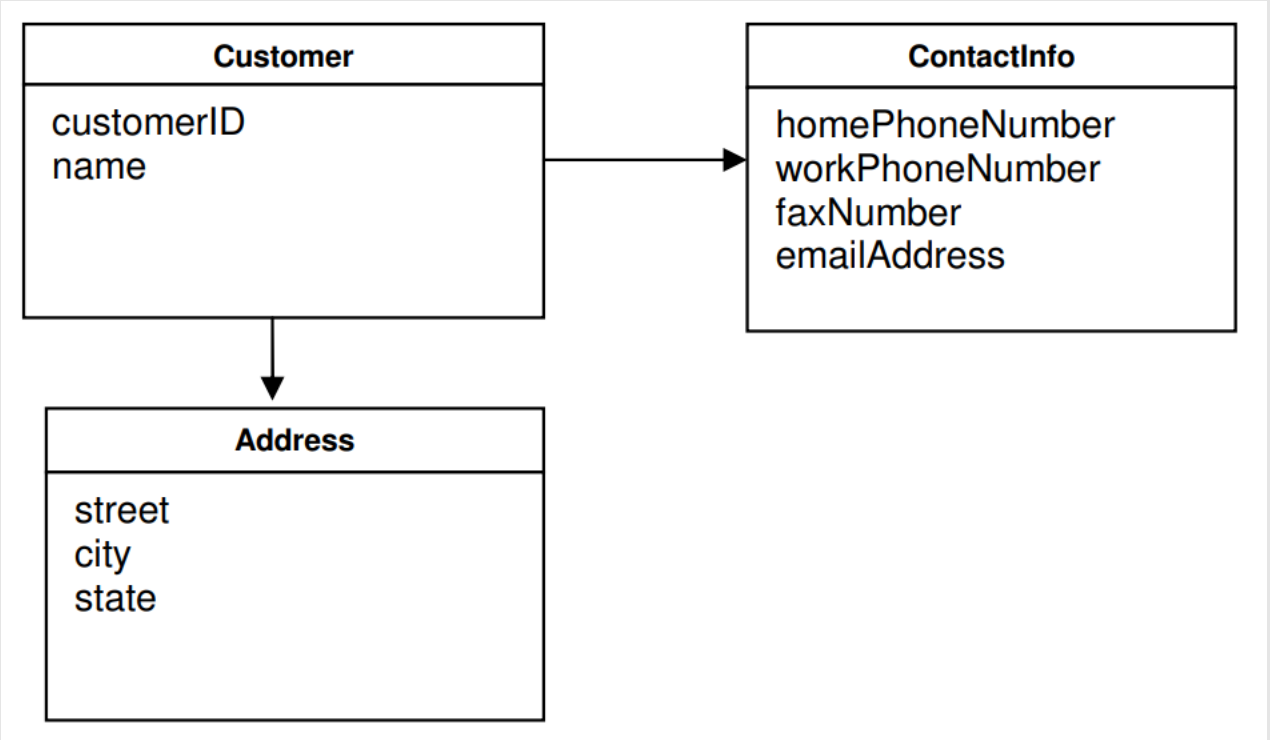
- Entidades podem ser modeladas apenas com atributos escalares ou podem usar VOs para deixar certos conceitos mais bem definidos. Além disso podem conter relacionamentos para outras entidades.
- Manter a integridade do conjunto de relacionamentos e objetos que compõe uma entidade é um desafio em si mesmo. O padrão aggreate busca auxiliar nesta solução.

Aggregate (3)

- Um “Aggregate” é um grupo de objetos associados que são considerado um só. É demarcado por um limite que separa os objetos “de dentro” dos “de fora”. Cada Aggregate tem uma raiz. A raiz é uma Entidade e é o único objeto acessível externamente. Um aggregate deve respeitar as seguintes regras:
  - Objetos externos só podem manter referencias para a Entidade raiz
  - A raiz pode manter referencias para qualquer dos agregados
  - Qualquer agregado pode manter referencias entre si ou para a raiz
  - Se houverem outras Entidades dentro do aggregate elas só podem fazer sentido dentro do aggregate
  - Qualquer modificação nos objetos internos só pode ser demandada através da entidade raiz
  - Não é possível passar referências para objetos internos “para fora”. Se houver necessidade de fazê-lo deve-se passar uma referência para uma cópia do objeto interno
- Dessa forma um aggregate garante a consistência de seus objetos internos

Aggregate (4)

- Um exemplo de Aggregate pode ser visto na figura
- O sistema mantém referências apenas para “Customer”
- Address e ContactInfo são VOs gerenciados por Customer



Repositórios (1)

- Em uma aplicação orientada a objetos é necessário obter referências para os objetos com os quais desejamos lidar, sejam eles simples ou complexos.
- A criação de objetos complexos pode ser simplificada com o uso do padrão Factory que será visto em seguida. Mas, ao longo de seu ciclo de vida, muitos objetos precisam ser persistidos e, mais tarde, serem recuperados novamente.
- Bancos de dados e outras formas de persistência são mecanismos da camada de infraestrutura. Seria muito ruim misturar esse tipo de código cada vez que fosse necessário obter a referência para um objeto que já foi persistido.

Repositórios (2)

- O padrão Repository vem ajudar nesta questão.
- Use um Repository com o propósito de encapsular toda a lógica necessária para obter referências para objetos. Os objetos de domínio não devem lidar com a infraestrutura necessária para obter referências para outros objetos de domínio. Eles devem simplesmente solicitar para o repositório, mantendo a clareza e o foco.

Factory

- Em alguns casos, Entidades e Aggregates podem ser complexos de criar. Nestas situações a criação de um objeto complexo pode envolver muito mais do que a simples ativação do construtor da classe.
- Se o objeto que necessita criar um objeto complexo estiver na camada de aplicação, então parte do conhecimento da camada de domínio terá de “extravasar” para a camada de aplicação, o que não é aconselhável.
- O padrão Factory resolve esse problema na medida que encapsula o conhecimento necessário para a criação de um objeto complexo.

Ver lista de exercícios