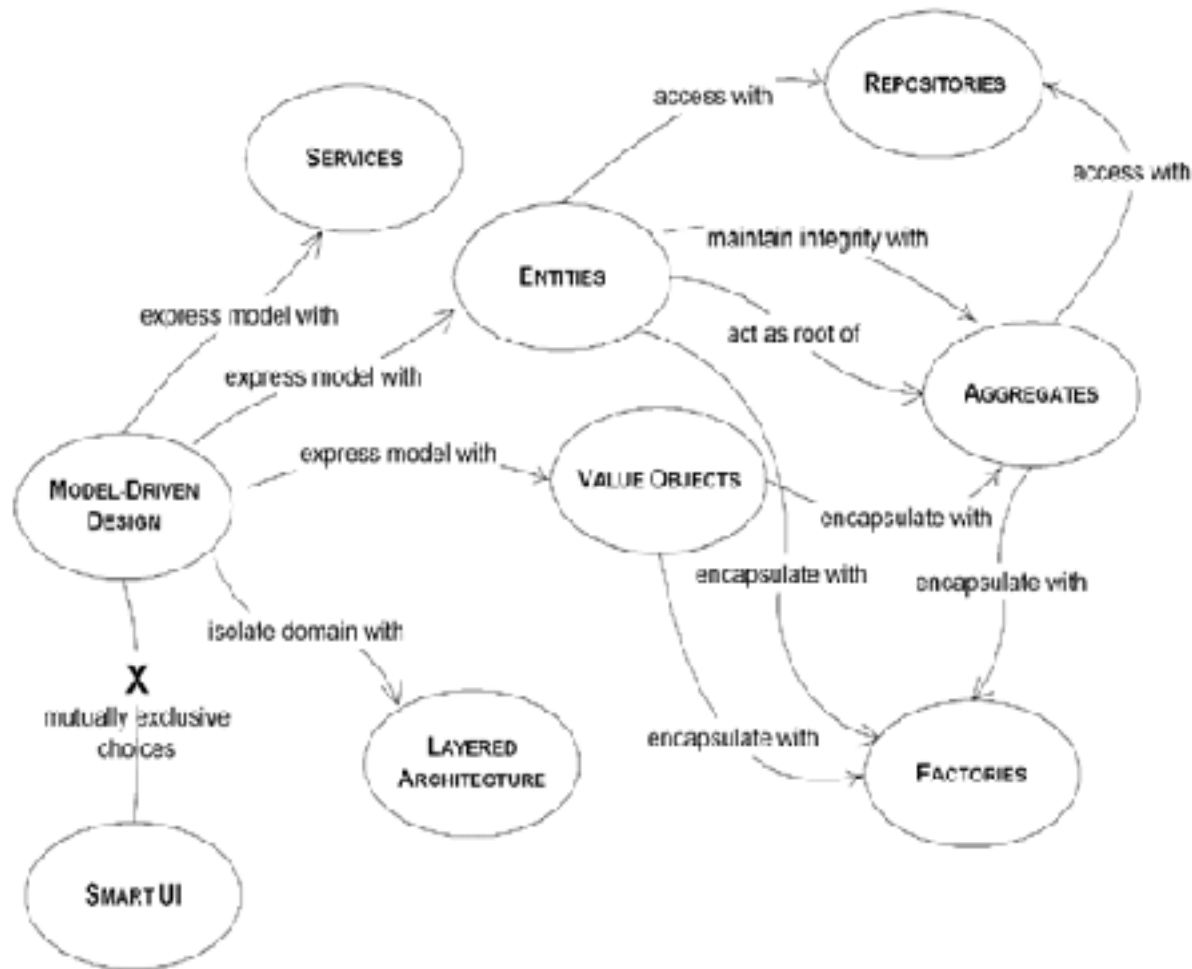




Domain-Driven Design - Parte 2



A modelagem tática é utilizada dentro do bounded context para construir o domain model



Nem sempre faz sentido adotar Domain-Driven Design no nível tático, depende do negócio

Quando evitar Domain-Driven Design no nível tático

- Exportação de dados
- ETLs
- CRUDs
- Relatórios
- Integrações



Resumindo, quando não envolver a implementação de regras de negócio

Outro ponto de atenção é com a equipe, desenvolver orientado ao domínio **requer mais experiência e esforço na modelagem, envolve muita discussão e amadurecimento do modelo de domínio**

Objetos de Domínio

- Entities
- Value Objects
- Domain Services
- Aggregates
- Repositories

Entities

Abstraem regras de negócio independentes, assim como no Clean Architecture, mas no DDD elas tem **identidade** e **estado**, podendo sofrer mutação ao longo do tempo

Exemplos

Order: Pode ter mais ou menos itens, com o tempo é pago, entregue ou até mesmo cancelado

Product: Muda de preço com frequência e pode ser descontinuado

Coupon: Pode ter a sua validade alterada, eventualmente ser restrito a uma categoria de itens ou até mesmo ter uma determinada quantidade disponível para utilização

Como gerar a identidade?

Manualmente: O próprio usuário pode gerar a identidade da entidade, nesse caso é mais complicado garantir a unicidade

Aplicação: A aplicação pode utilizar um algoritmo para gerar a identidade como um gerador de UUID e GUID

Banco de dados: O banco de dados por meio de uma sequence ou outro tipo de registro pode centralizar a geração da identidade

Outro bounded context: A geração pode ser delegada para outro bounded context

Como comparar?

A comparação entre entities se dá normalmente pela **identidade**, sem levar em consideração as características do objeto

Value Objects

Também contém regras de negócio independente, no entanto são identificados pelo seu valor, sendo **imutáveis**, ou seja, a sua mudança implica na sua substituição

Características

- Mede, quantifica ou descreve uma coisa no domínio
- Seu valor é imutável
- É substituído quando seu valor mudar
- Pode ser comparado pelo valor que representa

Exemplos

OrderCode: Representa uma determinada regra de formação de um número

Cpf: Garante que o número do documento é válido

Dimension: Abstrai a largura, altura, profundidade e peso de um item

Substituir por um inteiro

Uma técnica para **identificar** um value object é tentar substituí-lo por um tipo primitivo como uma string ou um número



Um **value object** é restrito a uma **entity** ou pode ser utilizado em vários lugares?

Domain Service

Realiza tarefas específicas do domínio, não tendo estado. É indicado quando a operação que você quer executar não pertence a uma entity ou a um value object

Exemplos


FreightCalculator: É responsável por analisar os itens de um pedido e determinar quanto deve ser pago de frete

DistanceCalculator: Pegando duas coordenadas retorna a distância

StockCalculator: Com base em várias entradas e saídas do estoque, calcula quantos itens estão disponíveis

Utilize em operações que envolvem
múltiplos objetos de domínio

Normalmente quando uma operação afeta
múltiplos objetos de domínio, não pertencendo a
nenhum deles, ela deve ser descrita em um
domain service



CAUTION

Cuidado para não criar **serviços procedurais**,
favorecendo um modelo anêmico



Como definir o **relacionamento** entre diferentes entities e value objects?

Dimension

Order

Cpf

OrderCode

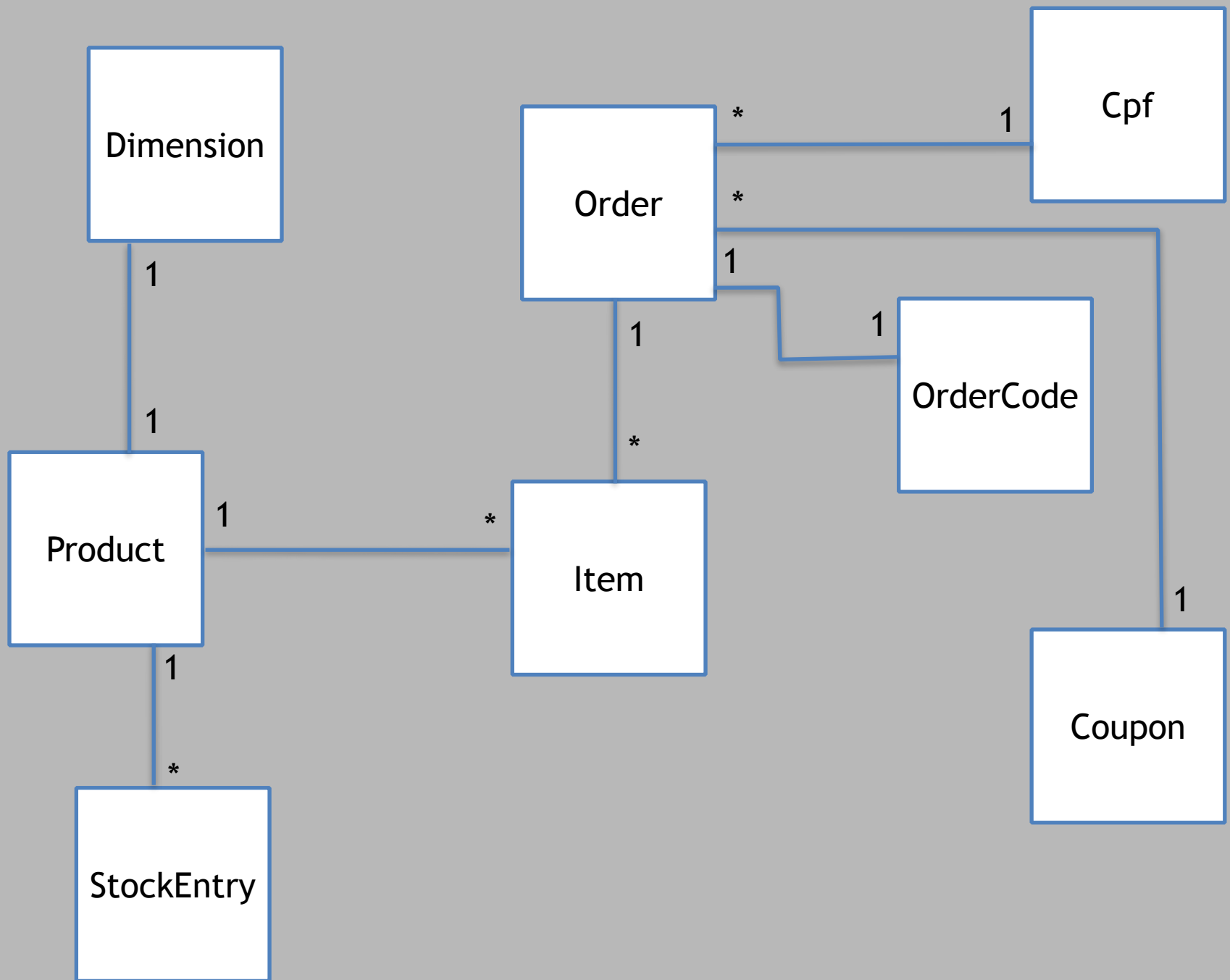
Product


Item

Coupon

StockEntry

<A>





CAUTION

A relação entre os objetos de domínio não é a mesma utilizada no **banco de dados**



Qual é o **problema** dessa abordagem?

Grandes aggregates podem trazer desperdício de memória, além de sobrecarregar o banco de dados sem necessidade já que nem sempre a camada de aplicação estará interessada em utilizá-lo na íntegra



O desafio é **balancear** a preservação da invariância com o consumo de recursos

Aggregates

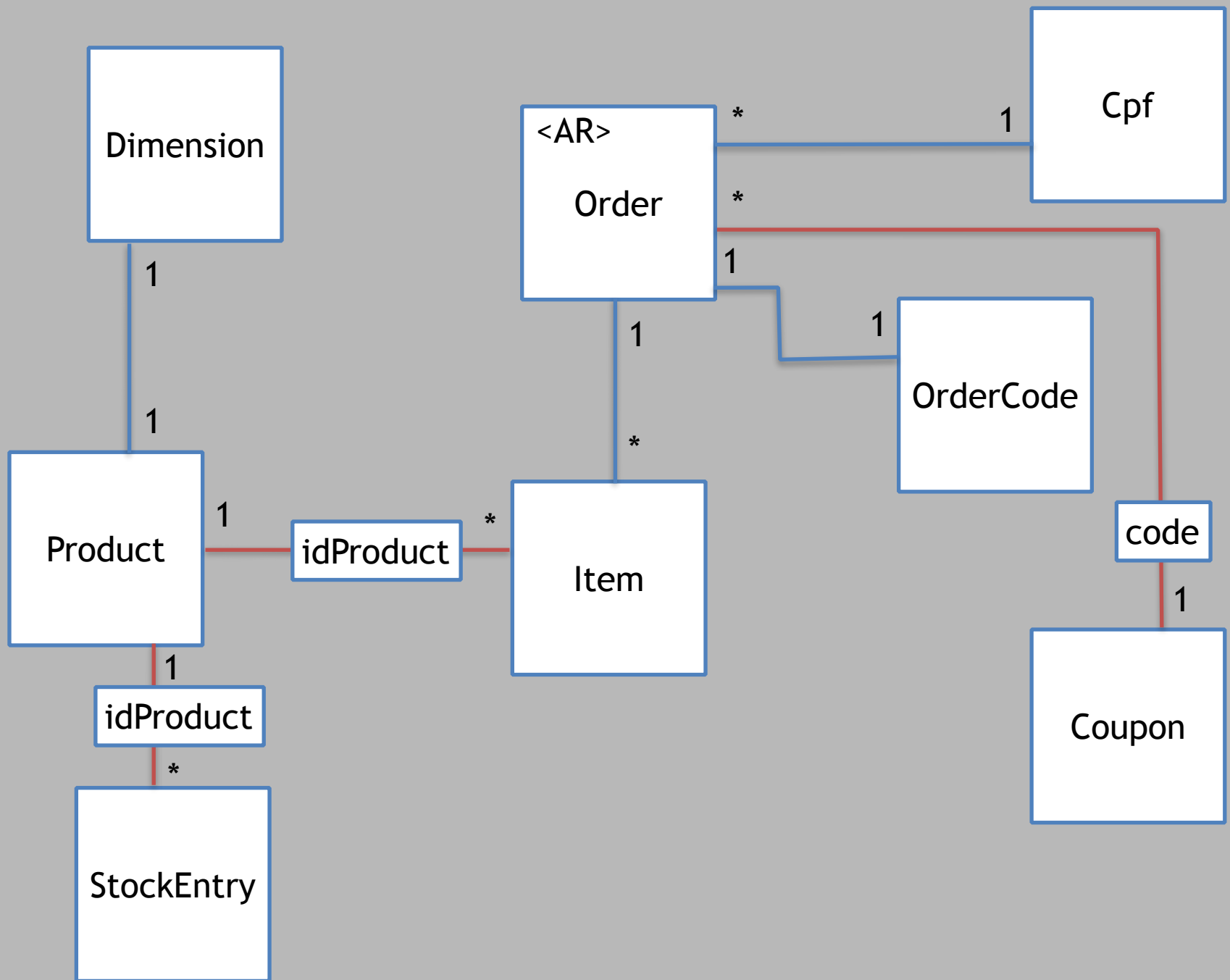
Um aggregate é um agrupamento, ou cluster, de objetos de domínio como entities e value objects, estabelecendo o relacionamentos entre eles, tratado como uma unidade

Todo aggregate tem uma raíz, que é uma entity
por meio da qual as operações sobre o
aggregate são realizadas



Qual é a **solução**?

<A>



<A>

Dimension

1

1

<AR>

Product

idProduct

idProduct

<A>

<AR>

StockEntry

<A>

<AR>

Order

*

1

1

*

Item

code

<A>

<AR>

Coupon

Cpf

1

1

OrderCode

Boas práticas na criação de aggregates

Crie aggregates pequenos: Comece sempre com apenas uma entidade e cresça de acordo com as necessidades

Referencie outros aggregates por identidade:
Mantenha apenas a referência para outros aggregates, isso reduz a quantidade de memória e o esforço que o repositório faz para recuperá-los



Como descobrir o **limite** do aggregate?

Se estiver difícil de implementar o repositório, talvez o aggregate seja muito grande e possa ser separado



Um aggregate deve refletir a base de dados?

Isso faria o aggregate ser **muito grande e
consumir muita memória**, tornando o
repository mais complexo do que deveria



Um aggregate pode referenciar outros
aggregates?

Pode, mas por identidade



Um aggregate pode ter apenas uma entidade?

Pode e quanto menor melhor



Uma entidade que faz parte de um aggregate pode fazer parte de outro?

Não faz muito sentido, uma mudança na entidade utilizada por um aggregate poderia causar a quebra em outro

Repositories

É uma extensão do domínio responsável por realizar a **persistência dos aggregates**, separando o domínio da infraestrutura



Qual é a diferença do padrão Repository no Domain-Driven Design e o DAO?

Um repository lida a persistência de um **aggregate inteiro**, enquanto um DAO não tem uma granularidade definida



Somente parte do aggregate mudou, posso persistir apenas essa parte?

A persistência é sempre realizada sobre o **aggregate inteiro**, no entanto a implementação do repository pode decidir quais registros banco de dados devem ser atualizados



Posso obter apenas parte do aggregate?

Isso significa que pode ser que o aggregate
seja grande mais e poderia ser quebrado
em aggregates menores



Posso utilizar lazy loading dentro do aggregate?

A preservação da invariância depende da integridade do aggregate, se parte dele não estiver populado pode perder o sentido



É possível utilizar diferentes filtros para obter um aggregate?

Com certeza, na obtenção do aggregate
diversos **filtros podem ser utilizados**



Posso gerar dados para a emissão de um relatório a partir de um repository?

A granularidade de um relatório é diferente da utilizada pelo aggregate e renderizar relatórios a partir de repositories pode ser **excessivamente complexo**, prefira a utilização de CQRS com a criação de consultas separadas

Application Service

É a API do Bounded Context, expondo operações de negócio relevantes para os clientes da aplicação, muito similar ao Use Case do Clean Architecture

Atua como uma **fachada para os clientes**,
que podem ser uma API REST, GraphQL ou
gRPC, uma interface gráfica, um CLI, uma
fila entre outros

Faz a **orquestração dos objetos de domínio** e
interage com a infraestrutura para fins de
persistência, comunicação, integração,
segurança entre outros

Mapeia DTOs para objetos de domínio e **vice-versa**, eventualmente utilizando recursos como presenters para converter a saída para o formato mais adequado



Não devem existir **regras de negócio** dentro do application service, apenas orquestração

A camada de aplicação é **naturalmente procedural** enquanto a camada de domínio é favorecida pela orientação a objetos



Um application service pode chamar outro?

Pode, mas isso vai criando uma dependência que quebra o SRP, tornando o código frágil, talvez o ideal seja publicar eventos de domínio



Somente o application service pode utilizar
um repository diretamente?

Quem é responsável por orquestrar os aggregates é o application service, por esse motivo o repository deve ser injetado nele e não nos aggregates



Podem existir várias operações em um application service?

Sim, sem problemas, isso não afeta o design, utilizar uma Screaming Architecture, conforme descrito pelo Robert Martin em Clean Architecture não é uma obrigação



Um application service pode persistir vários aggregates na mesma operação?

A consistência eventual pode trazer complexidade para o application service, nesse caso talvez uma abordagem orientada a eventos ou padrões como o Unit of Work possam ser utilizados



Qual é a diferença entre um use case e um application service?

michael...@gmail.com

12 de jun. de 2013 07:46:29



para clean-code...@googlegroups.com

Hello Uncle Bob :)

I've just watched a great conference where you talked about your way of thinking about Clean Architecture: <http://www.youtube.com/watch?v=esLUIJJqE>

You explained that any heart software should be completely independent of any delivery mechanism (DB, kind of UI etc...) => totally agree.

That's why you outline the need of an Interactor layer, representing any uses cases with pure data structures as input and pure data structures as output.

Currently, I develop a personal project using Hexagonal Architecture (following Domain-Driven Design).

Does the Application Services layer, in DDD sense, play the exactly same role as your Interactor layer (Use-Cases layer)?

Indeed, each application service deals with data structures as input (simple DTO in my case), and returns others. Each service represents a use-case and the whole is well testable quickly with data only in memory without any "details" like controller or anything else. Thus, it seems to be very similar with your definition of "Interactor layer".

Did I missed something ?

Thanks a lot :)

Michael

Uncle Bob

14 de jun. de 2013 05:21:15



para clean-code...@googlegroups.com

Michael,

You didn't miss anything. There is a difference in emphasis between the two architectures, but no difference in substance. DDD emphasizes the domain – ergo "Domain Driven". The Clean Architecture emphasizes the Use Cases, the services. Jacobson called it "Use-case driven". It's not really important which of the two drives. What is important is that the two are kept separate. If the domain drives, you might not get crisp divisions between your use cases; and so the service layer may appear muddled. If the use cases drive, you may not get crisp divisions between your domain elements, and so the domain model may appear muddled. It is the designers job, in each case, to prevent that muddle by viewing the design from the opposing view from time to time.

Infrastructure Service

Os detalhes de implementação de cada tecnologia ficam restritos aos **serviços de infraestrutura**, que ficam numa camada independente da aplicação

Modules

Fornecem uma **separação física para um conjunto de objetos de domínio**, geralmente separado por bounded context, facilitando a organização em projetos grandes



Posso ter um monolito com múltiplos
bounded contexts?



Nem sempre você precisa de **microservices**
para separar os bounded contexts