Estratégias de decomposição e o monólito distribuído

- ***Um breve bate-papo sobre arquitetura de software***

7COMm
Serviços e Soluções em TI

Arquitetura de Software

- Estrutura em alto nível que consiste em partes e suas dependências
- Define aspectos de uma implementação, como escalabilidade, resiliência e segurança
- Define como os elementos do software interagem entre si
- Facilita a divisão de trabalho.

Arquitetura de Software

The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both.

Documenting Software Architectures by Bass et al.

Padrões de desenvolvimento

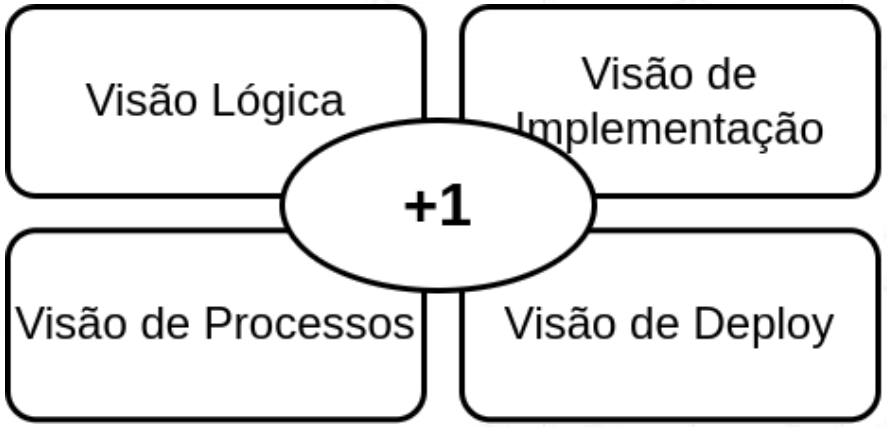
- https://conferences.oreilly.com/software-architecture
- https://resources.sei.cmu.edu/news-events/events/saturn/

O Modelo 4+1

- Desenvolvida por *Philippe Cruchten* com o objetivo de descrever o funcionamento de sistemas de software
- É baseado no uso de múltiplas visões concorrentes
- Define quatro visões de arquitetura: *visão lógica*, de *implementação*, *de processo* e de *deploy*.
- Cada uma dessas visões representam um aspecto da aplicação e seus respectivos relacionamentos.
- O "+1" é a visão de cenários, mostra como as classes contribuem entre si.

O Modelo 4+1

- **Visão lógica**: Elementos criados por desenvolvedores, como classes, pacotes e relacionamentos
- **Visão de implementação**: A saída do build. Módulos que representam o sistema empacotado e seus componentes
- **Visão de processo**: componentes em tempo de execução
- **Visão de deploy:** processos mapeados em máquinas
- **+ 1**: mostra como as classes contribuem entre si



Unified Modeling Language (UML)

- Linguagem padrão para elaboração de estrutura de projetos de software
- Auxilia na visualização do desenho e comunicação entre os processos
- Notação independente de processos
- Possui 14 diagramas na sua versão 2.5, divididos entre *comportamentos*, *estruturais* e *interação*

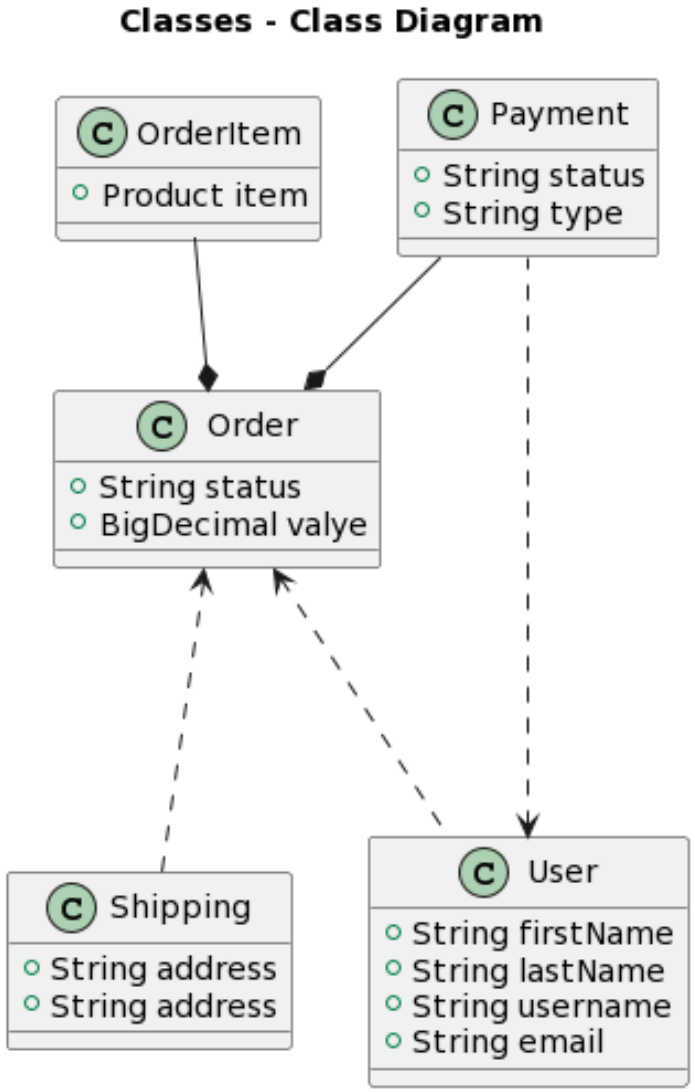
Unified Modeling Language (UML)

- Entre os diagramas mais importantes da UML podemos cita
  - Diagrama de Classes
  - Diagrama de sequencia

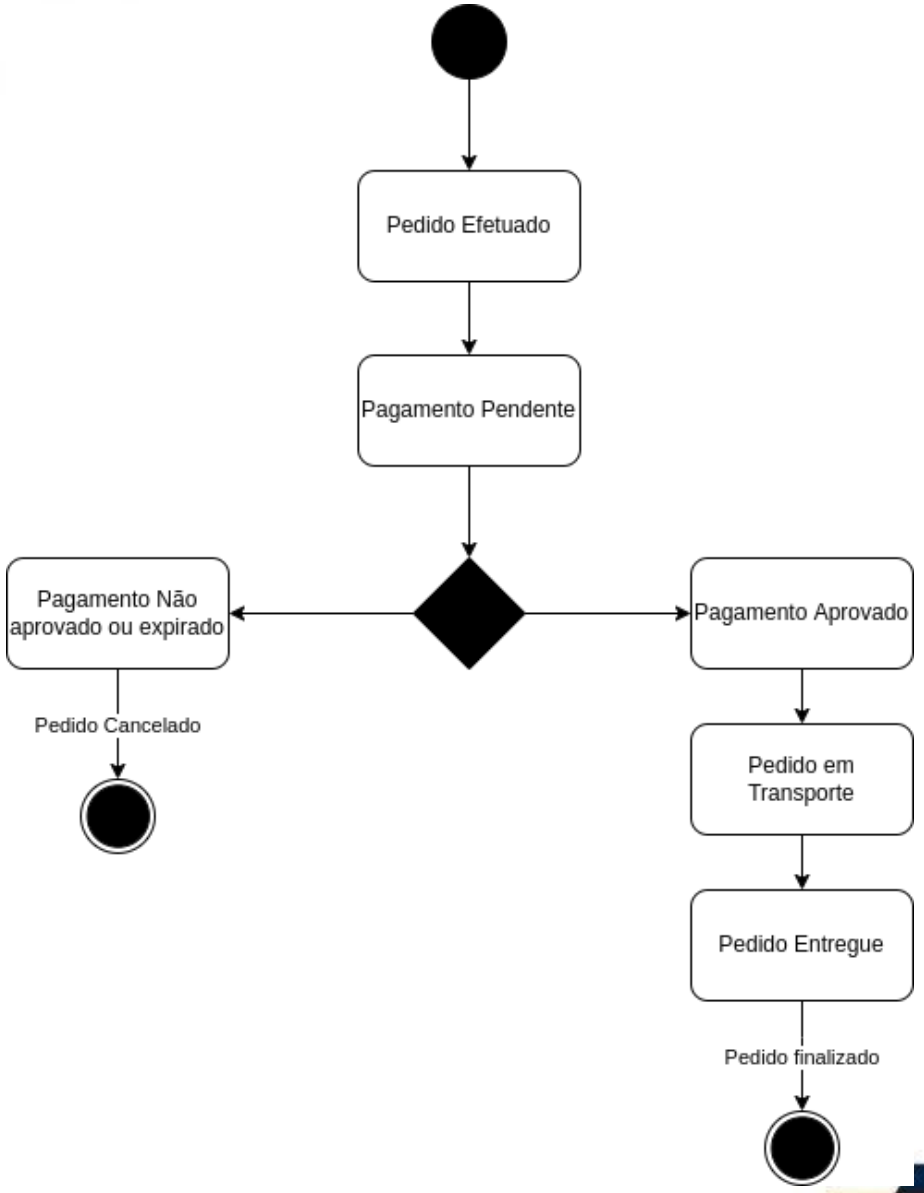
O Modelo 4+1 e UML

- Podemos mapear cada visão do modelo em um conjunto de diagramas UML
- Visão Lógica
  - Diagrama de classes
  - Diagrama de Estados
  - Diagrama de Sequencia

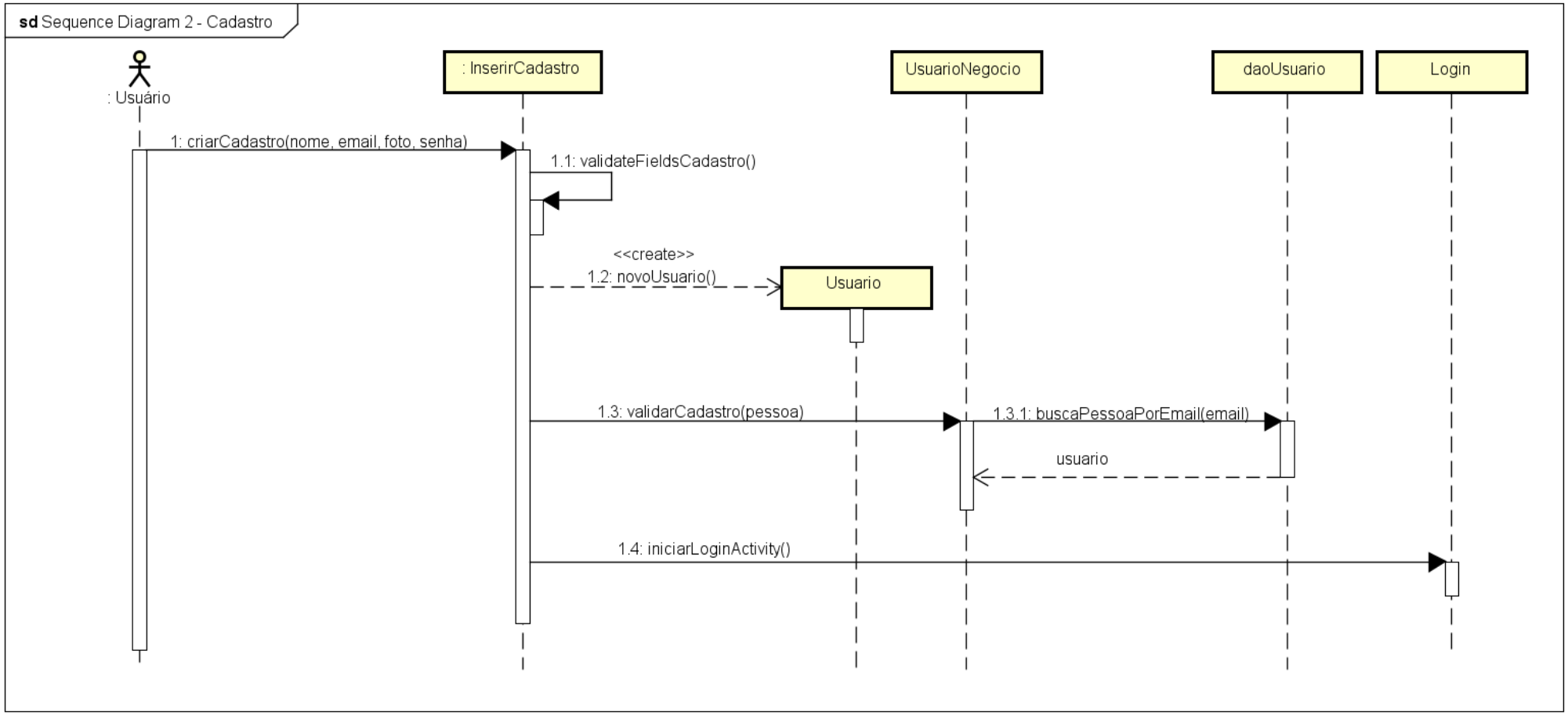
Visão Lógica



Visão Lógica



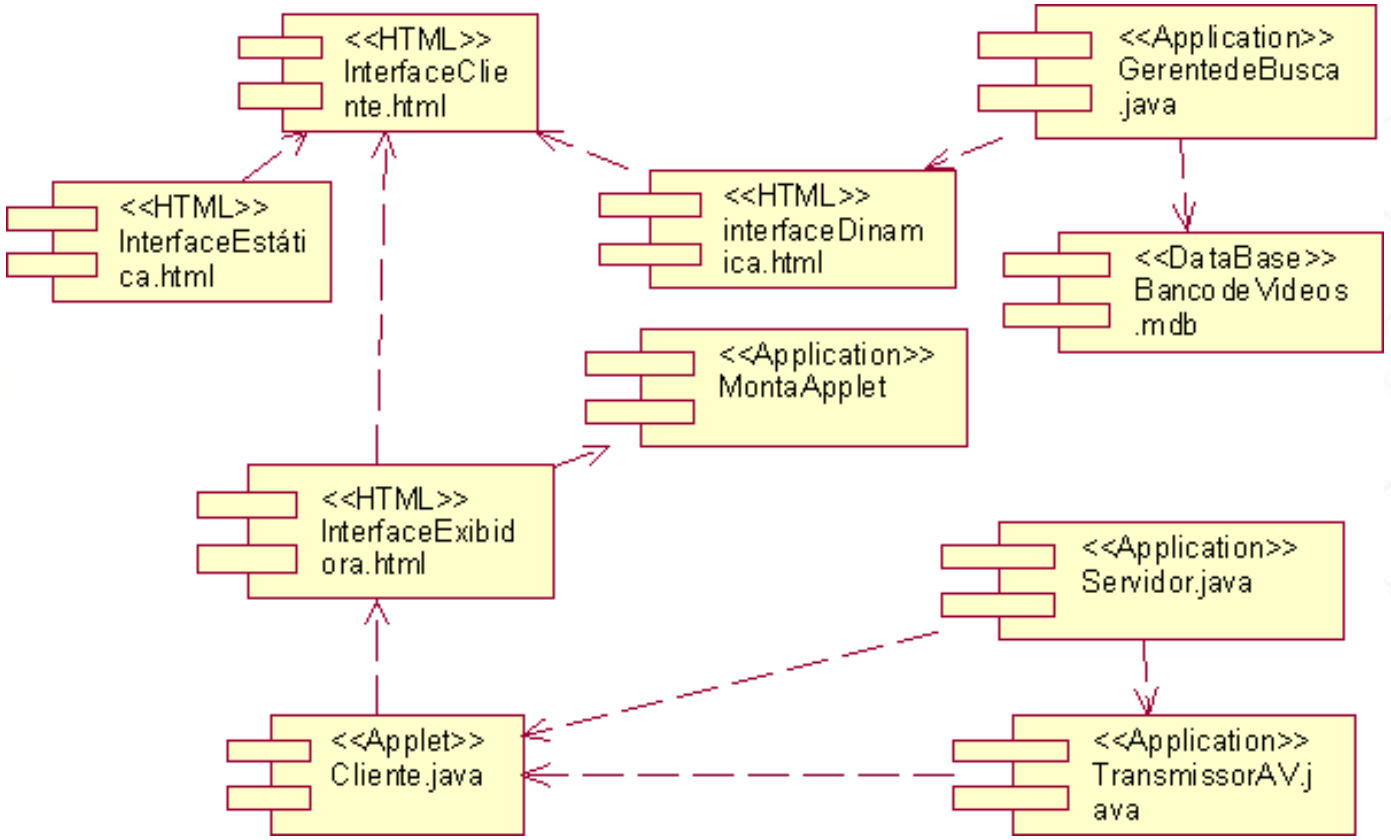
Visão Lógica



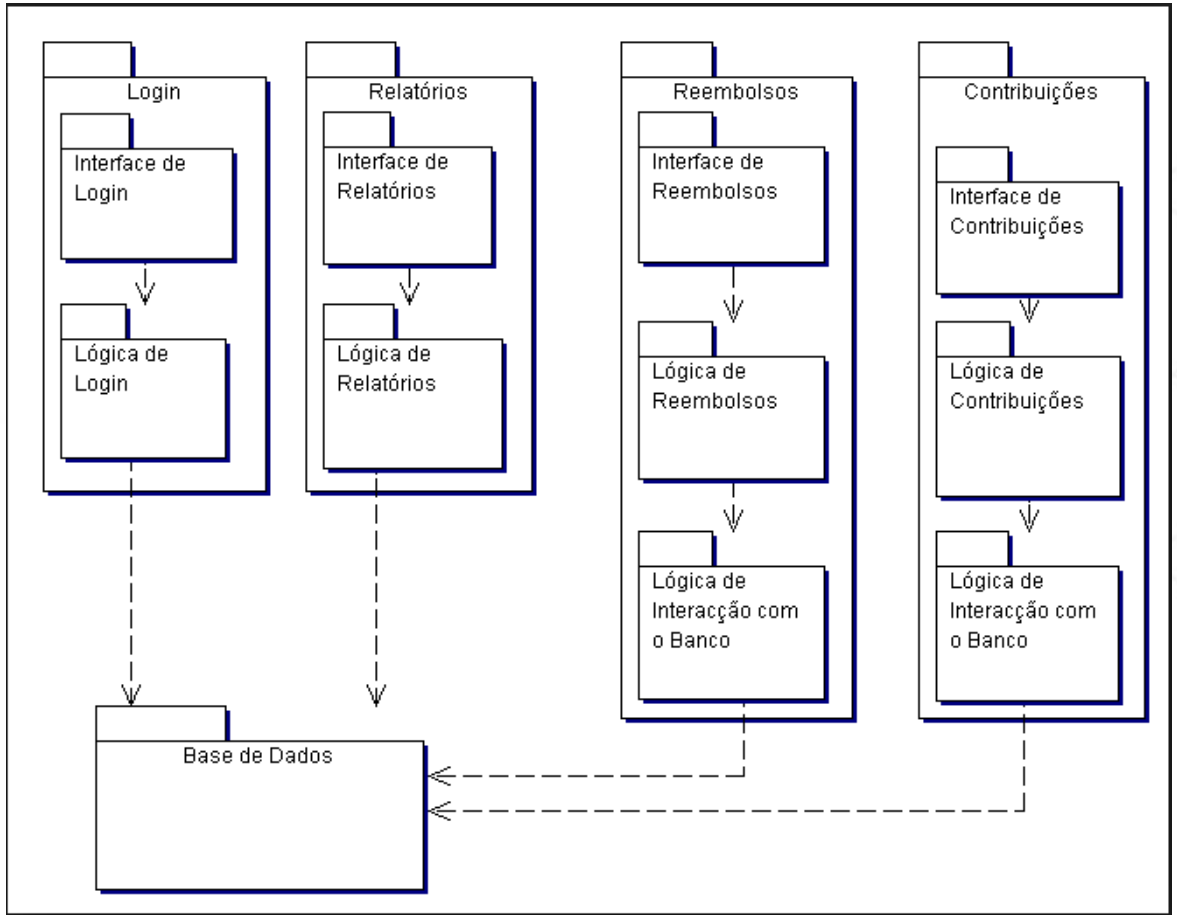
O Modelo 4+1 e UML

- Visão de Implementação
  - Diagrama de Componentes
  - Diagrama de pacotes

Visão de Implementação



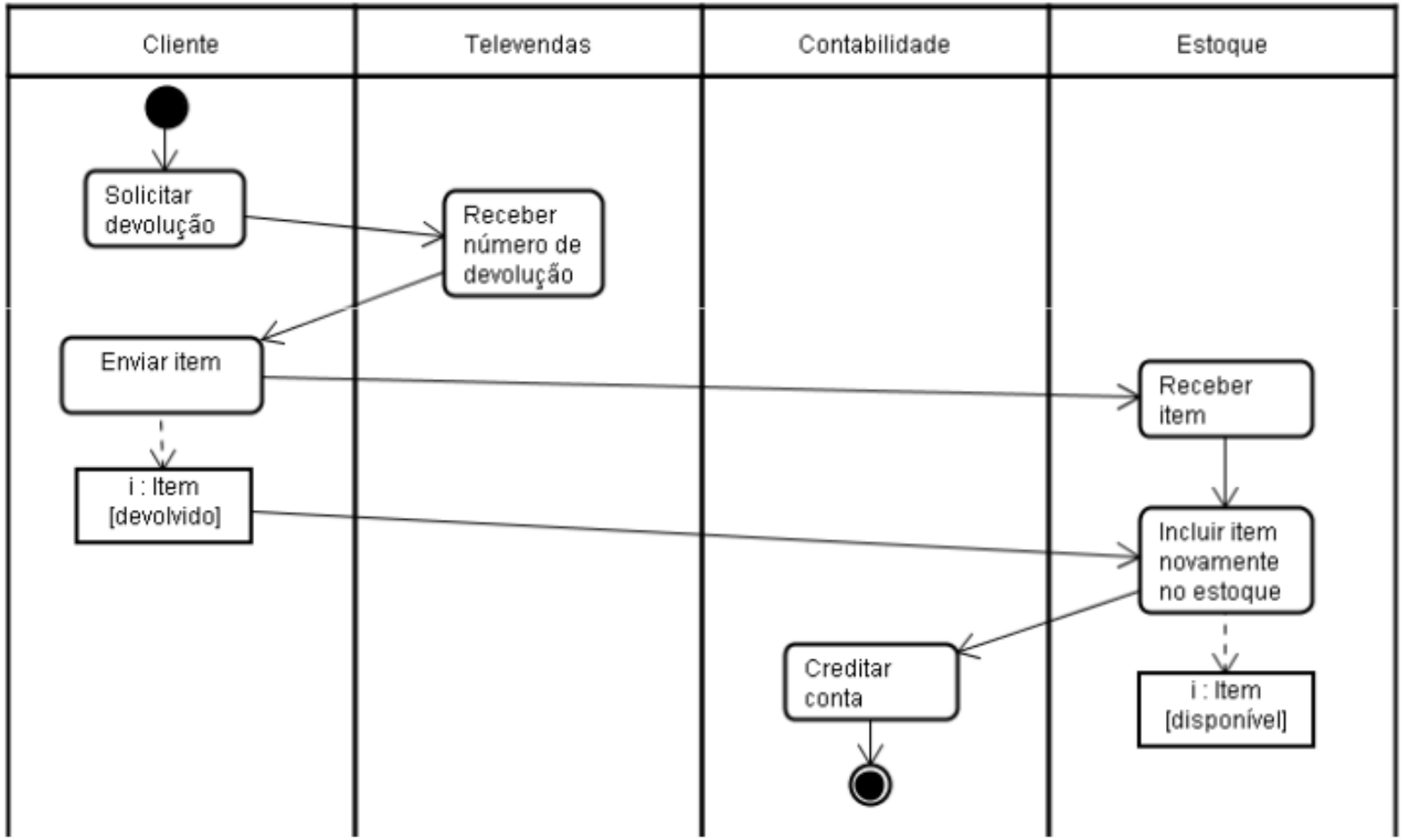
Visão de Implementação



O Modelo 4+1 e UML

- Visão de Processo
  - Diagrama de Atividades

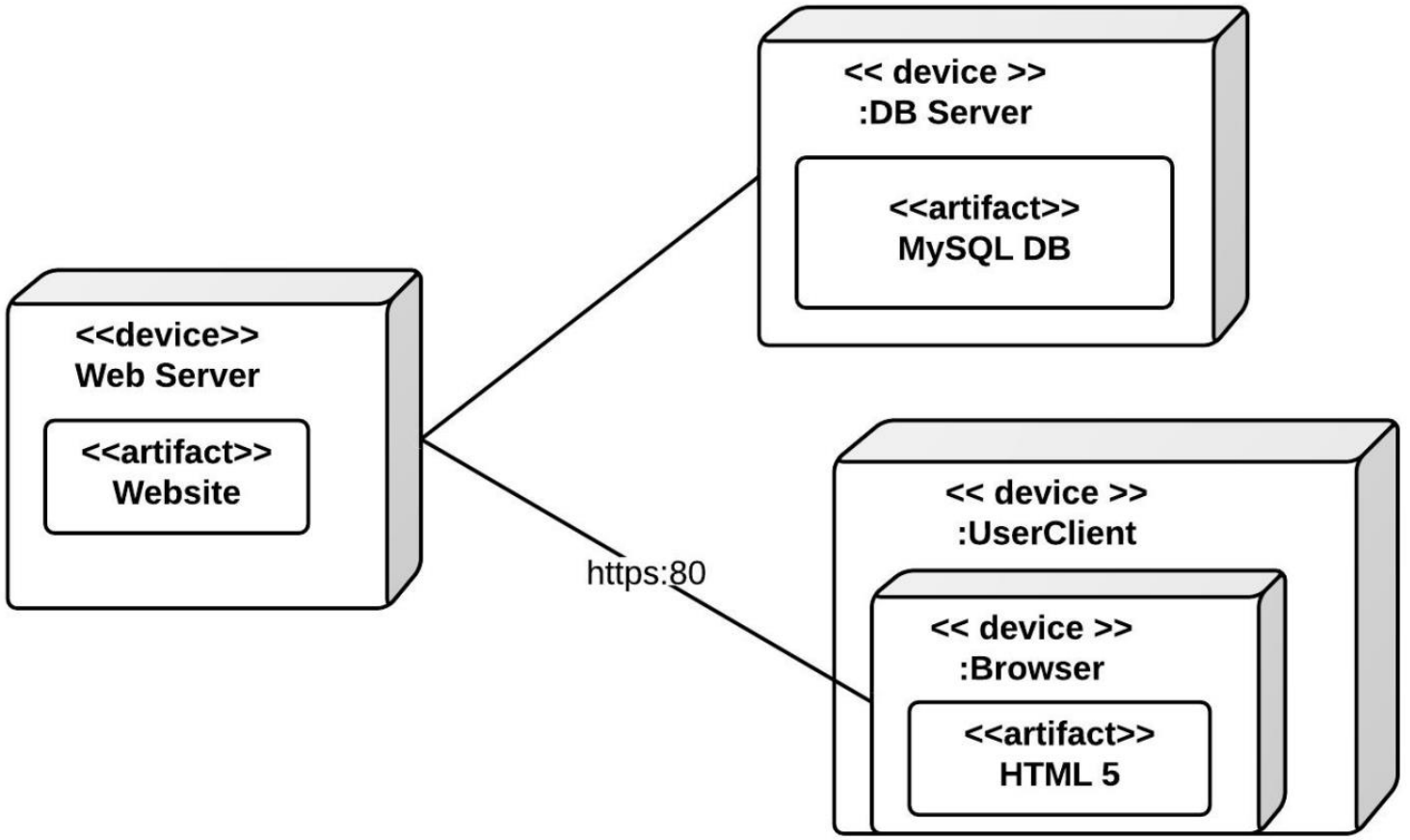
Visão de Processos



O Modelo 4+1 e UML

- Visão de Deploy
  - Diagrama de Deploy

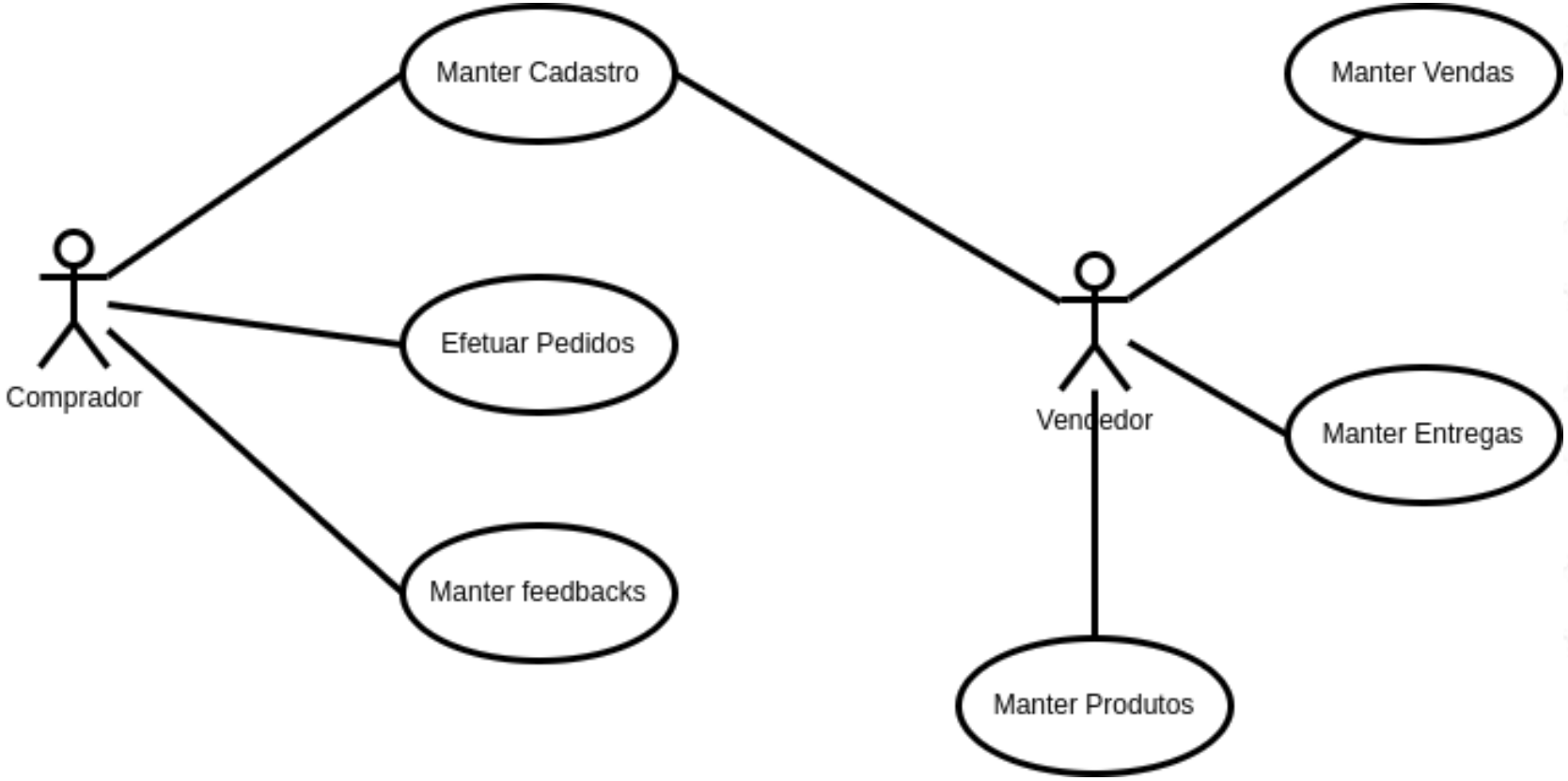
Visão de deploy



O Modelo 4+1 e UML

- Visão de Cenários ou +1
  - Diagrama de Casos de uso

Visão de Cenários



O Modelo 4+1 ?

- Garante que todos os requisitos arquiteturais são devidamente implementados.

Estilos Arquiteturais

- Um estilo arquitetural define uma espécie de paleta de elementos e relações entre eles a partir do qual podemos definir a arquitetura da aplicação
- É comum utilizar mais de um estilo arquitetural em uma arquitetura de *microsserviços*
- Esses estilos pode simplificar o problema de definição de arquiteturas de sistema
- A maioria dos sistemas de grande porte adere a vários estilos

Arquitetura em camadas

- Exemplo clássico. Organiza os elementos em camadas. Cada camada depende da camada imediatamente anterior
- Um dos padrões arquiteturais mais usados, desde que os primeiros sistemas de software de maior porte foram construídos nas décadas de 60 e 70.
- Cada camada possui um conjunto bem definido de responsabilidades e restrições
- São utilizados basicamente em aplicações desktop, pequenas aplicações web ou aplicações com limites rígidos de orçamento e de tempo

Arquitetura em Camadas

Desvantagens de uma arquitetura em camadas

- Há apenas uma camada de apresentação
- Há apenas uma camada de persistência
- A regra de negócios funciona de uma maneira "dependente" da camada de persistência.

Pipe-filter

- Transforma uma funcionalidade complexa em um fluxo de funções mais simples
- Muito comum sua utilização em uma arquitetura funcional
- É comum seu uso em compiladores

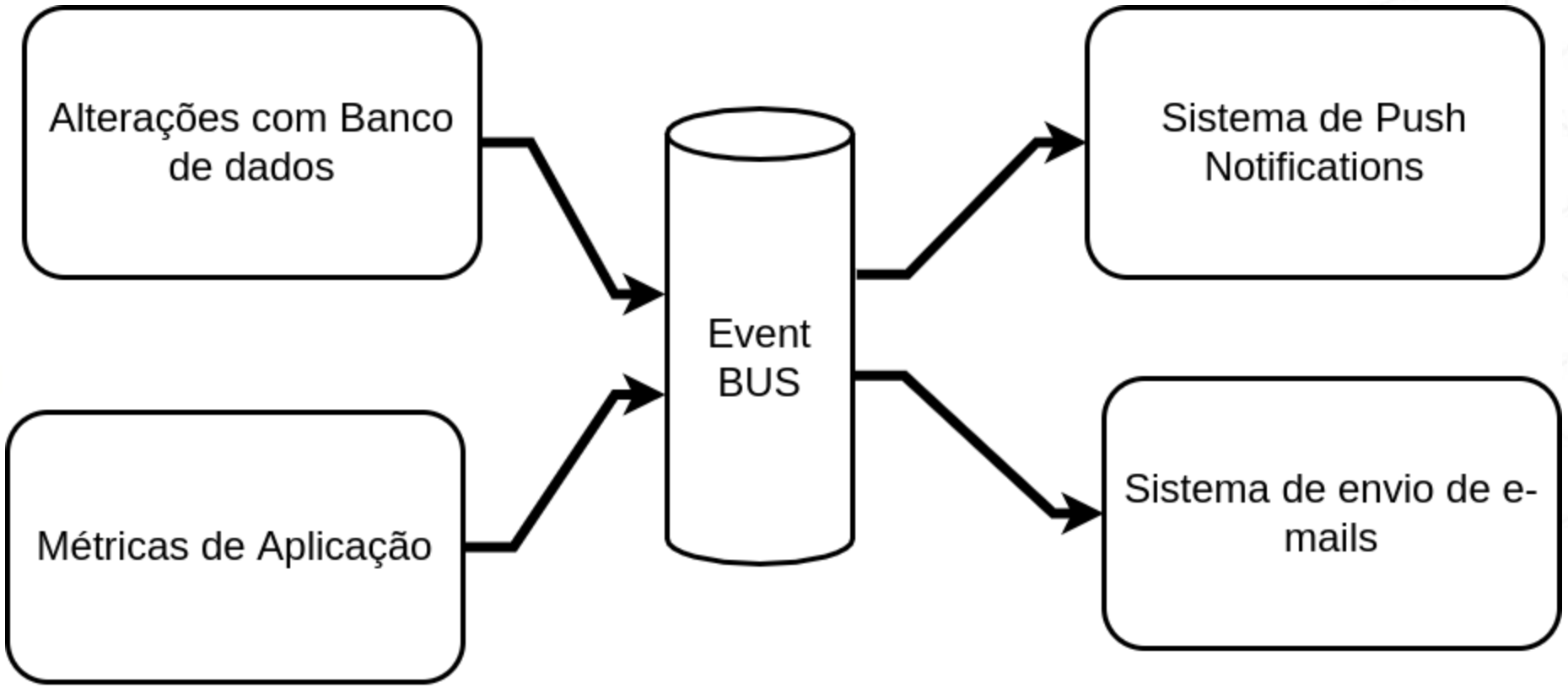
Pipe-filter


Análise Léxica
Análise Sintática
Análise Semântica
Análise Semântica
Geração de Código

Event Bus Pattern

- É um padrão de arquitetura assíncrona distribuída para criar aplicativos reativos altamente escaláveis.
- Pattern baseado em Eventos
- Contém quatro elementos principais: *publisher*, *listener*, *event bus* e *channel*
- A ideia principal é entregar e processar eventos de forma assíncrona.

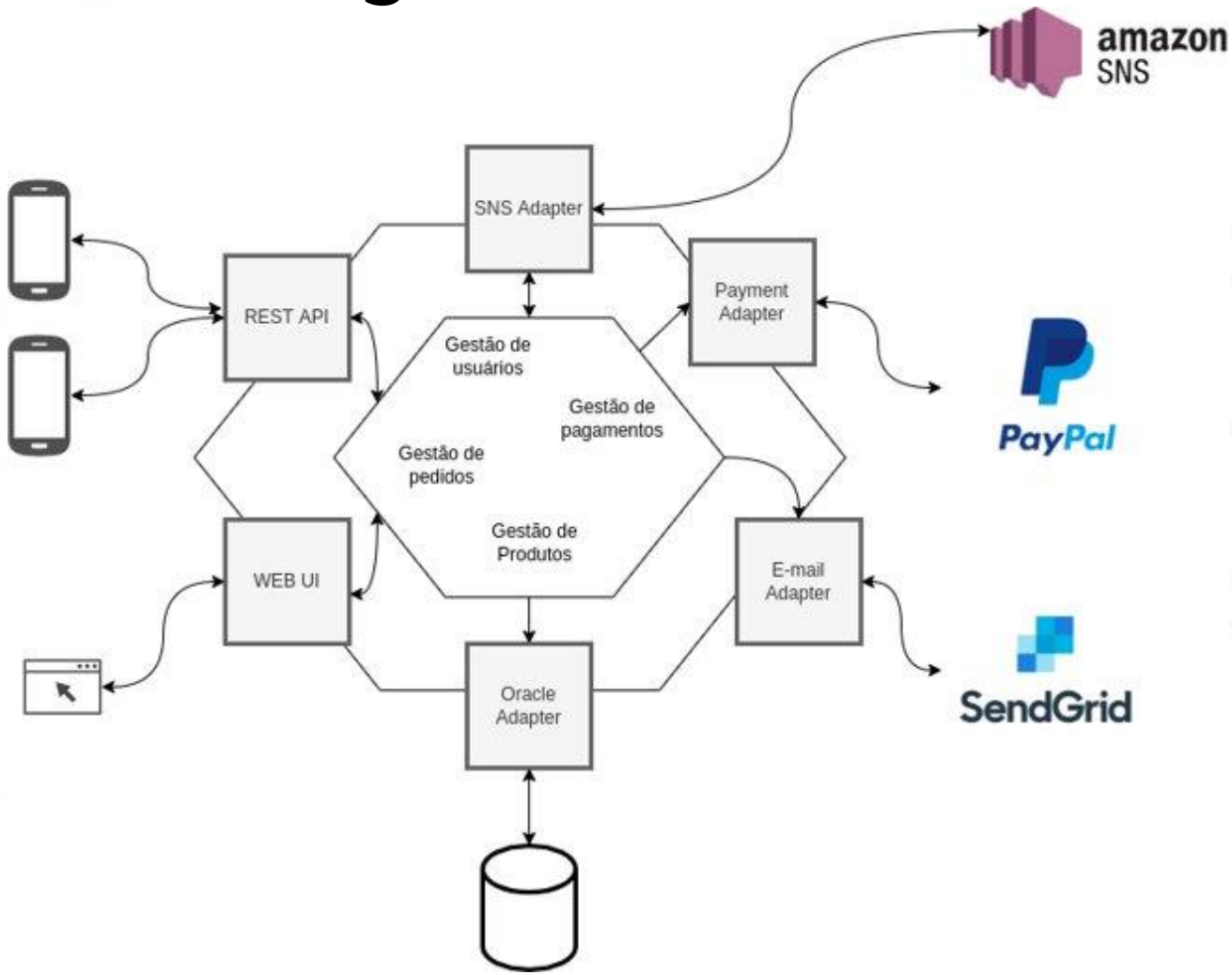
Event Bus Pattern



Arquitetura Hexagonal

- Divide as classes de sistema em dois tipos: *domínio* e *infraestrutura*
- Classes de **domínio** fornecem as **regras de negocio**
- Classes de **infraestrutura** fornecem as **interfaces com entidades externas**
- As classes de domínio não conhecem as tecnologias e com isso mudanças de tecnologia podem ser feitas sem impactá-las
- Mais robusta que o modelo em camadas
- A comunicação entre os dois tipos de classes é feita através de *adapters*

Arquitetura Hexagonal



Microsserviços

7COMm
Serviços e Soluções em TI

Arquitetura em microsserviços

- Arquitetura descentralizada na qual as funcionalidades são desacopladas em serviços diferentes
- A principal característica em uma arquitetura em microsserviços é o baixo acoplamento entre eles
- Cada serviço é independente e deve implementar uma única funcionalidade em um contexto limitado
- Uma grande vantagem dos sistemas arquitetados a microsserviços é a escalabilidade

O que é um serviço?

- Aplicação standalone
- Deploy independente
- Implemente uma funcionalidade útil no domínio da aplicação

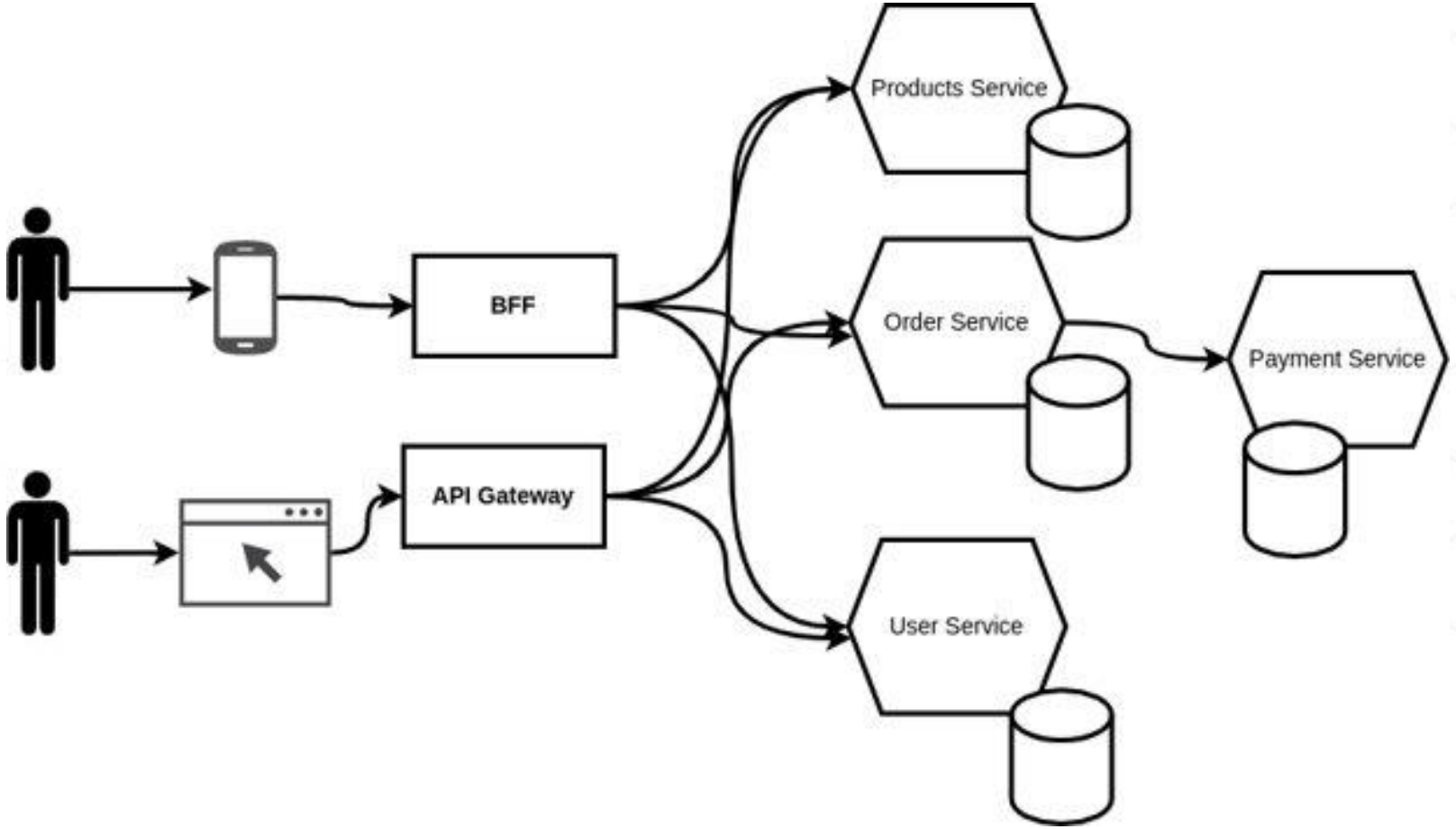
Serviço

- Fornece uma API que permite aos clientes acessarem os dados.
  - **Comandos**, que persistem as informações
  - **Queries**, que consultam e recuperam informações da aplicação
- Cada serviço pode ter sua própria arquitetura e sua própria *stack*

Baixo acoplamento

- Todas as interações com o serviço **devem acontecer via API**
  - Encapsulam os detalhes de implementação
  - Manutenção otimizada
  - Otimização do desenvolvimento sem impactos no restante do ecossistema
- Não é utilizada a integração via banco de dados

Arquitetura de microsserviços



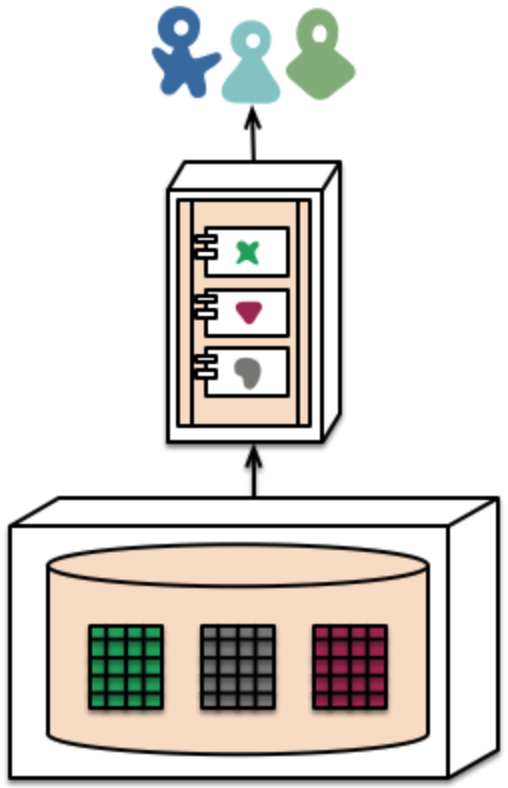
Qual o tamanho ideal?

- O nome *microsserviço* não ajuda.
- Não é uma métrica assertiva.
- Desenvolvido por um time no menor tempo possível, com a menor colaboração de outros times
- Se tiver mais de uma equipe trabalhando nele, pode ser um indicativo ruim
- Se muita manutenção é necessária quando ***outros*** serviços mudam é indicativo de alto acoplamento

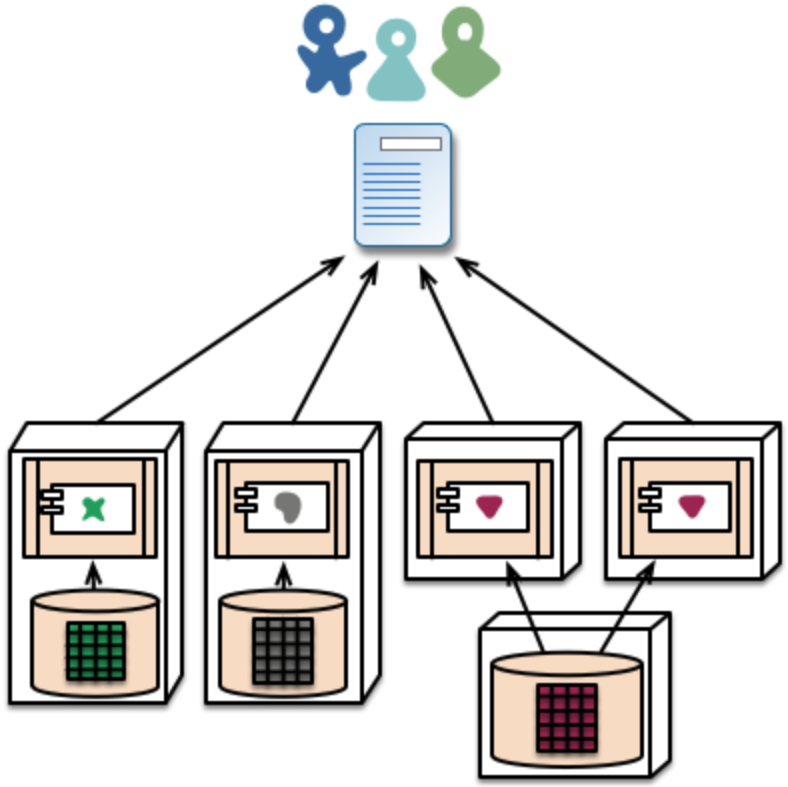
Bibliotecas compartilhadas

- A priori parece uma excelente ideia separar classes em múltiplas aplicações
- É preciso ter muito cuidado nessa decomposição pra não se manter o acoplamento distribuído

Monolito distribuído



monolith - single database



microservices - application databases

Definindo uma arquitetura em microsserviços

7COMm
Serviços e Soluções em TI

Definindo uma arquitetura em microsserviços

- Não existe um "gabarito". De acordo com as particularidades da empresa e do projeto, pode-se ter uma arquitetura diferente
- Não é um processo mecânico. Envolve bastante vivência e, até, criatividade em certo ponto
- Deve levar em conta as particularidades da empresa e do negócio no processo

Definindo uma arquitetura em três passos

1. **Identificar as operações do sistema**
   - Um *request* que a aplicação deve processar
   - Essas operações se transformam em um cenário arquitetural
2. **Identificar os serviços**
   - Entregar agrupamentos em conceitos de negócio
3. **Definir APIs e colaborações**
   - Todas as operações descobertas no passo 1 devem ser convertidas em APIs dentro dos serviços

1. Identificando Operações de Sistema

- Requisitos funcionais
- Modelo em alto nível do problema
- Identificar operações de sistema e descreve-las de acordo com o modelo gerado
- *Domínio = substantivos; Operações = verbos;*

Como ***Cliente ou vendedor****, eu quero* ***manter um cadastro*** *para que meus* ***pedidos*** *sejam* ***processados*** *corretamente com meus dados.*

Critérios de Aceite

- Os campos **Nome, CPF, e-mail, senha**, e pelo menos um **endereço** são obrigatórios
- O campo CPF deve ser **validado** com a regra vigente de validação de números e dígitos, assim como a formatação correta
- A qualquer momento quaisquer dados meus podem ser alterados por mim através da interface correta, desde que, devidamente logado e autorizado.

Como ***Cliente ou vendedor****, eu quero* ***ser credenciado*** *para* ***ter acesso ao sistema e suas funcionalidades de acordo com meu perfil***

Critérios de Aceite

- Os campos para credenciamento são: login e senha
- Em caso de inatividade, minha sessão deve ser invalidada em 15 minutos

Como ***Cliente****, eu quero* ***realizar uma compra*** *para ter acesso a produtos de meu interesse*

Critérios de Aceite

- O cliente deve estar habilitado e validado
- Cada compra pode possuir um ou mais produtos de um mesmo vendedor
- Os produtos devem estar disponíveis e em quantidades adequadas ao pedido do usuário
- O pedido só pode ser finalizado uma vez que o pagamento seja AUTORIZADO

Como ***Cliente****, eu quero* ***realizar o pagamento de um pedido*** *para* ***dar continuidade ao processamento do mesmo***

Critérios de Aceite

- A Forma de pagamento deve ser aceita pelo vendedor
- O pedido pode ser cancelado enquanto não for finalizado através da entrega

Como ***cliente****, eu quero poder ver meus* ***pedidos e seus respectivos status.***

Critérios de Aceite:

- Cliente deve estar logado no sistema
- Só poderá aparecer pedidos do cliente logado

Como ***vendedor****, eu quero poder* ***alterar o preço dos meus produtos***

Critérios de Aceite:

- Vendedor deve estar logado no sistema
- Só poderá alterar o preço dos produtos do vendedor em questão

Como ***cliente****, quero* ***poder cancelar minha compra*****.**

Critérios de Aceite:

- Cliente precisa estar logado
- Cliente só pode cancelar as compras feitas por ele
- Compras com status FINALIZADO não podem ser canceladas

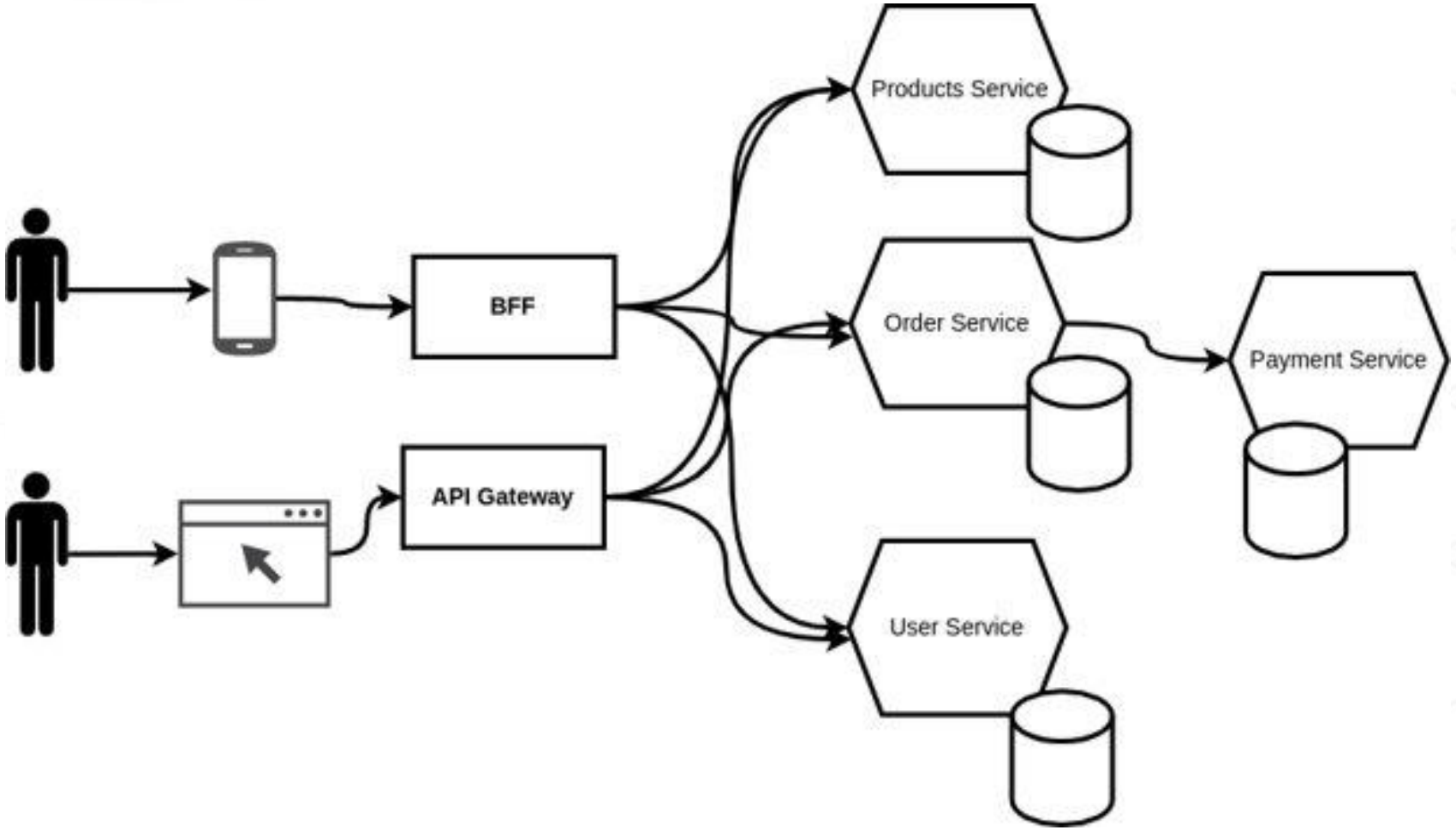
Como **vendedor** eu quero poder **ver todos os meus produtos listados**

Critérios de Aceite:

- Apenas produtos do vendedor aparecem na listagem

Modelo de Alto nível

- Modelo mais simples do que o que vai ser implementado
- Cada serviço terá seu próprio modelo, portanto, não gastemos mais energia do que o necessário


Products Service
BFF
Order Service
Payment Service
API Gateway
User Service

Operações do sistema

- Uma vez definido o modelo, utilizando as mesmas histórias, deve-se definir as requisições para este processamento
- Esta definição pode ser classificada como síncrona ou assíncrona, mas pode-se deixar para posteriori

Ator	História de usuário	Comando	Descrição
Cliente	Manter cliente	criaUsuario()	Cria um novo usuário
Cliente	Manter cliente	alteraUsuario()	Altera os dados do usuário logado
Cliente	Credenciamento	login()	Credencia e autoriza o usuário para o acesso a suas funcionalidade
Vendedor	gestão de produtos	criaProduto()	Vendedor cria um produto novo
Vendedor	Gestão de produtos	ZeraProduto()	Zerando o estoque do produto e não pode mais ser vendido

Operações do sistema

Ator	História do usuário	Comando	Descrição
Cliente	Ver Pedidos	pedidosDoUsuario()	Lista os pedidos do usuário logado no sistema
Vendedor	Gestão de produtos	alterarPrecoProduto()	Vendedor altera preço do produto
Cliente	Cancelar pedido	cancelarPedido()	Cliente cancela um pedido
Vendedor	Gestão de produtos	listaProdutos()	Lista os produtos do vendedor logado no sistema

2. Definindo serviços

- Uma vez que as operações são definidas, estamos aptos a identificar os serviços da aplicação
- Veremos algumas estratégias para este procedimento.

Business Capability Pattern

- Define, através de mercado, algo que a empresa faça que entrega valor
- Algo já mapeado através de outros projetos similares.
- É comum utilizar em projetos que já estão mapeados e definidos. O que a empresa *faz!*
- Essa quebra acaba fazendo sentido na primeira iteração. É possível, devido a particularidades da empresa ou do projeto, que precise ser ainda mais especializada: separar pagamentos de cobranças, gateways diferentes, etc.
- Acaba sendo uma primeira divisão estável.

DDD

- Abordagem para construir sistemas complexos
- Centralizada em domínios de objetos.
- Uma entidade acaba não sendo centralizada, como na abordagem tradicional
- Pode-se utilizar réplicas dessa mesma entidade, dependendo do domínio em que ela é aplicada

EXEMPLO CLASSES DDD

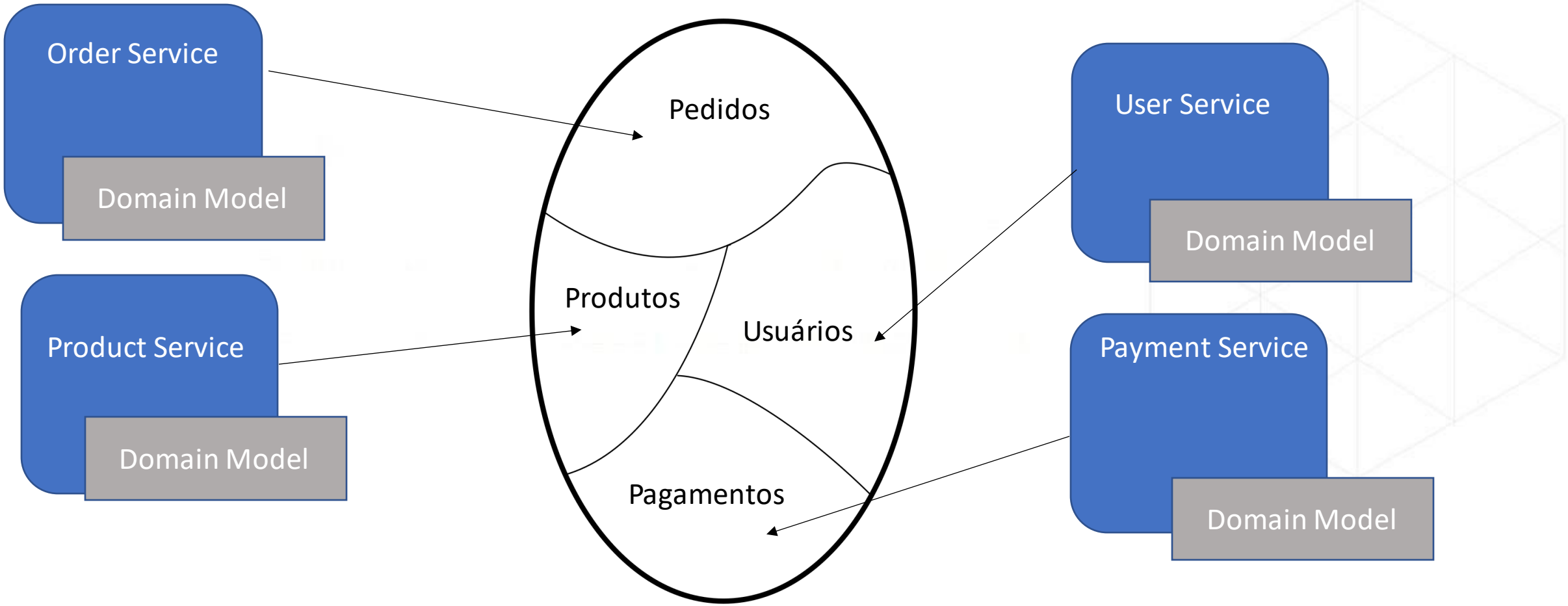
```java
public abstract class AggregateRoot {
    private List<DomainEvent> recordedDomainEvents = new LinkedList<>();

    final public List<DomainEvent> pullDomainEvents() {
        final var recordedDomainEvents = this.recordedDomainEvents;
        this.recordedDomainEvents = new LinkedList<>();

        return recordedDomainEvents;
    }

    final protected void record(DomainEvent event) {
        recordedDomainEvents.add(event);
    }
}
```

```java
public final class Video extends AggregateRoot {
    private final VideoTitle title;
    private final VideoDescription description;

    private Video(VideoTitle title, VideoDescription description) {
        this.title = title;
        this.description = description;
    }

    public static Video publish(VideoTitle title, VideoDescription description) {
        var video = new Video(title, description);

        var videoCreated = new VideoPublished(title.value(), description.value());

        video.record(videoCreated);

        return video;
    }
}
```

Subdomínios e Boundered Contexts

- Subdomínios são utilizados de uma maneira parecida com o pattern anterior
  - através das capabilities identifica-se as áreas de expertise
- **Boundered context** é o escopo de cada um desses domínios, incluindo o código-fonte
- Estão em perfeito alinhamento com uma arquitetura em microsserviços. Cada um subdominio pode ser considerado um serviço.

Subdomínios e Boundered Contexts



Algumas observações e diretrizes...

7COMm
Serviços e Soluções em TI

Single Responsability

- Uma das principais dificuldades no desenho de uma arquitetura, ou mesmo, na sua implementação.
- Cada responsabilidade do serviço, é um motivo em potencial para sua alteração. Portanto, cada **serviço**, deve ter uma única responsabilidade

Single Responsability

A class should have only one reason to change.

Robert C. Martin

Common Closure Principle

- Se dois **serviços/classes** devem ser alteradas pelo mesmo motivo, elas devem estar no mesmo *pacote*
- Caso um serviço implemente uma regra e esta regra muda, este deve ser o único serviço impactado
- http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Common Closure Principle

The classes in a package should be closed together against the same kinds of changes. A change that affects a package affects all the classes in that package.

Robert C. Martin

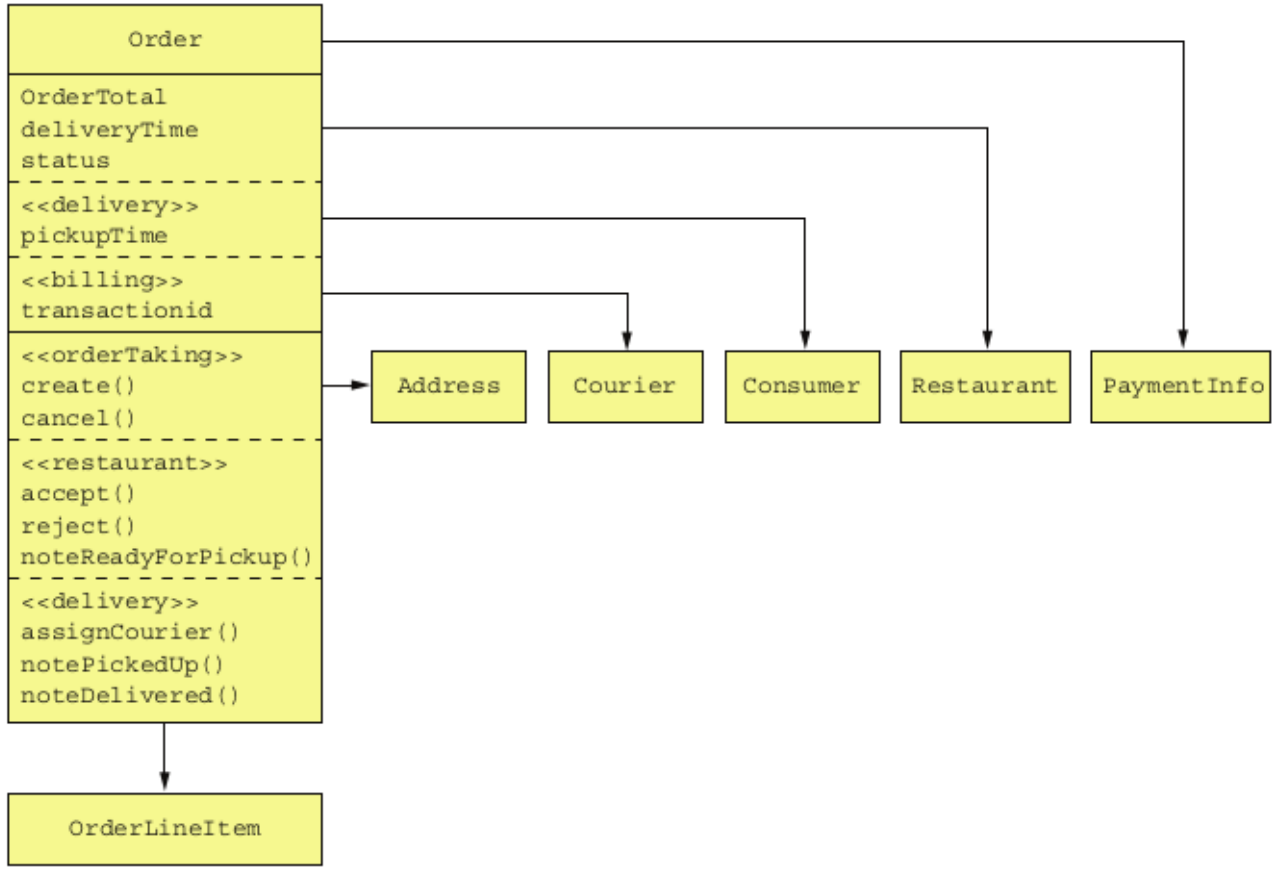
... e alguns obstáculos.

7COMm
Serviços e Soluções em TI

God Classes

- Classes *inchadas*, utilizadas em toda a aplicação
- Normalmente lógica que passa por todos os módulos da aplicação
- É comum existir ***pelo menos*** uma em cada projeto
- Sua implementação é difícil de ser *quebrada*

God Classes


Order
OrderTotal
deliveryTime
status
<<delivery>>
pickupTime
<<billing>>
transactionid
<<orderTaking>>
create()
cancel()
<<restaurant>>
accept()
reject()
noteReadyForPickup()
<<delivery>>
assignCourier()
notePickedUp()
noteDelivered()
OrderLineItem
Address
Courier
Consumer
Restaurant
PaymentInfo

ACID e microsserviços

A — **Atomic**

All changes to the data must be performed successfully or not at all

C — **Consistent**

Data must be in a consistent state before and after the transaction

I — **Isolated**

No other process can change the data while the transaction is running

D — **Durable**

The changes made by a transaction must persist

Consistência dos dados

- Algumas operações devem alterar dados em serviços múltiplos atomicamente
- Deve-se abordar uma estratégia para esse tipo de transação
- Em muitos casos devemos aplicar o conceito da **consistência eventual**
- Deve-se sempre observar a **real necessidade de uma transação distribuída**. *Esta não pode ser separada em um serviço só dela?*
- Veremos com muito mais profundidade as transações com sagas, mais a frente

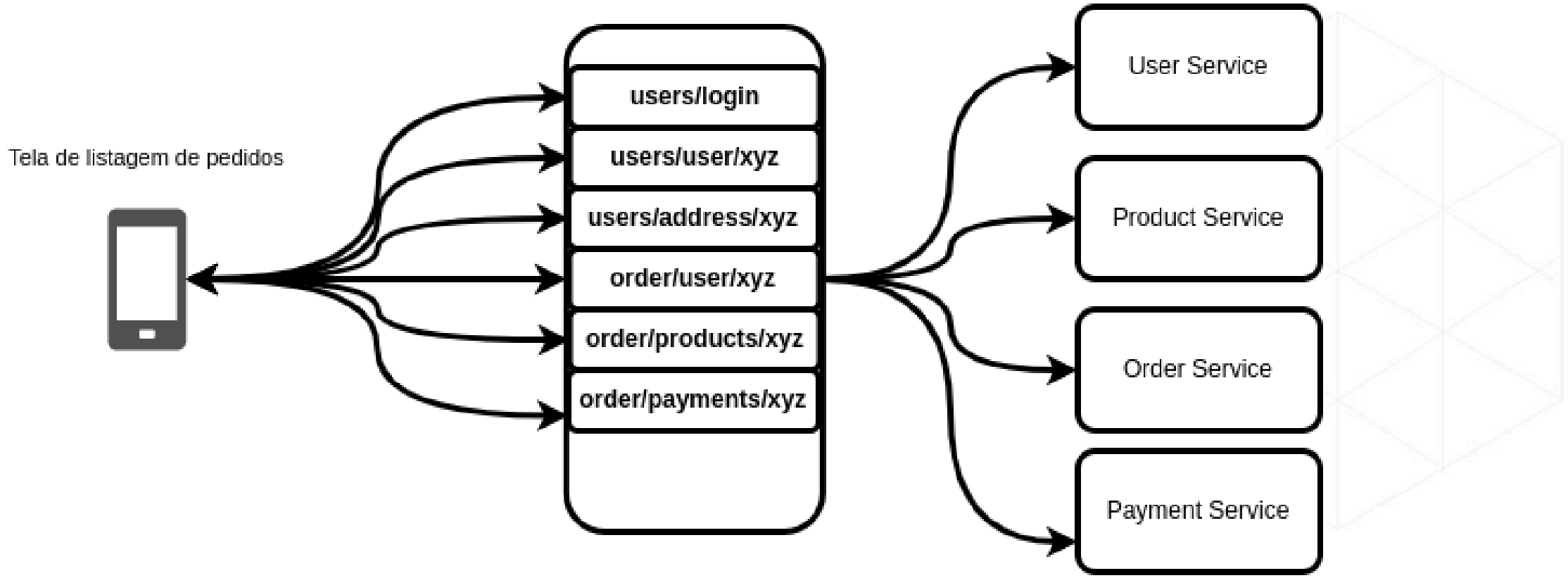
Visão consistente dos dados

- O mesmo problema da consistência eventual ocorre nas consultas
- Como manter a consistência de um dado que passa por múltiplos bancos de dados?
- É possível implementar **ACID** em uma arquitetura de microsserviços?
- Caso essa consistência seja algo fundamental, a informação deve ser mantida em um único serviço.

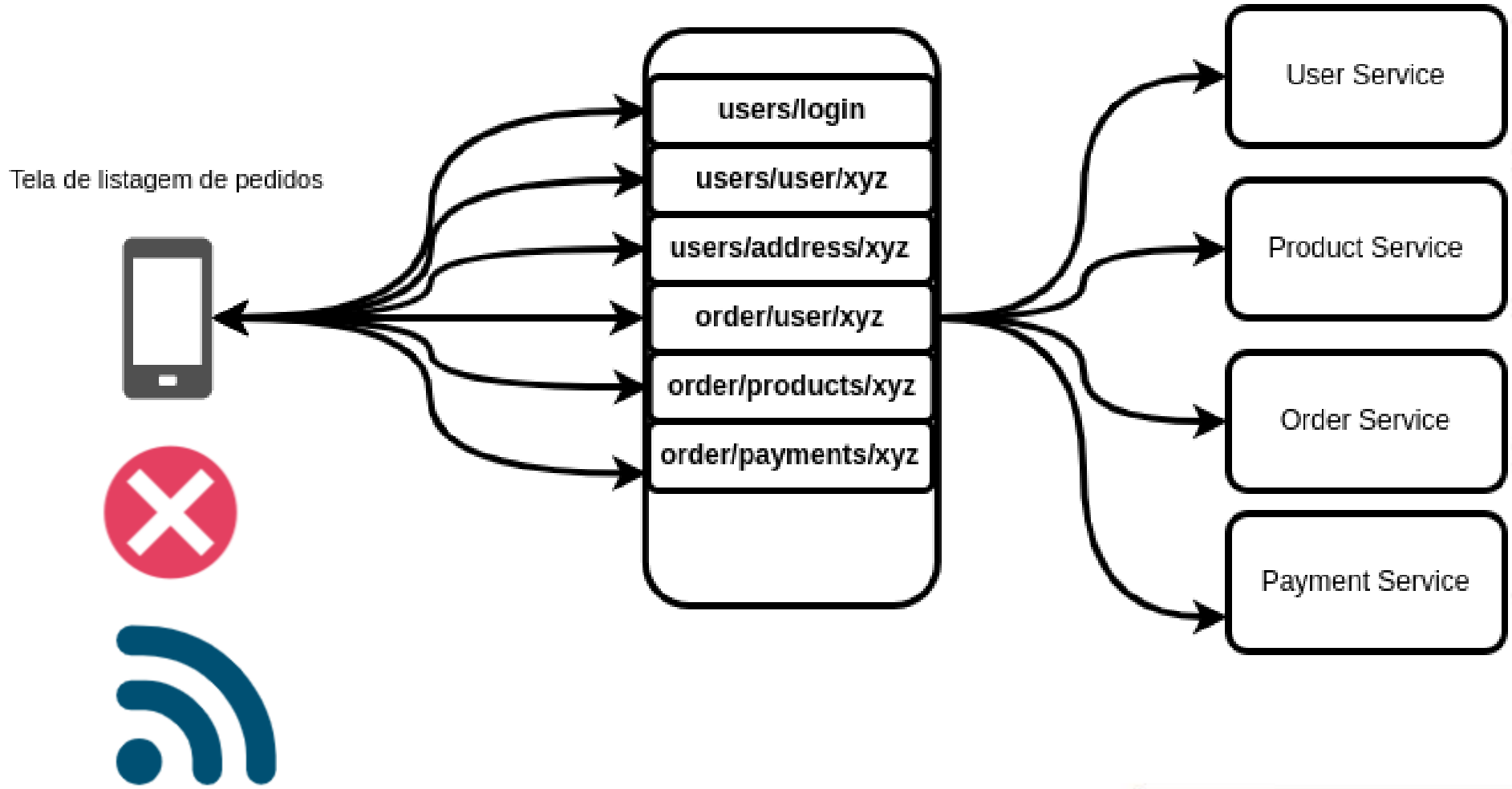
Latência de rede

- Preocupação sempre constante. Deve-se observar o número de idas e vindas na comunicação entre dois ou mais serviços
- Existem algumas estratégias para mitigar, que veremos mais a frente
- O ideal, é evitar um grande número de *idas e vindas*.

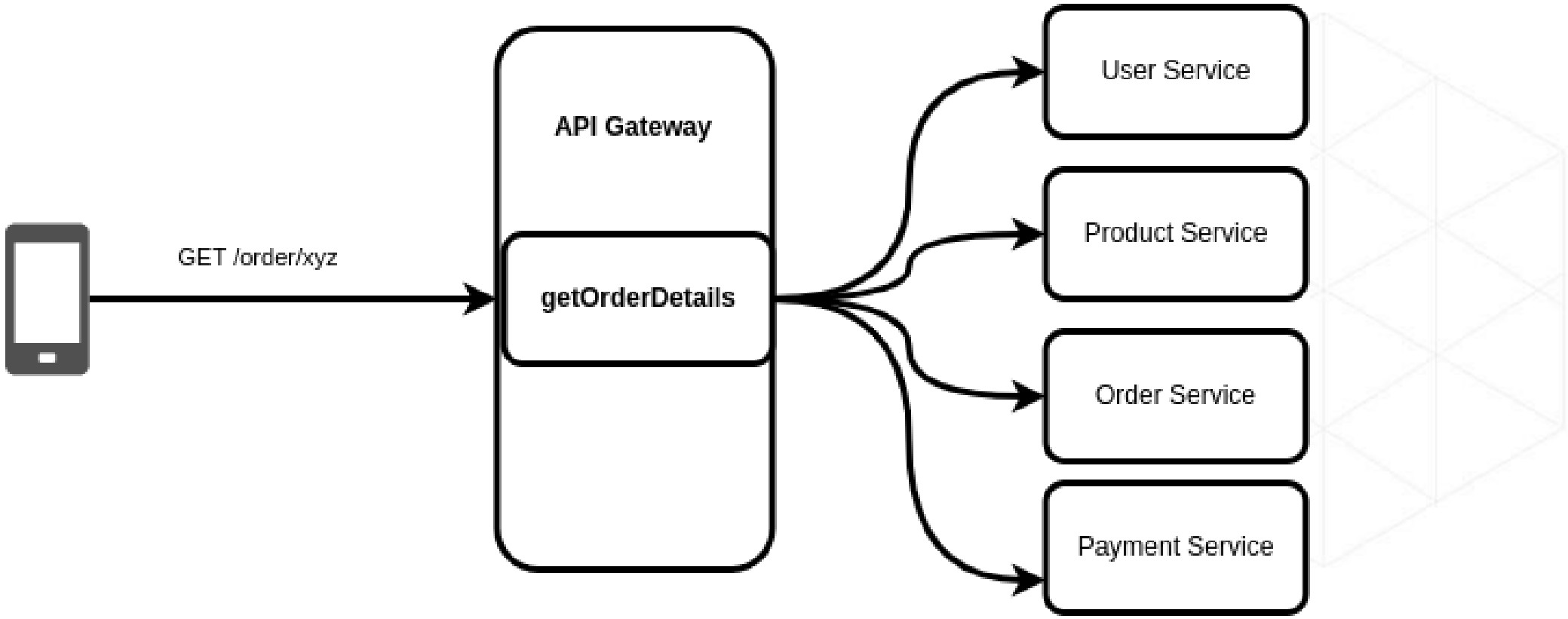
Latência de rede



Latência de rede



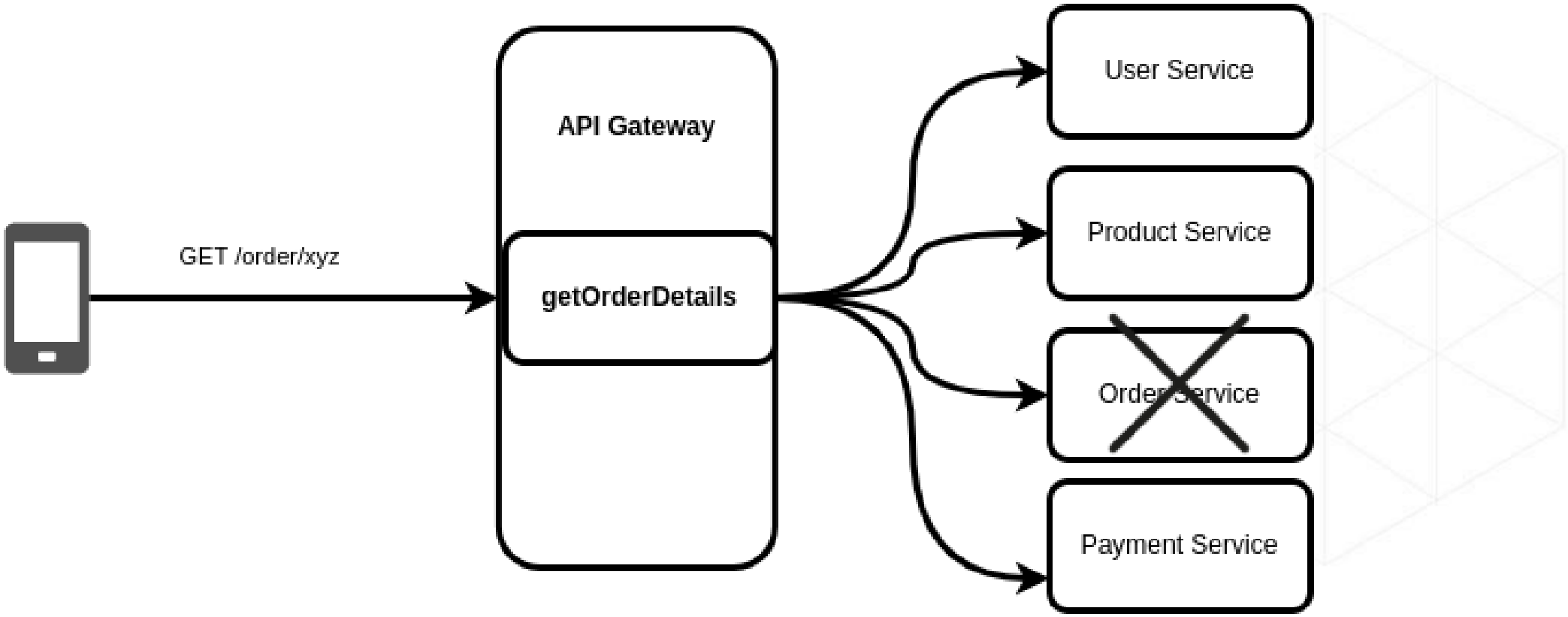
Latência de rede



Comunicação síncrona e a disponibilidade

- Comunicação síncrona reduz a disponibilidade
- Essa deve ser sempre uma preocupação no desenho da comunicação entre um serviço e outro
- Deve-se estudar caso a caso este tipo de comunicação

Latência de rede



3. Definindo as APIs

- Uma vez definidas as operações e mapeadas em serviços, decidimos se esses serviços precisam se comunicar com outros
- Caso seja necessário, são determinadas as APIs dessa comunicação.
- Algumas vezes esse mapeamento é bastante óbvio. Outras vezes precisamos de mais refinamento.

Definindo as APIs

Serviço	Operação	Colaboradores
User Service	• CreateUser(); • UpdateUser(); • Login(); • Logout(); • ValidateUser();	-
OrderService	CreateOrder()	• ProductService.getProduct(); • ProductService.processProduct(); • UserService.validateUser(); • PaymentService.ReceivePayment()
OrderService	ChangeStatus()	ProductService.processProduct()

Definindo as APIs

Serviço	Operação	Colaboradores
PaymentService	ReceivePayment()	-
PaymentService	AuthorizePayment()	OrderService.ChangeStatus()
PaymentService	DenyPayment()	OrderService.ChangeStatus()
PaymentService	CancelPayment	OrderService.ChangeStatus()
ProductService	ListProducts()	
ProductService	GetProduct()	
ProductService	ProcessProducts()	

Próximos Passos

- Comunicação entre os serviços

Para saber mais...

- https://en.wikiquote.org/wiki/Software_architecture
- https://www.youtube.com/watch?v=BrT3AO8bVQY
- https://en.wikipedia.org/wiki/Loose_coupling
- http://microservices.io/patterns/decomposition/decompose-by-business-capability.html
- http://wiki.c2.com/?GodClass
- https://plantuml.com/

OBRIGADO!

Centro
Rua Formosa, 367 - 29º andar Centro, São Paulo - SP, 01049-000

Alphaville
Avenida Ipanema, 165 - Conj. 113/114 Alphaville, São Paulo - SP,06472-002

+55 (11) 3358-7700
contact@7comm.com.br

7COMm
Serviços e Soluções em TI