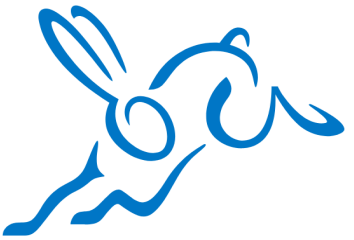
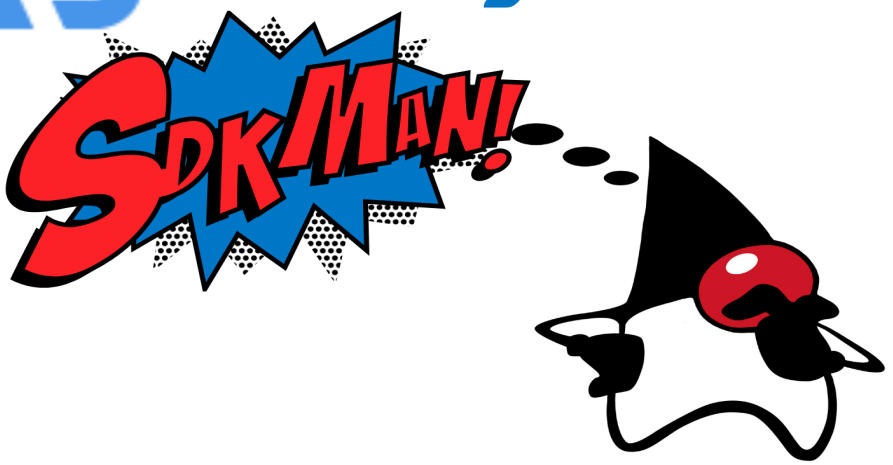
ADRIANO DOS
SANTOS RIBEIRO





ARQUITETURA JAVA MODERNA

PADRÕES E PRÁTICAS

APLICAÇÕES CORPORATIVAS
CLOUD-NATIVE COM QUARKUS

Padrões e Práticas para Arquitetura Java Moderna

Aplicações Corporativas Cloud-Native com Quarkus

ADRIANO DOS SANTOS RIBEIRO

Este livro está disponível em
https://leanpub.com/ppajm-apps-corporativas-cloud-native-com-quarkus

Esta versão foi publicada em 2025-05-27

Conteúdo

Parte 1: Fundamentos - Design Estratégico e Arquitetural

Capítulo 1: Decomposição com Mapa de Contexto DDD Estratégico

Quando encaramos sistemas corporativos, especialmente os legados ou aqueles que cresceram organicamente, a complexidade pode ser avassaladora. Sem uma visão clara de como as grandes peças do negócio se encaixam no software, corremos o risco de criar o temido "Big Ball of Mud". Para evitar isso e trazer ordem ao caos, vamos abordar diretamente o **Design Estratégico do Domain-Driven Design (DDD)**, especificamente usando o **Mapa de Contexto (Context Map)**.

Neste capítulo, não focaremos na granularidade de classes, mas sim em um nível mais alto: identificar as *grandes áreas de responsabilidade* e como as diferentes partes do negócio (e do software) interagem.

Mapas de Contexto, Coesão e Acoplamento

O **Mapa de Contexto** é uma ferramenta essencial para gerenciar a complexidade de sistemas, especialmente os distribuídos, ajudando a equilibrar **coesão** e **acoplamento** entre os diferentes **Bounded Contexts (BCs)**.

Coesão

Coesão refere-se ao quão bem os elementos dentro de um BC estão relacionados e trabalham juntos para cumprir uma responsabilidade específica. Um BC com alta coesão:

- Possui um modelo de domínio claro e consistente.
- Evita responsabilidades dispersas ou conflitantes.
- Facilita a evolução interna sem impactar outros BCs.

No Mapa de Contexto, a coesão é promovida ao:

- Definir fronteiras claras para cada BC.
- Garantir que cada BC tenha uma **Linguagem Ubíqua** própria e bem definida.
- Minimizar a sobreposição de responsabilidades entre BCs.

Acoplamento

Acoplamento refere-se ao grau de dependência entre BCs. Um baixo acoplamento é desejável para:

- Reduzir o impacto de mudanças em um BC sobre os outros.
- Permitir maior autonomia das equipes responsáveis por diferentes BCs.
- Facilitar a escalabilidade e a manutenção do sistema.

O Mapa de Contexto nos ajuda a gerenciar o acoplamento através de padrões de relacionamento, como:

- Identificar e documentar as relações entre BCs (e.g., **Customer-Supplier**, **Shared Kernel**, **Anticorruption Layer**).
- Escolher padrões de integração que minimizem dependências rígidas, como o uso de **Open Host Service (OHS)** e **Published Language (PL)**.
- Avaliar cuidadosamente o uso de **Shared Kernels**, considerando alternativas como a extração da funcionalidade compartilhada para um novo BC dedicado, consumido via OHS e ACL.

Não se preocupe em memorizar todos os padrões de relacionamento agora; vamos explorá-los em detalhes. O importante é entender que o Mapa de Contexto é uma ferramenta poderosa para visualizar e gerenciar a complexidade do sistema, ajudando a manter um equilíbrio saudável entre coesão e acoplamento.

Relação entre Coesão e Acoplamento

- **Alta Coesão e Baixo Acoplamento**: O cenário ideal. Cada BC é autônomo e bem definido, com interações mínimas e controladas.
- **Baixa Coesão e Alto Acoplamento**: Indica problemas de design, como responsabilidades mal distribuídas ou dependências excessivas entre BCs.

O Mapa de Contexto ajuda a visualizar e equilibrar esses aspectos, permitindo que os BCs sejam projetados para maximizar a coesão interna e minimizar o acoplamento externo, resultando em um sistema mais robusto e sustentável.

Nosso Estudo de Caso: Sakila Store (PostgreSQL)

Para tornar a discussão concreta, usaremos o conhecido schema da **Sakila Store** (versão PostgreSQL, disponível no projeto jOOQ e na documentação original do MySQL). Embora seja um schema monolítico, ele representa distintas *capacidades de negócio* que podemos usar para simular um processo de decomposição.

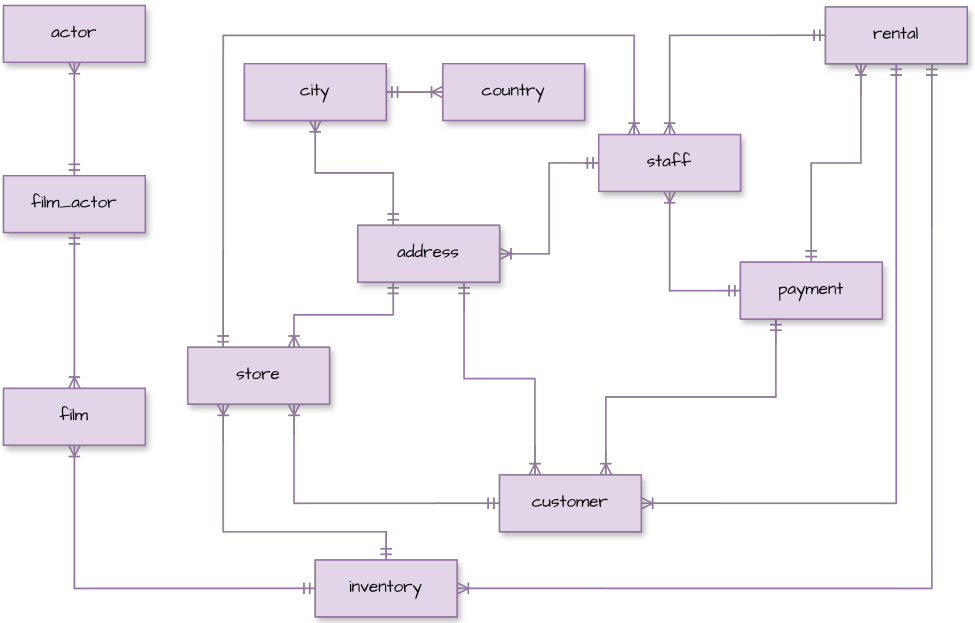


Figura 1: Modelo de Entidade-Relacionamento simplificado do Sakila.

Identificando Fronteiras: Domínios e Bounded Contexts

O primeiro passo é dissecar o monólito Sakila. Precisamos entender as áreas funcionais e identificar onde as regras de negócio mudam ou onde os termos têm significados ligeiramente diferentes. Técnicas como Event Storming ou análise de casos de uso são excelentes para isso. No entanto, olhando diretamente para o schema e a *cadeia de valor da locadora*, podemos fazer algumas inferências iniciais.

> **Foco no Core**: Ao decompor, priorize as áreas que entregam o valor principal do negócio. Otimizar domínios puramente de suporte pode não trazer o Retorno Sobre o Investimento (ROI) esperado. No Sakila, o aluguel de filmes é central.

Analisando as tabelas e restrições do Sakila, notamos alguns pontos cruciais que sugerem limites de domínio:

- `inventory`: Cada linha é uma *cópia física* de um filme em uma loja específica.
- `payment`: `rental_id` é opcional, sugerindo pagamentos não diretamente ligados a um aluguel (talvez taxas de adesão ou multas por atraso não vinculadas a um aluguel específico?).
- `address`: Usada por `customer`, `store` e `staff` – um ponto de acoplamento comum em modelos monolíticos.
- `customer`: Possui um `store_id` (loja de preferência/cadastro).
- `store`: Possui um `manager_staff_id`, ligando a um `staff`.

Com base nisso e pensando nas responsabilidades de negócio, podemos esboçar alguns domínios candidatos:

- **Core Domains**:
  - **Catalog**: Gerencia informações sobre filmes e atores (`film`, `actor`, `category`).
  - **Store Operations**: Gerencia lojas, estoque (`inventory`) e funcionários (`staff`) associados. Inclui endereços de lojas e funcionários (`address`, `city`, `country`).
  - **Rental**: Orquestra o processo de locação e devolução (`rental`), ligando `customer`, `inventory`, `staff`.
  - **Finance**: Lida com pagamentos (`payment`), vinculados a `rental`, `customer`, `staff`.
  - **Customer**: Gerencia dados dos clientes (`customer`) e seus endereços.
- **Supporting Domains**:
  - **HR**: (Poderia ser expandido a partir de `staff`) Gerencia dados mais detalhados dos funcionários.
  - **Procurement/Operations**: (Poderia ser inferido) Gerencia a aquisição de novos filmes para o catálogo.

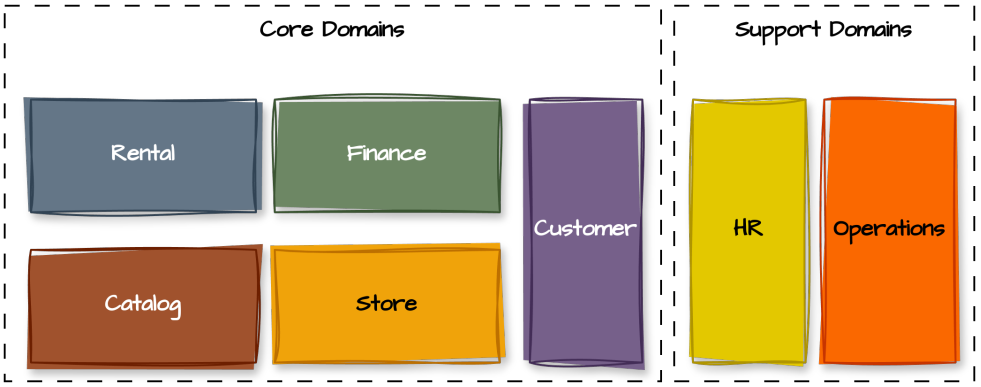


Figura 2: Possível divisão de domínios no Sakila.

Agora, traduzimos esses domínios em **Bounded Contexts (BCs)**. Lembre-se: um BC é uma fronteira explícita onde um modelo de domínio específico e sua **Linguagem Ubíqua** são consistentes.

1. **Catalog Management (BC)**: Foco em "título do filme", "gênero", "elenco". (Tabelas: `film`, `actor`, `category`, `film_actor`, `film_category`, `language`).
2. **Store & Inventory Management (BC)**: Linguagem sobre "inventário disponível na loja", "gerente da loja", "endereço comercial". (Tabelas: `store`, `inventory`, e parte de `staff`, `address`, `city`, `country` relevantes à loja).
3. **Customer Relationship (BC)**: Foco em "cliente ativo", "endereço principal do cliente", "histórico de aluguéis do cliente". (Tabelas: `customer`, e parte de `address`, `city`, `country` relevantes ao cliente).
4. **Rental Operations (BC)**: Core do negócio. Linguagem: "aluguel ativo", "devolução pendente", "item alugado". (Tabelas: `rental`, e IDs/referências a `customer`, `inventory`, `staff`).
5. **Payment Processing (BC)**: Linguagem: "pagamento recebido", "valor pendente", "data de processamento". (Tabelas: `payment`, e IDs/referências a `rental`, `customer`, `staff`).

Atenção ao Acoplamento Oculto: Note como `address`, `customer` e `staff` são referenciados ou gerenciados em múltiplos BCs propostos. Isso é um *sinal de alerta* no design monolítico e um ponto crucial a ser tratado na decomposição. Um "Cliente" no BC *Rental Operations* pode precisar apenas de ID e nome, enquanto no *Customer Relationship* temos o perfil completo.

Os domínios do negócio em complemento do mapa de contexto, nos dá uma visão mais clara das interações e dependências entre as diferentes áreas e de possíveis pontos de início para o o desenvolvimento de maneira a focar em funcionalidades que permitem o funcionamento do negócio o mais cedo possível embasados na cadeia de valor da empresa. Ex: é necessário um catálog de filmes para permitir o aluguel, as lojas são canais de venda e o processamento de pagamentos é necessário para o fechamento do aluguel. Nem tudo precisa ser automatizado desde o início da operação da empresa.

Mapeando as Relações: O Context Map

Identificar BCs é metade da batalha; a outra metade é definir *como* eles interagem. É aqui que entra o **Mapa de Contexto**. Ele nos força a nomear e padronizar as relações entre os BCs. Para visualizar isso, ferramentas como o Context Mapper são muito úteis.

Figura 3: Mapa de Contexto inicial para os BCs Sakila (sem relações definidas).

Agora, aplicamos os padrões de relacionamento do DDD:

- **Parceria (Partnership - P)**: Dois BCs/times colaboram intimamente, dependendo um do outro para o sucesso. Talvez *Rental Operations* e *Payment Processing*? Se um falhar, o outro é diretamente impactado de forma crítica. Exige alta sincronia e alinhamento.
- **Kernel Compartilhado (Shared Kernel - SK)**: Dois ou mais BCs compartilham um subconjunto do modelo (código, schema). *Cuidado aqui!* É tentador, mas pode gerar acoplamento forte e dificuldades de evolução. A tabela `address` no schema Sakila original é um exemplo clássico de SK problemático se não tratada.
- **Cliente-Fornecedor (Customer-Supplier - CS)**: Um BC (Fornecedor/Upstream) fornece serviços/dados para outro (Cliente/Downstream). O Cliente geralmente tem poder de influência sobre as prioridades do Fornecedor. Exemplo: *Rental Operations* (Cliente) precisa de informações de filmes do *Catalog Management* (Fornecedor).
- **Conformista (Conformist - CF)**: Um BC (Downstream) adere completamente ao modelo de outro BC (Upstream), sem tentar traduzir ou proteger seu próprio modelo. Isso ocorre quando o Upstream tem muito mais poder ou não tem interesse/capacidade em suportar as necessidades específicas do Downstream.

- **Camada Anticorrupção (Anticorruption Layer - ACL)**: O BC Downstream constrói uma camada de tradução para proteger seu modelo das "influências" do modelo do Upstream. É uma defesa crucial. Se *Rental Operations* consome dados de *Catalog Management*, ele pode ter uma ACL para traduzir o modelo rico de `Film` para um modelo mais simples, focado apenas no que é necessário para o aluguel.
- **Serviço Aberto ao Host (Open Host Service - OHS)**: O BC Upstream define um protocolo claro e bem documentado (ex: API REST) para acessar seus serviços/dados, aberto para qualquer consumidor autorizado.
- **Linguagem Publicada (Published Language - PL)**: Um formato bem documentado (JSON Schema, Protocol Buffers, Avro, etc.) usado para intercâmbio de informações entre BCs, geralmente associado a um OHS ou integração via eventos.

Analisando as Relações no Sakila

A análise das relações, considerando os papéis de Upstream (U - quem fornece/influencia) e Downstream (D - quem consome/é influenciado), é fundamental. A escolha do tipo de relacionamento deve alinhar-se com as necessidades do negócio e a organização das equipes para otimizar os processos.

Vamos analisar algumas interações chave:

1. **`Catalog Management` (Upstream) e `Rental Operations` (Downstream)**:
    - *Necessidade:* `Rental Operations` precisa de informações dos filmes para o aluguel.
    - *Relação Sugerida:* `Catalog Management` (U) expõe um **OHS** com uma **PL**. `Rental Operations` (D) consome esses dados, possivelmente através de uma **ACL** para adaptar as informações ao seu próprio modelo e proteger-se de mudanças no catálogo que não são relevantes para o aluguel. Se `Rental Operations` tiver influência sobre as funcionalidades do catálogo, uma relação **CS** poderia ser considerada, com `Rental Operations` como Customer.

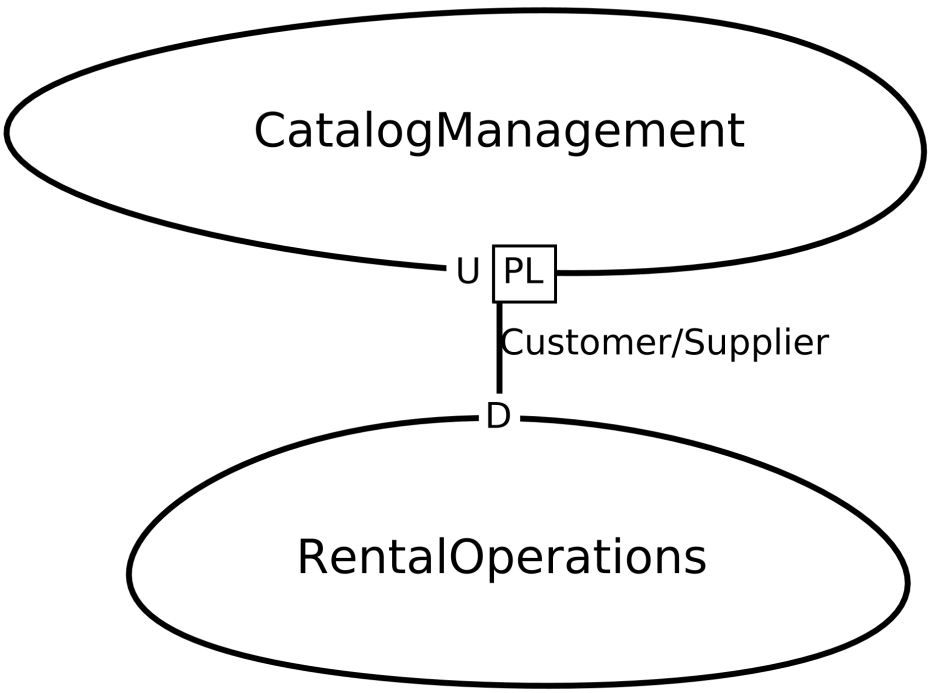


Figura 4: Exemplo de Mapa de Contexto com `Catalog Management` (Upstream/OHS+PL) e `Rental Operations` (Downstream/ACL). A seta D->U implícita na relação CS indicaria a influência do cliente no fornecedor.

2. **`Customer Relationship` (Upstream) e `Rental Operations` (Downstream)**:

    - *Necessidade:* `Rental Operations` precisa identificar o cliente e verificar seu status (ex: ativo, bloqueado).
    - *Relação Sugerida:* `Customer Relationship` (U) via **OHS/PL**. `Rental Operations` (D) pode ser **CF** se precisar apenas de dados básicos e estáveis (ID, nome, status), ou usar uma **ACL** se precisar de uma visão mais adaptada ou se o modelo de cliente for complexo/volátil.

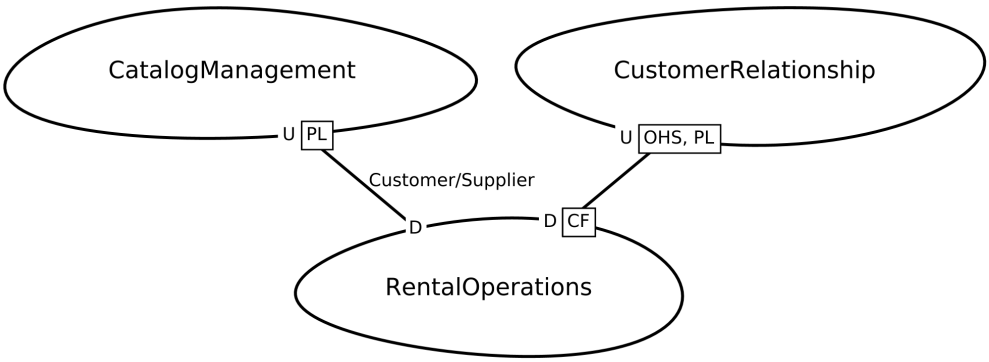


Figura 5: Mapa de Contexto com `Customer Relationship` (Upstream/OHS+PL) e `Rental Operations` (Downstream/CF).

3. **`Rental Operations` e `Payment Processing`**:

    - *Necessidade:* `Rental Operations` informa sobre aluguéis para cobrança. `Payment Processing` informa sobre o status dos pagamentos para conciliar e fechar aluguéis. Relação bidirecional.
    - *Opção 1 (Fortemente Acoplada):* Modelar como duas relações **CS** distintas, ou até uma **Partnership (P)** se a colaboração e dependência forem extremamente altas e os times trabalharem em conjunto nas evoluções.
    - *Opção 2 (Desacoplada via Eventos):* `Rental Operations` publica um evento "Aluguel Criado"; `Payment Processing` publica "Pagamento Recebido". Cada um consome os eventos do outro e reage. (Veremos mais sobre mensageria em capítulos futuros).

- *Opção 3 (Serviços):* Ambos podem expor **OHS/PL**. Por exemplo, `Rental Operations` (U) expõe um serviço para `Payment Processing` (D) consultar detalhes do aluguel, e `Payment Processing` (U) expõe um serviço para `Rental Operations` (D) consultar status de pagamento.

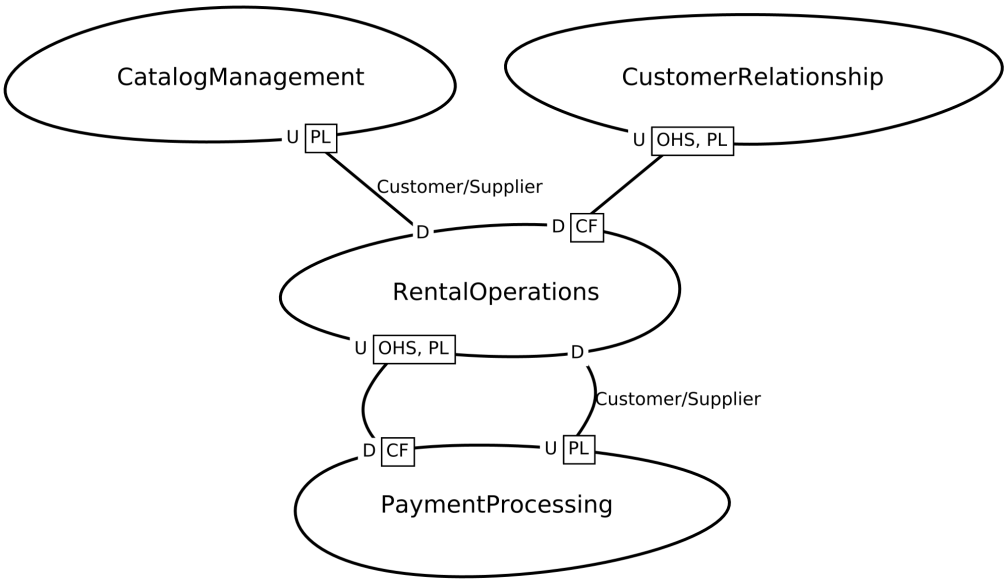


Figura 6: Mapa de Contexto ilustrando uma relação bidirecional entre `Rental Operations` e `Payment Processing`. Aqui, `Rental Operations` (U) provê dados para `Payment Processing` (D) via OHS/CF, e `Payment Processing` (U) provê dados para `Rental Operations` (D) via OHS/PL (poderia ser CS se `Rental` influenciar `Payment`).

4. **`Store & Inventory Management` (Upstream) e `Rental Operations` (Downstream)**:

   - *Necessidade:* `Rental Operations` precisa saber quais itens de inventário (`inventory`) estão disponíveis em qual loja (`store`) e qual funcionário (`staff`) processou o aluguel/devolução.
   - *Relação Sugerida:* `Store & Inventory Management` (U) via **OHS/PL**. `Rental Operations` (D) consome via **CF** ou **ACL**, dependendo da complexidade e necessidade de adaptação do modelo de inventário/loja/funcionário para o contexto de aluguel.

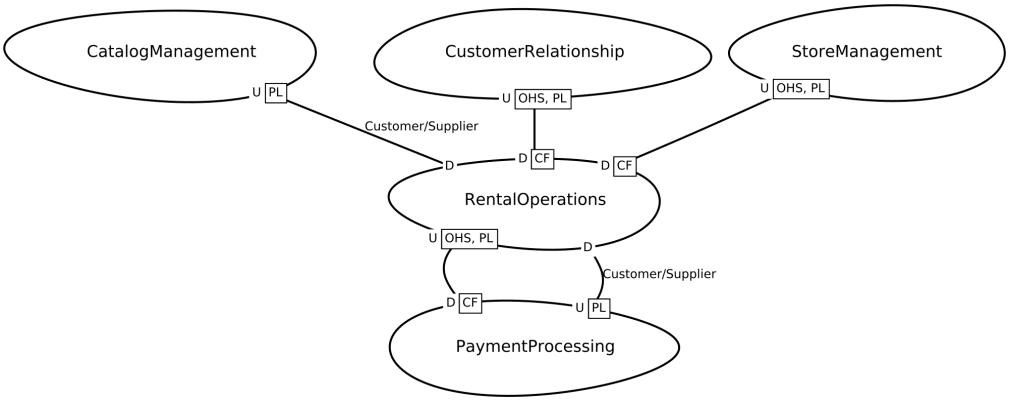


Figura 7: Mapa de Contexto com `Store & Inventory Management` (Upstream/OHS+PL) e `Rental Operations` (Downstream/CF).

5. **O Problema do `address` (Shared Kernel Implícito)**:

   - *Situação Atual (Monólito):* `Customer Relationship` e `Store & Inventory Management` (que lida com endereços de lojas e funcionários) precisam de informações de endereço. No schema original, ambos usariam a mesma estrutura (`address`, `city`, `country`). Isso configura um **Shared Kernel (SK)** problemático. Mudar a estrutura de `address` para clientes (ex: adicionar "complemento residencial") poderia quebrar a gestão de endereços de lojas ou funcionários (ex: "sala", "bloco comercial").

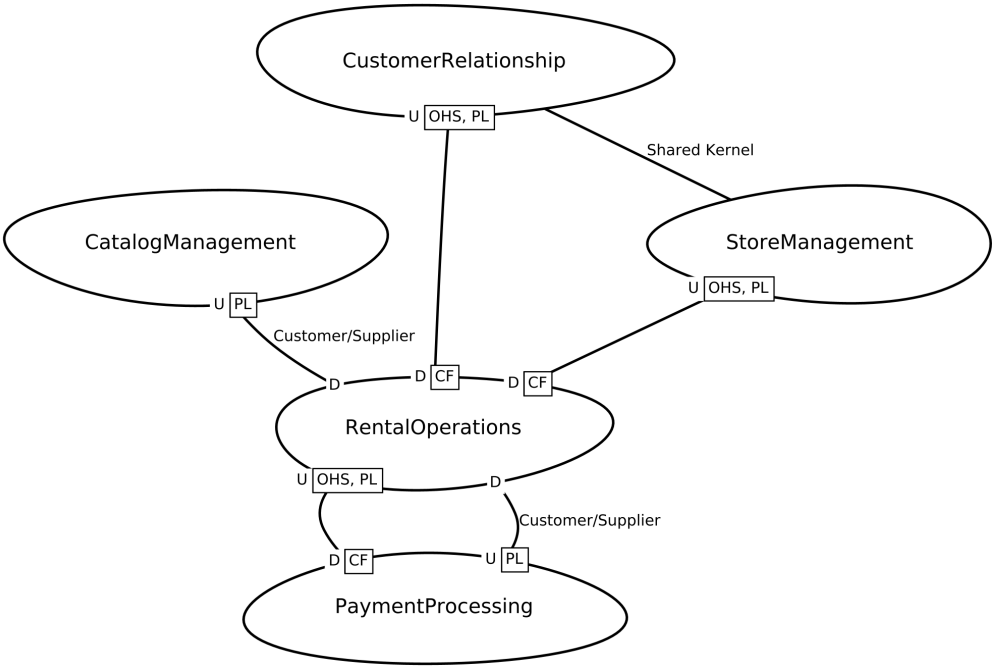


Figura 8: Mapa de Contexto mostrando o Shared Kernel implícito em `address` entre `Customer Relationship` e `Store & Inventory Management` (representado como `StoreManagement` na imagem).

- *Solução Proposta (Decomposição):* Extrair um novo **Address Management BC**.
  - Este BC seria o *dono* do modelo canônico de endereço e suas validações.
  - Exporia um **OHS/PL**.
  - `Customer Relationship` e `Store & Inventory Management` (ambos Downstream em relação ao novo BC) consumiriam esse serviço, cada um potencialmente com sua própria **ACL** para adaptar o modelo de endereço às suas necessidades específicas (ex: `Customer Address Model` vs. `Store Address Model`). Isso quebra o acoplamento indesejado do SK e permite que cada contexto evolua seu conceito de endereço de forma independente.

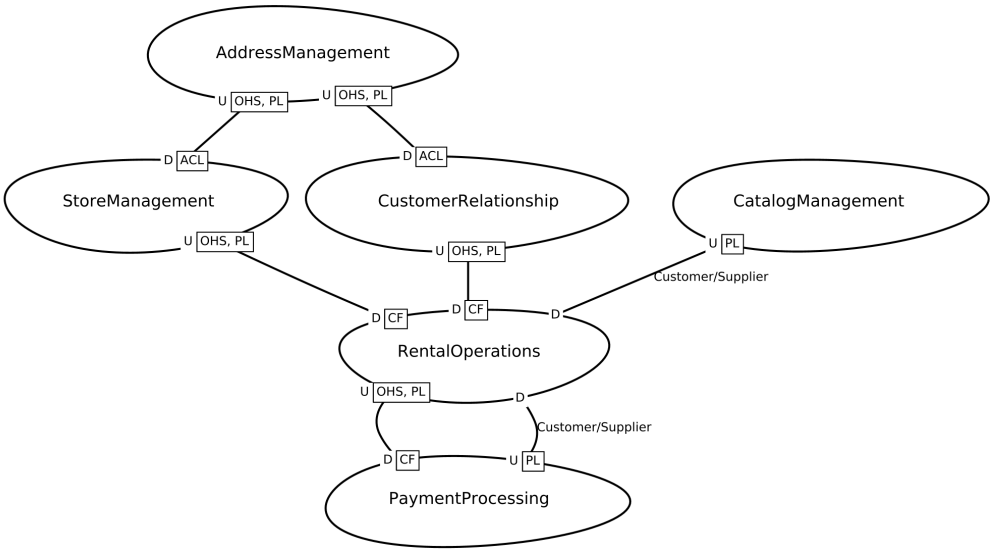


Figura 9: Mapa de Contexto com o novo `Address Management BC`, resolvendo o Shared Kernel com OHS e ACLs para `Customer Relationship` e `Store & Inventory Management` (representado como `StoreManagement` na imagem).

Por Que Investir Tempo Nisso?

- **Clareza Estratégica**: Entender as fronteiras e como as partes se conectam antes de escrever código.
- **Autonomia de Times**: BCs bem definidos com relações claras permitem que times trabalhem em paralelo com menos atrito.
- **Evolução Sustentável**: Isolar contextos com ACLs e OHS permite que um BC evolua sem quebrar os outros.
- **Tecnologia Adequada**: Cada BC pode, potencialmente, usar a tecnologia que melhor se adapta ao seu problema específico.
- **Base para Microsserviços**: Uma boa decomposição por BCs é um pré-requisito fundamental se você está considerando uma arquitetura de microsserviços.

Pontos Chave

- O Design Estratégico do DDD, com o Mapa de Contexto, ajuda a visualizar e gerenciar a complexidade em nível macro.

- Bounded Contexts definem limites explícitos para modelos de domínio e sua Linguagem Ubíqua.
- Analisar o schema existente e as capacidades de negócio ajuda a identificar candidatos a BCs.
- O Mapa de Contexto documenta as relações entre BCs usando padrões (OHS, ACL, CF, CS, SK, P, etc.).
- Identificar e tratar acoplamentos implícitos (como SKs em schemas monolíticos) é crucial para uma decomposição eficaz.
- Este mapeamento é um processo iterativo e evolui com o aprofundamento do entendimento do negócio.

Definir essas fronteiras e relações logo no início fornece um guia muito mais robusto para as próximas etapas de design tático e implementação.

Deseja se Aprofundar?

- **Livros**:
  - "Domain-Driven Design: Tackling Complexity in the Heart of Software" de Eric Evans.
  - "Implementing Domain-Driven Design" e "Domain-Driven Design Distilled" de Vaughn Vernon.
- **Comunidade e Recursos Online**:
  - DDD Crew no GitHub (inclui exemplos de Context Mapping): https://github.com/ddd-crew
  - Documentação do Context Mapper: https://contextmapper.org/docs/
  - Vídeo sobre Context Mapping
  - Vídeo sobre Context Mapping Avançado
- **Sobre o Sakila (para referência do estudo de caso)**:
  - Sakila Sample Database (MySQL): https://dev.mysql.com/doc/sakila/en/sakila-introduction.html
  - Port do Sakila para PostgreSQL (jOOQ): https://github.com/jOOQ/sakila
- **Conceitos Relacionados (Revisão)**:
  - Domain Model (Martin Fowler): https://www.martinfowler.com/eaaCatalog/domainModel.html

Capítulo 2: Visão Macro da Solução - Componentes e Serviços por Função

Este conteúdo não está disponível na amostra do livro. O livro pode ser adquirido no Leanpub em https://leanpub.com/ppajm-apps-corporativas-cloud-native-com-quarkus.

Componentes e Serviços por Função

Este conteúdo não está disponível na amostra do livro. O livro pode ser adquirido no Leanpub em https://leanpub.com/ppajm-apps-corporativas-cloud-native-com-quarkus.

Definindo as Interações e Escolhendo Padrões

Este conteúdo não está disponível na amostra do livro. O livro pode ser adquirido no Leanpub em https://leanpub.com/ppajm-apps-corporativas-cloud-native-com-quarkus.

Como Escolher Tecnologias?

Este conteúdo não está disponível na amostra do livro. O livro pode ser adquirido no Leanpub em https://leanpub.com/ppajm-apps-corporativas-cloud-native-com-quarkus.

Aplicando Padrões às Interações do Sakila

Este conteúdo não está disponível na amostra do livro. O livro pode ser adquirido no Leanpub em https://leanpub.com/ppajm-apps-corporativas-cloud-native-com-quarkus.

Considerações de Arquitetura Nesta Fase

Visualizando a Arquitetura

Capítulo 4: Organizando Regras de Negócio com Padrões do DDD Tático e Modelo de Domínio

Benefícios Desta Abordagem Macro:

Desafios a Considerar:

Concluindo

Deseja se Aprofundar?

Capítulo 3: Arquitetura de Aplicação BCE e o Princípio da Inversão de Dependência

No capítulo anterior, definimos os grandes blocos funcionais da nossa solução Sakila – os Serviços/Componentes como `Catalog Management`, `Rental Operations`, etc., baseados nos Bounded Contexts. Agora, a pergunta é: como organizar o código *dentro* de cada um desses blocos? Uma estrutura interna inconsistente ou ausente pode transformar cada serviço em uma caixa-preta distinta, tornando a manutenção um verdadeiro desafio. Essa falta de estrutura frequentemente leva ao surgimento de um ou mais dos quatro sintomas primários de degradação do design de software, conforme apresentados por Robert C. Martin em "Design Principles and Design Patterns" (2000):

1. **Rigidez**: O sistema é difícil de mudar. Pequenas alterações causam uma cascata de outras mudanças em módulos dependentes.
2. **Fragilidade**: O sistema quebra inesperadamente em múltiplos lugares como resultado de uma única mudança.
3. **Imobilidade**: É difícil reutilizar partes do sistema em outros contextos porque elas estão fortemente acopladas ao sistema original.
4. **Viscosidade**: Fazer as coisas "do jeito certo" (seguindo o design) é mais difícil do que fazer "gambiarras". Isso pode se manifestar tanto no design quanto no ambiente:

    - **Viscosidade do Design**: Ocorre quando é mais fácil implementar uma solução que viola o design original do que seguir os padrões estabelecidos. Isso geralmente acontece se o design não for flexível o suficiente ou se a equipe não compreender bem os padrões, levando a um acúmulo de dívida técnica.

        > **Exemplo**: Em um projeto onde o padrão é usar injeção de dependência para colaborações, um desenvolvedor encontra uma situação onde instanciar um colaborador diretamente parece mais rápido para uma tarefa específica. Se essa prática se espalha, o design original é minado.

    - **Viscosidade do Ambiente**: Refere-se a dificuldades impostas pelo ambiente de desenvolvimento, como ferramentas lentas, processos de build demorados ou ciclos de feedback de testes excessivamente longos.

> **Exemplo**: Se os testes automatizados levam horas para rodar, os desenvolvedores podem ser tentados a pular a execução completa antes de commitar alterações, arriscando a introdução de bugs.

Esses sintomas são sinais de alerta, indicando que a estrutura interna do código precisa de atenção. Felizmente, princípios e padrões de design nos ajudam a evitar esses problemas. A organização interna de um serviço ou componente é tão crucial quanto a arquitetura geral do sistema, especialmente em sistemas complexos desenvolvidos por múltiplas equipes.

Estrutura Interna: Padrão BCE e DIP

Em linhas gerais, a maioria das funcionalidades de software pode ser decomposta em três responsabilidades principais: receber uma entrada, realizar algum tipo de processamento ou transformação, e produzir uma saída. Essas responsabilidades se manifestam de várias formas para compor um software maior. Os padrões arquiteturais existem para organizar esses componentes e suas interações. É crucial que os padrões sirvam como guias para clareza e consistência, não como restrições artificiais.

Uma abordagem simples e eficaz para organizar a lógica dentro de uma camada de aplicação ou serviço é o padrão **Boundary-Control-Entity (BCE)**. Ele propõe a separação de responsabilidades em três tipos de objetos. Para alcançar essa separação de forma desacoplada e testável, o **Princípio da Inversão de Dependência (DIP)** é nosso aliado fundamental.

Origem do Estilo Arquitetural BCE

O estilo arquitetural **Boundary-Control-Entity (BCE)** foi introduzido por Ivar Jacobson em seu livro de 1992, *"Object-Oriented Software Engineering: A Use Case Driven Approach"*. O BCE surgiu como uma forma de organizar o design de sistemas orientados a objetos, promovendo a separação de responsabilidades em:

1. **Boundary (Fronteira)**: Objetos que interagem diretamente com atores externos ao sistema (usuários, outros sistemas). Eles isolam a interface do sistema do restante da lógica.
2. **Control (Controle)**: Objetos que gerenciam a coordenação e orquestração de casos de uso ou fluxos de negócio. Eles não contêm lógica de negócio detalhada, mas sabem quem chamar para realizá-la.

3. **Entity (Entidade)**: Objetos que encapsulam o estado e o comportamento do domínio de negócio. São o coração da aplicação, contendo as regras de negócio mais importantes.

O BCE visa criar sistemas modulares, reutilizáveis e de fácil manutenção. Sua simplicidade e clareza o tornam uma escolha interessante, especialmente quando combinado com o DIP para reforçar o desacoplamento.

O que é o DIP?

O **Princípio da Inversão de Dependência (DIP)**, o "D" do acrônimo SOLID, é crucial para construir sistemas flexíveis e resilientes a mudanças. Ele postula duas regras principais:

1. Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações (interfaces).
2. Abstrações não devem depender de detalhes. Detalhes (implementações concretas) devem depender de abstrações.

Em essência, o DIP nos instrui a depender de interfaces e não de implementações concretas, permitindo que as implementações variem sem afetar os módulos que as utilizam.

Juntando as Peças: BCE + DIP

Para manter uma arquitetura interna simples, flexível e que suporte melhoria contínua, podemos visualizar os componentes de um serviço ou aplicação em camadas, conforme a `Figura 3-1`.

> Pessoalmente, gosto de classificar as responsabilidades em algumas categorias principais para simplificar o raciocínio sobre a estabilidade e o propósito de cada parte:
>
> 1. **Lógica de Aplicação/Fluxo de Negócio (associada ao Control)**: É inerentemente dinâmica, pois os casos de uso e os fluxos de trabalho mudam com as necessidades do negócio. Não controlamos essas mudanças de requisitos; elas são motivadas por dores ou oportunidades de negócio.
> 2. **Modelo de Domínio (associado à Entity)**: Também pode ser instável à medida que o entendimento do negócio evolui. Deve ser mantido o mais puro possível (livre de dependências de infraestrutura) para que possamos controlar o impacto das mudanças nos objetos de negócio, suas relações e seus contratos de dados internos.

3. **Contratos/Interfaces (Boundary para o exterior e interfaces para Control/Entity)**: Definem os contratos funcionais (o "o quê" o sistema faz ou o "o quê" um componente oferece). Tendem a ser mais estáveis que as implementações detalhadas.
4. **Implementação/Infraestrutura (detalhes do Boundary e implementações concretas para interfaces de Control/Entity)**: Contém os detalhes tecnológicos (o "como" as coisas são feitas) e é projetada para ser extensível ou substituível. É onde frameworks como Quarkus e Spring entram, provendo implementações para as interfaces (ex: adaptadores para bancos de dados, mensageria) e a infraestrutura de execução (servidor HTTP, container DI).

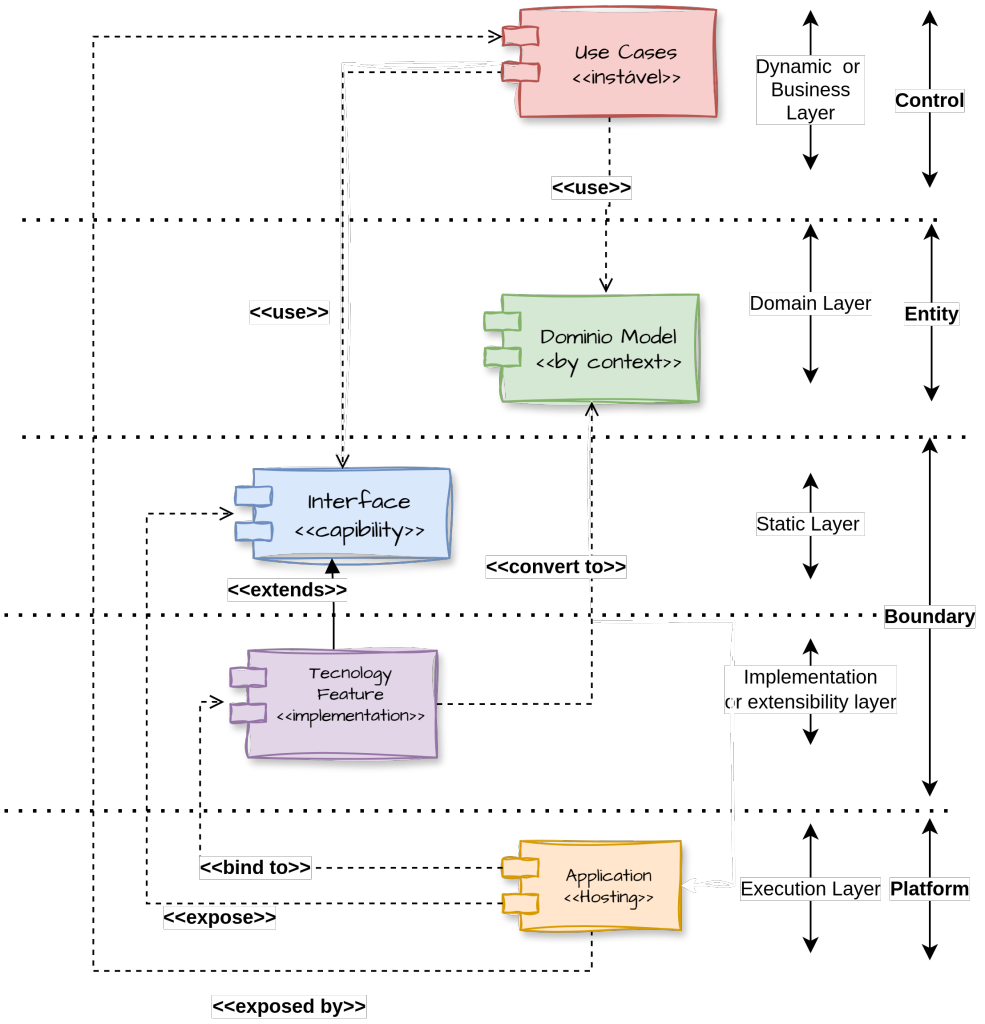


Figura 3-1: Relacionamento conceitual entre camadas com BCE e DIP.

Nesta visualização:

- **Use Cases (Casos de Uso)**: Residem na camada de Controle, orquestrando a lógica da aplicação.
- **Domain Model (Modelo de Domínio)**: Está na camada de Entidade, encapsulando as regras de negócio.
- **Interface (Contratos)**: Definem como os componentes interagem, permitindo que as implementações (detalhes tecnológicos e extensões) variem.
- **Technology Feature (Recursos Tecnológicos)**: São as implementações concretas.

- **Application (Aplicação)**: Hospeda os componentes e provê serviços de suporte (plataforma de execução).

Relação entre as Camadas com BCE + DIP

A tabela a seguir detalha as responsabilidades e exemplos de elementos Java (com foco em Quarkus) para cada estereótipo BCE, considerando o DIP:

Estereótipo / Camada	Responsabilidade	Exemplos de Elementos Java (Quarkus)
Boundary (Interface com o Exterior)	Lida com a comunicação com o mundo externo (APIs REST, UIs, Consumidores/Produtores de Mensagens). Traduz dados externos para chamadas internas e vice-versa. Não contém lógica de negócio.	Classes JAX-RS (`@Path`), WebSockets, Consumidores de Kafka (`@Incoming`), DTOs (Data Transfer Objects).
Control (Coordenação/Casos de Uso)	Orquestra os casos de uso (lógica da aplicação). Coordena interações entre Boundary e Entity. Delega a lógica de negócio às Entidades e o acesso a dados aos Repositórios (via interfaces).	Classes de Serviço de Aplicação (`@ApplicationScoped`, `@Service` em Spring), manipuladores de casos de uso. Dependem de interfaces de Repositório e, possivelmente, de Serviços de Domínio.

Estereótipo / Camada	Responsabilidade	Exemplos de Elementos Java (Quarkus)
Entity (Modelo de Domínio)	Encapsula o estado e as regras de negócio do domínio. Representa os conceitos fundamentais do negócio.	Classes de Domínio (POJOs ricos com lógica), Agregados, Entidades de Domínio, Objetos de Valor (DDD Tático). Idealmente, não dependem de frameworks de infraestrutura.
Abstrações (Interfaces Internas)	Definem os contratos para desacoplar componentes internos, especialmente para acesso a dados e outros serviços de infraestrutura.	Interfaces Java para Repositórios (ex: `CustomerRepository`, `InventoryRepository`), para Serviços de Domínio, para gateways de serviços externos (ex: `PaymentGateway`).
Infraestrutura (Implementação Detalhes)	Fornece implementações concretas para as abstrações (interfaces). Lida com preocupações de infraestrutura como persistência, mensageria, comunicação de rede.	Implementações de Repositório (ex: usando Panache ORM, JDBC), clientes HTTP para serviços externos, produtores de mensagens Kafka. Gerenciadas por Injeção de Dependência (CDI).

Nota: A tabela acima é uma interpretação aplicada, onde "Interfaces (Boundary)" do seu texto original foi explicitada como "Abstrações (Interfaces Internas)" e a camada de implementação dos detalhes de infraestrutura foi nomeada como "Infraestrutura (Implementação Detalhes)" para maior clareza na aplicação do BCE e DIP.

BCE + DIP na Prática: Estrutura de Projeto

Existem diversas formas de organizar a estrutura de pastas e módulos de um projeto. Adotar o estilo arquitetural BCE + DIP desde o início pode simplificar e padronizar o desenvolvimento. Uma granularidade interessante para o primeiro nível de módulos Java (se você estiver construindo um monólito modular ou um conjunto de microsserviços com estrutura similar) poderia ser o Contexto de Domínio (ex: `sakila-catalog`, `sakila-rental`). Dentro de cada um desses módulos de contexto, poderíamos ter sub-módulos ou pacotes para `boundary`, `control`, `entity`, e `infrastructure` (para as implementações de repositórios e outros adaptadores de infraestrutura).

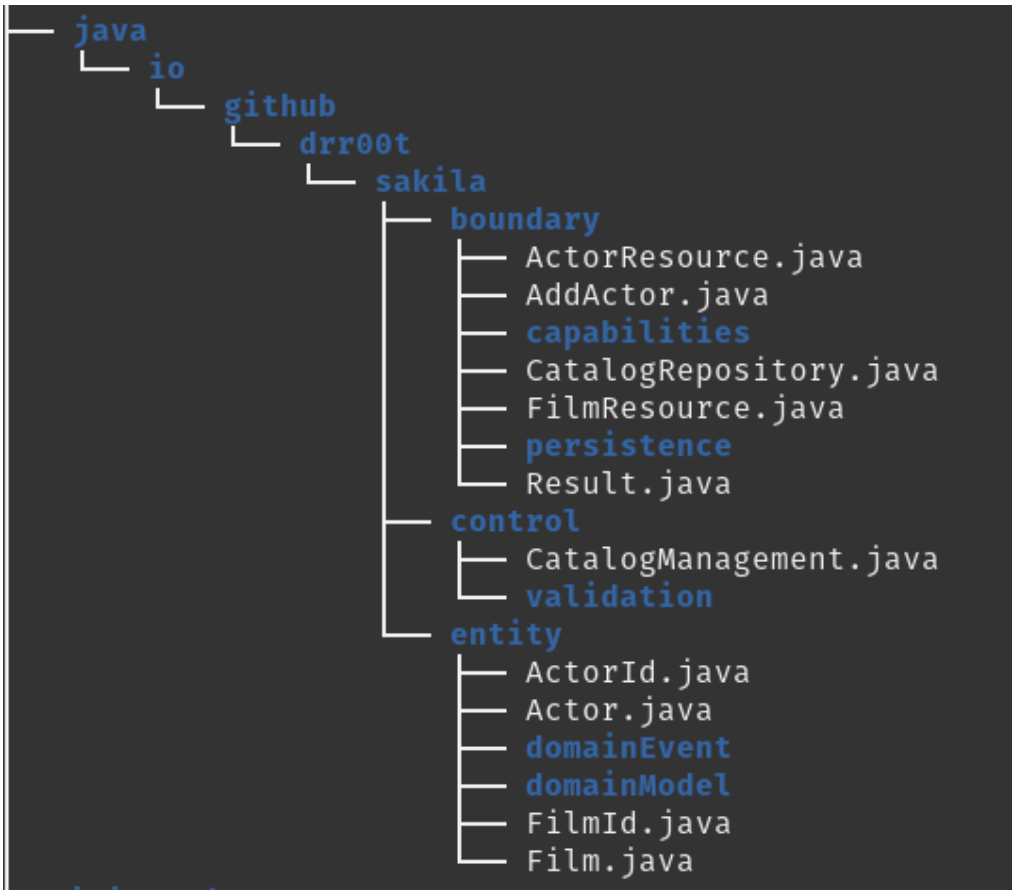


Figura 3-2: Exemplo de organização de um projeto (módulo ou serviço) seguindo BCE + DIP (Estrutura de Pacotes). Na imagem, `capabilities` poderia ser um nome alternativo ou um subpacote dentro de `boundary` para DTOs ou interfaces de Boundary, e `persistence`

dentro de `boundary` parece um equívoco, deveria estar mais associado a `infrastructure` (implementação dos repositórios).

Observação sobre a Figura 3-2: A estrutura mostrada na imagem é um bom começo. No entanto, `capabilities` e `persistence` dentro de `boundary` podem gerar confusão. Geralmente:

- DTOs (parte das `capabilities` de comunicação) podem estar em `boundary` ou um subpacote `boundary.dto`.
- Interfaces de Repositório (`CatalogRepository`) são abstrações e podem residir em `control` ou `entity` (se as entidades as usarem diretamente) ou em um pacote `domain.spi` (Service Provider Interface).
- Implementações de Repositório (`persistence`) são detalhes de infraestrutura e pertenceriam a um pacote `infrastructure.persistence`.

A estrutura exata de pacotes pode variar, mas o princípio é separar as responsabilidades.

Como Funciona com Java e Frameworks (Quarkus/CDI):

Vamos ilustrar com um exemplo simplificado de um caso de uso de locação:

1. **Definimos Interfaces (Abstrações):** Para nossos objetos `Control` (Casos de Uso/Serviços de Aplicação) e para os `Repositories` (que abstraem o acesso a dados das entidades).

```java
// Pacote: ...control ou ...application.service
// Control Interface (Abstração para o Caso de Uso de realizar uma nova locação)
public interface RentalPlacementService {
    RentalResult placeNewRental(int customerId, int inventoryId, int staffId);
    // outros métodos de caso de uso relacionados a locação...
}

// Pacote: ...domain.model.inventory ou ...domain.spi
// Repository Interface (Abstração para persistência de Inventory)
public interface InventoryRepository {
    Optional<Inventory> findAvailableById(int inventoryId);
    void update(Inventory inventory); // Para marcar como alugado, por exemplo
}

// Pacote: ...domain.model.rental ou ...domain.spi
// Repository Interface (Abstração para persistência de Rental)
```

```
17  public interface RentalRepository {
18      void save(Rental rental);
19  }
20
21  // Pacote: ...domain.model.customer ou ...domain.spi
22  // Repository Interface (Abstração para Customer - pode vir de outro BC/módulo se for\
23   um serviço separado)
24  public interface CustomerRepository {
25       Optional<Customer> findActiveById(int customerId);
26  }
27
28  // DTOs e Objetos de Resultado podem estar em boundary ou em um pacote compartilhado \
29  de API
30  // Pacote: ...boundary.dto ou ...api.dto
31  public record RentalRequestDTO(int customerId, int inventoryId, int staffId) {}
32  public record RentalConfirmationDTO(UUID rentalId, LocalDateTime rentalDate, LocalDat\
33  eTime expectedReturnDate) {}
34
35  // Pacote: ...application.result ou ...common
36  public class RentalResult { // Simplificado
37      private boolean success;
38      private List<String> errors;
39      private RentalConfirmationDTO data;
40
41      // Construtores estáticos para sucesso e falha
42      public static RentalResult success(RentalConfirmationDTO data) { /* ... */ return\
43   null;}
44      public static RentalResult failure(String... errors) { /* ... */ return null; }
45      public boolean isSuccess() { return success; }
46      public List<String> getErrors() { return errors; }
47      public RentalConfirmationDTO getData() { return data; }
48  }
```

2. **Implementamos as Entidades (Coração do Domínio):**

```
1   // Pacote: ...domain.model.rental
2   public class Rental { // Entity
3       private UUID id;
4       private Customer customer; // Ou apenas customerId se preferir
5       private Inventory inventory; // Ou apenas inventoryId
6       private int staffId;
7       private LocalDateTime rentalDate;
8       private LocalDateTime returnDate; // Data de devolução esperada/real
9
10      // Construtor privado para ser usado pelo método de fábrica estático
11      private Rental() {}
12
13      public static Rental create(Customer customer, Inventory inventory, int staffId) {
14          // Validações de negócio para criar um Rental
15          if (!customer.isActive()) {
```

```
                throw new BusinessRuleException("Cliente não está ativo.");
            }
            if (!inventory.isAvailableForRental()) {
                throw new BusinessRuleException("Item não está disponível para locação.");
            }

            Rental rental = new Rental();
            rental.id = UUID.randomUUID();
            rental.customer = customer;
            rental.inventory = inventory;
            rental.staffId = staffId;
            rental.rentalDate = LocalDateTime.now();
            // rental.returnDate = calcularDataDevolucaoEsperada();

            // Marcar o inventário como não disponível
            inventory.markAsRented(); // Método na entidade Inventory

            return rental;
        }
        // Getters, outros métodos de negócio...
        public UUID getId() { return id; }
        public LocalDateTime getRentalDate() { return rentalDate; }
        public LocalDateTime getExpectedReturnDate() { /* Lógica para calcular */ return \
    null; }
    }

    // Outras entidades como Customer, Inventory com sua própria lógica e estado.
    // Pacote: ...domain.model.inventory
    public class Inventory { // Entity
        private int id;
        private Film film; // Referência à entidade Film
        private Store store; // Referência à entidade Store
        private InventoryStatus status; // Ex: AVAILABLE, RENTED, MAINTENANCE

        public boolean isAvailableForRental() { return status == InventoryStatus.AVAILABL\
    E; }
        public void markAsRented() { this.status = InventoryStatus.RENTED; }
        // ...
    }
    public enum InventoryStatus { AVAILABLE, RENTED, MAINTENANCE }
    // Similarmente para Customer, Film, Store...
```

3. **Implementamos o Control (Serviço de Aplicação):**

```java
// Pacote: ...application.internal ou ...control.impl
@ApplicationScoped // Quarkus CDI (ou @Service em Spring)
public class RentalPlacementServiceImpl implements RentalPlacementService {

    private final InventoryRepository inventoryRepository;
    private final RentalRepository rentalRepository;
    private final CustomerRepository customerRepository;
    // private final DomainEventPublisher eventPublisher; // Para publicar eventos de\
 domínio

    @Inject // Construtor Injection é preferível para testabilidade e imutabilidade d\
e dependências
    public RentalPlacementServiceImpl(InventoryRepository inventoryRepository,
                                      RentalRepository rentalRepository,
                                      CustomerRepository customerRepository) {
        this.inventoryRepository = inventoryRepository;
        this.rentalRepository = rentalRepository;
        this.customerRepository = customerRepository;
    }

    @Override
    @Transactional // Gerenciamento de transação aqui!
    public RentalResult placeNewRental(int customerId, int inventoryId, int staffId) {
        try {
            // 1. Buscar entidades (via repositórios - que retornam entidades de domí\
nio)
            Customer customer = customerRepository.findActiveById(customerId)
                .orElseThrow(() -> new EntityNotFoundException("Cliente não encontrad\
o ou inativo: " + customerId));

            Inventory inventory = inventoryRepository.findAvailableById(inventoryId)
                .orElseThrow(() -> new EntityNotFoundException("Item de inventário nã\
o disponível: " + inventoryId));

            // Staff staff = staffRepository.findById(staffId)... (omitido para brevi\
dade)

            // 2. Executar lógica de negócio / Criar nova entidade de domínio
            // A lógica de criação e validação está encapsulada na entidade Rental
            Rental rental = Rental.create(customer, inventory, staffId);

            // 3. Persistir o estado (via repositórios)
            // O repositório de inventário precisa persistir a mudança de estado do i\
tem
            inventoryRepository.update(inventory); // Salva o estado 'RENTED' do Inve\
ntory
            rentalRepository.save(rental); // Salva o novo Rental

            // 4. (Opcional) Publicar eventos de domínio
            // eventPublisher.publish(new RentalCreatedEvent(rental.getId(), ...));
```

```
51
52              // 5. Preparar resultado de sucesso com dados para o Boundary
53              return RentalResult.success(
54                  new RentalConfirmationDTO(rental.getId(), rental.getRentalDate(), ren\
55  tal.getExpectedReturnDate())
56              );
57          } catch (BusinessRuleException | EntityNotFoundException e) {
58              // Logar o erro e retornar falha
59              // logger.warn("Falha ao processar locação: {}", e.getMessage());
60              return RentalResult.failure(e.getMessage());
61          } catch (Exception e) {
62              // Logar erro inesperado
63              // logger.error("Erro inesperado ao processar locação:", e);
64              return RentalResult.failure("Erro inesperado no sistema.");
65          }
66      }
67  }
```

4. **Implementamos o Boundary (Ex: JAX-RS Resource):**

```
1   // Pacote: ...boundary.rest
2   @Path("/rentals")
3   public class RentalResource { // Parte do Boundary
4
5       private final RentalPlacementService rentalPlacementService; // Injetando a abstr\
6   ação do Serviço de Aplicação
7
8       @Inject
9       public RentalResource(RentalPlacementService rentalPlacementService) {
10          this.rentalPlacementService = rentalPlacementService;
11      }
12
13      @POST
14      @Consumes(MediaType.APPLICATION_JSON)
15      @Produces(MediaType.APPLICATION_JSON)
16      public Response createRental(RentalRequestDTO request) {
17          // Chama o serviço de aplicação (Control)
18          RentalResult result = rentalPlacementService.placeNewRental(
19              request.customerId(),
20              request.inventoryId(),
21              request.staffId()
22          );
23
24          if (!result.isSuccess()) {
25              // Mapear para um status HTTP apropriado, ex: 400 Bad Request ou 422 Unpr\
26  ocessable Entity
27              return Response.status(Response.Status.BAD_REQUEST)
28                             .entity(Map.of("errors", result.getErrors()))
29                             .build();
30          }
```

```
            // Retorna 201 Created com a confirmação da locação
            return Response.status(Response.Status.CREATED).entity(result.getData()).buil\
d();
        }
    }
```

5. **Implementamos os Repositórios (Infraestrutura):**

```
    // Pacote: ...infrastructure.persistence.inventory
    @ApplicationScoped
    public class PanacheInventoryRepositoryImpl implements InventoryRepository {
        // Implementação usando Panache ORM do Quarkus para a entidade JPA InventoryEntit\
y.
        // Esta classe faz a tradução entre InventoryEntity (JPA) e Inventory (Domínio).
        @Override
        public Optional<Inventory> findAvailableById(int inventoryId) {
            // Exemplo:
            // return PanacheInventoryEntity.find("id = ?1 and status = 'AVAILABLE'", inv\
entoryId)
            //       .firstResultOptional()
            //       .map(this::toDomain); // Método para converter Entity JPA para Domín\
io
            return Optional.empty(); // Placeholder
        }
        @Override
        public void update(Inventory inventory) {
            // Exemplo:
            // PanacheInventoryEntity entity = PanacheInventoryEntity.findById(inventory.\
getId());
            // if (entity != null) {
            //    updateEntityFromDomain(entity, inventory); // Método para atualizar Ent\
ity JPA a partir do Domínio
            //    entity.persist();
            // }
        }
        // private Inventory toDomain(PanacheInventoryEntity entity) { /* ... */ }
        // private void updateEntityFromDomain(PanacheInventoryEntity jpaEntity, Inventor\
y domainEntity) { /* ... */ }
    }
    // ... outras implementações de repositório para Rental, Customer (ex: PanacheRentalR\
epositoryImpl)
```

6. **Container de Injeção de Dependência (DI):** O framework (Quarkus CDI) gerencia as instâncias. Ele encontra as implementações (`RentalPlacementServiceImpl`, `PanacheInventoryRepositoryImpl`) e as injeta onde as interfaces (`RentalPlacementService`, `InventoryRepository`) são requisitadas (`@Inject`). Assim, `RentalResource` não conhece a implementação de

`RentalPlacementService`, e `RentalPlacementServiceImpl` não conhece a implementação de `InventoryRepository`.

Por que BCE + DIP é uma Dupla Poderosa?

- **Organização Clara**: Separa as preocupações de forma lógica:
  - **Boundary**: Como o sistema interage com o mundo externo (APIs, mensagens, UI).
  - **Control**: O que o sistema faz (orquestração de casos de uso).
  - **Entity**: O coração do negócio (regras e estado do domínio).
- **Testabilidade Aprimorada**:
  - **Boundary** pode ser testado isoladamente, mockando o `Control` (Serviço de Aplicação).
  - **Control** pode ser testado isoladamente, mockando os `Repositories` e outras dependências de infraestrutura ou domínio.
  - **Entities** podem ser testadas como POJOs puros, focando em suas regras de negócio intrínsecas, sem a necessidade de um container ou banco de dados.
- **Desacoplamento Forte**: As camadas e componentes se conhecem predominantemente por interfaces. Trocar uma implementação de repositório (ex: de JPA para um NoSQL específico) ou o tipo de `Boundary` (ex: de REST para gRPC) tem impacto minimizado nas outras partes do sistema.
- **Flexibilidade e Manutenibilidade**: O código torna-se mais fácil de entender, modificar e estender, pois as responsabilidades estão bem localizadas. A adição de novos casos de uso ou a alteração de regras de negócio pode ser feita com maior confiança.

Pontos Chave

- BCE organiza o código *dentro* de um componente/serviço em Boundary, Control e Entity.
- **Boundary** lida com I/O, tradução de dados entre o formato externo e o interno, e delega para o Control.

- **Control** (Serviços de Aplicação/Casos de Uso) orquestra a lógica da aplicação, interagindo com Entidades (via seus próprios métodos ou via Serviços de Domínio) e Repositórios (para persistência).
- **Entity** (Modelo de Domínio) encapsula o estado e as regras de negócio intrínsecas do domínio.
- O **DIP**, implementado via interfaces e Injeção de Dependência, permite que Boundary, Control, Entity e Infraestrutura colaborem de forma desacoplada.
- Frameworks como Quarkus (com CDI) facilitam enormemente a implementação do DIP.
- A estrutura resultante é mais organizada, testável, flexível e manutenível.
- A lógica de negócio é distribuída de forma clara: coordenação e fluxos no Control, regras e comportamento intrínsecos nas Entities. O próximo capítulo detalhará mais as Entities e os padrões táticos do DDD (Agregados, Serviços de Domínio, Eventos de Domínio).

Deseja se Aprofundar Mais?

- **Livros Fundamentais**:
  - Martin, Robert C. *"Agile Software Development, Principles, Patterns, and Practices."* (Contém o artigo original "Design Principles and Design Patterns" de 2000, onde o DIP é bem explorado).
  - Evans, Eric. *"Domain-Driven Design: Tackling Complexity in the Heart of Software."* (2003) - A bíblia do DDD.
  - Vernon, Vaughn. *"Implementing Domain-Driven Design."* (2013) - Foco prático na implementação.
  - Fowler, Martin. *"Patterns of Enterprise Application Architecture."* (2002) - Descreve muitos padrões usados em conjunto com BCE/DIP.
- **Artigos e Recursos Online**:
  - Busque por "SOLID principles Robert C Martin" para artigos e vídeos sobre DIP.
  - Explore a documentação de Quarkus sobre CDI (Contexts and Dependency Injection) e design de aplicações.
  - Wikipedia - Entity Control Boundary
  - sebastian Daschner, How to Structure Modern enterprise Java Projects

Capítulo 4: Organizando Regras de Negócio com Padrões do DDD Tático e Modelo de Domínio

Nos capítulos anteriores, definimos as fronteiras estratégicas (Contextos Delimitados e Mapa de Contexto) e a estrutura macro dos nossos componentes/serviços (visão por função e o padrão BCE). Discutimos Fronteiras (Boundary), Controles (Control) e Entidades (Entity). Agora, é hora de aprofundar no "E" do BCE – as Entidades – e em como realmente organizar a lógica de negócio complexa que reside ali. É aqui que os **padrões táticos do Domain-Driven Design (DDD)** entram em jogo.

Se o DDD Estratégico nos forneceu o mapa geral do território, o DDD Tático nos oferece as ferramentas de construção detalhadas para modelar o núcleo do nosso software: o **Modelo de Domínio**. O objetivo é criar um modelo que não seja apenas um reflexo passivo dos dados, mas um participante ativo na execução das regras de negócio, um modelo **rico** e expressivo.

O Domain-Driven Design, tanto em seus aspectos estratégicos quanto táticos, é fundamentalmente sobre gerenciar e reduzir o acoplamento desnecessário, ao mesmo tempo em que reconhece que **algum nível de acoplamento é inevitável e até desejável** – desde que seja dentro de limites bem definidos, como dentro de um Agregado.

Os padrões táticos do DDD nos fornecem as abstrações e práticas para criar um modelo de domínio que seja coeso, expressivo e fácil de entender. Esses padrões ajudam a organizar a lógica de negócio, a proteger o estado das entidades e a garantir que as regras de negócio sejam aplicadas corretamente, promovendo a clareza e o controle da complexidade ao longo do tempo.

Antes de mergulharmos nos padrões táticos, vale revisitar dois conceitos fundamentais que permeiam todo o DDD: **acoplamento** e **coesão**. Eles são cruciais para garantir que nosso modelo de domínio seja não apenas funcional, mas também sustentável e fácil de evoluir.

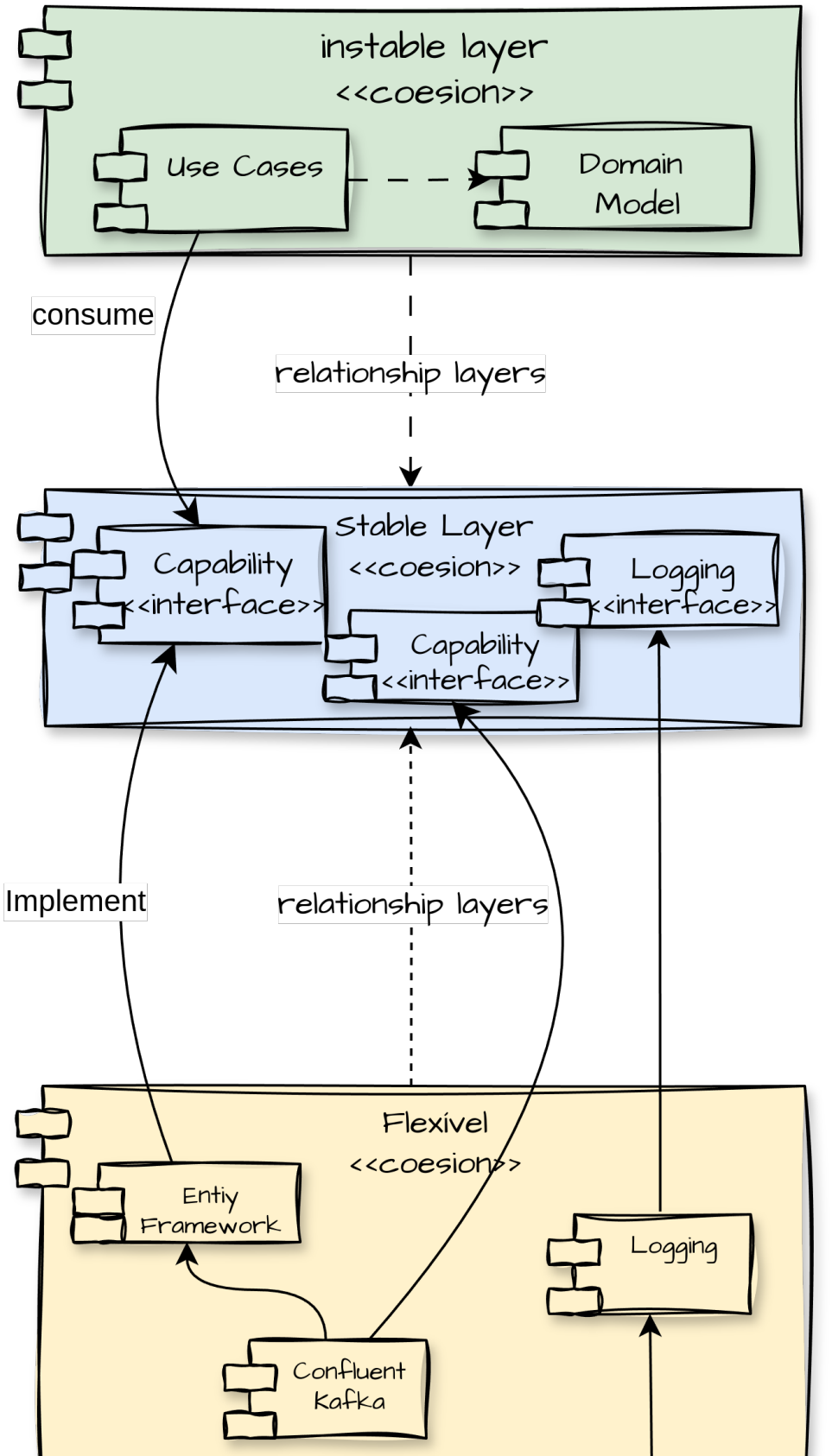

instable layer
<<coesion>>
Use Cases
Domain Model
consume
relationship layers
Stable Layer
<<coesion>>
Capability
<<interface>>
Logging
<<interface>>
Capability
<<interface>>
Implement
relationship layers
Flexível
<<coesion>>
Entiy Framework
Logging
Confluent Kafka

Figura 4-1: Visualização conceitual de acoplamento e coesão.

Acoplamento e Coesão na Prática

No Capítulo 1, discutimos acoplamento e coesão sob a perspectiva do Mapa de Contexto, analisando como diferentes contextos delimitados colaboram.

- **Acoplamento**: Refere-se à força da dependência entre componentes. Um sistema com baixo acoplamento possui componentes que podem ser alterados ou substituídos com impacto mínimo em outros. Isso facilita a manutenção e a evolução. No entanto, um acoplamento excessivamente baixo pode dificultar a colaboração necessária entre componentes.
- **Coesão**: Mede o quão bem os elementos dentro de um componente estão relacionados e focados em uma responsabilidade específica. Um componente altamente coeso é mais fácil de entender e manter.

Neste capítulo, nosso foco é no acoplamento e coesão *dentro* de um Bounded Context, especificamente no nível do código (classes e métodos). Um design de código bem pensado, utilizando padrões táticos, busca maximizar a coesão dentro dos elementos do domínio (como Agregados) e gerenciar cuidadosamente o acoplamento entre eles.

Modelo de Domínio Rico vs. Anêmico: Uma Decisão Crítica

Um erro comum, especialmente em abordagens mais antigas focadas em dados ou em interpretações superficiais do padrão MVC, é criar um **Modelo de Domínio Anêmico**. Nele, as Entidades são meros sacos de dados com getters e setters públicos para todos os seus atributos, e quase nenhuma lógica de negócio. Toda a lógica de negócio reside nos Serviços de Aplicação (os "Controls" do BCE).

```java
// Exemplo de Modelo Anêmico - EVITAR!
public class Rental { // Apenas dados, sem comportamento
    private int id;
    private Instant rentalDate;
    private Instant returnDate;
    private Customer customer;
    // ... getters e setters públicos para todos os campos ...

    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
    // ... e assim por diante ...
}

public class RentalApplicationService { // Toda a lógica aqui
    private RentalRepository rentalRepository;
    private PaymentService paymentService; // Outro serviço de aplicação

    public void processFilmReturn(int rentalId, Instant returnTime) {
        Rental rental = rentalRepository.findById(rentalId)
                            .orElseThrow(() -> new RuntimeException("Rental not found\
"));

        // Modificação direta do estado da entidade por fora
        rental.setReturnDate(returnTime);

        // Lógica de negócio espalhada no serviço de aplicação
        BigDecimal fine = calculateFineForRental(rental);
        if (fine.compareTo(BigDecimal.ZERO) > 0) {
            paymentService.chargeFine(rental.getCustomer().getId(), fine);
        }
        rentalRepository.save(rental);
    }

    private BigDecimal calculateFineForRental(Rental rental) {
        // ... lógica de cálculo de multa ...
        return BigDecimal.ZERO; // Placeholder
    }
}
```

Problemas do Modelo Anêmico:

- As regras de negócio ficam espalhadas pelos serviços de aplicação.
- O estado da Entidade pode ser modificado de forma inconsistente por qualquer consumidor.
- A coesão dentro das Entidades é baixa (são apenas dados).
- A coesão nos Serviços de Aplicação tende a diminuir à medida que acumulam mais lógica.

- A testabilidade da lógica de negócio se torna mais difícil, pois está atrelada aos serviços.

O **Modelo de Domínio Rico** é o oposto. As Entidades e Agregados contêm tanto os dados quanto o comportamento (métodos de negócio) que opera sobre esses dados, protegendo suas invariantes.

```
// Exemplo de Modelo Rico (primeira aproximação)
// Assumindo que RentalId, CustomerId, InventoryId, Money, Fine são VOs ou tipos defi\
nidos
// e BaseAggregateRoot fornece funcionalidade básica como gerenciamento de ID e regis\
tro de eventos.
public class Rental extends BaseAggregateRoot<RentalId> {
    private RentalId id;
    private CustomerId customerId;
    private InventoryId inventoryId;
    private Instant rentalDate;
    private Instant returnDate; // Null se não devolvido

    // Construtor privado ou protegido para ser chamado pelo método fábrica
    private Rental(RentalId id, CustomerId customerId, InventoryId inventoryId, Insta\
nt rentalDate) {
        super(id); // Passa o ID para a classe base
        this.customerId = Objects.requireNonNull(customerId, "CustomerId não pode ser\
 nulo.");
        this.inventoryId = Objects.requireNonNull(inventoryId, "InventoryId não pode \
ser nulo.");
        this.rentalDate = Objects.requireNonNull(rentalDate, "RentalDate não pode ser\
 nulo.");
    }

    // Método fábrica estático para criação controlada
    public static Rental createNewRental(RentalId rentalId, CustomerId customerId, In\
ventoryId inventoryId) {
        // Validações de pré-condição para a criação do aluguel
        // (Ex: verificar se cliente está ativo, se item está disponível - poderia us\
ar Domain Services aqui)
        Objects.requireNonNull(rentalId, "RentalId não pode ser nulo.");
        // ... outras validações ...

        Rental rental = new Rental(rentalId, customerId, inventoryId, Instant.now());
        rental.registerEvent(new FilmRentedEvent(rental.getId(), rental.customerId, r\
ental.inventoryId, rental.rentalDate));
        return rental;
    }

    // Método de negócio que encapsula regras e estado
    public Optional<Fine> processReturn(Instant actualReturnTime, OverdueFineCalculat\
or fineCalculator) {
```

```
        if (this.isReturned()) {
            throw new BusinessRuleException("Filme já devolvido em " + this.returnDat\
e);
        }
        if (actualReturnTime.isBefore(this.rentalDate)) {
            throw new BusinessRuleException("Data de devolução inválida (anterior à d\
ata de locação).");
        }

        this.returnDate = actualReturnTime; // Modificação de estado controlada

        Optional<Money> fineAmount = fineCalculator.calculate(this.rentalDate, this.r\
eturnDate, /* outros params */);

        if (fineAmount.isPresent() && fineAmount.get().isGreaterThanZero()) {
            this.registerEvent(new FilmReturnedOverdueEvent(this.id, this.customerId\
, fineAmount.get()));
            return Optional.of(new Fine(this.id, this.customerId, fineAmount.get()));
        } else {
            this.registerEvent(new FilmReturnedOnTimeEvent(this.id, this.customerId)\
);
            return Optional.empty();
        }
    }

    public boolean isReturned() {
        return this.returnDate != null;
    }
    // Getters para expor estado de forma controlada, se necessário.
    // Setters diretos para campos críticos são evitados.
}

public class RentalApplicationService { // Mais enxuto, focado na orquestração
    private RentalRepository rentalRepository;
    private OverdueFineCalculator fineCalculator; // Domain Service injetado

    @Inject // Assumindo injeção de dependência
    public RentalApplicationService(RentalRepository rentalRepository, OverdueFineCal\
culator fineCalculator) {
        this.rentalRepository = rentalRepository;
        this.fineCalculator = fineCalculator;
    }

    @Transactional // Gerencia a transação
    public Optional<Fine> processFilmReturn(RentalId rentalId, Instant actualReturnTi\
me) {
        Rental rental = rentalRepository.findById(rentalId)
            .orElseThrow(() -> new EntityNotFoundException("Aluguel não encontrado: "\
 + rentalId));

```

```
93          Optional<Fine> fine = rental.processReturn(actualReturnTime, fineCalculator);
94          rentalRepository.save(rental); // Persiste o Agregado (e seus eventos)
95          return fine;
96      }
97  }
```

> **Nota sobre Exceções**: No exemplo de Modelo Rico, `BusinessRuleException` é usada para simplicidade. Em capítulos futuros, discutiremos alternativas como o padrão Result para um tratamento de falhas mais funcional e explícito, evitando o uso excessivo de exceções para controle de fluxo.

A ideia de usar um sistema de padrões é como reconhecer uma melodia: mesmo sem saber tocar um instrumento, você percebe a harmonia das notas. Com os padrões de projeto, buscamos criar um vocabulário e uma estrutura que tornem a "música" do nosso código coesa e compreensível.

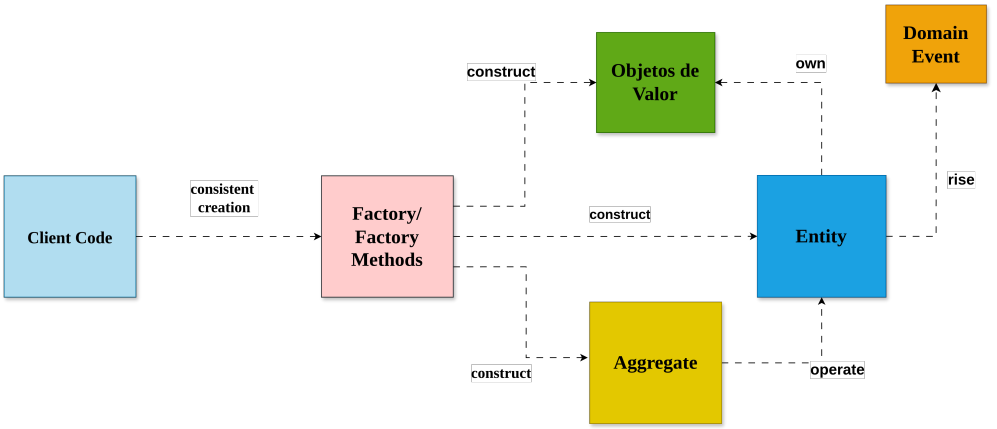


Figura 4-2: Sistema de Padrões Táticos do DDD e suas inter-relações.

Padrões Táticos do DDD

A tabela abaixo resume os principais padrões táticos e seus papéis:

Padrão Tático	Descrição Resumida	Propósito Principal
Value Object	Objeto imutável definido por seus atributos, sem identidade conceitual.	Encapsular conceitos descritivos, garantir validade, aumentar expressividade.
Entity	Objeto com identidade única que persiste ao longo do tempo, mesmo se seus atributos mudarem.	Representar coisas do domínio que têm um ciclo de vida e cuja identidade é importante.
Aggregate Root and Aggregate	A Entidade principal de um Agregado (AR) e o cluster de objetos (Entidades e VOs) que ele gerencia como uma unidade de consistência.	Controlar acesso, garantir consistência transacional e proteger invariantes dentro do Agregado.
Factory	Objeto ou método responsável por encapsular a lógica de criação de objetos complexos (VOs, Entidades, Agregados).	Simplificar a criação, garantir que objetos sejam criados em estado válido, isolar complexidade.
Domain Event	Objeto que representa algo significativo que aconteceu no domínio.	Comunicar mudanças de estado de forma desacoplada entre Agregados ou Bounded Contexts.

Value Objects (VOs): Descritivos, Imutáveis e Seguros

- **O que são?** Objetos definidos puramente pelo valor de seus atributos. Não possuem uma identidade conceitual própria. Dois VOs são iguais se todos os seus atributos forem iguais.
- **Características Chave**:
  - **Imutabilidade**: Uma vez criados, não podem ser alterados. Qualquer "modificação" resulta em uma nova instância. Isso simplifica o rastreamento de estado e evita efeitos colaterais.

  - **Auto-Validação**: A lógica para validar os atributos deve estar no construtor ou em métodos fábrica estáticos. Um VO nunca deve existir em um estado inválido.
  - **Comportamento**: Podem (e devem) ter métodos que encapsulam lógica relacionada aos valores que representam (ex: `Money.add(Money another)`, `Email.getDomainPart()`).
  - **Expressividade e Segurança**: Substituem tipos primitivos (String, int, BigDecimal) por conceitos do domínio, tornando o código mais claro, seguro e menos propenso a erros de atribuição (ex: usar `Email` em vez de `String`, `MonetaryAmount` em vez de `BigDecimal`).

- **Exemplos Sakila**:

  - `MonetaryAmount`: Para `payment.amount`, `film.rental_rate`. Conteria `BigDecimal value` e `Currency currency`.
  - `RentalDuration`: Para `film.rental_duration`. Poderia ser `int days`.
  - `FilmRating`: Para `film.rating`. Um Enum (`G, PG, PG_13, R, NC_17`) é uma forma simples de VO.
  - `AddressValue`: Se `address` não precisasse de um ID e fosse sempre gerenciado como parte de `Customer` ou `Store`, poderia ser um VO.
  - **IDs como VOs**: `ActorId`, `FilmId`, `CustomerId` podem ser VOs simples envolvendo um `int` ou `UUID`, melhorando a segurança de tipo.

```java
// Pacote: io.github.drr00t.sakila.domain.common.model (ou específico do contexto)
public final class ActorId { // 'final' para reforçar imutabilidade
    private final int value;

    private ActorId(int value) { // Construtor privado
        if (value <= 0) { // Auto-validação
            throw new IllegalArgumentException("Actor ID must be positive.");
        }
        this.value = value;
    }

    public static ActorId of(int value) { // Método Fábrica
        return new ActorId(value);
    }

    public int getValue() {
        return value;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
```

```
        if (o == null || getClass() != o.getClass()) return false;
        ActorId actorId = (ActorId) o;
        return value == actorId.value;
    }

    @Override
    public int hashCode() {
        return Objects.hash(value);
    }

    @Override
    public String toString() { // Útil para logging e debugging
        return String.valueOf(value);
    }
}

// Exemplo de uso com um teste
class ActorIdTest {
    @Test
    void testActorIdCreation() {
        ActorId actorId = ActorId.of(1);
        assertEquals(1, actorId.getValue());
    }

    @Test
    void testInvalidActorId() {
        assertThrows(IllegalArgumentException.class, () -> ActorId.of(-1));
    }

    @Test
    void testActorIdEquality() {
        ActorId actorId1 = ActorId.of(1);
        ActorId actorId2 = ActorId.of(1);
        ActorId actorId3 = ActorId.of(2);
        assertEquals(actorId1, actorId2);
        assertNotEquals(actorId1, actorId3);
    }
}
```

- O VO `ActorId` encapsula o identificador de um ator.
- A remoção da sobrecarga `of(ActorId)` simplifica, pois a conversão pode ser feita pelo chamador se necessário (`ActorId.of(outroActorId.getValue())`).
- O `toString()` foi simplificado para retornar apenas o valor, o que pode ser mais prático.

`Value Objects` são excelentes para encapsular lógica de validação e comportamento relacionado a um conceito de valor. No entanto, ao interagir

com frameworks de persistência (como JPA com `@Embeddable`) ou serialização (como Jackson), pode haver a necessidade de um construtor público (ou pelo menos `protected`) sem argumentos e/ou setters, o que pode comprometer a imutabilidade pura se não gerenciado com cuidado. Discutiremos estratégias para lidar com isso mais à frente.

Entities: Quando a Identidade Importa

- **O que são?** Objetos que possuem uma identidade única que persiste ao longo do tempo, mesmo que seus atributos mudem. Dois objetos Entidade são diferentes se tiverem IDs diferentes, mesmo que todos os outros atributos sejam iguais.
- **Características**:
  - Possuem um ciclo de vida (criação, modificações, exclusão).
  - Sua identidade (geralmente um ID, que pode ser um VO) é fundamental e estável.
  - São mutáveis, mas suas modificações de estado devem ser controladas por métodos de negócio que protegem as invariantes.
- **Exemplos Sakila**: `Customer`, `Film`, `Actor`, `Rental`, `Inventory`, `Staff`, `Store`, `Payment`.

Separando as Entidades segundo seu papel Primário ou secundário

- **Entidades Raiz de Agregado (Aggregate Roots - ARs)**: São as Entidades primárias que definem a fronteira de um Agregado. São acessadas diretamente por meio de Repositórios. Ex: `Rental`, `Customer`, `Film`.
- **Entidades Filhas (Child Entities)**: Existem dentro de um Agregado e seu ciclo de vida é gerenciado pelo AR. Não são acessadas diretamente de fora do Agregado. Ex: `RentalItem` (se existisse) dentro do Agregado `Rental`.

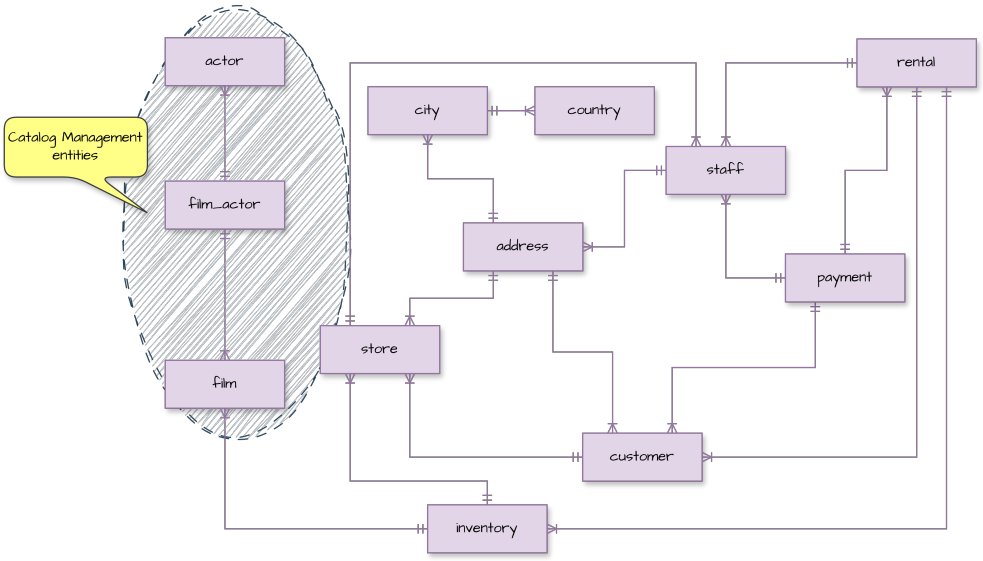


Figura 1. Imagem representando entidades do catálogo, como Actor, Film e FilmActor

Figura 4-3: Entidades do Domínio de Catálogo no Sakila (ator, filme, filme_ator).

Analisando a `Figura 4-3`, vemos o modelo de dados relacional para o catálogo, com `actor`, `film` e a tabela de junção `film_actor`. Essa estrutura normalizada permite cadastrar atores e filmes independentemente e depois associá-los.

Considerações de Negócio vs. Modelo de Dados Normalizado: O modelo de dados normalizado é flexível, mas nem sempre reflete como o negócio opera ou como os usuários interagem com o sistema. Perguntas a se fazer sobre o contexto de "Gerenciamento de Catálogo de Filmes":

- Faz sentido cadastrar um ator que não está em nenhum filme do catálogo?
- A principal função é gerenciar filmes e, como parte disso, listar seus atores?
- Qual a importância da unicidade e gestão detalhada de um `Actor` como uma entidade independente *neste contexto específico*?

Se, no contexto de gerenciamento do catálogo para a locadora, a ênfase for nos filmes, e os atores forem principalmente atributos descritivos do filme, o modelo de domínio pode divergir do modelo de dados relacional. Por exemplo, `Film` poderia ser um AR, e os atores poderiam ser uma lista de VOs (`ActorInfo`) ou referências (`ActorId`) dentro do Agregado `Film`. A decisão de tornar `Actor` um Agregado próprio depende se ele tem um ciclo de vida e regras de negócio independentes e complexas.

Se a decisão de negócio for que **ator não é um dado obrigatório para o cadastro do filme** e que a gestão detalhada de atores é secundária, poderíamos até considerar que, dentro

do Agregado `Film`, a informação do ator é apenas um nome. Isso simplificaria o Agregado `Film`, mas poderia levar à duplicação de nomes de atores se não houver uma entidade `Actor` separada para consulta e padronização.

> **Ponto Crítico**: É crucial entender o contexto e as prioridades do negócio. Um modelo de domínio rico não é apenas sobre código, mas sobre refletir fielmente o domínio e suas regras. A normalização de dados é importante para o armazenamento, mas o modelo de domínio foca na coesão e no comportamento.

```
// Pacote: io.github.drr00t.sakila.domain.catalog.model
// Assumindo que Actor é um AR independente (para gerenciamento de atores)
public class Actor extends BaseAggregateRoot<ActorId> { // Actor é um AR
    private Name firstName;
    private Name lastName;

    private Actor(ActorId id, Name firstName, Name lastName) {
        super(id);
        this.setFirstName(firstName); // Usa setters para validação
        this.setLastName(lastName);
    }

    public static Actor createNew(ActorId id, Name firstName, Name lastName) {
        Objects.requireNonNull(id, "ActorId não pode ser nulo.");
        // Validações de Name (VO) ocorrem em sua própria criação
        return new Actor(id, firstName, lastName);
    }

    public Name getFirstName() { return firstName; }
    public Name getLastName() { return lastName; }

    public void changeName(Name newFirstName, Name newLastName) {
        this.setFirstName(newFirstName);
        this.setLastName(newLastName);
        this.registerEvent(new ActorNameChangedEvent(this.getId(), newFirstName, newL\
astName));
    }

    private void setFirstName(Name firstName) {
        Objects.requireNonNull(firstName, "Primeiro nome não pode ser nulo.");
        this.firstName = firstName;
    }
    private void setLastName(Name lastName) {
        Objects.requireNonNull(lastName, "Último nome não pode ser nulo.");
        this.lastName = lastName;
    }
}

```

```
// Pacote: io.github.drr00t.sakila.domain.catalog.model
public class Film extends BaseAggregateRoot<FilmId> { // Film é um AR
    private Title title;
    private Description description;
    private final Set<Actor> actors; // Mantém um conjunto' de Atores

    private Film(FilmId id, Title title, Description description) {
        super(id);
        this.setTitle(title); // Usa setters para validação
        this.setDescription(description);
        this.actors = new HashSet<>();
    }

    public static Film createNew(FilmId id, Title title, Description description) {
        Objects.requireNonNull(id, "FilmId não pode ser nulo.");
        return new Film(id, title, description);
    }

    public Title getTitle() { return title; }
    public Description getDescription() { return description; }
    public Set<ActorId> getActorIds() { return Collections.unmodifiableSet(actorIds);\
 }

    public void setTitle(Title title) {
        Objects.requireNonNull(title, "Título não pode ser nulo.");
        this.title = title;
    }
    public void setDescription(Description description) { this.description = descript\
ion; }

    public void assignActor(Actor actor) {
        Objects.requireNonNull(actor, "Ator não pode ser nulo para atribuição.");
        if (this.actors.contains(actor)) {
            // Pode ser silencioso ou lançar exceção, dependendo da regra de negócio
            throw new BusinessRuleException("Ator " + actorId + " já está atribuído a\
 este filme.");
        }

        this.actors.add(actor);
        this.registerEvent(new ActorAssignedToFilmEvent(this.getId(), actor.getId()));
    }

    public void removeActor(Actor actor) {
        Objects.requireNonNull(actor, "Ator não pode ser nulo para remoção.");
        if (!this.actors.contains(actor)) {
            // Pode ser silencioso ou lançar exceção
            throw new BusinessRuleException("Ator " + actor.getId() + " não está atri\
buído a este filme.");
        }
        this.actors.remove(actor);
```

```
        this.registerEvent(new ActorRemovedFromFilmEvent(this.getId(), actor.getId())\
);
    }
}

// Exemplo de teste
class FilmTest {
    @Test
    void testAssignActorToFilm() {
        Film film = Film.createNew(FilmId.of(1), new Title("Um Filme Genial"), new De\
scription("Uma descrição."));
        ActorId actorId = ActorId.of(10); // Assumindo que este ator existe e é válido

        film.assignActor(actorId);

        assertTrue(film.getActorIds().contains(actorId));
        assertEquals(1, film.getActorIds().size());
    }

    @Test
    void testAssignDuplicateActorToFilm() {
        Film film = Film.createNew(FilmId.of(1), new Title("Filme com Duplicata"), ne\
w Description(""));
        Actor actor = Actor.of(actorId);
        film.assignActor(actor);
        film.assignActor(actor); // vai lançar exceção
        assertThrows(BusinessRuleException.class, () -> film.assignActor(actor));
        assertEquals(1, film.getActorIds().size()); // Garante que não foi adicionado\
 duas vezes
    }
}
```

Nota: A decisão de `Actor` ser um AR próprio ou apenas uma informação dentro de `Film` (ou `Film` conter uma lista de `ActorInfo` VOs) é uma decisão chave de modelagem de Agregado. Se `Actor` tem seu próprio ciclo de vida complexo e regras de negócio independentes do `Film` (ex: biografia, prêmios, etc.), ele deve ser um AR. Se, no contexto do catálogo de filmes, ele é apenas um nome associado a um filme, uma abordagem mais simples pode ser melhor. Para o Sakila, `Actor` tem sua própria tabela e identidade, sugerindo que é uma Entidade (e potencialmente um AR) por direito próprio.

Aggregates e Aggregate Roots (ARs): Guardiões da Consistência

Este é um dos conceitos táticos mais importantes do DDD, especialmente relevante quando se considera a decomposição em microsserviços, pois a fronteira do Agregado frequentemente influencia a fronteira de um microsserviço para garantir consistência transacional.

- **O que é um Agregado?** Um cluster de uma ou mais Entidades e Value Objects relacionados que são tratados como uma única unidade de consistência. Qualquer operação que mude o estado dentro do Agregado deve garantir que todas as suas regras de negócio (invariantes) sejam satisfeitas ao final da transação.
- **O que é um Aggregate Root (AR)?** Dentro de cada Agregado, uma única Entidade é designada como a Raiz do Agregado.
  - É a **única porta de entrada** para qualquer comando ou consulta que modifique ou acesse o estado interno do Agregado.
  - Referências externas (de outros Agregados ou de Serviços de Aplicação) só podem apontar para o AR (geralmente por seu ID).
- **Regras Fundamentais dos Agregados**:
  1. A Raiz do Agregado tem identidade global. Entidades internas ao Agregado têm identidade local (única dentro do Agregado).
  2. A Raiz do Agregado é responsável por carregar e gerenciar o ciclo de vida das Entidades e VOs internos.
  3. A Raiz do Agregado é responsável por garantir que todas as invariantes do Agregado sejam mantidas.
  4. Objetos fora do Agregado **não podem** manter referências diretas a Entidades internas; devem passar pelo AR.
  5. A exclusão da Raiz do Agregado implica na remoção de todo o Agregado.
  6. Apenas Raízes de Agregado devem ser obtidas diretamente dos Repositórios. Objetos internos são carregados junto com o AR.
  7. Objetos fora do Agregado **não podem** modificar o estado interno de partes do Agregado diretamente; devem invocar métodos no AR.
- **Regra de Ouro da Transação**: Uma transação de banco de dados deve modificar no máximo um único Agregado. Se uma regra de negócio requer a modificação de múltiplos Agregados, isso deve ser orquestrado usando consistência eventual (ex: via Domain Events e Sagas). Vaughn Vernon, em "Effective Aggregate Design", explora o impacto do tamanho da agregação no desempenho e na complexidade, sugerindo que Agregados muito grandes podem levar a problemas.

- **Aplicando ao Sakila (Exemplos de Modelagem de Agregados)**:
  - **`Rental` como Agregado**: `Rental` (AR). Contém `RentalId`, `CustomerId`, `InventoryItemId`, `StaffId`, `rentalDate`, `returnDate`.
  - **`Payment` e `Rental`**: Provavelmente Agregados distintos, como discutido anteriormente, devido a ciclos de vida e regras de consistência potencialmente separadas.
  - **`Film` como Agregado**: `Film` (AR). Contém `FilmId`, `title`, `description`, e uma coleção de `ActorIds`. `Actor` é outro Agregado.
  - **`Customer` como Agregado**: `Customer` (AR). Contém `CustomerId`, dados do cliente, e `AddressValue` (VO).
  - **`Store` como Agregado**: `Store` (AR). Gerencia uma coleção de `InventoryItems` (Entidades filhas) ou `InventoryItemIds` (se `InventoryItem` for seu próprio AR).

Agregados definem as fronteiras de consistência. Se um Agregado se torna muito grande (muitas entidades internas, muitas regras cruzadas), pode ser um sinal para reavaliar e, possivelmente, dividi-lo.

Qual o tamanho ideal de um agregado? Não há um número mágico. Guias:

- **Coesão**: O Agregado deve encapsular conceitos que mudam juntos e precisam ser consistentes entre si.
- **Transações**: Pense na unidade atômica de uma operação de negócio.
- **Performance**: Agregados menores geralmente carregam mais rápido.
- **Complexidade**: Agregados menores são mais fáceis de entender. Colaborar com analistas de negócio e especialistas em UX/UI é crucial, pois os Agregados frequentemente refletem como os usuários interagem com o sistema e como as operações de negócio são percebidas.

```java
// Exemplo de Agregado Rental (revisado e simplificado para focar no conceito de Agre\
gado)
// Pacote: io.github.drr00t.sakila.domain.rental.model
public class Rental extends BaseAggregateRoot<RentalId> {
    private final Customer customerId;
    private final InventoryItem inventoryItemId; // ID do item específico alugado
    private final Staff staff;
    private final Instant rentalDate;
    private Instant returnDate; // Mutável
    private Payment payment; // Referência opcional a um pagamento associado (outro A\
gregado)

    private Rental(RentalId id, Customer customer, InventoryItem inventoryItem, Staff\
 staff, Instant rentalDate) {
        super(id);
        this.customer = Objects.requireNonNull(customer);
        this.inventoryItem = Objects.requireNonNull(inventoryItem);
        this.staff = Objects.requireNonNull(staff);
        this.rentalDate = Objects.requireNonNull(rentalDate);
    }

    // Método fábrica para criar um novo aluguel.
    // Validações mais complexas (envolvendo outros agregados ou estado global)
    // seriam feitas ANTES de chamar este método, possivelmente em um Domain Service \
ou Application Service.
    public static Rental create(RentalId id, Customer customer, InventoryItem invento\
ryItem, Staff staff) {
        // Assume-se que o cliente está ativo, item está disponível, etc. (verificaçõ\
es prévias)
        Rental rental = new Rental(id, customerId, inventoryItemId, staffId, Instant.\
now());
        // Evento de criação é registrado. O InventoryItem associado precisaria ser m\
arcado como RENTED
        // em uma transação separada ou via consistência eventual (ex: por um listene\
r do FilmRentedEvent).
        rental.registerEvent(new FilmRentedEvent(rental.getId(), customerId, inventor\
yItemId, rental.rentalDate));
        return rental;
    }

    public void processReturn(Instant actualReturnTime) {
        if (this.returnDate != null) {
            throw new BusinessRuleException("Aluguel já devolvido em: " + this.return\
Date);
        }
        if (actualReturnTime.isBefore(this.rentalDate)) {
            throw new BusinessRuleException("Data de devolução inválida.");
        }
        this.returnDate = actualReturnTime;
        // O cálculo de multa e associação de pagamento seriam tratados separadamente,
```

```
        // possivelmente acionados por um evento FilmReturnedEvent.
        this.registerEvent(FilmReturnedEvent.of(this.getId(), customerId, inventoryIt\
emId, actualReturnTime));
    }

    public void associatePayment(Payment payment) {
        Objects.requireNonNull(payment);
        if (this.payment != null) {
            throw new BusinessRuleException("Aluguel já possui um pagamento associado\
: " + this.payment);
        }
        this.payment = payment;
        // Registrar evento se necessário
    }

    public boolean isReturned() { return this.returnDate != null; }
    // Getters...
    public Customer getCustomer() { return customer; }
    public InventoryItem getInventoryItem() { return inventoryItem; }
    // ...
}
```

O exemplo de `StoreInventory` foi removido desta seção para manter o foco no `Rental` como um Agregado mais central. O conceito de Agregado `Store` gerenciando `InventoryItems` é válido, mas pode ser detalhado em um contexto específico de inventário.

Domain Events: Notificando o Mundo sobre Mudanças Significativas

- **O que são?** Objetos que representam algo que aconteceu no domínio que é de interesse para outras partes do sistema (dentro do mesmo Bounded Context ou em outros).
- **Características**:
  - Nomeados no tempo passado (ex: `FilmRented`, `PaymentProcessed`, `CustomerAddressChanged`).
  - Imutáveis.
  - Contêm dados suficientes para que os consumidores do evento entendam o que aconteceu (ex: IDs relevantes, timestamps, valores alterados).
- **Propósito**:
  - Desacoplar Agregados: Um Agregado publica um evento, e outros Agregados ou Serviços de Aplicação podem reagir a ele sem um acoplamento direto.
  - Facilitar a consistência eventual entre Agregados ou Bounded Contexts.

  - Auditoria e rastreabilidade.
  - Integração com sistemas externos.

- **Como usar**:

  1. Um Agregado, ao final de uma operação de negócio bem-sucedida que altera seu estado, registra um ou mais Domain Events.
  2. Após a persistência bem-sucedida do Agregado (geralmente ao final da transação do Serviço de Aplicação), esses eventos são publicados.
  3. A publicação pode ser para um dispatcher de eventos síncrono (dentro do mesmo processo) ou para um message broker assíncrono (Kafka, RabbitMQ) para comunicação inter-serviços.

- **Exemplo de Registro de Evento (dentro do Agregado `Rental`)**:

```
// Assumindo que BaseAggregateRoot tem um método registerEvent(DomainEvent event)
// e uma forma de coletar esses eventos para publicação.

// Exemplo de um Domain Event (imutável, geralmente um record ou classe com campos fi\
nal)
// Pacote: io.github.drr00t.sakila.domain.rental.event
public record RentalPaymentProcessedEvent(
    RentalId rentalId,
    Payment payment,
    MonetaryAmount amountPaid, // MonetaryAmount é um VO
    Instant processedAt
) implements DomainEvent { // DomainEvent é uma interface marcadora
    public RentalPaymentProcessedEvent(RentalId rentalId, Payment payment, MonetaryAm\
ount amountPaid) {
        this(rentalId, payment, amountPaid, Instant.now());
    }
}


// Dentro do Agregado Payment (assumindo Payment é um AR)
// Pacote: io.github.drr00t.sakila.domain.payment.model
public class Payment extends BaseAggregateRoot<PaymentId> {
    private final RentalId rentalId; // Referência ao aluguel associado
    private final Customer customer;
    private final Staff staff;
    private final MonetaryAmount amount;
    private final Instant paymentDate;
    private PaymentStatus status;

    private Payment(PaymentId id, RentalId rentalId, Customer customer, Staff staff, \
MonetaryAmount amount) {
```

```
        super(id);
        this.rentalId = Objects.requireNonNull(rentalId);
        this.customer = Objects.requireNonNull(customer);
        this.staff = Objects.requireNonNull(staff);
        this.amount = Objects.requireNonNull(amount);


        this.paymentDate = Instant.now();
        this.status = PaymentStatus.PENDING; // Pagamento inicia como pendente
    }

    public static Payment createNew(RentalId rentalId, Customer customer, Staff staff\
, MonetaryAmount amount) {
        Payment payment = new Payment(PaymentId.new(), rentalId, customer, staff, amo\
unt);

         if (!amount.isGreaterThanZero())
            throw new IllegalArgumentException("Valor do pagamento deve ser positivo.\
");
        // Registrar evento de criação se for relevante para outros processos
        // payment.registerEvent(new PaymentInitiatedEvent(...));
        return payment;
    }

    public void confirmPayment() {
        if (this.status != PaymentStatus.PENDING) {
            throw new BusinessRuleException("Pagamento não está pendente, status atua\
l: " + this.status);
        }
        this.status = PaymentStatus.CONFIRMED;
        // Evento crucial para notificar que o pagamento foi processado.
        // O Rental Agregado pode escutar este evento para atualizar seu status de pa\
gamento.
        this.registerEvent(new RentalPaymentProcessedEvent(this.rentalId, this.getId(\
), this.amount, this.paymentDate));
    }

    public void failPayment(String reason) {
        if (this.status != PaymentStatus.PENDING) {
             throw new BusinessRuleException("Pagamento não está pendente, status atu\
al: " + this.status);
        }
        this.status = PaymentStatus.FAILED;
        // Registrar evento de falha se necessário
        // this.registerEvent(new PaymentFailedEvent(this.getId(), reason));
    }
    // Getters...
    public RentalId getRentalId() { return rentalId; }
    public MonetaryAmount getAmount() { return amount; }
    public PaymentStatus getStatus() { return status; }
```

```
}
enum PaymentStatus { PENDING, CONFIRMED, FAILED }
// MonetaryAmount seria um VO: record MonetaryAmount(BigDecimal value, Currency curre\
ncy) { ... }
```

A classe `BaseAggregateRoot` conteria uma lista de eventos e o método `registerEvent`. O Repositório ou um interceptador do Serviço de Aplicação seria responsável por coletar e despachar esses eventos após o `save()` do Agregado.

Design by Contract e o Modelo de Domínio Rico

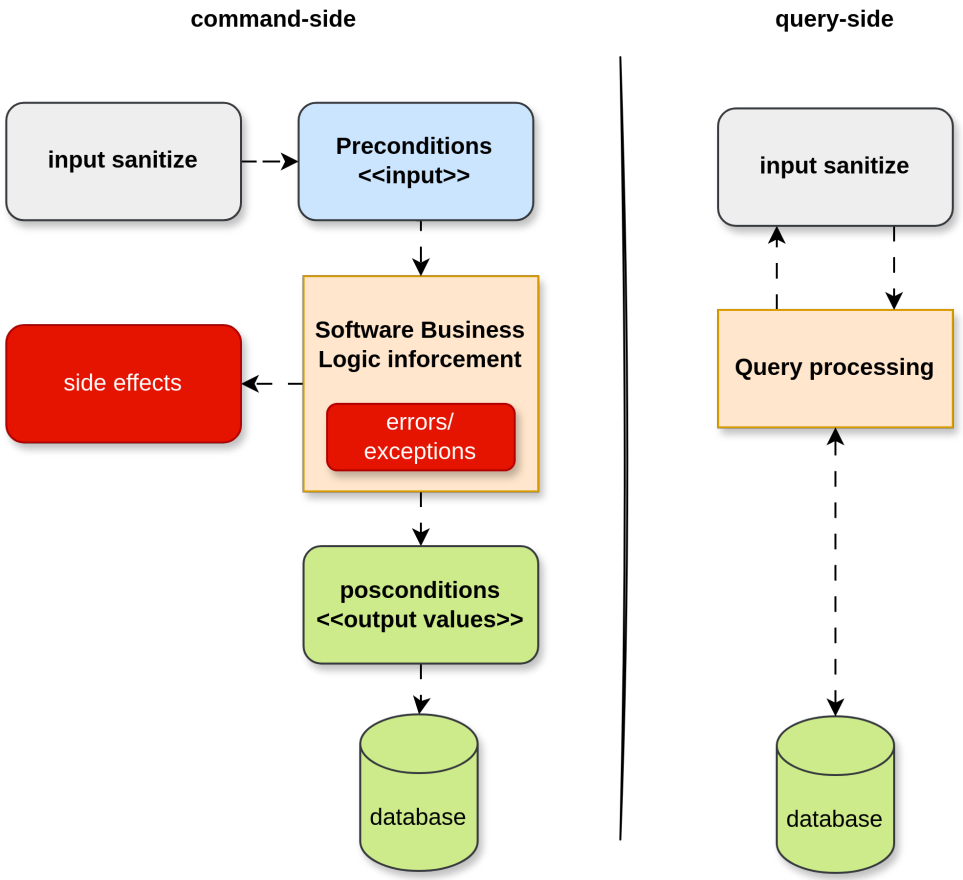


Figura 4-3: Separação entre Comandos (que podem ter pré/pós-condições e alterar estado) e Consultas (que não alteram estado).

O conceito de **Design by Contract (DbC)**, introduzido por Bertrand Meyer, complementa bem a ideia de um Modelo de Domínio Rico. DbC sugere que os métodos de um objeto devem definir um contrato claro:

- **Pré-condições**: O que deve ser verdadeiro *antes* que o método seja executado. Responsabilidade do chamador.
- **Pós-condições**: O que o método garante que será verdadeiro *após* sua execução bem-sucedida. Responsabilidade do método.
- **Invariantes**: Condições que devem ser verdadeiras para uma instância de uma classe sempre que ela estiver em um estado estável (ex: antes e depois de qualquer chamada a um método público).

Meyer também enfatizou a **Separação Comando-Consulta (CQS - Command-Query Separation)**:

- **Comandos**: Métodos que modificam o estado do objeto (efeitos colaterais) mas não retornam dados (geralmente `void` ou retornam apenas um status de sucesso/falha, ou o ID do objeto modificado/criado).
- **Consultas**: Métodos que retornam dados sobre o estado do objeto mas não o modificam.

A `Figura 4-3` (anteriormente referenciada) ilustra essa separação. Aplicar CQS e DbC ao projetar os métodos de seus Agregados e Entidades torna seus contratos mais claros e o sistema mais previsível.

```
// Exemplo de Comando no Agregado Customer (parte do Design by Contract implícito)
public class Customer extends BaseAggregateRoot<CustomerId> {
    private Name firstName;
    private Name lastName;
    private Email email; // Email é um VO
    private boolean active;
    private String deactivationReason;
    // StoreId storeId; // Loja de cadastro/preferência

    private Customer(CustomerId id, Name firstName, Name lastName, Email email /*, St\
oreId storeId */) {
        super(id);
        this.setFirstName(firstName); // Usa setters para validações
        this.setLastName(lastName);
        this.setEmail(email);
        // this.storeId = Objects.requireNonNull(storeId);
        this.active = true; // Clientes são criados ativos por padrão
```

```
    }

    public static Customer createNew(CustomerId id, Name firstName, Name lastName, Em\
ail email /*, StoreId storeId */) {
        // Validações adicionais se necessário
        return new Customer(id, firstName, lastName, email /*, storeId */);
    }

    public void deactivate(String reason) { // Comando
        // Pré-condição: cliente deve estar ativo
        if (!this.active) {
            throw new BusinessRuleException("Cliente já está inativo. Razão: " + this\
.deactivationReason);
        }
        // Pré-condição: razão deve ser fornecida
        if (reason == null || reason.isBlank()) {
            throw new IllegalArgumentException("Uma razão para desativação é obrigató\
ria.");
        }

        this.active = false;
        this.deactivationReason = reason;

        this.registerEvent(new CustomerDeactivatedEvent(this.getId(), reason));
    }

    public void reactivate() { // Comando
        if (this.active) {
            throw new BusinessRuleException("Cliente já está ativo.");
        }
        this.active = true;
        this.deactivationReason = null; // Limpa a razão anterior
        this.registerEvent(new CustomerReactivatedEvent(this.getId()));
    }

    // Getters (Consultas)
    public boolean isActive() { return this.active; }
    public Optional<String> getDeactivationReason() { return Optional.ofNullable(this\
.deactivationReason); }
    public Name getFirstName() { return firstName; }
    // ... outros getters e setters controlados ...

}
// Email seria um VO: record Email(String address) { public Email { if (!isValid(addr\
ess)) throw ... } }
```

Escopo das Regras de Negócio

As regras de negócio podem ser validadas em diferentes níveis:

1. **Nível do Value Object**: Validações intrínsecas ao valor (formato, intervalo). Ex: um `Email` VO valida se a string é um e-mail bem formado.
2. **Nível da Entidade (Raiz de Agregado ou filha)**: Regras que envolvem o estado interno da entidade e suas partes. Protege invariantes da entidade.
3. **Nível do Agregado (via Aggregate Root)**: Regras que garantem a consistência entre múltiplas Entidades e VOs *dentro* de um único Agregado. Essas são as invariantes mais amplas protegidas pela Raiz do Agregado.
4. **Nível do Serviço de Domínio**: Regras que envolvem múltiplos Agregados ou requerem acesso a informações externas ao Agregado que está sendo modificado.
5. **Nível do Serviço de Aplicação (Controle)**: Orquestração, validações de fluxo de caso de uso que não são puramente regras de domínio (ex: autorização), ou coordenação de etapas que envolvem múltiplos Agregados e consistência eventual.

Neste modelo, o Agregado `Rental` protege seu próprio estado e contém a lógica relevante para si. O Serviço de Aplicação orquestra, buscando o Agregado, invocando o método de negócio apropriado e persistindo o resultado. Isso resulta em um sistema muito mais coeso e encapsulado.

Conectando Tudo

- Os padrões táticos (Agregados, Entidades, VOs) formam o coração do Modelo de Domínio, que reside na camada "Entity" do BCE.
- Repositórios (interfaces definidas no domínio, implementações na infraestrutura) são usados pelos Serviços de Aplicação ("Control") para buscar e salvar Agregados.
- Serviços de Domínio são chamados pelos Serviços de Aplicação ou, às vezes, pelos próprios Agregados para executar lógica de negócio complexa que não se encaixa naturalmente em um único Agregado.
- Factories ajudam a criar esses objetos de domínio complexos, garantindo seu estado inicial válido.
- Domain Events comunicam mudanças significativas de forma desacoplada.

A arquitetura BCE com um modelo de domínio rico ajuda a deixar clara a intenção do código, separando as preocupações de negócio (lógica de domínio) das preocupações de infraestrutura (persistência, APIs, etc.). Isso resulta em um sistema mais fácil de entender, manter e evoluir.

```
|   |               └── catalog
|   |                   ├── boundary
|   |                   |   ├── ActorResource.java
|   |                   |   ├── AddActor.java
|   |                   |   ├── capabilities
|   |                   |   |   └── Repository.java
|   |                   |   ├── CatalogRepository.java
|   |                   |   ├── FilmResource.java
|   |                   |   ├── persistence
|   |                   |   |   ├── ActorRecord.java
|   |                   |   |   └── FilmRecord.java
|   |                   |   └── Result.java
|   |                   ├── control
|   |                   |   ├── CatalogManagement.java
|   |                   |   └── validation
|   |                   |       ├── AndValidationRule.java
|   |                   |       ├── NotValidationRule.java
|   |                   |       ├── OrValidationRule.java
|   |                   |       ├── ValidationResult.java
|   |                   |       └── ValidationRule.java
|   |                   └── entity
|   |                       ├── ActorId.java
|   |                       ├── Actor.java
|   |                       ├── domainEvent
|   |                       |   ├── ActorAddedEvent.java
|   |                       |   └── ActorUpdatedEvent.java
|   |                       ├── domainModel
|   |                       |   ├── Aggregate.java
|   |                       |   ├── BaseDomainEvent.java
|   |                       |   ├── BaseEntity.java
|   |                       |   ├── DomainEvent.java
|   |                       |   └── PrimaryBaseEntity.java
|   |                       ├── FilmDescription.java
|   |                       ├── FilmDuration.java
|   |                       ├── FilmId.java
|   |                       ├── Film.java
|   |                       ├── FilmRating.java
|   |                       ├── FilmYear.java
|   |                       ├── Name.java
|   |                       ├── Rating.java
|   |                       └── Title.java
```

Estrutura de diretórios/pacotes com o objetivo de organizar os objetos tornando mais expressivo o papel de cada pacote e sua responsabilidade.

Pontos Chave

- O DDD Tático fornece os blocos de construção (Agregados, Entidades, VOs, Serviços de Domínio, Repositórios, Factories, Eventos de Domínio) para construir um Modelo de Domínio robusto e expressivo.
- **Agregados** são o pilar para gerenciar a consistência transacional. Modele-os com cuidado, focando em pequenas unidades transacionais.
- **Value Objects** imutáveis aumentam a clareza, segurança e expressividade. Use-os generosamente para substituir tipos primitivos por conceitos de domínio.
- Diferencie claramente **Serviços de Aplicação** (orquestração de casos de uso, lidam com transações, segurança, etc.) de **Serviços de Domínio** (lógica de negócio "órfã" que não pertence a um Agregado específico).
- Busque um **Modelo de Domínio Rico**, onde Entidades/Agregados contêm comportamento e protegem suas invariantes, em contraste com um Modelo Anêmico.
- **Repositórios** abstraem a persistência, focando na recuperação e salvamento de Agregados inteiros.

Com essas ferramentas táticas, podemos construir um modelo de domínio para o Sakila (e para sistemas corporativos em geral) que seja não apenas um reflexo dos dados, mas um verdadeiro motor para as regras de negócio, tornando nosso software mais resiliente, compreensível e fácil de evoluir.

Deseja se Aprofundar Mais?

- **Livros**:
  - Evans, Eric. *"Domain-Driven Design: Tackling Complexity in the Heart of Software."*
  - Vernon, Vaughn. *"Implementing Domain-Driven Design."*
  - Vernon, Vaughn. *"Domain-Driven Design Distilled."*
- **Artigos**:
  - Effective Aggregate Design by Vaughn Vernon (disponível online) - Leitura essencial sobre design de agregados.

Parte 2: Construindo o Core - Domínio, Dados e Integração Confiável

Capítulo 5: Validação com Result: Sucesso e Falha sem Exceções

Capítulo 6: Persistência com Repositórios de Negócio usando Panache ORM

Capítulo 7: Mensageria: Produtor, Consumidor e Processador com Quarkus Reactive Messaging

CapÃtulo 8: CDC para Colocar Dados em Movimento: Quarkus com Debezium

Capítulo 9: Outbox Pattern com Eventos de Domínio usando Quarkus Outbox

Parte 3: Expondo e Entregando - APIs, Contêineres e Orquestração

Capítulo 10: Unindo Pontas Soltas: Agregando APIs com Quarkus GraphQL

Capítulo 11: Rodando a Solução em Containeres

Capítulo 12: Deploy da Solução no Kubernetes - Minikube