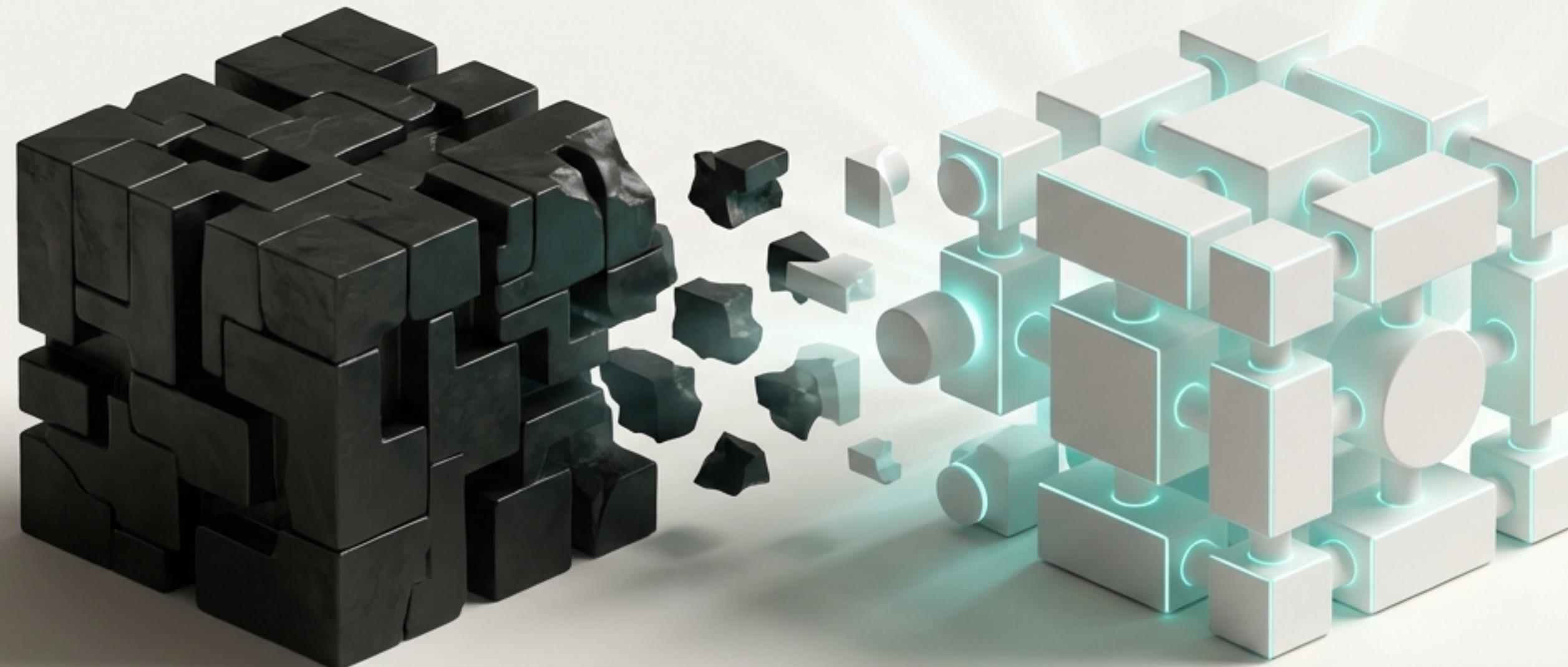


# **Do Monolito ao Modular: A Revolução dos Aspectos em CDS**

Um guia de melhores práticas para criar modelos de dados limpos, reutilizáveis e escaláveis com o SAP Cloud Application Programming Model.



# O Princípio Fundamental: Aumente a Modularidade, Separe as Preocupações

## Conceito Central

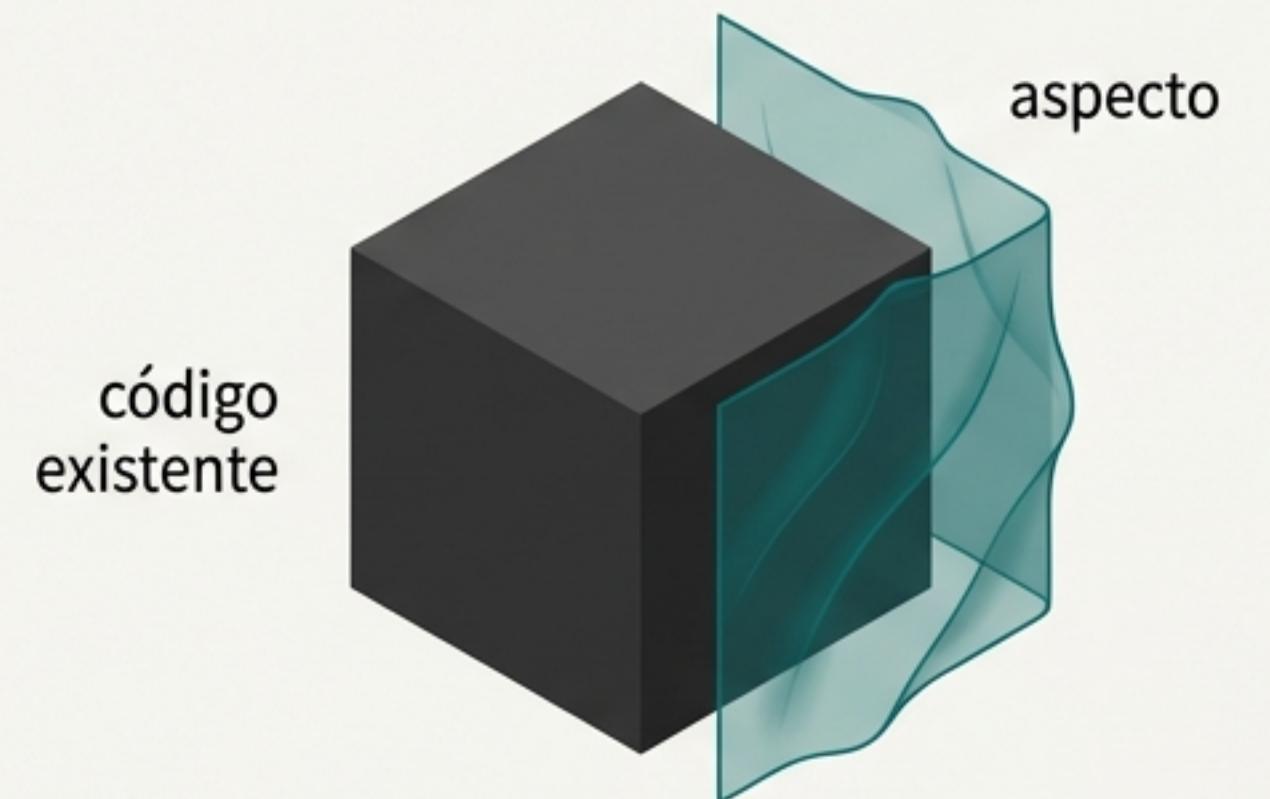
A Modelagem Orientada a Aspectos em CDS compartilha os mesmos objetivos da Programação Orientada a Aspectos (AOP).

## Aplicação em CDS

Em essência, os Aspectos CDS permitem que você espalhe arbitrariamente uma definição por diferentes locais e arquivos, até mesmo em projetos distintos com diferentes ciclos de vida. É a capacidade de **estender qualquer coisa, de qualquer lugar.**

- “A programação orientada a aspectos (AOP) é um paradigma de programação que visa aumentar a modularidade, permitindo a **separação de preocupações transversais**. Ela faz isso adicionando comportamento ao código existente... **sem modificar o código.**”

- Definição de AOP



# O Problema: Modelos 'Tudo-em-Um' que Poluem seu Domínio

## ✗ Abordagem Monolítica

É comum encontrar modelos de domínio poluídos com uma infinidade de anotações, especialmente para a interface do usuário (UI). Isso torna o modelo principal difícil de ler, entender e manter.

```
srv/cat-service.cds
```

```
service CatalogService {

    @UI.SelectionFields: [ ID, price, currency_code ]
    @UI.LineItem: [
        { Value: ID, Label: '{i1n>Title}' },
        { Value: author, Label : '{i1n>Author}' },
        { Value: genre.name},
        { Value: price},
        { Value: currency.symbol},
    ]
    @UI.HeaderInfo: {
        TypeName      : '{i1n>Book}',
        TypeNamePlural : '{i1n>Books}',
        Description   : { Value: author }
    }
    // ... more UI annotations ...

    entity Books { ... }
    ...
}
```

# A Solução: Mantenha seu Core Limpo e Isole as Preocupações



## \*\*Separação Clara de Preocupações\*\*

A melhor prática é manter seu modelo de domínio conciso e comprehensível, e fatorar as preocupações secundárias (como layout de UI) em arquivos separados usando a palavra-chave `annotate`.

### Core Model (srv/cat-service.cds)

```
service CatalogService {  
    entity Books {  
        ...  
    }  
    ...  
}
```

### UI Annotations (app/fiori-layout.cds)

```
using { CatalogService } from '../srv/cat-service';  
  
// Annotations for List Pages  
annotate CatalogService.Books with @UI:{  
    SelectionFields: [ ... ],  
    LineItem: [ ... ]  
}  
  
// Annotations for Object Pages  
annotate CatalogService.Books with @UI:{  
    HeaderInfo: { ... },  
    Facets: [ ... ]  
}
```

# Os Benefícios Imediatos da Separação de Preocupações



## 💡 Clareza Máxima

Seu modelo de domínio principal permanece puro e focado exclusivamente na lógica de negócios, facilitando a compreensão e a evolução.



## 🔧 Manutenção Simplificada

Atualizações na UI não exigem modificações no serviço de catálogo central. As responsabilidades são desacopladas, reduzindo o risco de efeitos colaterais.



## 🤝 Colaboração Paralela

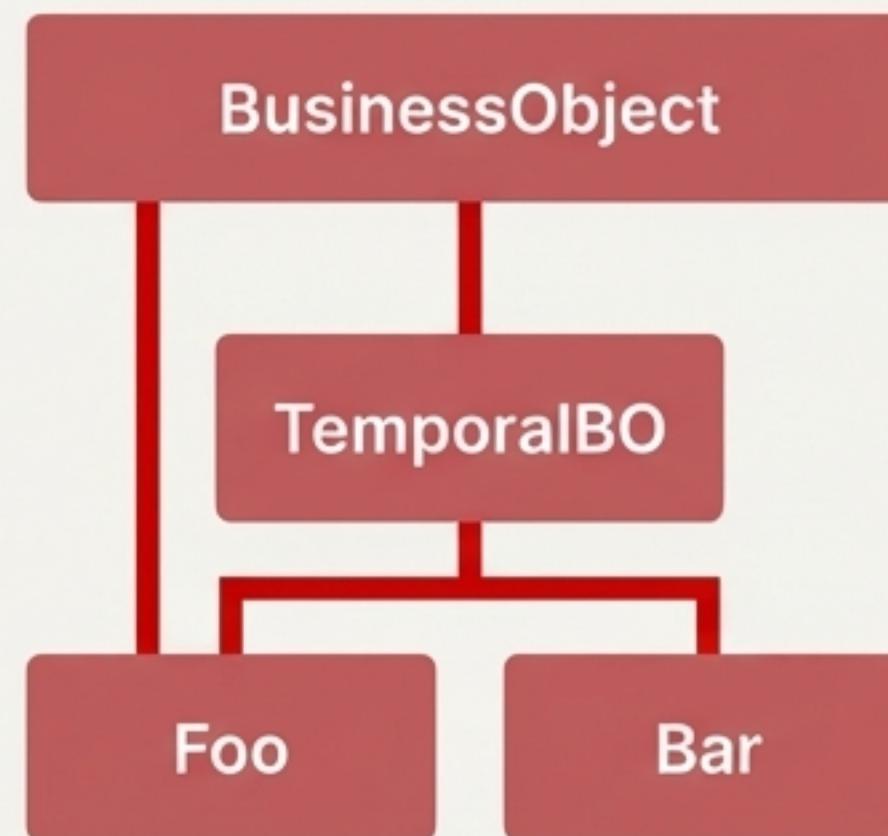
Equipes de frontend e backend podem trabalhar em seus respectivos arquivos de forma independente e simultânea, acelerando o desenvolvimento.

# O Desafio do Reuso: A Rigidez da Herança Única

A maneira clássica de reutilizar funcionalidades é criar hierarquias de herança com classes base. No entanto, a limitação de herança única muitas vezes força a combinação de múltiplos aspectos independentes em uma única definição, que os consumidores são obrigados a aceitar por completo.

✗ Evite

A Abordagem  
“Max Base Class”



```
// A single base class combining multiple concerns
abstract entity BusinessObject {
    key ID      : UUID;
    createdAt : DateTime;
    createdBy : User;
    modifiedAt : DateTime;
    modifiedBy : User;
    changes   : Composition of many Changes;
    extensions : PredefinedExtensionFields;
}

// Extending the base class
abstract entity TemporalBO : BusinessObject {
    validFrom : Date @cds.valid.from;
    validTo   : Date @cds.valid.to;
}

// Consumers have limited choices
entity Foo : BusinessObject {...} // Gets everything
entity Bar : TemporalBO {...}    // Gets even more
```

## Aspectos de Reuso Separados

# A Nova Abordagem: Componha Modelos com Aspectos Independentes

Com CDS, podemos fazer melhor. Em vez de herança, usamos a composição de múltiplos aspectos. Cada aspecto define uma única preocupação, permitindo que os consumidores escolham e combinem exatamente o que precisam.

### cuid

```
aspect cuid { key ID : UUID; }
```

### managed

```
aspect managed {  
    createdAt : DateTime;  
    createdBy : User;  
    modifiedAt : DateTime;  
    modifiedBy : User;  
}
```

### tracked

```
aspect tracked {  
    changes : Composition of many Changes;  
}
```

### temporal

```
aspect temporal {  
    validFrom : Date @cds.valid.from;  
    validTo   : Date @cds.valid.to;  
}
```

Esta abordagem permite a **propriedade distribuída** dos aspectos. Eles não dependem uns dos outros e podem ser mantidos por equipes diferentes.

# Flexibilidade em Ação: Misture e Combine Aspectos Conforme a Necessidade

Os consumidores agora podem compor suas entidades de forma declarativa, incluindo apenas os aspectos relevantes para cada caso de uso.

```
using { cuid, managed, tracked, extensible, temporal } from 'your-reuse-aspects';
// Foo precisa de rastreamento e extensibilidade
entity Foo : cuid, managed, tracked, extensible {
    // ... campos específicos de Foo
}
```



**Reuso Inteligente:** Fim do 'tudo ou nada'. Use apenas o que você precisa.

```
} from 'your-reuse-aspects';
// Bar é uma entidade temporal, mas não precisa
// de rastreamento
entity Bar : cuid, managed, temporal {
    // ... campos específicos de Bar
}
```



**Máxima Flexibilidade:** Crie qualquer combinação de funcionalidades sem estar preso a uma hierarquia de classes.

A sintaxe com ':' parece herança múltipla, mas na verdade é baseada em **mixins**. Isso é mais poderoso e evita problemas clássicos como o 'problema do diamante'.

# Adaptação Não-Intrusiva: Estenda e Personalize Definições Reutilizadas

**Cenário:** Imagine que você está usando um pacote de reuso (`some-reuse-package`), mas precisa de pequenas adaptações: um campo extra, um tipo de dado mais longo, ou até mesmo uma nova associação.

## A Solução

A palavra-chave `extend` permite modificar qualquer definição ‘de fora’, sem alterar o código original.

### Antes: Código do Pacote de Reuso (`some-reuse-package/index.cds`)

```
type CodeList : {  
    name : localized String;  
}  
  
entity Currencies : CodeList { key code : String(3); }  
entity Languages : CodeList { key locale : String(5); }
```

### Depois: Suas Adaptações (`db/common.cds`)

```
using { CodeList, Currencies, Languages } from 'some-reuse-package'  
  
// Adicionando um novo campo a um tipo  
extend CodeList with { descr: localized String }  
  
// Adicionando um campo a uma entidade específica  
extend Currencies with { symbol: String(2) }  
  
// Adaptando um campo existente  
extend Languages:locale with { length:15 };
```

# Leve a Adaptação ao Próximo Nível: Relacionamentos e Aspectos

A capacidade de extensão vai além de campos simples. Você pode adicionar **associações** e **composições**, ou até mesmo aplicar um **aspecto** inteiro a uma entidade que você não controla.

## Adicionando Relacionamentos

Adicionar um histórico de mudanças a todas as entidades que usam o aspecto `managed` de @sap/cds/common`.

```
using { User, managed } from '@sap/cds/common';

extend managed with {
  ChangeNotes : Composition of many {
    key timestamp : DateTime;
    author      : User;
    note        : String(1000);
  }
}
```

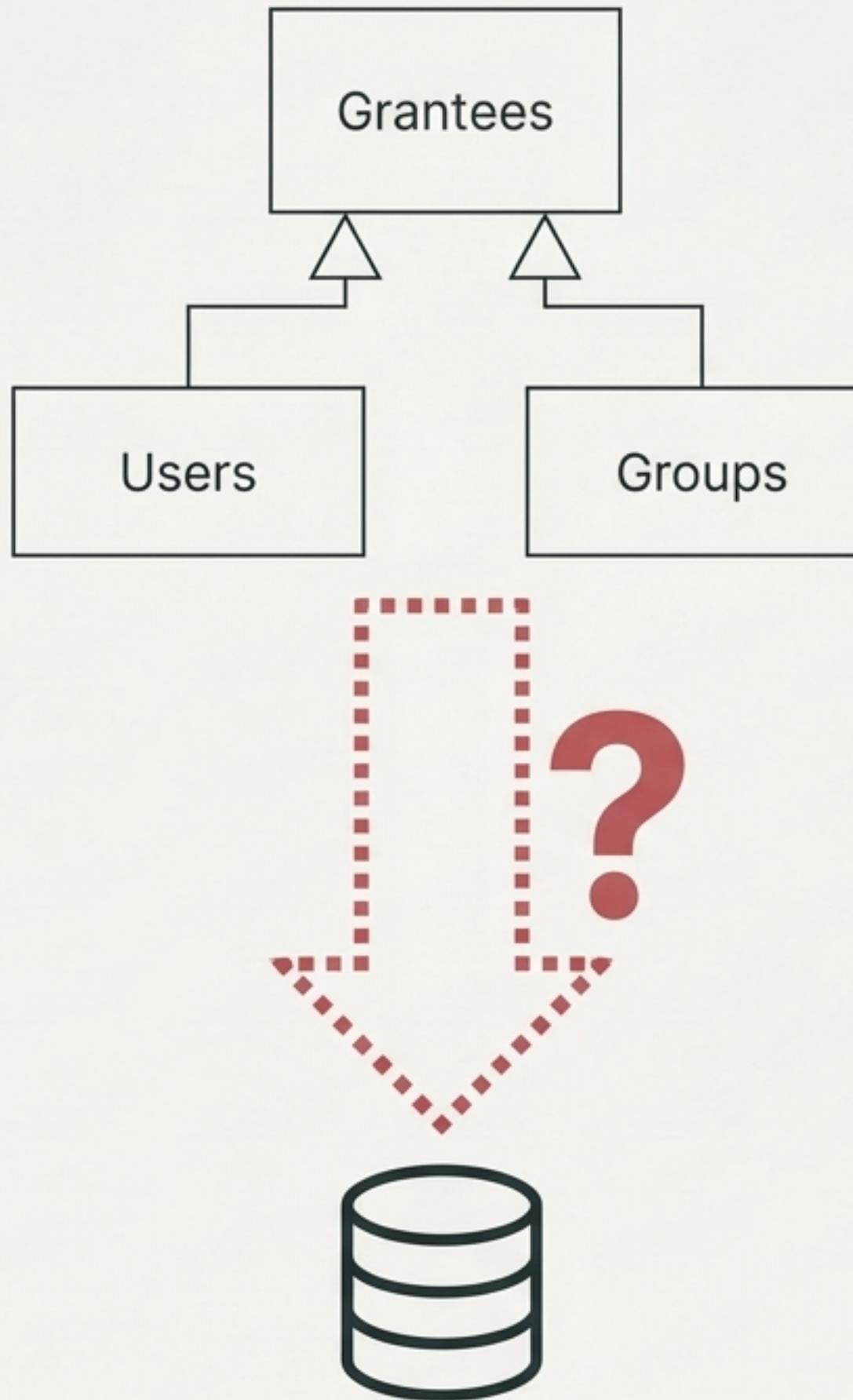
## Adicionando Aspectos de Reuso

Fazer com que uma entidade de um pacote reutilizado adote o comportamento do aspecto `managed`.

```
using { SomeEntity } from 'some-reuse-package';
using { managed } from '@sap/cds/common';

extend SomeEntity with managed;
```

Essas mesmas técnicas são a base para **Customização** e **Verticalização** em aplicações SaaS, permitindo que clientes adicionem campos (carrier : managees : `Releamer : Association to Carriers;`) ou **alteem** alterem anotações (@title: 'Patients'`).



# O Desafio Final: Mapeando Hierarquias de Herança para Tabelas

## O Cenário Tentador

Como desenvolvedores, muitas vezes somos tentados a modelar hierarquias como faríamos em linguagens orientadas a objetos.

```

// Uma "superclasse" abstrata ou aspecto
abstract entity Grantees {
    key name : String;
}
// "Subclasses" que herdam dela
entity Users : Grantees {
    group : Association to Groups;
}
entity Groups : Grantees {
    members : Composition of many Users on members.group = $self;
}

```

## O Problema Fundamental

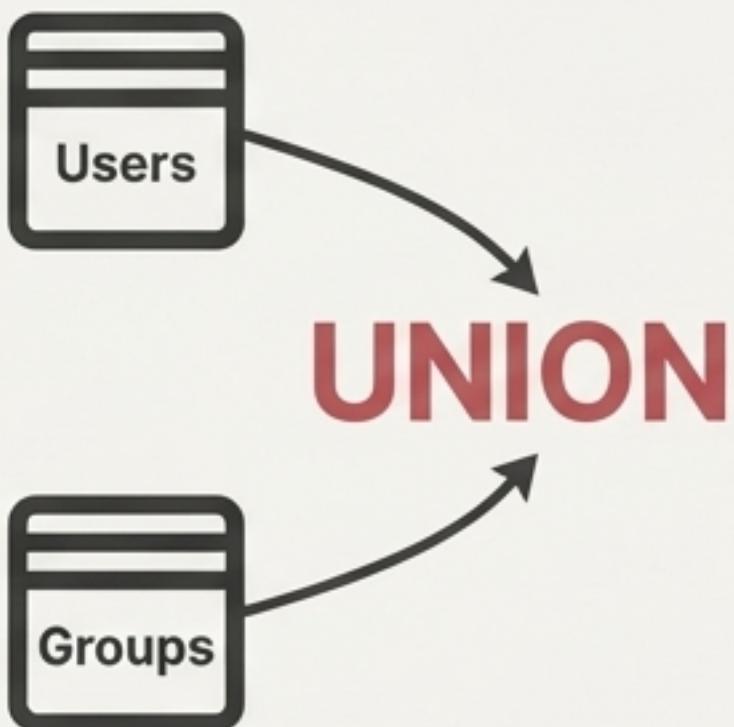
O CDS não mapeia automaticamente essas hierarquias para o banco de dados. Você deve escolher explicitamente uma estratégia, e as escolhas mais óbvias podem levar a grandes problemas de performance.

# Estratégias de Mapeamento a Serem Evitadas

## ✗ Tabela por Classe Folha (Table Per Leaf Class)

### Como Funciona

Cria uma tabela separada para 'Users' e outra para 'Groups'.



### O Problema

Consultar uma lista heterogênea de 'Grantees' (usuários e grupos juntos) requer 'UNION's caros, que prejudicam a performance.

```
entity UsersAndGroups as (
    SELECT from Users
) UNION ALL (
    SELECT from Groups
);
```

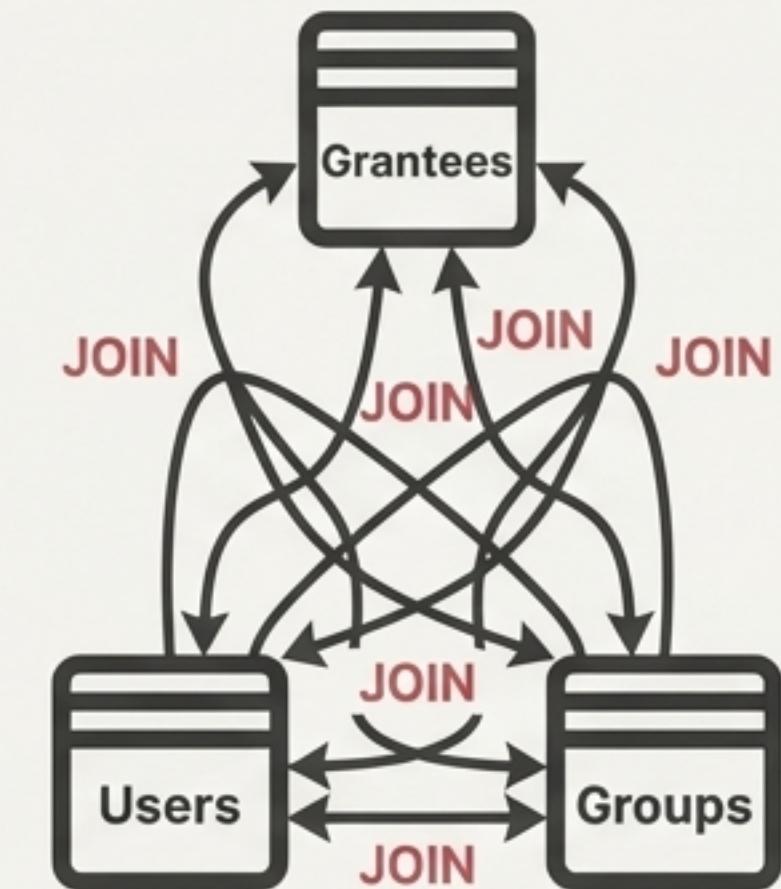
## ✗ Tabela por Classe (Table Per Class)

### Como Funciona

Cria uma tabela para 'Grantees', uma para 'Users' e uma para 'Groups', ligadas por associações.

### O Problema

Evita 'UNION's, mas requer múltiplos 'JOIN's para reconstruir um objeto completo, o que se torna complexo e lento em exemplos do mundo real.



```
entity Grantees { key name: String; }
entity Users { header: Association to Grantees; ... }
entity Groups { header: Association to Grantees; ... }
```

# ✓ Single Table Strategy

## A Solução Elegante e Performática: Estratégia de Tabela Única

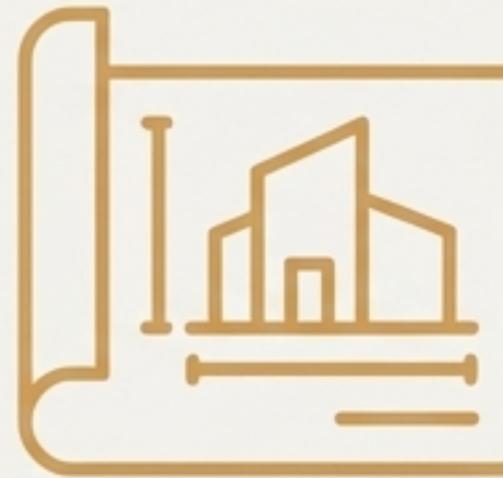
A terceira e melhor estratégia é colocar tudo em uma única tabela, usando um elemento adicional (um “**discriminador**”) para diferenciar os tipos de entidade.

Users			
...	...	kind	...
...	...	kind	...

```
// Tudo é modelado como uma única entidade 'Users'  
entity Users {  
    key name : String;  
  
    // O discriminador que diferencia os tipos  
    kind : String enum { user; group };  
  
    // Associações autorreferenciadas  
    group : Association to Users;  
    members : Composition of many Users on members.group = $self;  
}
```

# Por que a Estratégia de Tabela Única Vence

Esta abordagem alinha o modelo conceitual com um design de banco de dados físico otimizado, resultando em ganhos significativos.



## Performance Otimizada

Elimina a necessidade de `UNION`'s caros e `JOIN`'s excessivos. As consultas para listas heterogêneas são simples e diretas.

## Modelo Simplificado

A definição da entidade é mais limpa e fácil de entender, refletindo diretamente a estrutura da tabela subjacente.

## Consultas Diretas

Filtrar por tipo (`where kind = 'user'`) ou buscar todos os grantees é trivial e eficiente, sem a complexidade de unir múltiplas fontes.

# Do Monolito ao Modular: Seus Princípios Orientadores



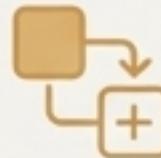
## 1. Isole Preocupações

Mantenha seu modelo de domínio puro. Use `annotate` para fatorar preocupações secundárias como UI, expondo-as em arquivos separados.



## 2. Componha, Não Apenas Herde

Prefira a composição de múltiplos aspectos pequenos e reutilizáveis em vez de classes base monolíticas e rígidas.



## 3. Estenda Sem Modificar

Use `extend` para adaptar tipos e entidades de forma não-intrusiva, mesmo que venham de bibliotecas externas.



## 4. Pense em Tabela Única

Para hierarquias, adote a estratégia de tabela única com um discriminador para obter máxima performance e simplicidade no modelo.

**A Modelagem Orientada a Aspectos não é apenas uma técnica; é a base para construir modelos CDS robustos, escaláveis e elegantes.**