Engenharia de Softwares Escaláveis

Design Patterns e Domain-Driven Design com Java

Agenda

Etapa 7: Projetar Softwares Usando Aggregates.

- Agregados.
- Ciclo de Vida de um Objeto de Domínio.
- Projeto de Aggregates.

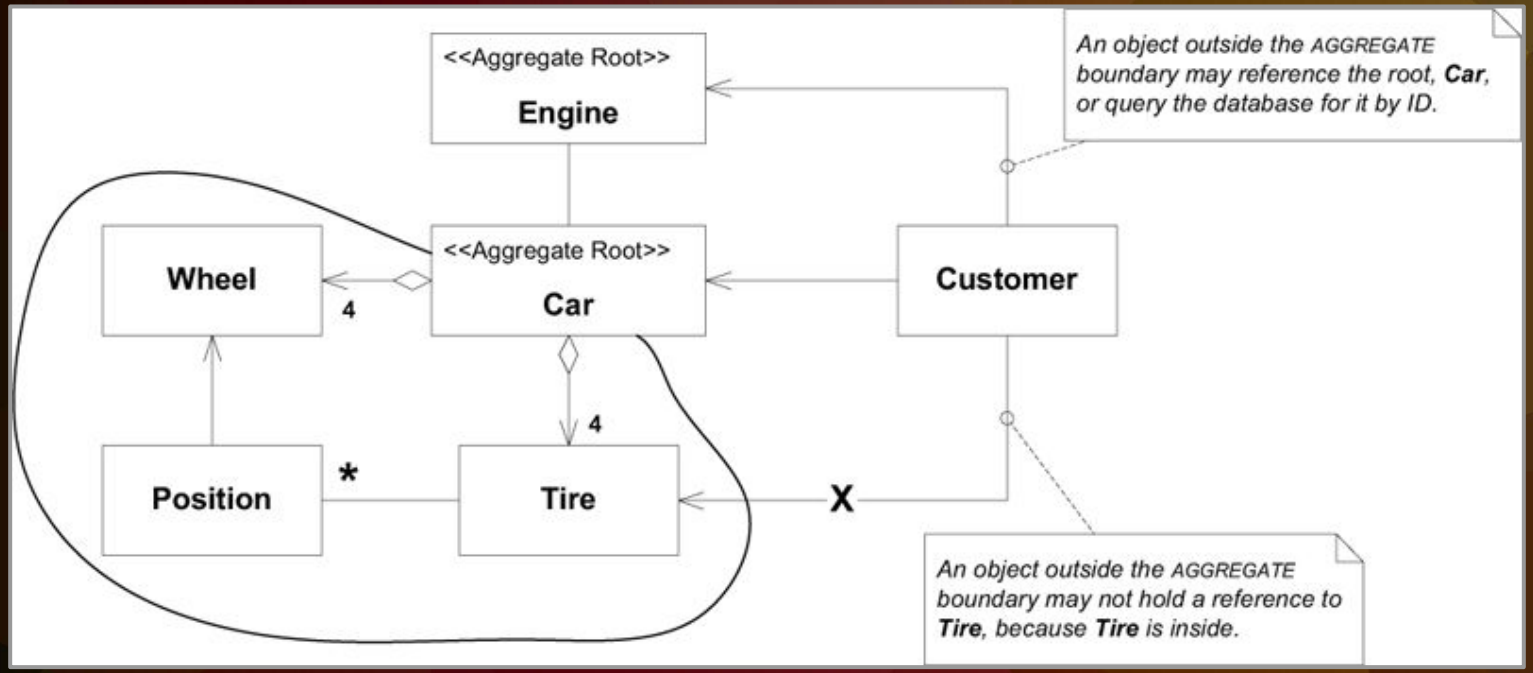
Agregados

Um **Aggregate** é um cluster de objetos associados que tratamos como uma unidade para fins de alterações de dados.

Cada **Aggregate** possui uma raiz e um limite.

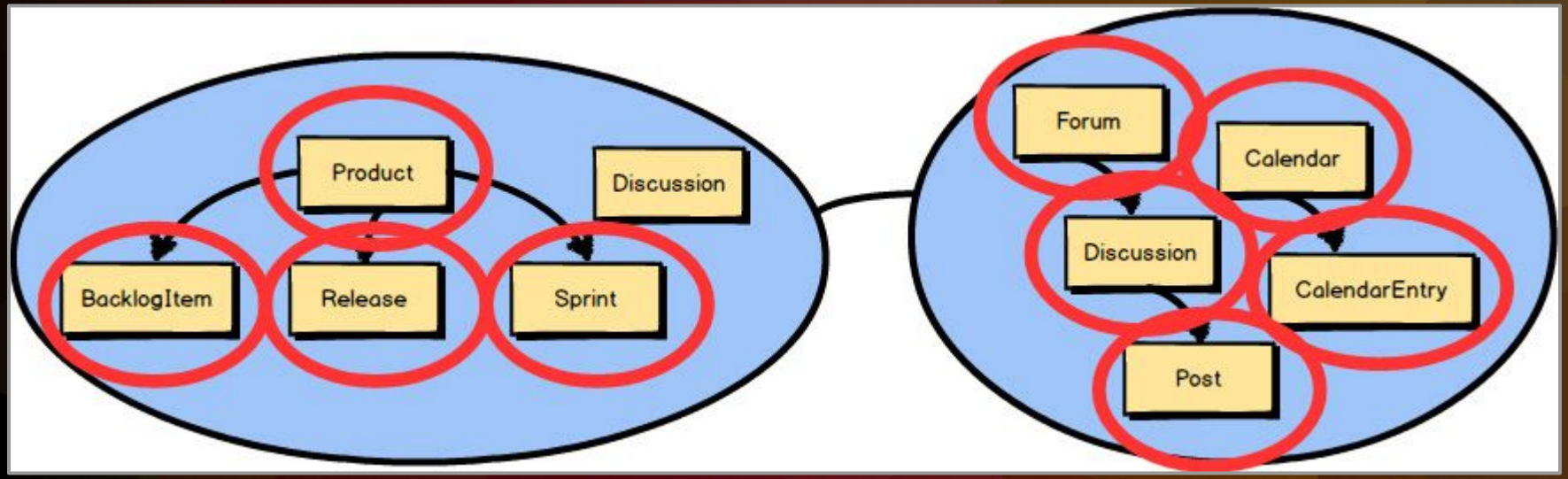
O limite define o que está dentro do **Aggregate**.

A raiz é uma **Entidade** única e específica contida no **Aggregate**.

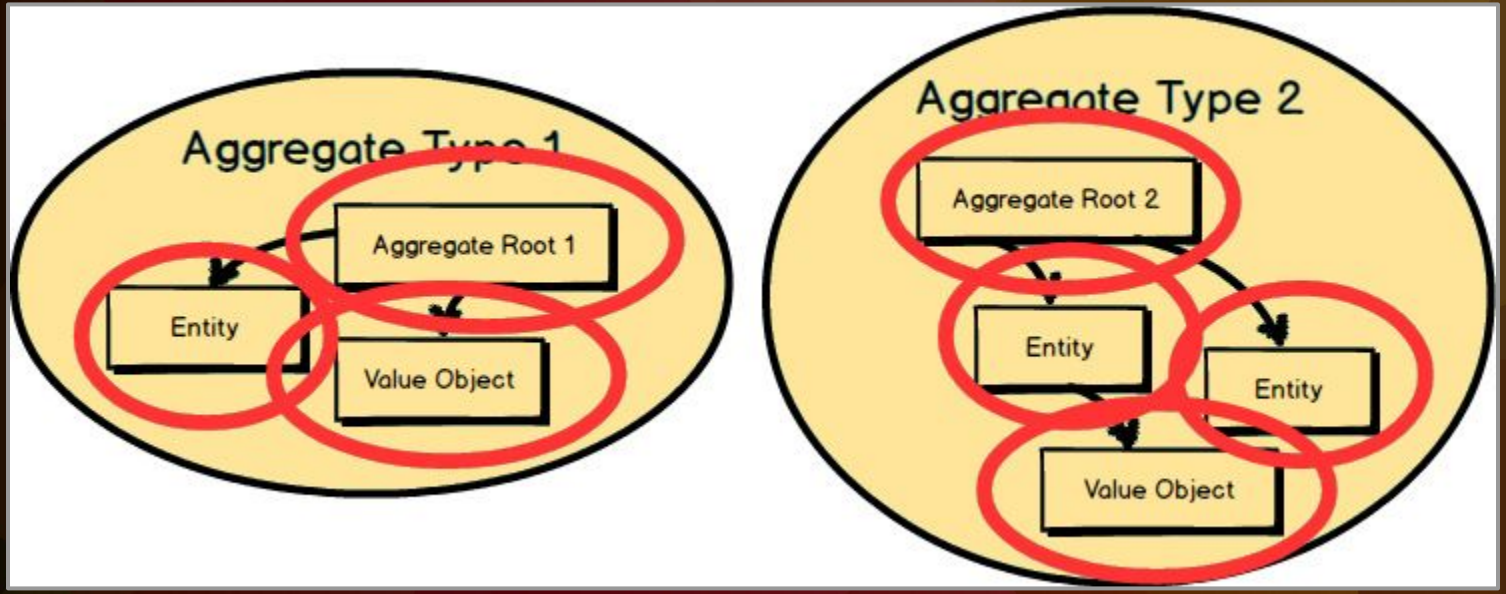


O carro é a **entidade** raiz do **agregado** cujo limite também envolve os pneus.

O bloco do motor têm números de série gravados e às vezes são rastreados independentemente do carro.



Cada um dos conceitos circulados que você vê dentro desses dois contextos limitados é um agregado e representam **entidades**.



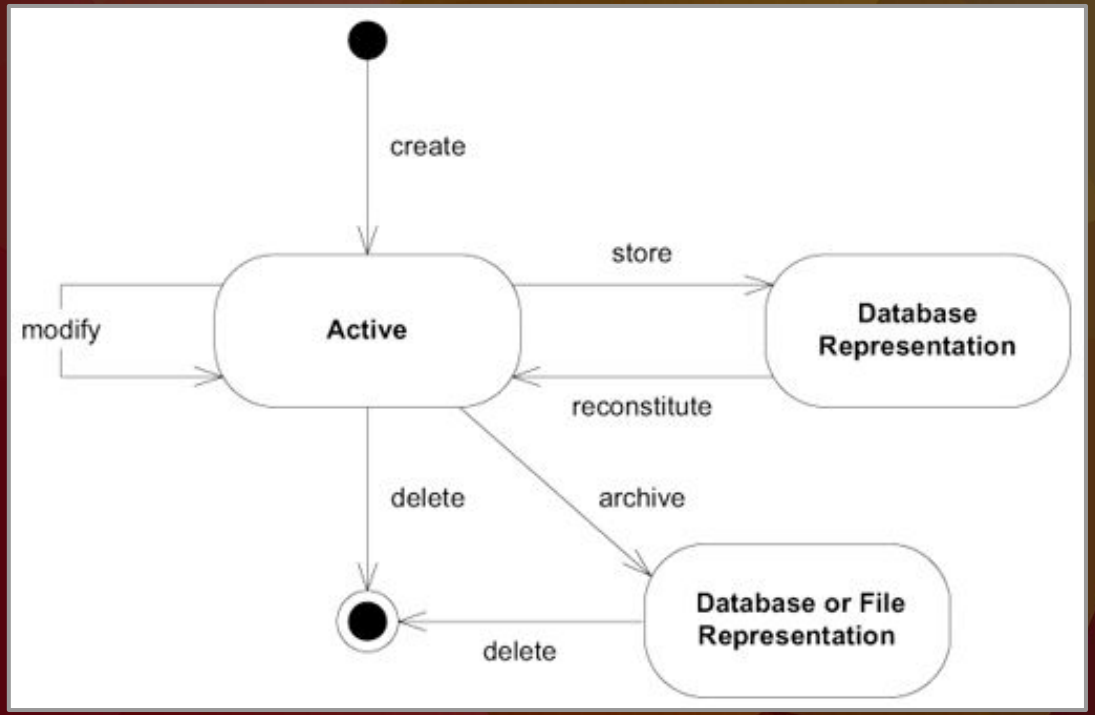
Cada **agregado** é composto por **entidades** e **objetos de valor**, onde uma **entidade** é chamada de **Raiz do Agregado**.

Ciclo de Vida de um Objeto de Domínio

Ciclo de Vida de um Objeto de Domínio

Os desafios se enquadram em duas categorias.

1. Manter a integridade durante todo o ciclo de vida.
2. Evitar que o modelo seja inundado pela complexidade do gerenciamento do ciclo de vida.



Mantendo a Integridade

Aggregate Root é o guardião das invariantes, pois todas as modificações no **Aggregate** deve passar por ele.

1. `Pedido.adicionarItem(produto, quantidade)` garante que quantidade seja `> 0` antes de criar um `ItemPedido`.

O estado do **Aggregate** centraliza as validações, evitando inconsistências internas e regras espalhadas em todos os lugares.

Reduzindo a Complexidade

Aggregate define uma fronteira clara de consistência onde as regras de negócio são aplicadas em transações.

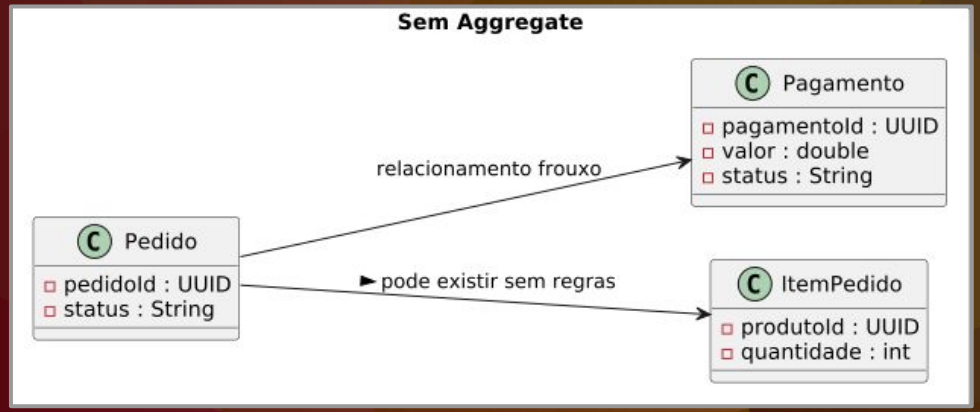
2 Fora dele, interações com outros **Aggregates** podem usar eventos de domínio e consistência eventual.

Isso evita que o modelo seja inundado com dependências cruzadas e regras espalhadas.

Cada **Aggregate** pode evoluir de forma independente.

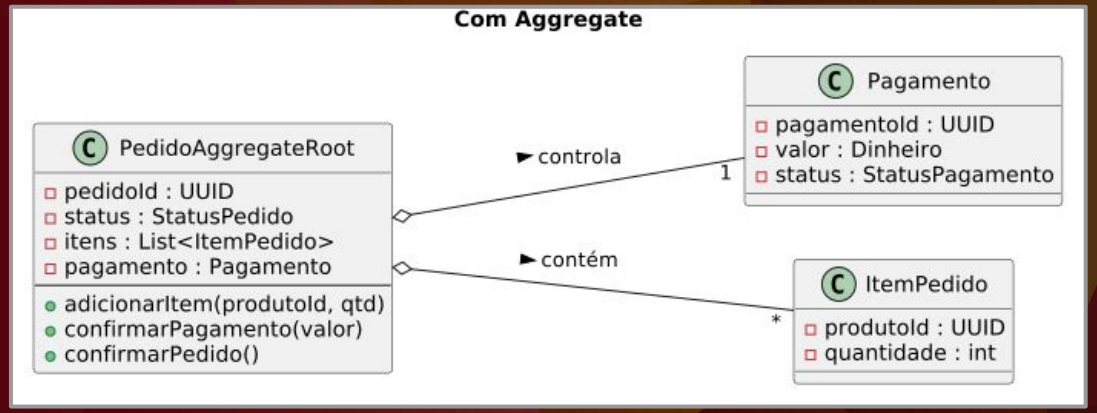
`ItemPedido` e `Pagamento` poderiam ser manipulados separadamente.

Risco: criar `Pedido` sem Itens, duplicar `Pagamento`, confirmar `Entrega` sem `Pedido`.

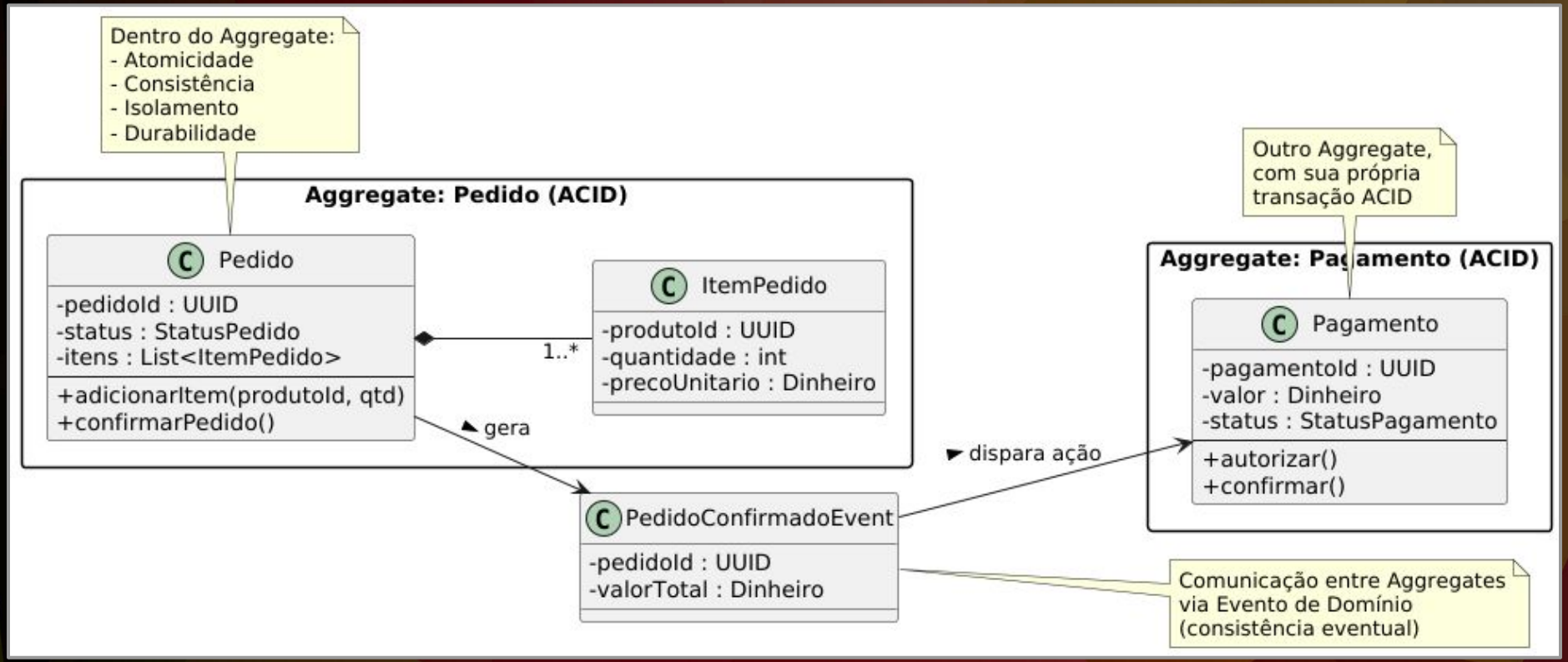


`Pedido` controla a consistência:

- Só permite adicionar `ItemPedido` válido.
- Só permite confirmar o `Pedido` se houver `Pagamento`.
- Gera um evento de domínio `PedidoConfirmado` para notificar outros contextos, como Entrega e Faturamento.



ACID	Relação com Aggregates no DDD
Atomicidade	**Aggregate** é a **unidade atômica de modificação**. Todas as mudanças em suas **entidades** e **value objects** devem ser aplicadas juntas em uma única transação.
Consistência	**Aggregate Root** garante as **invariantes de negócio** (ex.: Pedido não pode ter Item com quantidade ≤ 0). Assim, qualquer transação que termina deixa o **Aggregate** consistente.
Isolamento	É recomendável **manter Aggregates pequenos**, de modo que as transações concorrentes tenham menos chance de colisão. Dois **Aggregates** diferentes podem ser atualizados em paralelo.
Durabilidade	A persistência normalmente é feita via **repositórios** que salvam o estado do **Aggregate Root**. Uma vez commitado, o estado é durável.


Dentro do Aggregate:
- Atomicidade
- Consistência
- Isolamento
- Durabilidade
Aggregate: Pedido (ACID)
Pedido
-pedidoId : UUID
-status : StatusPedido
-itens : List<ItemPedido>
+adicionarItem(produtoId, qtd)
+confirmarPedido()
ItemPedido
-produtoId : UUID
-quantidade : int
-precoUnitario : Dinheiro
1..*
gera
PedidoConfirmadoEvent
-pedidoId : UUID
-valorTotal : Dinheiro
dispara ação
Outro Aggregate,
com sua própria
transação ACID
Aggregate: Pagamento (ACID)
Pagamento
-pagamentoId : UUID
-valor : Dinheiro
-status : StatusPagamento
+autorizar()
+confirmar()
Comunicação entre Aggregates
via Evento de Domínio
(consistência eventual)

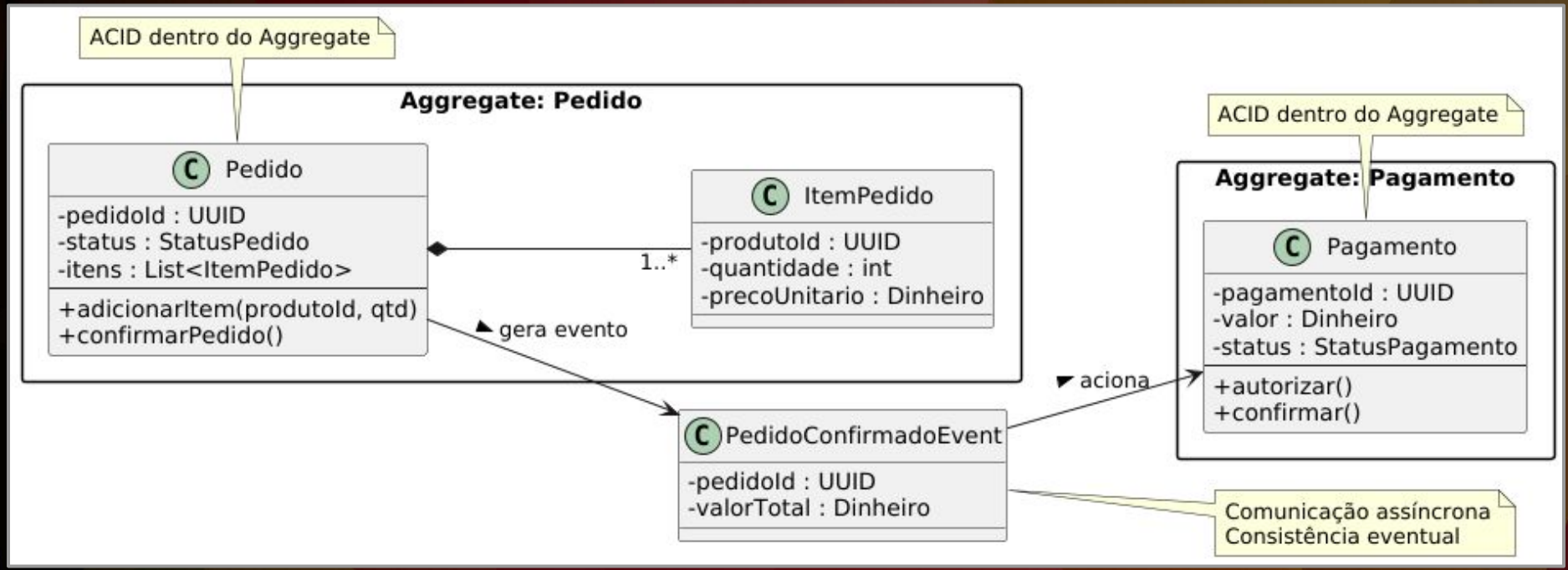
Aspecto	Dentro de um Aggregate (ACID)	Entre Aggregates (Consistência Eventual)
Unidade de transação	**Aggregate** é a **fronteira atômica**. Todas as mudanças em suas entidades e VOs ocorrem juntas.	Cada **Aggregate** tem sua própria transação independente.
Atomicidade	Ou todas as mudanças no **Aggregate** são aplicadas, ou nenhuma.	Não é garantida entre **Aggregates**, pois pode haver falhas parciais.
Consistência	Invariantes sempre mantidas pelo **Aggregate Root**.	Regras de negócio entre **Aggregates** podem ficar **temporariamente inconsistentes** até o processamento dos eventos.
Isolamento	Conflitos de concorrência são gerenciados dentro do **Aggregate**.	Concorrência entre **Aggregates** é **independente**, reduzindo contenção.
Durabilidade	Mudanças persistem após o commit do **Aggregate**.	Cada **Aggregate** persiste sua parte. Sincronização acontece via eventos.
Comunicação	Métodos e invariantes dentro do próprio **Aggregate**.	**Eventos de domínio** ou mensagens assíncronas publicadas.
Exemplo no E-Commerce	Adicionar `ItemPedido` a um `Pedido`: tudo ou nada na mesma transação.	`PedidoConfirmadoEvent` dispara processo no `Pagamento`, que pode ocorrer em outro momento.

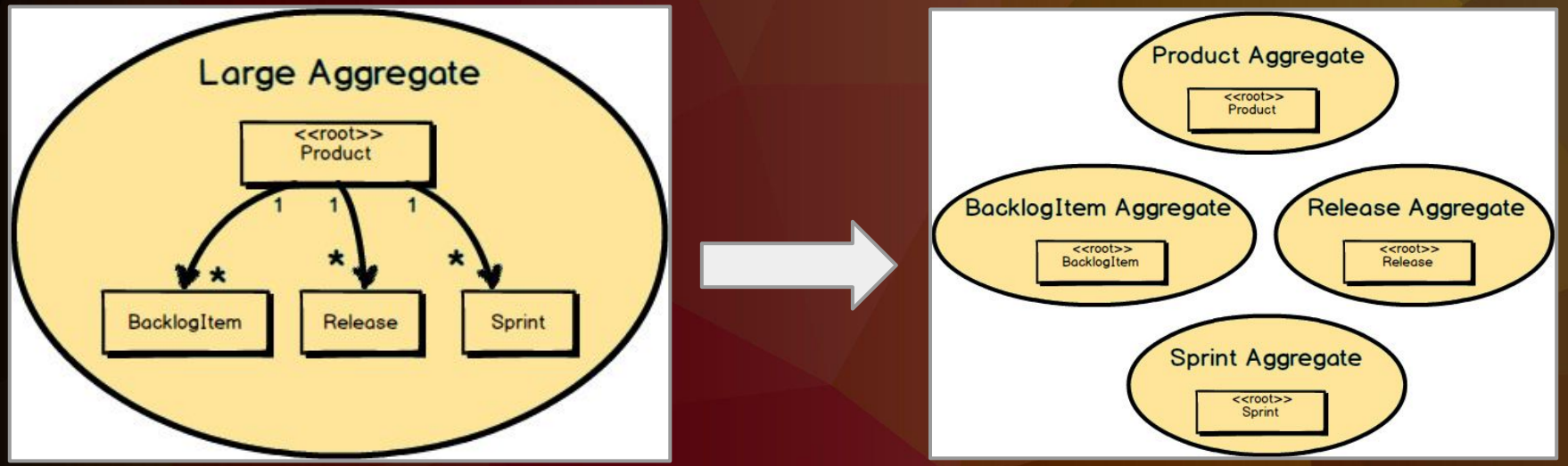
Projeto de Aggregates

Projeto de Aggregates

1. Cada **Aggregate** tem um **Aggregate Root**
2. Toda referência externa deve ser feita apenas para o **Aggregate Root**
3. **Aggregate Root** garante as invariantes de negócio
4. Transações devem ser consistentes apenas dentro do **Aggregate**

1 Cada **Aggregate** tem um **Aggregate Root**



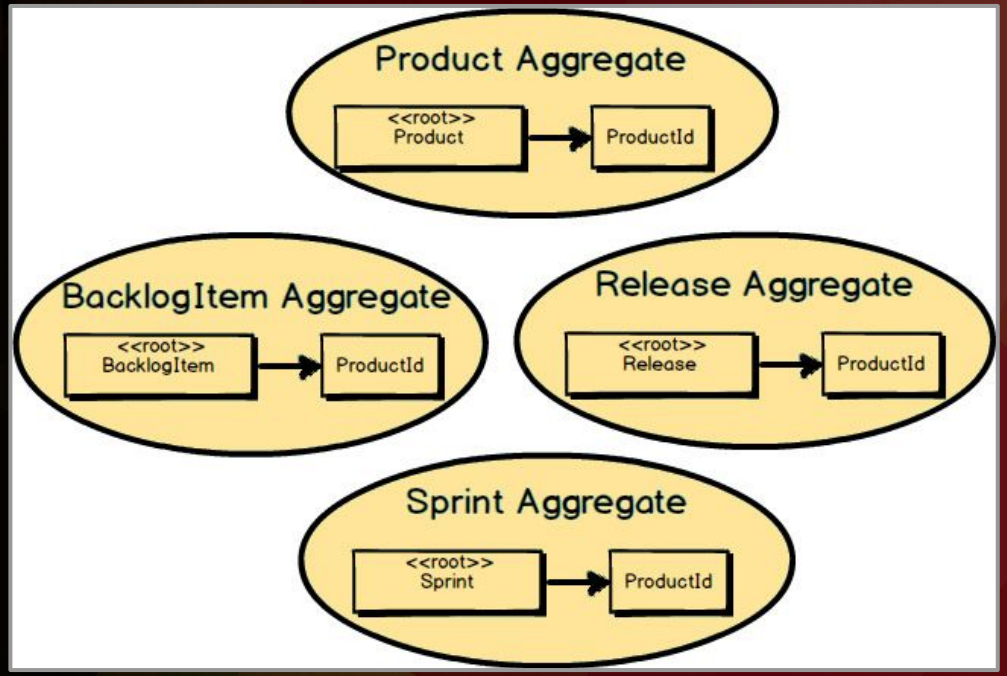


Dica: Projete Pequenos Agregados.

Se dividirmos o **Product** para formar quatro **agregados** separados, isso é o que obteremos: um **Product**, um **BacklogItem**, um **Release** e **Sprint**.

Eles carregam mais rapidamente, ocupam menos memória e são mais rápidos para coletar o lixo.

2 Toda referência externa deve ser feita apenas para o **Aggregate Root**

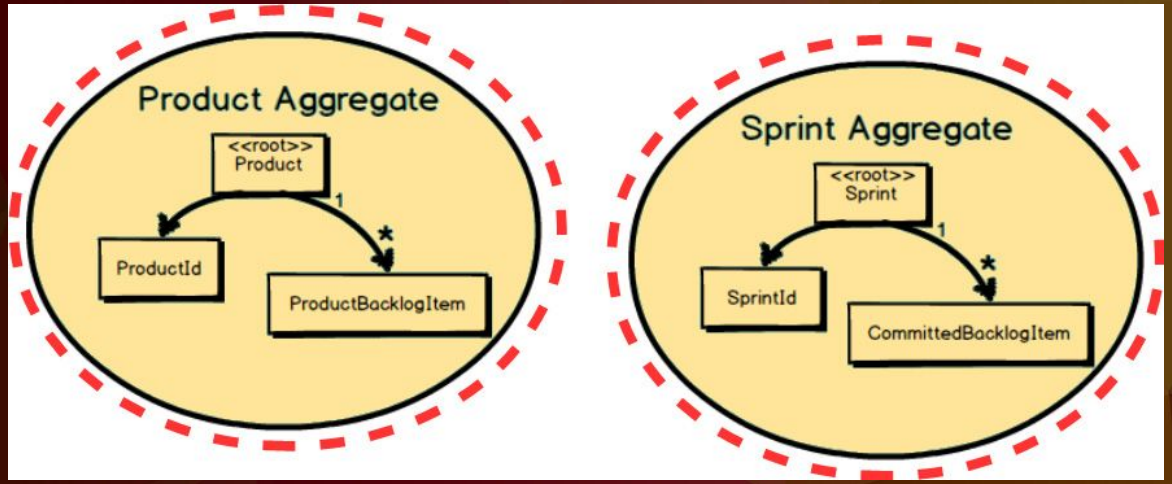


Isso ajuda a manter os **agregados** pequenos e evita a modificação de vários **agregados** na mesma transação.

Isso ajuda ainda mais a manter o design **agregado** pequeno e eficiente, reduzindo os requisitos de memória e carregando mais rapidamente de um armazenamento de persistência.

Também ajuda a impor a regra de não modificar outras instâncias do **agregado** na mesma transação.

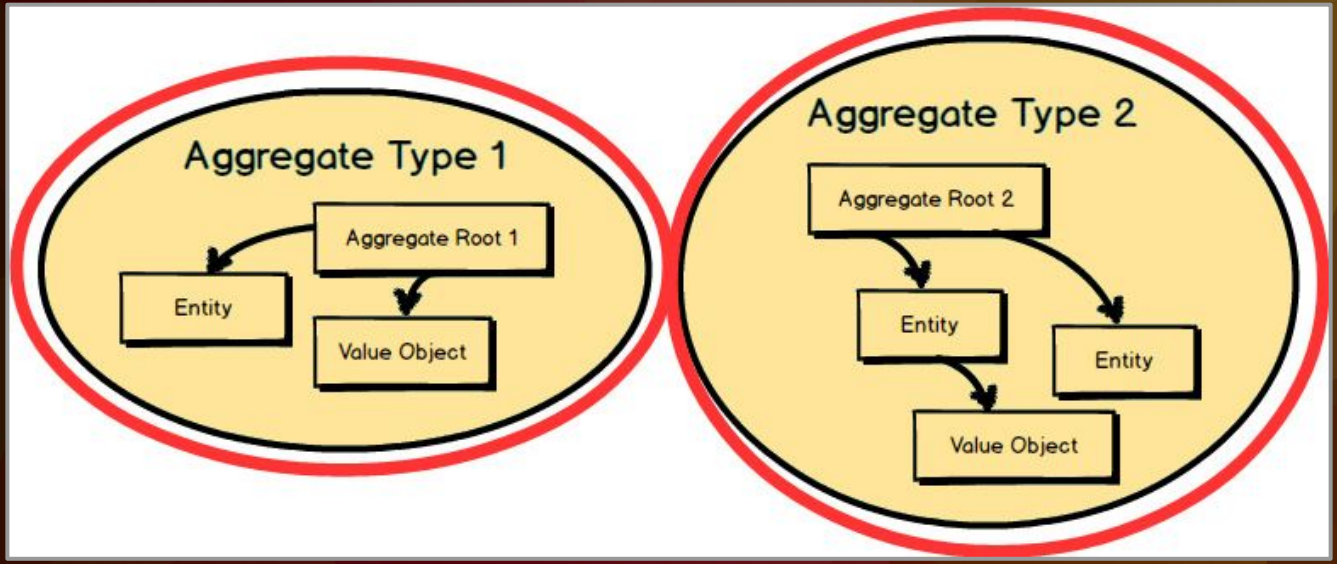
3 **Aggregate Root** garante as invariantes de negócio



Product é projetado de tal forma que, no final de uma transação, todos os **ProductBacklogItems** devem ser contabilizados e consistentes com o **Product** raiz.

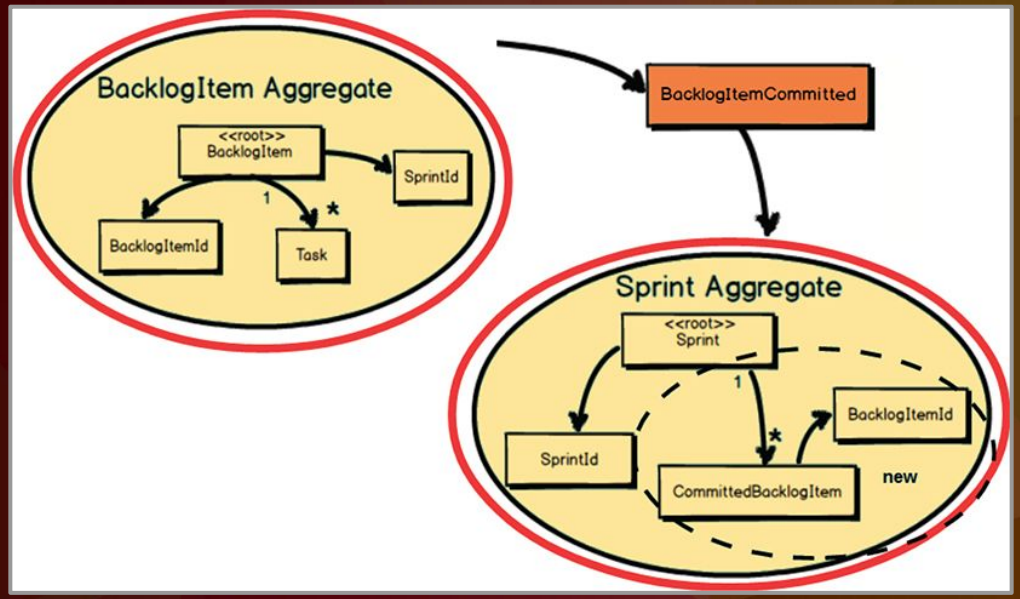
Sprint foi projetado de tal forma que, no final de uma transação, todos os **CommittedBacklogItem** compostos devem ser contabilizados e consistentes com o **Sprint** raiz.

4 Transações devem ser consistentes apenas dentro do **Aggregate**



Cada **agregado** forma um limite de consistência transacional.

Isso significa que dentro de um único **agregado**, todas as partes compostas devem ser consistentes, de acordo com as regras de negócio, quando a transação de controle for confirmada no banco de dados.



Dica: Use Consistência Eventual Entre Aggregates.

Como parte da transação do **BacklogItem**, ele publica um evento de domínio denominado **BacklogItemCommitted**.

A transação do **BacklogItem** é concluída e seu estado persiste junto com o **BacklogItemCommitted**.

Quando o **BacklogItemCommitted** chega a um assinante local, uma transação é iniciada e o estado do **Sprint** é modificado para manter o **BacklogItemId**.