

Engenharia de Softwares Escaláveis

Domain-Driven Design (DDD) e Arquitetura de
Softwares Escaláveis com Java

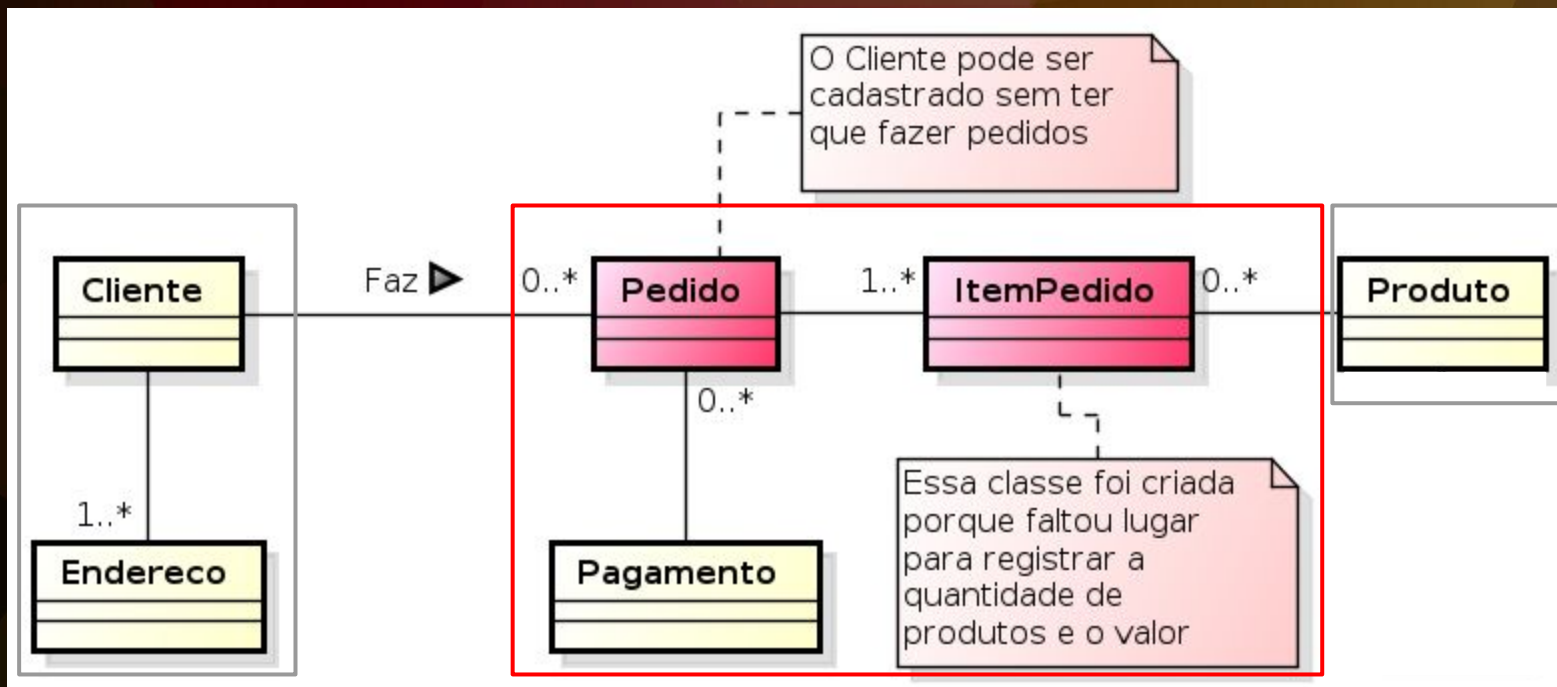
Agenda

Etapas 3: Implementação e Gerenciamento de Domain Events.

- Construção de Aggregates.
- Contextos Transacionais.
- Revisando o Projeto de Agregados.
- Eventos de Domínio - Parte 1.



Construção de Aggregates



1

The image shows an IDE interface with two main panels. The left panel displays a project tree for 'DR4_E2_A1'. The right panel shows the source code of 'Pedido.java'.

Project Tree (Left Panel):

- DR4_E2_A1
 - Source Packages
 - br.edu.infnet
 - br.edu.infnet.pedido.domain
 - ItemPedido.java
 - Pedido.java**
 - br.edu.infnet.pessoas.domain
 - Email.java
 - Pessoa.java
 - Telefone.java
 - br.edu.infnet.pessoas.infra
 - br.edu.infnet.produtos.domain
 - Produto.java
 - Test Packages
 - Other Sources
 - Dependencies
 - Runtime Dependencies
 - Test Dependencies
 - Java Dependencies
 - Project Files

Source Code (Right Panel):

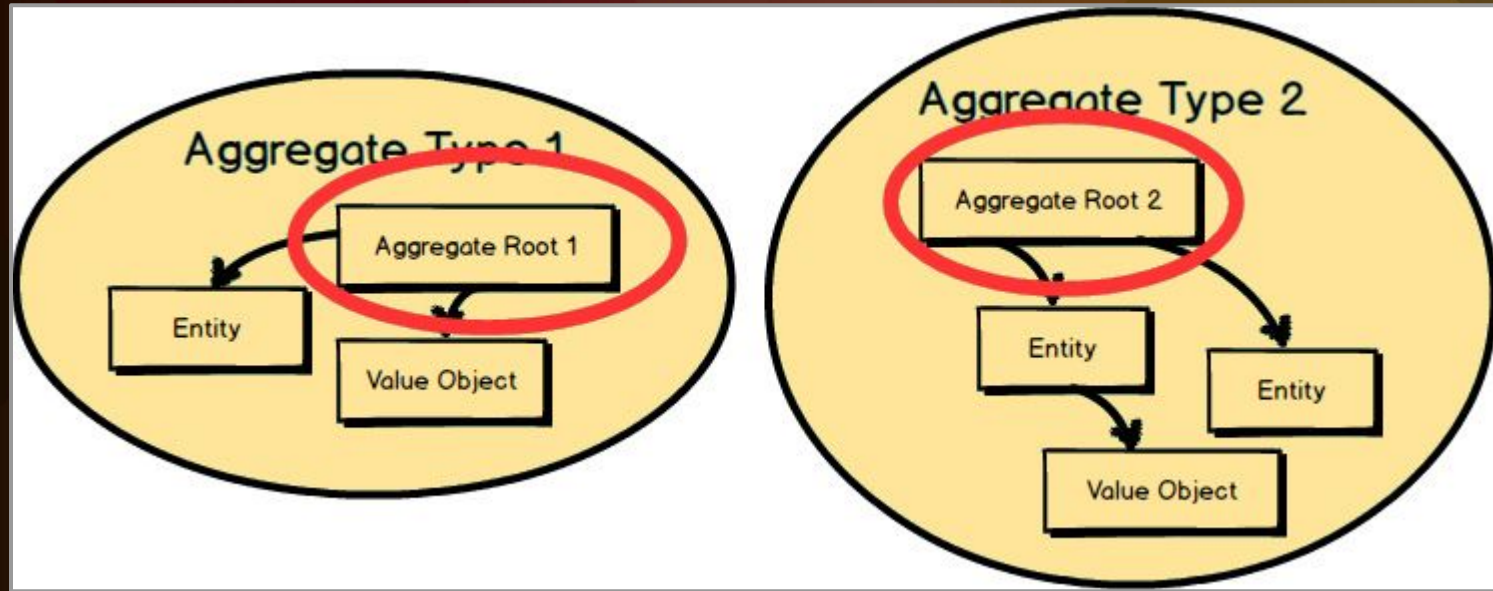
```
1 package br.edu.infnet.pedido.domain;
2
3 import ...8 lines
11
12 @Entity
13 @Table(name = "pedidos")
14 public class Pedido {
15
16     @Id
17     @GeneratedValue(strategy = GenerationType.IDENTITY)
18     private long id;
19
20     private Date data;
21     private String status;
22     private double total;
23     private List<ItemPedido> itensPedido;
24
25     public void adicionarItem(Produto produto, int quantidade) {
26     }
27
28     public void fazerPedido() {
29     }
30
31     public void cancelarPedido() {
32     }
33
34     public void enviarPedido() {
35     }
36 }
```

The methods `adicionarItem`, `fazerPedido`, `cancelarPedido`, and `enviarPedido` are highlighted with a red box.

3

```
Pedido.java x
Source History
1 package br.edu.infnet.pedidos.domain;
2
3 import ...14 lines
17
18 @Entity
19 @Table(name = "PURCHASE_ORDER", catalog = "DR4_1", schema = "PUBLIC")
20 public class Pedido implements Serializable {
21
22     private static final long serialVersionUID = 1L;
23     @Id
24     @GeneratedValue(strategy = GenerationType.IDENTITY)
25     @Column(nullable = false)
26     private Long id;
27     @Column(name = "ORDER_DATE")
28     @Temporal(TemporalType.DATE)
29     private Date orderDate;
30     @OneToMany(mappedBy = "orderId")
31     private List<ItemPedido> itemList;
32     @Column(name = "CUSTOMER_ID")
33     private Long customerId;
34     @Column(name = "STATUS")
35     private PedidoStatus status;
36     @Column(name = "VALOR_TOTAL")
37     private ValorMonetario valorTotal;
```

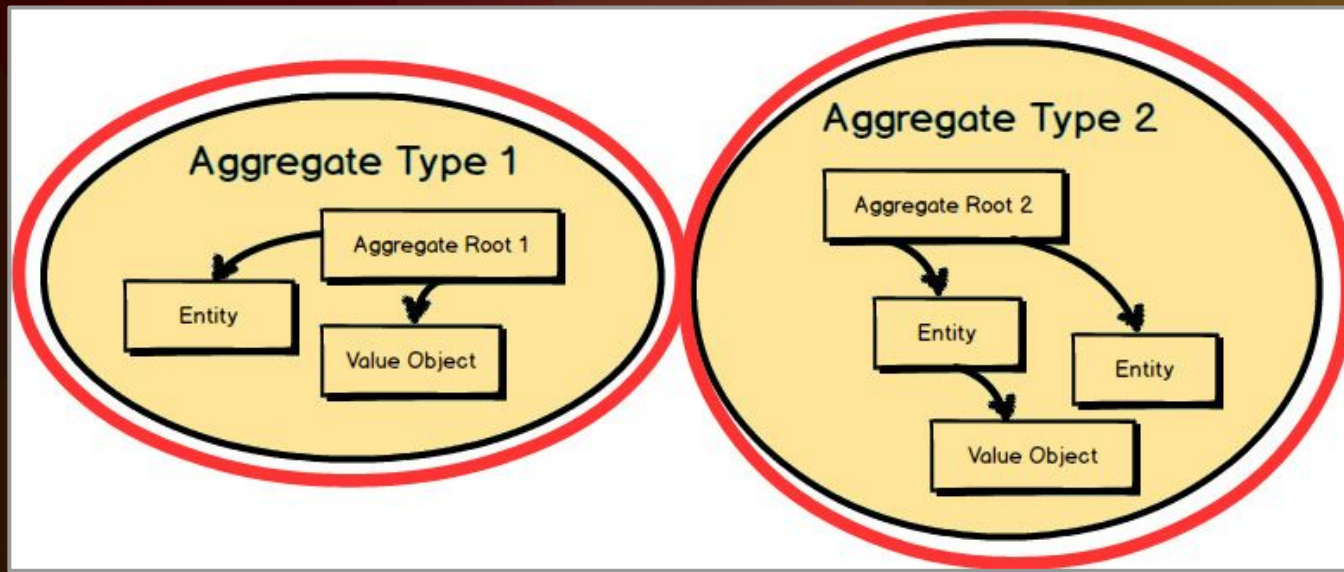
Contextos Transacionais



A **Entidade Raiz** de cada **Agregado** possui todos os outros elementos agrupados dentro dela.

O nome da Entidade Raiz é o conceito do Agregado.

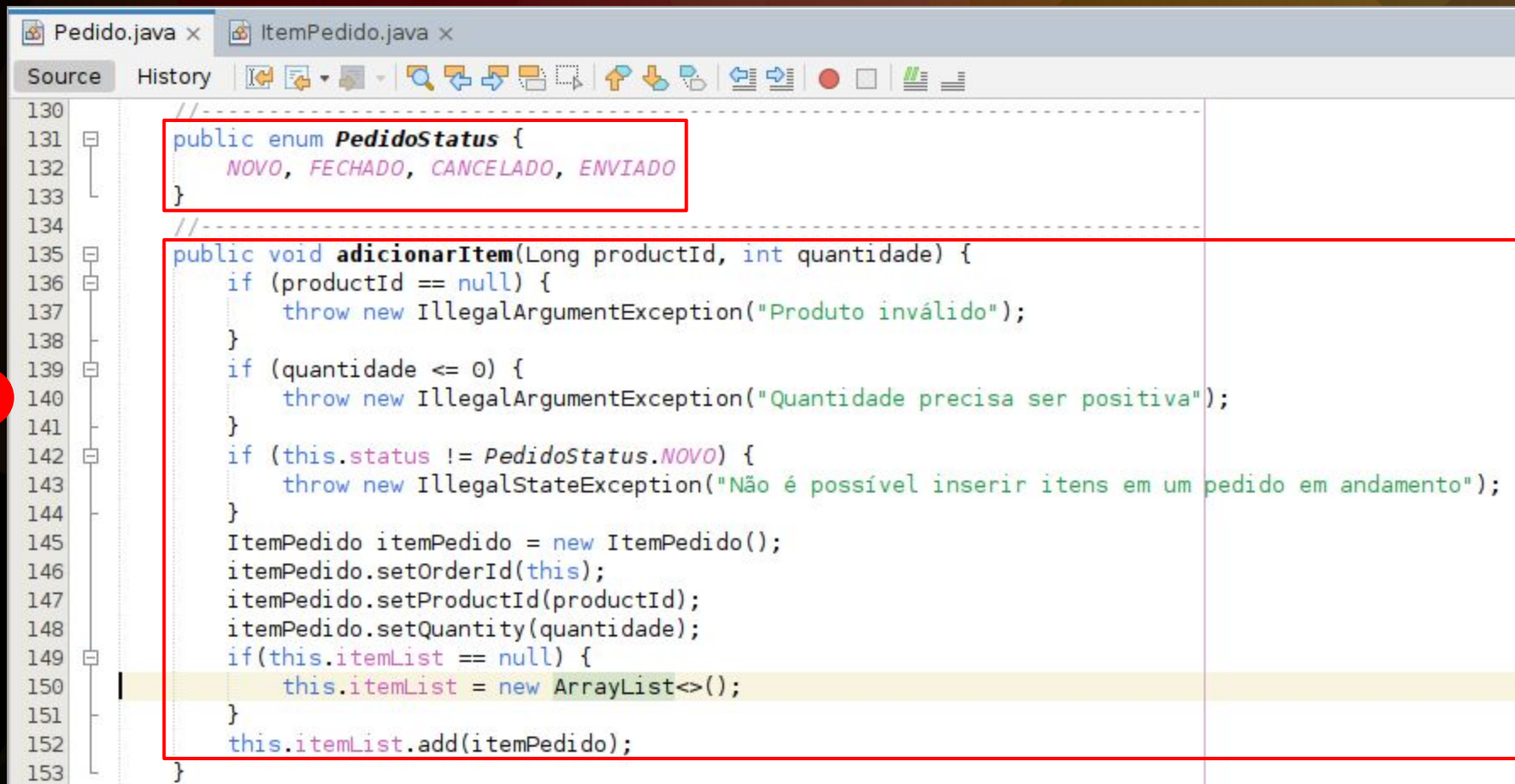
Você deve escolher um nome que descreva adequadamente o todo conceitual que o **Agregado** modela.



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 negócios, quando a transação de controle é confirmada no banco de dados.

Isso não significa necessariamente que você não deve compor outros elementos dentro de um **Agregado** que não precisam ser consistentes após uma transação.



```
130 //-----
131 public enum PedidoStatus {
132     NOVO, FECHADO, CANCELADO, ENVIADO
133 }
134 //-----
135 public void adicionarItem(Long productId, int quantidade) {
136     if (productId == null) {
137         throw new IllegalArgumentException("Produto inválido");
138     }
139     if (quantidade <= 0) {
140         throw new IllegalArgumentException("Quantidade precisa ser positiva");
141     }
142     if (this.status != PedidoStatus.NOVO) {
143         throw new IllegalStateException("Não é possível inserir itens em um pedido em andamento");
144     }
145     ItemPedido itemPedido = new ItemPedido();
146     itemPedido.setOrderId(this);
147     itemPedido.setProductId(productId);
148     itemPedido.setQuantity(quantidade);
149     if (this.itemList == null) {
150         this.itemList = new ArrayList<>();
151     }
152     this.itemList.add(itemPedido);
153 }
```



```
155 public void fecharPedido() {
156     if(this.status != PedidoStatus.NOVO) {
157         throw new IllegalStateException("Não é possível fechar um pedido que não é novo");
158     }
159     if(this.itemList.isEmpty()) {
160         throw new IllegalStateException("Não é possível fechar um pedido vazio");
161     }
162     this.status = PedidoStatus.FECHADO;
163     //DomainEvents.publish(new PedidoFechadoEvent(this.id);
164 }
165
166 public void cancelarPedido() {
167     if(this.status != PedidoStatus.FECHADO) {
168         throw new IllegalStateException("Não é possível cancelar um pedido que não esteja fechado");
169     }
170     this.status = PedidoStatus.CANCELADO;
171     //DomainEvents.publish(new PedidoCanceladoEvent(this.id);
172 }
173
174 public void enviarPedido() {
175     if(this.status != PedidoStatus.FECHADO) {
176         throw new IllegalStateException("Não é possível enviar um pedido que não esteja fechado");
177     }
178     this.status = PedidoStatus.ENVIADO;
179     //DomainEvents.publish(new PedidoEnviadoEvent(this.id);
180 }
181 //-----
```

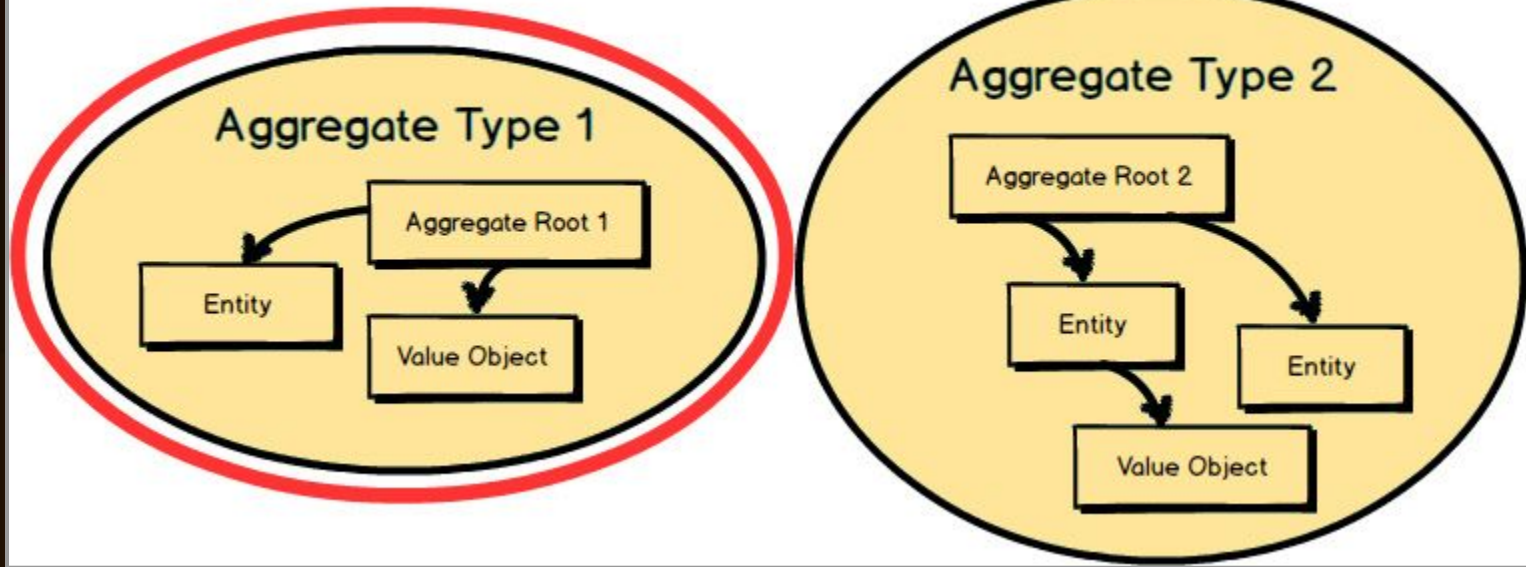
ACID

Quatro propriedades cruciais definem Transações:

- **Atomicidade** define todos os elementos que compõem uma transação completa.
- **Consistência** define as regras para manter os pontos de dados em um estado correto após uma transação.
- **Isolamento** mantém o efeito de uma transação invisível para outras pessoas até que ela seja confirmada, para evitar confusão.
- **Durabilidade** garante que as alterações de dados se tornem permanentes assim que a transação for confirmada.

Se o **Agregado** não for armazenado em um estado inteiro e válido, a operação de negócios que foi realizada seria considerada incorreta de acordo com as regras de negócios.

Single transaction



Uma regra geral do design do **Agregado**: modifique e confirme apenas uma instância do **Agregado** em uma transação.

O ponto principal a ser lembrado é que as regras de negócios são os motivadores para determinar o que deve ser completo, integral e consistente no final de uma única transação.

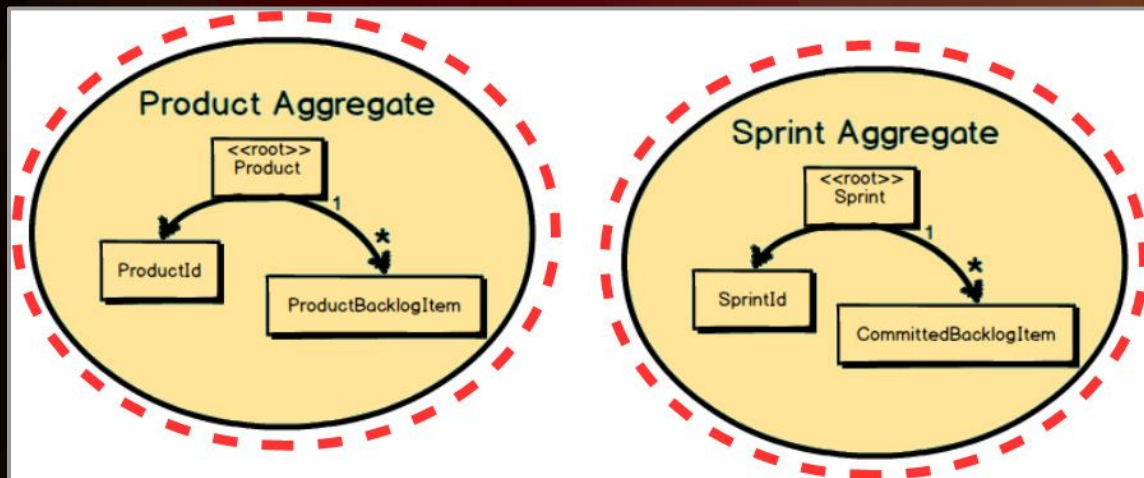
Revisando o Projeto de Agregados

Regras Básicas para o Projeto de Agregados

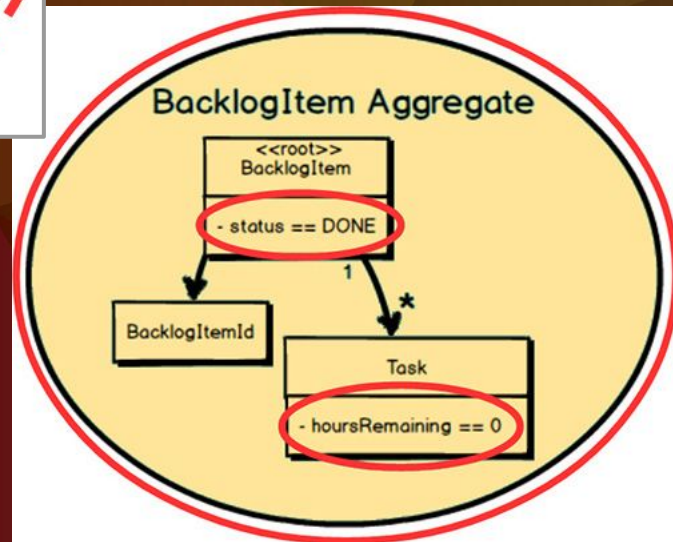
Vamos agora considerar as quatro regras básicas do projeto de agregados :

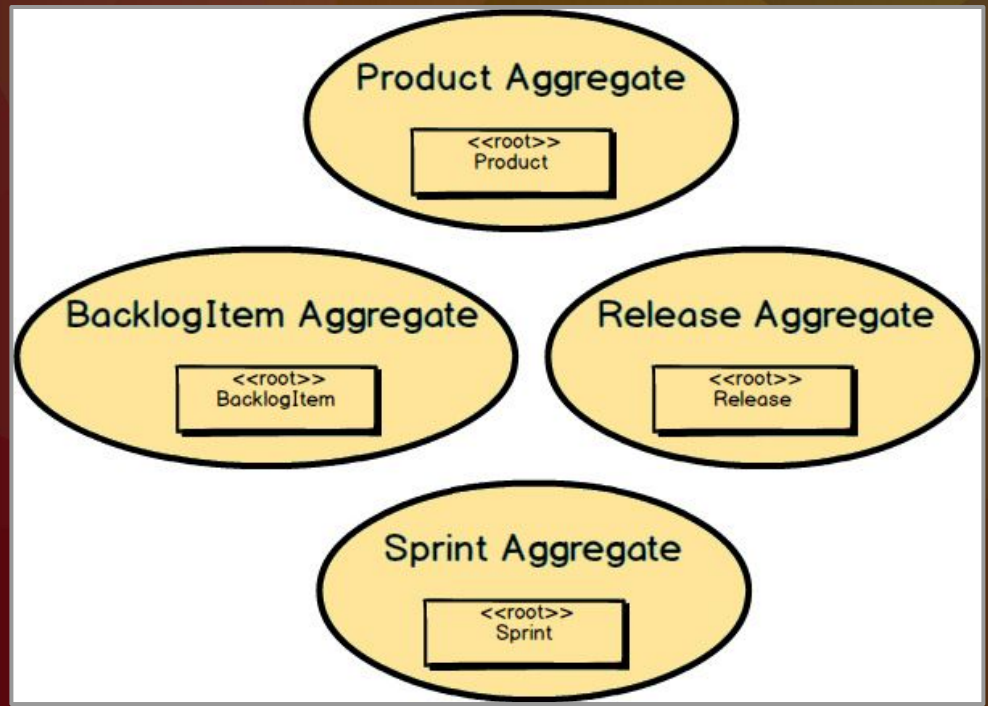
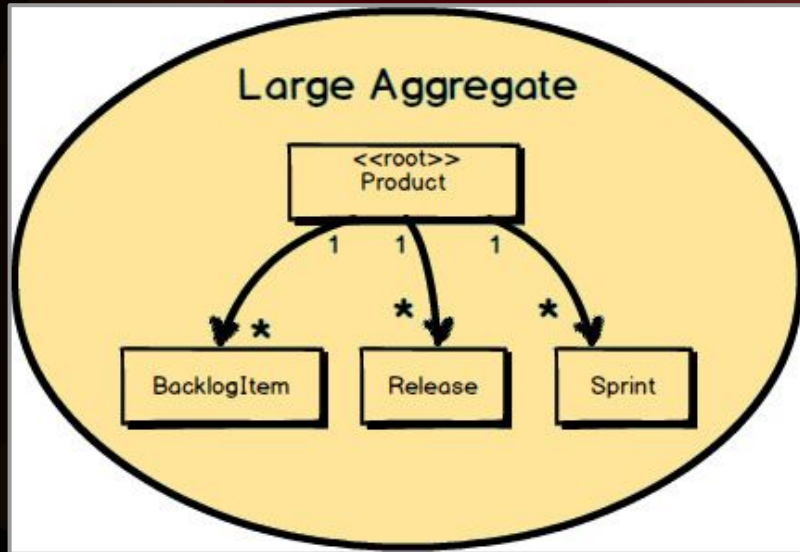


1. Proteja as invariantes de negócios dentro dos limites do **Agregado**.
2. Projetar pequenos agregados.
3. Referenciar outros agregados somente pela identidade.
4. Atualize outros agregados usando consistência eventual.



Proteja Invariantes de Negócios
dentro dos limites dos agregados





Projete Pequenos Agregados

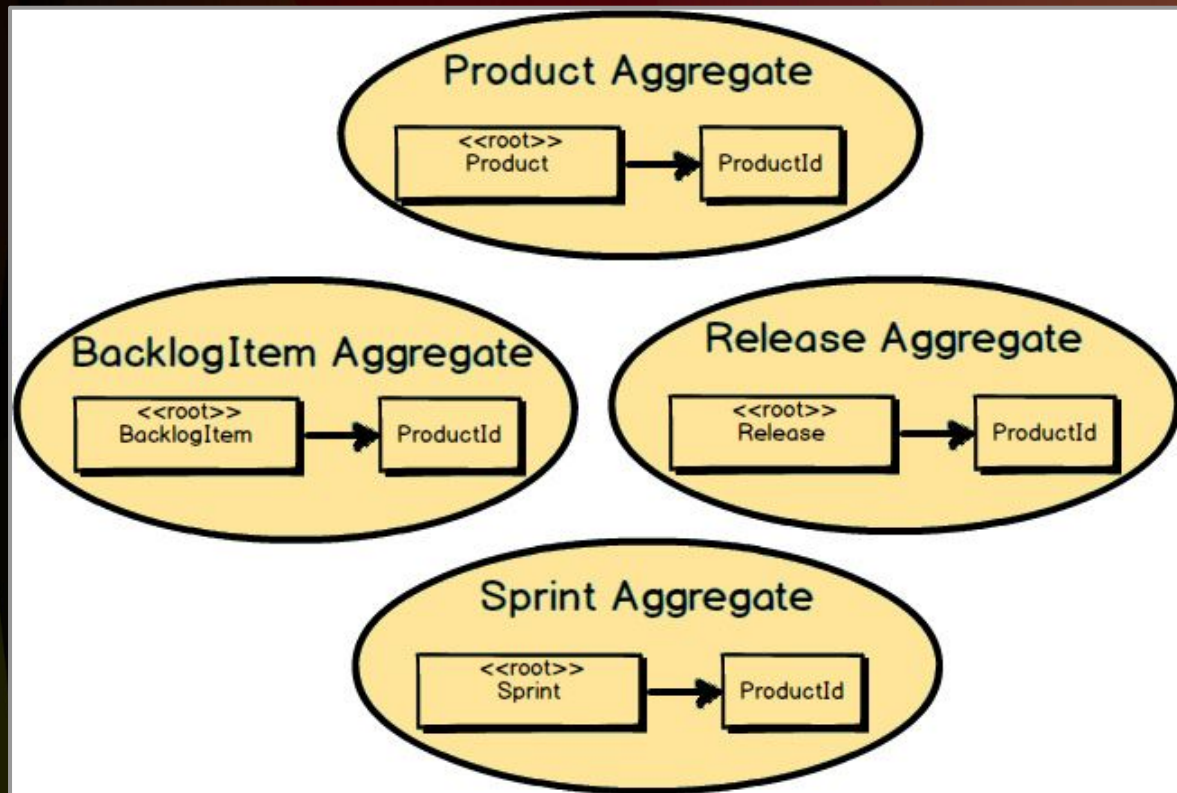
Ter em mente ao projetar Agregados o **Princípio da Responsabilidade Única**.

Se seu Agregado está tentando fazer muitas coisas, ele não está seguindo o SRP, e isso provavelmente será revelador em seu tamanho.

Referenciar Outros Agregados Somente pela Identidade

Isso ajuda ainda mais a manter o design do **Agregado** pequeno e eficiente, resultando em requisitos de memória mais baixos e carregamento mais rápido de um armazenamento de persistência.

Também ajuda a impor a regra de não modificar outras instâncias do **Agregado** dentro da mesma transação.



6

```
Pedido.java x
Source History
1 package br.edu.infnet.pedidos.domain;
2
3 import ...18 lines
21
22 @Entity
23 @Table(name = "PURCHASE_ORDER", catalog = "DR4_1", schema = "PUBLIC")
24 @NamedQueries({
25     @NamedQuery(name = "PurchaseOrder.findAll", query = "SELECT p FROM PurchaseOr
26     @NamedQuery(name = "PurchaseOrder.findById", query = "SELECT p FROM PurchaseO
27     @NamedQuery(name = "PurchaseOrder.findByOrderDate", query = "SELECT p FROM PU
28 public class Pedido implements Serializable {
29
30     private static final long serialVersionUID = 1L;
31     @Id
32     @GeneratedValue(strategy = GenerationType.IDENTITY)
33     @Basic(optional = false)
34     @Column(nullable = false)
35     private Long id;
36     @Column(name = "ORDER_DATE")
37     @Temporal(TemporalType.DATE)
38     private Date orderDate;
39     @OneToMany(mappedBy = "orderId")
40     private List<ItemPedido> itemList;
41     // @JoinColumn(name = "CUSTOMER_ID", referencedColumnName = "ID")
42     // @ManyToOne
43     // private Cliente customerId;
44     @Column(name = "CUSTOMER_ID")
45     private Long customerId;
46     @Column(name = "STATUS")
47     private PedidoStatus status;
48     @Column(name = "VALOR_TOTAL")
49     private ValorMonetario valorTotal;
```

7

```
Pedido.java x ItemPedido.java x
Source History
1 package br.edu.infnet.pedidos.domain;
2
3 import ...13 lines
16
17 @Entity
18 @Table(catalog = "DR4_1", schema = "PUBLIC")
19 @NamedQueries({
20     @NamedQuery(name = "Item.findAll", query = "SELECT i FROM Item i"),
21     @NamedQuery(name = "Item.findById", query = "SELECT i FROM Item i WHERE i.id = :id"),
22     @NamedQuery(name = "Item.findByQuantity", query = "SELECT i FROM Item i WHERE i.quantity = :quantity"),
23     @NamedQuery(name = "Item.findByTotal", query = "SELECT i FROM Item i WHERE i.total = :total")
24 public class ItemPedido implements Serializable {
25
26     private static final long serialVersionUID = 1L;
27     @Id
28     @GeneratedValue(strategy = GenerationType.IDENTITY)
29     @Basic(optional = false)
30     @Column(nullable = false)
31     private Long id;
32     private Integer quantity;
33     // @Max(value=?) @Min(value=?)//if you know range of your decimal fields consider it
34     @Column(precision = 20, scale = 2)
35     private BigDecimal total;
36     // @JoinColumn(name = "PRODUCT_ID", referencedColumnName = "ID")
37     // @ManyToOne
38     // private Produto productId;
39     @Column(name = "PRODUCT_ID")
40     private Long productId;
41     @JoinColumn(name = "ORDER_ID", referencedColumnName = "ID")
42     @ManyToOne
43     private Pedido orderId;
```

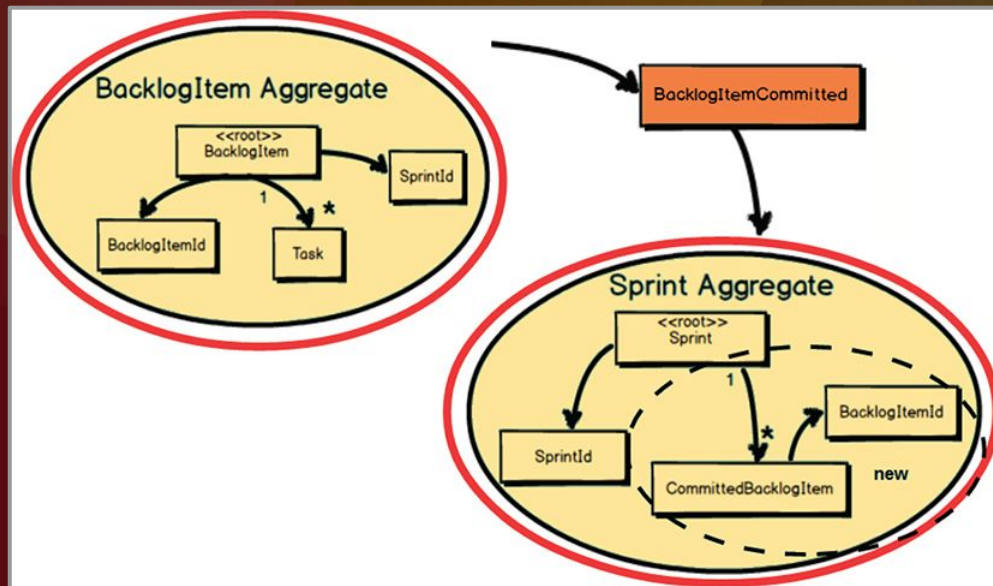

Atualizar Outros Agregados Usando Eventos de Domínio

Como parte da transação do agregado BacklogItem, ele publica um **Evento de Domínio** chamado BacklogItemCommitted.

A transação é concluída e seu estado é persistido junto com o evento BacklogItemCommitted.

Quando o evento fizer seu caminho para um assinante local, uma transação é iniciada e o estado de Sprint é modificado para conter o BacklogItemId.

Sprint contém o BacklogItemId dentro de uma nova entidade CommittedBacklogItem.



Pedido.java x

Source History

```
134 //-----
135 public void adicionarItem(Long productId, int quantidade) {...19 lines }
154
155 public void fecharPedido() {
156     if(this.status != PedidoStatus.NOVO) {
157         throw new IllegalStateException("Não é possível fechar um pedido que não é novo");
158     }
159     if(this.itemList.isEmpty()) {
160         throw new IllegalStateException("Não é possível fechar um pedido vazio");
161     }
162     this.status = PedidoStatus.FECHADO;
163     //DomainEvents.publish(new PedidoFechadoEvent(this.id);
164 }
165
166 public void cancelarPedido() {
167     if(this.status != PedidoStatus.FECHADO) {
168         throw new IllegalStateException("Não é possível cancelar um pedido que não esteja fechado");
169     }
170     this.status = PedidoStatus.CANCELADO;
171     //DomainEvents.publish(new PedidoCanceladoEvent(this.id);
172 }
173
174 public void enviarPedido() {
175     if(this.status != PedidoStatus.FECHADO) {
176         throw new IllegalStateException("Não é possível enviar um pedido que não esteja fechado");
177     }
178     this.status = PedidoStatus.ENVIADO;
179     //DomainEvents.publish(new PedidoEnviadoEvent(this.id);
180 }
181 //-----
```

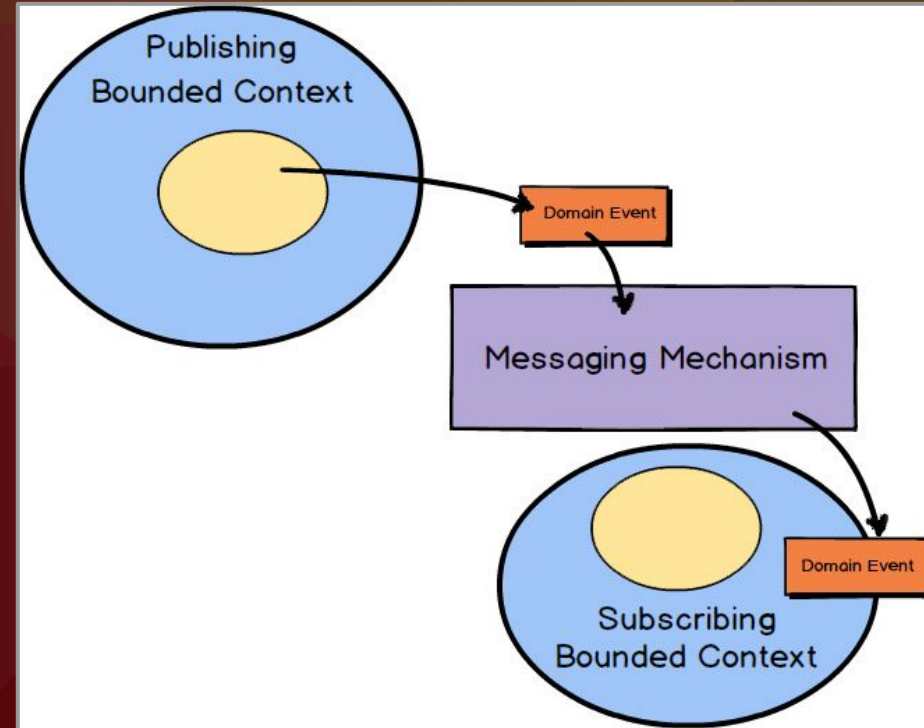
Eventos de Domínio - Parte 1

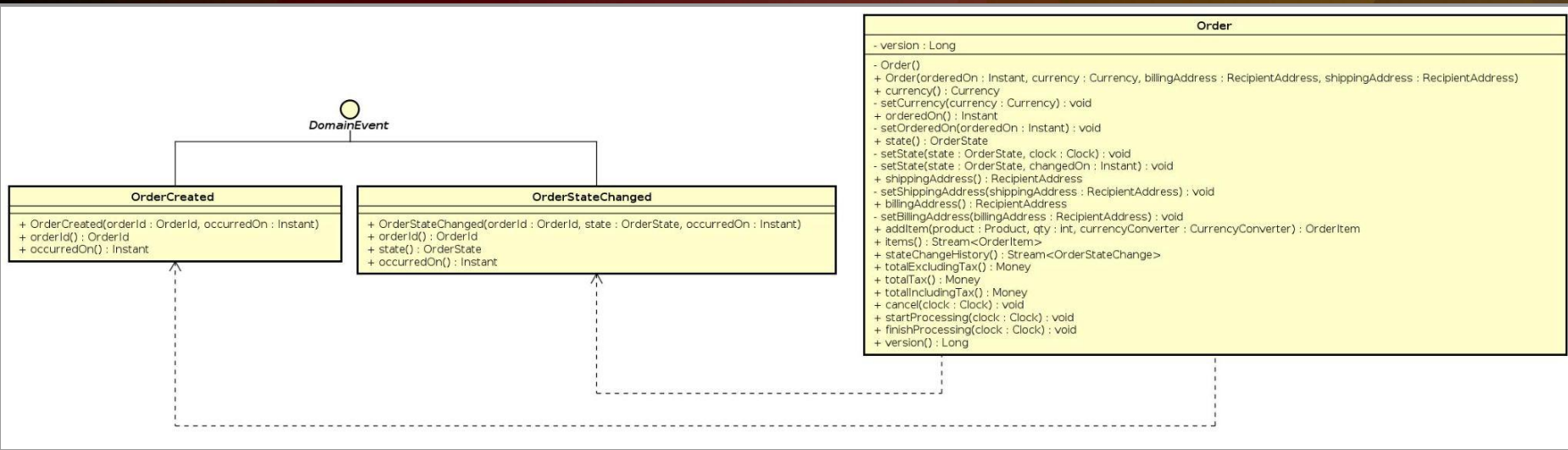
Um **Evento de Domínio** é um registro de alguma ocorrência significativa para os negócios em um Contexto Limitado.

Frequentemente durante o design tático os **Eventos de Domínio** são conceituados e se tornam parte do seu Domínio Principal.

Um domínio de negócios fornece consistência causal se suas operações têm relacionamento causa-efeito e são vistas por cada nó dependente de um sistema distribuído na mesma ordem.

Isso significa que operações causalmente relacionadas devem ocorrer em uma ordem específica e, portanto, uma coisa não pode acontecer a menos que outra coisa aconteça antes dela.





<https://github.com/peholmst/DDDExample>

```
1 package net.pkapps.ddd.shared.domain.base;
2
3 import org.springframework.lang.NonNull;
4
5 import java.time.Instant;
6
7 /**
8  * Interface for domain events.
9  */
10 public interface DomainEvent extends DomainObject {
11
12     /**
13      * Returns the time and date on which the event occurred.
14      */
15     @NonNull
16     Instant occurredOn();
17 }
```



```
DomainEvent.java x OrderCreated.java x OrderStateChanged.java x
Source History
1 package net.pkapps.ddd.orders.domain.model.event;
2
3 import ...8 lines
11
12 public class OrderCreated implements DomainEvent {
13
14     @JsonProperty("orderId")
15     private final OrderId orderId;
16     @JsonProperty("occurredOn")
17     private final Instant occurredOn;
18
19     @JsonCreator
20     public OrderCreated(@JsonProperty("orderId") @NonNull OrderId orderId,
21                         @JsonProperty("occurredOn") @NonNull Instant occurredOn) {
22         this.orderId = Objects.requireNonNull(orderId, "orderId must not be null");
23         this.occurredOn = Objects.requireNonNull(occurredOn, "occurredOn must not be null");
24     }
25
26     @NonNull
27     public OrderId orderId() {
28         return orderId;
29     }
30
31     @Override
32     @NonNull
33     public Instant occurredOn() {
34         return occurredOn;
35     }
36 }
```

```
DomainEvent.java x OrderCreated.java x OrderStateChanged.java x
Source History
1 package net.pkapps.ddd.orders.domain.model.event;
2
3 import ...9 lines
12
13 public class OrderStateChanged implements DomainEvent {
14
15     @JsonProperty("orderId")
16     private final OrderId orderId;
17     @JsonProperty("state")
18     private final OrderState state;
19     @JsonProperty("occurredOn")
20     private final Instant occurredOn;
21
22     @JsonCreator
23     public OrderStateChanged(@JsonProperty("orderId") @NonNull OrderId orderId,
24                             @JsonProperty("state") @NonNull OrderState state,
25                             @JsonProperty("occurredOn") @NonNull Instant occurredOn) {
26         this.orderId = Objects.requireNonNull(orderId, "orderId must not be null");
27         this.state = Objects.requireNonNull(state, "state must not be null");
28         this.occurredOn = Objects.requireNonNull(occurredOn, "occurredOn must not be null");
29     }
30
31     @NonNull
32     public OrderId orderId() {
33         return orderId;
34     }
35
36     @NonNull
37     public OrderState state() {
38         return state;
39     }
40
41     @Override
42     @NonNull
43     public Instant occurredOn() {
44         return occurredOn;
45     }
46 }
```



ProductCreated

- * tenantId
- * productId
- * name
- * description

SprintScheduled

- * tenantId
- * sprintId
- * productId
- * name
- * description
- * startsOn
- * endsOn

ReleaseScheduled

- * tenantId
- * releaseId
- * productId
- * name
- * description
- * targetDate

BacklogItemPlanned

- * tenantId
- * backlogItemId
- * productId
- * sprintId
- * story
- * summary

BacklogItemCommitted

- * tenantId
- * backlogItemId
- * sprintId