业务逻辑设计

- 在微服务架构下，服务的领域模型设计至关重要
  - 消除服务间的对象引用
  - 能够在微服务架构的事务管理约束下工作
- DDD 聚合模式
  - 业务逻辑由一组聚合构成
  - 聚合包含多个对象。聚合中的所有对象是一个单元

5.1 业务逻辑模式

- 业务逻辑是六边形架构的核心
- 通过入站适配器和出站适配器，业务逻辑与外部系统进行集成
- **一个服务由业务逻辑和以下适配器组成**
  - [x] REST API Adapter：提供 REST API 的入站适配器
  - [ ] OrderCommandHandlers：提供命令消息 API 的入站适配器
  - [x] Database Adapter：访问数据库的出站适配器
  - [ ] Domain Event Publishing Adapter：往消息代理（Broker）发送事件的出站适配器
- 相对适配器，业务逻辑的设计要复杂得多

事务脚本模式

> **把业务逻辑组织成一组面向过程的事务脚本，每个脚本能够处理一种请求**

- 使用面向过程的思想实现用例
- 把所有的业务逻辑都写到入站端口中
- 适合开发简单逻辑（CRUD）
- 对于复杂逻辑，会导致代码无法复用，此时应使用面向对象设计

领域模型模式

> **业务逻辑由一组具有状态和行为的对象构成**

使用 DDD 战术模式设计领域模型

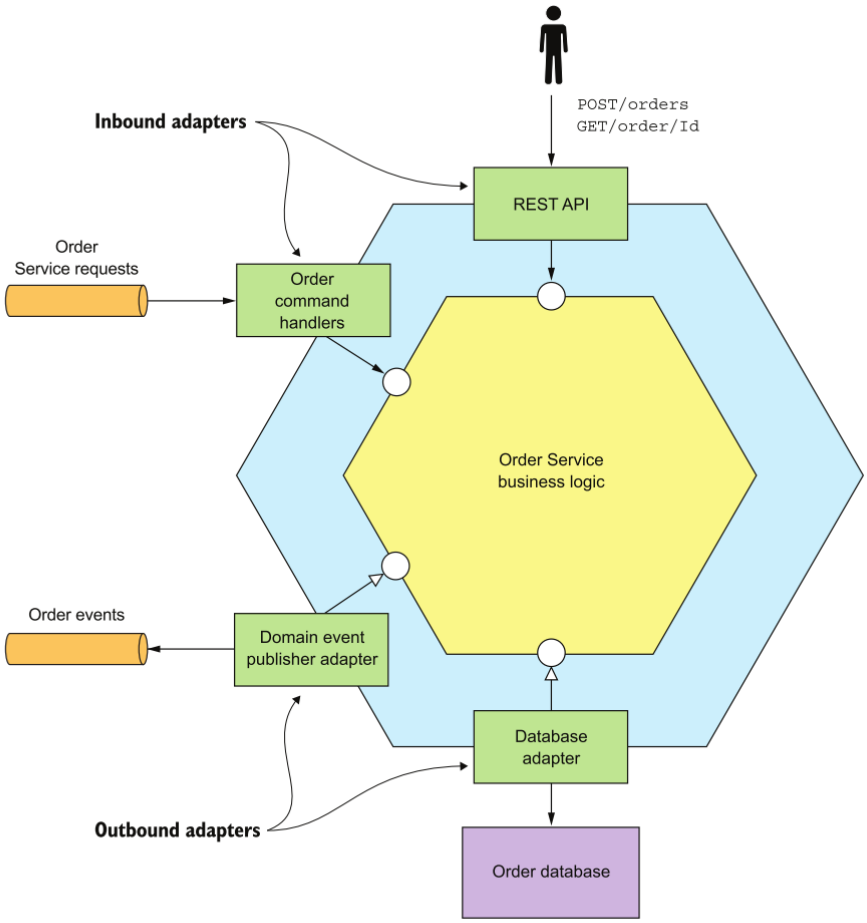


Figure 5.1 The `Order Service` has a hexagonal architecture. It consists of the business logic and one or more adapters that interface with external applications and other services.

- DDD 战术模式把领域模型中的类分为以下几类：
  - 实体（Entity）：具有持久化标识（ID）。只有ID不同，其他属性都相同的实体也是两个不同的对象。JPA中的实体类（@Entity）一般有对应的DDD的实体
  - 值对象（Value Object）：没有状态，只包括一组值。具有相同值的两个值对象可用来进行交换。比如，

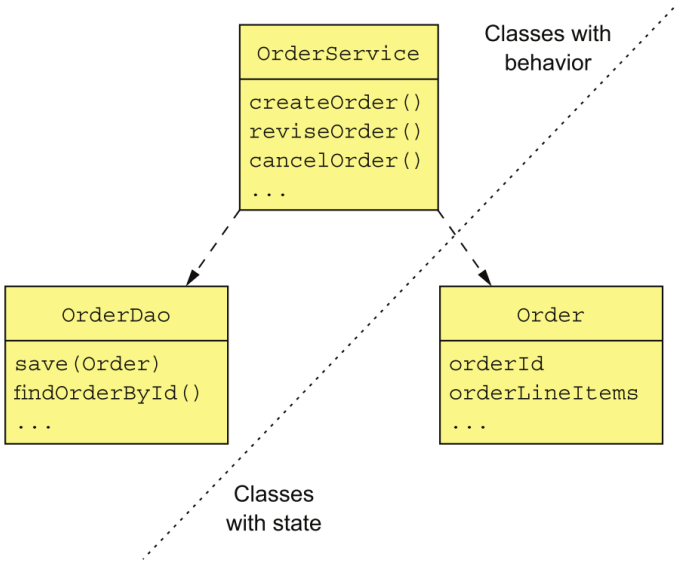


Figure 5.2 Organizing business logic as transaction scripts. In a typical transaction script–based design, one set of classes implements behavior and another set stores state. The transaction scripts are organized into classes that typically have no state. The scripts use data classes, which typically have no behavior.

Money，Height等

  - 工厂（Factory）：如果创建某个对象很复杂，可使用工厂模式对外屏蔽创建过程。具体参考设计模式。
  - 资源库（Repository）：提供并封装访问数据库的访问机制。
  - 领域服务（Service）：对象应该包含属性和行为，但有些行为不属于任何对象。把这些行为封装为一个服务。
    - 注意与之前讲到应用服务的区别。
    - 领域服务属于业务逻辑，应用服务不属于业务逻辑
    - 六边形架构中，应用服务会调用包括领域服务在内的业务逻辑
  - 聚合（Aggregate）：包含一组具有业务一致性的领域对象，提供了一种边界。外部无法访

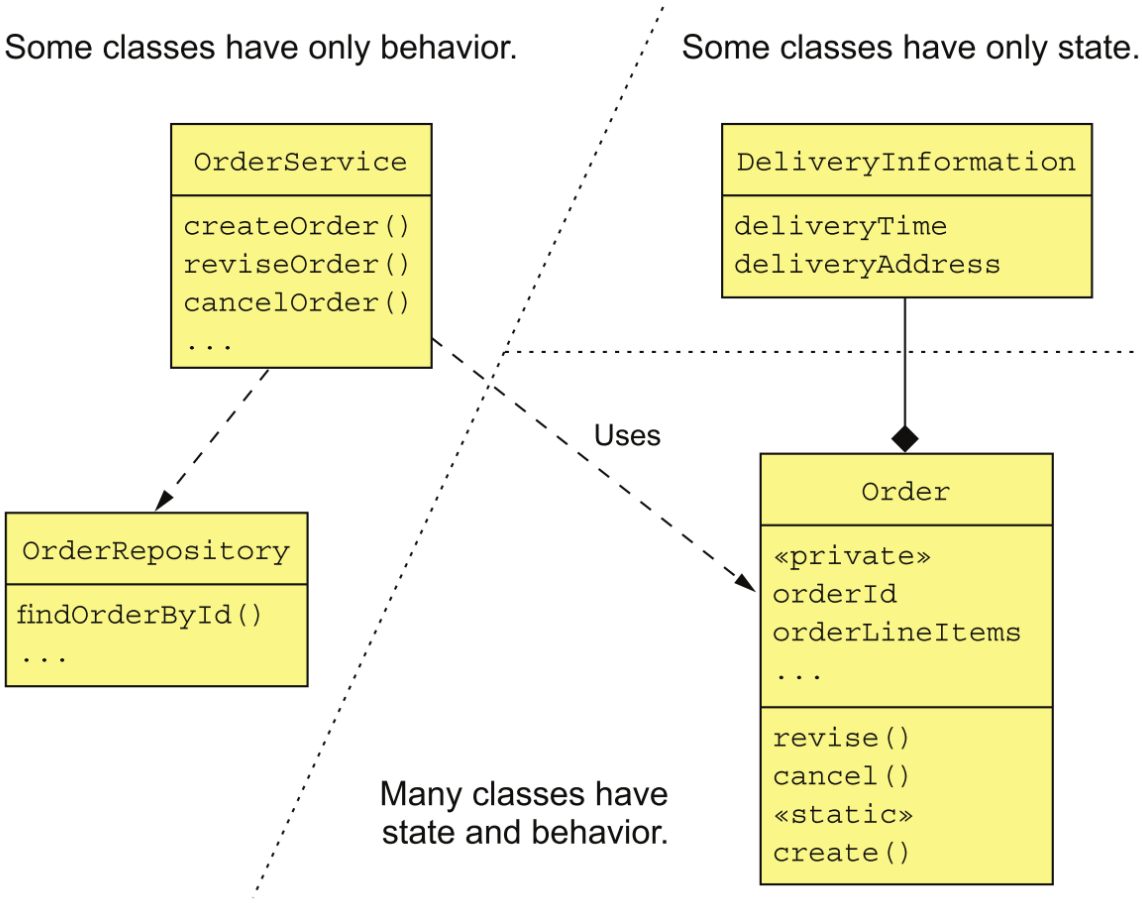


Figure 5.3 Organizing business logic as a domain model. The majority of the business logic consists of classes that have state and behavior.

问边界内的对象，只能通过聚合根（Aggregate Root）完成内部对象的操作。

实体

- 何时使用
  - 需要身份：唯一的Id，与其他对象进行区分
  - 具有生命周期：创建/撤销订单
  - 需要修改自身状态：修改订单
- 实现
  - 分配标识符（ID）
    - 自然键：ISBN，身份证号，学号，教工号，
    - 随机生成：`GUID`，`UUID`等
    - 存储（如数据库）生成或`ORM`框架生成
    - 复合主键：`@EmbeddedId`
  - 提供基类`Entity<TypeOfId>`：只要Id相等就相等（`equals`方法）
  - 避免成为数据容器：只关注行为，不关注数据（公开行为，而不是状态）
    - 当界面需要时：使用`DTO（Data Transfer Object）`模式
  - 自验证，拒绝无效状态（`validate`）
  - 具体情况具体分析：顾客手中的钱（值对象）/印钞厂中的钱（实体）
- 案例中的实体
  - 作业（Assignment）：可编辑，有状态（过期）
  - 提交（Submission）：可多次提交，每次提交都有分数需要保存（后期功能）

值对象

- 何时使用
  - 表示描述性的，不具有身份的概念：身高，钱，日期，重...
  - 增强明确性：避免使用原始类型：`Price price` vs `double price`
  - 不可变：永远不可变更. `Money.add() { return new Money }`
- 实现
  - 相等性：所有的属性都必须相等
  - 考虑创建基类`ValueObject<T>`，提供`equals`和`hashcode`的模板
  - 自验证：`validate() { if(price < 0) throw new NegativePriceException}`
  - 静态工厂方法：`new LocalDate(..) -> LocalDate.of(..)`
  - 微类型：`Price price`
  - 避免使用集合：`Set<PhoneNumber> -> PhoneBook`
  - 尽可能使用值对象
- 案例中的值对象
  - 截止日期（`DeadLine`）-> 具有一定的业务逻辑，封装进值对象可避免实体类的臃肿

领域服务

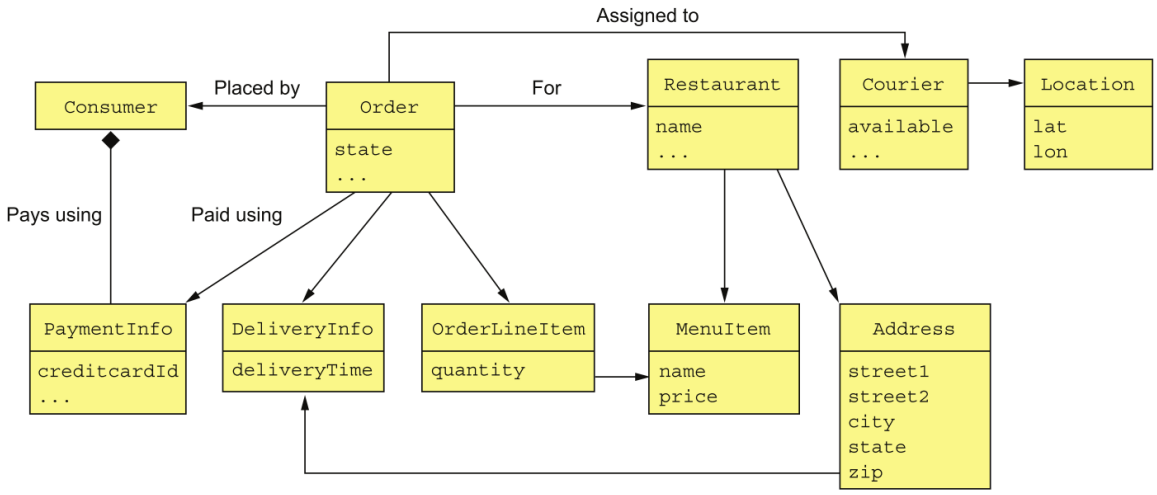
- 何时使用
  - 领域行为概念的行为不属于任何实体和值对象：银行转账、学生课程成绩统计等
- 实现
  - 可以接收Spring容器的管理
  - 将实体和值对象组织在一起进行无状态行为
  - 不要把所有的领域逻辑都推入到领域服务中
  - 可以在应用服务中使用领域服务：注意与应用服务的区别
  - 可以使用依赖注入模式将领域服务注入到领域模型中（如实体，值对象等）
  - 使用领域事件避免将领域服务与领域模型耦合
- 案例中的领域服务
  - 批量导入用户信息（`BatchImportUserService`）：不属于任何实体或值对象

资源库（Repository）

- 何时使用
  - 当需要对领域对象（以聚合为单位）进行检索与持久化时
  - 避免领域对象与基础架构代码（DAO）之间的直接耦合
  - 作为六边形架构的出站端口
- 实现
  - 仅允许通过聚合根进行检索与持久化
  - 保持持久化与查询机制对于领域对象的透明性
  - 定义了领域模型与数据模型的边界
  - 使用依赖注入模式将具体的实现注入到业务逻辑中
  - 每个聚合（注意不是实体）都要有一个资源库
- 案例中的资源库
  - AssignmentRepository
  - SubmissionRepository

5.2 使用 DDD 聚合模式设计领域模型

- 在传统的OOD设计中，领域模型（类图）是
  - 一组类及其之间的关系
  - 使用包对类进行组织
  - 问题：缺少清晰的边界



- 问题：哪些类属于订单（Order）业务？
  - 直接操作订单项（`OrderLineItem`）会导致状态不一致
  - 不一致的例子参考教材（5.2.1节）

使用聚合

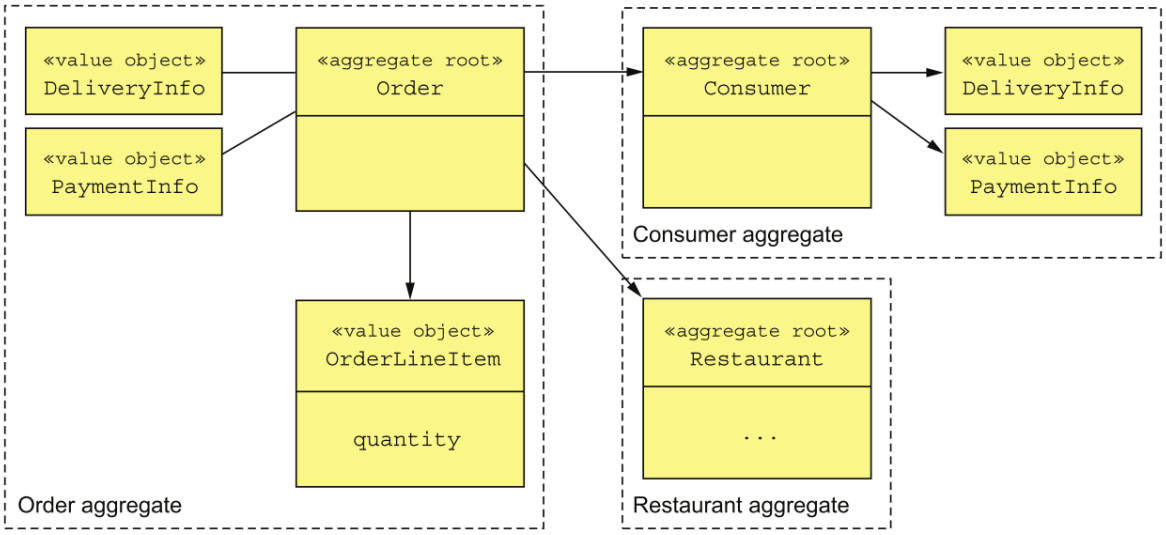


Figure 5.5　Structuring a domain model as a set of aggregates makes the boundaries explicit.

- 聚合把领域模型进行分组，每个聚合都确定某种业务的边界
- 业务操作（增删改查）是对整个聚合（通过聚合根如Order）进行操作，而不是直接操作其内部的对象（如`OrderLineItem`）
  - `order.items.stream().filter(i->item.productId() == productId).FindFirst().quantity = newQuantity`
  - `order.changeQuantityOf(productId, newQuantity);`
  - 这是有业务价值的
- **聚合也是数据库读写操作的最小单元。资源库（Repository）应该操作聚合，而不是简单的实体**
- 聚合规则
  - 聚合之间只引用聚合根，并使用聚合根ID进行引用

- **聚合内部使用主键进行引用（而不要使用对象）**
  - 在传统OOD中，应使用关联。而主键/外键属于数据库的概念
  - 它用于真正确保聚合的边界地位（如果还使用对象引用，有可能会深入到对象内部进行操作）
  - 技术上简化。适用于各种数据库
  - `class Customer { List<Order> orders; }`
  - `class Order { CustomerId customerId; }`

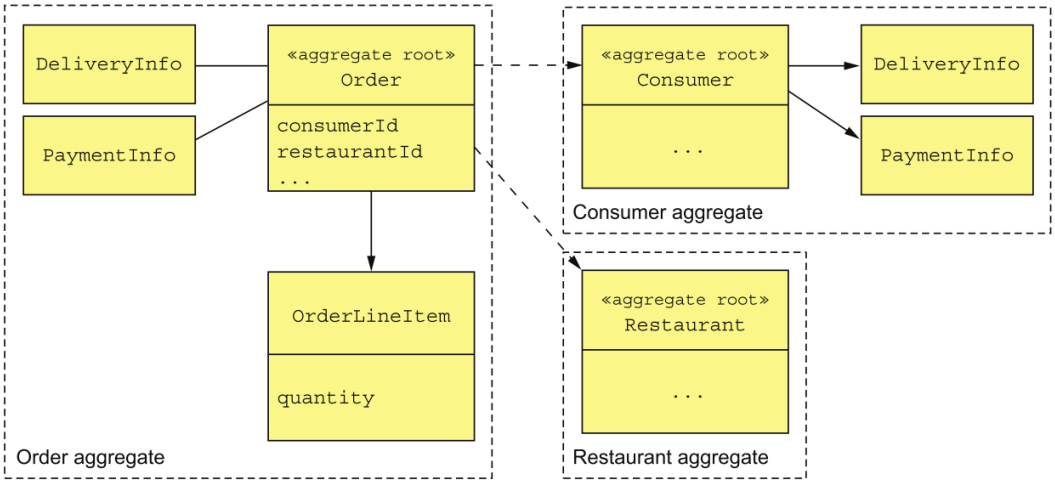


Figure 5.6 References between aggregates are by primary key rather than by object reference. The `Order` aggregate has the IDs of the `Consumer` and `Restaurant` aggregates. Within an aggregate, objects have references to one another.

  - 事务边界：一次事务只对一个聚合进行操作（参考）
  - 不要使用延迟加载
  - 被删除时，内部的所有领域对象都必须被删除
    - 失去聚合根，内部对象没有单独存活的意义
    - 类似于UML中的组合关系（整体-部分具有相同的生命周期）
- 真正的难题：如何确定聚合及其范围
  - 设计小聚合：将大模型分解为小模型
- **在一个微服务内部，其业务逻辑就是由一个或多个聚合组成的**
- **应用服务（ApplicatinService）是操作聚合的入口**

5.3 发布领域事件

- 领域事件模式

> 当一个聚合被创建或着发生重要的状态改变时，该聚合应发布一个领域事件

- 为什么需要领域事件