领域驱动设计(DDD)

引例

DDD

子域与界限上下文

经典分层架构

UI层与应用层

领域层

基础设施层

总结

引例

在电商系统中"修改商品数量"的业务需求：

> 在用户未支付订单（Order）之前，用户可以修改Order中商品（Goods）的数量, 数量更新后需要更新相应的订单金额（totalPrice）。

实现一： Service + 贫血模型

贫血类 `Order, OrderItem`，充当ORM持久化对象

```
public class Order {       // order 表
    private Long id;       // 订单id
    private Long totalPrice;
    private Status status;
    // address, time etc
    // getter, setter
}

public class OrderItem {   // order_item 表
    private Long id;
    private Long orderId;      // 订单id
    private Long goodsId;
    private Integer count;
    private Long price;
    // getter setter
}
```

`OrderServiceImpl`类

```
    @Transactional
    public void changeGoodsCount(long id, long goodsId, int count) {
        Order order = dao.findOrderById(id);
        if (order.getStatus() == PAID) {
            throw new OrderCannotBeModifiedException(id);
        }
        List<OrderItem> orderItems = dao.findItemByOrderId(id);
        findAndSetCount(orderItems);
        order.setTotalPrice(calculateTotalPrice(orderItems));
        dao.saveOrUpdate(order);
        dao.saveOrUpdate(item);
    }
```

面向过程编程，业务逻辑`Service`层中实现，随着项目演化，这些业务逻辑会分散在不同的Service类中。

实现二： 基于事务脚本的实现

> 事务脚本: 通过过程的调用来组织业务逻辑，每个过程处理来自表现层的单个请求。

```
@Transactional
public void changeGoodsCount(long id, long goodsId, int count) {
    OrderStatus orderStatus = DAO.getOrderStatus(id);
    if (orderStatus == PAID) {
        throw new OrderCannotBeModifiedException(id);
    }
    DAO.updateGoodsCount(id, command.getGoodsId(), command.getCount());
    DAO.updateTotalPrice(id);
}
```

实现三： 基于领域对象实现

业务表达逻辑被内聚到领域对象（`Order`）中， `Order`不再是一个贫血对象，除了属性还有行为。

```
public class Order {
    private List<OrderItem> items;

    public void changeGoodsCount(long goodsId, int count) {
        if (status == PAID) {
            throw new OrderCannotBeModifiedException(id);
        }
        OrderItem item = items.stream().filter(x -> x.getGoodsId() == goodsId).findFirst().get();
        item.setCount(count);
        this.totalPrice = items.stream().mapToLong(x -> x.getPrice() * x.getCount()).sum();
    }
}
```

`OrderApplicationService` 提供接口

```
@Transactional
public void changeGoodsCount(long id, long goodsId, int count) {
    Order order = repository.findById(id);
    order.changeGoodsCount(goods, count);
    repository.save(order);
}
```

Quote

> **I find this a curious term because there are few things that are less logical than business logic.**

> Matin Flowler @ Patterns of Enterprise Application Architecture

DDD简述

2004年Eric Evans发表Domain-Driven Design: Tackling Complexity in the Heart of Software，简称Evans DDD。领域驱动设计分为两个阶段：

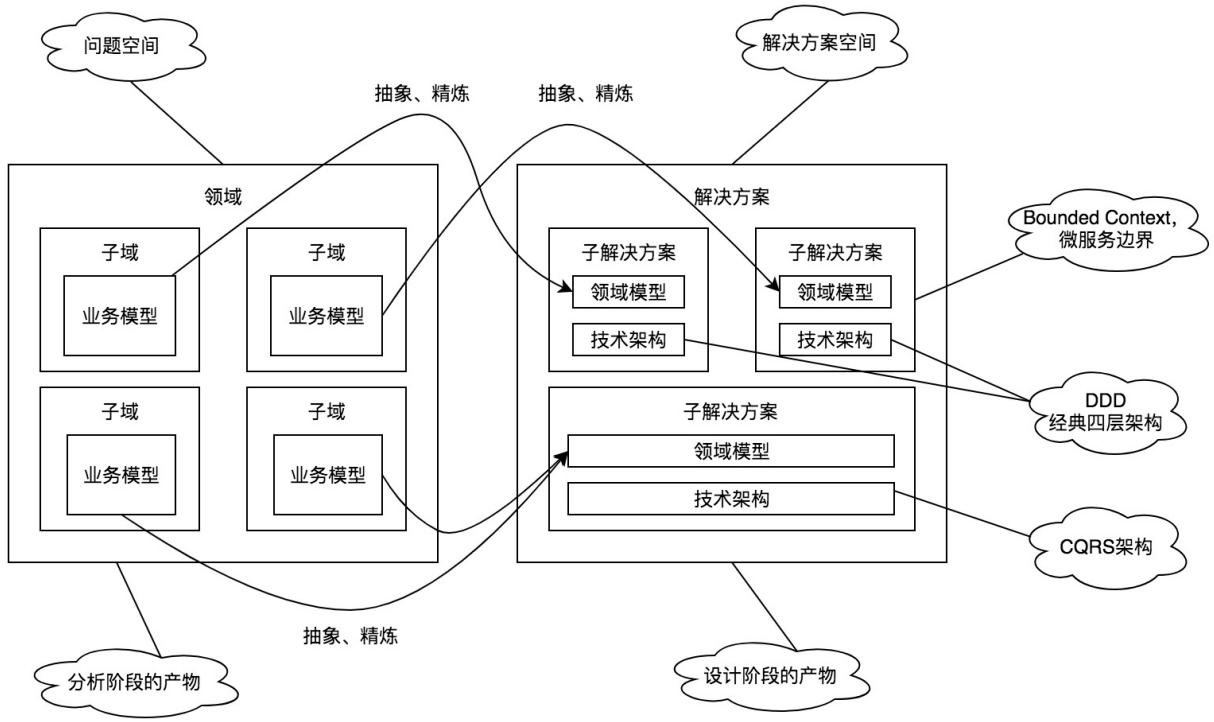
- 以一种领域专家、设计人员、开发人员都能理解的通用语言作为相互交流的工具，在交流的过程中发现领域概念，然后将这些概念设计成一个领域模型；
- 由领域模型驱动软件设计，用代码来实现该领域模型；

它是一套完整而系统的设计方法，尝试对业务复杂性和技术复杂性进行分离或者至少降低耦合性，达到软件架构和业务清晰的表达。

战略: 子域和界限上下文划分

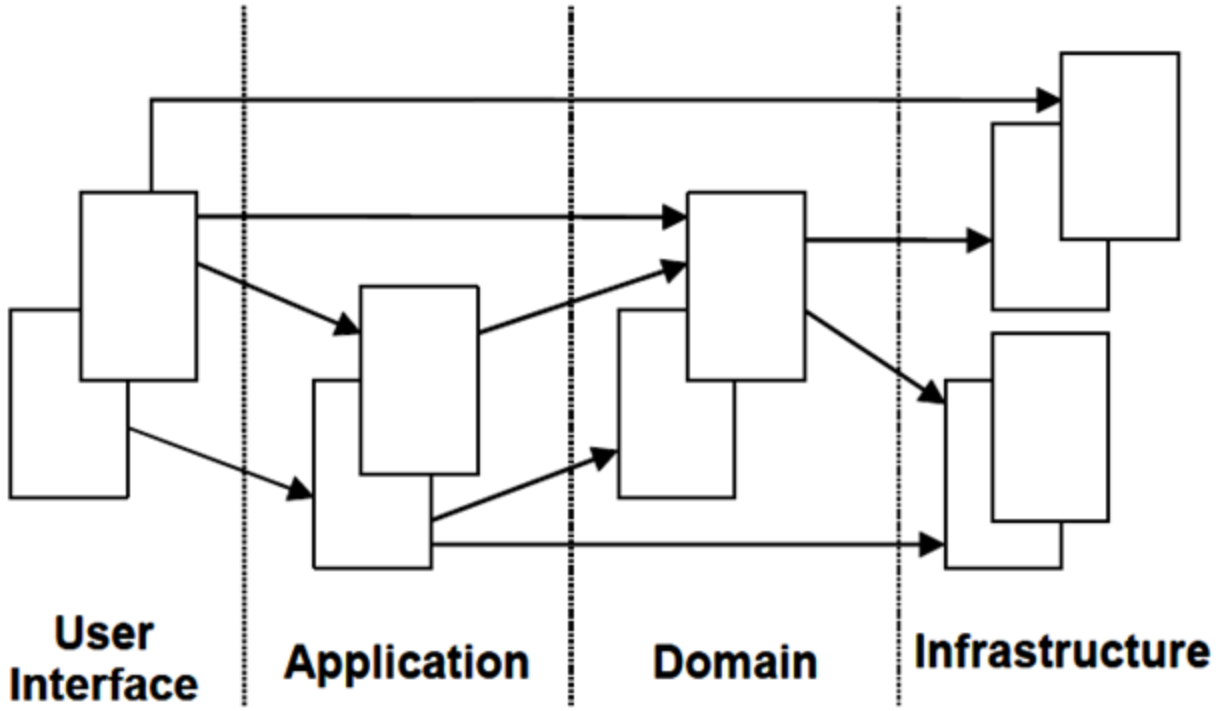
领域: 一种边界，范围，可以理解为业务边界;

Bounded Context: 界限上下文，定义了解决方案的边界;



战术: 经典分层架构

- UI 层: controller, outer-api；
- 应用层: 业务逻辑无关，一个业务用例对应 `ApplicationService` 上的一个方法；
- 领域层: 核心层, 表达业务，包含领域模型及领域服务；
- 基础设施层: 为上层提供基础服务，如持久化服务等。



用户界面层(User Interface): 上下游适配器

适配不同的协议: `REST`, `RPC`, `SOAP` 等，负责向用户展现信息以及执行用户命令。更细的方面来讲就是：

1. 请求应用层以获取用户所需要展现的数据；
2. 发送命令给应用层要求其执行某个用户命令；

```
    @PostMapping("/{id}/changeGoods")
    public void changeGoodsCount(@PathVariable(name = "id") Long id, @RequestBody @Valid ChangeGoodsCountCommand command) {
        orderApplicationService.changeGoodsCount(id, command.getProdcutId(), command.getCount());
    }
```

应用层 (ApplicationService): 领域模型的门面

作用: 对外为展现层提供各种应用功能（包括查询或命令），对内调用领域层（领域对象或领域服务）完成各种业务逻辑。

原则:

(1). 一个业务用例对应ApplicationService上的一个业务方法，比如修改产品个数: `OrderApplicationService.changeGoodsCount()`;

(2). 与事务一一对应;

(3). 不包含业务逻辑，所有业务内聚在聚合根中，只用对领域对象进行调用，无需知道领域模型内部实现;

(4). 作为领域模型的门面，封装领域模型的对外提供的功能，不应处理UI交互或者通信协议之类的技术细节

领域服务 (DomainService): 多领域模型协调者

领域中的一些操作比如涉及到多个领域模型对象不适合归类到某个具体的领域对象中，领域服务用来协调跨多个对象的操作，**无状态**。 比如在`OrderPayByService`中，代支付需收取1%手续费；

```
public void pay(Account account, Order order) {
    account.payby(order.getId(), order.getPrice()); // Account领域模型中实现1%业务逻辑
    order.paid();
    paymentGateway.pay(account.getId(), order.getPrice);
}
```

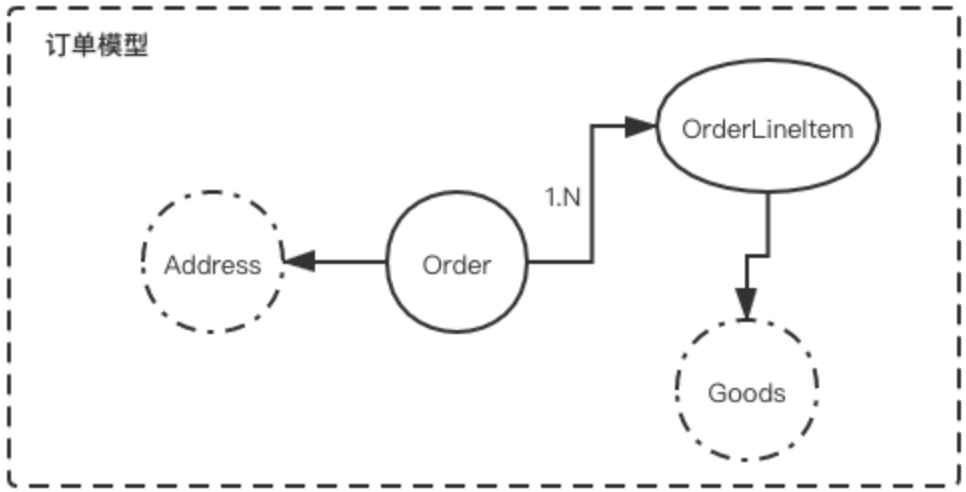
`OrderApplicationService`

```
@Transactional
public void orderPayBy(long orderId, long payByAccount) {
    Order order = DAO.findOrderById(id);
    Account account = DAO.findAccountById(payByAccount);
    orderPayByService.pay(account, order);
    DAO.saveOrUpdate(order, account);
}
```

领域(Domain)层: 业务逻辑的载体

核心: 识别对象之间的内在的关系，构建领域对象。

聚合(Aggreate): 定义了一组具有内聚关系的相关对象的集合: (1)修改数据的单元; (2)业务逻辑的载体。由聚合根，实体及值对象组成；
聚合根(Aggreate Root): 是集合的根节点，可被外界独立访问，具有独立的生命周期。



实体(Entity) vs 值对象(Value Object)

	聚合根	实体	值对象
有无标识(id)	有	有	无
是否只读	否	否	是
是否有生命周期	有	有	无
相等条件	对象标识符	对象标识符	对象属性
示例	Order	OrderLineItem	Goods, Address

聚合根的设计

1. 聚合用来封装不变性(Invariants) (业务规则)， 而不是简单对象从属关系
   帖子及回复关系？ 公司与部门之间关系?

```
public class Post extends AggregateRoot {
    private string title; private List<Reply> replies;
}
```

2. 聚合应尽量小
3. 聚合之间通过ID关联，而不是对象
4. 聚合内强一致性，聚合之间最终一致性

Order领域模型

```
public class Order { // Order聚合根
    private Long id;
    private Long totalPrice;
    private Status status;
    private List<OrderLineItem> items;  // 实体对象 1:N关系
    private Address address; // 值对象
}

public class OrderLineItem {  // 订单商品对象: entity
    private Long id; private Long orderId; private Goods goods;  private Integer count;
}

public class Goods { // 商品对象: value object, 只读
    private Long id; private String name; private String desc; private Long price;
}

public class Address {  // 送货地址: value object, 只读
    private String country; private String province; private String area; private String detail;
}
```

聚合根的创建 - Factory模式

> 1. 直接在聚合根中实现Factory方法，常用于简单的创建过程
> 2. 独立的Factory类，用于有一定复杂度的创建过程，或者创建逻辑不适合放在聚合根上

```
public static Order create(Long id, List<OrderLineItem> items, Address address) {
    return new Order(id, items, address);
}
```

```
public class OrderFactory {
    private OrderIdGenerator idGenerator;

    public Order create(List<OrderLineItem> items, Address address) {
        long orderId = idGenerator.get();
        return Order.create(orderId, items, address);
    }
}
```

基础设施层(Infrasture): 聚合根的家

仓储(`Repository`)为聚合根提供查询及持久化机制；类似`JPA`，可以看成聚合的容器。

- `Repository`与聚合根一一对应；
- `DAO`直接与表数据进行交互，比较薄。

```
@Repository
public class OrderRepository {

    public Order findById(Long id) {    }

    public Order save(Order order) {    }
}
```

模型到表映射

- 实体，聚合根与表一一对应
- 值对象通常与聚合根一起，是表的一列

对于上述的`Order`聚合根来说，可以映射到以下2张表中(`t_order_item`和`t_order`):

列名	类型	含义	对象映射
item_id	bigint	订单项目id	OrderLineItem.id
order_id	bigint	订单id, 外键	OrderLineItem.orderId; Order.id
goods	varchar	商品信息, json格式	OrderLineItem.goods
count	int	商品数量	OrderLineItem.count

`t_order`表

列名	类型	含义	对象映射
order_id	bigint	订单id	Order.id
user_id	bigint	用户id	Order.userId
total_price	bigint	订单总金额	Order.totalPrice
address	varchar	地址，json格式	Order.address
status	tinyint	订单状态	Order.status

代码结构

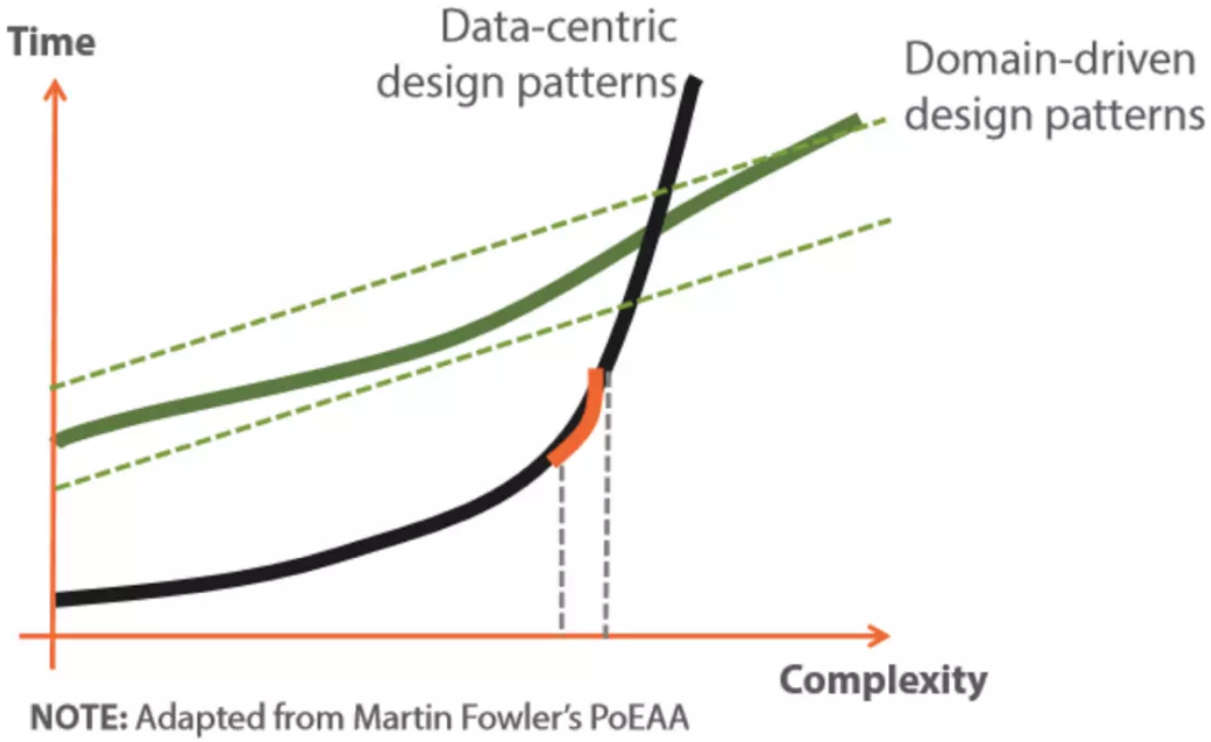
```
├── OrderDddApplication.java
├── ddd
│   ├── application
│   │   └── OrderApplicationService.java
│   ├── domain
│   │   ├── Address.java
│   │   ├── Goods.java
│   │   ├── Order.java
│   │   └── OrderLineItem.java
│   ├── factory
│   │   ├── OrderFactory.java
│   │   └── RedisOrderIdGenerator.java
│   ├── repository
│   │   ├── OrderLineItemPoDao.java
│   │   ├── OrderPoDao.java
│   │   ├── OrderRepository.java
│   │   └── po
│   └── ui
│       └── OrderController.java
```

Git: https://github.com/JiangWork/order-ddd-sample

DDD为什么有效

有效的边界划分,限定职责范围

但划分边界是件有难度的事，随着对业务深入的了解，划分也会不同。



总结


push state change with
DOMAIN EVENTS
access with
REPOSITORIES
access with
SERVICES
express model with
express change with
ENTITIES
encapsulate with
act as root of
express identity with
AGGREGATES
encapsulate with
express state & computation with
VALUE OBJECTS
encapsulate with
MODEL-DRIVEN DESIGN
model gives structure to
isolate domain expressions with
encapsulate with
LAYERED ARCHITECTURE
encapsulate with
FACTORIES
define model within
UBIQUITOUS LANGUAGE
cultivate rich model with
keep model unified by
CONTINUOUS INTEGRATION
names enter
BOUNDED CONTEXT
SHARED KERNEL
SHARED KERNEL
CORE DOMAIN
interdependent contexts form
overlap allied contexts through
relate allied contexts as
CUSTOMER /SUPPLIER
work in autonomous, clean
assess/overview relationships with
CONTEXT MAP
minimize translation
CONFORMIST
avoid overinvesting in
support multiple clients through
segregate the conceptual messes
free teams to go
OPEN HOST SERVICE
GENERIC SUBDOMAINS
translate and insulate unilaterally with
loosely couple contexts through
formalize
BIG BALL OF MUD
ANTI-CORRUPTION LAYER
SEPARATE WAYS
PUBLISHED LANGUAGE

Thanks