领域驱动设计与开发架构

-- 从宏观到微观

自我介绍

Workshop · 工作坊 mpd

FUJITSU

享佳健康 SHAREJOY

- **15年+ 大型软件产品架构经验**

MIZUHO

瑞穗银行 (*Mizuho Bank, Ltd.*)
服务器软件故障自动侦测与分析系统 (SIRMS)

JPX TOKYO STOCK EXCHANGE

东京证券 (*Tokyo Stock Exchange, Inc.*)
富士通数据库 Symfoware 加密连接系统 (ECG)

ISUZU

五十铃汽车 (*ISUZU Motors, Ltd.*)
车载系统云计算解决方案 SaaS (ITP)

- **10年+ 技术讲师经验**

擅长 企业级系统架构，领域驱动设计，重构与模式
深圳证券交易所，中国电信，中国平安，招商银行信用卡中心，中兴通讯 技术讲师
南京大学，东南大学，南瑞集团 特约技术讲师

Agenda

宏观 - 战略建模

- 通用语言
- 领域
- 界限上下文 / 上下文映射
- 架构

微观 - 战术建模

- 实体 / 值对象
- 应用服务 / 领域服务
- 领域事件
- 聚合
- 资源库

宏观 - 战略建模

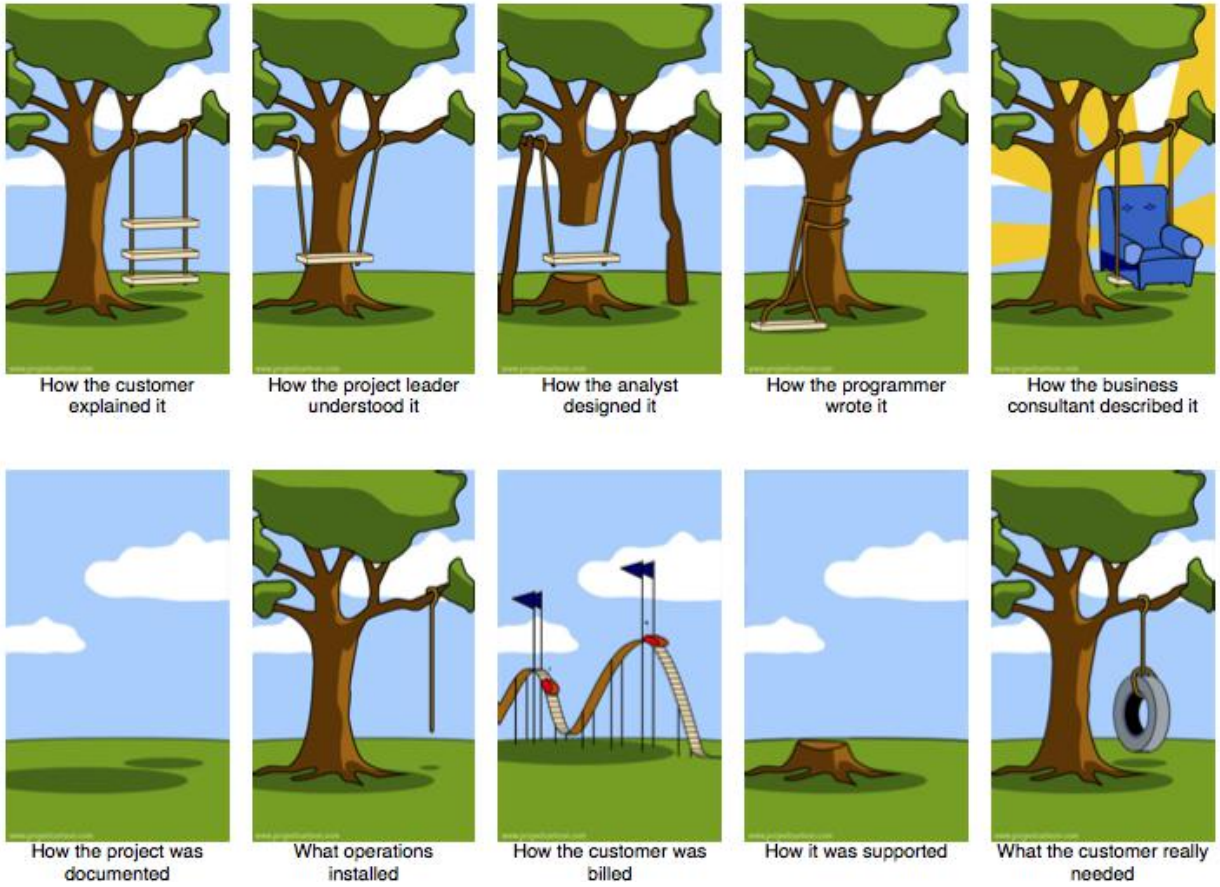
精准地划分领域以及
处理各个领域之间的关系

通用语言

(Ubiquitous Language)

沟通失调

软件开发最大的问题便是业务与技术人员需要某种翻译才能交流



工欲善其事必先利其器

通用语言应对沟通失调

通用语言用于消除 领域专家 与 开发者 之间的沟通失调

通用语言特性：

- 准确 通用 高效
- 不包含 任何 IT 术语
- 包含 术语 用例

UML 的 Use Case 图
是通用语言的一个简单工具

举个栗子

领域专家：请实现一下“高端客户优惠”这个需求

领域专家　　开发人员

通用语言关键点

Workshop · 工作坊 MPD

5W2H1E

Why	为了什么目的
What	要做什么事情
Who	谁会与此相关
When	什么时间做
Where	在什么地方做
How	怎么做
How much	需要什么资源
Exception	例外情况

前置条件 & 后置条件

Pre-Condition	必须的前提
Post-Condition	应该的结果 (与 Why 对应)

通用语言演习

Workshop · 工作坊 MPD

关于“高端客户优惠”的通用语言关键点

Why：	为了提升平台销售、刺激高端客户消费
What：	优惠指根据特定要素组合享受指定折扣
Who：	高端客户指金卡及以上级别客户
When：	促销时段可配置，精确到分钟
Where：	移动App端、Web端都可享受
How：	根据客户级别与订单商品，匹配特定折扣
How much：	开发成本 20人天
Exception：	特价商品不参加优惠
Pre-Condition：	优惠活动配置项处于激活状态
Post-Condition：	预计平台销售额增长 8% ~ 13%

通用语言演习

关于“高端客户优惠”的通用语言描述

为了提升平台销售、刺激高端客户消费，系统可以在指定时间段(精确到分钟)，为移动App端、Web端的金卡及以上客户提供消费折扣。

折扣额度根据客户级别与订单中商品的组合来匹配指定的折扣算法。折扣算法另附。

此优惠需要对应优惠活动配置项处于激活状态，才可生效。另外，特价商品不参加此优惠活动。

开发预算 20人天，激活期间预计平台销售额增长 8% ~ 13%

通用语言需要持续完善

通用语言的代码表达

通用语言包含“术语”和“用例场景”

“用例场景”可以转化为 应用服务 代码

此优惠需要对应优惠活动配置项处于激活状态，才可生效。

```
Promotion promotion = _promotionRepository.FindPromotionById(promotionId);

double amount = 0?

if (promotion.IsActive()) {
    Discount discout = _discountManager.FindMatchDiscount(promotionId,
                       customerGrade, orderId);

    amount = discout.CalculateAmount();
}
```

体现代码即设计的思想

通用语言实战

领域专家：提出一个你工作中的包含领域概念的 简单 需求

开发人员：和领域专家讨论出描述需求的通用语言

通用语言实战

Why :

What :

Who :

When :

Where :

How :

How much :

Exception :

Pre-Condition :

Post-Condition :

通用语言总结

Workshop · 工作坊 MPD

以领域专家的语言为基础

+

开发人员的逻辑性 (归纳 + 演绎)

+

专有时间

“高端客户优惠”还有什么可能的推演?

领域

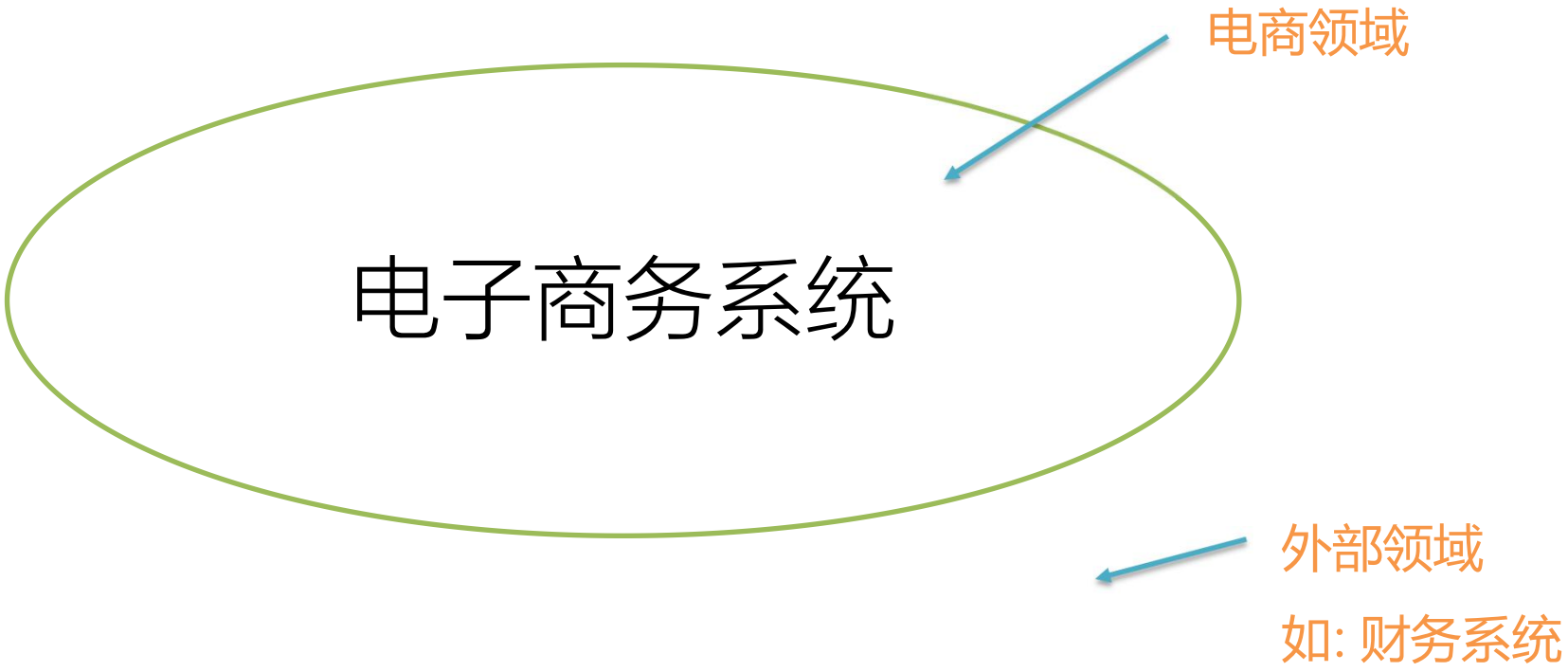
(Domain)

领域是什么

领域一词重在范围与界限

在软件开发里，就代表着要解决的 问题的范围与界限

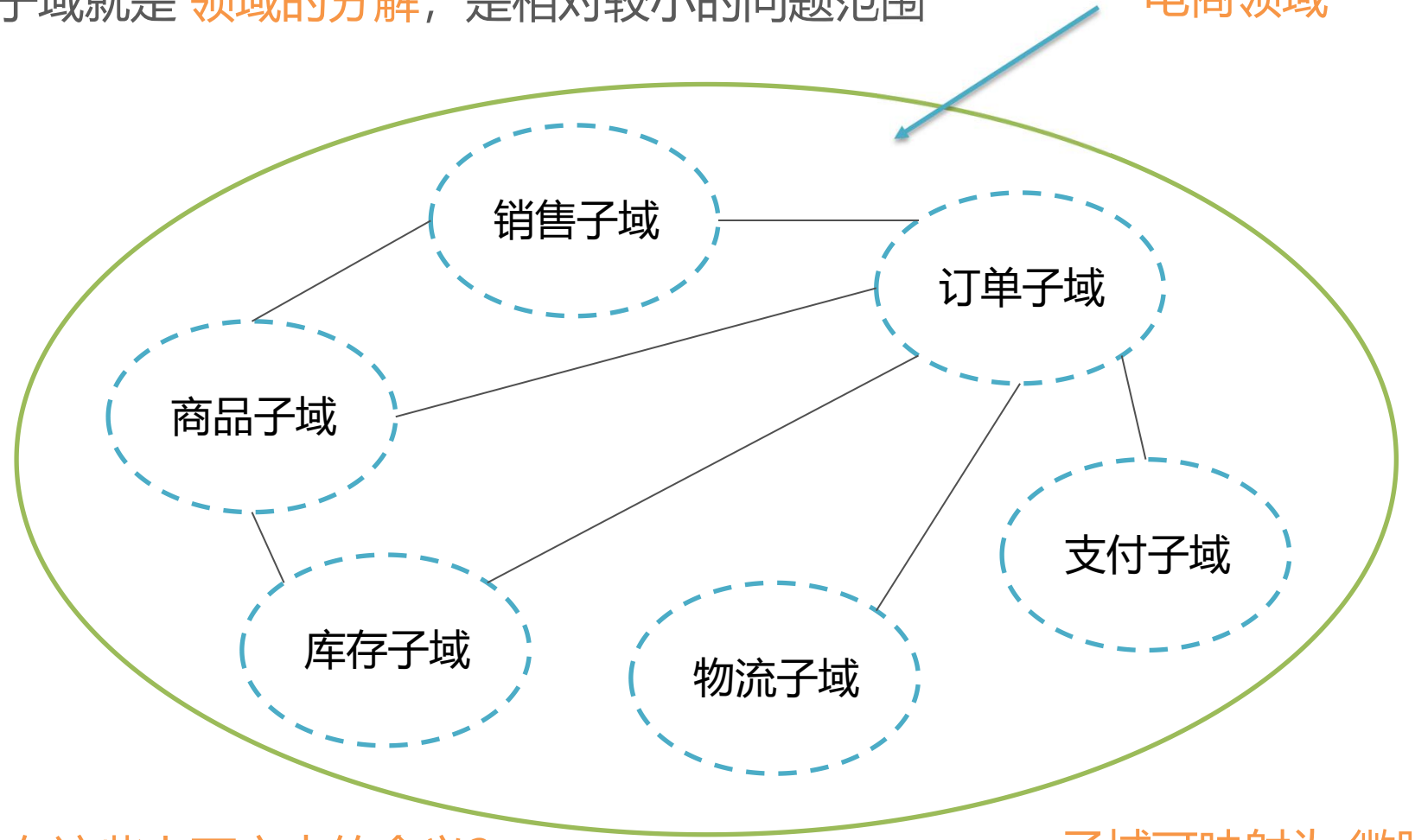
如：电子商务系统就是一个领域



子域是什么

分而治之 是人们应对复杂问题的通用方法

子域就是 领域的分解，是相对较小的问题范围



顾客在这些上下文中的含义?

子域可映射为 微服务

单一职责原则 (Single Responsibility Principle)

Workshop · 工作坊 MPD

就一个软件实体而言，应该仅有一个引起它变化的原因

There should never be more than one reason for a software entities(classes, modules, functions, etc.) to change.

Just because you can doesn't mean you should

核心域 支撑域 通用域

核心域：核心竞争力所在的子域

如：销售子域 (因人而异)

支撑域：支持核心域的子域

如：除销售子域以外的其他子域

通用域：服务于整个领域的的子域

如：除销售子域以外的其他子域

根据 SRP 对要解决的问题拆分细化

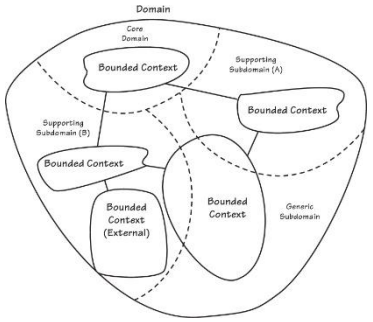
一句话总结领域

Workshop · 工作坊 MPD

领域 = 问题的范围

界限上下文

(Bounded Context)



领域与界限上下文

Workshop · 工作坊 MPD

领域 = 问题的范围

界限上下文 = 问题的解决方案

界限上下文是什么

DDD中与领域对应的概念是 界限上下文

理论上一个子域，对应一个界限上下文

界限上下文的含义：

一个单词或句子所出现的环境，此环境会反过来影响它们的含义。

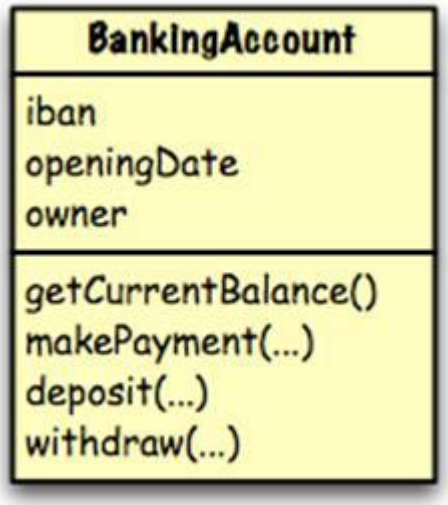
“我想静静”　　“静静是谁？”

“静静”这个概念有歧义，有两个与上下文相关的含义

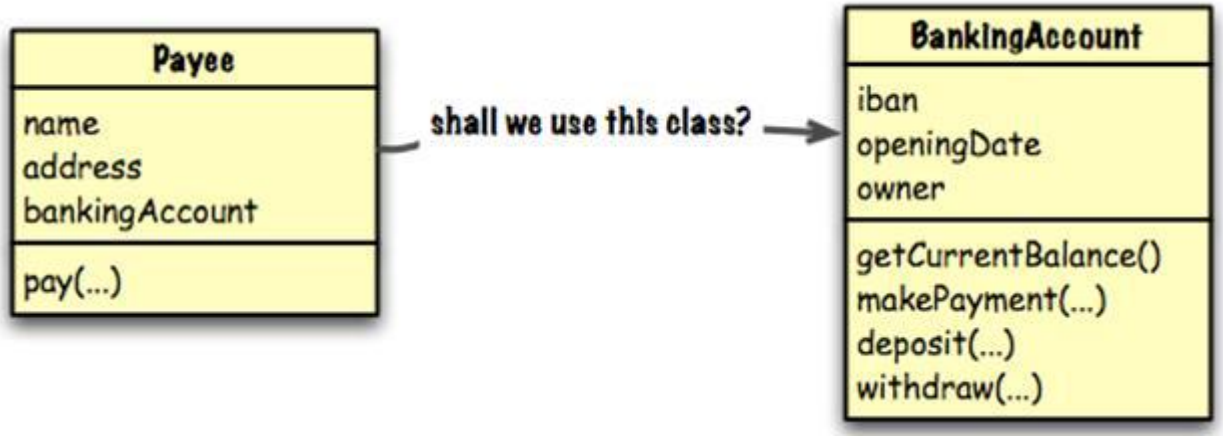
界限上下文最重要的作用之一就是 消除歧义

概念歧义：概念相同，用法不同

支付宝里的 银行账户 这个概念，
可以用右侧精简版的模型表示



在 给朋友转账 的场景里，收款人(Payee)类是否该关联上述 银行账户



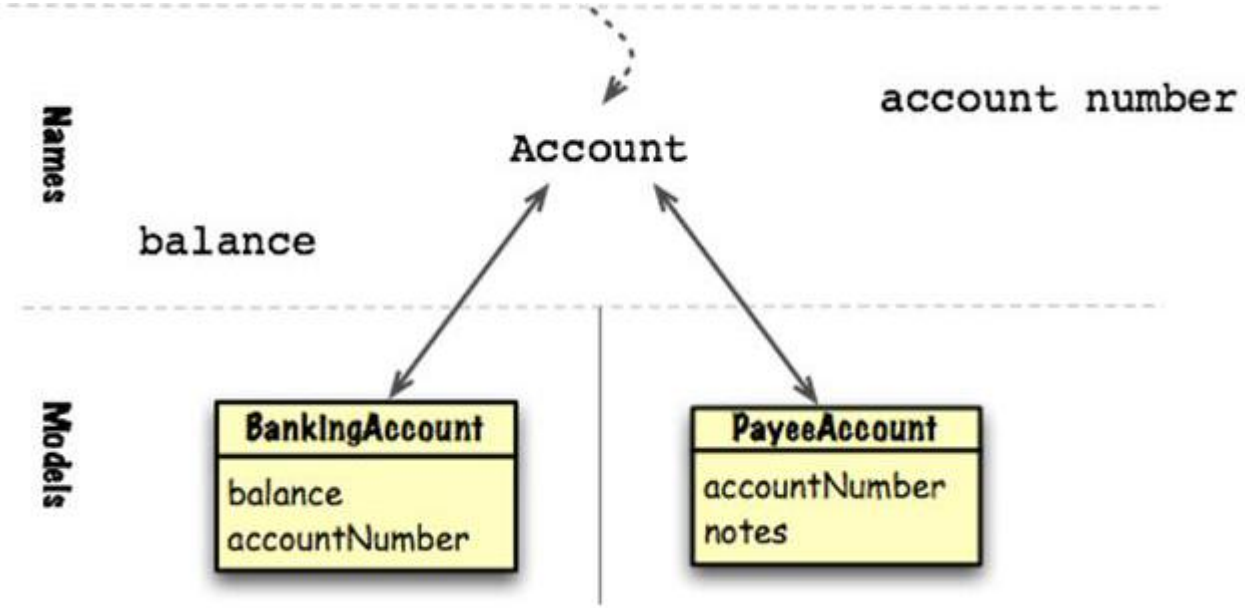
概念歧义：概念相同，用法不同

朋友的银行账户 也是 银行账户，概念相同，理应可以复用

但复用之后，就可以在代码中对收款人的账户做任何操作

银行账户，这个 领域概念 有着两种明显不同的使用场景

每一种都需要一种单独的类型

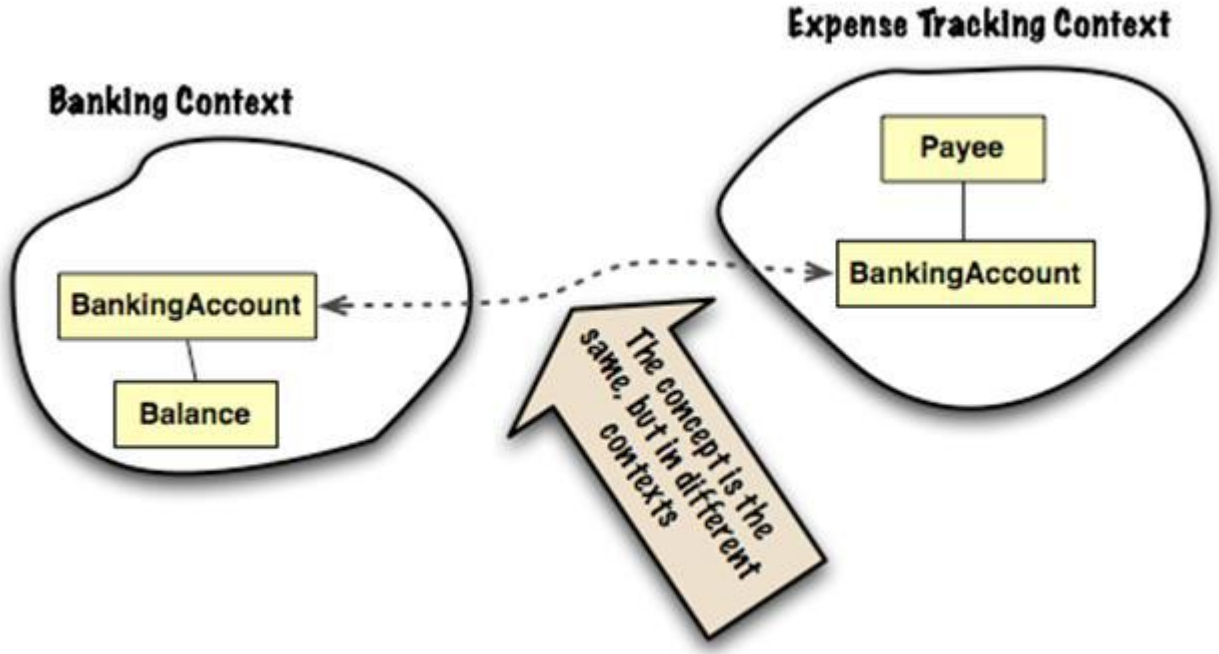


隔离有“二义性”的概念

应该有 银行账户上下文 以及 开销跟踪上下文

这样 BankingAccount 类甚至可以 相同名称 在不同的上下文中出现

但代表 不同的含义



外部系统交互也需要独立的上下文

交流时间

举一个你工作中，有二义性领域概念的例子

以及如何用界限上下文进行二义性隔离

图书(出版/销售)

用户(前台/后台)

Less is More

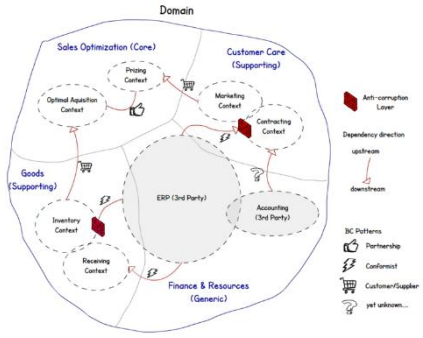
Workshop · 工作坊 MPD

“完美设计不是包罗万象无所不有，而是完整自治不可精简"

-- Antoine de Saint-Exupery

上下文映射

(Context Mapping)



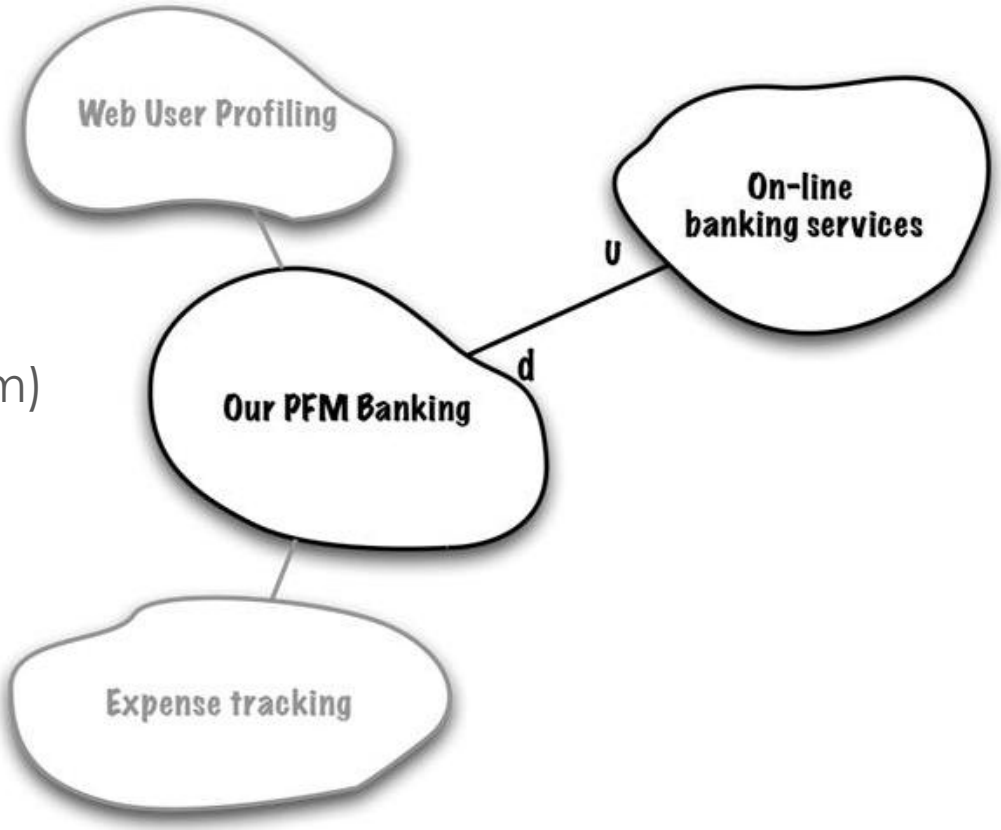
上下文映射

当应用程序跨越多个上下文后，需要管理 上下文之间的关系

上下文映射便是描述上下文之间关系的工具

两个上下文之间的关系是 有方向的

- 依赖发起方叫下游 (Downstream)
- 被依赖方叫上游 (Upstream)
- 上游会影响到下游



上下文映射 - 防腐层

上下游之间的集成关系为 直接依赖

可以使用 防腐层 (Anti Corruption Layer) 来隔离上游上下文的变化

防腐层可以在两个上下文之间

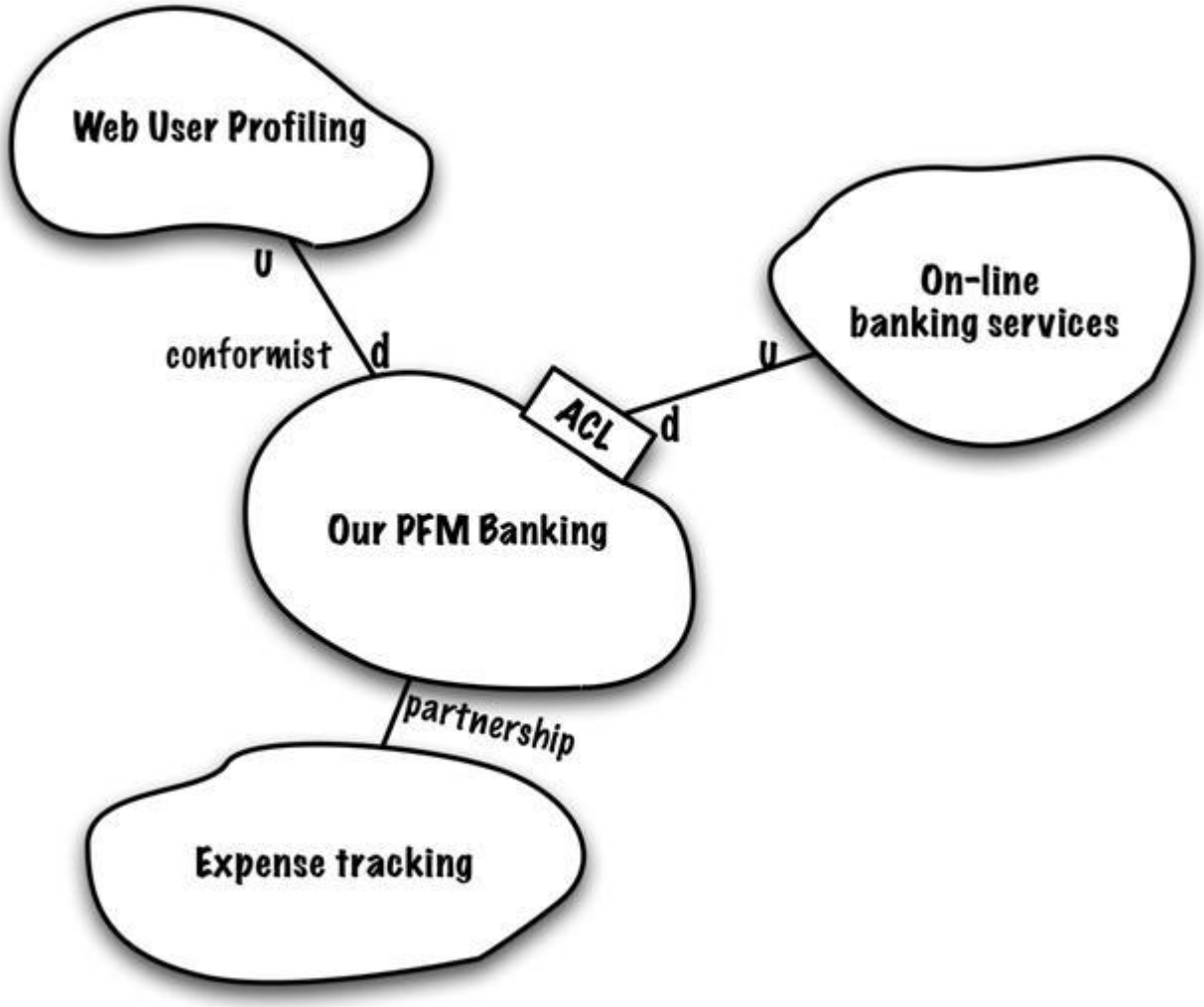
在代码层面上实现显式的转换

可以使用 适配器 模式实现防腐层

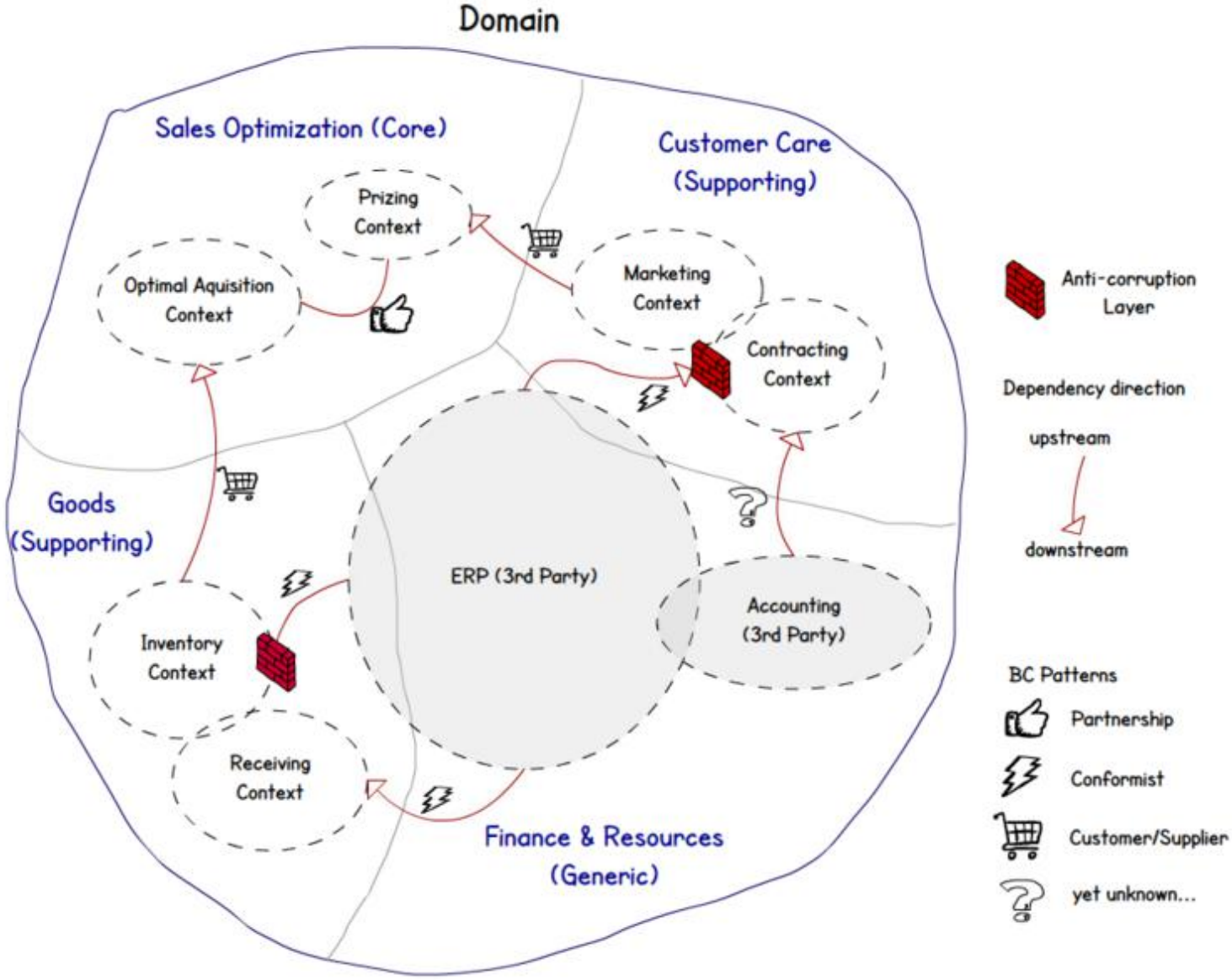
上下文映射 - 常见集成关系

- 合作关系 (Partnership): 上下文之间紧密结合，共生共荣
- 共享内核 (Shared Kernel): 上下文之间共享数据或者代码
- 消费供应 (Customer-Supplier): 上游的计划中，会顾及到下游的需求
- 遵奉者 (Conformist): 下游团队只能盲目地使用上游团队模型
- 防腐层 (Anticorruption Layer): 在上下文之间进行翻译转换
- 开放主机服务 (Open Host Service): 通过公开协议(如 REST)让子系统访问
- 发布语言 (Published Language): 上下文之间通过公共语言交流(如 Json)
- 另谋他路 (Separate Way): 上下文之间不存在任何关系
- 大泥球 (Big Ball of Mud): 系统中混杂在一起的模型

上下文映射 - 常见集成关系



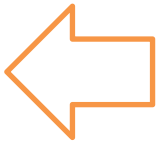
上下文映射图 – 更完整例子



一句话总结

Workshop · 工作坊 MPD

通用语言

领域 

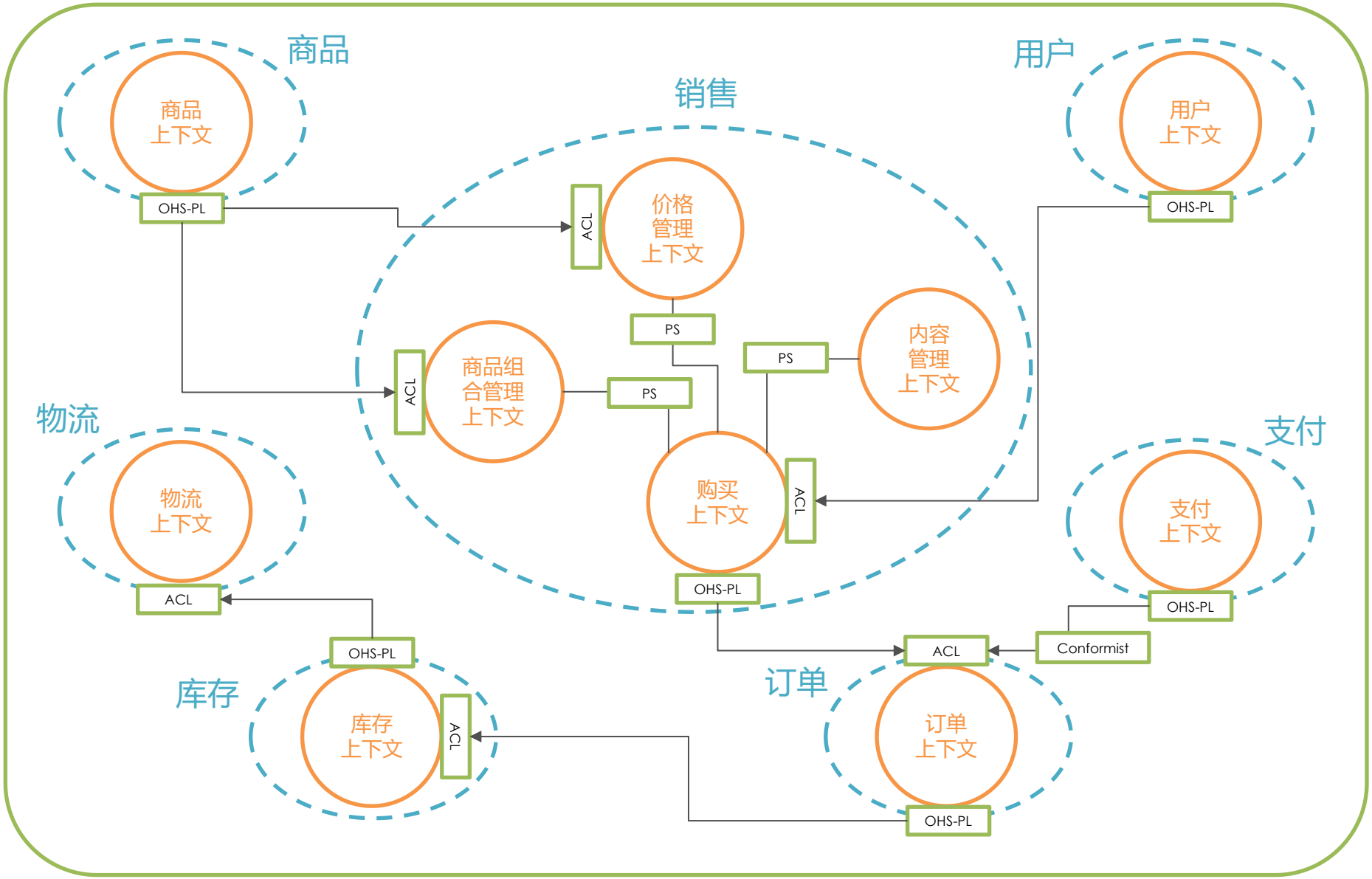
上下文映射

界限上下文

更清晰的宏观建模

电商系统界限上下文映射实战

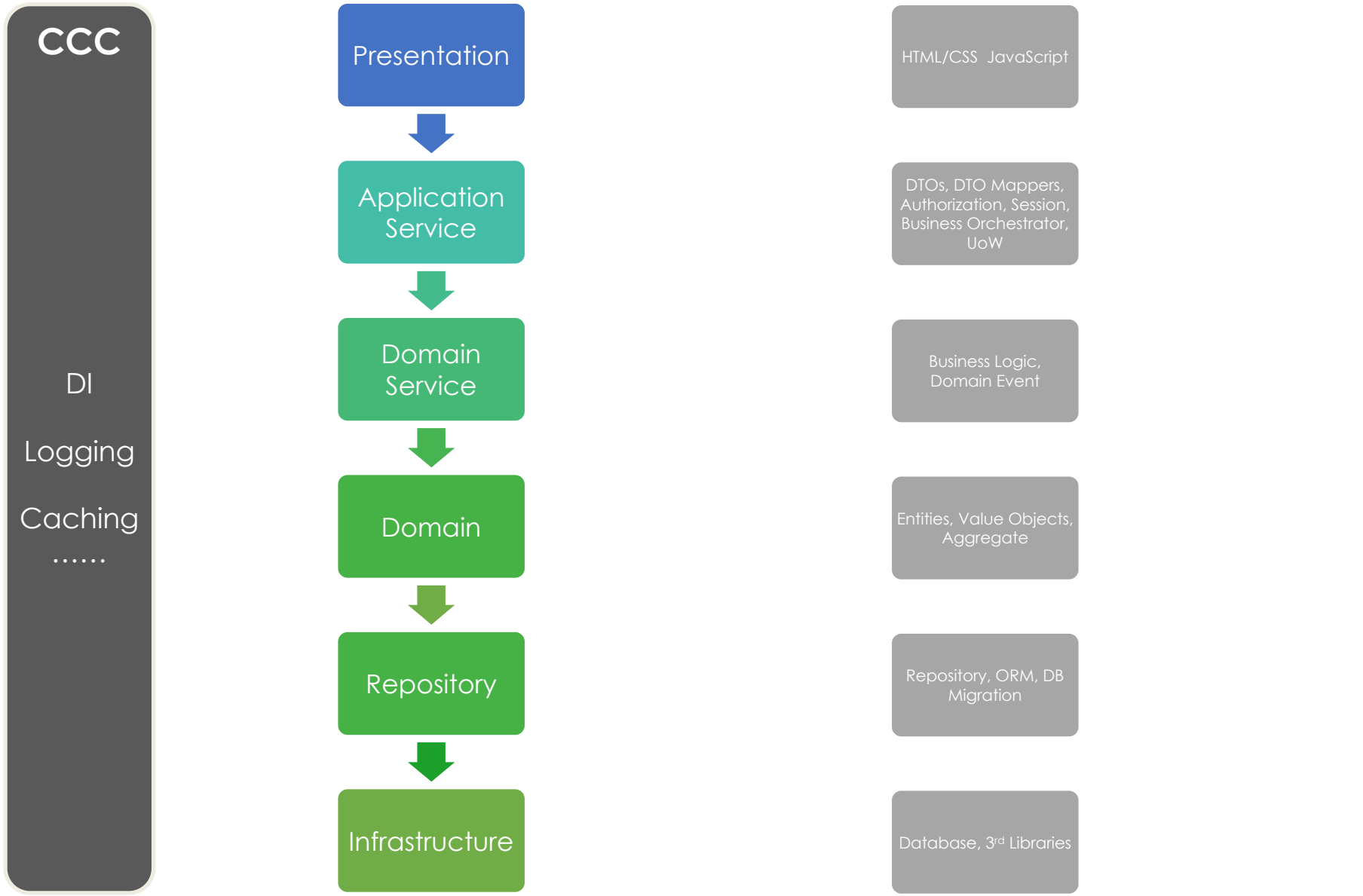
电商系统界限上下文实战



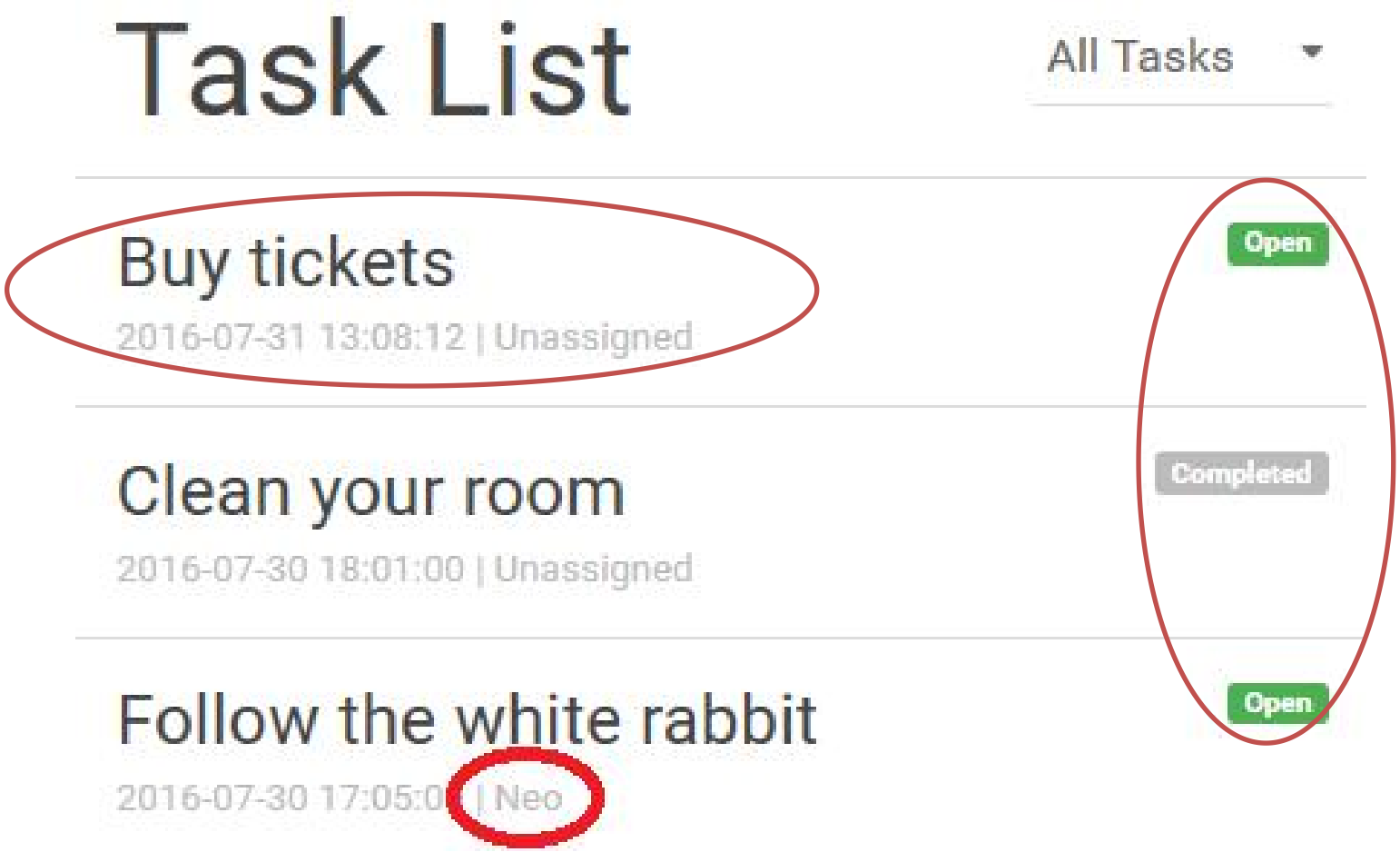
架构

(Architecture)

分层架构



Sample



Entity

```
public class Task : Entity {
    public string Title { get; set; }

    public string Description { get; set; }

    public DateTime CreationTime { get; set; }

    public TaskState State { get; set; }

    public Task(string title, string description = null)
        : this() {
        Title = title;
        Description = description;
        CreationTime = Clock.Now;
        State = TaskState.Open;
    }

    public Open() { State = TaskState.Open; }

    public Close() { State = TaskState.Open; }
}
```

Application Service

```
public class TaskAppService : TaskAppAppServiceBase, ITaskAppService {
    private readonly IRepository<Task> _taskRepository;

    public TaskAppService(IRepository<Task> taskRepository) {
        _taskRepository = taskRepository;
    }

    public async Task<ResultDto<TaskListDto>> GetAll(GetAllInput input) {
        var tasks = await _taskRepository
                    .GetAll()
                    .WhereIf(input.State.HasValue,
                             t => t.State == input.State.Value)
                    .OrderByDescending(t => t.CreationTime)
                    .ToListAsync();

        return new ListResultDto<TaskListDto>(
                    ObjectMapper.Map<List<TaskListDto>>(tasks)
                );
    }
}
```

Domain Service

```
public class TaskManager : DomainService, ITaskManager {
    private readonly ITaskRepository _taskRepository;

    public TaskManager(ITaskRepository taskRepository) {
        _taskRepository = taskRepository;
    }

    public void AssignTaskToPerson(Task task, Person person) {
        if (task.AssignedPersonId == person.Id) {
            return;
        }

        if (task.State != TaskState.Active) {
            throw new ApplicationException("Task is not active!");
        }

        if (HasPersonMaximumAssignedTask(person)) {
            throw new UserFriendlyException(L("MaxLimitMessage",
person.Name));
        }

        task.AssignedPersonId = person.Id;
    }
}
```

Repository

```
public class TaskRepository : RepositoryBase<Task, long>, ITaskRepository {
    public TaskRepository(IDbContextProvider<DbContext> dbContextProvider)
        : base(dbContextProvider) {
    }

    public List<Task> GetAllWithPeople(int? personId, TaskState? state) {
        var query = GetAll();

        if (assignedPersonId.HasValue) {
            query = query.Where(task => task.Person.Id == personId.Value);
        }

        if (state.HasValue) {
            query = query.Where(task => task.State == state);
        }

        return query
               .OrderByDescending(task => task.CreationTime)
               .Include(task => task.AssignedPerson)
               .ToList();
    }
}
```

依赖倒置原则 (Dependency Inversion Principle)

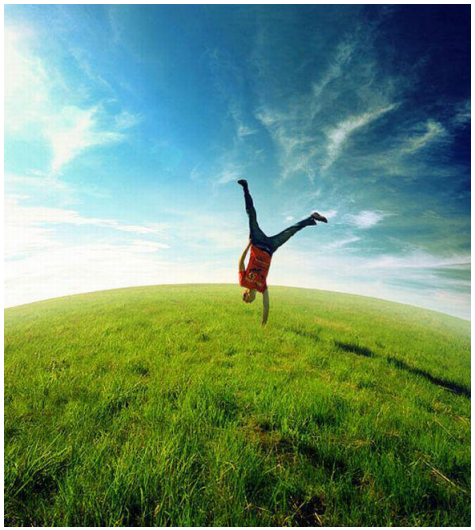
Workshop · 工作坊 MPD

高层模块不应该依赖于低层模块，它们都应该依赖于抽象

High-level modules should not depend on low-level modules. Both should depend on abstractions.

抽象不应该依赖于细节，细节应该依赖于抽象

Abstractions should not depend upon details. Details should depend upon abstractions.

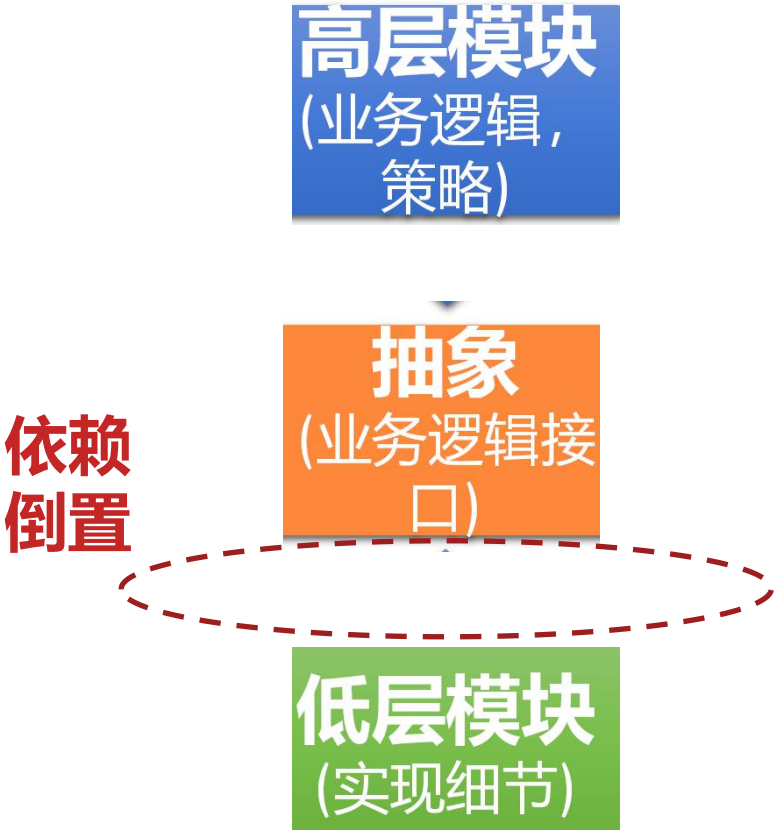


Oops! Upside down!

DIP：对象化的设计理念

面向对象的分析和设计：

- 高层模块不应该依赖于低层模块，它们都应该依赖于抽象
- 抽象不应该依赖于细节，细节应该依赖于抽象



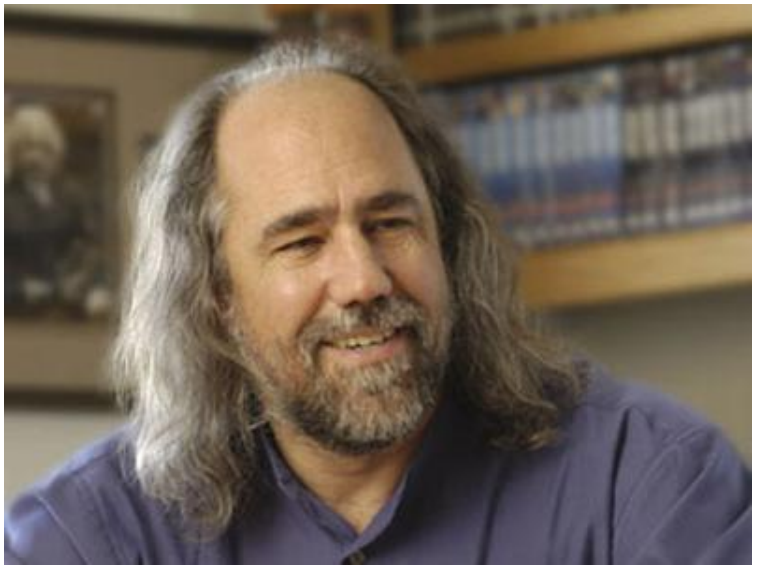
倒置了对低层模块的依赖

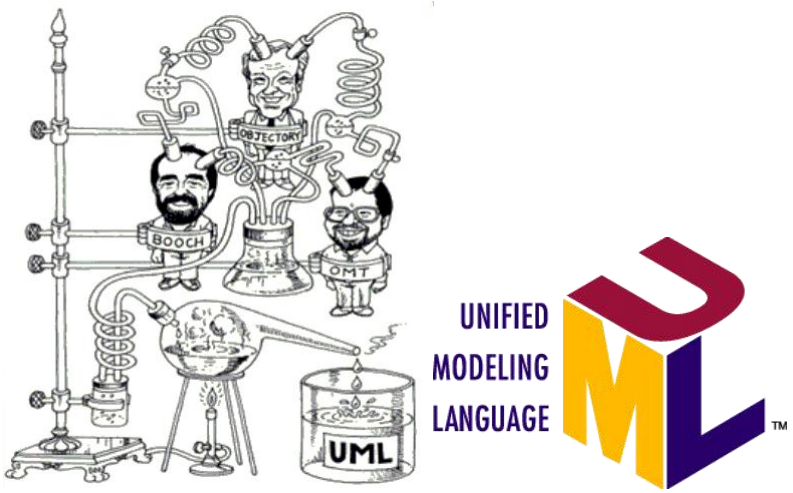
DIP：依赖倒置的关键 -- 分层

所有结构良好的面向对象架构拥有清晰的分层，每个分层通过良好定义的接口提供条理分明的一系列服务

All well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface.

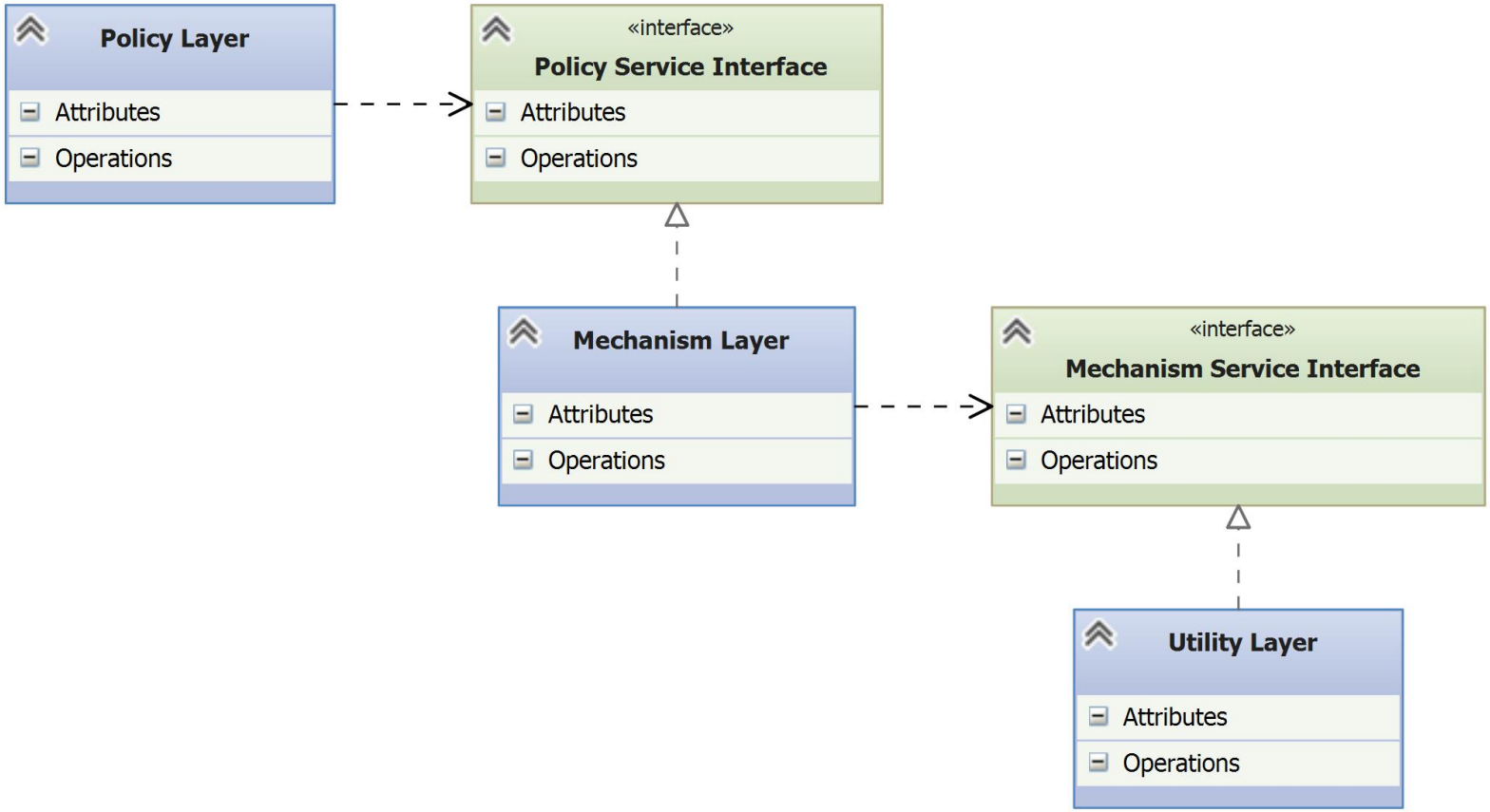
-- Grady Booch



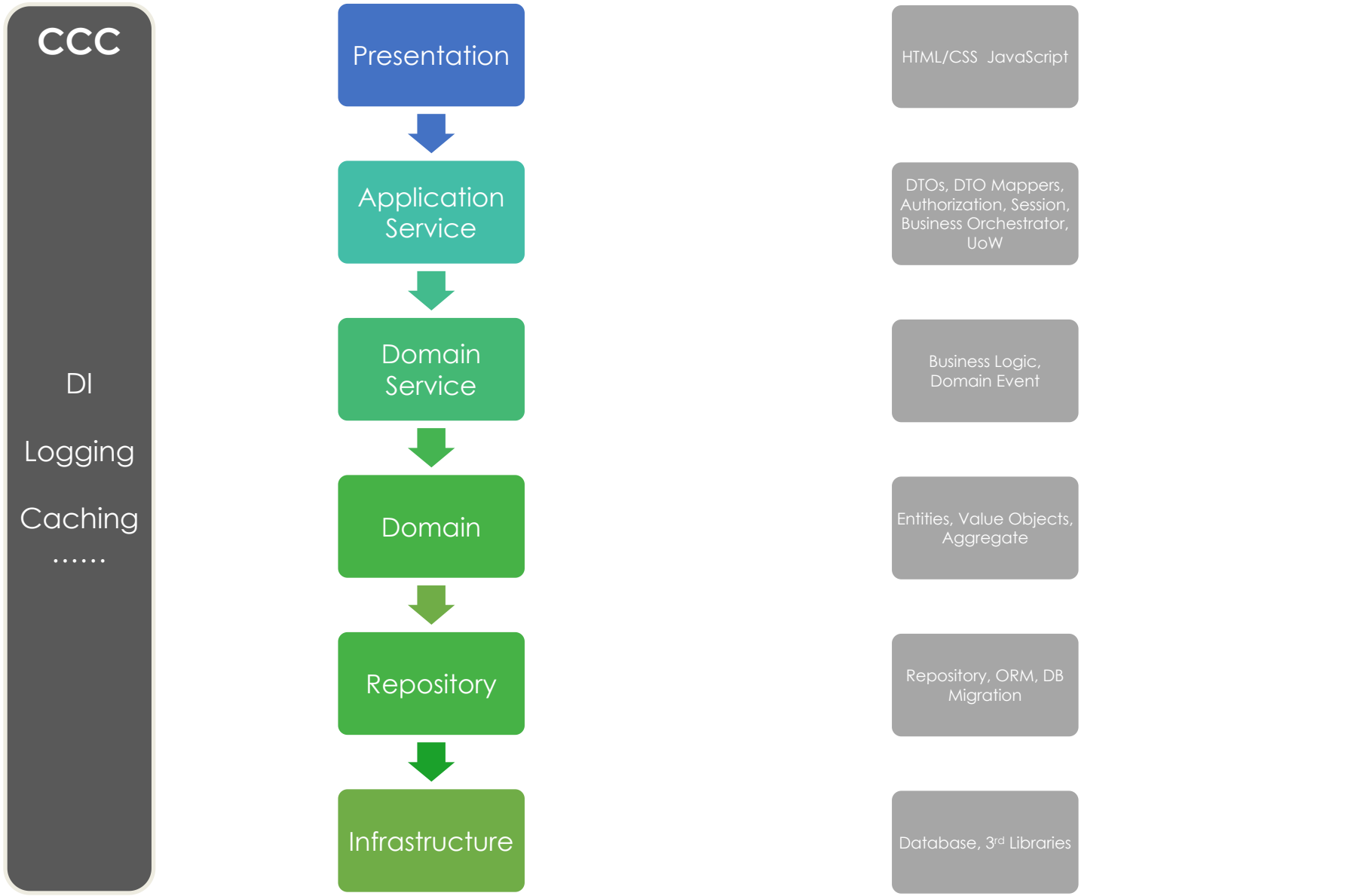


DIP：更好的层次化

- 每个较高的层次都为它所需要的服务声明一个抽象接口
- 每个高层类都通过该抽象接口使用下一层
- 较低的层次则实现这些抽象接口



分层架构

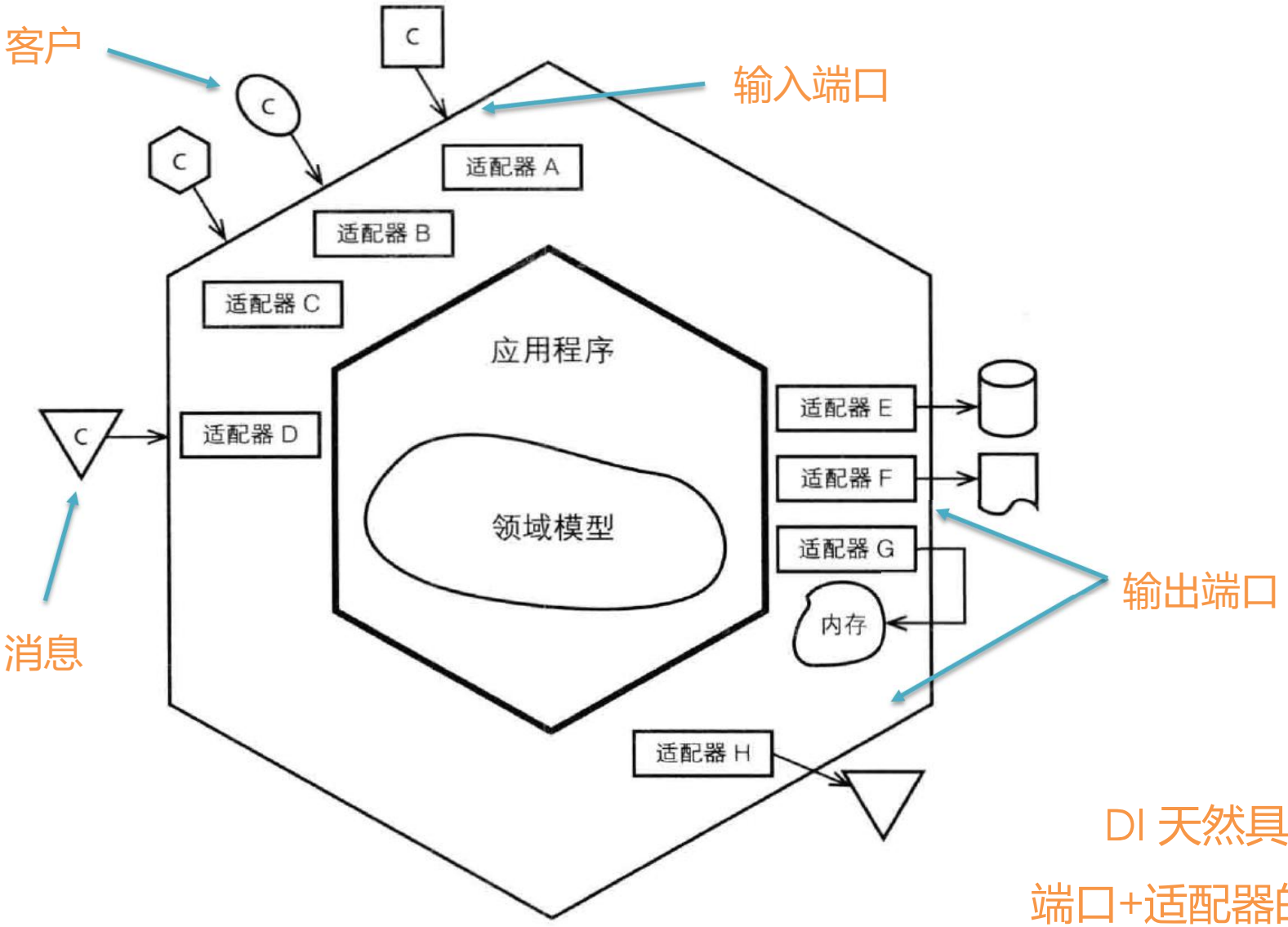


被推平的分层

依赖倒置把分层推平了

如果再加上对称性呢?

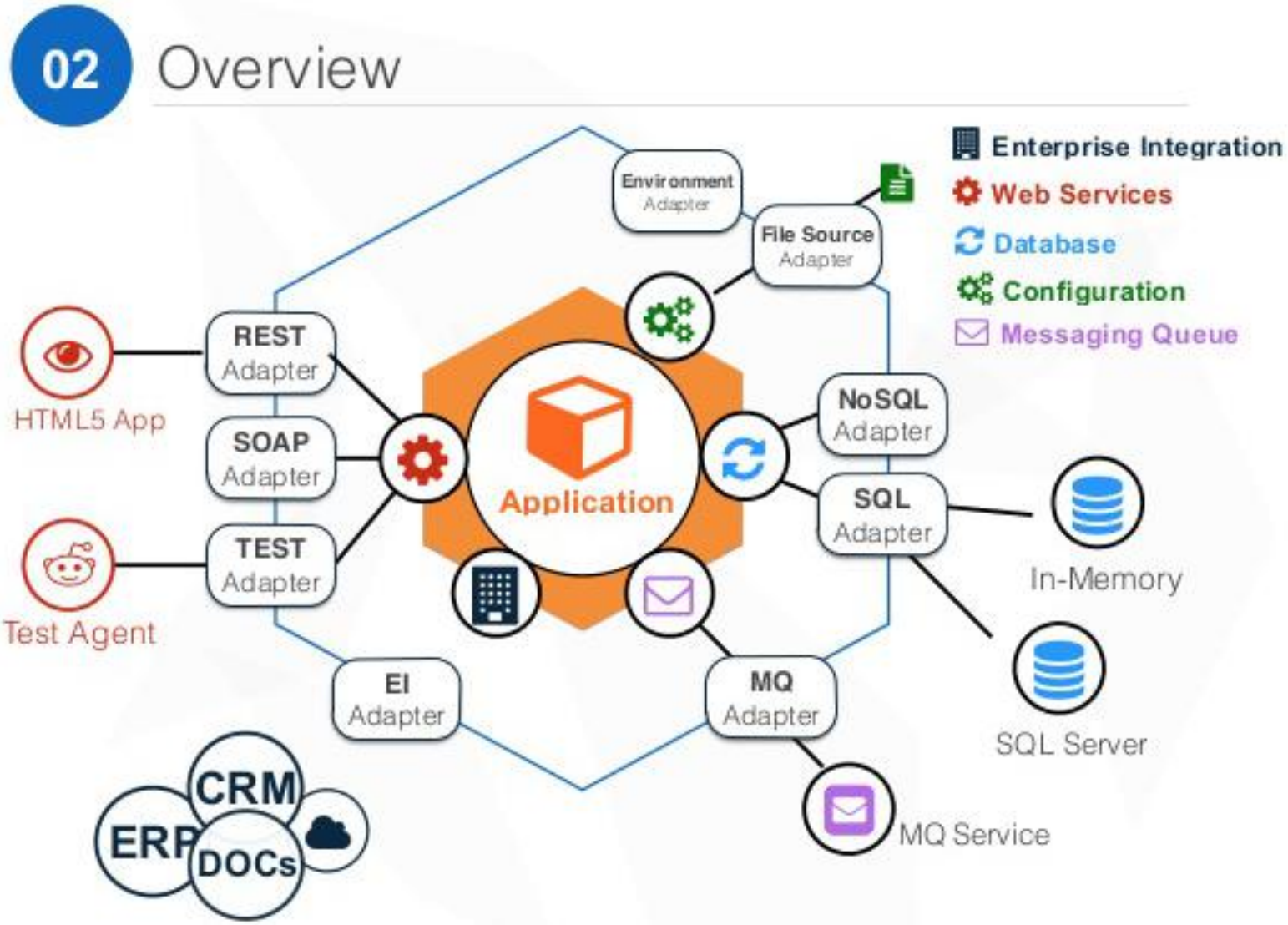
六边形架构



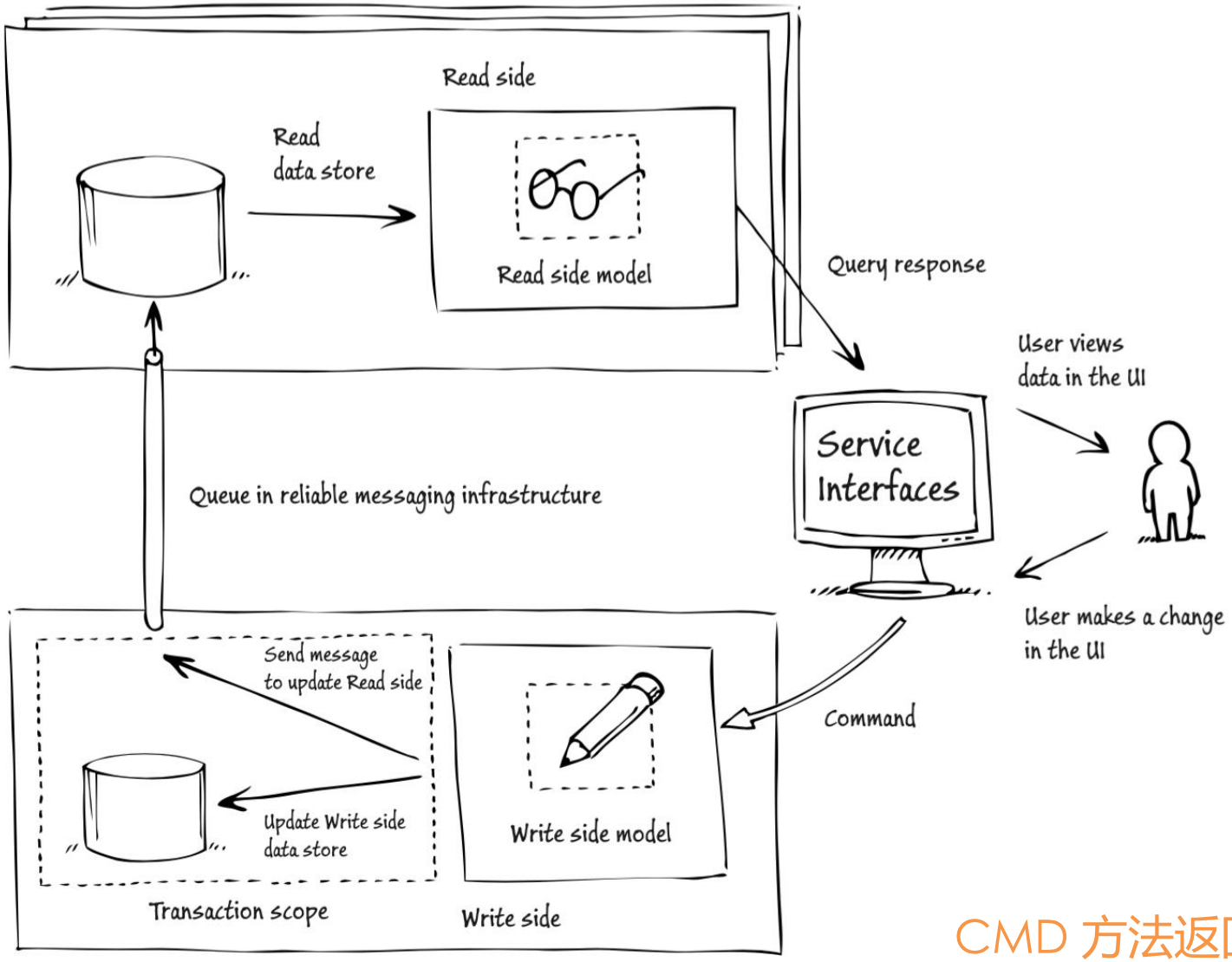
DI 天然具有

端口+适配器的风格

六边形架构



CQRS 架构



CMD 方法返回值?

宏观建模总结

Workshop · 工作坊 MPD

定好语言工具

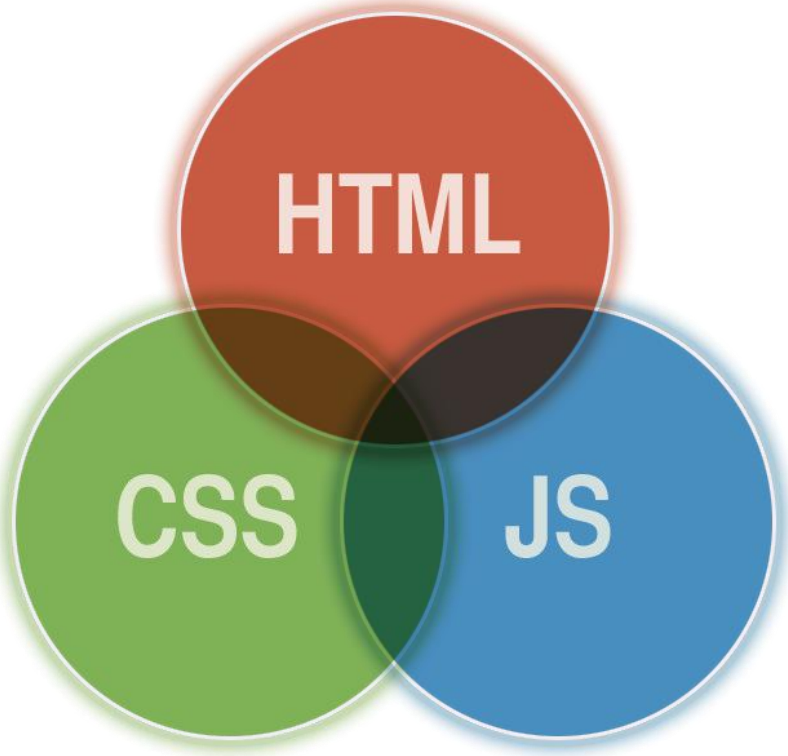
描述领域问题

细分解决方案

选择合适架构

Workshop · 工作坊 mpd

分离关注点

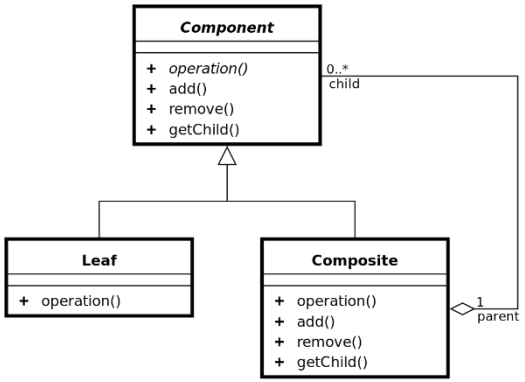


微观 - 战术建模

从技术实现层面如何具体地实施 DDD

实体

(Entity)



两种 DDD 与 实体

Workshop · 工作坊 MPD

Data - Driven - Development

关注点放在 数据 上，数据库建模优先 (Db First)

实体只是数据库模型的转换结果和 数据的容器

容易导致 贫血模型 (代码可读性也低)

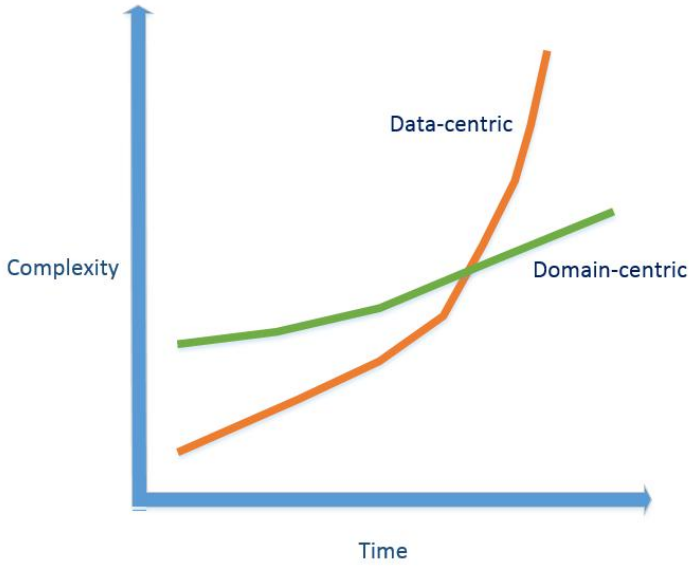
Domain - Driven - Design

关注点放在 领域概念 上

实体建模优先 (Code First)

实体是 服务的提供方

数据库表只是实体的持久化容器



实体的特征

Workshop · 工作坊 MPD

实体 =

唯一标识 + 可变性 (属性、行为)

唯一标识

DDD中要求实体是 唯一 的且 可持续变化

唯一性由唯一的 身份标识 来决定的

值对象 可以作为实体的唯一标识

唯一标识类型

- 简单系统：int 类型的的自增 Id ，简单高效
- 复杂系统 无意义标识：guid 类型，空间占用大，查询速度稍慢
- 复杂系统 有意义标识：string 类型，生成标识算法是以一种挑战

唯一标识生成时机

- 即时生成：持久化之前，由业务系统生成
- 延时生成：持久化时，由持久化组件生成 (Repository)

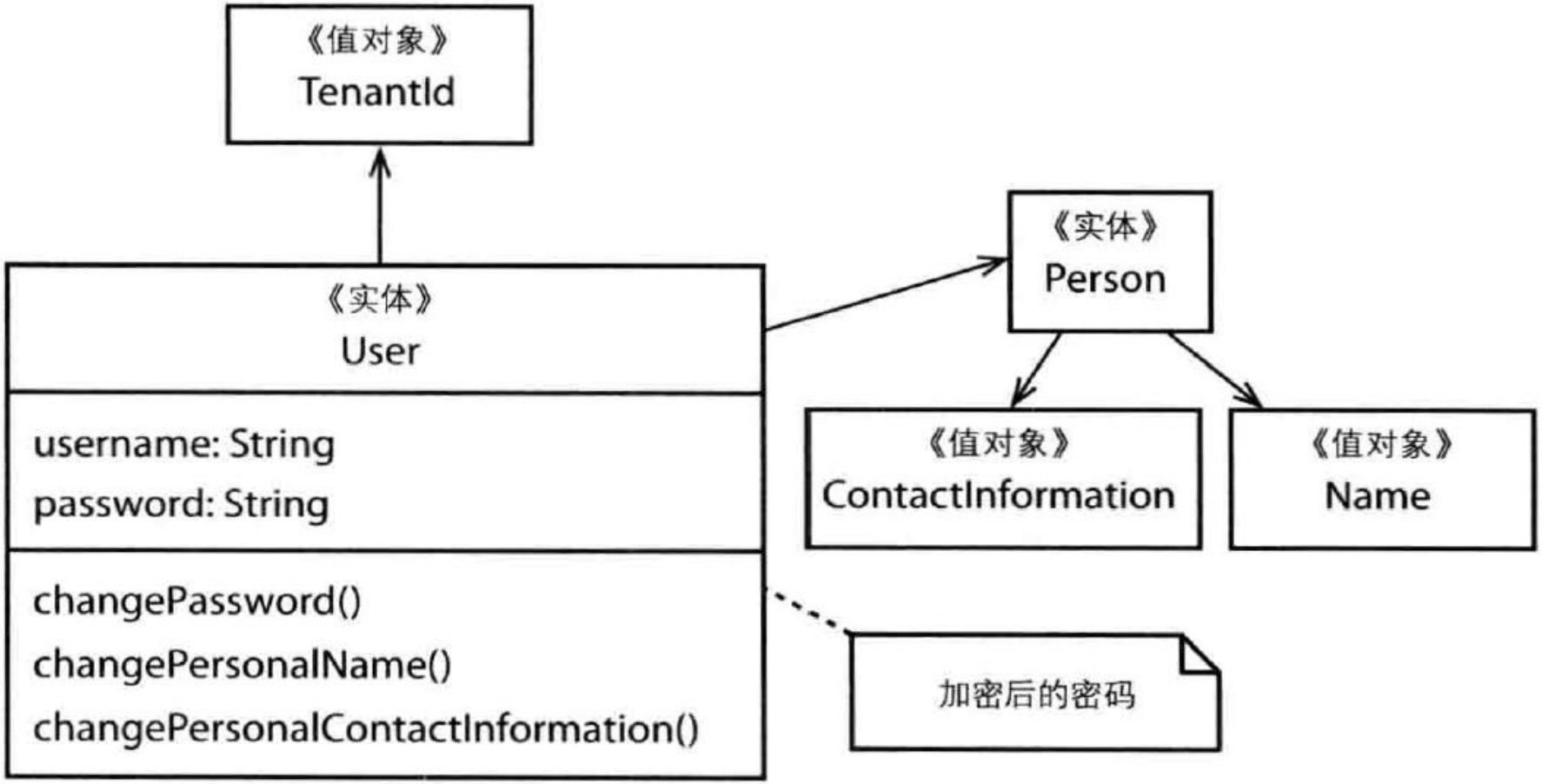
委派标识

基于领域实体概念确定的唯一身份标识，称为 领域实体标识

使用 ORM 工具时，它们有自己的方式来处理对象的身份标识
此时，委派标识 表现为 int 或 long 类型的实体属性，作为数据库主键
委派标识和领域实体标识可以无任何关系

需要创建一个称为层超类型 (Layer Supertype) 的实体基类来统一指定委派标识

实体图



实现层超类型

定义层超类型接口与缺省层超类型

```
public interface IEntity
{
}

public interface IEntity<TPrimaryKey> : IEntity
{
    TPrimaryKey Id { get; set; }
}
```

```
public class Entity : Entity<int>, IEntity
{
}
```

实现层超类型

实现层超类型

```
public abstract class Entity<TPrimaryKey> : IEntity<TPrimaryKey>
{
    public virtual TPrimaryKey Id { get; set; }

    public override bool Equals(object obj);

    public override int GetHashCode();

    public static bool operator == (Entity<TPrimaryKey> left,
                                    Entity<TPrimaryKey> right);

    public static bool operator != (Entity<TPrimaryKey> left,
                                    Entity<TPrimaryKey> right);
}
```

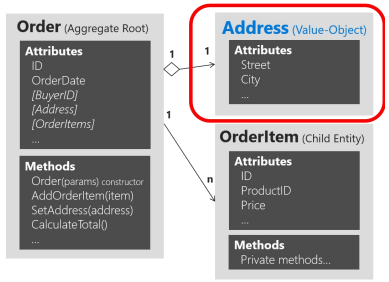
实体必须继承自Entity，即可实现委托标识的统一定义

实体 Sample

```
public class Person : Entity<long>
{
    public virtual string Name { get; set; }

    public virtual DateTime CreationTime { get; set; }

    public Person()
    {
        CreationTime = DateTime.Now;
    }
}
```

值对象

(Value Object)



什么是值对象

值对象 = 值 + 对象

将一个值用对象的方式进行表述，来表达一个具体的固定不变的概念

值对象 不应该 是领域里的一件东西

值对象可以 度量或描述 领域里的实体

值对象创建出来之后就 不能改变 了

> *"A small simple object, like money or a date range, whose equality isn't based on identity."*
>
> *Martin Flower*

值对象的特征

1. 表示一个具体的概念
2. 通过值的属性对其识别
3. 属性判等
4. 固定不变
5. 可以提升系统性能

说说你项目中的值对象

罕见的值对象其实很有威力

Workshop · 工作坊 MPD

数据建模的主要依据是数据库范式设计

数据库范式设计的目标是消除存储在多个位置上的冗余数据，以免导致更新异常

这需要进行不断的表拆分，直到每个表都只表示一个单一的概念

传统的以数据为中心的开发中，数据库表直接映射到实体

值对象的作用：

- 优化系统性能　　(工厂创建单例值对象实例)
- 简化持久化机制　(数据库逆范式设计)

实现值对象

```
public abstract class ValueObject<TValueObject> : IEquatable<TValueObject>
    where TValueObject : ValueObject<TValueObject> {

    public bool Equals(TValueObject other);

    public override bool Equals(object obj);

    public override int GetHashCode();

    public static bool operator == (Entity<TPrimaryKey> left,
                                    Entity<TPrimaryKey> right);

    public static bool operator != (Entity<TPrimaryKey> left,
                                    Entity<TPrimaryKey> right);

    private PropertyInfo[] GetPropertiesForCompare();
}
```

实现值对象

```
public bool Equals(TValueObject other)
{
    if ((object)other == null) {
        return false;
    }

    var publicProperties = GetPropertiesForCompare();

    return publicProperties.All(
                property => Equals(
                    property.GetValue(this, null),
                    property.GetValue(other, null)));
}
```

```
private PropertyInfo[] GetPropertiesForCompare()
{
    return GetType().GetTypeInfo().GetProperties().Where(
      t => ReflectionHelper
        .GetSingleAttributeOrDefault<IgnoreOnCompareAttribute>(t) == null)
        .ToArray();
}
```

值对象 Sample

```
public class Address : ValueObject<Address> {
    public string City { get; private set; }

    public string Street { get; private set; }

    public int Number { get; private set; }

    public Address(string city, string street, int number) {
        City = city;
        Street = street;
        Number = number;
    }
}
```

```
var address1 = new Address("New York", "Baris Manco Street", 42);
var address2 = new Address("New York", "Baris Manco Street", 42);

Assert.Equal(address1, address2);
Assert.Equal(address1.GetHashCode(), address2.GetHashCode());
Assert.True(address1 == address2);
Assert.False(address1 != address2);
```

值对象一句话总结

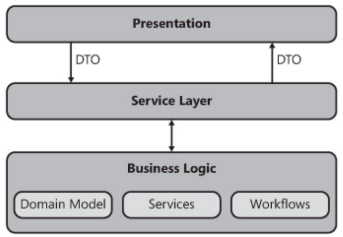
Workshop · 工作坊 MPD

值对象 =

用来描述实体某个属性的不变值

应用服务

(Application Service)



应用服务简介

Workshop · 工作坊 MPD

- 应用服务用来表达 用例 (Use Case)
  一个接口对应展现层中的一个用户操作

- 应用服务本身只并不具有业务逻辑
  只负责 编排协调 领域服务来实现用例

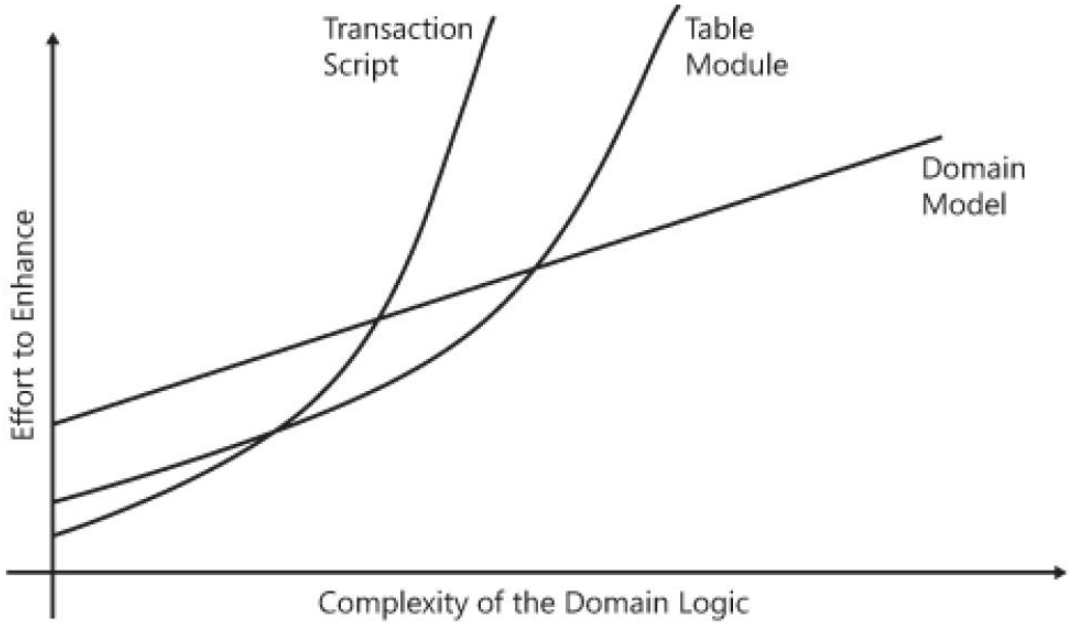
- 应用服务还负责：DTO转换、安全与权限校验、持久化事务控制(UoW)或向其他系统发生基于领域事件的消息通知

- 应用服务是一种 门面模式 (Facade)

BLL 中应该如何组织业务逻辑

业务逻辑层的架构模式

- Transaction Script (事务脚本)
- Table Module (表模块)
- Active Record (活动记录)
- Domain Model (领域模型)



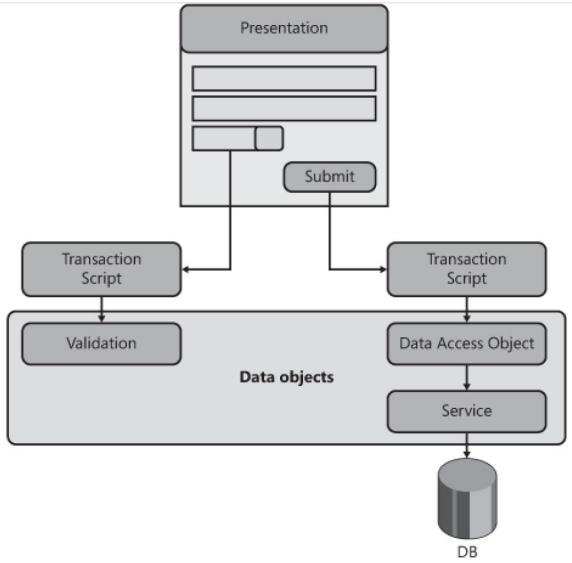
事务脚本介绍

事务脚本模式鼓励放弃所有面向对象设计

事务脚本的关注点：

- 用户通过表现层所能执行的操作
- 为每个操作编写一个专门的方法，这个方法就叫做事务脚本

事务脚本适合业务逻辑非常简单，且不会常变的场景中



TS 示例

```
public class CreateOrderCommand : IApplicationCommand {
    OrderInfo _orderinfo;
    int _orderId;

    public CreateOrderCommand(OrderInfo order) {
        _orderInfo = order;
    }

    public int OrderID { get { return _orderId; } }

    public int Run() {
        // 开始事务
        // 接收并检查产品信息
        // 检查库存中是否有足够的被订购产品
        // 检查信用卡状态
        // 计算价格，包括税费，运费
        // 在 Orders 数据库中添加纪录
        // 在 OrderDetails 数据库中添加纪录
        // 提交事务
        // 获取订单ID，并存储在局部变量 _orderID 中
    }
}
```

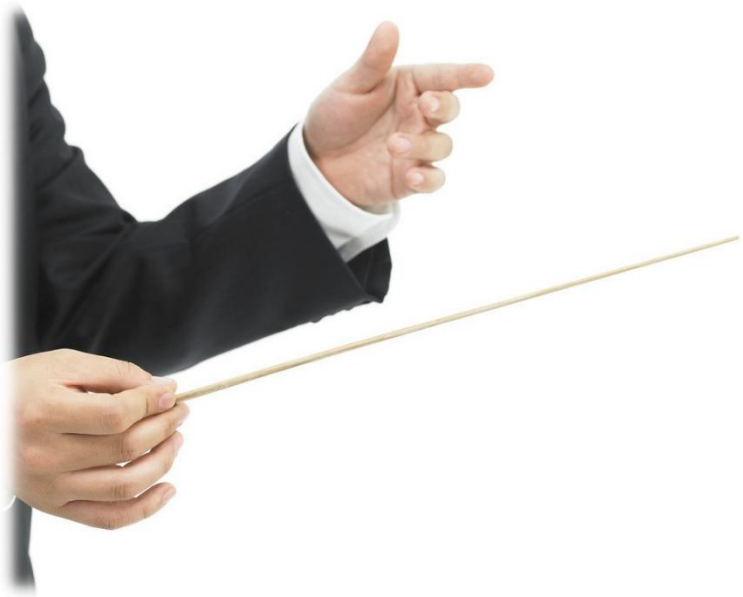
事务脚本 与 应用服务

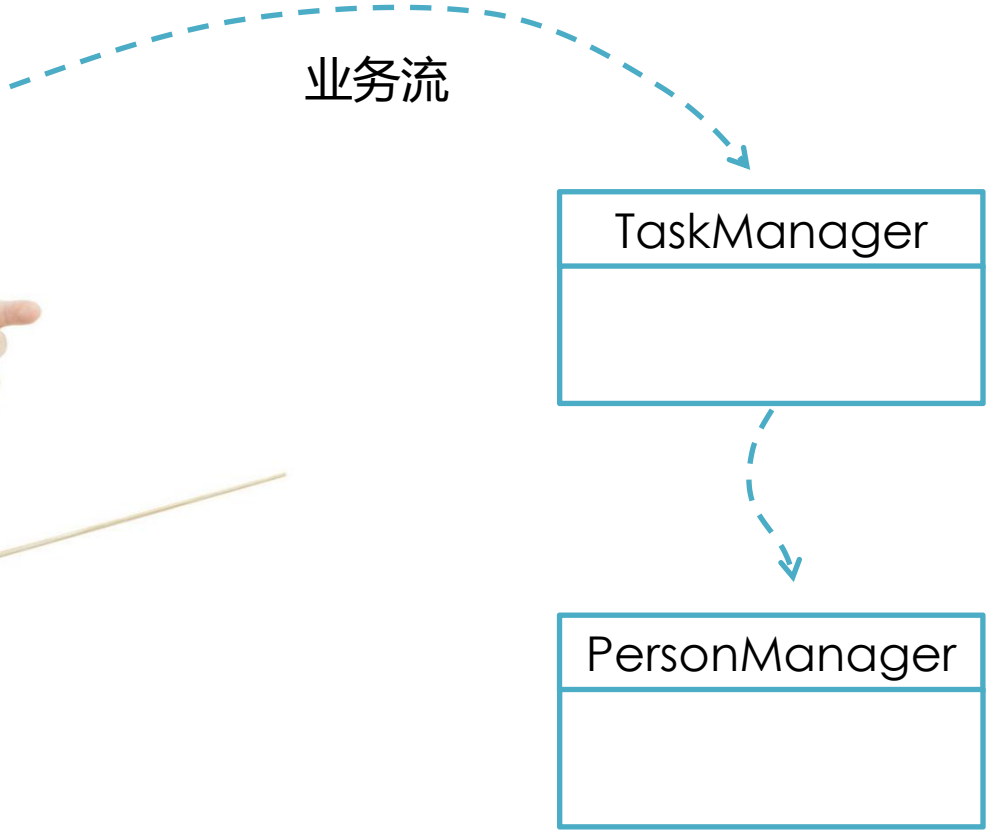
应用服务 =

只有调用顺序

没有业务逻辑的事务脚本

用例的协调者 - 应用服务





Application Service 协调
若干 Domain Service 完成特定业务流

实现应用服务

```
public interface IApplicationService : ITransientDependency
{
}
```

```
public abstract class ServiceBase
{
    public ISettingManager SettingManager { get; set; }

    public IUnitOfWorkManager UnitOfWorkManager { get; set; }

    protected IActiveUnitOfWork CurrentUnitOfWork { get; }

    public ILogger Logger { protected get; set; }

    public IObjectMapper ObjectMapper { get; set; }
}
```

实现应用服务

```
public abstract class ApplicationService : ServiceBase, IApplicationService
{
    public ISession Session { get; set; }

    public IPermissionChecker PermissionChecker { protected get; set; }

    protected virtual bool IsGranted(string permissionName)
    {
        return PermissionChecker.IsGranted(permissionName);
    }
}
```

```
public interface IPersonAppService : IApplicationService
{
    void CreatePerson(CreatePersonInput input);
}
```

实现应用服务

```
public class PersonAppService : IPersonAppService {
    private readonly IRepository<Person> _personRepository;

    public PersonAppService(IRepository<Person> personRepository) {
        _personRepository = personRepository;
    }

    public void CreatePerson(CreatePersonInput input) {
        var person = _personRepository.FirstOrDefault(
                        p => p.EmailAddress == input.EmailAddress);

        if (person != null) {
            throw new UserFriendlyException("Duplicated person");
        }

        person = new Person { Name = input.Name, Email = input.Email };
        _personRepository.Insert(person);

        Logger.Debug("Successfully inserted!");
    }
}
```

如何向一个5岁的小孩解释依赖注入

Workshop · 工作坊 MPD

- When you go and get things out of the refrigerator for yourself, you can cause problems.

- You might leave the door open, you might get something Mommy or Daddy doesn't want you to have.You might even be looking for something we don't even have or which has expired.

- What you should be doing is stating a need, "I need something to drink with lunch," and then we will make sure you have something when you sit down to eat.

http://stackoverflow.com/questions/1638919/how-to-explain-dependency-injection-to-a-5-year-old

为什么要有 DTO

Workshop · 工作坊 MPD

DTO 的作用

- 将 BLL 中的 Entity 与 PL 隔离
- 隐藏敏感信息
- 提升网络传输效率
- 避免序列化中的循环引用问题
- 实现延迟加载

理论上应该为每个应用层的接口定义 DTO

如何在 DTO 与 Entity 之间进行互转

为 Entity 与其对应的每个 DTO 写转换代码是一件很无聊的事情

Mapper 框架

- Dozer (Java)

  https://github.com/DozerMapper/dozer

- AutoMaper (.NET)

  https://github.com/DozerMapper/dozer

Dozer 示例

```
<mapping>
  <class-a>yourpackage.SourceClassName</class-a>
  <class-b>yourpackage.DestinationClassName</class-b>
    <field>
      <A>yourSourceFieldName</A>
      <B>yourDestinationFieldName</B>
    </field>
</mapping>
```

```
SourceClassName sourceObject = ...

Mapper mapper = DozerBeanMapperBuilder.createDefault();
DestinationObject destObject =
    mapper.map(sourceObject, DestinationClassName.class);

assertTrue(destObject.getYourDestinationFieldName()
          .equals(sourceObject.getYourSourceFieldName));
```

AutoMapper 示例

```
Mapper.Initialize(cfg =>
{
    cfg.CreateMap<Foo, FooDto>();
    cfg.CreateMap<Bar, BarDto>();
});
```

```
var fooDto = Mapper.Map<FooDto>(foo);
var barDto = Mapper.Map<BarDto>(bar);
```

权限管理

- 通过属性声明权限

```
[Authorize("UserManagement.CreateUser")]
public void CreateUser(CreateUserInput input) {
    // for users who granted the
    // "UserManagement.CreateUser" permission.
}
```

- 通过代码声明权限

```
public void CreateUser(CreateOrUpdateUserInput input) {
    if (!PermissionChecker.IsGranted("UserManagement.CreateUser")) {
        throw new AuthorizationException("Not authorized!");
    }
}
```

UoW

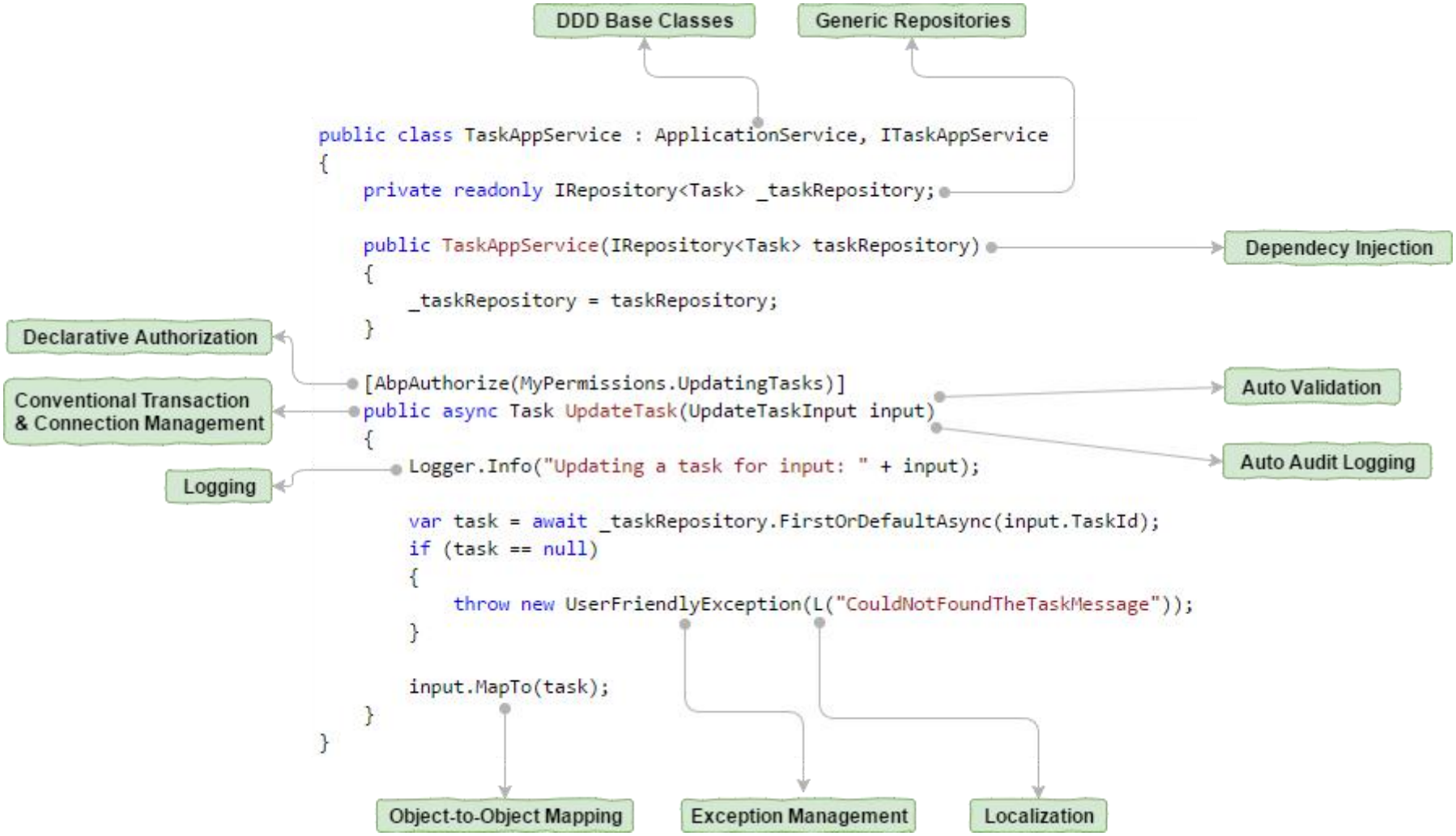
- 通过属性声明UoW

```
[UnitOfWork]
public void CreatePerson(CreatePersonInput input) {
    var person = new Person
        { Name = input.Name, EmailAddress = input.Email };
    _personRepository.Insert(person);
    _statisticsRepository.IncrementPeopleCount();
}
```

- 通过代码声明权限

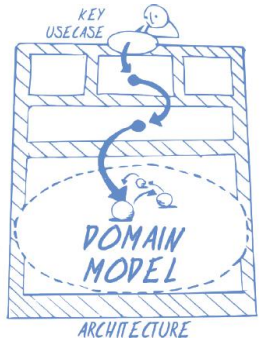
```
public void CreatePerson(CreatePersonInput input) {
    var person = new Person(input.Name, input.Email) };

    using (var unitOfWork = _unitOfWorkManager.Begin()) {
        _personRepository.Insert(person);
        _statisticsRepository.IncrementPeopleCount();
        unitOfWork.Complete();
    }
}
```

应用服务关注点



领域服务

(Domain Service)



什么时候使用领域服务?

- 领域中的某个操作不是实体或值对象的职责时
- 应将该操作放在一个单独的接口中，即领域服务
- 领域服务需要设计成无状态的 (幂等)

实现领域服务

```
public abstract class ServiceBase
{
    public ISettingManager SettingManager { get; set; }

    public IUnitOfWorkManager UnitOfWorkManager { get; set; }

    protected IActiveUnitOfWork CurrentUnitOfWork { get; }

    public ILogger Logger { protected get; set; }

    public IObjectMapper ObjectMapper { get; set; }
}
```

```
public interface IDomainService : ITransientDependency
{
}
```

```
public abstract class DomainService : AbpServiceBase, IDomainService
{
}
```

实现领域服务

```
public interface ITaskManager : IDomainService
{
    void AssignTaskToPerson(Task task, Person person);
}
```

```
public class TaskManager : DomainService, ITaskManager
{
    public void AssignTaskToPerson(Task task, Person person)
    {
        if (task.AssignedPersonId == person.Id)
        {
            return;
        }

        if (task.State != TaskState.Active)
        {
            throw new ApplicationException("Task is not active!");
        }

        task.AssignedPersonId = person.Id;
    }
}
```

应用服务 调用 领域服务

```
public class TaskAppService : ApplicationService, ITaskAppService
{
    private readonly IRepository<Task, long> _taskRepository;
    private readonly IRepository<Person> _personRepository;
    private readonly ITaskManager _taskManager;

    public TaskAppService(IRepository<Task, long> taskRepository,
       IRepository<Person> personRepository, ITaskManager taskManager)
    {
        _taskRepository = taskRepository;
        _personRepository = personRepository;
        _taskManager = taskManager;
    }

    public void AssignTaskToPerson(AssignTaskToPersonInput input)
    {
        var task = _taskRepository.Get(input.TaskId);
        var person = _personRepository.Get(input.PersonId);

        _taskManager.AssignTaskToPerson(task, person);
    }
}
```

应用服务 VS 领域服务

用例相关

业务逻辑相关

Application Service  Domain Service

Domain Service 可被不同的 Application Service 复用

领域事件

(Domain Event)

领域事件简介

- 领域事件作为领域模型的重要部分，是领域建模的工具之一
- 用来捕获领域中已经发生的事情
- 领域事件是领域专家所关心的，一般使用通用语言表达

当用户在购物车点击结算时，生成待付款订单
若支付成功，则更新订单状态为已支付，扣减库存，
并推送捡货通知信息到捡货中心

事务脚本会如何实现这个需求？

单次事务单个根聚合原则

Workshop · 工作坊 MPD

一个业务用例对应一个事务

一个事务对应一个聚合根

一次事务中只对一个聚合根进行操作

需要操作多个聚合的场景很多

Workshop · 工作坊 MPD

在电商网站上买了东西之后，你的积分会相应增加

订单（Order）对象，账户（Account）对象均为聚合根
且分属于订单上下文和用户上下文

如要在订单和积分之间维护数据一致性，则需要在同一个事务中同时更新两者
这会带来如下问题：

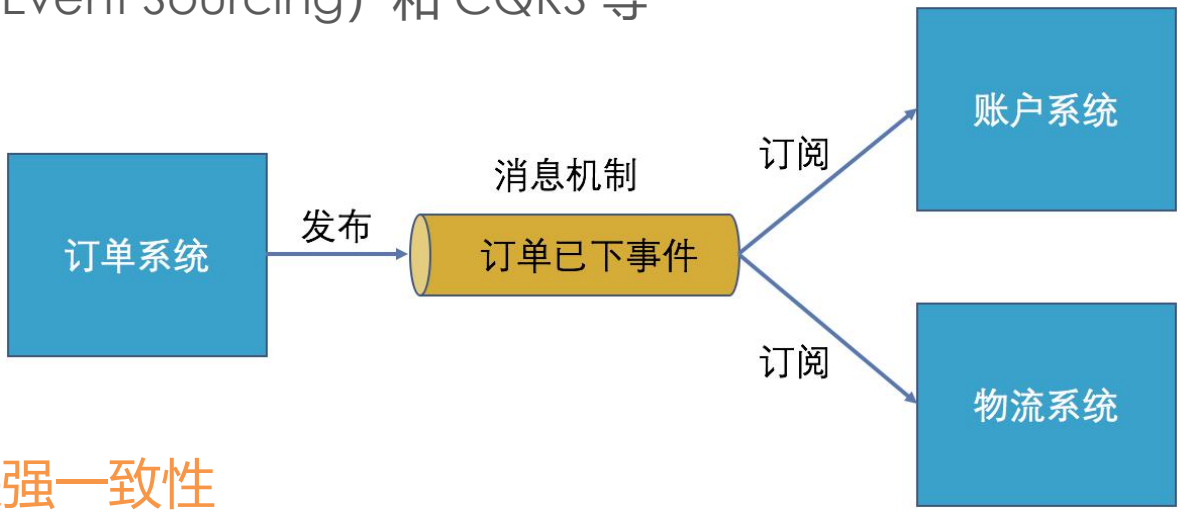
- 高并发下一个长事务无法被接受，锁冲突 频发影响系统性能
- 需要在不同的系统之间采用重量级的 分布式事务（Distributed Transactioin，也叫XA事务或者全局事务）
- 在不同系统之间产生 强耦合

领域事件的好处

通过引入领域事件，我们可以很好地解决上述问题

领域事件给我们带来以下好处：

- 解耦出微服务（限界上下文）；
- 帮助我们深入理解领域模型；
- 提供审计和报告的数据来源；
- 迈向事件溯源（Event Sourcing）和 CQRS 等



各服务之间不再是强一致性

而是基于事件的最终一致性

实现抽象领域事件 (Layer Supertype)

```
public interface IEventData {
    Guid EventGuid { get; private set; }
    DateTime EventTime { get; private set; }
    object EventSource { get; set; }
}
```

```
public abstract class EventData : IEventData {
    public Guid EventGuid { get; private set; }
    public DateTime EventTime { get; private set; }
    public object EventSource { get; set; }

    protected EventData() {
        EventGuid = Guid.NewGuid();
        EventTime = Clock.Now;
    }
}
```

领域事件是不可变的

(Immutable)

```
public class EntityEventData<TEntity> : EventData {
    public TEntity Entity { get; private set; }

    public EntityEventData(TEntity entity) { Entity = entity; }
}
```

实现抽象领域事件

```
public class EntityChangedEventData<TEntity> :
                    EntityEventData<TEntity> {
    public EntityChangedEventData(TEntity entity)
        : base(entity)
    {  }
}
```

```
public class EntityUpdatedEventData<TEntity> :
                    EntityChangedEventData<TEntity> {
    public EntityUpdatedEventData(TEntity entity)
        : base(entity)
    {  }
}
```

```
public class TaskStatusChangedEventData : EntityUpdatedEventData
{
    public int TaskId { get; private set; }
}
```

实现领域事件的发布和订阅

领域事件的发布可以使用 发布--订阅 模式来实现，通常会使用事件总线

```
public interface IEventBus
{
    void Register<TEventData>(IEventHandler<TEventData> handler)
        where TEventData : IEventData;

    void Unregister<TEventData>(IEventHandler<TEventData> handler)
        where TEventData : IEventData;

    void Trigger<TEventData>(object eventSource, TEventData eventData)
        where TEventData : IEventData;
}
```

实现领域事件的发布和订阅

```
public class EventBus : IEventBus {
    public static EventBus Default { get; } = new EventBus();
    private readonly ConcurrentDictionary<Type, List<IEventHandler>>
_handlers;

    public EventBus() {
        _handlers = new ConcurrentDictionary<Type, List<IEventHandler>>();
    }

    public void Register<TEventData>(IEventHandler<TEventData> handler)
       where TEventData : IEventData {
         //TODO: register logic here...
    }

    public void Unregister<TEventData>(IEventHandler<TEventData> handler)
       where TEventData : IEventData {
         //TODO: unregister logic here...
    }

    public void Trigger<TEventData>(object eventSource, TEventData eventData)
       where TEventData : IEventData {
         //TODO: trigger logic here...
    }
}
```

如何发布领域事件 Where&How

发布领域事件

- 聚合根对象的方法中

```
public class Task : Entity {
    private readonly IEventBus _eventBus;
    public Task(EventBus eventBus)
        : this() {
        _eventBus = eventBus;
    }

    public void Complete()
        //TODO: complete the task in the database...
        eventBus.Trigger(new TaskCompletedEventData
                                {TaskId = this.Id});
    }
}
```

API
污染

```
public class Task : Entity {
    public void Complete()
        //TODO: complete the task in the database...
        EventBus.Trigger(new TaskCompletedEventData
                                {TaskId = this.Id});
    }
}
```

紧
耦合

发布领域事件

- 应用服务中

```
public class TaskAppService : ApplicationService
{
    public IEventBus EventBus { get; set; }

    public TaskAppService()
    {
        EventBus = NullEventBus.Instance;
    }

    public void CompleteTask(CompleteTaskInput input)
    {
        //TODO: complete the task in the database...
        EventBus.Trigger(new TaskCompletedEventData {TaskId = 42});
    }
}
```

实现事件处理

```
public interface IEventHandler
{
}
```

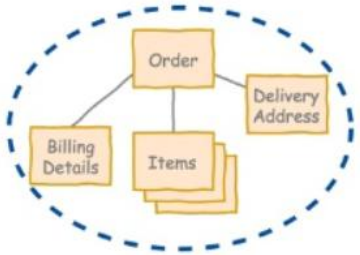
```
public interface IEventHandler<in TEventData> : IEventHandler
{
    void HandleEvent(TEventData eventData);
}
```

```
public class ActivityWriter : IEventHandler<TaskCompletedEventData>,
                              ITransientDependency
{
    public void HandleEvent(TaskCompletedEventData eventData)
    {
        WriteActivity("Task completed. Id = " + eventData.TaskId);
    }
}
```

```
EventBus.Register<TaskCompletedEventData, ActivityWriter>();
```

聚合

(Aggregate)



聚合是什么

UML 中，聚合表示一种强的关联关系，是一种整体与部分的关系，且部分能够离开整体而独立存在，如车和轮胎

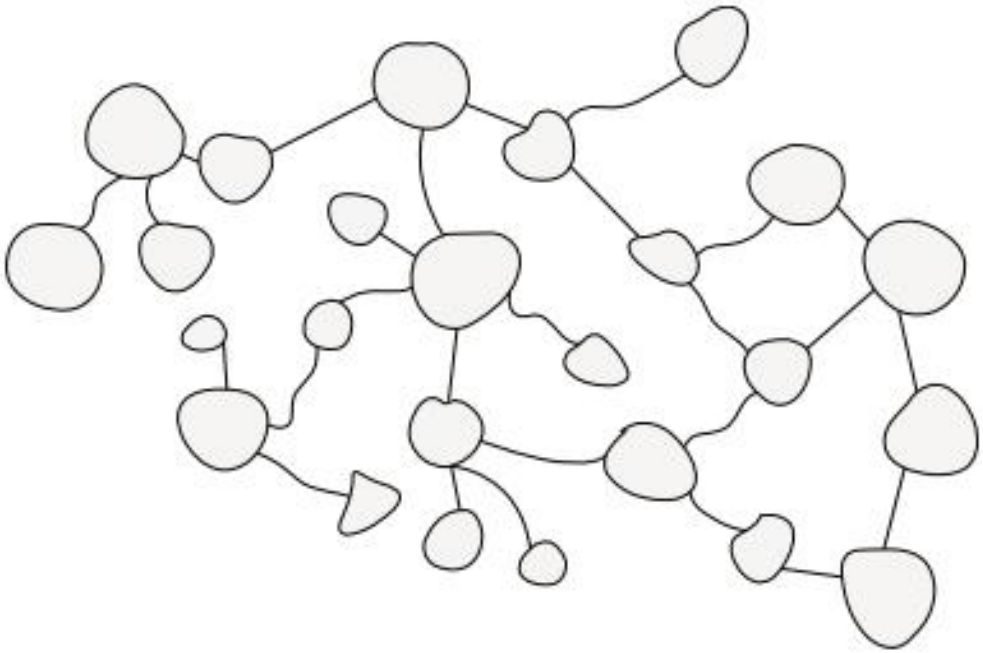
DDD中，聚合也表示整体与部分的关系，但不再强调部分与整体的独立性
聚合是将相关联的领域对象进行 显式分组，来表达整体的概念

如：将订单与订单项的领域对象进行组合，来表达领域中订单这个整体概念

封装这个在技术实现领域的基本原则
在建模时却很少被重视起来

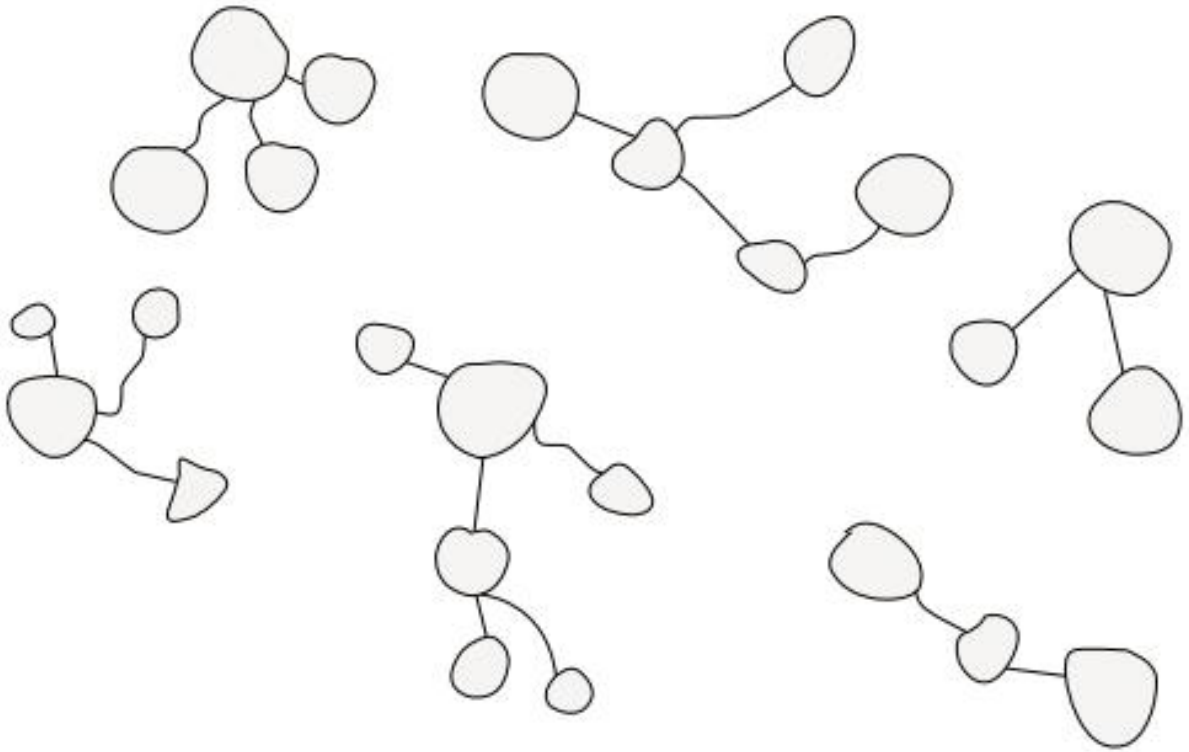
模型边界蔓延

- 设计初期，我们习惯专注于领域中的实体和值对象，以及它们之间的关联
- 这往往会忽略领域对象之间的逻辑界限，以至于建立的模型边界蔓延

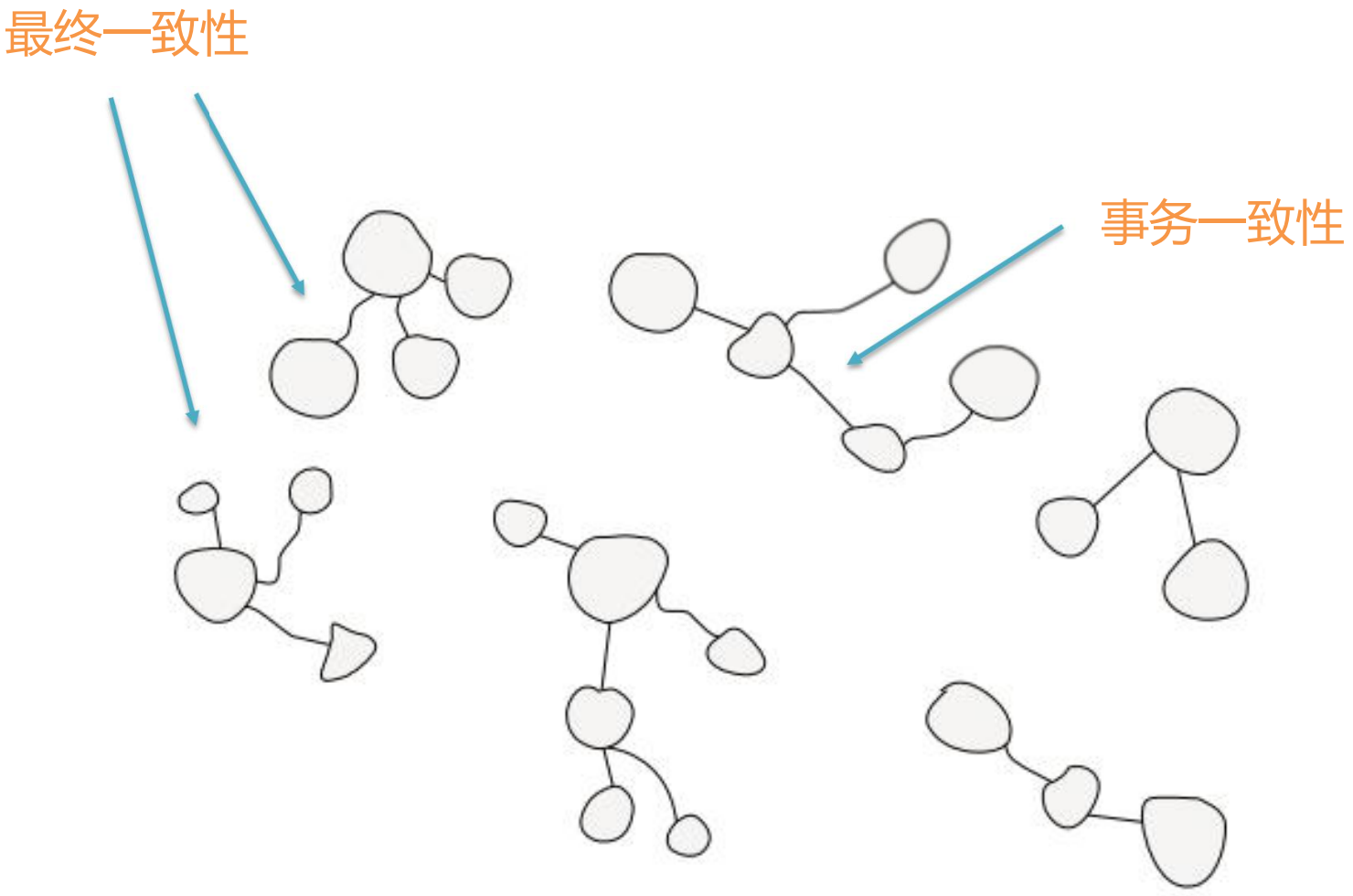


如何简化

1. 基于业务用例而非现实生活建立必要的关联
2. 划分逻辑界限，减少不必要的关联
3. 将双向的关联转换为单向关联



聚合与一致性



应用聚合

聚合是领域对象的 显式分组

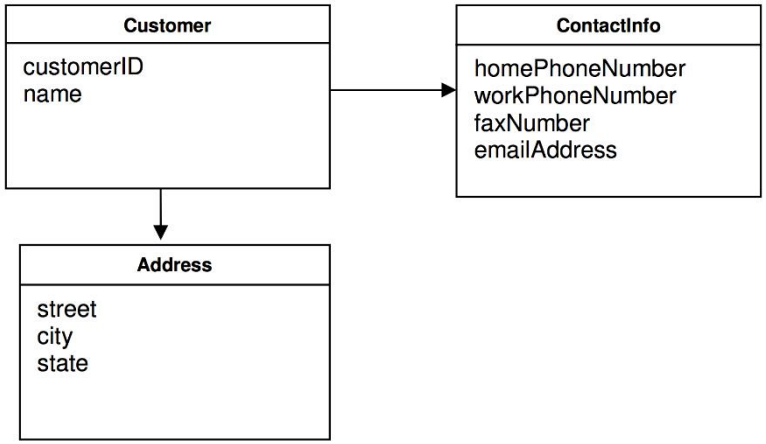
旨在支持领域模型的 行为 和 不变性 的范围

同时充当 一致性和事务性边界

每个聚合都有 一个根，根是一个实体，并且是 唯一 可 被外访问 的

因此，聚合可保证多个模型的不变性，因为这些模型都需参考聚合的根

想要改变模型，只能通过聚合的根去操作



应用聚合

订单支付成功后，订单状态要更新为已支付状态

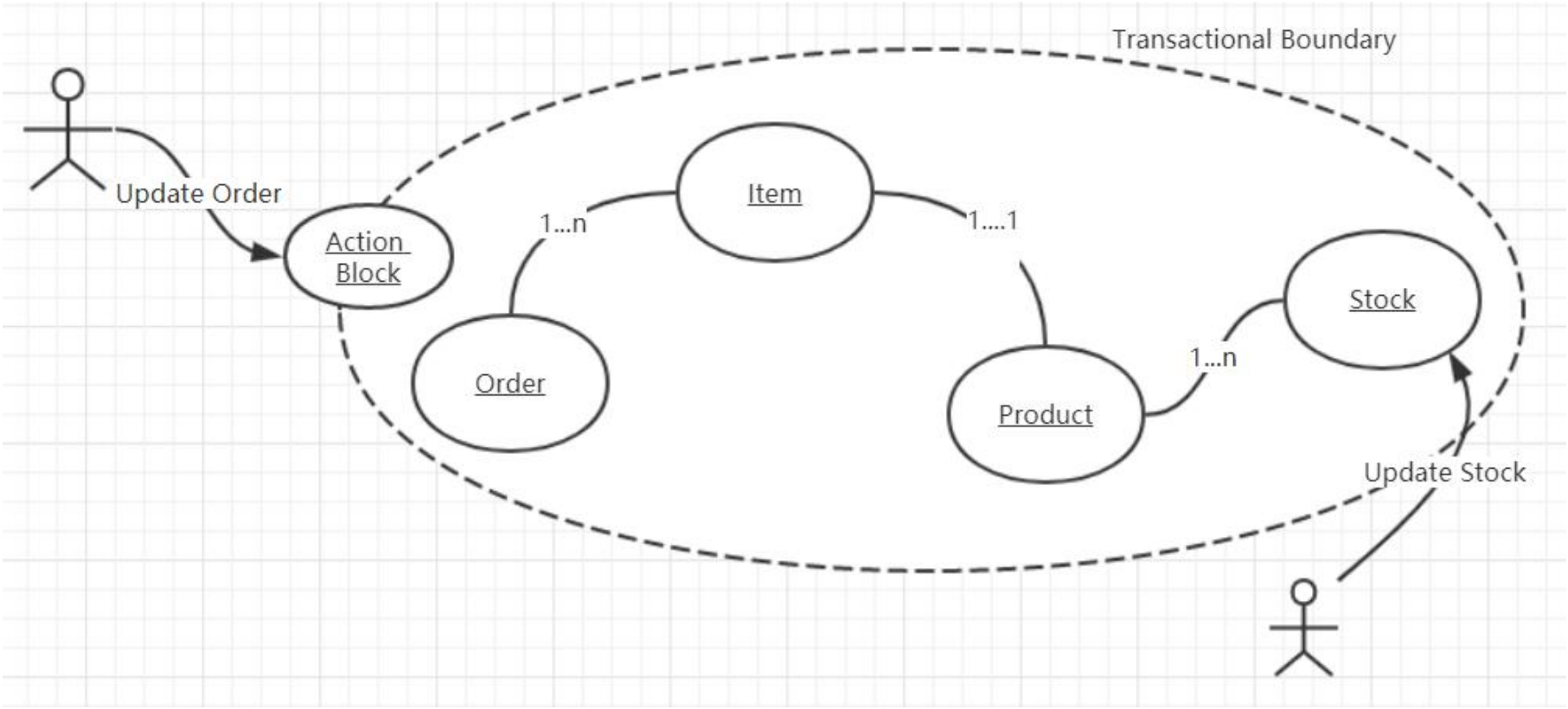
且现有库存要根据订单中商品实际销售数量进行扣减

事务一致性

传统的做法就是，在一个事务中，去更新订单状态和扣减库存

这样似乎满足了业务场景需求，但是我们不得不考虑另外一个问题 - 并发冲突

问题可能表现为数据库级别的阻塞或更新失败（由于悲观并发）

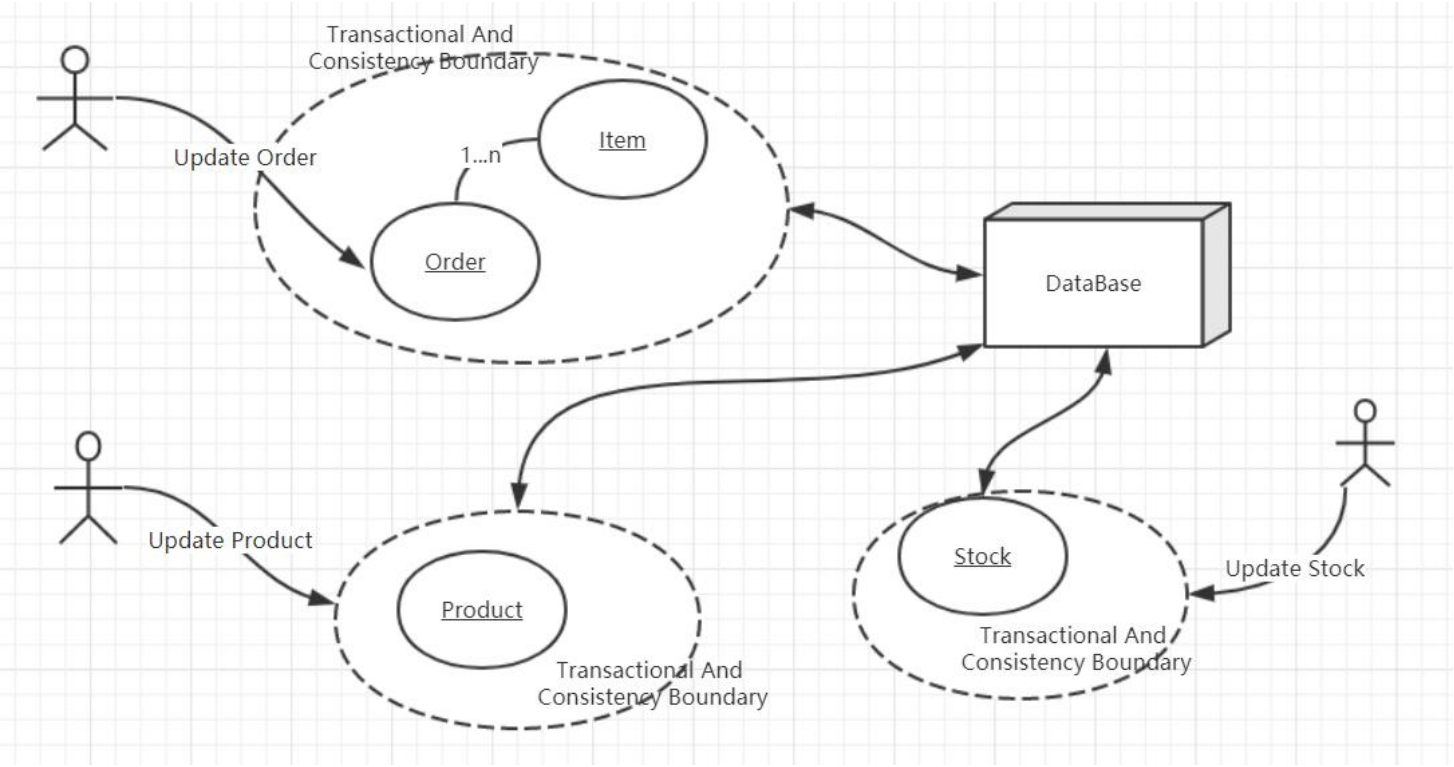


最终一致性

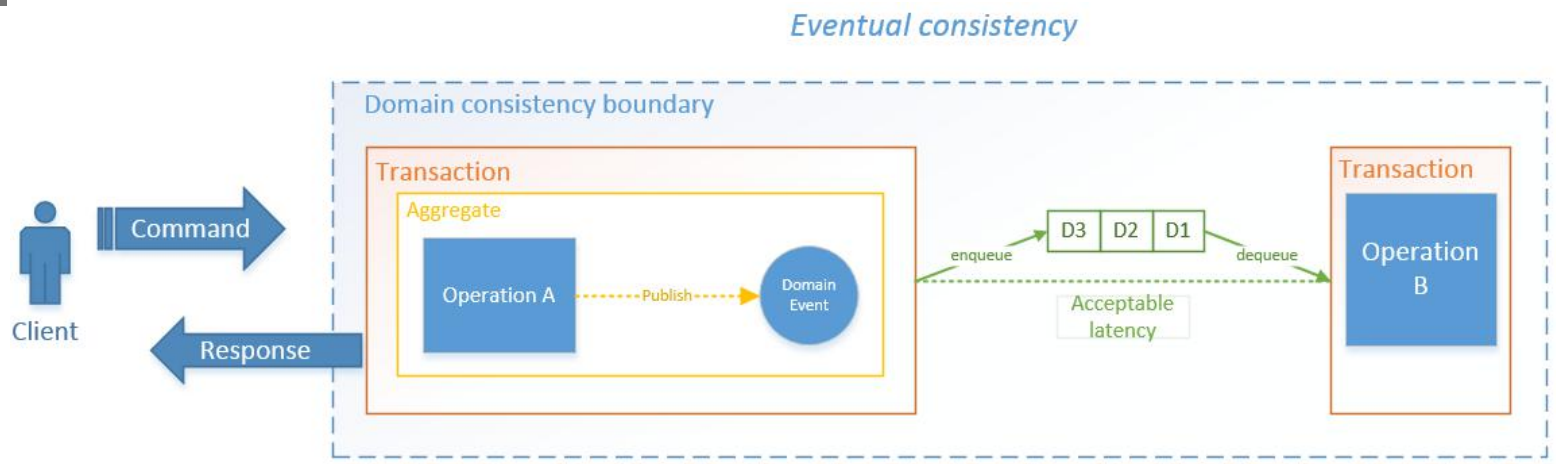
我们错误的将业务涉及到的所有领域对象都放到了一个事务性边界中去了

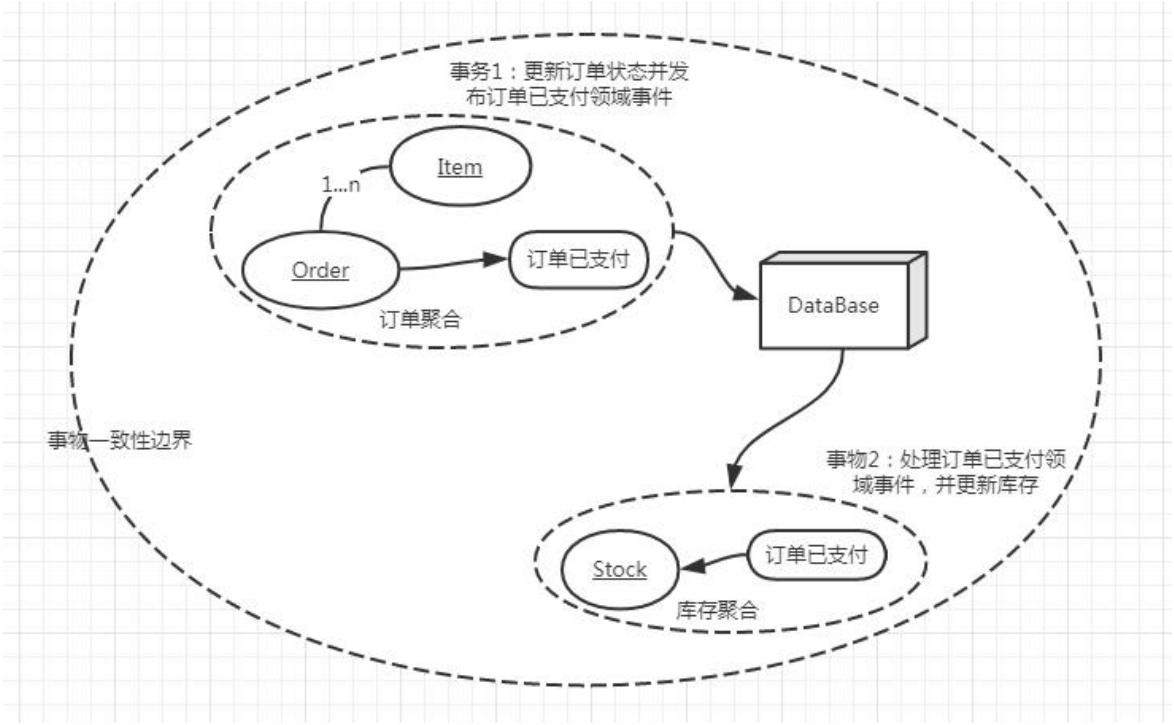
这个用例涉及到三个子域，销售、商品、库存子域

从领域不变性的角度来看，我们应该维护各自子域内业务规则的不变性



领域事件完成最终一致性





交流时间

Workshop · 工作坊 MPD

说一说你工作中遇到的聚合

资源库

(Repository)

资源库的作用

Workshop · 工作坊 MPD

资源库是聚合的管理集合，主要用于聚合的 持久化 和 检索

它隔离了 领域模型 和 数据模型

资源库通过隐藏聚合持久化和检索的底层技术实现领域层的 持久化无关性

资源库限定了只能通过 聚合根 来持久化和检索领域对象，以确保所有改动和不变性由聚合处理

分离模型

表明聚合检索的意图

隐藏聚合查询操作细节

分离模型

资源库定义应用服务及领域服务执行业务用例时需要的所有的数据访问方法

```
public interface ICustomerRepository
{
    Customer FindBy(Guid id);
    void Add(Customer customer);
    void Remove(Customer customer);
}
```

资源库的实现通常位于基础架构层，由持久化框架来支撑

将领域模型的持久化转移到基础设施层

隐藏了领域模型的技术复杂性，从而使领域对象能够专注于业务概念和逻辑

表明聚合检索的意图

对复杂领域行建模时，资源库是模型的扩展，它表明聚合检索的意图

资源库通过隐藏基础持久化框架的功能

使用命名查询方法来限制对聚合的访问

```
_userRepository.GetAllActiveUsers();
```

比以下两种写法更加能表明意图

```
select * from users where isactive = 1
```

```
var users = db.Users.Where(u => u.IsActive == 1)
```

除了查询，资源库仅暴露必要的持久化方法而不是提供所有的 CRUD 方法

持久化透明

资源库的要点是保持你的领域模型和技术持久化框架的独立性

这样领域模型可以隔离来自底层持久化技术的影响

在 Hibernate 里，类型是持久化透明的

Entity Framework 直到 v4.0，类型才是持久化透明的

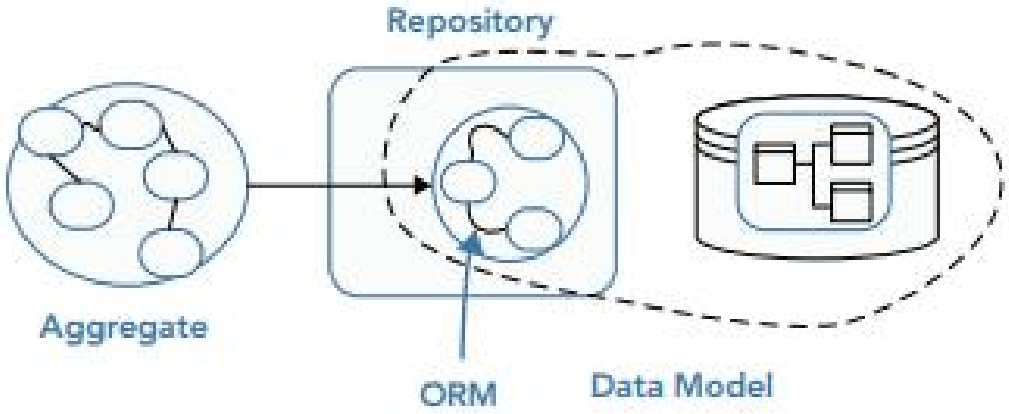


FIGURE 21-1: An ORM maps between the domain and persistence model.

定义泛型资源库

```
public interface IRepository : ITransientDependency
{
}
```

```
public interface IRepository<TEntity, TPrimaryKey> : IRepository
  where TEntity : class, IEntity<TPrimaryKey>
{
  IQueryable<TEntity> GetAll();
  TEntity Get(TPrimaryKey id);
  TEntity Insert(TEntity entity);
  TEntity Update(TEntity entity);
  void Delete(TEntity entity);
  int Count();
}
```

实现泛型资源库

```
public abstract class EfRepositoryBase<TEntity, TPrimaryKey> :
    IRepository<TEntity, TPrimaryKey>
    where TEntity : class, IEntity<TPrimaryKey> {
  private readonly DbContext _dbContext;
  public virtual DbSet<TEntity> Table => _dbContext.Set<TEntity>();

  public IQueryable<TEntity> GetAll() {
    return Table;
  }

  public virtual TEntity Get(TPrimaryKey id) {
    var entity = GetAll().FirstOrDefault(id);
    return entity;
  }
```

实现泛型资源库

```
public TEntity Insert(TEntity entity) {
    return Table.Add(entity);
}

public TEntity Update(TEntity entity) {
    AttachIfNot(entity);
    _dbContext.Entry(entity).State = EntityState.Modified;
    return entity;
}

public void Delete(TEntity entity) {
    AttachIfNot(entity);
    Table.Remove(entity);
}

public int Count() {
    return GetAll().Count();
}
```

使用资源库

```
public class PersonAppService : ApplicationService,
IPersonAppService
{
    private readonly IRepository<Person> _personRepository;

    public PersonAppService(IRepository<Person> personRepository)
    {
        _personRepository = personRepository;
    }

    public async Task<GetPeopleOutput> GetAllPeople()
    {
        var people = await _personRepository.GetAllListAsync();

        return new GetPeopleOutput
        {
            People = Mapper.Map<List<PersonDto>>(people)
        };
    }
}
```

通过 Hibernate 实现资源库

```
public class CustomerRepository : ICustomerRepository {
    private ISession _session;

    public CustomerRepository(ISession session) {
        _session = session;
    }
    public IEnumerable<Customer> FindBy(Guid id) {
        return _session.Load<Order> (id);
    }

    public void Add(Customer customer) {
        _session.Save(customer);
    }

    public void Remove(Customer customer) {
        _session.Delete(customer);
    }
}
```

资源库与缓存

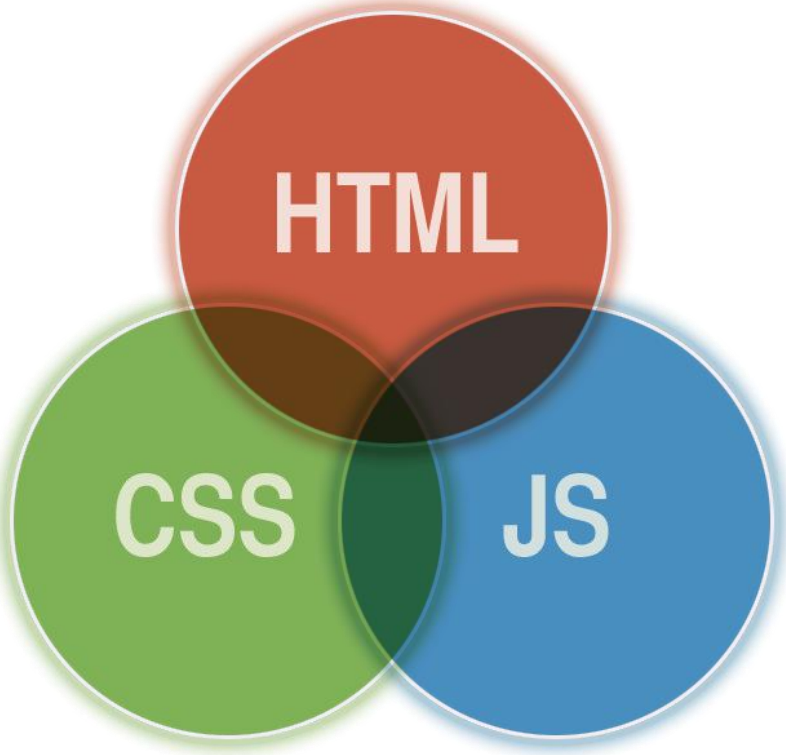
```
public class CustomerRepository : ICustomerRepository {
    private readonly ICacheManager _cacheManager;

    public CustomerRepository(ICacheManager cacheManager) {
        _cacheManager = cacheManager;
    }

    public Customer FindBy(Guid id)
    {
        return _cacheManager
                .GetCache("MyCache")
                .Get(id, () => GetFromDatabase(id));
    }

    public Customer GetFromDatabase(Guid id)
    {
        //... retrieve item from database
    }
}
```

资源库总结

- 资源库是一个契约而不是数据访问层，明确表明聚合所必需的数据操作
- 尽可能使用 IRepository<T>
- 如扩展了 IRepository<T>，派生类需要设计成无状态的
- 资源库不应该包含业务逻辑
- ORM 用面向对象的方式来表示数据模型
- 资源库使用 ORM 来协调领域模型和数据模型

Workshop · 工作坊 MPD

分离关注点



DDD

不是技术

而是 讨论、聆听、理解与发现


Thank you !