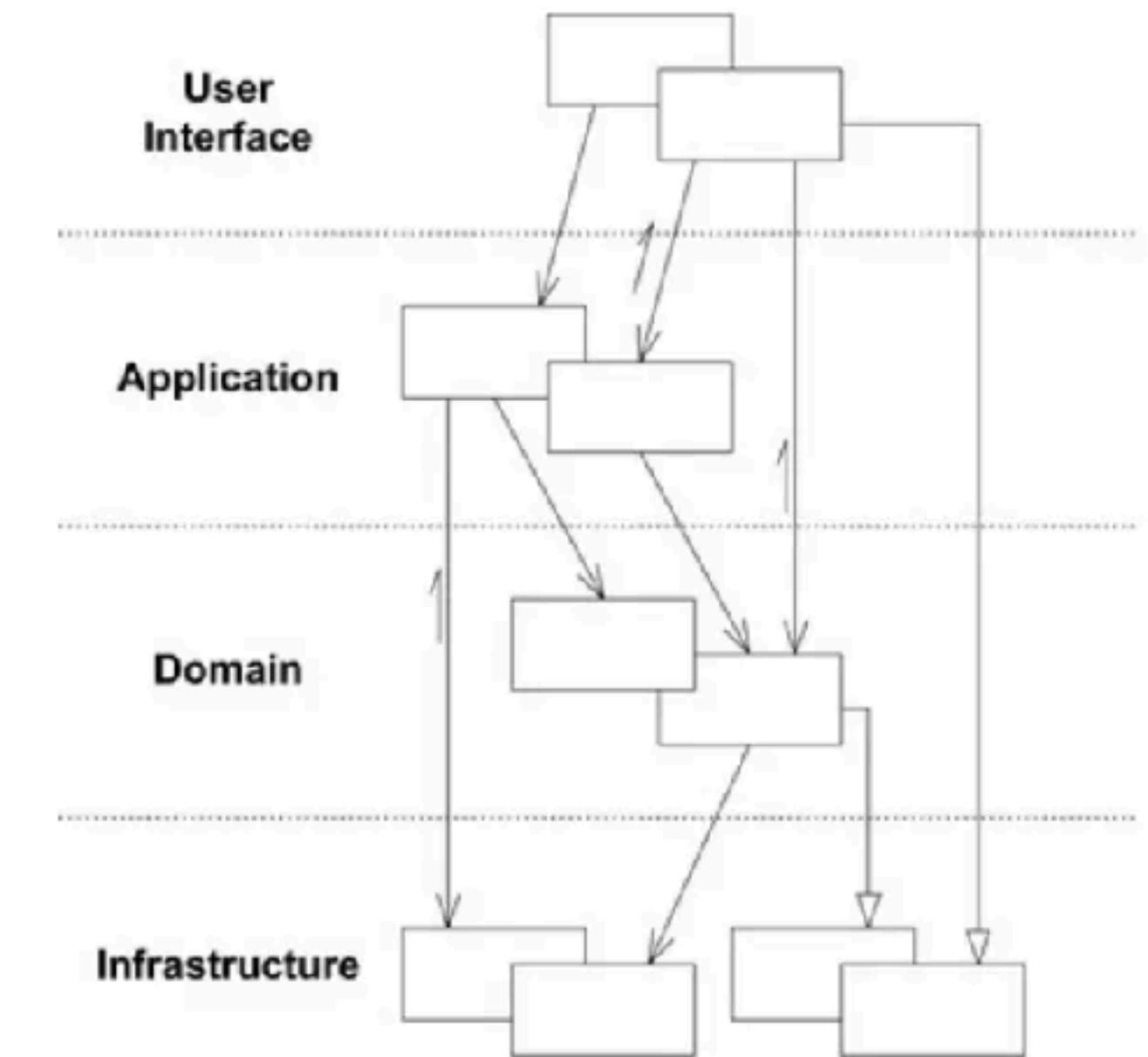
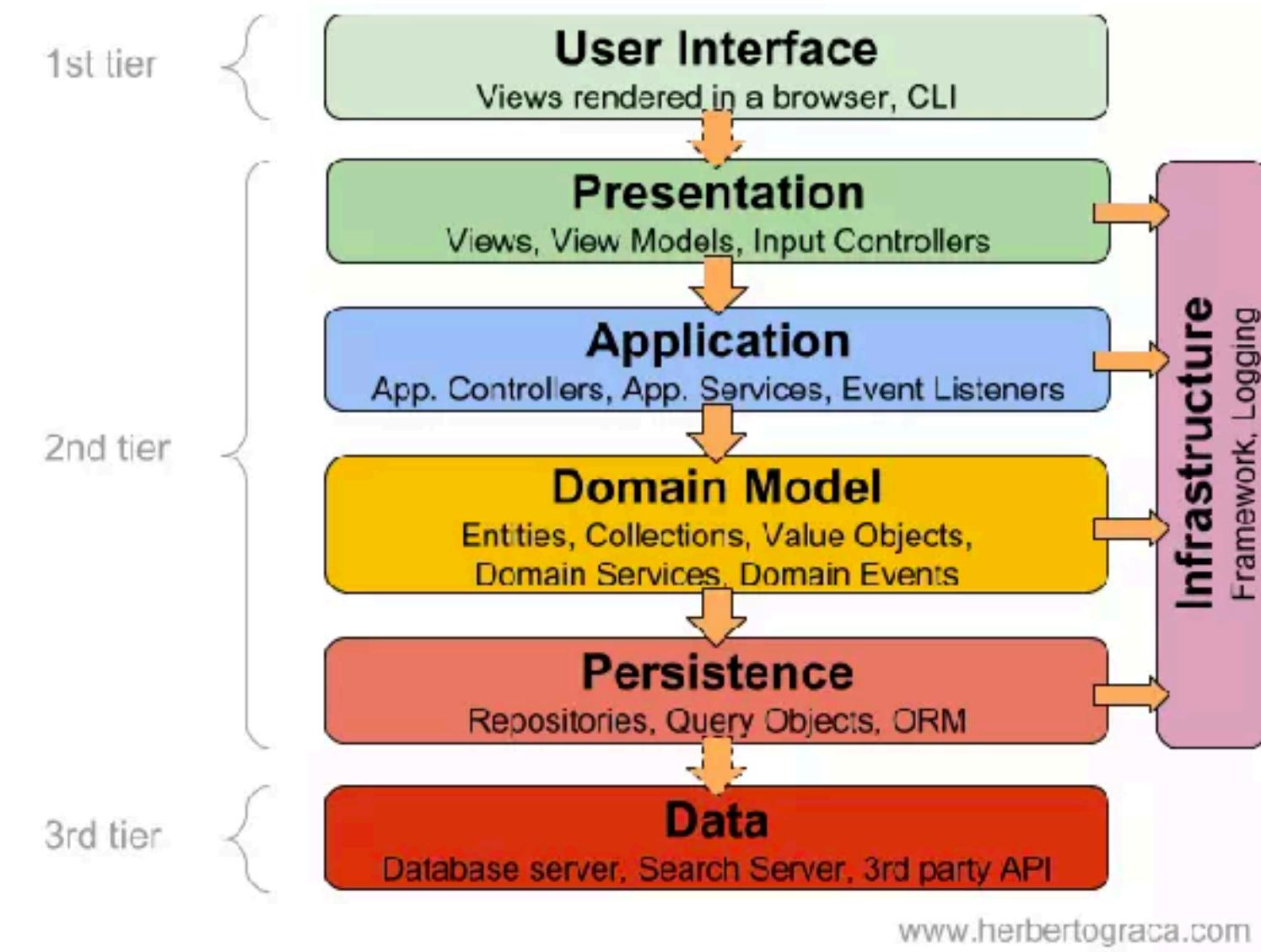
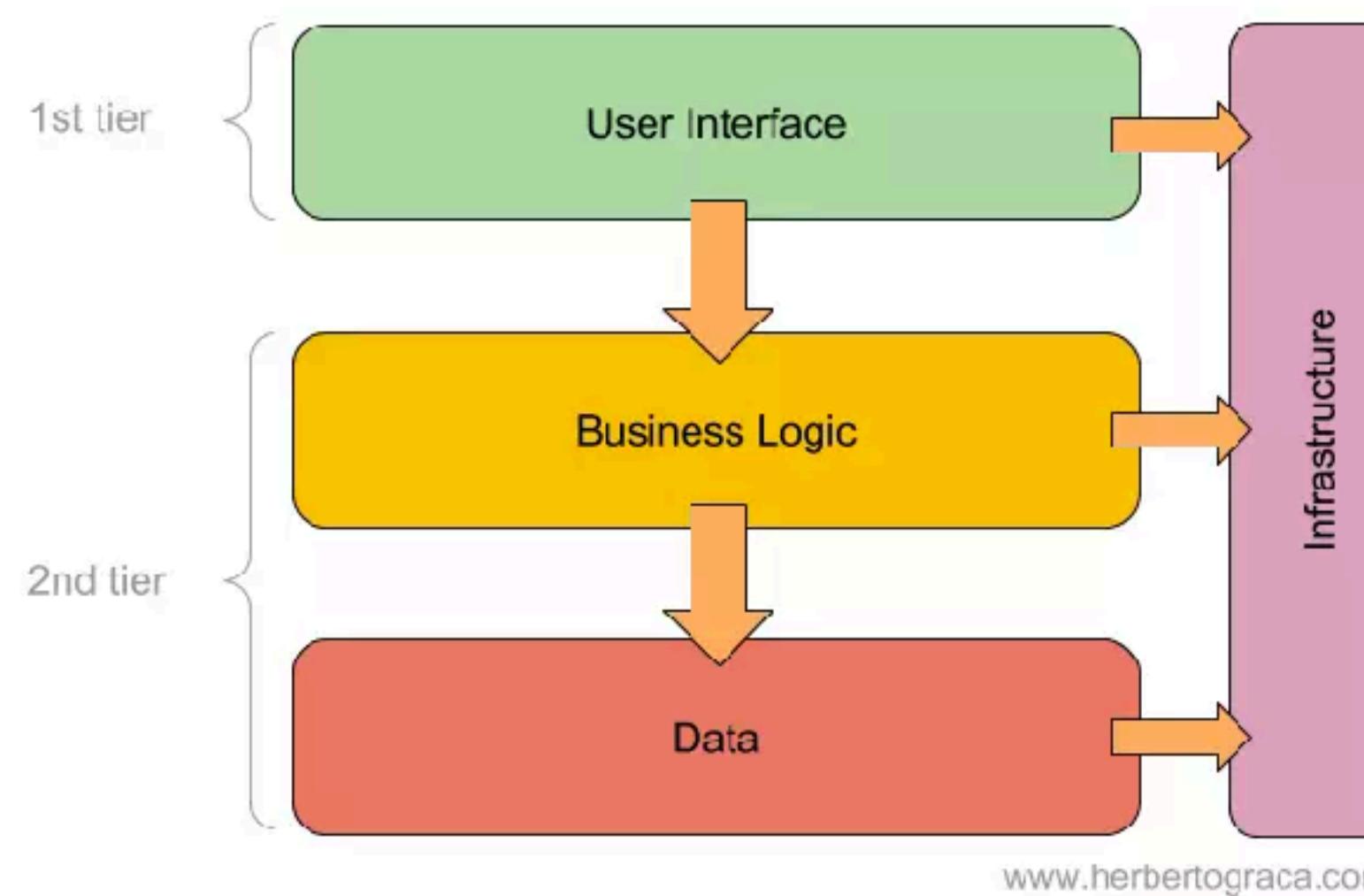


# 分层架构的演进

---

# 传统分层架构

## 传统分层架构的演进



优点：

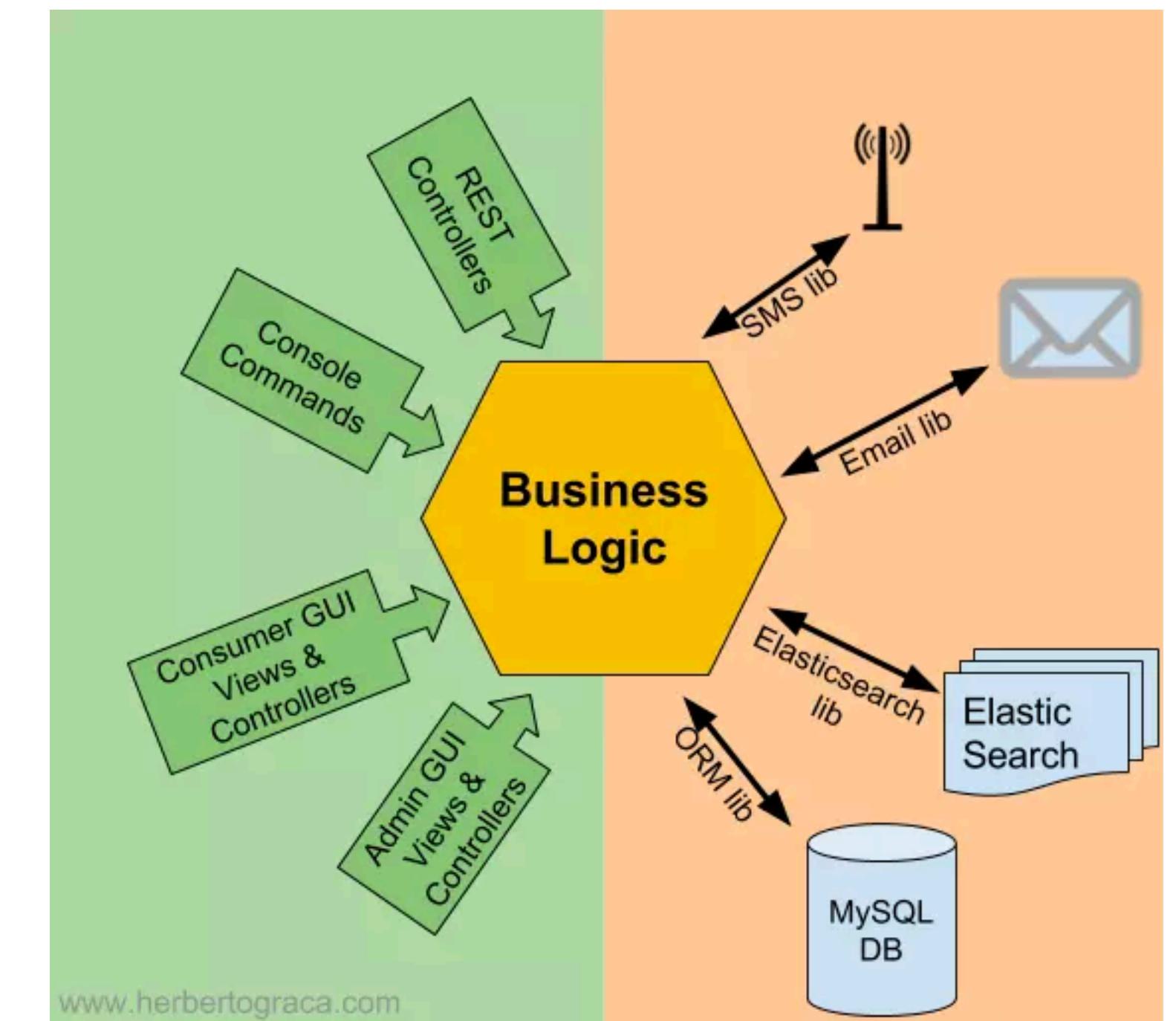
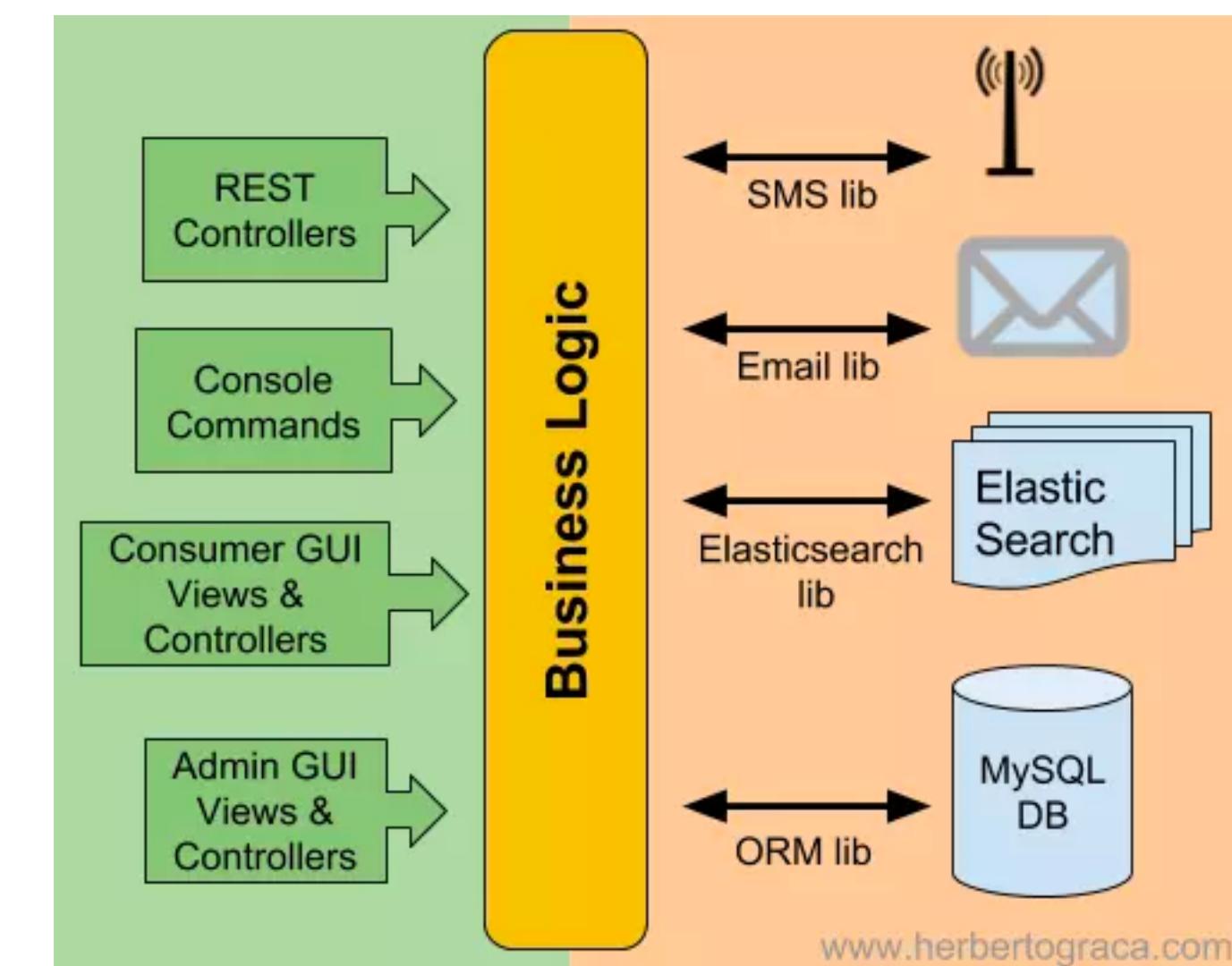
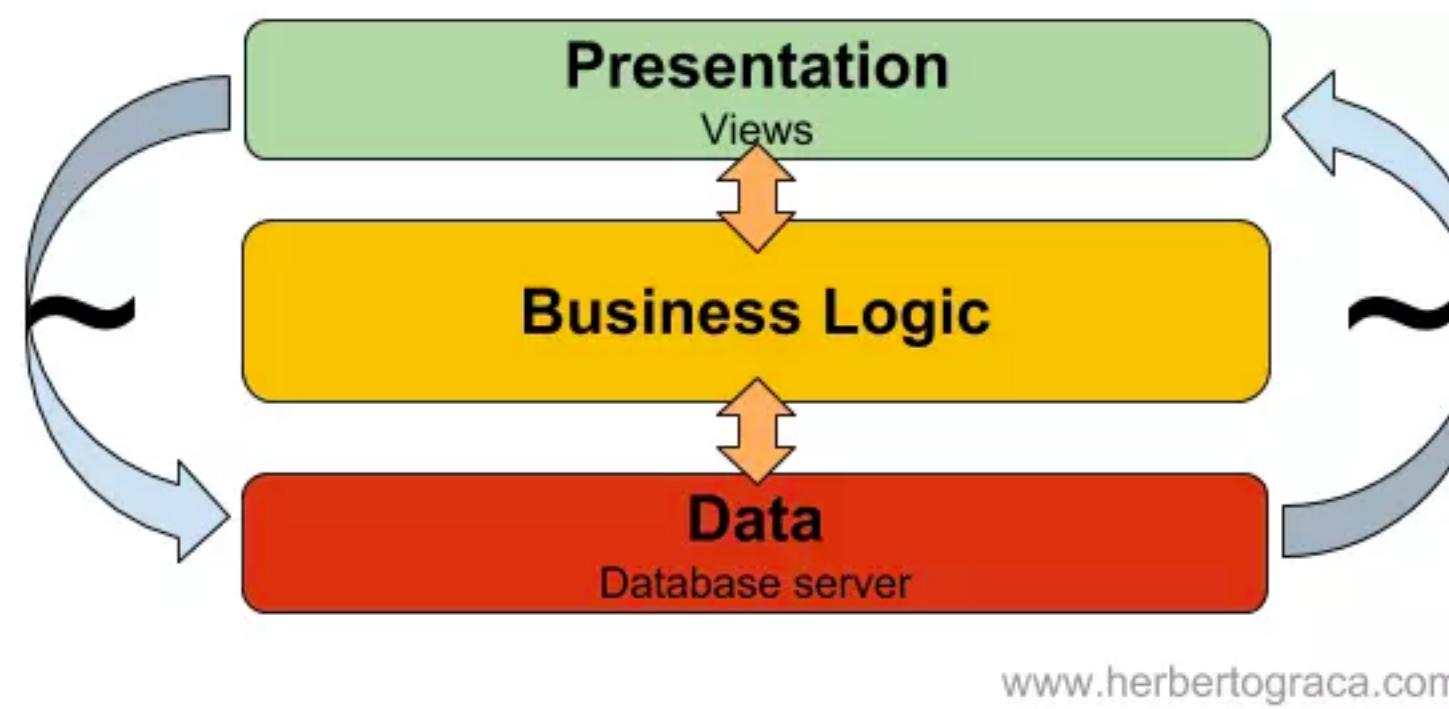
- 我们只需要了解我们工作的那层之下的层次；
- 每个层次都可以用等价的实现替换，而不会影响到其它层次；
- 层次是标准化的最佳候选；
- 层次可以被多个不同的上级层次使用。

缺点：

- 分层并不能隔离一切 (UI 中添加的字段，也要添加到数据库)；
- 额外的分层会影响性能，尤其是位于不同物理层的时候。

# 现代分层架构

## 六边形架构(端口与适配器架构)



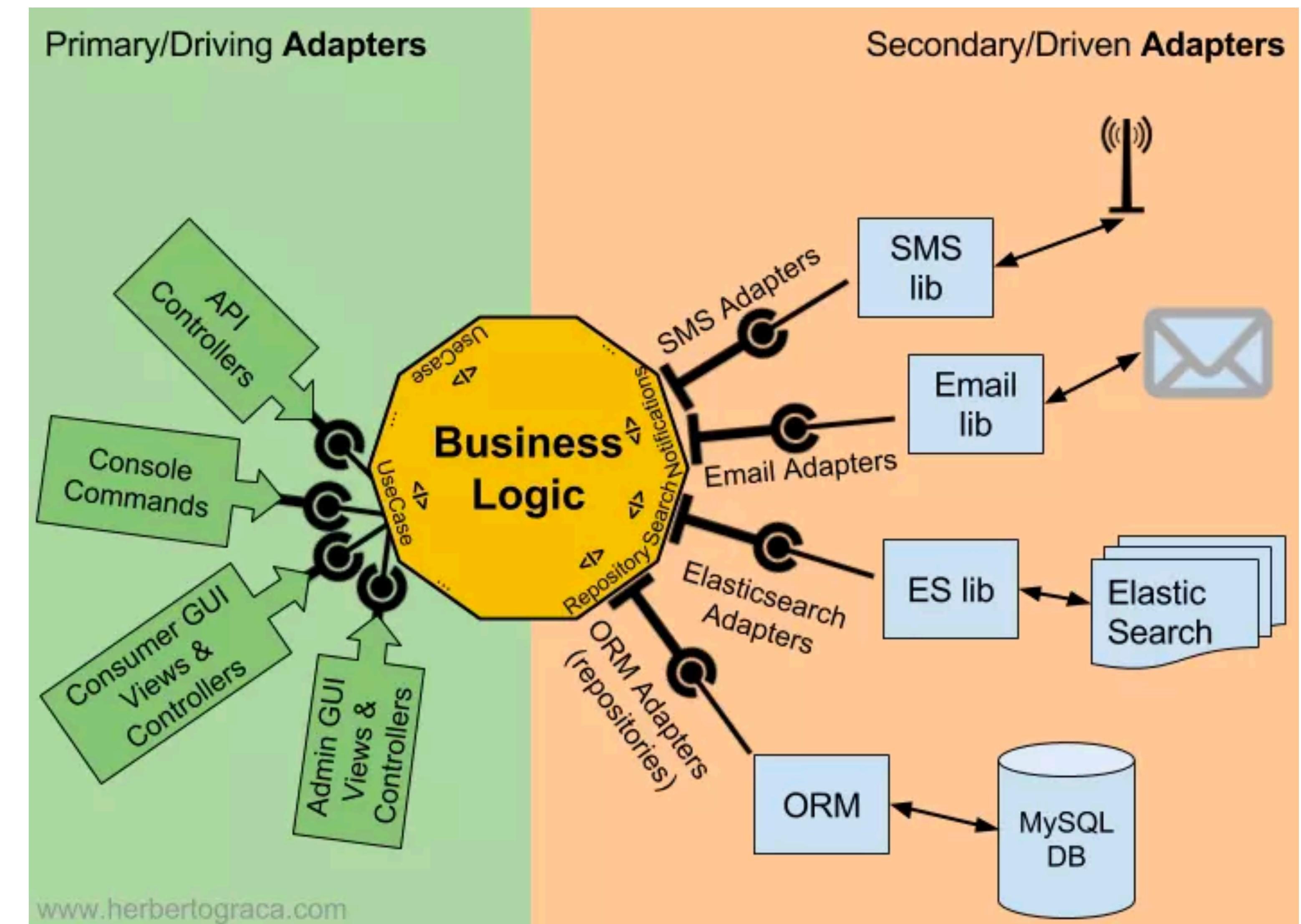
# 现代分层架构

## 端口与适配器架构

端口是对其消费者无感知的进入/离开应用的入口和出口。

适配器是将一个接口转换(适配)成另一个接口的类。

使用这种应用位于系统中心的端口/适配器设计，让我们可以保持应用和实现细节之间的隔离，这些实现细节包括技术、工具和用户界面。它还让可重用的概念更容易更快速地得到验证并被创建出来。

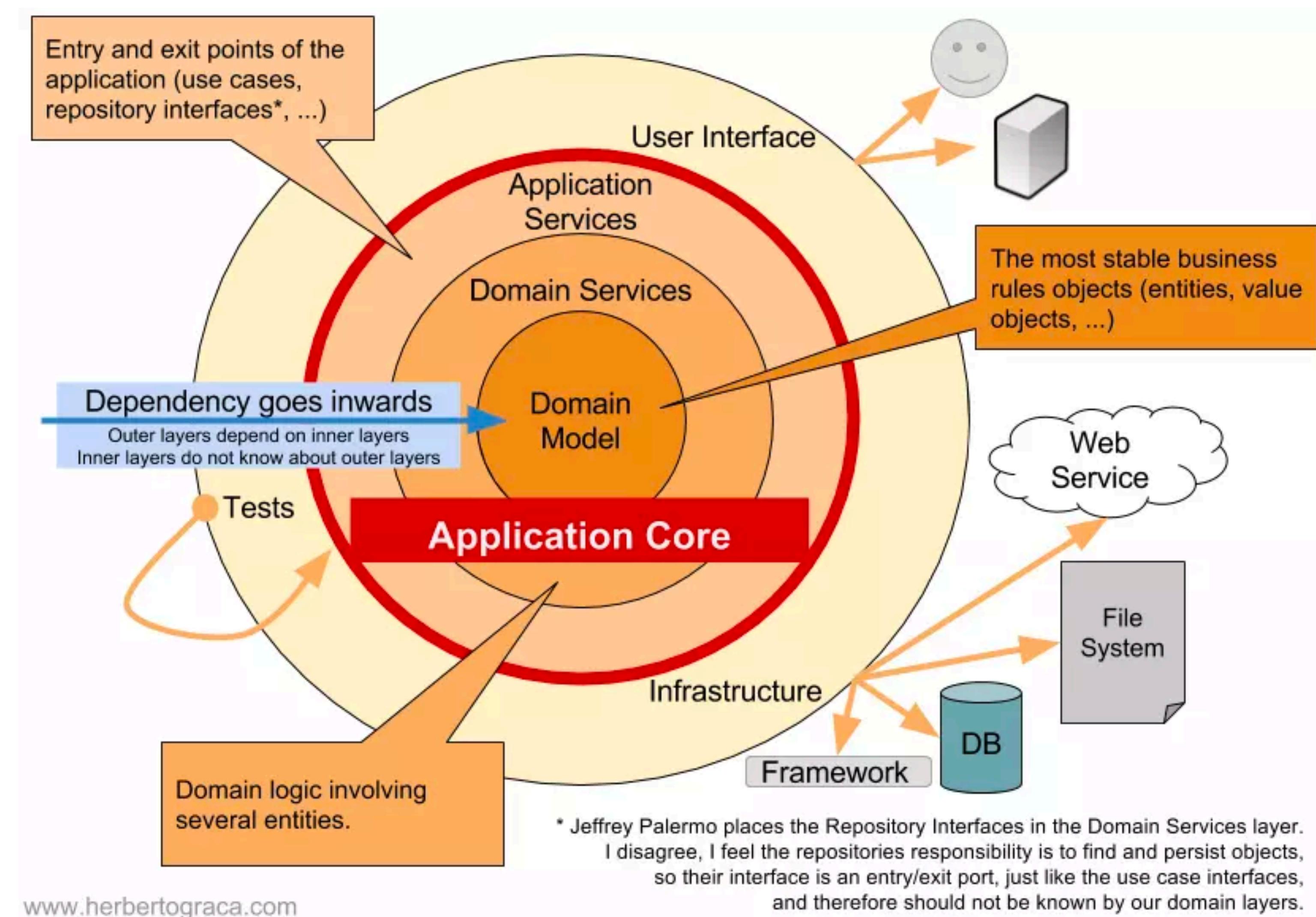


# 现代分层架构

## 洋葱架构

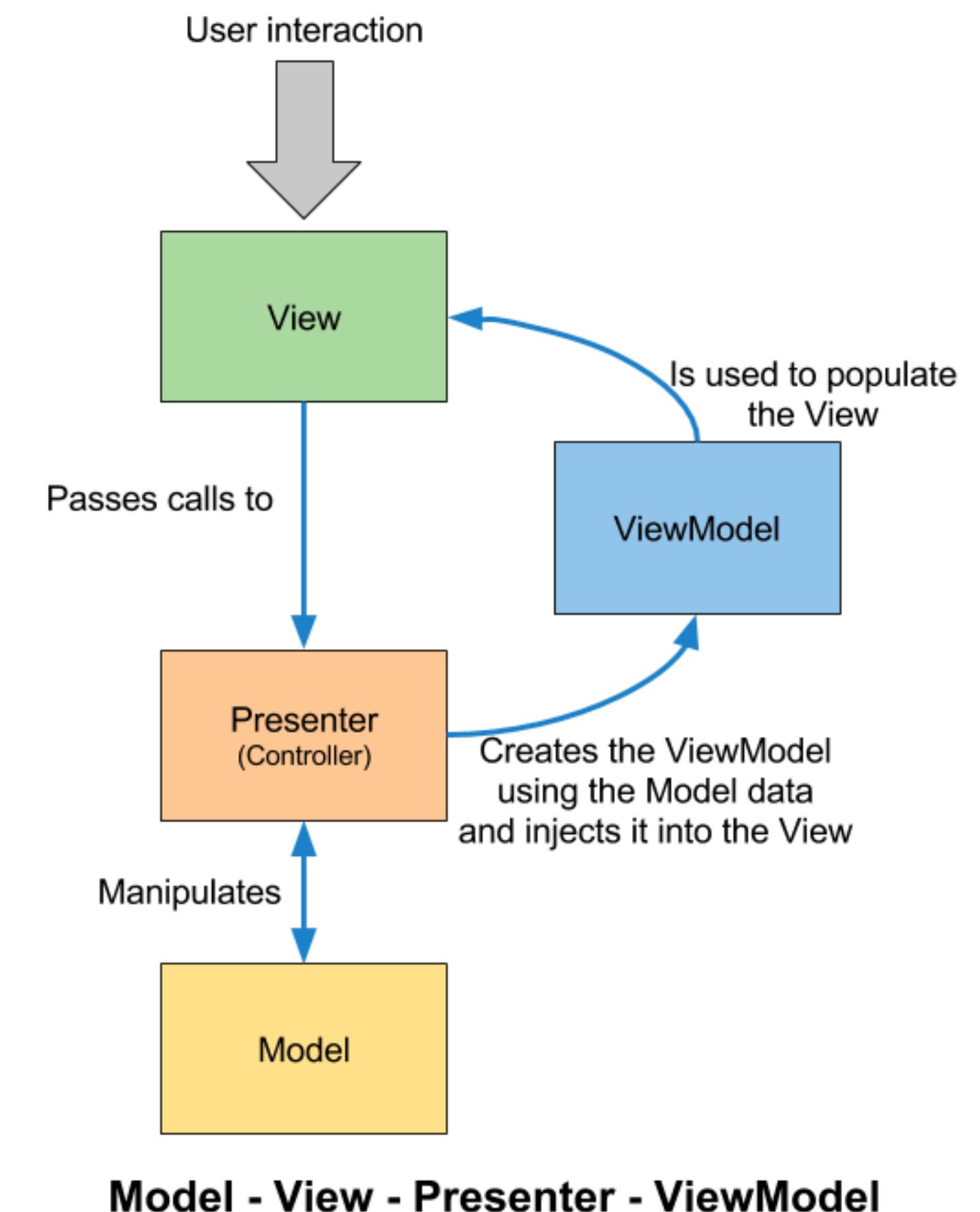
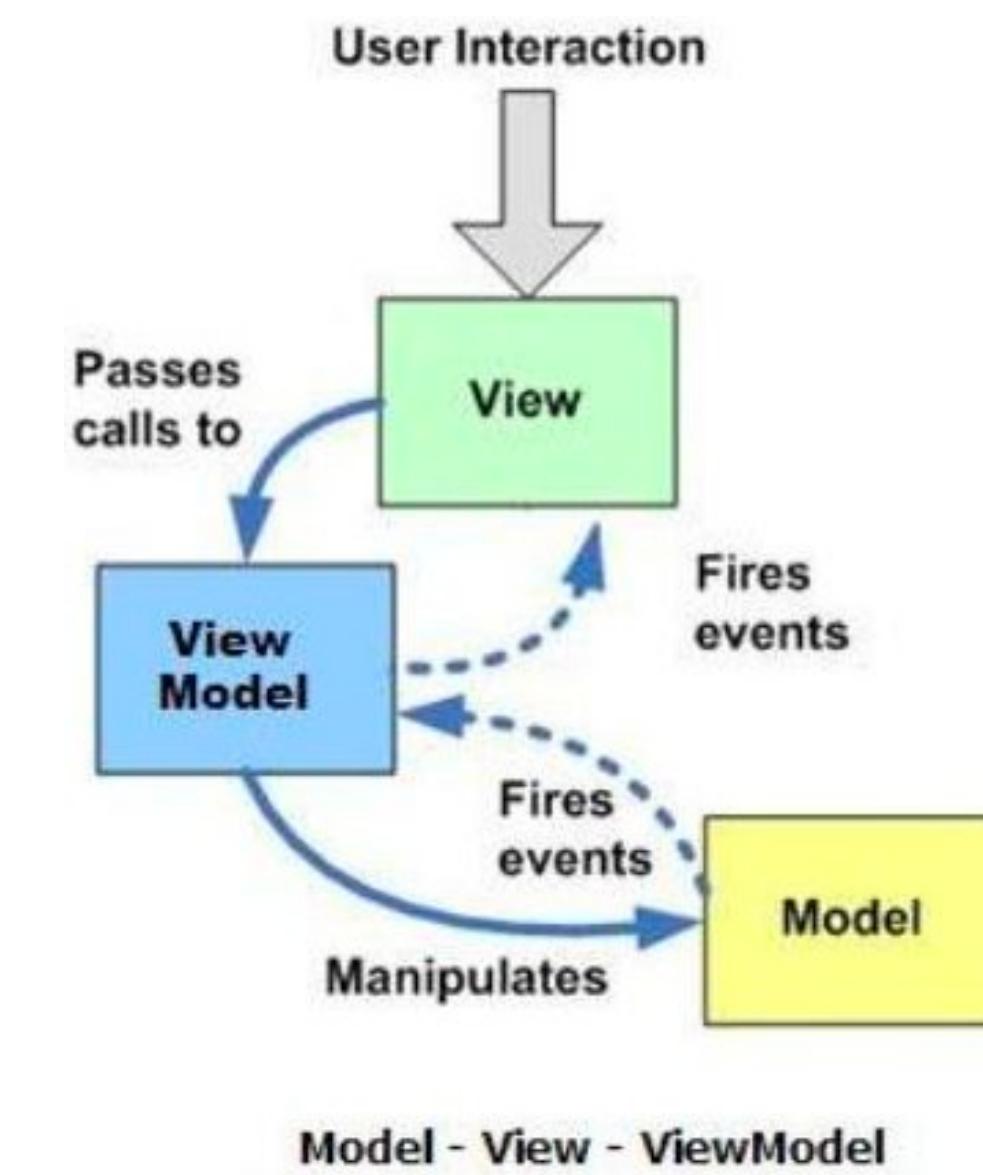
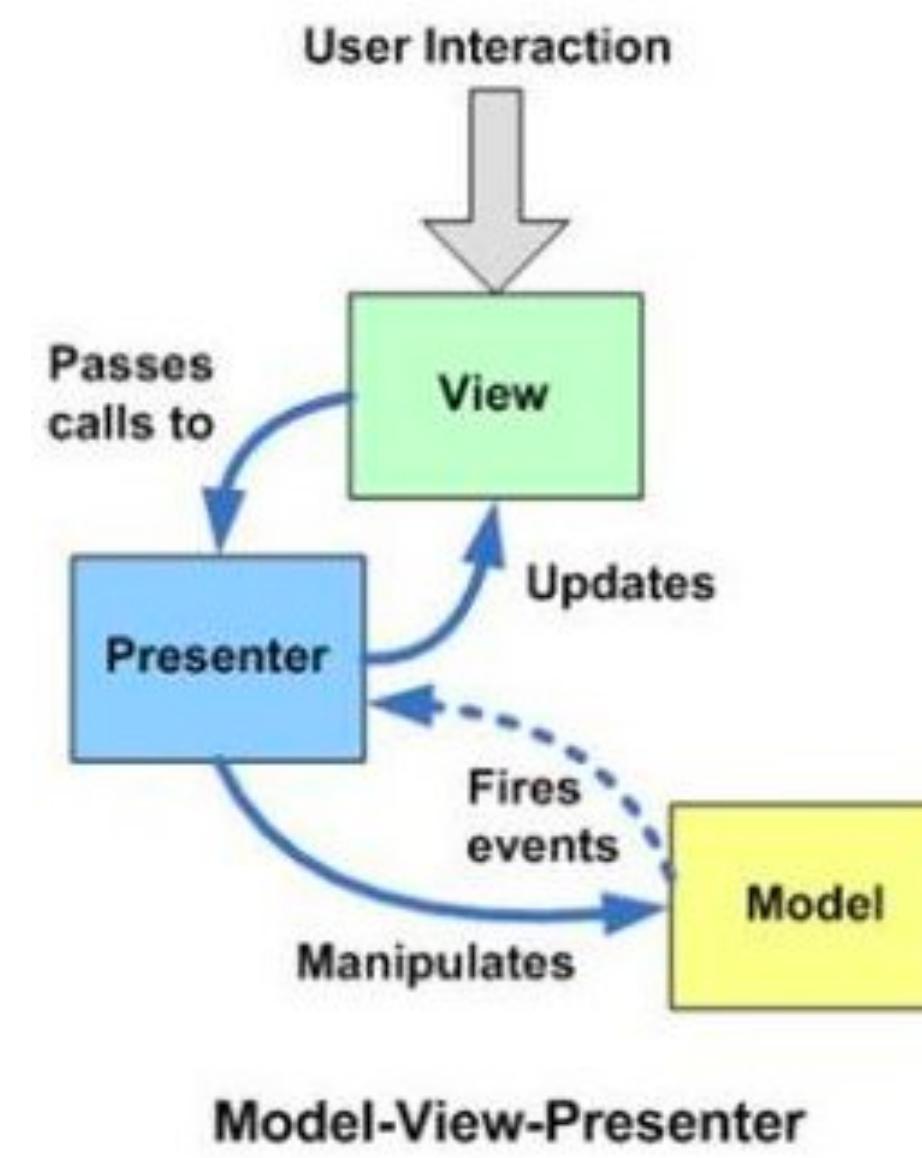
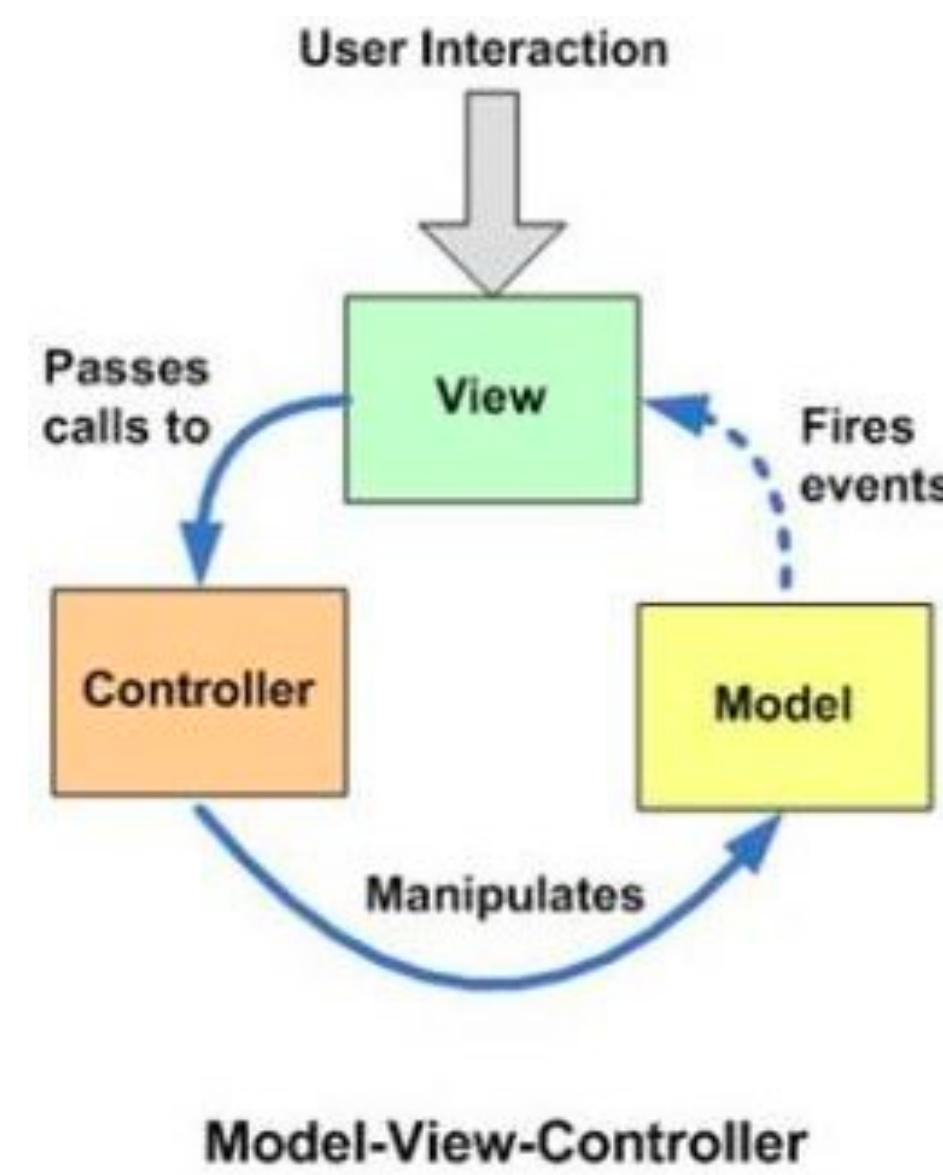
### 洋葱架构的关键原则：

- 围绕独立的对象模型构建应用
- 内层定义接口，外层实现接口
- **依赖的方向指向圆心**
- 所有的应用代码可以独立于基础设施编译和运行



# 客户端应用架构演进

## MVC架构及其变种



Model-View-Controller

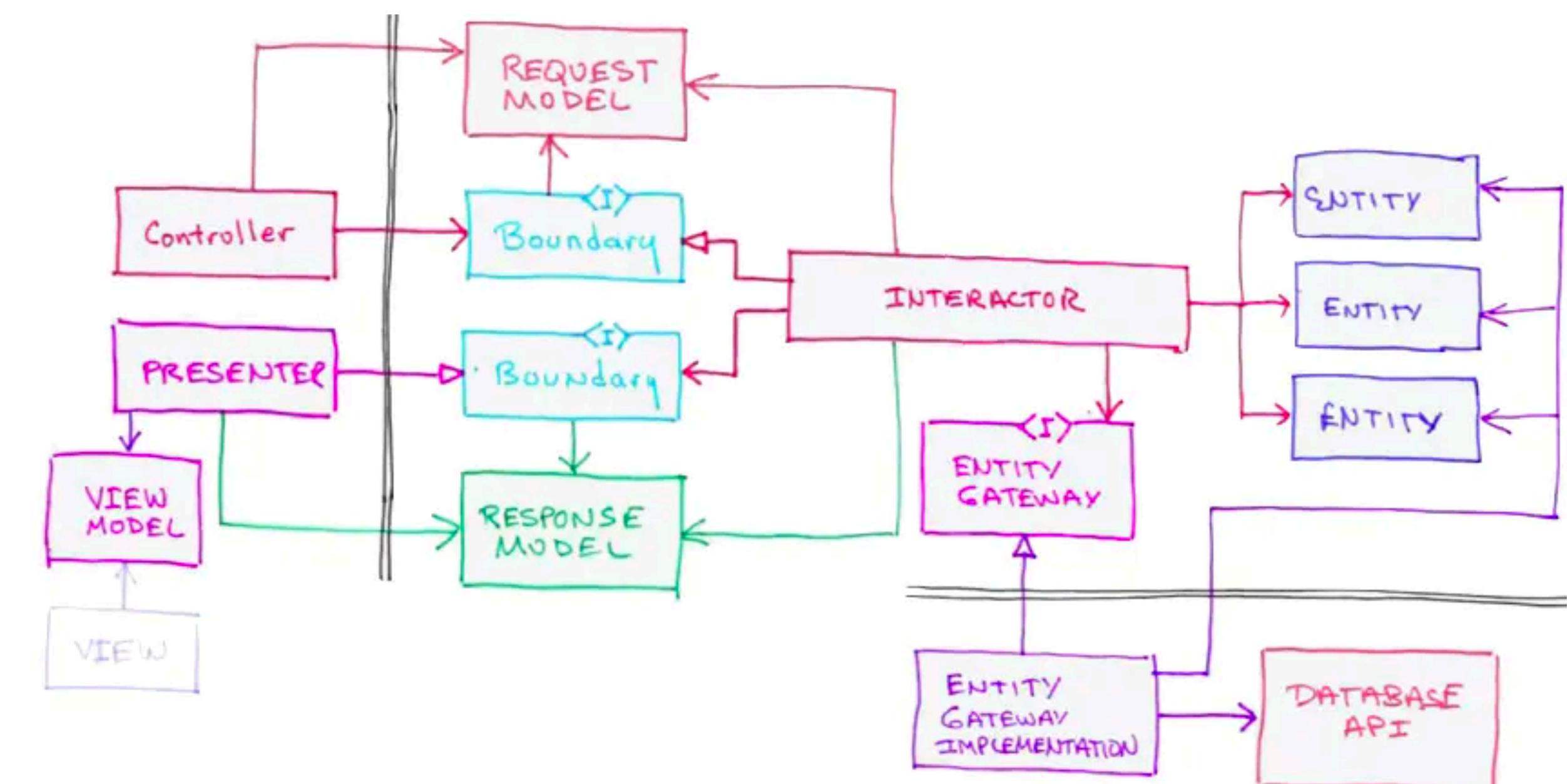
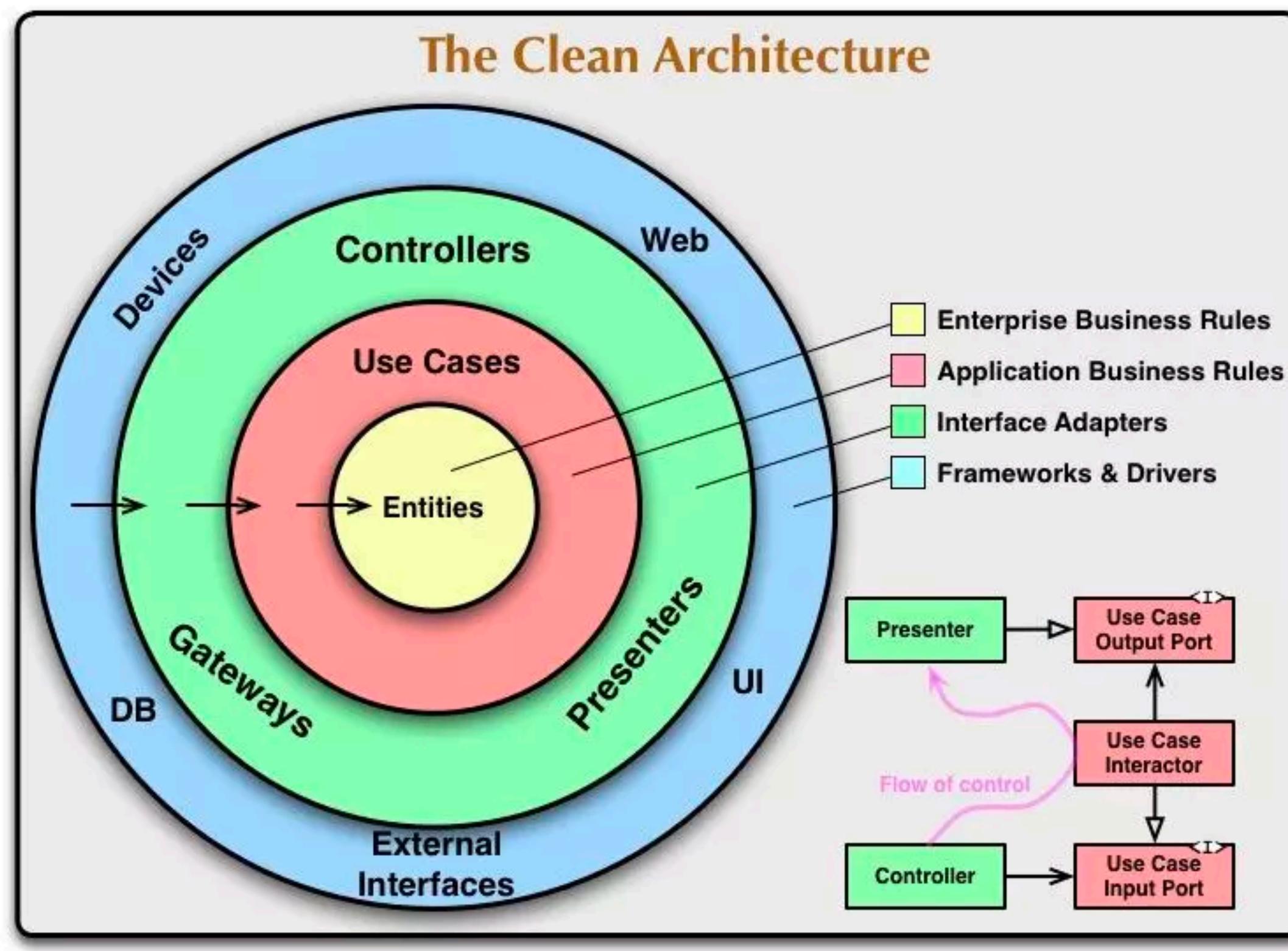
Model-View-Presenter

Model - View - ViewModel

Model - View - Presenter - ViewModel

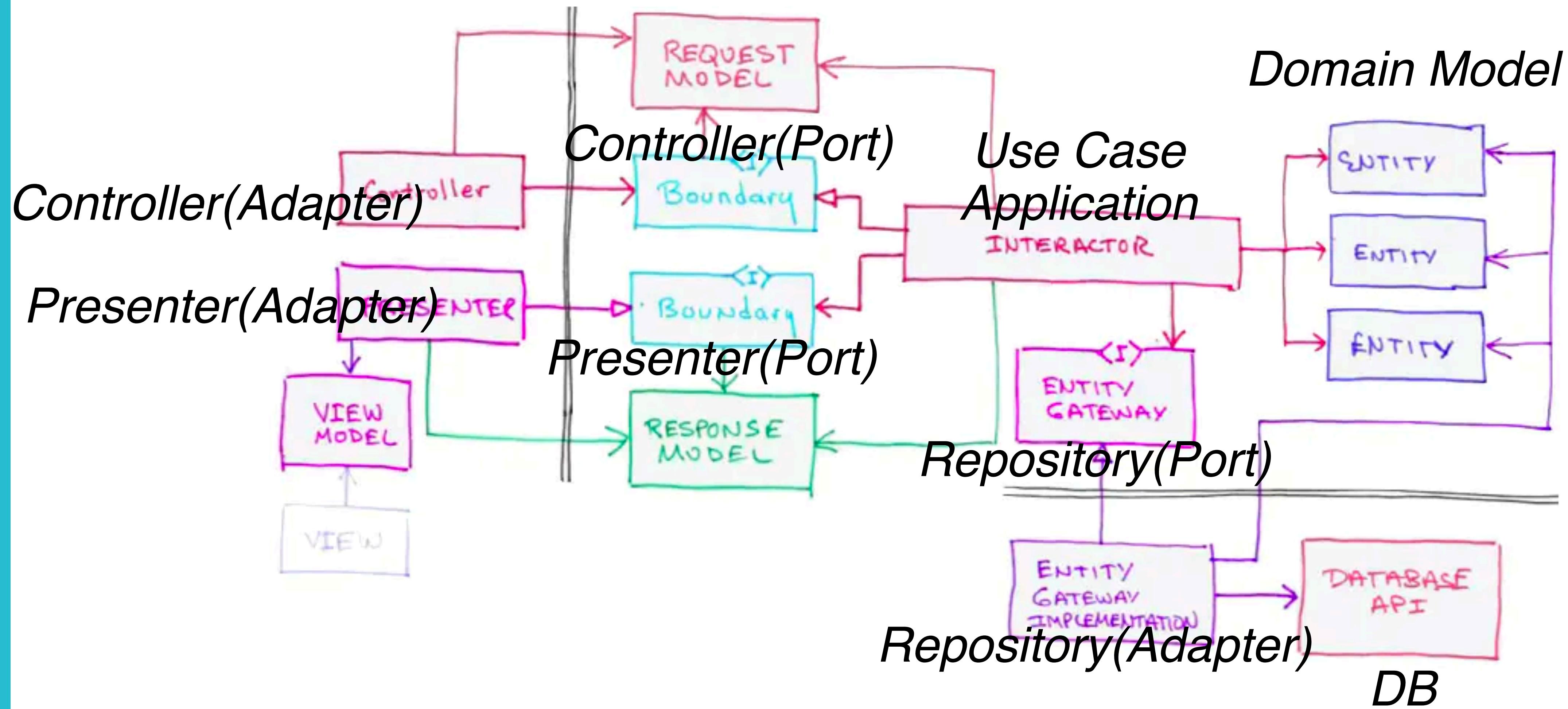
# 现代分层架构

## 集大成者：整洁架构



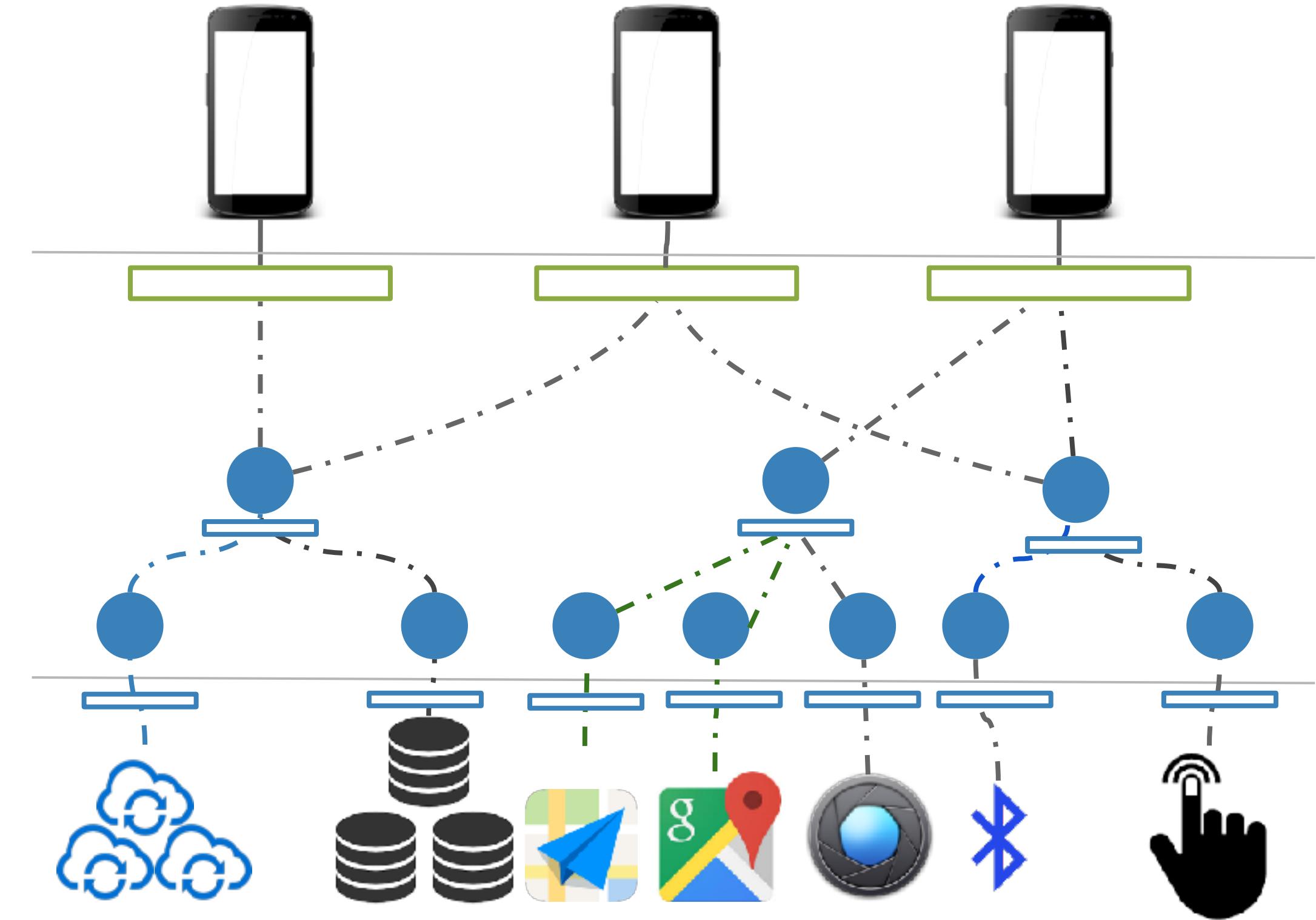
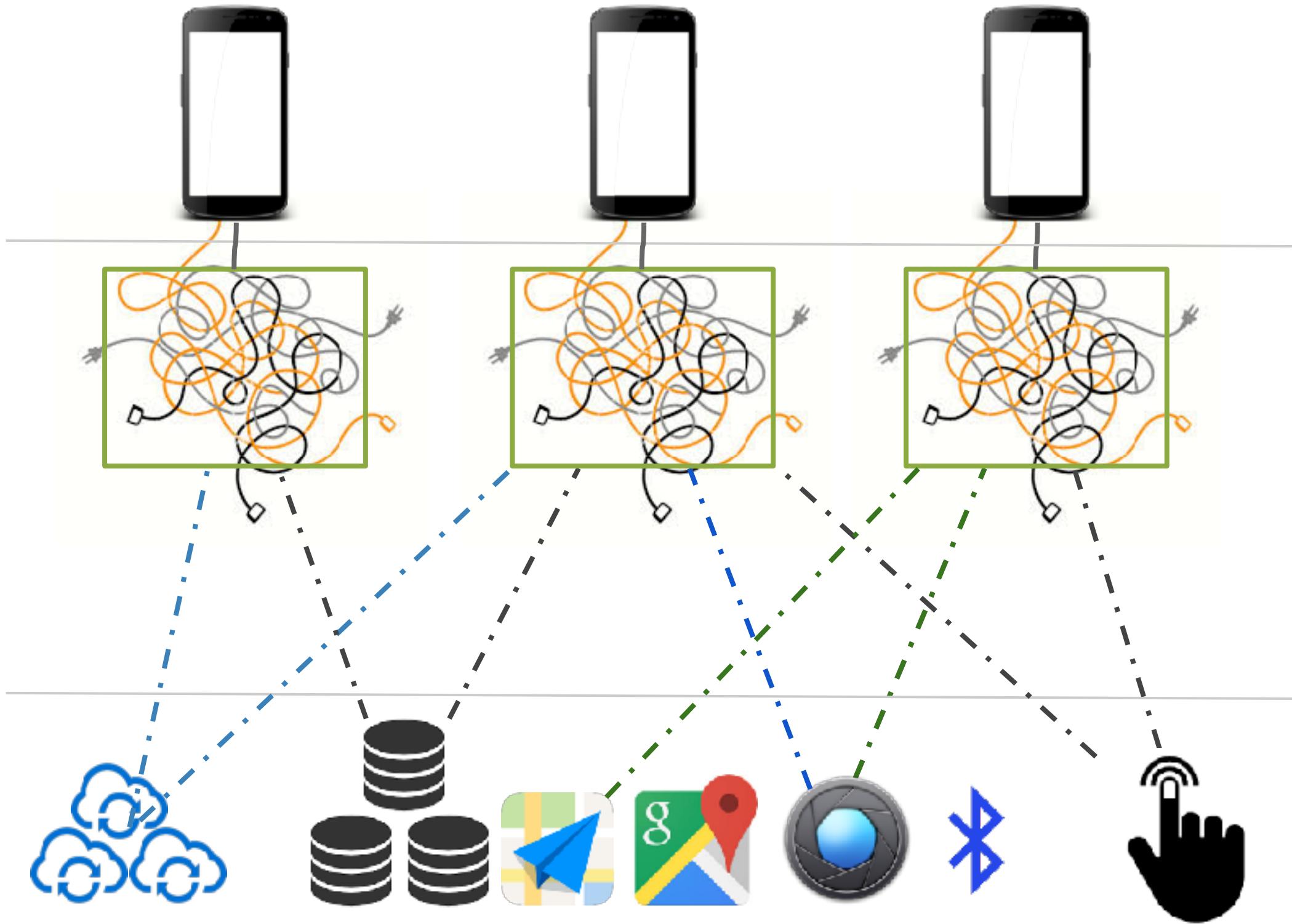
## 现代分层架构

### 集大成者：整洁架构



现代分层架构

## 根据整洁架构隔离变化



# 现代分层架构

## 用代码表述架构意图

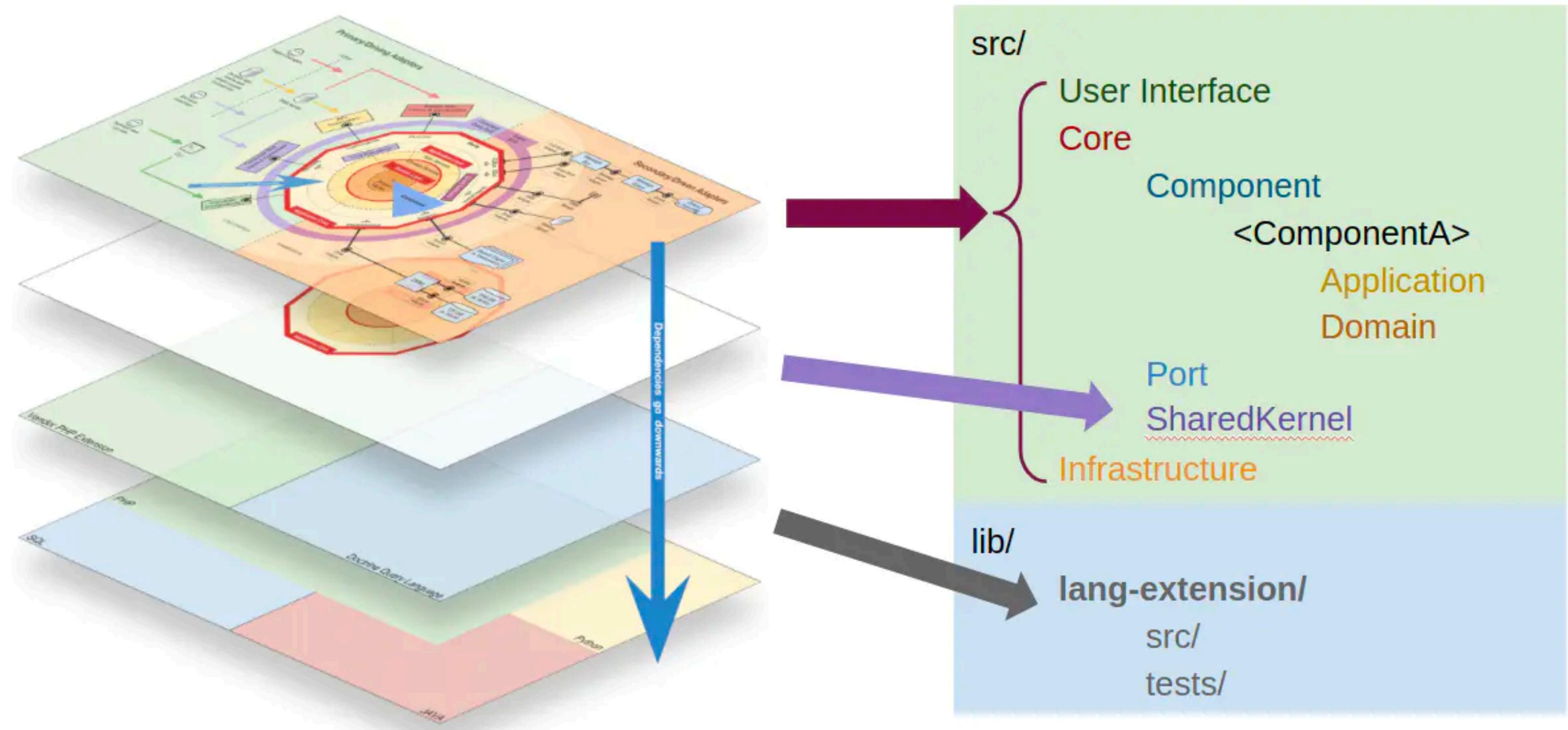
Application UI, Core and Adapters

Shared Kernel

Userland language extensions  
(owned by the project)

Programming languages  
and their native extensions

[www.herbertograca.com](http://www.herbertograca.com)

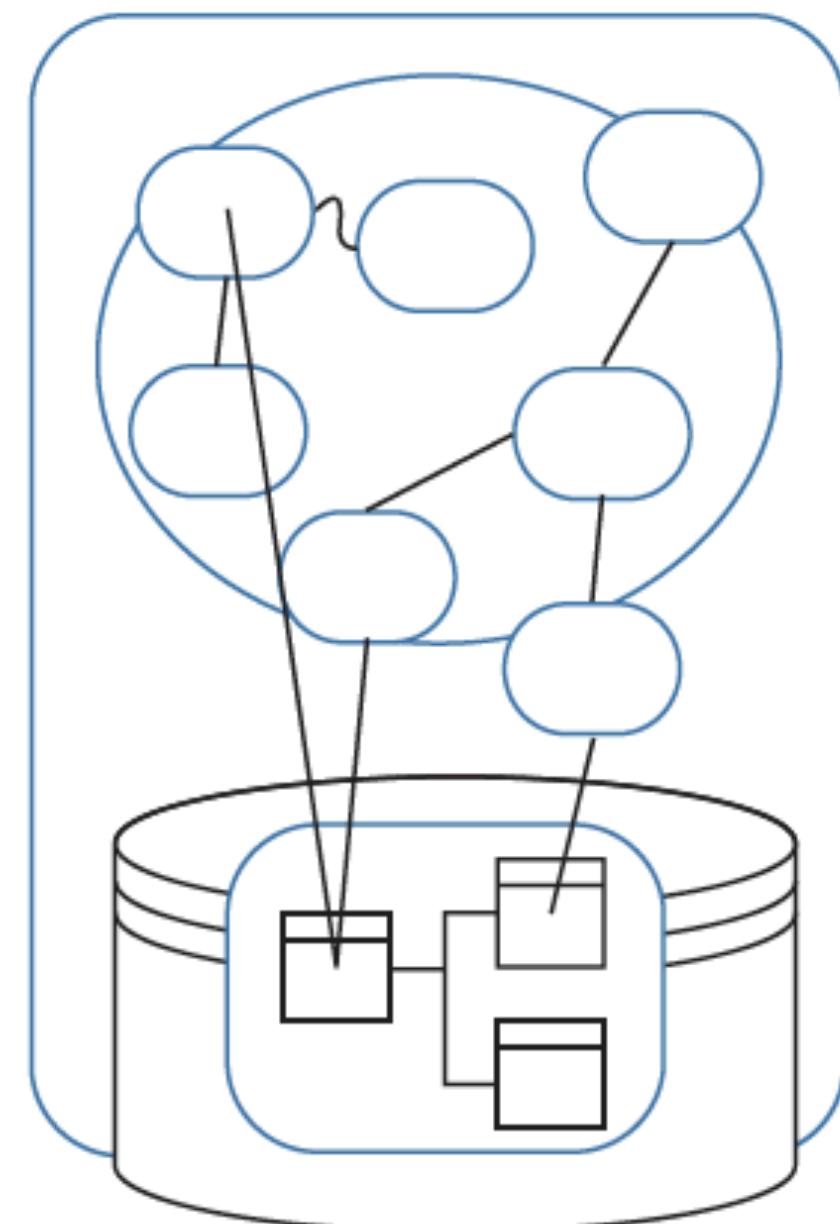


<https://www.jianshu.com/p/dd992f3fe370>

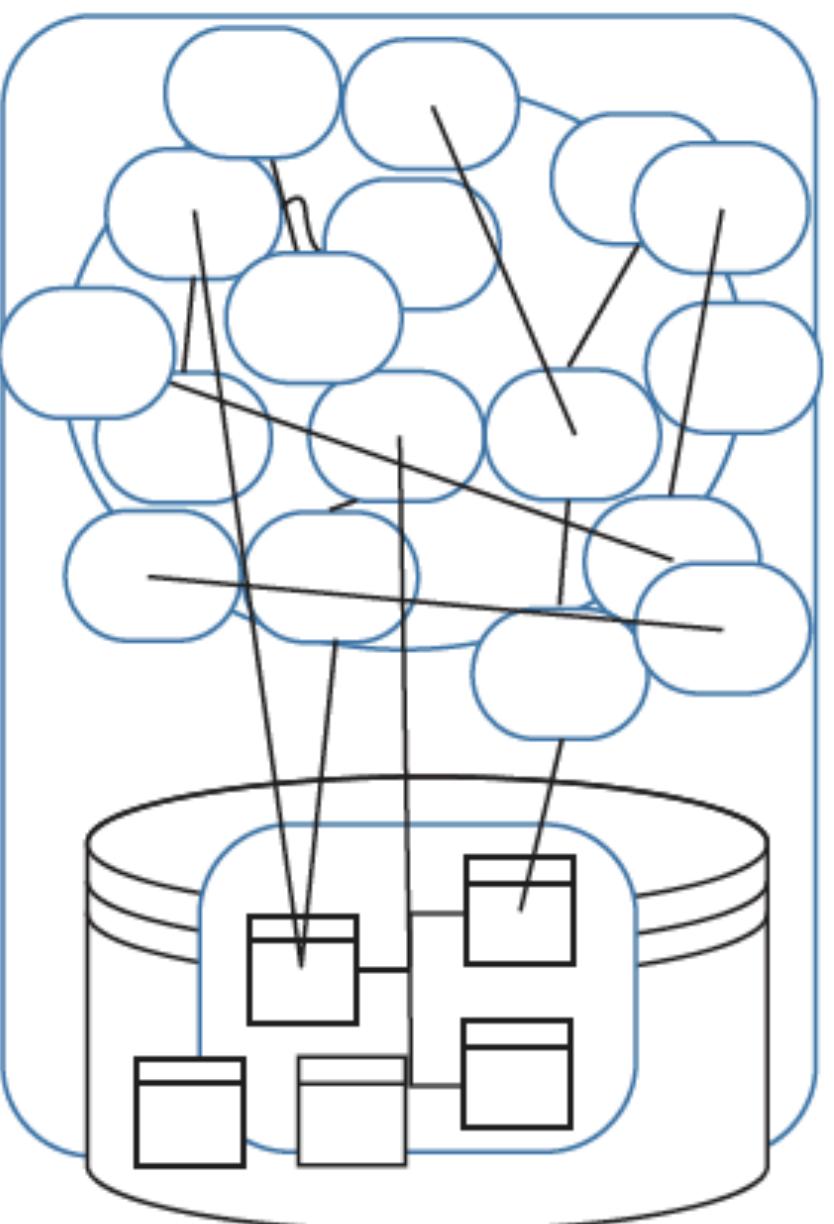
# DDD概述

---

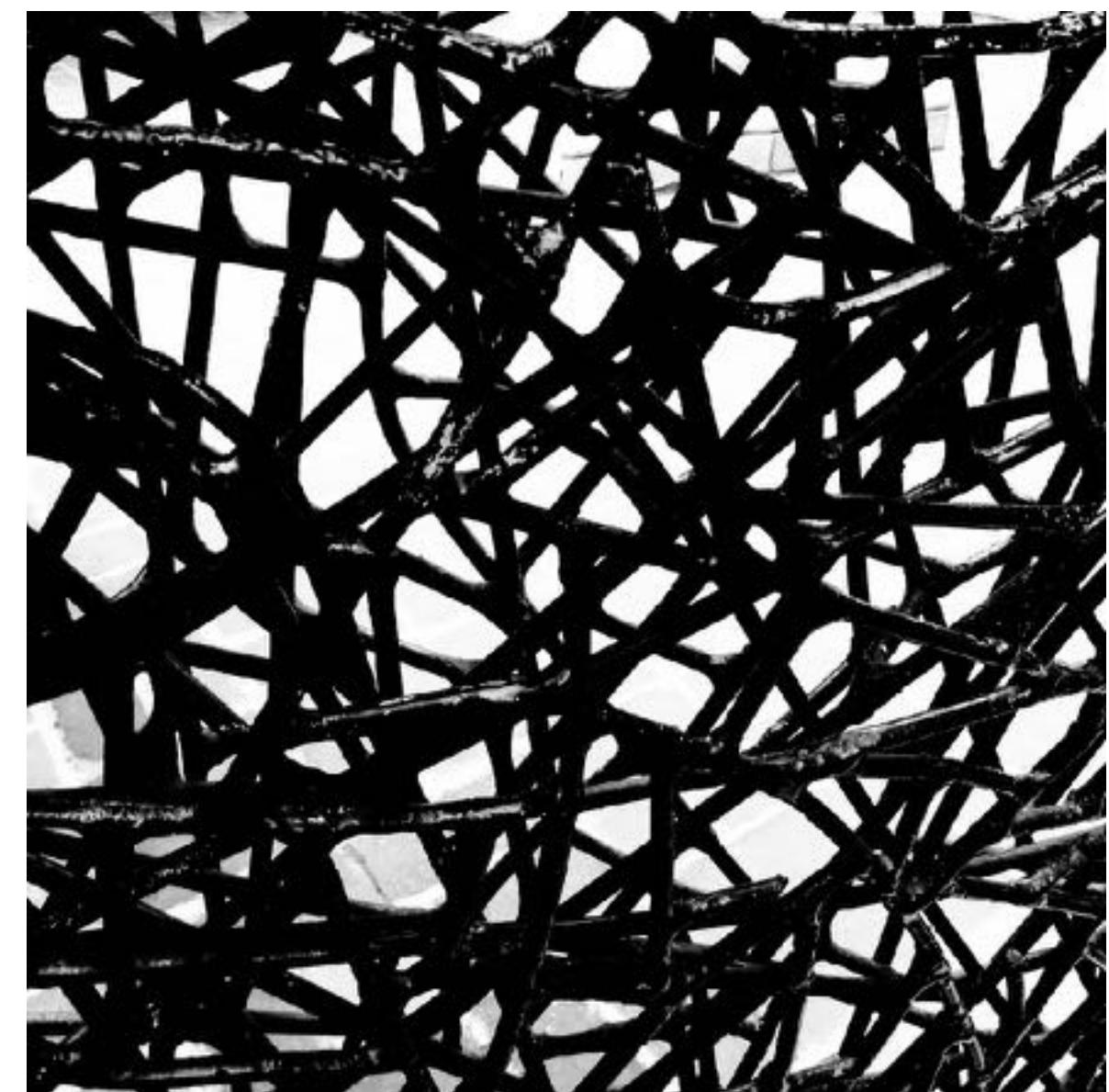
# 软件系统因不确定性必然走向腐坏



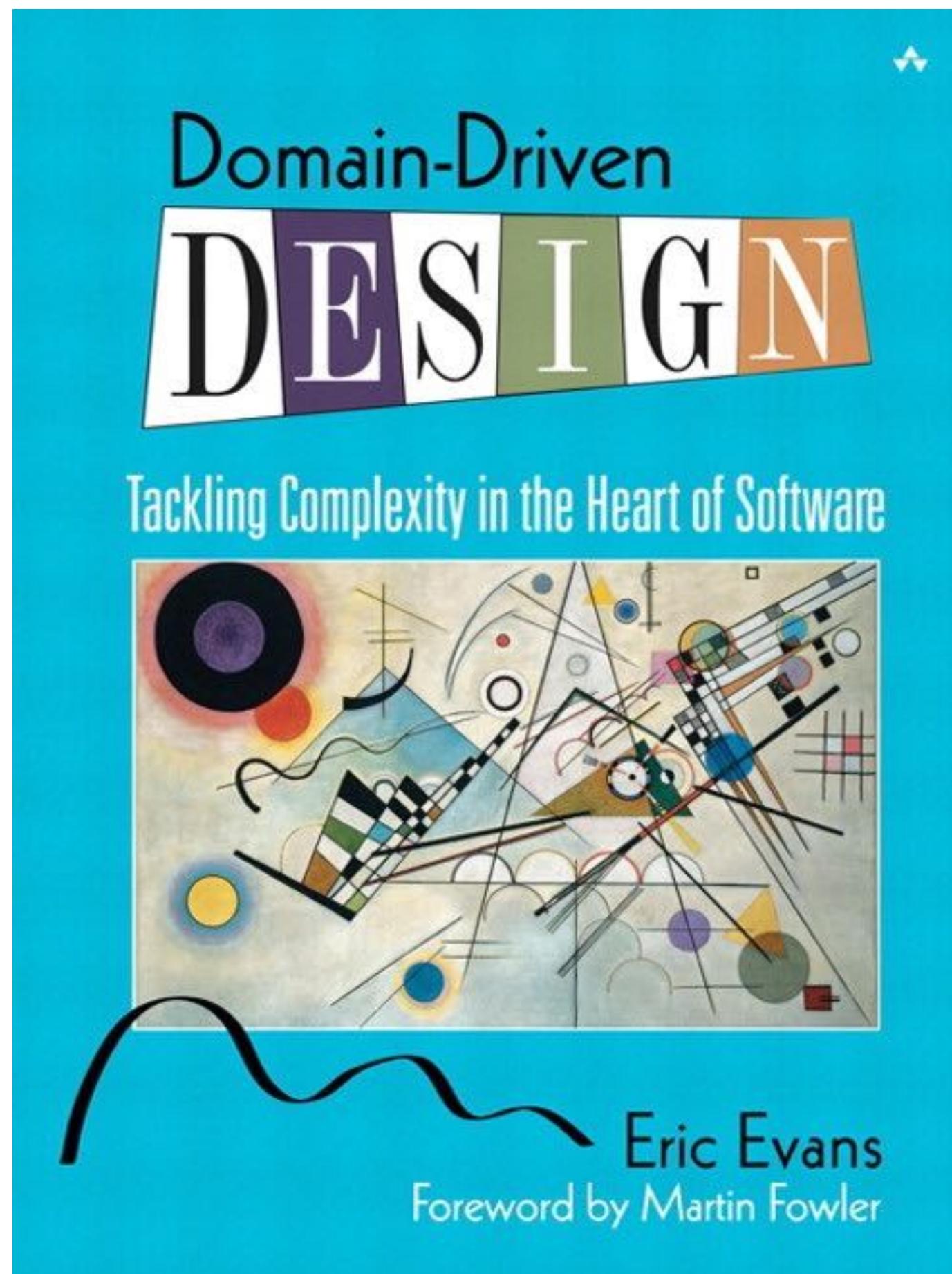
Initial software  
incarnation fast to  
produce



Over time without care  
and consideration,  
software turns to ball  
of mud



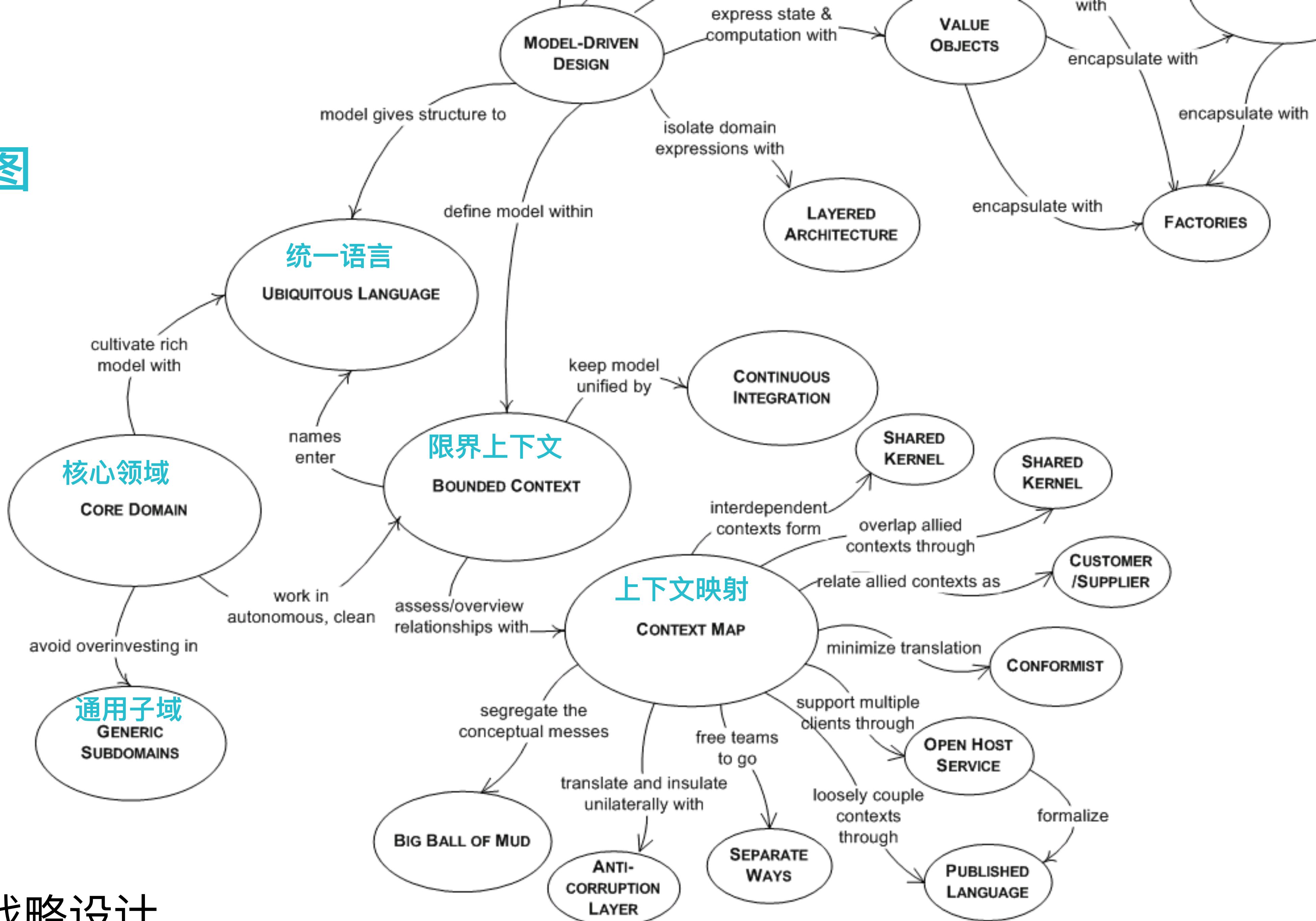
## 什么是领域驱动设计



►领域驱动设计是一种处理高度复杂域的设计思想，试图分离技术实现的复杂性，围绕**业务概念**构建**领域模型**来控制业务的复杂性，以解决软件难以理解，难以演化等问题。团队应用它可以成功地开发复杂业务软件系统，使系统在增大时仍然保持敏捷。

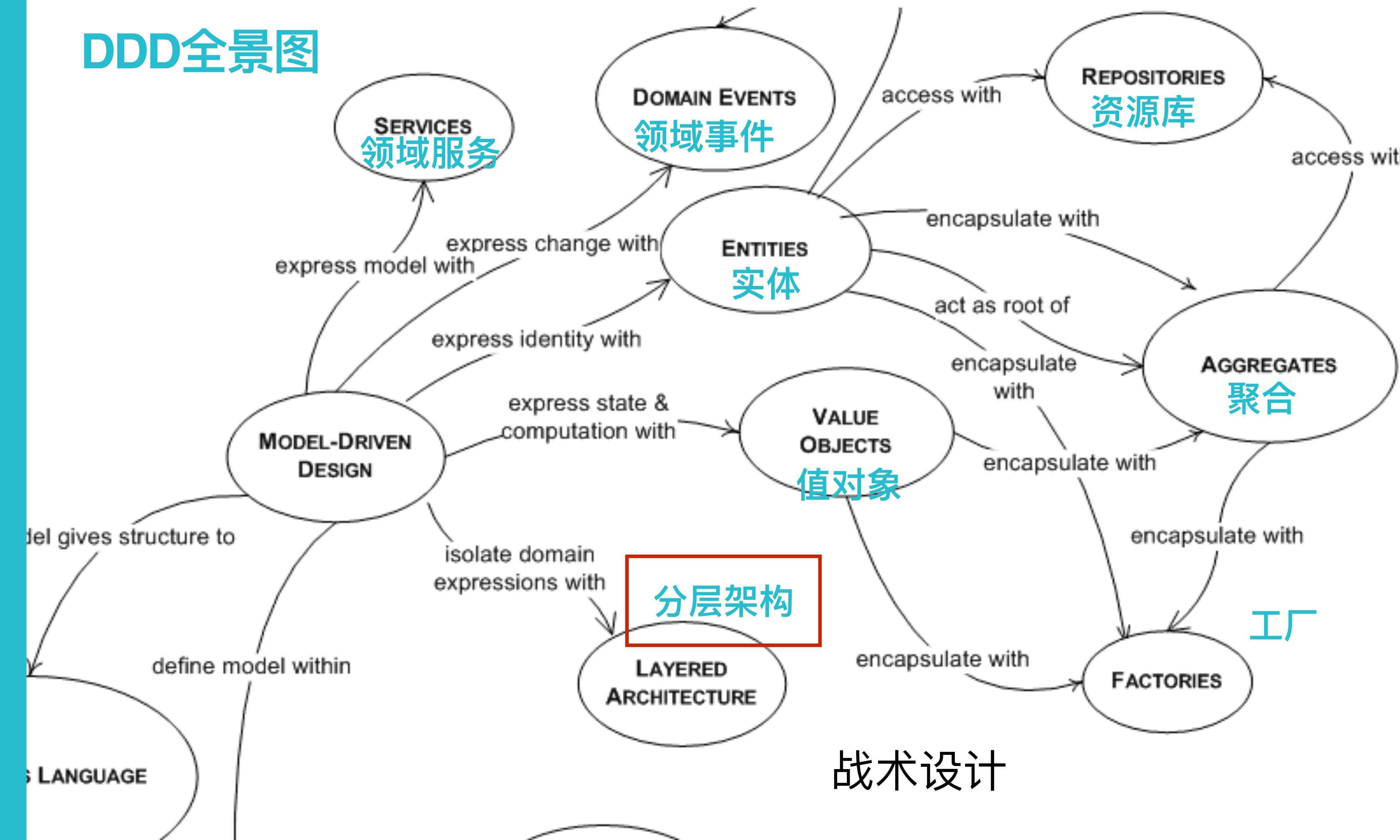
# DDD概述

## DDD全景图

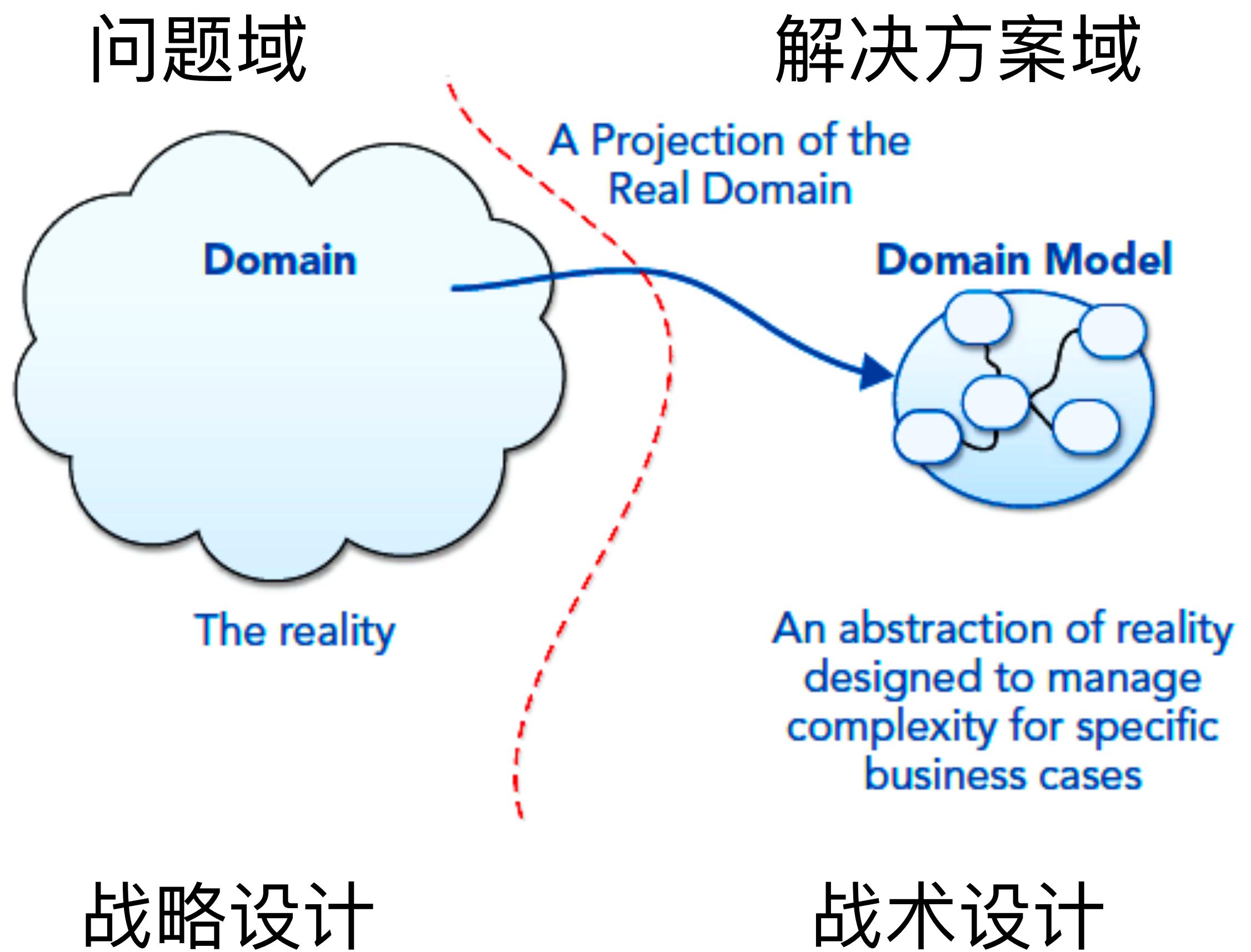


# DDD概述

## DDD全景图



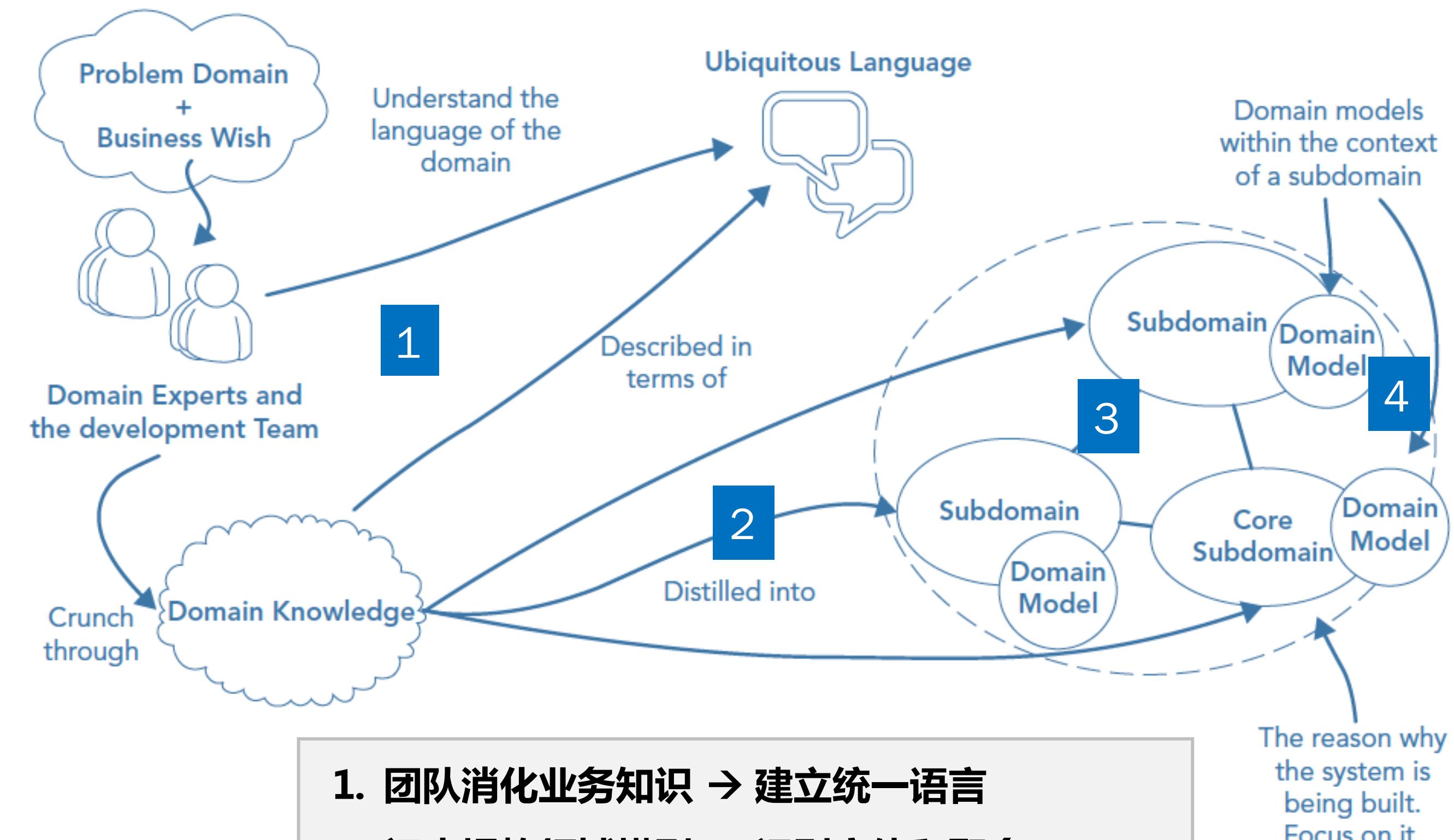
## 领域驱动设计的基本思路



DDD认为：

1. 软件项目的主要聚焦点应该在领域和领域逻辑，不应该直接从需求开始设计数据库表结构
2. 复杂的领域设计应该基于领域模型，领域模型是封装了数据和行为的对象
3. 技术专家和领域专家一起进行创造性的合作，从而可以不断的切入问题领域的核心。

# 领域驱动设计四步走



1. 团队消化业务知识 → 建立统一语言
2. 初步提炼领域模型 → 识别实体和聚合
3. 分解领域模型复杂度 → 识别限界上下文
4. 细分领域模型内部元素 → 识别事件和命令

## 领域驱动设计三原则



**P1: 聚焦核心领域**

*Focus on your core domain.*



**P2: 通过协作迭代式探索模型**

*Iteratively explore models in a creative collaboration of domain practitioners and software practitioners.*



**P3: 在限界上下文内统一语言**

*Speak a Ubiquitous Language within an explicitly Bounded Context.*

# 事件风暴

---

# 事件风暴

## 简介

### 简介

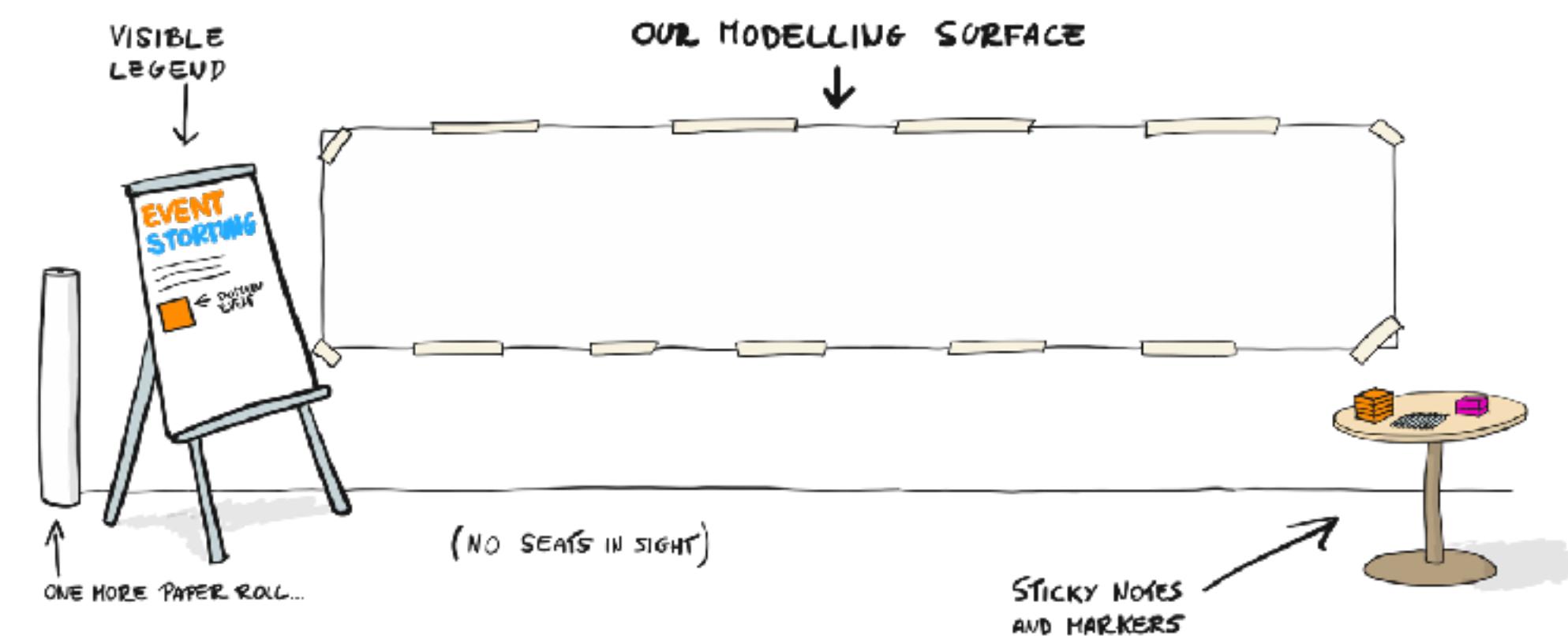
**Event Storming**是一种领域建模的实践，最初由Alberto Brandolini 开发，经过DDD社区和TW的很多团队实践验证后，于2015年11月进入ThoughtWorks技术雷达。**Event Storming**被验证是一种快速探索复杂业务领域的方法：

- › **Powerful**: 可以让实践者在数小时内理解复杂的业务模型
- › **Engaging**: 把带着问题的人和拥有答案的人共聚一堂构建模型
- › **Efficient**: 跟DDD的实现模型高度一致，并能快速发现Aggregate 和 Bounded Context
- › **Easy**: 标记都很简单，没有复杂的UML
- › **Fun**

<http://eventstorming.com/>

### 活动准备

- › 正确的人：业务人员，领域专家，技术人员，架构师，测试人员等关键角色都要参与其中
- › 开放空间：有足够的空间可以将事件流可视化，让人们可以交互讨论
- › 彩色即时贴：至少三种颜色



### 什么是领域事件

“任何的业务事件都会以某种数据的形式留下足迹。我们对于事件的追溯可以通过对数据的追溯来完成。……你无法回到从前去看看到底发生了什么，但是却可以在单据的基础上，一定程度的还原当时事情发生的场景。当我们把这些数据的足迹按照时间顺序排列起来，我们几乎可以清晰的推测出这个在过往的一段时间内到底发生了那些事情”

### 要点

- › 业务关心的事件
- › 用“已发生”时态描述
- › 有时间顺序

### (Domain) Events

*It really became clear to me in the last couple of years that we need a new building block and that is the Domain Event.*

Eric Evans

*An event is something that has happened in the past.*

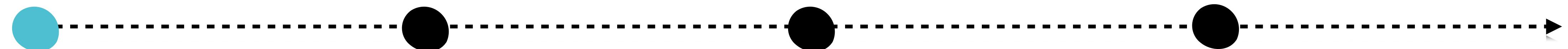
Greg Young

*A domain event ... captures the memory of something interesting which affects the domain*

Martin Fowler

# 事件风暴

## 步骤



### 事件风暴

### 事件风暴

- › 领域专家介绍业务
- › 参与者可以任意提问
- › 参与者根据自己对业务的理解，将领域事件写在橙色即时贴上
- › 每个即时贴一个事件
- › 事件采用“xx已xx”的格式，如“订单已创建”

### 命令风暴

### 事件排序

- › 参与人员将自己的事件贴纸贴在白板纸上
- › 每个事件从左到右按时间顺序排列
- › 不同参与者的事件需保证相对顺序

### 寻找聚合

### 持续探索

#### WHERE ARE DOMAIN EVENTS COMING FROM?

MAYBE AN ACTION  
STARTED BY A USER



MAYBE THEY'RE COMING  
FROM AN EXTERNAL SYSTEM



MAYBE THEY'RE JUST THE  
RESULT OF TIME PASSING

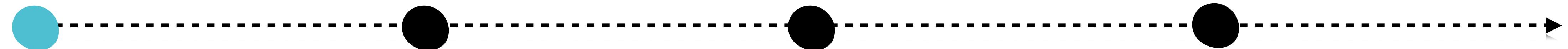


OR MAYBE, THEY'RE JUST  
THE CONSEQUENCE  
OF ANOTHER DOMAIN EVENT



# 事件风暴

## 步骤



### 事件风暴

#### 事件风暴

- › 领域专家介绍业务
- › 参与者可以任意提问
- › 参与者根据自己对业务的理解，将领域事件写在橙色即时贴上
- › 每个即时贴一个事件
- › 事件采用“xx已xx”的格式，如“订单已创建”

### 命令风暴

上架

商品  
已创建

库存  
已增加

### 寻找聚合

购买

订单  
已创建

库存  
已扣减

订单  
已支付

订单  
已撤销

### 持续探索

配送

仓储  
已发货

仓储  
已配货

订单  
已接收

收货

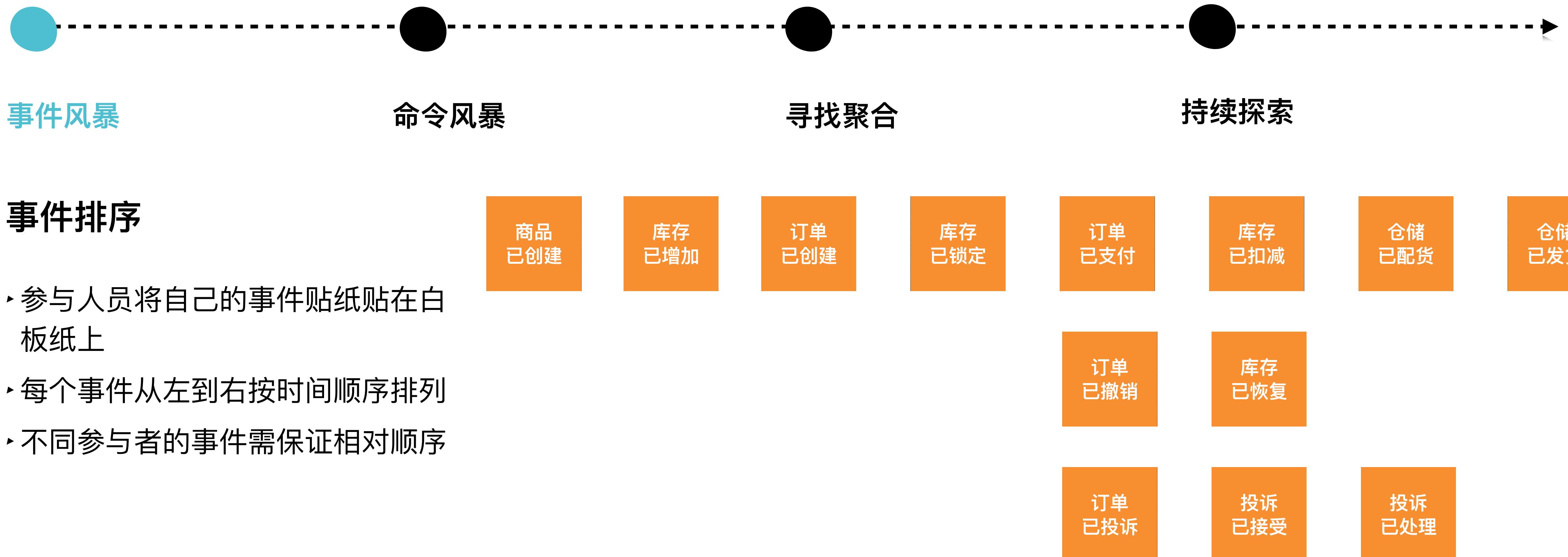
订单  
已投诉

投诉  
已接受

投诉  
已处理

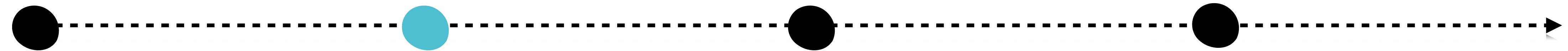
# 事件风暴

## 步骤



# 事件风暴

## 步骤



事件风暴

命令风暴

寻找聚合

持续探索

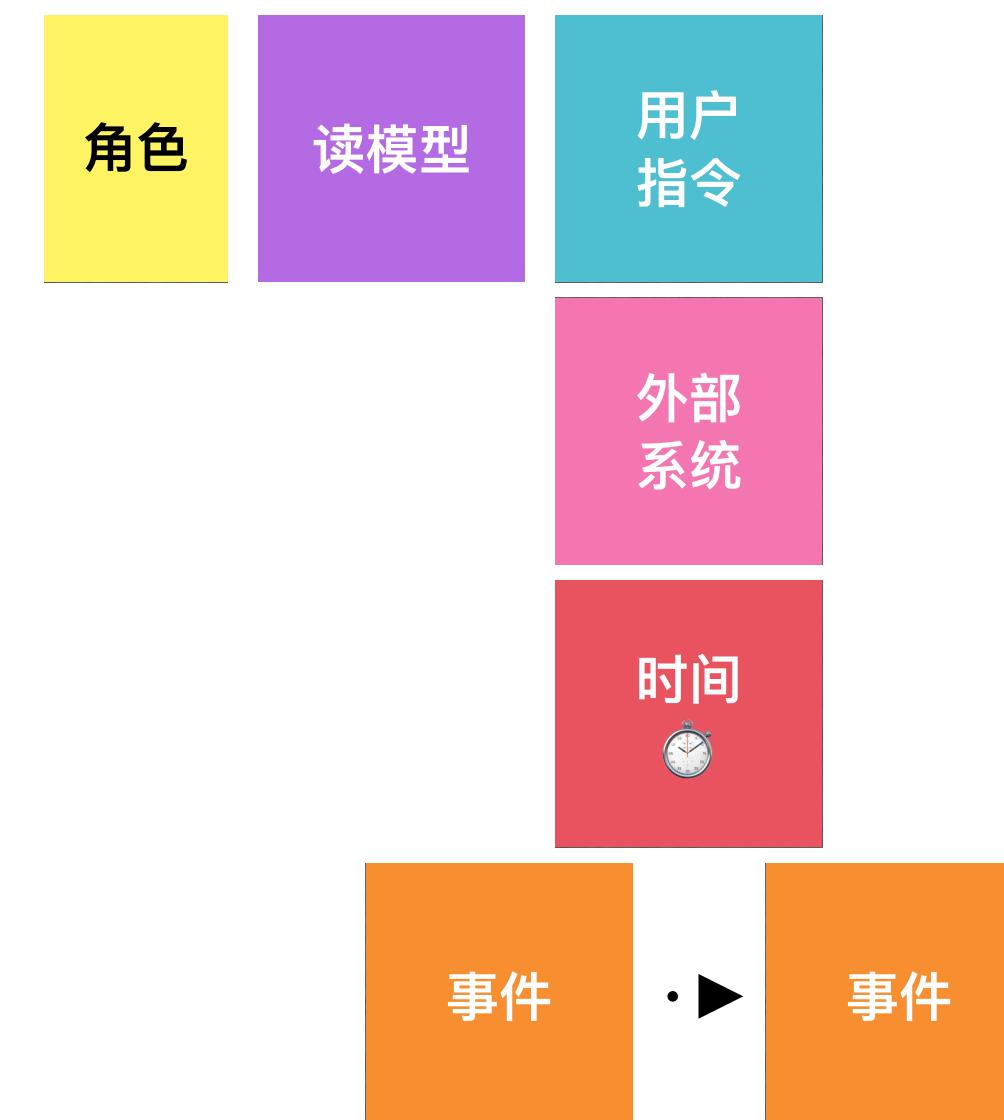
### 决策

- › 将发出命令的决策写在蓝色即时贴上
- › 将决策贴在所产生的事件左边
- › 有的决策可能产生多个事件
- › 在这个过程中可以识别出触发命令的用户（黄色即时贴）并进行标识

### 什么是决策

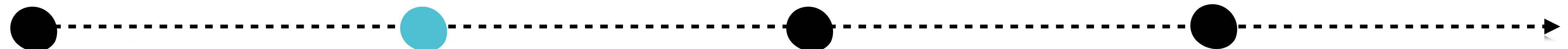
对可能产生事件的决策进行建模，决策可以是：

- › 用户的指令（蓝色）
- › 外部系统触发（粉红色）
- › 定时任务（深红色）



# 事件风暴

## 步骤



事件风暴

命令风暴

寻找聚合

持续探索

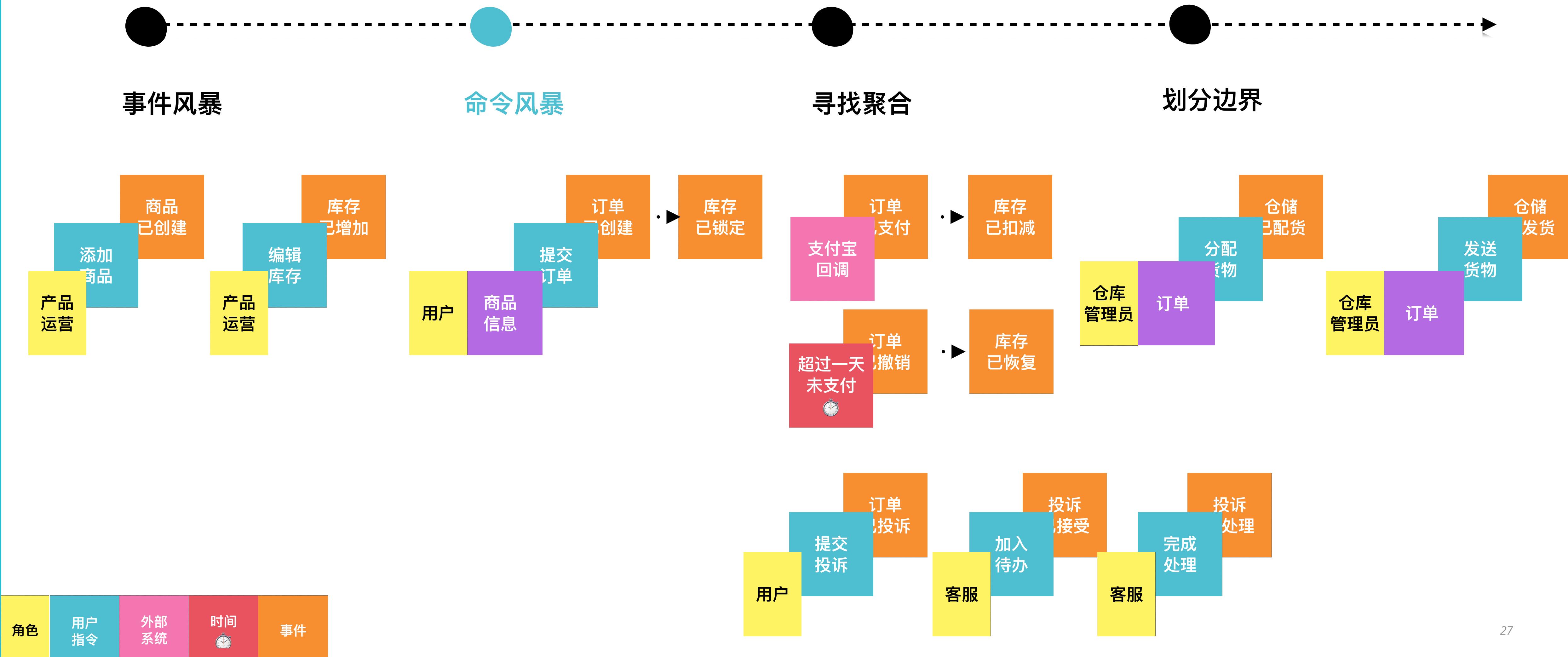
### 命令风暴

- › 将发出命令的决策写在蓝色即时贴上
- › 将决策贴在所产生的事件左边
- › 有的决策可能产生多个事件
- › 在这个过程中可以识别出触发命令的用户（黄色即时贴）并进行标识



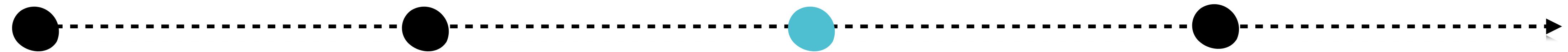
# 事件风暴

## 步骤2 - 结果展示



# 事件风暴

## 步骤



事件风暴

命令风暴

寻找聚合

持续探索

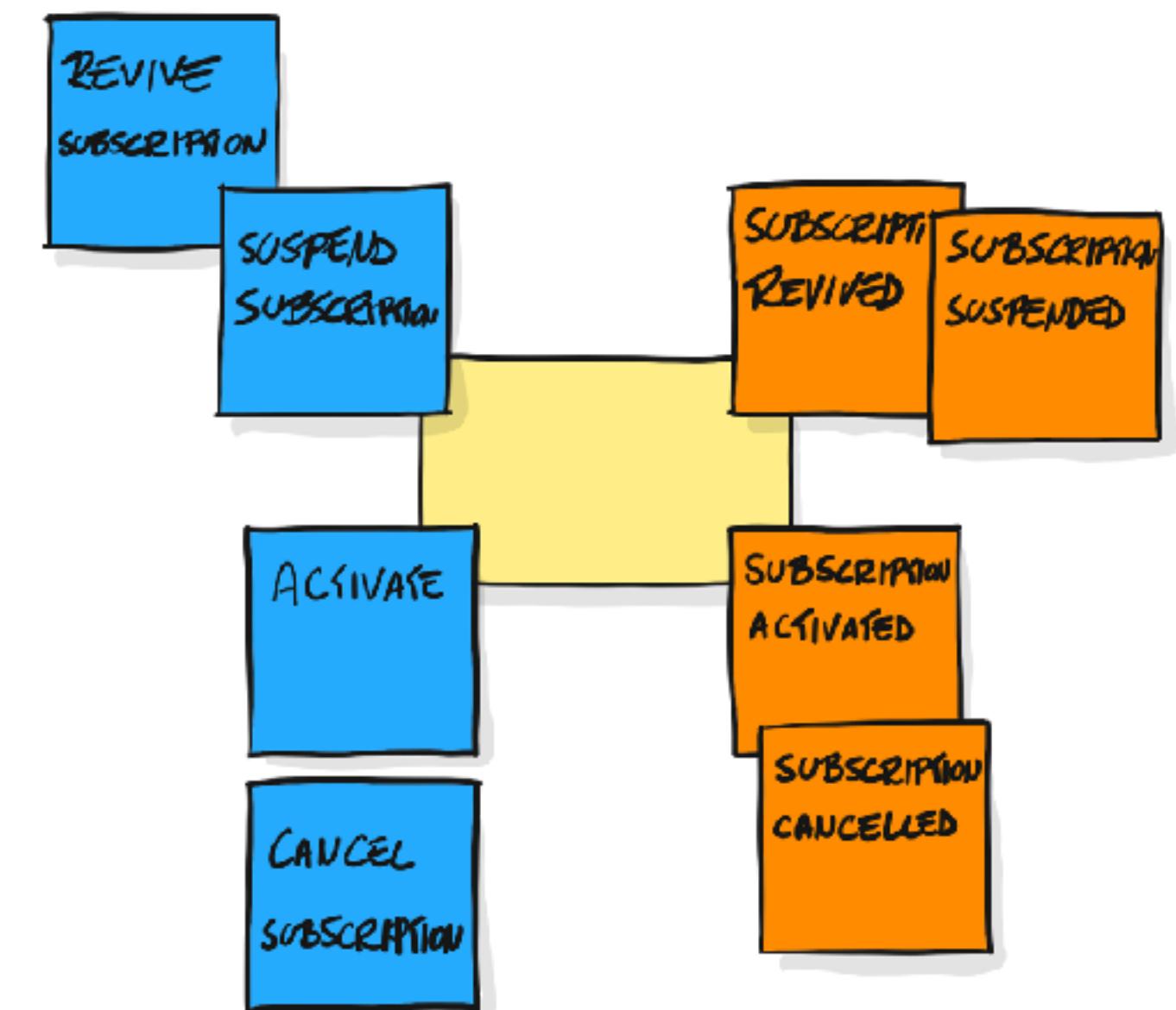
### 寻找聚合

- 对命令和事件进行划分找到聚合边界
- 利用聚合定义进行确认
- 通过业务上统一概念，识别出分布在时间轴不同位置的同一个聚合
- 利用 *persona* 帮助判断聚合
- 对聚合使用大的黄色即时贴进行标记

### 什么是聚合

在领域驱动设计中，聚合是一组相关领域对象，其目的是要确保业务规则在领域对象的各个生命周期都得以执行：

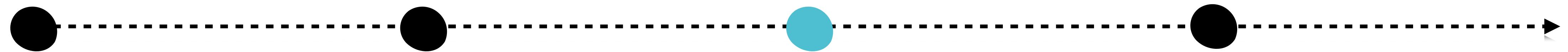
- 聚合边界内保证业务不变性 (*invariant*)
- 只能通过聚合根修改边界内的对象
- 聚合根有全局标识



聚合接受决策，产生事件

# 事件风暴

## 步骤



### 事件风暴

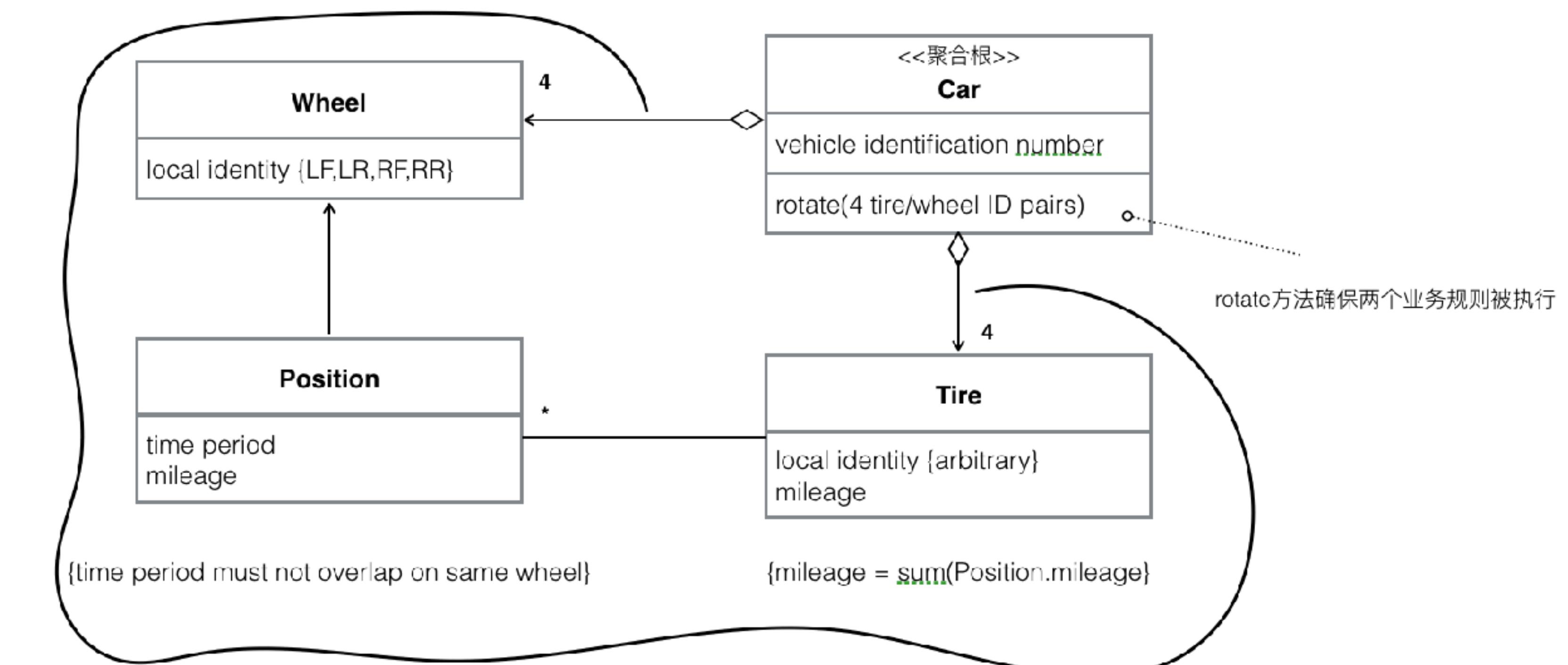
### 命令风暴

### 寻找聚合

### 持续探索

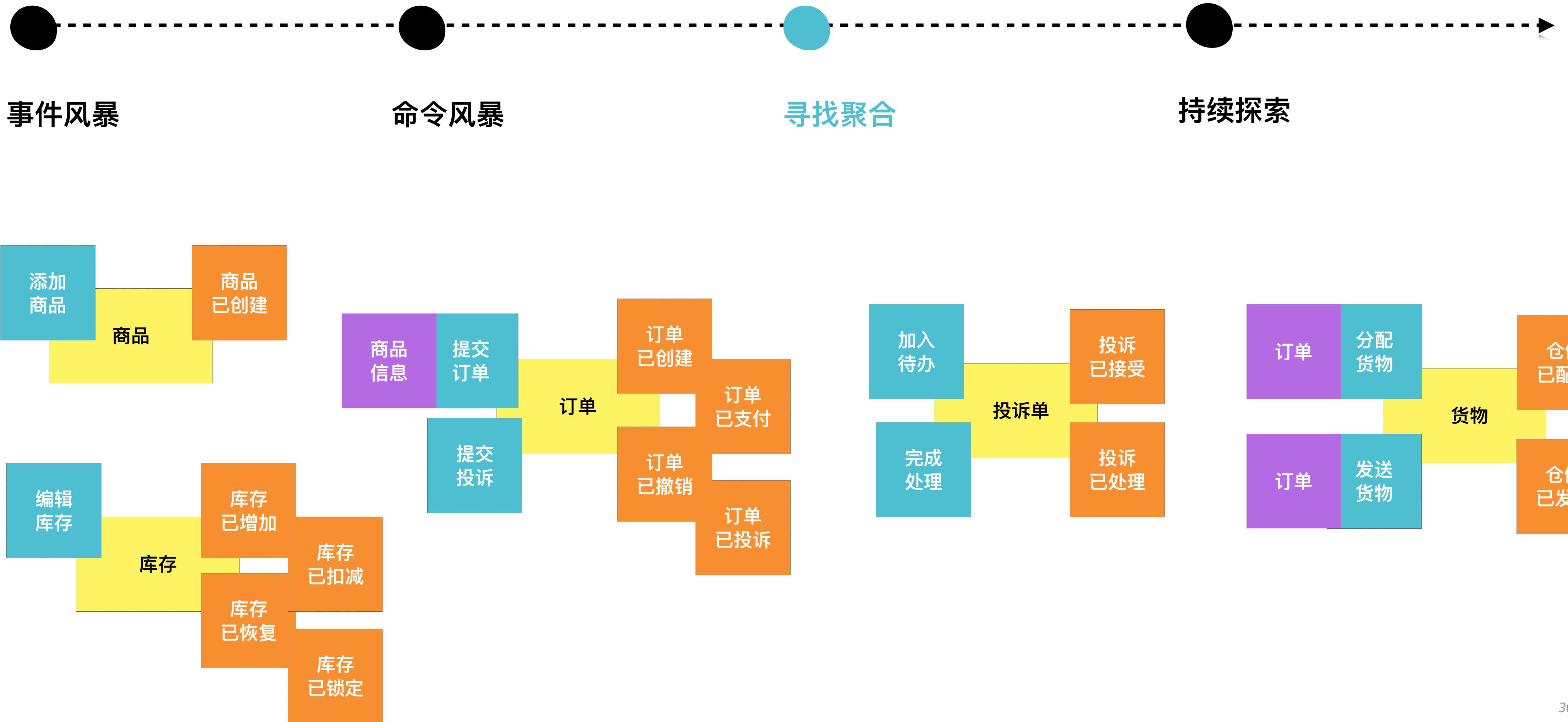
## 聚合根

- 外部对象只能引用聚合根
- 外部对象只能通过聚合根导航到边界内的对象
- 聚合根上的行为要确保业务不变性
- 只能通过聚合根修改边界内的对象
- 聚合根有全局标识
- 聚合内实体只能有局部标识



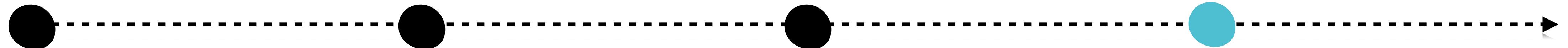
# 事件风暴

## 步骤



# 事件风暴

## 步骤

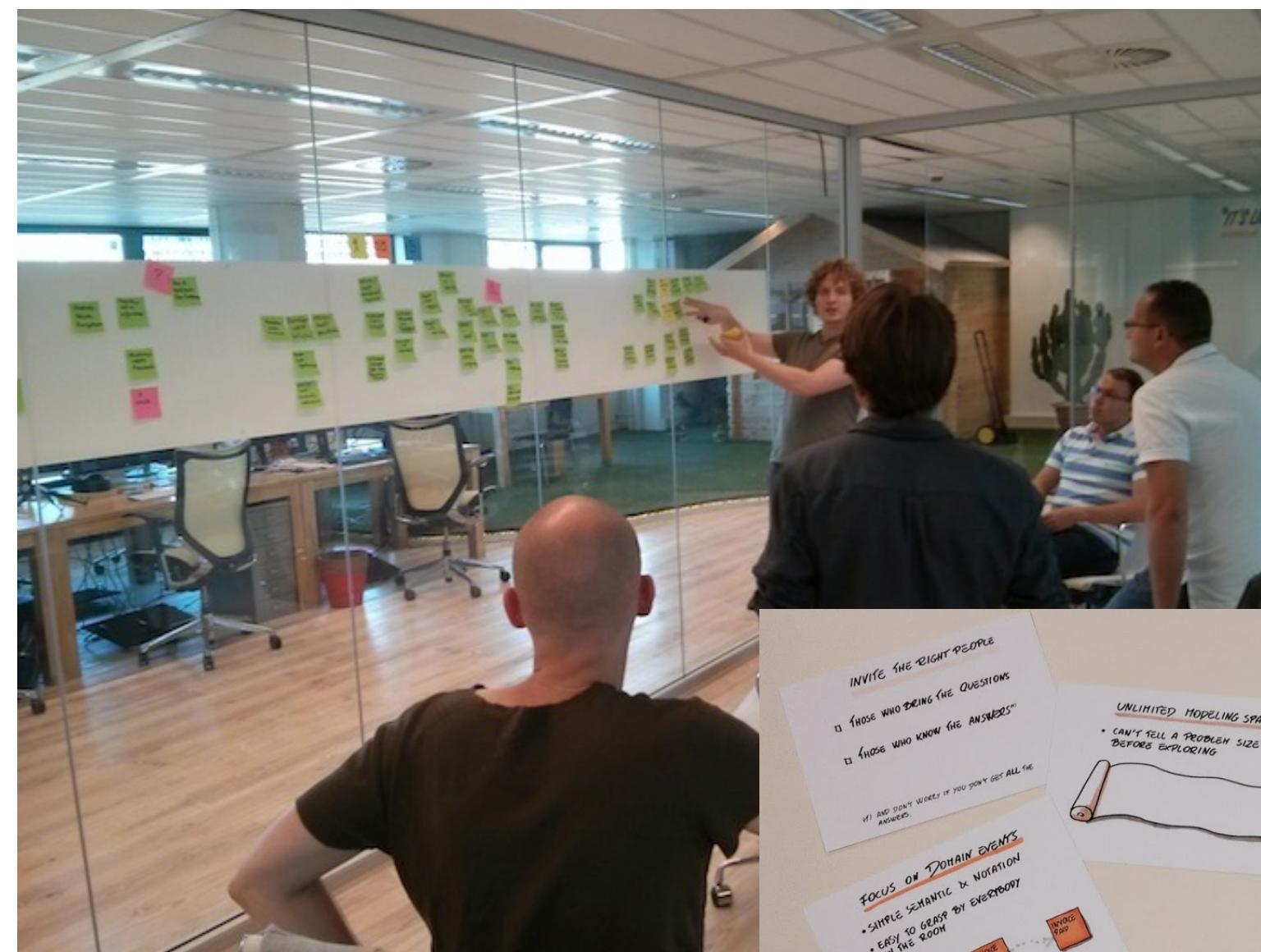


### 事件风暴

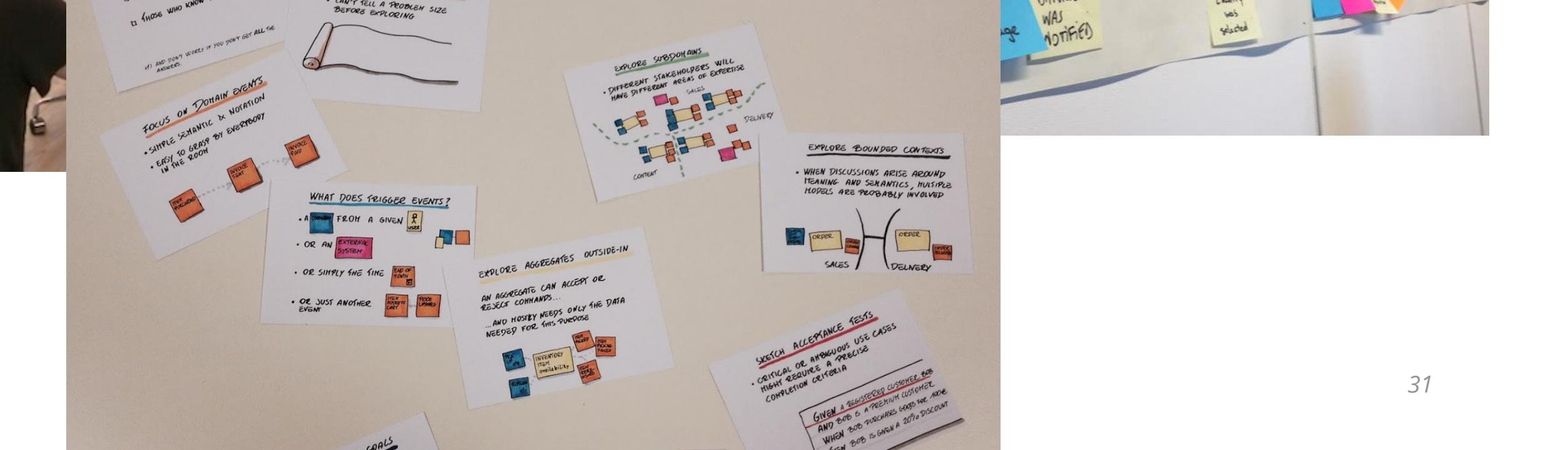
### 持续探索

- 探索子域
- 探索界限上下文
- 拟定用户画像
- 拟定关键验收测试
- 拟定关键读模型

### 命令风暴



### 寻找聚合



### 持续探索



# 界限上下文及映射

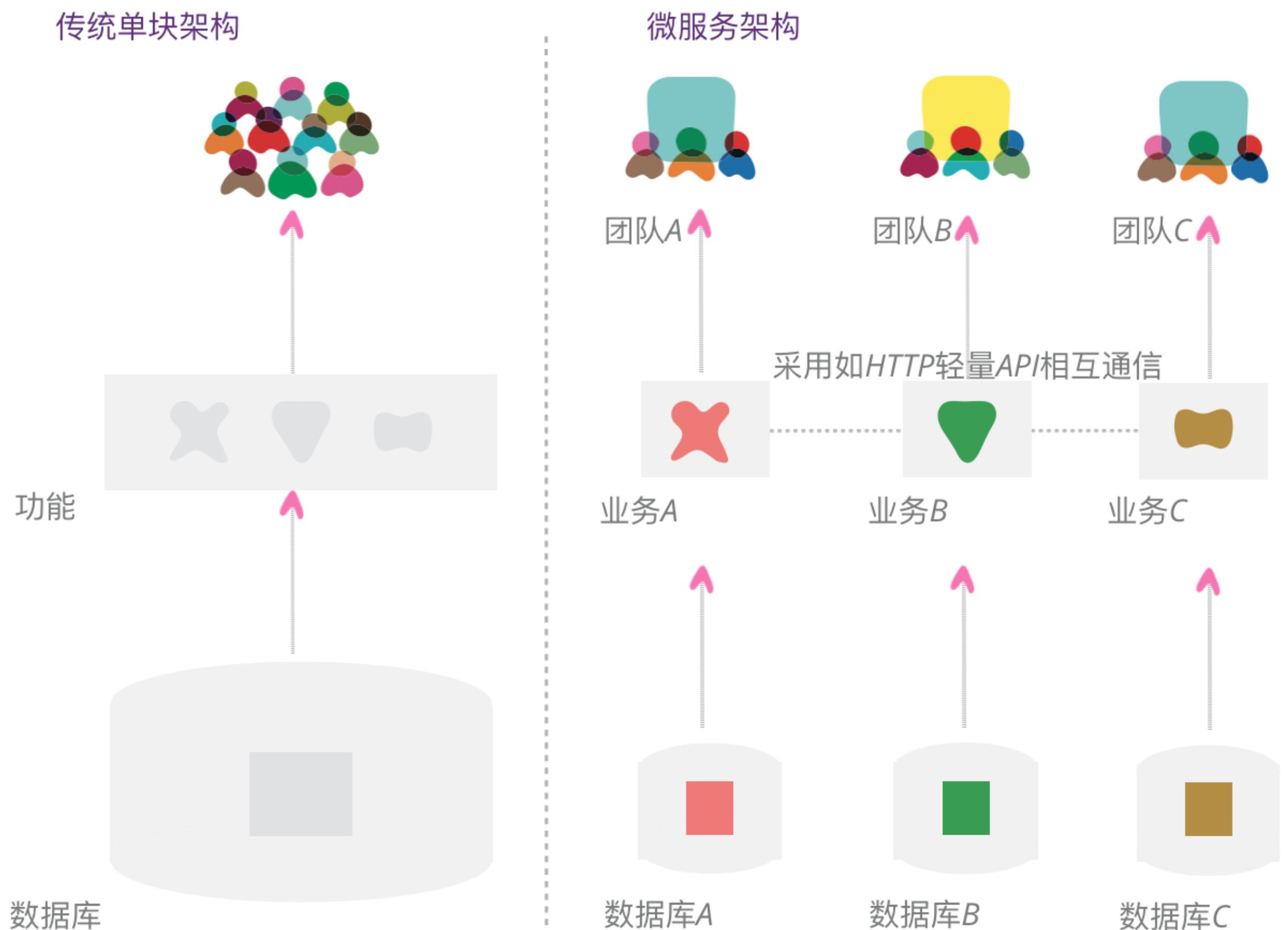
---

限界上下文及映射

# 领域驱动设计与微服务架构

WHY?

领域驱动设计的提出距今已经十多年，但真正火热起来大约是在2013年微服务架构被提出之后。



“  
微服务架构将业  
务和数据剥离  
”

每条业务做到4个独立：  
独立进程

以开发一组小型服务的方式来开发一个独立的应用系统，其中每个小型服务都运行在自己的进程中。  
独立部署

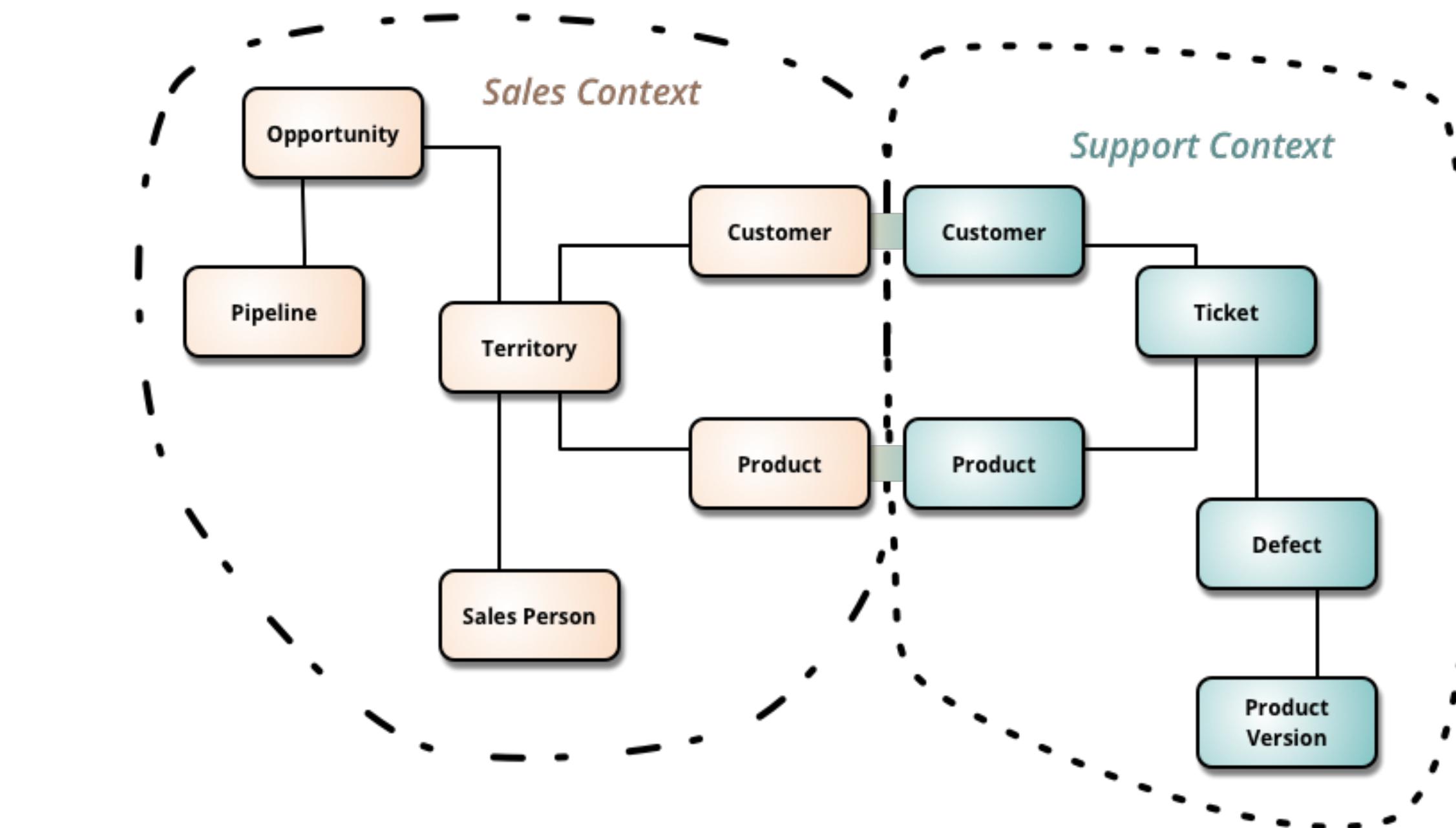
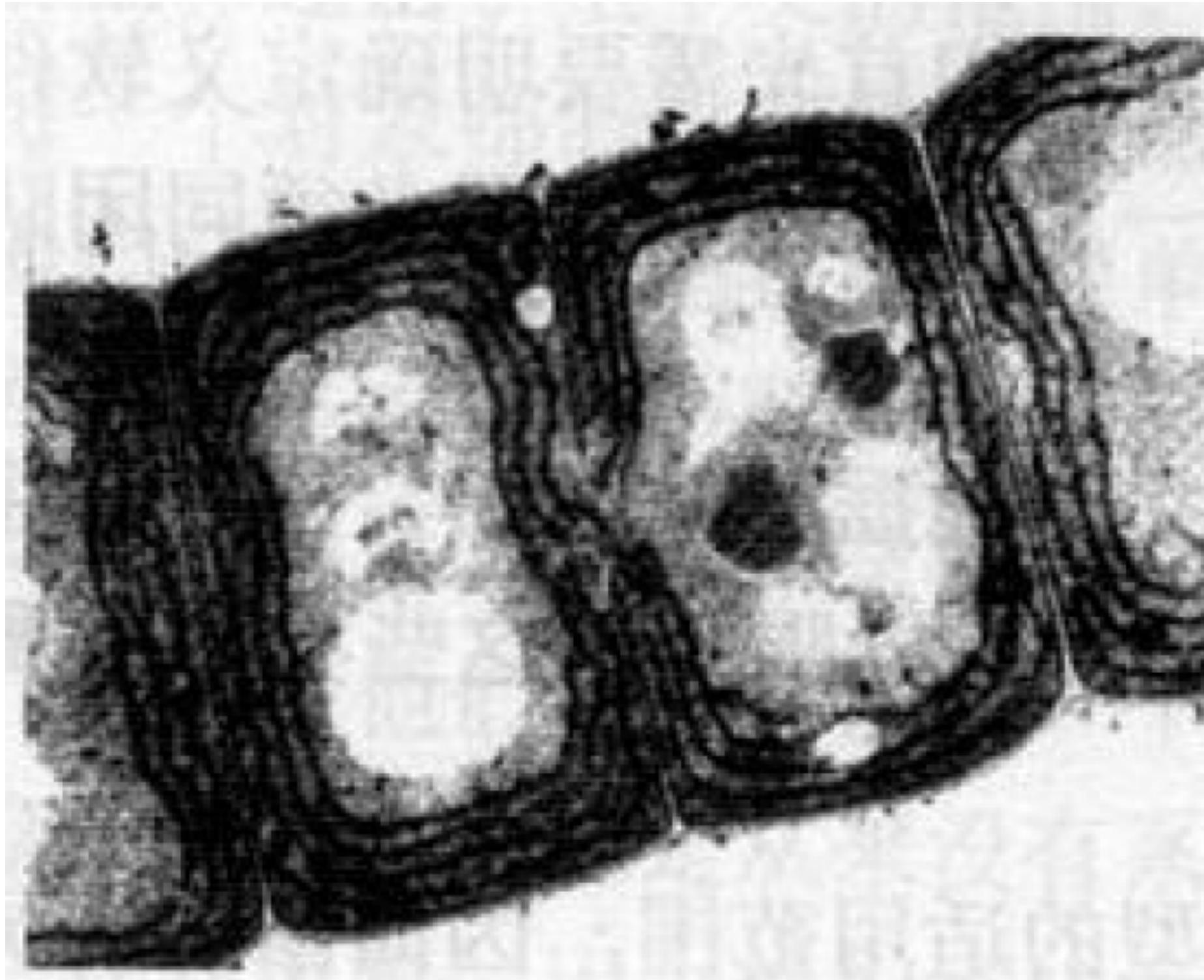
服务围绕业务功能进行构建，并能通过全自动的部署机制来进行独立部署  
独立技术

每条业务线可以使用不同的开发语言、数据存储技术，并保持最低限制的集中式管理  
独立团队

每条业务线均配备开发、测试、运维、DBA等更具生产力、更对结果负责的全功能团队

限界上下文及映射

## 什么是限界上下文



限界上下文主要用来封装通用语言和领域模型，显式地定义了领域模型的边界（所应用的上下文），以便保证模型的概念完整性。

## 限界上下文及映射

# 限界上下文地图（映射）

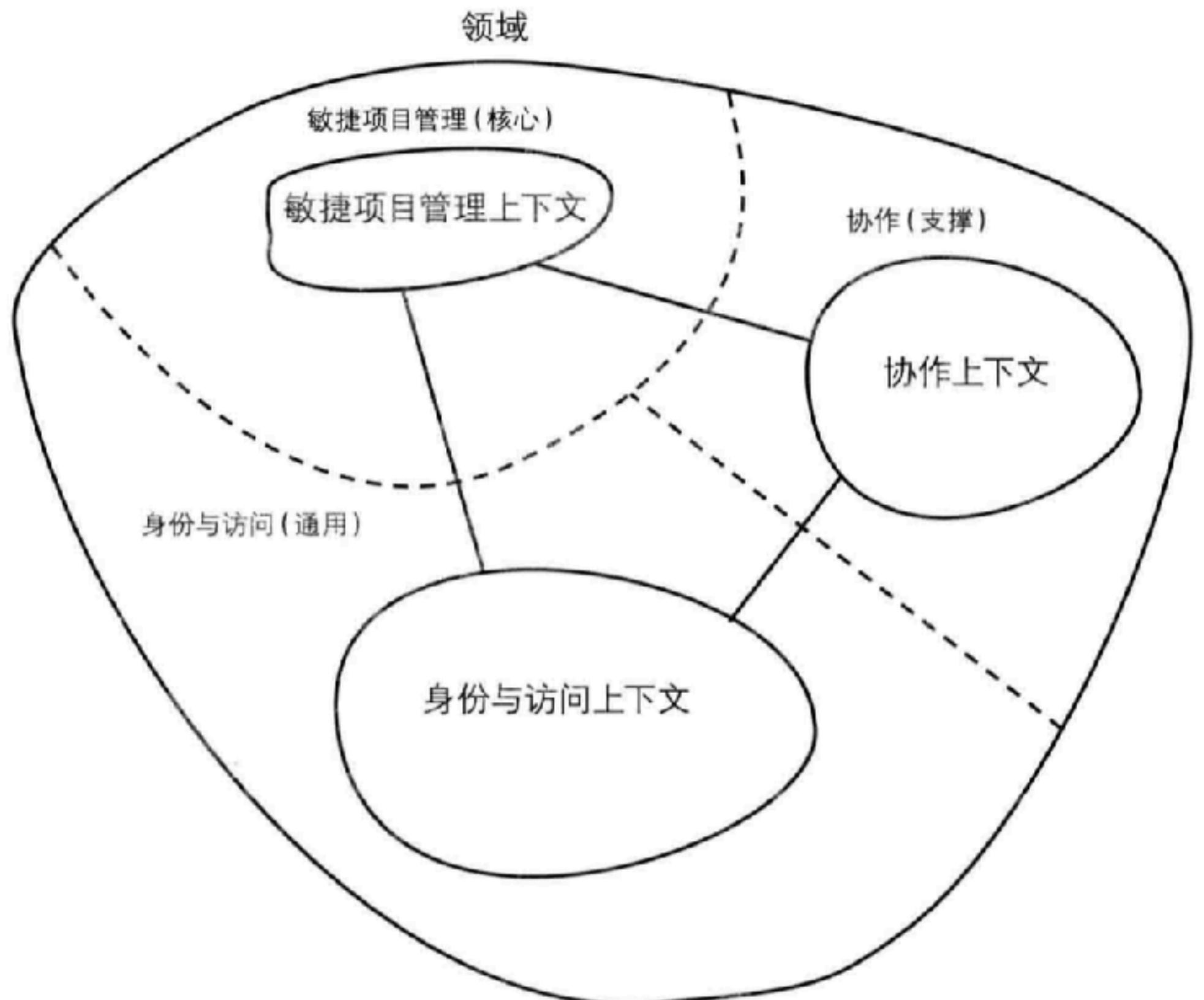


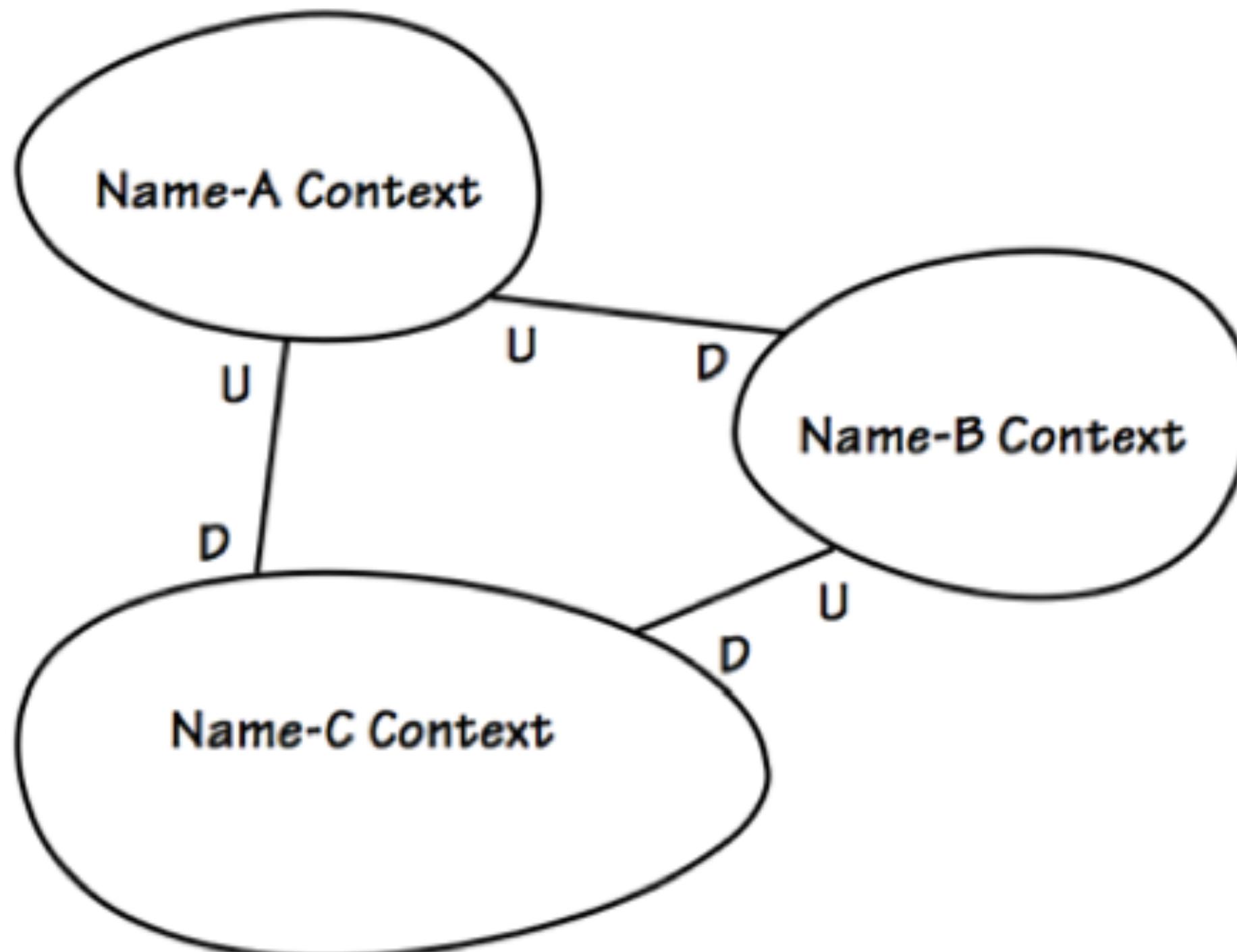
图 2.7 一个拥有清晰子域的示例限界上下文

- ▶当有很多上下文时，不同团队负责不同的上下文，如何保证有效的合作？

限界上下文及映射

## 上下文映射

通过定义不同上下文之间的关系，中创建的一个所有模型上下文的全局视图。



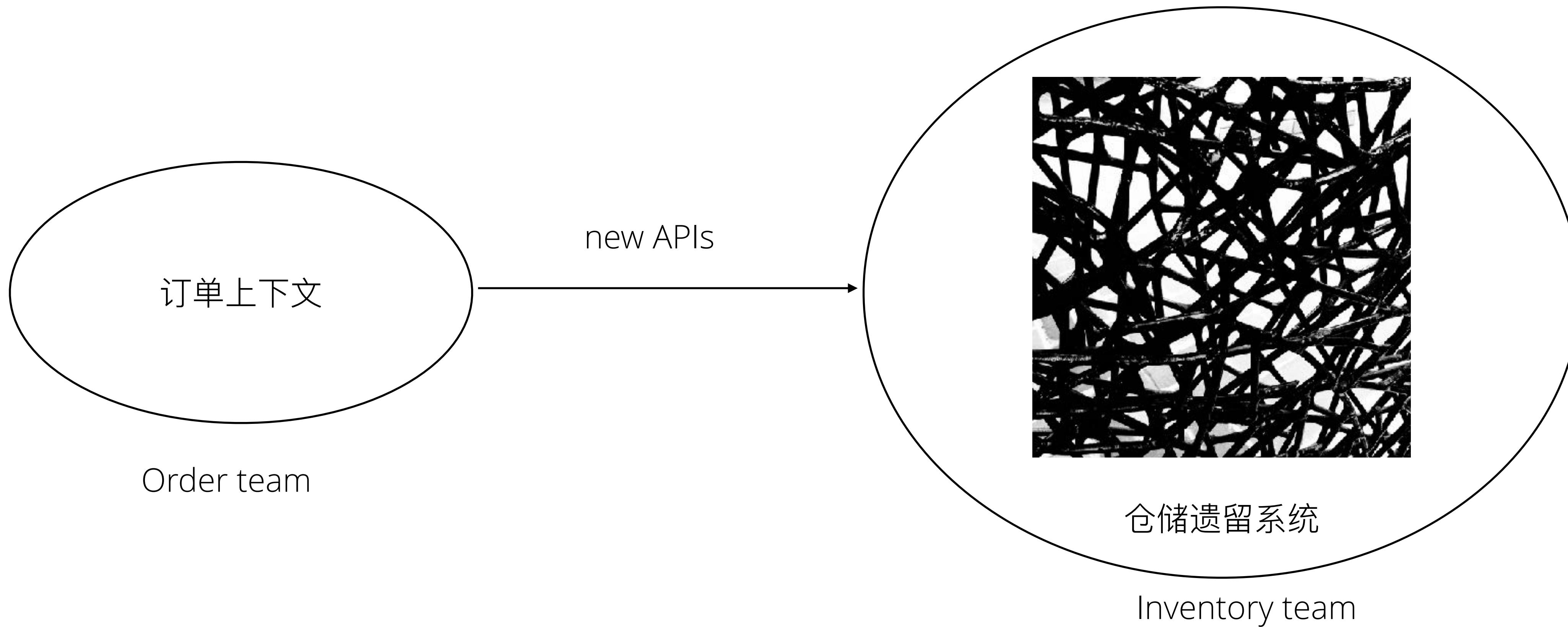
两个上下文之间的联系是有方向的：

“上游” (U | Upstream)

“下游” (D | Downstream)

限界上下文及映射

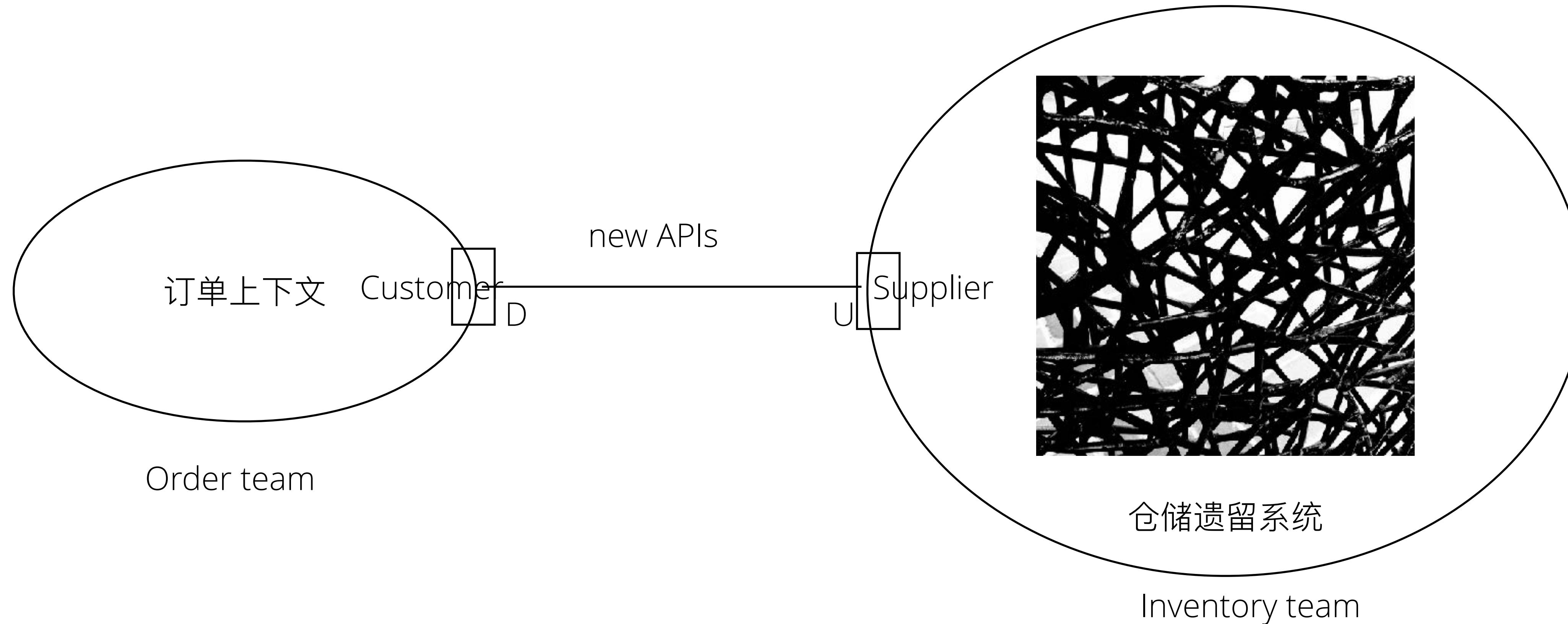
## 大泥球



## 限界上下文及映射

### 客户-供应方

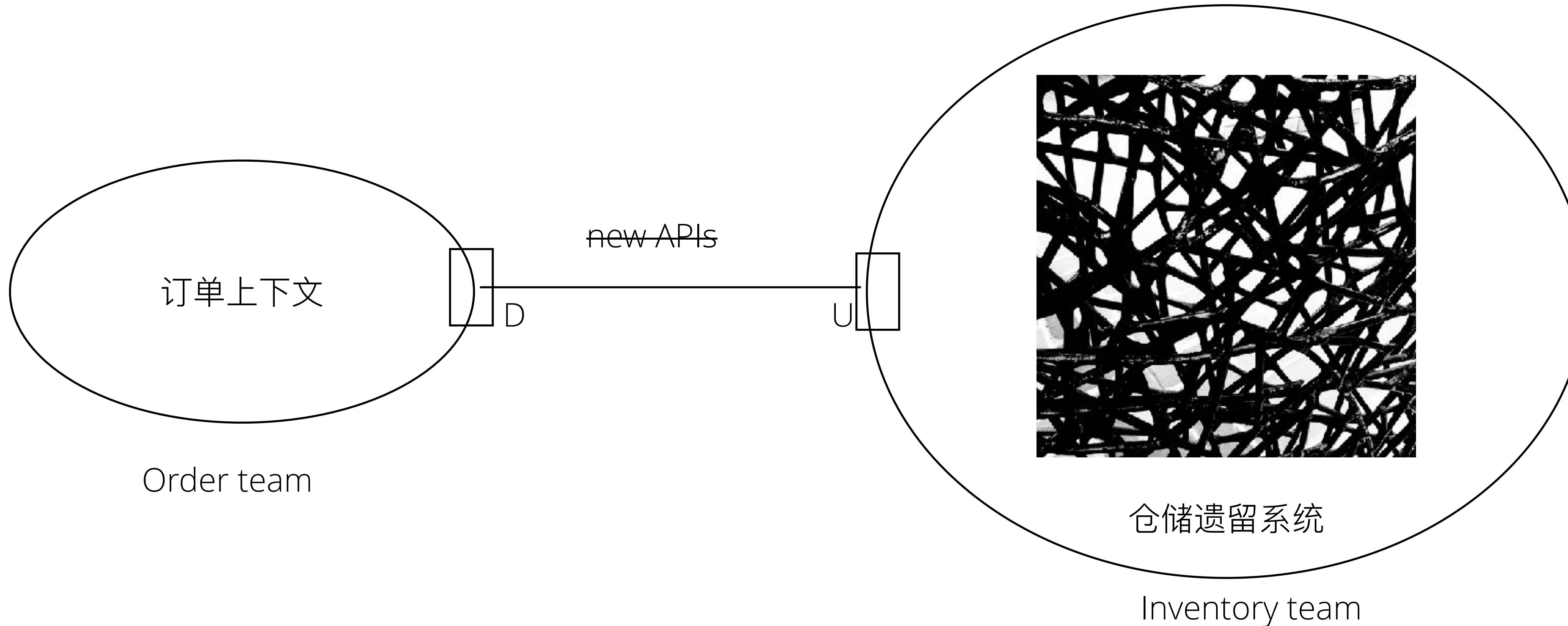
上游可能独立于下游团队完成开发也可能晚于开发。下游团队就会收到影响。所以上游团队的计划中，需要兼顾下游团队的需求，而下游团队需要尽早识别这种依赖关系，并找到解决方案。



限界上下文及映射

## 遵奉者 (Conformist)

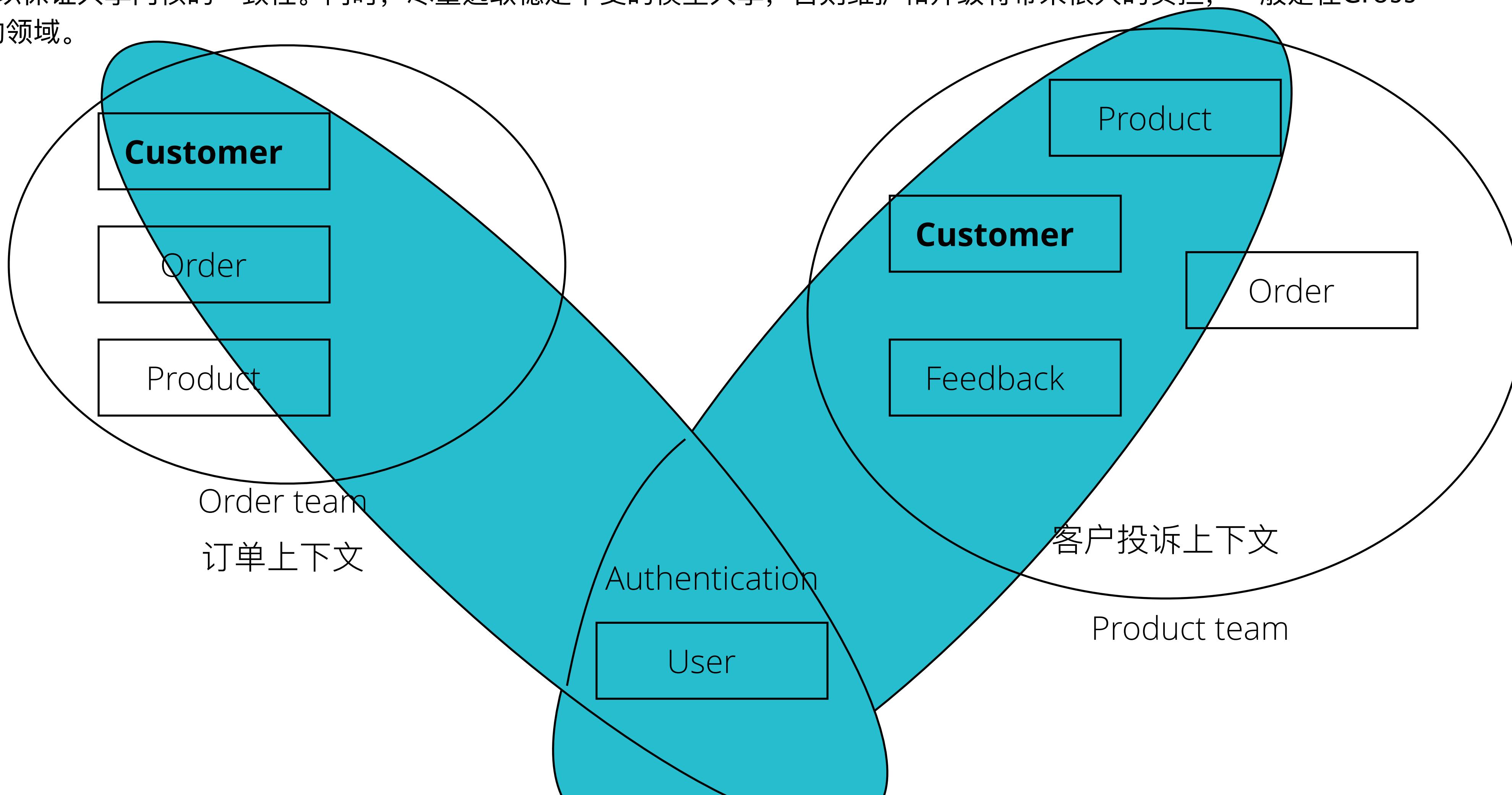
如果上游团队已经没有动力提供下游团队指需，下游团队便孤军无助。处于利他主义，上游团队可能会做出承诺，但很可能这些承诺并无法实现，下游团队不得不使用现有的模型。



限界上下文及映射

## 共享内核（Shared Kernel）

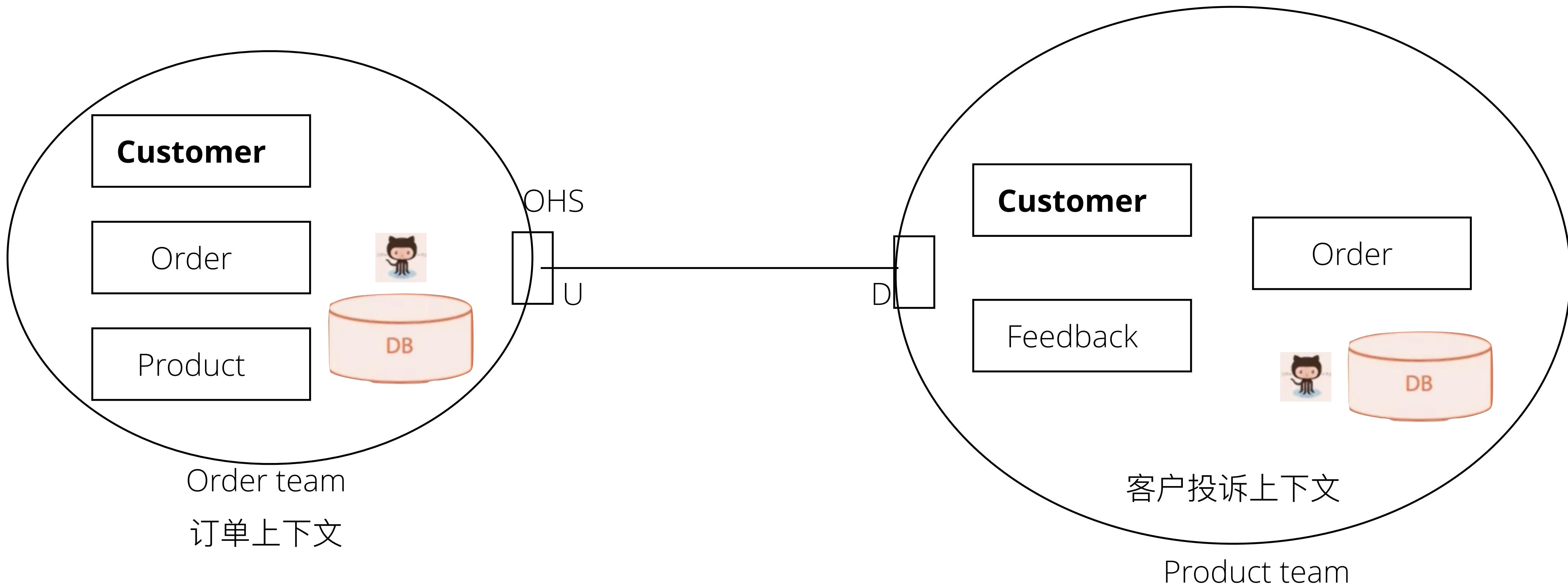
两个团队可以选择对模型和代码进行共享，这将产生强依赖性。共享内核部分代码的更改需要双方的协调，团队应尽早引入持续集成的方式以保证共享内核的一致性。同时，尽量选取稳定不变的模型共享，否则维护和升级将带来很大的负担，一般是在Cross-cutting的领域。



限界上下文及映射

## 开放主机 (Open-Host)

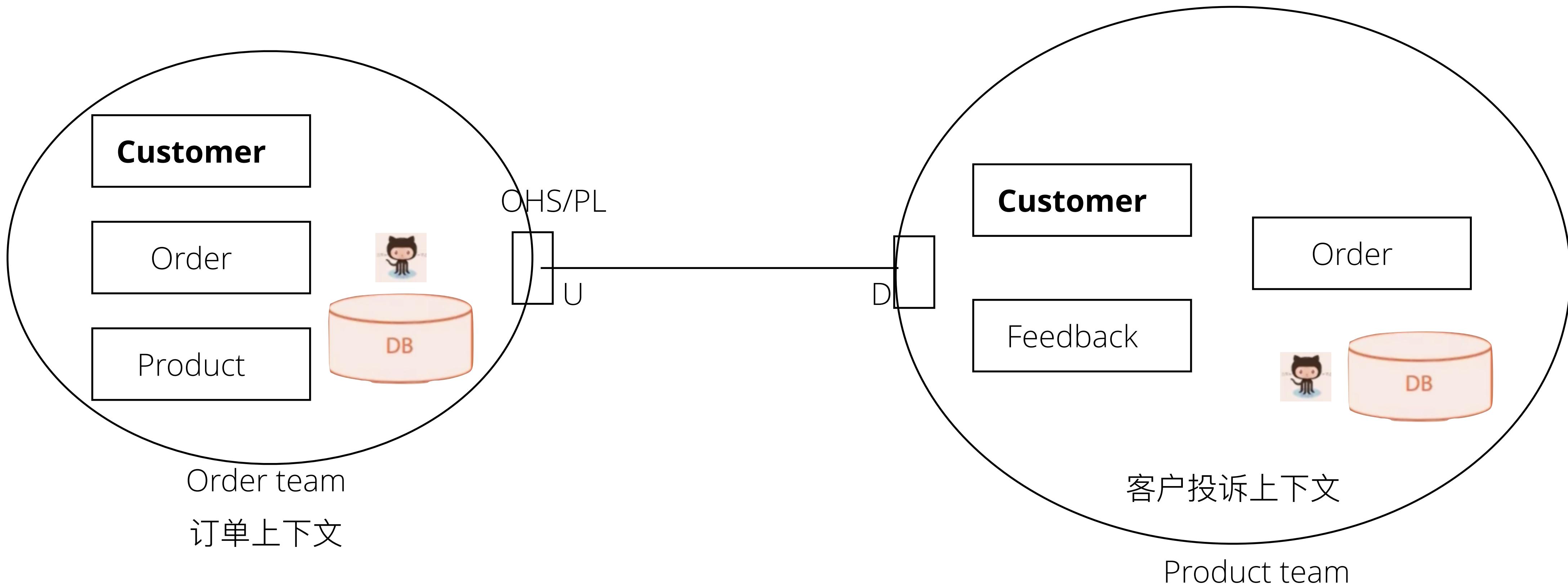
上游将自己的服务通过定义一种协议进行公开，下游可以通过协议来访问上游的服务。



限界上下文及映射

## 发布语言 (Published Language)

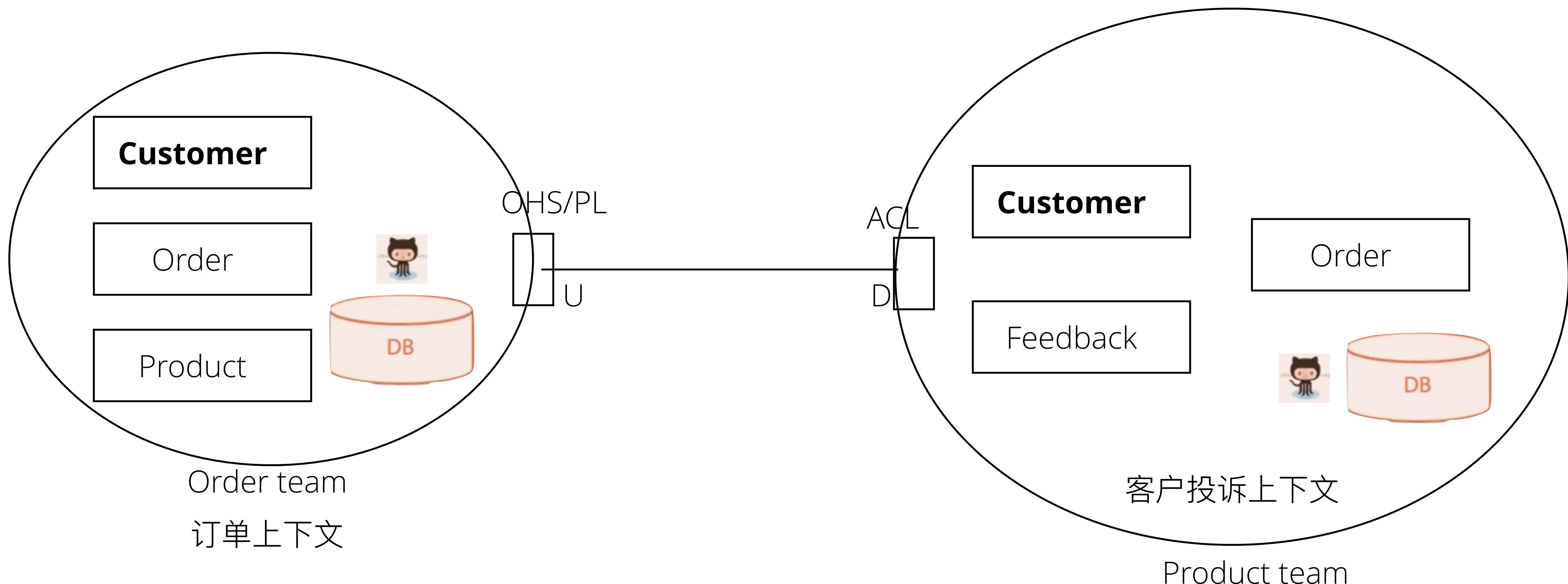
上下游进行集成的时候需要有一种统一的语言对模型进行翻译和集成。在使用REST服务的时候，发布语言展现领域模型，可以用JSON。一般与开放主机关系同时存在。



限界上下文及映射

## 防腐层 (ACL)

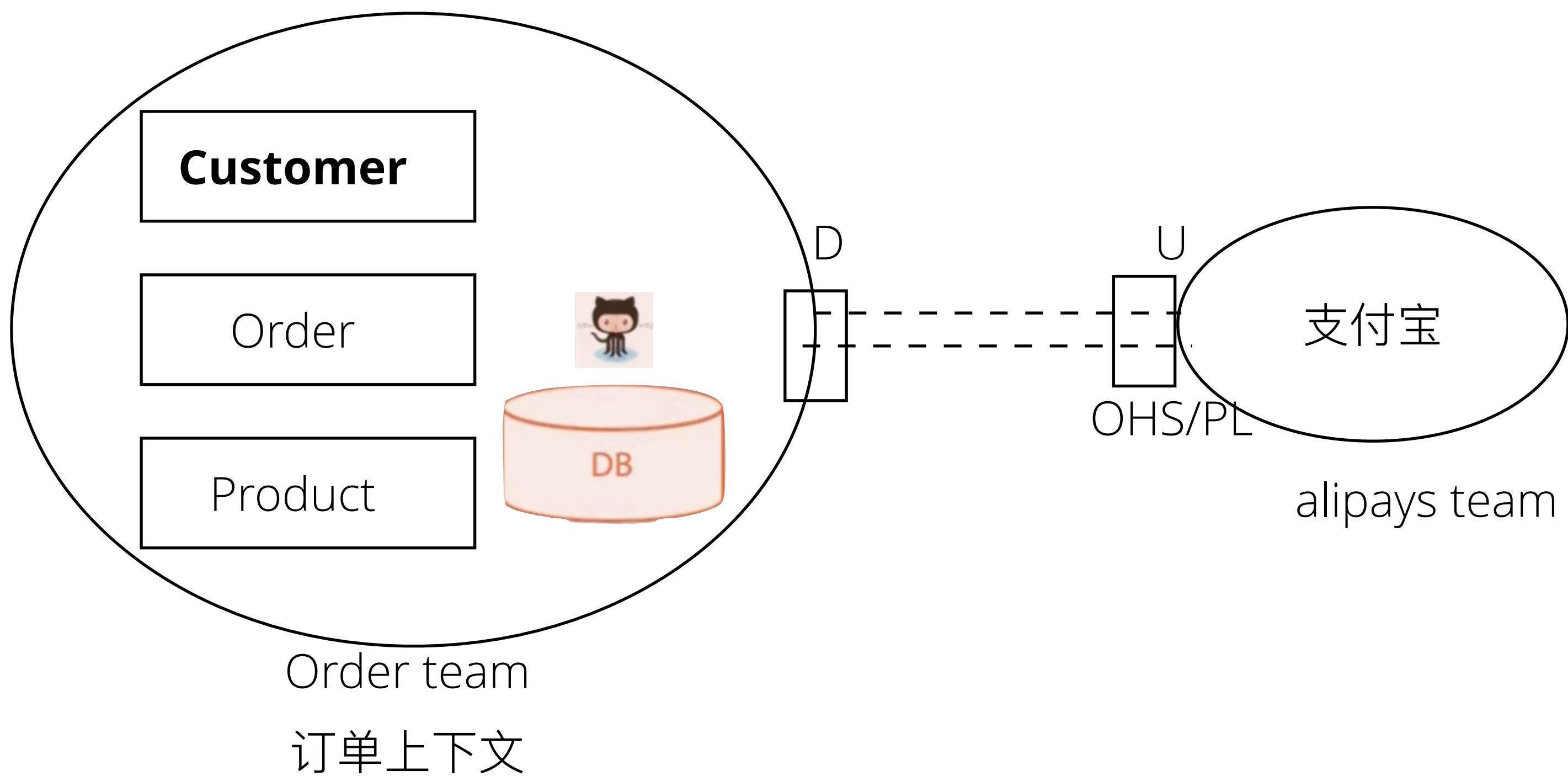
在很多时候，与上游的合作关系并不能顺利地展开，尤其在跨组织的两个团队合作的时候。此时，对于下游系统来说，需要建立一个单独的层作为与上游的代理向你的系统提供功能，以保护内部集成的稳定性。



限界上下文及映射

## 另谋他路 (Separate Way)

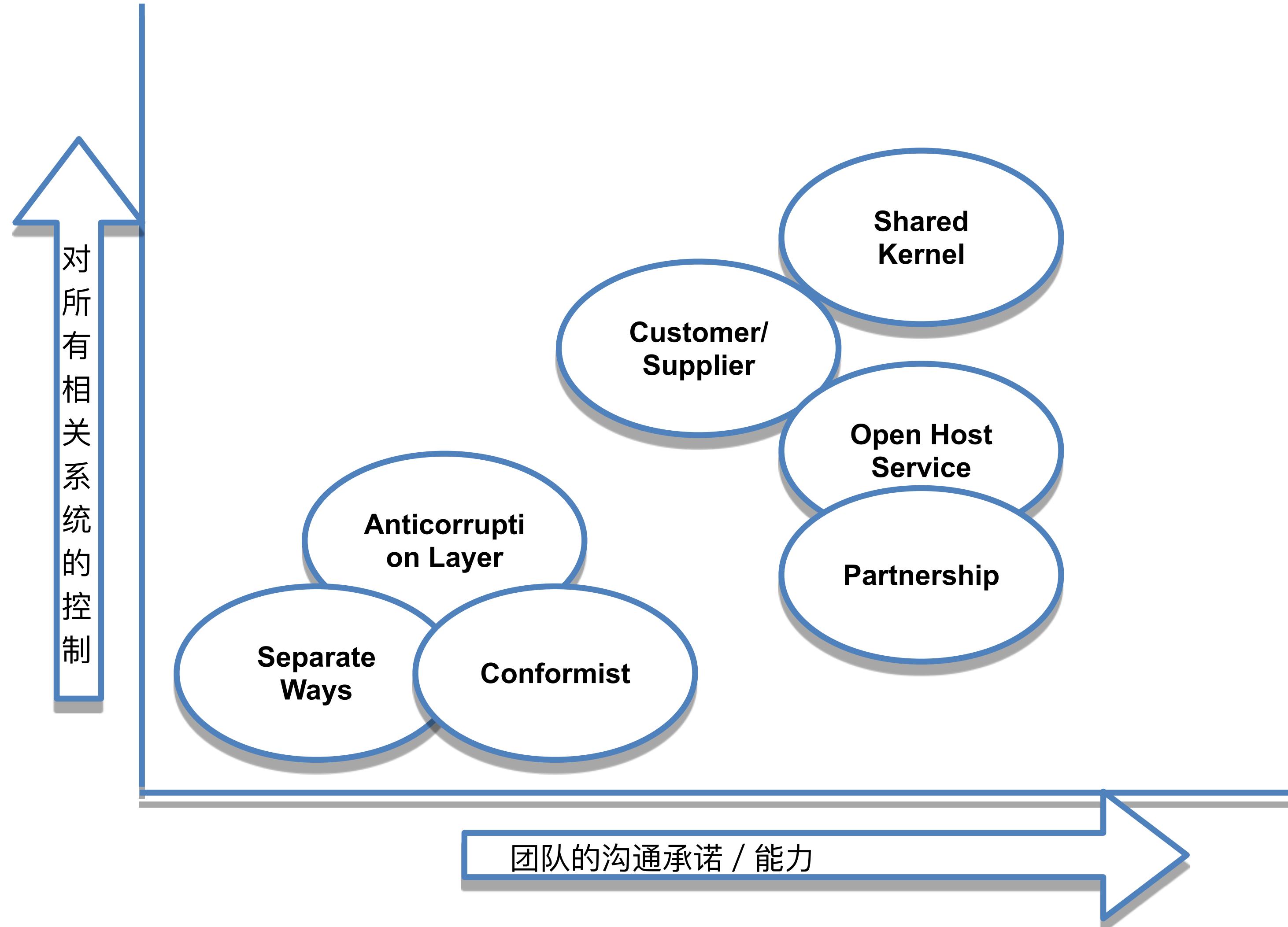
两个上下文并没有显著的关系，可以被完全解耦，因此可以采用更松散的集成方式。



限界上下文及映射

## 上下文映射关系

关系涉及的团队，  
对相关系统中模型  
变化的感知要求



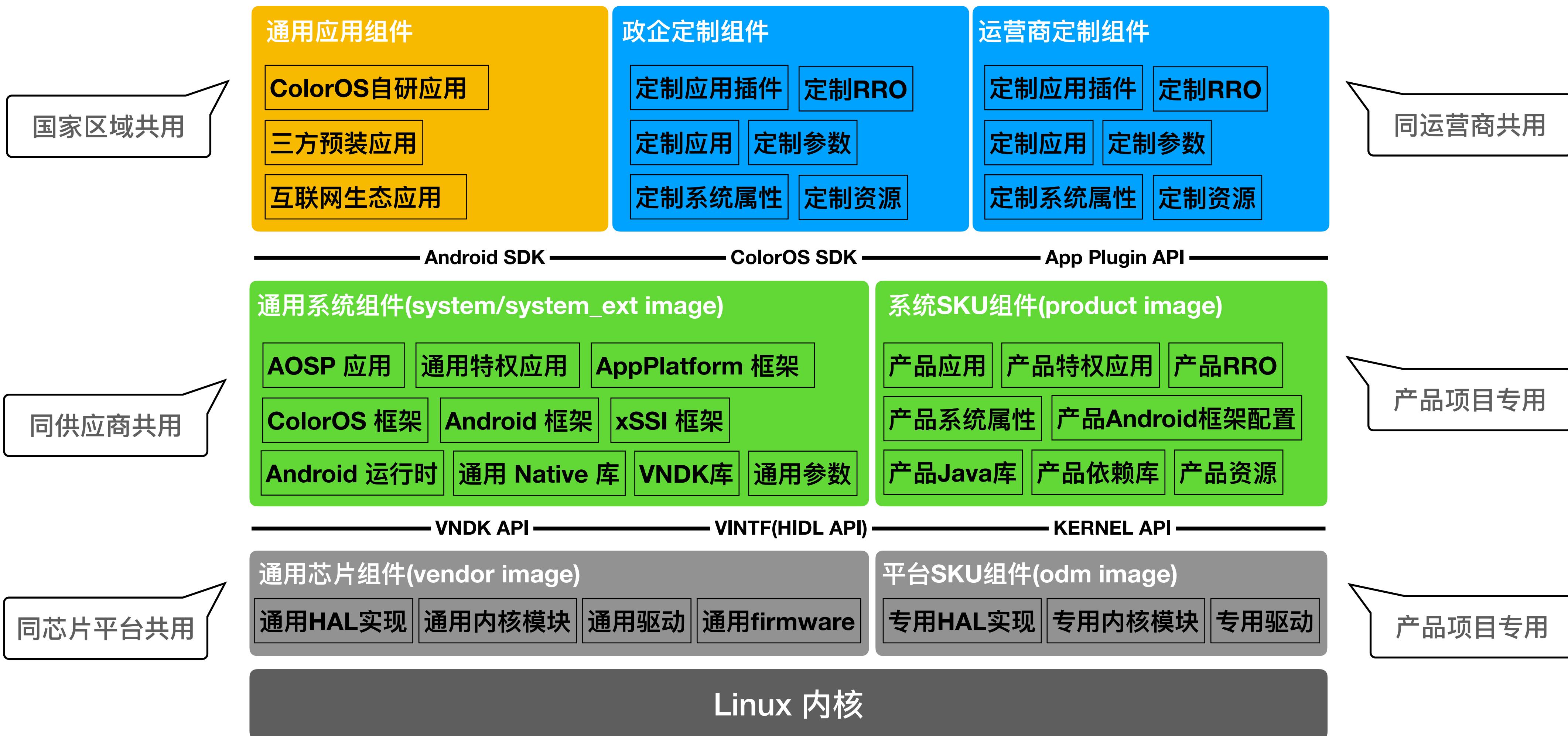
限界上下文及映射

## 上下文映射

- ▶ 可以用于架构分析，造成集成瓶颈的架构问题。
- ▶ 展现了组织动态的能力，帮助我们识别出阻碍项目进展的一些管理（协作和沟通）问题。
- ▶ 保持简单，促进沟通。

限界上下文及映射

## 组件拆分需要体现业务架构



## 限界上下文及映射

# 用上下文映射指导组件间协作

	共享内核	客户-供应方	跟随者	防腐层
适用场景	双方有重复的代码需要修改	上游依赖（供应方）服务于多个下游（客户）	上游是已成熟的技术生态	与随时可能淘汰的老系统或者变化不受控制的系统集成
规则	代码规范，单元测试	接口规范，兼容性规范	增加防腐层	依赖通用接口，用接口实现包装外部API；或特性开关切换不同实现
守护	对共享代码模块的修改必须经过对方代码检视，必须有单元/接口测试覆盖并在流水线中持续执行	契约测试、接口测试在上游依赖流水线中持续执行	遵循上游技术生态规范，通过上游的代码扫描和自动化测试持续验证	接口定义必须有单元/接口测试覆盖，并持续验证
协作	完全开放代码仓库，面对面沟通、结对编程。由一方主导共享代码的维护	双方定期协商定义接口，完成接口文档以及示例。联合开发接口测试	跟踪上游依赖更新计划，及时调整	积极跟踪上游依赖变动，及时调整
举例	定制SKU	互联网服务与客户端，应用间解耦，服务进程与应用	Vendor、AOSP	HIDL、ColorSDK

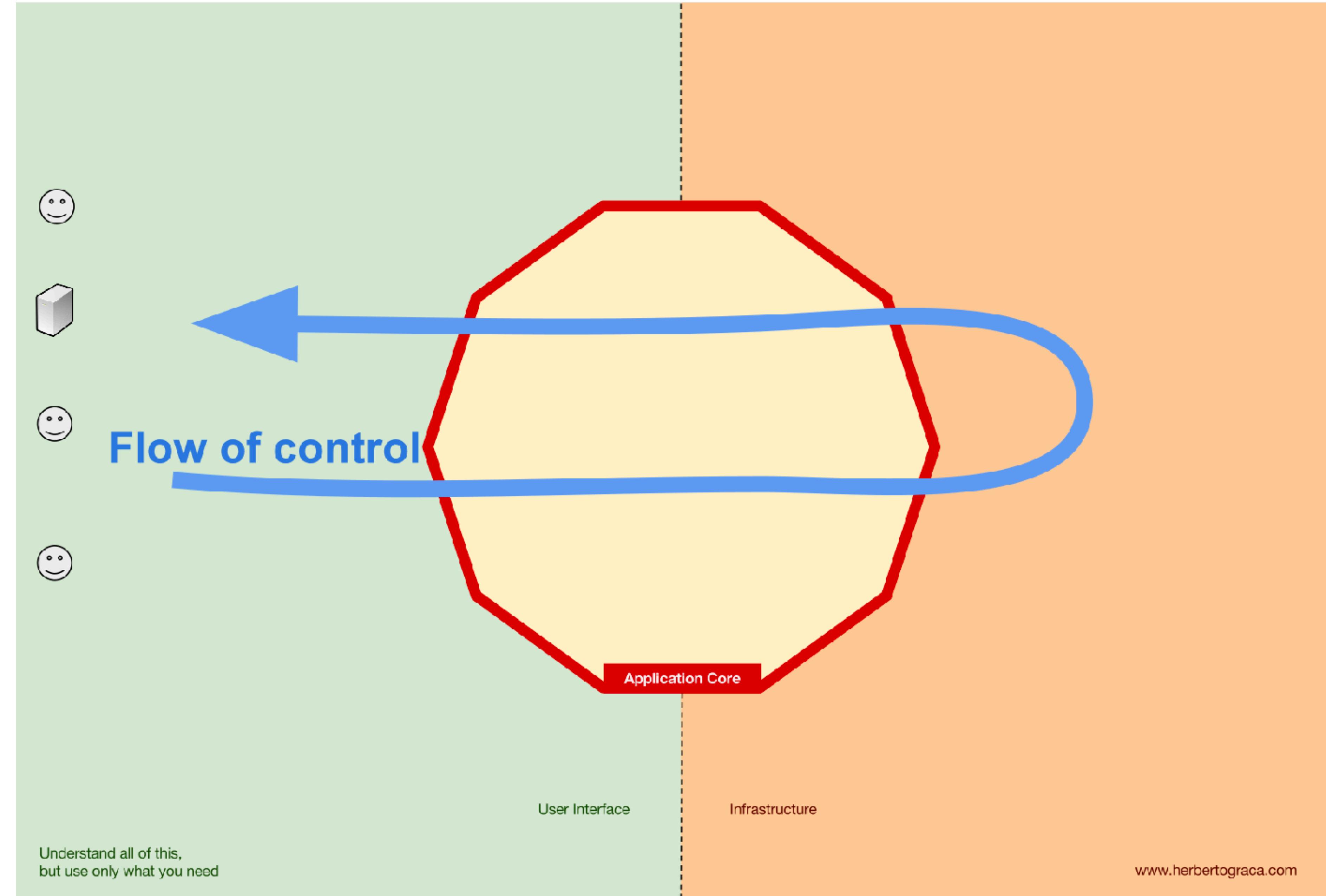
# 附：清晰架构

---

## 分层架构模块

### 基本构建块

- 运行用户界面所需的构建块
- 应用核心，用户界面要使用这个构建块达成目的
- 基础设施，将应用核心和数据库、搜索引擎或第三方 API 等连接起来

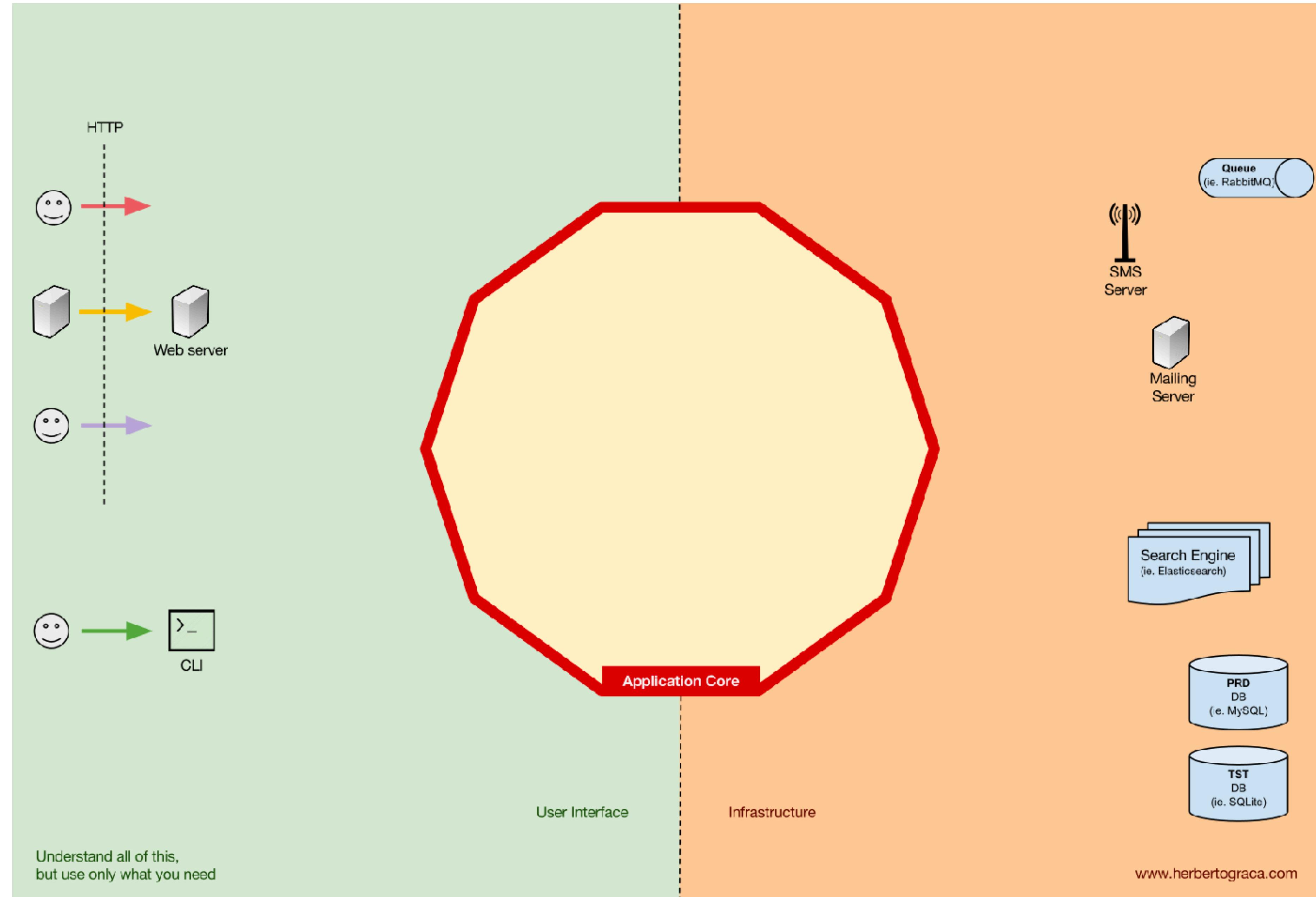


# 分层架构模块

## 工具

命令行控制台和 Web 服务器  
告诉我们的应用它要做什么

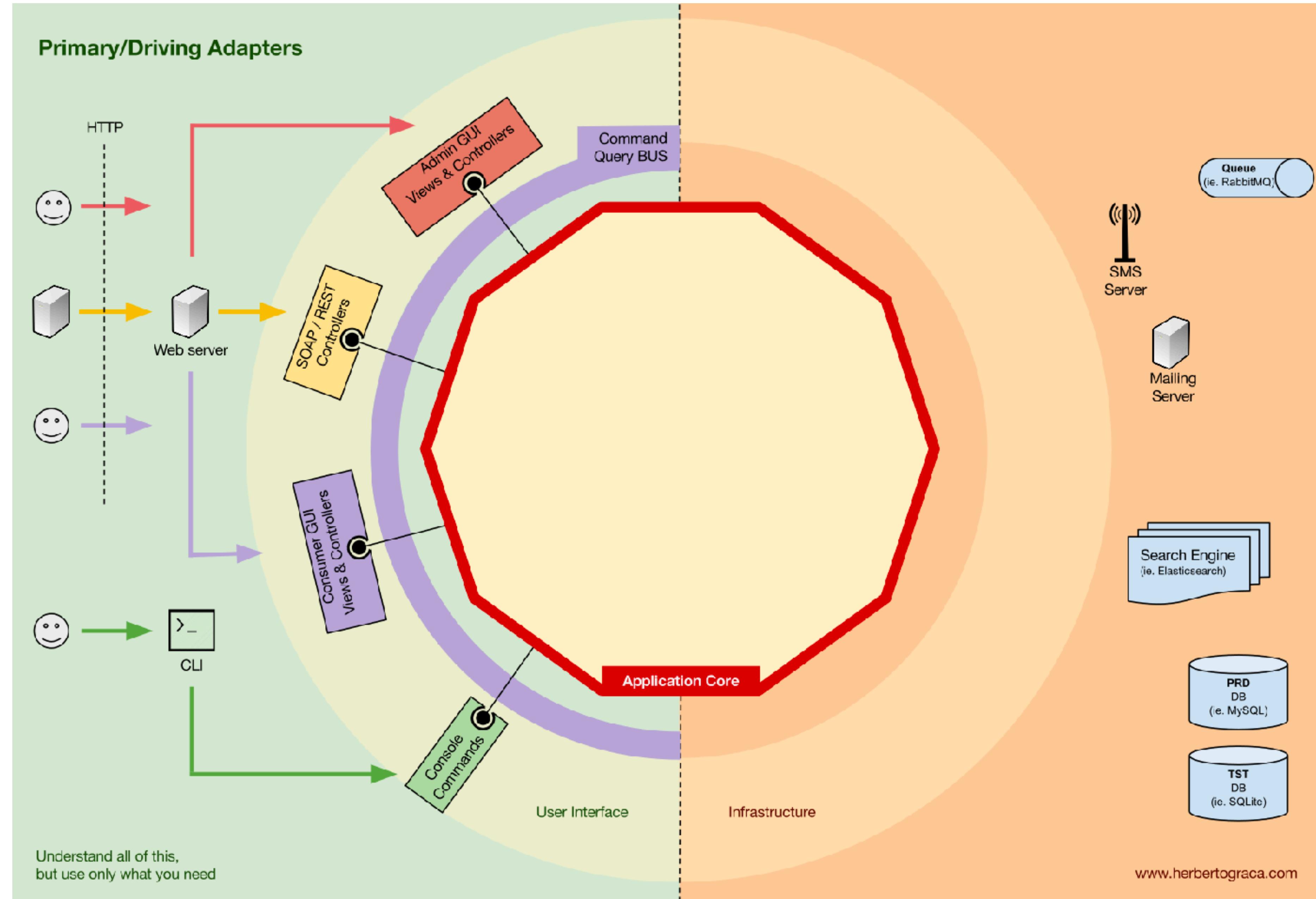
数据库引擎是由我们的应用  
来告诉它做什么



# 分层架构模块

## 主适配器

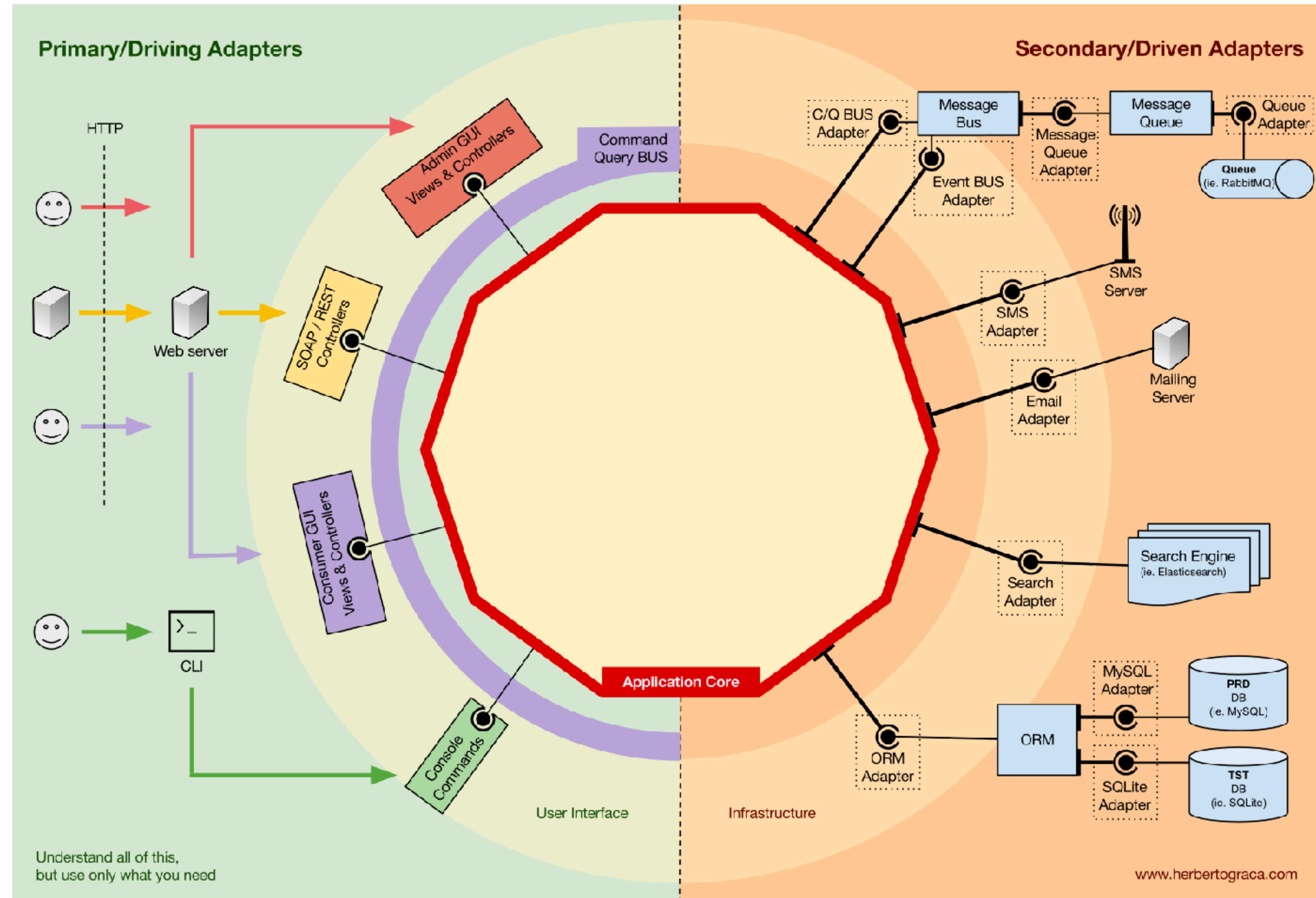
主适配器或主动适配器包装端口并通过它告知应用核心应该做什么。它们将来自用户界面的信息转换成对应用核心的方法调用。



# 分层架构模块

## 从适配器

被动适配器实现一个端口（接口）并被注入到需要这个端口的应用核心里。



# 分层架构模块

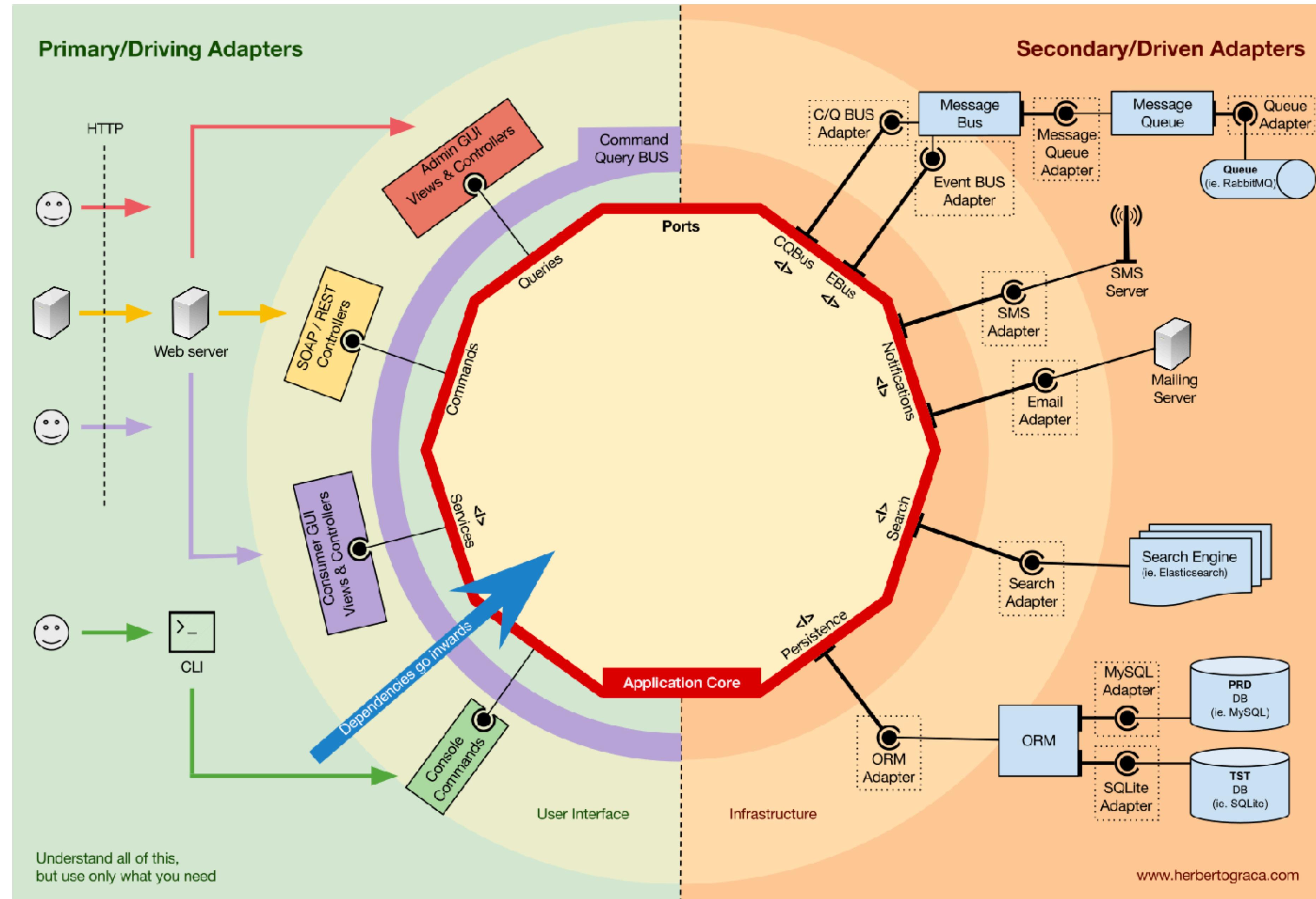
## 控制反转

适配器需要按照应用核心某个特定入口的要求来创建，即端口。

在大多数语言里最简单的形式就是接口，也可能由多个接口和 DTO 组成。

端口（接口）位于业务逻辑内部，而适配器位于其外部。

适配器依赖特定的工具和特定的端口，而业务逻辑只依赖按照它的需求设计的端口（接口）



# 分层架构模块

## 应用层

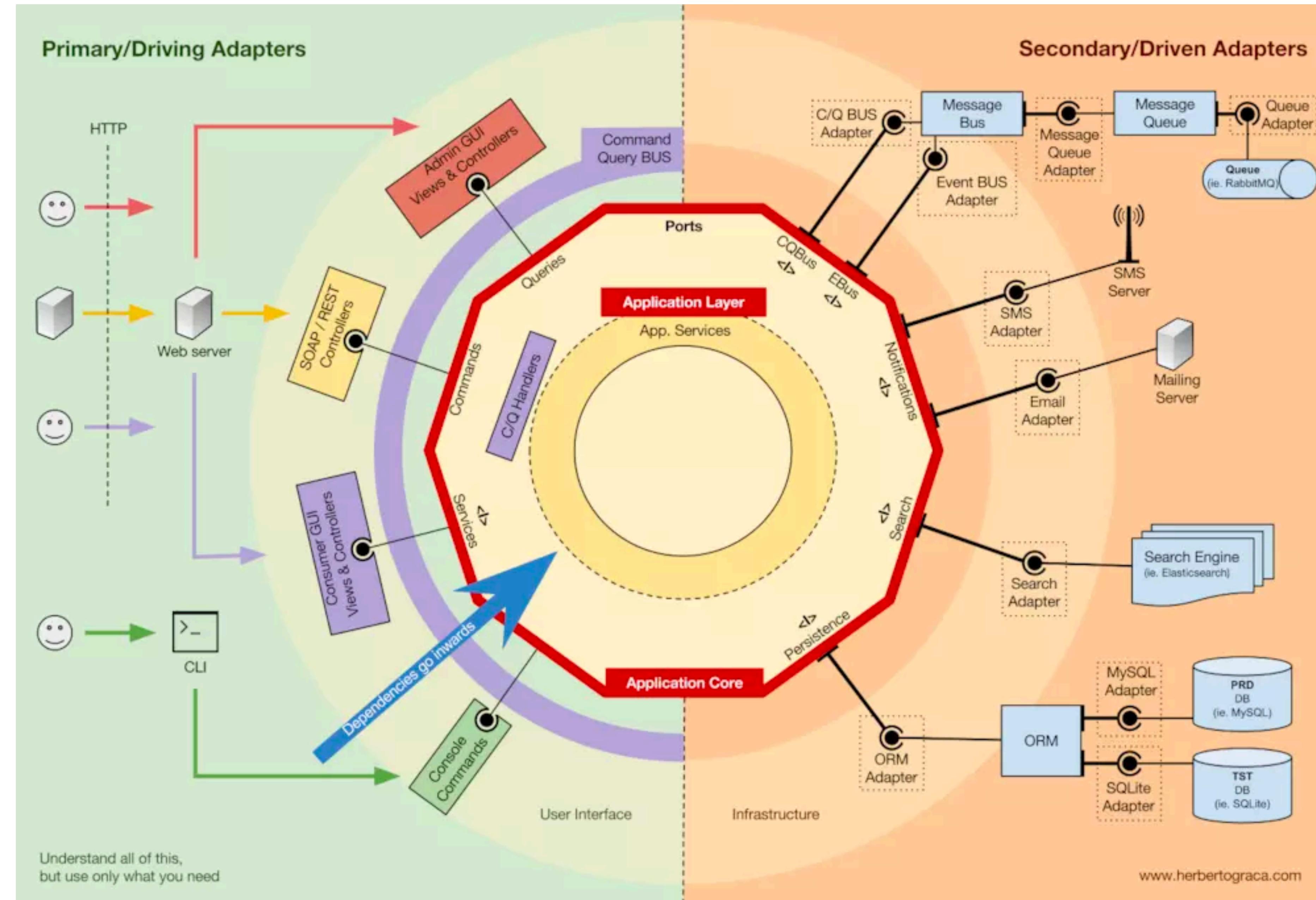
这个层包括：

应用服务(以及它们的接口)

端口，例如 ORM 接口、搜索引擎接口、消息接口等等

如果使用了命令总线和查询总线，命令和查询分别对应的处理程序也属于这一层

作为应用服务副作用触发的应用事件，比如发送邮件，通知第三方 PAI，发送推送通知等



# 分层架构模块

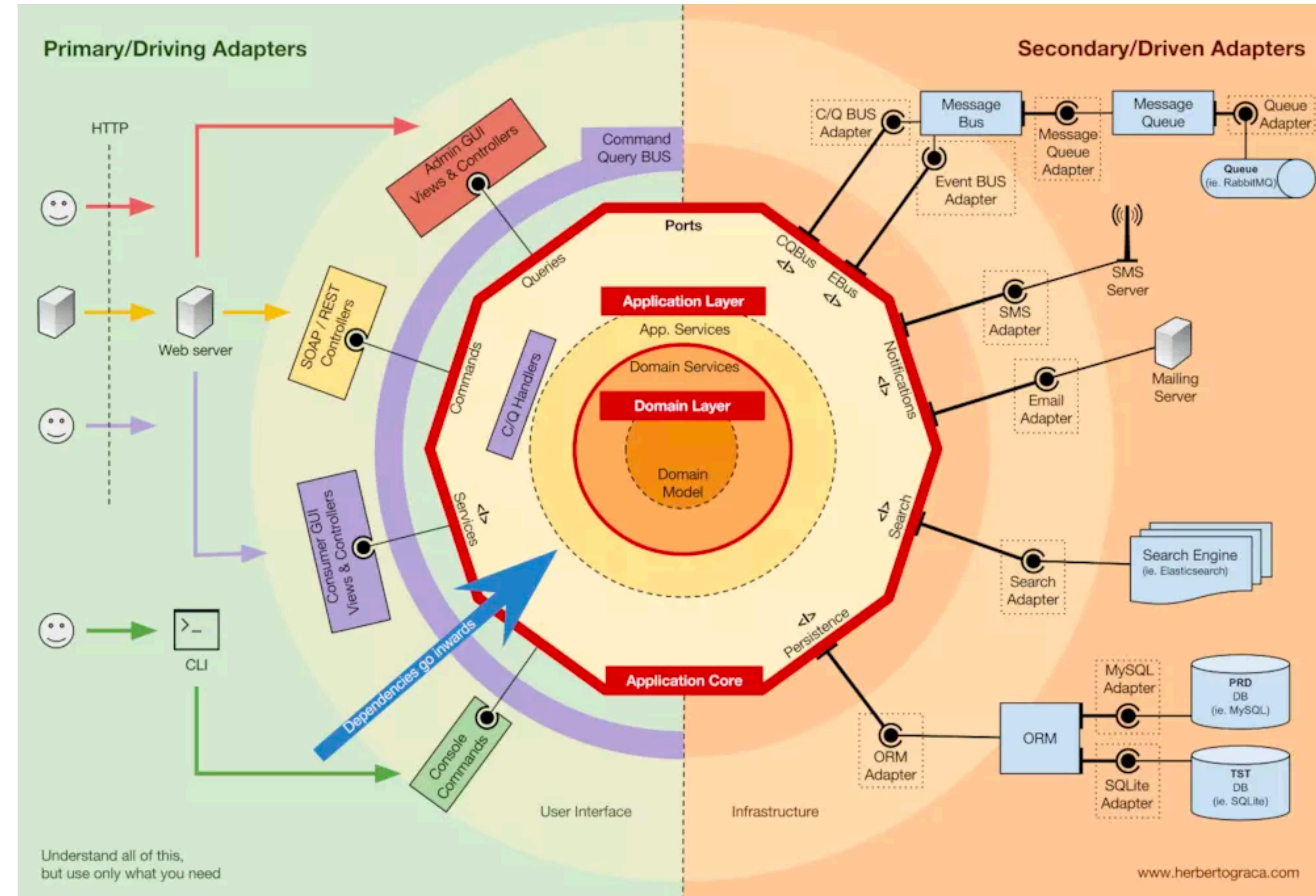
## 领域层

这个层包括：

**领域服务**，它的作用是接收一组实体并对它们执行业务逻辑，它可以使用其他领域服务，还可以使用其它领域模型对象。

**领域模型对象**，表示领域中某个概念的业务对象。这些对象包括实体、值对象、枚举以及其它任何领域模型使用的对象。

**领域事件**，一组特定的数据发生变化时会触发这些事件。

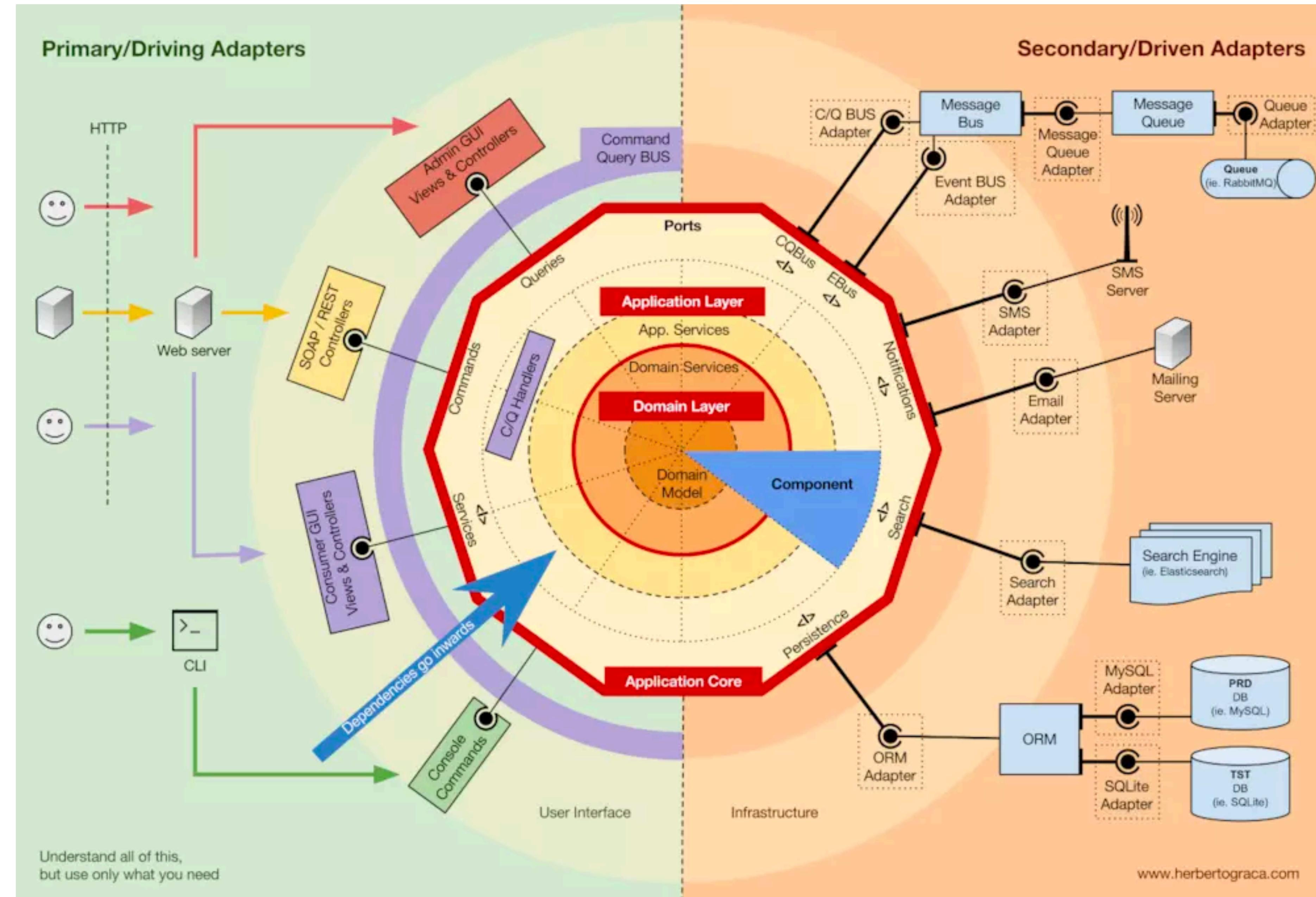


# 分层架构组件

## 组件

组件是例如账单、用户、评论或帐号，其划分与业务领域相关

组件的划分是分层划分的横切（与其正交）



划分包结构

## 包的定义

将类划分成包可以让我们在更高的抽象级别来思考设计。其目标是将你的应用中的类按照某种条件进行分片，然后将这些分片分配到包中。这些包之间的关系表达出了应用高级别的组织方式。

—— Robert C. Martin 1996

## 划分的目的

高内聚低耦合：

- 修改一个包而不会影响其它的包，减少出现的问题；
- 修改一个包而不需要修改其它的包，加快交付的节奏；
- 让团队专注于特定的包，带来更快、更健壮和设计更优的变化；
- 团队开发活动之间的依赖和冲突更少，提升产能。
- 更仔细地斟酌组件之间的关系，让我们更好地将应用作为一个整体建模，交付质量更高的系统。

## 划分包结构

# 划分原则

## 包内聚原则

- REP - 重用发布等价原则  
重用的粒度等价于发布的粒度
- CCP - 共同封闭原则  
一起被修改的类应该放在一个包里
- CRP - 共同重用原则  
一起被重用的类应该放在一个包里

## 包耦合原则

- ADP - 无环依赖原则  
包的依赖图中不能出现循环
- SDP - 稳定依赖原则  
依赖应该朝着稳定的方向前进
- SAP - 稳定抽象原则  
抽象的级别越高，稳定性就越高

## 划分包结构

# 建议按组件分包

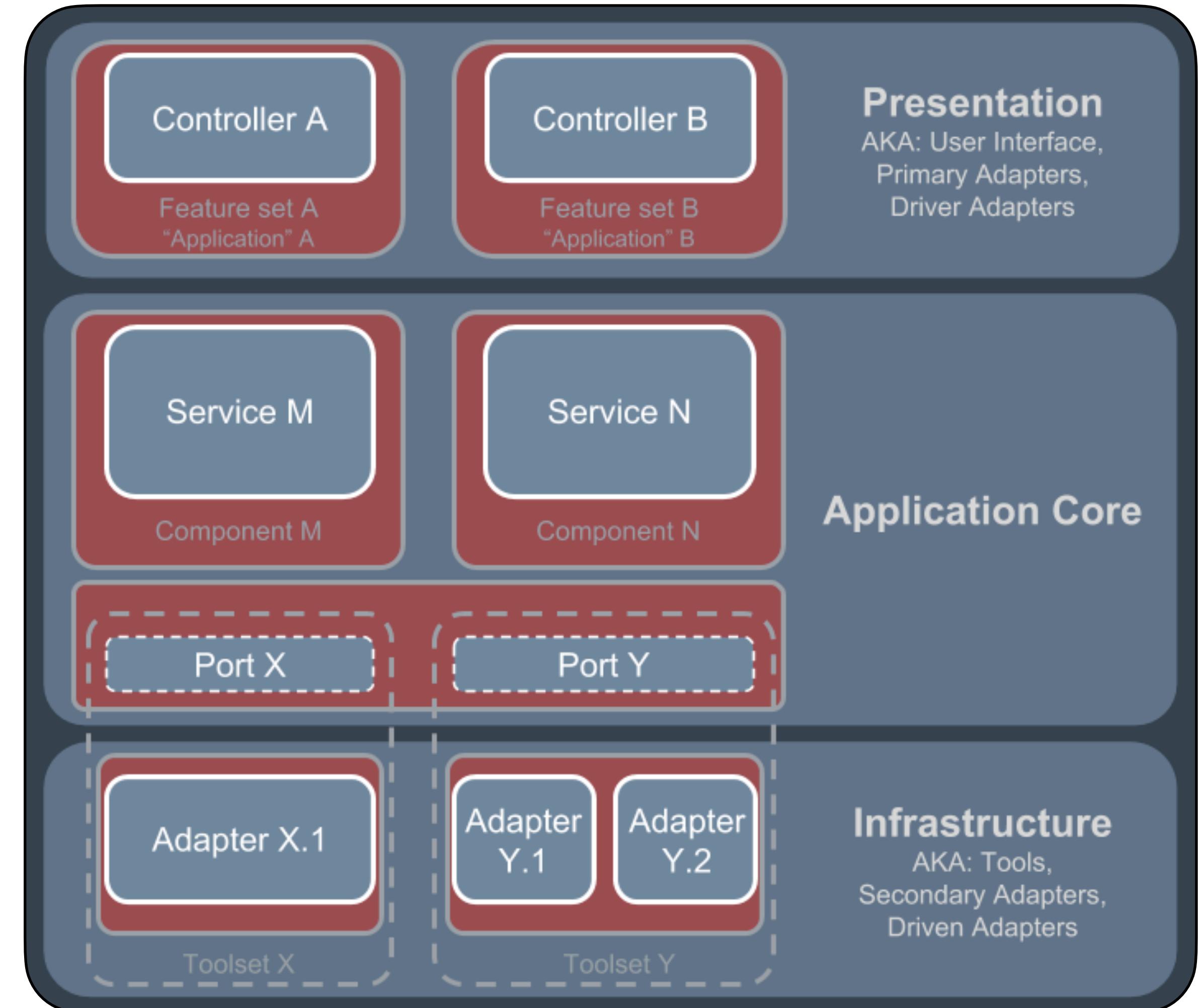
源代码文件夹里出现的第一级目录应该和领域概念有关，即最顶层的限界上下文（例如，患者、医生、预约等）。

“如果我想去掉一个业务概念，是不是删除掉它的组件根目录就能把这个业务概念的所有代码删除而且应用的剩余部分还不会被破坏？”

如果答案是肯定的，那么我们就有了一个解耦得不错的组件。

在包内，我们才会在必要时按照功能作用区分类

任何代码只能存在于一个逻辑上的位置，即使对项目中的新手和初级开发者来说，这个位置也是十分明了的。



## 分层架构

## 参考资料



软件架构编年史

<https://www.jianshu.com/p/b477b2cc6cfa>

《领域驱动设计》中的例子

<https://github.com/citerus/dddsample-core>

《实现领域驱动设计》中的例子

[https://github.com/VaughnVernon/IDDD\\_Samples](https://github.com/VaughnVernon/IDDD_Samples)



# THANK YOU



ThoughtWorks®