什么是微服务架构

什么是架构

- 架构：**描述组成系统的各部件及其之间的关系**
  - 架构决定了软件的质量属性 bility
    - 可维护性，可测试性，可部署性
    - 安全性，可靠性，可扩展性
  - 4+1架构模型：（逻辑、实现、进程、部署）+ 场景

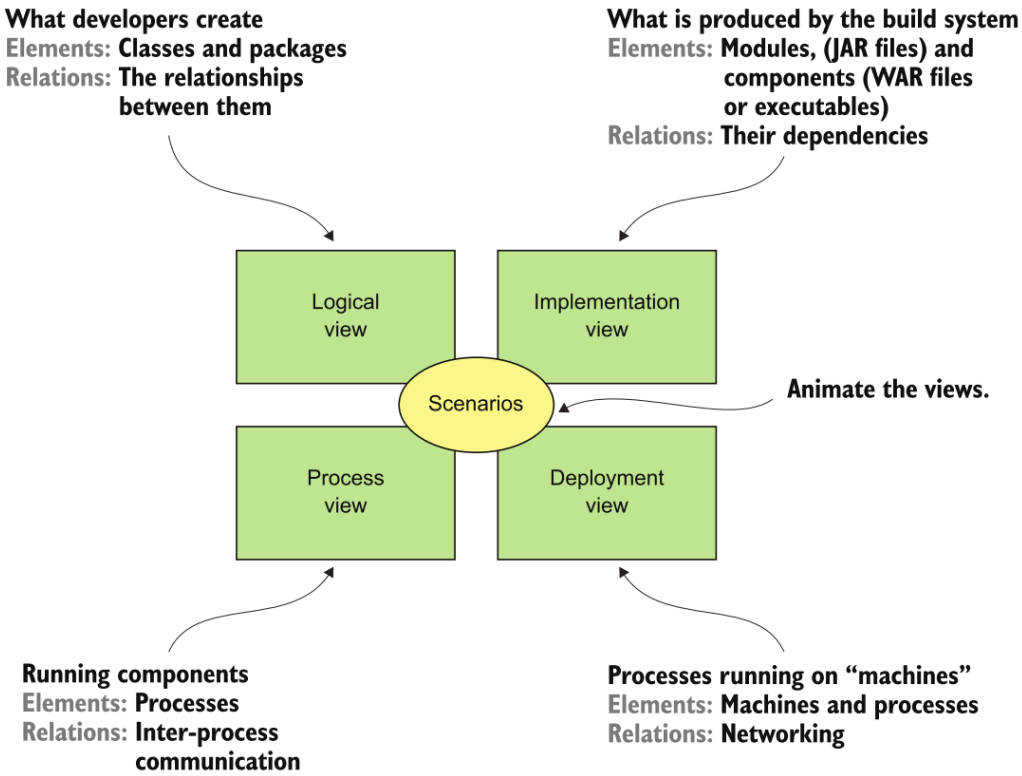
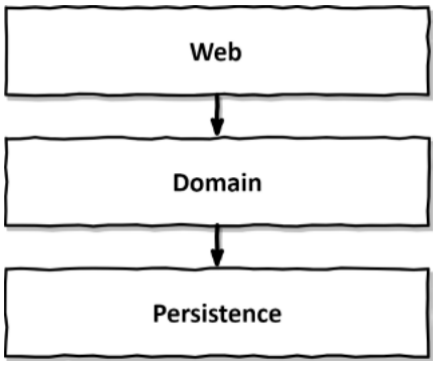


Figure 2.1 The 4+1 view model describes an application's architecture using four views, along with scenarios that show how the elements within each view collaborate to handle requests.

  - 逻辑视图：开发时的架构
    - 关注功能性需求，考虑问题域分解与抽象
    - 组成部件：包（模块）、类、接口
    - 关系：关联、继承、依赖、组合、聚合

    - 实现（开发）视图：构建时的架构
        - 关注软件打包，与团队分配有关
        - 部件：打包好的可执行，可部署的模块和组件， 如 Jar包，War包，exe文件
        - 关系：依赖关系
    - 进程视图：运行时的架构
        - 关注组件间的协作，考虑进程间通信
        - 部件：各个运行当中的进程
        - 关系：进程间的通信，如 HTTP， 信号量，共享内存
    - 部署视图：运行机器的架构
        - 关注性能与可用性，考虑可扩展性
        - 部件：运行着系统进程的物理机或虚拟机
        - 关系：机器间的网络通信
    - 四种视图并不互斥，可互相组合
- 分层架构（三层架构为例）



    - 属于逻辑视图
    - 表示层（用户接口或外部API）、业务逻辑层、持久层
    - 缺点： 不支持多客户端，不支持多数据库，领域层依赖持久层
- 六边形架构

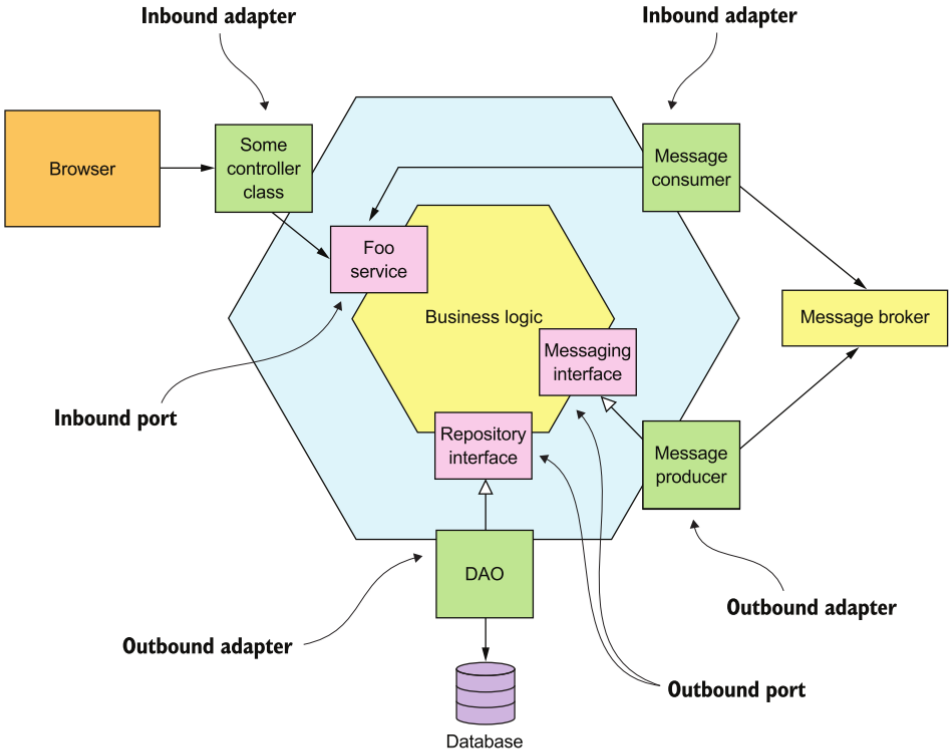
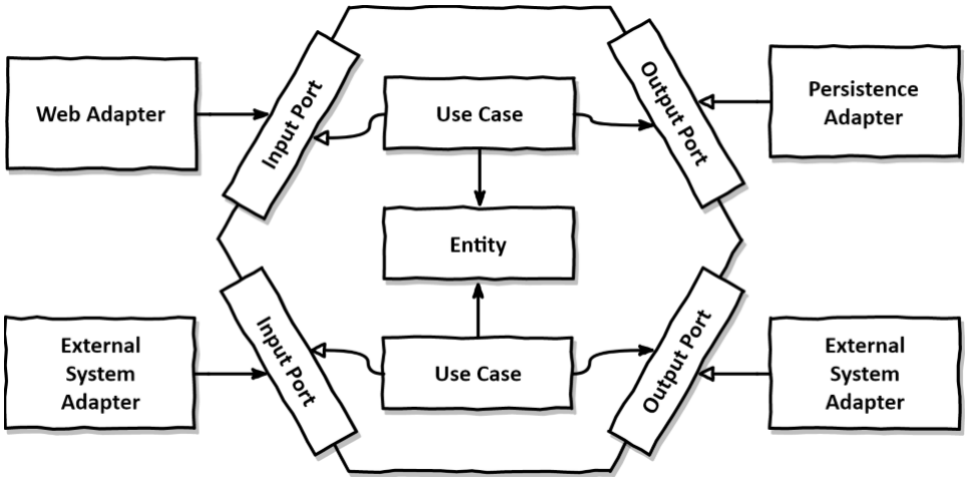


Figure 2.2 An example of a hexagonal architecture, which consists of the business logic and one or more adapters that communicate with external systems. The business logic has one or more ports. Inbound adapters, which handled requests from external systems, invoke an inbound port. An outbound adapter implements an outbound port, and invokes an external system.



- 属于逻辑视图
- 引入多个入站适配器（inbound adapter）
  - 取代单一的表示层，
  - 完成对多种外部请求的处理
  - 调用业务逻辑
- 引入多个出站适配器（outbound adapter）
  - 取代单一的持久层
  - 被业务逻辑调用
  - 同时调用外部应用程序，如数据库
- **但是，业务逻辑并不依赖这些适配器，而是适配器依赖业务逻辑**
- 业务逻辑具有一个或多个端口
  - 一个端口定义了一组操作供外界调用
  - 可分为入站端口和出站端口

    - 入站端口
        - 定义业务逻辑提供的API，供外部应用调用
        - 入站适配器调用入站端口
        - REST 适配器是最常见的入站适配器
    - 出站端口
        - 定义业务逻辑如何调用外问应用
        - 出站适配器实现了出站端口
        - DAO 是最常见的出站适配器
    - 六边形架构的好处
        - 把业务逻辑与表示层，持久层完全解耦
        - 保持业务逻辑的独立性：业务逻辑与环境、技术、框架无关
        - 因为解耦，所以可与多种外部应用进行适配
        - 六边形架构应作为每个微服务的架构
    - **实战：三层架构 VS 六边形架构**
        - 代码请参照 https://github.com/walshzhang/eaap
- 单体架构
    - 实现视图：打包成一个可执行文件或WAR包（Java）
    - 进程视图：运行在一个进程中
    - 部署视图：部署到一台服务器
    - 与逻辑视图不冲突：可采用分层架构，也可采用六边形架构
- 微服务架构：应用被分解为多个服务
    - 实现视图：每个服务被打包成一个可执行文件、JAR包或WAR包
    - 进程视图：每个服务运行在一个进程中
    - 部署视图：每个服务被部署到一台机器上
    - 每个服务一般采用六边形架构
    - **每个服务是一个小的单体应用**

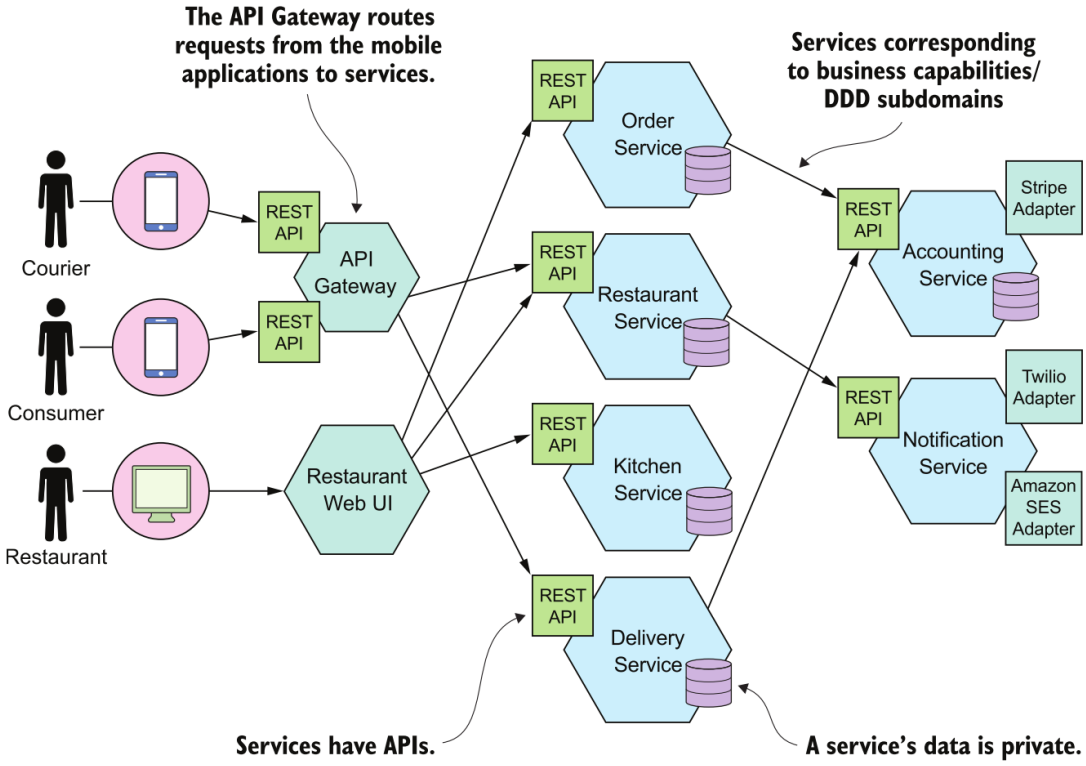


Figure 2.3 A possible microservice architecture for the FTGO application. It consists of numerous services.

什么是服务

- 实现了某种功能的可独立运行，单独部署的软件组件
- 对外提供操作，以API的形式
  - 命令：写数据，创建订单（createOrder）
  - 查询：读数据，查看订单（getOrder）
  - 事件：供客户端消费，如新订单被创建（OrderCreated）
- 逻辑架构：每个服务都拥有自己独立的架构（六边形）、技术栈
- 实现架构：Jar包，镜像，云函数
- 进程架构：进程、容器，Pod（K8S），Serverless
- 这些API由适配器（入站或出站）实现

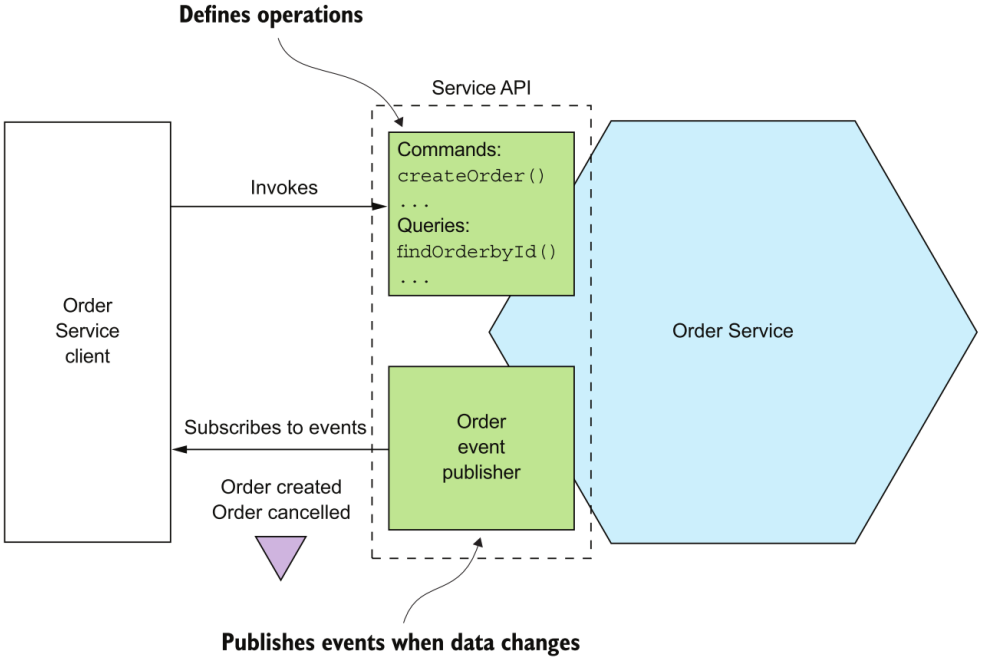


Figure 2.4 A service has an API that encapsulates the implementation. The API defines operations, which are invoked by clients. There are two types of operations: commands update data, and queries retrieve data. When its data changes, a service publishes events that clients can subscribe to.

- 松耦合
  - 所有的交互都通过 API 完成
  - API 的实现对外不可见（私有）
  - 数据库是对外不可见（私有）
- 服务可作为共享库使用，解决服务之间代码复用
  - 但是，对于稳定的，不会变化的功能，还是使用库为好
- **服务的尺寸不重要**
  - 微服务不是小尺寸服务
  - **关键是如何分解服务**
  - 一个团队开发一个服务
    - 团队大小：两个Pizza原则（两个披萨能喂饱整个团队:）
  - 因为小，所以敏捷，更快地开发、测试，部署

定义应用的微服务架构

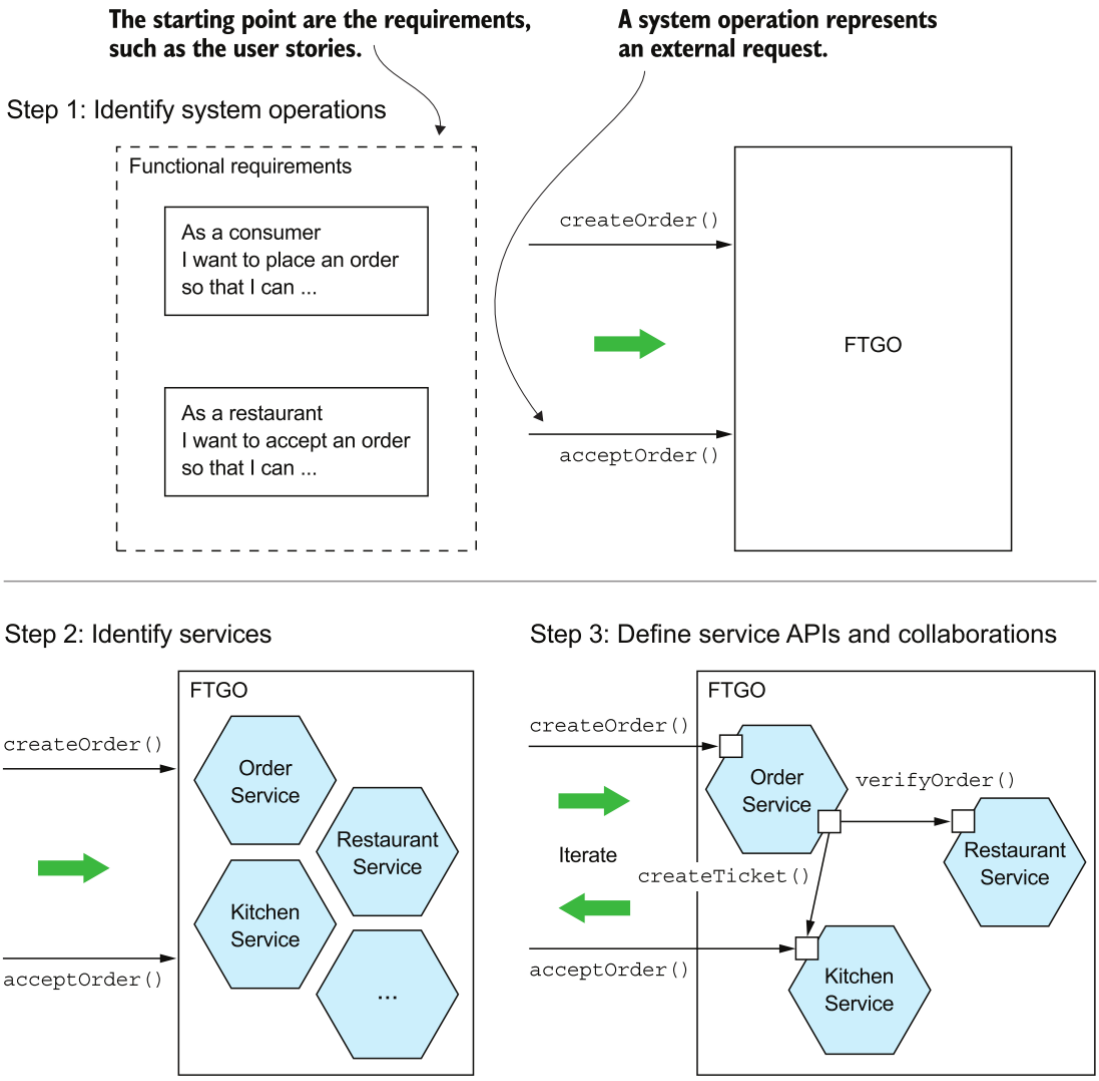


Figure 2.5 A three-step process for defining an application's microservice architecture

- 标识系统操作（行为）
  - 后续会某种进程间通信（IPC）取代
  - 高层领域模型：从需求中提取名词，只用于描述系统操作
    - 并不是最后的领域模型（教材第5章）
    - 最后每个服务都有自己的领域模型
  - 系统操作：从需求（用户故事）提取动词

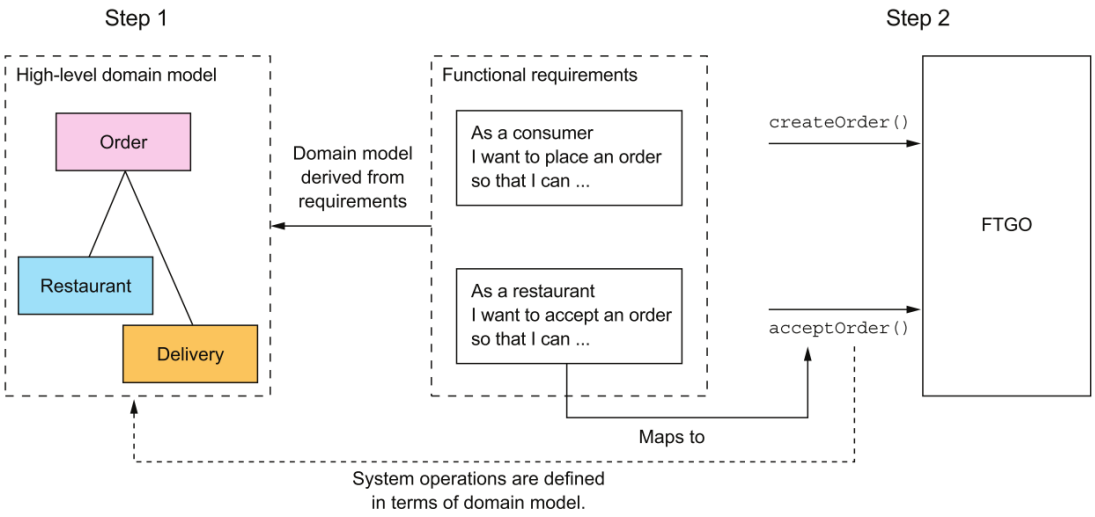


Figure 2.6 System operations are derived from the application's requirements using a two-step process. The first step is to create a high-level domain model. The second step is to define the system operations, which are defined in terms of the domain model.

Table 2.1 Key system commands for the FTGO application

Actor	Story	Command	Description
Consumer	Create Order	`createOrder()`	Creates an order
Restaurant	Accept Order	`acceptOrder()`	Indicates that the restaurant has accepted the order and is committed to preparing it by the indicated time

Operation	`acceptOrder(restaurantId, orderId, readyByTime)`
Returns	—
Preconditions	▪ The `order.status` is `PENDING_ACCEPTANCE`. ▪ A courier is available to deliver the order.
Post-conditions	▪ The `order.status` was changed to `ACCEPTED`. ▪ The `order.readyByTime` was changed to the `readyByTime`. ▪ The courier was assigned to deliver the order.

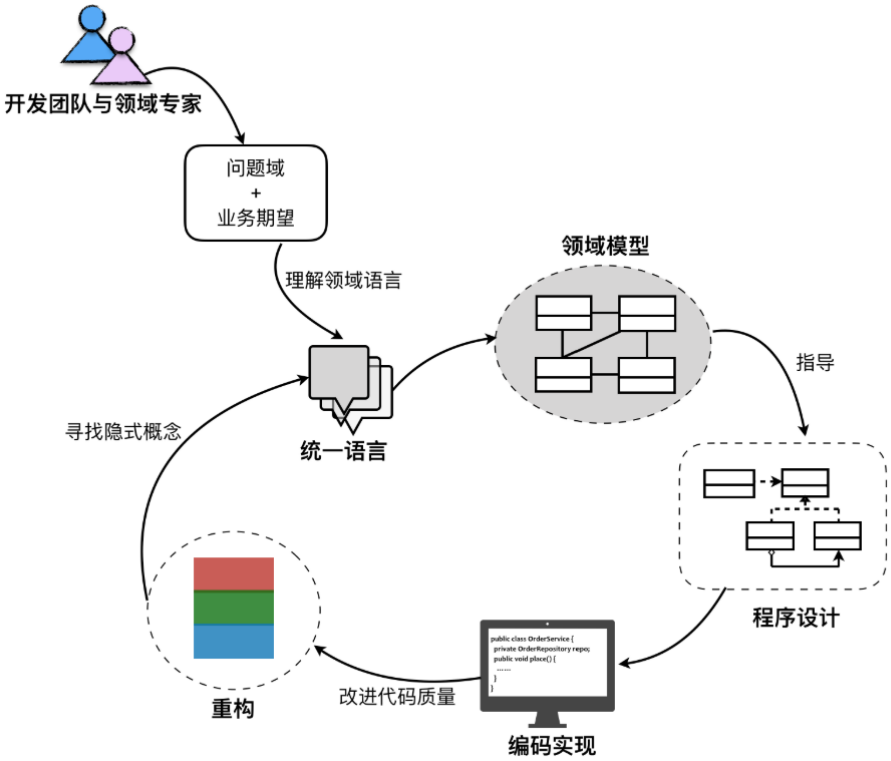
- 标识应用的服务
  - 使用功能分解方法（略）
  - 使用领域驱动设计：识别限界上下文，每个限界上下文对应一个服务
- 定义服务 API 及其之间的关系
  -

实战：作业提交系统架构设计

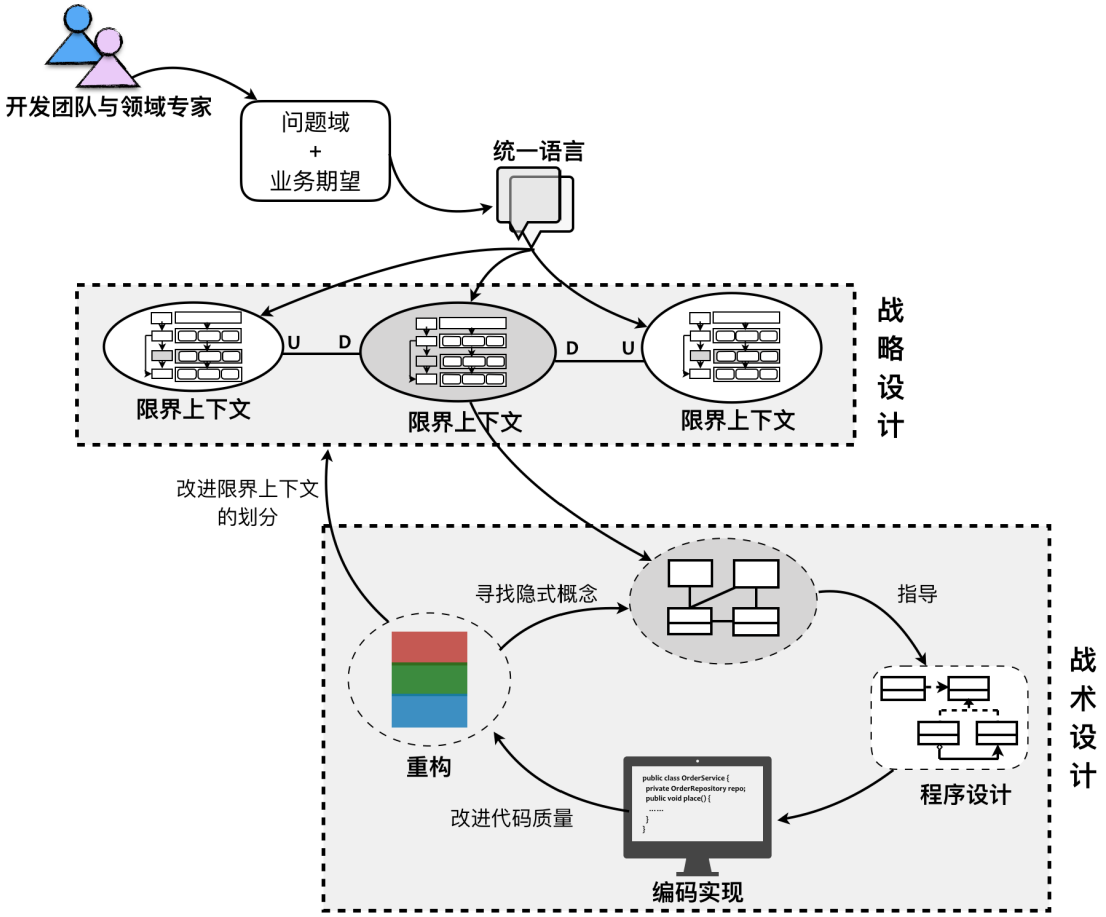
- 用户故事
-

领域驱动设计概述

- 此部分课件引用了[张逸. 领域驱动设计（战略篇）. GitChat 专栏课程]
- 领域驱动设计：Domain Driven Design
- Eric Evans. 在2003年出版的同名著作中提出



    - 提炼统一语言（Ubiquitous Language）
    - 根据统一语言建立领域模型
    - 使用领域模型指导程序设计
    - 编码、重构、完善统一语言
    - 螺旋式迭代过程
- 对于大型复杂应用，可分为战略设计、战术设计两部分进行
- 战略设计：（教材第2章分解服务）
    - 问题分解：限界上下文（Bounded Context）和上下文映射（Context Map）
    - 限界上下文 ---- 对应着 -----> 服务
    - 上下文映射 ---- 对应着 -----> 服务之间的通信
- 战术设计：（教材第5章）
    - 针对每个限界上下文进行设计
    - 模式：值对象（Value Object），实体（Entity），领域服务（Domain Service），领域事件（Domain Event），资源库（Repository），工厂（Factory），聚合（Aggregate），应用服务（Application Service）
- 演进后的领域驱动设计过程



使用用户故事提炼需求

- 在收集需求时，可使用用例或用户故事

  > As a（作为）<**角色**>
  >
  > I would like（我希望）<**活动**>
  >
  > so that（以便于）<**业务价值**>

- 实战：作业提交系统的用户故事

根据需求建立统一语言

- 获取统一语言就是需求分析的过程，也是团队就系统目标、范围与具体功能达成一致的过程
- 统一语言建立后，团队中所有成员都要使用统一语言，包括业务专家、程序员（代码）
- 统一语言提供了统一的领域术语和领域行为描述

作业与实验

- 规划一个自己的应用，用于本课程后续的设计、开发与部署。要求如下：
  - 后端应该至少可划分为两个相互依赖的服务
  - 后端至少有两个服务需要对数据库进行读写操作
  - 至少有一个前端服务

  使用用户故事描述该应用的需求。每个用户故事至少要包含一个场景。
- 使用本章所讲的内容，完成