**DDD**

**软件的目标**

降本增效：可复用、可维护、可读

**软件的挑战**

熵增：规模增加导致复杂度增加

复杂度：技术复杂度和业务复杂度

**背景**

**在 OOA 阶段**，分析师（领域专家或业务专家）以需求为输入，完全从业务视角出发，对需求中的领域概念进行分析和提炼，并定义它们之间的关系，建立分析模型；

**在 OOD 阶段**，设计师（架构师）以分析模型为输入，从技术视角出发，补充被分析师忽略但却与软件开发效率息息相关的因素（软硬件平台、数据存储约束、并发、编程框架...），进一步按照软件工程的要求（各种设计原则和模式）重塑了分析师给出的分析模型，建立设计模型；

**在 OOP 阶段**，程序员（开发人员）根据设计模型来编写代码，并根据代码的反馈来演进设计模型。

DDD 抛弃了将分析模型与设计模型分离的做法，而是寻找单个模型来满足两方面的要求，这就是领域模型。领域模型是业务视角和技术视角的交集，它反映了业务人员和技术人员的共识。对领域模型的建立，必须由业务人员和技术人员达成一致。随着领域模型的演进及精炼，技术实现越来越逼近业务的本质，软件系统可以真实反映业务需求，且易于理解和维护。

**领域驱动设计**

**什么是领域驱动设计**

一种模型驱动的设计思想，使用模型捕捉领域知识，通过领域模型构造更易维护的软件

领域模型的最大价值是它提供了一种通用语言，这种语言是将领域专家和技术人员联系在一起的纽带。

知识消化：

领域模型与实现关联

基于模型统一语言

充血模型

精炼模型

**背景**

克服（业务系统的）复杂度，就需要非常严格地使用领域逻辑设计方法。在近20年的时间内，一种有效的领域逻辑设计方法就是Eric Evans提出的领域驱动设计(domain-driven design)。

**其他领域逻辑设计还包括事务脚本、领域模型、表模块**。

前提

1. 迭代开发
2. 开发与领域专家一起

通过战略和战术设计而成的领域模型应该是架构中立的。

**目标**

解决业务复杂度

**挑战**

• 为创建通用语言腾出时间和精力

• 持续地将领域专家引入项目

• 改变开发者对领域的思考方式

DDD 的精髓是对边界的划分和控制，共有四重边界：

第一重边界是在问题空间分离子域，包括核心域，支撑域和通用域。

第二重边界是在解决方案空间拆分 BC(限界上下文，Bounded Context)，BC 之间的协作关系通过 Context Mapping(上下文映射) 来表达。

第三重边界是在 BC 内部分离业务复杂度和技术复杂度，形成分层架构，包括用户界面层，应用层，领域层和基础设施层。

第四重边界是在领域层引入聚合这一最小的设计单元，它从完整性与一致性对领域模型进行了有效的隔离，聚合内部包括实体、值对象、领域服务、工厂和仓储等设计元素。

**战略设计**

**前提**

统一语言：业务方与开发达成一致，降低沟通成本

**领域（定义问题）**

一个领域由多个子域组成。子域可以分为核心子域、支撑子域、通用子域

**限界上下文（解决问题）**

一个限界上下文并不一定只包含在一个子域中，但这是可能的。

**落地方法**

我们先将每个子域对应一个候选限界上下文，然后再根据业务边界、工作边界和技术边界来拆分或合并不满足约束的候选限界上下文，形成真正的限界上下文。

|  |  |
| --- | --- |
| 类型 | 说明 |
| 业务边界 | 语义相关、功能相关、数据相关、变化方向相关、生命周期相关、变化方向相关 |
| 工作边界 | 对团队协作的控制，建立了团队之间的合作模式，避免团队之间的沟通变得混乱，从而降低系统的管理复杂度 |
| 技术边界 | 对技术风险的控制，做出对系统质量属性的响应与承诺，功能复用，管理变化，确定服务之间的集成方式，从而降低系统的技术复杂度 |

子域和限界上下文的关系有三种：

* 一对一的映射关系，即一个子域对应一个限界上下文（这是最简单的关系）；
* 一对多的映射关系，即一个子域可以拆分为多个限界上下文（这里用了拆分两字，意味着这个子域的任何一个限界上下文都不能再包含其他子域）；
* 多对一的映射关系，即多个子域可以被一个限界上下文包含（这里用了包含两字，意味着这个限界上下文的任何一个子域不能再被包含在其他限界上下文中）。

**领域专家**

领域专家并不是一个职位，他可以是精通业务的任何人。他们可能了解更多的关于业务领域的背景知识，他们可能是软件产品的设计者，甚至有可能是销售员。

**通用语言**

通用语言是团队自己创建的公用语言，团队中同时包含领域专家和软件开发人员。

**上下文映射图**

不同界限上下文映射

合作关系（Partnership）

共享内核（Shared Kernel）

客户-供应商（Customer-Supplier）

跟随者（Conformist）

防腐层（Anticorruption Layer）

开放主机服务（Open Host Service）

发布语言（Published Language）

各行其道（Separate Ways）

大泥球（Big Ball of Mud）

**战术设计**

**实体**

**值对象**

**聚合**

聚合可以由单个实体（Entity）组成，也可以由一组实体和值对象（Value Object）组成

* 在聚合边界内保护业务规则不变性；
* 聚合要设计得小巧；
* 只能通过标识符引用其他聚合；
* 使用最终一致性更新其他聚合。

**资源库**

持久化聚合实例

**领域服务**

执行跨聚合的操作

**领域事件**

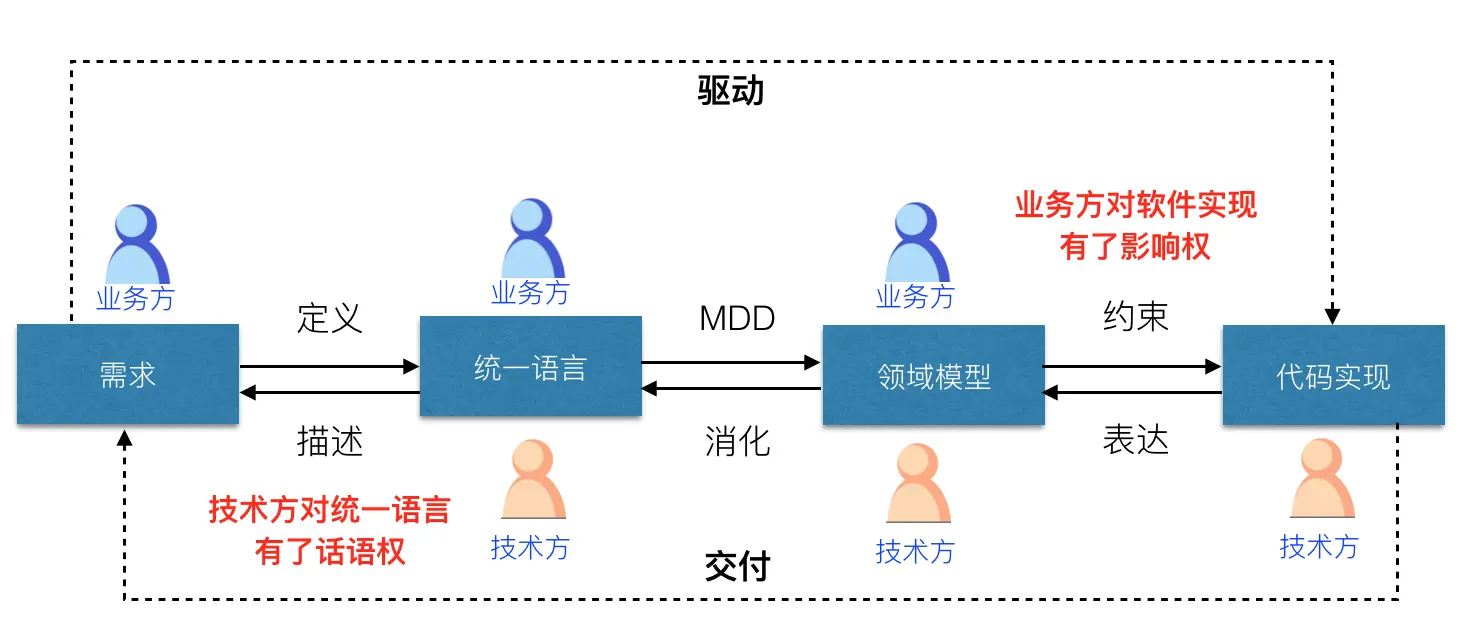
聚合发布领域事件

**模块**

一个模块包含并组织内聚在一起的领域对象

**两关联一循环**

* 领域模型与统一语言关联；
* 领域模型与代码实现关联；
* 领域模型的演进和精炼循环。



业务人员（业务方）梳理需求，并向技术人员（技术方）澄清；

业务人员和技术人员一起通过事件风暴捕获行为需求，消化领域知识，定义统一语言，同时尝试用统一语言来描述需求；

业务人员和技术人员一起完成战略设计和战术设计，完善统一语言，更深刻的理解业务，同时确定分层架构规范；

技术人员的代码实现要受到领域模型和分层架构规范的约束，分离业务和技术的关注点，凸显领域模型，同时基于面向模型的实现模式来准确的表达领域模型，让模型和代码一一映射；

领域的业务需求“驱动”出了代码实现，同时代码实现又“交付”了领域的业务需求，它们形成了一个正向反馈的循环，可以反复交替，迭代改进，使代码的实现复杂度逐步逼近业务的本质复杂度。

软件工程

一个特定的限界上下文中只使用一套通用语言，并且保证它的清晰性和简洁性。

经验萃取

首先，在领域建模过程中不应将概念与实现割裂开来。高效的领域建模人员不仅应该能够在白板上与会计师进行讨论，而且还应该能与程序员一道编写代码。

领域模型并不是按照“先建模，后实现”这个次序来工作的。像很多人一样，我也反对“先设计，再构建”这种固定的思维模式。Eric的经验告诉我们，真正强大的领域模型是随着时间演进的，即使是最有经验的建模人员也往往发现他们是在系统的初始版本完成之后才有了最好的想法。

文档，注释很难和代码保持一致，代价太高，因此可读代码非常关键

**复杂性**

行为：可预测角度，分为预测，复杂难测，混沌

结构：可理解角度，分为简单，复杂难解

**复杂性**

问题空间（功能性，质量属性），软件工程的复杂（人为可以控制）

专，多，变

DDD解决了业务复杂度

DDD是OO的扩展，提现了分治的思想

界限上下文

领域模型的知识语境

业务能力的纵向切分

一个微服务对应DDD的一个界限上下文

聚合对依赖的约束

**应用场景**

产品性软件：持续演进

业务复杂：

**使用的前提**

团队能力：对DDD的理解和认可

领域专家：不是一个角色而不是一个人。测试，开发，业务都可以。引导者，决策者

并发，事物，性能会影响领域模型，不要过于僵化遵守原则

模型是必须的吗？是否只通过统一语言即可？

统一语言是必须的吗？是否只通过模型即可？

隐藏在业务知识的复杂度如何理解？

DDD尝试解决复杂问题，因此，建立迭代循环，不断试错，修正才是最关键的。领域模型不是一蹴而就的，因人而异。

定义问题

解决问题

业务建模

是什么

首先是一个定义问题的方法，其次才是解决问题的方法

目标

与相关干系人达成共识

方法

1. 定义问题：理解，建模，共识，迭代
2. 解决问题：架构约束下实现建模

软件难题

信息经不同人传递的损耗

技术难题

业务难题：隐藏在业务中的复杂度

协作难题：业务与技术直接信息不能等价转换

注：不是复杂业务不一定需要ddd

领域建模

目标：对业务进行简化和精炼，提取领域知知识

要点

1. 统一语言描述模型：
2. 模型与实现绑定：面向对象
3. 模型与业务绑定
4. 迭代：不断优化1、2、3

修改代码就是修改模型，修改模型就修改代码

修改统一语言就修改模型，修改模型就修改统一语言

业务-统一语言-模型-代码实现-开发

软件是复杂的

复杂的软件需要多人写作

ddd本质是解决开发和业务的信息传递问题

通过模型解决业务与开发沟通中的信息丢失

1. 让开发学习业务知识而不是业务人员学习开发
2. 通过领域模型建立开发和业务沟通的桥梁

业务 - 开发的转换 - 代码（数据结构+算法）

业务- 领域模型 - 代码（数据结构+算法）

模型就是万金油吗？

业务需要将业务翻译为领域模型

开发需要讲领域模型翻译为代码

领域模型典型的是通过增加一层来解决两者之间的鸿沟，信息传递仍然有损耗

领域模型是对已有业务的抽象与描述，无法建模未知业务知识

模型是数据角度，描述不同业务维度下数据的流转

仅仅有模型就够了吗？

不，模型是数据视角，业务从业务视角理解，模型只能建模已有业务，必须有一种东西可以表达未知的业务

模型是精确的，统一语言是模糊的

模型驱动架构失败了。还需要统一语言

架构师是业务和技术沟通的桥梁。

根据需求建立模型，业务无法直接理解模型，从模型提炼统一语言，业务通过统一语言影响模型。

**统一语言**

为什么需要统一语言？

一种被实践检验过的方法。

如何建立统一语言？

统一语言是基于模型的共同语言

业务对统一语言的修改体现在代码实现上，业务对模型的修改得到开发的监督

开发对代码的重构也体现在统一语言上，重构对统一语言的修改得到业务的监督，两者之间建立了一种博弈

统一语言只有真正被使用才算统一语言，是事实而不是标准

统一语言需要在各种场景各种工种都在使用时才发挥其价值

统一语言是必须的吗？

可以通过不同的建模方法替代统一语言

领域建模的挑战

1. 业务与技术信任度不够，难以形成迭代
2. 变革的阻力比较大
3. 团队协作能力要求比较高

模型与实现关联

1. 关联对象：替代集合明确表达对象关系，解决逻辑和性能问题
2. 角色对象：同一用户在不同的领域扮演不同的角色
3. 上下文对象：显示讲不同领域之间的上下文表示为对象
4. 架构分层：领域模型不是最稳定的，将基础设施层作为能力供应商

模型作为统一语言

如何展开业务维度

1. 催化剂法

优点

缺点

1. 角色-目标-实体法

优点

缺点

方法：提炼知识

1. 需求发现
2. 沟通反馈
3. 与业务方共建模型

8X Flow

1. 微服务，分布式事务

魔球服务建模法

1. 运营角度构建saas化服务

参考

https://www.jianshu.com/p/a0d8891ee954