2004年Eric Evans发表的<领域驱动设计—软件核心复杂性应对之道>(<mDomain-Driven Design –Tackling Complexity in the Heart of Software>)一书中，提出了领域驱动设计(简称 DDD)”的概念。可以毫不夸张的说，DDD让面向对象的分析和设计进入了一个新的阶段，对企业级软件开发起到了巨大的推动作用。

DDD的思想认为，业务逻辑不再集中在几个大型的类上，而是由大量相对小的领域对象(类)组成，这些类具备自己的状态和行为，每个类是相对完整的独立体，并与现实领域的业务对象映射。领域模型就是由这样许多的细粒度的类组成。基于领域驱动的设计，保证了系统的可维护性、扩展性和复用性，在处理复杂业务逻辑方面有着先天的优势。

四个逻辑层的划分:

领域层:本层包含关于领域的信息。这是业务软件

的核心所在。在这里保留业务对象的状态,

对业务对象和它们状态的持久化被委托给

了基础设施层。

领域模型具备自己的属性行为状态，并与现实世界的业务对象相映射。各类具备明确的职责划分，领域对象元素之间通过聚合和引用等关系配合解决实际业务应用和规则。可复用，可维护，易扩展，可以采用合适的设计模型进行详细设计。

例子中的DomainModel, 上层:逻辑层，业务服务+事务处理；下层:持久化层

模型驱动设计(Model-Driven Design)抛弃了分裂分析模型与设计的做法，使用单一的模型来满足这两方面的要求。这就是领域模型。

单一的领域模型同时满足分析原型和软件设计，如果一个模型实现时不实用，重新寻找新模型。

如果模型没有忠实表达领域关键概念时，也必须重新寻找新的模型。

建模和设计成为单个迭代循环。将领域模型和设计紧密联系。因此，建模专家必须懂设计，会编程。

应用层：定义软件可以完成的工作，并且指挥具有丰富含义的领域对象来解决问题，保持精练；

不包括业务规则或知识，无业务情况的状态；

领域层：负责表示业务概念、业务状态的信息和业务规则，是业务软件核心。

领域模型种类

　　传统模型分为两种：实体（Entity）和值对象（Value Object），现在服务（Service）成为第三种模型元素。

　　实体（Entity）定义：通过一系列连续性（continuity）和标识（identity　ID）来定义；

个人认为它和分析领域的四色原型中的PPT原型非常类似，可以看成是PPT原型延续。

　　实体必须拥有自己的唯一ID，主键，如果没有一个ID标识，为每个实例加上一个具有唯一性ID，可能是内部使用。

　　值对象（Value Object）：如果一个对象代表了领域的某种描述性特征，且没有概念性的标识。

个人认为它是四色原型中Description原型延续。如果我们只关心模型中一个元素的属性，那么把这个元素划为值对象。

值对象是不可变的，不要给它任何标识，避免实体的维护性，降低设计复杂性。我们不关心值对象是哪个实例。

# 重看那本书

实体，值对象，服务，模块，聚合，工厂，资源库

# 需求分析方法演变

历史上，对需求分析方法可以说经过三个阶段：

第一阶段：围绕数据库的驱动的分析设计，新软件项目总是从设计数据库及其字段开始。这个阶段特征就是围绕数据库编程，

典型的是 DBase/Foxpro，以及后来的Delphi/VB技术。

第二阶段：面向对象的分析设计方法诞生后，有了专门的分析和设计阶段之分，我们使用UML符号来表达分析设计思想，

分析设计进入了一个相对更高的层 次，拥有了自己一套科学且艺术的方法论。

但是有一个致命缺点：分析阶段和设计阶段是断裂的，互相不能很好衔接，为什么？

首先，我们看看分析人员和设计人员在职责重点工作是什么？

分析人员的职责：是负责从需求领域中收集基本概念。

而设计人员的职责：必须指明一组能北项目中适应编程工具构造的组件，

这些组件必须能够在目标环境中有效执行，并能够正确解决应用程序出现的问题两个阶段两者目标不一致，

分析人员只管需求分析，至于是否适合设计，或者能够导出适宜设计的分析结果，这个尺度很难衡量和把握；

而设计人员因为照顾代码可运行，因此，经常可能会抱怨分析员给出的结果过于粗糙，不适合设计，这样分析设计两个阶段就导致分裂，项目失败。

在这个阶段，虽然有UML帮助，但是UML不是思想，打个比喻：会CAD的绘图员就是建筑师吗？很显然，UML就是CAD图符号，

UML不等于分析设计思想。 所以，有人说UML不是银弹，这些就象说中医不是科学一样绕人（中医就不是西医，当然就不是科学）。

第三阶段：融合了分析阶段和设计阶段的领域驱动设计（Evans: DDD）。

2004年Eric Evans 发表Domain-Driven Design –Tackling Complexity in the Heart of Software

简称Evans DDD，

领域建模是一种艺术的技术，它是用来解决复杂软件快速应付变化的解决之道，

不是科学，所以，从Evans DDD通篇文章中，你找不到科学象征的定理和公式，

当然如果 你试图寻找这样寻找，你也就陷入了“中医是不是科学”怪圈了。

Evans DDD抛弃了分裂分析模型与设计的做法，使用单一的模型来满足这两方面的要求。这就是领域模型。

单一的领域模型同时满足分析原型和软件设计，如果一个模型实现时不实用，重新寻找新模型。

如果模型没有忠实表达领域关键概念时，也必须重新寻找新的模型。建模和设计成为单个迭代循环。将领域模型和设计紧密联系。

因此，建模专家必须懂设计。

1. 什么是模型

2004年Eric Evans发表的<领域驱动设计—软件核心复杂性应对之道>(<mDomain-Driven Design –Tackling Complexity in the Heart of Software>)一书中，提出了领域驱动设计(简称 DDD)”的概念。可以毫不夸张的说，DDD让面向对象的分析和设计进入了一个新的阶段，对企业级软件开发起到了巨大的推动作用。

DDD的思想认为，业务逻辑不再集中在几个大型的类上，而是由大量相对小的领域对象(类)组成，这些类具备自己的状态和行为，每个类是相对完整的独立体，并与现实领域的业务对象映射。领域模型就是由这样许多的细粒度的类组成。基于领域驱动的设计，保证了系统的可维护性、扩展性和复用性，在处理复杂业务逻辑方面有着先天的优势。

四个逻辑层的划分:

领域层:本层包含关于领域的信息。这是业务软件

的核心所在。在这里保留业务对象的状态,

对业务对象和它们状态的持久化被委托给

了基础设施层。

领域模型具备自己的属性行为状态，并与现实世界的业务对象相映射。各类具备明确的职责划分，领域对象元素之间通过聚合和引用等关系配合解决实际业务应用和规则。可复用，可维护，易扩展，可以采用合适的设计模型进行详细设计。

例子中的DomainModel, 上层:逻辑层，业务服务+事务处理；下层:持久化层

模型驱动设计(Model-Driven Design)抛弃了分裂分析模型与设计的做法，使用单一的模型来满足这两方面的要求。这就是领域模型。

单一的领域模型同时满足分析原型和软件设计，如果一个模型实现时不实用，重新寻找新模型。

如果模型没有忠实表达领域关键概念时，也必须重新寻找新的模型。

建模和设计成为单个迭代循环。将领域模型和设计紧密联系。因此，建模专家必须懂设计，会编程。

应用层：定义软件可以完成的工作，并且指挥具有丰富含义的领域对象来解决问题，保持精练；

不包括业务规则或知识，无业务情况的状态；

领域层：负责表示业务概念、业务状态的信息和业务规则，是业务软件核心。

# 分层架构

DDD其实告诉我们如何让实现业务层！

分层架构是现代OO软件企业系统的基本架构，只有分层才能达到良好的可拓展性和维护性。

基本三层：表现层、业务层和持久层 ;J2EE中表现层和持久层有成熟框架支持，应用重点在业务层。

业务层根据Evans DDD，可以再细分为应用层和领域层两种，在业务层设计编码中，大量应用OO设计原则和设计模式。

领域层定义：负责表达业务领域概念、业务状态以及业务规则，是整个业务软件核心和重点。

应用层定义：负责完成功能，并且协调丰富的领域对象来实现功能，不能包括业务规则，无业务状态；

每个层都是内聚的，并且只依赖它的下层，为了实现各层的最大解耦，IOC/DI容器是当前Java业务层的最好选择。

没有分层架构的快速开发基本是旁门左道，不如返回Foxpro和Delphi/VB两层时代。

将本属于业务层的逻辑交由表现层来处理的快速UI方式也是一种旁门左道。

快速开发必须基于良好的质量，虽然良好的分层架构带来开发效率的降低，但是这些也是可以有方法解决。

模型提炼

如果你没有恰当的OO设计思想，Java就会用性能惩罚你，这可能是Java世界的一个潜规则。

那么，一个正确的OOA/OOD/OOP步骤是什么呢？目前围绕模型驱动设计(MDD)的设计思想成为主流思想，

MDA更是在MDD基础上提升和升华。下面让我们首先了解，如何使用领域驱动设计思想来分析设计一个软件系统。

当我们不再对一个新系统进行数据库提炼时，取而代之的时面向对象的模型提炼。

我们必须大刀阔斧地对业务领域进行细分，将一个复杂的业务领域划分为多个小的子领域，

同时还必须分清重点和次要部分，抓住核心领域概念，实现重点突破。

核心领域模型

精简模型，找出核心领域，将业务需求中最有价值的概念体现出来，

让核心变精要，这实际就是一个使复杂问题变简单的过程，也是对我们软件设计人员真正能力的考验。

核心领域模型不是轻易能够发现，特别是他处于一个纷乱复杂的众多领域模型结构中时，

核心模型通常是我们某个子领域关注的重点，例如订单模型是订单管理领域的核心；消息模型是论坛或消息领域系统的核心。

目前，分析领域有很多模式来帮助我们来提炼核心模型，例如四色原型、Martin Fowler 的分析模式等，

例如MF的"分析模式"(Analysis Patterns)中的记帐模型就是不仅仅用来记录账目数值，而且可以记录和控制账目的每一次修改。

而四色原型则是一种高于分析模式的一种原型基本模式。

1. 领域驱动设计DDD

围绕数据库设计的结果必然扭曲业务对象，不如不用OO

• 有助于团队创建一个业务部门与IT部门都能理解的通用模型，并用该模型来沟通业务需求、数据实体、过程模型。

• 模型是模块化、可扩展、易于维护的，同时设计还反映了业务模型。

• 提高了业务领域对象的可重用性和可测性。

2. DDD对应用的分层架构，及Django的对应

在Eric Evans《领域驱动设计--软件核心复杂性应对之道》中对领域的分层架构如下：

• 用户界面（表现层）：负责给用户展示信息，并解释用户命令。

• 应用层：该层协调应用程序的活动。不包括任何业务逻辑，不保存业务对象的状态，但能保存应用程序任务过程的状态。

• 领域层：这一层包括业务领域的信息。业务对象的状态在这里保存。业务对象的持久化和它们的状态可能会委托给基础设施层。

• 基础设施层：对其它层来说，这一层是一个支持性的库。它提供层之间的信息传递，实现业务对象的持久化，包含对用户界面层的支持性库等。

层间交互的对应：

A、页面提交表单数据到Action，Action创建DTO对象并设置相应属性值为表单数据

B、Action传递DTO对象给Facade

C、Facade中套用ServiceTemplate事务模板以加入事务管理，在ServiceTemplate中根据具体业务调用Factory或Reposistory，分别create或者load出DomainModel对象

D、Facade传递DomainModel对象给Service

E、Service执行具体业务逻辑（调用DomainModel对象相应的业务方法）

F、Service调用Reposistory对状态已改变的DomainModel对象进行持久化操作（调用相应DAO）

3. Domain Model的创建和开发

多次迭代，每次之迭代先检查Domain是否会变更，如果有，首先做这一步

过程：

• 寻找概念类

类图（推荐用草图）

• 开发

• 单元测试

其他层的开发

集成测试

完善、重构领域模型

使用更新的领域模型重复上述步骤

3、对于DomainModel的设计

DomainModel基于贫血模型来设计。在Robbin《总结一下最近关于domain object以及相关的讨论》一文对于贫血模型领域的切分原则写到：

引用

Rod Johnson提出原则是“case by case”，可重用度高的，和domain object状态密切关联的放在Item中，可重用度低的，和domain object状态没有密切关联的放在ItemManager中。

我提出的原则是：看业务方法是否显式的依赖持久化。

在该项目中DomainModel内的业务逻辑方法没有任何对DAO接口的依赖，如果需要对DomainModel进行持久化操作全部由 Service调用Reposistory的store()方法进行处理，对DomainModel的持久化操作已全部封装于Reposistory中。 DomainModel只提供改变自身状态的业务处理方法。

4、对DomainModel的访问1. 领域驱动设计DDD

• 有助于团队创建一个业务部门与IT部门都能理解的通用模型，并用该模型来沟通业务需求、数据实体、过程模型。

• 模型是模块化、可扩展、易于维护的，同时设计还反映了业务模型。

• 提高了业务领域对象的可重用性和可测性。

2. DDD对应用的分层架构，及Django的对应

在Eric Evans《领域驱动设计--软件核心复杂性应对之道》中对领域的分层架构如下：

• 用户界面（表现层）：负责给用户展示信息，并解释用户命令。

• 应用层：该层协调应用程序的活动。不包括任何业务逻辑，不保存业务对象的状态，但能保存应用程序任务过程的状态。

• 领域层：这一层包括业务领域的信息。业务对象的状态在这里保存。业务对象的持久化和它们的状态可能会委托给基础设施层。

• 基础设施层：对其它层来说，这一层是一个支持性的库。它提供层之间的信息传递，实现业务对象的持久化，包含对用户界面层的支持性库等。

层间交互的对应：

A、页面提交表单数据到Action，Action创建DTO对象并设置相应属性值为表单数据

B、Action传递DTO对象给Facade

C、Facade中套用ServiceTemplate事务模板以加入事务管理，在ServiceTemplate中根据具体业务调用Factory或Reposistory，分别create或者load出DomainModel对象

D、Facade传递DomainModel对象给Service

E、Service执行具体业务逻辑（调用DomainModel对象相应的业务方法）

F、Service调用Reposistory对状态已改变的DomainModel对象进行持久化操作（调用相应DAO）

3. Domain Model的创建和开发

多次迭代，每次之迭代先检查Domain是否会变更，如果有，首先做这一步

过程：

• 寻找概念类

类图（推荐用草图）

• 开发

• 单元测试

其他层的开发

集成测试

完善、重构领域模型

使用更新的领域模型重复上述步骤

3、对于DomainModel的设计

DomainModel基于贫血模型来设计。在Robbin《总结一下最近关于domain object以及相关的讨论》一文对于贫血模型领域的切分原则写到：

引用

Rod Johnson提出原则是“case by case”，可重用度高的，和domain object状态密切关联的放在Item中，可重用度低的，和domain object状态没有密切关联的放在ItemManager中。

我提出的原则是：看业务方法是否显式的依赖持久化。

在该项目中DomainModel内的业务逻辑方法没有任何对DAO接口的依赖，如果需要对DomainModel进行持久化操作全部由 Service调用Reposistory的store()方法进行处理，对DomainModel的持久化操作已全部封装于Reposistory中。 DomainModel只提供改变自身状态的业务处理方法。

4、对DomainModel的访问原则

A、必须从领域模型的根部操作模型的各个部分

从聚合的对象访问具体对象

B、必须保持领域模型的完整性。最初由Factory构造一个模型的时候，就必须将其进行完整的初始化，而不是只构造一部分，然后在接下来的处理过程中逐步完善。

5、Factory和Reposistory的职责

Factory只负责领域模型对象的创建（从无到有）；

Reposistory对持久化进行了封装，它操作的是已存在的领域模型对象（从数据库中读取数据or重新持久化到数据库）

6、项目待优化部分

在Service调用Reposistory的store()方法进行持久化操作时，如果某个业务仅对DomainModel的一部分属性进行了 修改，在持久化到数据库中时仍然会将DomainModel的所有属性全部进行一次update操作。这样的数据库操作没有必要，只需对 DomianModel中修改过的属性进行持久化操作即可。可以考虑在DomainModel上提供一个类似保存所有修改属性的列表，在 Reposistory持久化时仅对该列表中的修改属性进行操作。

1. 领域驱动设计DDD

• 有助于团队创建一个业务部门与IT部门都能理解的通用模型，并用该模型来沟通业务需求、数据实体、过程模型。

• 模型是模块化、可扩展、易于维护的，同时设计还反映了业务模型。

• 提高了业务领域对象的可重用性和可测性。

2. DDD对应用的分层架构，及Django的对应

在Eric Evans《领域驱动设计--软件核心复杂性应对之道》中对领域的分层架构如下：

• 用户界面（表现层）：负责给用户展示信息，并解释用户命令。

• 应用层：该层协调应用程序的活动。不包括任何业务逻辑，不保存业务对象的状态，但能保存应用程序任务过程的状态。

• 领域层：这一层包括业务领域的信息。业务对象的状态在这里保存。业务对象的持久化和它们的状态可能会委托给基础设施层。

• 基础设施层：对其它层来说，这一层是一个支持性的库。它提供层之间的信息传递，实现业务对象的持久化，包含对用户界面层的支持性库等。

层间交互的对应：

A、页面提交表单数据到Action，Action创建DTO对象并设置相应属性值为表单数据

B、Action传递DTO对象给Facade

C、Facade中套用ServiceTemplate事务模板以加入事务管理，在ServiceTemplate中根据具体业务调用Factory或Reposistory，分别create或者load出DomainModel对象

D、Facade传递DomainModel对象给Service

E、Service执行具体业务逻辑（调用DomainModel对象相应的业务方法）

F、Service调用Reposistory对状态已改变的DomainModel对象进行持久化操作（调用相应DAO）

3. Domain Model的创建和开发

多次迭代，每次之迭代先检查Domain是否会变更，如果有，首先做这一步

过程：

• 寻找概念类

类图（推荐用草图）

• 开发

• 单元测试

其他层的开发

集成测试

完善、重构领域模型

使用更新的领域模型重复上述步骤

3、对于DomainModel的设计

DomainModel基于贫血模型来设计。在Robbin《总结一下最近关于domain object以及相关的讨论》一文对于贫血模型领域的切分原则写到：

引用

Rod Johnson提出原则是“case by case”，可重用度高的，和domain object状态密切关联的放在Item中，可重用度低的，和domain object状态没有密切关联的放在ItemManager中。

我提出的原则是：看业务方法是否显式的依赖持久化。

在该项目中DomainModel内的业务逻辑方法没有任何对DAO接口的依赖，如果需要对DomainModel进行持久化操作全部由 Service调用Reposistory的store()方法进行处理，对DomainModel的持久化操作已全部封装于Reposistory中。 DomainModel只提供改变自身状态的业务处理方法。

4、对DomainModel的访问原则

A、必须从领域模型的根部操作模型的各个部分

从聚合的对象访问具体对象

B、必须保持领域模型的完整性。最初由Factory构造一个模型的时候，就必须将其进行完整的初始化，而不是只构造一部分，然后在接下来的处理过程中逐步完善。

5、Factory和Reposistory的职责

Factory只负责领域模型对象的创建（从无到有）；

Reposistory对持久化进行了封装，它操作的是已存在的领域模型对象（从数据库中读取数据or重新持久化到数据库）

6、项目待优化部分

在Service调用Reposistory的store()方法进行持久化操作时，如果某个业务仅对DomainModel的一部分属性进行了 修改，在持久化到数据库中时仍然会将DomainModel的所有属性全部进行一次update操作。这样的数据库操作没有必要，只需对 DomianModel中修改过的属性进行持久化操作即可。可以考虑在DomainModel上提供一个类似保存所有修改属性的列表，在 Reposistory持久化时仅对该列表中的修改属性进行操作。

原则

A、必须从领域模型的根部操作模型的各个部分

从聚合的对象访问具体对象

B、必须保持领域模型的完整性。最初由Factory构造一个模型的时候，就必须将其进行完整的初始化，而不是只构造一部分，然后在接下来的处理过程中逐步完善。

5、Factory和Reposistory的职责

Factory只负责领域模型对象的创建（从无到有）；

Reposistory对持久化进行了封装，它操作的是已存在的领域模型对象（从数据库中读取数据or重新持久化到数据库）

6、项目待优化部分

在Service调用Reposistory的store()方法进行持久化操作时，如果某个业务仅对DomainModel的一部分属性进行了 修改，在持久化到数据库中时仍然会将DomainModel的所有属性全部进行一次update操作。这样的数据库操作没有必要，只需对 DomianModel中修改过的属性进行持久化操作即可。可以考虑在DomainModel上提供一个类似保存所有修改属性的列表，在 Reposistory持久化时仅对该列表中的修改属性进行操作。

# 架构

典型的企业应用架构由下面四个概念上的层组成：

* 用户界面（表现层）：负责给用户展示信息，并解释用户命令。
* 应用层：该层协调应用程序的活动。不包括任何业务逻辑，不保存业务对象的状态，但能保存应用程序任务过程的状态。
* 领域层：这一层包括业务领域的信息。业务对象的状态在这里保存。业务对象的持久化和它们的状态可能会委托给基础设施层。
* 基础设施层：对其它层来说，这一层是一个支持性的库。它提供层之间的信息传递，实现业务对象的持久化，包含对用户界面层的支持性库等。

让我们更详细地看一下应用层和领域层。应用层：

* 负责应用中UI屏幕之间的导航，以及与其它系统应用层之间的交互。
* 还能对用户输入的数据进行基本（非业务相关）的验证，然后再把数据传到应用的其它层（更底层）。
* 不包含任何业务、领域相关的逻辑、或数据访问逻辑。
* 没有任何反映商业用例的状态，但却能处理用户会话或任务进展的状态。

领域层：

* 负责业务领域的概念，业务用例和业务规则的相关信息。领域对象封装了业务实体的状态和行为。贷款处理应用中的业务实体例子有抵押（Mortgage）、房产（Property）和贷款人（Borrower）。
* 如果用例跨越多个用户请求（比如贷款登记过程包含多个步骤：用户输入贷款详细信息，系统基于贷款特性返回产品和利率，用户选择特定的产品/利率组合，最后系统会用这个利率锁定贷款），还可以管理业务用例的状态（会话）。
* 包含服务对象，这些服务对象只包含一个定义好的、不属于任何领域对象的可操作行为。服务封装了业务领域的状态，而业务领域并不适用于领域对象本身。
* 是商业应用的核心，应该与应用的其它层隔离开来。而且，它不应该依赖于其它层使用的应用框架（JSP/JSF、[Struts](http://struts.apache.org/)、EJB、[Hibernate](http://www.hibernate.org/)、[XMLBeans](http://xmlbeans.apache.org/)等）。

DDD中：

业务层＝应用服务层＋领域模型层。

应用服务与具体的架构有关，

领域模型独立于具体的架构，可以在不同的架构下移植的。

业务核心放在领域模型层里

应用服务通过领域模型来实现具体的业务操作

领域模型＝实体＋值对象＋服务

实体：需要跟踪状态，需要标识

值对象：主要用来描述系统状态，不需要跟踪状态变化

服务：也分为领域层服务和业务层服务，我们的业务逻辑是要放在领域层服务里，业务层的服务只是协调领域层的服务来完成业务操作。这样依赖我们的领域层通过实体，值对象，服务将是充血的。       至于到底是充血还是贫血，要从整个领域层来考虑，不能片面的从个别模型元素来看待贫血还是充血。有些实体从代码级别看是失血的，但是从整个领域层的角度来看是充血的，而所谓的设计模式也正是让领域层充血的法宝。

其他要素

模块--聚合--工厂--资源库

模块:对大型的复杂项目,模型作为整体很难讨论,理解不同部件之间的关系和交互变得很困难.基于此原因,很有必要将模型组织进模块.模块被用来作为组织相关概念和任务以便降低复杂性的一种方法.

聚合是一个用来定义对象所有权和边界的领域模式. 工厂和资源库是另外的两个涉及模式,用来帮助我们处理对象的创建和存储问题.

工厂用来封装对象创建所必须的知识,它们对创建聚合特别有用.当聚合的根建立时,所有聚合包含的对象将随之建立,所有的不变量得到了强化. 将创建过程原子化非常重要,如果不这样做,创建过程就会存在对某个对象执行了一半操作的机会,将这些对象置于未定义的状态,对聚合而言更是如此.

 资源库:使用一个资源库,它的目的是封装所有获取对象引用所需的逻辑.领域对象不需要处理基础设施,以得到领域中对其它对象的所需的引用.只需从资源库中获取它们,于是模型重获它应有的清晰和焦点.

 代码设计应该围绕模型展开,模型自身也会基于设计决定而有所增进.脱离了模型的设计会导致软件不能反映它所服务的领域,甚至可能得不到期望的行为.建模如果得不到设计的反馈或者缺少了开发人员的参与,会导致必须实现模型的人很难理解它,并且可能对所用的技术不太适合.

建模的的一件事是阅读业务规范,从中寻找名词和动词.名词转换成类,而动词则成为方法.将产生前层次的模型.

# 开发

没有实际的实现，模型就没有用处。实现阶段应该尽可能多地自动化完成开发任务。为了看看什么任务能自动完成，让我们看看涉及领域模型的一个典型用例。下面是用例的步骤列表：

输入请求：

* 客户端调用外观类，以XML文档（XSD兼容的）的方式发送数据；外观类为UOW初始化一个新的事务。
* 验证输入的数据。验证包括基本验证（基本的/数据类型/属性级检查）和业务验证。如果有任何的验证错误，抛出适当的异常。
* 将描述转换为代码（以成为简单的领域）。
* 改变数据格式，以成为简单的领域模型。
* 进行所有的属性分割（比如，在客户实体对象中，将客户姓名分成名字和姓）。
* 把DTO拆分为一或多个领域对象。
* 持久化领域对象的状态。

输出响应：

* 从数据存储中获取领域对象的状态。
* 如果必要，缓存状态。
* 将领域对象组装为对应用有利的数据对象（DTO）。
* 进行所有的数据元素合并或分离（比如结合名字和姓，组成单一的客户姓名属性）。
* 将代码转换为描述。
* 必要时改变数据格式，以处理客户端数据使用的要求。
* 如果有必要，缓存DTO的状态。
* 事务提交（如果有错误则回滚），退出控制流。

下表显示了应用中不同的对象，这些对象将一个层的数据传到另一个层。

表3. 应用层间的数据流向

|  |  |  |  |
| --- | --- | --- | --- |
| 层 | 起点对象 | 终点对象 | 框架 |
| DAO | 数据库表 | DO | Hibernate |
| 领域委托 | DO | DTO | Dozer |
| 数据传输 | DTO | XML | [JAXB](https://jaxb.dev.java.net/) |

正如你所看到的，相同的数据以不同形式（DO、DTO、XML等）在应用架构中传递的层并不多。大部分持有数据的这些对象（Java或XML），还 有像 DAO、DAOImpl、DAOTest这些类实际上都是基础设施。这些有模板代码和结构的类、XML文件都很适合代码生成。

# DDD与代码

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 层**/**功能 | 模式 | 你写的代码 | 生成的代码 | 框架 |
| 数据访问 | DAO/资源库 |  | DAO接口， DAO实现类， DAOTest， 测试种子数据 | Unitils, DBUnit |
| 领域 | DO | 领域类 | DomainTest |  |
| 持久化 | ORM | 领域类 | ORM映射， ORM映射测试 | Hibernate, ORMUnit |
| 数据传输 | DTO | XSD | DTO | JAXB |
| DTO组装 | 组装 | 映射 | DO-DTO映射文件 | Dozer |
| 委托 | 业务委托 | DO到DTO的转换代码 |  |  |
| 外观 |  | 外观 |  | 远程服务， EJB, Web Service |
| 控制器 | MVC | 控制器映射文件 | Struts/Spring MVC |  |
| 表示层 | MVC | 视图配置文件 | Spring MVC |  |

委托层是唯一同时理解领域对象和DTO的层。其它层，例如持久层，不应该察觉到DTO。

面向对象与领域建模

需求分析方法演变

历史上，对需求分析方法可以说经过三个阶段：

第一阶段：围绕数据库的驱动的分析设计，新软件项目总是从设计数据库及其字段开始。这个阶段特征就是围绕数据库编程，

典型的是 DBase/Foxpro，以及后来的Delphi/VB技术。

第二阶段：面向对象的分析设计方法诞生后，有了专门的分析和设计阶段之分，我们使用UML符号来表达分析设计思想，

分析设计进入了一个相对更高的层 次，拥有了自己一套科学且艺术的方法论。

但是有一个致命缺点：分析阶段和设计阶段是断裂的，互相不能很好衔接，为什么？

首先，我们看看分析人员和设计人员在职责重点工作是什么？

分析人员的职责：是负责从需求领域中收集基本概念。

而设计人员的职责：必须指明一组能北项目中适应编程工具构造的组件，

这些组件必须能够在目标环境中有效执行，并能够正确解决应用程序出现的问题两个阶段两者目标不一致，

分析人员只管需求分析，至于是否适合设计，或者能够导出适宜设计的分析结果，这个尺度很难衡量和把握；

而设计人员因为照顾代码可运行，因此，经常可能会抱怨分析员给出的结果过于粗糙，不适合设计，这样分析设计两个阶段就导致分裂，项目失败。

在这个阶段，虽然有UML帮助，但是UML不是思想，打个比喻：会CAD的绘图员就是建筑师吗？很显然，UML就是CAD图符号，

UML不等于分析设计思想。 所以，有人说UML不是银弹，这些就象说中医不是科学一样绕人（中医就不是西医，当然就不是科学）。

第三阶段：融合了分析阶段和设计阶段的领域驱动设计（Evans: DDD）。

2004年Eric Evans 发表Domain-Driven Design –Tackling Complexity in the Heart of Software

简称Evans DDD，

领域建模是一种艺术的技术，它是用来解决复杂软件快速应付变化的解决之道，

不是科学，所以，从Evans DDD通篇文章中，你找不到科学象征的定理和公式，

当然如果 你试图寻找这样寻找，你也就陷入了“中医是不是科学”怪圈了。

Evans DDD抛弃了分裂分析模型与设计的做法，使用单一的模型来满足这两方面的要求。这就是领域模型。

单一的领域模型同时满足分析原型和软件设计，如果一个模型实现时不实用，重新寻找新模型。

如果模型没有忠实表达领域关键概念时，也必须重新寻找新的模型。建模和设计成为单个迭代循环。将领域模型和设计紧密联系。

因此，建模专家必须懂设计。

分层架构

分层架构是现代OO软件企业系统的基本架构，只有分层才能达到良好的可拓展性和维护性。

基本三层：表现层、业务层和持久层 ;J2EE中表现层和持久层有成熟框架支持，应用重点在业务层。

业务层根据Evans DDD，可以再细分为应用层和领域层两种，在业务层设计编码中，大量应用OO设计原则和设计模式。

领域层定义：负责表达业务领域概念、业务状态以及业务规则，是整个业务软件核心和重点。

应用层定义：负责完成功能，并且协调丰富的领域对象来实现功能，不能包括业务规则，无业务状态；

每个层都是内聚的，并且只依赖它的下层，为了实现各层的最大解耦，IOC/DI容器是当前Java业务层的最好选择。

没有分层架构的快速开发基本是旁门左道，不如返回Foxpro和Delphi/VB两层时代。

将本属于业务层的逻辑交由表现层来处理的快速UI方式也是一种旁门左道。

快速开发必须基于良好的质量，虽然良好的分层架构带来开发效率的降低，但是这些也是可以有方法解决。

建模与项目管理

在我们大多数从软件项目管理上寻找软件永恒解决之道时，他们可能没有意识到又在范“缘木求鱼”老毛病了，

打个比喻很容易明白这个道理：冷兵器时代（也就是火枪没有没有发明之前），各种排兵布阵可能在作战指挥时很有效；

但是到了火器时代，所有的过去作战方式就落伍了；当然到了现在信息化战争时代，更是天壤之别。

Evans DDD领域驱动建模的诞生，对过去传统的项目管理都提出挑战，当我们还在争论RUP好还是敏捷好的时候，

谁会想到我们应该采取围绕统一领域模型的迭代驱动开发呢？

有人可能还在疑惑？我接到一个大项目，那么我的建模和架构设计时间应该是5个月还是5年呢？

当然应该回答他：都不行，需求是多变且复杂的，计划赶不上变化，现在就应该开始DDD建模。

三、领域模型驱动设计(Evans DDD)之模型提炼

如果你没有恰当的OO设计思想，Java就会用性能惩罚你，这可能是Java世界的一个潜规则。

那么，一个正确的OOA/OOD/OOP步骤是什么呢？目前围绕模型驱动设计(MDD)的设计思想成为主流思想，

MDA更是在MDD基础上提升和升华。下面让我们首先了解，如何使用领域驱动设计思想来分析设计一个软件系统。

当我们不再对一个新系统进行数据库提炼时，取而代之的时面向对象的模型提炼。

我们必须大刀阔斧地对业务领域进行细分，将一个复杂的业务领域划分为多个小的子领域，

同时还必须分清重点和次要部分，抓住核心领域概念，实现重点突破。

核心领域模型

精简模型，找出核心领域，将业务需求中最有价值的概念体现出来，

让核心变精要，这实际就是一个使复杂问题变简单的过程，也是对我们软件设计人员真正能力的考验。

核心领域模型不是轻易能够发现，特别是他处于一个纷乱复杂的众多领域模型结构中时，

核心模型通常是我们某个子领域关注的重点，例如订单模型是订单管理领域的核心；消息模型是论坛或消息领域系统的核心。

目前，分析领域有很多模式来帮助我们来提炼核心模型，例如四色原型、Martin Fowler 的分析模式等，

例如MF的"分析模式"(Analysis Patterns)中的记帐模型就是不仅仅用来记录账目数值，而且可以记录和控制账目的每一次修改。

而四色原型则是一种高于分析模式的一种原型基本模式。

1. 领域驱动设计DDD

围绕数据库设计的结果必然扭曲业务对象，不如不用OO

• 有助于团队创建一个业务部门与IT部门都能理解的通用模型，并用该模型来沟通业务需求、数据实体、过程模型。

• 模型是模块化、可扩展、易于维护的，同时设计还反映了业务模型。

• 提高了业务领域对象的可重用性和可测性。

2. DDD对应用的分层架构，及Django的对应

在Eric Evans《领域驱动设计--软件核心复杂性应对之道》中对领域的分层架构如下：

• 用户界面（表现层）：负责给用户展示信息，并解释用户命令。

• 应用层：该层协调应用程序的活动。不包括任何业务逻辑，不保存业务对象的状态，但能保存应用程序任务过程的状态。

• 领域层：这一层包括业务领域的信息。业务对象的状态在这里保存。业务对象的持久化和它们的状态可能会委托给基础设施层。

• 基础设施层：对其它层来说，这一层是一个支持性的库。它提供层之间的信息传递，实现业务对象的持久化，包含对用户界面层的支持性库等。

层间交互的对应：

A、页面提交表单数据到Action，Action创建DTO对象并设置相应属性值为表单数据

B、Action传递DTO对象给Facade

C、Facade中套用ServiceTemplate事务模板以加入事务管理，在ServiceTemplate中根据具体业务调用Factory或Reposistory，分别create或者load出DomainModel对象

D、Facade传递DomainModel对象给Service

E、Service执行具体业务逻辑（调用DomainModel对象相应的业务方法）

F、Service调用Reposistory对状态已改变的DomainModel对象进行持久化操作（调用相应DAO）

3. Domain Model的创建和开发

多次迭代，每次之迭代先检查Domain是否会变更，如果有，首先做这一步

过程：

• 寻找概念类

类图（推荐用草图）

• 开发

• 单元测试

其他层的开发

集成测试

完善、重构领域模型

使用更新的领域模型重复上述步骤

3、对于DomainModel的设计

DomainModel基于贫血模型来设计。在Robbin《总结一下最近关于domain object以及相关的讨论》一文对于贫血模型领域的切分原则写到：

引用

Rod Johnson提出原则是“case by case”，可重用度高的，和domain object状态密切关联的放在Item中，可重用度低的，和domain object状态没有密切关联的放在ItemManager中。

我提出的原则是：看业务方法是否显式的依赖持久化。

在该项目中DomainModel内的业务逻辑方法没有任何对DAO接口的依赖，如果需要对DomainModel进行持久化操作全部由 Service调用Reposistory的store()方法进行处理，对DomainModel的持久化操作已全部封装于Reposistory中。 DomainModel只提供改变自身状态的业务处理方法。

4、对DomainModel的访问1. 领域驱动设计DDD

• 有助于团队创建一个业务部门与IT部门都能理解的通用模型，并用该模型来沟通业务需求、数据实体、过程模型。

• 模型是模块化、可扩展、易于维护的，同时设计还反映了业务模型。

• 提高了业务领域对象的可重用性和可测性。

2. DDD对应用的分层架构，及Django的对应

在Eric Evans《领域驱动设计--软件核心复杂性应对之道》中对领域的分层架构如下：

• 用户界面（表现层）：负责给用户展示信息，并解释用户命令。

• 应用层：该层协调应用程序的活动。不包括任何业务逻辑，不保存业务对象的状态，但能保存应用程序任务过程的状态。

• 领域层：这一层包括业务领域的信息。业务对象的状态在这里保存。业务对象的持久化和它们的状态可能会委托给基础设施层。

• 基础设施层：对其它层来说，这一层是一个支持性的库。它提供层之间的信息传递，实现业务对象的持久化，包含对用户界面层的支持性库等。

层间交互的对应：

A、页面提交表单数据到Action，Action创建DTO对象并设置相应属性值为表单数据

B、Action传递DTO对象给Facade

C、Facade中套用ServiceTemplate事务模板以加入事务管理，在ServiceTemplate中根据具体业务调用Factory或Reposistory，分别create或者load出DomainModel对象

D、Facade传递DomainModel对象给Service

E、Service执行具体业务逻辑（调用DomainModel对象相应的业务方法）

F、Service调用Reposistory对状态已改变的DomainModel对象进行持久化操作（调用相应DAO）

3. Domain Model的创建和开发

多次迭代，每次之迭代先检查Domain是否会变更，如果有，首先做这一步

过程：

• 寻找概念类

类图（推荐用草图）

• 开发

• 单元测试

其他层的开发

集成测试

完善、重构领域模型

使用更新的领域模型重复上述步骤

3、对于DomainModel的设计

DomainModel基于贫血模型来设计。在Robbin《总结一下最近关于domain object以及相关的讨论》一文对于贫血模型领域的切分原则写到：

引用

Rod Johnson提出原则是“case by case”，可重用度高的，和domain object状态密切关联的放在Item中，可重用度低的，和domain object状态没有密切关联的放在ItemManager中。

我提出的原则是：看业务方法是否显式的依赖持久化。

在该项目中DomainModel内的业务逻辑方法没有任何对DAO接口的依赖，如果需要对DomainModel进行持久化操作全部由 Service调用Reposistory的store()方法进行处理，对DomainModel的持久化操作已全部封装于Reposistory中。 DomainModel只提供改变自身状态的业务处理方法。

4、对DomainModel的访问原则

A、必须从领域模型的根部操作模型的各个部分

从聚合的对象访问具体对象

B、必须保持领域模型的完整性。最初由Factory构造一个模型的时候，就必须将其进行完整的初始化，而不是只构造一部分，然后在接下来的处理过程中逐步完善。

5、Factory和Reposistory的职责

Factory只负责领域模型对象的创建（从无到有）；

Reposistory对持久化进行了封装，它操作的是已存在的领域模型对象（从数据库中读取数据or重新持久化到数据库）

6、项目待优化部分

在Service调用Reposistory的store()方法进行持久化操作时，如果某个业务仅对DomainModel的一部分属性进行了 修改，在持久化到数据库中时仍然会将DomainModel的所有属性全部进行一次update操作。这样的数据库操作没有必要，只需对 DomianModel中修改过的属性进行持久化操作即可。可以考虑在DomainModel上提供一个类似保存所有修改属性的列表，在 Reposistory持久化时仅对该列表中的修改属性进行操作。

1. 领域驱动设计DDD

• 有助于团队创建一个业务部门与IT部门都能理解的通用模型，并用该模型来沟通业务需求、数据实体、过程模型。

• 模型是模块化、可扩展、易于维护的，同时设计还反映了业务模型。

• 提高了业务领域对象的可重用性和可测性。

2. DDD对应用的分层架构，及Django的对应

在Eric Evans《领域驱动设计--软件核心复杂性应对之道》中对领域的分层架构如下：

• 用户界面（表现层）：负责给用户展示信息，并解释用户命令。

• 应用层：该层协调应用程序的活动。不包括任何业务逻辑，不保存业务对象的状态，但能保存应用程序任务过程的状态。

• 领域层：这一层包括业务领域的信息。业务对象的状态在这里保存。业务对象的持久化和它们的状态可能会委托给基础设施层。

• 基础设施层：对其它层来说，这一层是一个支持性的库。它提供层之间的信息传递，实现业务对象的持久化，包含对用户界面层的支持性库等。

层间交互的对应：

A、页面提交表单数据到Action，Action创建DTO对象并设置相应属性值为表单数据

B、Action传递DTO对象给Facade

C、Facade中套用ServiceTemplate事务模板以加入事务管理，在ServiceTemplate中根据具体业务调用Factory或Reposistory，分别create或者load出DomainModel对象

D、Facade传递DomainModel对象给Service

E、Service执行具体业务逻辑（调用DomainModel对象相应的业务方法）

F、Service调用Reposistory对状态已改变的DomainModel对象进行持久化操作（调用相应DAO）

3. Domain Model的创建和开发

多次迭代，每次之迭代先检查Domain是否会变更，如果有，首先做这一步

过程：

• 寻找概念类

类图（推荐用草图）

• 开发

• 单元测试

其他层的开发

集成测试

完善、重构领域模型

使用更新的领域模型重复上述步骤

3、对于DomainModel的设计

DomainModel基于贫血模型来设计。在Robbin《总结一下最近关于domain object以及相关的讨论》一文对于贫血模型领域的切分原则写到：

引用

Rod Johnson提出原则是“case by case”，可重用度高的，和domain object状态密切关联的放在Item中，可重用度低的，和domain object状态没有密切关联的放在ItemManager中。

我提出的原则是：看业务方法是否显式的依赖持久化。

在该项目中DomainModel内的业务逻辑方法没有任何对DAO接口的依赖，如果需要对DomainModel进行持久化操作全部由 Service调用Reposistory的store()方法进行处理，对DomainModel的持久化操作已全部封装于Reposistory中。 DomainModel只提供改变自身状态的业务处理方法。

4、对DomainModel的访问原则

A、必须从领域模型的根部操作模型的各个部分

从聚合的对象访问具体对象

B、必须保持领域模型的完整性。最初由Factory构造一个模型的时候，就必须将其进行完整的初始化，而不是只构造一部分，然后在接下来的处理过程中逐步完善。

5、Factory和Reposistory的职责

Factory只负责领域模型对象的创建（从无到有）；

Reposistory对持久化进行了封装，它操作的是已存在的领域模型对象（从数据库中读取数据or重新持久化到数据库）

6、项目待优化部分

在Service调用Reposistory的store()方法进行持久化操作时，如果某个业务仅对DomainModel的一部分属性进行了 修改，在持久化到数据库中时仍然会将DomainModel的所有属性全部进行一次update操作。这样的数据库操作没有必要，只需对 DomianModel中修改过的属性进行持久化操作即可。可以考虑在DomainModel上提供一个类似保存所有修改属性的列表，在 Reposistory持久化时仅对该列表中的修改属性进行操作。

原则

A、必须从领域模型的根部操作模型的各个部分

从聚合的对象访问具体对象

B、必须保持领域模型的完整性。最初由Factory构造一个模型的时候，就必须将其进行完整的初始化，而不是只构造一部分，然后在接下来的处理过程中逐步完善。

5、Factory和Reposistory的职责

Factory只负责领域模型对象的创建（从无到有）；

Reposistory对持久化进行了封装，它操作的是已存在的领域模型对象（从数据库中读取数据or重新持久化到数据库）

6、项目待优化部分

在Service调用Reposistory的store()方法进行持久化操作时，如果某个业务仅对DomainModel的一部分属性进行了 修改，在持久化到数据库中时仍然会将DomainModel的所有属性全部进行一次update操作。这样的数据库操作没有必要，只需对 DomianModel中修改过的属性进行持久化操作即可。可以考虑在DomainModel上提供一个类似保存所有修改属性的列表，在 Reposistory持久化时仅对该列表中的修改属性进行操作。

# 示例应用设计

在贷款处理示例应用中用到的领域类列举如下：

实体：

* Loan
* Borrower
* UnderwritingDecision
* FundingRequest

值对象：

* ProductRate
* State

服务：

* FundingService

资源库：

* LoanRepository
* BorrowerRepository
* FundingRepository

图3展示了示例应用的领域模型图。

图**3.** 分层应用领域模型（点击查看大图）

在本文中讨论的大部分DDD设计概念和技术都在示例应用中进行了运用。像DI、AOP、注解、领域级别安全、持久化这些概念都用到了。另外，我还使用了几个开源框架来助力DDD开发和实现任务。这些框架列举如下：

* Spring
* Dozer
* Spring安全
* JAXB（用于封送处理和取消封送处理数据的Spring-WS）
* Spring Testing（用于单元测试和集成测试）
* DBUnit
* Spring Dynamic Modules

示例应用中的领域类利用Equinox和Spring DM框架部署为OSGi模块。下表显示了示例应用的模块打包细节。

表5. 打包、部署细节

|  |  |  |  |
| --- | --- | --- | --- |
| 层 | 部署工件名称 | 模块内容 | **Spring**配置文件 |
| 客户端/控制器 | loanapp-controller.jar | 控制器，客户端代理类 | LoanAppContext-Controller.xml |
| 外观 | loanapp-service.jar | 外观（远程）服务，服务代理类，XSD | LoanAppContext-RemoteServices.xml |
| 领域 | loanapp-domain.jar | 领域类、DAO，通用的DTO | LoanAppContext-Domain.xml, LoanAppContext-Persistence.xml |
| 框架 | loanapp-framework.jar | 框架，实用工具，监视（JMX）类，方面 | LoanAppContext-Framework.xml, LoanAppContext-Monitoring.xml, LoanApp-Aspects.xml |