**基本概念**

　　领域驱动设计(简称 DDD)概念来源于2004年著名建模专家Eric Evans发表的他最具影响力的书籍:《Domain-Driven Design –Tackling Complexity in the Heart of Software》(中文译名：领域驱动设计—软件核心复杂性应对之道)一书。，书中提出了“领域驱动设计(简称 DDD)”的概念。

         领域驱动设计一般分为两个阶段：

        1.   以一种领域专家、设计人员、开发人员都能理解的“通用语言”作为相互交流的工具，在不断交流的过程中发现和挖出一些主要的领域概念，然后将这些概念设计成一个领域模型；

         2.   由领域模型驱动软件设计，用代码来表现该领域模型。领域需求的最初细节，在功能层面通过领域专家的讨论得出。

       领域驱动设计告诉我们，在通过软件实现一个业务系统时，建立一个领域模型是非常重要和必要的，因为领域模型具有以下特点：

     1*.   领域模型是对具有某个边界的领域的一个抽象，反映了领域内用户业务需求的本质；领域模型是有边界的，只反应了我们在领域内所关注的部分；*

*2.   领域模型只反映业务，和任何技术实现无关；领域模型不仅能反映领域中的一些实体概念，如货物，书本，应聘记录，地址，等；还能反映领域中的一些过程概念，如资金转账，等；*

*3.   领域模型确保了我们的软件的业务逻辑都在一个模型中，都在一个地方；这样对提高软件的可维护性，业务可理解性以及可重用性方面都有很好的帮助；*

*4.   领域模型能够帮助开发人员相对平滑地将领域知识转化为软件构造；*

*5.  领域模型贯穿软件分析、设计，以及开发的整个过程；领域专家、设计人员、开发人员通过领域模型进行交流，彼此共享知识与信息；因为大家面向的都是同一个模型，所以可以防止需求走样，可以让软件设计开发人员做出来的软件真正满足需求；*

*6.  要建立正确的领域模型并不简单，需要领域专家、设计、开发人员积极沟通共同努力，然后才能使大家对领域的认识不断深入，从而不断细化和完善领域模型；*

*7.  为了让领域模型看的见，我们需要用一些方法来表示它；图是表达领域模型最常用的方式，但不是唯一的表达方式，代码或文字描述也能表达领域模型；*

*8.  领域模型是整个软件的核心，是软件中最有价值和最具竞争力的部分；设计足够精良且符合业务需求的领域模型能够更快速的响应需求变化；*

**领域驱动设计中的一些基本概念**：

**1．实体(Entity)：**

        根据Eric Evans的定义，”一个由它的标识定义的对象叫做实体”。通常实体具备唯一ID，能够被持久化，具有业务逻辑，对应现实世界业务对象。

         实体一般和主要的业务/领域对象有一个直接的关系。一个实体的基本概念是一个持续抽象的生命，可以变化不同的状态和情形，但总是有相同的标识。

需要注意的是：

         一些开发人员将实体当成了ORM意义上的实体，而不是业务所有和业务定义的领域对象。在一些实现中采用了Transaction Script风格的架构，使用贫血的领域模型。这种认识上的混乱，在领域驱动架构中，不愿意在领域对象中加入业务逻辑而导致贫血的领域模型，同时还可能使混乱的服务对象激增。

**2．值对象(Value Object)**

        值对象的定义是：描述事物的对象；更准确的说，一个没有概念上标识符描述一个领域方面的对象。这些对象是用来表示临时的事物，或者可以认为值对象是实体的属性，这些属性没有特性标识但同时表达了领域中某类含义的概念。

        通常值对象不具有唯一ID，由对象的属性描述，可以用来传递参数或对实体进行补充描述。

        作为实体属性的描述时，值对象也会被存储。在UML的类图上显现为一对多或一对一的关系。在ORM映射关系上需要采用较复杂的一对多或一对一关系映射。

        关于实体与值对象的一个例子：比如员工信息的属性，如住址，电话号码都可以改变；然而，同一个员工的实体的标识将保持不变。因此，一个实体的基本概念是一个持续抽象的生命，可以变化不同的状态和情形，但总是有相同的标识。

**实体与值对象的区别**

         实体具有唯一标识，而值对象没有唯一标识，这是实体和值对象间的最大不同。

        实体就是领域中需要唯一标识的领域概念。有两个实体，如果唯一标识不一样，那么即便实体的其他所有属性都一样，也认为是两个不同的实体；一个实体的基本概念是一个持续抽象的生命，可以变化不同的状态和情形，但总是有相同的标识。

        不应该给实体定义太多的属性或行为，而应该寻找关联，发现其他一些实体或值对象，将属性或行为转移到其他关联的实体或值对象上。

        如果两个对象的所有的属性的值都相同，我们会认为它们是同一个对象的话，那么我们就可以把这种对象设计为值对象。值对象在判断是否是同一个对象时是通过它们的所有属性是否相同，如果相同则认为是同一个值对象；而实体是否为同一个实体的区分，只是看实体的唯一标识是否相同，而不管实体的属性是否相同。

        值对象另外一个明显的特征是不可变，即所有属性都是只读的。因为属性是只读的，所以可以被安全的共享；当共享值对象时，一般有复制和共享两种做法，具体采用哪种做法还要根据实际情况而定。

        箴言：如果值对象时可共享的，它们应该是不可变的。（值对象应该保持尽量的简单）

         值对象的设计应尽量简单，不要让它引用很多其他的对象，因为本质上讲值对象只是代表一个值。

**3．聚合及聚合根(Aggregate、Aggregate Root)：**

        聚合是用来定义领域对象所有权和边界的领域模式。聚合的作用是帮助简化模型对象间的关系。聚合，它通过定义对象之间清晰的所属关系和边界来实现领域模型的内聚，并避免了错综复杂的难以维护的对象关系网的形成。聚合定义了一组具有内聚关系的相关对象的集合，我们把聚合看作是一个修改数据的单元。

        划分Aggregation是对领域模型的进一步深化，Aggregation能阐释领域模型内部对象之间的深层关联．对Aggregation的划分会直接映射到程序结构上．比如：ＤＤＤ推荐按Aggregation设计model的子包．每个Aggregation配备一个Repository.Aggregation内部的非root对象是通过导航获得的．

        一个聚合是一组相关的被视为整体的对象。每个聚合都有一个根对象（聚合根实体），从外部访问只能通过这个对象。根实体对象有组成聚合所有对象的引用，但是外部对象只能引用根对象实体。

         只有聚合根才能使用仓储库直接查询，其它的只能通过相关的聚合访问。如果根实体被删除，聚合内部的其它对象也将被删除。

         通常，我们把聚合组织到一个文件夹或一个包中。每一个聚集对应一个包，并且每个聚集成员包括实体、值对象，domain事件，仓储接口和其它工厂对象。

**聚合有以下一些特点**：

　　1. 每个聚合有一个根和一个边界，边界定义了一个聚合内部有哪些实体或值对象，根是聚合内的某个实体；

　　2. 聚合内部的对象之间可以相互引用，但是聚合外部如果要访问聚合内部的对象时，必须通过聚合根开始导航，绝对不能绕过聚合根直接访问聚合内的对象，也就是说聚合根是外部可以保持对它的引用的唯一元素；

　　3. 聚合内除根以外的其他实体的唯一标识都是本地标识，也就是只要在聚合内部保持唯一即可，因为它们总是从属于这个聚合的；

　　4. 聚合根负责与外部其他对象打交道并维护自己内部的业务规则；

　　5. 基于聚合的以上概念，我们可以推论出从数据库查询时的单元也是以聚合为一个单元，也就是说我们不能直接查询聚合内部的某个非根的对象；

　　6. 聚合内部的对象可以保持对其他聚合根的引用；

　　7. 删除一个聚合根时必须同时删除该聚合内的所有相关对象，因为他们都同属于一个聚合，是一个完整的概念。

**如何识别聚合**？

        聚合中的对象关系是内聚的，即这些对象之间必须保持一个固定规则，固定规则是指在数据变化时必须保持不变的一致性规则。

        当我们在修改一个聚合时，我们必须在事务级别确保整个聚合内的所有对象满足这个固定规则。

        作为一条建议，聚合尽量不要太大，否则即便能够做到在事务级别保持聚合的业务规则完整性，也可能会带来一定的性能问题。

        有分析报告显示，通常在大部分领域模型中，有70%的聚合通常只有一个实体，即聚合根，该实体内部没有包含其他实体，只包含一些值对象；另外30%的聚合中，基本上也只包含两到三个实体。这意味着大部分的聚合都只是一个实体，该实体同时也是聚合根。

**如何识别聚合根**？

　　如果一个聚合只有一个实体，那么这个实体就是聚合根；如果有多个实体，可以思考聚合内哪个对象有独立存在的意义并且可以和外部直接进行交互。

       并不是所有的实体都是聚集根，但只有实体才能成为聚集根。

**4．工厂(Factories)：**

       工厂用来封装创建一个复杂对象尤其是聚合时所需的知识，作用是将创建对象的细节隐藏起来。客户传递给工厂一些简单的参数，然后工厂可以在内部创建出一个复杂的领域对象然后返回给客户。当创建 实体和值对象复杂时建议使用工厂模式。

       不意味着我们一定要使用工厂模式。如果创建对象很简单，使用构造器或者控制反转/依赖注入容器足够创建对象的依赖。此时，我们就不需要通用工厂模式来创建实体或值对象。

**良好工厂的要求**：

       每个创建方法都是原子的。一个工厂应该只能生产透明状态的对象。对于实体，意味着创建整个聚合时满足所有的不变量。

      一个单独的工厂通常生产整个聚合，传出一个根实体的引用，确保聚合的不变量都有。如果对象的内部聚合需要工厂，通常工厂方法的逻辑放在在聚合根上。这样对外部隐藏了聚合内聚的实现，同时赋予了根确保聚合完整的职责。如果聚合根不是子实体工厂的合适的家，那么继续创建一个单独的工厂。

**5．仓储（Repositories）：**

        仓储是用来管理实体的集合。

         仓储里面存放的对象一定是聚合，原因是Domain是以聚合的概念来划分边界的；聚合作为一个整体概念，要么一起被取出来，要么一起被删除。外部访问不会单独对某个聚合内的子对象进行单独操作。因此，我们只对聚合设计仓储。

         仓储还有一个重要的特征就是分为仓储定义部分和仓储实现部分，我们在领域模型中定义仓储的接口，而在基础设施层实现具体的仓储。也符合按照接口分离模式在领域层定义仓储库接口的原则。

        注意：Repositories本身是一种领域组件，但Repositories的实现却不是领域层中的。

**Respositories和DAO：**

         DAO和Repository在领域驱动设计中都很重要。DAO是面向数据访问的，是关系型数据库和应用之间的契约。

        Repository：位于领域层，面向Aggregation Root。Repository是一个独立的抽象，使用领域的通用语言，它与DAO进行交互，并使用领域理解的语言提供对领域模型的数据访问服务的“业务接口”。

　　DAO方法是细粒度的，更接近数据库，而Repository方法的粒度粗一些，而且更接近领域。领域对象应该只依赖于Repository接口。客户端应该始终调用领域对象，领域对象再调用DAO将数据持久化到数据 存储中。

　　处理领域对象之间的依赖关系（比如实体及其Repository之间的依赖关系）是开发人员经常遇到的典型问题。解决这个问题通 常的设计方案是让服务类或外观类直接调用Repository，在调用Repository的时候返回实体对象给客户端。

**6．服务（Services）：**

         服务这个词在服务模式中是这么定义的：服务提供的操作是它提供给使用它的客户端，并突出领域对象的关系。

         所有的Service只负责协调并委派业务逻辑给领域对象进行处理，其本身并真正实现业务逻辑，绝大部分的业务逻辑都由领域对象承载和实现了。

         Service可与多种组件进行交互，这些组件包括：其他的Service、领域对象和Repository 或 DAO。

         通常，应用中一般包括：Domain模型服务和应用层服务：

        \*  Domain services encapsulate domain concepts that just are not naturally modeled as things.

        \*  Application services constitute the application, or service, layer.

        当一个领域操作被视为一个重要的领域概念，一般就应该作为领域服务。 服务应该是无状态的。

        设计实现领域服务来协调业务逻辑，只在领域服务中实现领域逻辑的调用。

        领域服务逻辑须以非常干净简洁的代码实现。因此，我们必须实现对领域低层组件的调用。通常应用的调用，例如仓储库的调用，创建事务等，不应该在这里实现。这些操作应该在应用层实现。

          通常服务对象名称中都应包含一个动词。 Service接口的传入传出参数也都应该是DTO，可能包含的工作有领域对象和DTO的互转换以及事务。

      服务的3个特征：

　　a. 服务执行的操作涉及一个领域概念，这个领域概念通常不属于一个实体或者值对象

　　b. 被执行的操作涉及到领域中其它的对象

　　c. 操作时无状态的

推荐：最好显式声明服务，因为它创建了领域中一个清晰的特性，封装了一个概念领域层服务和基础设施层服务：均建立在领域实体和值对象的上层，以便直接为这些相关的对象提供所需的服务；

**领域服务与Domain对象的区别**

        一般的领域对象都是有状态和行为的，而领域服务没有状态只有行为。需要强调的是领域服务是无状态的，它存在的意义就是协调领域对象共同完成某个操作，所有的状态还是都保存在相应的领域对象中。

        通常，对开发人员来说创建不应该存在的服务相当容易；要么在服务中包含了本应存在于领域对象中的领域逻辑，要么扮演了缺失的领域对象角色，而这些领域对象并没有作为模型的一部分去创建。

**7．Domain事件**

        Domain Event模式最初由udi dahan提出，发表在自己的博客上：http://www.udidahan.com/2009/06/14/domain-events-salvation/

        企业级应用程序事件大致可以分为三类：系统事件、应用事件和领域事件。领域事件的触发点在领域模型（Domain Model）中。它的作用是将领域对象从对repository或service的依赖中解脱出来，避免让领域对象对这些设施产生直接依赖。它的做法就是当领域对象的业务方法需要依赖到这些对象时就发出一个事件，这个事件会被相应的对象监听到并做出处理。

        通过使用领域事件，我们可以实现领域模型对象状态的异步更新、外部系统接口的委托调用，以及通过事件派发机制实现系统集成。另外，领域事件本身具有自描述性。它不仅能够表述系统发生了什么事情，而且还能够描述发生事件的动机。

         Domain事件也用表进行存储。

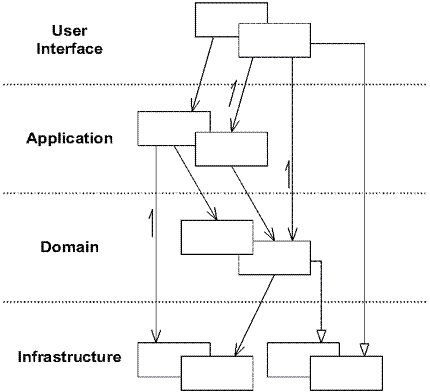
**8．DTO**

       DTO- DataTransfer Object（数据传输对象）：DTO在设计之初的主要考量是以粗粒度的数据结构减少网络通信并简化调用接口。

**领域驱动架构与N层架构设计**

**领域驱动架构**

        Eric  Evans的“领域驱动设计- 应对软件的复杂性“一书中描述和解释了建议的N层架构高层次的图：



**User Interface：**

        该层包含与其他系统/客户进行交互的接口与通信设施，在多数应用里，该层可能提供包括Web Services、RMI或Rest等在内的一种或多种通信接口。该层主要由Facade、DTO和Assembler三类组件构成，三类组件均是典型的J2EE模式。

        DTO的作用最初主要是以粗粒度的数据结构减少网络通信并简化调用接口。在领域驱动设计中，采用DTO模型，可以起到隐藏领域细节，帮助实现独立封闭的领域模型的作用。

        DTO与领域对象之间的相互转换工作多由Assembler承担，也有一些系统使用反射机制自动实现DTO与领域对象之间的相互转换，如Apache Common BeanUtils。

        Facade的用意在于为远程客户端提供粗粒度的调用接口。Facade本身不处理任何的业务逻辑，它的主要工作就是将一个用户请求委派给一个或多个Service进行处理，同时借助Assembler将Service传入或传出的领域对象转化为DTO进行传输。

**Application**：

         Application层中主要组件就是Service。这里需要注意的是，Service的组织粒度和接口设计依据与传统Transaction Script风格的Service是一致的，但是两者的实现却有质的区别。

　　Transaction Script(**事务脚本)的核心是过程，通过过程的调用来组织业务逻辑，**业务逻辑在服务（Service）层进行处理。大部分业务应用都可以被看成一系列事务。

**Transaction Script的特点是简单容易理解，面向过程设计。**  如果应用相对简单，在应用的生命周期里不会有基础设施技术的改变，尤其是业务逻辑很少会变动，采用Transaction Script风格简单自然，性能良好，容易理解。

        Transaction Script的缺点在于，对于复杂的业务逻辑难以保持良好的设计，事务之间的冗余代码不断增多。应用架构容易出现“胖服务层”和“贫血的领域模型”。同时，Service层积聚越来越多的业务逻辑，导致可维护性和扩展性变差

**领域模型**属于面向对象设计，领域模型具备自己的属性行为和状态，领域对象元素之间通过聚合配合解决实际业务应用。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。缺点是相对复杂，要求设计人员有良好的抽象能力。

        TransactionScript风格业务逻辑主要在Service中实现，而在领域驱动设计的架构里，**Service只负责协调并委派业务逻辑给领域对象进行处理。因此，我们可以考察这一点来识别系统是Transaction Script架构还是Domain Model架构**。在实践中，设计良好的领域设计架构在开发过程中也容易向Transaction Script架构演变。

**Domain**：

        Domain层是整个系统的核心层，该层维护一个使用面向对象技术实现的领域模型，几乎全部的业务逻辑会在该层实现。Domain层包含Entity（实体）、ValueObject(值对象)、Domain Event（领域事件）和Repository（仓储）等多种重要的领域组件。

**Infrastructure**：

        Infrastructure（基础设施层）为Interfaces、Application和Domain三层提供支撑。所有与具体平台、框架相关的实现会在Infrastructure中提供，避免三层特别是Domain层掺杂进这些实现，从而“污染”领域模型。Infrastructure中最常见的一类设施是对象持久化的具体实现。

**N层架构设计**

        层（Layers）被视为构成应用或服务的水平堆叠的一组逻辑上的组件。它们帮助区分完成不同任务的组件，提供一个最大化复用和可维护性的设计。简言之，是关于在架构方面应用关注点分离的原则。  
        在传统的多层架构中，每个解决方案的组件必须分隔到不同的层。每层的组件必须内聚而且有大约相同的抽象级别。每个一级层应该和其他的一级层松耦合。

        从最底层的抽象级别看，例如第1层。这是系统的基础层。这些抽象的步骤是一步一步的最后到最顶层。

        多层应用的关键在于对依赖的管理。传统的多层架构，层内的组件只能和同级或者低级层的组件交互。这有利于减少不同层内组件的依赖。通常有两种多层架构的设计方法：严格和灵活的。  
         \*   “严格的层设计”限定层内的组件只能和同一层、或者下一层的组件通信。即第N层只能和第N-1层交互，N-1层只能和N-2层交互，等等。

         \*  “灵活的层设计”允许层内的组件和任何低级别层交互。这种设计中，第N层可以和N-1，N-2层交互。

        这种设计由于不需要对其他层进行重复的调用，从而可以提高性能。然而，这种设计不提供层之间的同层隔离级别，使得它难以在不影响多个高级层的时候替换一个低级的层。

        由于层之间是通过定义明确的接口进行交互这一事实，很容易为各层添加替代的实现(例如 Mock  and Stubs)。

         因为高层的组件只能和底层的交互，在单独的组件上进行测试是很容易的。

**使用层的好处**   
-  功能容易确定位置，解决方案也就容易维护。层内高内聚，层间松耦合使得维护/组合层更容易。  
-  其他的解决方案可以重用由不同层暴露的功能。  
-  当项目按逻辑分层时，分布式的部署更容易实现。  
-  把层分布到不同的物理层可以提高可伸缩性；然后这一步应该进行仔细的评估，因为可能对性能带来负面影响。

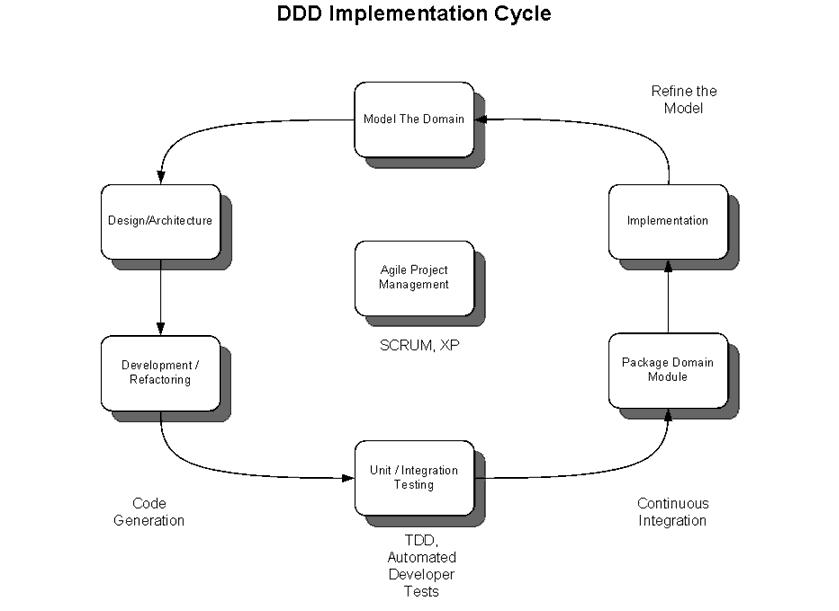
面向领域架构的分层：

        在面向领域架构中，关键是要清楚界定和分离领域模型层和其余的层。

**领域驱动与项目开发**

         一般适合结合使用scrum(适用于项目管理)和XP（适用于软件开发目标）方法对处理DDD实施项目。敏捷方法注重于交付商业价值，而DDD侧重于结合软件系统和业务模型。此 外，就DDD迭代的特性来说，SCRUM或DSDM这样的敏捷方法对项目管理来说也是更好的框架。

       DDD迭代周期的项目管理模型如图所示。



本图根据《Domain Driven Design and Development In Practice》一文中插图进行了部分修改。

　　领域建模结束时可以开始领域驱动设计。关于如何开始实现领域对象模型，Ramnivas Laddad推荐如下的步骤。他强调要更侧重于领域模型中的领域对象，而不是服务。

       \*   从领域实体和领域逻辑开始。

       \*   不要一开始就从服务层开始，只添加那些逻辑不属于任何领域实体或值对象的服务。

       \*   利用通用语言、契约式设计（DbC）、自动化测试、 CI和重构，使实现尽可能地与领域模型紧密结合。

**设计领域模型的一般步骤**：

       1.   根据需求建立一个初步的领域模型，识别出一些明显的领域概念以及它们的关联，关联可以暂时没有方向但需要有（1：1，1：N，M：N）这些关系；可以用文字精确的没有歧义的描述出每个领域概念的涵义以及包含的主要信息；

       2.   分析主要的软件应用程序功能，识别出主要的应用层的类；这样有助于及早发现哪些是应用层的职责，哪些是领域层的职责；

       3.   进一步分析领域模型，识别出哪些是实体，哪些是值对象，哪些是领域服务；

       4.   分析关联，通过对业务的更深入分析以及各种软件设计原则及性能方面的权衡，明确关联的方向或者去掉一些不需要的关联；

       5.   找出聚合边界及聚合根，这是一件很有难度的事情；因为你在分析的过程中往往会碰到很多模棱两可的难以清晰判断的选择问题，所以，需要我们平时一些分析经验的积累才能找出正确的聚合根；

       6.   为聚合根配备仓储，一般情况下是为一个聚合分配一个仓储，此时只要设计好仓储的接口即可；

       7.   走查场景，确定我们设计的领域模型能够有效地解决业务需求；

       8.   考虑如何创建领域实体或值对象，是通过工厂还是直接通过构造函数；

       9.   停下来重构模型。寻找模型中觉得有些疑问或者是蹩脚的地方，比如思考一些对象应该通过关联导航得到还是应该从仓储获取？聚合设计的是否正确？考虑模型的性能怎样，等等；

         领域建模是一个不断重构，持续完善模型的过程，大家会在讨论中将变化的部分反映到模型中，从而是模型不断细化并朝正确的方向走。

　　从设计和实现的角度来看，**典型的DDD框架应该支持以下特征**。

       \*   应该是一个以POJO为基础的架构。

       \*   应该支持使用DDD概念的业务领域模型的设计和实现。

       \*   应该支持像依赖注入（DI）和面向方向编程（AOP）这些概念的开箱即用。

       \*   与单元测试框架整合。

       \*   与其它Java/Java EE框架进行良好的集成，比如JPA、Hibernate、TopLink等。

一些反模式：

       \*   贫血的领域对象

       \*   重复的DAO

       \*   肥服务层：服务类在这里最终会包含所有的业务逻辑。

       \*   依恋情结（Feature Envy）：函数对某个类的兴趣高过对自己所处类的兴趣。

**一些思考**

1.   建立完整自封闭的领域模型。

        领域驱动架构相对比较容易理解，但建立一个完整自封闭的领域模型却很困难。“领域模型”是一个针对业务逻辑抽象的分析模型，它反映出对领域问题的整体描述。领域模型不是编程的实现模型，而是一组抽象概念的集合。一个领域概念不一定映射成一个类，也有可能会映射很多的类（包括多个实体或值对象）。领域需求的最初细节，在功能层面通过领域专家的讨论得出。领域专家并不一定需要熟知软件开发领域的知识，相反强调的是具有领域中的相关知识。领域需求在相互讨论中不断得到细化，还有可能在开发过程出现需求的反复或变更，这都要求领域模型的建立完善是一个反复重构的过程。敏捷开发是一种应对快速变化的需求的一种软件开发能力。强调程序员团队与业务专家之间的紧密协作、面对面的沟通（认为比书面的文档更有效）、频繁交付新的软件版本、紧凑而自我组织型的团队、能够很好地适应需求变化的代码编写和团队组织方法。故采用敏捷开发有利于领域模型的建立完善，以更能符合用户的实际需求。

      关于领域模型分析存在有多种分析方法。也许并不是能经常能有机会去实践这些分析方法或分析领域模型。但关于领域驱动架构的理解，有助于帮助我们去理解领域驱动的设计，实现一些高内聚、低耦合的代码实现。

2.  领域服务建模

          建立和识别领域服务也比较容易出错。通常的SSH分层架构与领域驱动架构相近，而SSH架构开发更容易导致Transaction Script架构而非领域驱动架构。在SSH分层架构上，开发人员更容易建立”贫血”模型，而在Service里实现业务逻辑。而DDD强调“充血模型”，“薄”Service层。建立领域服务需要识别出领域业务逻辑，并将业务实现到领域模型中。一方面，业务需求充满着变化，在开发过程中难以把握。当业务不明需求不清时，“贫血模型”就更容易被人接受。另一方面，在构建领域模型时，ORM映射显示十分重要并且也非常复杂，包括类继承体系与数据库的映射，抓取策略和缓存管理在内的一系列问题等．“贫血模型”有时会简化这种映射关系，同时，在处理对象依赖关系上显得更加灵活性。而领域模型强调了领域边界，对领域对象的访问总是通过聚合根开始，在有时候，模型的某些遍历会带来更大的性能和稳定性上的问题。而解决这些问题时，又常常会从实效性上出发而牺牲模型个别的清晰性和纯洁性。

3．领域对象、领域服务以及Repository之间的互相依赖

        在实际开发中，开发人员会经常需要处理领域对象之间的依赖关系，以及领域对象与Repository间的依赖。通常可能的方案是让service或Façade直接调用Repository，从而获得返回的领域对象。这种方式导致各层间的依赖，通常我们应该考虑解耦这种依赖。当前实现组件解耦常用的技术无非是：控制反转(IoC)、依赖注入(DI)、面向方面编程（AOP）以及分布式服务接口。因此，解决依赖的一种思路利用DI或AOP将Repository和服务注入到领域对象中。Spring框架提供了相关的机制。在Spring环境中，利用Spring实例化对象，注入依赖，将服务、Repository等领域对象联系起来。

         还有一种思路是采用事件驱动架构，通过Domain Event异步消息机制来实现。

4．架构不能保证领域驱动设计的贯彻与执行

        领域模型的建立完善是一个反复重构的过程。重构本身需要改变或调整应用代码，但不修改应用已有功能或行为。重构可以是设计相关的，也可以是代码相关的。设计重构就是为了不断完善模型、重构代码来提升领域模型。在代码重构时，由于各方面的原因，如需求变更、开发人员对领域驱动的理解以及对需求的理解，都有可能导致“胖服务”“贫血模型”等反模式的出现。在每次重构过程中都有出现这种趋势。如没有严格的规定，对重构代码的审查，以及可靠的测试（包括单元测试、自动代测试）。代码的变化都无没保证领域驱动设计的贯彻与执行。因此进行重构应该结合使用单元测试、自动化测试，同时的代码审查，以确保代码变动不会破坏已有的功能或改变原有的行为模式，同时严格实现建立的领域模型。

参考：[Domain Driven Design and Development In Practice](http://www.infoq.com/articles/ddd-in-practice)

[Domain-Driven Design in an Evolving Architecture](http://www.infoq.com/articles/ddd-evolving-architecture)

[Microsoft – Domain Oriented N-Layered .NET 4.0 App Sample](http://microsoftnlayerapp.codeplex.com/documentation)