今天的企业应用程序无疑是复杂的，需要依靠一些专门技术（持久性，AJAX，Web服务等）来完成他们的工作。作为开发人员，我们倾向于关注这些技术细节，这是可以理解的。但事实是，一个不能解决业务需求的系统对任何人都没用，无论它看起来多么漂亮或者如何很好地构建其基础设施。

领域驱动设计（DDD）的理念- 首先由Eric Evans在他的同名书中描述 - 是关于将我们的注意力放在应用程序的核心，关注业务领域固有的复杂性本身。我们还将核心域（业务独有）与支持子域（通常是通用的，如钱或时间）区分开来，并将更多的设计工作放在核心上。

领域驱动设计包含一组用于从领域模型构建企业应用程序的模式。在您的软件生涯中，您可能已经遇到过许多这些想法，特别是如果您是OO语言的经验丰富的开发人员。但将它们一起应用将允许您构建真正满足业务需求的系统。

在本文中，我将介绍DDD的一些主要模式，了解一些新手似乎很难解决的问题，并突出显示一些工具和资源，以帮助您在工作中应用DDD。

代码和模型……

使用DDD，我们希望创建问题域的模型，持久性，用户界面和消息传递的东西可以在以后再创建，这是需要理解的业务领域，因为正在构建的系统中，可以区分公司的业务、核心竞争力以及竞争对手情况。（如果不是这样，那么考虑购买一个包装好的产品）。

根据模型，我们不是指一个或一组图表; 确实，图表很有用，但它们不是模型，只是模型的不同*视图*（参见图）。模型是我们选择在软件中实现的一组概念，用代码表示，以及用于构建交付系统的任何其他软件工件。换句话说，代码就是模型。文本编辑器提供了一种使用此模型的方法，尽管现代工具也提供了大量其他可视化（UML类图，实体关系图，Spring beandocs，Struts / JSF流等）。

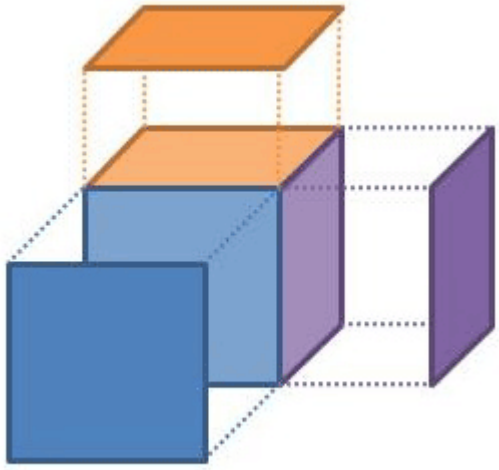

图1：模型与模型的视图

这是DDD模式中的第一个概念：***模型驱动设计***。这意味着能够将模型中的概念映射到设计/代码的概念（理想情况下），模型的变化意味着代码的变化; 更改代码意味着模型已更改。DDD并没有强制要求您使用面向对象来构建领域 - 例如，我们可以使用规则引擎构建模型 - 但是考虑到主要的企业

编程语言是基于OO的，大多数模型本质上都是OO。毕竟，OO基于建模范例。模型的概念将表示为类和接口，作为类成员的职责。

谈谈语言

现在让我们看一下领域驱动设计的另一个基本原则。回顾一下：我们想要捕获一个问题域的域模型，并且我们将在代码/软件工件中表达成某种理解。为了帮助我们做到这一点，DDD提倡领域专家和开发人员有意识地使用模型中的概念进行沟通。因此，领域专家不会根据屏幕上的字段或菜单项来描述新的用户故事需求，而是讨论领域对象所需的基础属性或行为。类似地，开发人员不会讨论数据库表中的数据列以及新字段类型。

DDD严格要求我们开发出一种**无处不在的语言**。如果一个想法不能轻易地明确表达，那么它实际上在暗示背后有一个概念，这个概念在领域模型中缺失了，并且团队需要共同努力找出缺失的概念是什么。一旦建立了这个，那么数据库表中的屏幕或数据表列上的新字段等结果就自然产生。

像DDD一样，这种发现无处不在的语言的想法并不是一个新想法：XPers称之为“名称系统”，多年来DBA将数据字典放在一起。但无处不在的语言*是*一个令人回味的术语，可以出售给商业和技术人员。现在，“整个团队”敏捷实践正在成为主流，这也很有意义。

模型和上下文……

每当我们讨论模型时，它总是在某种情况下（某种背景条件下），通常可以从使用该系统的最终用户的使用情况来推断出这个上下文背景。比如，我们有一个部署到交易员前台的交易系统，或超市收银员使用的销售点系统，这些用户以特定方式与模型的概念相关，并且模型的术语对这些用户有意义，但不一定对该上下文之外的任何其他人有意义。DDD称之为***有界上下文***（*BC*）。每个域模型都只存在于一个BC中，BC只包含一个域模型。

我必须承认，当我第一次读到关于BC时，我看不出重点：如果BC与领域模型一样，为什么要引入一个新术语？如果只有最终用户与BC进行了互动，那么也许就不需要这个术语了。然而，不同的系统（BC）也相互交互，发送文件，传递消息，调用API等。如果我们知道有两个BC相互交互，那么我们知道我们必须注意进行概念之间进行转换：此域和其他域之间。

在模型周围设置明确的边界也意味着我们可以开始讨论这些BC之间的关系。实际上，DDD确定了BC之间的一整套关系，因此当我们需要将不同的BC链接在一起时，我们可以合理地确定应该做什么：

- **已发布的语言published language**：交互式BC是就共同的语言（例如企业服务总线上的一堆XML模式）达成一致，通过它们可以相互交互;
- **开放主机服务open host service**：BC指定任何其他BC可以使用其服务的协议（例如RESTful Web服务）；
- **共享内核shared kernel**：两个BC使用一个共同的代码内核（例如一个库）作为一个共同的通用语言，但是否则以他们自己的特定方式执行其他的东西;
- **发布/订阅customer/supplier**：一个BC使用另一个BC的服务，并且是另一个BC的利益相关者（客户端）。因此，它可以影响该BC提供的服务;
- **跟从者conformist**：一个BC使用另一个BC的服务，但不是其他BC的利益相关者。因此，它使用“原样”（符合）BC提供的协议或API;

- **防腐败层anti-corruption layer**：一个BC使用另一个服务而不是利益相关者，但旨在通过引入一组适配器 - 反腐败层来最小化BC所依赖的变化所带来的影响。

你可以看到，在这个列表中，两个BC之间的合作耦合水平是逐渐降低（见图2）。使用已**发布的语言，**我们从BC建立一个可以与之交互的共同标准，我们不拥有这种语言，而是拥有它们所在的企业（甚至可能是行业标准）。有了**开放主机服务，**我们仍然做得很好; BC提供其作为任何其他BC调用的运行时服务的功能，但随着服务的发展（可能）将保持向后兼容性。

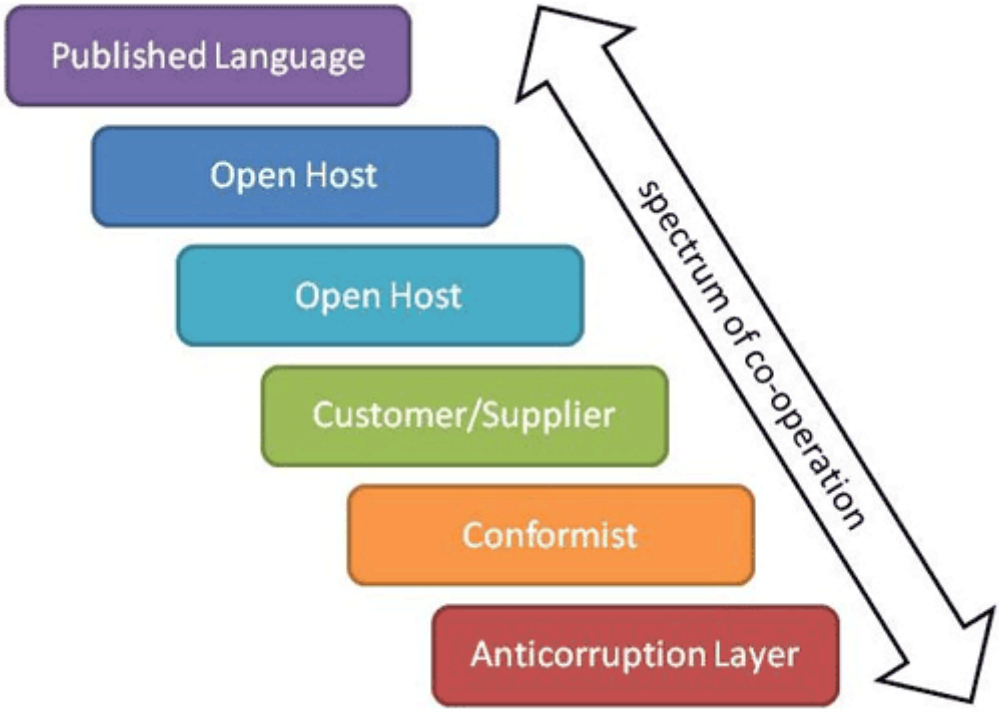

图2：有界上下文关系的谱

然而，当我们走向**跟从模式时，**我们只是一起调用和被调用; 一个BC明显屈服于另一个。如果我们必须与购买megabucks的总分类帐系统集成，那可能就是我们所处的情况。如果我们使用**反腐败层，** 那么我们通常会与遗留系统集成，但是额外的层将我们尽可能地隔离开来。当然，这需要花钱来实施，但它降低了依赖风险。反腐败层也比重新实现遗留系统便宜很多，这最多会分散我们对核心域的注意力，最坏的情况是以失败告终。

DDD建议我们制定一个**BC图**来识别我们的BC以及我们依赖或依赖的BC，以确定这些依赖关系的性质。图3显示了我过去5年左右一直在研究的系统的上下文映射。

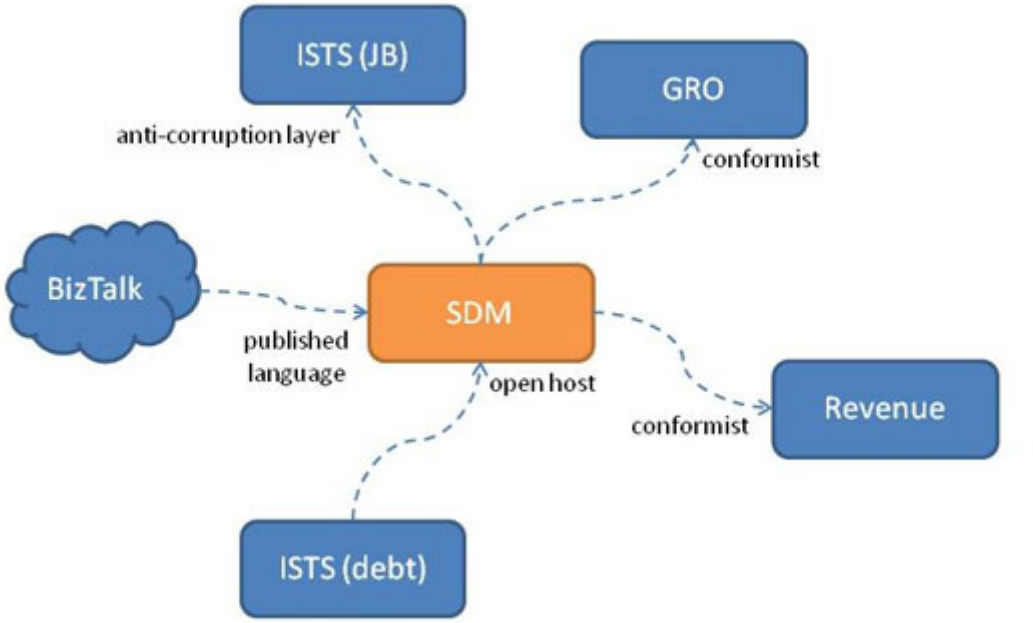

图3：上下文映射示例

所有这些关于有界上下文图和BC的讨论有时被称为**战略性DDD**，并且是有充分的理由的。毕竟，当你想到它时，弄清楚BC之间的关系是非常具有战略重要的：我的系统将依赖哪些上游系统，我是否容易与它们集成，我是否有利用它们，我相信它们吗？下游也是如此：哪些系统将使用我的服务，如何将我的功能作为服务公开，他们是否会对我有利？误解了这一点，您的应用程序可能很容易失败。

层和六边形

现在让我们转向内部并考虑我们自己的BC（系统）的架构。从根本上说，DDD只关心领域层，实际上它并没有很多关于其他层的说法：比如表现层，应用程序层或基础架构层（或持久层）。但它确实期望它们存在。这是**分层架构**模式（图4）。

图4：分层架构

当然，我们多年来一直在构建多层系统，但这并不意味着我们必须擅长它。确实，过去的一些主流技术 - 例如EJB 2，对，我说的是它！ - 对领域模型可以作为有意义的层存在的想法产生了积极的影响。所有的业务逻辑似乎渗透到应用层或（更糟糕的）表现层，留下一组贫血的领域对象作为数据持有者的空壳(DTO或VO)，这不是DDD的意思。

因此，要绝对清楚，应用程序层中不应存在任何领域逻辑。相反，应用程序层负责事务管理和安全性等事务。在某些架构中，它还可能负责确保从基础结构/持久层中检索的领域对象在与之交互之前已正确初始化（尽管我更喜欢基础结构层执行此操作）。

如果表现层有单独的存储空间中（比如手机终端），应用层也充当表现层和领域层之间的中介。表现层通常处理领域对象或其他对象（数据传输对象或DTO）的可序列化表示，通常每个“视图”一个。如果这些被修改，则表示层将对应用程序层的任何更改发送回去，而应用程序层确定已修改的领域对象，并从持久层加载它们，然后转发对这些领域对象的更改。

分层架构的一个缺点是：它从表现层一直到基础结构层的依赖性是线性的。但是，我们可能希望在表现层和基础结构层中支持不同的实现。如果我们想测试我们的应用程序肯定是这样的：

- 例如，FitNesse等工具允许我们从最终用户的角度验证我们系统的行为。但是这些工具通常不会通过表示层，而是直接返回到下一层，即应用层。所以从某种意义上说，FitNesse就是另一种观察者。
- 同样，我们可能有多个持久性实现。我们的生产实现可能使用RDBMS或类似技术，但是对于测试和原型设计，我们可能有一个轻量级实现（甚至可能在内存中），因此我们可以模拟持久性。

我们可能还想区分“内部”和“外部”层之间的交互，其中内部我指的是两个层完全在我们的系统（或BC）内的交互，而外部交互跨越BC。

因此，不要将我们的应用程序视为一组图层，另一种方法是将其视为六边形，如图5所示。我们的最终用户使用的是查看器以及FitNesse测试使用内部客户端API（或端口），而来自其他BC的调用（例如，RESTful用于开放主机交互，或来自ESB适配器的调用用于已发布的语言交互）命中外部客户端端口。对于后端基础架构层，我们可以看到用于替代对象存储实现的持久性端口，此外，领域层中的对象可以通过外部服务端口调用其他BC。

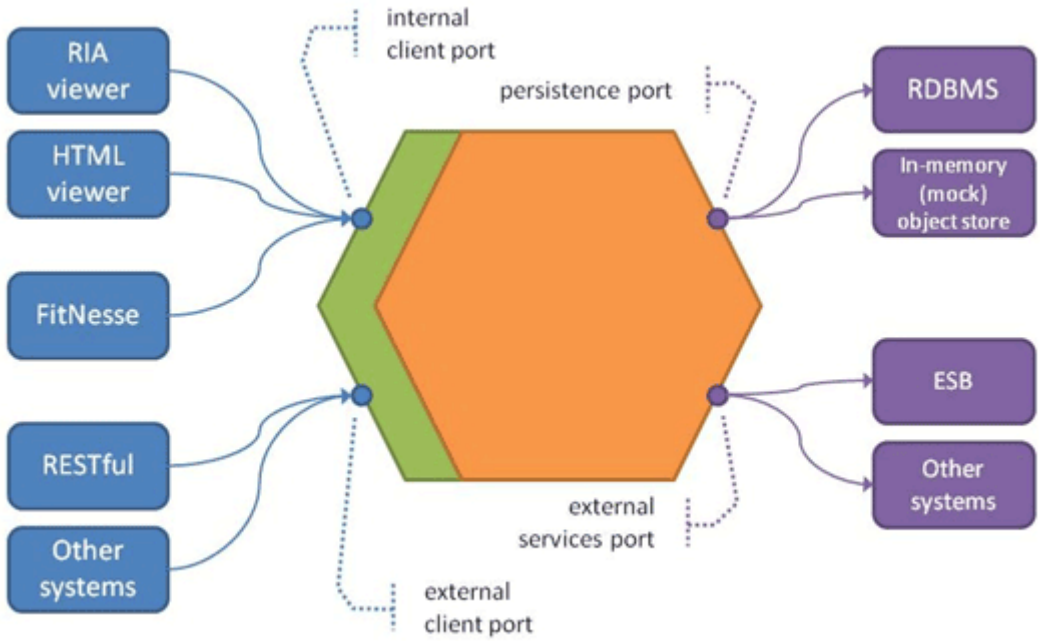

图5：六边形结构

架构模块

正如我们已经指出的那样，大多数DDD系统可能会使用OO范例。因此，我们对领域模型的元素可能很熟悉，例如**实体，值对象**和**模块**。例如，如果您是Java程序员，那么将DDD实体视为与JPA实体基本相同（使用@Entity注释）就足够安全了。

值对象是字符串，数字和日期之类的东西; 一个模块就是一个包。但是，DDD倾向于更多地强调**值对象，而**不是过去习惯于强调实体。

虽然您可以使用String类型来保存*Customer*的*givenName*属性的值，这可能是合理的，但是一笔钱例如*产品*的价格呢？也可以使用原始类型int或double，但是（甚至忽略可能的舍入错误）1或1.0是什么意思？$ 1吗？€1？¥1？1美分？

相反，我们应该引入一个*Money*值类型，它封装了*Currency* 和任何舍入规则（将特定于*Currency*）。而且，值对象应该是不可变的，并且应该提供一组无副作用的函数来操作它们。我们应该写成：

```
Money m1 = new Money("GBP", 10);
Money m2 = new Money("GBP", 20);
Money m3 = m1.add(m2);
```

将m2添加到m1不会改变m1，而是返回一个新的*Money* 对象（由m3引用），它表示一起添加的两个*Money*。

值也应该具有值语义，这意味着（例如在Java和C＃中）它们实现了*equals（）*和*hashCode（）*。它们通常也可以序列化，可以是字节流，也可以是*String*格式。当我们需要持久保存它们时，会很有用。

值对象常见的另一种情况是标识符。因此，（US）*SocialSecurityNumber(美国的社会安全码或身份证号)*是一个很好的例子，*车辆*的*车架号*也是如此。因此，它一个*网址URL*。

因为我们已经重写了*equals（）*和*hashCode（）*，所以这些都可以安全地用作哈希映射中的键key。

引入值对象不仅扩展了我们无处不在的语言，还意味着我们可以将行为推向值本身。因此，如果我们认为*Money*永远不会包含负值，我们可以在*Money*内部实现此检查，而不是在使用*Money的*任何地方。如果*SocialSecurityNumber*具有校验和数字（在某些国家/地区就是这种情况），那么该校验和的验证可以在值对象中。我们可以要求*URL*验证其格式，返回其方案（例如http），或者确定相对于其他*URL*的资源位置。

我们的另外两个元素可能需要更少的解释。 **实体**通常是持久的，通常是可变的并且（因此）倾向于具有一生的状态变化。在许多体系结构中，实体将作为行数据保存在数据库表中。**同时，模块**（包或命名空间）是确保领域模型保持解耦的关键，并且不会成为一团泥球。埃文斯在他的书中谈到了**概念轮廓**，一个优雅的短语来描述如何分离领域的主要关注领域。模块是实现这种分离的主要方式，以及确保模块依赖性严格非循环的接口。我们使用诸如Bob Martin大叔的依赖倒置原则之类的技术来确保依赖关系是严格单向的。

实体，值和模块是核心构建元素，但DDD还有一些不太熟悉的构建块。我们现在来看看这些。

聚合和聚合根

如果您精通UML，那么您将记住，它允许我们将两个对象之间的关联建模为简单关联、聚合或使用组合。一个**聚合根** （有时简称为AR）是由组合物构成的实体（以及它自己的值）。也就是说，聚合实体仅由根（可能是可传递的）引用，并且可能不被聚合外部的任何对象（永久地）引用。

换句话说，如果实体具有对另一个实体的引用，则引用的实体必须位于同一聚合内，或者是某个其他聚合的根。

许多实体是聚合根，不包含其他实体。对于不可变的实体（相当于数据库中的引用或静态数据）尤其如此。例子可能包括*Country*，*VehicleModel*， *TaxRate*，*Category*，*BOOKTITLE*等。

但是，更复杂的可变（事务）实体在建模为聚合时确实会受益，主要是通过减少概念开销。我们不必考虑每个实体，而只考虑聚合根；聚合实体仅仅是聚合的“内部运作”。它们还简化了实体之间的相互作用；我们遵循以下规则：只能将聚合根保存到数据库，而不是聚合中的任何其他实体。

另一个DDD原则是聚合根负责确保聚合实体始终处于有效状态。例如，*Order*（root）可能包含*OrderItem*的集合（聚合）。可能存在以下规则：*订单*发货后，任何*OrderItem*都无法更新；或者，如果两个*OrderItem*引用相同的*产品*并具有相同的运输要求，则它们将合并到同一个*OrderItem中*。或者，*Order*的派生*totalPrice* 属性应该是*OrderItem*的价格之和。维护这些不变量是聚合根的责任。

但是……只有聚合根才能完全在聚合中维护对象之间的不变量。*OrderItem* 中引用的*产品*几乎肯定不会在聚合根AR中，因为还有其他用例需要与*Product*交互， 而不管是否有*订单*。所以，如果有，一个规则：不能放入已经停产的产品，那么订单将需要以某种方式解决这个问题。实际上，这通常意味着：在*订单更新时* 使用隔离级别2或3来“锁定” *产品，* 这样保证以事务方式更新。或者，可以使用一种外部协调流程来协调保证聚合不变量不会被破坏。

在我们继续之前退一步，我们可以看到我们有一系列粒度：

value <entity <aggregate <module <有界上下文

现在让我们继续研究一些DDD构建方面的元素。

存储库，工厂和服务

在企业应用程序中，实体通常是持久的，其值表示这些实体的状态。但是，我们如何从持久性存储中获取实体呢?

一个数据库**库**是在持久存储的抽象，满足某些条件返回实体。例如，*Customer*存储库将返回*Customer*聚合根实体，订单存储库将返回*Order*s（及其*OrderItem*）。通常，每个聚合根有一个存储库。

因为我们通常希望支持持久性存储的多个实现，所以存储库通常由具有不同持久性存储实现的不同实现的接口（例如，*CustomerRepository*）组成（例如，*CustomerRepositoryHibernate*或*CustomerRepositoryInMemory*）。由于此接口返回实体（领域层的一部分），因此接口本身也是领域层的一部分。接口的实现（与特定的持久性实现耦合）是基础结构层的一部分。

通常，我们要搜索的条件隐含在方法名称中。因此，*CustomerRepository*可能会提供*findByLastName（String）*方法来返回具有指定姓氏的*Customer*实体。或者，我们可以有一个*OrderRepository*返回*Orders*，用*findByOrderNum（ORDERNUM）*返回匹配*ORDERNUM*（请注意使用的值类型在这里，顺便！）的*订单*。

更复杂的设计将标准包装到查询或规范中，例如*findBy（Query <T>）*，其中*Query*包含描述标准的抽象语法树。然后，不同的实现解包*查询*以确定如何以他们自己的特定方式定位满足条件的实体。

也就是说，如果你是.NET开发人员，那么值得一提的是LINQ 。因为LINQ本身是可插拔的，所以我们通常可以使用LINQ编写存储库的单个实现。然后变化的不是存储库实现，而是我们配置LINQ以获取其数据源的方式（例如，针对实体框架或针对内存中的对象库）。

每个聚合根使用特定存储库接口的变体是使用通用存储库，例如*Repository <Customer>*。这提供了一组通用方法，例如每个实体的*findById（int）*。当使用*Query <T>*（例如*Query <Customer>*）对象来指定条件时，这很有效。对于Java平台，还有一些框架，例如Hades (banq注：Spring data jdbc也支持)，它允许混合和匹配方法（从通用实现开始，然后在需要时添加自定义接口）。

存储库不是从持久层引入对象的唯一方法。如果使用对象关系映射（ORM）工具（如Hibernate），我们可以在实体之间导航引用，允许我们透明地遍历图。根据经验，对其他实体的聚合根的引用应该是延迟加载的，而聚合中的聚合实体应该被急切加载。但与ORM一样，期望进行一些调整，以便为最关键的用例获得合适的性能特征。

在大多数设计中，存储库还用于保存新实例，以及更新或删除现有实例。如果底层持久性技术支持它，那么它们很可能存在于通用存储库中，但是从方法签名的角度来看，没有什么可以区分保存新*客户*和保存新*订单*。

最后一点……直接创建新的聚合根很少见。相反，它们倾向于由其他聚合根创建。一个*订单*就是一个很好的例子：它可能是通过调用*Customer的一个*动作来创建*的*。

工厂

如果我们要求*Order*创建一个*OrderItem*，那么（因为毕竟*OrderItem*是其聚合的一部分），*Order*知道要实例化的具体*OrderItem*类是合理的。实际上，实体知道它需要实例化的同一模块（命名空间或包）中的任何实体的具体类是合理的。

假设*客户模块*使用*Customer*的*placeOrder*操作创建*订单*（参见图6）。如果*客户*知道*订单*的具体类别，则表示*客户*模块依赖于*订单*模块。如果*订单*具有对*客户*的反向引用，那么我们将在两个模块之间获得循环依赖。

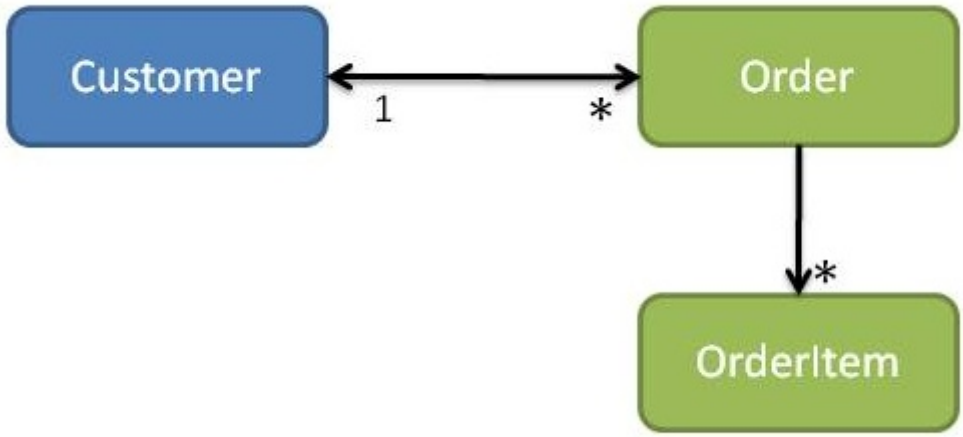

图6：客户和订单（循环依赖）

如前所述，我们可以使用依赖性反转原则来解决这类问题：从*订单* - > *客户*模块*中*删除依赖关系，将引入*OrderOwner*接口，使*Order*引用为*OrderOwner*，并使*Customer*实现*OrderOwner*（参见图7））。

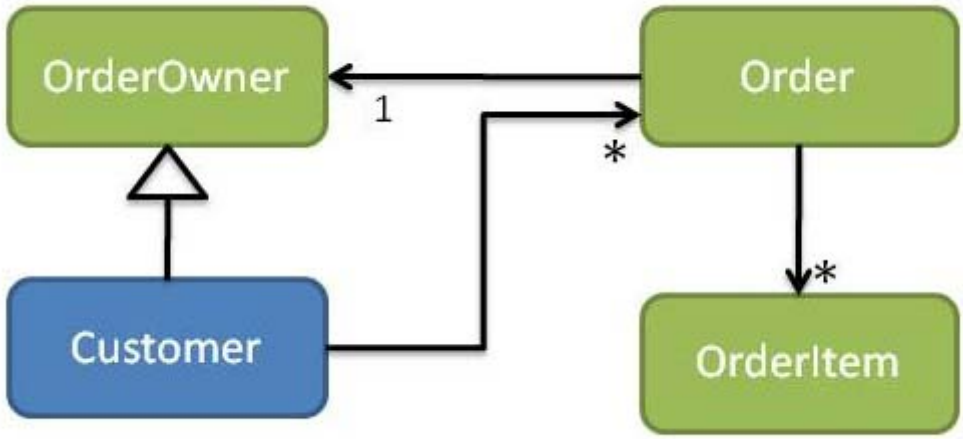

图7：客户和订单（客户取决于订单）

那么另一种方式呢：如果我们想要*订单 - > 客户*？在这种情况下，需要 在*客户*模块中有一个表示*Order*的接口（这是*Customer*的 *placeOrder*操作的返回类型）。然后，*订单*模块将提供*订单的*实现。由于*客户*不能依赖*订单*，因此必须定义*OrderFactory*接口。然后，*订单*模块依次提供*OrderFactory*的实现（参见图8）。

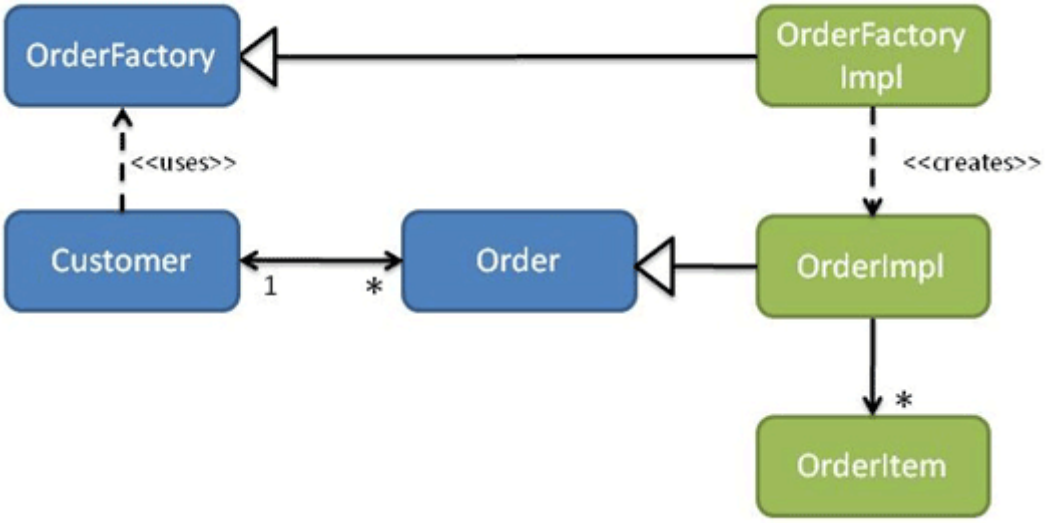

图8：客户和订单（订单取决于客户）

可能还有相应的存储库接口。例如，如果*客户*可能拥有数千个*订单*，那么我们可能会删除其*订单*集合。相反，客户将使用*OrderRepository* 根据需要定位其*订单*的（子集）。或者（如某些人所愿），您可以通过将对存储库的调用移动到应用程序体系结构的更高层（例如领域**服务**或可能是**应用程序服务**）来避免从实体到存储库的显式依赖性。

实际上，服务是我们需要探索的下一个话题。

领域服务，基础设施服务和应用服务

domain service 是限定在领域层内的，虽然实现可以是基础设施层的一部分。存储库是领域服务，其实现确实在基础结构层中，而工厂也是领域服务，其实现通常在领域层内。特别是在适当的模块中定义了存储库和工厂：*CustomerRepository*位于*客户*模块中，依此类推。

更一般地说，领域服务是任何不容易在实体中生存的业务逻辑。埃文斯建议在两个银行账户之间进行转账服务，但我不确定这是最好的例子（我会将*转账*本身建模为一个实体）。但另一种领域服务是一种充当其他有界上下文的代理。例如，我们可能希望与暴露开放主机服务的General Ledger系统集成。我们可以定义一个公开我们需要的功能的服务，以便我们的应用程序可以将条目发布到总帐Ledger。这些服务有时会定义自己的实体，这些实体可能会持久化; 这些实体实际上影响了在另一个BC中远程保存的显着信息。

我们还可以获得技术性更强的服务，例如发送电子邮件或SMS文本消息，或将*Correspondence* 实体转换为PDF，或使用条形码标记生成的PDF。接口在领域层中定义，但实现在基础架构层中非常明确。因为这些技术服务的接口通常是根据简单的值类型（而不是实体）来定义的，所以我倾向于使用术语**基础结构服务**而不是领域服务。但是如果你想成为一个“电子邮件”BC或“SMS”BC的桥梁，你可以想到它们。

虽然领域服务既可以调用领域实体，也可以由领域实体调用，但***应用服务***位于领域层之上，因此领域层内的实体不能调用领域服务，只能相反。换句话说，应用层（我们的分层架构）可以被认为是一组（无状态）应用服务。

如前所述，应用程序服务通常处理交叉和安全等交叉问题。他们还可以通过以下方式与表现层进行调解：解组入站请求; 使用领域服务（存储库或工厂）获取对与之交互的聚合根的引用; 在该聚合根上调用适当的操作; 并将结果编组回表现层。

我还应该指出，在某些体系结构中，应用程序服务调用基础结构服务。因此，应用服务可以直接调用*PdfGenerationService*，传递从实体中提取的信息，而不是实体调用*PdfGenerationService*将其自身转换为PDF 。这不是我特别喜欢的，但它是一种常见的设计。我很快就会谈到这一点。

好的，这完成了我们对主要DDD模式的概述。在Evans 500 +页面书中还有更多内容 - 值得一读 - 但我接下来要做的是强调人们似乎很难应用DDD的一些领域。