

领域对象的生命周期

每个对象都有它的生命周期。一个对象在创建以后，可能要经历各种不同的状态，并最终消亡(被存档或者删除)，如图 6-1 所示。当然，许多对象都是简单的临时对象(transient object)，我们只是调用其构造函数把它们创建出来，在某种计算中使用它们，然后就把它扔到垃圾收集器中去了。像这样的对象我们没有必要把它们搞得太复杂。但是，其他对象的生命周期更长，在生命周期中有时并不驻留内存。它们与其他的对象存在着复杂的依赖关系，在经历状态变迁时还要满足一些不变量的约束条件。如何管理这些对象给我们提出了挑战，处理不好就很容易使我们的工作偏离模型驱动设计的方向。

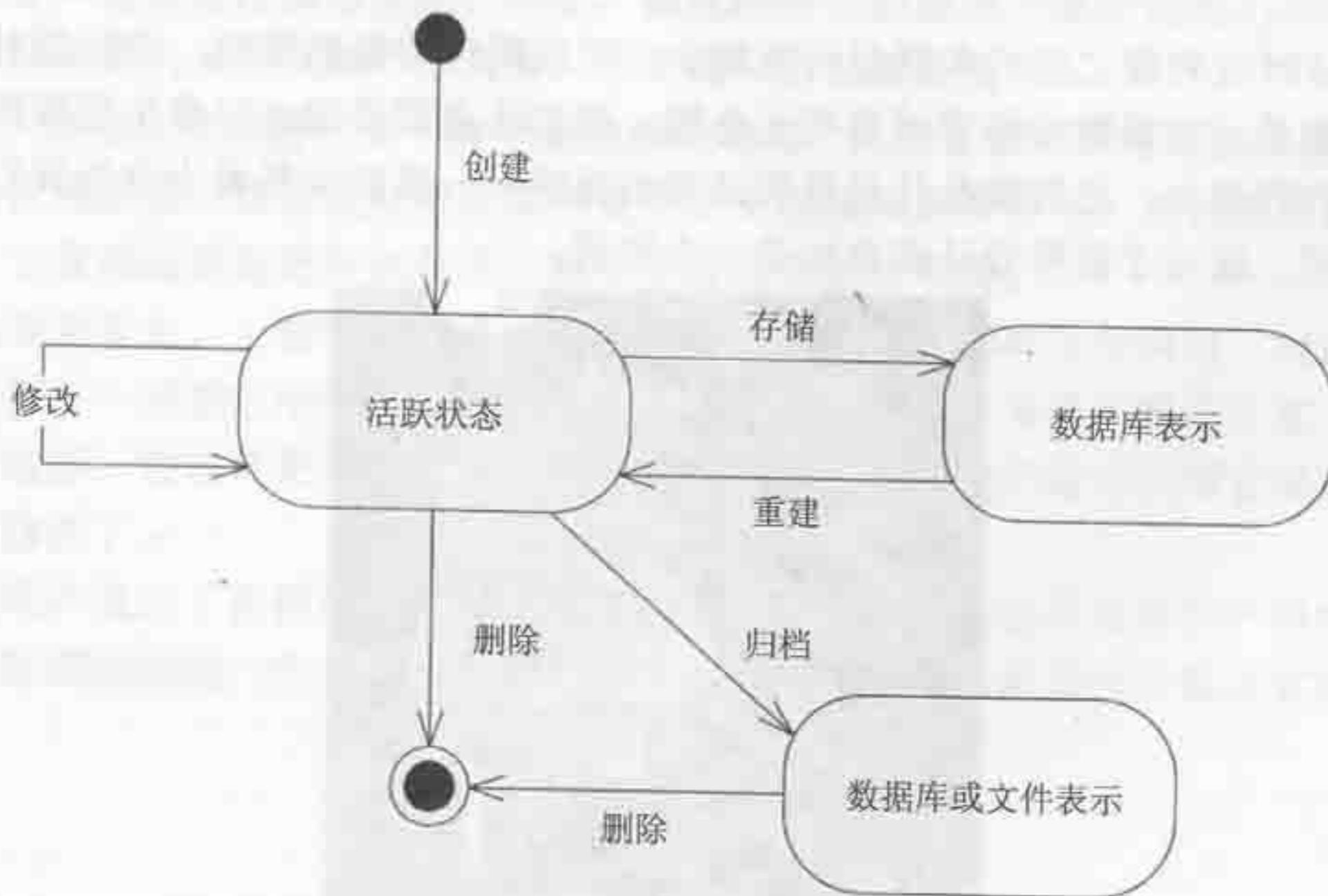


图 6-1 一个领域对象的生命周期

挑战可以分为两类：

- 在生命周期中维护对象的完整性；
- 避免模型由于管理生命周期的复杂性而陷入困境。

本章将通过 3 个模式来处理这些问题。首先，聚合(Aggregate)通过定义清晰的所有权和边界来使模型变得更紧凑，避免出现盘根错节的对象(关系)网。这个模式对于在生命周期的各个阶段中维护完整性是非常关键的。

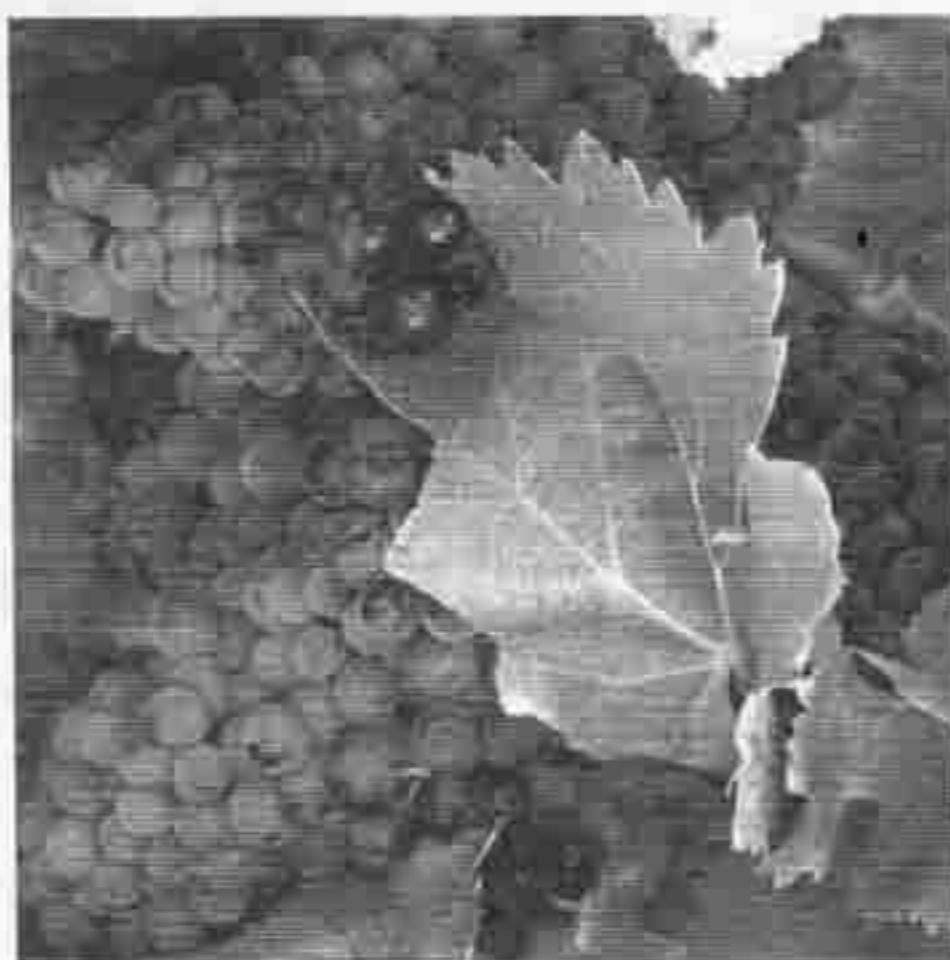
接下来，我们将重点转向生命周期的开始部分，用工厂(Factory)来创建和重组复杂的对象和聚合，并保持对其内部结构的良好封装。最后，仓储(Repository)用于处理生命周期的中间和结束部分，为我们提供查找和提取持久对象的方法，同时把与生命周期管理有关的复杂基础设施封装起来。

虽然工厂和仓储本身并不是从领域中产生的，但它们在领域设计中作用不容忽视。这些构造为我们提供了访问和控制模型对象的方法，完善了模型驱动设计。

建立聚合的模型，并把工厂和仓储加入设计中来，可以使我们系统地对模型对象进行操纵，同时使得这些对象的生命周期成为一个个意义明确的单元。聚合圈出一个范围，在这个范围中无论对象处于生命周期的哪个阶段，都应该保持不变性。工厂和仓储对聚合进行操作，将特定生命周期变迁的复杂性封装起来。

6.1 聚合

在设计时让对象之间的关联尽可能地少，可以简化对象的导航，并限制对象间关系的膨胀。但是，大多数业务领域都相互连接，我们还是需要通过对象引用跟踪较长的路径。在某种程度上，这种混乱性是真实世界的反映——真实世界很少会为我们提供清晰可见的边界。这对于软件设计而言却是一个问题。





假设我们正在从数据库中删除一个 Person 对象。附带删除的还有这个人的名字、生日和职位描述等信息。但是地址呢？其他人可能也住在同一个地址。如果把地址也删掉，那么其他 Person 对象将引用一个被删除了的地址对象。但是，如果把地址留下来，数据库中会积累一些无用的地址对象。自动垃圾收集机制可以消除这些无用的地址对象。但是，即使我们的数据库系统也提供这种机制，它终究还是一种技术上的措施，并没有解决根本的建模问题。

当我们考虑一个孤立的事务时，也有同样的问题：在一个典型的对象模型中，对象之间的关系网无法清晰地告诉我们，一个修改会产生哪些潜在的影响。我们也不能仅仅因为存在某些依赖，就把系统中的所有对象都刷新一遍，这样做显然是不现实的。

如果系统的多个客户并发地访问同一个对象，那么问题会变得更加尖锐。当系统中有许多用户对不同对象进行查询和修改时，我们必须提供保护措施(如锁定)，以阻止相互依赖的对象被同时修改。如果把保护范围弄错了，就会产生严重的后果。

在包含着复杂关联的模型中，要保证对象修改的一致性是很困难的。我们必须保证紧密关联的对象组也能保持不变性，而不仅仅只保证各个离散的对象。如果锁定策略过于谨慎，就会导致多个用户毫无必要地互相干扰，使系统变得无法使用。

换一种说法，如果一个对象是由其他对象构成的，那么我们怎样才能知道它何去何从呢？任何包含了数据持久存储的系统，肯定会有执行数据修改事务的作用域，也会有维护数据一致性(即维护数据的不变量)的机制。数据库为我们提供了不同的锁定方案，我们可以对各种情况进行编程测试。但是，像这种特定的解决方案会把我们的注意力从模型移开，很快我们又会回到“尽人事，听天命”的老路上来了。

实际上，要为这种问题找到一种平衡的解决方案，我们必须对领域理解得更加深刻，了解更多的扩展因素，例如某些类的实例发生改变的频率是多少等。我们需要找到这样一个模型，它使得远离高竞争多发点，而不变量约束更加牢固。

尽管从表面看去，上面的问题是一个很难的技术性的数据库事务问题，但问题的实际来自于模型——模型之中缺乏明确的边界。如果从模型出发来寻找解决方案，就能使模型更易于理解，使设计更易于交流。在修正了模型之后，我们就可以用它来指导我们对实现进行修改了。

人们已经开发出了在模型中定义所有权关系的方案。下面这个简单但严格的系统是由那些概念精炼而成的，它包含一系列的事务实现规则，对如何修改对象及其所有者作出了规定¹。

¹ David Siegel 发明了这个系统并在 20 世纪 90 年代将其用于项目中，但没有发表过这个系统。



首先，我们需要一种抽象机制用于在模型中对引用进行封装。一个聚合是一簇相关的对象，出于数据变化的目的，我们将这些对象视为一个单元。每个聚合都有一个根（root）和一个边界。边界定义了聚合中包含什么；根是包含在聚合中的单个特定的实体。根是聚合中唯一允许被外部对象引用的元素，但在聚合的边界内，对象之间可以互相引用。根之外的实体具有本地标识，但是它们仅仅在聚合内部才需要区分其标识，因为根实体上下文以外的外部对象不会看到它们。

我们来看一个汽车修配厂的软件中可能会用到的轿车模型，如图 6-2 所示。轿车是一个具有全局标识的实体：我们希望将一辆轿车与世界上所有其他的轿车区分开来，即使它们非常相似。这一点我们可以用车辆识别代号(VIN)来做到，每辆新车都会被指定一个唯一的车架识别代号。我们可能希望跟踪 4 个不同位置的轮胎的旋转情况，了解每个轮胎的胎面磨损度和里程。为了分清楚哪只是哪只，轮胎必须也是包含标识的实体。但是，我们只有在研究一辆特定的轿车时才会去关心那些轮胎的标识。如果我们换了胎，然后把旧胎送到回收工厂去，那么我们的软件就再也不需要对那些旧胎进行跟踪了，它们变成了一堆轮胎中的“无名之辈”。没有人会关心这些轮胎的旋转历史了。更进一步讲，就算把旧胎装到一辆轿车上，也不会有人去向系统查询一个特定的轮胎，然后问它装在哪部车上。他们会查询数据库，找到一部车，然后找它要一个轮胎的暂时引用。因此，这个聚合的根实体是轿车，而且其边界把轮胎也包含在内。另一方面，引擎组上都刻了序列号，有时需要独立于轿车单独对它们进行跟踪，因此在某些应用中，引擎可能会是它自己的聚合根。

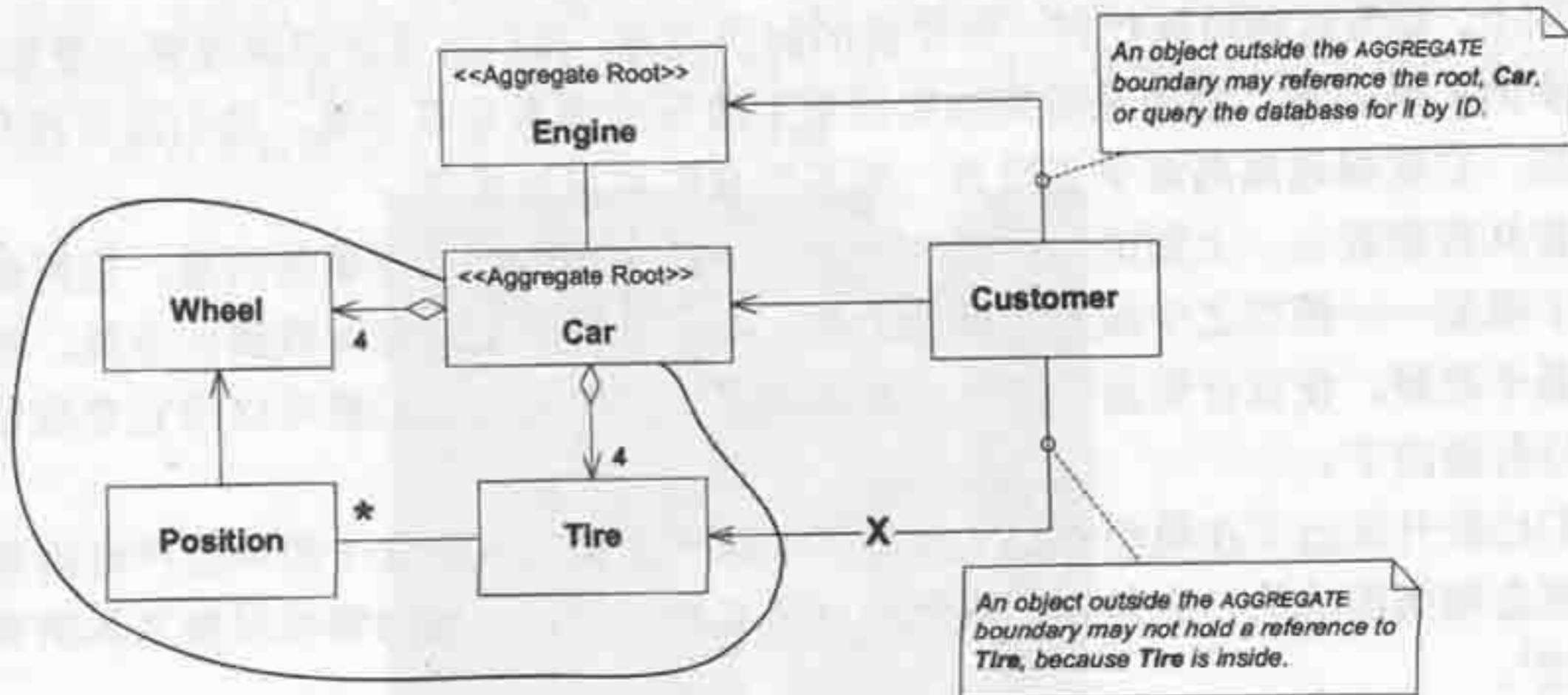


图 6-2 本地标识和全局标识以及对象引用



不变量(invariant)是指无论何时数据发生变化都必须满足的一致性规则。聚合的成员之间可能存在不变量的关系，如图 6-3 所示。我们不能指望涉及到聚合的任何规则是每时每刻都被满足的，一些依赖关系只能在某些特定的时刻，通过事件处理、批处理或其他更新机制来解决。但是，聚合内部的不变量必须在每次事务完成时就被满足。

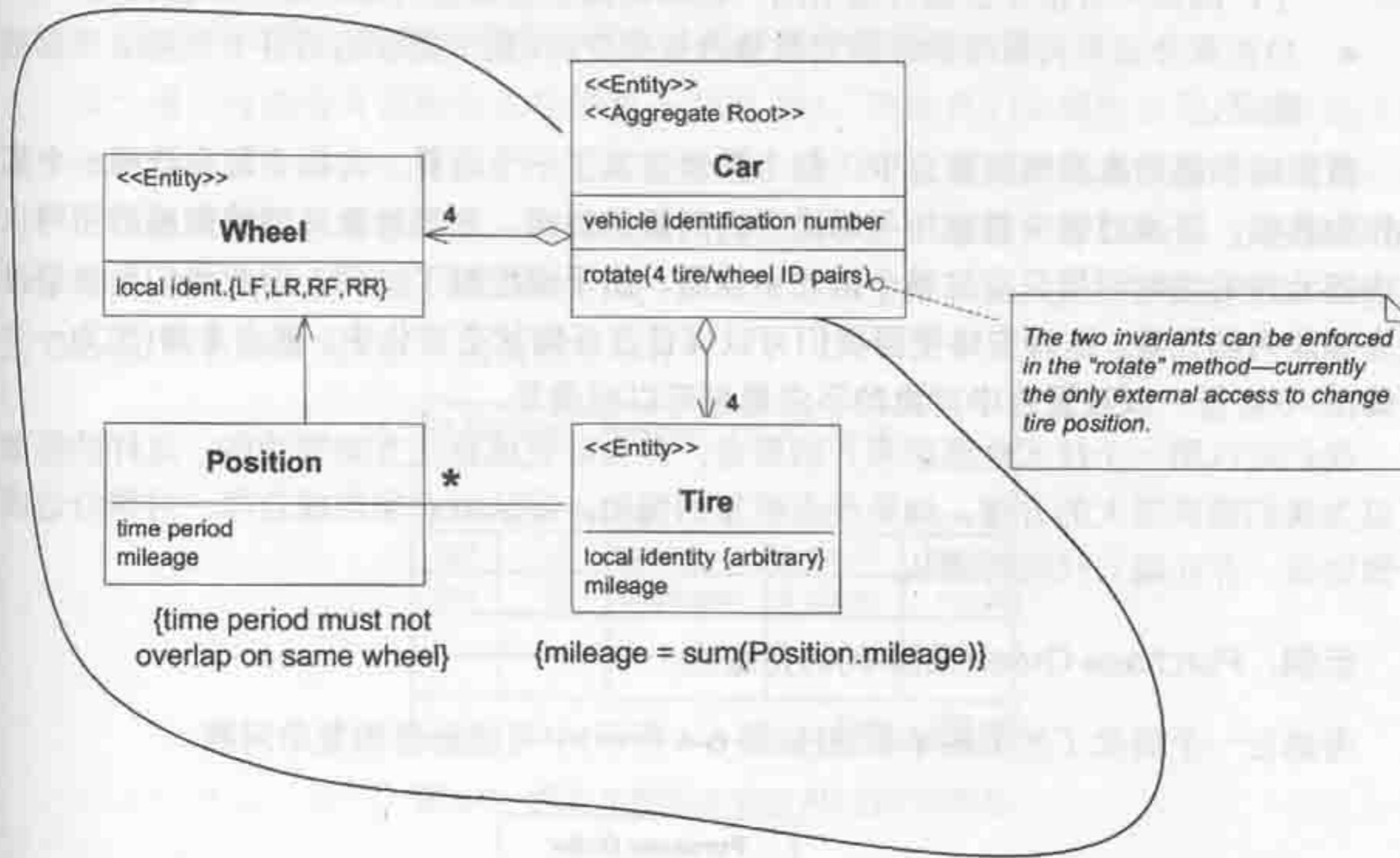


图 6-3 聚合的不变量

现在，为了将概念上的聚合转换为实现，我们需要在所有的事务中应用一系列的规则。

- 根实体具有全局标识，并最终负责对不变量的检查；
- 根实体具有全局标识。边界之内的 ENTITY 具有本地标识，这些标识仅在聚合内部是惟一的；
- 聚合边界以外的任何对象除了可以引用根实体，不能持有任何对其内部对象的引用。根实体可以把其内部实体的引用传递给其他对象，但是它们只能临时使用这种引用，而不能持有这种引用。根还可以复制一个值对象的副本传给另一个对象。它并不关心这个副本会发生什么变化，因为那只是一个值，而且与聚合已经不再有任何关联了。



- 作为上一条规则的推论，能通过数据库查询直接获得的对象只有聚合根。所有其他对象必须通过导航关联来访问；
- 聚合内的对象可以持有其他聚合根的引用。
- 删除操作必须一次性删除聚合边界内的所有对象(如果有垃圾收集器就很容易了，因为只有根才会被外部引用，删除根就会使其他内部对象全部被收集掉)。
- 当在聚合边界内发生的任何对象修改被提交时，整个聚合的所有不变量必须都被满足。

将实体和值对象聚集到聚合中。每个聚合定义了一个边界。为每个聚合选择一个实体作为其根，并通过根来控制所有对边界内对象的访问。外部对象只能持有根的引用；对内部元素的临时引用只能在单个操作中使用。由于根控制了访问，因此我们无法绕过它去修改内部元素。这种安排使得我们可以保证在任何状态变化中，聚合本身(作为一个整体)的不变量，以及聚合中对象的不变量都可以被满足。

我们可以用一个技术性框架来声明聚合，并自动完成锁定方案等功能，这样的框架可以为我们提供很大的方便。如果没有框架的帮助，团队就必须形成自律，对聚合达成一致协议，并在编写代码时遵从。

示例：Purchase Order(采购单)的完整性

考虑在一个简化了的采购单系统(如图 6-4 所示)中可能出现的复杂问题。

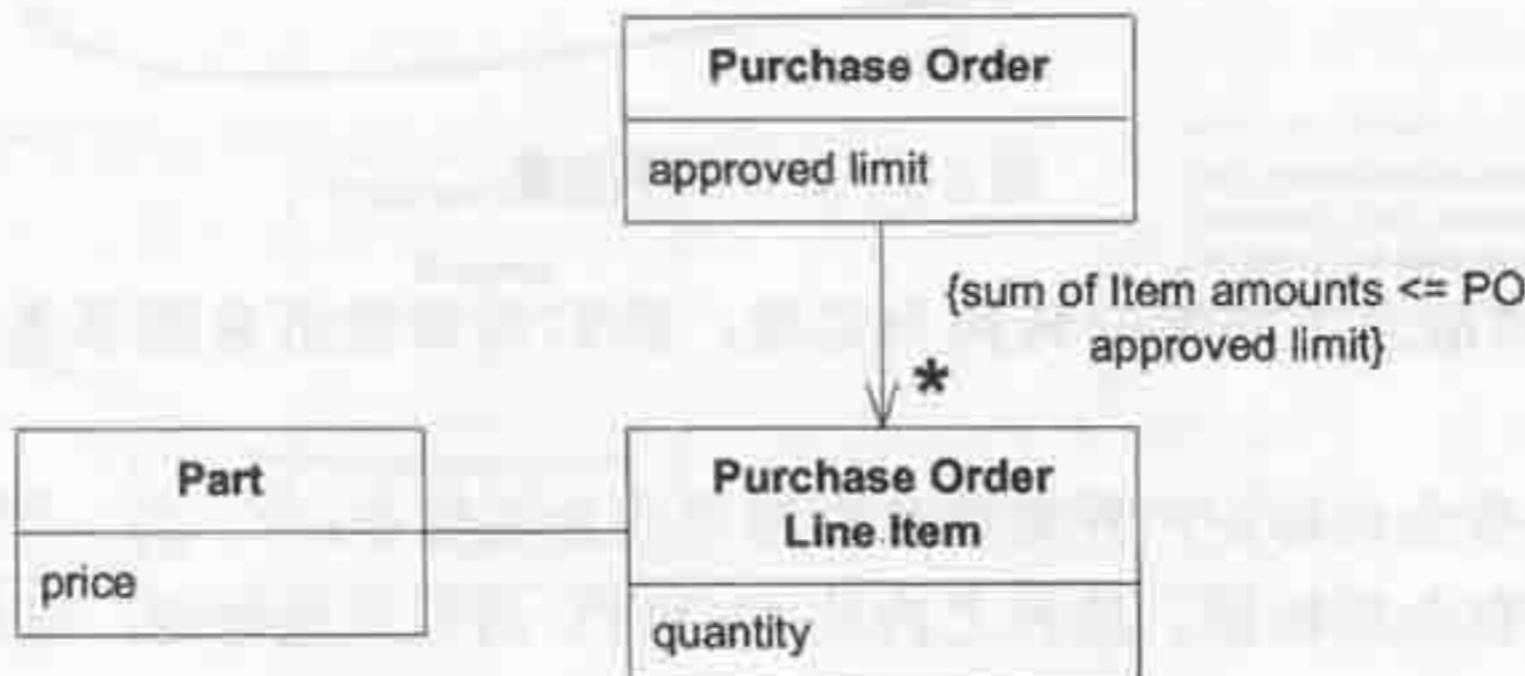


图 6-4 一个采购单系统的模型

图中对采购单(PO)作了一种常识性的描述。采购单被分解为采购单项(line item)，并包含一个不变量规则：所有采购单项的金额之和不得超过采购单的最高限额。这个实现存在 3 个互相关联的问题。



- 不变量的保证。当加入新的子项时，PO 对总金额进行检查，如果一个子项导致总金额超限，就把自己标记为非法。我们将看到这种保护并不充分。
- 变更管理。当删除或归档 PO 时，子项也被同时删除或归档。但是，关于这种级联关系何时终止，模型没有提供任何指示。在不同的时间修改商品价格会造成哪些影响也不得而知。
- 数据库共享。数据库将会出现多个用户竞争使用的问题。

多个用户可能会并发地输入和更新不同的 PO，因此我们必须防止用户对其他人的工作造成干扰。如图 6-5 所示，刚开始的时候，我们可以使用一个非常简单的策略：把用户正在编辑的所有对象锁定，直到他提交自己的事务。这样，当 George 正在编辑 001 号子项时，Amanda 就不能访问这条子项。她可以编辑任何 PO 上的任何其他订单子项(包括 George 正在编辑的 PO 上的其他子项)。

PO #0012946 Approved Limit: \$1,000.00				
Item #	Quantity	Part	Price	Amount
001	3	Guitars	@ 100.00	300.00
002	2	Trombones	@ 200.00	400.00
Total:				700.00

图 6-5 保存在数据库中的 PO 的初始情形

对象从数据库中读出，并在各个用户的内存空间中实例化(然后那些对象才能被查看和编辑)。由于我们只有在开始编辑时才请求对数据库上锁，因此 George 和 Amanda 可以并发地工作，只要他们互不编辑对方的子项，如图 6-6 所示。一切运转正常，直到 George 和 Amanda 开始编辑同一个 PO 上的不同订单子项。

George adds guitars in his view					Amanda adds a trombone in her view				
PO #0012946 Approved Limit: \$1000.00					PO #0012946 Approved Limit: \$1,000.00				
Item #	Quantity	Part	Price	Amount	Item #	Quantity	Part	Price	Amount
001	5	Guitars	@ 100.00	500.00	001	3	Guitars	@ 100.00	300.00
002	2	Trombones	@ 200.00	400.00	002	3	Trombones	@ 200.00	600.00
Total:				900.00	Total:				900.00

图 6-6 在不同事务中同时编辑



对两个用户和他们的软件来说，一切都好，因为软件不会理会数据库的其他记录在事务过程中发生了什么变化，而两个用户都没有去修改对方锁定了的子项。

如图 6-7 所示当两个用户的修改都保存完毕之后，数据库中保存的一个 PO 就违反了领域模型的不变量。一个重要的业务规则被破坏了。更糟的是，甚至没有人知道这一点。

PO #0012946 Approved Limit: \$1,000.00				
Item #	Quantity	Part	Price	Amount
001	5	Guitars	@ 100.00	500.00
002	3	Trombones	@ 200.00	600.00
Total: 1,100.00				

图 6-7 修改后的 PO 超过了批准限额，不变量被破坏了

很明显，锁定单条订单子项不能提供足够的保护。如果我们换一种方法，每次都把整个 PO 锁定，那么问题应该就可以避免了，如图 6-8 所示。

George edits his view				
PO #0012946 Approved Limit: \$1000.00				
Item #	Quantity	Part	Price	Amount
001	5	Guitars	@ 100.00	500.00
002	2	Trombones	@ 200.00	400.00
Total: 900.00				

Amanda is locked out of PO #0012946				
George's changes have been committed				

Amanda gets access; George's change shows				
PO #0012946 Approved Limit: \$1000.00				
Item #	Quantity	Part	Price	Amount
001	5	Guitars	@ 100.00	500.00
002	3	Trombones	@ 200.00	600.00
Limit exceeded →				Total: 1,100.00

图 6-8 锁定整个 PO 保证了不变量能被满足

现在，在 Amanda 解决问题之前(例如提高批准限额，或者少买一把吉他)，程序是不会允许保存这个事务的。这种机制防止了不变量被破坏的问题，而且只要用户的工作广泛地散布在许多 PO 之中，那它就是一个很好的解决方案。但是，如果多个用户常常需要同时对一个大 PO(包含很多子项)的不同订单子项进行编辑，那么这种锁定就会造成很多不便了。



即使有许多小的 PO，也还是有其他的方法来违反这个断言。考虑一下商品(part)这个对象。如果当 Amanda 正在添加她的订单时，有人修改了长号(trombone)的价格，那是不是也造成了不变量的破坏呢？

让我们试着把整个 PO 连同其 part 一起锁定起来。图 6-9 所示是当 George、Amanda 和 Sam 对不同的 PO 进行编辑时的情况。

George editing PO	Amanda adds trombones; must wait on George																																																																						
Guitars and trombones locked	Violins locked																																																																						
PO #0012946 Approved Limit: \$1,000.00	PO #0012932 Approved Limit: \$1,850.00																																																																						
<table border="1"> <thead> <tr> <th>Item #</th><th>Quantity</th><th>Part</th><th>Price</th><th>Amount</th></tr> </thead> <tbody> <tr> <td>001</td><td>2</td><td>Guitars</td><td>@ 100.00</td><td>200.00</td></tr> <tr> <td>002</td><td>2</td><td>Trombones</td><td>@ 200.00</td><td>400.00</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td align="right" colspan="2">Total:</td><td align="right">600.00</td><td></td><td></td></tr> </tbody> </table>	Item #	Quantity	Part	Price	Amount	001	2	Guitars	@ 100.00	200.00	002	2	Trombones	@ 200.00	400.00																Total:		600.00			<table border="1"> <thead> <tr> <th>Item #</th><th>Quantity</th><th>Part</th><th>Price</th><th>Amount</th></tr> </thead> <tbody> <tr> <td>001</td><td>3</td><td>Violins</td><td>@ 400.00</td><td>1,200.00</td></tr> <tr> <td>002</td><td>2</td><td>Trombones</td><td>@ 200.00</td><td>400.00</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td align="right" colspan="2">Total:</td><td align="right">1,600.00</td><td></td><td></td></tr> </tbody> </table>	Item #	Quantity	Part	Price	Amount	001	3	Violins	@ 400.00	1,200.00	002	2	Trombones	@ 200.00	400.00																Total:		1,600.00		
Item #	Quantity	Part	Price	Amount																																																																			
001	2	Guitars	@ 100.00	200.00																																																																			
002	2	Trombones	@ 200.00	400.00																																																																			
Total:		600.00																																																																					
Item #	Quantity	Part	Price	Amount																																																																			
001	3	Violins	@ 400.00	1,200.00																																																																			
002	2	Trombones	@ 200.00	400.00																																																																			
Total:		1,600.00																																																																					
Sam adds trombones; must wait on George																																																																							
PO #0013003 Approved Limit: \$15,000.00																																																																							
<table border="1"> <thead> <tr> <th>Item #</th><th>Quantity</th><th>Part</th><th>Price</th><th>Amount</th></tr> </thead> <tbody> <tr> <td>001</td><td>1</td><td>Piano</td><td>@ 1,000.00</td><td>1,000.00</td></tr> <tr> <td>002</td><td>2</td><td>Trombones</td><td>@ 200.00</td><td>400.00</td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td></td><td></td><td></td><td></td><td></td></tr> <tr> <td align="right" colspan="2">Total:</td><td align="right">1,400.00</td><td></td><td></td></tr> </tbody> </table>	Item #	Quantity	Part	Price	Amount	001	1	Piano	@ 1,000.00	1,000.00	002	2	Trombones	@ 200.00	400.00																Total:		1,400.00																																						
Item #	Quantity	Part	Price	Amount																																																																			
001	1	Piano	@ 1,000.00	1,000.00																																																																			
002	2	Trombones	@ 200.00	400.00																																																																			
Total:		1,400.00																																																																					

图 6-9 过于谨慎的锁定会干扰人们的工作

事情变得越来越不方便了，因为很多地方都会出现对乐器(货物)的竞争，如图 6-10 所示。

George 添加小提琴 (violin); 必须等待 Amanda(!)

图 6-10 死锁

那3个人都得干等上一段时间。

现在，我们可以开始对模型进行改讲，把下面的业务知识会并进来。

- part 在多个 PO 中使用(高度竞争):



- 对 part 的修改比对 PO 的修改要少；
- 修改 part 的价格不一定会导致已有的 PO 发生变化。这取决于修改价格的时间与 PO 的当前状态。

如图 6-11 所示当我们考虑已经交货的被归档的 PO 对象时，第 3 点就非常显而易见了。当然，这些对象所显示的价格应该是填订单时的价格，而不是当前的价格。

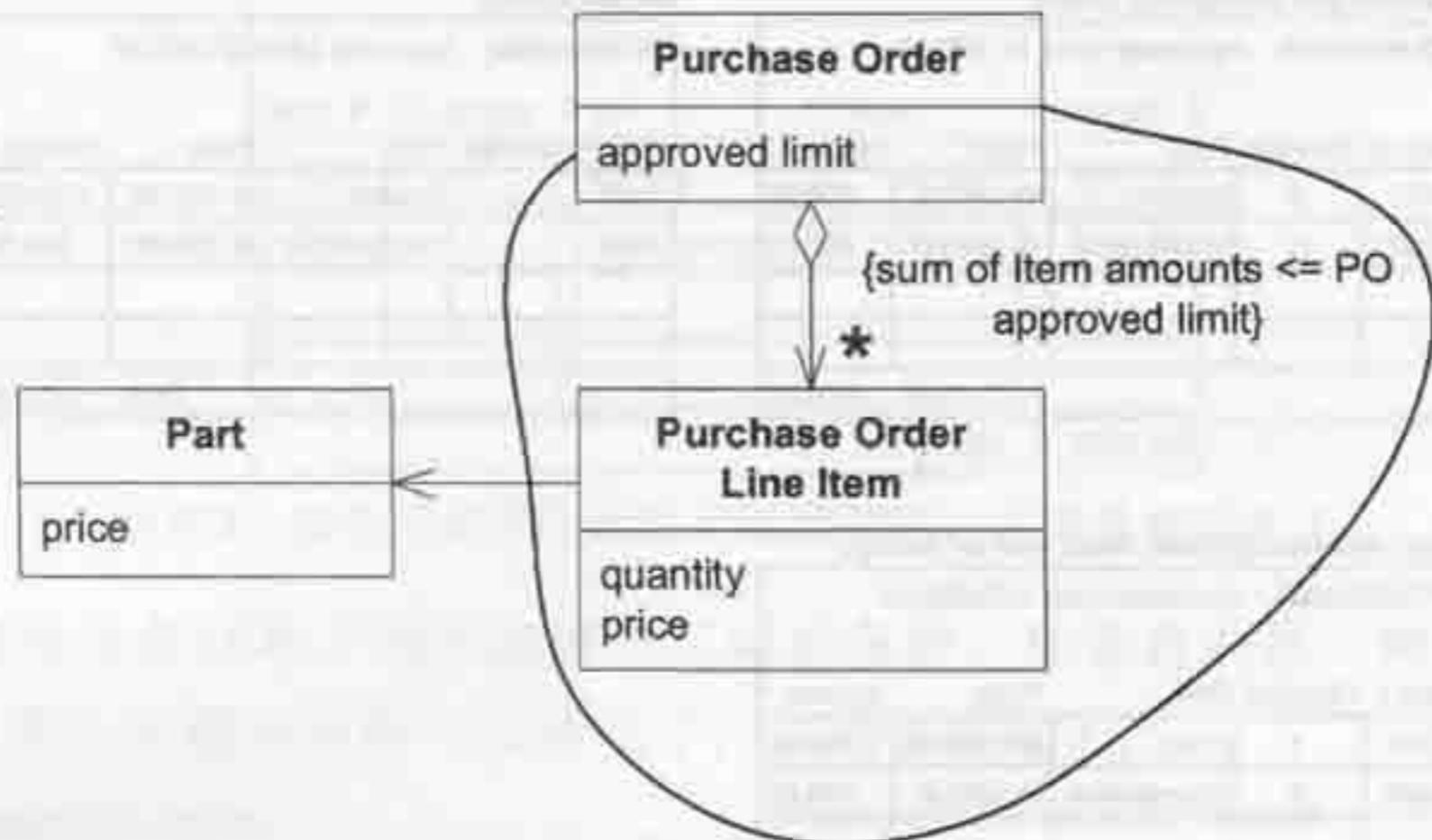


图 6-11 价格(price)被复制到子项。现在聚合的不变量可以得到保证了

按照这个模型来实现，能够保证 PO 与其子项之间的不变量约束，同时修改一个 part 的价格也不会直接影响到引用该 part 的订单子项。更广泛的一致性规则可以用其他的方法来处理。例如，系统可以每天为用户列出价格已经过期了的子项队列，由用户来逐个决定是采用新价还是去掉子项。但是，这不是一个必须随时满足的不变量。通过把子项和 part 的依赖关系变得松散，我们就避免了不必要的竞争，并能将实际业务更好地反映出来。同时，加强 PO 与其子项之间的联系又保证了软件能够遵守一条重要的业务规则。

聚合强制了 PO 及其子项之间的的所有权关系，这是符合业务实际的。PO 及其子项的创建和删除被自然地关联起来，同时 part 的创建和删除则是独立的。

聚合圈定出一个范围，在这个范围内无论处于生命周期的哪个阶段，都应该满足不变量。接下来的工厂和仓储模式用来对聚合进行操作，将特定生命周期变迁的复杂性封装起来。

6.2 工厂

当创建一个对象或整个聚合的逻辑变得非常复杂，或者过多地暴露了内部结构时，



工厂提供了封装。



对象的许多能力来自于其内部复杂的配置以及与其他对象的关联。对象应该是高度精练的，不应包含任何与其含义或者支持交互角色无关的东西。生命周期管理具有复杂的职责，因此如果让一个复杂对象来负责自身的创建工作，就会由于职责过载而产生问题。

轿车引擎是一种非常复杂的机件，由几十种零件协作完成引擎的职责：推动曲柄。我们可以设想试着设计这样一个引擎组，它能够自己抓来一些活塞，把它们插入自己的汽缸，然后找到火花塞的插口，把火花塞给拧进去。但是，看起来这么复杂的一台机器却并不会像我们平时见到的引擎那样可靠和有效。相反，我们认为应该用其他的东西来组装这些零件，也许是一个技师，也许是一台工业机器人。人和机器人实际上都比它们所组装的引擎更加复杂。组装零件的工作与推动曲柄的工作是毫不相干的。只有在生产轿车时才需要组装功能——我们在开车时并不需要一个机器人或者一个技师。因为轿车从来都不会一边组装一边行驶，因此把这些功能塞进同一个机制中是没有价值的。类似地，组装一个复杂的复合对象的工作，与该对象被组装成功后所执行的任何其他工作，最好是分离开来。

但是，如果把组装的职责转移给其他对此感兴趣的对象，如应用中的客户对象，那就会使问题变得更加糟糕。客户知道需要让哪些领域对象来执行必须的计算，从而完成自己的工作。如果我们希望让客户程序来组装其所需的领域对象，那它就必须知道一些有关该对象内部结构的事情。为了保证领域对象中各个部分之间的关系满足所有不变量，



客户程序又必须知道该对象上的某些规则。即使是调用构造函数也会使客户与它正在创建的具体类关联起来。对领域对象的实现所作的所有修改都需要客户作出相应修改，导致重构更加困难。

让客户来负责创建对象的工作，使问题不必要地复杂化了，也模糊了客户程序的职责。这样也破坏了被创建的领域对象和聚合的封装。更糟糕的是，如果客户是应用层的一部分，那就导致这个职责完全从领域层泄漏到了应用层。这种应用与实现细节的紧密关联使我们丧失了领域层抽象的大部分优点，而进一步的修改所需的代价则更加昂贵了。

“创建”可以作为对象自身一个主要的操作，但是，把对象的创建职责与复杂的组装操作组合起来是不恰当的。这样会使得设计非常笨拙而且难以理解。让客户直接构造对象又让客户的设计非常混乱，破坏了对被组装对象或聚合的封装，使得客户和被创建的对象的实现完全关联到了一起。

复杂的对象创建工作是一种领域层的职责，然而创建工作并不属于表达模型的对象。在有些情况下，对象的创建和组装对应着领域中的一个重要的里程碑，例如“开一个银行账户”。但是，对象创建和组装在领域中通常没有含义；它们只是实现上的需要。为了解决这个问题，我们必须在领域设计中加入一个构造，它既不是实体，也不是值对象或服务。这和前面的章节有些相悖，但弄清楚这一点非常重要：我们正在向设计中加入的元素不对应于模型中的任何事物，但是它们确实要完成领域层的部分职责。

每种面向对象语言都提供了一种创建对象的机制(例如，在 Java 和 C++ 中是构造函数，在 Smalltalk 中是实例创建类方法)，但是我们还是需要一种更加抽象的，与其他对象解耦的构造。担负了创建其他对象的职责的程序元素被称为工厂，如图 6-12 所示。

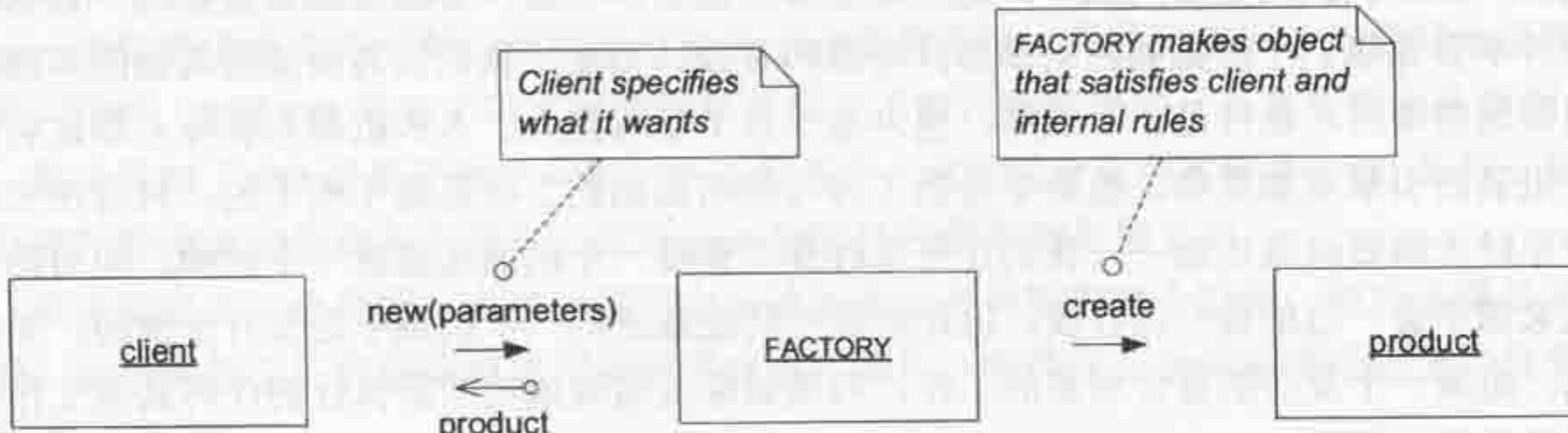


图 6-12 和工厂的基本交互

和对象接口应该封装其实现一样，工厂允许客户使用其行为而无需知道它是如何工作的，从而将创建一个复杂对象或聚合所需的知识封装起来。它提供了一个反映客户目的的接口，以及一个被创建对象的抽象视图。



因此：

将创建复杂对象或聚合的实例的职责分离到一个单独的对象中来，这个对象本身可能在领域模型中没有职责，但仍然是领域设计的一部分。它提供了一个将所有复杂的组装封装起来的接口，这样客户就无需引用它要实例化的对象的具体类了。用工厂把聚合作为一个整体创建出来，并保证其不变量得到满足。

设计工厂的方法有很多种。Gamma et al. 1995 中透彻地研究了一些针对特定目的的创建模式——工厂方法(Factory Method)、抽象工厂(Abstract Factory)和构建器(Builder)。该书主要探讨了一些用来解决最困难的对象构造问题的模式。我们这里不是要深入研究工厂的设计，而是要说明工厂作为一种重要的组件在领域设计中所占的位置。适当地使用工厂有助于使模型驱动设计不会出轨。

任何好的工厂都有两个基本的要求：

- 每个创建方法都是原子的，并保证所创建的对象或聚合能满足所有不变量。工厂所构造出来的对象在状态上必须是一致的。对于实体来说，这意味着它创建的是一个完整的、满足所有的不变量的聚合，但有一些可选的元素可能还有待加入。对于具有不变性的值对象来说，这意味着其所有属性都已经初始化为正确的最终状态。有时工厂可能无法根据外部请求把对象正确地构造出来，如果工厂的接口允许这种请求，那么它应该抛出一个异常，或者调用其他机制来确保不会返回错误的对象。
- 工厂应该将构造结果抽象到所需的类型，而不是它所创建的具体类的类型。Gamma et al. 1995 中精巧的工厂模式会对此有所帮助。

6.2.1 工厂及其应用场所的选择

一般而言，我们创建工厂的目的是为了把待创建对象的细节隐藏起来，并在我们希望进行控制的地方使用工厂。这些决定通常是围绕聚合来考虑的。

例如，如果您需要向一个已有的聚合中加入元素，那么可以在聚合根中创建一个工厂方法。这可以把聚合的内部实现向所有外部客户隐藏起来，同时由根负责保证聚合在加入元素之后的完整性，如图 6-13 所示。

另一个在对象中实现工厂方法的例子是，这个对象与另一个对象的生成密切相关，但是却并不拥有它生成的对象。当一个对象所提供的数据或规则在另一个对象的创建过程中起到支配作用的时候，我们用它的工厂方法来创建那个对象会很方便，因为如果在其他地方创建的话，我们还得把这个对象的内部信息都提取出来。此外，这种做法还能表达出生产者对象与其产品之间的特殊联系。

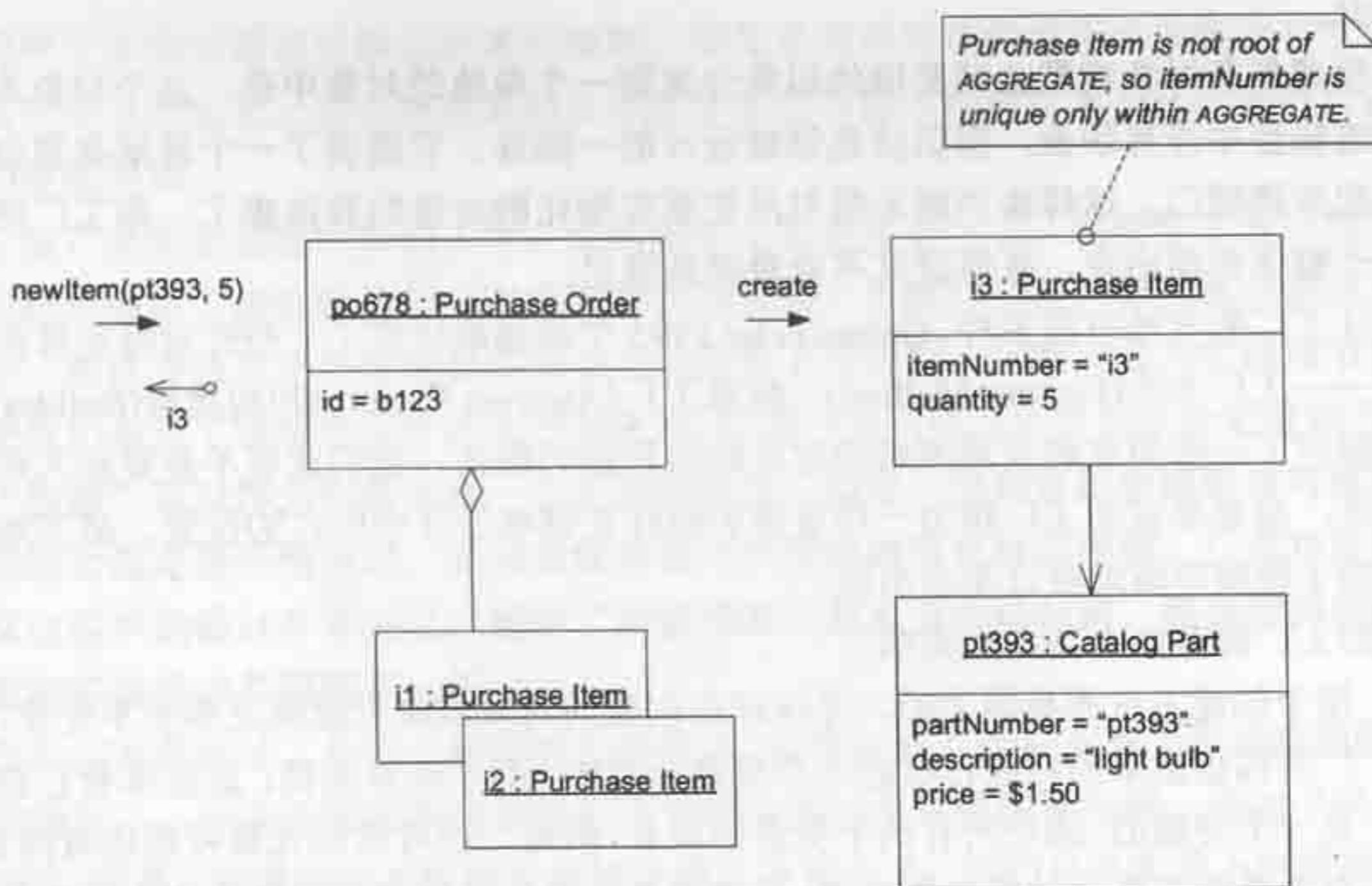


图 6-13 一个工厂方法封装了聚合的扩充逻辑

在图 6-14 中, Trade Order 不属于 Brokerage Account 所在的聚合的一部分, 因为它在生成后将继续与交易执行系统打交道, 把它放在 Brokerage Account 中反而碍事。虽然如此, 让 Brokerage Account 来控制 Trade Order 的生成看起来还是很自然的。Brokerage Account 包含了 Trade Order 中必须嵌入的信息(从其自身的标识开始), 还包含了一些规则来控制哪些交易是允许的。还有一个好处是我们可以把 Trade Order 的实现隐藏起来。例如, Trade Order 可能会被重构为一个层次结构, 包括 Buy Order 和 Sell Order 等不同的子类。工厂解除了客户与具体类的关联。

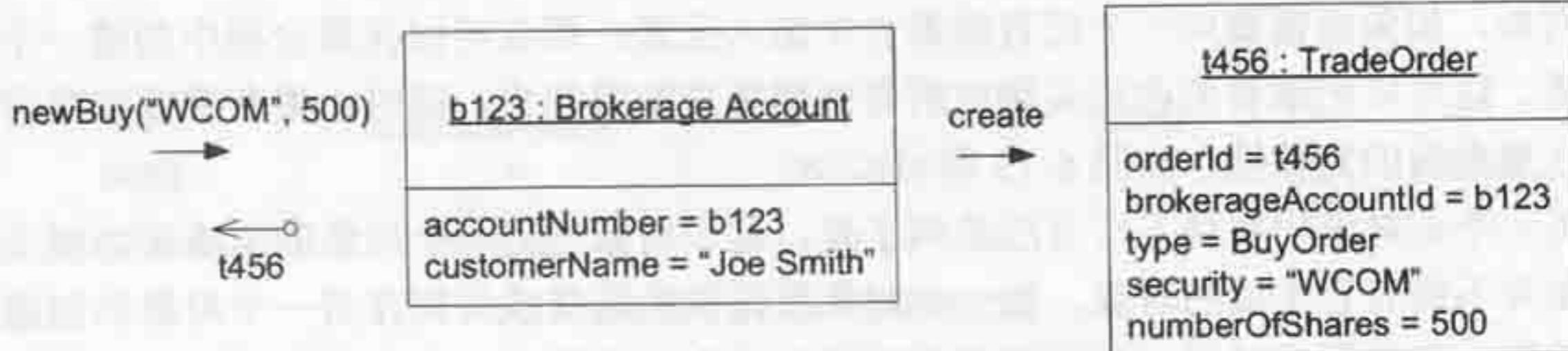


图 6-14 工厂方法生成的实体不是其聚合的一部分

工厂与其产品之间的关联是非常紧密的, 因此它所关联的对象应该与产品具有自然

的紧密联系。每当我们希望隐藏一些信息(具体的实现细节或者是复杂的构造逻辑),但是又似乎没有一个自然的地方可以容纳它们时,我们就应该创建一个专门的工厂对象或者服务。如图 6-15 所示独立的工厂通常用来生成整个聚合,返回其根的引用,并保证生成的聚合满足所有不变量。如果聚合内部的对象需要一个工厂,而聚合根又不适合充当这个角色的话,那么就可以创建一个独立的工厂。但是,聚合必须仍然遵守访问限制规则,确保聚合之外的对象只能获得产品的临时引用。

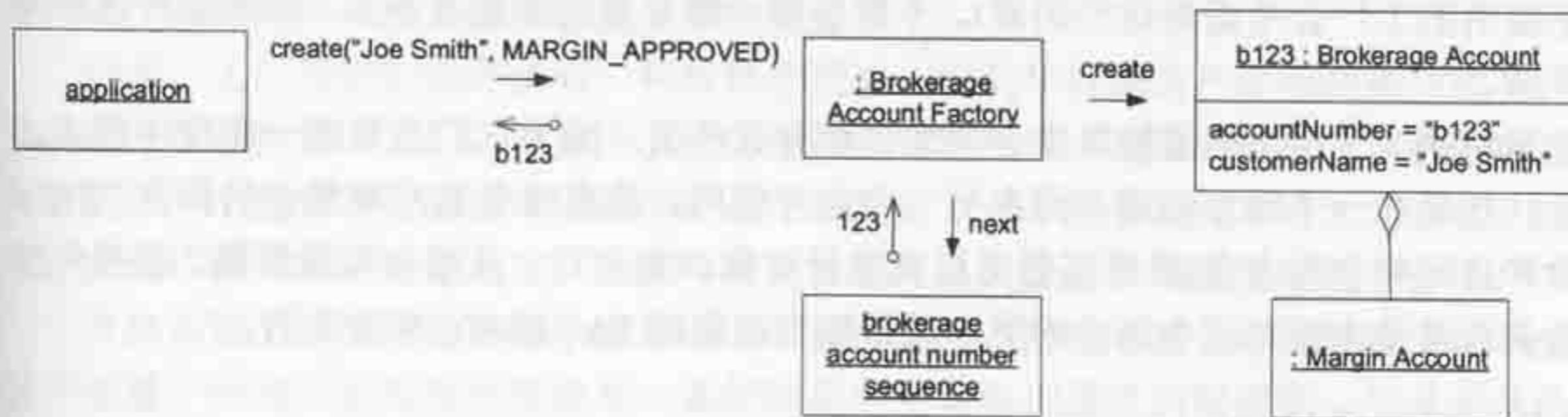


图 6-15 用独立的工厂来构造聚合

6.2.2 只需构造函数的情况

有些程序全部通过直接调用类构造函数来创建对象实例(或者使用编程语言提供的最低级的实例创建机制)。这种代码我见得太多了。工厂的引入可以为我们带来极大的好处,可惜它经常被忽视了。然而,有的时候直接调用构造函数是最好的选择。对于没有使用多态的简单对象,工厂只会把问题弄复杂。

在下面的情况下,我们倾向于使用原始的公有构造函数:

- 类(class)就是类型(type)。它不是任何层次结构的一部分,也没有通过实现接口来使用多态;
- 客户关心具体的实现,可能作为选择策略的一种方法;
- 客户可以使用对象的所有属性,因此提供给客户的构造函数中没有嵌入对象创建逻辑;
- 构造函数并不复杂;
- 公有构造函数必须遵循和工厂相同的规则:它必须是一个原子操作,并保证创建出来的对象满足所有不变量。

避免在构造函数中调用其他类的构造函数。构造函数应该尽量简单。复杂的组合对象(特别是聚合)最好使用工厂。选择使用小小工厂方法的阈值并不高。

Java 类库中提供了一些有意思的例子。所有的集合都通过接口解除了客户与其具体



实现的关联。然而，它们都是通过直接调用构造函数来创建的。我们本来可以用一个工厂把集合的层次结构封装起来。让客户告诉工厂的方法它需要哪些特性，然后由工厂选择适当的具体类来实例化。这样创建集合的代码会表达得更加清楚，而且我们可以加入新的集合类而不会破坏已有的 Java 程序。

但是在这个例子中，我们赞成使用具体的构造函数来创建对象。首先，在许多应用中，实现选择对性能的影响是很大的，因此我们希望能够控制使用哪种实现(当然，一个真正聪明的工厂会考虑到这些因素)。不管怎样，集合类的数量并不多，因此这种选择并不复杂。

虽然缺少工厂，但是抽象集合类型仍然有其价值，因为它们具有统一的使用模式。集合往往是在一个地方创建，而在另一个地方使用。这意味着客户对集合的操作(增加、删除和访问集合的内容)最终还是可以直接针对接口来进行，从而与实现解耦。选择何种集合类通常是由拥有这个集合的类，或是拥有对象的 FACTORY 来决定的。

6.2.3 接口的设计

当设计工厂的方法签名时，无论工厂是独立的工厂还是工厂方法，都要记住以下两点：

- 每个操作都必须是原子的。您必须把所需的所有信息传递给工厂，使之只需一次交互就能创建出完整的产品来。您还必须决定当创建失败时如何处理，即出现某些不变量无法满足时。您可以抛出一个异常，或者直接返回一个 null。为了保持一致，您可以把工厂失败的处理方式写入编码规范。
- 工厂将与其变元的类型产生关联。在选择输入参数时，我们可能会不小心产生隐含的依赖关系。关联的程度取决于我们是怎样处理变元的。如果变元可以简单地插入到产品之中，那么依赖关系还不是很大。如果我们要从参数中挑出一些部分用来构造对象的话，那么关联就会变得更紧。

最安全的参数是那些来自更低设计层中的参数。即使是在同一层中，也会存在自然的分层倾向，有一些更加基础的对象被更高层次的对象使用(这种分层将在第 10 章“柔性设计”中从另一个角度来讨论。第 16 章“大比例结构”还会再次讨论到这个问题)。

另一个不错的参数选择是在模型中与产品密切相关的对象，这样就不会增加新的依赖关系了。在前面的 Purchase Order Item 的例子中，工厂方法将 Catalog Part 作为一个变元，而 Catalog Part 是 Item 的一个本质关联。这使得 Purchase Order 和 Part 之间增加了一个直接依赖。但是，这 3 个对象构成了一个紧密的概念组。不管怎样，Purchase Order 的聚合已经引用了 Part。因此，让聚合根来控制和封装其内部结构是一种不错的权衡。



使用变元的抽象类型，而不是其具体类型。工厂与产品的具体类已经关联在一起了，因此无需把它与具体参数也关联起来。

6.2.4 如何放置不变量的逻辑

工厂负责保证它创建的对象或聚合能够满足所有不变量。然而，如果要把应用于一个对象的规则提到对象之外，我们还是应该先仔细思考一番。工厂可以把不变量的检查委托给产品来执行，而且这往往是最好的选择。

但是，工厂与它们的产品有一种特殊的联系。它们早就知道产品的内部结构了，而且我们之所以要加入工厂，就是为了让它把产品实现出来。在某些情况下，把不变量逻辑放到工厂中是有好处的，可以减少产品中的混乱。对于聚合规则来说尤其如此，因为它涉及到许多对象。但是把工厂方法与其他领域对象关联起来则显得特别不合适。

虽然原理上不变量是在每个操作完成时被检查的，但是对象允许的转换往往不会违反不变量。例如，在实体中可能有一条规则是约束其标识属性的赋值的，但是实体的标识属性在创建之后就不会再发生变化了。值对象则完全是不变的。由于这种规则在对象的活跃生命周期中从来不会被用到，因此对象可以不需要携带这些逻辑。在这样的情况下，把不变量存放在工厂中就非常合理了，也能使得产品更加简单。

6.2.5 实体工厂与值对象工厂

实体工厂和值对象工厂有两个不同之处。值对象具有不变性，产品一旦生成就是最终状态，因此其工厂操作必须考虑到产品的完整描述。实体工厂则倾向于仅仅要求构造一个有效聚合所需的本质属性，而那些不变量不要求的细节可以在以后再追加进来。

接下来我们考虑一下，在为实体指定标识时会涉及到哪些问题(这显然与值对象无关)。我们在第5章指出，标识既可以是由程序自动指定的，也可以由外部(通常是用户)提供的。如果一个顾客的标识要根据电话号码来进行跟踪，那么那个电话号码就必须显式地作为变元传递给工厂。如果标识是由程序来指定的，那么工厂就是一个控制标识的好地方。在实际应用中，虽然惟一的流水ID号实际上是利用数据库的“序列”或者其他基础结构提供的机制来产生的，不过只有工厂才知道需要什么标识，以及把标识放到哪里。

6.2.6 存储对象的重建

到现在为止，我们还只是讨论了工厂在对象生命周期的最开始部分所起的作用。从



某种程度上，许多对象都要保存到数据库中，或者通过网络进行传输，而目前的数据库技术一般都不能按对象的内容把它们存储起来。许多传输方法是将对象“碾平”，通过非常有限的一些类型来描述它们。因此，我们可能会需要一个比较复杂的过程才能把各个部分重新组装为一个活的对象。

我们可以用工厂来重建对象。这个过程与用工厂创建对象的过程非常相似，只有两点主要的区别：

- 用来重建对象的实体工厂需为实体指定一个新的流水 ID 号，因为指定新 ID 将会导致对象失去与以前存储的实例的连续性(即变成了另一个实体)。因此，用工厂来重建存储对象时，标识属性必须成为输入参数之一。
- 工厂在重建对象时，对违反不变量的处理可能会发生变化。如果在创建新对象时发生不变量无法满足的情况，工厂只需简单地中止就行了，但是在重建时可能会需要更加灵活的响应。如果有一个对象已经存在于系统(例如数据库)之中，那么我们不能忽略这个事实，也不能听任规则被它违反。我们必须提供某种策略来修复这种不一致的状况。这使得重建对象比创建新对象更加困难。

图 6-16 和 6-17 演示了两种重建的方法。对于从数据库中重建对象的情况，对象映射技术可以为我们提供一些或者全部的服务，这是非常方便的。无论何时，如果我们需要进行复杂的处理来从另一种媒介中将对象重建出来，那么工厂就是一种很好的选择。

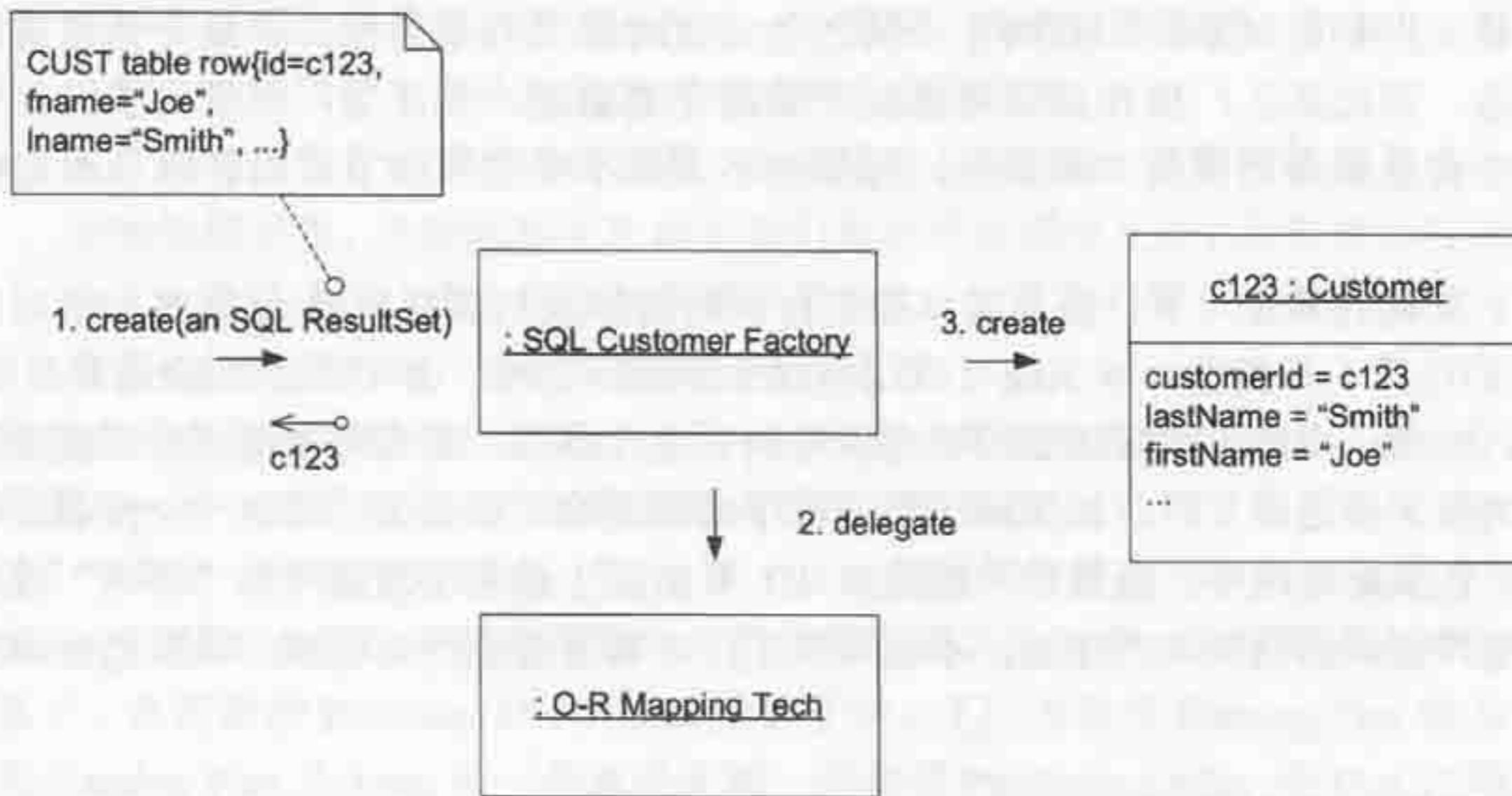


图 6-16 从关系数据库中查询并重建一个实体

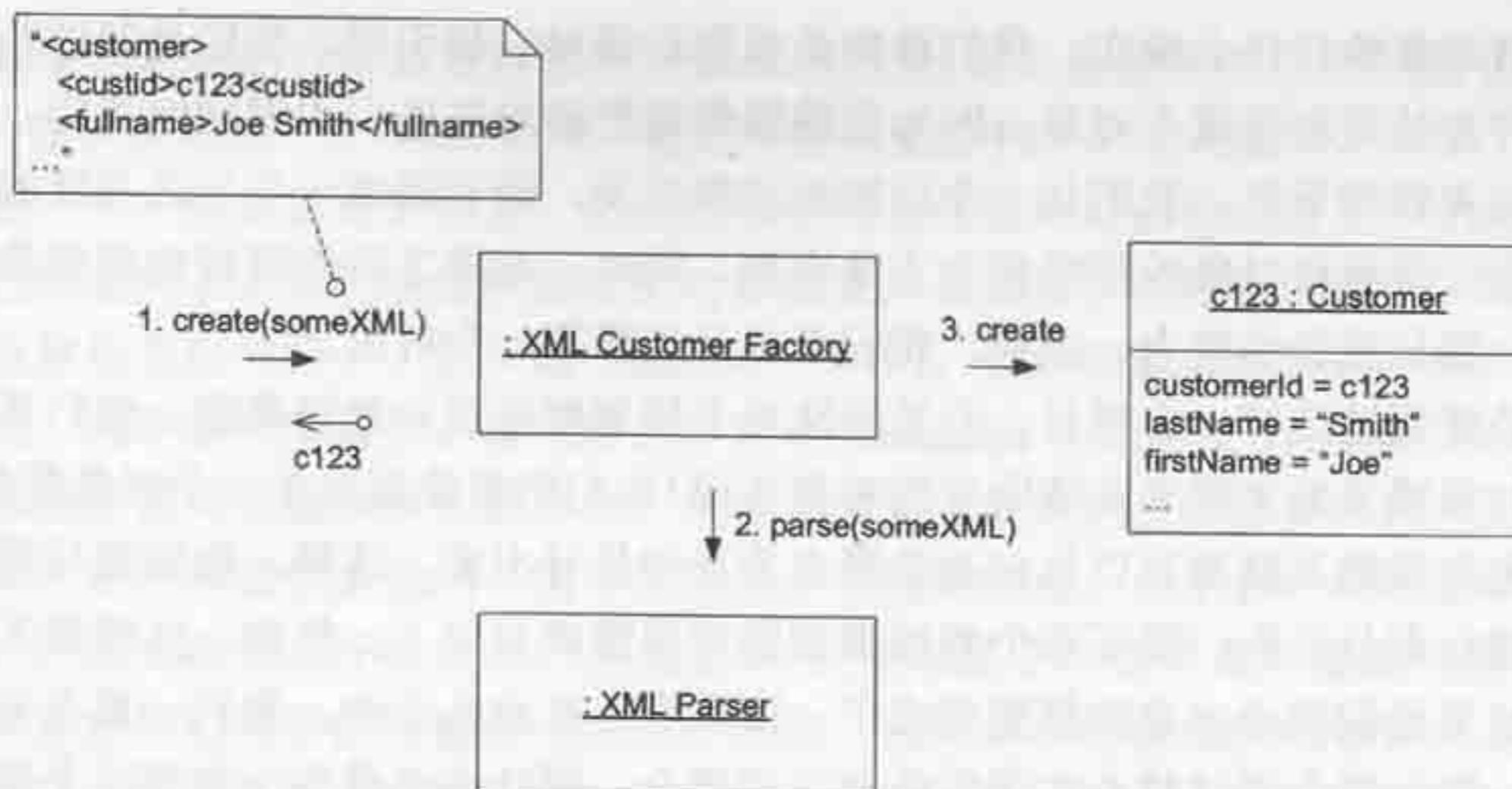


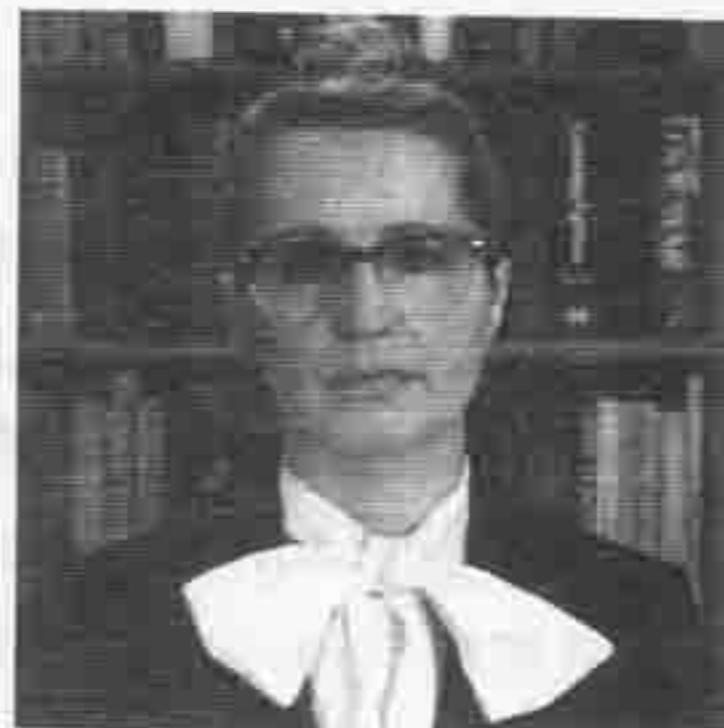
图 6-17 重建作为 XML 传输的实体

总而言之，我们必须把实例创建的访问点标识出来，并显式地定义它们的作用域。这些访问点可以就是构造函数，但我们往往需要一种更加抽象或精巧的实例创建机制。这就需要在设计中引入一种新的构造：工厂。工厂通常不表达模型中的任何部分，然而它们确实是领域设计的一部分，并有助于使模型中表达的对象更加清晰。

工厂封装了对象在创建和重建时的生命周期变迁。还有另一种变迁，它在技术上的复杂性也会导致领域设计陷入混乱，那就是对象和存储之间的双向转换。这种转换是另一种领域设计构造——仓储的职责。

6.3 仓储

我们通过关联就可以从一个对象找到另一个与之相关的对象。但是，如果我们要引用的实体或者值对象已经处于生命周期之中了，那么就必须首先获得一个访问的起点。





无论对对象执行什么操作，我们首先必须要获得对它的引用。我们是怎样获得引用的呢？一种方法是创建这个对象，因为创建操作将把新对象的引用返回给我们。另一种方法是通过关联的导航。我们从一个已知的对象出发，向它请求一个与之关联的对象。这种方法在任何面向对象的程序都会大量用到。同时，对象之间的链接也使对象模型获得了相当一部分的表达能力。但是，我们必须得到那第一个对象。

我曾经碰到过这样一个项目，开发团队对于模型驱动设计热情高涨，他们甚至试图通过创建对象或导航关联来实现所有的对象访问！他们把对象保存在一个对象数据库中，并认为所有必要的关联都可以从已有的概念关系中推导出来。这样，他们就只需对这些概念关系进行充分分析，保证整个领域模型的内聚性就可以了。然而，这样做无异于作茧自缚，结果他们构造出来的模型变成了一团乱麻。在前几章中，我们一直在努力避免这种情况，例如努力通过精心实现实体和应用聚合。团队成员最终没有把这个策略坚持多久，但是他们也一直没有改用其他一致的方法。他们修修补补地提出了很多临时方案，也不再像开始那样雄心勃勃的了。

很少会有人想到这种方法，更不用说试验它了，因为他们的大部分对象都存储在关系数据库中。这种存储技术会使我们很自然地用到第3种获得对象引用的方法：执行数据库查询，根据对象的属性把它查找出来，或者先查找出对象的组成部分，然后把它重建出来。

数据库查询是全局可访问的，因此我们可以直接得到想要的对象。对象网是通过对象之间的联系来管理的，但是我们并不需要把所有对象都相互连接起来。是提供导航还是依赖于查询，只是一个设计上的决定，取决于我们在查询的解耦与关联的内聚之间如何权衡。一个 Customer 对象是应该持有他下的所有 Order 对象的集合，还是应该通过对 Customer ID 字段进行查询来把 Order 从数据库中筛选出来？把查询和关联正确地结合起来，可以使我们的设计更易于理解。

遗憾的是，开发人员通常不会去多想这些设计中的细微问题，因为他们满脑子都是实现机制，他们需要用这些机制来完成一项技巧性的任务——存储一个对象，把它重建出来，最终把它从存储系统中删除出去。

从技术的视角来看，获取一个已存储的对象实际上是一种创建对象的过程，因为我们从数据库中得到的数据是用来组装新对象的。确实，我们经常要编写这个过程的实现代码，因此很难忘记“重建就是一种创建”这个事实。但是，从概念上说，这个过程发生在实体对象的生命周期的中间时段。一个 Customer 对象存储到数据库然后再取出来，并不能表示是一个新的顾客。为了记住这种区别，我把从已存储数据创建实例的过程称为“重建(reconstitution)”。



领域驱动设计的目标是通过聚焦于领域的模型(而不是技术)来生产更好的软件。但是,当开发人员构造一个SQL查询,把它传给基础结构层的查询服务,得到一个表记录的结果集,从中取出必需的信息,然后把那些信息传给构造函数或者工厂的时候,模型的焦点已经丢失了。我们很自然就会把对象想象为查询提供的数据的容器,而整个设计则偏向数据处理的风格了。技术细节虽然各不相同,但是问题依然存在:客户正在和技术打交道,而不是模型概念。像元数据映射层(Metadata Mapping Layers, Fowler 2002)这样的基础结构可以提供极大的帮助,使我们能更方便地把查询结果转换为对象,但是开发人员还是在考虑技术机制,而不是领域。更糟糕的是,由于客户代码能够直接使用数据库,开发人员会忍不住绕过一些模型特性(例如聚合,甚至是对象的封装),取而代之的是直接把他们需要的数据取出来进行操作。越来越多的领域规则被嵌入到查询代码中,或者直接就不见了。对象数据库确实能够消除转换问题,但是其查询机制通常还是一种技术机制,开发人员仍然对如何从数据库抓取他们想要的对象感兴趣。

客户需要一种可行的方法来获得已有的领域对象的引用。如果基础结构允许客户的开发人员很容易地加入关联,那么他们就会加入更多的导航关联,把模型弄得一团糟。另一种可能是,开发人员会使用查询提取他们需要的额外数据,或者提取一些特殊的对象,而它们本来是应该通过聚合根来访问。领域逻辑跑到查询代码和客户代码中去了,而实体和值对象变成了纯粹的数据容器。大部分数据库访问基础结构的技术复杂性,很快使得客户代码陷入混乱,最终开发人员只好抛开领域层,把模型变成了一个摆设。

利用我们前面讨论过的[设计原则](#),我们可以在某种程度上缩小对象访问问题的范围。假设我们找到了一种访问方法,保证模型的焦点足够明确来应用上述原则:首先,我们可以不用关注临时对象。临时对象(通常是值对象)的生命周期很短,它们在客户操作中被创建出来,用完之后就被丢弃了。我们也不需要为那些可以通过导航更方便地访问到的持久对象提供查询访问。例如,一个人的地址可以通过 Person 这个对象来获取。还有最重要的一点,即聚合的任何内部对象都只能通过根来导航,其他的访问途径都是禁止的。

持久化的值对象通常可以从某些实体出发来进行访问,这个实体充当着封装那些值对象的聚合根。实际上,为一个值提供全局的查询访问通常是毫无意义的,因为根据属性来查找一个值就相当于按照那些属性来创建一个新的实例。当然,也有例外。例如,当我在线计划旅程时,有时会把一些可行的路线先保存起来,稍后再回过头来选择一种路线进行预订。那些路线都是值(如果有两个路线所包含的班机完全相同,我们并不会介意哪个是哪个),但是它们已经跟我的用户名关联起来了,可以根据用户名来进行查询。另一种情况是“枚举”,枚举类型只有一批严格限制的、预先确定的可能值。但是,对值



对象进行全局访问的情况远比实体要少见。如果您发现自己需要查询数据库来获得一个已有的值，那么不妨考虑一下，您也许实际上是得到了一个实体，只不过还没有辨认出它的标识而已。

从上面的讨论中可以明显看出，大部分对象都不应该通过全局查询来访问。如果设计能把那些确实需要全局查询的对象表现出来，那就太好了。

现在我们可以更精确地把问题重申如下：

部分持久对象必须通过按对象属性进行查询的方式来实现全局访问。对不便于通过导航来访问的聚合根来说，这种访问方式是必需的。这些对象通常是实体，有时是包含复杂内部结构的值对象，有时是枚举值。为其他对象提供这种访问会使一些重要的区别变得模糊。不受限制的数据库查询实际上会破坏领域对象和聚合的封装。把技术基础结构和数据库访问机制暴露出来，会使客户变得复杂，同时掩盖了模型驱动设计。

有很多技术可以用来解决数据库访问的技术挑战，其中包括把 SQL 封装为查询对象 (QUERY OBJECTS)，或者通过元数据映射层(Fowler 2002)实现对象和表之间的转换。工厂也有助于存储对象的重建(本章稍后会讨论到)。这些技术和许多其他的技术都可以用来限制问题的复杂性。

但即便有这些技术，我们还是要注意失去了些什么。我们不再考虑领域模型中的概念了。我们的代码也不再描述与业务有关的事情；它只是在使用数据检索技术。仓储模式是一个简单的概念框架，用来把上面那些解决方案封装起来，并找回模型的焦点。

一个仓储将某种类型的所有对象描述为一个概念性的集合(通常是模拟的集合)。它的行为与集合类似，但是包含更精细的查询能力。仓储可以加入和删除具有合适类型的对象，并通过仓储背后的机制将它们插入数据库或从数据库中删除。从这个定义可以推断出，仓储具有一系列紧密相关的职责，为我们提供了对聚合根从产生之初直到其生命周期结束期间的访问能力。

仓储的查询方法根据客户指定的标准 (通常是某些属性的值)来筛选对象，然后把结果返回给请求的客户程序。如图 6-18 所示，仓储负责取出被请求的对象，并把数据库查询和元数据映射的机制封装起来。仓储可以根据客户要求的任何标准实现各种不同的选择对象的查询，进行对象筛选。它们还可以返回统计信息，例如统计总共有多少个符合标准的实例。它们甚至可以进行统计计算，例如计算出所有匹配对象的某个数值属性的累加和。

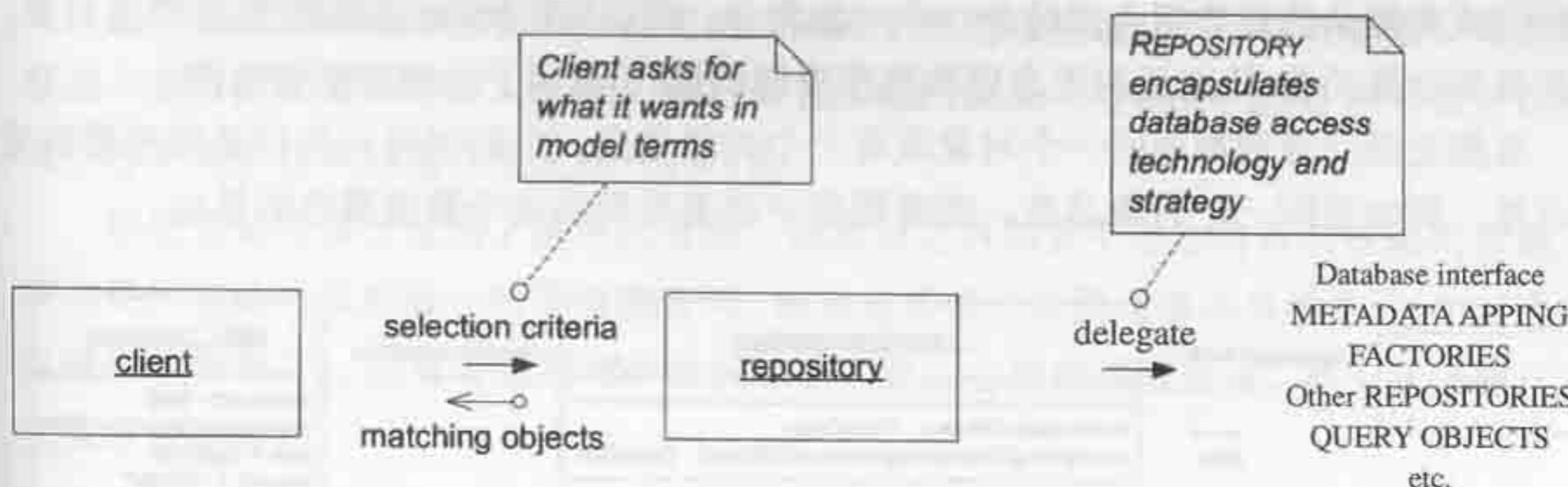


图 6-18 仓储为客户执行查询

仓储为客户免除了一个艰巨的任务，使之能够使用简单的表明意图的接口，并按照模型请求所需的对象。为此我们需要许多复杂的技术基础结构，但是接口非常简单，而且在概念上与领域模型联系起来了。

因此：

为每种需要全局访问的对象类型创建一个对象，该对象为该类型所有对象在内存中的集合提供影像。用一个众所周知的全局接口来设立访问入口。提供增删对象的方法，把对数据存储的实际的插入和删除封装起来。提供根据某种标准筛选对象的方法，返回完整实例化了的属性值符合标准的对象或对象集合，把实际的存储和查询技术封装起来。仅为确实需要直接访问的聚合根提供仓储。让客户聚焦于模型，把所有对象存储和访问的工作委托给仓储来完成。

仓储具有许多优点，包括：

- 它们为客户提供了一个简单的模型，来获取持久对象并管理其生命周期；
- 它们把应用和领域设计从持久技术、多种数据库策略或甚至多种数据来源解耦出来；
- 它们传达了对象访问的设计决策；
- 它们可以很容易被替换为哑实现，以便在测试中使用(通常使用一个内存中的集合)。

6.3.1 查询仓储

所有仓储都要提供方法，来允许客户请求符合某些标准的对象。但是，对如何设计这个接口的选择范围是很大的。

最容易实现的仓储是把具有特定参数的查询硬编码出来，如图 6-19 所示。这些查询可以是各种各样的：根据标识来检索一个实体(几乎所有仓储都会提供)；根据一个特定



属性值或者复杂的参数组合来请求一个对象集合；根据值区间(如日期范围)来筛选对象；甚至有些计算功能(特别是利用底层数据库提供的操作)也属于仓储的职责范围。

虽然大部分查询都返回一个对象或者一个对象集合，它们也可以执行某些类型的统计运算，例如返回一个对象总数，或者模型中要求得到的某个数值属性的累加。

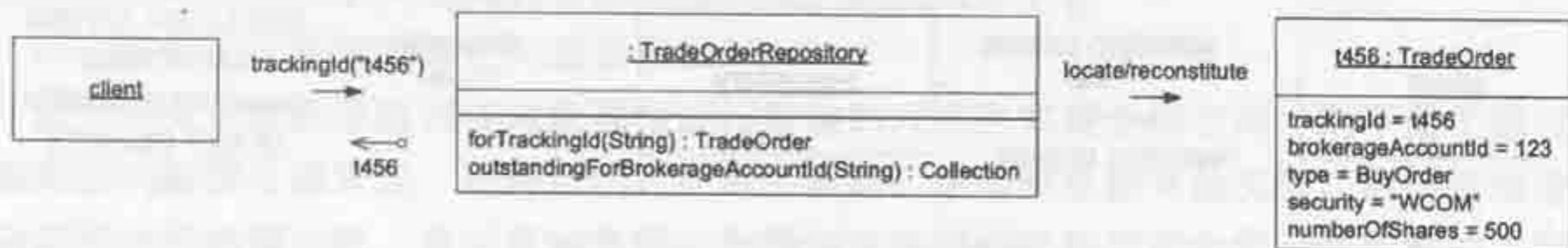


图 6-19 一个简单仓储中的硬编码查询

硬编码查询可以构建于任何基础结构之上，而且也不需要额外的投入，因为如果它们不做那些事情的话，总会有某些客户会设法去做到的。

如果项目中包含大量的查询，那么我们可以构造一个仓储框架，来提供更加灵活的查询。这就需要项目中有人对所需的技术非常熟悉。支持性的基础结构也可以为实现这样的框架提供极大的帮助。

有一种方法特别适合于通过框架来实现通用的仓储，那就是使用基于 Specification(规格)的查询。Specification 允许客户把它所希望的结果描述(更准确地说是指定)出来，而无需考虑如何获得这个结果。在处理过程中，我们创建一个对象来最终执行筛选工作，如图 6-20 所示。这个模式将在第 9 章作深入讨论。

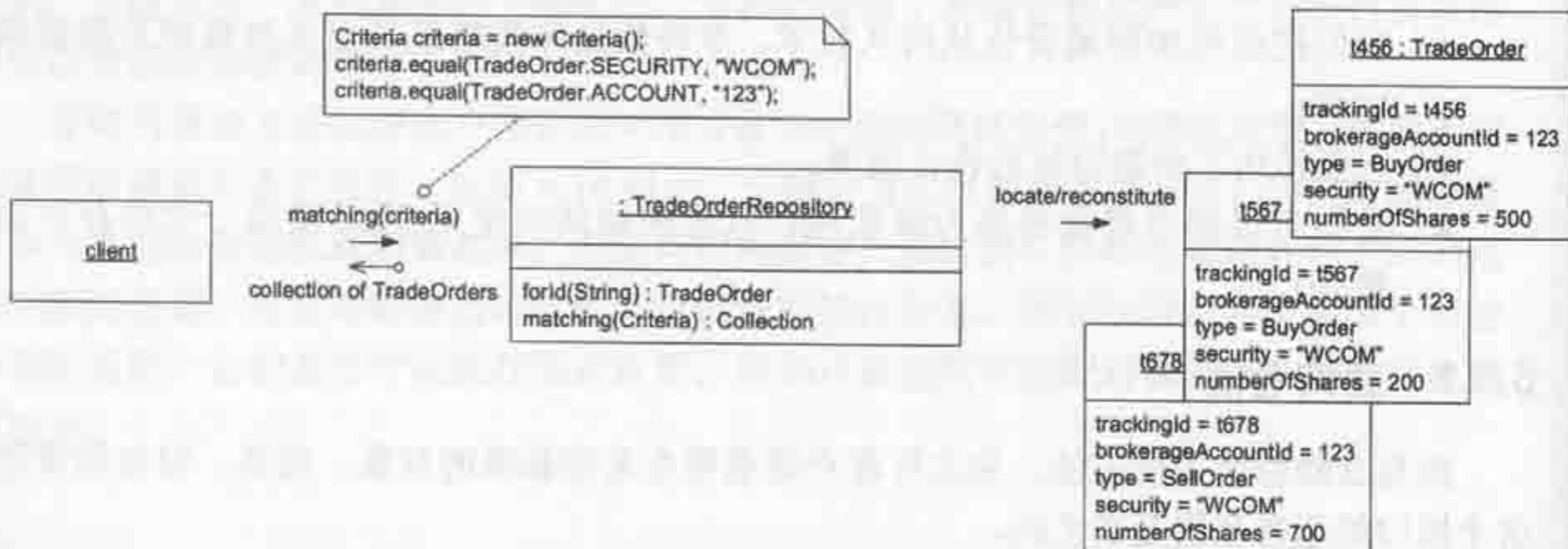


图 6-20 在一个成熟的仓储中，查询标准是用一个灵活的、声明性的规格来指定的



基于规范的查询非常优雅而且灵活。依赖于所用的基础结构，它可能是一个小型的框架，也可能会复杂得惊人。Rob Mee 和 Edward Hieatt 在 Fowler 2002 中对设计这种仓储时涉及到的技术问题作了更深入的讨论。

即使仓储的设计使用了灵活的查询，也应该允许向其中追加特殊的硬编码查询。这些方便的方法可以封装一些常用的查询，或者是那些不返回对象本身的查询(如对选定对象进行数学统计)。没有考虑到例外情况的框架要么会扭曲领域设计，要么就会被开发人员故意绕过去。

6.3.2 了解仓储实现的必要性

封装持久技术使得客户可以非常简单，与仓储的实现完全解耦。但是，和其他封装一样，开发人员必须理解幕后发生了什么事情。对于仓储来说，不同的使用方法或不同的工作方法可能会对性能造成极大的影响。

Kyle Brown 告诉过我一个故事。有一次他被请求去查看一个制造系统，该系统是基于 WebSphere 的，当时正值产品发布阶段。那个系统在使用几个小时之后，就会莫名其妙地出现内存溢出的问题。Kyle 把代码全部浏览了一遍之后发现了问题所在：系统有时需要对车间中所有项目的某些信息进行汇总，而开发人员在实现这个功能的时候使用了一个叫做“all objects”的查询，它为各个对象创建实例，然后再从中选出所需的。这样的代码相当于把整个数据库一次性全部调入内存！在测试过程中并没有出现这个问题，那是因为测试的数据量比较小的缘故。

这种问题显然是不应发生的，但还有更多容易被人忽略的细微问题，也会带来同样严重的后果。开发人员需要理解使用封装行为的隐含问题，但这并不意味着他们必须对实现细节非常熟悉。设计良好的组件是可以被刻画出来的(这是第 10 章“柔性设计”中的一个主要主题)。

正如我们在第 5 章中讨论的，底层的技术可能会限制我们的建模选择。例如，关系数据库可能会对对象结构的组合深度施加某种实现上的限制。同样，开发人员之间，包括仓储的使用者及其查询的实现者，也必须建立双向反馈机制。

6.3.3 实现仓储

根据我们所使用的持久技术和基础结构，仓储可以有不同的实现方法。理想的实现能将所有内部工作向客户隐藏起来(但不是向客户的开发人员)，这样无论数据是保存在



对象数据库还是关系数据库中，或者只是简单地驻留在内存中，客户代码都是完全相同的。仓储将委托合适的基础结构来完成自己的工作。把存储、检索和查询封装起来是仓储实现的最基本特性，如图 6-21 所示。

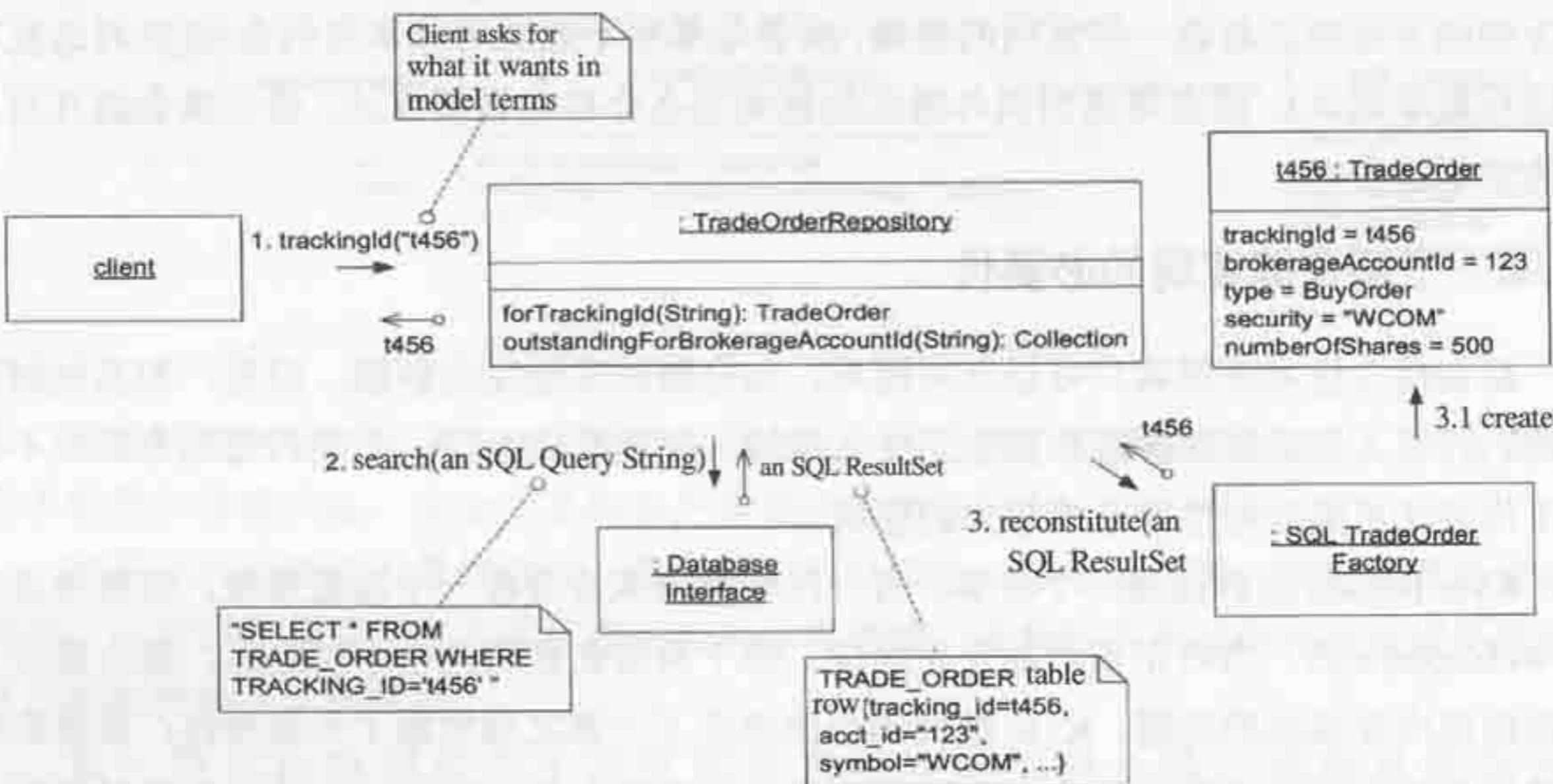


图 6-21 仓储封装了底层的数据存储

仓储的概念可以适用于许多情况。可能的实现方法不胜枚举，这里我只能把一些需要记住的关注点罗列出来。

- **抽象类型。** 仓储“包含”一个特定类型的所有实例，但是这并不意味着我们必须为每个类实现一个仓储。这种类型可以是一个层次结构中的抽象超类(例如一个 TradeOrder 可以是一个 BuyOrder 或者一个 SellOrder)，也可以是一个接口(即使该接口的实现没有形成继承结构也无妨)，或者一个特定的具体类。记住，由于数据库技术中缺乏这种多态机制，我们很可能会因此而遇到一些限制。
- **与客户解耦大有好处。** 我们可以更自由地修改仓储的实现。如果客户直接调用仓储机制，修改起来就会困难多了。我们还可以借此进行性能优化，例如使用不同的查询技术，或者把对象缓存到内存中，或者随时自由地切换持久化策略。我们还可以提供一个易于操纵的、模拟的内存存储策略，来为客户代码和领域对象的测试提供方便。



- 让客户来控制事务。虽然仓储负责数据库的插入和删除，但是它通常不会提交任何东西。例如，虽然保存之后都需要提交事务，但通常都是由客户管理事务上下文，正确地初始化和提交工作单元。如果仓储不插手的话，事务管理就会简单多了。

团队通常会在基础结构层中加入一个框架来支持仓储的实现。仓储的超类除了与低层的基础结构组件进行协作之外，还可能会实现一些基本的查询，特别是需要实现的灵活查询。遗憾的是，在像 Java 这样的类型系统中，这种方法可能会迫使我们用 Object 作为返回对象的类型，而让客户将其强制转换为仓储包含的类型。当然，对于返回 Java 集合的查询来说，客户总是需要执行这种强制转换的。

在 Fowler(2002)中，我们还可以找到一些更多的信息，里面对仓储的实现以及支持其实现的技术模式(如查询对象，Query Object)提供了指导。

6.3.4 在框架内工作

在实现某些像仓储这样的东西之前，需要仔细考虑将要使用的基础结构，特别是所有架构框架。我们可能会发现，利用框架提供的服务可以很容易创建出一个仓储，或者发现框架总在给我们帮倒忙。一些架构框架可能早就提供了与仓储等价的、用来持久化对象的模式，也有可能它们的模式与仓储毫无相似之处。

例如，我们的项目可能构建于 J2EE 之上。找找看框架中有哪些概念与模型驱动设计的模式相近(记住，实体 bean 和实体不是同一个东西)。我们可能会选择把实体 bean 作为聚合根。在 J2EE 的架构框架中，负责为这些对象提供访问入口的构造是“EJB Home”。企图把 EJB Home 包装成一个仓储可能会造成其他的问题。

一般来说，不要与框架对着干。寻找合适的方法来保持领域驱动设计的基本方向。如果框架与设计发生矛盾的话，在一些细节问题上顺其自然。寻找领域驱动设计的概念与框架概念有哪些相近之处。当然，这里的假设是我们除了使用该框架之外别无选择。许多 J2EE 项目根本就没有使用实体 bean。如果可以自由选择的话，我们应该选择那些能够与我们希望使用的设计风格协调一致的框架(或部分框架)。

6.3.5 与工厂的关系

工厂处理的是对象生命周期的开始，而仓储则帮助管理生命周期的中间和结束部分。如果对象都保存在内存中或者对象数据库中的话，实现这一点会很简单。但是，通常总是会有一些对象保存在关系数据库，文件或其他非面向对象的系统中。在这种情况下，取出来的数据就必须通过重建才能成为对象的形式。

由于在这种情况下仓储是根据数据来创建对象的，因此许多人认为仓储就是一种工厂——从技术的角度来看，它确实如此。但是，把对象的重建模型放在显著的位置会更



加有用；而且我们上面提到过，重建一个存储对象与创建一个新的概念对象并不是相同的。在这种领域驱动的设计视角中，工厂和仓储具有完全不同的职责。工厂创建新的对象；而仓储寻找旧的对象。仓储必须让客户觉得那些对象好像就在内存中。那些对象也许不得不重建(是的，必须要创建一个新的实例)，但是它是同一个概念对象，仍然处于它的生命周期之中。

这两个视角可以融合起来，方法是让仓储把对象创建委托给一个工厂，工厂能够从头开始创建对象了(这样的情况理论上存在，但实际很少见)，如图 6-22 所示。

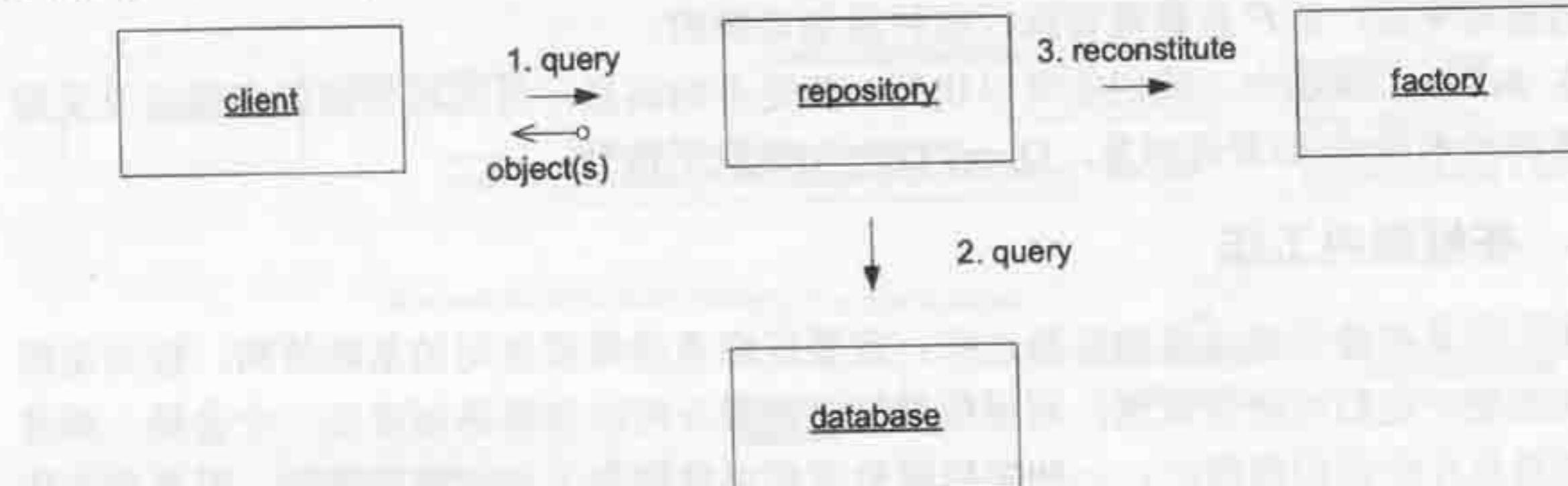


图 6-22 仓储用一个工厂来重建已有的对象

清晰地区分创建和重建有助于把所有与持久有关的职责从工厂中分离出来。工厂的工作是根据数据来实例化一个可能非常复杂的对象。如果产品是一个新的对象，那么客户会知道这一点，并可以把它添加到仓储中去，而把对象保存到数据库中的工作则由仓储来封装，如图 6-23 所示。

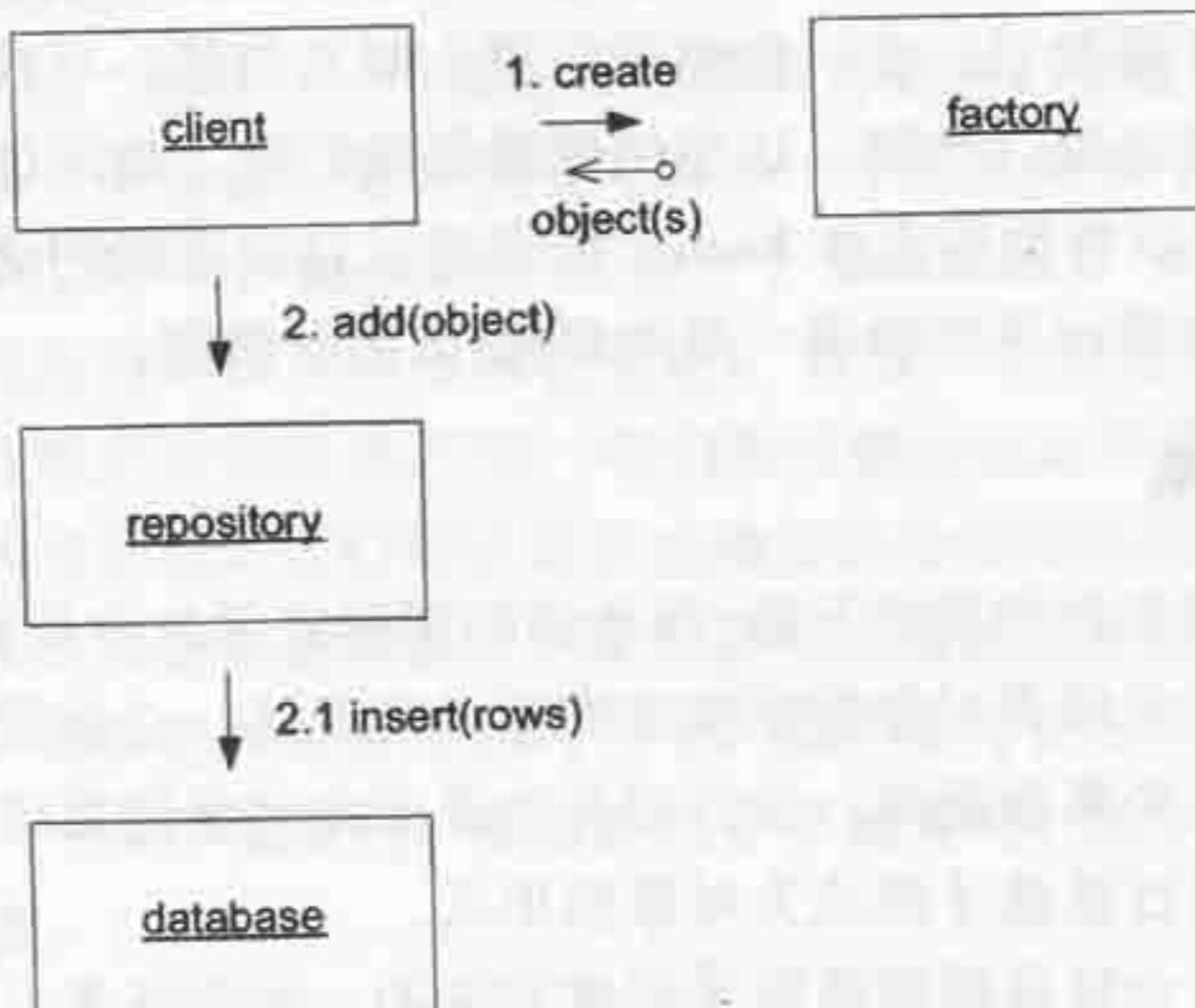


图 6-23 客户使用仓储来保存一个新对象



另一种促使人们把工厂和仓储结合起来的情况是，希望实现一个“查找或创建”功能：客户把它希望的对象描述出来，如果没有找到这样的对象，那么就为它创建一个新的。这种功能是应该避免的。它最多也只带来了很小的一点方便。只要把实体和值对象区分开来，许多看上去有用的情况就都不会存在了。想要一个值对象的客户可以直接让工厂为它创建一个新的。在很多情况下，一个对象到底是新对象还是已有对象，把这个问题区分清楚在领域中是很重要的。用一个框架把创建和重建透明地结合起来，最终只会把问题搅乱。

6.4 为关系数据库设计对象

在以面向对象为主的软件系统中，最常见的非对象组件就是关系数据库。这个事实表现为一个常见的问题：范型的混合(参见第5章)。但是，数据库与对象模型的关系比大多数其他组件都要密切。数据库不仅仅与对象进行交互，还要把组成对象本身的数据持久保存起来。如何把对象映射到关系表、如何有效地保存和读取对象等都是非常复杂的技术，有相当多的文献对它们进行了讨论。最近的讨论可以参考 Fowler 2002。有一些相当精巧的工具可以用来创建和管理对象和关系表之间的映射。除了技术上的考虑，这二者的不匹配对于对象模型也具有显著的影响。

常见的情况有3种：

- 数据库主要用来作为对象仓储；
- 数据库是为另一个系统设计的；
- 数据库是为本系统设计的，但是用来充当其他角色，而不是存储对象。

如果数据库方案(schema)是特意针对对象存储而创建的，那么我们可以接受某些模型方面的限制，以保证映射可以非常简单。如果在设计数据库方案时没有其他的需求，那么我们可以通过精心设计数据库，来使得聚合完整性更加安全，使得数据更新更加高效。从技术上说，关系表的设计并不需要反映领域模型。我们有足够的成熟的映射工具来弥合二者之间的重大差别。问题在于，多个互相重叠的模型实在是太复杂了。许多针对模型驱动设计的争论(例如避免把分析和设计割裂开来)对这个不匹配也是有效的。因此，我们确实要在对象模型中作出一些牺牲，或者有时候在数据库设计中作出一些折衷(例如有选择性的反规范化)；但是如果这样做的话，就可能会出现导致模型和实现失去紧密关联的风险。当然，这并不是说必须使用一个对象/一个表的简单对应关系。根据映射工具所提供的功能，有的表可能可以映射为对象的聚合和组合。但是，关键的一点是映射必须是透明的，映射关系应该通过查看代码或映射工具中的配置项就能够很容易理解。



- 当数据库被视为一种对象存储时，不要让数据模型和对象模型相差太远，无论映射工具有多强大。牺牲一些对象关系的丰富性来保持与关系模型的紧密关联。如果有助于简化对象映射的话，还可以对一些形式上的关系标准(如规范化)作一些折衷。
- 在对象系统之外的处理不应访问这种对象存储机制。那样会违反对象的不变量。此外，这样的访问也会绑定到数据模型之上，导致对对象进行重构时难于修改。

另一方面，在许多情况中，数据来自老式系统或者外部系统，它们从来就不是用来保存对象的。在这种情况下，实际上是有两个领域模型同时存在于同一个系统中。第14章“维护模型完整性”深入研究了这个问题。遵循其他系统中隐含的模型可能会有道理；也有可能创建一个完全不同的模型是更好的解决之道。

出现例外的另一个原因是性能问题。为了解决执行速度的问题，我们可能需要对设计作出一些古怪的修改。

但是，在关系数据库用作面向对象领域的持久形式这种重要的常见情况中，简单的直接映射是最好的方式。表中的每一行应该包含一个对象，也许还有聚合的子对象。外键应该转换为另一个实体对象的引用。虽然有时候会对这种简单的直接映射作一些调整，但是它不应该导致把简单映射原则全盘抛弃。

通用语言有助于把对象和关系组件绑定起来，使之成为一个单一的模型。对象中元素的名称和关联应该与那些关系表小心地对应起来。虽然有的映射工具功能很强大，似乎可以使我们不需要这种对应关系，但这样可以避免由于关系之间的细微差别而引发的诸多混乱。

按惯例，重构是在对象的世界里逐步进行的，因此不会对关系数据库设计造成什么实际影响。而且，修改数据库设计会导致严重的数据迁移问题，因此我们也不鼓励频繁的改变。这可能会对对象模型的重构产生一些阻力，但是一旦对象模型和数据库模型开始出现分歧，(二者的对应关系)可能很快就会失去透明性了。

最后，有些原因会导致数据库方案与对象模型大不相同，即使数据库是专为本系统而建时也是如此。可能会有其他的软件要使用这个数据库，而那些软件不实例化对象。也可能对象的行为发生了许多变化，但是我们只能对数据库作出少量的修改。让两个模型互不干扰也是一条诱人的道路。有时候开发团队没能让数据库与模型保持同步，结果无意之中走上了这条道路。如果是特意选择将两个模型分割开来，那么我们就能得到一个清晰的数据库方案——而不是一个为了与原来的对象模型保持一致而到处进行折衷的笨拙的模型。



第7章

使用语言：扩展示例

利用前面 3 章介绍的模式语言，我们可以对模型的细节进行精炼，同时使设计严格遵循模型驱动设计。在以前的例子中，我们基本上是每次应用一个模式；但在实际项目中，不同的模式必须结合起来使用。本章将给出一个精细的示例(当然比实际项目仍然简单得多)，在这个示例中，我们假想了一个开发团队来负责处理需求和实现问题，并进行模型驱动设计开发。我们将逐步对模型和设计进行精化，阐述其中的约束，并说明怎样应用第 II 部分的模式来解决问题。

7.1 货物运输系统概述

我们要为一个货运公司开发新软件。最初的需求有 3 个基本功能。

- 跟踪顾客货物的关键装卸事件；
- 对货物进行事先预约；
- 当货物在装卸中抵达某个点时，自动向顾客发送发票。

在实际项目中，我们可能要花些时间、经过多次迭代才能得到一个轮廓分明的模型。这种发现过程将在本书的第 III 部分作深入讨论。在这里，我们一开始就有了一个模型，它已经把所需的概念以恰当的形式描述出来了，如图 7-1 所示。我们将对模型的细节稍作调整来支持设计。

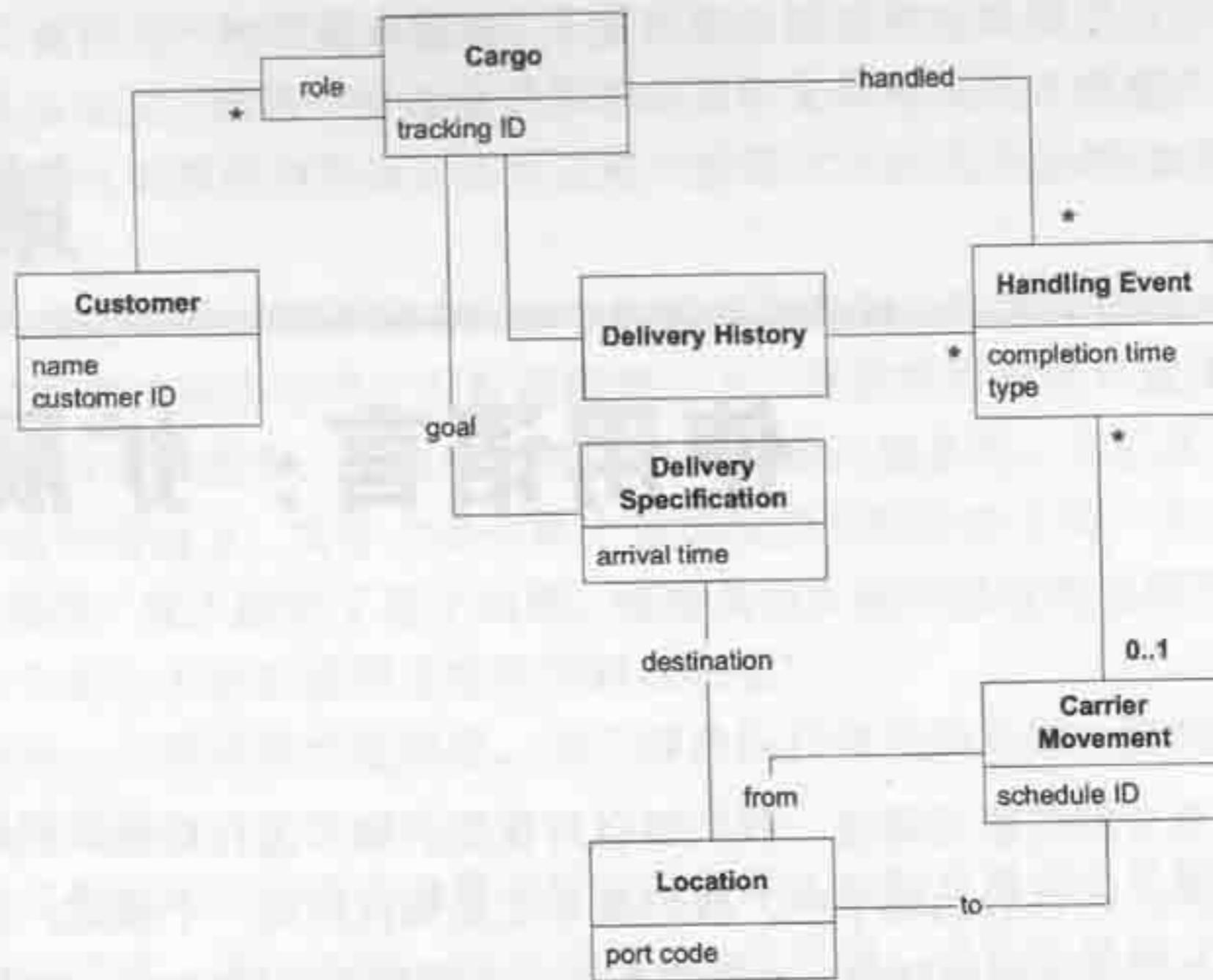


图 7-1 一个表示运输领域模型的类图

这个模型组织了领域知识，并为开发团队提供了一种语言。我们可以给出如下的陈述：

“多个 Customer(顾客)涉及到一个 Cargo(货物)，每个 Customer 充当一种不同的 role(角色)。”

“已经指定了提货目标。”

“满足 Specification 的一系列 Carrier Movement(运输动作)将实现提货目标。”

模型中的每个对象都有清晰的含义：

Handling Event(装卸事件)是发生在 Cargo 上的一种离散的活动，例如货物装船或者出关。这个类也许可以细分为一个包含不同种类事件的层次结构，例如装货、卸货，或者收货人提货等。

Delivery Specification(提货规格)定义了一个提货目标，它至少包含一个目的地和一个抵达时间，但是还可以更加复杂。这个类遵循规格模式(参见第 9 章)。

Delivery Specification 的职责本来可以由 Cargo 对象来承担，但是把它抽象出来至少可以获得 3 个好处：

- 如果没有 Delivery Specification，所有与指定提货目标有关的属性和关联就都要由 Cargo 对象来负责。这会使 Cargo 变得散乱，理解或修改会更困难。
- 当把模型作为一个整体来说明时，这种抽象使我们可以很容易且安全地把细节隐藏起来。例如，Delivery Specification 可能还封装了其他的标准，但是在这个细



节层次上，我们并不需要把它暴露出来。这个图只是告诉读者，这里有一个提货规格，但是它的细节并不重要，无需考虑(而且，以后要作修改实际上也很容易)。

- 这个模型更富有表达能力。添加 Delivery Specification 把问题说得很明白了：运输 Cargo 的实际方式可以自由确定，但是它必须达到由 Delivery Specification 给出的目标。

role 将顾客在一次运输中扮演的不同角色区分开来。例如，有“托运人”、“收货人”、“付款人”等。对于一件特定的 Cargo，一个顾客只能扮演一个角色，因此 Customer 和 Cargo 之间的关联成为限定的多对一关联，而不是多对多关联。角色可以实现为一个简单的字符串，如果还需要其他行为的话，也可以实现为一个类。

Carrier Movement 代表一个特定的 Carrier(例如一辆卡车或者一艘货轮)从一个 Location(地点)到另一个地点的转移。Cargo 就可以装上 Carrier，在一个或者多个 Carrier Movement 期间，从一个地方转移到另一个地方。

Delivery Specification 描述目标，而 Delivery History(提货历史)则反映了发生在 Cargo 上的实际行为。通过分析最后一次装货或卸货对应的 Carrier Movement 的目的地，我们可以从 Delivery History 对象计算出 Cargo 的当前 Location (位置)。如果提货成功的话，那么 Delivery History 最后应该满足 Delivery Specification 的目标。

我们把上述需求所涉及的所有概念都在模型中描述出来，并假设存在合适的机制来实现对象的持久和查找等功能。这些实现问题在模型中不会考虑，但是在设计中必须妥善处理。

为了把模型可靠地实现出来，我们还需要让模型更加清晰紧凑一些。

记住，模型精化、设计和实现一般应该在一个迭代开发过程中紧密地配合进行。但是在本章中，为了阐述的清晰性，我们使用了一个相对成熟的模型作为自己的起点，并将对修改的目的作出严格限制，即它们的目的必须把模型与实际实现连接起来(应用那些基本模式)。

一般来说，随着模型的不断精化，除了能更好地支持设计，它还应该能够反映出对领域的新的理解。但是在本章中，为了阐述的清晰性，我们将对修改的目的作出严格限制，即应用那些基本模式将模型与实际实现相连接。

7.2 隔离领域：系统简介

为了避免领域职责与系统的其他部分互相混杂，我们用分层架构把领域层划分出来。无需作深入分析，我们就可以标识出 3 个用户级的应用功能，并将其分配给 3 个应



用层类。

- Tracking Query(跟踪查询)可以访问一件特定 Cargo 过去的和现在的装卸信息;
- Booking Application(预订应用)用来注册一件新的 Cargo, 并让系统准备对其进行处理;
- Incident Logging Application(事件日志应用)用来记录 Cargo 的各次装卸信息(它提供的信息由 Tracking Query 查询得到)。

这些应用类都是起协调作用的。它们只管提出问题, 但是不管解决问题。解决那些问题是领域层的工作。

7.3 区分实体和值对象

现在我们将逐个来考察这些对象, 看它是实体(包含必需跟踪的标识)还是值对象(只代表一个基本的值)。首先我们研究一些较容易看出来的情况, 然后再考虑比较模糊的情况。

1. Customer(顾客)

我们从一个容易一些的对象开始。一个 Customer(顾客)对象代表一个人或者一个公司, 在通常意义下这是一个实体。Customer 显然有标识, 而且对用户很重要, 因此它在模型中是一个实体。怎样跟踪它呢? 在某些情况下税号(Tax ID)可能比较合适, 但是跨国公司不会这样做。这个问题需要向领域专家咨询。我们与货运公司的一个业务人员对这个问题进行了讨论, 发现公司早就有顾客数据库了, 其中的每个顾客分配了一个 ID 数字, 在与顾客进行首次销售接触时确定的。整个公司都在使用这种 ID, 因此我们的软件也会这样做, 以便保持与其他系统的标识连续性。客户 ID 在初始时是用手工输入的。

2. Cargo(货物)

两个同样的货箱必须进行区分, 因此 Cargo(货物)对象是实体。实际上, 所有货运公司都会给每件货物指定一个跟踪 ID。这个 ID 将是自动产生的, 用户可以看到它, 在预订时它还可能发给顾客。

3. Handling Event(装卸事件)和 Carrier Movement(承运人运输)

我们关心每个个体事件, 因为我们需要这些信息来跟踪进展。这些事件反映了现实



世界中发生的事件，通常是不可互换的，因此它们是实体。每个 Carrier Movement 都会从运输计划中得到一个代码作为标识。

我们与领域专家进行了另一次讨论，发现把 Cargo ID、完成时间和类型组合起来可以惟一确定一个 Handling Event。例如，同一个 Cargo 不会同时既装货又卸货。

4. Location(地点)

名称相同的两个地点不是同一个地点。我们可以把经度和纬度作为地点的惟一主键，但是这可能不太现实，因为这种方法与系统的大部分目标都没有多大关系，而且它相当复杂。Location 地点更有可能会包含在某种地理模型之中，将地点和运输航线及其他特定的领域问题联系起来。因此，用随机的、内部自动产生的标识就足够了。

5. Delivery History

Delivery History(提货历史)有些复杂。Delivery History 是不可互换的，因此它们是实体。但是，Delivery History 与它发运的 Cargo 具有一对一关联，因此它实际上并没有自己的标识。它的标识是从所属的 Cargo 借用过来的。用聚合来建模将使问题更加清晰。

6. Delivery Specification(提货规格)

虽然 Delivery Specification 代表 Cargo 的目标，但是这个抽象并不依赖于 Cargo。它实际上描述了某些 Delivery History 的假想状态。我们希望与一件 Cargo 连结的 Delivery History 最终能够满足与它连结的 Delivery Specification。如果两件 Cargo 的目的地相同，那么它们可以共享同一个 Delivery Specification；但是，虽然二者的历史在最开始时都相同(都为空)，但它们不能共享同一个 Delivery History。Delivery Specification 是值对象。

Role(角色)和其他属性

Role 角色提供了与其限定的关联有关的信息，但是它没有历史的或连续性的要求。它是一个值对象，可以在不同的 Cargo/Customer 关联中进行共享。

其他属性，如时间戳或者名字，都是值对象。

7.4 运输领域中的关联设计

在前面的类图中，所有关联都没有指定游历方向。但是，设计中的双向关联容易引起问题。此外，游历方向往往捕获了对领域的理解，可以使模型本身更加深入。



如果让 Customer 直接引用它托运的所有 Cargo，那对于长期、多次合作的 Customer 来说就会非常不便。在大型系统中，Customer 可能在许多对象中充当不同的角色，因此最好是避免让它承担这种特定的职责。如果需要通过 Customer 来查询 Cargo，那可以通过数据库查询来实现。本章稍后在讨论仓储的部分将回过头来讨论这个问题。

如果我们的系统要对货轮进行跟踪，那么从 Carrier Movement 游历到 Handling Event 的关联就非常重要了。但是，我们的业务只要求对 Cargo 进行跟踪。为了反映这一点，我们把关联修改为只允许从 Handling Event 游历到 Carrier Movement。这个关联的实现也被简化为一个简单的对象引用，因为“多”那个方向的关联不允许。

图 7-2 解释了其他修改决定的理由。

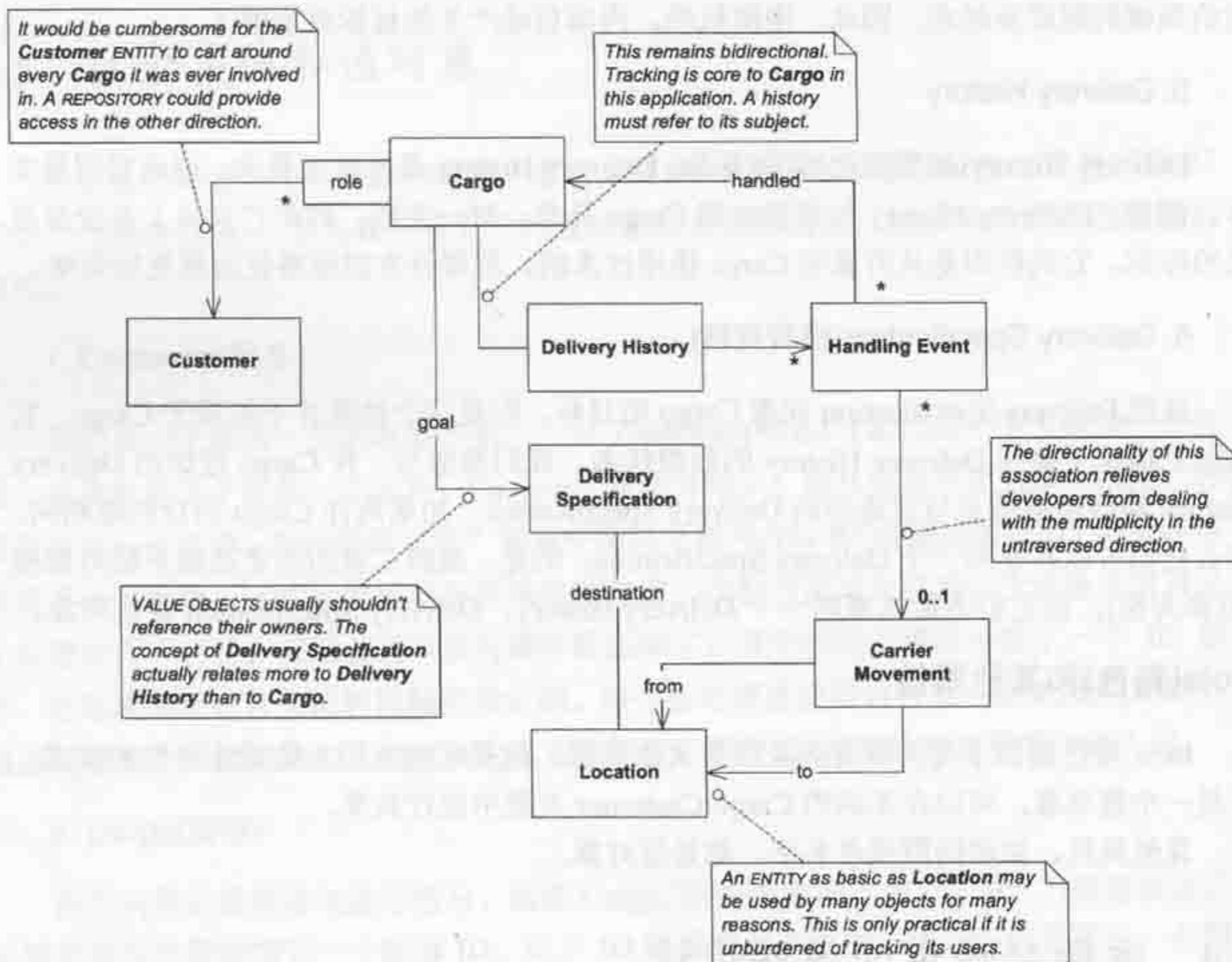


图 7-2 限制某些关联的游历方向

在我们的模型中有一个循环引用关系：Cargo 知道它的 Delivery History，Delivery History 包含一系列 Handling Event，而 Handling Event 又反过来指向 Cargo。循环引用合

理地存在于许多领域当中，有时在设计中也是需要的，但是维护起来很复杂。合理选择实现可以避免把需要同步的同一个信息保存到两个不同的地方。在这个例子中，我们可以使用一种简单但脆弱的方法，用 Java 语言来实现一个初步的原型——在 Delivery History 中用一个 List 对象来保存 Handling Event。但是，有时候我们可能不想使用集合，而选择把 Cargo 作为键来执行数据库查询。这个问题在选择仓储时会再次讨论到。如果查看历史的频率不是很高，那么通过数据库来查询就可以获得较好的性能，简化维护问题，并降低新增 Handling Event 的开销。如果这个查询非常频繁，那么最好还是按原来的方式维护直接指针。这些都是实现的简单性与性能之间的设计权衡。模型同样也包含了那个循环引用和双向关联。

7.5 聚合的边界

Customer、Location 和 Carrier Movement 都有自己的标识，并可以被许多 Cargo 共享，因此它们必须是自己的聚合根。聚合包含了它们的属性，可能还有一些其他的下层对象。Cargo 显然也是一个聚合根，但是它的边界如何确定还要思考一番才行。

如图 7-3 所示 Cargo 聚合可以把所有因为 Cargo 而存在的事物都包揽进去，包括 Delivery History、Delivery Specification 以及 Handling Event。Delivery History 确实也可以放到 Cargo 中——如果不是为了某个特定的 Cargo，是不会有人去直接查看 Delivery History 的。由于 Delivery History 无需直接的全局访问，其标识实际上又是从 Cargo 获取的，因此把它放在 Cargo 的边界之中非常适合，而且它也无需充当根。Delivery Specification 是一个值对象，因此把它放在 Cargo 聚合中没有什么复杂之处。

但是，Handling Event 有所不同。前面我们已经考虑了针对 Handling Event 的两种可能的数据库查询：一种是查询 Delivery History 所包含的 Handling Event，以此作为位于 Cargo 聚合内的集合的一种可能替代；另一种用来查询某次特定的 Carrier Movement 所需准备的货物和装卸操作。在第二种情况下，货物装卸活动本身，即使是脱离 Cargo 来考察，也已经具有了某种意义。因此 Handling Event 应该成为它自身的聚合根。

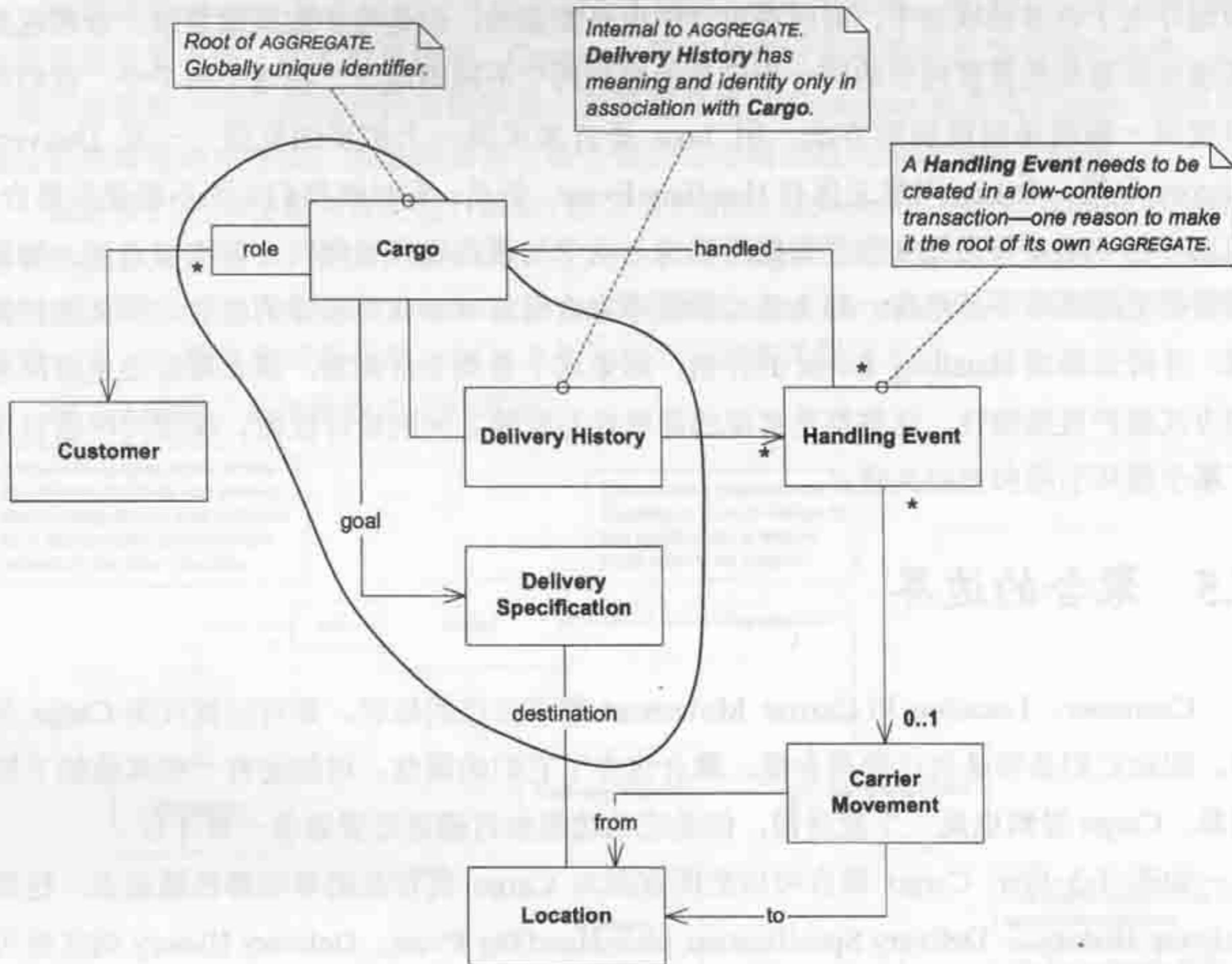


图 7-3 模型中设定的聚合边界(注意：画出的边界之外的实体意味着是其自身的聚合根)

7.6 选择仓储

设计中有 5 个实体是聚合根。我们只需考虑这几个对象，因为其他对象都不允许有仓储。

为了决定哪些候选者确实需要仓储，我们必须重温一下应用需求。为了在 Booking Application 中进行预约，用户需要选择不同角色(托运人、收货人等)的 Customer。因此，我们需要一个 Customer Repository。用户还要查找 Location 来指定 Cargo 的目的地，因此 Location Repository 也是需要的。

Activity Logging Application 需要允许用户查找装载了给定 Cargo 的 Carrier Movement，因此我们需要一个 Carrier Movement Repository。用户需要告诉系统哪个

Cargo 已经装载，因此还需要一个 Cargo Repository，如图 7-4 所示。

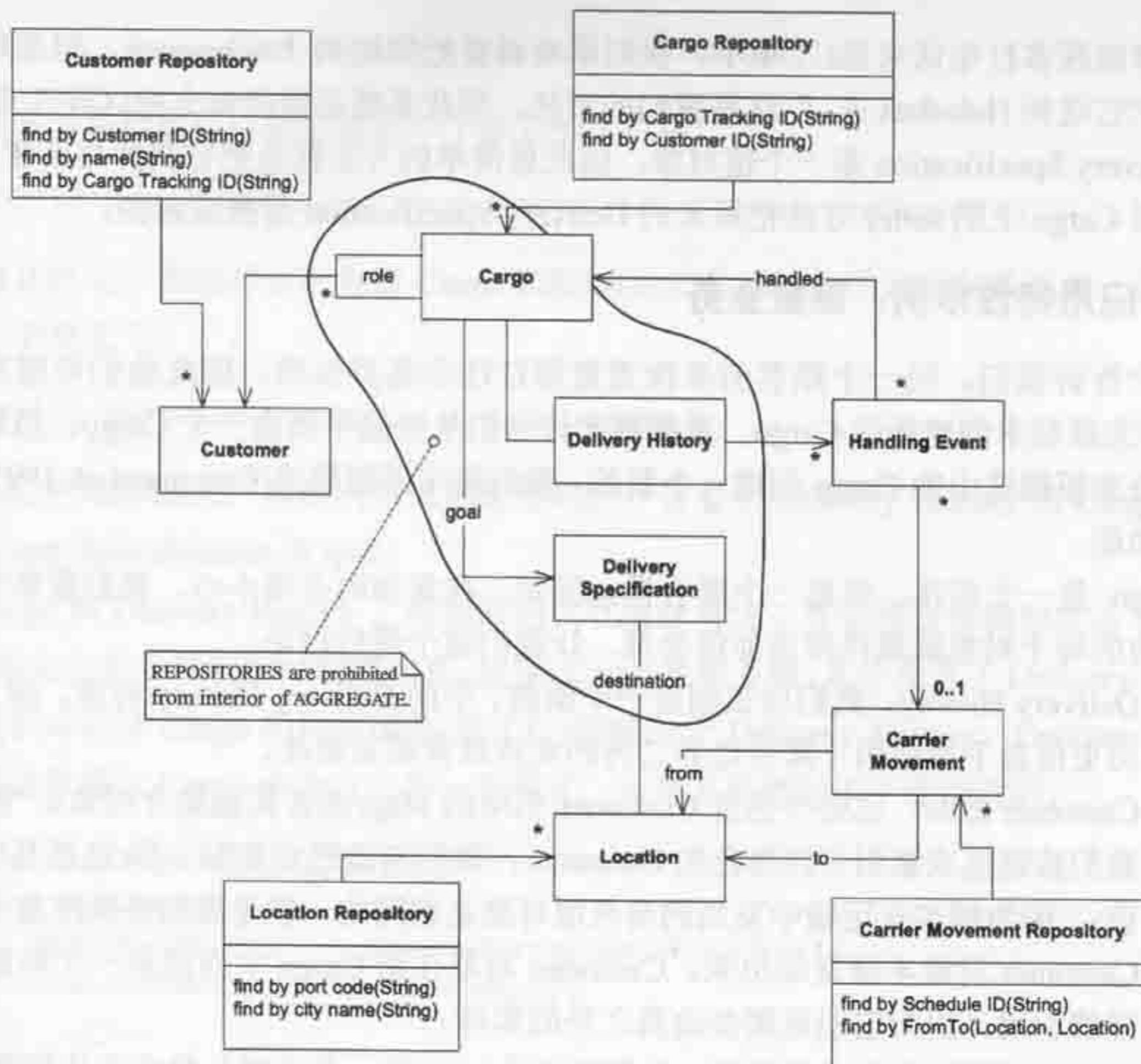


图 7-4 仓储为聚合根提供访问

这里并没有建立 Handling Event Repository，一个原因是决定在第一次迭代时把 Handling Event 和 Delivery History 的关联实现为一个集合；另一个原因是尚未把“找出 Carrier Movement 上装载了哪些货物”这样的功能列入应用需求。这两个原因都有可能发生变化；一旦真的发生了变化的话，我们就要把 Handling Event Repository 添加进来。

7.7 场景概述

为了反复检验这些设计决定，我们必须经常性地进行场景走查，以确保我们可以有效地解决应用问题。



7.7.1 应用特性示例：改变一件货物的目的地

有时候顾客打电话来说：“哦不！我们原来说要把货送到 Hackensack，但是现在我们必须把它送到 Hoboken 去。”这是我们的责任，因此系统必须能够支持这种改变。

Delivery Specification 是一个值对象，因此最简单的方法就是把它扔掉再新建一个，然后调用 Cargo 上的 setter 方法把原来的 Delivery Specification 替换成新的。

7.7.2 应用特性示例：重复业务

用户告诉我们，同一个顾客的多次重复预订往往是类似的，因此他们希望用旧的 Cargo 作为原型来创建新的 Cargo。系统将允许他们从仓储中找出一个 Cargo，然后执行一条命令来根据选出的 Cargo 创建一个新的。我们将用原型模式(Gamma et al. 1995)来设计这个功能。

Cargo 是一个实体，也是一个聚合根。因此，在复制时必须小心。我们需要考虑聚合边界内的每个对象或属性应该如何处理。让我们逐个进行讨论：

- **Delivery History:** 我们应该创建一个新的、空的 Delivery History 对象，因为老的历史信息不再适用。聚合边界之内的实体通常都是如此。
- **Customer Role:** 这是个包含 Customer 引用的 Map(或者其他集合对象)，它允许我们按键值来索引不同角色的 Customer。我们应该把它复制一份(包括其中的键值)，因为顾客在运输中充当的角色很可能是相同的。但是我们必须注意不要把 Customer 对象本身复制出来。Customer 对象在新 Cargo 中应该是一个和原来一样的引用，因为它们是聚合边界之外的实体。
- **Tracking ID:** 我们必须提供一个新的 Tracking ID，其来源与我们从头创建一个新 Cargo 对象时相同。

注意我们复制了 Cargo 聚合边界内的所有东西。虽然我们在复制时作了一些修改，但是这没有对聚合边界以外的任何对象产生任何影响。

7.8 对象的创建

7.8.1 Cargo 的工厂和构造函数

无论是为 Cargo 创建一个强大的工厂，还是用其他 Cargo 作为工厂(如“重复业务”中那样)，我们都需要有一个基本的构造函数。我们希望构造函数产生的对象能够满足它的不变量；或者对于实体来说，至少能保持其标识不变。



根据这些考虑，我们可以在 Cargo 上创建一个如下所示的工厂方法：

```
public Cargo copyPrototype(String newTrackingID)
```

或者，我们也可以把它放到一个独立的工厂中：

```
public Cargo newCargo(Cargo prototype, String newTrackingID)
```

独立的工厂类还可以把为新 Cargo 获取(自动产生)新 ID 的过程封装起来，这样它就只需一个变元了：

```
public Cargo newCargo(Cargo prototype)
```

从任何这样的工厂得到的结果都是相同的：带着空 Delivery History 的 Cargo，其中的 Delivery Specification 为 null。

Cargo 和 Delivery History 之间的双向关联意味着这二者必须互相指向对方才算是完整的，因此它们必须被同时创建出来。注意，Cargo 是聚合根，其中包含了 Delivery History，因此我们可以让 Cargo 的构造函数(或工厂)创建一个 Delivery History。Delivery History 的构造函数将以 Cargo 作为自己的一个变元，结果有些类似于下面的代码：

```
public Cargo(String id) {  
    trackingID = id;  
    deliveryHistory = new DeliveryHistory(this);  
    customerRoles = new HashMap();  
}
```

上面的代码将生成一个新的 Cargo 对象，其中包含了一个指向它本身的 Delivery History。Delivery History 的构造函数仅能被其聚合根使用，因此 Cargo 的组合被封装了起来。

7.8.2 添加一个 Handling Event

当货物在现实世界中发生装卸时，就会有某个用户通过 Incident Logging Application 来输入一个 Handling Event。

每个类都必须有基本的构造函数。由于 Handling Event 是一个实体，因此所有用来定义其标识的属性都应该被传递给它的构造函数。我们在前面讨论过，把 Cargo ID、完成时间和事件类型三者组合起来可以惟一确定一个 Handling Event 的标识。Handling Event 中剩下的惟一一个属性是到 Carrier Movement 的关联，但是某些类型的 Handling



Event 根本就不需要这样的关联。创建一个有效的 Handling Event 的基本构造函数可以是这样的：

```
public HandlingEvent(Cargo c, String eventType, Date timeStamp) {  
    handled = c;  
    type = eventType;  
    completionTime = timeStamp;  
}
```

实体的非标识性属性通常可以在以后添加进来。在这里，Handling Event 的所有属性将在初始事务中设置好，以后不会再修改了(除非要更正一个错误的数据字段)。因此我们可以在 Handling Event 中为每种事件类型加入一个简单的工厂方法，并提供所有必需的变元。这样会非常方便，客户代码也会更有表达力。例如，创建 loading event 时包含一个 Carrier Movement：

```
public static HandlingEvent newLoading(  
    Cargo c, CarrierMovement loadedOnto, Date timeStamp) {  
    HandlingEvent result =  
        new HandlingEvent(c, LOADING_EVENT, timeStamp);  
    result.setCarrierMovement(loadedOnto);  
    return result;  
}
```

模型中的 Handling Event 是一种抽象，它可能封装了许多特化的 Handling Event 类，包括装货、卸货、加封条、存放，以及其他与 Carrier 无关的活动等。我们可以把它们实现为成多个子类，然后分别创建不同的子类；或者通过复杂的初始化来创建这些对象；也可以同时使用这两种方法。通过在基类(Handling Event)中为每种类型加入一个工厂方法，我们可以把实例创建工作抽象化。这样客户就无需知道实现细节，而由工厂来负责了解实例化哪个类，以及如何初始化的问题。

遗憾的是，故事并不像这么简单。Cargo 到 Delivery History 到 Handling Event 再回到 Cargo 的这个引用循环把实例创建问题搞复杂了。Delivery History 持有一个与其 Cargo 相关的 Handling Event 集合，而新创建的 Handling Event 对象必须在同一个事务中加入到这个集合中来。如果不创建这种反向指针，对象将是不一致的，如图 7-5 所示。

反向指针的创建可以用工厂封装起来(并放在其所属的领域层中)，但是现在我们将看到另一种设计方法，它能完全避免图中这种难处理的交互过程。

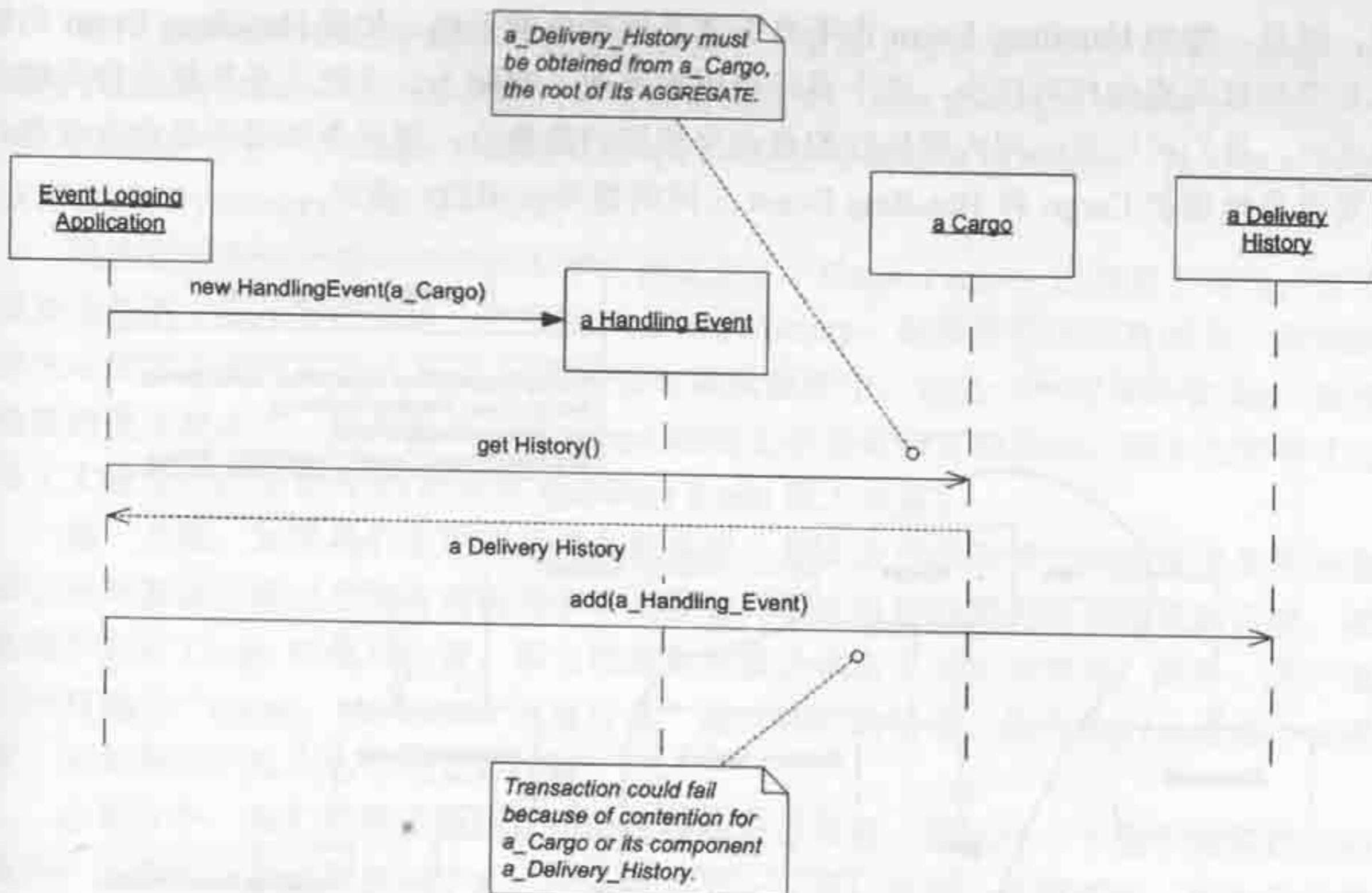


图 7-5 添加入 Handling Event 要求把它插入 Delivery History 中

7.9 停下来重构：Cargo 聚合的另一种设计

建模和设计不是一个匀速向前的过程。如果不频繁重构，利用新的理解来改进模型和设计，我们就会逐渐变得寸步难行。

现在，我们的设计可以开始工作了，也确实反映出了模型，但其中仍有一些笨拙之处。一些在开始设计的时候并不显得重要的问题现在变得越来越烦人了。让我们充当一回事后诸葛亮，回过头来对付其中的一个问题，使我们的设计能为下一步的工作打好基础。

由于在增加 Handling Event 时需要更新 Delivery History，因此 Cargo 聚合也被牵扯到这个事务中来了。如果此时某个其他用户正在修改 Cargo，那 Handling Event 事务就会失败或者被推迟。输入 Handling Event 的操作活动必须简单快捷，因此无竞争地输入 Handling Event 成为一个重要的应用需求。这迫使我们考虑另一种设计。

我们可以把 Delivery History 中的 Handling Event 集合用一个查询来代替，如图 7-6 所示。这样修改以后，加入 Handling Event 不会在其自身的聚合之外引起任何完整性问



题；而且，增加 Handling Event 也不再受到其他事务的干扰。如果 Handling Event 的输入非常频繁而查询相对较少，这个设计会更加高效。实际上，当把关系数据库作为底层技术时，我们可以设法用内部执行的查询来模拟对象集合。使用查询而不是集合使我们能更容易地维护 Cargo 和 Handling Event 之间的循环引用的一致性。

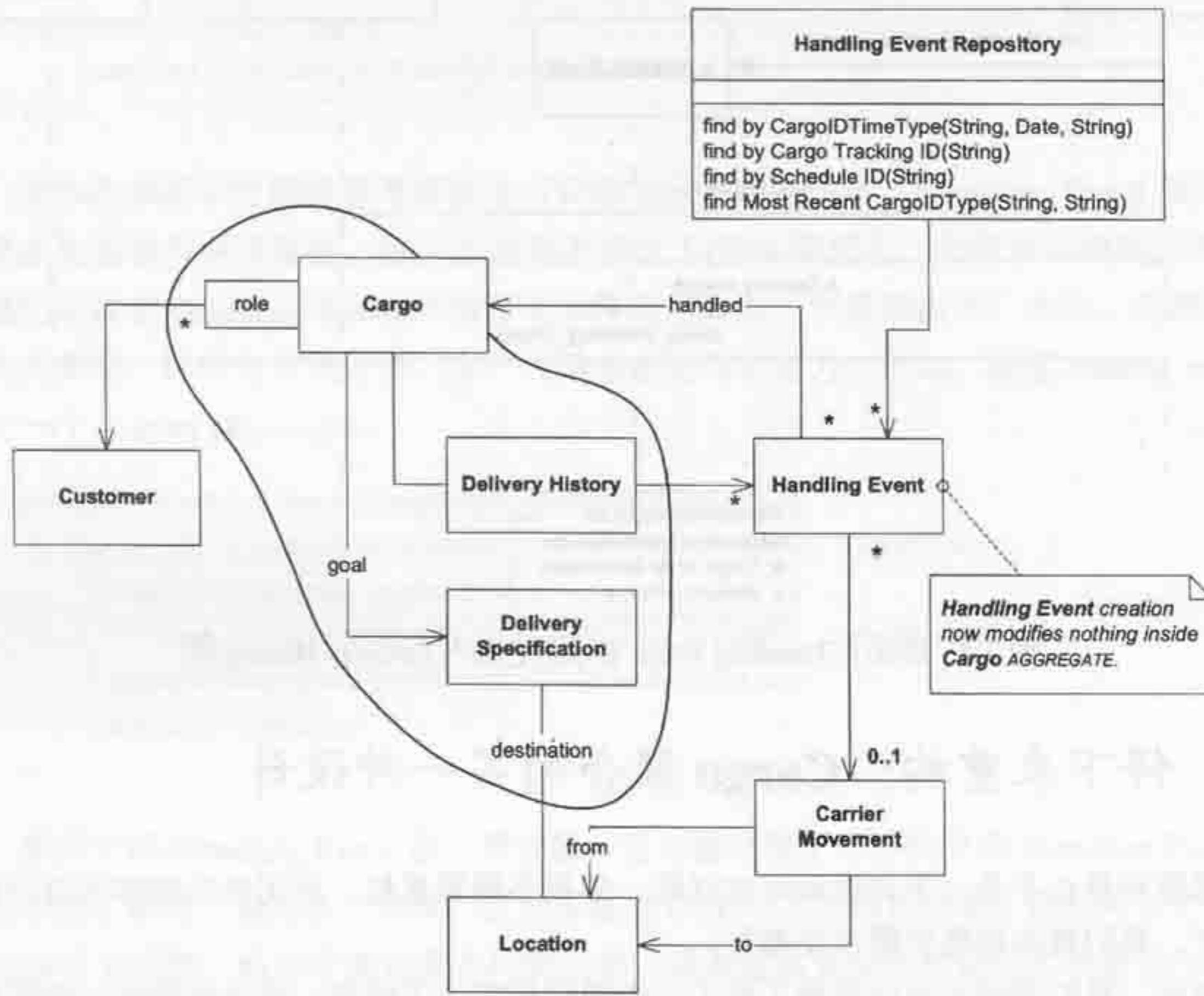


图 7-6 把 Delivery History 中的 Handling Event 集合用一个查询代替之后，
插入 Handling Event 就很简单了，也不会与 Cargo 聚合出现竞争

我们为 Handling Event 增加一个仓储来承担查询的职责。Handling Event Repository 将提供一个根据给定 Cargo 查询其相关 Event 的功能。此外，仓储还可以为某些特定的问题进行优化，使其获得更高的查询效率。例如，为了推断 Cargo 的当前状态，我们需要根据 Delivery History 来查找最后一次报告的装货或卸货。如果这个访问路径使用非常频繁，那么我们可以设计一个专门的查询，仅仅返回与最后一次报告相关的 Handling Event。如果我们希望用一个查询来得出某个给定 Carrier Movement 中装载的所有 Cargo，也只要增加一个查询就行了。



这样 Delivery History 就不再有持久状态了，实际上保留它已经没有任何必要。当需要用 Delivery History 来回答问题时，可以查询结果。虽然我们每次就得重新创建这个实体，但是它与同一个 Cargo 对象的关联使得不同的实例能够连贯起来，因此通过查询来获得 Delivery History 对象是可行的。

现在创建和维护循环引用的工作不再复杂了。Cargo Factory 也得到了简化，它不再需要为新的 Cargo 实例创建一个空的 Delivery History。数据库空间稍有减少，而实际的持久对象的个数可能会大为减少(在某些对象数据库中，这是一种有限的资源)。如果在通常的使用模式下，用户极少会在 Cargo 抵达之前去查询它的状态，那么这种设计也避免了大量不必要的开销(把所有 Handling Event 载入内存)。

另一方面，如果我们使用的是对象数据库，那么应该注意通过关联或者显式的集合来访问对象会比通过仓储查询快得多。如果访问模式包括频繁列出全部装卸历史，而不是偶尔定位 Cargo 的最后位置，那么性能权衡就会偏向于显式的集合。此外，用户现在还没有提出“Carrier Movement 上有什么”这个额外的特性，也许他们永远都不会提出来，因此我们不想为它付出太多代价。

在设计中，我们经常会遇到这样的选择和设计权衡，像这样一个很小的简化过的系统中，我都可以举出许多例子来。但是重要的一点是，在同一个模型中，自由是有限度的。通过把值、实体及其聚合建模出来，我们已经把那些设计改变的影响大大降低了。例如，在这个例子中所有修改都被封装在 Cargo 聚合边界之内。我们还加入了一个 Handling Event Repository，但是它并不需要重新设计 Handling Event 本身(但是根据仓储框架的细节，可能会涉及到某些实现上的修改)。

7.10 运输模型中的模块

到现在为止，我们所看到的对象都非常少，所以模块化根本不是一个问题。现在让我们来看一个运输模型中稍大一点的部分(当然还是作了简化)，来讨论把它组织到模块之中将如何影响这个模型。

图 7-7 演示了一个假想的模型，它划分得很整洁。在第 5 章中，我们讨论过基础结构驱动打包的问题，这个图是那个问题的一个变体。在图中，对象是根据其所用的模式来进行组织的，结果那些在概念上相去甚远的对象勉强堆在了一起(低内聚)，而模块之间的关联则连得乱七八糟(高关联)。每个包都有含义，但不是运输业务上的含义，而是开发人员眼中的含义。



第 II 部分 模型驱动设计的构建块

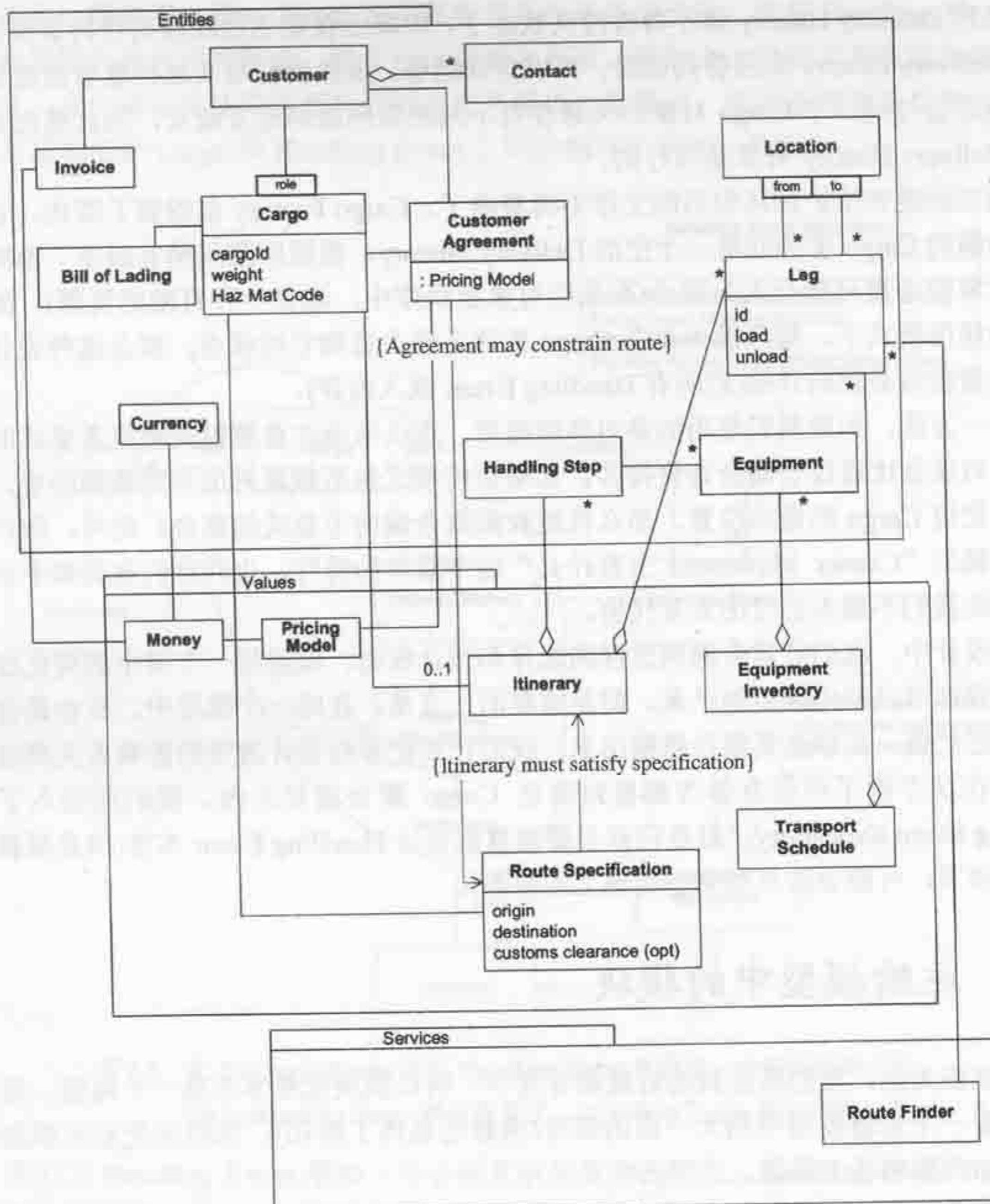


图 7-7 这些模块没有传达领域知识

按模式来划分看起来像是一个明显的错误。但是，如果按对象是持久对象还是临时对象来划分，或者按任何其他技术方案而不是根据对象的含义来划分，那也不会比按模式划分好到哪里去。

相反，我们应该寻找具有内聚性的概念，并集中考虑我们希望与项目中的其他人交

流什么。对于规模较小的建模来说，我们可以有多种划分的方法。图 7-8 演示了一种比较直接的方法。

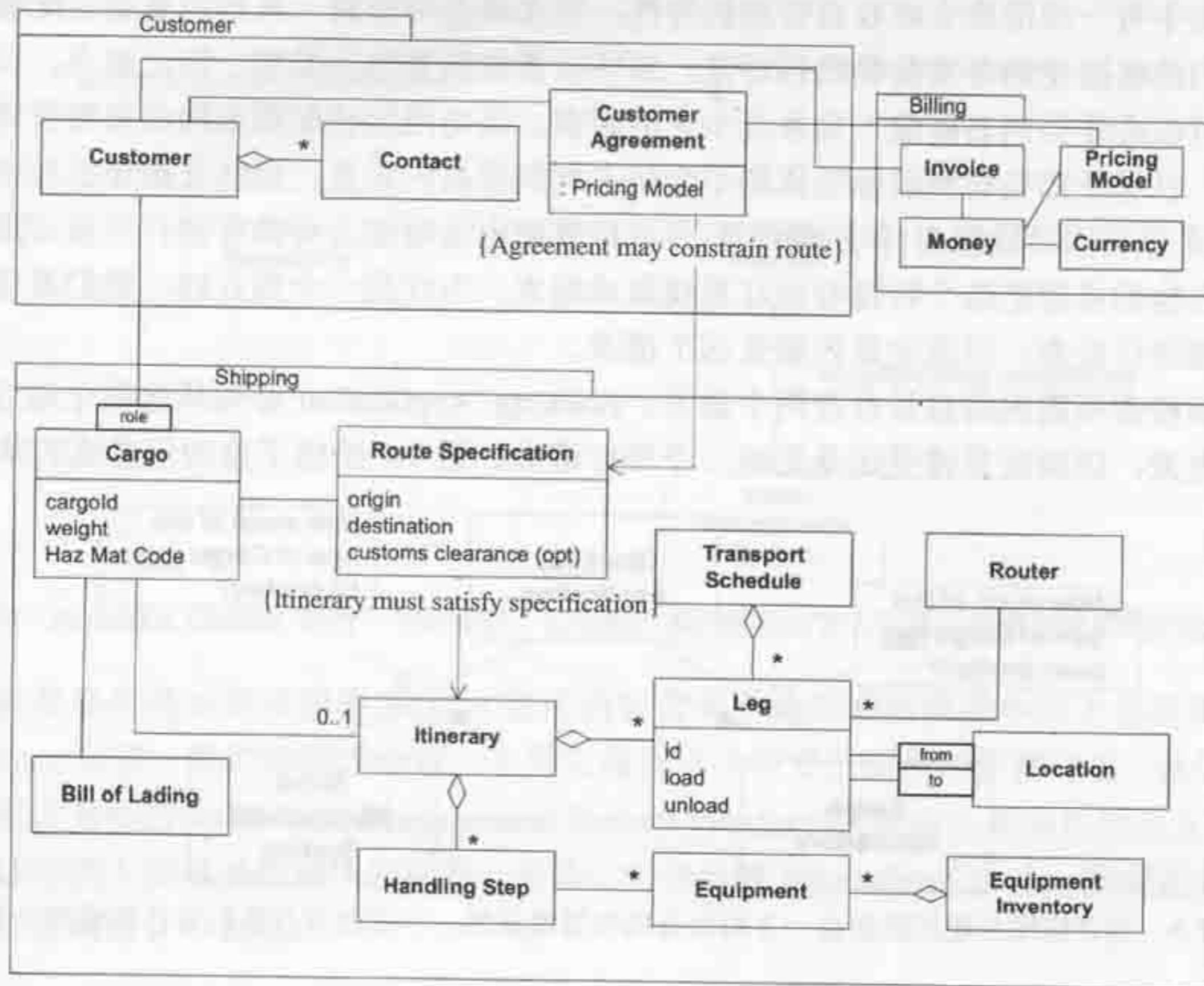


图 7-8 根据大范围的领域概念划分模块

图 7-8 中的模块的名称丰富了开发团队的语言。我们的公司向顾客(Customer)发货(Shipping)，因此我们可以向他们发出账单(Billing)。销售和市场人员和顾客打交道，操作人员执行发货(Shipping)，把货物送到指定的目的地。财务部门负责处理帐单(Billing)，根据顾客同意的价格为其开具发票。我们可以根据这些模块说出一个故事。

当然，上面这种按直觉的划分可以通过连续的迭代来精化，甚至用一种更好的方法来替代它。但是，它为我们的模型驱动设计提供了帮助，同时还丰富了我们的通用语言。

7.11 引入新特性：配额检查

到这里，我们已经逐步解决了最初的需求和建模问题。现在我们将加入第一个比较



大的新功能。

在我们假想的运输公司中，销售部门使用其他的软件来管理客户关系、营销计划等业务。其中有一项用来支持收益管理的特性，它允许公司根据一系列因素制定配额，来限制他们能够接受的各类货物的预订量。那些因素包括货物的类型、起止地点，以及其他任何可以选作类别名称输入到系统中去的因素。公司用这些配额来构成每种货物的销售目标，以保证那些利润较低的货物不会挤占利润较高的业务，同时又避免出现预订不足(没有充分利用运输能力)和超额预订(导致频繁超出运输能力而损害客户关系)的情况。

现在他们希望把这个特性与预订系统集成起来。当收到一个预订时，他们希望能够根据配额进行检查，以决定是否接受这个请求。

配额检查所需的信息保存在两个地方。Booking Application 必须从这两个地方把信息查询出来，以决定是接受还是拒绝一个预订请求。图 7-9 给出了总的信息流的草图：

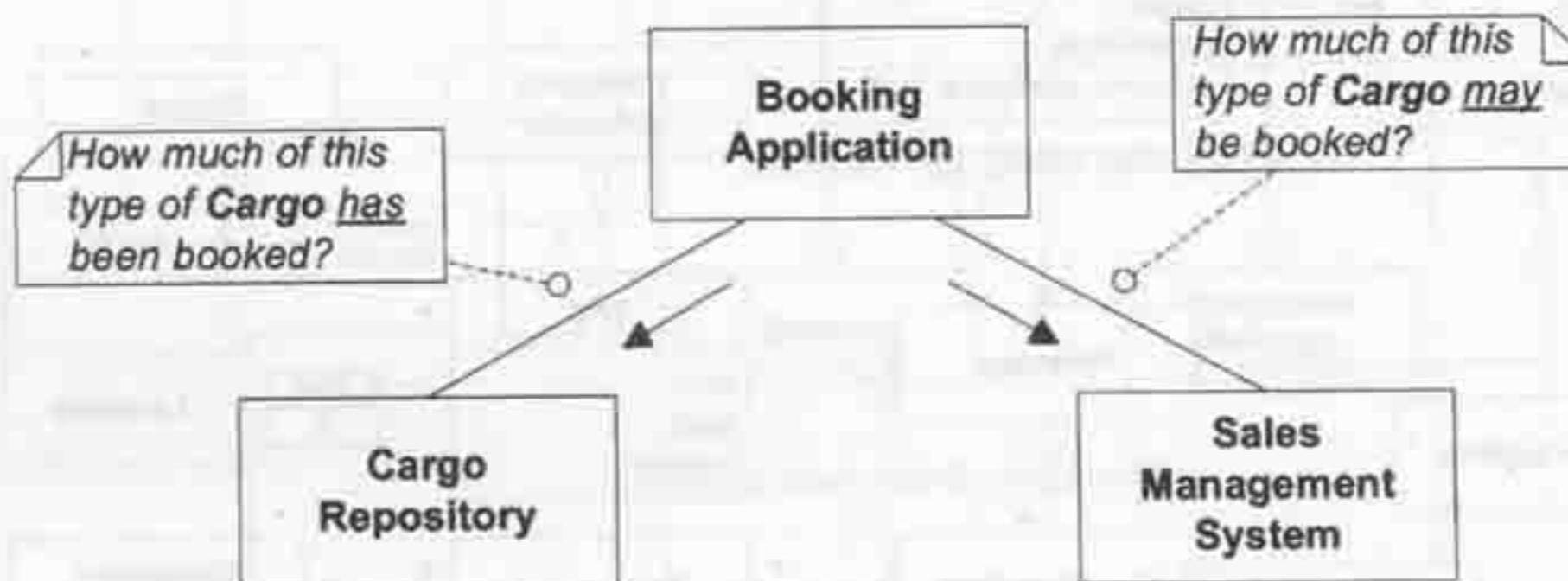


图 7-9 预订应用所使用的信息一方面来自销售管理系统，一方面来自我们自己领域的仓储

7.11.1 连接两个系统

销售管理系统(Sales Management System)并不是按照我们设计的模型来编写的。如果让预订应用程序(Booking Application)直接与它交互，那么我们的系统就必须兼容其他系统的设计，这将使我们更难保持模型驱动设计的方向，也会扰乱通用语言。相反，我们可以创建另一个类，让它来充当我们的模型与销售管理系统所使用的语言之间的翻译。这个类并不是要提供一种通用的翻译机制，而是仅仅提供我们的系统所需的特性，并根据我们的领域模型对其进行重新抽象。这个类将充当一个防腐层(在第 14 章讨论)。

由于这是一个销售管理系统接口，我们首先可能会想到把它命名为 Sales Management Interface。但是我们可以抓住这个机会，用更加有用的说法把问题重申出来。对每个需要从其他系统得到的配额功能，我们可以为它定义一个服务，然后用这个服务的实现类的类名来反映其在我们的系统中的职责：配额检查者(Allocation Checker)，如图 7-10 所示。

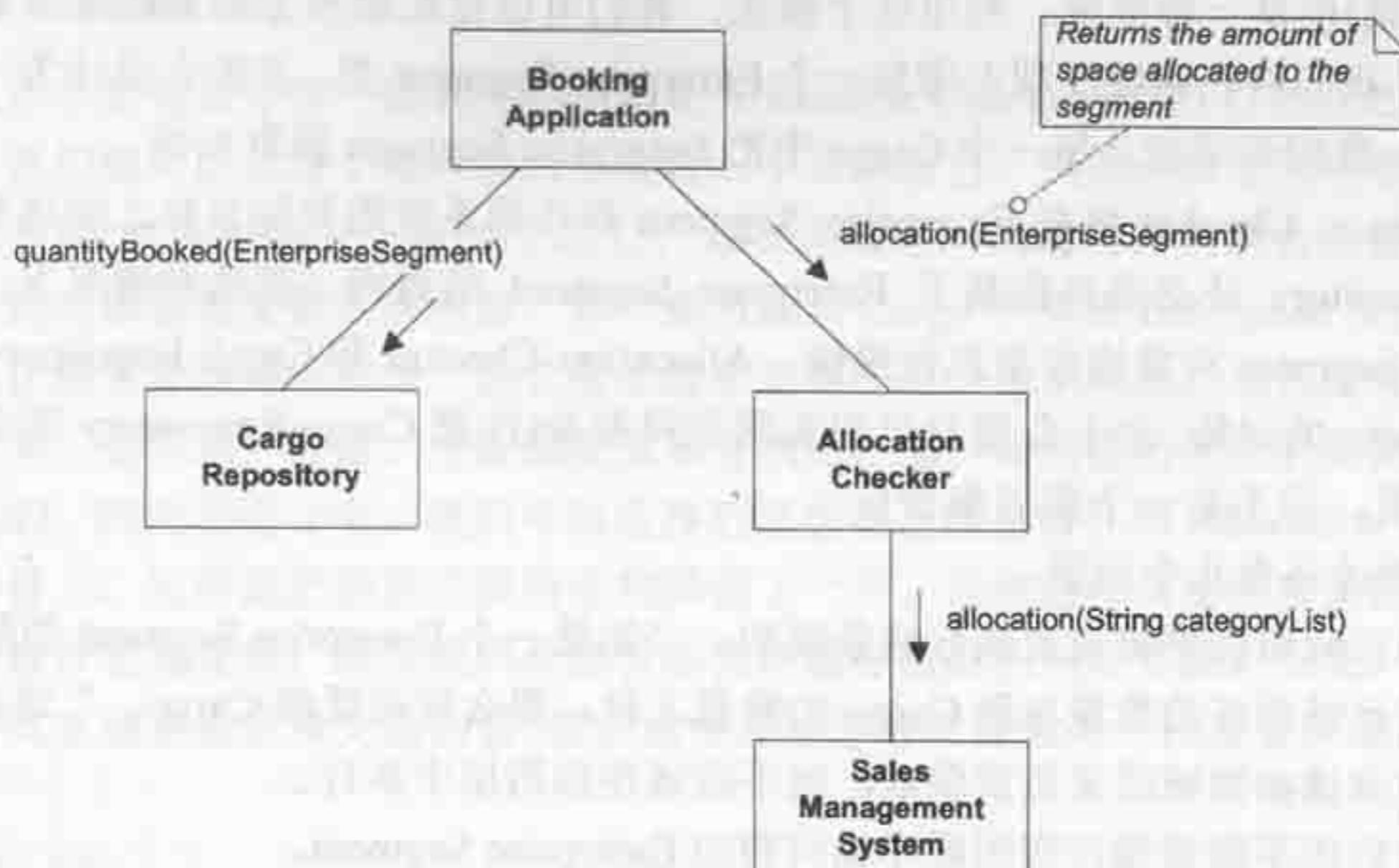


图 7-10 Allocation Checker 充当一个防腐层，它用我们领域模型的方式展现了销售管理系统的适配性接口

如果有其他的集成需要(例如，使用销售管理系统的顾客数据库而不是我们自己的 Customer 仓储)，我们还可以创建一个带有履行这个职责的服务的新翻译器。我们仍然可以用层次更低的类(如 Sales Management System Interface)来操纵与其他程序交互的机制，但是这种类不应该承担翻译的职责。另外，它也会被 Allocation Checker 隐藏起来，因此不会在领域设计中出现。

7.11.2 改进模型：划分业务

现在两个系统之间的交互已经大体清楚了，那么我们应该提供一个什么样的接口，才能回答“可以预定多少某种类型的 Cargo”这个问题呢？这个问题的一个棘手之处是 Cargo 的“类型”究竟是什么，因为我们的领域模型还没有对 Cargo 进行分类。在销售管理系统中，Cargo 类型只是一系列类别关键字，我们可以使类型遵从这个类别列表，然后把它们放到一个字符串集合中作为变元传进去。但是这样我们又会错失另一个机会：重新抽象其他系统的领域。我们需要把货物类别的知识添加进来，使领域模型更加丰满。我们应该与领域专家进行头脑风暴，把新概念界定出来。

有时(就像在第 11 章中将要讨论的那样)，分析模式可以为我们的建模方案提供思路。Analysis Patterns(Fowler 1996)书中描述了一个用来处理这种问题的模式：Enterprise Segment(企业部门单元)。企业部门单元是一个维度集合，每个维度定义了一种对业务进行分解的方法。我们已经提到的所有分解运输业务的方法都属于这些维度，此外时间维



(如月份日期)也是一种维度。利用这个概念，我们可以使配额模型的表达能力更强，并简化接口。我们将在领域模型中增加一个 Enterprise Segment 类，并把它设计为一个额外的值对象。我们必须能从每一个 Cargo 中把 Enterprise Segment 提取出来。

Allocation Checker 将在 Enterprise Segment 和外部系统的类别名称之间进行转换。Cargo Repository 还必须提供基于 Enterprise Segment 的查询。在两种情况下，通过与 Enterprise Segment 对象协作来执行操作，Allocation Checker 和 Cargo Repository 既不会破坏 Segment 的封装，也不会使自己的实现变得复杂(注意 Cargo Repository 的查询是返回一个总量，而不是一个实例集合)。

这个设计还有几个问题。

- 预订应用程序将负责执行这条规则：“如果一个 Enterprise Segment 的配额大于其已经预订的数量与新 Cargo 的数量之和，那么就接受该 Cargo。”这条业务规则应该由领域层来负责保证，而不应该在应用层中执行。
- 我们还不清楚预订应用程序如何得出 Enterprise Segment。

这两个职责看起来都属于 Allocation Checker。我们可以修改 Allocation Checker 的接口，把这两个服务分离开来，使交互过程更加清楚和明显，如图 7-11 所示。

这个集成方案只有一个严重的约束：销售管理系统不能使用无法由 Allocation Checker 转换为 Enterprise Segment 的维度(如果不用企业部门单元模式，同样的约束会使得销售管理系统仅仅只能使用 Cargo Repository 中的查询所使用的维度。虽然这样也行，但是它把销售管理系统的细节泄漏到领域的其他部分中去了。在我们的设计中，Cargo Repository 只需对 Enterprise Segment 进行处理(无需关心销售管理系统)；而销售管理系统的变化只会波及到 Allocation Checker。首先我们可以把 Allocation Checker 视为一个外观)。

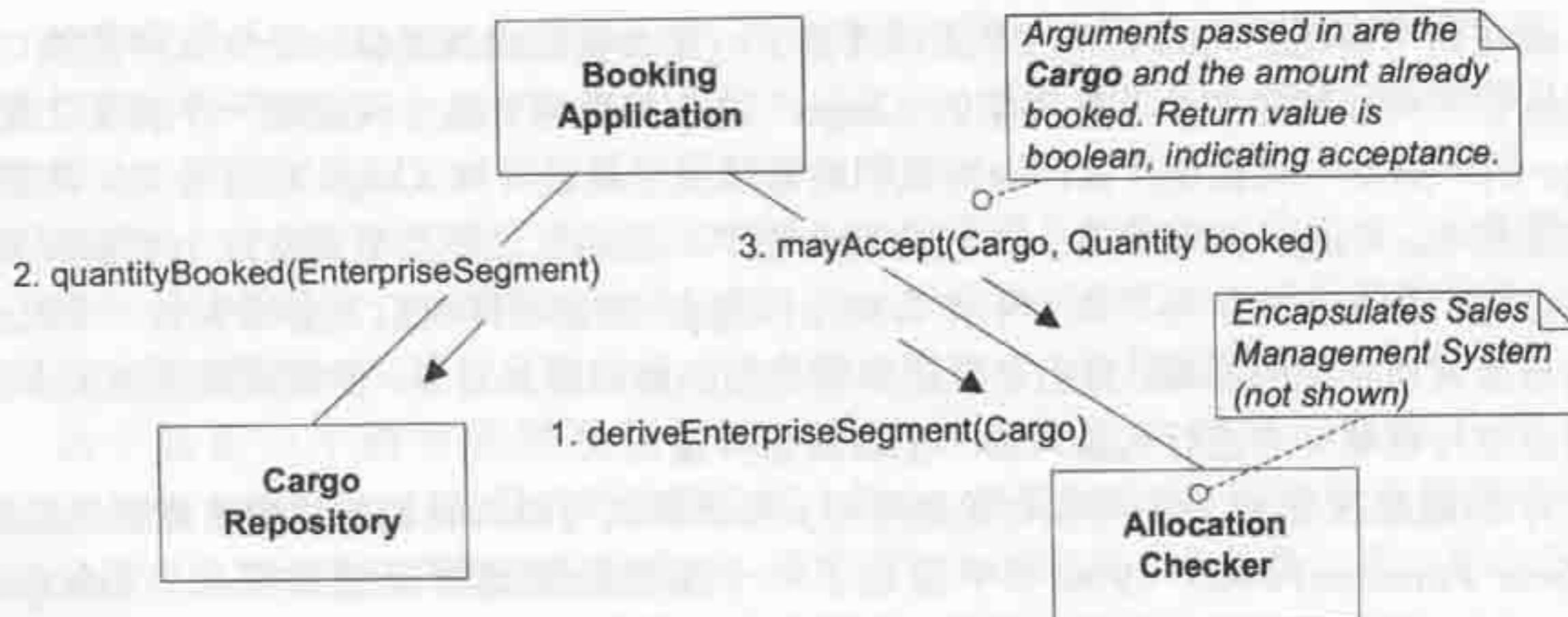


图 7-11 把 Booking Application 的领域职责转移到 Allocation Checker



7.11.3 性能调整

领域设计中其他部分惟一需要关心的就是 Allocation Checker 的接口。如果出现性能问题的话，我们就可以从这个接口的内部实现入手来解决它。例如，如果销售管理系统在另一个服务器(也许在另一个地点)上运行，那么系统之间的通信开销就会比较显著。每次检查配额时都需要交换两条消息：第一条消息从 Cargo 中得出 Enterprise Segment，第二条消息让销售管理系统回答“是否接受给定货物”这个基本问题。我们没有其他的方法来发送第二条消息；但是，与配额决策本身相比，第一条消息所使用的数据和行为是相对静态的。因此在设计中，我们可以选择把这些静态信息缓存到 Allocation Checker 所在的服务器上，这样就把消息交换的开销降低了一半。当然这种灵活性也是需要代价的，它使得设计更加复杂，而且我们必须设法保证缓存数据的同步更新。但是，如果性能对于分布式系统来说非常关键的话，那么这种灵活部署就会成为一种重要的设计目标。

7.12 小结

这种集成问题弄不好就会把我们原本简单、概念一致的设计搅得一团糟，但是现在，通过使用一个防腐层、一个服务，以及一些企业部门单元，我们把销售管理系统的功能清晰地集成到预订系统中来了，也丰富了领域。

最后还有一个设计问题：为什么不让 Cargo 来承担获取 Enterprise Segment 的职责呢？乍一看起来这样好像很优雅，如果 Enterprise Segment 的所有数据都是从 Cargo 中获得的，那么它就可以变成 Cargo 的一个派生属性了。遗憾的是，问题没有那么简单。Enterprise Segment 是可以任意定义的，只要它的划分方法对于业务策略有用就行。同一个实体可以根据不同的目的作不同的划分。出于预订配额的目的，我们从一个特定的 Cargo 中获取 Enterprise Segment，但是税务记账使用的 Enterprise Segment 可能会完全不同。即使是用于配额的 Enterprise Segment，也可能因为销售管理系统根据新的销售策略进行了重新配置而发生变化。所以 Cargo 必须知道 Allocation Checker，但是 Allocation Checker 与 Cargo 的概念职责完全不搭界。而且，每种特定类型的 Enterprise Segment 都需要用一个方法来获取，这也会使 Cargo 不堪重负。因此，适于承担获取 Enterprise Segment 这个职责的，应该是那些知道划分规则的对象，而不是包含那些规则所针对的数据的对象。我们可以把那些规则分离到一个单独的策略对象中，然后把它传给 Cargo，让 Cargo 把 Enterprise Segment 计算出来。使用策略看起来好像超出了我们需求的范围，但是在以后的设计中它可以成为一种选择，而且应该不需要设计做太大的修改。

