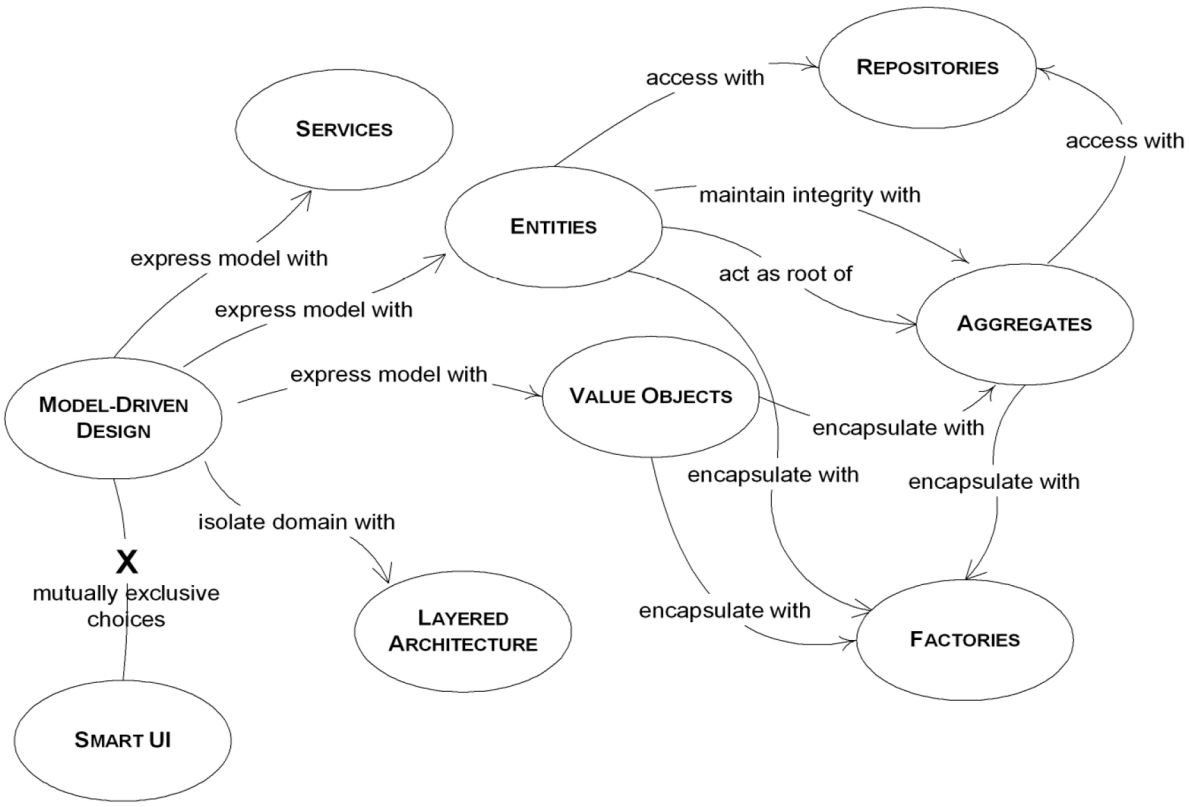
Eric Evans《领域驱动设计》总结之作

Domain-Driven Design Quickly

领域驱动设计精简版

全新修订



Abel Avran & Floyd Marinescu 著

孙向晖 霍泰稳 李锟 译

InfoQ 企业软件开发系列图书

C4Media 是 InfoQ.com 这一企业软件开发社区的出版商

本书属于 InfoQ 企业软件开发丛书

如果您打算订购 InfoQ 的图书，请联系 books@c4media.com

英文版责任编辑：Floyd Marinescu
英文版封面设计：Gene Steffanson
英文版美术编辑：Laura Brown 和 Melissa Tessier
向 Eric Evans 致以特别的谢意

中文版翻译：孙向晖　霍泰稳
中文版修订：李锟
中文版责任编辑：霍泰稳
中文版美术编辑：吴志民

欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译等，请联系 china-editorial@infoq.com 。

10 9 8 7 6 5 3 2 1

修订版序

笨蛋，问题是领域

“笨蛋，问题是经济！”（It's the economy, stupid!）

上面这句带有冒犯性的话，是比尔·克林顿在与老布什竞选美国总统时提出的竞选标语。事实证明，克林顿是美国近 30 年来最懂经济的总统，在他的任内，启动了著名的“信息高速公路”计划，美国的经济蒸蒸日上。所以即使他是一位花花太岁，一度声名狼藉，美国人民至今仍然很怀念他。

对于我们所从事的软件开发行业来说，如果有什么东西像是经济一样永恒，那么无疑就是领域（domain）了。在计算机发展的最初岁月，它主要是用来完成一些计算密集型的工作，例如核试验、天气预报中的庞大的计算工作。后来人们发现计算机很适合用来模拟人类世界很多领域中所存在的业务逻辑，把人们从繁重的重复性人工处理中解放出来，去做更有创造力和乐趣的工作。特别是在 20 世纪 80 年代以苹果电脑和 IBM PC 为代表的个人电脑的出现后，计算机迅速普及到了各行各业。

计算机之所以很有用，是因为通过使用计算机软件能够解决人们所从事的某个领域的问题。然而，假如软件的开发人员并没有深入理解软件所处的领域，软件就无法很好地解决该领域的问题，其实用价值就会大打折扣。因此对于业务软件的开发人员，应该时常提醒自己的一句话是：“笨蛋，问题是领域”。深入理解领域的开发人员，即使使用过时技术所开发出来的软件，也远远好过完全不理解领域的开发人员，使用最时髦技术所开发出来的软件。

遗憾的是，这个很明显的道理，在软件开发人员中并没有被普遍接受。在很多兴趣驱动的开发人员看来，深入理解领域中的业务逻辑，是一件枯燥乏味的事情。他首先会考虑：“我要在这个项目中使用苹果公司新推出的 Swift 编程语言，在服务器端要使用

Hadoop，最好再尝试一下深度学习方面的技术”，然后就一头扎进这些时髦和高大上的技术之中。三个月后，你去问他需要解决的领域中的真实问题是什么，他还是一脸茫然。

哥们，别王顾左右而言他，作为软件开发人员，很可能你也有这样的倾向。实话实说，在大约十年前，笔者也曾经是这样一位兴趣驱动的开发人员。当然，责任并不是完全在我身上。IT 行业，尤其是软件开发行业，是一个非常喜欢炒作新概念的行业。新的编程语言、新的开发框架层出不穷，让开发人员疲于跟随。以有涯之人生，去追随无涯的技术变迁，实在是一件很痛苦的事情。

如何跳出这样的恶性循环，获得人生的解脱呢？我们需要有某种方法论的指导。幸运的是，这种方法论已经有了，那就是领域驱动设计——DDD。DDD 就是我们做业务软件开发的指路明灯。它指导我们专注于最重要的问题——领域，并且为如何设计好的领域模型，以便更好地解决领域问题提出了很多具体的做法。

DDD 并不是突然冒出来的天生石猴，它其实是 30 年来业务软件开发（不同于科学计算类软件）实践经验的结晶。在笔者看来，它的起源甚至可以追溯到将近 40 年前的《人月神话》这本经典名著（你还没有读过？那你出门可不大好意思跟别人打招呼啊）。在《人月神话》中，布鲁克斯老先生将维护软件的“概念完整性”作为软件开发的核心问题。软件之所以很复杂、难以维护，根本原因就在于软件的概念完整性遭到了破坏，甚至开发团队的成员从来就没有意识到有必要去维护软件的概念完整性，他们并不是一个真正的团队，只是一些自行其事的开发人员，碰巧在一个团队中一起堆代码而已。

在项目开发最初的时候，他也有过一段狂欢般的快乐时光，不久之后，事情就越来越艰难。项目的代码越来越难以维护，工作越来越像是一种煎熬，合作的同事对他越来越不满。“该是与这个项目，与这个公司说 bye bye 的时候了”，他想。他换了一家公司，涨了一点工资，开始了另一段狂欢。周而复始，一年又一年过去了。随着年龄的增长，他不再能够从软件开发中享受到乐趣，软件开发的职业生涯，对于他而言痛苦多于快乐。学习新的技术，他也比不上年轻人。运气好的话，他可以完全脱离开发，转去做管理、做产品设计、做业务，运气不好的话，他只得继续做开发，因为毕竟还要养家糊口。“继续混着呗”，他悲观地想。他从一家公司换到另一家公司，从一次失败，走向另一次失败，他始终都是一个 loser。

上面这段描述，其实就是目前国内大多数软件开发人员的真实写照。对于缺乏学习能力的开发人员而言，从事软件开发的职业前景是灰暗的，最终他们会被这个行业所抛弃。想想看，国内目前的软件开发人员有几百万人，还会有越来越多的年轻人加入到这个行业，如果他们只能无奈地走上这样一条不归路，这可真是一个严重的社会问题。

在笔者所在的公司里面，有一个很重要的项目，最初的架构师（不止一个人）在错误的架构设计风潮的影响下，决意把这个项目设计成一个全分布式的系统。他们设计了十几个分布式服务，相互之间通过 WebService 通信。他们还引入了一个开源的 ESB（企业服务总线）中间件——Mule。项目开发的鼎盛时期，曾经有二十几个开发人员，每个人负责开发不同的分布式服务。因为架构师对这些程序员缺乏必要的指导和控制，每个程序员只熟悉自己负责的一小块工作，相互之间缺乏沟通和协作。这个项目的概念完整性很快就遭到了破坏，架构师所画的架构图，在代码实现中消失了。程序员很快就不再关心最初画的那些架构图，他们开始自行其事，只求尽快完成手头的工作早点下班。十几个分布式服务所导致的大量远程调用给系统的性能和可伸缩性带来了巨大损失，每一个远程调用都有可能出错，全部可能出现的异常情况难以计数，极大增加了异常处理的开发和测试工作量。同时给配管、运维人员的发布工作也带来了巨大的困难，每一次的上线发布都像是一次艰难的战役。Mule 这个中间件，并不适用于大流量互联网应用的环境，其性能和可伸缩性有很大的问题。最初引入 Mule 时，架构师甚至都没有想到应该提前做一些压力测试！项目的开发人员，离职的越来越多，也包括最初的架构师，最后只剩下了 4 个人。这么少的人很难维护十几个分布式服务，架构师只好决定将 Mule 完全废弃掉，将十几个分布式服务合并为三个相对独立的集中式应用，将远程调用改为了本地方法调用。这样做起码配管、运维人员的工作能够大幅减少。然而，这种合并仅仅是把代码简单地合在一起，原先因为存在十几个分布式应用，程序员自行其事，所导致概念完整性遭受的破坏，仍然完全没有解决，因此这个项目的代码仍然难以维护。这真是一个 design by buzzword 的典型案例，软件开发的很多反模式都可以在这个项目中找到。这个项目的开发人员流动性很大，一个重要原因是他们无法从这个项目中获得开发的乐趣。

那么，这个项目就完全无药可治了，只能等待寿终正寝了吗？答案是并非如此，只要下定决心，坚持正确的工作方法，在完善自动化验收测试的前提下，对项目代码进行持续的重构和改造，提炼出良好的领域模型，这个项目仍然有恢复健康的可能性。当然，改造工作的重要性和所需要投入的成本，也需要得到公司管理者的认可和支持。

代码的质量如果不加以控制，就一定会迅速腐烂变质。这是一个客观规律，就像在热力学第三定律中，熵总是会增加一样。对于软件开发而言，“概念完整性”就相当于热力学第三定律中的熵，是衡量软件项目混乱程度的重要指标。DDD 就是目前维护软件项目“概念完整性”的最佳良药，虽然永远不可能出现某种银弹式的技术，但是 DDD 能够很好地解决软件开发中的这个核心问题。DDD 与重构技术（参见 Martin Fowler 的《重构:改善既有代码的设计》）都有助于改善代码的概念完整性，但是它们有不同的定位。重构是在代码实现层面对抗腐烂变质，而 DDD 是在代码架构设计层面对抗腐烂

变质，两者是互补的关系。实践 DDD，能够控制好软件开发的复杂性，最终交付令用户感到满意的业务软件。

DDD 创造者 Eric Evans 大师的经典著作《领域驱动设计——软件核心复杂性应对之道》，对于初学者而言还是很抽象的。InfoQ 几年前出版的 *Domain-Driven Design Quickly*（dddquickly）是一本 DDD 方面非常棒的入门书。因为工作需要，笔者有强烈的愿望想要把 DDD 介绍给合作的同事们。然而，dddquickly 之前的中文版的质量不大令人满意。因此笔者决定在之前中文版的基础上，重新翻译 dddquickly 这本书。笔者的老朋友 ,dddquickly 之前中文版的译者 Kevin Huo 对这个工作给予了大力支持。笔者和 Kevin，还有 InfoQ 中文站的编辑马国耀、丁雪丰，都有强烈的愿望，希望 DDD 能在中国尽快普及，生根发芽、开花结果。这个工作很有意义，这是一次非常愉快的合作。建议读者从阅读 dddquickly 中文版入门，然后再去坚持读完 Evans 大师的原著，相信可以取得张无忌修炼九阳神功之后的效果。

正如对 Evans 的访谈中 Evans 所说的，对于业务开发人员而言，衡量一种技术好坏的最终标准，应该是这种技术是否有助于开发人员专注于研究领域问题。任何会导致开发人员分心的技术，哪怕这种技术再高大上，在项目开发中也应该坚决抛弃。学习新的技术很重要，但是专注于研究领域问题更为重要。只有更好地解决用户真正关心的领域问题，软件开发人员才能获得职业生涯的长青。

李锟

2014 年 11 月 8 日

译序一

在 2004 年之前的某一天，我和所在部门的一个设计师进行沟通，当时他为自己的一个思路兴奋不已，而我要做的事情就是跟他讨论清楚他头脑中的那个想法，然后写出需求和设计文件来。大家可能会注意到，很多时候，需求是从设计中反推出来的，这被一些专家称为“需求的反向工程”。其实我更多地认为这是由于我们现在糟糕的工作现状决定的，有诸多的因素导致需求或者设计被局限在仅有的几个人的知识体系中，但如果有心去细察，会发现他们各自的理解又各不相同。

回到刚才说到的那次沟通上，当那个设计师把自己的得意之作描述完毕后，我在纸上用 UML 图画出了他的主题思路，然后我们针对细节开始探讨并在图上改改画画。最后的修订结果显示，他的很多“创举”是多余的，经过精简后的 UML 基本上颠覆了他原有的思路。现在我还记得那位同事的一声叹息：“一周的功夫白费了……”

其实在整件事情中，他有他的得失，我也有我的收获和困惑：我意识到对统一的核心模型进行探讨和简化的重要性，但应该如何把这样的过程程式化，让它在更多的同事的工作中发挥作用呢？

这样的问题纠缠了我好久，终于有一天，我得到了一本如字典般的硬皮 *Domain Driven Design*（我们亲切地称它为“DDD”）原版书，从中找到了答案。然后，我参与了 DDD 中文版的审校工作和 DDD 注释版的注释工作。再后来，InfoQ 中文站的总编霍泰稳又邀请我一起做了 *Domain Driven Design Quickly* 这本书的翻译工作。

曾经有人要我用简练的词汇描述 DDD 的中心思想，我个人认为这是一个比较难的工作，但我愿意去尝试。我的回答是“关注精简的业务模型及实现的匹配”。

1. 如果你了解“模型”的定义是对现实的有选择性的精简，然后用这样的观点去读 DDD 这本书，你就会发现，DDD 其实没有什么太多的新鲜玩意，它更多地是可以

看作是面向对象思潮的回归和升华。在一个“万事万物皆对象”的世界里，哪些对象是对我们的系统有用的？哪些是对我们拟建系统没有用处的？我们应该如何保证我们选取的模型对象恰好够用？

2. 前面的选择性问题只是解决了一个初步框选的问题，对象并不是独立存在的，它们之间有着千丝万缕的联系。这种扯不断理还乱的联系构成了系统的复杂性。一个具体的体现就是，我们修改了一处变更，结果引发了一系列的连锁反应。虽然对象的封装机制可以帮我们解决一部分问题，但那只是有限的一部分。我们应该如何在一个更高点的层次上，通过保留对象之间有用的关系去除无用的关系，并且限定变更影响的范围以来降低系统的复杂度呢？

3. 在 DDD 以及传统 OO 的观点中 ,业务而不是技术是一个开发团队首先要关注的内容，众多的框架和平台产品也在宣称把开发人员解放出来，让他们有更多的精力去关注业务。但是，当我们真正去看待时，会发现，开发人员大多还是沉溺于技术中，对业务的理解和深入付出的太少太少。其实要解决这个问题，就要先看清楚我们提炼出来的模型,在整个架构和整个开发过程中所处的位置和地位。我们经常听到两个词，一个是 MDD(模型驱动设计),一个是 MDA(模型驱动架构)。如果 DDD 特别关注的是“M”(以及其实现)，那么，这个 M 应该如何与架构和开发过程相融合呢？我经常会看到我们辛苦提取出来的领域模型被肢解后，分散到系统的若干角落。这真是一件可怕的事情，因为一旦形成了"人脑拼图" ,就很难再有一个人将它们一一复原 ,除非这个人是个天才。

4. 很多面向对象的教材，都会告诉你若干的技巧，让你去机械化地处理模型和对象实现，但是这些教材通常会忽略一个大的上下文环境，就是应该由哪些具有什么样素质和技能的人来处理模型和对象实现,或者说白了 ,就是应该用什么样的团队模型来匹配业务模型。不同的模型，需要不同的团队模型的支撑，不同的团队模型也会让一个模型实现更优秀或者更糟糕。

5. 相信很多人读过 ATM 机的例子，你发现自己彻底明白了用例应该怎么编写、模型怎么提取和实现，但是当你信心十足地去开始你自己的项目时，你又会发现你的思路片段化了，所有你明白了的技能在你的新项目中好像用不上。算了，还是老的工作思路和工作方式比较顺手，于是一切都照旧会到了老的套路上。那么，面向对象技术或者说我们提炼出来的模型应该如何在大型项目/团队中使用呢？我们是应该要求一个项目使用统一的模型，还是应该把它们分成不同的模型？我们应该如何抉择？

……

在实际的应用过程中，我相信每一个有心人都会提出比我在这里列举出来的还要多的问题，没有关系，DDD 这本书都能给你所需的答案。

当然，“DDD”作为传统 OO 范型的探索和升华版，也存在着一些不足和未涉及的领域，例如：在安全、权限方面的考虑，其基本构造块的适用范围和决策标准，与开发框架之间更好的融合性等问题。还好，我们都知道"尽信书不如无书"的道理，在阅读任何著作时，都应该带着自己的疑惑和批判精神去阅读，你的任何关于 DDD 的深层思考和讨论，都可以在它的配套网站或者 InfoQ 中文站网站中发表。

如果你认为阅读 DDD 这么一本如字典版的大厚书时间上不太允许，那么请允许我为你介绍一本简化版的小书 *Domain Driven Design Quickly* ,经过我们的努力 ,InfoQ 中文站已经将其翻译成中文版——《领域驱动设计精简版》，并作为国庆礼物奉献给大家！

那还等什么呢，祝你开卷有益，阅读愉快！

孙向晖

2007 年 9 月 26 日于泰安

译序二

Domain Driven Design 这本书的中文版已经由清华大学出版社在 2006 年出版，人民邮电出版社图灵出版公司新近即将出版这本书的中文注释版。据我了解该书得到了读者的认可，由此可见领域驱动设计这一概念正在或者已经得到了国内开发者社区的认可。

关于领域驱动设计的重要性，本书作者之一 Floyd Marinescu 和中文版译者之一孙向晖兄弟都已经做了详细的阐述，我就不在这儿画蛇添足了。我相信通过这本迷你书的介绍，读者会在短时间对领域驱动有一个更加清晰的认识。

在参与 InfoQ 中文站前期筹备工作的时候，Floyd 就告诉我 *Domain Driven Design Quickly* 这本书在 InfoQ.com 网站上特别受欢迎，下载量已经超过了 3 万。也正是这个信息，让我萌生了在 InfoQ 中文站上首先发布本书的想法，让中文社区的开发人员也能早日领略领域驱动设计的精彩，普及这一概念。

很幸运的是，我邀请到了对领域驱动很有研究也很有兴趣的孙向晖（网名“豆豆他爹”）参与到本书的翻译中来。在领域驱动设计方面，向晖曾参与了 *Domain Driven Design* 中文版的审校和 *Domain Driven Design* 注释版的注释工作，这让我对这本书的翻译质量有了更多地信心，在我对本书的审校过程中也确实证明了这一点。向晖是我多年的好友，在从前去浪潮软件“楼上”平台采访时，我们一起登临了泰山，他丰富的从业经历和对朋友的热情，都给我留下深刻的印象。与他合作，我非常愉快！

因为 Floyd 和向晖对这本书的推荐，我对领域驱动设计的好感也与日俱增。恰好在向晖翻译的过程中，公司对他的工作有了新的安排，时间上一时忙不过来，由我接手了本书下半部分的翻译工作。感谢向晖前面的精彩翻译，在我通读了之后，对 Domain Driven Design 概念理解的更加清晰，也庆幸原书行文的流畅，让我得以顺利完成相关

内容的翻译。所以本书的 1~4 章为孙向晖翻译，5~6 章为我翻译，然后我们又互相进行了审校。

尽管在本书的翻译和后期制作过程中，我们小心又小心，但肯定还有许多不足的地方，欢迎读者指正并反馈给我们，以备在我们收集后对译作进行修正，让以后的朋友读到更完美的作品。意见可以直接反馈在 InfoQ 中文站关于本书的网页评论中，或者邮件至 kevin@infoq.com。

最后感谢女友志民对我工作的支持，我现在所工作的工作台、座椅，包括用于制作本书的笔记本都是她陪同我购买的。尤其感谢在我沉浸于工作无暇陪伴她时，她所给予我的耐心和嗔怪。十一期间，我们将完成从“恋人”到“爱人”的转变，所以我也想借用本书献上我对她的爱。

霍泰稳

2007 年 9 月 26 日于北京花家地

前言：编者按

我第一次听说领域驱动设计并认识 Eric Evans 是在 2005 年夏天由 Bruce Eckel 组织的一次小型架构师的顶级聚会上。参加聚会的很多人都是我非常尊敬的，包括 Martin Fowler、Rod Johnson、Cameron Purdy、Randy Stafford 和 Gregor Hohpe 等。

这个小组对领域驱动设计的愿景好像都很感兴趣。我也有这样一种感觉，所有的人都希望这些概念能更为主流一些。当我注意到 Eric 如何使用领域模型来解释以前小组讨论过的各种技术挑战的解决方案，以及他对业务领域加以特别强调，而对特定于技术的虚假宣传却不为所动，我猛然意识到他所说的这个愿景正是社区特别需要的东西。

我们在企业开发社区中，尤其是 Web 开发社区，被多年的虚假宣传所扰乱，因而偏离了正确的面向对象软件开发方向。在 Java 社区，优秀的领域建模也在 1999 年到 2004 年间的 EJB 和容器/组件模型虚假宣传下迷失了方向。幸运的是，技术的变迁和软件开发社区积累的经验正推动着我们回到传统的面向对象设计方法。然而，社区对于如何在企业规模应用面向对象缺乏清晰的愿景，这就是为什么我认为 DDD 很重要的原因。

不幸的是，除了这些少数顶尖架构师外，我看到很少有人理解 DDD，这也是为什么 InfoQ 委派我来编写这本书的原因。

发布这样一本书，针对 DDD 的基础知识进行简短且易于快速阅读的概括和介绍，读者可以在 InfoQ 网站上免费下载电子版，或者购买便宜的印刷口袋书。我希望通过这样做，这个愿景能够为更多的人所接受。

本书没有介绍任何新的概念，它只是简明地概括了 DDD 的基本知识，内容大多来自于 Eric Evans 的《领域驱动设计 软件核心复杂性应对之道》英文名 *Domain-Driven Design: Tackling Complexity in the Heart of Software*）原书中关于这一主题的内容，以及其他相关资料，包括已经出版的书籍，例如 Jimmy Nilsson 的《领域驱动设计与

模式实战》（英文名 *Applying DDD*）和各种领域驱动设计讨论群组中的内容等。本书可以让你快速了解 DDD 的基础知识，但无法替代 Eric 书中提供的大量实例和案例研究或者 Jimmy 书中提供的动手实例等。我非常鼓励大家去阅读这两本卓越的书籍。同时，如果你也认同 DDD 这一概念需要成为社区关注的重点，那么请让更多的人知道本书和 Eric 的工作！

InfoQ.com 共同创始人和总编
Floyd Marinescu

目 录

简　介

软件是一种被创建用来帮助我们处理现代生活中复杂问题的工具，它只是到达目的的一种方法，而这个目的通常就是非常实际和真实的事情。例如，我们使用软件控制空中交通，这就和我们日常的生活有直接的联系。我们想从一个地方飞到另一个地方，要通过使用复杂的机器达到目的，于是我们制造软件来协调那些在空中飞行的数以千计的飞机。

软件必须是实际和有用的，否则我们不会花那么多时间和资源去创建它。这就使它和我们生活的某个方面有非常密切的联系。一个有用的软件包不能和现实的范围（也就是假定能帮助我们管理的领域）分割开来。相反，它们应该紧密相连！

软件设计是一门艺术，像其他艺术一样，无法通过定理和公式以一门精确科学的方式来教授和学习软件设计。我们能够发现可用于整个软件创建过程的一些很有用的规律和技巧，然而我们也许永远不能提供一个精确的路径，能够将现实世界直接映射到满足其需求的代码模块。如同一幅画或者一个建筑，软件产品中包括了设计和开发它的那些人的个性化感觉（personal touch），也就是促成软件的启动开发和成长这些人（富有或缺乏）的才能和天分（charisma and flair）。

完成软件设计的方法多种多样。在过去的 20 年中，软件产业发现和使用了多种方法来创建软件产品，每一种方法都有自己的优点和不足之处。本书的目的是专注于介绍一个出现和发展了 20 多年但近几年才开花结果的设计方法：领域驱动设计。Eric Evans 通过撰写一本领域驱动设计相关经验的书籍，在这个话题上做了大量的工作。关于这个主题的更详细讨论，我们推荐阅读他的的《领域驱动设计：软件核心复杂性应对之道 》（*Domain-Driven Design: Tackling Complexity in the Heart of Software*）一书，由 Addison-Wesley 出版。

通过参与领域驱动设计的讨论组也可以学习到很多有价值的观点：

http://groups.yahoo.com/group/domaindrivendesign

这本书只是对领域驱动设计话题的介绍，希望能让你快速了解它的基础知识，而不是相关的详尽理解。我们希望通过介绍领域驱动设计世界中所使用的原理和指导，激发出你对优秀软件设计的探索欲望！

第 1 章

何为领域驱动设计

软件开发通常被用于将真实世界中已经存在的流程自动化，或者为真实的业务问题提供解决方案。需要自动化的业务流程或者真实世界的问题，就是软件的领域。从一开始，我们就必须明白软件起源于其领域，并且与其领域密切相关。

软件是由代码最终构成的。我们可能会被代码所诱惑，在它上面花费了太多的时间，因此将软件看作是简单的对象或者方法。

我们可以以汽车制造来做类比。参与汽车制造的工人会专门负责汽车的某个部件，但这样做的后果是工人们通常对整体的汽车制造流程缺乏了解。他们可能将汽车视为一大堆需要固定在一起的零件的集合体，但一辆汽车的意义远不只于此。一辆好车起源于一个好的创意，开始于认真制定的规格说明，然后再交付给设计。经历若干道设计工序，花费上几个月甚至几年时间去设计、修改、精化直至达到完美、能够完全反映出最初的愿景。设计的过程也不全然是在纸上进行的。设计工作的很大一部分包括了建造汽车的模型（doing models of the car）、并且在特定条件下对它进行测试，以验证该模型是否能工作。设计会根据测试的结果做出修改。汽车最终被交付到生产线上，在那里，所有的部件已经就绪，然后被组装到一起。

软件开发也是一样。我们不能直接坐下来敲代码。当然在开发价值不大的软件时，也可以这样做，但我们不能用这种方法开发复杂的软件。

为了创建一个好软件，你必须知道这个软件究竟是什么。在你充分了解金融业务是什么之前，你是做不出一个好的银行业软件系统的，你必须理解银行业的领域。

没有丰富的领域知识能做出复杂的银行业业务软件吗？没门。答案永远是否定的。那么谁了解银行业业务？软件架构师吗？不，他只是在使用银行来保护他的财产安全，并且确保需要钱的时候能够取出来；软件分析师吗？也不是，他只懂得在已获取到所有材料的情况下，对一个给定的主题进行分析；软件开发人员？别难为他了。那么还有谁？当然是银行的从业者了。银行业务系统被银行的内部人员和专家所熟知。他们知道所有的细节、所有的困难、所有可能出现的问题、所有的业务规则。这些就是我们永远的起始点：**领域**。

在启动一个软件项目时，我们应该关注软件涉及的领域。软件的最终目的是增进一个特定的领域。为了达到这个目的，软件需要跟要它服务的领域和谐相处，否则，它就会给领域引入束缚，引起故障、造成破坏甚至导致很大的混乱。

我们怎样才能让软件和领域和谐相处呢？最佳方式是让软件成为领域的一个映射。软件需要包含领域里重要的核心概念和元素，并精确实现它们之间的关系。也就是说，软件需要对领域进行建模。

对银行业业务不了解的人，应该能够通过阅读该领域模型相关的代码学习到大量知识。这是必不可少的。随着时间的推移，没有深深扎根于领域的软件将无法很好地应对变化。

所以我们从领域开始着手。接下来要做什么呢？领域是真实世界中某些事物，不要企图能轻而易举地捕获它们，以为敲几下键盘就能出来代码。我们需要为该领域创建一个抽象。当我们跟领域专家交流时，我们会学到很多领域知识，但这些未加工的知识难以被转换成软件构造，除非我们为它建造一个抽象——在脑海中的一个蓝图。开始时，这个蓝图总是不完整的，但随着时间的推移，经过不断的努力，我们会让它越来越好，让它看上去越来越清晰。这个抽象是什么？它是一个模型，一个关于领域的模型。按照 Eric Evans 的观点，领域模型不是一幅具体的图，它是那幅图想要去传达的那个思想。它也不是一个领域专家头脑中的知识，而是一个经过严格组织并进行选择性抽象的知识。一幅图能够描绘和传达一个模型，同样，经过精心编写的代码和一段英语句子都能达到这个目的。

模型是我们对目标领域的内部展现方式，它是非常必须的，会贯穿设计和开发的全过程。在设计过程中，我们会记住并大量引用模型中的内容。我们周遭的世界中有着太多的内容等待着我们的大脑去处理。甚至一个特定的领域所包含的内容都远远超出了人脑一次可以处理的范围。我们需要组织信息，将其系统化，把它分割成小一点的信息块，将这些信息块分类放到逻辑模块中，每次只处理其中的一个逻辑模块。我们需要忽略领域中的很多部分，因为领域包含了如此之多的信息，不能一下子放到一个模型中。而且，它们当中的很多部分我们也不必去考虑。这对它自身而言也是一个挑战：要保留哪些内容放弃哪些内容呢？这些取舍是设计和软件创建过程的一部分。银行业软件肯定会跟踪客户的住址，但它决不会关心客户的眼睛是什么颜色的。这是个很明显的例子，但其他的例子可能不会如此明显。

模型是软件设计中最基本的部分。我们需要它，是因为能够用它来处理复杂问题。我们对领域的所有的思考过程被汇总到这个模型中。这样甚好，但它必须要脱离开我们的头脑，如果它始终停留在我们的头脑中，其实并不会有多大的作用，不是吗？我们需要就这个模型跟领域专家进行交流，跟相关的设计人员进行交流，跟开发人员进行交流。模型是软件的根本，但我们需要找到一些方法来表达它，与其他人交流这个模型。在这个过程中我们并不是孤立的，所以我们需要彼此共享知识和信息，而且我们需要把它做得更好、更精确、更完整，没有二义性。要做到这点有多种方式。一种方式是图形化的：图形、用例、绘画、图片等。另一种方式是文字描述，我们会写下我们对领域的愿景。还有一种方式是使用语言，我们能够也应该创建一种语言，来对与领域相关的特定问题进行交流。在以后的章节中我们会详细讲解它们，但要点就是，**我们需要用模型来交流**。

当我们完成了模型的表达时，我们可以开始做代码的设计。这跟软件设计有很大的不同。软件设计类似于创建房子的架构，那是跟一个总图相关的。从另一个方面讲，代码设计是非常

细节性的工作，类似于在一面墙上定位一幅绘画作品。代码设计也是非常重要的，但它却不象软件设计那样基础。一个代码设计中的错误通常更容易修正，但要想修复软件设计中的错误需要更多的成本。要想让一幅油画向左边移动非常简单，而想要拆除房子的一个边却是一个完全不同性质的事情。话虽这么说，缺少了良好的代码设计，最终产品肯定好不到哪儿去。代码的设计模式唾手可得，当需要时，它们应该被很好地应用。优良的编码技巧可以帮助我们建立清晰的、可维护的代码。

软件设计有不同的方法，其中之一是瀑布设计方法。这种方法包含了一些阶段。业务专家提出一堆需求同业务分析人员进行交流，分析人员基于那些需求来创建一个模型，然后将建模的结果传递给开发人员，开发人员根据他们收到的东西开始编码。在这个方法中，知识只有单一的流向。虽然这种方法作为软件设计的一个传统方法，这么多年来已经取得了某种程度上的成功应用，但它还是有它的缺点和局限。主要问题是业务专家得不到分析人员的反馈信息，分析人员也得不到开发人员的反馈信息。

另一种方法是敏捷方法学，例如极限编程（XP）。这些方法学是不同于瀑布方法的一些活动，其产生的原因是很难预先确定所有的需求，特别是在需求经常变化的情况。要想预先创建一个覆盖领域所有方面的完整模型确实非常困难。需要做出大量的思考，而且在初期常常无法看到涉及到的所有的问题，也无法预见到设计中某些带有负面影响或错误的部分。敏捷方法试图解决的另一个问题被称为“分析瘫痪”，团队成员会因为害怕做出任何设计决定而止步不前。尽管敏捷方法的倡导者承认设计决定的重要性，但他们反对做出预先设计。相反，他们借助于实现层面的灵活性，通过由业务涉众持续参与的迭代开发和很多重构，开发团队更多地学习到了客户的领域知识，从而能够产出满足客户需要的软件。

敏捷方法也存在自己的问题和局限：他们提倡简单，但每个人都对“简单”的含义有着自己的观点。同时，缺乏了真实可见的设计原则，由开发人员完成的持续重构会导致代码更难理解或者更难改变。虽然瀑布方法可能会导致过度工程，但对过度工程的担心可能会带来另一种担心：害怕做出深度、彻底的设计。

本书描述了领域驱动设计的原则，在任何开发过程中应用这些原则，开发团队以一种可维护的方式对领域内复杂问题进行建模和实现的能力，都将会得到极大提升。领域驱动设计结合了设计和开发实践，展示了设计和开发如何协同工作以创建一个更好的解决方案。优良的设计会加速开发的过程，而开发过程中的反馈也会进一步优化设计。

构建领域知识

让我们考虑一个飞机飞行控制系统项目的例子，看领域知识是如何被构建的。

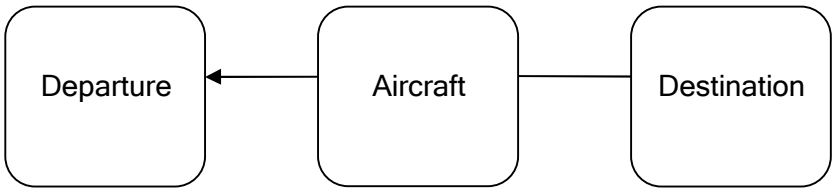
在一个给定的时刻，空中四处会有成千上万的飞机。它们会朝着各自的目的地按照自己的路线飞行，很重要的事情是要确保它们不会在空中发生碰撞。我们不会试图详细描述一个完整

的交通控制系统，而只会关注其中的一个小小的子集：飞行监控系统。这里要讨论的项目是一个监控系统，它会跟踪在指定区域内的任意航班，判断班机是否遵照了预定的航线，以及是否有可能发生碰撞。

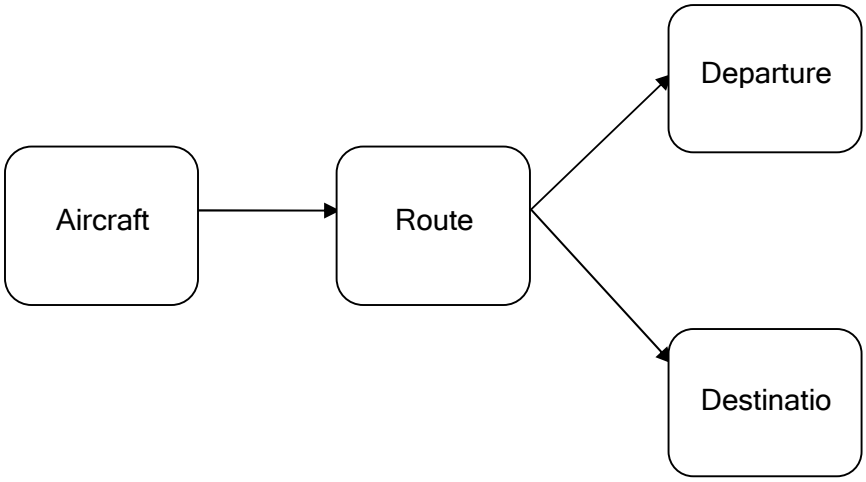
按照一个软件开发的视角，我们应该从何处开始呢？在前面的章节，我们说过会从理解领域开始，在本例中就是从空中交通监控开始。空中交通控制人员是这个领域内的专家。但是控制人员不是系统的设计人员或者软件的专家，你不能期望他们会给你提供一个关于他们问题域的完整描述。

空中交通控制人员对他们的领域拥有广博的知识，但为了能构建模型你需要提取出基本信息（essential information）并将其通用化。当你开始跟他们讨论时，你会听到很多关于飞机起飞、着陆、半空中的飞机和碰撞的危险、飞机等待允许着陆等知识。为了找到看似杂乱无章的信息中的规律，我们需要从某个地方开始。

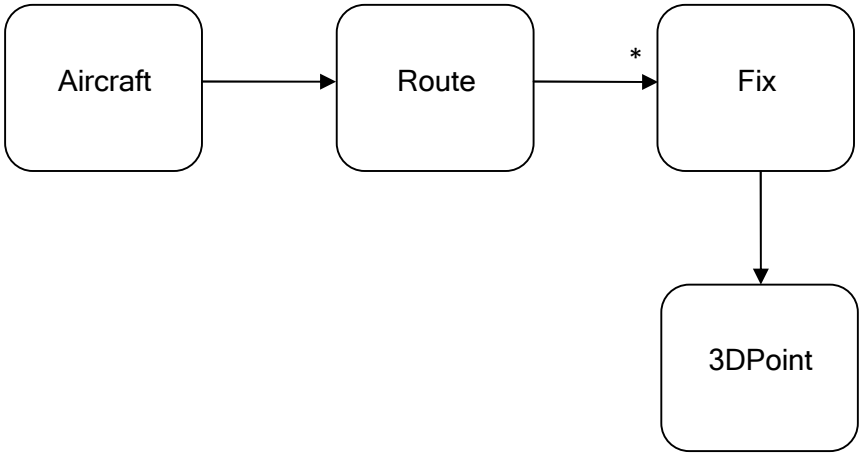
控制人员和你自己都认同每一个飞行器有一个出发机场和目的机场。所以我们找到了“飞行器”、“起始机场”和“目的机场”，见下图。



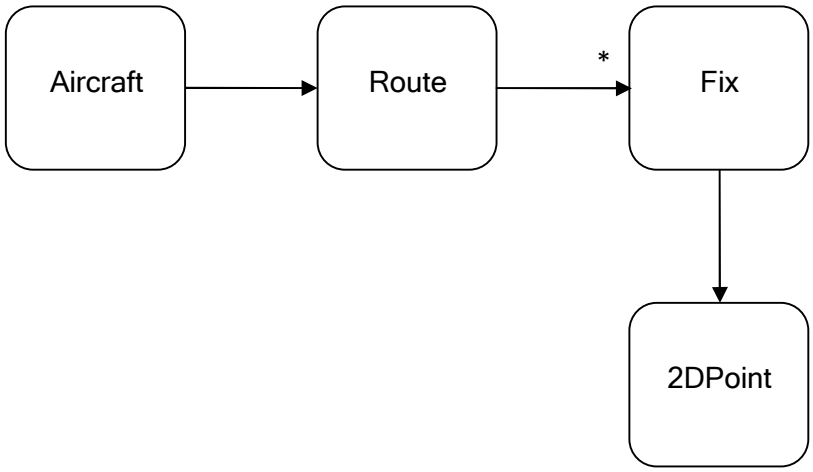
那么，飞机从某地起飞又在另一地降落。但在空中发生了什么？班机会按照什么路线飞行？事实上，我们更关心的是它在飞行时所发生的事情。控制人员说，他们会给每架飞机指派一个飞行计划，在飞行计划中应该描述空中飞行的全过程。当听到“飞行计划”时，你可能会在脑海中想到这是一个飞机在空中飞行时必须遵循的路径。在后边的讨论中，你听到了一个有趣的单词：路线（route）。这个单词当之无愧地立即引起了你的注意。路线中包含了飞机飞行中的一个重要概念，那就是飞机在飞行时所做的事：它们必须遵循一条路线。很明显飞行器的出发地和目标点也就是路线的开始和结束点。所以，与其将飞行器与出发地和目标点关联起来，不如将它与“路线”关联起来，这样看起来会更自然。然后再将路线与适当的出发地和目的地关联起来。



跟控制人员交流飞机需要遵循的路线时，你会发现路线是由小的区间段组成，这些区间段按照一定的次序组织起来，就会构成从出发地到目的地的一条曲线。这条曲线应该穿过预定的一些方位点。所以，路线可以被考虑成一系列连续的方位点。从这个角度看，你不会再将出发地和目的地看作是路线的结束点，而只是将它们看作是那些方位点中的两个点。这跟从控制人员的角度来看可能有很大的不同，但这是一个必要的抽象，对后续的工作会产生帮助。根据这些发现，导致模型所产生的变化如下：



这副图显示了另外一个元素，事实上路线中需要遵照的每个方位点是在空间中的一个点，它表现为一个 3 维的点。但是当你跟控制人员交谈时，你会发现他并不按这种方式思考。实际上，他将路线看作是飞机航班在地球上的映射。方位点只是地球表面上的点，可以由经度和纬度来决定。所以正确的图是：



实际上发生了什么呢？你和领域专家在交谈，你们在交换知识。你开始问问题，然后他们回答。当他们这样做时，他们从空中交通领域挖掘出基本概念。那些概念可能看上去未经雕琢、没有经过组织，但它们对于理解领域来说却是必不可少的。你需要从专家那里尽可能多地学习领域知识。通过提出正确的问题，正确地处理得到的信息，你和专家开始勾勒出领域的视图，也就是领域模型。这个视图既不完整也不能保证是正确的，但它却是你需要的开始点，你应该尽力提炼出领域的那些基本概念。

这是设计中很重要的一点。通常，软件架构师或开发人员都会和领域专家之间进行很长时间的讨论。软件专家希望从领域专家那里获取到知识，他们也必须将这些知识转换成有用的形式。在某个时刻，他们可能想要建造一个早期的原型，以验证它是否能按照预期工作。在这个过程中，他们可能会发现自己的模型或解决方法中的某些问题，可能想要修改模型。交流不再是从领域专家到软件架构师再到更后面的开发人员的单向关系，它存在着反馈，这会帮助我们创建一个更好的模型，获得更清晰更准确的对领域的理解。领域专家掌握很多专业技能，然而他们是按照特殊的方式来组织和使用他们的知识，这对于将这些知识实现为软件系统而言，并不总是一件好事情。

通过与领域专家的交谈，软件设计人员的分析型思维（analytical mind）会帮助他们挖掘出领域中的一些关键概念，并且帮助构建出可用于将来讨论中的结构，我们将在下一章中看到这种结构。作为软件方面的专家（软件架构师和开发人员）和领域专家，我们会在一起创建领域的模型，这个模型会体现两个专业领域的交汇。这看上去是个很消耗时间的过程，并且确实如此，但是我们原本就应该这样做，因为软件的最终目的是去解决真实领域中的业务问题，所以它必须和领域完美结合。

第 2 章

通用语言

对公共语言的需要

通过前一章的案例，我们认识到由软件专家和领域专家通力合作开发出一个领域模型是绝对必要的，但是，那种解决方法通常会由于一些根本上的交流的障碍而困难重重。开发人员满脑子都是类、方法、算法、模式，总是想将实际生活中的概念和程序中的工件做对应。他们希望看到要建立哪些对象类，要如何对对象类之间的关系建模。他们会按照继承、多态、面向对象的编程等方式去思考，会随时随地这样交谈，这对他们来说这太正常不过了，开发人员就是开发人员，但是领域专家通常对这一无所知。他们对软件类库、框架、持久化甚至数据库没有什么概念。他们只了解他们特定领域的专业知识。

在空中交通监控的例子中，领域专家知道飞机、路线、海拔、经度、纬度，知道飞机偏离了正常路线，知道飞机的飞行轨道。他们用他们自己的术语讨论这些事情，对于外行来说有时候很不容易理解。

为克服这种交流方式的不同，在建立模型时，我们必须通过沟通来交换对模型和模型中所包括的元素的想法，应该如何连接它们，哪些是有关的，哪些不是？在这种层次上的交流对一个成功的项目而言是极为重要的。如果一个人说了什么事情，其他的人不能理解，或者更糟的是错误理解成其他事情，又有什么机会来保证项目成功呢？

当团队成员不能共享一个公共的语言来讨论领域时，项目会面临严重的问题。领域专家使用自己的行话，而技术团队成员在设计的过程中，也使用自己（为讨论领域而调整过）的语言。

代码是一个软件项目中最重要的产物，但每天用来讨论的术语却与代码中使用的术语脱节了。即使是同一个人都需要使用不同的语言来交谈和书写，而对领域的最深刻的表达通常需要使用一种暂时的形式（transient form），但这种形式不会出现在代码甚至是书写的内容中。

在交流的过程中，需要做翻译才能让其他的人理解这些概念。开发人员可能会努力使用外行人的语言来解释一些设计模式，但这并不一定都能成功奏效。领域专家也可能

会创建一种新的行话以努力表达他们的一些想法。在这个痛苦的交流过程中，这种类型的翻译并不能对知识的构建过程产生帮助。

在设计过程中，我们倾向于使用自己的方言，但是没有一种方言能成为一种公共的语言，因为它们都不能满足所有的需要。

在讨论模型和定义模型时，我们确实需要讲同一种语言。那么是哪种语言呢？开发人员的语言？领域专家的语言？介乎两者之间的语言？

领域驱动设计的一个核心的原则是使用一种基于模型的语言。因为模型是软件满足领域的共同点，它很适合作为这种通用语言的构造基础。

使用模型作为语言的主干。要求团队在进行所有的交流时都使用一致的语言，在代码中也是这样。在共享知识和推敲模型时，团队会使用演讲、文字和图形。需要确保团队使用的语言在所有的交流形式中看上去都是一致的。因为这个原因，这种语言被称为“通用语言”（Ubiquitous Language）。

通用语言连接起设计中的所有的部分，为设计团队今后顺利开展工作建立了前提。完成一个大规模项目的设计可能需要花费数周乃至数月的时间。团队成员会发现一些初始的概念是不正确的或者不合时宜，或者发现一些需要考虑并放进总体设计中的新的设计元素。没有一种公共的语言，所有的这一切都是不可能的。

这种语言的形成可不是一日之功，需要开展艰难的工作，重点在于确保发现语言的那些关键元素。我们需要发现定义领域和模型的那些关键概念，发现描述它们的适当用词，并开始使用它们。它们当中的一些概念可能很容易被发现，但另一些则不然。

通过尝试反映其他可选模型的其他的表述方式，可以消除这个难点。然后重构代码、重命名类、方法和模型以适应新的模型。使用我们能够正常理解的普通词汇，化解交谈所使用术语之间的混乱。

构建一个类似这样的语言会得到一个清晰的结果：模型和语言相互密切关联。一个对语言的变更会变成对模型的变更。

领域专家会反对用那些很笨拙的或者不适当的字眼或者结构来传达对领域的理解。如果领域专家不能理解模型或者语言中的某种内容，那么就如同是说这种内容存在某种错误。从另一方面讲，开发人员应该留意那些与他们试图呈现在设计中的内容存在二义性或者不一致的部分。

创建通用语言

我们应该如何开始去构建一种语言？看一个假想的软件开发人员和领域专家之间关于空中交通监控项目的对话吧。需要留意粗体的那些词。

开发人员：我们想监控空中交通，应该从哪里开始？

专家：让我们从最基础的开始吧。所有的交通由飞机组成。每架飞机从一个**出发地**起飞，并在一个**目的地点**着陆。

开发人员：很容易嘛。在飞行时，飞机会按照驾驶员的意愿选择任何空中线路吗？是不是等于说他们可以决定他们能走哪条路，只要他们能到达终点？

专家：哦不。驾驶员会收到一条他们应该遵照的**飞行路线**。并且他们必须尽可能地跟那条飞行路线吻合。

开发人员：我会把这条**路线**考虑成空中的 3D 线路。如果我们使用笛卡尔系统坐标，那么一条飞行路线会被简化成一系列 3D 的点。

专家：我可不这么认为。我们不会这样看待**飞行路线**的。**飞行路线**实际上是飞机预期的空中线路在地面上的映射。**飞行路线**会穿过一系列地面上的点，而这些点我们可以用经度和纬度来决定。

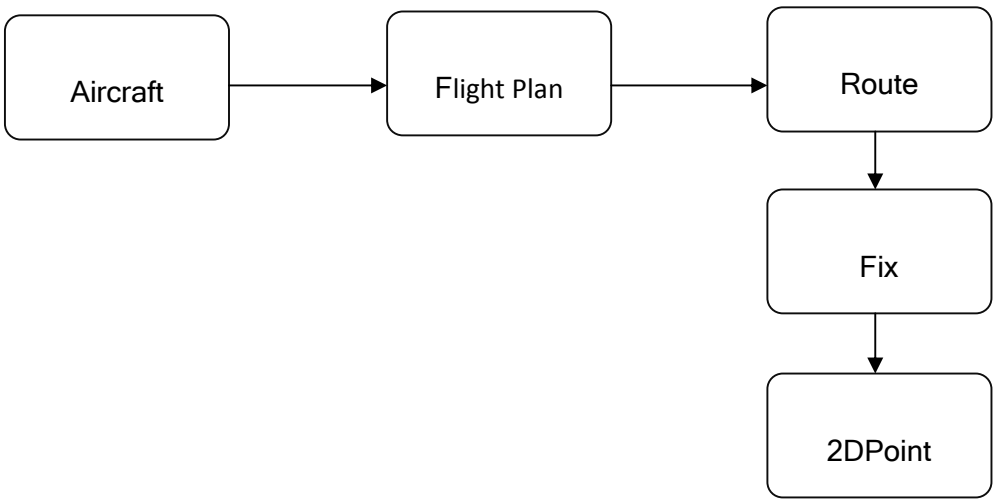
开发人员：哦，那我们可以称每一个这样的点为一个**方位**，因为它是地球表面上的一个固定的点。我们将使用一系列 2D 的点来描述线路。顺便说一句，**出发地**和**终点**都属于**方位**。我们不再会将它们考虑成其他不同的概念。**飞行路线**到达终点就如同它到达其他的**方位**一样。飞机必须遵照飞行路线，但这是否意味着它可以按照自己的意愿选择飞行高度呢？

专家：不。飞机在一个特定的时刻的**海拔高度**也会在**飞行计划**中有规定。

开发人员：**飞行计划**？那是什么意思？

专家：在离开机场之前，驾驶员会接到一个详细的**飞行计划**，包括所有关于这次**飞行**的信息：**飞行路线**、巡航**高度**、巡航**速度**和**飞机的类型**甚至机组成员的信息等。

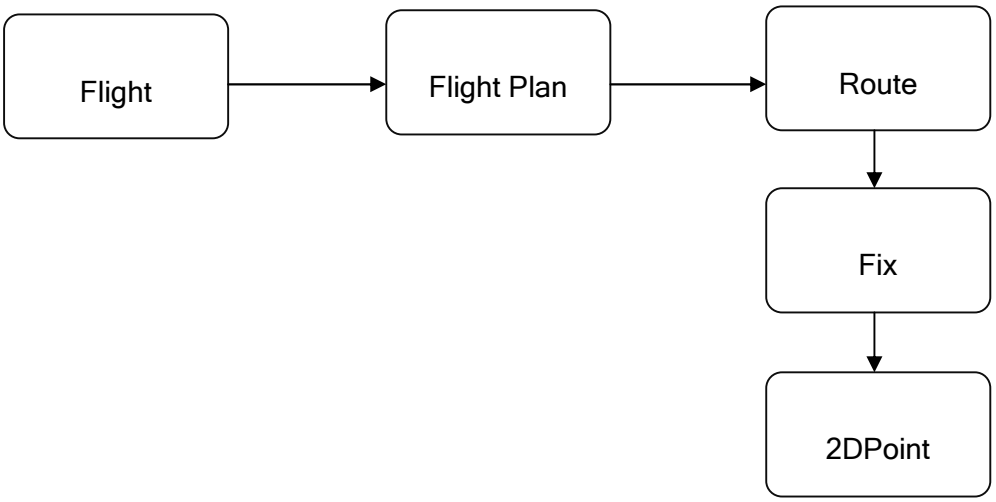
开发人员：噢，**飞行计划**看起来相当的重要。我们可得把它加到模型中。



开发人员：好多了。当我看到这副图时，我会了解到很多事情。在监控空中交通时，我们其实并不对飞机本身感兴趣，不管它是白色的还是蓝色的，也不管它是“波音”的还是“空客”的。我们对它们的“**飞行**”（flight）感兴趣。这才是我们实际上要跟踪和度量的东西。我认为我们应该对模型做些改变以确保其更加准确。

注意这个团队是如何围绕他们的初始模型讨论空中交通监控领域的，他们逐渐用粗体的词汇建立了一种语言。也要注意那种语言是如何变更模型的。

然而，在实际生活中像这样的对话太冗长了，人们经常间接地讨论事情，或者深入到非常细节的部分，或者会选择错误的概念。这会让语言的产生过程非常艰难。为了解决这个问题，所有的团队成员都应该意识到需要创建一种通用的语言，并且时常提醒大家对基本的内容保持专注，在任何需要的时候都使用这种语言。我们应该尽量少地在这些场合使用我们自己的行话，应该使用通用语言，因为它能帮助我们更清晰、更精确地交流。



强烈建议开发人员把这些模型中的主要概念实现到代码中。可以为 Route 写一个类，而另一个应该是 Fix。Fix 类应该从 2DPoint 类继承，或者应该包含一个 2DPoint 对象作为它的主要的属性。依赖于其他因素的问题，我们会在后面加以讨论。通过为模型概念建立对应的类，我们在模型和代码之间以及在语言和代码之间做映射。这非常有帮助，它会让代码的可读性更好，通过代码来重现模型。代码能够表达出模型会让项目得益，如果不对代码进行适当地设计，在模型变大或代码中发生了变化时，会导致意料之外的结果。

我们已经看到了语言是如何在整个团队中被共享的，也看到了它是如何帮助我们构建知识和创建模型的。我们应如何使用这些语言呢，只是口头交谈吗？我们已经用图了，还有其他用法吗？需要写下来吗？

有人会说 UML 很适合用来构建模型，它也真是一种很好的记录关键概念（如类）和表现它们之间的关系的工具。你可以在画板上画下 4 到 5 个类，写下它们的名字，展示它们之间的关系。对每个人来说都可以容易地理解你的想法，一个某种想法的图形化

表达方式很容易被人理解。每个人立即分享到了关于特定主题的相同的愿景，以此为基础，沟通变得更加简单。当有新的想法出现时，会修改图以反映出概念的变化。

UML 图在元素数量较少时会很有帮助，但 UML 会像夏日雨后的蘑菇那样快速增长。当你的数以百计的类充斥在一张像密西西比河那样长的纸上时，你应该怎么办？即使是对软件专家而言，它也很难阅读，就更不用说领域专家了。当它变大时，会变得难以理解，甚至一个中等规模项目的类图就会是这样。

同时，UML 擅长表达类，它们的属性和相互之间的关系。但类的行为和约束并不容易表达。为此 UML 求助于文字，将它们作为注释加在图上。所以 UML 不能传达出一个模型很重要的两个方面：它所要表现的概念的意义和对象应该做什么。但问题不大，因为我们可以增加其他的沟通工具来做这些。

我们可以使用文档。对模型进行沟通的一个明智的方式是创建一些小的图，每一张小图包含了模型的一个子集。这些图会包含若干个类以及它们之间的关系。这样就很好地包括了所涉及到的概念中一部分。然后我们可以向图中添加文本。文本将解释图所不能表现的行为和约束。每一个这样的部分都试图解释领域中的一个重要的方面，类似于一个聚光灯那样只专注于领域的一个部分。

这些文档甚至有可能是手绘的，因为这传递了一种感觉：它们是临时的，可能不久就要发生变化。这种感觉是对的，因为模型从一开始到它达到比较稳定的状态会发生很多次变化。

努力去创建一个反映整个模型的大图可能会很有诱惑力，但是大多数时候像这样的图不可能组合在一起。此外，即使你成功地制成了这样的统一的大图，它也会非常混乱，并不能传达比小图的集合更好的理解。

冗长的文档，会让我们花费很多的时间来书写它们，有可能在完成之前它们就已经作废了。文档必须与模型同步。陈旧的文档、使用了错误的语言、或者不能如实反映模型，都是没有什么用的。应尽可能的避免这样文档。

也可能使用代码来沟通，这个方法被 XP 社区广泛倡导。优良的代码也非常适合用来沟通。尽管用方法来表现行为是清楚的，但方法名也能与方法实现一样清楚吗？可以为它们提供一个测试断言，但是变量名和整体的代码结构该怎么办？它们能大声地、清楚地讲出整个故事吗？代码能够完成正确的功能，但不一定能清楚表达出其所作的事情。将模型直接写成代码，这样做是非常困难的。

在设计过程中还有其他的沟通方式，本书不想全部介绍这些方法。有一件事情是非常清楚的：由软件架构师、开发人员和领域专家组成的设计团队，需要有一种语言来统一它们的行动，以帮助它们创建一个模型，并使用代码来表达模型。

第 3 章

模型驱动设计

前面的章节强调过，以业务领域为中心，对于软件开发来说是极为重要的。我们说过，最重要的是，创建一个植根于领域、并精确反映出领域中的基本概念的模型。通用语言应该被充分运用在建模过程中，以推动软件专家和领域专家之间的沟通，以及发现应该被用在模型中的关键领域概念。建模过程的目的是创建一个优良的模型，下一步是将模型实现成代码。这是软件开发过程中同等重要的一个阶段。创建了一个优良的模型，但却未能将其成功地转换成代码将会得到质量有问题的软件。

常常出现这样的情况，软件分析人员和业务领域专家在一起工作了若干个月，一起发现了领域的基础元素，强调了元素之间的关系，创建了一个正确的模型，这个模型也正确捕获了领域知识。然后模型被传递给了软件开发人员。开发人员看模型时可能会发现模型中的有些概念或者关系无法使用代码来正确地表达。所以他们使用模型作为灵感的源泉，但是创建了自己的设计，虽然其设计借鉴了模型的某些思想，但是他们还增加了一些自己的东西。开发过程继续进行，更多的类被添加到代码中，进一步加大了原始模型和最终实现的差距。在这种情况下，很难保证产生好的结果。优秀的开发人员可能会做出一个能够工作的产品，但它能经得起时间的考验吗？它能被很容易地扩展吗？它能被容易地维护吗？

任何领域都能被表达成多种模型，每一种模型都能以不同的方式被表现达代码。对每一个特定问题而言，可能会有不止一种解决方案。我们应该选择哪一种呢？拥有一个在分析层面正确的模型，并不代表模型能被直接表达成代码。或者它的实现会违背某些软件设计原则，这是我们不建议做的事情。选择一个能够被轻易和准确地转换成代码的模型是很重要的。根本的问题是：我们应该如何完成从模型到代码的转换。

一个推荐的设计技术是创建分析模型，它被认为是与代码设计相互分离的、通常是由不同的人完成的。分析模型是业务领域分析的结果，此模型不需要考虑软件如何实现。这样的一个模型可用来理解领域，它建立了特定级别的知识，模型在分析层面是正确的。软件实现不是这个阶段要考虑的，因为这会被看作是一个导致混乱的因素。这个模型到达开发人员那里后，由他们来做设计的工作。因为这个模型中没有包含设计原则，它可能无法很好地为目标服务。因此开发人员不得不修改它，或者创建分离的设计。在模型和代码之间也不再存在映射关系。最终的结果是分析模型在编码开始后就被抛弃了。

这种方法中存在的一个主要的问题是分析无法预见模型中存在的某些缺陷以及领域中的所有的复杂性。分析人员可能会过多深入到模型中某些组件的细节，但其他的部分却缺乏足够的细节。非常重要的细节直到设计和实现过程才被发现。如实反映领域的模型可能会导致对象持久化出现严重问题，或者导致无法接受的性能表现。开发人员会被迫做出他们自己的决定，并会做出设计变更以解决一个实际问题，而这个问题在模型创建时是没有考虑到的。他们创建了一个偏离了模型的设计，使得设计与模型更加不相关。

如果分析人员独立工作，他们最终也会创建出一个模型。当这个模型被传递给开发人员，分析人员的一些关于领域和模型的知识丢失了。虽然模型可以被表达成图形或者文字形式，开发人员常常难以掌握模型的完整含意、或者某些对象的关系或者它们之间的行为。模型中的一些细节不容易表达为图形的形式，甚至也可能无法完全用文字来表达。开发人员理解这些细节非常困难。在某种情况下他们会对想要的行为做出一些假设，而这些假设有可能是错误的，最终会导致错误的程序功能。

分析人员会举行一些他们内部的会议，在会议上他们讨论领域，会享有很多的知识。他们创建了一个应该包含所有信息的浓缩模型，开发人员不得不阅读他们提供的文档以吸收所有知识。如果开发人员能够参加分析讨论会议，并在开始做编码设计之前对领域和模型获得一个清晰完整的视图，他们的工作将会更有效率。

一种更好的方法是将领域建模和设计紧密关联起来。模型在构建时就考虑到软件实现和设计。开发人员应该被加入到建模的过程中来。主要的想法是选择一个能够在软件实现中恰当地表达的模型，这样设计过程会很顺畅并且基于模型。将代码与其所基于的模型紧密关联，将会使得代码更有意义，并且与模型保持相关。

有了开发人员的参与，就会获得反馈，它能确保模型能够在软件中得到实现。如果其中某处有错误，会在早期就被标识出来，问题也能够很容易得到纠正。

写代码的人应该很好地了解模型，应该感觉到自己有责任保持它的完整性。他们会意识到对代码的一个变更其实就隐含着对模型的变更，否则，如果哪里的代码不能表达最初模型的话，他们将会对代码做重构。如果分析人员从实现过程中分离出去，他会不再关心开发过程中引入的局限性，最终的结果是模型将不再实用。

任何技术人员对模型做出贡献，必须花费一些时间来接触代码，无论他在项目中担负的是什么主要角色。任何一个负责修改代码的人都必须学会用代码来表达模型。每位开发人员都必须参与到一定级别的领域讨论中并和领域专家保持联络。那些按不同方式贡献的人必须自觉地与接触代码的人密切协作，使用通用语言来动态交换模型思想。

如果设计或者设计中的核心部分没有映射到领域模型，模型就没有什么价值，而软件是否正确也就令人怀疑。同时，模型和设计功能之间的映射如果很复杂，就会很难理解，当设计变更了实际上模型是不可能维护的。分析和设计之间出现了致命的割裂，这样一个活动（分析或设计）中产生的想法将无法对另外一个产生影响。

设计软件系统的一部分，确保它能如实反映领域模型，让（设计与模型之间的）映射显而易见。重新访问模型，修改它，使它能在软件中更自然地得到实现，甚至可以让它如你所愿反映出对领域更深层的理解。除了支持一种流畅的通用语言外，这要求一个单独的模型能够服务好这两个目的。

从模型中提取出在设计中使用的术语和所分配的职责之后，代码就成了模型的表达方式，所以对代码的一个变更就可能成为对模型的变更。变更的影响相应地还会波及到项目的其余活动中。

为了实现实现和模型之间的紧密捆绑，通常需要支持建模范型（例如面向对象编程）的软件开发工具和语言。

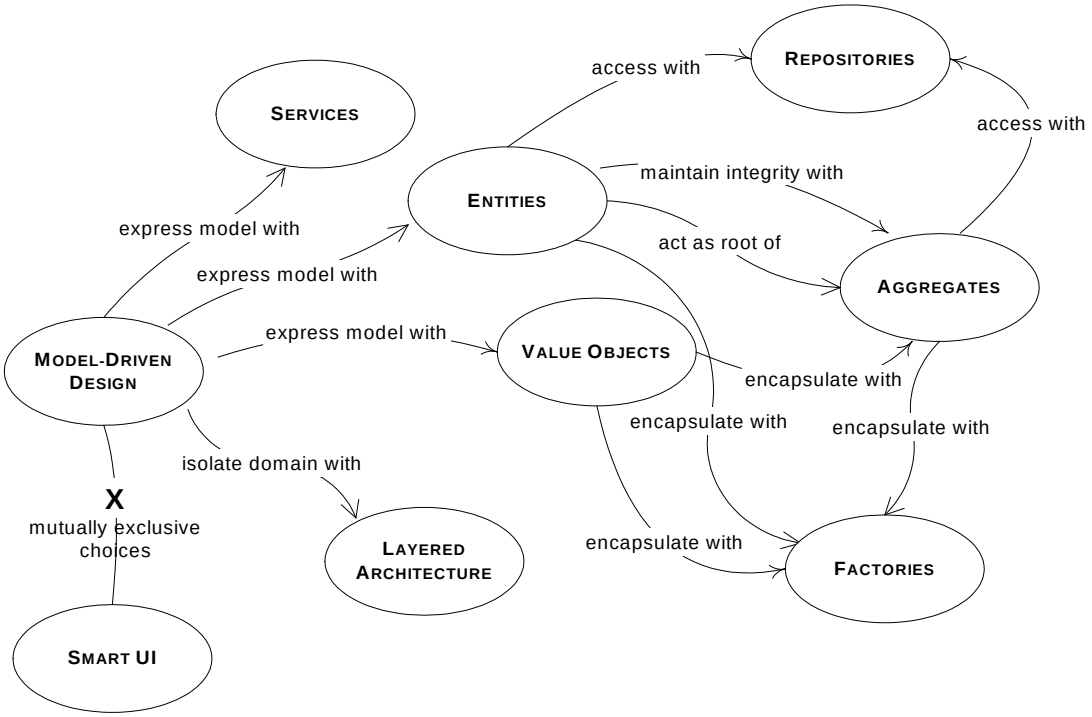
面向对象编程非常来实现模型，因为它们基于同一个范型（即，面向对象）。面向对象编程提供了对象的类、类之间的关联关系、对象实例、以及对象实例之间的消息通信。面向对象编程语言使得在带有关联关系的模型对象与它们的编程对等物之间创建直接的映射成为了可能。

过程化语言仅仅为模型驱动设计提供了有限的支持。这样的语言不能提供实现模型的关键组件所必须的构造。有人说象 C 语言这样的过程化语言也能够实现面向对象编程，确实，某些功能可能被用某种方式再现：对象可能会用数据结构来模拟。这样的数据结构无法包含对象的行为，所以必须独立添加函数。这种数据的意义仅仅存在于开发人员的脑海中，因为代码本身并不是那么明确。一段用过程化语言写就的程序通常被认为是一组函数，一个函数调用另一个，相互协作以达到一个特定的结果。这样的程序无法轻易地封装概念性连接，因此很难实现领域和代码之间的映射。

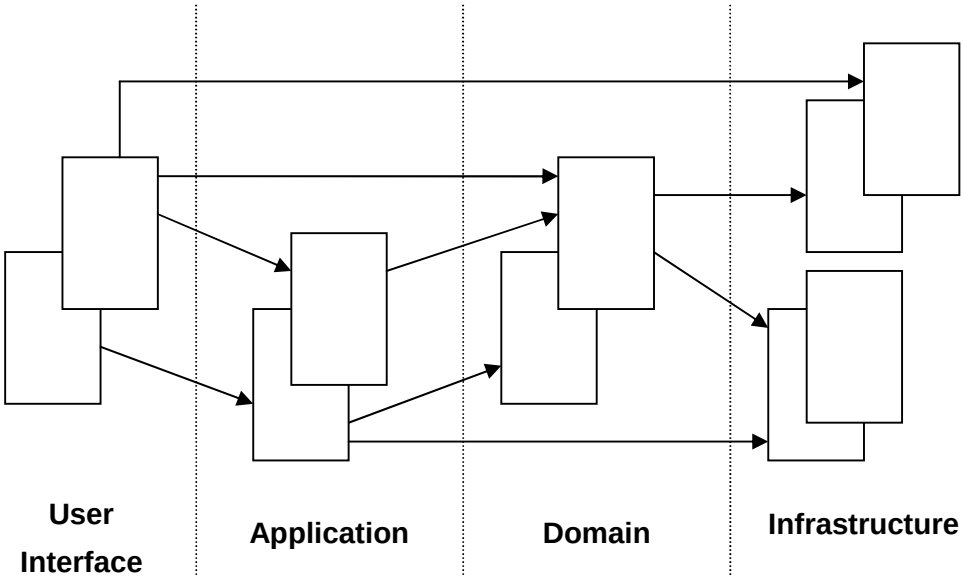
某些特殊的领域（例如数学）可以使用过程化编程来轻易地建模和实现，那是因为很多数学理论大多数都是关于计算的，可以简单地使用函数调用和数据结构来解决。更加复杂的领域并非仅仅是一组包含有计算的一套抽象的概念，无法被简化成一系列的算法，因此过程化语言不足以表达相应的模型。因为这个原因，对于模型驱动设计而言，不推荐使用过程化编程。

模型驱动设计的基本构成要素

本章接下来的几节中会展现在模型驱动设计中要使用的最重要的模式。这些模式的作用是从领域驱动设计的角度展现一些对象建模和软件设计中的关键元素。下图是将要展现的模式和模式间关系的总图。



分层架构



当我们创建一个软件应用时，这个应用的很大一部分并没有直接与领域关联，但它们是基础设施的一部分或者是为软件本身提供服务的。最好能让应用中的领域部分与其

余部分相比保持尽可能小（而不是和其余部分掺杂在一起），因为一个典型的应用包含了大量访问数据库、访问文件或网络、用户界面等相关的代码。

在一个面向对象的程序中，用户界面、数据库以及其他支持性代码经常被直接写到业务对象中。附加的业务逻辑被嵌入到 UI 组件和数据库脚本的行为中。之所以有时候这样做，原因是这样可以很容易地让事情快速工作起来。

但是，当领域相关的代码被混入到其他层时，要阅读和思考这些代码也变得极其困难。表面看上去是对 UI 的修改，却变成了对业务逻辑的修改。对业务规则的变更可能需要谨慎跟踪用户界面代码、数据库代码以及其他程序元素。实现粘连在了一起，模型驱动对象（model-driven objects）于是变得不再可行。也很难开展自动化测试。对于所有活动中包含的全部技术和逻辑而言，程序必须保持简单，否则就会变得很难理解。

因此，将一个复杂的程序划分成多个层。为每一个层开发一个内聚的设计，让每个层仅依赖于它底下的那些层。遵照标准的架构模式实现与其上面的那些层的低耦合。将领域模型相关的代码集中到一个层中，把它从用户界面、应用和基础设施代码中隔离开来。领域对象不必再承担显示自己、保存自己、管理应用任务的职责，而是专注于表达领域模型。这会让一个模型逐渐进化得足够丰满、足够清晰，以便捕获基本的业务知识，并且能够正常工作。

领域驱动设计的一个通用的架构解决方案包含了 4 个概念层：

用户界面/展现层	负责向用户展现信息以及解释用户命令
应用层	很薄的一层，用来协调应用的活动。它不包含业务逻辑。它也不保留业务对象的状态，但它保留有应用任务的进度状态
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
基础设施层	本层作为其他层的支撑库存在。它提供了层间的通信，实现对业务对象的持久化，包含对用户界面层的支持库等作用

将应用划分成分离的层并建立层间的交互规则，这样做是很重要的。如果代码没有被清晰地隔离到一些层中，就会很快发生混乱，因为管理变更将会变得非常困难。在某处对代码的一个简单修改会对其他地方的代码造成难以预测和并不希望出现的结果。领域层应该关注核心的领域问题。它不应该包括基础设施方面的活动。用户界面既不应该与业务逻辑紧紧捆绑，也不应该与通常属于基础设施层的任务紧紧捆绑。在很多情况下应用层是必要的。它会成为业务逻辑之上的管理者，用来监督和协调应用的一切活动。

例如，应用层、领域层和基础设施层之间的一个典型交互，看上去会是这样：用户想要预定一个飞行路线，请求一个位于应用层中的应用服务来做这件事情。应用层从基

础设施层中取得相关的领域对象，然后调用它们的相关方法，例如检查与其他已经被预定的飞行线路的安全界限（security margins）。当领域对象执行完所有的检查并将它们的状态修改为“已决定”（decided）之后，应用服务将对象持久化到基础设施中。

实体

有一类对象看上去好像拥有标识符，它的标识符在历经软件的各种状态变更后仍能保持一致。对这些对象而言，重要的不是其属性，而是其延续性和标识，对象的延续性和标识会跨越甚至能够超出软件系统的生命周期。我们把这样的对象称为实体。

OOP 语言会把对象的实例放于内存中，它们会为每个对象保持一个对象引用或者记录一个对象地址。在给定的某个时刻，这种引用对每一个对象而言是唯一的，但是很难保证在不确定的某个时间段内它也是如此。实际上恰恰相反。对象经常被移出或者移回内存，它们被序列化后在网络上传输，然后在另一端被重新创建，或者它们被销毁了。在程序的运行环境中，那个看起来像标识符的对象引用其实并不是我们正在谈论的标识符。如果有一个存放了天气信息（如温度）的类，很容易产生同一个类的不同实例，这两个实例都包含了同样的值，这两个对象是完全相当的，相互之间可以互换，但是它们拥有不同的引用，它们并不是实体。

如果我们要用软件程序实现一个“人”的概念，我们可能会创建一个 Person 类，这个类会带有一系列的属性，例如：名称，出生日期，出生地等。这些属性中有哪个可以作为 Person 的标识符吗？名字不可以作为标识符，因为可能有很多人拥有同一个名字。如果我们只考虑两个人的名字的话，我们不能使用同一个名字来区分他们两个。我们也不能使用出生日期作为标识符，因为会有很多人出在同一天出生。同样也不能用出生地作为标识符。一个对象必须与其他的对象区分开来，即使是它们拥有着相同的属性。错误的标识符可能会导致数据混乱。

考虑一下一个银行会计系统。每一个账户拥有它自己的编号。每一个账户可以用它的编号来精确地标识。这个编号在系统的生命周期中会保持不变，并保证延续性。账户编号可以作为一个对象存在于内存中，也可以在内存中被销毁并保存到数据库中。当这个账户被关闭时，它还可以被归档，只要还有人对它感兴趣，它就依然在某处存在。不论它的表现形式如何，编号会保持一致。

因此，在软件中实现实体意味着创建标识符。对一个人而言，其标识符可能是属性的组合：名称、出生日期、出生地、父母姓名、当前地址。在美国，社会保险编号也会被用来创建标识符。对一个银行账户来说，账户编号看上去已经足以作为标识符了。通常标识符或是对象的一个属性（或属性的组合），一个专门为保存和表达标识符而创建的属性，甚至可以是一种行为。非常重要的是，系统能够轻易区分开两个拥有不同标识符的对象，或者将两个使用了相同标识符的对象看做是相同的。如果不能满足这个条件，整个系统可能会变得破烂不堪。

有不同的方法来为每一个对象创建一个唯一标识符：ID 可能是由一个模块自动生成的，仅仅在软件中内部使用，不会暴露给用户；ID 可能是数据库表的一个主键，可以确保在数据库中是唯一的。只要从数据库中获取到该对象，它的 ID 也会被获取到并在内存中被重建；ID 也可能是由用户创建的，例如每个机场会有一个关联的代码。每个机场拥有一个唯一的字符串 ID，这个字符串是在世界范围内通用的，被世界上所有的旅行代理用来标识它们的旅行计划中涉及的机场。另一种解决方案是使用对象的属性来创建标识符，当这个属性不足以代表标识符时，可以添加另一个属性来帮助标识相应的对象。

当一个对象可以用其标识符而不是它的属性来区分时，可以将标识符作为在模型中该对象定义的主要部分。使类的定义保持简单并专注于生命周期的延续性和标识符。为每个对象定义一个区分手段，该手段与对象的形式或历史无关。警惕要求使用属性来匹配对象的需求。定义一个可以确保对每一个对象产生一个唯一结果的操作，这个过程可能需要附加一个可以确保唯一性的标志。这意味着对象的标识可以来自外部，也可以是由系统产生和使用的任意标识符，但它必须符合模型中的身份区别。模型必须定义满足什么条件，两个东西可以被看作同一事物。

实体是领域模型中非常重要的对象，并且它们应该在建模过程开始时就被考虑。决定一个对象是否需要成为一个实体也很重要，这会在下一个模式中加以讨论。

值对象

我们已经讨论了实体以及在建模阶段及早识别实体的重要性。实体在领域模型中是必需的对象。我们应该将所有的对象视为实体吗？每一个对象都应该有一个标识符吗？

我们可能被诱导将所有对象看成实体。实体是可以被跟踪的，但跟踪和创建标识符需要很大的成本。我们需要确保每一个实体的实例都有唯一标识，跟踪标识也并非易事。需要花费很多仔细的考虑来决定由什么来构成一个标识符，因为一个错误的决定可能会让对象拥有相同的标识，而这并不是我们所期望的。将所有的对象实现为实体也会带来隐含的性能问题，因为需要对每个对象产生一个实例。如果 Customer 是一个实体对象，那么这个对象的一个实例代表了一个特定的银行客户，不能被对应于其他客户的账户操作所复用，造成的结果是必须为每一个客户创建一个这样的实例。当处理成千上万的实例时，系统的性能会严重下降。

让我们考虑一个绘画应用。用户会看到一个画布且他能够用任何宽度、样式和颜色来画任何点和线。创建一个叫做 Point 的类非常有用，程序会对画布上的每一个点创建这个类的一个实例。这样的一个点会包含两个属性，对应于屏幕或者画布的坐标。是否有必要考虑每一个点都拥有一个标识符？这个标示符会有延续性吗？对于这样一个对象而言，看起来重要的事情只有它的坐标而已。

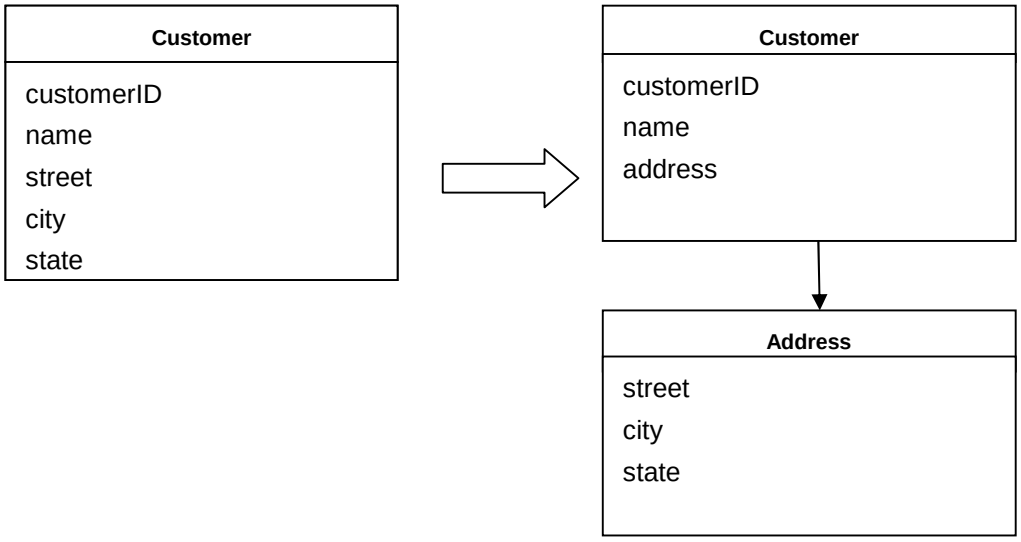
有的时候我们需要包含一个领域对象的某些属性。我们对它是哪一个对象并不感兴趣，而是只关心它所拥有的属性。用来描述领域的特定方面、并且没有标识符的一个对象，叫做值对象。

区分实体对象和值对象非常必要。出于想要统一的缘故而将所有对象实现为实体对象，这样做没有多大帮助。实际上，只建议选择那些符合实体定义的对象作为实体，而将剩下的对象实现为值对象（我们会在下一节引入其他类型的对象，但我们假设现在只有实体对象和值对象两种）。这样做会简化设计，并且将会产生某些其他的积极影响。

没有标识符，值对象就可以被轻易地创建或者丢弃。没有人关心创建一个标识符，在没有其他对象引用时，垃圾回收会处理这个对象。这极大简化了设计。

极力推荐将值对象实现为不可变的。它们由一个构造器创建，并且在它们的生命周期内永远不会被修改。当你想要得到这个对象的不同的值时，你简单地创建另一个对象就行了。这会对设计产生重要的影响。实现为不可变的，并且不具有标识符，值对象就能够被共享了。这对某些设计是必要的。在性能很重要的场合，不可变的对象是可以共享的。它们也能维持一致性，例如：数据一致性。设想一下共享一个可变的对象会意味着什么。一个航空旅行预定系统能够为每个航班创建对象，这个对象会有一个可能是航班号的属性。一个客户会为一个特定目的地预定一个航班。另一个客户想要订购同一个航班。因为是同一个航班，系统选择了重用持有那个航班号的对象。这时，客户改变了主意，选择换成一个不同的航班。因为它不是不可修改的，所以系统修改了航班号。这会导致第一个客户的航班号也发生了变化。

一条箴言是：如果值对象是可共享的，那么它们应该是不可变的。值对象应该保持很小、很简单。当其他参与方需要一个值对象时，可以简单地传递值，或者创建一个副本。制作一个值对象的副本是非常简单的，通常不会有什么副作用。如果没有标识符，你可以按你所需创建若干副本，然后根据需要来销毁它们。



值对象可以包含其他的值对象，它们甚至还可以包含对实体对象的引用。尽管值对象仅仅用来包含一个领域对象的属性，但这并不意味着它应该包含一长列所有的属性。属性可以被分组到不同的对象中。被选择用来构成一个值对象的属性应该形成一个概念上的整体。一个客户会跟其姓名、街道、城市、州相关。最好用一个单独的对象来包含这些地址信息，客户对象会包含一个对这个对象的引用。街道、城市、州应该归属于一个对象，因为它们在概念上属于一体的，而不应该作为客户对象分离的属性。

服务

当我们分析领域并试图定义构成模型的主要对象时，我们发现领域的有些方面难以被映射成对象。对象通常被认为是拥有属性，一个由对象管理的内部状态、并且暴露出一种行为。在我们开发通用语言时，领域中的关键概念被引入到语言中，语言中的名词很容易被映射成对象。语言中对应那些名词的动词成为了那些对象的行为。但是有些领域中的动作，它们是一些动词，看上去却不属于任何对象。它们代表了领域中的一个重要的行为，所以不能忽略它们或者简单地把它们合并到某个实体或者值对象中。给一个对象增加这样的行为会破坏这个对象，让它看上去拥有了本不该属于它的功能。但是，要使用一种面向对象语言，我们必须使用一个对象才行。我们不能只拥有一个单独的功能，它必须附属于某个对象。通常这种行为类的功能会跨越若干个对象，或许是不同的类。例如，为了从一个账户向另一个账户转钱，这个功能应该放到转出的账户还是在接收的账户中？感觉放在这两个中的哪一个也不对劲。

当这样的行为从领域中被识别出来时，最佳实践是将它声明成一个服务。这样的对象不再拥有内置的状态，它的作用仅仅是为领域提供相应的功能。服务所能提供的帮助作用是非常重要的，一个服务可以将服务于实体和值对象的相关功能进行分组。最好显式地声明服务，因为它在领域中创建了一个清晰的区分，它封装了一个概念。把这样的功能放入实体或者值对象都会导致混乱，因为那些对象所代表的含义将会变得不再清晰。

服务担当了一个提供操作的接口。服务在技术框架中是很常见的，但它们也能被运用到领域层中。一个服务不是与执行服务的对象相关，而是与操作所要执行的对象相关。在这种情况下，一个服务通常变成了多个对象的一个连接点。这也是为什么行为应该很自然地隶属于一个服务而不是被包含在领域对象中的一个原因。如果这样的功能被包含在领域对象中，就会在领域对象和作为操作受益者的对象之间建立起一个密集的关联网。众多对象之间的高耦合度是糟糕设计的一个信号，因为这会让代码很难阅读与理解，更重要的是，这会导致很难进行变更。

一个服务不应该替代通常隶属于领域对象的操作。我们不应该为每一个需要的操作创建一个服务。但是当一个操作凸现为领域中的一个重要概念时，就需要为它创建一个服务了。以下是服务的 3 个特征：

1. 服务执行的操作代表了一个领域概念，这个领域概念无法自然地隶属于一个实体或者值对象。
2. 被执行的操作涉及到领域中的其他的对象。
3. 操作是无状态的。

当领域中的一个重要的过程或者转换不属于一个实体或者值对象的职责时，向模型中添加一个操作，作为一个单独的接口将其声明为一个服务。根据模型的语言定义一个接口，并确保操作的名字是通用语言的一部分。使服务成为无状态的。

当使用服务时，保持领域层的隔离非常重要。很容易弄混属于领域层的服务和属于基础设施层的服务。应用层中也可能会有服务，这增加了额外的复杂性。将这些服务与领域层中相关服务分离开来会更加困难。当我们在设计阶段建立模型时，我们需要确保将领域层与其他的层隔离开来。

不论是应用服务还是领域服务，通常都是建立在领域的实体对象和值对象之上，以便提供与这些对象直接相关的服务。决定一个服务应该归属于哪一层是很困难的。如果所执行的操作概念上属于应用层，那么服务就应该放到这个层。如果操作是关于领域对象的，而且确实是与领域有关的、为领域的需要服务，那么它就应该属于领域层。

让我们考虑一个实际的 Web 报表应用的例子。报表使用存储在数据库中的数据，它们会基于模版来生成。最终的结果是一个在 Web 浏览器中可以显示给用户查看的 HTML 页面。

用户界面层被包含在 Web 页面中，允许用户登录，选择想要查看的报表，单击一个按钮就来请求这个报表。应用层是很薄的一层，它位于用户界面层与领域层、基础设施层之间。在登录操作时，它会跟数据库基础设施进行交互；在需要创建报表时会和领域层进行交互。领域层中包含了领域的核心部分，与报表直接相关的对象。有两个这样的对象，Report 和 Template，它们是生成报表的基础。基础设施层将支持数据库访问和文件访问。

当用户选择创建一个报表时，他实际上从名称列表中选择了一个报表名称。这是一个字符串类型的 reportID。还会传递其他的参数，例如要在报表中显示的条目、报表中所包括数据的时间间隔等。但出于简化的考虑我们将只提到 reportID。这个名称会通过应用层被传递到领域层。由领域层负责根据所给的名称来创建并返回报表。因为报表会基于模版产生，我们需要创建一个服务，它的作用是根据一个 reportID 获得对应的模版，这个模版被保存在一个文件或者数据库中。保存操作不适于作为 Report 对象自身的一个操作。它也同样不属于 Template 对象。所以我们创建了一个独立的服务，这个服务的目的是基于一个报表的标识符来获取一个报表模版。这会是一个位于领域层的服务。它会通过使用文件方面的基础设施，从磁盘上获取模版。

模块

对一个大型的复杂项目而言，模型趋向于越来越大。当模型发展到了某个规模，将它作为整体来讨论很困难，理解不同部件之间的关系和交互变得很困难。基于这个原因，必须将模型组织到模块中。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。

模块被广泛使用在很多项目中。如果你查看模快包含的内容以及那些模块间的关系，就会很容易从中掌握大型模型的概况。理解了模块之间的交互之后，人们就可以开始处理模块中的细节了。这是管理复杂性的简单有效的方法。

另一个使用模块的原因跟代码质量有关。普遍认为软件代码应该具有高层次的内聚性和低层次的耦合度。虽然内聚开始于类和方法级别，它也可以应用于模块级别。推荐的做法是将高关联度的类分组到一个模块，以提供尽可能大的内聚性。有很多类型的内聚性。最常用到的两个是通信性内聚（communicational cohesion）和功能性内聚（functional cohesion）。在模块中的部件操作相同的数据时，可以得到通信性内聚。把它们分到一组很有意义，因为它们之间存在很强的关联性。在模块中的部件协同工作以完成定义好的任务时，可以得到功能性内聚。功能性内聚被认为是最佳的内聚类型。

在设计中使用模块是一种提高内聚性和消除耦合度的方法。模块应该由在功能上或者逻辑上属于一体的元素构成，以确保内聚性。模块应该具有定义好的接口，这些接口可以被其他的模块访问。最好用访问一个接口的方式，而不是调用模块中的三个对象，因为这样做可以降低耦合度。低耦合降低了复杂性并提高了可维护性。与每个模块同其他的模块间存在许多连接相比，如果模块间仅有极少的连接，通过这些连接来执行定义好的功能，这样做会让人更容易理解系统是如何工作的。

我们应该选择那些能够表达出系统功能并且包含具有内聚性的一组概念的模块。这样做常常会降低模块之间的耦合度，即使我们并没有寻求一种方式来修改模型，以解除这组概念之间的耦合，或者找到一个可以作为模块基础的宏观概念，使得各个元素有机地组合在一起。寻求降低概念之间的耦合度，使得概念能够被独立理解和推导。重新定义模型，直到能够按照高级别的领域概念将它区分开来，而且对应的代码也被很好地解耦。

给定的模块名称会成为通用语言的组成部分。模块和它们的名称应该能够反映出对领域的深层理解。

设计人员会习惯地从一开始就创建模块，这在我们的设计过程中是很普通的部分。模块的角色被决定以后通常会保持不变，尽管模块的内部会发生很多变化。模块的设计应该拥有一些灵活性，允许模块随着项目的进展而进化，并且不应该被冻结。大家都明

白模块的重构成本要比类的重构昂贵的多，但是如果发现了一个模块设计方面的错误，还是需要解决这个问题的。可以先改变模块，然后再寻求更进一步的解决途径。

聚合

本章的最后3个模式将处理不同的建模挑战，其中一个跟领域对象的生命周期相关。领域对象在它们的生命周期内会历经若干种状态，它们被创建、放在内存中、在计算中被使用、直到最后消亡。有时它们会被保存到一个永久位置，例如数据库中，这样可以在以后的日子里被获取到，或者被存档。有时它们会被完全从系统中清除掉，包括从数据库和归档介质上。

管理领域对象的生命周期自身就会遇到一个挑战，如果做得不恰当，就会对领域模型产生负面影响。我们将介绍 3 个模式来帮助我们处理这个挑战。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设计模式，用来帮助我们处理对象的创建和存储问题。我们将从聚合开始讨论。

一个模型会包含众多的领域对象。无论在设计时做了多少考虑，我们都会看到很多对象会跟其他的对象发生关联，形成了一个复杂的关系网。这些关联的类型有很多种。对模型中的每一个可导航的关联而言，都应该有对应的软件机制来加强它。领域对象间实际的关联存在于代码中，很多时候甚至存在于数据库中。一位客户和用它名字开立的银行账户之间存在的一个 1 对 1 的关系，会被表达为两个对象之间的引用，并且在两个数据库表中隐含着一个关联关系，一个表存放有客户信息，另一个表存放有账户信息。

来自模型的挑战常常不是让它们尽量完整，而是让它们尽量地简单和容易理解。这意味着，直到模型中嵌入了对领域的深层理解，否则大多数时候需要对模型中的关系进行消减和简化。

一个 1 对多的关联关系就更复杂了，因为它涉及到了相关的多个对象。这种关系可以被简单地转换成一个对象与一个其他对象的集合之间的关联，虽然这并不总能行得通。

多对多的关联关系大部分情况下是双向的。这又增加了复杂度，使得管理对象的生命周期非常困难。关联的数字应该被尽可能减小。首先，要删除模型中非基本的关联关系。它们可能在领域中是存在的，但它们在我们的模型中不是必要的，所以我们要删除它们。其次，可以通过添加约束的方式来减少多重性。如果很多对象满足一种关系，那么在这个关系上加入正确的约束之后，很有可能只有一个对象会继续满足这种关系。第三，很多时候双向关联可以被转换成非双向的关联。每一辆汽车都有一台发动机，并且发动机在运转时，都会属于一辆汽车。这种关系是双向的，但是很容易将其简化为汽车拥有发动机，而不用考虑反向的。

在我们减少和简化了对象之间的关联后，我们仍然会得到很多的关系。一个银行系统会保留并处理客户数据。这些数据包括客户的个人数据（例如姓名、地址、电话号码、

工作描述等）和账户数据：账户、余额、执行的操作等。当系统归档或者完全删除一个客户的信息时，必须要确保所有的引用都被删除了。如果许多对象持有这样的引用，则很难确保它们全被清除了。同样地，如果一个客户的某些数据发生了变化，系统必须确保在整个系统中执行了适当的更新，数据的一致性必须得到保证。这通常是在数据库层面进行处理的。通常会使用事务来确保数据的一致性。但是如果模型没有被仔细地设计过，会产生很大程度的数据库争夺，导致性能很差。当数据库事务在这样的操作中担负重要角色时，我们会期望直接在模型中解决跟数据一致性相关的一些问题。

强化不变量通常也是有必要的。不变量是在数据发生变化时必须维护的那些规则。在许多对象持有正在发生变化的数据对象的引用时，不变量是很难实现的。

在模型中拥有复杂关联的对象发生变化时，很难保证其一致性。很多时候不变量会被应用到密切相关的对象之上，而不是离散的对象。但是谨慎的锁定模式又会导致多个用户之间发生不必要的冲突，系统会变得不可用。

因此，使用聚合。聚合是针对数据变化可以考虑成一个单元的一组关联的对象。聚合使用边界将内部和外部的对象划分开来。每个聚合都有一个根。这个根是一个实体，并且它是外部可以访问的唯一的对象。根对象可以持有对任意聚合对象的引用，其他的对象可以互相持有彼此的引用，但一个外部对象只能持有对根对象的引用。如果边界内还有其他的实体，那些实体的标识符是本地化的，只在聚合内有意义。

聚合是如何保持数据一致性和强化不变量的呢？因为其他对象只能持有对根对象的引用，这意味着它们不能直接变更聚合内的其他的对象。它们所能做的就是对根做变更，或者让请求根来执行某些动作。根能够变更其他对象，但这是聚合内包含的操作，并且它是可控的。如果根从内存中被删除并移除，聚合内的其他所有的对象也将被删除，因为再不会有其他的对象持有对它们当中的任何一个的引用了。当针对根对象的变更间接影响到聚合内的其他对象，强化不变量变得很简单，因为根将做这件事情。如果外部对象能直接访问内部对象并且变更它们时，强化不变量将变得困难的多。在这种情况下，强化不变量意味着将某些逻辑放到外部对象中去处理，这不是我们所期望的。

根对象可能会将内部对象的临时引用传递给外部对象，作为限制，当操作完成之后，外部对象不能再持有这个引用。一种简单的实现方式是向外部对象传递值对象的副本。在这些副本对象上发生了什么将不再重要，因为它不会以任何方式影响到聚合的一致性。

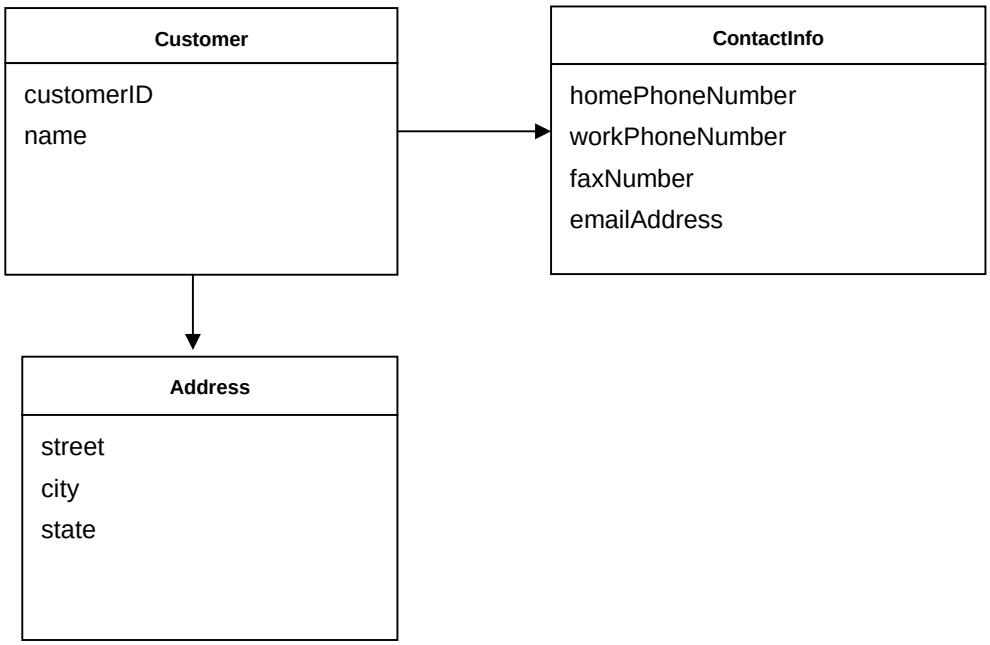
如果一个聚合中的对象被保存到数据库中，可以通过查询来获得的应该只有根对象。其他的对象只能通过从根对象出发导航关联对象来获得。

聚合内的对象可以被允许持有对其他聚合的根对象的引用。

根实体拥有全局的标识符，并且有责任维护不变量。内部的实体拥有内部的标识符。

将实体和值对象聚集在聚合之中，并且定义各个聚合之间的边界。为每个聚合选择一个实体作为根，并且通过根来控制所有对边界内的对象的访问。允许外部对象仅持有对根的引用。对内部成员的临时引用可以被传递出来，但是仅能用于单个操作之中。因为由根对象来进行访问控制，将无法盲目地对内部对象进行变更。这种安排使得强化聚合内对象的不变量变得可行，并且对聚合而言，它在任何状态变更中都是作为一个整体。

聚合的一个简单的例子如下图所示。客户（Customer）是聚合的根，并且其他所有的对象都是内部的。如果需要地址（Address），一个它的副本将被传递给外部对象。



工厂

实体和聚合常常会很大很复杂，过于复杂以至于难以通过根实体的构造器来创建。实际上通过构造器努力构建一个复杂的聚合，并不是领域本身通常应该做的事情，在领域中，某些事物通常是由别的事物创建的（例如电器是在生产线上被创建的）。通过根实体的构造器来构建复杂的聚合，看上去就像是要用打印机构建打印机本身。

当一个客户对象（client object）想创建另一个对象时，它会调用它的构造器，可能还会传递某些参数。但是当对象构建是一个很费力的过程时，创建这个对象会涉及到大量的知识，例如对象的内部结构、所包含对象之间关系的以及应用在这些对象上的规则等。这意味着该对象的每个客户将持有关于该对象构建的专用知识。这破坏了对于领域对象和聚合的封装。如果客户属于应用层，领域层的一部分将被移到了外边，从而打乱整个设计。在真实生活中，这就像是给我们塑胶、橡胶、金属、硅，让我们来构建自己的打印机。这不是不可能完成的，但这样做值得吗？

创建一个对象可以是它自身的主要操作，但是复杂的组装操作不应该成为被创建对象的职责。组合这样的职责会产生笨拙的设计，也很难让人理解。

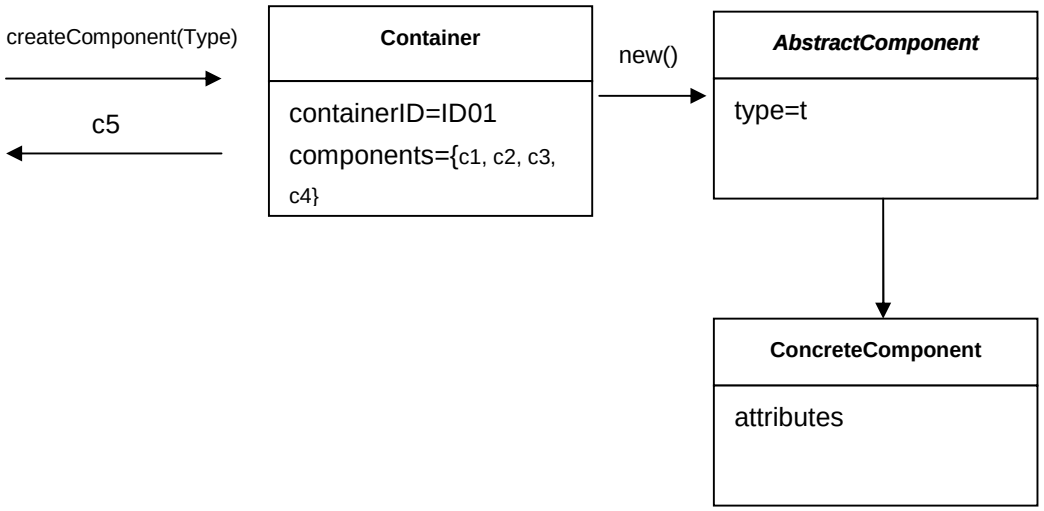
因此，有必要引入一个新的概念，这个概念可以帮助封装复杂的对象创建过程，它就是工厂（Factory）。工厂被用来封装对象创建所必需的知识，它们对创建聚合特别有用。当聚合的根被创建后，所有聚合包含的对象将随之创建，所有的不变量得到了强化。

保持创建过程的原子性非常重要。如果不这样做，创建过程就有可能会对某个对象执行了一半操作，将这些对象置于未定义的状态，对于聚合而言更是如此。当根被创建之后，所有对象服从的不变量也必须被创建完毕，否则，不变量将无法得到强化。对不变的值对象而言则意味着所有的对象属性都被初始化成有效的状态。如果一个对象无法被正确地创建，将会产生一个异常，确保没有返回一个无效的值。

因此，为复杂对象和聚合创建实例的职责，应该转交给一个单独的对象。虽然这个对象本身在领域模型中没有职责，但它仍是领域设计的一部分。提供一个接口来封装所有复杂的组装过程，客户不需要引用正在初始化的对象所对应的具体类。将整个聚合当作一个单元来创建，强化它们的不变量。

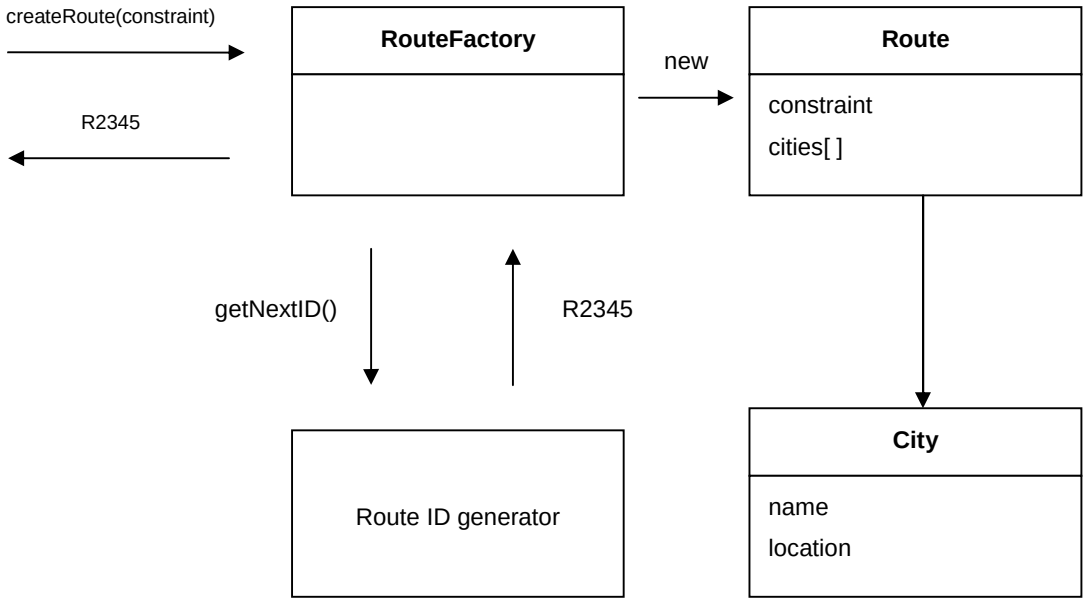
有很多的设计模式可以用来实现工厂模式。由 Gamma 等人著的《设计模式》一书中对此有详细描述，并介绍了两种不同的模式：工厂方法（Factory Method）和抽象工厂（Abstract Factory）。我们不会尝试从设计的角度介绍模式，而是从领域建模的角度来介绍它。

工厂方法是一个对象方法，包含并隐藏了必要的创建其他对象的知识。这在客户试图创建一个属于某聚合的对象时是非常有用的。解决方案是给聚合的根添加一个方法，由这个方法来负责对象的创建，强化所有的不变量，返回对那个对象的一个引用或者一个副本。



在容器（container）中包含着具有某种特定类型的组件。当这样的一个组件被创建后能自动归属于一个容器是很有必要的。客户调用容器的 createComponent(Type t)方法，容器实例化一个新的组件。组件的具体类取决于它的类型（通过 createComponent 传入）。在创建之后，组件被增加到容器所包含的组件的集合中，将组件的一个副本返回给客户。

有时候创建一个对象的逻辑会更为复杂，或者创建一个对象还涉及到创建其他一系列对象时（例如，创建一个聚合）。可以使用一个单独的工厂对象来隐藏聚合的内部构造所需要的任务。让我们考虑一个程序模块的例子，在给定一组约束条件的情况下，计算一辆汽车从起点行驶到终点的路线。用户登录 Web 站点后运行应用程序，并指定一个要遵守的约束条件：最短的路线，最快的路线，或者最便宜的路线。可以在被创建的路线上标注用户信息，这些信息需要被保存下来，以便客户下次登录时能够检索到。



路线 ID 的生成器被用来为每一条路线创建一个唯一的标识符，这对一个实体而言是必要的。

当创建一个工厂时，我们被迫违反一个对象的封装原则，而这必须谨慎行事。每当对象中发生了某种变化时，会对构造规则或者某些不变量造成影响，我们需要确保工厂也被更新以支持新的条件。工厂和它们要创建的对象是紧密关联的。这可能是个弱点，但它也有长处。一个聚合包含了一系列密切相关的对象。根的构建与聚合内的其他对象的创建是相关的。会有一些逻辑一同放到聚合中，这些逻辑并不天然属于任何一个对象，因为它总是跟其他对象的构建有关。看起来比较合适的做法是，使用一个专用的工厂类来负责创建整个聚合，在这个工厂类中将包含应该为聚合强化的规则、约束和不变量。这个对象会保持简单，并将完成特定的目的，不会使复杂的构建逻辑混乱不堪。

实体工厂和值对象工厂是有差异的。值对象通常是不可变的对象，并且其所有必需的属性需要在创建时完成。当一个对象被创建之后，它必须是有效的，也是最终的，不会再发生变化。实体对象并非是不可变的。在被创建以后，它们可以通过设置某些属性

发生变化，但是需要保持所有的不变量。另一个差异源于实体对象需要标识符，而值对象不需要。

有时工厂是不需要的，一个简单的构造器就足够了。在如下情况下应该使用构造器：

- 构造过程并不复杂。
- 一个对象的创建不涉及到其他对象的创建，可以将所有需要的属性传递给构造器。
- 客户对实现很感兴趣，可能希望选择使用策略（Strategy）模式。
- 类是特定的类型，不存在到层级，所以不用在一系列的具体实现中进行选择。

另一个观察角度是工厂需要从无到有创建一个新对象，也或者它们需要对先前已经存在但可能已经持久化到数据库中的对象进行重建。将实体对象从它们所在的数据库中取回内存中，包含的是一个与创建一个新对象完全不同的过程。重建的新对象不需要一个新的标识，这个对象已经有一个标示符了，对不变量的违反也将区别对待。当从无到有创建一个新对象时，任何对不变量的违反都会产生一个异常。对于从数据库重建的对象，我们不能也这样处理。这个对象需要以某种方式加以修复，这样它们才能正常工作，否则就会有数据的丢失。

资源库

在模型驱动设计中，对象有一个生命周期，从被创建开始，直到被删除或者被归档结束。可以使用一个构造器或者工厂来负责对象的创建。创建对象的目的完全是为了使用它们。在一种面向对象的语言中，我们必须保持对一个对象的引用以便能够使用它。为了获得这样的引用，客户必须创建一个对象或者通过导航已有的关联关系从另一个对象中获得它。例如，为了从一个聚合中获得一个值对象，客户程序需要向聚合的根发送请求。问题是现在客户程序必须先拥有一个对根的引用。对大型的应用而言，这会变成一个问题，因为我们必须确保客户程序始终持有所需要对象的引用，或者持有另一个对象的引用，这个对象持有所需要对象的引用。在设计中使用这样的规则将强制要求对象持有一系列它们并不需要持有的引用。这增加了耦合性，创建了一系列并非真正需要的关联。

要使用一个对象，则意味着这个对象已经被创建完毕了。如果这个对象是聚合的根，那么它是一个实体，它会被保存为一个持久化的状态，可能是在数据库中，也可能是其他的持久化形式。如果它是一个值对象，可以通过导航一个关联关系从一个实体中获得它。实际上大量的对象都可以从数据库中直接获取到。这解决了获取对象引用的问题。当一个客户程序需要使用一个对象时，它可以访问数据库，从中检索出对象并使用它。这看上去是个非常快捷并且简单的解决方案，但它对设计会产生负面的影响。

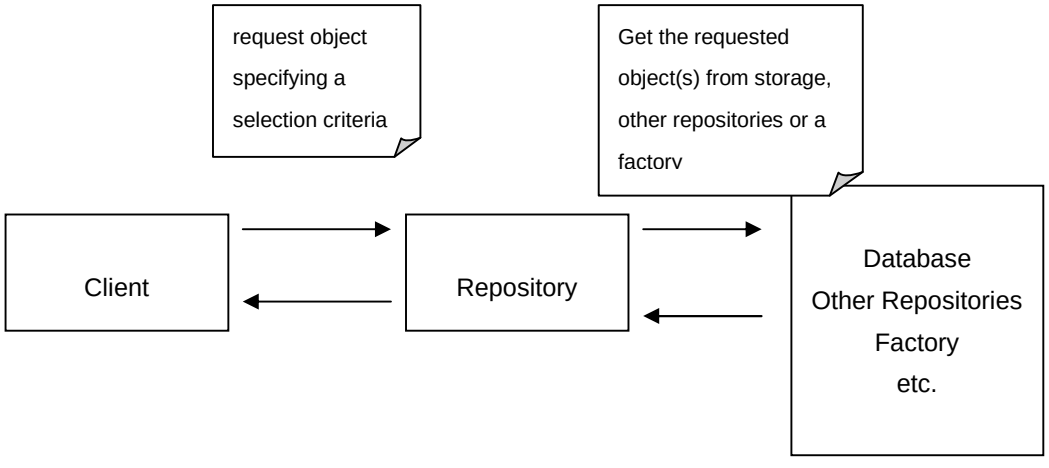
数据库是基础设施的一部分。一个糟糕的解决方案是客户程序必须知道访问数据库所需的细节。例如，客户需要创建 SQL 查询语句来检索想要的数据。数据库查询可能会返回一组记录，甚至会暴露出其内部更多的细节。当许多客户程序不得不直接从数据库创建对象时，会导致这样的代码扩散到整个模型中。从这点上讲领域模型遭受了损害。它必须处理大量基础设施的细节而不是处理领域概念。如果我们对底层数据库做出了变更，将会发生什么事情呢？所有扩散的代码都需要变更，以便能够访问新的存储。当客户代码直接访问一个数据库时，它极有可能会恢复聚合内部的一个对象。这样做会破坏聚合的封装性，带来未知的结果。

客户程序需要有一个实用的手段来获取已存在领域对象的引用。如果基础设施使得获得引用很容易，客户程序的开发人员可能会增加更多可导航的关联，导致模型混乱不堪。从另一方面讲，他们可能使用查询从数据库中获取所需的数据，或者拿到几个特定的对象，而不是通过聚合的根来导航。领域逻辑分散到查询和客户代码中，实体和值对象变得仅仅是数据的容器。应用了众多数据库访问基础设施，技术复杂性会迅速蔓延在客户代码中，开发人员降低了领域层的重要性，所做的工作跟模型无关了。总体的结果是丢失了对领域的专注，设计遭受了损害。

因此，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。领域对象不需处理基础设施，以得到领域中对其他对象的引用。只需要从资源库中获取它们，于是模型重获它应有的清晰和专注。

资源库会保存对某些对象的引用。当一个对象被创建之后，它可以被保存到资源库中，可以从资源库中获取到以备后续使用。如果客户程序从资源库中请求一个对象，而资源库中不存在，就会从存储介质中获取它。不管怎样，资源库扮演了一个全局可访问对象的存储地点。

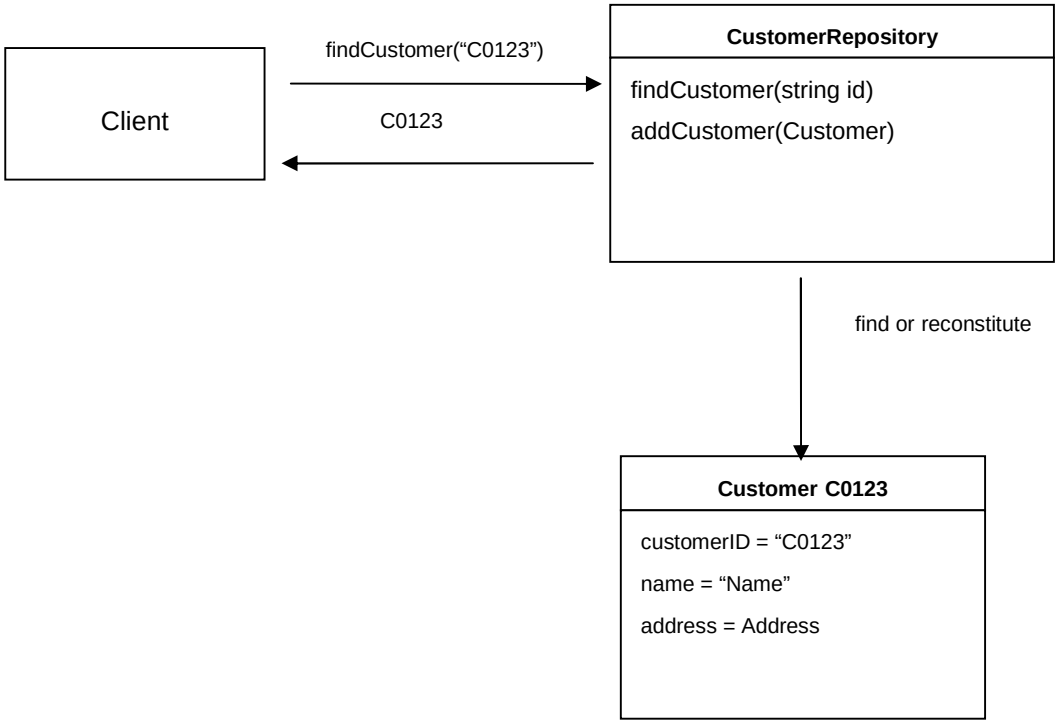
资源库可以包含一个策略。它可能基于特定的策略来访问某个或者另一个持久化存储介质。它可能会对不同类型的对象使用不同的存储位置。总体的结果是领域模型本身与需要保存对象或它们的引用、访问底层持久化基础设施实现了解耦。



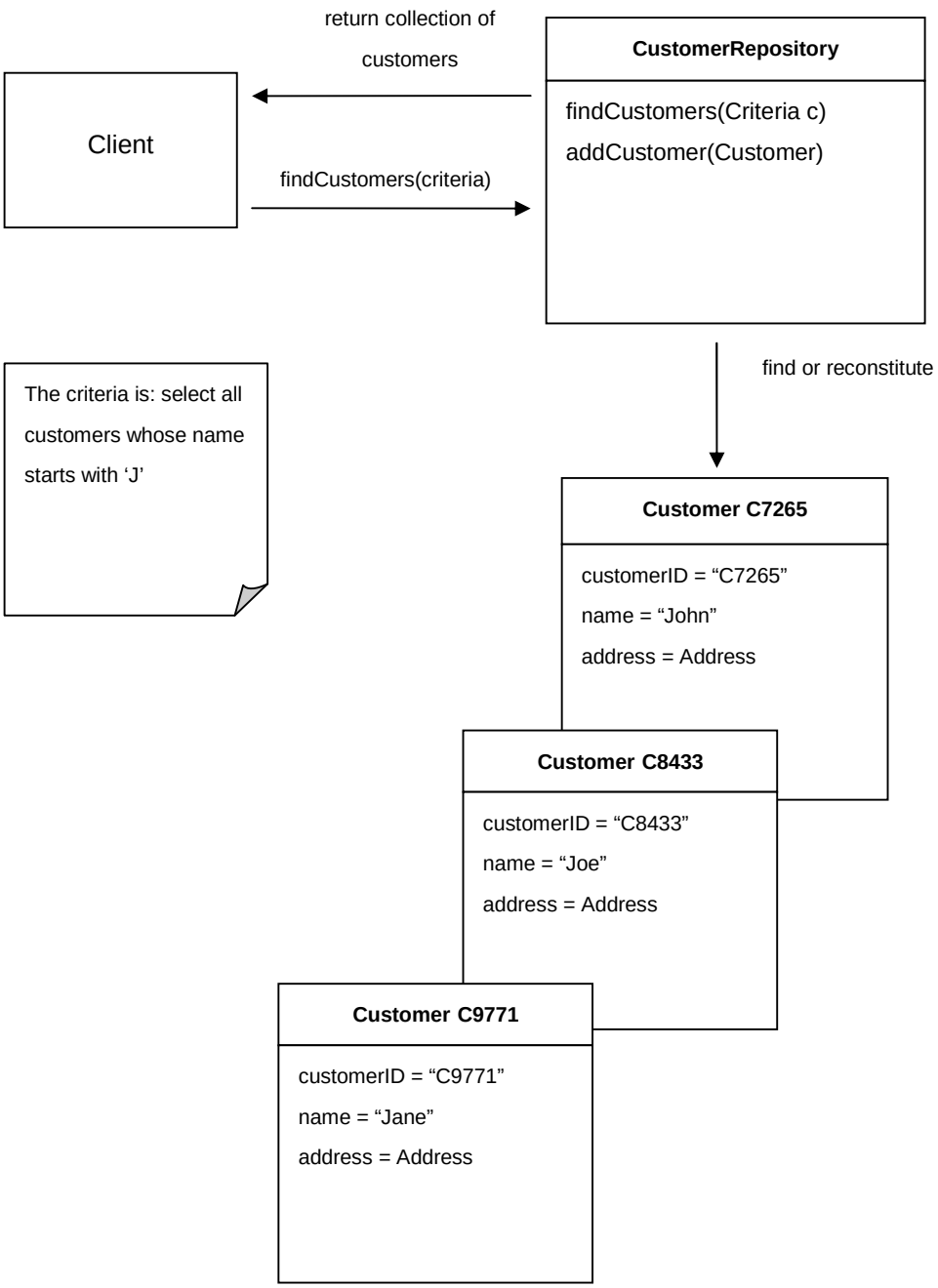
对于需要全局访问的每种类型的对象，创建一个对象来提供该类型所有对象都在内存中的假象。通过一个众所周知的全局接口来设置访问途径。提供方法来添加或者删除对象，封装向数据存储中插入或者删除数据的实际操作。提供基于某些条件选择对象的方法，返回属性值符合条件的完全实例化的对象或对象集合，从而封装实际的存储和查询技术。仅仅为真正需要直接访问的聚合根提供资源库。让客户程序保持对模型的专注，将所有的对象存储和访问细节都委托给资源库。

资源库可以包含用来访问基础设施的细节信息，但它的接口应该非常简单。资源库应该拥有一组用来检索对象的方法。客户程序调用这样的方法，传递一个或者多个代表筛选条件的参数用来选择一个或者一组匹配的对象。可以通过传递实体的标识符来轻易指定一个实体。其他筛选条件可能由一组对象属性构成。资源库将所有的对象与这组条件来比较，并返回符合条件的那些对象。资源库接口可能还包含用来执行某些辅助计算（例如获取特定类型对象的数量）的方法。

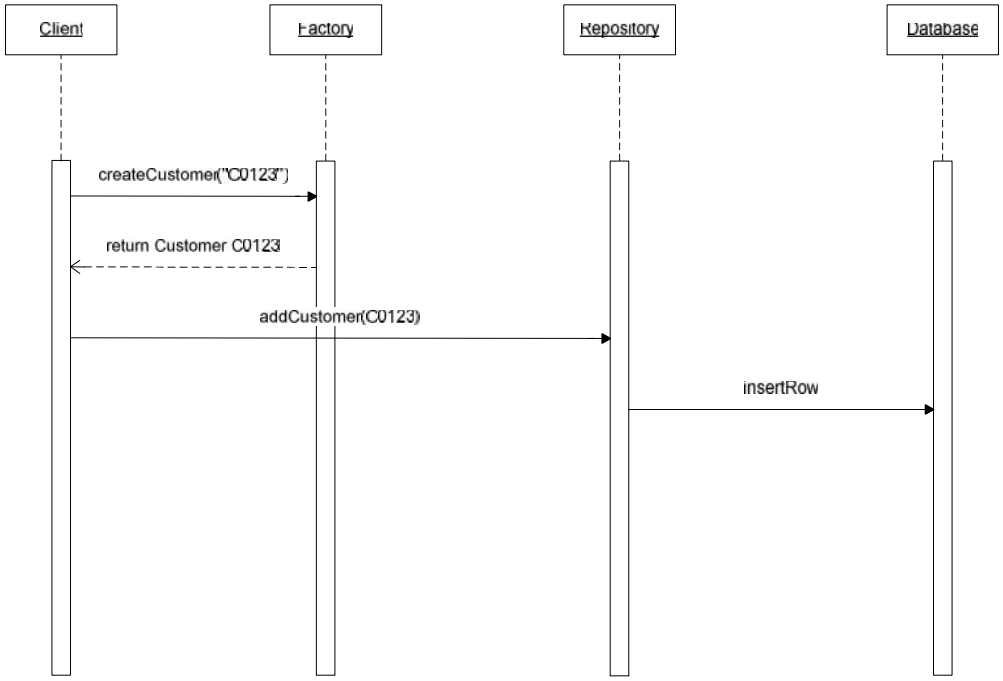
需要注意的是，资源库的实现可能会非常像是基础设施，然而资源库的接口却是纯粹的领域模型。



另一种选项是将筛选条件定义为一个规约（Specification）。规约允许定义更复杂的条件，见下图：



工厂和资源库之间存在一定的关系。它们都是模型驱动设计中的模式，它们都能帮助我们管理领域对象的生命周期。然而工厂关注的是对象的创建，而资源库关注的是已经存在的对象。资源库可能会在本地缓存对象，但更常见的情况是需要从一个持久化存储中检索它们。对象可以通过构造器创建，也可以通过一个工厂来构建。出于这个理由，资源库也可以被看作是一个工厂，因为它会创建对象。然而它不是从无到有创建新的对象，而是重建已有的对象。我们不应该将资源库与工厂混合在一起。工厂应该用来创建新的对象，而资源库应该用来发现已经创建的对象。当一个新对象被添加到资源库时，它应该是先由工厂创建好的，然后它应该被传递给资源库，由资源库来保存它，见下面的例子：



另外要注意的是工厂是“纯的领域”，而资源库会包含到基础设施的连接，例如数据库。

第 4 章

面向深层理解的重构

持续重构

迄今为止我们已经讨论过了领域以及创建一个能够表达领域的模型的重要性。我们给出了一些可用来创建有价值模型的技术指南。模型必须与它所源自于的领域紧密关联。我们也说过，代码设计应该围绕模型开展，模型自身应该基于设计决定而有所改善。脱离了模型的设计会导致软件无法真实表达它所服务的领域，很可能会得不到期望的行为。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，并且对于所用的技术而言可能不太适合。

在设计和开发过程中，我们时不时需要停下来，查看一下代码。这意味着到了重构的时间了。重构是不改变应用的行为而重新设计代码使得它更好的过程。重构通常是非常谨慎的，按照小幅且可控的步骤进行，这样我们就不会破坏功能或者引入一些bug了。毕竟，重构的目的是让代码更好而不是更坏。自动化测试可以为我们提供很大帮助，确保我们没有破坏任何事情。

代码重构有很多种方式，甚至存在重构的模式。这些模式代表了一个重构的自动化方法。基于这些模式的一些工具可以让开发人员的生活比以前更容易。缺少了那些工具的支持，重构工作会非常困难。这类重构更多是与代码和它的质量有关的。

还有另一种类型的重构，与领域和它的模型相关。有时会对领域有新的理解，有些事物变得更加清晰，或者发现了两个元素间的关系。所有的这些会通过重构工作被包括到设计中。得到容易阅读和理解的、有表现力的代码是非常重要的。通过阅读代码，一个人应该不仅仅能够了解代码是做什么的，同时了解它为什么要这样做。只有这样才能让代码真正捕获模型的实质。

基于模式的技术性重构，可以被组织并结构化。面向更深层理解无法按照同样的方式进行，我们不能为它创建模式。模型的复杂性和模型的可变性，使我们不可能按照机械的方式进行建模。一个优秀的模型产生于深层的思考、理解、经验和天分。

我们被教教授的关于建模的第一件事是阅读业务规范，从中寻找名词和动词。名词被转换成类，而动词则转换成方法。这是一种简化，将产生浅层次的模型。所有的模型开始时都缺乏深度，但我们可以面向越来越深的理解来重构模型。

设计必须灵活，僵硬的设计很难做重构。在编写者的头脑中若没有代码灵活性的概念，其编写出的代码就会很难维护。当需要发生变更时，你会看到代码在与你搏斗，原本应该很容易重构的事情，你却要花费很多时间。

与一致的语言一道使用经过验证的基础构造块，这样做会使得开发工作在某种程度上保持明智。这带来了一个挑战：如何发现一个深刻的模型（incisive model），这个模型能够捕获到领域专家头脑中微妙的概念，并且以此来驱动实际的设计。一个忽略肤浅的表面内容且捕捉到基本内容的模型是一个深层模型（deep model）。这会让软件更加与领域专家的思路合拍，也更能满足用户的需要。

从传统意义上讲，重构描述的是出于技术动机的代码转换。重构的动机同样可以出于对领域的深入理解，以及对模型及其代码表达进行相应的改进，。

除非使用迭代的重构过程，加上领域专家和开发人员一起密切关注对领域的学习，否则一个复杂成熟的领域模式是很难开发出来的。

凸现关键概念

重构是小幅度进行的，其结果也必然是一系列小的改进。有时，会有很多次小的变更，给设计增加的价值很小，有时，会有很少的变更，但却会造成很大的差异。这就是突破。

我们会从一个粗糙的、浮浅的模型开始，然后基于对领域的更深入的理解、对关注点更好的理解来改进这个模型和基于模型的设计。我们会为模型添加新的概念和抽象，然后对基于模型的设计做重构。每一次改进都会让设计更加清晰。这为取得突破创建了必要的前提。

突破常包括思维上的变化，如同我们理解模型一样。它也是项目中取得巨大进展的源泉，然而它也有一些缺点。突破可能隐含了大量的重构。这意味着需要时间和资源，而这两者看上去从来都不够。突破也是有风险的，因为大量的重构会在应用中引入行为上的变化。

为达到一次突破，我们需要将隐含的概念显现出来。当我们跟领域专家交谈时，我们交换了大量的想法和知识。某些概念成为了通用语言的一部分，但也有一些概念在起初未被重视。它们是隐含的概念，用来解释已经在领域中的其他概念。在改进设计的过程中，一些隐含的概念吸引了我们注意力。我们发现这些概念在设计中担任了重要的角

色。因此我们需要将这些隐含的概念显现出来。我们应该为它们创建类和关系。当这种情况出现时，我们就拥有了突破的机会。

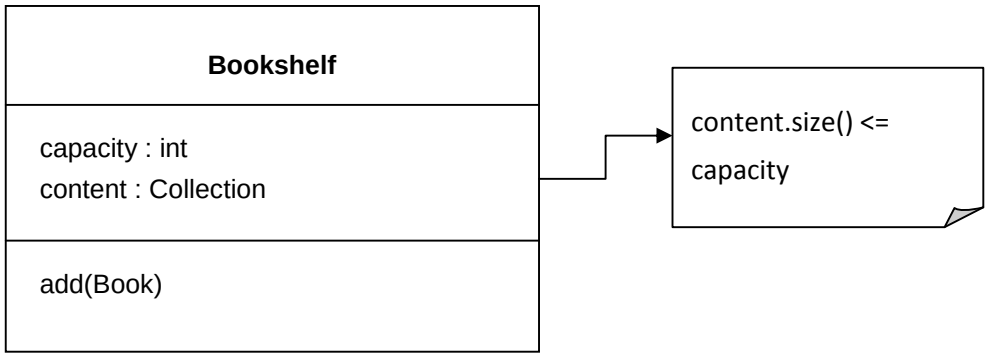
隐含的概念可能不会仅于此。如果它们是领域概念，它们应该被引入到模型和设计中。我们应该如何识别它们呢？第一种发现隐含概念的方式是倾听用到的语言。我们在建模和设计过程中使用的语言中包含了大量关于领域的信息。起初可能不会很多，或者某些信息没有被正确地使用。某些概念可能无法被完全理解，甚至理解是完全错误的。这是在学习一个新的领域所必须经历的一部分。但因为我们建造了我们的通用语言，关键概念会被加入其中。我们应该从那里开始查找隐含的概念。

有时设计的一些部分可能不会那么清晰，有一组关系让路径的计算变得难以进行，或者其过程会复杂到难以理解。这些部分在设计中显得十分笨拙，但这也是寻找隐藏的概念的绝佳之所，可能我们错过了什么。如果某个关键概念在破解谜团时缺失了，其他的事物就不得不替代它完成它的功能。这会让某些对象变胖，给它们增加了一些本不应该属于它的行为。设计的清晰度受到了损害。努力寻找是否有缺失的概念，如果找到一个，就将它显现出来。对设计做重构，让它更简单、更具灵活性。

当我们构建知识时很可能会遇到矛盾的情况。某个领域专家所讲的看上去与另一个领域专家所持的观点发生了矛盾。一个需求可能看上去与另一个需求矛盾。有一些矛盾其实不是真正的矛盾，只是因为看待同一事物的方式不同，或者只是因为在讲解时缺乏精确度造成的。我们应该努力去解决矛盾，有时这确实会帮助我们发现重要的概念。即使并没有发现它们，能够保持所有事物清晰也是很重要的。

挖掘模型概念的另一种明显的方式是使用领域文献。现在有众多为几乎任何可能的主题而编写的书，它们包含了大量关于特定领域的知识。这些书通常不包含所介绍领域的模型，它们包含的信息需要进一步处理、提炼和改进。但是，在书中发现的信息是有价值的，会给我们提供对领域的深层视图。

在将概念显现出来时，还有其他一些非常有用的概念：约束、过程和规约。约束是一个很简单的表达不变量的方式。无论对象的数据如何变化，不变量都要得到保持。简单的实现方式是将不变量的逻辑放在一个约束中。下面是一个简单的例子，其目的是为了解释这个概念，而不是为了描述对相似情况的建议解决方法。



我们可以向一个书架添加书籍，但是我们永远不能添加超过它容量的部分。这个约束可以被视为书架行为的一部分，见下面的 Java 代码。

```
public class Bookshelf {
    private int capacity = 20;
    private Collection content;
    public void add(Book book) {
        if(content.size() + 1 <= capacity) {
            content.add(book);
        } else {
        throw new IllegalOperationException(
            “The bookshelf has reached its limit.”);
            }
       }
}
```

我们可以重构这个代码，将约束提取为一个单独的方法。

```
public class Bookshelf {
    private int capacity = 20;
    private Collection content;
    public void add(Book book) {
        if(isSpaceAvailable()) {
            content.add(book);
        } else {
        throw new IllegalOperationException(
“The bookshelf has reached its limit.”);
       }
       }
       private boolean isSpaceAvailable() {
        return content.size() < capacity;
       }
}
```

将约束置于一个单独的方法，将它显现出来，这样做有很多优点。它很容易阅读，并且每个人都会注意到 add()方法服从于这个约束。如果约束变得更为复杂，还有空间可以向该方法添加更多的逻辑。

处理过程（process）通常在代码中被表达为 procedure。从我们开始使用面向对象语言后我们就不再用一种过程化的方法，所以我们需要为处理过程选择一个对象，然后给它添加行为。最好的实现过程的方式是使用服务。如果还有其他的实现处理过程的不同

方式，我们可以将其算法封装在一个策略对象中。并不是所有的处理过程都必须要显现出来。如果通用语言中提到了某个处理过程，那就应该将它显现出来。

我们在此要介绍的最后一个将概念显现出来的方法是规约。简单来说，规约是用来测试一个对象是否满足特定条件的。

领域层包含了应用到实体和值对象上的业务规则。那些规则通常与它们要应用到的对象合成一体。其中的一些规则仅仅是一组答案为“是”和“否”的问题，这样的规则可以被表达为一系列在布尔值上执行的逻辑操作，最终的结果也是一个布尔值。一个这样的例子是在一个客户对象上执行测试，看他是否有资格获得特定的贷款。这个规则可以被表达为一个方法，起名叫 isEligible()，并且可以附加在客户对象上。但这个规则不是一个严格地基于客户数据执行操作的简单方法。评估规则涉及到验证客户的信用，检查他过去是否偿还过他的债务，检查他是否具有未清余额（outstanding balance）等。这样的业务规则可能会很大、很复杂，是的对象的功能膨胀，不再满足其原始的目的。在这种情况下，我们可能会尝试将整个规则移动到应用层，因为它看上去已经超越了领域层了。实际上，重构的时候到了。

规则应该被封装到其自身的一个对象中，这将成为客户的规约，并且被保留在领域层中。新的对象将包含一组布尔方法，这些方法用来测试一个客户对象是否有资格获得贷款。每一个方法承担了一个小的测试功能，通过组合所有的方法，可以给出某个原始问题的答案。如果业务规则没有被包含在一个规约对象中，对应的代码会散布到很多对象中，难以保证规则的一致性。

规约用来测试对象是否满足某种需要，或者他们是否已经准备好达成某种目的。它也可以被用来从集合中筛选出一个特定的对象，或者作为某个对象的创建条件。

通常情况下，单个规约负责检查一个简单的规则是否得到了满足，若干个这样的规约组合在一起，表达一个复杂的规则，例如：

```
Customer customer =
customerRepository.findCustomer(customerIdentiy);
…
Specification customerEligibleForRefund = new Specification(
      new CustomerPaidHisDebtsInThePast(),
      new CustomerHasNoOutstandingBalances());
if(customerEligibleForRefund.isSatisfiedBy(customer) {
   refundService.issueRefundTo(customer);
}
```

测试简单的规则很简单，只需阅读这段代码，这段代码的含义很明显，表示了一个客户是否有资格获得偿还资金。

第 5 章

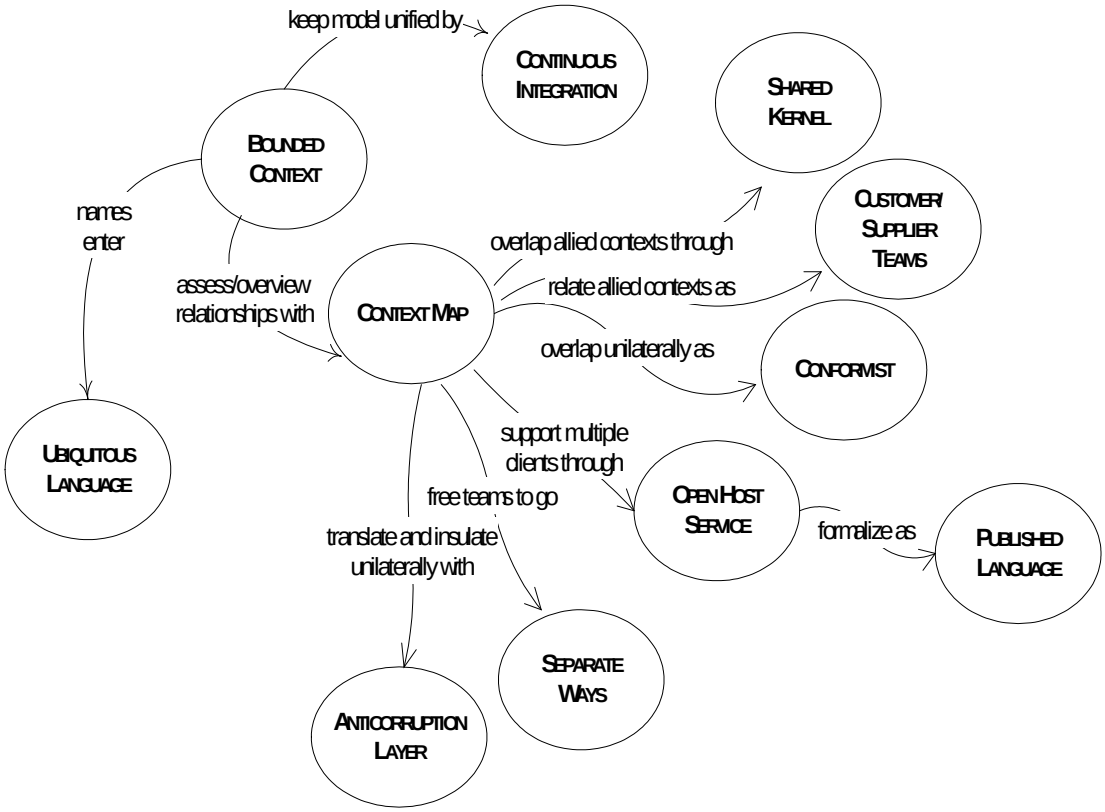
保持模型的一致性

本章涉及的是需要多个团队通力配合的大型项目。当存在多个团队，有不同的管理和协作时，我们会面对一系列不同的挑战。企业级项目通常都是大型项目，需要使用多种的技术和资源。这样的项目的设计仍然应该基于一个领域模型来开展，并且我们需要采用适当的度量（measure）以确保项目的成功。

当多个团队开发一个项目时，代码开发是并行完成的，每个团队被分配了模型的一个特定部分。那些部分不是独立的，相互之间存在或多或少的关联。它们都是从一个大的模型出发，然后实现其中的一部分。假设某个团队创建了一个模块，然后提供给其他的团队使用。另一个团队的开发人员开始使用这个模块，但发现还缺少一些自己模块需要的功能，于是他添加了这个功能并提交了代码，以便所有的人都能使用。但是他可能没有意识到，这其实是对模型的一个变更，这个变更有可能会破坏应用的功能。这种情况很容易发生，因为没有人会花时间去完全理解整个模型。每个人都知道自己的后院里有什么，但对其他地方却并不是非常了解。

从一个良好的模型开始，发展到后来却变成了一个不一致的模型，这种情况很容易出现。模型的首要需求是：模型必须是一致的，保持不变的术语，并且没有矛盾。模型内部的一致性被称为“统一”（unification）。一个企业项目应该有一个模型，涵盖企业的整个领域，没有矛盾和重叠的术语。一个统一的企业模型是难以达到的理想状态，有的时候甚至都不值得去尝试。这些项目需要很多团队的通力协作。在开发过程中，团队需要高度的独立性，因为他们没有时间去经常开会和讨论设计。要协调这些团队是一项非常艰难的任务。也许他们属于不同的部门，有着独立的管理。当模型的设计在局部独立进化时，我们就面临着失去模型完整性的可能。努力为整个企业项目维护一个大的统一模型，以获得模型完整性，这个办法也不会有什么作用。解决方案并非显而易见，因为它与我们目前已经学到所有知识恰恰相反。不是试图保持一个迟早要四分五裂的大模型，我们应该做的是有意识地将大模型分解成多个较小的模型。只要遵守它们所绑定的契约，良好整合的小模型能够独立进化。每个模型都应该有一个清晰的边界，模型之间的关系也应该被精确地定义。

我们将会介绍一组技术，用来维护模型的完整性。下面的图展示了这些技术，以及它们之间的关系。



界定的上下文

每一个模型都有一个上下文。在我们处理一个独立的模型时，上下文是隐含的，我们不需要去定义它。当我们创建一个假定要和其他软件（例如一个遗留应用）交互的应用时，很明显新的应用有自己的模型和上下文，独立于遗留模型及其上下文。它们无法被合并、混合或者混淆起来。所以当我们开发大的企业应用时，我们需要为每一个我们创建的模型定义上下文。

采用多个模型，对任何大型项目都能起到作用。如果将基于明显不同模型的代码合并在一起，软件就会变得有很多 bug、不可靠而且很难理解。团队成员之间的沟通会容易产生混淆。模型不应该被应用于哪些上下文，通常不是非常明确。

如何将一个大的模型分解成小的模型没有什么固定的准则。尽量把那些相关联的以及能形成一个自然概念的元素放在一个模型里。模型应该足够小，以便能分配给一个团队去实现。团队协作和沟通会更加流畅，这会有助于开发人员共同完成一个模型。模型的上下文是一些条件的集合，这些条件可以确保应用在模型里的术语都有一个明确的含义。

主要的思想是定义模型的范围，定出它的上下文的边界，然后尽最大可能保持模型的统一。在模型跨越整个企业项目时，要保持它的纯洁是很困难的；但是在它被限定到一个特定区域时，要保持它的纯洁就容易得多。明确定义模型所应用的上下文，根据以

下因素来明确设置边界：团队的组织结构、应用的特定部分中的惯例、物理表现（例如代码库、数据库 Schema）。保持模型在这些边界里严格一致，不要因外界因素而产生干扰或混淆。

界定的上下文并不是模块。界定的上下文提供有模型在其中进化的逻辑框架。模块是被用来组织模型的元素，因此界定的上下文包含了模块。

当不同的团队不得不共同工作于一个模型时，我们必须小心不要踩到别人的脚（译者注：意思为各司其职，不越界）。要时刻意识到任何针对模型的变更都有可能破坏现有的功能。当使用多个模型时，每个人在自己的模型之上可以自由地工作。我们都知道自己模型的界限，都恪守在这些边界里。我们需要确保模型的纯洁、一致和统一。每个模型应能使重构尽可能容易，而不会影响到其他的模型。设计能够被改进和提炼，以达到最高的纯洁性。

有多个模型时总是会付出些代价。我们需要定义不同模型间的边界和关系。这需要额外的工作和设计努力，可能还有不同模型间的翻译。我们不能在不同模型间传递任何对象，也不能像是没有边界一样自由地调用行为。但这并不是一个非常困难的任务，而且带来的好处证明克服这些困难是值得的。

例如，我们要创建一个用来在互联网上卖东西的电子商务应用。这个应用允许客户注册，然后我们收集他们的个人数据，包括信用卡号码。数据保存在一个关系型数据库里面。客户被允许登录，通过浏览网站寻找商品，然后下单。不论在什么时候下单，应用都需要发布一个事件，因为必须有人邮寄请求的货物。我们还想做一个用于创建报表的报表界面，这样我们就能够监视可售卖货物的状态、哪些是客户感兴趣购买的、哪些是不受欢迎的等等。开始的时候，我们用一个模型涵盖整个电子商务的领域。这样做有很大的诱惑性，毕竟我们被要求创建一个大的应用。但是仔细考虑手头的任务之后，我们发现这个在线购物应用其实和报表关联度不大。它们有不同的关注点，操作不同的概念，甚至需要用到不同的技术。唯一共通的地方是客户和商品的数据都存储在数据库里，两个应用都会访问这些数据。

推荐的做法是为每一个领域创建一个独立的模型，一个是电子商务的模型，一个是报表的模型。它们两个可以在互不关注的情况下自由进化，甚至可以变成独立的应用。也许报表应用会用到电子商务应用应该存储在数据库里的一些特定数据，但多数情况下它们彼此独立发展。

还需要有个消息发送系统将订单信息通知仓库工作人员，这样他们就可以邮寄被购买的商品。邮寄人员也会用到一个应用，该应用可以提供给他们关于所购买的商品条目、数量、客户地址以及交付需求等详细信息。不需要使在线购物（e-shop）模型来涵盖两个活动领域。对在线购物而言，用异步消息的方式将包含购买信息的值对象发送给仓库，这样做要简单的多。这样就清楚地得到了两个可以独立开发的模型，我们只需要确保它们之间的接口工作良好就可以了。

持续集成

一旦界定的上下文被定义好，我们就必须保持它的完整性。但多人工作于同一个界定的上下文时，模型很容易碎片化。团队越大，问题就会越大，不过通常只有三四个人会遇到严重的问题。但是，系统被破坏成更小的上下文后，最终失去了完整性和一致性的价值。

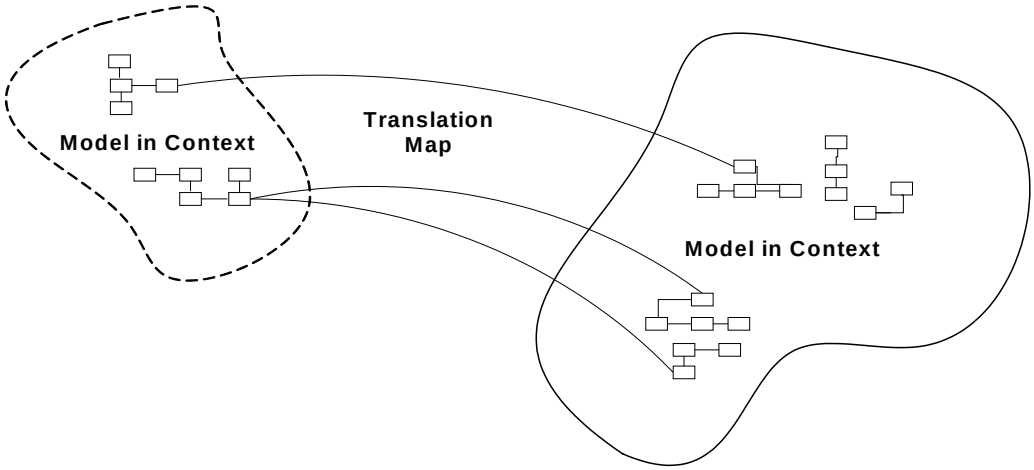
即使是只有一个团队工作于一个界定的上下文，也有犯错误的时候。在团队内部我们需要充分的沟通，以确保每个人都能理解模型中每个元素所扮演的角色。如果一个人不理解对象之间的关系，他就可能会以和原意完全相反的方式修改代码。如果我们不能百分之百地专注于模型的纯洁性，就会很容易犯这种错误。团队的某个成员可能会添加重复的代码，因为他不知道这些代码已经存在，或者因为担心破坏现有的功能而不去改变已有的代码，却选择了添加重复的代码。

模型不是一开始就被完全定义。而是先被创建，然后基于对领域新的理解和来自开发过程的反馈持续进化。这意味着新的概念会进入模型，新的元素也会被添加到代码中。所有的这些需求都会被集成进一个统一的模型，进而用代码来实现。这也就是为什么持续集成在界定的上下文中如此必要的原因。我们需要这样一个集成的过程，以确保所有新增的元素和模型原有部分能够和谐相处，在代码中也被正确地实现。我们需要有一个过程用来合并代码。合并得越早越好。对于单个小团队，推荐做每日合并。我们还需要有一个适当的构建过程（build process）。合并的代码需要自动地被构建，这样才能够被测试。另外一个必要的需求是执行自动化测试。如果团队有测试工具，并创建了一个测试套件，那么每次构建都可以运行测试，任何错误都可以被检测出来。而这时也可以较容易地修改代码以修正报告的错误，因为它们被发现的很早，然后合并、构建、和测试过程会重新开始。

持续集成是基于模型中概念的集成，然后通过测试来实现。任何模型的不一致性在实现中都会被检测出来。持续集成应用于界定的上下文，不会被用来处理相邻上下文之间的关系。

上下文映射

一个企业应用有多个模型，每个模型都有自己的界定的上下文。建议使用上下文作为团队组织的基础。在同一个团队里的人们能更容易地沟通，也能更好地将模型和实现集成。尽管每个团队都工作于自己的模型，最好让每个人都能了解总体的图景。上下文映射（Context Map）是描绘不同的界定上下文和它们之间关系的一份文档。它可以是像下面所展示的一个图表（diagram），也可以是其他任何形式的文档，细节层次可以有所不同。重要的是，要让每个在项目中工作的人都能够分享并理解它。

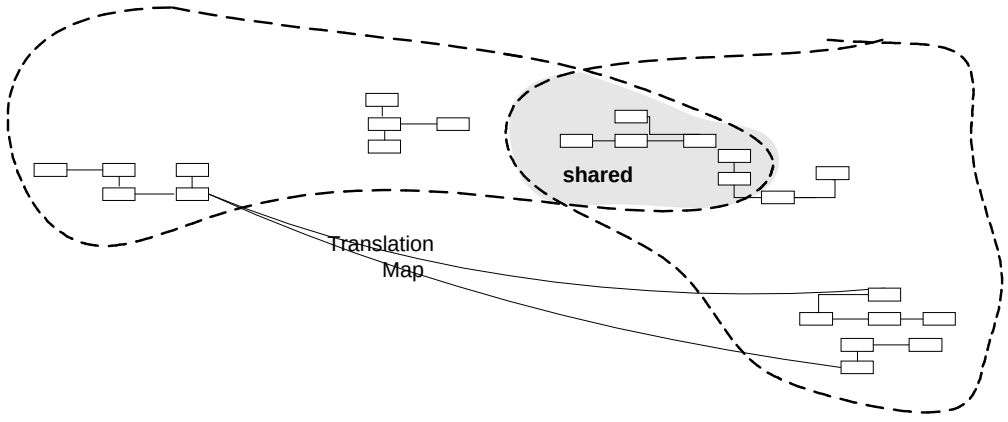


只有独立的统一模型还不够，它们还必须被集成在一起，因为每个模型的功能都只是整个系统的一部分。在最后，单个的部分要被组装在一起，整个的系统必须能正确地工作。如果上下文定义的不清晰，很有可能彼此之间互相覆盖。如果上下文之间的关系没有被描绘出来，在系统被集成的时候它们就有可能无法工作。

每个界定的上下文都应该有一个作为通用语言一部分的名字。当团队之间讨论整个系统的时候，这会对团队之间的沟通产生极大帮助。每个人也应该知道每个上下文的边界以及在上下文和代码之间的映射等。一个常用的做法是先定义上下文，然后为每个上下文创建模块，再用一个命名约定来指明每个模型所属的上下文。

在本章接下来的内容中，我们要讨论不同上下文之间的交互。我们会介绍一系列可用来创建上下文映射的模式，在这些映射中，上下文有清晰的角色，上下文之间的关系也被标出。共享内核（Shared Kernel）和客户-供应商（Customer-Supplier）是处理上下文之间的高级交互的模式。隔离通道（Separate Way）是在我们想让上下文高度独立和独立进化时要用到的模式。还有两个模式用来处理系统与一个遗留系统或一个外部系统之间的交互，它们是开放主机服务（Open Host Service）和防崩溃层（Anticorruption Layer）。

共享内核



当缺少功能集成时，持续集成的成本会显得过于高昂。尤其是在团队不具备相关的技能或者行政组织来维护持续集成，或者是单个团队又大又笨拙的时候。所以独立的界定上下文可能会由多个团队来定义和形成。

工作于紧密关联的应用程序上团队如果缺乏协调，有时会进展得很快，但他们的工作成果有可能会很难整合。他们在转换层（translation layers）和改造（retrofitting）上花费的时间比一开始就做持续集成会更多，做了许多重复劳动，失去了公共的通用语言带来的好处。

因此，需要指派两个团队同意共享的领域模型子集。当然除了包括模型的子集部分，还要包括模型相关联的代码或数据库设计的子集。这个明确被共享的东西有特殊的状态，在没有咨询另一个团队之前不能做修改。

要经常整合功能系统，但是可以不用像在团队内部进行持续集成那么频繁。在集成的时候，两个团队开发的测试都要运行。

共享内核的目的是减少重复，但是仍保持两个独立的上下文。对于共享内核的开发需要多加小心。两个开发团队都有可能修改内核的代码，还必须对所做的修改做集成。如果团队用的是内核代码的副本，那么要尽可能早地合并代码，至少每周一次。还应该使用一个测试套件，这样每一个针对内核的修改都能快速地被测试。内核的任何改变都应该与另一个团队进行沟通，并且通知相关团队，使大家都能了解新增的功能。

客户-供应商

有的时候两个子系统之间存在特殊的关系：一个子系统严重依赖另一个。两个子系统所在的上下文是不同的，并且一个系统的处理结果被作为另外一个的输入。它们没有共享的内核，因为有这样一个内核从概念上说是错误的，或者两个子系统要共享代码在技术上不可能实现。

让我们回到先前的例子。我们曾讨论了一个关于电子商务应用的模型，包括了报表和发送消息两部分内容。我们已经解释说最好为所有的上下文创建各自独立的模型，因为只有一个模型时会在开发过程中遇到瓶颈和资源的争夺。假设我们同意创建独立的模型，那么在 Web 商店子系统和报表子系统间之的关系是什么样子的呢？共享内核看上去并不是好的选择。子系统很可能会用不同的技术来实现。一个是纯浏览器的体验，而另一个则可能是一个富 GUI 应用。即使报表应用是用 Web 界面来实现，各自模型的主要概念也是不同的。也许会有重叠的情况，但还不足以应用共享内核。所以我们选择采用不同的做法。另外，在线购物子系统完全不依赖报表子系统。在线购物应用的用户是那些浏览商品并下单的 Web 客户。所有的客户、商品和订单数据被放在一个数据库里。就是这样。在线购物应用不会真的关心各自的数据发生了什么。而同时，报表应用非常关心和需要由在线购物应用保存的数据。它还需要一些额外的信息以执行它提供的生成

报表的服务。客户可能在购物篮里放了一些商品，但在结账的时候又去掉了。客户访问某个链接的次数可能多于其链接等。这样的信息对在线购物应用没有什么意义，但是对报表应用却意义重大。由此，供应商子系统不得不实现一些客户子系统需要的规约。这就是联系两个子系统的纽带。

另外一个需求与所用到的数据库相关，更确切地说，与它的 schema 相关。两个应用将使用同一个数据库。如果在线购物应用是唯一访问数据库的应用，那么数据库 schema 可以在任何时间被改变以反应它的需要。但是报表子系统也需要访问数据库，所以它需要数据库 schema 尽量保持稳定。在开发过程中，数据库 schema 一点也不能改变的情况是不可想像的。对在线购物应用来说，这并不代表是个问题，但对报表应用这肯定是一个问题。这两个团队需要沟通，可能他们不得不一起在同一个数据库上工作，然后决定什么时候可以执行变更。对报表子系统来说这会是一个限制，因为团队会倾向于随着开发的进展快速地执行变更，而不是等待在线购物应用。如果在线购物应用团队有否决权，他们也许会对要在数据库上执行的变更强制加上限制，从而伤害到报表团队的活动。如果在线购物团队能独立行动，他们迟早会破坏约定，然后执行一些报表团队还没有准备好的变更。当团队处于相同管理之下时， 这个模式非常有效，它会使决策过程变得容易，也能够产生默契。

当我们面对这样一个场景时，我们应该采取行动。报表团队应该扮演客户的角色，而在线购物团队应该扮演供应商的角色。两个团队应该定期碰面或者提出碰面邀请，像一个客户对待他的供应商那样交谈。客户团队应该介绍它的需求，而供应商团队根据需求制定计划。尽管客户团队的所有需求最后都必须得到满足，实现这些需求的时间表是由供应商团队来决定的。如果认为一些需求确实非常重要，那么应该先实现它们，延迟实现其他的需求。客户团队还需要由供应商团队分享的输入和知识。这个过程是单向的，但是在有的时候是必要的。

需要精确定义两个子系统之间的接口。另外还要创建一个顺从的测试套件，在关注任何接口需求的时候用来做测试。供应商团队能够在他们的设计上大胆地工作，因为接口测试套件构成的保护网会在任何有问题的时候报警。

在两个团队之间建立一个清晰的客户/供应商关系。在制定计划的过程中，让客户团队扮演和供应商团队打交道的客户角色。对满足客户需求的任务进行协商并做出预算，让每个人都理解相关的承诺和日程表。

联合开发可以对预期接口做验证的自动化验收测试。将这些测试添加到供应商团队的测试套件里，作为团队的持续集成过程的一部分运行。这个测试能使供应商团队放心地做修改，而不用担心会产生影响客户团队应用的副作用。

顺从者

在两个团队对彼此的关系都有兴趣时，客户-供应商关系是可行的。客户非常依赖于供应商，然而供应商却不依赖客户。如果有管理手段来保证合作的执行，供应商会给予客户需要的关注，并聆听客户的要求。如果管理手段没有清晰地界定在两个团队之间需要完成什么，或者管理很糟糕，或者缺乏管理，供应商慢慢地会更加关注它的模型和设计，对帮助客户不再感兴趣。毕竟他们有自己的 deadline。即使他们是好人，愿意帮助其他团队，时间的压力却不允许他们这么做，客户团队会受到损害。在团队属于不同公司的情况下，这样的事情也会发生。沟通是困难的，供应商的公司也许没兴趣在维持关系上投资太多。他们要么提供零星的帮助，或者直接拒绝合作。结果是客户团队孤立无援，只能尽自己的努力摸索模型和设计。

当两个开发团队有客户-供应商关系，而且供应商团队没有动力为客户团队提供需要的帮助时，客户团队是无助的。利他精神也许会使供应商开发者做出许诺，但是他们很有可能会完成不了。相信这些美好的许诺，会导致客户团队根据那些从来都不会实现的功能来制定计划。客户的项目会被一直耽搁，直到团队最终学会如何处理这种情况。符合客户团队需要的接口注定不会实现。

客户团队没有多少选择。最明显的做法是将它与供应商分离开，完全自力更生。在后面的“隔离通道”模式中我们再对它做详细介绍。有时供应商子系统提供的好处不值得所付出的努力。创建一个独立的模型，并且在不必考虑供应商模型的情况下做设计也许更简单些。但这样做并不总是管用。

有时候供应商的模型会有一些价值，这时不得不维持一个连接。但是因为供应商团队不会帮助客户团队，所以后者不得不采取一些措施来保护自己，以防止前者对模型所做变更带来的影响。他们需要实现连接两个上下文的转换层。也有可能供应商团队的模型没有被很好地构思，导致其实现非常糟糕。虽然客户上下文仍然可以使用它，但是它应该通过使用一个我们后面要讨论的“防崩溃层”来保护自己。

如果客户不得不使用供应商团队的模型，而且这个模型做得很好，那么就需要顺从这个模型了。客户团队遵从供应商团队的模型，完全顺从它。这和共享内核很相似，但有一个重要的不同之处。客户团队不能对内核做更改。他们只能将它作为自己模型的一部分，可以在所提供的现有代码上完成构建。在很多情况下，这种方案是可行的。当有人提供一个丰富的组件，并提供了一个访问该组件的接口时，我们就可以建造包括了这个组件的模型，就好像这个组件是我们自己的东西。如果组件有一个小的接口，那么最好只为它简单地创建一个适配器，在我们的模型和组件模型之间做转换。这会将我们的模型隔离出来，可以有很高的自由度去开发它。

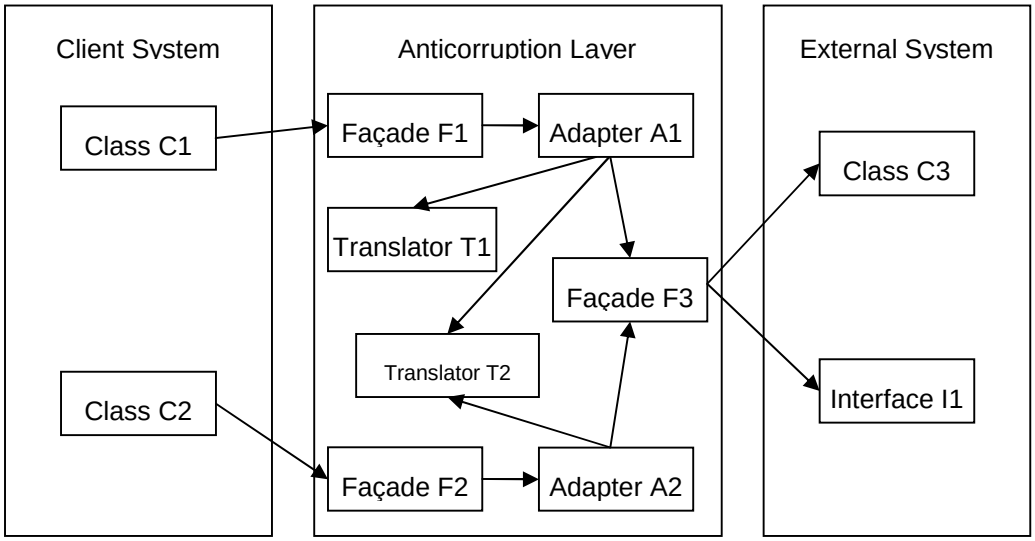
防崩溃层

我们会经常遇到以下情况：所创建的新应用需要与遗留软件或者一个独立应用交互。对领域建模者而言，这又是一个挑战。很多遗留应用从前没有用领域建模技术来构建，而且它们的模型模糊不清、盘根错节，难于理解，也难以使用。即使做得很好，遗留应用的模型对我们也没有多大作用，因为我们的模型很可能与它完全不同。尽管如此，在我们的模型和遗留模型之间就必须有一个集成层，这也是使用老旧应用的需求之一。

我们的客户端系统和一个外部系统交互有很多种方法。一种方法是通过网络连接，两个应用需要使用相同的网络通信协议，客户端需要遵从外部系统所使用的接口。另一种交互的方法是通过数据库。外部系统使用存储在数据库里的数据，客户端系统被假定访问同样的数据库。在这两个案例中，我们所处理的都是在两个系统之间传输的原始数据。这看上去相当简单，然而事实是原始数据不包括任何关于模型的信息。我们不能将数据从数据库中取出来，全部作为原始数据来处理。在这些数据后面隐藏着大量的语义。一个包含有与原始数据相关的其他原始数据的数据库，构成了一个关系网。数据的语义非常重要，并且需要被充分考虑。客户端应用不能在不理解被使用数据含义的情况下就访问数据库并执行写操作。我们看到外部模型的一些部分被反映在数据库里，然后影响了我们的模型。

如果我们允许这样的事情发生，那么就会存在外部模型修改客户端模型的风险。我们不能忽视与外部模型的交互，但是我们也应该小心地将我们的模型与外部模型隔离开来。我们应该在我们的客户端模型和外部模型之间建造一个防崩溃层。从我们模型的观点来看，防崩溃层是模型天然的一部分，并不像一个外来的东西。它操作的是与我们的模型相似的概念和动作，但是防崩溃层使用外部语言与外部模型交流，而不是客户端语言。这个层在两个领域和语言之间扮演双向转换器，它最大的好处在于可以使客户端模型保持纯洁和一致，不会受到外部模型的污染。

我们应该如何来实现防崩溃层？一个非常好的解决方案是将这个层看作来自客户端模型的一个服务。使用服务是非常简单的，因为它抽象了其他系统并让我们以自己的术语来定位它。服务会处理所需要的转换，所以我们的模型可以保持绝缘。考虑到实际的实现，可以将服务实现为一个 Facade（参见 Gammaetal 在 1995 年写作《设计模式》）。除了这一点，防崩溃层最有可能还需要一个适配器（Adapter）。适配器可以使你将一个类的接口转换成客户端能够理解的另一个接口。在我们的这个案例中，适配器不需要一定包装一个类，因为它的工作是在两个系统之间做转换。



防崩溃层也许包含多个服务。每一个服务都有一个相应的 Facade，对每一个 Facade 我们为之添加一个适配器。我们不应该为所有的服务使用一个适配器，因为这样会将很多功能混在一起，从而导致杂乱无章。

我们必须再添加一个组件。适配器将外部系统的行为包装起来。我们还需要对象和数据转换（object and data conversion），可以使用一个转换器（translator）来完成这个任务。它可以是一个非常简单的对象，有很少的功能，满足数据转换的基本需要。如果外部系统有一个复杂的接口，最好在适配器和接口之间再添加一个额外的 Facade。这会简化适配器的协议，将它和其他系统分离开来。

隔离通道

到目前为止，我们尝试找到集成子系统的各种方式，使它们以一种能够使模型和设计保持完好的方式协同工作。这需要付出努力和做出妥协。工作于各自子系统的团队需要花费大量时间理清子系统之间的关系，需要持续不断地合并他们的代码，执行测试以确保没有破坏任何东西。有的时候，某个团队还需要花费很多时间去实现其他团队需要的一些需求。还需要做出一些妥协。独立做开发，自由地选择概念和关联是一回事，而确保你的模型适合另一个系统的框架则是另一回事。我们有可能会为了能和另一个子系统协同，而修改模型。或者有可能需要引入特殊的层，在两个子系统之间做转换。有时我们必须这样做，但有时也可以采用其他做法。我们需要严格地评估集成的价值，只在做这件事确实有价值时才去做。如果我们得出的结论是集成的难度很大，不值得这样去做，那么就应该考虑隔离通道。

隔离通道模式适合于以下情况：一个企业应用可由几个较小的应用组成，而且从建模的角度来看彼此之间有很少或者没有公共之处。它有一组自己的需求，从用户角度看这是一个应用，但是从建模和设计的观点来看，它可以由具有不同实现的独立模型来完成。我们应该查看一下需求，思考一下它们是否可以被划分成两个或者多个几乎没有相

通之处的部分。如果可以这样做，那么我们就创建独立的界定上下文（Bounded Context），并独立建模。这样做的好处是可以自由地选择用来实现模型的技术。我们正创建的应用可能会共享一个公共的瘦 GUI，作为带有链接或按钮的一个门户来访问每一个应用。相对于集成后端的模型，将应用组织在一起是一个较小的集成。

在采用隔离通道模式之前，我们需要确信我们将不会回到一个集成的系统。独立开发的模型是很难做集成的，它们的相通之处很少，不值得这样做。

开放主机服务

当我们试图集成两个子系统时，通常要在它们之间创建一个转换层。这个层在客户端子系统和我们想要集成的外部子系统之间扮演了缓冲的角色。这个层可以是始终如一的，这要看关系的复杂度和外部子系统是如何设计的。如果外部子系统不是被一个客户端子系统使用，而是被多个子系统使用的话，我们需要为所有的子系统创建转换层。所有的这些层都会重复相同的转换任务，也会包含相似的代码。

当一个子系统要和其他很多子系统集成时，为每一个子系统定制一个转换器会使整个团队陷入困境。会有越来越多的代码需要维护，当做出变更时，有越来越多的事情需要担心。

这个问题的解决方案是，将外部子系统看作服务提供者。如果我们能为这个系统封装一组服务，那么所有的其他子系统将会访问这些服务，我们也就不需要任何转换层。难点在于每一个子系统也许需要以一种特殊的方式和外部子系统交互，那么要创建一组一致的服务可能会比较麻烦。

定义一个能以一组服务的形式访问你的子系统的协议。将这个协议开放出来，使得所有需要和你做集成的人都能使用它。然后增强和扩展这个协议，使其能够处理新的集成需求，但某团队有特殊需求时除外。对于特殊的需求，使用一个一次性的转换器来增强协议，从而使得共享的协议保持简单和一致。

提炼

提炼是从一种混合物中分离出其组成物质的过程。提炼的目的是从混合物中提取出某种特殊的物质。在提炼的过程中，可能会得到某些副产品，它们可能也是有价值的。

即使在我们改进和创建很多抽象之后，一个大的领域还是会有一个大的模型。就是在做了很多次重构之后，模型依然会很大。对于这样的情况，就需要做一次提炼了。其思路是定义一个代表领域本质的核心域（Core Domain）。提炼过程的副产品将是包含了领域中其他部分的普通子域（Generic Subdomain）。

在设计一个大型系统时，有那么多分布式组件，所有的都是那么复杂而且绝对必须不出差错，而领域模型的基本内容，也就是真正的业务资产（business asset），却变得模糊不清和不受重视。

当我们处理一个大的模型时，应该尝试将基本概念与普通概念分离开来。一开始我们曾经举过一个关于飞机空中交通监控系统的例子。我们说飞行计划包含了飞机必须遵循的设计好的路线。在这个系统里，路线好像是一个无时不在的概念。实际上，这个概念是一个普通的概念，不是一个基本概念。在很多领域里都会用到路线概念，可以设计一个普通的模型来描述它。空中交通监控的基本内容是在其他地方。监控系统知道飞机应该遵循的路线，但是它还会接收来自对飞行中飞机做跟踪的雷达网络的输入。这个数据显示飞机真正遵循的飞行路线，而它经常和预先描述好的路线有些偏差。系统必须基于飞机当前的飞行参数、飞机的特性和天气情况来计算飞行轨道。这一轨道是一个能完全描述飞机当前飞行路线的四维路线，它可能会在接下来的两分钟里被计算出来，也可能是几十分钟，或者是两小时。每一个计算都有助于决策制定过程。计算飞机轨道的目的是看看是否有可能与其他飞机的飞行路线产生交叉。在机场附近，在飞机起飞或者降落时，有很多飞机在空中盘旋或者移动。如果一个飞机偏离了它的计划路线，很有可能会和其他飞机相撞。空中交通监控系统会计算飞机的轨道，在路线出现交叉的可能性时发出警报。空中交通控制人员需要快速做出决策，指挥飞机改变飞行路线，以避免相撞发生。飞机之间相距越远，计算轨道的时间就越长，做出反应的时间也越长。根据已有的数据合成飞机轨道的模块才是这个业务系统的核心。可以将这个模块标记为核心域。而路线模型应该作为一个普通域。

系统的核心域是什么，取决于我们如何看待系统。一个简单的路线系统会将路线和与它相关的概念看作核心域，而空中交通监控系统却将路线看作普通子域。一个应用的核心域有可能会变成另一个应用的普通子域。正确标识核心域，以及它与模型其他部分之间的关系是非常重要的。

对模型做提炼，找到核心域，提供一种手段将核心域与支持模型和代码（mass of supporting model and code）容易地区分开来。对最有价值的和专业的概念加以强调。尽量使核心域小一些。

将你们的顶尖人才投入到核心域上，并且开展所需的招聘。在核心域上付出努力，找到一种深层的模型，开发一个足够灵活的设计以实现系统的远景。根据其他部分如何支持提炼出的核心域，判断这些部分投资价值。

分配最好的开发人员去承担实现核心域的任务是很重要的。开发人员经常沉溺于技术，喜欢学习最好的和最新的编程语言，相对于业务逻辑他们更关注基础设施。一个领域的业务逻辑在他们看来是令人厌倦的，也看不到什么回报。还记得我们前面所谈到的飞机轨道案例的要点吗？当项目完成时，所有的知识都成为几乎毫无益处的过去事物。但是领域的业务逻辑是业务的核心所在，这个核心的设计和实现中如果存在错误，将会

导致整个项目遭到抛弃。如果核心业务逻辑不起作用，所有技术上的亮点都等于做了无用功。

核心域的创建无法在一步完成。这需要有一个改进的过程，在核心越来越清晰之前必须开展持续的重构。我们需要强化将核心作为设计工作的中心的地位，并划定它的边界。我们还需要根据其与新核心的关系来重新考虑模型的其他元素，也许它们也需要被重构，一些功能也许需要改变等。

模型的某些部分会增加复杂度，对于捕获和交流专业知识没有帮助。任何题外的事物都会使核心域难于辨认和被理解。因为广为人知的普遍原则（general principles everyone knows），或者那些并非是你首要关注而是扮演支持角色的细节都会影响模型。但是这些其他元素对系统功能和模型的完整表达来说依然是必要的。

标识出并非是项目动机的的相关子域。找出那些子域的普通模型，并将它们放在分离的模块中。不要在这些模块中留下什么专业知识。

一旦它们被分离开，就将对它们做持续开发的优先级调得比核心域更低，而且注意不要把你的核心开发人员分配到这些任务中（因为它们从中获取不了多少领域知识）。另外要考虑现成的解决方案，或者那些已发布的针对普通子域的模型。

每个领域会用到其他领域所使用的概念。钱和与其相关的概念，例如货币和汇率，可以被包扩在不同的系统里。图表是另外一个被广泛使用的概念，就它本身而言是非常复杂的，但是可以被用在很多应用中。

有下面几种方法可以实现普通子域：

1. **购买现成的解决方案**。这个方法的好处是可以使用别人已经完成的全套解决方案。随之而来的是学习曲线的问题，而且这样的方案还会引入一些依赖。如果代码有很多 bug，你只得等待别人来解决。你还需要使用特定的编译器和类库版本。与自己实现的系统相比，将这样的方案与自己系统做集成也不是那么容易。
2. **外包**。将设计和实现交给另外一个团队，有可能是其他公司的团队。这样做可以使你专注于核心域，不再承受处理另一个领域的负担。不便的地方是集成外包的代码。用来与子域通信的接口需要预先定义好，并且与其他团队进行沟通。
3. **已有模型**。一个取巧的方案是使用一个已经创建的模型。市面上已经有一些关于分析模式的书，可以用来作为我们子域的灵感来源。直接复制原有的模式不太现实，但确实有些模式只需要做少许改动就可以用了。
4. **自己实现**。这个方案的好处是能够做到最好的集成，但这也意味着额外的付出，包括维护的压力等。

第 6 章

DDD 在今天仍然很重要：专访 Eric Evans

InfoQ.com 采访了领域驱动设计的创始人 Eric Evans，以了解这一设计技术的最新进展。

⇨为什么 DDD 一直都很重要？

基本上，DDD 是我们应该专注于用户所从事领域里的深层问题的指导原则。我们最好的精力应该致力于理解那个领域，与那个领域的专家们一起合作，将它抽象成一个我们可以用来建造强大而灵活的软件的概念化形式。

这是一个不会过时的指导原则。无论何时我们**操作**一个复杂的、难以理解的领域，这个原则都是有用的。

长期的趋势是软件会被应用于解决越来越复杂的问题，越来越深入这些业务的核心。在我看来，这一趋势中断了几年，因为 Web 突然出现在我们面前。人们的注意力被从富于逻辑和深层次的解决方案上转移开，因为仅仅将数据传递到 Web 上，同时实现简单的行为，就能产生出巨大的价值。因为太多的数据要传递，但一段时间内在 Web 上做简单的事情都比较困难，所以消耗了软件开发的所有努力。

但是现在人们已经跨越了这一 Web 应用的基本层次，软件项目的雄心再次更多地集中在业务逻辑上。

最近，Web 开发平台逐渐成熟，Web 开发的生产力已经提高到足以应用领域驱动设计，有很多积极的趋势。例如，SOA，如果应用的好，就可以提供给我们一个非常有用的**隔离**领域的方法。

同时，敏捷开发过程也有了足够的影响力，大多数项目现在多少都意识到了迭代、与业务伙伴密切协作、应用持续集成和在强沟通环境（high-communication environment）下工作的重要性。

所以 DDD 在可预见的未来会越来越重要，目前已经有了些基础。

⇨技术平台，像 Java、.NET、Ruby 或者其他的等都一直在变化。领域驱动设计如何适应这一情况？

实际上，应该由它们是否支持团队专注于他们的领域来验证新技术和开发过程的价值，而不是使团队分心以至于无暇顾及领域。DDD 并不特定于哪一个技术平台，但是有些平台为创建业务逻辑提供了表达力更强的方式，在有些平台中使人分心的杂乱更少一些。对于后面这些平台，过去几年的发展显示出大有希望的方向，尤其在可怕的 20 世纪 90 年代后期之后。

Java 是过去这几年默认的选择，从表达能力的角度看，它是一种典型的面向对象语言。对于使人分心的杂乱而言，Java 的基础语言还不算太坏。它有垃圾收集功能，实践证明这一功能很有必要（这一点是相比于需要非常关注底层细节的 C++而言的）。Java 语法有些地方比较乱，但是仍然可以通过使用 POJO 来获得可读性。Java 5 语法的一些革新提高了代码的可读性。

但是在 J2EE 框架最初出现时，完全将那些基本的表达能力淹没在大量的框架代码之中。根据早期的约定（例如 EJB Home、为所有变量写的 get/set 方法等）生产出可怕的对象。这些工具是如此笨重，使得开发团队不得不全力以赴才能让它工作。改变对象是如此困难，以至于一旦生成了大量代码和 XML，人们往往不再修改它们。这样的平台几乎不可能进行高效的领域建模。

与此同时还要使用相当原始的第一代工具来勉强开发以 http 和 html 作为媒介的 Web UI（它们不是为此目的而设计的）。在那个时期，创建和维护一个像样的 UI 是如此困难，以至于很少有精力再去关注复杂内部功能的设计。具有讽刺意味的是，恰恰在这时，对象技术横空出世，复杂的建模和设计方法发生了极大变化。

这一情况在.NET 平台上也是类似的，有些问题处理的会更好一些，而其他问题处理的则更为差劲。

那是一个令人沮丧的时期，但是在过去的四年里，趋势发生了转变。首先来看 Java，社区里（关于如何有选择地使用框架的）新出现的更为老练的做法，与增量式逐渐改进的很多新的框架（大多数是开源的）交汇在一起，相得益彰。例如 Hibernate 和 Spring 这些框架可以用一种轻量得多的方式来处理 J2EE 试图解决的特定工作。像 AJAX 这样试图解决 UI 问题的方法也更加便捷。现在的项目在选择使用可以提供价值的 J2EE 元素并将其与新元素结合时也更加聪明。术语 POJO 就是在这个时期产生的。

结果是项目的技术成本有了明显的降低，在把业务逻辑和系统的其他部分隔离方面也有了明显的进步，这样业务逻辑就可以基于 POJO 来编写了。这不会自动产生出领域驱动设计，但是它提供了一个可行的机会。

这就是 Java 世界。然后就是像 Ruby 这样的新来者。Ruby 有表达力非常强大的语法，从这个基础层面而言，它会是适合于 DDD 的一种非常棒的语言（尽管我尚未听说过在应用了 DDD 的那些应用中有哪些实际案例）。Rails 给人们带来了很多兴奋的地方，因为它最终好像能够使得开发 Web UI 像上世纪 90 年代初期在 Web 出现之前开发 UI 时一样容易。在目前，这种能力大多数时候被用于建造一些没有很多复杂领域逻辑的 Web 应用上，因为就是开发这些简单的应用在过去来说也是很困难的。但是我的希望是，当问题里的 UI 实现部分减少时，人们可以把这看成将他们的注意力更加专注于领域的好机会。如果使用 Ruby 是向着这个方向努力的，那么我认为它会为 DDD 提供一个优秀的平台（可能还需要补充一些基础设施）。

更多前沿的话题发生在领域特定语言（DSL）这个领域，我一直深信 DSL 会是 DDD 发展的下一大步。现在，我们还没有一个工具可以真正给我们想要的东西。但是人们在这一领域比过去做了更多的实验，这使我对未来充满了希望。

在目前，我所能说的是大多数尝试应用领域驱动设计的人们是使用 Java 或者.NET，也有少数人在使用 Smalltalk。在 Java 世界里已经取得了直接的效果，这是一个积极的趋势。

⇨从你写完这本书，在 DDD 社区里发生了那些值得注意的事情？

一个让我很兴奋的事情是，人们会用一些我所没有预料到的方式来应用我在书中提到的原则。例如，在挪威国家石油公司的 StatOil 项目中对战略（strategic）设计的使用。那儿的一个架构师根据他的经验写了一篇报道（参见 http://domaindrivendesign.org/articles/.）。

在其他项目里，还有人做了上下文映射，并应用来评估是自己构建还是购买已有软件。

另外还有一个相当不同的例子，我们中的一些人通过为很多项目都需要的一些基础的领域对象开发一个代码库，探索了一些问题。人们可以从这里检出这个代码库：

http://timeandmoney.domainlanguage.com

例如，我们探索了仍然使用 Java 来实现对象时，究竟能把一种流畅的、领域特定的语言推动多远。

我们已经取得了很多进展。在人们联系到我，告诉我他们正在做什么事情的时候，我一直都很感激。

⇨请您给要学习 DDD 的人一些建议？

请读一下我的书！　　另外，在项目中舍得投入时间和资金。我们最初的目标之一就是提供一个好的案例，让人们能够通过使用它进行学习。

要谨记一点的是 DDD 主要是由团队来完成的事情，所以你也许要成为一个布道者。现实一点讲，可能需要你找到一个大家正在努力完成的项目。

另外还要注意下面几个领域建模时的陷阱：

1. 亲自动手，建模者需要写代码。
2. 专注于具体场景。抽象思维需要落地于具体案例。
3. 不要试图对任何事情都应用 DDD。画一张上下文映射，然后决定在哪里需要努力应用 DDD，在哪里不需要。不要担心边界之外的事情。
4. 进行大量的尝试，期望能产生大量错误。模型是一个创造性的过程。

关于 Eric Evans

Eirc Evans 是《领域驱动设计——软件核心复杂性应对之道》（Addison-Wesley 2004，已由清华大学出版社翻译出版）一书的作者。

早 20 世纪 90 年代，他就参与了很多项目，通过使用多种不同方法、多种不同效果的对象，来开发大型的业务系统。这本书是那些经验的总结。它提供了一个建模和设计技术的系统，成功的团队已经应用这一系统来组装有业务需求的复杂软件系统，并使系统在逐渐增大时仍然能够保持敏捷。

Eric 现在是“Domain Language”的负责人。Domain Language 是一个咨询小组，它指导和训练团队实施领域驱动设计，帮助他们使自己的开发工作对于业务而言更有生产力和更有价值。

广　告

我们的服务

我们帮助渴望进步的软件项目实现领域驱动设计和敏捷过程的潜能。

为了使领域建模和设计真正服务于项目，它需要高级和详细的设计同时出现。这是为什么我们要提供能真正获得领域驱动设计过程的整合服务的原因。

我们的培训课程和经验丰富的老师会加强团队在建模和部署有效应用时的基本技巧。我们的教练专注于团队的努力，并帮助解决在设计系统以符合业务需要时遇到的流程问题。我们的战略设计咨询师会负责解决那些影响项目整体进展的问题，使得所有的开发工作变得容易，并使项目向组织要达到的目标迈近。

评估

我们的评估会提供给你 Perspective 和具体的推荐。我们需要知道你的所在地，想去哪儿，并且开始设计蓝图帮助你到达目标。

预定评估

要询问价格、日程安排和其他信息，请致电 415-401-7020 或者发邮件到 info@domainlanguage.com。

www.domainlanguage.com

InfoQ 中文站使命：

成为关注企业软件开发领域变化和创新的专业网站

受众：面向决策人群，如团队领导者、技术架构师、项目经理和企业架构师。

社区/主题：Java、.NET、SOA、Agile、Ruby 和 Architecture

与众不同之处：

- 关注高级决策人员和大中型企业：InfoQ 是目前业界唯一致力于关注技术架构师和相关角色的网站。
- 个性化：读者可以根据自己的喜好定制 InfoQ 提供的内容，例如 Java 人群可以屏蔽.NET 内容，项目经理只浏览 Agile 相关内容等。
- 广告与内容相关：广告发布前，我们的编辑会为广告编制内容相关标签。
- 编辑均为领域专家：超过 30 位领域架构师/开发者定期为 InfoQ 撰写新闻和其他内容，没有哪家网站有如此强悍的编辑队伍。
- QCon 大会：2007 年 3 月份在伦敦举行了 QCon 大会，超过 500 人参加。主题涉及 InfoQ 网站覆盖的 6 大领域，包括金融 IT。11 月份在美国旧金山举行。
- 国际化：2007 年 3 月份 InfoQ 中文站上线，9 月份 InfoQ 日文站发布。
- 独一无二的内容：InfoQ 提供免费的电子书下载，附幻灯片的流媒体大会演讲，视频采访和深度文章。

口碑营销下的成长速度：

- 至 2007 年 7 月份每月有 17 万 2000 名独立访问用户，而 InfoQ 是在 2006 年 6 月份才发布。
- 技术指数位列互联网领域最权威网站的前 2000 名。
- Reddit.com 网站上 Programming.reddit.com 领域热点新闻的最主要来源。

销售联系：

请致电 010-84725788 或者 13811662678 或者邮件至 china-sales@infoq.com。

注：本说明中所提 InfoQ 包括 InfoQ 中文站。

领域驱动设计精简版

这本书没有介绍任何新的概念，它只是概要总结了领域驱动设计的本质，抽取了Eric Evans原书中关于这一主题的大部分内容，以及其他相关资料，包括已经出版的书籍和各种领域驱动设计讨论群组等。这本书可以让你快速了解领域驱动设计的基础知识，但不能替代Eric书(*Domain Driven Design*)中提供的大量事例和案例研究或者Jimmy书(*Applying Domain-Driven Design*)中提供的动手事例等。

我们非常鼓励大家去阅读这两本绝对优秀的书籍。同时，如果你也认同领域驱动设计这一概念需要成社区关注的重点，那么请让更多的人知道本书和Eric的工作！

本书主题：

- 何为“领域驱动设计”
- 构建领域知识
- 对通用语言的需要
- 创建通用语言
- 模型驱动设计
- 面向深层理解的重构
- 保持模型一致性
- 领域驱动设计新进展

无论你是企业架构师、团队领导者、项目经理还是高级软件开发人员，本书都可以帮助你正确理解领域驱动设计，并同时实现你的业务目标和技术目标。

译者简介

孙向晖，儿子小名“豆豆”，常被人称为“豆豆他爹”。1998年开始步入IT行业，现任浪潮软件质保中心副主任。专注于研究和实践MDA/UP/UML/SCM等相关技术在团队中的大规模应用，对产品化的软件项目管理、需求管理和配置管理略有心得。他的博客为http://blog.csdn.net/xiaosun/

霍泰稳，InfoQ中文站总编，有多年的软件开发经验和媒体从业经历，在《程序员》杂志社工作期间参与《程序员》、《MSDN 开发精选》、《BEA dev2dev专刊》和《开源大本营》等刊物的编辑策划工作，并主持负责了“软件中国2006年度风云榜”评选活动。此外，他还是BEA北京User Group的负责人。

InfoQ中文站 时刻关注企业软件开发领域的变化和创新