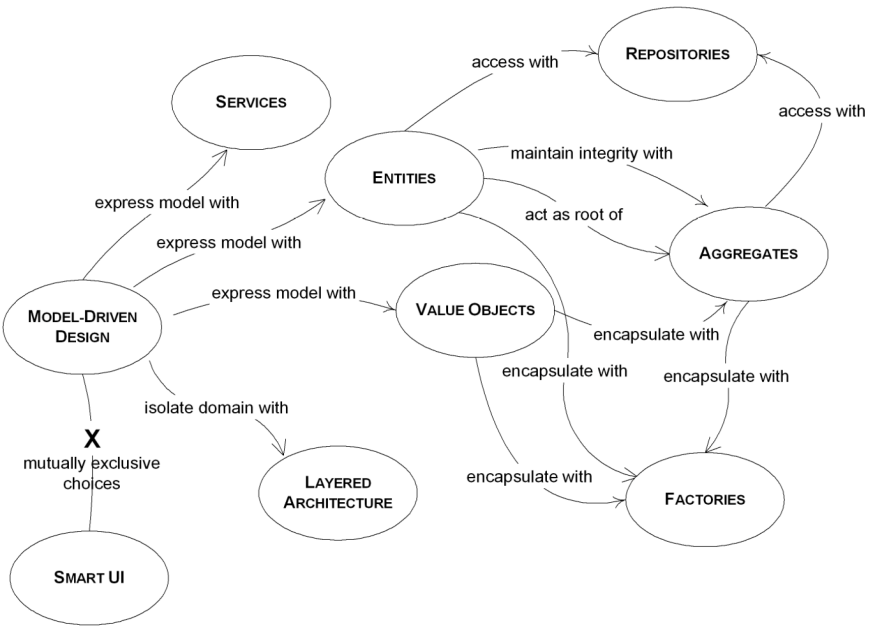
Eric Evans《领域驱动设计》总结之作

领域驱动设计

Domain Driven Design

精简版

Quickly



Abel Avran & Floyd Marinescu 著

孙向晖 霍泰稳 译

InfoQ 企业软件开发系列图书

免费在线版本

（非印刷免费在线版）

InfoQ 中文站出品

本书主页为

http://infoq.com/cn/minibooks/domain-driven-design-quickly

领域驱动设计

精简版

C4Media 是 InfoQ.com 这一企业软件开发社区的出版商

本书属于 InfoQ 企业软件开发系列图书

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

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

目录

前言：编者按

我第一次听说领域驱动设计和认识 Eric Evans 是在 2005 年夏天由 Bruce Eckel 组织的一个小型架构师顶级聚会上。参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主流一些。当我察觉到 Eric 使用领域模型来解释从前小组讨论的一些技术挑战的解决方案，以及他对业务领域而不是某个技术青睐有加的时候，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，尤其是 Web 开发社区，已经被多年的宣传从正确的面向对象软件开发方向上偏离了许久。在 Java 社区，优秀的领域建模也在 1999 年到 2004 年间的 EJB 和容器/组件建模宣传下迷失了方向。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。但是，社区也面临着缺少如何实现企业级规模面向对象清晰愿景的困境，这也是为什么我认为领域驱动设计重要的原因。

不幸的是，除了这些顶尖的小部分架构师，我看到很少有人理解面向领域设计，这也是为什么 InfoQ 执意要做这本小书的原因。

我希望，通过发布这样一个针对领域驱动设计基础的精短且易于阅读的概要和介绍，加上读者可以在 InfoQ 网站上免费下载电子版，或者购买便宜的印刷口袋书，这个愿景能够为更多的人所接受。

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 书中提供的大量事例和案例研究或者 Jimmy 书中提供的动手事例等。我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！

InfoQ.com 共同创始人和总编 **Floyd Marinescu**

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。比如，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，也就是假定能帮助我们管理的领域，分割开来。相反，它们紧密相连！

软件设计是一门艺术，像其他艺术一样，它不能通过定理和公式以一门精确科学的方式被教授和学习。通过软件创建的过程，我们可以发现有用的规律和技巧，但是我们也许永远不能提供一个准确的方法，以满足从现实世界映射到代码模型的需要。如同一幅画或者一个建筑，软件产品既包括设计和开发它的那些人的个人劳动，也包括致力于它发端和成长的那些人的某些领导力和洞察力（或者没有这个）。

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张方法创建它的产品，每一个都有自己的优点和不足之处。本书的目的是专注介绍一个出现和发展了 20 多年但近几年才有所成果的设计方法：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，我们推荐阅读他的由 Addison-Wesley 出版的 ISBN 号为 0-321-12521-5 的“Domain-Driven Design: Tackling Complexity in the Heart of Software”一书（中文名字《领域驱动设计--软件核心复杂性应对之道》，清华大学出版社出版）。

通过参与领域驱动设计的讨论组也可以学习到许多有价值的观点：

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。我们只想通过领域驱动设计世界中使用的原理和指导，激发出你对优秀软件设计的探索欲望！

1 何为“领域驱动设计”

软件开发通常被应用到真实世界中已经存在的自动化流程，或者给真实的业务问题提供解决方案，即要自动化的业务流程或者可以用软件解决的现实问题。从一开始，我们就必需明白软件脱胎于领域，并跟领域密切相关。

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

假设以汽车制造来类比。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，（历经岁月），用上几个月甚至几年的时间去设计、修改、精化直至完美，直至它反映出最初的愿景。设计的过程也不全然是在纸上进行的。许多的设计工作包括制模、在极端条件下对它们进行测试，以验证它们是否能工作等。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。当然也可以这样做，在开发价值不大的软件时。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来吗？没门。答案永远是否定的。那谁了解银行业业务？软件架构师吗？不，他只是在使用银行来保护他的财产安全，以保证他的急时所需；软件分析师吗？也不是，他只精通于如何运用所有能够获得的必要因素去分析一个给定的主题；软件开发人员？别难为他了。那还有谁？当然是银行的业务人员了。银行业务系统被银行的内部人员所熟知，我们称其为专家。他们知道所有的细节，所有的困难、所有可能出现的问题，以及所有的规章等。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，它会给领域引入麻烦，产生障碍、灾难甚至导致混乱等。

我们怎样才能让软件和领域和谐相处呢？最佳的方式是让软件成为领域的反射（映射）。软件需要具现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。软件需要对领域进行建模。

如果某人不了解银行业业务，他可能需要阅读很多针对领域模型的代码。这是非常重要的。随着时间的推移，没有植根于领域的软件不能很好地应对变化。

所以我们从领域开始着手。接下来要做什么呢？领域是世界中的某些事物，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。我们需要建立领域的抽象。当我们跟领域专家交流时，我们会学到好多领域知识，但这些未加工的知识不能被容易地转换成软件构造，除非我们为它建立一个抽象——在脑海中建立一个蓝图。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，它是那幅图要极力去传达的那个思想。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。在设计过程中我们记住并会对模型的很多方面进行引用。我们周遭的世界中有着海量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畴。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？这在软件建立过程中属于设计部分的范畴。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础的部分。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，但它必须要超越我们的头脑，如果它止步不前，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，跟资深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现它，让它和其他事物交流。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。常见的一种方式是将模型图形化：图、用例、画和图片等。另一种方式是写，我们会写下我们对领域的愿景。还有一种方式是使用语言，我们能够也应该针对要交流的领域内的特定问题建立一种语言。在以后的章节中我们会详细讲解它们，但主要观点是说，**我们需要用模型来交流**。

当我们有一个表现的模型时，就可以开始做代码设计了。这跟软件设计有很大的不同。软件设计类似于构建房子的架构，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油画。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。代码设计模型唾手可得，当需要时，它们会被很好地应用。优良的编码技巧可以帮助我们建立清晰、高可维护性的代码。

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模型并作为结果传递给开发人员，开发

人员根据他们收到的内容开始编码。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，这么多年来已经有了一定级别的成功应用，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。

另一个方法是敏捷方法学，例如极限编程（XP）。这些方法学是不同于瀑布方法的一堆动作，产生背景是预先很难确定所有的需求，特别是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很困难。需要做出很多的思考，而且开始时常不能看到涉及到的所有的问题，也不能预见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无所事事。尽管敏捷方法的倡导者承认设计决定的重要性，但他们反对预先设计。相反，他们使用大量灵活的实现，通过由业务涉众持续参与的迭代开发和许多重构，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”的意义有着自己的观点。同时，缺乏了真实可见的设计原则，由开发人员执行地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，应用这些原则会增进对领域内复杂问题进行建模和实现的开发过程能力。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

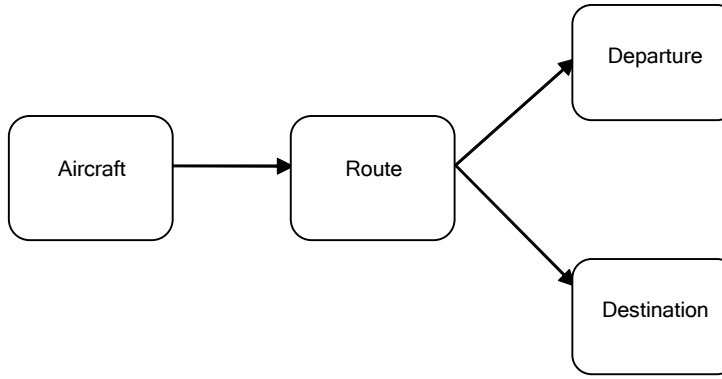
在空中发生碰撞。这儿我们不会试图演化成一个完整的交通控制系统，只会关注其中的一个小的子集：飞行监控系统。这个提到的项目是一个监控系统，它会跟踪在指定区域内的任意航班，判断班机是否遵照了预定的航线，以及是否有可能发生碰撞。

如果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。你不能期望他们会交给你一个关于他们问题域的完整描述。

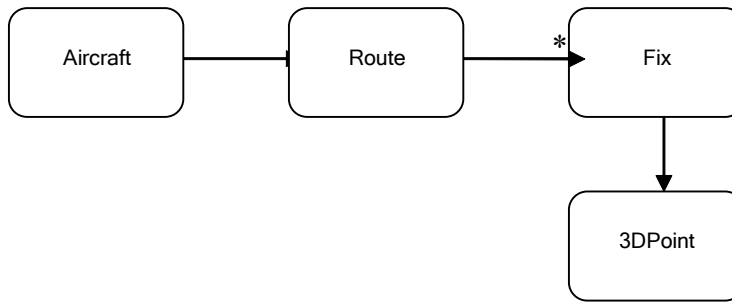
空中控制人员对他们的领域拥有广博的知识，但为了能构建模型你需要提取基础信息并归纳它。当你开始跟他们讨论时，你会听到很多关于飞机起飞、着陆、飞机升空和碰撞危险、飞机请求允许着陆等知识。为了找到看似杂乱无章的信息中的规律，我们需要从某个地方开始。

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

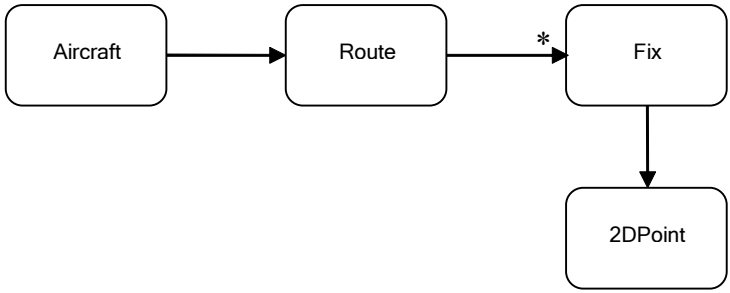
那么，飞机从某地起飞又在另一地降落。但空中发生了什么？班机会按照什么路线航行？事实上，我们更关心的是它在航行时所发生的事情。控制人员说会给每架飞机指派一个飞行计划，飞行计划会用来描述假定的整个空中旅行。当听到“飞行计划”时，你可能会在你脑海中想到这是一个飞机在空中必须遵守的路径。在后边的讨论中，你会听到一个有趣的词：路线(route)。它能很快引起你的注意，这是个很好的理由。路线包含了飞机航行中的一个重要概念。那就是飞机在飞行时要做的事，它们必须遵照一条路线。很明显飞机的出发点和目标点也就是路线的开始和结束点。所以，不同于将飞行器与出发点和目标点管理，看上去更自然的是将它与“路线”进行关联，然后路线再与适当的出发点和目的地关联。



跟控制人员交流飞行器需要遵照的路线时，你会发现路线由小的区间段组成，这些区间段按照一定的次序组织起来就会构成从出发点到目的地的一条曲线。这条线被假定穿过预定的方位点。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，你不会再将出发点和目的地看作是路线的结束点，而只是将它们看作那些方位中的两个点。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。根据这些发现产生出来的变化结果如下：



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是一个在空间中的一个点，它表现为一个 3 维的点。但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，然后他们回应。当他们这样做时，他们从空中交通领域挖掘出基础性的概念。那些概念可能看上去未经雕琢、没有经过组织，但它们却是理解领域的基础。你需要尽可能多地从专家处学习领域知识。通过提出正确的问题，正确地处理得到的信息，你和专家会开始勾勒领域的骨架视图，也就是领域模型。这种骨架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，可以尽力判断出领域的基础性概念。

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知识，他们也不得不将它们转换成有用的形式。从某种角度上讲，他们可能希望建立一种早期的原型以验证它是否能按照预期工作。在这当中，他们可能会发现一些自己模型或者方法中的某些问题，可能想改变模型。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，这会帮助我们更好地建立模型，获得更清晰更准确的对模型的理解。领域专家掌握很多的专业技能，只是他们按照特殊的方式组织和使用这些知识，而这通常对要将它们实现到软件系统中不是件好事情。

通过与领域专家的交谈，软件设计人员的分析型思维会帮助他挖掘出一些领域中的关键概念，并且帮助构建出可用于将来讨论用的结构，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，但是它也应该被这样做，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。

2 通用语言

对通用语言的需要

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的模型是绝对需要的，但是，那种方法通常会由于一些基础交流的障碍而存在难点。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。他们只了解他们特有的专业技能。

在空中交通监控样例中，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，知道飞机的发射。他们用他们自己的术语讨论这些事情，有时这对于外行来说很难直接理解。

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涉及到的元素的想法，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？

当团队成员不能享用一个公共语言来讨论领域时，项目会面临严重的问题。领域专家使用自己的行话，技术团队成员在设计中也用自己的语言讨论领域。

代码可能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，所以要想完成对领域的深刻表达通常需要产生一种临时形式，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析一些设计模式，但这并一定都能成功奏效。领域专家也可能会创建一种新的行话以努力表达他们的这些想法。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通用的语言，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核心骨架。要求团队在进行所有的交流是都使用一致的语言，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。这儿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，建立了设计团队良好工作的前提。可能会花费数周乃至数月的时间才能让一个大规模项目的设计成型。团队成员会发现一些初始的概念是不正确的或者不合适宜，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。没有了通用语言，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，它需要做出艰苦卓绝的工作并关注很多方面的内容，才能确保语言的核心元素被发现。我们需要发现定义领域和模型的那些关键性概念，发现描述它们的相应的用

词，并开始使用它们。它们当中的一些可能很容易被发现，但有些却不能。

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**出发点**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。

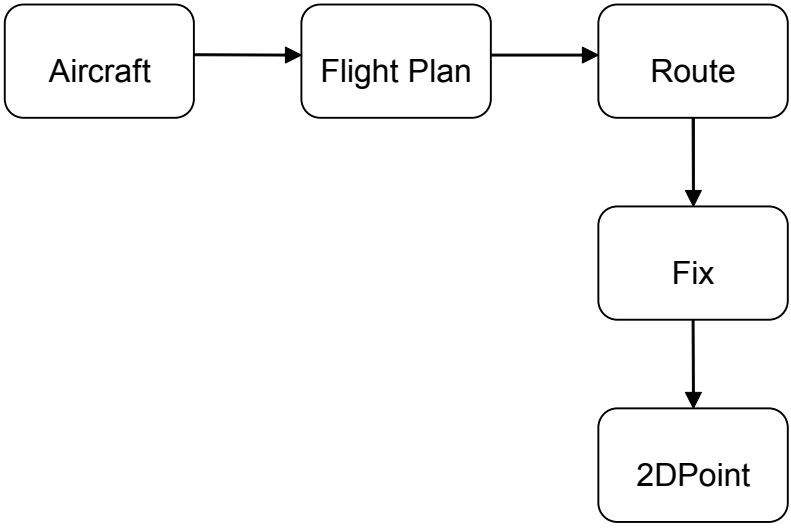
开发人员：哦，那我们可以称每一个这样的点为一个**方位**，因为它是地球表面上的一个固定的点。我们将使用一系列 2D 的点来描述线路。顺便说一句，**出发点**和**终点**都属于**方位**。我们不再会将它们考虑成其他不同的概念。**飞行路线**到达终点就如同它到达其他的**方位**一样。飞机必须遵照飞行路线，但这是否意味着它可以按照自己的意愿选择飞行高度？

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。

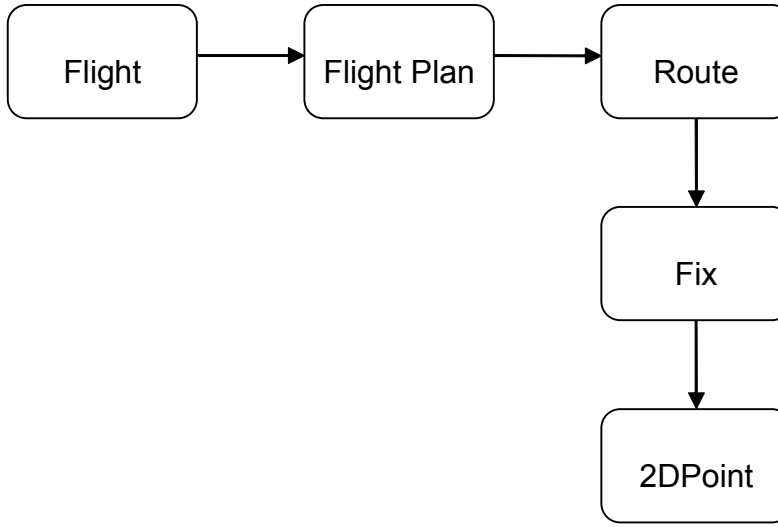
开发人员：噢，**飞行计划**看起来相当的重要。我们可得把它加到模型中。



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出改变以保证其正确性。

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。为了解决这点，所有的团队成员应该都意识到需要建立一种通用的语言，并应保持专注核心的部分，在任何需要之时使用这种语言。我们应该尽可能少地在这种场景中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。那依赖于其他的因素，我们会在以后讨论它们。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，它会让代码更可读，让模型得到完美实现。代码表现模型会让项目得益，如果代码没有得到适当地设计，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，只是语言交谈吗？我们已经用图了，还有其他用法吗？文档吗？

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，画出它们自己的关系。对每个人来说都可以很容易地掌握你的想法，一个某种想法的图形化展现方式很容易被人理解。每个人立即共享到关于特定主题的相同的愿景，基于此的沟通因此变得简单。当有新的想法出现时，会修改图以反映概念的变化。

UML 图在只涉及少量元素时很有帮助。但 UML 会像雨后的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的长纸上时，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，就会很难理解，一个中等规模项目的类图也会如此。

同时，UML 擅长表现类，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现。为此 UML 求助于文字，将它们作为注释放在图上。所以 UML 不能传达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准备做什么。但那依然很好，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。一个明智的沟通模型的方式是创建一些小的图，让每副小图包含模型的一个子集。这些图会包含若干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。每个这样的片断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聚焦领域的一个部分。

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体模型的大图可能会很有诱惑力，但是，大部分时间像这样的图很有可能不能组合在一起。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，并不能传递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。文档必须跟模型同步。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。

也可能使用代码来交流，这个方法被 XP 社区广泛倡导。优良的代码也具有很好的可交流性。尽管用方法表现行为是清楚的，但方法体也能跟方法名一样清楚吗？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得表达出正确的事情。将模型写入代码是非常困难的。

在设计中还有其他沟通的方式，本书不想全部描述它们。有一个件事情是非常清楚的：由软件架构师、开发人员和领域专家构成的开发团队，需要一种语言来统一它们的行动，以帮助它们创建一个模型，并使用代码来表现模型。

3

模型驱动设计

前面的章节强调过软件开发过程的重点：它必须以业务领域为中心。我们说过让模型植根于领域、并精确反映出领域中的基础概念是建立模型的一个最重要的基础。通用语言应该在建模过程中广泛尝试以推动软件专家和领域专家之间的沟通，以及发现要在模型中使用的主要的领域概念。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两个阶段。创建了优良的模型，但却未能将其成功地转换成代码将把软件的质量带入未知境地。

曾经发生过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，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不能被正确地转换成代码。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，虽然某些设计借鉴了模型的思想，另外他们还增加了很多自己的东西。开发过程继续进行，更多的类被加入到代码中，进一步加大了原始模型和最终实现的差距。在这种情况下，很难保证产生优良的最终结果。优秀的开发人员可能会让一个产品最终交付使用，但它能经得起生产环境的考验吗？它能容易地被扩展吗？它能容易地被维护吗？

任何领域都能被表现成多种模型，每一种模型都能用不同的方式表现成代码。对每一个特殊问题而言，可能会对应不止一个解决方案。我们应该选择哪一个呢？拥有一个看上去正确的模型不代表模型能被直接转换成代码。也或者它的实现会违背某些我们所不建议的软件设计原则呢。选择一个能够被轻易和准确转换成代码的模型很重要。最根本的问题是：我们应该如何动手处理从模型到代码的转换。

一个推荐的设计技巧是使用分析模型，它被认为是从代码设计中分离出来、通常是由多个人完成的。分析模型是业务领域分析的结果，其产生的模型不考虑软件需要如何实现。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，模型看上去会很正确。软件不是这个阶段要考虑的，因为它的介入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涉及到设计原则，它可能不能很好地完成目标。因此开发人员不得不修改它，或者建立分离的设计。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放弃了。

这个方法中存在的一个主要的问题是分析不能预见模型中存在的某些缺陷以及领域中的所有的复杂关系。分析人员可能深入到了模型中某些组件的细节，但却未深入到其他的部分。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的一系列问题，或者导致不可接受的性能行为。开发人员会被强制做出他们自己的决定，会做出设计变更以解决实际问题，而这个问题在模型建立时是没有考虑到的。他们建立了一个偏离了模型的设计，让它们二者越来越不相关。

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，分析人员的一些领域知识和模型就丢失了。虽然模型可以被表现成图或者文字形式，极有可能的还是开发人员不能掌握整个模型、或者某些对象的关系或者他们之间的行为。模型中的很多细节不适合表现成图的方式，甚至也可能不能完全用文字来表现。开发人员需要花费大量精力去处理它们。在某种情况下他们会对想要的行为做出某种假设，这极有可能让他们做出错误的决定，最终导致错误的程序功能。

分析人员会参加频繁的领域讨论会议，他们会享有很多知识。他们建立假定包含了所有信息的浓缩模型，开发人员不得不阅读所有他们给的文档以吸收精华。如果开发人员能够加入分析讨论会议，并在开始设计编码前获得清晰完整的领域和模型视图会更有效率。

一个更好的方法是紧密关联领域建模和设计。模型在构建时就考虑到软件和设计。开发人员会被加入到建模的过程中来。主要的想法是选择一个能够恰当在软件中表现的模型，这样设计过程会很顺畅并且基于模型。代码和其下的模型紧密关联会让代码更有意义并与模型更相关。

有了开发人员的参与就会有反馈。它能保证模型被实现成软件。如果其中某处有错误，会在早期就被标识出来，问题也会容易修正。

写代码的人会很好地了解模型，会感觉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，另外，如果哪里的代码不能表现原始模型的话，他们会重构代码。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。最终结果是模型不再实用。

任何技术人员想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表现模型。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使用通用语言，动态交换模型思想。

如果设计或者设计中的核心部分不能映射到领域模型，模型基本上就没有什么价值，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复杂映射很难理解，实际上可能很难维护设计变更。分析和设计之间铁定被割裂开来，这样一个（分析或设计）活动中获取到的想法将不能对另外一个产生影响。

设计软件系统的一部分，保证它能如实反映领域模型，让映射显而易见。重新访问模型，修改它，让它能在软件中更自然地得到实现，甚至可让它能按你所愿反映对领域的更深层理解。除了支持流畅的通用语言，还可以要求一个单独的模型来服务好这两个目的。

从模型中去除在设计中使用的术语和所赋予的基本职责后，代码就成了模型的表达式，所以对代码的一个变更就可能称为对模型的变更。这个影响会波及到项目的其余活动中。

为了紧密捆绑起实现和模型，通常需要支持建模范型的软件开发工具和语言，例如面向对象编程。

面向对象编程非常适合对模型的实现，因为它们基于同一个范型。面向对象编程提供了对象的类和类之间的关联关系、对象实例、以及对象实例之间的消息通信。面向对象编程语言让建立模型对象、对象关系与它们的编程副本之间的直接映射成为可能。

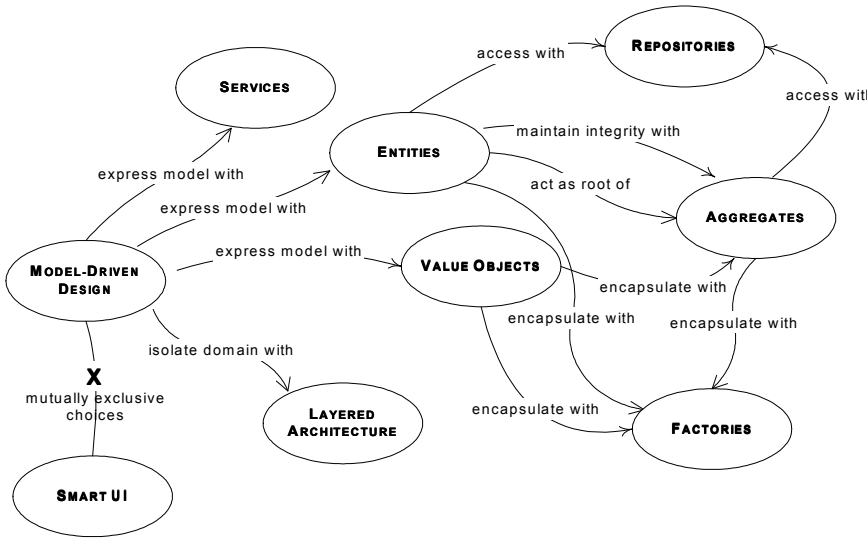
过程化语言提供了有限的模型驱动设计的支持。这样的语言不能提供实现模型关键组件所必须的构建能力。有人说象 C 这样的过程化语言能够实现面向对象编程，确实，某些功能可能被用某种方式再

现：对象可能被模拟成数据结构。这样的数据结构不能包含对象的行为，所以必须额外加入函数。这种数据的意义仅存在开发人员的脑海中，因为代码本身都不那么明显。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，为达到一个特定的结果共同协作。这样的程序不能轻易地封装概念性连接，很难实现领域和代码之间的映射。

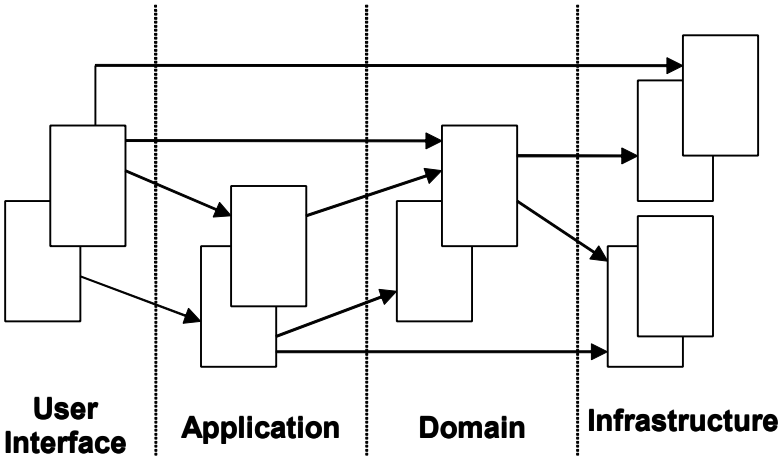
某些特殊的领域（例如数学）可以借助过程化编程被轻易地建模和实现，是因为许多数学理论大多数都是关于计算的，可以用函数调用和数据结构简单解决。许多复杂的领域不仅是一组抽象概念涉及到的计算，所以不能简化成一系列的算法，因此过程化语言不足以完成表述各自模型的任务。因为这个原因，模型驱动设计中不推荐过程化编程。

模型驱动设计的基本构成要素

本章接下来的章节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基本元素。下图是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是不能直接跟领域关联的，但它们是基础设施的一部分或者是为软件服务的。最好能让应用中的领域部分尽可能少地和其他的部分掺杂在一起，因为一个典型的应用包含了很多和数据库访问，文件或网络访问以及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。之所以这样做的某些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，要阅读和思考它也变得极其困难。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层代码、数据库代码以及其他程序元素。实现粘连在了一起，模型驱动对象于是变得不再可行。也很难使用自动化测试。对于每个活动中涉及到的技术和逻辑，程序必须保持简单，否则就会变得很难理解。

因此，将一个复杂的程序切分成层。开发每一个层中内聚的设计，让每个层仅依赖于它底下的那层。遵照标准的架构模式以提供层的低耦合。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分隔开来。释放领域对象的显示自己、

保存自己、管理应用任务等职责，让它专注于展现领域模型。这会让一个模型进一步富含知识，更清晰地捕获基础的业务知识，让它们正常工作。

一个通用领域驱动设计的架构性解决方案包含 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。它不保留业务对象的状态，但它保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撑库等作用。

将应用划分成分离的层并建立层间的交换规则很重要。如果代码没有被清晰隔离到某层中，它会迅即混乱，因为它变得非常难以管理变更。在某处对代码的一个简单修改会对其他地方的代码造成不可估量的结果。领域层应该关注核心的领域问题。它应该不涉及基础设施类的活动。用户界面既不跟业务逻辑紧密捆绑也不包含通常属于基础设施层的任务。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整个活动。

例如，对一个典型的交互型应用，领域和基础设施层看上去会这样：用户希望预定一个飞行路线，要求用一个应用层中的应用服务来完成。应用依次从基础设施中取得相关的领域对象，调用它们的相关方法，比如检查与另一个已经被预定的飞行线路的安全边界。当领域对象执行完所有的检查并修改了它们的状态决定后，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后仍能保持一致。对这些对象来讲这已经不再是它们关心的属

性，这意味着能够跨越系统的生命周期甚至能超越软件系统的一系列的延续性和标识符。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，它们对每个对象会保持一个对象引用或者是记录一个对象地址。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，它被序列化后在网络上传输，然后在另一端被重新建立，或者它们都被消除。在程序的运行环境中，那个看起来像标识符的引用关系其实并不是我们在谈论的标识符。

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，可以用其中一个跟另一个交换，但它们拥有不同的引用，它们不是实体。

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，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数字码。每一个账户可以用它的数字码来精确标识。这个数字码在系统的生命周期中会保持不变，并保证延续性。账户码可以作为一个对象存在于内存中，也可以被在内存中销毁，发送到数据库中。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，数字码会保持一致。

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，父母名称、当前地址。在美国，社会保险号也会用来创建标识符。对一个银行账户来说，账号看上去已经足可以作为标识符了。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现标识符而创建的属性，也或是一种行为。对两个拥有不同标识符的对象来说，能用系统轻易地把它们区分开来，或者两个使用了相同标识符

的对象能被系统看成是相同的，这些都是非常重要的。如果不能满足这个条件，整个系统可能是有问题的。

有很多不同的方式来为每一个对象创建一个唯一的标识符：可能由一个模型来自动产生 ID，在软件中内部使用，不会让它对用户可见；它可能是数据库表的一个主键，会被保证在数据库中是唯一的。只要对象从数据库中被检索，它的 ID 就会被检索出并在内存中被重建；ID 也可能由用户创建，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，被世界上的每一个旅行代理使用以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，另一个属性就会被加入以帮助确定每一个对象。

当一个对象可以用其标识符而不是它的属性来区分时，可以将它作为模型中的主要定义。保证类定义简洁并关注生命周期的延续性和可标识性。对每个对象定义一个有意义的区分，而不管它的形式或者历史。警惕要求使用属性匹配对象的需求。定义一个可以保证对每一个对象产生一个唯一的结果的操作，这个过程可能需要某个符号以保证唯一性。这意味着标识可以来自外部，或者它可以是由系统产生、使用任意的标识符，但它必须符合模型中的身份差别。模型必须定义哪些被看作同一事物。

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，这会在下一个模型中被讨论。

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？

我们可能被引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。我们需要保证每一个实体都有唯一标识，跟踪标识也并非易事。我们需要确保每个实例拥有它唯一的标识，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，而这并不是我们所预期的。将所有的对象视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标识一个特殊的银行客户，不能被对应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建立一个这样的实例。这会导致系统在处理成千上万的实例时性能严重下降。

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 的对象类非常有用，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屏幕或者画布的坐标。考虑每一个点都拥有标识符是必要的吗？它会有延续性吗？看上去与这样一个对象相关的事情只有它的坐标。

这是我们需要包含一个领域对象的某些属性时的例子。我们对某个对象是什么不感兴趣，只关心它拥有的属性。用来描述领域的特殊方面、且没有标识符的一个对象，叫做值对象。

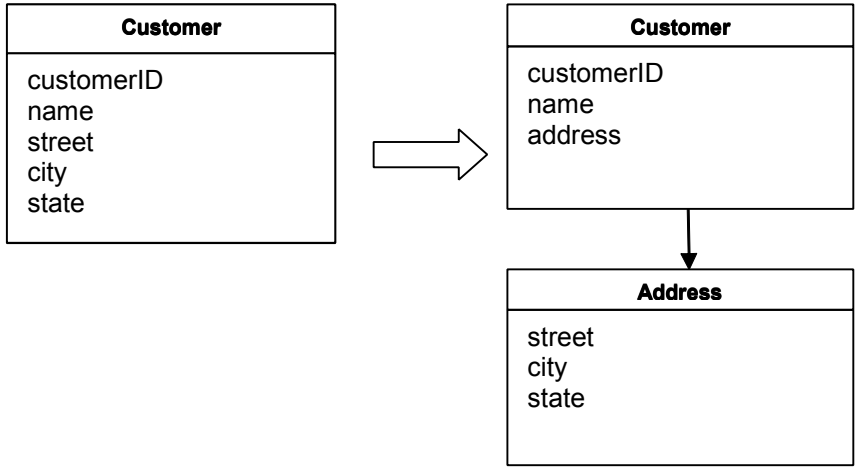
区分实体对象和值对象非常必要。出于要统一的缘故而将所有对象处理成实体对象并没有太大帮助。实际上，只建议选择那些符合实体定义的对象作为实体，将剩下的对象处理成值对象（我们会在下一个章节引入其他类型的对象，但我们假设现在只有实体对象和值对象两种）。这会简化设计，并且将会产生某些其他的积极的意义。

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。

极力推荐值对象是不变的。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。当你希望一个对象的不同的值时，你会简单地去创建另一个对象。这会对设计产生重要的结果。保持不变，并且不具有标识符，值对象就可以被共享了。这对某些设计是必要的。不变的对象可在重要的性能语境中共享。它们也能表明一致性，如：数据一致性。设想一下共享可变的对象会意味着什么。一个航空旅行预定系统可能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的目的地预定一个航班。另一个客户希望订购同一个航班。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，所以系统改变了航班号。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。值对象应该保持尽量的简单。当其他当事人需要一个值对象时，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。虽然值对象可用来简化一个领域对象要包含的属性，但这并不意味着它应该包含所有的一大长列的属性。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。一个客户会跟其名称、街道、城市、州县相关。最好分离出一个对象来包含地址信息，客户对象会包含一个对地址对象的引用。街道、城市、州县应该归属于一个对象，因为它们在概念上属于一体的，而不应该是作为分离的客户属性。

服务

当我们分析领域并试图定义构成模型的主要对象时，我们发现有些方面的领域很难映射成对象。对象要通常考虑的是拥有属性，对象会管理它的内部状态并暴露行为。在我们开发通用语言时，领域中的主要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变成那些对象的行为。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的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该属于它的功能。但是，要使用一种面向对象语言，我们必须用到一个对象才行。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了，它的作用是为了简化所提供的领域功能。服务所能提供的协调作用是非常重要的，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。最好显式声明服务，因为它创建了领域中的一个清晰的特性，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，因为那些对象的立场将变得不清楚。

服务担当了一个提供操作的接口。服务在技术框架中是通用的，但它们也能被运用到领域层中。一个服务不是在执行服务的对象，而与被执行操作的对象相关。在这种情况下，一个服务通常变成了多个对象的一个链接点。这也是为什么行为很自然地依附于一个服务而不是被包含到其他领域对象的一个原因。如果这样的功能被包含进领域对象，就会在领域对象和成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。

一个服务应该不是对通常属于领域对象的操作的替代。我们不应该为每一个需要的操作来建立一个服务。但是当一个操作凸现为一个

领域中的重要概念时，就需要为它建立一个服务了。以下是服务的 3 个特征：

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

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变化不属于一个实体或者值对象的自然职责时，向模型中增加一个操作，作为一个单独的接口将其声明一个服务。根据领域模型的语言定义一个接口，确保操作的名字是通用语言的一部分。让服务变得无状态。

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。服务层也可能有服务，这会继续增加层级的复杂性。这些服务甚至更难从与领域层中的近似的服务中分离开来。当我们在设计阶段建立模型时，我们需要确保领域层保持从其他层中隔离开来。

不论是应用服务还是领域服务，通常都是建立在领域实体和值对象的上层，以便直接为这些相关的对象提供所需的服务。决定一个服务所应归属的层是非常困难的事情。如果所执行的操作概念上属于领域层，那么服务就应该放到这个层。如果操作和领域对象相关，而且确实也跟领域有关，能够满足领域的需要，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，它们会基于模版产生。最终的结果是一个在 Web 浏览器中可以显式给用户查看的 HTML 页面。

用户界面层被合并成 Web 页面，允许用户登录，选择所期望的报表，单击一个按钮就可以发出请求。应用层是非常薄的一个层，它位于用户界面和领域层以及基础设施层的中间位置。它在登录操作时，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，对象直接关联到报表。有两个这样的对象是报表产生的基础，它们是 Report 和 Template。基础设施层将支持数据库访问和文件访问。

当一个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的名称。这会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项目以及报表中要包括的数据的时间间隔等。但出于简化的考虑我们将只提到 reportID。这个名字会通过应用层传递到领域层。领域层有义务根据所给的名字来创建并返回报表。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，这个模版被存放在一个文件或者数据库中。这不适于作为 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分离服务出来，这个服务的目的是基于一个 report 的标识符来检索一个报表模版。这会是一个位于领域层的服务。它也会用到文件类的基础设施，以从磁盘上检索模版。

模块

对一个大型的复杂项目而言，模型趋向于越来越大。模型到达了一个作为整体很难讨论的点，理解不同部件之间的关系和交互变得很困难。基于此原因，很有必要将模型组织进模块。模块被用来作为组织相关概念和任务以便降低复杂性的一种方法。

模块在许多项目中被广泛使用。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。理解了模型间的交互之后，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。强烈推荐将高关联度的类分组到一个模块以提供尽可能大的内聚。有很多类型的内聚。最常用到的两个是通信性内聚和功能性内聚。通信性内聚通常在模块的部件操作相同的数据时使用。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。功能性内聚在模块中的部件协同工作以完成定义好的任务时使用。这被认为是最佳的内聚类型。

在设计中使用模块是一种增进内聚和消除耦合的方法。模块应该由在功能上或者逻辑上属于一体的元素构成，以保证内聚。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替代调用模块中的三个对象，因为这可以降低耦

合。低耦合降低了复杂性并增强了可维护性。当要执行定义好的功能时，模块间仅有极少的连接会让人很容易理解系统是如何工作的，这要比每个模块同其他的模块间存在许多关联好很多。

我们应该选择那些能够表述系统内涵并且包含具有内聚性概念集的模块。这种方式通常会使得模块之间松耦合，但是如果没有让模型中的概念更加清晰明白，它也能找到一个或许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。寻找每一个能够被独立理解和辨别的概念间的低耦合。重定义模型，直到它被按照高级别的领域概念区分开来，而且对应的代码也被很好地解耦合。

给定的模块名称会成为通用语言的组成部分。模块和它们的名字应该能反映对领域的深层理解。

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的变化。强烈推荐拥有一些弹性，允许模块随这项目的进展而演化，并且不被冻结。大家都明白模块的重构成本要比类的重构昂贵的多，但如果一个模块被发现存在设计错误，最好是通过变更模块来定位它，然后再找其他的解决途径。

聚合

本章的最后 3 个模式将处理不同的建模挑战，它们跟领域对象的生命周期相关。领域对象在它们的生命期内会历经若干种状态，直到最后消亡。有时它们会被保存到一个永久的位置，例如数据库中，这样可以在以后的日子里被检索到，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一个负面的影响。我们将引入 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。不管在设计时做了多少考虑，我们都会看到许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。对模型中的每个可导航的关联而言，都应该有对应的软件机制来强调它。领域对象间实际的关联在代码中结束，有时甚至却在数据库中。客户和用它名字开立的银行账户之间存在的一个 1 对 1 的关系会被表现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，一个表存放客户信息，另一个表存放账户信息。

来自模型的挑战是通常不让它们尽量完整，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，否则就要时常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转化成一个对象和一组其他对象之间的一个关联，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，使得对这样的对象的生命周期管理变得困难。关联的数字应该被尽可能消减。首先，要删除模型中非本质的关联关系。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，所以我们要清除它们。其次，可以通过增加约束的方式来消减多重性。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。每一辆汽车都有一个引擎，并且引擎在运转时，都会属于一辆汽车。这种关系是双向的，但是很容易成为汽车拥有引擎，而不用考虑反向的。

在我们消除和简化了对象间的关联后，我们仍然会遭遇到很多的关系。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，必须要保证所有的引用都被删除了。如果许多对象保有这样的引用，则很难确保它们全被清除了。同样，如果一个客户的某些数据发生了变化，系统必须确保在系统中执行了适当的更新，数据一致性必须得到保证。这通常被提交到数据库层面进行处理。事务通常用来确保数据一致性。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，导致性能极差。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。

通常也有必要确保不变量。不变量是在数据发生变化时必须维护的那些规则。这在许多对象与数据发生变化的对象保持引用时更难实现。

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。许多时候不变量被应用到密切相关的对象，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相关的对象。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。根可以保持对任意聚合对象的引用，并且其他的对象可以持有任意其他的对象，但一个外部对象只能持有根对象的引用。如果边界内有其他的实体，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进行变更，或者让根来执行某些活动。根能够变更其他的对象，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或者移除，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它们当中的任何一个了。当针对根对象的修改间接影响到聚合内的其他的对象，强化不变量变得简单了，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，这将变得越发困难。在这种情况下想强化不变量意味着讲某些逻辑放到外部对象中去处理，这不是我们所期望的。

根对象可能将内部的临时引用传递给外部对象，作为限制，当操作完成后，外部对象不能再持有这个引用。一个简单的实现方式是向外部对象传递一个值对象的拷贝。在这个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。

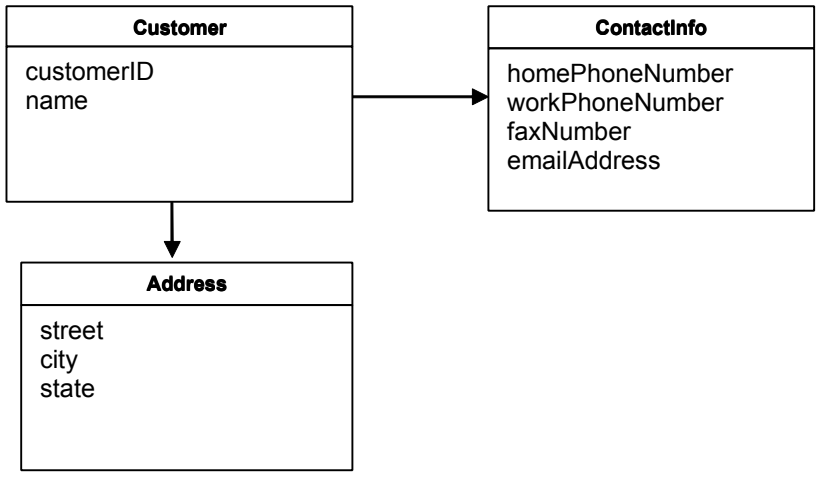
如果聚合对象被保存到数据库中，只有根可以通过查询来获得。其他的对象只能通过导航关联来获得。

聚合内的对象可以被允许持有对其他聚合的根的引用。

根实体拥有全局的标识符，并且有责任管理不变量。内部的实体拥有内部的标识符。

将有关的实体和值对象放置到聚合中并且围绕它们定义边界。选择一个实体作为每个聚合的根，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。临时对内部成员的引用仅可以被传递给一个单独的操作使用。因为根控制了访问，将不能盲目对内部对象进行变更。这种安排让强化聚合内对象的不变量变得可行，并且对聚合而言，它是一个处于任何变更状态的整体。

一个简单的聚合的案例如下图所示。客户是聚合的根，并且其他所有的对象都是内部的。如果需要地址，一个它的拷贝将被传递到外部对象。



工厂

实体和聚合通常会很大很复杂，根实体的构造函数内的创建逻辑也会很复杂。实际上通过构造器努力构建一个复杂的聚合也与领域本身通常做的事情相冲突，在领域中，某些事物通常是由别的事物创建的（例如电器是在组装线上被创建的）。这看上去是用打印机构建自己。

当一个客户程序对象想创建另一个对象时，它会调用它的构造函数，可能传递某些参数。但是当对象构建是一个很费力的过程时，对象创建涉及了好多的知识，包括：关于对象内部结构的，关于所

含对象之间的关系的以及应用其上的规则等。这意味着对象的每个客户程序将持有关于对象构建的专有知识。这破坏了领域对象和聚合的封装。如果客户程序属于应用层，领域层的一部分将被移到了外边，扰乱整个设计。实际上，给我们塑胶、橡胶、金属、硅，让我们构建自己的打印机。这不是不可能完成的，但这样做值得吗？

一个对象的创建可能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。

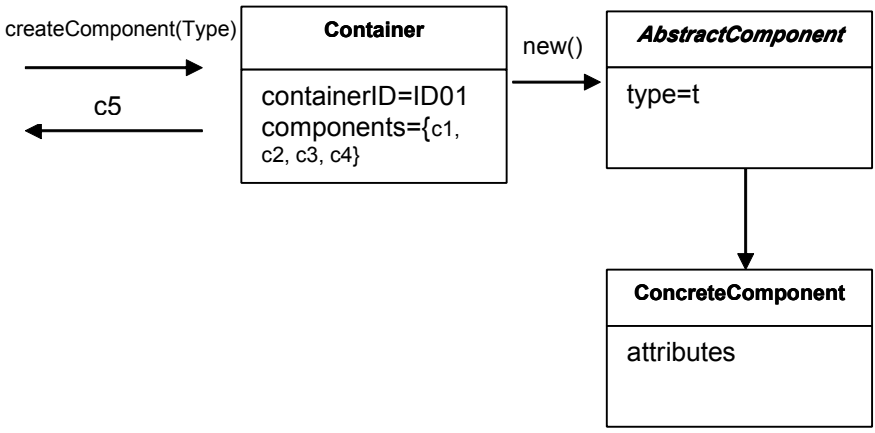
因此，有必要引入一个新的概念，这个概念可以帮助封装复杂的对象创建过程，它就是**工厂**（Factory）。工厂用来封装对象创建所必需的知识，它们对创建聚合特别有用。当聚合的根建立时，所有聚合包含的对象将随之建立，所有的不变量得到了强化。

将创建过程原子化非常重要。如果不这样做，创建过程就会存在对某个对象执行了一半操作的机会，将这些对象置于未定义的状态，对聚合而言更是如此。当根被创建时，所有对象服从的不变量也必须被创建完毕，否则，不变量将不能得到保证。对不变的值对象而言则意味着所有的对象被初始化成有效的状态。如果一个对象不能被正常创建，将会产生一个异常，确保没有返回一个无效值。

因此，转变创建复杂对象和聚合的实例的职责给一个单独的对象，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。提供一个封装了所有复杂组装的接口，客户程序将不再需要引用要初始化的对象的具体的类。将整个聚合当作一个单元来创建，强化它们的不变量。

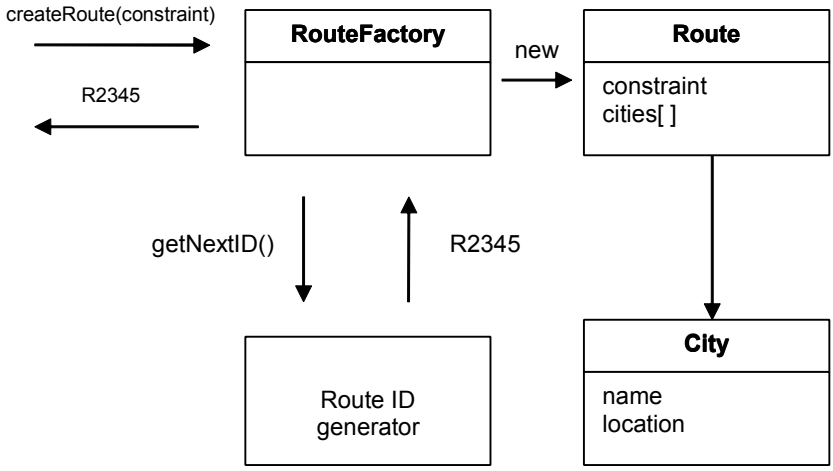
有很多的设计模式可以用来实现工厂模式。由 Gamma 等人著的《设计模式》一书中对此有详细描述，并介绍了两种不同的模式：工厂方法和抽象工厂。我们不会努力从设计的视角引入模式，而是从领域建模的角度来引入它。

工厂方法是一个对象的方法，包含并隐藏了必要的创建其他对象的知识。这在一个客户程序试图创建一个属于某聚合的对象时是很有用的。解决方案是给聚合的根增加一个方法，这个方法非常关心对象的创建，强化所有的不变量，返回对那个对象的引用或者拷贝。



Container 包含着许多组件，这些组件都是特定类型的。当这样的一个组件被创建后能自动归属于一个 Container 是很有必要的。客户程序调用 Container 的 createComponent(Type t)方法，Container 实例化一个新的组件。组件的具体类取决于它的类型。在创建之后，组件被增加到 Container 所包含的组件的集合中，并且返回给客户程序一个拷贝。

当创建一个极其复杂的对象或者创建的对象涉及到创建其他一系列的对象时（例如，创建一个聚合），需要大量的时间。隐藏聚合的内部构建所需要的任务可以用一个单独的工厂对象来完成。让我们考虑一个程序模块的案例，计算一辆汽车从起点行驶到终点的路线，而且被给定了很多的约束。用户登录 Web 站点后运行应用程序，并指定其中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。被创建的路线可以被登录的用户评论，而且这些信息都需要被保存下来。这样就可以在客户下次登录时检索出它们。



路线 ID 的生成器被用来给每条路线创建一个唯一的标识符，这对一个实体而言是非常必要的。

当创建一个工厂时，我们被迫违反一个对象的封装原则，而这应该小心行事。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，我们需要确认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。跟的构建与聚合内的其他对象的创建是相关的。会有一些逻辑放置到聚合中，这些逻辑不属于任何一个对象，因为它总跟其他对象的构建有关。看上去，使用一个专用的工厂类，给定创建整个聚合的任务比较合适，聚合中的包含的规则、约束和不变量将被确保在聚合内有效。这个对象会保持简单，并将完成特定的目的，不会造成复杂的构建逻辑的混乱。

实体工厂和值对象工厂是有差异的。值通常是不可变的对象，并且其所有的必需的属性需要在创建时完成。当一个对象被创建时，它是有效的，也是最终的，不会再发生变化。实体是非不可变的。它们会在以后发生变化，前面提及过设置某些属性时需要考虑所有的不变量。另一个差异源于实体需要标识符，而值对象不需要。

有时工厂是不需要的，一个简单的构造函数就足够了。在如下情况下使用构造函数：

- 构造过程并不复杂。

- 对象的创建不涉及到其他对象的创建，所有的属性需要传递给构造函数。
- 客户程序对实现很感兴趣，可能希望选择使用策略模式。
- 类是特定的类型，不涉及到继承，所以不用在一系列的具体实现中进行选择。

其他观察结果是工厂需要从无到有创建一个新对象，也或者它们需要从先前已经存在但可能已经持久化到一个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，涉及了一个和创建一个新对象完全不同的过程。对象已经有一个了，对不变量的违反将区别对待。当一个新的对象从无到有创建时，任何对不变量的违反都会产生一个异常。我们不能在对象从数据库重建时也这样处理。这个对象需要的是如何修复，这样它们才是可工作的，否则就会有数据的丢失。

资源库

在模型驱动设计中，对象从被创建开始，直到被删除或者被归档结束，是有一个生命周期的。一个构造函数或者工厂可应用来处理对象的创建。创建对象的整体作用是为了使用它们。在一个面向对象的语言中，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，客户程序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保证客户始终对需要的对象保持一个引用，或者是对关注的对象保有引用。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可能其实并不需要保持的一系列的引用。这增加了耦合性，创建了一系列本不需要的关联。

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，它会被保持到一个被持久化的状态，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，它会通过一个实体经由对关联的导航来获得。我们可以推导出大部分的对象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。一个不好的解决方案是客户程序必须知道要访问数据库所需的细节。例如，客户需要创建 sql 查询语句来检索所需的数据。数据库查询可能会返回一组记录，甚至暴露其内部的更细节信息。当许多客户程序不得不直接从数据库创建对象时，会导致这样的代码在整个模型中四散。从这点上讲领域模型做出了妥协。它必须处理许多基础设施的细节而不是处理领域概念。如果我们做出了一个对数据库进行变更的变更那会怎么样呢？所有四散的代码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，它极有可能存储聚合中的一个内部对象。这会破坏聚合的封装性，带来未知的结果。

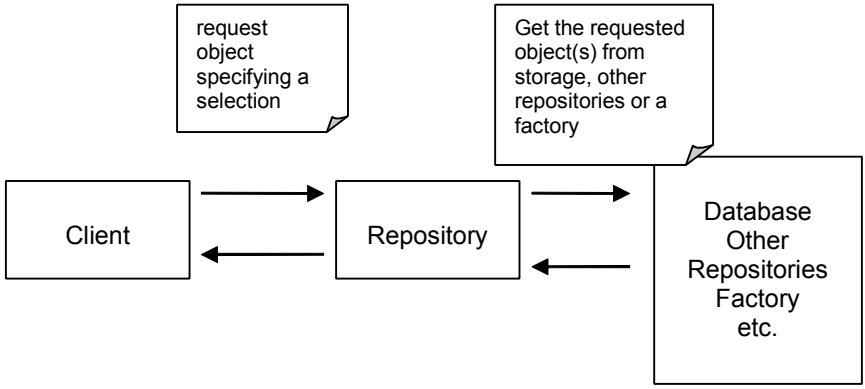
客户程序需要有一个获取已存在领域对象引用的实际方式。如果基础设施让这变得简单，客户程序的开发人员可能会增加更多的可导航的关联进一步混乱模型。从另一方面讲，他们可能使用查询从数

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递归。领域逻辑分散到查询和客户代码中，实体和值对象变得更像数据容器。应用到众多数据库访问的基础设施的技术复杂性会迅速蔓延在客户代码中，开发人员不再关注领域层，所做的工作跟模型无关了。最终的结果是丢失了对领域的关注，设计做了妥协。

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所需的引用。只需从资源库中获取它们，于是模型重获它应有的清晰和焦点。

资源库会保存对某些对象的引用。当一个对象被创建出来时，它可以被保存到资源库中，然后以后使用时可从资源库中检索到。如果客户程序从资源库中请求一个对象，而资源库中并没有它，就会从存储介质中获取它。换种说法是，资源库作为一个全局的可访问对象的存储点而存在。

资源库可能包含一定的策略。它可能基于一个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。最终结果是领域模型同需要保存的对象和它们的引用中解耦，可以访问潜在的持久化基础设施。

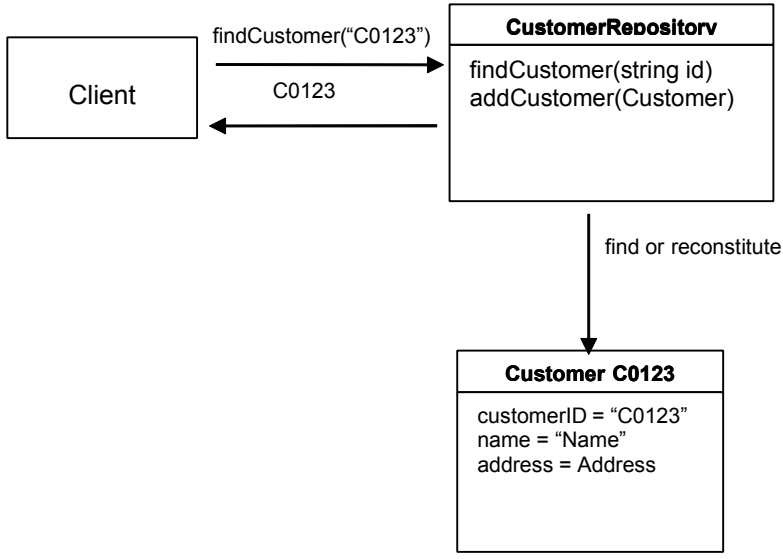


对于需要全局访问的每种类型的对象，创建一个新对象来提供该类型所有对象的内存列表的对应。通过一个共知的全局接口设置访问。提供方法来增加或者删除对象，封装向数据存储中插入或者删

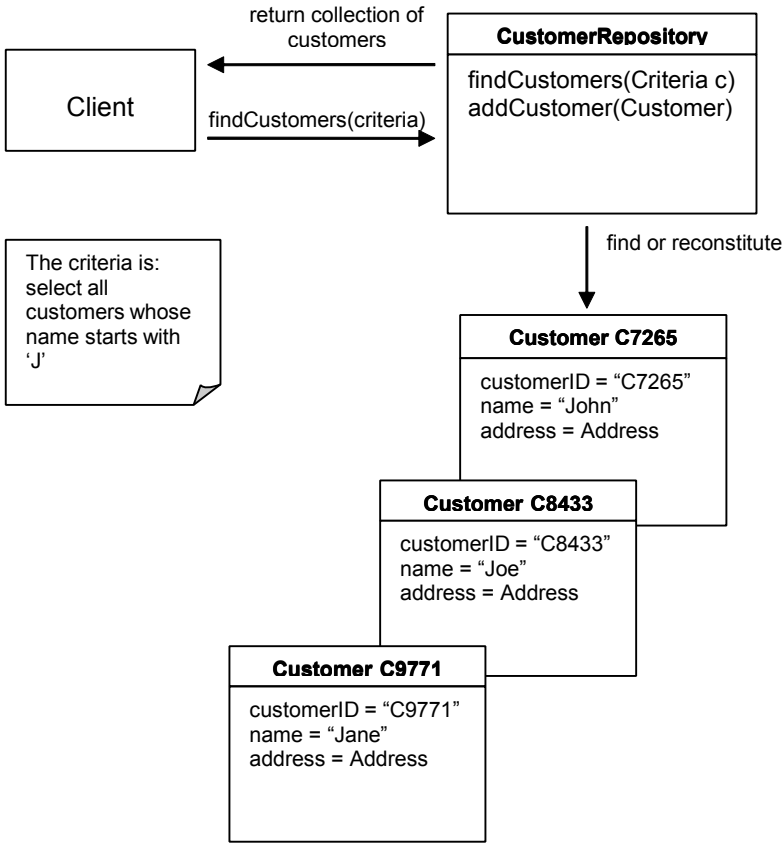
除数据的实际操作。提供基于某种条件选择对象的方法，返回属性值符合条件的完全实例化的对象或者一组对象，继而封装实际的存储和查询技术。仅对真正需要直接访问的聚合根提供资源库。让客户程序保持对模型的关注，把所有的对象存储和访问细节委托给资源库。

资源库可能包含用来访问基础设施的细节信息，但它的接口应非常简单。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。实体可能通过传递其标识符被轻易指定。其他选择条件可能由一组对象的属性构成。资源库将针对这组条件来比对所有的对象，并返回符合条件的那些对象。资源库接口可能包含用来执行某些辅助计算（例如特定类型对象的数量）的方法。

看上去资源库的实现可能会非常类似于基础设施，但资源库的接口是纯粹的领域模型。

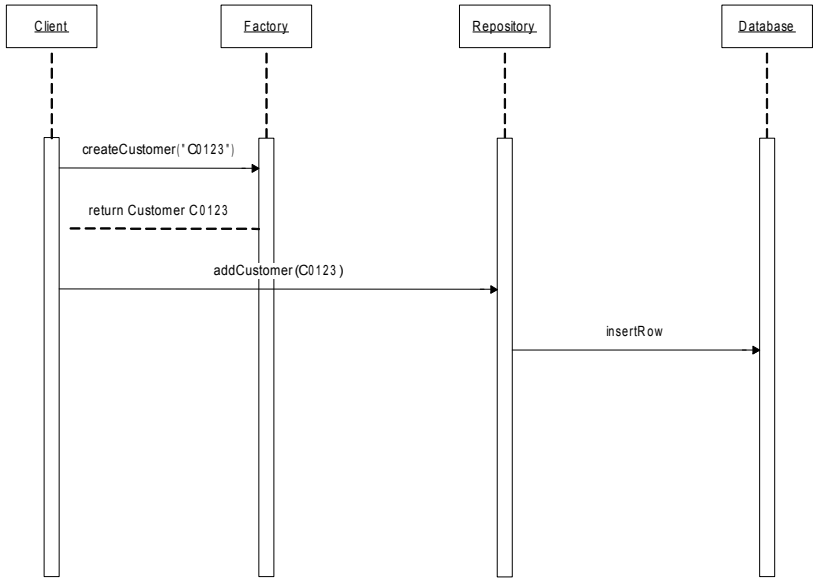


另一种选项是用规约（specification）指定一个查询条件。规约允许定义更复杂的条件，见下图：



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，它们都能帮助我们关联领域对象的生命周期。然而工厂关注的是对象的创建，而资源库关心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。对象可以用构造函数创建，也可以被传递给一个工厂来构建。从这个原因上讲，资源库也可以被看作一个工厂，因为它创建对象。不过它不是从无到有创建新的对象，而是对已有对象的重建。我们将不把资源库视为一个工厂。工厂创建新的对象，而资源库应该是用来发现已经创建过的对象。当一个新对象被添加到资源

库时，它应该是先由工厂创建过的，然后它应该被传递给资源库以便将来保存它，见下面的例子：



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

4

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表现领域的模型的重要性了。我们给出了一些可以用来创建有效模型的技术指南。模型现在已经跟它所源自的领域紧密关联了。我们也已经说过代码设计应该围绕模型展开，模型自身也会基于设计决定而有所增进。脱离了模型的设计会导致软件不能反映它所服务的领域，甚至可能得不到期望的行为。建模如果得不到设计的反馈或者缺少了开发人员的参与，会导致必须实现模型的人很难理解它，并且可能对所用的技术不太适合。

在设计和开发过程中，我们需要一次次得停下来，查看代码。这意味着到了重构的时间了。重构是不改变应用行为而重新设计代码以让它更好的过程。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某些 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确保未破坏任何事情提供很大的帮助。

代码重构有很多种方式，甚至存在重构模式。这些模式描述了一个重构的自动化方法。有些基于这些模式的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，重构起来会非常困难。这类重构更多处理的是代码和它的质量。

还有另一种类型的重构，跟领域和它的模型相关。有时会对领域有新的理解，有些事物变得更清晰，或者两个元素间的关系被发现。所有的这些会通过重构被包括到设计中。让拥有表现力的代码更易读和理解是非常重要的。通过阅读代码，我们可能不光了解代码是

什么，同时了解它为什么要这样。只有这样才能让代码真正捕获模型的主旨。

基于模式的技术性重构，可以被组织并结构化。面向更深层理解不能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多变性不可能提供让我们按照机械化的方式进行建模的方式。一个好的模型产生于深层的思考、理解、经验和才能。

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，而动词则成为方法。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。头脑中若没有对代码建立灵活性的概念，那么这样的代码就会很难维护。当需要发生变化时，你会看到代码与你交锋，会在原本应该很容易重构的事情上花费很多的时间。

使用经过验证的基础构造元素并使用一致的语言将给开发工作带来成效。这会减少不良模型所带来的挑战，我们可以捕获到领域专家非常关心的内容，并且以此来驱动实际的设计。一个忽略表面内容且捕捉到本质内涵的模型是一个深层模型。这会让软件更加和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，重构描述的是从技术动机的代码转换。重构同样可以由对领域的深入理解，以及对模型及其代码表达进行相应的精化所推动，。

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，否则一个极其复杂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，每次给设计增加非常小的价值，有时，会有很少的变更，但造成很大的差异。这就是突破。

我们会从一个粗糙的、浮浅的模型开始，然后基于对领域的深层理解以及对关注点的理解来细化它和设计。我们会对它增进新的概念和抽象，然后执行设计的重构。每一次精化会让设计更清晰。这就建立好了突破的前提。

突破常包括思维上的变化，如同我们理解模型一样。它也是项目重要进程中的一个发起者，但它也有一些缺点。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。它也是有风险的，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，我们需要让隐式的概念显式化。当我们跟领域专家交谈时，我们交互了很多的思想和知识。某些概念成为了通用语言，但也有些一直未被重视。它们是隐式概念，用来解释以及在领域中的其他的概念。在精化设计的过程中，某些隐式概念吸引了我们注意力。我们发现它们当中的某些在设计中担当了重要的角色。因此我们需要将这些概念显式化。我们应该为它们建立类和关系。当它们发生时，我们就拥有了突破的机会。

隐式概念可能不会仅于此。如果它们是领域概念，它们可能被引入模型和设计中。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话语。我们在建模和设计过程中使用的话语包含了许多关于领域的信息。开始时可能不会太多，或者某些信息没有被正确地使用。某些概念可能不能被完全理解，或者理解完全是错误的。这是在学习一个新的领域所必须经历的。但因为我们构建了我们的通用语言，关键概念会被加入其中。在那里我们可以开始查找隐含概念。

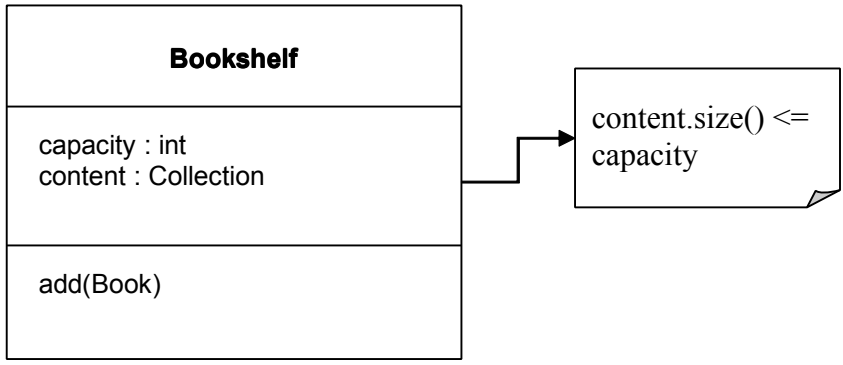
有时设计的部分可能不会那么清晰，一堆关系会让路径的计算变得难以进行，或者计算的过程会复杂到难以理解。这在设计中显得极其笨拙，但这也是寻找隐藏的概念的绝佳之所，可能我们错过了什么。如果某个关键概念在破解谜团时缺失了，其他的事物就不得不替代它完成它的功能。这会让某些对象变胖，给它们增加了一些本不应该属于它的行为。设计的清晰度受到了破坏。努力寻找是否有缺失的概念，如果找到一个，就把它显式化。对设计进行重构让它更简单更具柔性。

当我们构建知识时很可能会遭遇到矛盾。某个领域专家所讲的看上去跟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跟另一个需求矛盾。某些这样的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，或者只是因为讲解时缺乏了精确度。

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。

其他明细的挖掘模型概念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，它们包含了许多关于特定领域的知识。这些书通常不包含要展现的领域的模型，它们包含的信息需要进一步处理、精炼并细化。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。

在让概念显式化时，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示不变量的方式。无论对象的数据发生了变化，都要考虑不变量。这可以简单地通过把不变量的逻辑放置在一个约束中去实现它。下面是一个简单的案例，其目的是为了解释这个概念，而不是为了表现对相似案例的建议方法。



我们可以向一个书架添加书籍，但是我们永远不能增加超过它容量的部分。这儿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让它显示化有很多优点。它很容易阅读，并且每个人都会注意到 add（）方法从属于这个约束。如果约束变得更复杂，这可以为向该方法增加更多逻辑提供增长空间。

过程通常在代码中被表现为 procedure。从我们开始使用面向对象语言后我们就不再用一个过程化的方法，所以我们需要为过程选择一个对象，然后给它增加行为。最好的实现过程的方式是使用服务。其他的处理过程的不同的方式如，将算法封装进一个策略对象。并不是所有的过程都必须显式化。如果通用语言中提到了某个过程，那就是将它显式实现的时机了。

我们在此要介绍的最后一个将概念显式化的方法是规约。简单得说，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。在这些规则中，某些只是用来回答“是”和“否”的一组问题的，某些规则可以被表现成一系列操作布尔值的逻辑上的操作，最终结果也是一个布尔值。一个这样的案例是在一个客户对象上执行测试，看它是否符合特定的信用条件。这个规则可以被表现成一个方法，起名叫 isEligible（），并且可以附加到客户对象上。但这个规则不是严格基于客户的数据进行操作的一个简单的方法。评估规则涉及到验证客户的信用，检查他过去是否支付过他的债务，检查他是否具有足够的余额等。这样的业务规则可能非常的大，非常复杂，让对象的功能肿胀，不再满足其原始的目的。在这种情况下我们可能会试图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，到了重构的时候了。

规则应该被封装到一个负责它的对象中，这将成为客户的规约，并且被保留在领域层中。新的对象将包含一系列布尔方法，这些方法用来测试一个客户对象是否符合某种信用。每一个方法担负了一个小的测试的功能，所有的方法可以通过组合对某个原始问题给出答案。如果业务规则不能被包含到一个规约对象中，对应的代码会遍布到无数的对象中，让它不再一致。

规约用来测试对象是否满足某种需要，或者他们是否已经为某种目的准备完毕。它也可以被用来从一个集合中筛选一个特定的对象，或者作为某个对象的创建条件。

通常情况下，一个单个的规约负责检查是否满足一个简单的规则，若干个这样的规约组合在一起表现一个复杂的规约，例如：

```
Customer customer =
customerRepository.findCustomer(customerIdentiy);
```

```
…
Specification customerEligibleForRefund = new Specification(

        new CustomerPaidHisDebtsInThePast(),

        new CustomerHasNoOutstandingBalances());

if(customerEligibleForRefund.isSatisfiedBy(customer) {
        refundService.issueRefundTo(customer);

}
```

测试简单的规则变得非常简单，只需阅读这段代码，这段代码很明显地告诉我们一个客户是否符合偿还条件的真正含义。

5 保持模型一致性

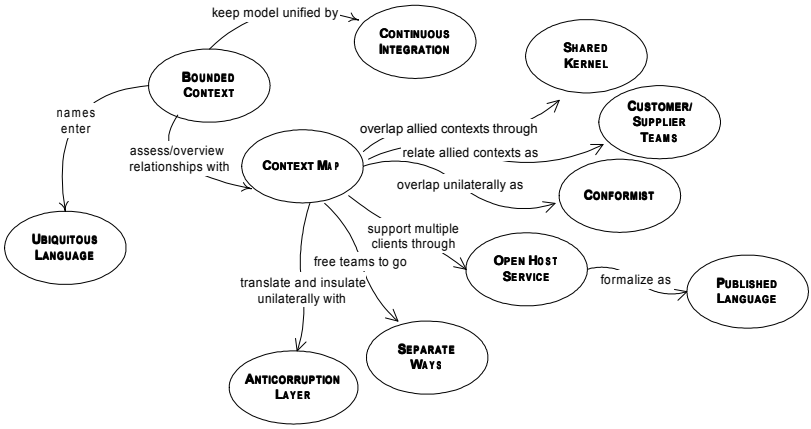
本章涉及的是需要若干个团队通力配合的大型项目。当有多个团队时，我们必须面对一系列不同的挑战，开发这样一下项目的认为需要不同的管理和协作。企业级项目通常都是大型项目需要使用许多的技术和资源。这样的项目的设计将仍然基于一个领域模型展开，并且我们需要采用适当的 measure 来确保项目成功。

当多个团队开发一个项目时，代码开发是并行的，每个团队都会被指派模型的一个特定部分。那些部分不是独立的，多少都有些关联性。它们都是从一个大的模型出发，然后实现其中的一部分。我们可以这样说，其中的一个团队创建了一个模块，然后提供给其他的团队使用。某个团队的开发人员开始在自己的模块中使用这个模块，但发现还缺少一些功能，于是他增加了这个功能并放到代码库里面，以便所有的人都能使用。但是他也许没有意识到，这其实是对模型的一个变更，这个变更很有可能破坏系统的功能。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，但对其他地方则并不是非常了解。

好的开始未必就是成功的一半。比较常见的情况是，很多开始良好的模型和流程到最后都一塌糊涂。模型的首要需求是一致性，条款统一和没有矛盾。模型的内部一致被称为“统一”。一个企业项目应该有一个模型，涵盖企业的整个领域，没有矛盾和重叠的条款。统一的企业模型是不容易实现的理想状态，有时甚至都不值得尝试。这些项目需要许多团队的通力协作。在开发流程中，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。要调和这些团队是很有挑战性的。也许他们属于不同的部门，有着不相干的管理。当模型的设计开始部分独立时，我们就开始面临失去模型完整性的可能性。试图为整个企业项目维持一个大的统一模型以获得模型完整性的做法，将不会有什么作为。解决方案不是那么明显

的，因为它远超乎我们所已经学习到的知识。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数个较小的部分。只要遵守相绑定的契约，整合得好的小模型会越来越有独立性。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。

我们会提供一整套技术来保持模型的完整性。下面的章节阐述了这些技术，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，上下文是固定的。我们不需要去定义它。当我们创建一个假定要和其他软件，比如一个遗产应用，交互的应用时，很明显新的应用有自己的模型和上下文，它们和遗产模型和它的上下文相分离。它们不能被合并、混合或者模糊定义。所以当我们开发大的企业应用时，需要为每一个我们创建的模型定义上下文。

在任何大型项目中都存在多个模型。如果基于不同模型的代码被合并，软件就变得不稳定、不可靠而且很难理解。团队之间的沟通也会不通畅。在哪些上下文里不应该有模型通常都不是非常明确的。

如何把一个大的模型分解成小的部分没有什么具体的公式。尽量把那些相关联的以及能形成一个自然概念的因素放在一个模型里。模型应该足够小，以便能分给一个团队去实现。团队协作和沟通如果

更畅通，会有助于开发人员共同完成一个模型。模型的上下文是一个条件集合，用这些条件可以确保应用在模型里的条款都有一个明确的含义。

这儿主要的思想是定义模型的范围，画出它的上下文的边界，然后尽最大可能保持模型的一致性。要在模型涵盖整个企业项目时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就相对容易很多。要在应用到模型的地方明确定义上下文。在团队组织里明确定义边界，在应用的具体部分明确定义用法，以及像代码库和数据库 Schema 的物理显示。保持模型在这些边界里的严格一致，不要因外界因素的干扰而有异动。

被界定的上下文不是模型。界定的上下文提供有模型参与的逻辑框架。模块被用来组织模型的要素，因此界定的上下文包含模块。

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化都有可能破坏现有的功能。当使用多个模型时，每个人可以自由使用自己的那一部分。我们都知道自己模型的局限，都恪守在这些边界里。我们需要确保模型的纯洁、一致和完整。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。而且为了达到纯洁的最大化，设计还要可以被精简和提炼。

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付出，以及可能出现的不同模型间的翻译。我们不能在不同模型间传递任何对象，也不能在没有边界的情况下自由地激活行为。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。

比如，我们要创建一个用来在互联网上卖东西的在线应用。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，下单等。不论在什么时候下单，应用都需要触发一个事件，因为有人要邮寄需求的货物。我们还想做一个用于创建报表的报表界面，所以还应该能监视已有的货物数量、哪些是客户感兴趣购买的、哪些是不受欢迎的等信息的状态。开始的时候，我们用一个模型涵盖整个在线应用的领域。很自然地就会这样做，因为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 E 商店应用其实和报表并不是那么相关联。它们有不同的考虑，在不同的概念下操作，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，两个应用都访问到它们。

推荐的做法是为每一个领域创建一个独立的模型，一个为在线交易，一个为报表。它们两个可以在互不干涉的情况下继续完善，甚至可以变成独立的应用。也许报表应用会用到在线交易（e-commerce）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。

还需要有个通讯系统来通知仓库管理人员订单信息，这样他们就可以邮寄被购买的货物。邮寄人员也会用到一个可以提供给他们详细的关于所购买的物品条目、数量、客户地址以及邮寄需求等信息的应用。不需要使在线商店（e-shop）模型覆盖两个活动领域。对在线商店而言，用异步通讯的方式给仓库发送包含购买信息的 Value 对象要相对简单的多。这样就明确地有两个独立开发的模型，我们只需要保证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，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，模型很容易被分解。团队越大，问题越大，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，基本上也就失去了完整性和一致性的价值。

就是一个团队工作于一个界定的上下文，也有犯错误的空间。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部分所扮演的角色。如果一个人不理解对象间的关系，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在不知道已经有自己所需代码的情况下增加重复代码，或者担心破坏现有的功能而不改变已有的代码选择重复增加。

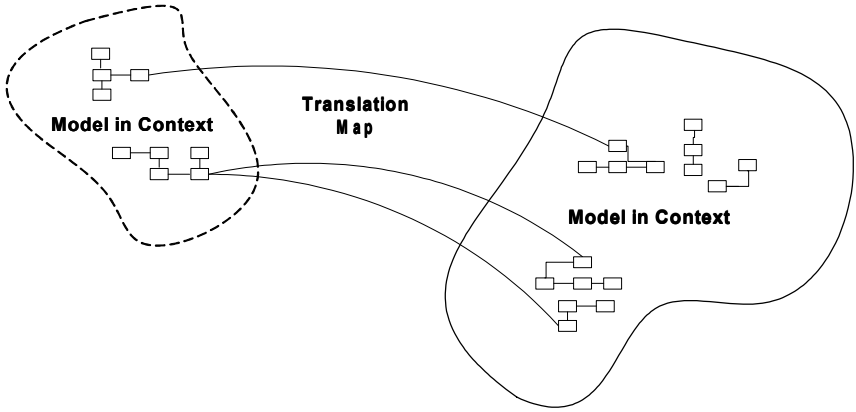
模型不是一开始就被完全定义。先被创建，然后基于对领域新的发现和来自开发过程的反馈等再继续完善。这意味着新的概念会进入模型，新的部分也会被增加到代码中。所有的这些需求都会被集成进一个统一的模型，进而用代码实现之。这也就是为什么持续集成在界定的上下文中如此必要的原因。我们需要这样一个集成的过

程，以确保所有新增的部分和模型原有的部分配合得很好，在代码中也被正确地实现。我们需要有个过程来合并代码。合并得越早越好。对小的独立团队，推荐每日合并。我们还需要适当地采用构建流程。合并的代码需要自动地被构建，以被测试。另外一个必须的邀请是执行自动测试。如果团队有测试工具，并创建了测试集，那么测试就可以运行在每个构建上，任何错误都可以被检查出来。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，合并、构建、和测试流程才刚开始。

持续集成是基于模型中概念的集成，然后再通过测试实现。任何不完整的模型在实现过程中都会被检测出来。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，每个模型有自己的界定的上下文。建议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，也能很好地将模型集成和实现。但是每个团队都工作于自己的模型，所以最好让每个人都能了解所有的模型。上下文映射（Context Map）是指抽象出不同界定上下文和它们之间关系的文档，它可以是像下面所说的一个试图（Diagram），也可以是其他任何写就的文档。详细的层次各有不同。它的重要之处是让每个在项目中工作的人都能够得到并理解它。

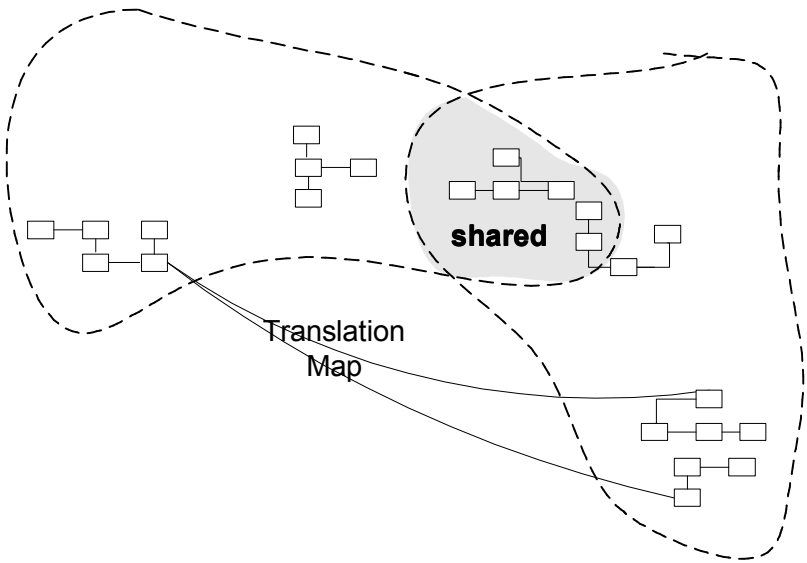


只有独立的统一模型还不够。它们还要被集成，因为每个模型的功能都只是整个系统的一部分。在最后，单个的部分要被组织在一起，整个的系统必须能正确工作。如果上下文定义的不清晰，很有可能彼此之间互相覆盖。如果上下文之间的关系没有被抽象出来，在系统被集成的时候它们就有可能不能工作。

每个界定的上下文都应该有一个作为 Ubiquitous Language 一部分的名字。这在团队之间沟通整个系统的时候非常有用。每个人也应该知道每个上下文的界限以及在上下文和代码之间的映射等。一个通常的做法是先定义上下文，然后为每个上下文创建模型，再用一个约定的名称指明每个模型所属的上下文。

在接下来的章节中，我们要讨论不同上下文之间的交互。我们会列举很多可用来创建上下文映射的模式，被创建的上下文有清晰的角色和被指明的关系。在上下文之间，共享内核（Shared Kernel）和客户-供应商（Customer-Supplier）是具有高级交互的模式。隔离通道（Separate Way）是在我们想让上下文高度独立和分开运行时要用到的模式。还有两个模式处理系统和继承系统或者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，持续集成可能就遥不可及了。尤其是在团队不具备相关的技术或者行政组织来维护持续集成，或者是某个团队又大又笨拙的时候。所以这些界定的上下文可能要被良好地定义和multiple teams formed。

协同工作于有紧密关系的应用程序上的不协调团队有时会进展很快，但他们所做的有可能很难整合。他们在转换层和技巧花样上花费了过多的时间，而没有在最重要的持续集成上下功夫，做了许多重复劳动也没有体味到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。当然除了模型的子集部分，还要包括代码自己或者和模型相关联的数据库设计子集。这个明确被共享的东西有特别的状态，没有团队之间的沟通不能做修改。

要经常整合功能系统，但是可以不用像在团队里进行持续集成那么频繁。在集成的时候，在两个团队里都要运行测试。

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代码，还要必须整合所做的修改。如果团队用的是内核代码的副本，那么要尽可能早地融合（Merge）代码，至少每周一次。还应该使用测试工具，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通知另一个团队，团队之间密切沟通，使大家都能了解最新的功能。

客户-供应商

我们经常会遇到两个子系统之间关系特殊的时候：一个严重依赖另一个。两个子系统所在的上下文是不同的，而且一个系统的处理结果被输入到另外一个。它们没有共享的内核，因为从概念上理解也许不可以有这样一个内核，或者对两个子系统而言要共享代码在技术上也不可能实现。

让我们回到先前的例子。我们曾讨论了一个关于在线商店应用的模型，包括报表和通讯两部分内容。我们已经解释说最好要为所有的那些上下文创建各自分开的模型，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。假设我们同意有分开的模型，那么在

Web 商店子系统和报表系统间的关系是什么样子的呢？共享内核看上去不是好的选择。子系统很可能会用不同的技术被实现。一个是纯浏览器体验，而另一个可能是丰富的 GUI 应用。尽管如果报表应用是用 Web 接口实现，各自模型的注意概念也是不同的。也许会有越界的情况，但还不足以应用共享内核。所以我们选择走不同的道路。另外，E 商店子系统并不全依赖报表系统。E 商店应用的用户是 Web 客户，是那些浏览商品并下单的人。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。E 商店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 E 商店应用保存的数据。它还需要一些额外的信息以执行它提供的报表服务。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。某个客户访问的链接可能多于其他人等。这样的信息对 E 商店应用没有什么意义，但是对报表应用却意义非凡。由此，供应商子系统不得不实现一些客户子系统会用到的规范。这是联系两个子系统的纽带。

另外一个和所用到的数据库相关联的需求是它的 Schema。两个应用将使用同一个数据库。如果 E 商店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，所以它需要一些 Schema 的稳定性。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 E 商店应用这不代表是个问题，但对报表应用这肯定是一个问题。这两个团队需要沟通，可能还需要在同一个数据库下工作，然后决定什么时候执行变更。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变更和进展，而不是在 E 商店应用上等待。如果 E 商店应用团队有否决权，他们也许会对要在数据库上做的变更强制加上限定，从而伤害到报表团队的行为。如果 E 商店团队能独立行动，他们就会迟早破坏约定，然后做一些报表团队还没有准备好的变更。但这个模式在团队处于统一管理的情况下有效，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，应该就开始“演出”了。报表团队应该扮演客户角色，而 E 商店团队应该扮演供应商角色。两个团队应该定期碰面或者提邀请，像一个客户对待他的供应商那样交谈。客户团队应该代表系统的需求，而供应商团队据此设置计划。当客户团队所有的需求都被激发出来后，供应商团队就可以决定实现它们的时间表。如果认为一些需求非常重要，那么应该先实现它们，延迟其他的需求。客户团队还需要输入和能被供应商团队分享的知识。这个过程 Flows one way，但是有时是必要的。

两个子系统之间的接口需要预先明确定义。另外还要创建一个统一的测试集，在任何接口需求被提出的时候用于测试。供应商团队可以在他们的设计上大胆地工作，因为接口测试集的保护网会在任何有问题的时候报警。

在两个团队之间确定一个明显的客户/供应商关系。在计划场景里，让客户团队扮演和供应商团队打交道的客户角色。为客户需求做充分的解释和任务规划，让每个人理解相关的约定和日程表。

联合开发可以验证期望（Expected）接口的自动化验收测试。将这些测试增加到供应商团队的测试集里，作为它的持续集成的一部分运行。这个测试能使供应商团队放心地做修改，而不用担心影响客户团队的应用。

顺从者

在两个团队都有兴趣合作时，客户-供应商关系是可行的。客户非常依赖于供应商，但供应商不是。如果有管理保证合作的执行，供应商会给于客户需要的关注，并聆听客户的要求。如果管理没有清晰地界定在两个团队之间需要完成什么，或者管理很差，或者就没有管理，供应商慢慢地会越来越关心它的模型和设计，而也越来越疏于帮助客户。毕竟他们有自己的工作完成底线。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，客户团队深受其害。在团队属于不同公司的情况下，这样的事情也会发生。交流是困难的，供应商的公司也许没兴趣在关系沟通上投资太多。他们要么提供少许帮助，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的需要提供帮助时，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，但是他们很有可能完不成。美丽的许诺导致客户团队根据那些从来都不会存在的功能制定计划。客户的项目会被一直耽搁，知道团队最终学会如何协同工作。符合客户团队需要的接口不在卡片里。

客户团队没有多少选择。最常见的现象是将它从供应商那儿分割出来，自己完全独立。在后面的“分割方法”模式中我们在对它做详细介绍。有时供应商子系统提供的好处不值得所付出的努力。创建

一个分割的模型，以及在不考虑供应商模型的情况下做设计也许更简单些。但并不总是管用。

有时在供应商模型里会有些数据，这时不得不维持一个连接。但是因为供应商团队不帮助客户团队，所以后者也不得不采取一些措施保护自己，以防止前者对模型所做更改带来的影响。他们需要实现连接两边上下文的转换层。也有可能供应商团队的模型没有被很好地理解，效用发挥不出来。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。这和共享内核很类似，但有一个重要的不同之处。客户团队不能对内核做更改。他们只能用它做自己模型的一部分，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，并提供了相应的接口时，我们就可以将这个组件看作我们自己的东西构建我们的模型。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。这会隔离出我们的模型，可以有很高的自由度去开发它。

防崩溃层

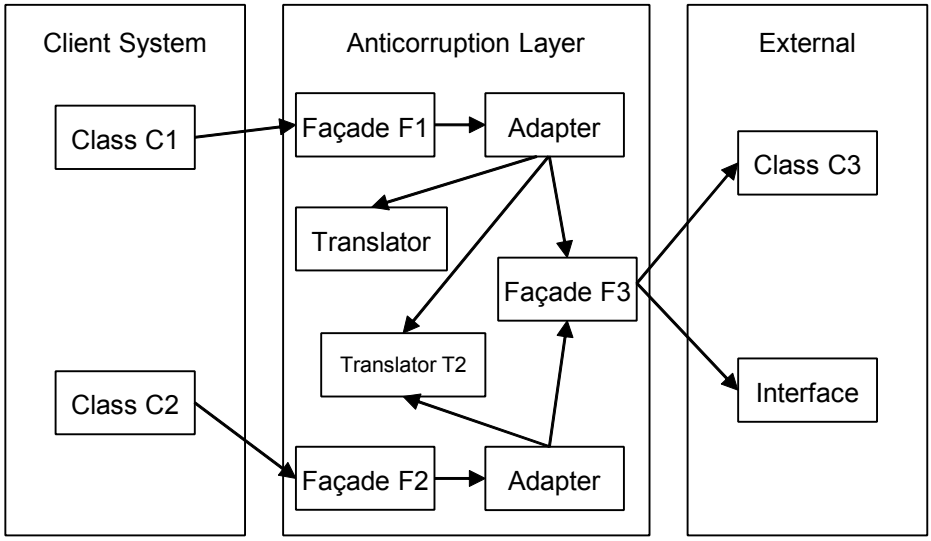
我们会经常遇到所创建的新应用需要和遗留软件或者其他独立应用相交互的情况。对领域建模器而言，这又是一个挑战。很多遗留应用从前没有用领域建模技术构建，而且它们的模型模糊不清，难于理解，也很难使用。即使做得很好，遗留应用的模型对我们也不是那么有用，因为我们的模型很可能与它完全不同。因此，在我们的模型和遗留模型之间就须要有一个集成层，这也是使用旧应用的需求之一。

让我们的客户端系统和外面的系统交互有很多种方法。一种是通过网络连接，两个应用需要使用同一种网络通信协议，客户端需要遵从使用外部系统使用的接口。另外一个交互的方法是数据库。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，我们所处理的两个系统之间传输的都是原

始数据。但是这看上去有些简单，事实是原始数据不包括任何和模型相关的信息。我们不能将数据从数据库中取出来，全部作为原始数据处理。在这些数据后面隐含着很多语义。一个关系型数据库含有和创建关系网的其他原始数据相关的原始数据。数据语义非常重要，并且需要被充分考虑。客户端应用不能访问数据库，也不能不理解被使用数据的含义就进行写入操作。我们看到外部模型的部分数据被反映在数据库里，然后进入我们的模型。

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。我们不能忽视和外部模型的交互，但是我们也应该小心地将我们的模型和它隔离开来。我们应该在我们的客户端模型和外部模型之间建立一个防崩溃层。从我们模型的观点来看，防崩溃层是模型很自然的一部分，并不像一个外部的什么东西。它对概念和行为的操作和我们的模型类似，但是防崩溃层用外部语言和外部模型交流，而不是客户端语言。这个层在两个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持久，不会受到外部模型的干扰。

我们怎么实现防崩溃层？一个非常好的方案是将这个层看作从客户端模型来的一个服务。使用服务是非常简单的，因为它抽象了其他系统并让我们在自己的范围内定位它。服务会处理所需要的转换，所以我们的模型保持独立。考虑到实际的实现，可以将服务看作比作一个 Facade（参见 Gamma et al 在 1995 年写作《设计模式》）。除了这一点，防崩溃层最可能需要一个适配器（Adapter）。适配器可以使你将一个类的接口转换成客户端能理解的语言。在我们的这个例子中，适配器不需要一定包装类，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 我们为之增加一个适配器。我们不应该为所有的服务使用一个适配器，因为这样会使我们无法清晰地处理繁多的功能。

我们还必须再增加一些组件。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，这会使用一个转换器来解决。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增加一个额外的 Facade。这会简化适配器的协议，将它和其他系统分离开来。

独立方法

到目前为止，我们尽己所能找到了整合子系统的途径，使它们协同工作，并且是在建模和设计都做得很好的情况下。这需要努力和妥协。工作于各自子系统的团队需要花费大量的时间理清子系统之间的关系，需要持续不断地融合他们的代码，执行测试以保证没有破坏任何部分。有时，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请求。妥协也是很有必要的。一方面是你独立做开发，

自由地选择概念和联盟，另一方面要确保你的模型适合另一个系统的框架。我们也许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的方法。我们需要严格地评估整合的价值，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整合难度很大，不值得这样去做，那么就应该考虑独立方法。

独立方法模式适合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相同之处的情况。它有一套自己的需求，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，它可以由有独立实现的独立模型来完成。我们应该现看看需求，然后了解一下它们是否可以被分割成两个或者多个不太相同的部分。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。这样做的好处是有选择实现技术的自由。我们正创建的应用可能会共享一个通用的瘦 GUI，作为链接和按钮的一个门户来访问每一个程序。相对于集成后端的模型，组织应用是一个较小的集成。

在继续谈论独立方法之前，我们需要明确的是我们不会回到集成系统。独立开发的模型是很难集成的，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缓冲的角色。这个层可以是个永久层，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，而是被多个服务端子系统的话，我们就需要为所有的服务端子系统创建转换层。所有的这些层会重复相同的转换任务，也会包含类似的代码。

当一个子系统要和其他许多子系统集成时，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，当需要做出改变时，也会越来越担心。

解决这一个问题的方法是，将外部子系统看作服务提供者。如果我们能为这个系统创建许多服务，那么所有的其他子系统就会访问这些服务，我们也就不需要任何转换层。问题是每一个子系统也许需要以某种特殊的方式和外部子系统交互，那么要创建这些相关的服务可能会比较麻烦。

定义一个能以服务的形式访问你子系统的协议。开放它，使得所有需要和你集成的人都能获取到。然后优化和扩展这个协议，使其可以处理新的集成需求，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增加协议，从而使得共享的协议保持简洁和精干。

精炼

精炼是从一个混合物中分离物质的过程。精炼的目的是从混合物中提取某种特殊的物质。在精炼的过程中，可能会得到某些副产品，它们也是很有价值的。

即使在我们提炼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。就是在重构多次之后，也依然会很大。对于这样的情况，就需要精炼了。思路是定义一个代表领域本质的核心域（Core Domain）。精炼过程的副产品将是组合领域中其他部分的普通子域（Generic Subdomain）。

在设计一个大系统时，有那么多分布式组件，所有的都是那么复杂而且绝对须要不出差错，这使得领域模型的本质，也就是真正的商业资产，变得模糊和不被重视。

当我们处理一个大的模型时，应该试图将本质概念从普通概念中分离出来。一开始我们举了一个关于飞空中交通监控系统的例子。我们说飞行计划包括飞机必须遵照的设计好的路线。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，不是本质上的。路线概念被用在许多领域里，可以设计一个普通的模型去描述它。空中交通监控的本质在其他地方。监控系统知道飞机要遵照的路线，但是它还会接收跟踪飞行中飞机的雷达网络输入。这个数据显示飞机真正遵照的飞行路线，而它经常和预先描述好的有些偏差。系统不得不基于飞机当前的飞行参数、飞机特

点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思维路线，它可能会在接下来的几分钟里被计算出来，也可能是几十分钟，或者是好几个小时。每一个计算都有助于决策制定过程。计算飞机轨道的终极目的是看看是不是会和其他飞机的飞行路线有交叉。在机场附近，飞机起飞或者降落时，有很多在空中盘旋或者有其他要求。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，在出现这种交叉飞行时发出警报。空中交通控制人员需要快速做出决策，改变飞机飞行路线，防止相撞发生。飞机飞得越远，计算轨道的时间就越长，做出反应的时间也越长。根据已有的数据同步飞机轨道的模块才是这个业务系统的“心脏”。可以将它标识为核心域。路线模型更像是一个普通域。

系统的核心域要看我们如何理解系统。一个简单的路线系统会将路线和与它相关概念看作核心域，而空中交通监控系统却把它们看作普通子域。一个系统的核心域有可能会变成另一个系统的普通子域。正确标识核心，以及它和其他模型之间的关系是非常重要的。

精炼模型。找到核心域，发现一个能轻松地从支持模型和代码中区分核心域的方法。强调最有价值和特殊的概念。使核心变小。

将你所有的才能都投入到核心域上，不停地努力，找到一种深刻的模型，做一个足够灵活的设计以满足系统的远景。根据其他部分对提炼后核心的支持，判断投资。

让最好的开发人员去承担核心域的任务是重要的。开发人员经常沉溺于技术，喜欢学习最好的和最新的语言，相对于业务逻辑他们更关注基础架构。领域的业务逻辑于他们毫无兴趣可言，也看不到什么回报。还记得我们前面所谈的飞机轨道案例的要点吗？当项目完成时，所有的知识都成为毫无意义的过去。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有了错误，将会导致整个项目的失败。如果核心业务逻辑不起作用，所有的技术亮点都等于垃圾。

核心域的创建不是能一朝一夕完成的。这需要一个提炼的过程，另外重构在核心越来越明显之前也是很必要的。我们需要硬性地将核心放在设计的中心位置，并划定它的界限。我们还需要重新考虑和新核心相关的其他模型的要素，也许它们也需要被重构，相关的功能也许需要被改变等。

没有特殊的知识，模型的某些部分也会增加复杂度。任何额外的事情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，或

者那些不是你首要关注但是扮演支持角色的细节都会影响模型。但是那些部分对系统功能完整性和模型的完整表达都依然是必要的。

标识出在你项目中不是推动因素的相关子域。找出那些子域的普通模型，并将它们放在不同的模型中。不要在这些模型中留下什么特殊的印记。

一旦它们被分离开，就将对它们的持续开发的优先级调得比核心域要低，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。

每个领域使用其他领域使用的概念。钱和它们相关的概念，比如行情和汇率，可以被包含在不同的系统里。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，但是可以被用在许多应用中。有下面集中方法可以实现普通子域：

1. **购买现成的方案。**这个方法的好处是可以使用别人已经做好的全套方案。随之而来的是学习曲线的问题，而且这样的方案还会引入其他麻烦。比如如果代码有凑五，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。和自己系统的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。这样做可以使你专注于核心域，从处理其他领域的重压下释放出来。不便的地方是集成外包的代码。需要和子域通信的结构需要预先定义好，还要和外包团队保持沟通。

3. **已有模型。**一个取巧的方案是使用一个已经创建的模型。世面上已经有一些关于分析模型的书，可以作为我们子域的灵感来源。直接复制原有的模型不太现实，但确实有些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。

6

领域驱动设计新进展：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。

为什么领域驱动设计一直都很重要？

基本上，领域驱动设计是我们应该专注于用户所关心领域里的重要问题的指导原则。我们的智慧应该用在理解这一领域上，和那个领域的其他专家一起将它抽象成一个概念。这样，我们就可以应用这个抽象出来的概念构造强大而灵活的软件。

它是一个永远不会过时的指导原则。不论我们何时**操作**一个复杂的领域，它都有用。大趋势是软件会应用于越来越复杂的问题，越来越趋近于业务的核心。对我来说，这一趋势好像中断了很多年，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艰深的解决方案上移开，因为有太多的数据需要传递到 Web 上，只需要简单的动作即可。因为太多的数据要传递，但短时间内在 Web 上做这一简单的事情又是比较困难的，所以消耗了软件开发的所有能力。

但是现在人们大步跨越了这一 Web 应用的基本层次，又把注意力集中在业务逻辑上了。

最近，Web 开发平台逐渐成熟，足以应用领域驱动设计来做 Web 开发，有很多积极的信号。比如，SOA，如果应用的好，就可以提供给我们一个非常有用的**解析**领域的方法。

同时，敏捷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、和业务伙伴亲密协作、应用持续集成和在强沟通环境下工作的重要性。

所以领域驱动设计在未来会越来越重要，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，新的技术和流程应该由它们是否支持团队专注于他们的领域来验证，而不是置领域于不理。领域驱动设计不依附于哪一个特定的平台，但是有些平台为创造业务逻辑提供了更多的好方法，**有些平台更加专注**。最近几年的发展显示后一种平台是一个有希望的方向，尤其在可怕 20 实际 90 年代后期。

Java 是这几年默认的选择，从表达角度看，它是一种典型的面向对象语言。对于 Distracting Clutter 来讲，基础语言是个不错的选择。它有垃圾收集功能，实践证明这一功能很有用。（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 POJO 仍然可以设计的让人理解。Java 5 语法的一些创新之处提高了**代码的**可读性。

但是在 J2EE 框架初次出来时，完全将那些基本的表达淹没在大量的框架代码之中。根据早期的**契约**（比如 EJB Home，为所有变量写的 Get/Set 前缀存取等）生产出可怕的对象。这一工具是如此笨重，使得开发团队不得不全力以赴才能让它工作。而且改变对象非常困难，一旦产生的大量代码和 XML 出现问题，人们往往束手无策。这样的平台很难开发出高效的领域模型。

另外就是使用初级的第一代工具勉强开发藉由 Http 和 Html（不是为此目的而设计的）**来完成**的 Web UI。在那个时候，创建和维护一个像样的 UI 是如此困难，以至很少去关注有着复杂内部功能的设计。有意思的是，恰在这时，对象技术出现了，很好地解决了复杂建模和设计的问题。

这一情况在.NET 平台上也是如此，有些事情处理的比较好一些，有些还更糟糕。

那是一个让人沮丧的时代，但是在过去的四年里，总算有了些转变。首先来看 Java，关于怎样有选择地使用框架社区里有了新的看法，很多新的优秀的框架（多数是开源的）也应运而生。比如 Hibernate 和 Spring 这些框架可以用一种更轻量级的方法处理 J2EE 试图做到的特定工作。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和**混合**的新 J2EE 项时也更加聪明。术语 POJO 就是在这时产生的。

结果是项目的技术贡献有了一个增大而且明显的减少，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，这样逻辑就可以基于 POJO 来写了。这不会自动产生领域驱动设计，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 有表达力很强的语法，在这一基础层面上它会是领域驱动设计的一个很好的语言（尽管我没有听说在那些应用程序中有哪些实际案例）。Rails 带给人们很多兴奋的地方，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 时一样简单。很快，这种能力就被大量使用在构建众多没有太多领域背景的 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，人们可以把这看成专注领域的机遇。如果 Ruby 使用是从这个方向起步的，那么我认为它会为领域驱动设计提供一个很棒的平台。（一些基础设施可能要被添加进来）

更多前沿的话题发生在域描述语言（DSL）领域，我一直深信 DSL 会是领域驱动设计发展的下一大步。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。

现在，我所能说的是大多数尝试使用领域驱动设计的人们是基于 Java 或者.NET 平台，也有少部分在 Smalltalk 上。在有直接效果的 Java 世界里，这是一个积极的信号

从你写完这本书，在领域驱动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，人们采用我在书中提到的原则然后用一些我所没有预料到的方法实践。比如，在挪威国家石油公司的 StatOil 项目中对**战略**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，并应用到在抉择是自己构建还是购买时到对已有软件的评估当中。

我们一直在调查在仍然用 Java 实现对象时，究竟能把这种域描述语言推动多远。

已经走了很远。在人们告诉我他们在做什么事情的时候，我一直都很感激。

请您给要学习领域驱动设计的人一些建议？

读我的书！☺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，让人们可以通过它进行学习。

要谨记一点的是领域驱动设计主要是由团队来操作，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) 事必躬亲。模型需要代码。

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进行领域驱动设计。画一张范围表，然后决定哪些应该进行领域驱动设计，哪些不用。不要担心边界之外的事情。

4) 不停地实验，期望能产生错误。模型是一个创造性的流程。

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，用具有多种不同的方法多种不同的输出的对象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，成功的团队应用这一系统可以组装有业务需求的复杂软件系统，并使系统在增大时仍然保持敏捷。

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，比如 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。

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！

本书讨论的主题包括：

- 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