

本书仅提供部分阅读，如需完整版，请联系QQ: 2404062482

提供各种IT类书籍pdf下载，如有需要，请QQ:2404062482

注：链接至淘宝，不喜者勿入！整理那么多资料也不容易，请多多见谅！非诚勿扰！

[点击购买完整版](#)

实现 领域驱动设计

IMPLEMENTING
DOMAIN-DRIVEN DESIGN

Vaughn Vernon 著

滕云

译

张逸

审

- 涵盖DDD各个方面
- 大量示例代码
- 案例研究贯穿全书
- 理论和实践紧密结合
- 程序员进阶佳作

DDD之父Eric Evans作序



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

“对于那些希望提升自己技能的软件开发来说，《实现领域驱动设计》将是一本绝佳的好书。”

——Randy Stafford, 自由架构师, Oracle Coherence产品部

“对于那些希望实际应用DDD的人来说,这是一本必读之作。”

——Udi Dahan, NServiceBus创始人

《实现领域驱动设计》采用一种自顶向下的方式向我们讲述了DDD的战略设计模式和战术编程工具,并使这两者之间自然地衔接起来。Vaughn Vernon向我们展示了如何将DDD实现应用于现代的软件架构,并且强调业务领域的重要性和价值,同时又不失向技术层面的折中考虑。

本书建立在Evic Evans的《领域驱动设计》之上,通过我们所熟知的示例领域向我们讲解实际的DDD实现技术。每种设计原则都有真实的Java例子作为支撑,并且这些例子对于C#程序员来说也适用。所有的Java例子都出自于同一个案例研究:一个大型的基于Scrum的SaaS多租户系统。

作者带领我们超越了“DDD-Lite”的局限,DDD-Lite即是将DDD单纯地作为一套技术工具集来使用。通过讲解限界上下文、上下文映射图和通用语言,作者全面地向我们展示了DDD的“战略设计模式”。通过书中所讲到的技术和例子,我们可以加快软件开发速度,提升软件质量,使我们的软件更具灵活性和可伸缩性,同时更加紧密地与软件的业务目标保持一致。

本书内容包括:

- 以正确的方式带领你进入DDD世界,从而快速地从获取价值。
- 将DDD用于不同的架构中,包括六边形架构、SOA、REST、CQRS、事件驱动架构和基于数据网格的架构。
- 适当地设计和实现实体——并且何时应该使用值对象而不是实体。
- 掌握DDD的领域事件技术。
- 通过ORM、NoSQL等实现资源库。

作者介绍: Vaughn Vernon是一个经验丰富的软件工匠,在软件设计、开发和架构方面拥有超过25年的从业经验。他提倡通过创新来简化软件的设计和实现。从20世纪80年代开始,他便开始使用面向对象语言进行编程;在90年代早期,他便在领域建模中应用了领域驱动设计,那时他使用的是Smalltalk语言。他在全球范围之内提供软件咨询和演讲,此外,他还在许多国家教授《实现领域驱动设计》的课程。

译者介绍: 滕云, ThoughtWorks软件工程师。当初抱着“非飞行器设计专业不读”的想法考入西北工业大学,却不料学起了机械和汽车。在尝尽了“从天上掉到地下”的滋味之后,又转行软件开发。目前主要从事银行、保险等领域的企业级软件开发,感兴趣的技术领域包括Java EE、Linux、领域驱动设计和构建自动化等。个人博客: <http://www.davenkin.me> (无知者云)。

上架建议: 计算机科学 > 软件工程

PEARSON

www.pearson.com



ISBN 978-7-121-22448-5



定价: 99.00元



责任编辑: 张春雨
封面设计: 李玲

实现领域驱动设计

IMPLEMENTING DOMAIN-DRIVEN DESIGN

Vaughn Vernon 著

滕云 译

张逸 审

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

领域驱动设计（DDD）是教我们如何做好软件的，同时也是教我们如何更好地使用面向对象技术的。它为我们提供了设计软件的全新视角，同时也给开发者留下了一大难题：如何将领域驱动设计付诸实践？Vaughn Vernon 的这本《实现领域驱动设计》为我们给出了全面的解答。

本书分别从战略和战术层面详尽地讨论了如何实现 DDD，其中包含了大量的最佳实践、设计准则和对一些问题的折中性讨论。全书共分为 14 章，在 DDD 战略部分，本书向我们讲解了领域、限界上下文、上下文映射图和架构等内容，战术部分包括实体、值对象、领域服务、领域事件、聚合和资源库等内容。一个虚构的案例研究贯穿全书，这对于实例讲解 DDD 实现来说非常有用。

本书在 DDD 的思想和实现之间建立起了一座桥梁，架构师和程序员均可阅读，同时也可以作为一本 DDD 参考书。

Authorized translation from the English language edition, entitled IMPLEMENTING DOMAIN-DRIVEN DESIGN, 1E, by VERNON, VAUGHN, published by Pearson Education, Inc., publishing as Addison-Wesley Professional, Copyright©2013 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

CHINESE SIMPLIFIED language edition published by PEARSON EDUCATION ASIA LTD., and PUBLISHING HOUSE OF ELECTRONICS INDUSTRY Copyright ©2014.

本书简体中文版专有出版权由 Pearson Education 培生教育出版亚洲有限公司授予电子工业出版社。未经出版者预先书面许可，不得以任何方式复制或抄袭本书的任何部分。

本书简体中文版贴有 Pearson Education 培生教育出版集团激光防伪标签，无标签者不得销售。

版权贸易合同登记号 图字：01-2014-0513

图书在版编目（CIP）数据

实现领域驱动设计/（美）弗农（Vernon,V.）著；滕云译. —北京：电子工业出版社，2014.3

书名原文：Implementing domain-driven design

ISBN 978-7-121-22448-5

I.①实… II.①弗… ②滕… III.①软件设计 IV.①TP311.5

中国版本图书馆 CIP 数据核字(2014)第 021688 号

策划编辑：张春雨

责任编辑：张春雨

印 刷：北京丰源印刷厂

装 订：河北省三河市路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：36.5 字数：817.6 千字

印 次：2014 年 3 月第 1 次印刷

定 价：99.00 元

凡购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

译者序

2013年秋天的某个周末，我在公司翻译本书。对面一个刚入职的大学毕业生向我问起一个关于软件建模的问题，我给她讲到了实体、领域服务和限界上下文等概念。语毕，我立即有两点感触：第一，我已经被DDD深深影响了；第二，即便是对于那些小的软件项目，DDD依然有用武之地。

通常的看法是，DDD更适合于大型的软件系统，这也使很多软件开发者对DDD敬而远之。其实不然，DDD首先作为一种思想而存在着，它无关系统大小，而是教我们如何做好软件。像实体、值对象这样的概念，我相信大家都在自己的项目中或多或少地用到，这些概念并不只存在于大型系统中，而是对任何系统皆然，因为它们体现了软件模型的本质所在。

曾经有一段时间，我希望在软件开发技能上有另一个层次的提高，于是我选择了DDD。在看完Eric Evans那本DDD开山之作之后，我了解了不少，也迷茫了许多。和其他人一样，我更希望看到一些实际的DDD例子。在网上搜索一番，我发现了本书；读完示例章节，我便爱不释手了。

在我看来，DDD绝非是什么标新立异之物，我更倾向于将其看成是软件发展的自然结果。就像在20世纪六七十年代出现了软件危机之后，面向对象成为了人们的救赎；瀑布式开发过程遇到瓶颈时，敏捷被搬上了舞台；而DDD则是对传统的以数据为中心的建模方式的反思结果。另外，我们还有“OO Done Right”的说法，即DDD是以正确的方式来使用面向对象的。

我很看重事物发展过程的“自然”面。你把一组随机数交给一个中学生排序，估计她/他也是能琢磨出一种排序算法的，此时的排序算法很可能就是我们在大学里才学到的直接排序法。我想很多人也都在程序中创建过一些服务类，并且将事务边界放在这些类的方法上，那么此时，我们所创建的便是DDD所称为的应用服务（Application Service）。于是我们看到，很多知识都可以通过我们自己的自然逻辑思考而获得。但是，我们和那些大牛的区别在于，他们有能力将这些知识概念化、理论化、抽象化和系统化。对于本书的作者Vaughn Vernon来说，也是如此。

如果说Eric Evans的《领域驱动设计》为我们提出了一些高屋建瓴的思想，那么本书便把这些思想落到了实处。而在本书中，作者也多次引用了《领域驱动设计》的相关章节。除此之外，本书还大量地引用了Gamma等人的《设计模式》和Martin Fowler的《企业应用架构模式》。所以，本书不只是关于软件建模的，而

是正如本书的赞美者之一Randy Stafford所说,它还可以用于更加宽阔的软件架构领域。

在收到电子工业出版社的张春雨编辑寄来的英文原著时,我便开始盘算:一本600页的书,每天翻译一页,我需要将近两年;每天翻译两页,也得十个月……一次次的组合之后,我已经厌倦了。于是不管三七二十一,白天工作,晚上翻译,平时工作,假期翻译。结果,四个月完工。

退却的时候也是有的,并且总会有这样或那样的借口:“今天不在状态”、“明天再翻译也不迟”,诸如此类,凡此种种。每当此时,我便想起在故乡的山坡上顶着烈日当午、弯腰锄地汗流滴土的父亲,于是再次打开已经合上的笔记本电脑……

感谢我的父母赋予我的精神动力,使我得以顺利地完成本书的翻译。

在翻译的过程中,我得到了我的同事,在DDD方面颇有建树的张逸¹的大力帮助。忘不了的,是那些在下班路上和他一起探讨聚合、CQRS的日子。

感谢郑晔²和格茸扎西在我还没有开始翻译本书时便主动提出帮我审校。另外,还要感谢澳大利亚的Mark Ryall和加拿大的Rick Harcus向我提供的有关英语语言文化上的帮助。



2013年10月 成都

¹ 张逸, ThoughtWorks咨询师, 著有《软件设计精要与模式》, 译有《恰如其分的软件架构》等书。

² 郑晔, ThoughtWorks咨询师, 2013年OracleDuke选择奖得主, 译有《Scala程序设计》和《Clojure编程乐趣》等书。

本书赞誉

“在《实现领域驱动设计》中，Vaughn不仅为DDD领域做出了卓越的贡献，还为更宽阔的企业应用架构领域写上了厚重的一笔。例如，在架构和资源库等核心章节中，Vaughn向我们展示了如何将DDD与各种架构风格和持久化技术融合在一起——包括SOA、REST、NoSQL和数据网格等——其中很多都是在Eric Evans那本DDD开山之作出版之后才出现的。另外，书中还讲到了对实体、值对象、聚合、领域服务、事件、工厂和资源库的实现，其中包括大量的例子。一言以蔽之，我认为这本书非常全面。对于那些希望提升自己技能的软件开发者来说，《实现领域驱动设计》将是一本绝佳的好书。”

——Randy Stafford，自由架构师，Oracle Coherence产品部

“领域驱动设计是一套非常强大的思想工具，它深远地影响着软件开发团队的效率。问题在于，许多开发者在应用这套思想工具时会不时地迷失方向，他们需要更实际的指导建议。在本书中，Vaughn将理论与实践联系在了一起。除了为我们讲解那些易被误解的DDD概念之外，Vaughn还讲到了一些新的概念，比如命令/查询职责分离（CQRS）和事件源等。对于那些希望实际应用DDD的人来说，这是一本必读之作。”

——Udi Dahan，NServiceBus创始人

“多年以来，DDD的开发者们都希望获得一些更实际的帮助。Vaughn缝合了理论和实践之间的间隙，向大家提供了一套完整的DDD实现参考。他向我们展示了如何在当前软件项目中使用DDD，并且向我们提出了大量的实际建议。”

——Alberto Brandolini，DDD导师（由Eric Evans和Domain Language, Inc颁发证书）

“《实现领域驱动设计》清晰地向我们展示了DDD的核心话题。本书的写作风格非常友好，就像一个值得信赖的导师在给你讲课一样。读完本书，你将能够应用DDD的各个重要概念。我在阅读本书的时候，在很多章节中都做上了着重标记……我会经常地参考并推荐本书。”

——Paul Rayner，首席咨询师，DDD导师（由Eric Evans和Domain Language, Inc颁发证书），DDD Denver创始人。

“在我所教的DDD课程中，很重要的一点便是如何将所有的DDD理论付诸实践。有了本书，DDD社区便有了可供参考的资料。《实现领域驱动设计》包含了创建DDD系统的方方面面，从具体的实现细节到高层的设计思想。这是一本了不起的DDD参考书，同时也是Eric Evans那本DDD开山之作的极佳伴侣。”

——Patrik Fredriksson，DDD导师（由Eric Evans和Domain Language, Inc颁发证书）

“如果你关心软件工艺——你也应该这么做——那么领域驱动设计便是非常重要的一项技能，而《实现领域驱动设计》则向我们提供了一条迈向成功的捷径。本书详尽地讨论了DDD的战略模式和战术模式，使开发者能够立即将理论付诸实践。今后的业务软件系统将从本书中受益匪浅。”

——Dave Muirhead, 首席咨询师, Blue River Systems 集团

“DDD既有理论，也有实践，这些都是每个开发者应该了解的，而本书则很好地弥补了理论与实践之间的差距。强烈推荐本书！”

——Rickard Oberg, Java开发者, Neo Technology公司

“在《实现领域驱动设计》中，Vaughn采用了自顶向下的方法，首先讲到了DDD的战略模式，比如限界上下文和上下文映射图，然后讲到了战术模式，比如实体、值对象和领域服务等。案例研究贯穿全书，要从中有所学，你需要在该案例研究上下足功夫。如果你这么做了，你便能看到将DDD应用于复杂领域的意义所在。书中包含了大量的旁注、图标和示例代码。如果你希望使用当下最常见的架构风格来创建一个DDD系统，那么Vaughn的这本《实现领域驱动设计》便是我所推荐的。”

——Dan Haywood, 《Domain-Driven Design with Naked Objects》作者

“本书采用了一种自顶向下的方式来讲解DDD，这种方式将DDD的战略模式和战术模式自然地衔接起来。在本书中，Vaughn强调了业务领域的价值，同时也给出了技术上的讨论。因此，DDD在软件开发中的角色也变得非常清晰。很多时候，我的团队，包括我本人，在应用DDD时都会遇到这样那样的麻烦。有了《实现领域驱动设计》的指导，我们得以克服种种挑战，进而将付出立即转化为业务价值。”

——Lev Gorodinski, 首席架构师, DrillSpot.com

序

在本书中, Vaughn Vernon以一种特有的方式向我们展示了领域驱动设计 (Domain-Driven Design, DDD) 的各个方面, 其中包括对新概念的解释、新的例子和原创的话题组织方式。我相信, 这种新颖的方式可以帮助大家掌握DDD的各种微妙之处, 特别是非常抽象的聚合和限界上下文。不同的人习惯用不同的方式来理解这些概念, 而在缺少多种解释的情况下, 想要了解这些微妙的抽象概念是非常困难的。

本书包含了在过去9年中出现在各种论文和讲稿中的对DDD的深层剖析, 而这些是在之前的书籍中没有的。本书将领域事件与实体和值对象一道看作是模型的基础部件。另外, 书中还讨论了“大泥球” (Big Ball of Mud) 架构和如何将其放置在上下文映射图 (Context Map) 中。Vaughn还向我们阐述了六边形架构 (Hexagonal Architecture), 这种新兴的架构与分层架构相比, 能够更好地描述我们要完成的事情。

我是在将近两年前第一次接触到本书内容的, 那时Vaughn已经开始撰写本书有一段时间了。在第一次DDD峰会上, 我们中的几个编写了关于DDD的若干话题, 比如有关DDD的新知识, 或者DDD社区所期待的一些针对性建议等。Vaughn负责写聚合部分, 这一写便是一个有关聚合的文章系列, 并且写得非常出色, 最后, 这个系列成为了本书中的一个章节。

在那次峰会上, 与会人员们一致认为: 一套更加具有规约性的DDD模式是大有裨益的。诚实地讲, 对于软件开发中的任何问题, 答案都是“得看情况”。然而, 这对于那些希望学到实际应用技术的人来说却没什么大用处。人们需要更加实际的指导。经验法则不见得一定要放之四海而皆准, 但在通常情况下, 他们可以工作得很好, 也应该被首先尝试。出于自身的果决性, 这些经验法则蕴含着解决问题的思想方法。Vaughn的这本《实现领域驱动设计》将各种明晰的建议很好地融合在一起, 同时又给出了一些折中性的讨论, 从而避免了将这些建议过于简单化。

一些额外的DDD模式,比如领域事件,已经成为了DDD的主流模式,人们也学会了如何应用这些模式,并尝试着在新架构和新技术中采用这些模式。在我的《领域驱动设计:软件核心复杂性应对之道》出版9年后,有太多关于DDD的新知识需要谈及,Vaughn的这本书则是最全面的阐述。

—Eric Evans

Domain Language, Inc.

前言

所有的计算都表明它不工作，唯一的做法是：使其工作。

——Pierre-Georges Latécoère
早期法国航空企业家

是的，我们将使其工作。然而，在软件开发过程中采用领域驱动设计却是困难的。即便是有能力的开发者，也很难找到实现领域驱动设计的正确方法。

起飞，着陆

在我小的时候，我的父亲学习过驾驶小型飞机。我们经常会全家出去飞行，有时会飞到另一个机场，在那里吃过午饭后再返回。当父亲时间有限而他依然想飞时，父亲便带上我一起在机场上空盘旋，起飞，着陆，再起飞，再着陆。

也会有些长途飞行，这时我们会带上一张由父亲先前绘制好的路线图。我们几个小孩便当起了领航员：将图上的标志对应着陆地上的地标，以确保我们没有跑偏航线。这是一件很有趣的事情，因为要识别远在地面上的物体是很有挑战性的。事实上，我敢肯定父亲根本不用我们领航便知道我们处于什么方位——他能看到仪表盘上的所有信息，并且他拥有仪表飞行执照。

空中的景观的确改变了我的视野。不时地，父亲和我会飞过我们乡下的房子。在几百英尺的高空中，我体会到了另一种“家”的概念，而这在之前是没有过的。当我们飞过自家的房子时，母亲和我的姐妹们便会跑到院子里向我们挥手。我知道那是她们，即便我看不清楚她们是谁。谈话肯定是不行的，连大声喊都不行，她们是听不见的。我还可以看到将我家和外面公路分开的护栏，平时我们会像走平衡木一样在护栏上面走来走去。从空中看，它们就像被细心编排过的小树枝一样。我们

家的院子很大，每每到了夏天，我都会开着割草机一排一排地修理院子里的草坪。而在空中时，我只能看到一片绿色，小草的叶子肯定是看不清楚的。

我喜欢在空中的时刻，直到现在我还不时回想起这些时刻，好像那个降落飞机的黄昏就发生在不久以前一样。虽然如此，在地面上的感觉依然是无法取代的，因为它给我一种脚踏实地的感觉。

着陆于领域驱动设计

一开始接触领域驱动设计 (DDD) 就像一个小孩之于飞行一样。天空中的景色是令人惊叹的，但有时我们却因为过于陌生而搞不明白它们到底是什么。要从甲地到乙地显得如此的遥远。然而，DDD的“成年人”们却总知道他们所处的方位，因为他们在很早之前便绘制好了路线图，并且能够完全按照仪表进行相应的操作。而还有很多人找不到“在地面上”的感觉，此时我们需要的是“稳定着陆”的能力，然后找到一张地图给我们指引方向。

Eric Evans的《领域驱动设计：软件核心复杂性应对之道》是一本经得住时间考验的经典之作。我坚定地相信，在接下来的几十年里，本书依然会是开发者的实用指导。和其他模式一样，该书为我们建立起了一种高屋建瓴式的宽阔视野。然而，对于如何实现DDD，我们可能将面对更多的挑战。通常来说，我们更渴望看到一些具体的例子。

我的目标之一便是帮助你来一个“软着陆”，保全飞机，然后沿着一条周知的线路带你回家。这将帮助你如何更好地去实现DDD，并且通过你所熟悉的工具和技术给出示例演示。当然，任何一个人都不可能一直呆在家里，所以我还会带领你到新的地带去冒险，这些地带你可能从来没有去过。冒险之路是险峻的，但是在正确的战术应对下，征服这些困难是可能的。在这条冒险之路上，你将学到另外的架构和模式来集成多个领域模型。你将接触到先前没有被研究过的集成方法，并且学到如何开发自治性服务。

我将向你提供一张对短途旅行和长途旅行均适用的地图，它可以帮助你更好地享受沿途风景，同时又不至于迷失途中。

对照地形, 绘制飞行图

在软件开发的过程中, 我们经常做的一件事便是将一种东西映射到另一种东西。我们将对象映射到数据库, 映射到用户界面, 或者映射到不同的应用层展现(包括作为消费方的其他系统或应用程序)。在所有这些映射中, 我们很自然地希望在Evans提出的高层模式和具体实现之间存在一种映射。

即便你已经接触过DDD, 你依然有很多可以获益的地方。有时, DDD首先被看作是一套技术工具集, 有人将此称为DDD-Lite。我们可能已经对实体、服务等DDD概念非常熟悉了, 并且大胆地尝试着设计聚合, 还通过资源库来管理持久化。这些模式是大家相对熟知的, 使用起来很容易, 我们甚至还使用了值对象。以上这些都属于战术设计模式范畴, 也即更加偏向技术层面。这些模式可以很好地帮我们解决软件问题。而同时, 对于战术性模式, 我们依然有许多需要学习的。我将战术模式映射到实现层面。

你曾了解过战术建模之外的东西吗? 你曾了解过被称为DDD“另一半”的战略设计模式吗? 如果你还没有使用过限界上下文和上下文映射图, 那么你很有可能也没有使用过通用语言。

如果说Evans在软件开发社区有一项发明, 那便是通用语言。通用语言是一种团队协作模式, 用于捕捉特定业务领域中的概念和术语。一个特定领域的软件模型通过不同的名词、形容词和动词来表达, 这些词汇是开发团队正式使用的, 而团队中应该包含一个或多个领域专家。然而, 将通用语言仅限定于一些词汇则是错误的。就像自然语言反映人们的思想一样, DDD的通用语言反映了领域专家对于软件系统的思维模型。通用语言和那些战略和战术性的建模模式同等重要, 在有些情况下甚至更具有持久性。

简单地讲, DDD-Lite将导致劣质的领域对象, 因为通用语言、限界上下文和上下文映射图的作用太大了, 你从其中获得的并不只是一套团队共用的语言。在限界上下文中用通用语言来表述一个领域模型可以增加业务价值, 并且使我们确信所开发软件的正确性。即使从技术的角度, 它也可以帮助我们创建更好的领域模型, 这样的模型行为丰满, 业务纯净, 并且可以减少犯错误的可能性。因此, 我将战略设计模式映射到了可理解的实际例子中。

本书对于DDD的映射可以帮助你同时体会到战略设计和战术设计的好处。通过一些具体的例子, 你将感受到这些DDD映射的业务价值和技术展现力。

如果我们对于DDD的所有实践都只是停留在“地面上”，那将是令人失望的。过度地拘泥于细节将使我们丧失在空中俯瞰的机会。所以，不要将自己局限在地面的细节上，要勇敢地飞翔在空中，居高临下。搭上战略设计的航班，去了解限界上下文和上下文映射图，你将获得更广阔的视野。当你从DDD的航班中获益时，我的目的也就达到了。

各章概要

以下是各章的主要内容以及你将如何从中获益。

第1章：DDD入门

本章向你介绍DDD的好处，并且教你如何尽可能多地去实现DDD。你将学到当你在应对复杂的软件系统时，DDD可以为你的项目和团队带来什么。同时，你将了解到通常的DDD替代方案以及这些方案为什么会导致问题。作为对DDD的基础讲解，本章将教你如何在项目中开始采用DDD，还有如何向你的领域专家和技术团队推销DDD。在DDD的武装下，你将学会如何迎接挑战，勇往直前。

本章将介绍关于一个公司及其团队的案例研究，虽然该公司是虚构的，但是他们所面临的DDD挑战却是真实存在的。该公司旨在开发一个新的多租户SaaS (Software as a Service, 软件即服务) 软件产品。不出所料，在使用DDD时，他们犯了一些常见的错误。不过还好，他们发现了这些错误，并解决了一些问题，因此项目还算没有偏离正轨。该团队需要开发一套基于Scrum的项目管理软件。该案例还会在本书的后续章节中连续讲到。每一种战略和战术模式都将教给这个团队。在这个过程中，团队有误入歧途的时候，但最终他们将向着成功的DDD实践昂首阔步。

第2章：领域、子域和限界上下文

领域、子域和核心域分别是什么？限界上下文是什么，我们为什么要使用它，并且如何使用？这些问题将在这个SaaS项目团队犯错误的时候给予解答。在他们的第一个DDD项目中，他们并不了解子域、限界上下文和通用语言这些概念。事实上，他们根本不知道什么是战略设计，只是采用了战术设计来解决一些技术问题。这样他们在开始设计领域模型的时候便遇到了不少问题。幸运的是，他们及时地意识到了这些问题，项目还有挽回的余地。

本章还讲到了如何使用限界上下文对模型进行分离，这是非常重要的；同时还讲到了一些模型分离不当的反例，并且给出了有效的实现建议。在采用了这些建议之后，该团队的成员们重新创建了两个不同的限界上下文。这种合理的模型分离带来的好处是引出了第三个限界上下文——核心域，这将是本书使用的主要例子。

对于那些苦于单单从技术层面应用DDD的人来说，本章应该能引起你的共鸣。如果你还是DDD战略设计的外行，那么本章将为你指明方向。

第3章：上下文映射图

上下文映射图帮助我们理解业务领域、模型间的边界，以及这些模型之间的集成方式。

上下文映射图绝对不只是绘制系统架构图这么简单，它处理的是不同限界上下文之间的关系，以及如何在不同的模型之间映射对象。对于在复杂的业务系统中使用好限界上下文，这是至关重要的。在第2章中，团队成员们在首次尝试限界上下文时碰到了问题。本章中，他们将学着如何利用上下文映射图来解决这些问题。这样的结果是产生了两个体面的限界上下文，这两个上下文将被另外一个负责核心域的团队所使用。

第4章：架构

我们都知道分层架构，但它是开发DDD软件的唯一方式吗，也或许还存在另外的方式？在本章中，我们将讲到：六边形架构（端口和适配器）、面向服务架构、REST、CQRS、事件驱动（管道和过滤器，长时处理过程，事件源）和数据网格，其中好几种架构都将被该团队成员所采用。

第5章：实体

在DDD的战术模式中，我们将首先讲到实体。团队成员们一开始过于强调实体的作用而忽视了值对象。受到数据库和持久化框架的影响，实体被该团队滥用了，此时他们开始讨论如何避免大范围地使用实体。

在本章中，你将看到很多优秀的实体设计例子。同时，本章还将讲到如何使用实体来表达通用语言，以及如何对实体进行测试、实现和持久化。

第6章: 值对象

早些时候, 团队成员们错过了采用值对象的好机会。他们过于注重为实体创建一些单一的属性, 这种方式是欠妥的, 更好的方式是将这些单一的属性聚合成一个不变的整体。本章将从不同的角度讲解如何设计值对象, 以及在什么时候采用值对象会优于实体。同时, 本章还包含了一些其他话题, 比如值对象在集成中的角色和对标准类型的建模等。然后, 本章讲到了如何设计以领域为中心的测试, 如何实现值对象。此外, 本章还讲到了在聚合中存储值对象时, 如何避免持久化机制所带来的不利影响。

第7章: 领域服务

本章将讲到, 在领域模型中, 什么时候应该将一个概念建模成粒度适中, 并且无状态的领域服务。你将学到何时应该使用领域服务而不是实体或值对象, 以及如何使用领域服务来处理业务逻辑和技术上的集成。团队成员们向我们展示了何时应该使用领域服务, 以及如何设计领域服务。

第8章: 领域事件

Eric Evans并没有在他的书中正式介绍领域事件, 领域事件是在他那本书出版之后才进入人们视野的。在本章中, 你将学到为什么领域事件如此有用, 以及使用领域事件的不同方法。领域事件甚至被用来辅助集成和自治性服务。在软件系统中, 我们经常使用一些技术层面的事件机制, 但本章将着重讲解领域事件与这些事件机制的区别。本章还将指导你如何设计并实现领域事件, 包括一些可行的方案和对这些方案的权衡选择。然后, 本章将讲到如何创建一个发布-订阅机制; 如何利用事件来集成整个企业软件中的各个订阅方; 如何创建和管理事件存储; 如何处理消息机制所面临的常见挑战等。

第9章: 模块

对于模型中的对象, 我们应该如何将他们组织在大小适中的容器中呢? 我们又如何保证不同容器中的对象之间只存在有限的耦合? 另外, 我们如何对这些容器进行命名以体现通用语言? 除了包和命名空间之外, 我们如何使用由语言和框架提供的现代模块化机制, 比如OSGi和Jigsaw? 在本章中, 你将看到SaaS团队成员是如何在不同的项目中使用模块的。

第10章: 聚合

在DDD的战术模式中, 聚合可能是最不容易理解的了。然而, 在遵循一定的经验法则的情况下, 我们是能够更简单、更快地实现聚合的。在本章中你将学到: 如何利用聚合在不同的小规模对象集群间创建一致性边界, 从而降低模型的复杂性。由于在细枝末节上花了太多精力, SaaS团队成员们在设计聚合时总是磕磕绊绊。我们将仔细研究该团队所面临的挑战, 并且分析错误的原因以及他们的应对策略。结果, 团队成员们对他们的核心域有了更深层次的理解。我们将看到, 在合理的事务处理和保证最终一致性 (Eventual Consistency) 的前提下, 该团队更正了他们所犯的错误, 并且在一个分布式环境中设计出了更具有伸缩性和更高效的模型。

第11章: 工厂

工厂已经在[Gamma et al.]中被大量地谈及了, 为什么还要讲呢? 本章并不打算重蹈覆辙, 而是将重点放在“工厂应该存在于何处”这个问题上。在本章中, 我们将讲到在DDD中实现工厂的技巧。团队成员在他们的核心域中创建的工厂可以简化客户端接口, 并且对模型的消费方起到保护作用, 从而避免了在多租户环境中引入灾难性的bug。

第12章: 资源库

资源库只是一个数据访问对象 (Data Access Object, DAO) 吗? 如果不是, 它们之间有什么区别呢? 我们为什么应该将资源库看成是对集合的模拟而非数据库呢? 在本章中, 我们将讲到如何利用ORM来实现资源库, 其中有两种ORM方案, 一种采用基于网格的分布式缓存, 另一种则采用NoSQL的键值对存储。团队成员们可以采用任何一种作为他们的持久化机制。

第13章: 集成限界上下文

到现在为止, 你已经了解了战略层次的上下文映射图和多种战术层次的模式。本章将讲到, 在DDD中, 我们如何通过上下文映射图来集成不同的模型。在团队对核心域和其他辅助性的限界上下文进行集成时, 我们将给出相应的建议和指导。

第14章: 应用程序

对于每一个核心域的通用语言,我们都设计了相应的模型,并且进行了足够的测试,模型工作正常。然而,客户应该如何使用我们的模型呢?他们应该使用DTO将数据在模型和用户界面之间传输吗?或者存在其他方案可以实现模型和展现组件间的数据传递?DDD中的应用服务和基础设施是如何工作的?对于这些问题,本章都将做出解答。

附录A: 聚合与事件源: A+ES

事件源是一种持久化聚合的重要技术,同时也是事件驱动架构的基础。事件源通过一系列的事件来表示聚合的所有状态。通过有序的事件重放,我们可以重新构建聚合的状态。当然,使用事件源的前提是:它能够简化对数据的持久化,并且能够捕捉到那些具有复杂行为属性的概念。

Java和开发工具

本书中的绝大多数例子都是使用Java语言编写的。我本来可以用C#的,但是我有意识地使用了Java。

首先,我认为Java社区正在抛弃好的软件设计和开发实践。现在,对于多数Java项目而言,要在其中找到一个好的领域对象恐怕是困难的。在我看来,Scrum和敏捷被人们看成了优良设计的替代品,而其中的产品待定项(Product Backlog)被看成了设计本身。多数敏捷人士并不会过多地去思考这些待定项是否会影响到业务模型。我得说明,Scrum的本意绝对不是要取代设计。不管有多少项目经理想将你捆绑在持续交付这条路上,我得说Scrum并不仅仅是要取悦于那些甘特图(Gantt chart)的追随者们。然而,太多的时候,情况的确是这样的。

我认为这是个很大的问题,所以我想鼓励Java社区重新回到领域建模中来,同时我会通过本书向大家说明,设计是可以使我们获益的。

此外,在.NET社区中已经有很好的DDD资源了,比如Jimmy Nilsson的《领域驱动设计与模式实战》[Nilsson]。由于Jimmy的出色工作和其他人对Alt.NET的倡导,.NET社区中正掀起一阵优秀设计的开发浪潮,这是Java社区需要注意的。

其次,我意识到C#.NET人员在理解Java代码上并不存在什么困难。由于很多DDD社区的人都在使用C#.NET,而本书的早期校对人员也都是C#程序员,但是我从来就没有收到他们的抱怨。因此,我便不用顾虑这些了。

在我写这本书时,业内正将目光从关系型数据库转向基于文档和键值对的存储方案。这是有原因的,Martin Fowler将这些存储方案称为“面向聚合存储”。这种命名是恰当的,它很好地描述了在DDD中使用NoSQL的好处。

但是,就我从事咨询的经验来看,很多开发者还是认定了关系型数据库和对象-关系映射。因此我想,NoSQL追随者们应该能够理解我在书中包含对象-关系映射的章节。然而,我的确得承认,这可能会招致那些认为存在对象-关系阻抗失配(Object-Relational Impedance)的人的鄙视。这无所谓,对此我表示接受,因为绝大多数人在他们的日常工作中都还得面对这种对象-关系阻抗失配。

当然,在第12章“资源库”中,我同样提供了基于文档的、键值对的和数据网格的存储方案。在多处地方,我都讨论到了NoSQL对聚合设计的影响。NoSQL趋势很有可能持续下去,那些对象-关系型的开发者们应该注意了。在本书中你将看到,我能够同时理解两个阵营的观点,并且对于双方的观点我都同意。这些都是技术趋势所导致的摩擦,而这对于积极的变革是有必要的。

致谢

非常感谢Addison-Wesley出版社给我机会出版本书。正如我之前在上课和演讲时所说,我将Addison-Wesley看成是一个懂得DDD价值的出版商。在本书的编辑过程中,Christopher Guzikowski和Chris Zahn (Dr. Z)给了我很大的支持。那天,Christopher Guzikowski打电话给我,说他希望我成为他的签约作家。我是不会忘记那一天的,我也不会忘记Christopher Guzikowski对我的鼓励。当然,是Dr. Z将本书的文本变成了可出版的状态。感谢我的出版编辑Elizabeth Ryan协调本书的出版细节。同时,我还要感谢我的技术编辑,Barbara Wood。

回到从前,Eric Evans花了他职业生涯里的5年时间完成了DDD的定义工作。没有他的努力,没有从SmallTalk和模式社区中迸发出来的智慧,许多开发者都只能依旧苦苦摸索,最终交付劣质的软件。可悲的是,这样的问题太常见了。正如Eric所说,那些劣质的软件以及开发团队无创新式的枯燥性几乎使他离开软件领域。因此,我们欠Eric一个大大的感谢。

Eric邀请我参加了2011年的DDD峰会。会毕,大家一致认为,DDD的领导层应该提供一套指导以帮助更多的开发者在DDD上取得成功。那时,我已经写本书有很长一段时间了,并且我们充分地体会到了开发者们所缺少的东西。我自告奋勇,决定写一个文章系列来介绍有关聚合的“经验法则”。之后,我将这个名为“高效聚合设计 (Effective Aggregate Design)”的文章系列当成了本书第10章的基础。当该系列文章在dddcommunity.org网站上发布时,我才知道,人们对这样的指导真是如饥似渴。感谢那些DDD领导层中审阅了这个文章系列的同仁们,并感谢他们为本书提供的建议和反馈。Eric Evans和Paul Rayner对该文章系列做了多次细致的审阅。另外,我还从Udi Dahan、Greg Young、Jimmy Nilsson、Niclas Hedhman和Rickard Oberg处获得了反馈。

特别感谢DDD社区的资深成员,Randy Stafford。几年前,我在丹佛举行DDD演讲,Randy也参加了。之后,他敦促我更多地参与到更大的DDD社区中去。一段时

间之后, Randy将我介绍给了Eric Evans, 由此我得以在DDD社区中与大家一起讨论问题。我的一些想法并不那么容易达到, 而Eric则说服我们将关注点放在一些具有近期价值的东西上。正是有了那次讨论, 才有了后来2011年的DDD峰会。虽然Randy由于忙于Oracle Coherence相关工作而无法参与本书的撰写, 我想以后我是可以和他合作来写点什么的。

非常感谢Rinat Abdullin、Stefan Tilkov和Wes Williams, 他们都为本书撰写了一些专题内容。要了解有关DDD的一切几乎是不可能的, 要在软件开发的各个领域都成为专家更不可能。这也是为什么我邀请他们撰写本书的第4章和附录A中的专题。感谢Stefan Tilkov在REST方面给我的帮助, 感谢Wes Williams在GemFire上的经验, 也感谢Rinat Abdullin与我们分享有关事件源和聚合实现方面的知识。

本书早期审阅者之一是Leo Gorodinsk。我第一次见到Leo是在丹佛。他根据自己的项目中采用DDD的经历向本书提出了很多宝贵的反馈。我也希望本书能够像他帮助我一样帮助他。我将Leo看成是DDD未来的一部分。

还有很多人都为本书的至少一章提出了反馈。其中, 那些更具批评性的反馈提供者有Gojko Adzic、Alberto Brandolini、Udi Dahan、Dan Haywood、Dave Muirhead和Stefan Tilkov。特别是, Dan Haywood和Gojko Adzic提供了很多早期的反馈, 其中主要是关于本书“最难读”的那些内容。我很高兴他们能够忍耐下去并且帮我做出更正。Alberto Brandolini在战略设计, 特别是上下文映射图方面的洞见使得我将关注点集中在这些概念的核心上。Dave Muirhead在面向对象设计、领域建模、对象持久化和内存数据网格方面——包括GemFire和Coherence——都拥有非常丰富的经验。本书中对对象持久化历史和实现细节的讲解便是受他的影响而完成的。除了在REST方面的贡献, Stefan Tilkov还在SOA、管道和过滤器方面向我提供了额外的支持。最后, Udi Dahan帮助我澄清了有关CQRS、长时处理过程(即Sagas)和NServiceBus方面的概念。其他为本书提供了有价值反馈的还有: Rinat Abdullin、Svein Arne Ackenhausen、Javier Ruiz Aranguren、William Doman、Chuck Durfee、Craig Hoff、Aeden Jameson、Jiwei Wu、Josh Maletz、Tom Marrs、Michael McCarthy、Rob Meidal、Jon Slenk、Aaron Stockton、Tom Stockton、Chris Sutton和Wes Williams。

Scorpio Steele为本书提供了非常棒的插图。Scorpio使IDDD团队的每一个人都成为了超级英雄。我的朋友Kerry Gilbert为本书做了非技术性的审阅。其他人的帮助使得本书在技术上是正确的, 而Kerry则在行文语法方面给了我很大的帮助。

我的父母为我的写作提供了灵感，在我这一生中，他们一直在支持着我。我的父亲——本书“牛仔的逻辑”幽默片段中的AJ——并不只是一个牛仔。不要搞错了。成为一个不错的牛仔已经非常好了，而我的父亲则在很多方面都展现出了他的才艺。除了喜欢飞行之外，我的父亲还是一个优秀的土木工程师、土地测量员，一个有天赋的谈判高手。另外，他还依旧喜欢着数学，并且研究星系。在我10岁的时候，我父亲就教我如何求解直角三角形。谢谢您，父亲，在我很小的时候就教给我这些。还要感谢我的母亲，她总是在我面临挑战时给予我鼓励和支持。

虽然本书是献给我的妻子Nicole和我们的儿子Tristan的，我还是想在这里再特别提及一下。他们使得我坚持写下去并最终完成本书。没有他们的支持和鼓励，这些都是不可能的。太感谢你们了，我亲爱的Nicole和Tristan。

关于作者

Vaughn Vernon是一个经验丰富的软件工匠，在软件设计、开发和架构方面拥有超过25年的从业经验。他提倡通过创新来简化软件的设计和实现。从20世纪80年代开始，他便开始使用面向对象语言进行编程；在20世纪90年代早期，他便在领域建模中应用了领域驱动设计，那时他使用的是Smalltalk语言。他在很多业务领域都有从业经验，包括航空、环境、地理、保险、医学和电信等领域。同时，Vaughn在技术上也取得了很大的成功，包括开发可重用的框架和类库等。他在全球范围之内提供软件咨询和演讲，此外，他还在许多国家教授《实现领域驱动设计》的课程。你可以通过www.VaughnVernon.co访问到他的最新研究成果。他的Twitter: @VaughnVernon。

如何使用本书

Eric Evans在他那本《领域驱动设计》中向我们展示了一整套模式语言。模式语言是相互关联的众多软件模式的一个集合,任何一种模式都会引用并依赖于其他一种或多种模式。这意味着什么呢?

这意味着当你在阅读本书时,某个章节中出现的有些DDD模式并不会在该章节中讲到,甚至在该章节之前都没有被谈及到。不要担心,继续往下读,被引用的模式将在本书的其他章节中做详细讲解。

在本书中,我将使用下表中的行文惯例:

表G.1 本书的行文惯例

出现文本	含义
模式名字 (#)	1.该模式是第一次出现在本书中, 或者 2.该模式已经在本章中出现了, 并且非常重要。
限界上下文 (2)	表示所引用的限界上下文在第2章中有详细的讲解。
限界上下文	表明限界上下文已经在本章中出现过了, 这里我并不会每次引用一个模式时都将其标为粗体并后加章号。
[REFERENCE]	表明对参考文献的引用
[Evans] 或 [Evans, Ref]	表明对于被引用的模式, 你可以参考Evans的著作以获得更多信息。[Evans]表示他那本经典的《领域驱动设计》, [Evans, Ref]并不表示引用Evans那本书本身, 而是另外的引用源, 该引用源也引用了Evans书中模式, 并且在此基础上有更新和扩展。
[Gamma et al.] 和 [Fowler, P of EAA]	[Gamma et al.]表示Gamma等人所著的那本经典的《设计模式》。 [Fowler, P of EAA]表示Martin Fowler的《企业应用架构模式》。 在本书中, 我会经常引用到这两本书, 虽然我还引用了其他的, 但这两本是主要的。.

在阅读的过程中,如果遇到对某个模式的引用,比如限界上下文,此时你通常可以在另外某个章节找到对该模式的讲解。

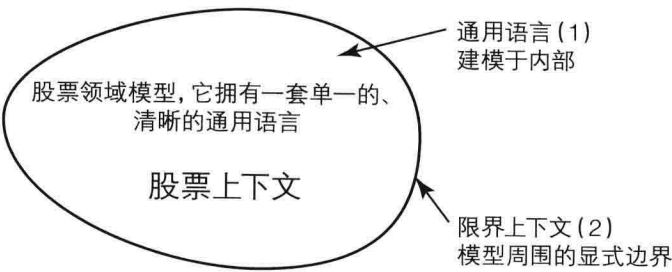
如果你已经读过[Evans],并且对其中的模式有一定的了解,那么本书可以帮助你进一步澄清DDD的概念,然后引导你对既有的模型进行改进。此时你可能并不需要一个总览式的介绍。但是,如果你还是DDD的新手,那么下面的内容将为你讲到不同的DDD模式是如何协同工作的,并且如何更好地使用本书,接着往下读吧。

DDD总览

早些时候,我讲到了DDD的通用语言(Ubiquitous Language, 1)。通用语言作用于某个限界上下文(Bounded Context, 2),它对于领域建模是非常重要的,你应该好好地熟悉一下。请记住,不管你是在战术上还是战略上设计软件模型,你都应该保证:在一个特定的限界上下文中只使用一套通用语言,并且保证它的清晰性和简洁性。

战略建模

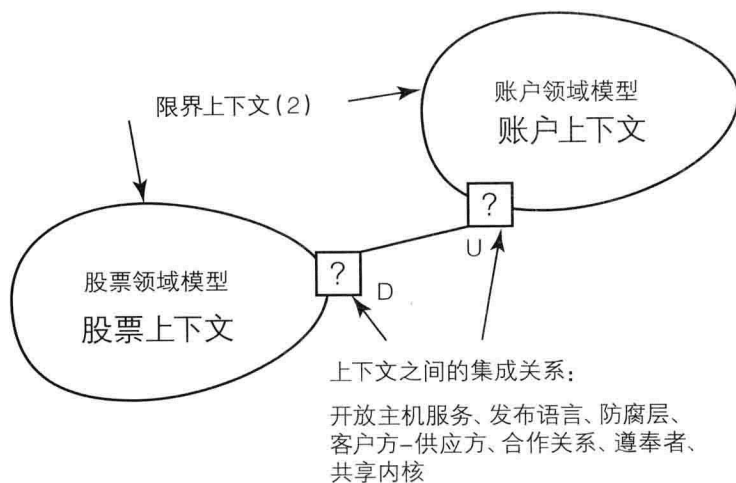
限界上下文是一种概念上的边界,领域模型便工作于其中。同时,限界上下文为通用语言提供了一套环境,项目成员便通过通用语言来表达软件模型,如图G.1所示。



图G.1 限界上下文和通用语言

在战略设计的过程中,你将发现上下文映射图(Context Map, 3)是非常有用的,如图G.2所示。你的团队将使用上下文映射图来理解项目的范围。

以上我们简要地了解了DDD的战略设计,这是我们必须好好理解的概念。



图G.2 上下文映射图展示限界上下文之间的关系

架构

有时,一个新的限界上下文或上下文映射图可能需要一种新的架构 (Architecture, 4)。你应该牢记: 通过战略和战术设计而成的领域模型应该是架构中立的。当然,在模型周围和模型之间则是存在架构的。一种能够支撑限界上下文的架构是六边形 (Hexagonal) 架构,它可以辅助其他架构风格,比如面向服务 (Service-Oriented) 架构、REST和事件驱动 (Event-Driven) 等。六边形架构如图G.3所示,从表面看,这种架构有点复杂,但是事实上却恰恰相反。

有时我们过于强调架构而忽略了DDD建模的重要性。架构固然是好的,但是架构并非一成不变。此时我们须要正确地处理优先级,将重点放在领域模型上,因为领域模型将产生更多的业务价值,并且更具有持久性。

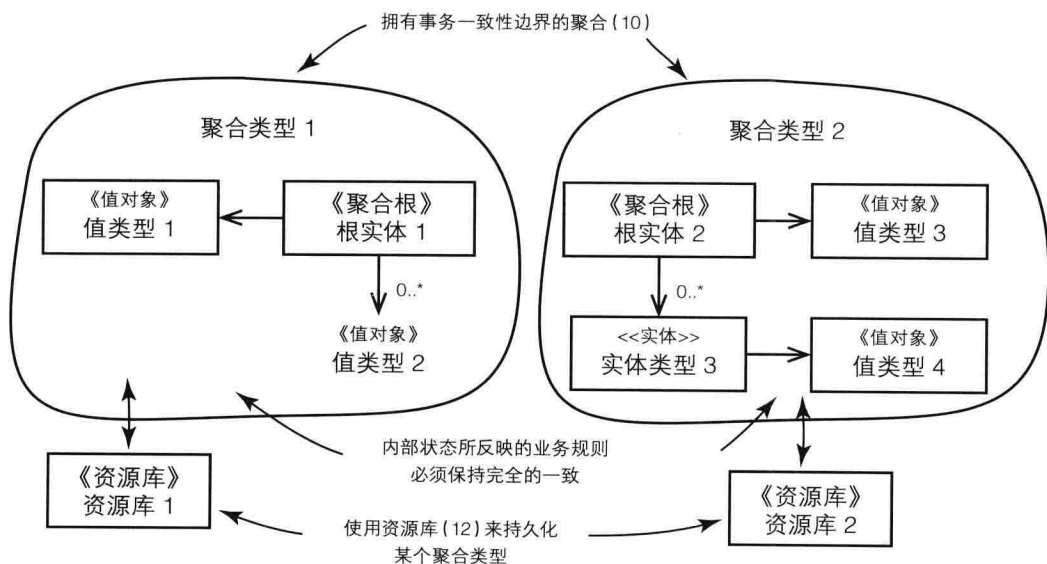
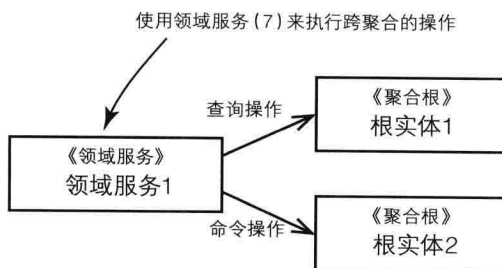


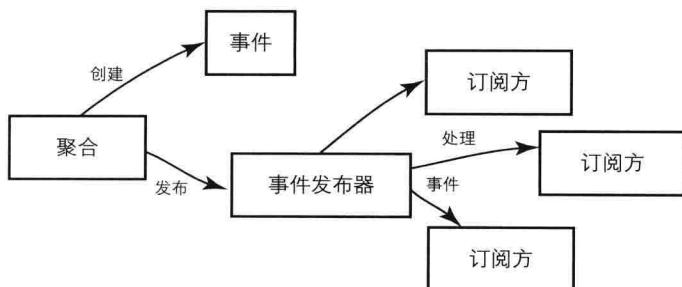
图 G.4 两个聚合类型，它们拥有各自的事务一致性边界

在领域模型中，有些业务操作并不能自然地放在实体或值对象上，此时我们可以使用无状态的领域服务 (Domain Service, 7)，如图G.5所示。



图G.5 领域服务执行特定于领域的操作，其中可能涉及到多个领域对象

领域事件 (Domain Event, 8) 表示领域模型中发生的重要事件。有多种方式可以对领域事件进行建模。在对聚合进行命令操作时，聚合本身将发布领域事件，如图G.6所示。



图G.6 领域事件可以由聚合发布

我们通常忽略了模块（Module, 9），但是正确地设计模块同样是重要的。简单来讲，我们可以将模块看成是Java中的包或C#中的命名空间。请记住，如果只是机械式地设计模块，而不是根据通用语言，那么我们将得不偿失。模块中包含的领域对象应该是内聚在一起的，如图G.7所示。

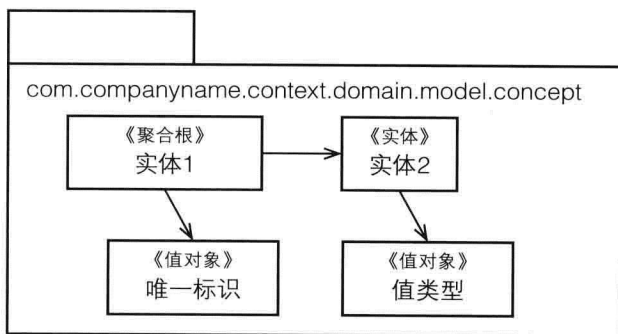


图 G.7 一个模块包含并组织内聚在一起的领域对象

好吧，现在来熟悉一下“牛仔的逻辑”幽默片段，以下便是一则：

牛仔的逻辑

AJ: “不要担心你的嘴巴包不下这块大肥肉，你的嘴巴比你想象的大多了。”

LB: “你说的是‘头脑’吧，J。你的头脑比想象的大多了才对。”



目录

序	xix
前言	xxi
致谢	xxxix
关于作者	xxxv
如何使用本书	xxxvii
 第1章 DDD入门	 1
我能DDD吗?	2
为什么我们需要DDD	5
如何DDD	17
使用DDD的业务价值	22
1.你获得了一个非常有用的领域模型	22
2.你的业务得到了更准确的定义和理解	23
3.领域专家可以为软件设计做出贡献	23
4.更好的用户体验	23
5.清晰的模型边界	24
6.更好的企业架构	24
7.敏捷、迭代式和持续建模	24
8.使用战略和战术新工具	24
实施DDD所面临的挑战	25
虚构的案例, 真实的实践	33
本章小结	36
 第2章 领域、子域和限界上下文	 37
总览	37
工作中的子域和限界上下文	38
将关注点放在核心域上	42
战略设计为什么重要	45
现实世界中领域和子域	48

理解限界上下文.....	53
限界上下文不仅仅只包含模型.....	57
限界上下文的大小.....	59
与技术组件保持一致.....	61
示例上下文.....	62
协作上下文.....	63
身份与访问上下文.....	69
敏捷项目管理上下文.....	71
本章小结.....	73
第3章 上下文映射图.....	75
上下文映射图为什么重要.....	75
绘制上下文映射图.....	77
产品和组织关系.....	79
映射3个示例限界上下文.....	82
本章小结.....	97
第4章 架构.....	99
采访一个成功的CIO.....	100
分层.....	104
依赖倒置原则.....	107
六边形架构(端口与适配器).....	110
面向服务架构.....	114
REST.....	117
REST作为一种架构风格.....	117
RESTful HTTP服务器的关键方面.....	118
RESTful HTTP客户端的关键方面.....	119
REST和DDD.....	120
为什么是REST?.....	121
命令和查询职责分离——CQRS.....	121
CQRS的各个方面.....	123
处理具有最终一致性的查询模型.....	128
事件驱动架构.....	129

管道和过滤器	131
长时处理过程 (也叫Saga)	134
事件源	140
数据网织和基于网织的分布式计算	143
数据复制	144
事件驱动网织和领域事件	145
持续查询	145
分布式处理	146
本章小结	148
第5章 实体	149
为什么使用实体	149
唯一标识	151
用户提供唯一标识	152
应用程序生成唯一标识	153
持久化机制生成唯一标识	156
另一个限界上下文提供唯一标识	160
标识生成时间	161
委派标识	163
标识稳定性	165
发现实体及其本质特征	167
揭开实体及其本质特征的神秘面纱	168
挖掘实体的关键行为	172
角色和职责	176
创建实体	181
验证	183
跟踪变化	192
本章小结	192
第6章 值对象	193
值对象的特征	194
度量或描述	195
不变性	195

概念整体.....	196
可替换性.....	199
值对象相等性.....	200
无副作用行为	201
最小化集成.....	204
用值对象表示标准类型.....	206
测试值对象.....	210
实现.....	214
持久化值对象	219
拒绝由数据建模泄漏带来的不利影响	220
ORM与单个值对象.....	221
多个值对象序列化到单个列中.....	224
使用数据库实体保存多个值对象.....	225
使用联合表保存多个值对象	229
ORM与枚举状态对象.....	230
本章小结.....	233
第7章 领域服务	235
什么是领域服务（首先，什么不是领域服务）	237
请确定你是否需要一个领域服务.....	238
建模领域服务.....	241
独立接口有必要吗.....	244
一个计算过程.....	246
转换服务.....	249
为领域服务创建一个迷你层	250
测试领域服务	250
本章小结.....	253
第8章 领域事件	255
何时/为什么使用领域事件	255
建模领域事件.....	258
创建具有聚合特征的领域事件	263
身份标识.....	264

从领域模型中发布领域事件	265
发送方	265
订阅方	269
向远程限界上下文发布领域事件	271
消息设施的一致性	271
自治服务和系统	272
容许时延	273
事件存储	274
转发存储事件的架构风格	279
以REST资源的方式发布事件通知	279
通过消息中间件发布事件通知	283
实现	284
发布NotificationLog	285
发布基于消息的事件通知	290
本章小结	297
第9章 模块	299
通过模块完成设计	299
模块的基本命名规范	302
领域模型的命名规范	302
敏捷项目管理上下文中的模块	305
其他层中的模块	308
先考虑模块, 再是限界上下文	309
本章小结	310
第10章 聚合	311
在Scrum核心领域中使用聚合	312
第一次尝试: 臃肿的聚合	313
第二次尝试: 多个聚合	314
原则: 在一致性边界之内建模真正的不变条件	317
原则: 设计小聚合	319
不要相信每一个用例	321
原则: 通过唯一标识引用其他聚合	322

通过标识引用使多个聚合协同工作	324
建模对象导航性	325
可伸缩性和分布式	326
原则：在边界之外使用最终一致性	327
谁的任务？	328
打破原则的理由	329
理由之一：方便用户界面	329
理由之二：缺乏技术机制	330
理由之三：全局事务	331
理由之四：查询性能	331
遵循原则	332
通过发现，深入理解	332
重新思考设计	332
估算聚合成本	334
常见用例场景	335
内存消耗	336
探索另外的设计	337
实现最终一致性	338
这是Scrum团队成员的任务吗？	339
决定的时候到了	341
实现	341
创建具有唯一标识的根实体	342
优先使用值对象	343
使用迪米特法则和“告诉而非询问”原则	344
乐观并发	346
避免依赖注入	348
本章小结	349

第11章 工厂 351

领域模型中的工厂	351
聚合根中的工厂方法	352
创建CalendarEntry实例	353
创建Discussion实例	357

领域服务中的工厂	358
本章小结	361
第12章 资源库	363
面向集合资源库	364
Hibernate实现	369
TopLink实现	377
面向持久化资源库	379
Coherence实现	381
MongoDB实现	386
额外的行为	391
管理事务	393
警告	397
类型层级	397
资源库 vs 数据访问对象 (DAO)	400
测试资源库	401
以内存实现进行测试	404
本章小结	407
第13章 集成限界上下文	409
集成基础知识	409
分布式系统之间存在根本性区别	411
跨系统边界交换信息	411
通过REST资源集成限界上下文	417
实现REST资源	418
使用防腐层实现REST客户端	421
通过消息集成限界上下文	428
从Scrum的产品负责人和团队成员处得到持续通知	428
你能处理这样的职责吗?	434
长时处理过程, 以及避免职责	439
长时处理过程的状态机和超时跟踪器	450
设计一个更复杂的长时处理过程	460

当消息机制或你的系统不可用时	464
本章小结	465
第14章 应用程序	467
用户界面	469
渲染领域对象	470
渲染数据传输对象	471
使用调停者发布聚合的内部状态	471
通过领域负载对象渲染聚合实例	472
聚合实例的状态展现	473
用例优化资源库查询	474
处理不同类型的客户端	474
渲染适配器以及处理用户编辑	475
应用服务	478
示例应用服务	478
解耦服务输出	485
组合多个限界上下文	487
基础设施	489
企业组件容器	490
本章小结	494
附录A 聚合与事件源: A+ES	495
应用服务内部	496
命令处理器	505
Lambda语法	508
并发控制	510
A+ES所带来的结构自由性	513
性能	513
实现事件存储	516
关系型持久化	520
BLOB持久化	522
专注的聚合	523
读模型投射	524

与聚合设计一道使用	527
增强事件.....	527
工具和模式	529
事件序列器	530
事件不变性.....	531
值对象.....	531
协议生成.....	534
单元测试和需求规范	535
事件源和函数式语言	536
参考文献	539

第1章

DDD入门

设计不只是感观，设计就是产品的工作方式。

——Steve Jobs

我们都致力于开发高质量的软件。通过测试，我们可以消除软件系统中大量的bug。然而，即便我们的软件中没有bug，也不能表示我们设计的软件模型本身就是好的。软件中存在少量的瑕疵是无可厚非的，而同时，我们是可以设计出能够准确表达业务意图的软件模型的。

领域驱动设计（DDD）作为一种软件开发方法，它可以帮助我们设计高质量的软件模型。在正确实现的情况下，我们通过DDD完成的设计恰恰就是软件的工作方式。本书便是帮助你如何正确实现DDD的。

你可能是个DDD新手；也可能做过一些DDD尝试而目前正苦苦地挣扎着；还有可能你已经成功地运用了DDD。不管如何，你都希望通过本书来提高自己的DDD技能，我相信你是可以的。以下是本章的学习路线图：

本章学习路线图

- 了解DDD可以为你的项目和团队带来哪些好处
- 如何确定你的项目是否适合采用DDD
- 了解DDD的常见替代方案和它们将导致问题的原因
- 学习DDD的基础
- 学习如何向你的管理层、领域专家和技术成员推销DDD
- 了解使用DDD时所面临的挑战
- 看看一个正在学习采用DDD的团队是如何工作的

那么，你应该期待从DDD中得到什么呢？首先，DDD不应该是一个仪式性的过程，更不应该成为你项目进度的阻碍。此时你可以采用敏捷开发方法，或者寻找另外的方法来帮你更深层次地了解自己的业务领域。我们的目标应该是创建一个可测试的、可伸缩的、组织良好的软件模型。

DDD同时提供了战略上的和战术 (Tactical) 上的建模工具来帮助我们设计高质量的软件模型。

我能DDD吗?

你是可以实施DDD的, 如果你:

- 有开发卓越软件的激情和毅力
- 渴望学习和进步
- 有能力理解软件模式, 并懂得如何应用这些模式
- 有发掘不同设计方法的能力和耐性
- 勇于改变现状
- 看重细节, 希望亲自试验
- 希望编写更好的代码

DDD不是没有学习曲线, 而且学习曲线有可能很陡。不用着急, 本书将尽可能地为你降低学习曲线, 我的目的就是挖掘你成功的潜能, 帮助你和你的团队实现DDD。

DDD首先并不是关于技术的, 而是关于讨论、聆听、理解、发现和业务价值的, 而这些都是为了将知识集中起来。如果你了解公司的业务, 那么你至少可以为DDD的通用语言 (Ubiquitous Language) 做出贡献。当然, 你可能需要学习更多的业务知识。由于你对业务概念的理解, 你已经开始走在DDD的康庄大道上了。

多年的软件开发经验能够帮助我更好地实现DDD吗? 可能会。然而, 你的开发经验并没有教给你向领域专家聆听和学习的能力。在实施DDD的过程中, 你最好将那些不怎么使用技术语言的人加进自己的团队, 此时你得仔细地聆听他们, 还应该尊重他们的观点, 并且相信他们比你了解得更多。

将领域专家引入到团队是大有好处的。

在实施DDD的过程中, 你最好将那些不怎么使用技术语言的人加进自己的团队。就像你会向他们学习一样, 他们也会向你学习。

可能你最希望看到的便是：领域专家同样得听你的。大家都是同一个团队的成员。领域专家不见得就知道所有的业务，他们也得学习。就像你会向他们学习一样，他们也会向你学习。你向领域专家提出的问题有可能暴露出他们不知道的地方。你将直接帮组团队更好地理解业务，甚至确定业务。

这样一来，团队中的所有成员都在学习和成长——是DDD使之成为了可能。

但是，我们还没有领域专家

领域专家并不是一个职位，他可以是精通业务的任何人。他们可能了解更多的关于业务领域的背景知识，他们可能是软件产品的设计者，甚至有可能是销售员。

如果你发现有人比你更加了解业务知识，找到他们，聆听他们，并向他们学习。

现在，我们已经开了一个好头。我并不是说技术不重要，而是说在本书中你得掌握领域建模中更高层次的概念。如果你的能力能够位于理解《Head First设计模式》[Freeman et al]和《设计模式》[Gamma et al]之间，或者你还学了一些更高级的模式，那么你已经具备很好的DDD基础了。在本书中，我将尽量顾及到各个层次的学习者。

什么是领域模型？

领域模型是关于某个特定业务领域的软件模型。通常，领域模型通过对象模型来实现，这些对象同时包含了数据和行为，并且表达了准确的业务含义。

不同角色的人都可以从DDD中获益，看看自己属于以下角色的哪一种：

- 新手，初级开发者：“我还年轻，有很多点子，对写代码充满了热情。但是我所在的那个项目简直让人崩溃，我才不想一毕业就被那些重复性的工作纠缠来缠去。这个项目的架构为什么如此复杂？这到底是怎么回事？我修改了一点代码，却破坏了更多的代码。有人知道这本来应该是什么样子吗？现在，我还得添加一些复杂的新特性。我在遗留代码之上添加了一个适配器来屏蔽那些难看的遗留代码，但是没有用。我相信除了整天写代码和调试外，还有更好的办法。于是有人向我介绍DDD，听说DDD是领域模型中的‘四人帮（Gang of Four）’，不错不错。”
- 中级开发者：“在过去的几个月中，我加入了一个新的项目，这次轮到我来做出改变了。那时，当我和高级开发人员在一起工作时，我发现我缺少对事物的洞察力。有时团队非常涣散，但是我又不知道其中的原因，我决定改变团

队成员们的做事方式。我需要一种能够助我成功的软件开发技术。一个高级架构师向我推荐DDD, 我打算了解了解。

听起来你已经是个高级开发者了, 继续往下读。你超前思考的态度自然会得到回报的。

- 高级开发者, 架构师: “我曾在多个项目中都使用过DDD, 不过目前所在项目还未使用。我喜欢DDD战术模式的威力, 但是我还打算应用更多的DDD模式, 比如战略设计等。在阅读[Evans]时, 我发现通用语言的功能非常强大。我已经与团队成员和管理层讨论过采用DDD的事情了, 但其中有些中高级开发者对DDD在项目中的前景并不看好, 而管理层也不是那么热衷于DDD。我是最近才加入公司的, 虽然我是团队带头人, 但是整个团队似乎并不愿意被一些新奇玩意儿打断了开发进度。不管如何, 我是不会放弃的。虽然其他开发者对DDD没有信心, 但是我是能做到的。我决定将领域专家引入到团队中来, 使他们跟技术人员一起工作。”

这就是一个领导应该做的。本书包含了很多关于战略设计的例子。

- 领域专家: “我已经在IT部门帮他们解决业务问题好长一段时间了, 我希望开发者们能够更好地理解他们所做事情。他们总是认为我们业务人员是愚蠢的。但是他们不知道的是, 如果不是我们, 他们早就丢了工作。开发者总会以一些奇怪的方式来讨论我们的软件, 如果我说这是A, 他们却说这是B, 好像我们需要有本词典才能交流一样。如果我们试图纠正他们的错误叫法, 他们就不愿意合作了。我们在这上面浪费了太多的时间。软件为什么不能像真正的业务专家所想象的那样工作呢?”

你说对了。软件开发的最大问题之一便是业务人员和技术人员需要某种翻译才能交流。本章将对此做出讨论。你将看到, DDD将业务人员和技术人员放在同一个层面上。惊讶吧, 你已经使一些开发者向你靠近了, 多帮帮他们。

- 项目经理: “我们要交付软件, 但结果并不总是让人舒心, 有时开发时间拖得太长了。开发者在谈论到领域时总是各执一词, 莫衷一是。我不确定我们是否需要另一种银弹般的技术或者方法。之前我们不是没有尝试过, 但是每次都失败了, 结果还是得回到原来的位置。我总是说, 我们应该放弃幻想, 准备战斗, 但是团队成员并不这么认为。他们工作非常努力, 我总觉得欠他们些什么, 于是听听他们的意见吧。他们都是聪明之人, 并且都希望做出些改变。对于我来说, 如果我上面的管理层同意, 我是完全允许开发者们花

时间学习的。团队成员们希望有个集中化的业务知识体系，我想我可以以此说服我的老板。这样一来，我自己的工作也将变得简单，我还可以促进团队和业务专家之间的合作与互信。”

多好的项目经理啊！

不管你是谁，有一点是重要的：要在DDD之路上取得成功，你肯定得学习，并且大量地学习。学习不是什么问题，你是聪明的，并且一直都在学习。然而，我们都面临这样一种困境：

就个人来讲，我时刻都在准备着学习，但是我不喜欢被人教。
——Winston Churchill

在本书中，我会尽量将知识传递变成一件愉悦之事，并且保证你在DDD上有所收获。

你可能又有问题了：“为什么我们需要DDD呢？”

为什么我们需要DDD

事实上，在前面我已经提到了一些应该采用DDD的原因。冒着有悖DRY原则（Don't repeat yourself，不要做重复的事情）的风险，我重新说说我们需要采用DDD的原因。

- 使领域专家和开发者在一起工作，这样开发出来的软件能够准确地传达业务规则。当然，对于领域专家和开发者来说，这并不表示单单地包容对方，而是将他们组成一个密切协作的团队。
- “准确传达业务规则”的意思是说，此时的软件就像如果领域专家是编码人员时所开发出来的一样。
- 可以帮助业务人员自我提高。没有任何一个领域专家或者管理者敢说他对业务已经了如指掌了，业务知识也需要一个长期的学习过程。在DDD中，每个人都在学习，同时每个人又是知识的贡献者。
- 关键在于对知识的集中，因为这样可以确保软件知识并不只是掌握在少数人手中。

- 在领域专家、开发者和软件本身之间不存在“翻译”，意思是当大家都使用相同的语言进行交流时，每人都能听懂他人所说。
- 设计就是代码，代码就是设计。设计是关于软件如何工作的，最好的编码设计来自于多次试验，这得益于敏捷的发现过程。
- DDD同时提供了战略设计和战术设计两种方式。战略设计帮助我们理解哪些投入是最重要的；哪些既有软件资产是可以重新拿来使用的；哪些人应该被加到团队中？战术设计则帮助我们创建DDD模型中各个部件。

就像其他高回报率的投入一样，DDD需要我们在时间和精力上都有所投入。但是，考虑到我们在开发软件的过程中经常遇到的各种问题和挑战，这样的投入是值得的。

难以捉摸的业务价值

开发能够传递真正业务价值的软件和开发普通的软件是不同的。具有真正业务价值的软件能够很好地符合业务战略，并且可以将竞争优势融合到解决方案中。此时的软件并不是关于技术的，而是关于业务的。

业务知识从来就没有被集中过。开发团队必须在多方之间权衡各种需求，并确定其中的优先级。同时，团队成员的技能也是良莠不齐的。在获得所有的信息之后，团队所面临的问题在于：如何确定某种需求确实能够传递真正的业务价值？还有，我们如何去发现并暴露出这些业务价值，如何安排它们之间的优先级，并且如何实现它们？

在开发过程中，最大的鸿沟之一便存在于领域专家和开发者之间。通常来说，领域专家将关注点放在交付业务价值上，而开发者则将注意力放在技术实现上。当然，并不是说开发者的动机是错误的，而是说开发者的眼光被自然而然地吸引到了实现层面上。即便让领域专家和开发者一同工作，他们之间的协作也只是表面的，这时在所开发的软件中便产生了一种映射：将业务人员所想的映射到开发者所理解的。这样一来，软件便不能完全反映出领域专家的思维模型。随着时间的推移，这种鸿沟将增加软件的开发成本。而随着开发者转到其他项目或者离职，本应该驻留在软件中的领域知识也就丢失了。

另一个问题发生在当多个领域专家之间存在分歧的时候。这是很有可能发生的，因为每个专家只是熟悉某个或者某些特定的领域。另外，在某个领域里找不到真正的专家也是可能的，此时，有人可能对该领域有所了解，但是他更像一个业务分析员。这些问题将导致相互矛盾的软件模型。

更糟的是，软件的技术实现可能错误地改变软件的业务规则。比如，ERP软件通常需要修改业务操作以满足某个特定用户的需求，因此ERP的成本不能单以使用许可和维护费用来计算，对业务规则的修改所产生的成本远远大于前两者。另外一个相似的例子是当开发团队将业务需求翻译成软件功能的时候。这对于业务、用户和合作方来说都是一笔很大的成本。还有，技术上的翻译和解释是没有必要的，并且在使用适当开发方式的情况下是可以避免的。解决方案才是主要的投入。

DDD如何帮助我们

DDD作为一种软件开发方法，它主要关注以下三个方面：

1. DDD将领域专家和开发人员聚集到一起，这样所开发的软件能够反映出领域专家的思维模型。这并不意味着我们将精力都花在了对“真实世界”的建模上，而是交付最具业务价值的软件。有时在实用和理想之间存在冲突，根据它们的互异程度，在DDD中我们将选择实用性。

领域专家将和开发人员一起创建一套适用于领域建模的通用语言。通用语言必须在全队范围之内达成一致；所有成员都使用通用语言进行交流，通用语言也是对软件模型的直接反映。请注意，虽然团队中同时包含领域专家和开发人员，但并不是“我们”和“他们”的关系，团队中只有“我们”的概念。

通用语言也有助于促使原本存在分歧的领域专家们达成一致意见。此外，通过将领域知识传达给所有的团队成员，包括开发人员，整个团队也将更具凝聚力。我们甚至可以认为，这是每个公司都应该有的对于知识型工作者的起码训练。

2. DDD关注业务战略。虽然说战略 (Strategic) 设计自然地包含了战术设计，但是战略设计关注更多的则是业务的战略方向。它帮助我们定义不同团队之间的组织关系，并在这些关系有可能导致项目失败的时候提供早期预警。DDD的战略设计用于清楚地界分不同的系统和业务关注点，这样可以保护每个业务层面的服务。更进一步，这将指引我们如何实现面向服务架构 (service-oriented architecture) 或者业务驱动 (business-driven architecture) 架构。
3. 通过使用战术设计建模工具，DDD满足了软件真正的技术需求。这些战术设计工具使开发人员能够按照领域专家的思维模型开发软件。同时，所开发出来的软件是可测试的，能够尽量避免错误，能执行服务层面协议 (Service-Level Agreement, SLA)，具有很好的伸缩性，并且允许分布式计算。DDD的

最佳实践同时包含了高层的架构性实践和底层设计实践,关注业务规则和数据不变性,并且可以对业务规则起到保护作用。

通过这种方式开发软件,你和你的团队将能成功地交付真正的业务价值。

处理领域复杂性

在使用DDD时,我们首先希望将它应用在最重要的业务场景下。对于那些可以轻易替换的软件来说,你是不会有所投入的。相反,值得你投入的是那些重要的、复杂的东西,因为这些东西将为你带来可观的回报。正因如此,我们将这样的模型命名为核心域(Core Domain, 2),而那些相对次要的称为支撑子域(Supporting Subdomain, 2)。那么现在,我们需要搞明白的是,“复杂”到底是什么意思?

DDD的作用是简化,而不是复杂化

在使用DDD时,我们应该采用最简单的方式对复杂领域进行建模,而不是使问题变得更加复杂。

不同的业务领域对于复杂的定义是不一样的。另外,不同的公司所面临的挑战不一样;成熟度不一样;软件开发能力也不一样。因此,与其去定义什么是复杂的,还不如定义什么是重要的。这时,你的团队和管理层应该做出决定:你们开发的软件系统是否值得做出DDD投入。

DDD计分卡: 使用表1.1来决定你的项目是否值得做出DDD投入。如果你的项目情况在某行的描述范围之内,那么请在右边的列中记上相应的分数,最后将这些分数相加得到总分。如果得分为7分或者以上,那么,你应该考虑使用DDD了。

表1.1 DDD计分卡

你的项目是否得到7分或者7分以上?			
如果你的项目.....	得分	备注	你的打分
<p>如果你的软件完全以数据为中心,所有操作都通过对数据库的CRUD完成,那么你并不需要DDD。此时你的团队只需要一个漂亮的数据库表编辑器。</p> <p>换言之,你可以指望用户对你的数据进行直接操作,包括更新和删除数据。你并不需要提供用户界面。如果你甚至可以用一个简单的数据库开发工具来完成开发,那么,你完全没有必要在DDD上浪费时间和金钱。</p>	0	<p>这似乎是一个傻瓜化的问题,但是要分清简单和复杂的区别却不是那么容易的。并不是说只要不是纯粹的CRUD软件,便可以采用DDD。因此我们需要采用另外的方法来判别简单和复杂.....</p>	
<p>如果你的系统只有25到30个业务操作,这应该是相当简单的。这意味着你的程序中不会多于30个用户故事 (user story) 或用例流(use case flow),并且每个用例流仅包含少量的业务逻辑。如果你可以使用Ruby on Rails或者Groovy和Grails来快速地开发出这样的系统,并且你没有感觉到由复杂性和业务变化所带来的痛苦,那么你是不需要使用DDD的。</p>	1	<p>澄清一下,我是说25到30个业务方法,而不是说25到30个拥有多个方法的服务接口,后者可能是复杂的。</p>	
<p>当你的系统中有30到40个用户故事或者用例流时,此时软件的复杂性便暴露出来了,你可以考虑采用DDD了。</p>	2	<p>通常情况下,复杂性并不能被及时发现。我们开发者很容易低估软件的复杂性。我们希望使用Ruby on Rails来开发软件并不代表我们就必须使用Ruby on Rails。而长远看来,这是不利的。</p>	

continues

你的项目是否得到7分或者7分以上?	如果你的项目……	得分	备注	你的打分
	即便我们的软件目前并不复杂,但是之后呢? 在真正的用户开始使用软件之前,我们是无法预测软件的复杂性的,但是在右边的“备注”栏中有一项可以帮助我们应对这种情况。	3	这时我们有必要和领域专家一起探讨那些复杂的用例。 如果领域专家…… 1. 已经要求加入更复杂的功能。这表明软件已经开始变得复杂,此时单纯的CRUD是不能满足需求的。 2. 认为既有的功能没什么可以探讨的。此时我们的软件可能并不那么复杂。	
	请注意,如果有暗示说明系统已经足够复杂,这往往意味着我们的系统实际上比目前更加复杂,采用DDD吧。			
	软件的功能在接下来的几年里将不断变化,而你并不能预期这些变化只是些简单的改变。	4	DDD可以帮助你管理软件的复杂性,随着时间的推移,你可以对软件模型进行重构。	
	你不了解软件所要处理的领域(2)。你的团队中也没有人曾经从事过该领域的开发工作。此时,软件很可能是复杂的,因此你们应该讨论复杂等级。	5	你需要和领域专家一起工作了。你肯定也在前面的计分行中打了分,采用DDD吧。	

通过对以上DDD计分卡打分,我们可以得出以下结论:

当我们在复杂性问题上犯错时,我们很难轻易地扭转颓势。

这意味着我们应该在项目计划早期便对简单性和复杂性做出判断,这将为我们节约很多时间和开销,并免除很多麻烦。

一旦我们做出了重要的架构决策,并且已经在该架构下进行了深入地开发,通常我们也被绑定在这个架构下了,所以在决定时一定要慎重。

如果你对以上几点产生了共鸣,表明你已经在认真地思考问题了。

贫血症和失忆症

贫血症严重危害着人类健康,并且伴随有危险的副作用。当贫血领域对象(Anemic Domain Object) [Fowler, Anemic]被首次提出来时,它并不是一个博得赞美的词汇,它描述的是一个缺少内在行为的领域对象。奇怪的是,人们对于贫血领域对象的态度褒贬不一。问题在于,多数开发者认为这样的领域对象是正常的,他们并没有意识到这是一个严重的问题。

你是否想知道你所建模型的健康状况呢? 如果你突然患上了技术上的“忧郁症”, 这里你可以做个自我检查。你可能心情愉悦,也可能无比恐惧。通过表1.2中的步骤开始检查吧。

表1.2 领域对象病历

Yes / No	
你的领域对象中是不是主要是些公有的getter和setter方法, 并且几乎没有业务逻辑, 或者甚至完全没有业务逻辑——对象嘛, 主要就是用来容纳属性值的?	
软件组件经常使用的领域对象是否包含了系统主要的业务逻辑, 并且多数情况下你需要调用那些getter和setter? 你可能会将这样的客户代码称为服务层 (Service Layer) 或者应用层 (Application Layer) (4, 14) 代码。又或者, 如果这描述的是你的用户界面, 请回答 “Yes”, 然后好好反省一下, 告诫自己一定不要再这么做了。	
提示: 正确的答案是: 要么两项均为 “Yes”, 要么均为 “No”。	

如果你对以上两个问题的回答都是 “No”, 表明你的领域对象是健康的。

如果都是 “Yes”, 表明你的领域对象已经病得不轻了, 这便是贫血对象。好消息是, 你是可以获得帮助的, 继续往下读吧。

如果你对其中一个回答“Yes”，而另一个回答“No”，你可能是在自欺或者患上了由贫血症导致的神经系统紊乱，此时你应该怎么办呢？回到第一个问题重新来一遍，不要着急，要确保对两个问题都回答“Yes”。

正如[Fowler, Anemic]所说，贫血领域对象是不好的，因为你花了很大的成本来开发领域对象，但是从中却获益甚少。比如，由于存在对象-关系阻抗失配 (Object-Relational Impedance)，开发者需要将很多时间花在对象和数据存储之间的映射上。这样的代价太大，而收益太小。我得说，你所说的领域对象根本就不是领域对象，而只是将关系型数据库中的模型映射到了对象上而已。这样的领域对象更像是活动记录 (Active Record) [Fowler, P of EAA]，此时你可以对架构做个简化，然后使用事务脚本 (Transaction Script) [Fowler, P of EAA]进行开发。

为什么会有贫血领域对象

如果说贫血领域对象是由设计不当造成的，为什么还有如此多的人认为他们的领域对象是健康的呢？其中一个原因是：贫血领域对象反映了一种自然的过程式的编程风格，但我并不认为这是首要原因。软件业中有很多开发者都是学着示例代码做开发的，这并不是什么坏事，只要示例代码本身是好的。然而，通常情况是，示例代码只是用尽可能简单的方式来展示某个特定的概念或者API特性，而并不强调要遵循多好的设计原则。一些极度简化的示例代码总是包含了大量的getter和setter，于是这些getter和setter随着示例代码每天被程序员们原封不动地来回复制。

还有历史的影响。Microsoft的Visual Basic对我们现在的软件开发产生了很大的影响。我并不是说Visual Basic是门不好的语言和集成开发环境 (IDE)，因为它的确是种高效的开发方式，并且在某些方面对软件开发产生过正面的影响。当然，有些人可能会拒绝Visual Basic的直接影响，但是最终它却间接地影响着每一个程序员。请注意表1.3中的时间线。

表1.3 从富有行为的对象到贫血对象的时间线

1980s	1991	1992–1995	1996	1997	1998–
由于Smalltalk和C++，对象开始产生影响	Visual Basic 属性和属性 列表	可视化工具 和IDE遍地 开花	Java JDK 1.0 发布	JavaBean 规范	Java和.NET平台 均出现了大量的 通过反射机制 处理对象属性的 工具

这里，我想谈及的是对象属性和属性列表带来的影响。对象属性和属性列表都得益于getter和setter的支持，而Visual Basic的窗体设计器将getter和setter

变得过于流行了。你需要做的只是将自定义控件拖到窗体上,然后编辑控件的属性列表,大功告成,一个功能完备的窗体程序开发完毕。如果直接采用C语言的Windows API来开发相同的窗体,可能需要几天时间,而采用Visual Basic只是几分钟的事情。

那这和贫血领域对象有什么关系呢? JavaBean标准最早是用来辅助Java的可视化设计工具的,旨在将Microsoft的Active X开发方式带到Java平台。Java此举希望开创一个第三方自定义控件市场,就像Visual Basic一样。此后不久,几乎所有的框架和类库都涌入到了JavaBean潮流中,其中包括Java本身的SDK/JDK和第三方类库,比如Hibernate。在.NET平台推出之后,这样的趋势还在继续。

有趣的是,在早期的Hibernate版本中,所有需要持久化的领域对象都必须暴露公有的getter和setter,不管是对于简单类型的属性,还是对复杂类型皆如此。这意味着,即便你希望将自己的POJO (Plain Old Java Object) 设计成富含行为的对象,你都必须将对象的内部暴露给Hibernate以保存或重建对象。诚然,你可以隐藏公有的JavaBean接口,但是多数开发者都懒得这样做,或者甚至都不知道为什么应该这么做。

我应该考虑在DDD中使用对象-关系映射 (Object-Relational Mapping, ORM) 吗?

前面主要是从历史的角度对Hibernate进行了批评。现在, Hibernate已经不需要对象暴露getter和setter了,它甚至可以对对象属性进行直接操作。我将在后面的章节中讲到在使用Hibernate或其他持久化机制时如何避免贫血对象。

此外,多数的Web框架依然只支持JavaBean规范。如果你想将一个Java对象显示在网页上,该Java对象最好是支持JavaBean规范的。如果你想将HTML表单中的数据传到一个Java对象中,该Java对象也最好是支持JavaBean规范的。

市场上的几乎每种框架都要求对象暴露公有属性。这样一来,多数开发者只能被动地接受那些贫血对象。于是我们便到了“到处都是贫血对象”的地步。

看看贫血对象都对你的模型做了些什么

好吧,我们同意这已经是烦人的即成事实。但是这些无处不在的贫血对象和失忆症又有什么关系呢? 当你在阅读一个贫血领域对象的示例代码时,比如应用服务(4, 14)中的事务脚本,你通常会看到类似如下的代码片段:

```
@Transactional
public void saveCustomer(
    String customerId,
    String customerFirstName, String customerLastName,
```

```
String streetAddress1, String streetAddress2,
String city, String stateOrProvince,
String postalCode, String country,
String homePhone, String mobilePhone,
String primaryEmailAddress, String secondaryEmailAddress) {

    Customer customer = customerDao.readCustomer(customerId);

    if (customer == null) {
        customer = new Customer();
        customer.setCustomerId(customerId);
    }

    customer.setCustomerFirstName(customerFirstName);
    customer.setCustomerLastName(customerLastName);
    customer.setStreetAddress1(streetAddress1);
    customer.setStreetAddress2(streetAddress2);
    customer.setCity(city);
    customer.setStateOrProvince(stateOrProvince);
    customer.setPostalCode(postalCode);
    customer.setCountry(country);
    customer.setHomePhone(homePhone);
    customer.setMobilePhone(mobilePhone);
    customer.setPrimaryEmailAddress(primaryEmailAddress);
    customer.setSecondaryEmailAddress (secondaryEmailAddress);

    customerDao.saveCustomer(customer);
}
```

刻意保持例子的简单

必须得承认，以上代码并不表示一个有趣的领域，但是却帮助我们看到了一个欠妥的设计，我们可以将其重构成更好的模型。这里我们关注的并不是如何保存Customer数据，而是如何向模型中添加业务价值，即便就这个例子本身来说意义并不大。

以上代码完成了什么功能呢？事实上，以上代码的功能是相当强大的。不管一个Customer是新建的还是先前存在的；不管是Customer的名字变了还是他搬进了新家；不管是他的家用电话号码变了还是他有了新的移动电话；也不管他是改用Gmail还是有了新的E-mail地址，这段代码都会保存这个Customer。哇，好厉害的方法啊！

情况真是这样的吗？其实，我们并不知道saveCustomer()方法的业务场景。为什么一开始会创建这个方法？有人知道它的本来意图吗，还是它原本就是用来满足不同业务需求的？几周或几个月之后，我们便将这些忘得一干二净了。你不相信？那请看看该方法的下一个版本：

```
@Transactional
public void saveCustomer(
    String customerId,
    String customerFirstName, String customerLastName,
    String streetAddress1, String streetAddress2,
    String city, String stateOrProvince,
    String postalCode, String country,
    String homePhone, String mobilePhone,
    String primaryEmailAddress, String secondaryEmailAddress) {

    Customer customer = customerDao.readCustomer(customerId);

    if (customer == null) {
        customer = new Customer();
        customer.setCustomerId(customerId);
    }

    if (customerFirstName != null) {
        customer.setCustomerFirstName(customerFirstName);
    }
    if (customerLastName != null) {
        customer.setCustomerLastName(customerLastName);
    }
    if (streetAddress1 != null) {
        customer.setStreetAddress1(streetAddress1);
    }
    if (streetAddress2 != null) {
        customer.setStreetAddress2(streetAddress2);
    }
    if (city != null) {
        customer.setCity(city);
    }
    if (stateOrProvince != null) {
        customer.setStateOrProvince(stateOrProvince);
    }
    if (postalCode != null) {
        customer.setPostalCode(postalCode);
    }
    if (country != null) {
        customer.setCountry(country);
    }
    if (homePhone != null) {
        customer.setHomePhone(homePhone);
    }
    if (mobilePhone != null) {
        customer.setMobilePhone(mobilePhone);
    }
    if (primaryEmailAddress != null) {
        customer.setPrimaryEmailAddress(primaryEmailAddress);
    }
    if (secondaryEmailAddress != null) {
```

```
        customer.setSecondaryEmailAddress (secondaryEmailAddress);  
    }  
  
    customerDao.saveCustomer(customer);  
}
```

我得说，以上方法还算不上糟糕到了极点。很多时候数据-映射 (data-mapping) 代码将变得非常复杂，此时大量的业务逻辑便不能反映在代码里了。

现在，除了customerId之外，所有的参数都是可选的，我们至少可以在某些业务场景下使用该方法。但是，我们就能说这是好的代码吗？我们如何测试这段代码以保证在错误的业务场景下该段代码不应该保存一个Customer呢？

都不用讨论过多的细节我们便知道，在很多情况下该方法是不能正常工作的。可能数据库约束会防止对非法状态的保存，但你是不是又得去查看数据库啦？你会在Java对象属性和数据库表的列名之间辗转反侧，然后可能发现你缺少数据库约束或者约束并不完全。

你可能会查看很多客户代码，然后比较代码历史，找出saveCustomer()的来龙去脉。你会发现，没有人能够解释这个方法为什么会成为现在这个样子，也没有人知道究竟有多少客户代码在正确地使用saveCustomer()方法。要自己去搞明白这里的缘由，你得花上几个小时甚至几天的时间。

牛仔的逻辑

AJ: “这哥们儿疑惑了，他不知道是应该吃土豆呢，还是吃牛肉？”



这个时候，领域专家是帮不上忙的，因为他们看不懂代码。即便领域专家能够看懂代码，他可能也会被这段代码搞得一头雾水。我们难道就不能用另外一种方式来改善这段代码吗？如果可以，怎么修改？

上面的saveCustomer()至少存在三大问题：

1. saveCustomer()业务意图不明确。
2. 方法的实现本身增加了潜在的复杂性。
3. Customer领域对象根本就不是对象，而只是一个数据持有器 (data holder)。

我们将这种情况称为“由贫血症导致的失忆症。”在实际项目中,这种症状发生得太多了。

等等!

这时你可能在想,“我们的设计都是在白板上进行的啊。我们会绘制设计框图,只有大家都达成一致时,我们才开始编码实现。”

如果情况是这样,那么请不要将设计和实现分开。记住,在实施DDD时,设计就是代码,代码就是设计。换句话说,白板图并不是设计,而只是我们讨论模型的一种方式。

现在你可能有些担心,“我要如何才能做到更好的设计呢?”用不着担心,你会成功的,继续读下去。

如何DDD

让我们暂时撇开关于实现细节的讨论,现在来看看DDD最具威力的特性之一:通用语言。通用语言和**限界上下文 (Bounded Context)**同时构成了DDD的两大支柱,并且它们是相辅相成的。

上下文术语

就现在来说,可以将限界上下文看成是整个应用程序之内的一个概念性边界。这个边界之内的每种领域术语、词组或句子——也即通用语言,都有确定的上下文含义。在边界之外,这些术语可能表示不同的意思。我们将在第2章中对限界上下文做深入探讨。

通用语言

通用语言是团队共享的语言。领域专家和开发者使用相同的通用语言进行交流。事实上,团队中每个人都使用相同的通用语言。不管你在团队中的角色如何,只要你是团队的一员,你都将使用通用语言。

那么,你认为你已经知道了什么是通用语言了?

很明显,通用语言是一种业务语言。

抱歉,不是。

通用语言必须采用工业标准术语。

不完全是。

通用语言是领域专家专用的。

对不起,不是。

通用语言是团队自己创建的公用语言，团队中同时包含领域专家和软件开发人员。

对了。

自然地，领域专家对通用语言有很大的影响，因为他们最了解业务，并且深受工业标准的影响。但是，通用语言更多地是关于业务本身如何思考和运作的。此外，很多时候，不同领域专家会在概念和术语上产生分歧，他们甚至也会犯错，因为他们也无法了解每种业务用例。因此，当领域专家和开发者一起创建领域模型的时候，他们有时会达成一致，有时会做一些妥协，但最终目的都是为了创造最适合项目的通用语言。团队成员们妥协的绝对不应是通用语言的质量，而是概念、术语和含义。然而，最初的一致并不表示始终一致，就像其他语言一样，通用语言也会随着时间推移而不断演化改变。

要使开发者和领域专家一样了解业务没有什么窍门。通用语言也不是强加在开发者身上的晦涩业务术语。通用语言是由整个团队共同创建的一门语言，其中包括领域专家、开发者、业务分析员等。在开始的时候，通用语言可能只包含由领域专家使用的术语，但是随着时间推移，通用语言将不断壮大成长。

在表1.4中，对于“注射流感疫苗”这个业务用例，我们不仅要完成建模，还应该让团队使用相同的通用语言。当团队讨论到业务模型时，他们会说：“护士给病人注射标准剂量的流感疫苗。”

表1.4 分析“注射流感疫苗”的最佳模型

哪种更能描述业务？	
虽然第2栏和第3栏的业务描述有些相似，但是它们的代码呢？	
可能的业务描述	生成的代码
“谁管呢？写代码就行了。” 哎，不着边际。	<pre>patient.setShotType(ShotTypes.TYPE _ FLU); patient.setDose(dose); patient.setNurse(nurse);</pre>
“我们给病人注射流感疫苗。” 好点了，但丢失了某些重要的概念。	<pre>patient.giveFluShot();</pre>
“护士给病人注射标准剂量的流感疫苗。” 这才是我们想要的。	<pre>Vaccine vaccine = vaccines. standardAdultFluDose(); nurse.administerFluVaccine(patient, vaccine);</pre>

由于通用语言最初只来自于领域专家,分歧是难免的。然而,这正是创建最佳通用语言的自然过程。在这个过程中,团队成员通过讨论、参考资料、引用标准、查阅词典等对通用语言进行改进。有时我们发现,有些我们曾经认为能很好表达业务的词汇不再适用了,而另外的一些词汇具有更好的效果。

那么,你该如何掌握通用语言呢?这里有一些试验性的方法:

- 同时绘制物理模型图和概念模型图,并标以名字和行为。虽然这些图并不是正式的设计图,但它们却包含了软件建模的某些方面。即使你的团队在使用统一建模语言(Unified Modeling Language, UML)来完成正式建模,也不要得意忘形,因为这样可能反而不利于团队的讨论,最终将阻碍通用语言的产生。
- 创建一个包含简单定义的术语表。将你能想到的术语都罗列出来,包括好的和不好的,并注明好与不好的原因。在你给术语下定义时,你在不经意间就会创造出一些可重用的词汇,因为此时你使用的是领域中的通用语言。
- 如果你不喜欢术语表,可以采用其他类型的文档,但记得将那些“非正式”的模型图也包含进去。同样,这里最终的目的也是发现通用语言中的术语和词组。
- 由于团队中有些人工作在术语表上,还有些人工作在文档上,此时你需要找到团队的其他人员来检查你的成果。分歧肯定是有,你应该对此有所准备。

以上是建立通用语言的一些理想化步骤,这样建立起来模型肯定不能直接用来指导开发,而只是建立通用语言的起步而已。此后,改进之后的通用语言将反映到系统的源代码中,比如Java、C#或者Scala等。以上的模型图和文档并未表明通用语言会随着时间而扩大。在通用语言开发早期,这些材料可能会对我们产生鼓舞,但是时间一久,它们也将变得过时。这也是为什么只有团队的交流和代码才能持续到最后的原因,也只有这两者才能实时地反映通用语言。

由于团队交流和代码才是对通用语言的持续表达,你应该试着抛弃那些模型图、术语表和文档。虽然这并不是DDD所要求的,但是这样做的确很实用,因为我们很难将项目文档和软件系统保持同步。

有了以上认识,我们便可以重新设计saveCustomer()方法了。我们将修改Customer,使其能够反映出它应该支持的业务操作:

```
public interface Customer {
    public void changePersonalName(
        String firstName, String lastName);
    public void postalAddress(PostalAddress postalAddress);
    public void relocateTo(PostalAddress changedPostalAddress);
    public void changeHomeTelephone(Telephone telephone);
    public void disconnectHomeTelephone();
    public void changeMobileTelephone(Telephone telephone);
    public void disconnectMobileTelephone();
    public void primaryEmailAddress(EmailAddress emailAddress);
    public void secondaryEmailAddress(EmailAddress emailAddress);
}
```

当然，以上的Customer并不是一个完美的模型，然而在实施DDD时，对设计的反思正是我们所期望的。作为一个团队，我们可以自由地讨论什么样的模型才是最好的，在对通用语言达成了一致之后，才开始着手开发。然而，即便我们可以对通用语言进行一遍又一遍地提炼，此时上面的例子已经能够反映出一个Customer应该支持的业务操作了。

另外，我们还应该知道，对领域模型的修改也将导致对应用层的修改。每一个应用层的方法都对应着一个单一的用例流：

```
@Transactional
public void changeCustomerPersonalName(
    String customerId,
    String customerFirstName,
    String customerLastName) {

    Customer customer = customerRepository.customerOfId(customerId);

    if (customer == null) {
        throw new IllegalStateException("Customer does not exist.");
    }

    customer.changePersonalName(customerFirstName, customerLastName);
}
```

这和最开始的saveCustomer()例子是不同的，在那个例子中，我们使用了同一个方法来处理多个用例流。在这个新的例子中，我们只用一个应用层方法来修改Customer的姓名，除此之外，该方法别无其他业务功能。因此，在使用DDD时，我们应该对照着模型的修改相应地修改应用层。同时，这也意味着用户界面所反映的用户操作也变得更加狭窄。但是无论如何，这个特定的应用层方法不再要求我们在用户姓名参数之后跟上10个null了。

对这个新例子你还算满意吧? 通过阅读代码你便能理解它的业务意图。你还可以通过测试来保证它的功能, 即只修改Customer的姓名。

因此, 我们使用通用语言来捕捉特定核心业务领域中的概念和术语, 它是一种团队模式。软件模型包含名词、形容词、动词和一些富有含义的语句等, 团队成员便通过这些语言进行交流。软件实现和测试中也使用和团队语言一样的通用语言。

是通用, 不是万能

我想, 关于通用语言, 有必要再做一点澄清。在理解通用语言时, 我们必须牢牢记住以下几点:

- 这里的“通用”意思是“普遍的”, 或者“到处都存在的”。通用语言在团队范围内使用, 并且只表达一个单一的领域模型。
- “通用语言”并不表示全企业、全公司或者全球性的万能的领域语言。
- 限界上下文和通用语言间存在一对一的关系。
- 限界上下文是一个相对较小的概念, 通常比我们起初想象的要小。限界上下文刚好能够容纳下一个独立的业务领域所使用的通用语言。
- 只有当团队工作在一个独立的限界上下文中时, 通用语言才是“通用”的。
- 虽然我们只工作在一个限界上下文中, 但是通常我们还需要和其他限界上下文打交道, 这时可以通过**上下文映射图 (3)**对这些限界上下文进行集成。每个限界上下文都有自己的通用语言, 而有时语言间的术语可能有重叠的地方。
- 如果你试图将某个通用语言运用在整个企业范围之内, 或者更大的、夸企业的范围内, 你将失败。

当你开始一个项目, 而该项目已经在使用DDD了, 此时你需要将你正在开发的独立限界上下文识别出来, 这样便在你的领域模型周围加上了一个显式的边界。此时, 你应该在这个限界上下文中使用其专属的通用语言。对于那些不包含在通用语言中的概念, 你应该拒绝使用。

使用DDD的业务价值

如果你的经验和我相当,你就应该知道软件开发者不应该只是热衷于技术,而是应该将眼界放得更宽。我认为不管使用什么技术,我们的目的都是提供业务价值。而如果我们采用的技术确实产生了业务价值,人们就没有理由拒绝我们在技术上的建议。

如果我们提供的技术方案比其他方案更能够产生业务价值,那么我们的业务能力也将增强。

业务价值最重要吗?

当然啦,我甚至都在想是不是应该将“使用DDD的业务价值”这一节再往前放一些。更确切地讲,该章节的题目叫“如何向你的老板推销DDD”更为合适。我并不希望你将本书看作只是些理论,而是希望本书对你的公司具有实际的指导意义。

让我们来看看DDD所带来的一些非常理想化的业务价值。记得将这些分享给你的管理层、领域专家和技术人员。我们可以将DDD的业务价值大致总结为以下几点:

1. 你获得了一个非常有用的领域模型
2. 你的业务得到了更准确的定义和理解
3. 领域专家可以为软件设计做出贡献
4. 更好的用户体验
5. 清晰的模型边界
6. 更好的企业架构
7. 敏捷、迭代式和持续建模
8. 使用战略和战术新工具

1. 你获得了一个非常有用的领域模型

DDD强调将精力花在对业务最有价值的东西上。我们并不过度建模,而是关注业务的核心域。有些模型是用来支撑核心域的,它们同样是重要的。但是,这些起支撑作用的模型在优先级上没有核心域高。

当我们将关注点放在自己的业务和别人业务的区别上时,我们便能更好地理解自己的任务所在,同时我们将更具竞争优势。

2.你的业务得到了更准确的定义和理解

业务人士能够更好地理解业务本身。我甚至听说通用语言曾经出现在某些公司的市场营销材料中。

随着业务模型的不断改善,人们对业务的理解也将更加深刻。在团队讨论的过程中,一些业务细节被不断地暴露出来,这些细节有助于掌握业务价值。

3.领域专家可以为软件设计做出贡献

当人们对自己的核心业务有了更深的了解时,业务价值自然就出来了。领域专家并不总是同意某些概念和术语。有时,分歧源自于领域专家们在其他公司工作时所积累起来的经验,而有时分歧则源自于公司内部。不管如何,当领域专家们在一起工作时,他们最终将达成一致意见,这对于整个公司来说都是件好事。

开发者和领域专家共享同一套交流语言,领域专家将知识传递给开发者。开发者总是会离开的,有可能去接触一个新的核心域,也有可能跳槽到其他公司,这时培训和工作移交也将变得更加简单,而“只有少数人才了解模型”的情况将大大减少。领域专家、剩下的开发者和新进人员可以继续使用通用语言进行交流。

4.更好的用户体验

用户体验可以更好地反映出领域模型的好坏。

如果软件留下太多的地方让用户自己去理解,用户往往需要经过培训才能做出操作决定。实际上,用户只是将他们所理解的转移到表单(form)中的数据而已。数据将被存储起来,如果用户不知道数据的用途,那么结果也将是错误的。

当用户体验是按照领域专家心中的模型来设计时,就不会出现以上的问题了。这时软件本身便能对用户起到培训作用,而不需要业务人员来提供培训。效率提高了,培训减少了——这就是业务价值。

接下来我们看看技术为业务创造的价值。

5.清晰的模型边界

我们并不鼓励技术团队将精力单纯地放在编码和算法上,而是期望他们能够面向业务。明确的目标产生高效的解决方案,而要达到这样的目的往往需要更好地理解项目的限界上下文。

6.更好的企业架构

一旦限界上下文得到了较好的理解和仔细的划分,那么团队的所有成员应该知道限界上下文间的集成是必要的。上下文之间的边界和关系是明晰的。当不同上下文的模型间存在依赖关系时,我们将使用上下文映射图来集成不同的限界上下文,而这又有助于我们全面地了解整个企业的架构。

7.敏捷、迭代式和持续建模

“设计”这个词可能并不能取悦业务管理层。然而,DDD并不是一个重量级的设计方法和开发过程。DDD并不是画模型图,而是将领域专家的思维模型转化成有用的业务模型。DDD不是创建一个真实世界的模型,而是模仿现实。

团队的工作遵循敏捷方法——迭代式的,增量式的。任何一种敏捷方法,只要团队认为合适,都可以用于DDD项目。通过DDD创建出来的模型便是可工作的软件。团队会对模型做持续的改进,直到业务层没有新的需求为止。

8.使用战略和战术新工具

限界上下文为团队创建了一个建模边界,成员在边界内部为特定的业务领域创建解决方案。在单个限界上下文中团队成员共享一套通用语言。不同的团队有时各自负责一个限界上下文,此时可以使用上下文映射图在战略层面上对限界上下文进行界分和集成。在某个建模边界内部,团队将使用战术建模工具:聚合(Aggregate, 10)、实体(Entity, 5)、值对象(Value Object, 6)、领域服务(Domain Service, 7)和领域事件(Domain Event, 8)等。

实施DDD所面临的挑战

在实施DDD的过程中, 挑战是不可避免的。那么, 有人成功过吗? DDD都有哪些常见的挑战, 我们又如何处理它们? 我将讨论以下三点最常见的挑战:

- 为创建通用语言腾出时间和精力
- 持续地将领域专家引入项目
- 改变开发者对领域的思考方式

使用DDD最大的挑战之一便是: 我们需要花费大量的时间和精力来思考业务领域, 研究概念和术语, 并且和领域专家交流, 以发现、捕捉和改进通用语言。如果你想完全采用DDD来最大化业务价值, 你需要做出很多努力, 并且花费很多时间。事实就是这样的。

要将领域专家引入你的项目恐怕也不是一件易事。但是不管有多么困难, 这是你必须做的。如果你连一个领域专家都找不到, 那么你根本无法对一个领域有深入的理解。当你找到领域专家的时候, 此时开发者应该表现出主动。开发者应该找领域专家交谈并仔细聆听, 然后将你们的谈话转化成软件代码。

如果你所工作的领域和业务相去甚远, 领域专家所了解的也只是一些边边角角, 那么此时你应该将这种问题暴露出来。在我曾经工作的一个项目里, 真正的领域专家很难找到, 有时他们还会到处出差, 我得等上好几周才能和他们开上一次会。在一些小型的公司里, 领域专家通常是CEO或者副总裁, 他们的事情太多了, 这时你也别指望他们能做好你的领域专家。

牛仔的逻辑

AJ: “如果你逮不到那头公牛, 你就得挨饿咯!”



引入领域专家需要创造性……

如何在项目中引入领域专家

咖啡。使用这种通用语言:

“Hi, Sally, 我给你泡了一杯泡沫牛奶咖啡, 你有时间聊聊……?”

学习C级经理使用的通用语言: “……利润……收入……竞争优势……市场优势。”



多数开发者在采用DDD时都需要转变自己思考问题的方式。作为开发者,我们都是技术思想者,技术实现对于我们来说并不是什么难事。我并不是说技术地思考不好,只是说有时少从技术层面去思考会更好。这么多年来,我们都习惯了单从技术层面完成软件开发,那么现在,是时候考虑一种新的思考方式了。为你的业务领域开发一门通用语言便是一个好的出发点。

牛仔的逻辑

LB: “那家伙的靴子太小了, 如果他不换双新的, 他的脚指头可能要受罪了。”

AJ: “对, 如果他不听的话, 就有他好受的了。”



在DDD中,我们会谈及到对概念的命名。对于概念命名而言,我们有更高层面的要求。当我们对一个领域进行建模时,我们需要仔细地考虑什么样的对象做什么样的事情,这是关于对象行为设计的。我们希望对对象行为的命名能够传达准确的业务含义,也即反映通用语言。要达到这样的目的,肯定不是先在类上定义属性,然后向客户端代码暴露getter和setter那么简单。

现在让我们来看看一个更有趣的领域,这个领域比之前那个Customer例子更具挑战性。这里,我刻意重复一下先前所讲的。

如果我们只是对领域模型提供getter和setter会怎么样? 答案是,结果我们只是在创建纯数据模型。看看下面的两个例子,自己思考一下,哪一个在设计上是欠妥的,哪一个对客户代码更有益。在这两个例子中是一个Scrum模型,我们需要将一个待定项 (Backlog Item) 提交到冲刺 (Sprint) 中去。这样的事情你可能一直在做,因此对这个领域你应该是很熟悉的。

第一个例子, 通常的做法, 使用属性访问的方式:

```
public class BacklogItem extends Entity {
    private SprintId sprintId;
    private BacklogItemStatusType status;
    ...
    public void setSprintId(SprintId sprintId) {
        this.sprintId = sprintId;
    }

    public void setStatus(BacklogItemStatusType status) {
        this.status = status;
    }
    ...
}
```

客户代码如下:

//客户端通过设置sprintId和status将一个BacklogItem提交到Sprint中

```
backlogItem.setSprintId(sprintId);
backlogItem.setStatus(BacklogItemStatusType.COMMITTED);
```

第二个例子使用了领域对象的行为, 这种行为表达出了领域中的通用语言:

```
public class BacklogItem extends Entity {
    private SprintId sprintId;
    private BacklogItemStatusType status;
    ...

    public void commitTo(Sprint aSprint) {
        if (!this.isScheduledForRelease()) {
            throw new IllegalStateException(
                "Must be scheduled for release to commit to sprint.");
        }

        if (this.isCommittedToSprint()) {
            if (!aSprint.sprintId().equals(this.sprintId())) {
                this.uncommitFromSprint();
            }
        }

        this.elevateStatusWith(BacklogItemStatus.COMMITTED);

        this.setSprintId(aSprint.sprintId());

        DomainEventPublisher
```

```
        .instance()
        .publish(new BacklogItemCommitted(
            this.tenant(),
            this.backlogItemId(),
            this.sprintId()));
    }
    ...
}
```

此时的客户代码如下：

```
//客户端通过特定于领域的行为将BacklogItem提交到Sprint中

backlogItem.commitTo(sprint);
```

第一个例子采用的是以数据为中心的方式，此时客户代码必须知道如何正确地将一个待定项提交到冲刺中。这样的模型是不能称为领域模型的。如果客户代码错误地修改了sprintId，而没有修改status会发生什么呢？或者，如果在将来有另外一个属性需要设值时又该怎么办？我们需要认真分析客户代码来完成从客户数据到BacklogItem属性的映射。

这种方式同时也暴露了BacklogItem的数据结构，并且将关注点集中在数据属性上，而不是对象行为。你可能会反驳道：“setSprintId()和setStatus()就是行为啊。”问题在于，这里的“行为”没有真正的业务价值，它并没有表明领域模型中的概念——此处即“将待定项提交到冲刺中”。开发者在开发客户代码时，他并不清楚到底需要为BacklogItem的哪些属性设值，而这样的属性有可能存在很多，因为这是一个以数据为中心的模式。

现在，我们来看看第二个例子。有别于第一个例子，它将行为暴露给客户，行为方法的名字清楚地表明了业务含义。这个领域的专家在建模时讨论了以下需求：

允许将每一个待定项提交到冲刺中。只有在一个待定项位于发布计划 (Release) 中时才能进行提交。如果一个待定项已经提交到了另外一个冲刺中，那么需要先将其回收。提交完成时，通知相关客户方。

在第二个例子中，客户代码并不需要知道提交BacklogItem的实现细节。实现代码所表达的逻辑恰好能够描述业务行为。我们很容易地添加了几行代码，以确保在发布计划之外的待定项是不能被提交的。诚然，在第一个例子中，你可以修改setter以达到同样的目的，但此时该setter的职责便不单一了，它需要了解BacklogItem对象的内部状态，而不再只是对sprintId和status属性赋值。

这里还有一个微小的区别。如果一个待定项已经被提交到了另外的冲刺中，那么我们应该先从那个冲刺中回收该待定项。这一点也是重要的，因为当一个待定项从冲刺中回收时，将有领域事件发出以通知客户方：

允许从冲刺中回收任何一个待定项，回收时通知相关客户方。

此时，我们并不需要关心如何发布回收事件，因为`uncommitFrom()`方法会为我们处理这些。而`commitTo()`方法甚至都不知道发布回收事件这码事，它只需要知道，在将待定项提交给一个新的冲刺时，必须先将该待定项从它当前所在的冲刺中回收。另外，`commitTo()`的领域行为还包括：在提交待定项完毕后，以事件形式通知相关客户方。如果不是这个富含行为的`BacklogItem`，我们得在客户代码中发布领域事件，这显然是一种领域逻辑的泄漏。

很明显，在第二个例子中，我们对`BacklogItem`有了更多的思考，但同时我们也获得更多的回报。沿着这条路往下走，我们将越走越容易。到后来，我们肯定需要更多的思考、付出和团队协作，但是这并不会使DDD变得笨重。

白板时间

- 对于你目前正在工作的业务领域，思考一下模型中的通用术语和业务操作。
- 将术语写在白板上。
- 然后，将项目中所用到的短语也写下来。
- 与真正的领域专家交流一下，看看哪些词汇是可以改善的（记得带上咖啡哦）。

为领域建模正名

通常来说，战术建模比战略建模复杂。因此，如果你打算采用DDD的战术模式（聚合、领域服务、值对象和领域事件等）来建立领域模型的话，你需要更仔细的思考和更大的投入。那么，我们有什么理由依然要采用战术建模呢？我们又拿什么标准来衡量在DDD上的投入是值得的呢？

你可能已经在盘算，这将把你带到一个陌生的领地，你发现你得好好研究一下周边的情况。你的团队可能会学着既有的线路图，甚至开辟一条新路来决定自己

的战略设计方案。你可能会仔细捉摸这片新的领地，然后试图使其为你所用。然而，不管你事先做了多少准备，这都将是一条荆棘丛生之路。

如果你发现你需要在战略的岩石上攀爬，那么你得找到一套合适的战术工具来辅助你。站在低处往上看，你有可能看到一些特别的挑战和危险地带。然而，如果不爬到那样的高度，你又是看不清楚的。你可能需要在坚硬的岩石上打孔插钉，但是也可以找到那些自然形成的裂缝。你可能还需要带上锁环以保证安全。你可以沿着一条路线顺直而上，也可以打点布阵、步步为营。有时随着岩石形状的走势，你可能需要往回撤，再重新设计路线。有人认为攀岩是种危险的运动，但是那些尝试过的人会告诉你，攀岩实际上比驾驶汽车和飞机还安全。攀岩者需要知道如何使用工具和运用好技能，并且能够根据岩石状况做出相应的反应。

如果说开发一个业务子域 (Subdomain, 2) 就像攀岩一样困难，那么我们需要随身携带DDD的战术模式来武装自己。对于满足核心域标准的业务来说，我们不应该将战术模式拒之门外。半途而废的项目时有发生，而正确的战术模式可以帮助我们减少这种情况的发生。

这里是一些实际的指导意见，我会先讲高层次的，然后讲更具体的：

- 如果一个限界上下文被当成核心域来开发，那么从战略上来说，这个限界上下文对业务的成功是极其重要的。核心模型是不易理解的，需要不断地尝试和重构。通过持续改进，我们可以延长它的效用生命，这样的做法显然是值得的。当然，这个限界上下文不见得始终是你的核心域。即便如此，如果它是复杂的，创新性的，并且需要在不断的变化中持续存在很长时间，我们还是建议在该限界上下文中使用战术模式。这里，我们假设你的核心域是值得配置最好的开发者的。
- 一个领域，对于消费方来说有可能成为通用子域 (Generic Subdomain, 2) 或者支撑子域，但是却有可能成为你自己的核心域。我们并不站在最终消费方的角度来评价一个领域。如果你正在开发的限界上下文是你主要的业务，那么它便是你的核心域，而不管消费方是如何看待的。此时，一定记得使用战术模式。
- 如果你正开发一个支撑子域，但是由于种种原因，该支撑子域不能从第三方的通用子域直接获得，那么此时战术模式将帮上你大忙。在这种情况下，你需要考虑团队成员的技能水平，还有模型是否具有创新性。如果此时的模