分类号：　　密级：

U D C ：　　学号：406107708094

南 昌 大 学 硕 士 研 究 生

学 位 论 文

领域驱动设计方法的研究及其应用

Research and Application of Domain-Driven Design

严欣喆

培养单位（院、系）：信息工程学院计算机系

指导教师姓名、职称：林仲达 副教授

申请学位的学科门类：工学

学科专业名称：计算机应用技术

论文答辩日期：2010 年 12 月 18 日

答辩委员会主席：_____

评阅人：_____

_____

年　月　日

学位论文独创性声明

本人声明所呈交的学位论文是本人在导师指导下进行的研究工作及取得的研究成果。据我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表或撰写过的研究成果，也不包含为获得南昌大学或其他教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文作者签名（手写）：[illegible]　签字日期：2010 年 12 月 22 日

学位论文版权使用授权书

本学位论文作者完全了解南昌大学有关保留、使用学位论文的规定，有权保留并向国家有关部门或机构送交论文的复印件和磁盘，允许论文被查阅和借阅。本人授权南昌大学可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编本学位论文。同时授权中国科学技术信息研究所和中国学术期刊（光盘版）电子杂志社将本学位论文收录到《中国学位论文全文数据库》和《中国优秀博硕士学位论文全文数据库》中全文发表，并通过网络向社会公众提供信息服务。

（保密的学位论文在解密后适用本授权书）

学位论文作者签名（手写）：[illegible]　导师签名（手写）：[illegible]

签字日期：2010 年 12 月 22 日　签字日期：[illegible] 年 [illegible] 月 [illegible] 日

摘 要

领域驱动设计思想自诞生以来就引起了人们的广泛重视，被认为是未来软件设计的主导思想。它完全颠覆了传统基于数据库设计的开发方式，强调了领域的概念，将软件系统的复杂性从技术转移到了领域本身，使开发人员因此能够以更加自然的观点去看待要解决的问题，扩大了人们一次性所能处理的问题规模。领域驱动设计思想诞生不过数年，还是一种年轻的思想，整个思想体系也并未完全成型，尽管如此，它依然吸引了越来越多的软件开发人员加入它的研究。领域驱动设计的研究在未来是一个充满挑战的工作。

本文对领域驱动设计的基本理论进行了研究，将理论中抽象的概念具体化，并根据领域驱动设计的基本理论，提出了一种轻量级领域驱动设计框架，使用该框架可以方便快速的进行软件开发，减轻开发人员的负担。

本文的研究工作主要包括以下几个方面：

1. 从本质上分析了领域驱动设计的基本理论，将六种抽象的领域模型元素模式概念具体化，对一些经常被误解的模式作出了解释，并从宏观角度给出了基本模式之间的关联。
2. 提出了一种轻量级领域驱动设计框架。该框架作为领域驱动设计思想的一种实现方式，能够将复杂的领域逻辑变得有条理，并封装繁琐的技术细节。该框架对系统架构、领域模型、数据访问模块以及数据库表结构的设计方法都作出了详细规定，并提供了数据访问模块的具体实现。
3. 将领域驱动设计方法应用于实际系统开发中。该系统为某水泥厂综合管理系统，涵盖产品管理、销售管理、部门管理、员工管理、客户管理等多方面内容，具有代表意义。

关键词：领域驱动设计；轻量级；框架；应用

ABSTRACT

The concept of Domain-Driven Design has been widely paid attention to since it was born,people think it will be the main methodology in software design.It emphasized the concept of domain,droped the traditional methodology in software design based on the database,transfered the complexity of software system from technology to the domain,make the software developers regard the problems with a more natural way,and expanded the scale of problems which people can process at a time.Though the concept of Domain-Driven Design was just born for several years, is still a young concept,and has not been completed,it attracted more and more software developers to research it.The research of Domain-Driven Design is a challenging work in the future.

This thesis researched the basic theory of Domain-Driven Design,materialized the abstract concepts in the theory,and proposed a lightweight framework based on Domain-Driven Design.This framework could make the software development convenient and rapid,and lighten the burden on the soft development personnel.

The main researches of this thesis are as follows:

1. Analyzed the Domain-Driven Design theory essentially, materialized abstract concepts of 6 domain model element patterns in Domain-Driven Design theory,explained some concept which misunderstood frequently,and given connections among the basic concepts from the macroscopic way.
2. Proposed a lightweight framework based on Domain-Driven Design.As a realization of Domain-Driven Design,this framework could make the complex domain logic in order,and encapsulate the tedious technical details.This framework made detailed stipulation on design method of system architecture,domain model,data accessing module and database table structure,and supplied the concrete realization of the data accessing module.
3. Applyed Domain-Driven Design in the actual system development.This

system is a management system of a cement plant,included management of product,sales,department,employee,customer and so on.This system has the typical significance.

Key Words: Domain-Driven Design; Lightweight; Framework; Application

目 录

第 1 章 引言

1.1 概述

上世纪 80 年代以来，随着电脑技术的飞速发展，计算机软件开始在各种领域得到广泛应用，尤其是互联网技术产生之后，Web 应用的规模越来越大，其复杂性也越来越高[1]。因此，如何高效高质量的开发软件成为人们日益重视的问题，出现了诸如软件工程、RUP 等系统化软件开发模型[2]。面向对象思想按照对象（事物、概念、或实体）的观点考虑问题域和逻辑解决方案[3]，渐渐成为了软件开发方法的主流。在面向对象思想基础上，人们提出了模式的概念。“每一个模式描述了一个在我们周围不断重复发生的问题以及该问题解决方案的核心。这样，你就能一次又一次的使用该方案而不必做重复劳动”[4]。模式有效而且有足够的通用性，能解决重复出现的问题[5]。软件框架作为这些成熟设计思想的具体实现，可以在后台自动解决与设计要求无关的基础问题，使得人们在开发软件时能够集中精力于设计需求，而无需关注技术细节，因此，近年来，软件框架的设计变得十分流行，各种各样的软件框架层出不穷，软件开发得到了极大发展。

然而，与技术的发展相比，软件开发的思想却没有得到相应提升。长期以来，无论使用何种软件框架，软件开发的指导原则依然是基于数据库设计：项目一开始就根据需求建立数据库模型[6]。软件开发退化成了对数据库的 CRUD 操作，面向对象思想没有在实践中贯彻下去。我们使用大量的构件组件，却在编制面向过程的体系[7]。

领域驱动设计思想认为，基于数据库设计的软件开发原则背离了面向对象思想，软件开发应该以领域模型作为基础，给开发人员提供一个虚拟的现实领域模型，并将底层与具体平台有关的操作封装起来，使得开发人员能够集中精力于设计需求而无需关注技术细节。领域驱动设计思想摒弃了传统设计与实现分离的做法，强调了设计与实现的同步进行，认为领域专家与开发人员的合作应该贯穿整个软件系统开发的过程，而不是仅仅在需求分析阶段。

领域驱动设计思想将系统设计的复杂性从技术转移到了领域本身，大大降

低了复杂系统的开发难度，因而迅速引起了人们的广泛兴趣。然而，领域驱动设计是一种抽象思想，不同的人对此有着不同理解，不同的理解又产生出不同的实现，再加上其自身体系结构也尚未完善，使得领域驱动设计一直没有一个统一的标准，这成为了阻碍领域驱动设计思想流行的最大障碍。针对这种情况，本文对领域驱动设计的基本理论进行了研究，将理论中抽象的概念具体化，并根据领域驱动设计的基本理论，提出了一种轻量级领域驱动设计框架，该框架具有简单易用、配置轻巧等特点，能方便快速的进行软件开发，减轻开发人员的负担。

1.2 研究现状

自诞生以来，领域驱动设计就引起了人们极大的兴趣，但是，作为一种新型软件设计思想，领域驱动设计还未完全经过实践的检验。人们进行了大量研究，并尝试性的建立了不少基于领域驱动设计的软件框架，也取得了一定成就，如 RoR、RIFE、Jdon Framework、JavATE 等[8]，但总体来看，领域驱动设计还尚未在软件开发行业流行起来。

2004 年，建模大师 Eric Evans 提出了“领域驱动设计”的概念，标志着软件开发的本质从基于数据库设计提升到了基于模型设计。领域驱动设计将软件系统的复杂性从技术转移到了领域本身，使软件开发与实际生活更接近了，开发人员因此能够以更加自然的观点去看待要解决的问题，人们一次性所能处理的问题规模也随之扩大了许多。

2005 年，RoR 和 Jdon Framework 等支持领域驱动设计的开发框架诞生，尽管最初的版本功能略显单薄，但这是人们对领域驱动设计思想的第一次尝试实现。

2006 年，Jimmy Nilsson 发表了文献[9]，全面详细的解释了领域驱动设计理论，并以实例的形式展示了领域驱动设计在具体项目中的应用，第一次为开发人员从头至尾的揭示了完整的领域驱动开发路线。

同年，挪威国家石油公司将领域驱动设计思想应用于旗下项目的战略设计中，并取得了良好效果，这意味着领域驱动设计在复杂领域当中是一个不错的选择。文献[10-12]展示了领域驱动设计在该项目的具体应用。

如今，领域驱动设计作为公认的未来软件设计的指导原则，正在吸引越来

越多的软件开发人员参与其中、分享心得。关于领域驱动设计的研究，是一个充满希望、激情和挑战的工作。

1.3 本文组织结构

本文主要介绍领域驱动设计思想的本质，同时给出了一个领域驱动设计的轻量级实现框架，该框架建立在.NET 平台上，利用 C#语言实现，能较方便的实现复杂软件系统的开发。

本文共分五章，组织结构如下：

第一章：引言。介绍领域驱动设计的概念、研究现状以及本文组织结构。

第二章：领域驱动设计理论。从本质上分析领域驱动设计思想理论。

第三章：领域驱动设计框架。详细介绍利用领域驱动设计思想开发软件的的设计方法，并提出了一个轻量级领域驱动设计框架。

第四章：领域驱动设计方法的应用。介绍领域驱动设计方法在实际系统开发中的应用。

第五章：结论与展望。总结领域驱动设计的未来发展，对本文提出的软件框架给出了一些改进建议。

第 2 章 领域驱动设计理论

2.1 领域驱动的基本理论

领域驱动设计（Domain-Driven Design，简称 DDD），是建模大师 Eric Evans 于 2004 年出版的著作《Domain-Driven Design—Tackling Complexity in the Heart of Software》中提出的一种应用于复杂业务逻辑领域的软件开发方法，它将软件开发的着眼点集中于领域本身，通过建立领域模型来将复杂的领域逻辑清晰的表达出来，使得混乱的业务有序化，降低开发难度，提高开发效率。

在传统基于数据库设计的开发方式中，经常都能看到这样的设计：整个业务层到处看见的是数据表的影子（包括数据表的字段），而不是业务对象[13]，业务逻辑被机械化为对数据库的 CRUD 操作。这样开发出来的项目，没有领域知识的浸润，枯燥无味，苦涩难懂，使得领域专家没有插入的余地，而失去领域专家的协助，开发人员在项目需求发生变更需要对项目进行修改时，又会觉得难以入手，最终导致程序的设计逐渐腐败变质，程序员愈来愈难通过阅读源码而理解原本设计[14]，使得软件在以后的扩展和维护过程中存在很大的困难[15]。大量统计资料表明，软件测试阶段投入的成本和工作量往往要占软件开发总成本和总工作量的 40%到 50%，甚至更多[16][17]。

领域驱动设计完全颠覆了传统基于数据库设计的开发方式。传统基于数据库设计的开发方式认为，数据库系统的核心和基础是数学模型，而数学模型是对现实世界数据特征的抽象[18]。领取驱动设计则强调了领域模型的概念，认为数据库只是用于存储信息的仓库，与领域本身是无关的。在设计领域模型时，不应该考虑信息是如何存储的，信息存储在 XML 文件中或是各种不同类型的数据库中，都不影响领域模型的设计。

领域模型是对现实领域的模拟，它构造真实完整体现领域的领域对象[19]，通常以 UML 图（UML 介绍详见文献[20]）的形式表现，但绝对不能与图划等号，领域模型是图所要表达的思想，是一种仿真的、虚拟的、立体的结构，是开发人员与领域专家沟通的桥梁。为了描绘出标准的 UML 图，而淡化它要表达出的思想，这有点本末倒置[21]。借助于领域模型，开发人员和领域专家可以

交流协作，传达彼此的思想，领域专家可以告诉开发人员需要做什么、为什么要做这些，使得开发更有理性，开发人员可以告诉领域专家这是怎么做的、为什么要这么做，使得领域专家可以根据现实重新思考需要解决的领域问题。领域建模并不专注于建立一个逼真的模型，而是展示现实领域中我们需要的某些方面，这种展示是有目的的，并且会带有特定修饰。

领域模型经常被误认为是设计的基础：架构师吃掉需求，设计师吃掉架构，而程序员则消化设计[22]。实际上，敏捷软件开发认为，响应变化胜过遵循计划[23]。领域建模、设计、实现是密不可分的，领域驱动设计中不存在“建模——设计——实现”的定向思维。领域模型关注的是需求分析，它与编程和设计相互影响[24]。一个领域模型会随着系统研究的深入不断成熟、不断精化，它与设计、实现始终处于一种同步变化当中，这正是开发人员和领域专家在领域模型中不断交流的结果。一个无法实现的设计会立刻从开发人员反馈到领域专家，迫使领域专家重新思考领域问题，从而改变模型设计，这样避免了由于设计与实现割裂导致问题发现延迟可能带来的巨大损失。随着技术的进步，领域专家在软件开发中将发挥更大的作用[25]。

2.2 领域驱动设计的分层架构

架构是指软件平台的整体设计[26]。分层架构是目前软件行业确认的一种对软件系统分割的流行方法，流行的三层架构和四层架构等都是典型的分层架构[27]。分层架构的基本原则是：某一层中的所有元素只能依赖同一层的其他元素，或者依赖其下层元素[28]。分层的意义在于将软件系统的不同职责划分到不同层次，使得每一层只关注于某一特定方面的职责，提高了设计的内聚性与可理解性。领域驱动设计的分层架构如下[29]：

- 用户界面层（表示层）：负责向用户显示信息，并且解析用户命令。外部的执行者有时可能会是其他的计算机系统，不一定非是人。
- 应用层：定义软件可以完成的工作，并且指挥具有丰富含义的领域对象来解决问题。这个层所负责的任务对业务影响深远，对跟其他系统的应用层进行交互非常必要。这个层要保持简练，它不包括处理业务规则或知识，只是给下一层中相互协作的领域对象协调任务、委托工作。在这个层次中不反映业务情况的状态，但反映用户或程序的任务进度

的状态。

- 领域层（模型层）：负责表示业务概念、业务状况的信息以及业务规则。尽管保存这些内容的技术细节由基础结构层来完成，反映业务状况的状态在该层中被控制和使用。这一层是业务软件的核心。
- 基础结构层：为上层提供通用的技术能力：应用的消息发送、领域持久化，为用户界面绘制窗口等。通过架构框架，基础结构层还可以支持这四层之间的交互模式。

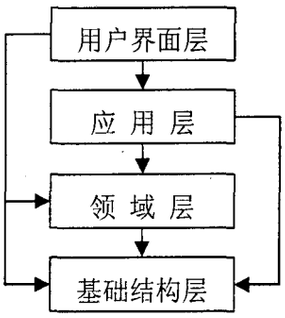

图 2.1 领域驱动设计的分层架构

把领域层隔离开来，可以使开发人员将重点集中在设计领域模型上，抓住问题的本质，而不去关心诸如显示、存储这些细枝末节的东西。

2.3 领域模型元素的模式

Eric Evans 提出了 6 种领域模型元素的模式：实体（Entity）、值对象（Value Object）、聚合（Aggregate）、服务（Service）、工厂（Factory）、仓储（Repository）。很多文章在描述领域驱动设计时都没有正确理解它们包含的内在含义。它们表达的是一种思想，而不是书本上教条式的定义。下面将从本质上阐述这 6 种模式的内在含义。

2.3.1 实体

实体是指领域中的事物以及可以施加在该事物上的所有作用。

事物并不仅仅是指有形的东西，书、汽车可以看成事物，规则、思想也可以看成事物。同类事物的不同具体事物凭借特定的、唯一的、不可更改的标识

来确认，并且标识不依赖于事物的属性，例如两本一模一样的书，依然是两个不同的具体事物。标识是用以区别同类事物的根本手段，也是事物存在的根本，事物通过标识来表明自己的存在。除了标识之外，事物还必须至少拥有一种属性，属性是事物存在的价值，我们研究事物，实际上就是关心它的属性。一个没有属性的事物是没有存在的意义的。事物不会随时间而改变，一个具体事物从创建到消亡，不论经历怎样的状态，始终保持自我的独立性。事物的标识和属性也被看作是实体的标识和属性。

施加在事物上的作用是指“与且仅与该事物有关的”操作，而非“该事物发起的”操作。“迪米特法则”指出，对象间的联系应当尽可能少，这样有利于保持对象的独立性。如果一个操作与且仅与某事物有关，那么这个操作就应当与该事物划分到一个实体，而无论它是否由该事物发起。

对实体认识的最大误区就是认为它仅仅是某种事物，忽略了施加在其上的作用。现实软件开发中许多充斥着“贫血实体类”的设计就是这种认识的具体表现。实际上，面向对象理论及软件工程早就指出，数据和对数据的处理原本是密切相关的，把数据和操作人为地分离成两个独立的部分，自然会增加软件开发与维护的难度[30]。一个完整的实体是事物及施加在该事物上的作用的统一体。

实体是领域中最重要的概念，领域模型的所有其它的概念都是围绕实体展开的。实体是领域真正感兴趣的对象。在软件设计中，实体通常是以类的形式展现，因此实体也被称作实体对象。

2.3.2 值对象

值对象本质上是实体的分解。世上的事物是可以无限分解的，大事物分解成小事物，小事物分解成更小的事物，一直到分子、原子……但在一个特定领域中，人们需要理解的事物的粒度是有限的，例如，大多数情况下，我们讨论一个人，并不需要明确他是由哪些细胞组成的，在这里，“人”就是我们需要理解的最小粒度。一个事物可以被看成值对象，意味着它的所有属性都已经被接受，它的标识就可以代表它的一切。也正因为如此，值对象是不可更改的，因为任何一个细小改动都会引起已经约定俗成的认识上的巨大混乱。

很多人误以为值对象是没有标识的，并例举出 Eric Evans 对值对象的定义

作为依据。实际上，这是对值对象肤浅片面的理解。Eric Evans 的意思是不需要人为的给予值对象标识，因为值对象在产生的时候本身就已经蕴含了标识，并不是说值对象是没有标识的。字母和数字都是值对象，我们在使用它们时，并不需要对“1”或“A”赋予标识，在多次使用时，也不关心我们使用的是哪个“1”或哪个“A”。但是，我们如何知道我们使用的是“1”而不是“2”，或者是“A”而不是“B”？“1”和“2”，以及“A”和“B”，实际上已经通过它们蕴含的标识区分开了，我们在不同地方使用的“1”和“A”，其实都是同一个“1”和“A”。

2.3.3 聚合

与值对象相反，聚合是实体的合并。在复杂领域中，实体数量繁多，关系错综复杂，很容易形成盘根错节的关系网，使得领域模型看起来非常混乱无序。因此，有必要把一些联系紧密的实体合并成一个更大的结构，以减少实体的数量，降低领域的复杂性。

聚合很好的担任了这个职责。一个聚合包括至少 2 个实体与一个“根（Root）”，根包含了聚合内所有实体的引用，并作为聚合的代表供外界访问，外界只能通过根来访问聚合内的实体，而聚合内的实体之间可以随意互相访问。包含在聚合内的实体构成了一个整体，当聚合消亡时，它们也将随之消亡。聚合使得实体间的关系网大大简化了，联系紧密的实体可以被放入聚合内部封装起来，聚合之间的联系则通过根变得清晰而有条理。

聚合并不是实体合并的顶点，正如分解是无穷的，合并也是无穷的，聚合与聚合可以再次合并，形成“超聚合”，并依次类推，一直到满足领域模型的需要为止。聚合也被称作聚合对象。

2.3.4 服务

服务指的是不属于任何实体和值对象的操作，包括与任何实体都无关的基础操作以及跨实体的操作。例如提供字符串加密功能的加密操作和银行系统中涉及到 2 个账户的转账操作。若领域中存在聚合，则跨实体的操作可以看成聚合的方法，此时，跨聚合的操作才被看成服务。

在传统软件开发中，服务经常被极端化。一种极端是抹杀掉服务的存在，

为了使之适合面向对象而将服务强行并入一些对象中，尽管这些对象与该服务所代表的操作没有很必然直接的联系。这种做法增加了对象间的依赖，为了使并入对象的服务能够正常工作，服务所在的对象不得不引用该服务所要求的另一些对象，这种额外增加的依赖破坏了对象的封装性，使对象变得混乱难懂，增加了二次开发的难度。

另一种极端是将服务作为操作的唯一存在，将所有操作都看成服务，不论它是否可以很恰当的划分到对象中去。这种做法在现行软件开发中非常流行，并且还得到了相当多的支持，认为它体现了“职责分离”的原则，减轻了实体数据与行为的依赖性。然而美国软件开发教父 Martin Fowler 却在自己主页上撰文强烈抨击这种做法，并形象的称这种做法造就的对象是“贫血”的，由此引发了业界关于“贫血模型”和“充血模型”的大讨论。实际上，这种将对象与操作分离的做法，尽管在一定程度上有利于提高中小型项目开发的效率，但却是违背面向对象原则的。缺少领域操作、仅仅拥有弱方法 getter 和 setter 的对象，与其说是对象，不如说是结构体更为恰当。脱离了操作的对象完全无法表达出领域的内涵，而脱离了对象的操作更是会迷失在复杂的领域逻辑中。

服务应当与对象内部的方法区分开。对象内部的方法，是指那些仅与该对象本身相关的操作，任何不涉及该对象或者同时涉及到其它对象的操作，都不应当划分为该对象内部的方法，而应该作为服务单独提出来。

在软件开发中，为了符合面向对象原理，通常将一组相关服务划分为一个类。并且，服务也是分层的。如果服务是与领域密切相关的，那它将属于领域层，如果它是纯技术上的，则应当属于基础结构层，如果它是面对具体应用的，则属于应用层。服务的分层如下：

- 应用层服务：读取输入，发送消息要求领域服务处理，确认消息，使用基础结构层的服务发送通告。
- 领域层服务：根据应用层服务的要求完成领域操作，并返回操作结果。
- 基础结构层服务：由应用层服务选择发送通告的方法（电子邮件、短信或其它通信方式）。

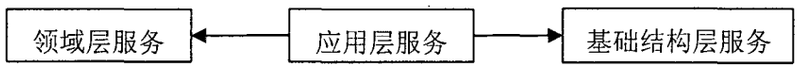

图 2.2 服务的分层

2.3.5 工厂

工厂是一组特殊的服务，它负责创建领域对象（实体、值对象、聚合），封装复杂的、没有显现价值的一些创建对象逻辑[31]。领域对象都是有生命周期的，它们的生命周期由工厂开始，由用户显式或系统自动结束。

实体、值对象、聚合是供应用层使用的，但如果将它们的创建工作也交由应用层负责，则会使应用层的设计变得混乱。更严重的是，如果创建工作包含了复杂的组装操作，则有可能将领域层的知识泄露到应用层去，使得层与层之间的耦合度大大增加了，破坏了封装性，不利于进一步的修改与重构。创建领域对象应当是领域层自己的工作，这个工作交由工厂实现，应用层通过工厂来创建满足自己需要的各种领域对象。工厂将领域对象的创建工作封装在其内部，应用层只需要使用工厂，不需要了解其内部细节，这样就隐藏了领域层的内部信息，增强了领域层的灵活性与独立性，也减轻了应用层的负担。

大多数面向对象语言都提供了创建对象的操作，如 Java、C#提供的 new 操作符。但是，对于一些复杂的、需要组装的领域对象，如聚合对象等，我们需要一种更加独立的机制来完成它们的创建工作。文献[32]中介绍的工厂方法（Factory Method）、抽象工厂（Abstract Factory）、建造者（Builder）三种模式为工厂在软件开发中的实现提供了很好的思路。此外，工厂还有一个重要意义，就是分离对象的使用和创建权限。如果直接使用面向对象语言本身提供的创建对象操作，那么由于规则的限制，所有可以使用对象的地方也同时可以创建对象，这就使得权限限制变得复杂甚至不可行。而如果将对象的创建工作全部放在工厂中完成，就可以很好的划分创建对象和使用对象的权限。

2.3.6 仓储

仓储是一组负责领域对象的持久化的服务。领域对象在其生命周期中并不时时刻刻都在被使用，当其闲置时，需要将其归档存储，以便腾出空间给其它需要使用的对象。在软件开发中，活跃的领域对象位于内存中以便随时调用，闲置的领域对象则存储在持久化介质中，如 XML 文件和各种数据库。当用户需要引用一个闲置的领域对象时，由仓储从持久化介质中将它取出，如果用户认为一个领域对象的生命周期结束了，则由仓储将它从持久化介质中删除。

在现有技术环境下，仓储是领域中最复杂的设计。尽管受到了面向对象数

据库和状态对象的挑战[33]，目前主流的持久化介质仍然是各种不同类型的关系数据库。关系数据库中数据的逻辑结构是二维表，但二维表不能很好的反映“对象”这个概念，对象是一个立体的结构，而二维表是一个平面的结构，将对象存储在关系数据库中，可以形象地比喻为将对象“碾平”了，给人不自然的感觉。更本质地说，数据表建模属于数学范畴，而面向对象建模属于哲学思维[34]。

为了解决面向对象与关系数据库的这种不匹配现象，使软件开发人员的思维集中在面向对象领域，对象-关系映射（Object/Relational Mapping，简称 ORM）技术应运而生。ORM 希望实现一个转换器的功能，在其内部实现对象-关系的互相转换，而在其外部给出一个面向对象形式的接口，使得软件开发人员能够以面向对象的思维访问关系数据库。一些 ORM 框架在软件业中取得了一定成功，比如 Hibernate（Hibernate 介绍详见文献[35]），已经成为了事实上的 Java ORM 工业标准。

然而，现行的成功 ORM 框架却并未必适合领域驱动的仓储设计。领域驱动设计希望软件开发人员能够真正融入到领域中去，将自己想象成在领域中工作，但现行的 ORM 框架却未能很好的实现这一点，毕竟它们不是为了领域驱动设计而生的。以 Hibernate 为例，软件开发人员在使用它时，需要借助一门类似于 SQL 语言的 Hibernate 专用语言 HQL 语言，这就把软件开发人员从领域模型中拖回了冷冰冰的计算机世界，不论是 SQL、HQL 或是 Java、C#，都是僵硬的代码，而非有生命的领域对象。领域驱动设计，目的是让软件开发人员感觉是在真实的领域中工作一般，使用领域提供的操作而非自己的输入来设计软件，这样才能使得软件开发人员集中精力投入到需求设计中去。

2.3.7 模式间的关联

6 种领域模型元素的基本模式中，实体、值对象、聚合构成了领域模型的数据框架，服务协调领域对象之间的相互作用，工厂和仓储则负责管理领域对象的生命周期。领域模型元素基本模式的关系如图 2.3 所示：

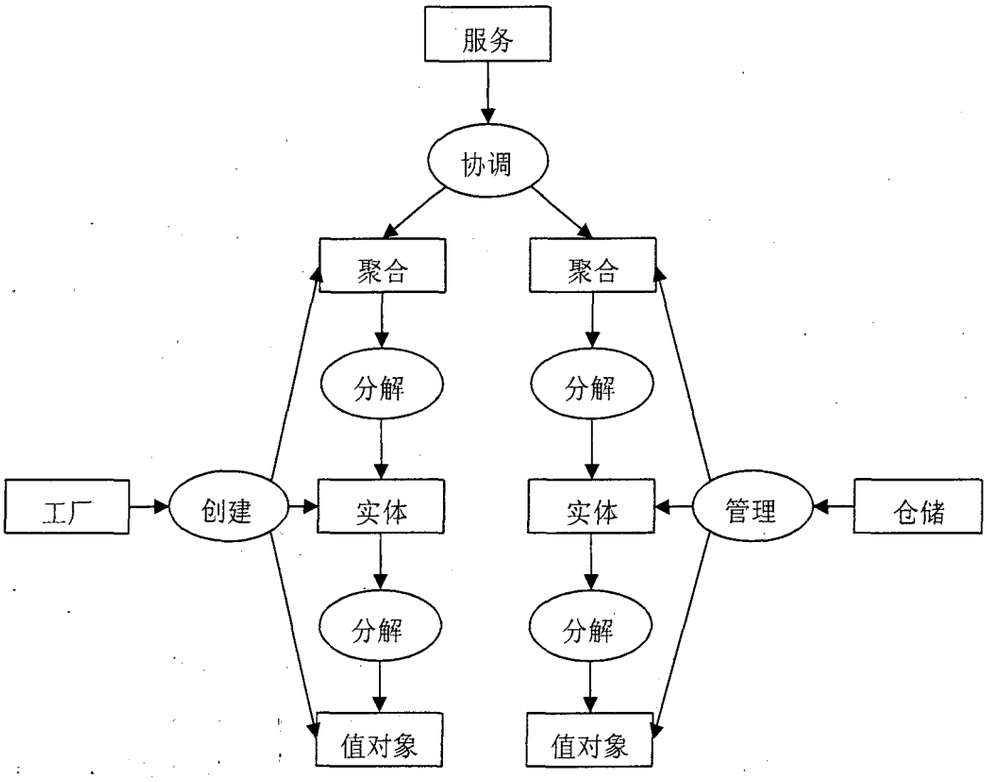

图 2.3 领域模型元素基本模式的关系

2.4 本章小结

本章对领域驱动设计理论做了详细分析，并对其中一些容易被理解错误的概念做了详细解释。领域驱动设计是一种抽象思想，在抽象思想基础上进行的扩展是多种多样的，我们应该充分理解思想的精髓，明白思想真正要表达的内容是什么，这样才能发掘更深层次的理念。本章从本质上阐述了领域驱动设计的基本理论、分层架构以及模型元素的模式，并引入层次结构从宏观角度给出了基本模式之间的关联。本章为下一章提出具体的领域驱动设计框架打下了基础。

第 3 章 领域驱动设计框架

领域驱动设计是一种思想，思想在实践中有多种实现方法，下面提出一种轻量级领域驱动设计框架，该框架对系统架构、领域模型、数据访问模块以及数据库表结构的设计方法都作出了详细规定，并提供了数据访问模块的具体实现。框架建立在.NET 平台上，用到的示例语言为 C#语言（C#语言介绍详见参考文献[36]），持久化介质为关系数据库。

3.1 框架总体结构

设计框架总体结构采用领域驱动设计的分层架构，但不同的是，该结构严格遵循分层架构的基本原则，不允许用户界面层越过应用层直接接触领域层。由于领域层是领域驱动设计的核心，而其它层经常在各类分层架构中出现，已经有了比较成熟的设计方法，因此，该框架主要讨论领域层以及与领域层密切相关的数据访问模块的设计。领域层是根据不同实际领域，利用统一编码规则实现的动态结构，又可分为聚合模块、工厂模块和仓储模块三部分。数据访问模块属于基础结构层，是封装好的静态模块。框架总体结构如图 3.1 所示：

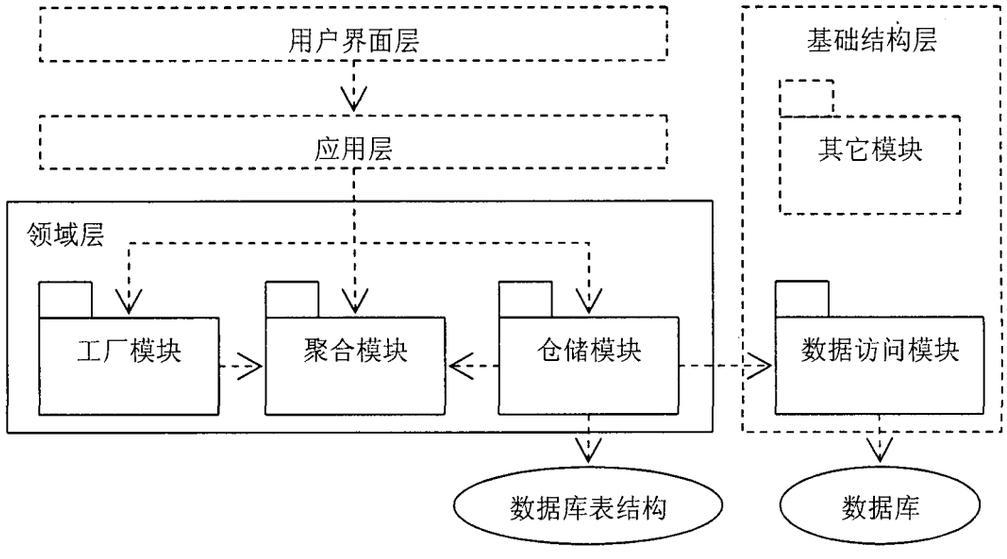

图 3.1 框架总体结构

3.2 数据库表结构设计

目前我们所使用的关系数据库存储的是二维表，利用关系数据库存储对象需要将对象的属性拆分成数个二维表。人们通常习惯凭借感性认识来拆分对象，毕竟大多数时候对象并不复杂，况且也没有一个介绍如何将对象的属性拆分成二维表的统一标准。事实上，如果能将这一行为规范化，对提高整个系统结构的条理性是非常有帮助的。

以一个典型类 EntityClass 为例，如图 3.2 所示：

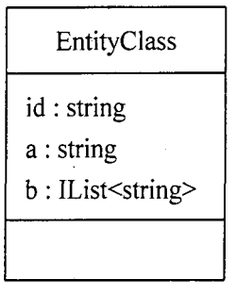

图 3.2 EntityClass 类的结构

id 为基本类型，是 EntityClass 类的标识，a 也为基本类型，被称为 EntityClass 类的固定成员，b 为集合类型，被称为 EntityClass 类的可变成员集合。类和类的固定成员之间是 1 对 1 的关系，类和类的可变成员之间是 1 对多的关系，如果用一张二维表来存储 EntityClass 类，则会得到形如表 3.1 所示的表：

表 3.1 存储 EntityClass 类的二维表

ID	A	B
1	1	1
1	1	2
2	2	3
2	2	4

从表中可以看出，类的固定成员被重复存储了多次。这说明一张二维表无法很好的表达 1 对 1 与 1 对多两种关系，实际上，一张关系表原本就是针对表达 1 种关系设计的，2 种关系必须使用 2 张关系表来实现。作为改进，我们接下来使用表 3.2-表 3.3 两张表来存储 EntityClass 类：

表 3.2 存储 EntityClass 类固定成员的二维表

ID	A
1	1
2	2

表 3.3 存储 EntityClass 类可变成员集合 b 的二维表

ID	B
1	1
1	2
2	3
2	4

经过改进之后的存储方式将类的固定成员和可变成员分成 2 张表存储，消除了之前产生的冗余，并且每增加一种可变成员集合，只需要再建立一张新表来存储新的对应关系即可。这种存储方式方便且易于理解，也是大多数情况下开发人员的选择。

但是，这种存储方式依然存在不足。让我们来看这样一种情况，假设在 EntityClass 类中删去 a，则数据库中对 EntityClass 类的存储仅需要一张用来存储 EntityClass 类的可变成员集合 b 的二维表，如果某个 EntityClass 类的实例的可变成员集合 b 在某一时刻都没有任何成员，是空集合，那么在这两张二维表中将没有该对象的任何信息，这就意味着对象从数据库中消失了。显然，在这种情况下，上述存储方式无法满足存储要求。为了应对这种情况，可以在上述存储方式基础上，再为每个类新建一个名单表，记录所有对象的 id，无论对象的属性如何，都能根据名单表找到相应对象。EntityClass 类的名单表如表 3.4 所示：

表 3.4 存储 EntityClass 类对象名单的二维表

ID
1
2

至此，类已经成功的分解成了二维表。每个类在数据库中都由 1 个储存类对象名单的二维表、1 个储存类固定成员的二维表（如果该类拥有固定成员）和若干个储存类可变成员集合的二维表（每一种可变成员集合一张表）组成。

自定义的值对象在它的成员表中存储它所包含的值或引用的值对象的 id，实体在它的成员表中存储它所包含的值或自定义的值对象的 id，聚合在它的成员表中存储它所包含的实体的 id。另外，对于不是位于最顶层的领域对象来说，可以不需要名单表，因为所有领域对象都会储存自己成员的名单。

3.3 数据访问模块设计

数据访问模块用于屏蔽数据库之间的差异，与具体的需求无关，因此可以采用通用的 ORM 框架。这里使用一个本人设计的简单模块，它可以分为四个子模块：参数模块、信息模块、转换模块和执行模块。四个模块之间的关系如图 3.3 所示：

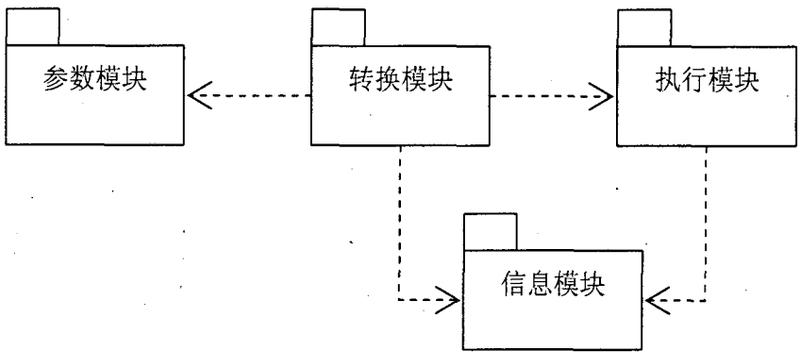

图 3.3 参数模块、信息模块、转换模块和执行模块的关系

3.3.1 参数模块

CRUD（Create、Retrieve、Update、Delete，即增加、查询、更新、删除）是 SQL 语句的基本操作，所有针对表数据的命令都是通过这 4 类操作完成的。在数据库中执行这些命令需要运行相应的 Insert、Select、Update、Delete 语句，这些语句在不同的数据库中写法不一，导致开发人员不仅需要了解数据库的结构，甚至还需要了解不同数据库的语句写法以使软件可以在不同数据库之间移植。参数模块的作用即是屏蔽数据库的结构与不同数据库之间的差异，以参数的形式提供统一的接口，使程序和具体数据库完全解耦，大大减轻了开发人员的负担，也降低了开发人员手动书写 SQL 语句产成错误的几率。

参数模块一共包含 4 个参数类，分别对应 Insert、Select、Update、Delete 这 4 类操作语句，另外还设置了 4 个辅助类用来帮助填充参数类。参数模块的结构如图 3.4 所示：

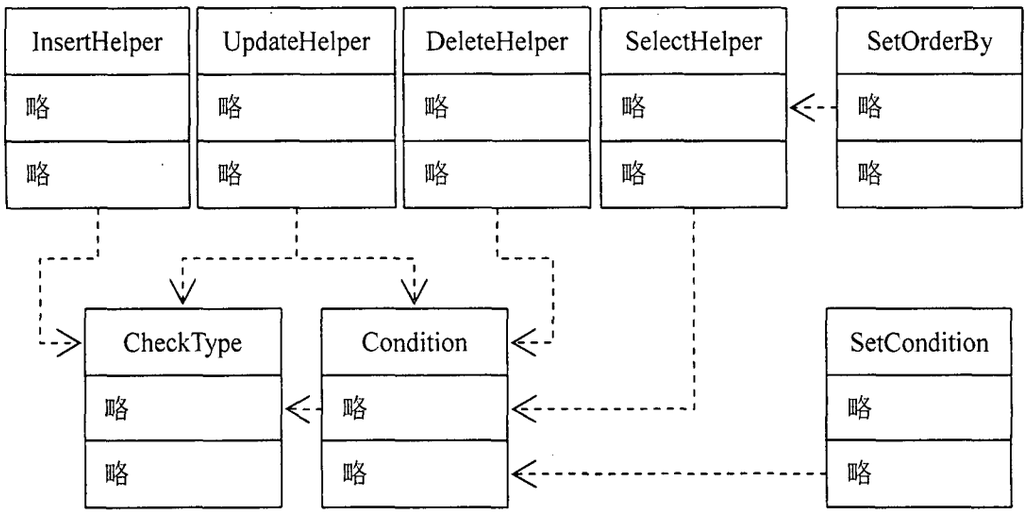

图 3.4 参数模块结构

- CheckType 类：CheckType 类是一个服务类，作用是判断输入值是否是数据库表中允许直接存储的值。非基本数据类型，如用户自定义的类等，显然不能直接存储在数据库的表中，需要进行转换。CheckType 类的介绍如表 3.5 所示：

表 3.5 CheckType 类的方法介绍

方法	bool IsBasicType (object value)
作用	若 value 为基本数据类型（如 int、string 等），则返回 true，否则返回 false。

- Condition 类：Condition 类是一个辅助参数类，它以参数的形式存储用户进行表操作时对表中数据的要求，如 SQL 语句中 WHERE 或 JOIN…ON 之后的条件。Condition 类的介绍如表 3.6-表 3.7 所示：

表 3.6 Condition 类的字段介绍

字段	IDictionary<string, object> atLeast
意义	表操作条件集合，其中每一个键值对存储一个“表字段>=给定值”的条件，键存储表字段，值存储给定值。
字段	IDictionary<string, object> atMost

意义	表操作条件集合，其中每一个键值对存储一个“表字段<=给定值”的条件，键存储表字段，值存储给定值。
字段	IDictionary<string, object> lessThan
意义	表操作条件集合，其中每一个键值对存储一个“表字段<给定值”的条件，键存储表字段，值存储给定值。
字段	IDictionary<string, object> moreThan
意义	表操作条件集合，其中每一个键值对存储一个“表字段>给定值”的条件，键存储表字段，值存储给定值。
字段	IDictionary<string, object> equalTo
意义	表操作条件集合，其中每一个键值对存储一个“表字段=给定值”的条件，键存储表字段，值存储给定值。
字段	IDictionary<string, object> notEqualTo
意义	表操作条件集合，其中每一个键值对存储一个“表字段<>给定值”的条件，键存储表字段，值存储给定值。
字段	IDictionary<string, object> contain
意义	表操作条件集合，其中每一个键值对存储一个“表字段 LIKE 给定值”的条件，键存储表字段，值存储给定值。
字段	IDictionary<string, object> notContain
意义	表操作条件集合，其中每一个键值对存储一个“表字段 NOT LIKE 给定值”的条件，键存储表字段，值存储给定值。
字段	IDictionary<string, IList<object>> among
意义	表操作条件集合，其中每一个键值对存储一个“表字段 IN 给定值集合”的条件，键存储表字段，值存储给定值集合。
字段	IDictionary<string, IList<object>> notAmong
意义	表操作条件集合，其中每一个键值对存储一个“表字段 NOT IN 给定值集合”的条件，键存储表字段，值存储给定值集合。
字段	IDictionary<string, IList<SelectHelper>> amongSubSelectHelper
意义	表操作条件集合，其中每一个键值对存储一个“表字段 IN 给定值集合”的条件，键存储表字段，值存储给定值集合。这里的给定值集合为子查询结果。
字段	IDictionary<string, IList<SelectHelper>> notAmongSubSelectHelper
意义	表操作条件集合，其中每一个键值对存储一个“表字段 NOT IN 给定值集合”的条件，键存储表字段，值存储给定值集合。这里的给定值集合为子查询结果。

字段	bool isSet
意义	标识是否设置了表操作条件，若所有表操作条件集合都为空集合，则为 false，否则为 true。

表 3.7 Condition 类的方法介绍

方法	void FieldAtLeast (string field, object value)
作用	若 value 为基本数据类型，则以 field 为表字段，value 为给定值，填充字段 atLeast，并置字段 isSet 为 true。
方法	void FieldAtMost (string field, object value)
作用	若 value 为基本数据类型，则以 field 为表字段，value 为给定值，填充字段 atMost，并置字段 isSet 为 true。
方法	void FieldLessThan (string field, object value)
作用	若 value 为基本数据类型，则以 field 为表字段，value 为给定值，填充字段 lessThan，并置字段 isSet 为 true。
方法	void FieldMoreThan (string field, object value)
作用	若 value 为基本数据类型，则以 field 为表字段，value 为给定值，填充字段 moreThan，并置字段 isSet 为 true。
方法	void FieldEqualTo (string field, object value)
作用	若 value 为基本数据类型，则以 field 为表字段，value 为给定值，填充字段 equalTo，并置字段 isSet 为 true。
方法	void FieldNotEqualTo (string field, object value)
作用	若 value 为基本数据类型，则以 field 为表字段，value 为给定值，填充字段 notEqualTo，并置字段 isSet 为 true。
方法	void FieldContain (string field, object value)
作用	若 value 为基本数据类型，则以 field 为表字段，value 为给定值，填充字段 contain，并置字段 isSet 为 true。
方法	void FieldNotContain (string field, object value)
作用	若 value 为基本数据类型，则以 field 为表字段，value 为给定值，填充字段 notContain，并置字段 isSet 为 true。
方法	void FieldAmong (string field, IList<object> value)
作用	若 value 为基本数据类型集合，则以 field 为表字段，value 为给定值集合，填充字段 among，并置字段 isSet 为 true。

方法	void FieldNotAmong (string field, IList<object> value)
作用	若 value 为基本数据类型集合，则以 field 为表字段，value 为给定值集合，填充字段 notAmong，并置字段 isSet 为 true。

- SetCondition 类：SetCondition 类是一个辅助参数类，它为用户提供一个统一的填充 Condition 类实例的接口。SetCondition 类的介绍如表 3.8-表 3.9 所示：

表 3.8 SetCondition 类的字段介绍

字段	Condition condition
意义	Condition 类的实例引用，指向需要填充的 Condition 类的实例。
字段	string field
意义	填充 condition 时用到的表字段。

表 3.9 SetCondition 类的方法介绍

方法	void AtLeast (object value)
作用	以 field 和 value 为参数，调用 condition 的 FieldAtLeast 方法填充 condition。
方法	void AtMost (object value)
作用	以 field 和 value 为参数，调用 condition 的 FieldAtMost 方法填充 condition。
方法	void LessThan (object value)
作用	以 field 和 value 为参数，调用 condition 的 FieldLessThan 方法填充 condition。
方法	void MoreThan (object value)
作用	以 field 和 value 为参数，调用 condition 的 FieldMoreThan 方法填充 condition。
方法	void EqualTo (object value)
作用	以 field 和 value 为参数，调用 condition 的 FieldEqualTo 方法填充 condition。
方法	void NotEqualTo (object value)
作用	以 field 和 value 为参数，调用 condition 的 FieldNotEqualTo 方法填充 condition。
方法	void Contain (object value)
作用	以 field 和 value 为参数，调用 condition 的 FieldContain 方法填充 condition。
方法	void NotContain (object value)
作用	以 field 和 value 为参数，调用 condition 的 FieldNotContain 方法填充 condition。
方法	void Among (IList<object> value)
作用	以 field 和 value 为参数，调用 condition 的 FieldAmong 方法填充 condition。

方法	void NotAmong (IList<object> value)
作用	以 field 和 value 为参数，调用 condition 的 FieldNotAmong 方法填充 condition。

- InsertHelper 类：InsertHelper 类是一个参数类，它存储执行一个 Insert 操作需要的所有参数。InsertHelper 类的介绍如表 3.10-表 3.11 所示：

表 3.10 InsertHelper 类的字段介绍

字段	string tableName
意义	执行 Insert 操作的表名。
字段	IDictionary<string, object> fieldValue
意义	执行 Insert 操作需要的表字段及其给定值集合，其中每一个键值对存储一个表字段及其给定值，键存储表字段，值存储给定值。

表 3.11 InsertHelper 类的方法介绍

方法	void SetFieldValue (string field, object value)
作用	若 value 为基本数据类型，则以 field 为表字段，value 为给定值，填充 fieldValue。

- DeleteHelper 类：DeleteHelper 类是一个参数类，它存储执行一个 Delete 操作需要的所有参数。DeleteHelper 类的介绍如表 3.12 所示：

表 3.12 DeleteHelper 类的字段介绍

字段	string tableName
意义	执行 Delete 操作的表名。
字段	Condition fieldCondition
意义	执行 Delete 操作的条件。

- UpdateHelper 类：UpdateHelper 类是一个参数类，它存储执行一个 Update 操作需要的所有参数。UpdateHelper 类的介绍如表 3.13-表 3.14 所示：

表 3.13 UpdateHelper 类的字段介绍

字段	string tableName
意义	执行 Update 操作的表名。
字段	IDictionary<string, object> newFieldValue
意义	执行 Update 操作需要的表字段及其新值集合，其中每一个键值对存储一个表字段及其新值，键存储表字段，值存储新值。
字段	Condition fieldCondition

意义	执行 Update 操作的条件。

表 3.14 UpdateHelper 类的方法介绍

方法	void SetNewFieldValue (string field, object newValue)
作用	若 newValue 为基本数据类型，则以 field 为表字段，newValue 为新值，填充 newFieldValue。

- SelectHelper 类：SelectHelper 类是一个参数类，它存储执行一个 Select 操作需要的所有参数。SelectHelper 类的介绍如表 3.15-表 3.16 所示：

表 3.15 SelectHelper 类的字段介绍

字段	string tableName
意义	执行 Select 操作的表名。
字段	string outJoinField
意义	执行表连接操作时的外表字段。
字段	string selfJoinField
意义	执行表连接操作时的本表字段。
字段	IDictionary<string, bool> outputField
意义	执行 Select 操作需要输出的表字段集合，其中每一个键值对存储一个表字段，键存储表字段，值标识是否需要去重复值。
字段	Condition fieldCondition
意义	执行 Select 操作的条件。
字段	IDictionary<string, bool> orderBy
意义	执行 Select 操作需要依次进行排序的表字段集合，其中每一个键值对存储一个表字段，键存储表字段，值标识是否按升序排序。
字段	IList<string> groupBy
意义	执行 Select 操作需要分组的表字段集合。
字段	IDictionary<string, SelectHelper> joinSelectHeper
意义	执行 Select 操作需要连接的表的 Select 参数集合，其中每一个键值对存储一个表的 Select 参数，键存储表名，值存储表的 Select 参数。
字段	int maxNum
意义	执行 Select 操作需要输出的最大记录数。

表 3.16 SelectHelper 类的方法介绍

方法	void SetOutputField (string field, bool isDistinct)
作用	若 field 尚未要求输出，则以 field 为表字段，isDistinct 为标识，填充 outputField。
方法	void SetOrderBy (string field, bool isASC)
作用	若 field 尚未要求排序，则以 field 为表字段，isASC 为标识，填充 orderBy。
方法	void SetGroupBy (string field)
作用	若 field 尚未要求分组，则以 field 为表字段，填充 groupBy。
方法	void SetMaxNum (int maxNum)
作用	设置 maxNum。
方法	void SetJoinSelectHelper ()
作用	设置 joinSelectHelper。

- SetOrderBy 类：SetOrderBy 类是一个辅助参数类，它为用户提供一个统一的填充 SelectHelper 类 orderBy 字段的接口。SetOrderBy 类的介绍如表 3.17-表 3.18 所示：

表 3.17 SetOrderBy 类的字段

字段	SelectHelper selectHelperList
意义	执行 Select 操作的表的 Select 参数。
字段	SelectHelper selectHelperField
意义	需要排序的表字段所在的表的 Select 参数。
字段	string field
意义	需要排序的表字段。

表 3.18 SetOrderBy 类的方法

方法	void Asc ()
作用	以 field 为表字段，true 为标识，调用 selectHelperList 的 SetOrderBy 方法。以 field 为表字段，false 为标识，调用 selectHelperList 的 SetOutputField 方法。置 selectHelperField 的 isSet 字段为 true。
方法	void Desc ()
作用	以 field 为表字段，false 为标识，调用 selectHelperList 的 SetOrderBy 方法。以 field 为表字段，false 为标识，调用 selectHelperList 的 SetOutputField 方法。置 selectHelperField 的 isSet 字段为 true。

3.3.2 信息模块

信息模块包含一个数据类 DbInfo，它存储执行持久化操作需要了解的数据库的所有信息。DbInfo 类的介绍如表 3.19 所示：

表 3.19 DbInfo 类的字段介绍

字段	string connstr
意义	数据库连接字符串，从配置文件中读取。
字段	string dbType
意义	数据库类型，如 SqlServer、Oracle 等，从配置文件中读取。

3.3.3 转换模块

转换模块包含一个服务类 FABHelper，它负责解析参数模块中的参数类包含的信息，并根据所使用的数据库类型，将得到的信息组装成参数化的 SQL 语句，调用执行模块相应服务执行。转换模块类的介绍如表 3.20 所示：

表 3.20 FABHelper 类的方法介绍

方法	DbCommand CreateCommand ()
作用	根据信息模块中的信息创建 DbCommand 对象。
方法	DbParameter CreateParameter ()
作用	根据信息模块中的信息创建 DbParameter 对象。
方法	string CreateParameterName (int parameterID)
作用	根据信息模块中的信息与 parameterID 创建参数化 SQL 语句中的占位符。
方法	string PrepareCondition (DbCommand cmd, string tableName, Condition condition)
作用	根据信息模块中的信息，将 condition 中包含的条件信息转化为参数化 SQL 语句。
方法	string PrepareTableName (DbCommand cmd, SelectHelper selectHelper)
作用	根据信息模块中的信息将 selectHelper 中包含的子查询信息转化为参数化 SQL 语句。
方法	DbCommand CreateInsertCommand (InsertHelper insertHelper)
作用	根据 insertHelper 中的信息创建执行该 Insert 操作的 DbCommand 对象。
方法	DbCommand CreateDeleteCommand (DeleteHelper deleteHelper)
作用	根据 deleteHelper 中的信息创建执行该 Delete 操作的 DbCommand 对象。
方法	DbCommand CreateUpdateCommand (UpdateHelper updateHelper)

作用	根据 updateHelper 中的信息创建执行该 Update 操作的 DbCommand 对象。
方法	DbCommand CreateSelectCommand (SelectHelper selectHelper)
作用	根据 selectHelper 中的信息创建执行该 Select 操作的 DbCommand 对象。
方法	IDictionary<string, int> Insert (IDictionary<string, InsertHelper> insertHelpers)
作用	根据 insertHelpers 中存储的 Insert 参数，依次调用执行模块相应服务执行 Insert 操作，并返回执行结果。
方法	IDictionary<string, int> Delete (IDictionary<string, DeleteHelper> deleteHelpers)
作用	根据 deleteHelpers 中存储的 Delete 参数，依次调用执行模块相应服务执行 Delete 操作，并返回执行结果。
方法	IDictionary<string, int> Update (IDictionary<string, UpdateHelper> updateHelpers)
作用	根据 updateHelpers 中存储的 Update 参数，依次调用执行模块相应服务执行 Update 操作，并返回执行结果。
方法	DataSet Select (IDictionary<string, SelectHelper> selectHelpers)
作用	根据 selectHelpers 中存储的 Select 参数，依次调用执行模块相应服务执行 Select 操作，并返回执行结果。

3.3.4 执行模块

执行模块包含一个服务类 EXEHelper，它负责执行 DbCommand 对象蕴涵的操作。执行模块类的介绍如表 3.21 所示：

表 3.21 EXEHelper 类的方法

方法	DbConnection CreateConnection ()
作用	根据信息模块中的信息创建 DbConnection 对象。
方法	DbDataAdapter CreateDataAdapter ()
作用	根据信息模块中的信息创建 DbDataAdapter 对象。
方法	void PrepareCommand (DbCommand cmd)
作用	打开 cmd 的数据连接。
方法	IDictionary<string, int> ExecuteNonQuery (IDictionary<string, DbCommand> cmds)
作用	依次执行 cmds 中包含的 DbCommand 对象蕴涵的操作，并返回执行结果。DbCommand 对象蕴涵的操作可为 Insert、Delete 或 Update。
方法	DataSet ExecuteDataSet (IDictionary<string, DbCommand> cmds)

作用	依次执行 cmds 中包含的 DbCommand 对象蕴涵的操作，并返回执行结果。DbCommand 对象蕴涵的操作只能为 Select 的非统计操作。
方法	IDictionary<string, object> ExecuteScalar (IDictionary<string, DbCommand> cmds)
作用	依次执行 cmds 中包含的 DbCommand 对象蕴涵的操作，并返回执行结果。DbCommand 对象蕴涵的操作只能为 Select 的统计操作。

3.4 领域层设计

3.4.1 实体设计

实体设计是领域设计中最基本、最常用、也最容易弄错的一环。习惯于传统软件设计的开发人员会很自然的将实体与类等同起来，用设计类的方式去设计实体。诚然，在软件设计中，实体确实是由类来表现的，实体的设计也是通过类的设计来实现的，但是，实体的设计仅仅是类设计的子集，施加在实体设计上的限制要比施加在类设计上的限制多得多。根据领域驱动设计理论，实体代表的事物是值对象的集合，实体的方法是跨越它所包含的值对象的操作。这就意味着实体不能访问其它实体，实体与实体之间是严格独立的，互相之间无法通信，也根本无需知道对方的存在，实体是一个严格封闭的集合。实体能进行的操作，仅仅局限于协调自己内部的值对象，对外界一无所知，这样做的好处是，大大增强了实体的灵活性，使得实体与实体之间是一种“零耦合”状态，实体可以任意的修改而无需考虑其它实体的状态。实体的结构如图 3.5 所示：

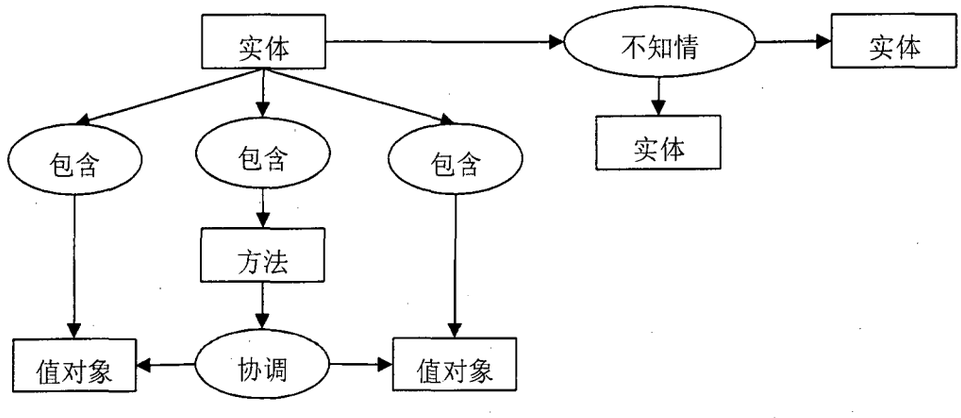

图 3.5 实体的结构

3.4.2 值对象设计

大多数情况下，值对象是不需要专门设计的。作为约定俗成的事物，程序设计语言通常提供了丰富的值对象供开发人员使用，但在某些情况下，开发人员依然需要创建适合自己领域的值对象。为了区别二者，我们将程序设计语言提供的值对象称为值，开发人员自己定义的值对象依然称为值对象。

值对象位于领域结构的最底层，它只能包含值或其它值对象。由于值对象代表的是一种约定，只需要按照现实领域的规则设计即可，灵活性不高，并且在使用过程中也只能调用而不允许更改，因此通常值对象的设计难度不大。利用类实现的值对象的结构如图 3.6 所示：

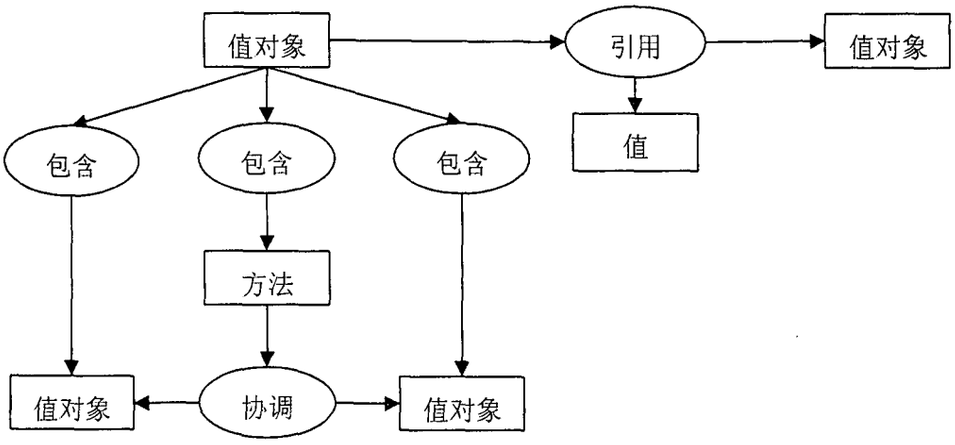

图 3.6 值对象的结构

3.4.3 聚合设计

聚合设计是领域设计的核心。领域驱动设计最重要的一点就是通过聚合建立起了立体的领域模型，降低了领域复杂度。聚合并不是一个单独的层次，它可以通过不断的重复聚集形成多层结构，以形成更为复杂的领域模型，这也是领域驱动设计的最大优点。与实体一样，聚合与聚合之间也是严格独立的，互不知情，互不通信。这种独立性使得聚合尽管体型庞大，却又不乏灵活。

根是聚合的组织核心，它持有聚合的标识，并且组织管理聚合包含的实体。在软件设计中，根也是通过类来实现。除了标识之外，根不包含任何值对象，仅包含聚合内所有实体的引用以及协调这些实体行为的方法。作为聚合与外界

通信的唯一渠道，根协调聚合内各实体的行为，保证聚合正常工作。根的标识和方法也可以看成聚合的标识和方法。聚合的结构如图 3.7 所示：

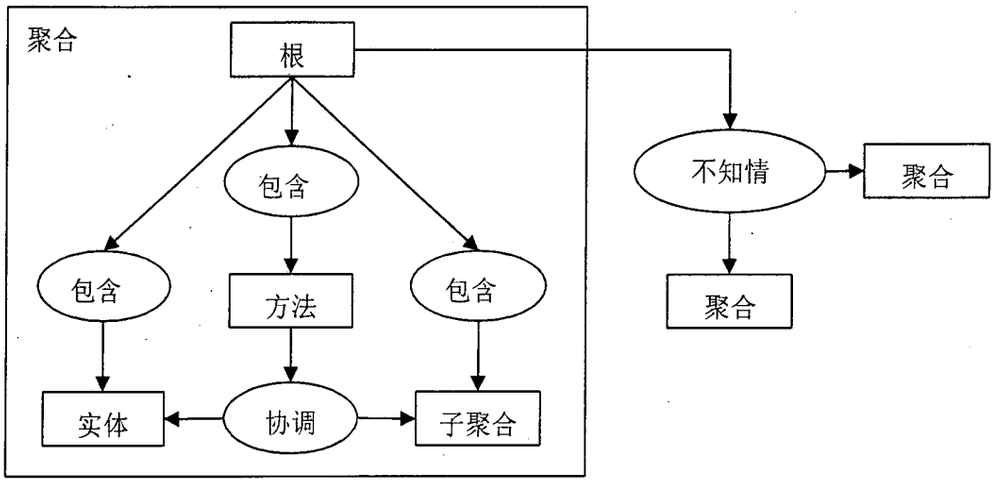

图 3.7 聚合的结构

3.4.4 服务设计

服务本质上是操作的集合。在多层次的聚合中，许多原本属于服务范畴的操作被划分为聚合的方法，使得服务的范围越来越窄，最终只包括跨越最高层聚合的操作和基础操作。实际上，如果把整个领域看成一个大的顶层聚合，那么领域服务就是这个顶层聚合的方法。

在面向对象语言中，服务也是通过类来实现。实现服务的类只有全局数据和方法，因此通常以单例模式（Singleton）的形式出现。服务的结构如图 3.8 所示：

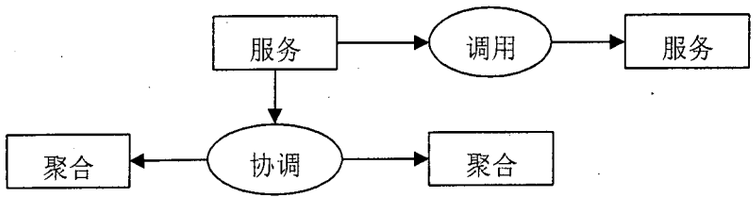

图 3.8 服务的结构

3.4.5 工厂设计

每一类领域对象都有自己的工厂。工厂以服务的形式出现，并且按照领域结构形成层次。通常情况下，值对象都是一次性生成，并不需要动态创建，因此工厂主要负责实体级别以上的领域对象的创建工作。位于底层的实体工厂只需要返回实体的引用，而位于高层的聚合工厂在返回根的引用之前，要先创建自己内部的实体，并将它们组装起来，位于更高层的聚合则需要创建并组装好自己内部的聚合之后才返回根的引用。由于采用层次结构，因此无论聚合有多少层，工厂都不需要了解自己内部成员的组装过程，仅仅需要创建及组装它们即可。工厂的结构如图 3.9 所示：

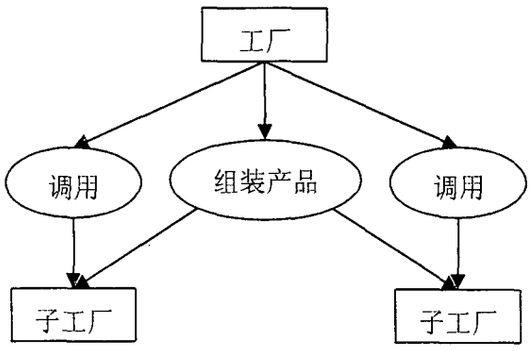

图 3.9 工厂的结构

3.4.6 仓储设计

仓储的设计比较复杂，包括领域对象的增、删、改、查四个方面。仓储是领域对象与持久化介质之间的接口，因此它对双方的信息都需要了解。每一类领域对象都拥有自己的仓储，仓储了解自己所属领域对象的所有数据信息以及它们在持久化介质中的存储格式，而持久化介质本身的信息则通常蕴含在公共模块中。仓储根据自己掌握的领域对象信息，调用公共的持久化模块完成仓储工作，例如在.NET 平台上写入 SQL 语句并调用 ADO.NET 模块完成数据库操作。

下面介绍一种基于上文介绍的数据访问模块的仓储的设计方案，该方案中

仓储需要实现 7 个功能，分别是添加领域对象、添加领域对象可变成员、删除领域对象可变成员、删除领域对象、修改领域对象可变成员、查询领域对象标识、查询领域对象信息。这 7 个功能分别由 7 个仓储模块实现，此外，还设置了一个数据类存储仓储需要了解的数据库结构信息。为了应对复杂的领域对象，仓储也采用层次结构，由上层仓储调用下层仓储完成仓储工作，具体的实现细节则由最底层仓储封装，上层仓储无需了解。仓储的结构如图 3.10 所示：

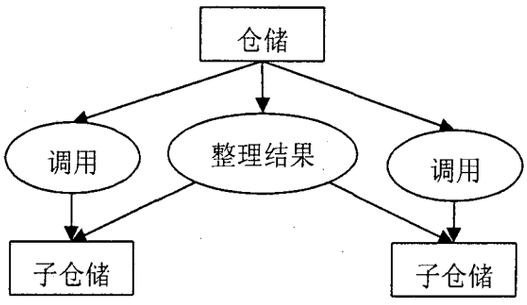

图 3.10 仓储的结构

- DbStruct 模块。DbStruct 模块负责存储领域对象在数据库中的存储结构，包括表以及表字段的信息。所有仓储类都需要引用 DbStruct 模块中的信息。DbStruct 模块只包含一个静态数据类 DbStruct，由于每一个具体领域对象的数据库表结构都是固定的，因此 DbStruct 类被设置成只读类。
- AddVarMember 模块。AddVarMember 模块负责将一个领域对象的可变成员插入数据库中。它包含数个子类 AddMember，每一个子类负责一个可变成员的插入操作。AddVarMember 模块的结构如图 3.11 所示：

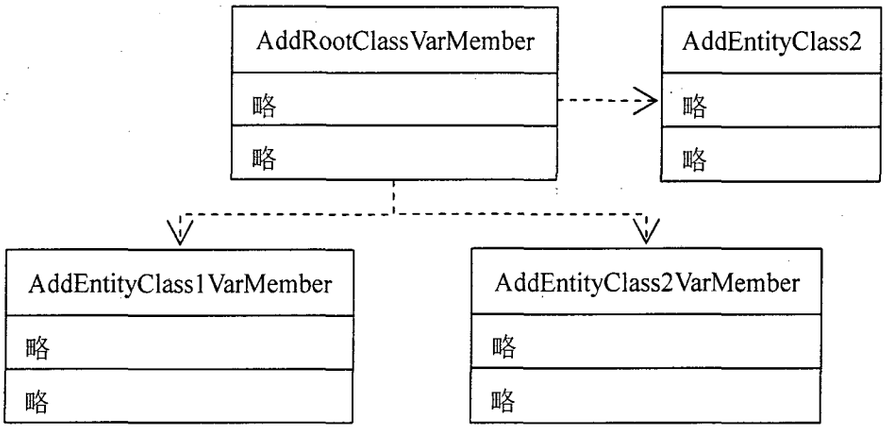

图 3.11 AddVarMember 模块的结构

- Add 模块。Add 模块负责将一个领域对象插入数据库中，它为开发人员提供一个统一的服务接口，使得开发人员无须关心需要插入的领域对象的类型与规模。Add 模块的结构如图 3.12 所示：

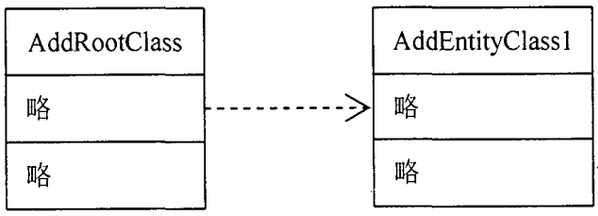

图 3.12 Add 模块的结构

- DelVarMember 模块。DelVarMember 模块负责将一个领域对象的可变成员从数据库中删除。它包含数个子类 DelMember，每一个子类负责一个可变成员的删除操作。DelVarMember 模块的结构如图 3.13 所示：

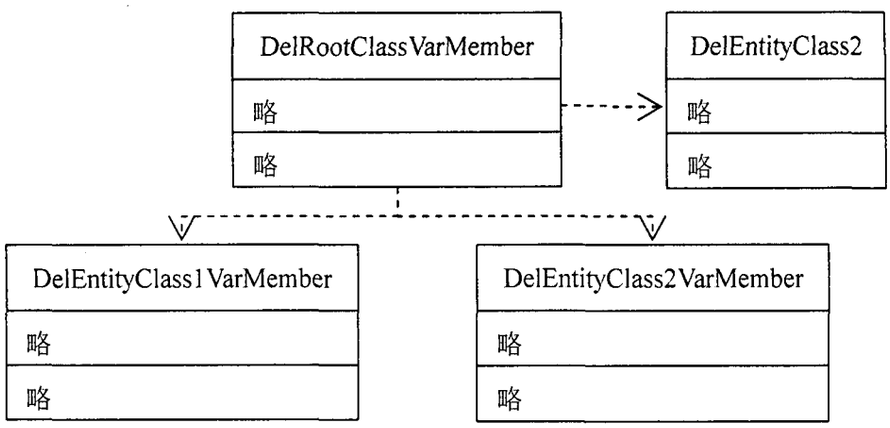

图 3.13 DelVarMember 模块的结构

- Del 模块。Del 模块负责将一个领域对象从数据库中删除，它为开发人员提供一个统一的服务接口，使得开发人员只需要了解领域对象的标识即可。Del 模块的结构如图 3.14 所示：

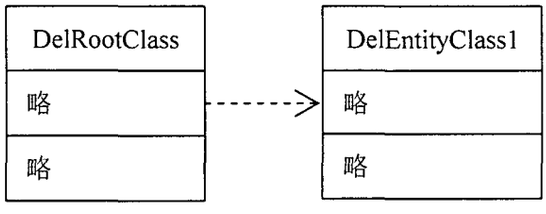

图 3.14 Del 模块的结构

- Update 模块。Update 模块负责更新领域对象在数据库中的信息，它以层次化的接口为开发人员提供服务。Update 模块的结构如图 3.15 所示：

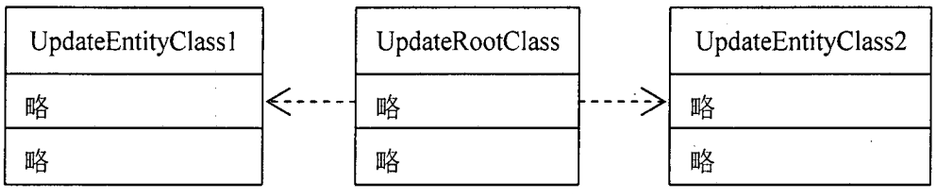

图 3.15 Update 模块的结构

- GetID 模块。GetID 模块负责在数据库中查询满足条件的领域对象，返回它们的标识。它以函数式的查询服务替代手工输入的查询代码，很好的向开发人员屏蔽了底层实现的细节，避免了手工输入可能产生的错误。GetID 模块包含 2 个子类 SetCondition 和 SetOrderBy，分别用于设置查询条件和排序方式。GetID 模块的结构如图 3.16 所示：

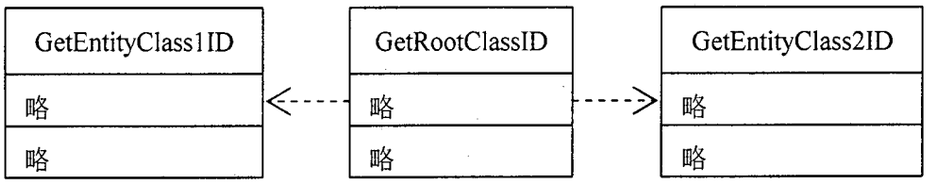

图 3.16 GetID 模块的结构

- Get 模块。Get 模块负责在数据库中根据标识查询领域对象，并返回它们的引用。它包含 1 个子类 SetOutput，用于设置需要输出的成员。Get 模块的结构如图 3.17 所示：

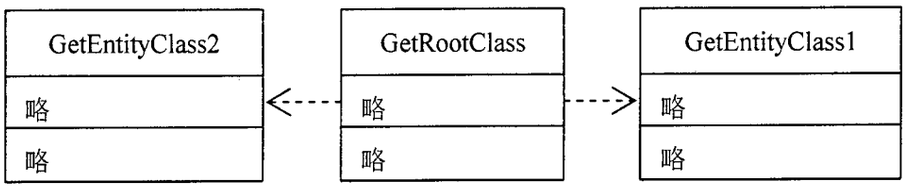

图 3.17 Get 模块的结构

3.5 本章小结

本章根据领域驱动设计理论，提出了具体的领域驱动设计方法，包括系统架构、领域模型、数据访问模块以及数据库表结构的设计方法，并在此基础上提出了一种轻量级领域驱动设计框架。该框架对本章讨论的设计方法都作出了详细规范，并提供了数据访问模块的具体实现。下一章将通过该框架介绍领域驱动设计方法在实际系统开发中的应用。

第 4 章 领域驱动设计方法的应用

为了展示领域驱动设计方法在具体项目中的应用，下面介绍一个使用上文提出的领域驱动设计框架开发的水泥厂的综合管理系统，该系统涵盖产品管理、销售管理、部门管理、员工管理、客户管理、财务管理、ID 卡管理等多方面内容，具有代表意义。

4.1 系统总体结构

根据需求信息，该系统所有领域对象可划分为 6 个聚合：ID 卡（IDCard）、产品（Product）、部门（Department）、客户（Customer） 、合同（Contract）、权限（Permission）。这 6 个聚合是可以单独存在的领域对象，其它所有领域对象都存在于这 6 个聚合之中。此外，领域服务（DomainService）负责协调各聚合间的通信。工厂和仓储依据聚合结构，按照框架规则实现。系统的总体结构如图 4.1 所示：

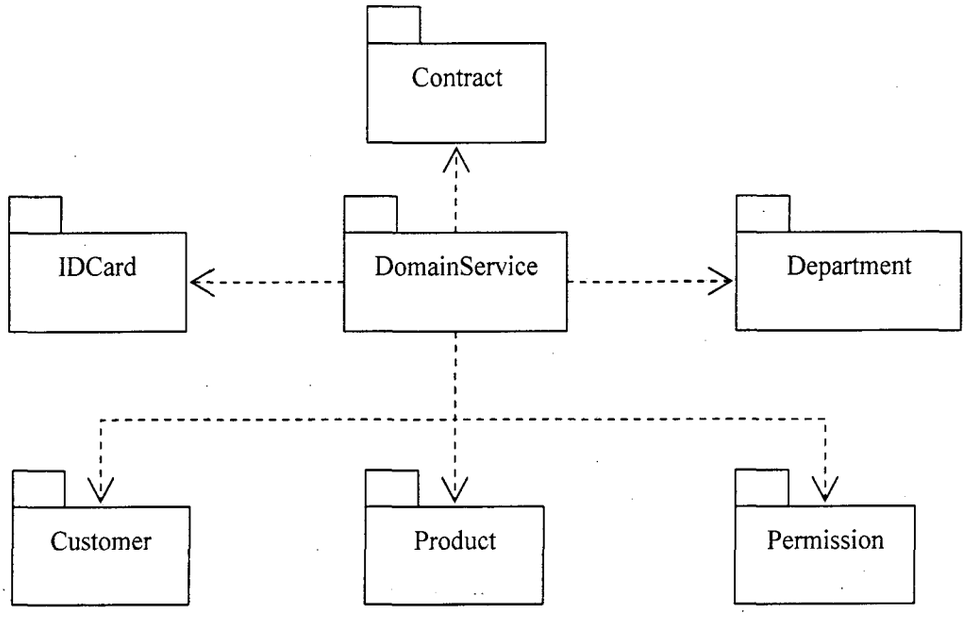

图 4.1 系统的总体结构

4.2 聚合模块

聚合是依据实体的独立性来划分的，以部门聚合为例，员工信息和登录信息是员工的一部分，当员工被删除后，员工信息和登录也一并被删除，因此，它们是员工聚合的一部分。每一个员工都有自己的职位，当某个职位取消时，该职位的员工要么转移到其它职位，要么离职，不存在没有职位的员工。因此，员工聚合应当作为职位聚合的子聚合。同理，职位也应当成为部门聚合的一部分，因此职位聚合是部门聚合的子聚合。所有聚合都依据此原则划分。下面展示部门聚合的结构，其它聚合的结构类似。部门聚合的结构如图 4.2 所示：

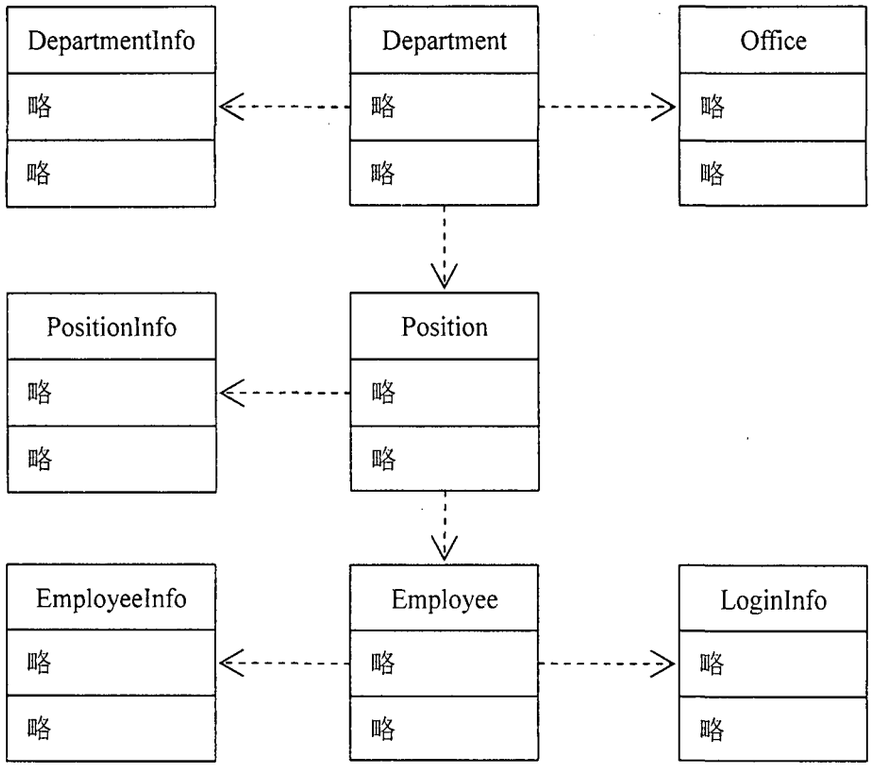

图 4.2 部门聚合的结构

所有实体都能由聚合导航而出，如 Department.Position.Employee.LoginInfo。聚合的层次结构使得实体间的联系更加清晰有条理。另外，为了有效进行权限管理，编程语言提供给各实体和聚合根的类创建操作（如 Java 或 C#的 new 操作符）将不对开发人员开放，开发人员对聚合只能引用，不能随意创建。聚合创建服务将统一由工厂提供。

4.3 工厂模块

工厂为开发人员提供聚合的创建服务，但作为聚合固定成员的子聚合和实体的工厂将不为开发人员所知。因为聚合的固定成员是与聚合同生命周期的，所以聚合固定成员的工厂只供聚合内部调用。工厂的结构按照聚合结构进行设计，下面展示部门工厂的结构，其它工厂的结构类似。部门工厂的结构如图 4.3 所示：

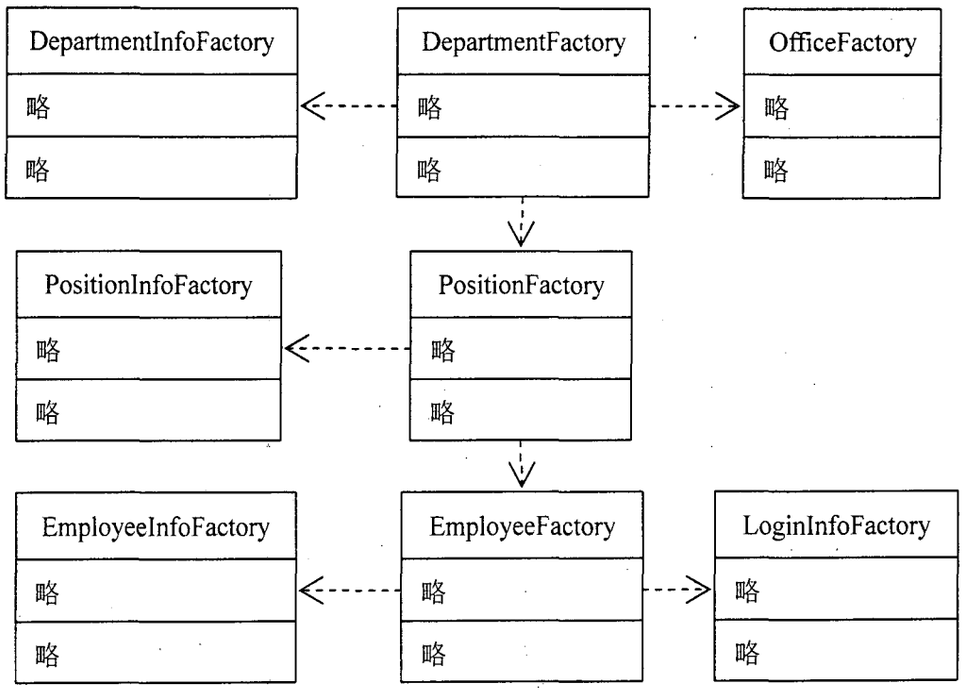

图 4.3 部门工厂的结构

开发人员可以使用工厂创建一个新的聚合或实体，该聚合或实体被创建时，将自动调用固定成员工厂创建自己的固定成员。可变成员将在聚合或实体创建后动态添加。

4.4 仓储模块

仓储负责聚合的持久化，每个聚合都有自己的仓储。需要注意的是，并非所有聚合都拥有全部仓储功能，根据需求的不同，有些聚合仅拥有部分仓储功能。例如没有可变成员的聚合不存在 AddVarMember 模块和 DelVarMember 模

块，只读聚合则仅拥有 GetID 模块和 Get 模块。仓储的结构按照聚合结构进行设计。下面展示部门仓储的结构，其它仓储的结构类似。部门仓储拥有所有模块，结构如图 4.4-图 4.10 所示：

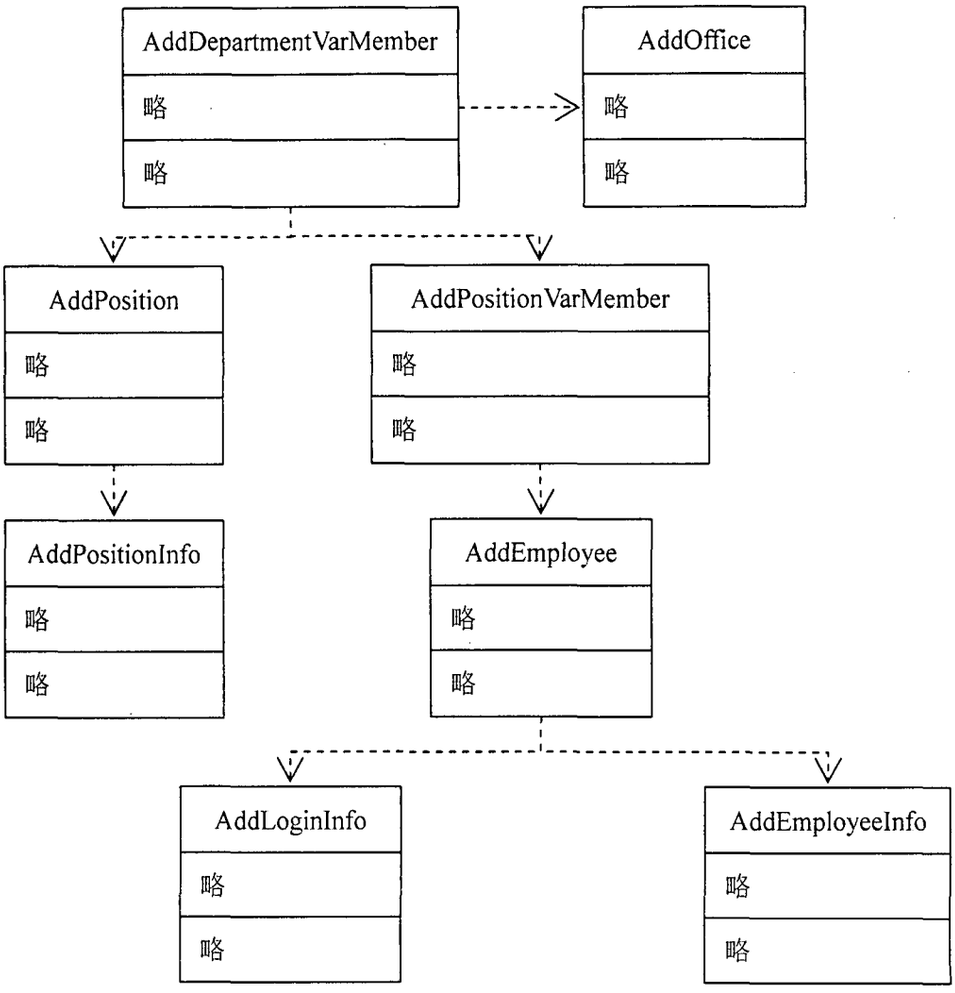

图 4.4 部门仓储的 AddVarMember 模块结构

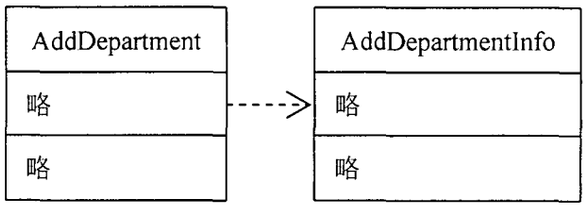

图 4.5 部门仓储的 Add 模块结构

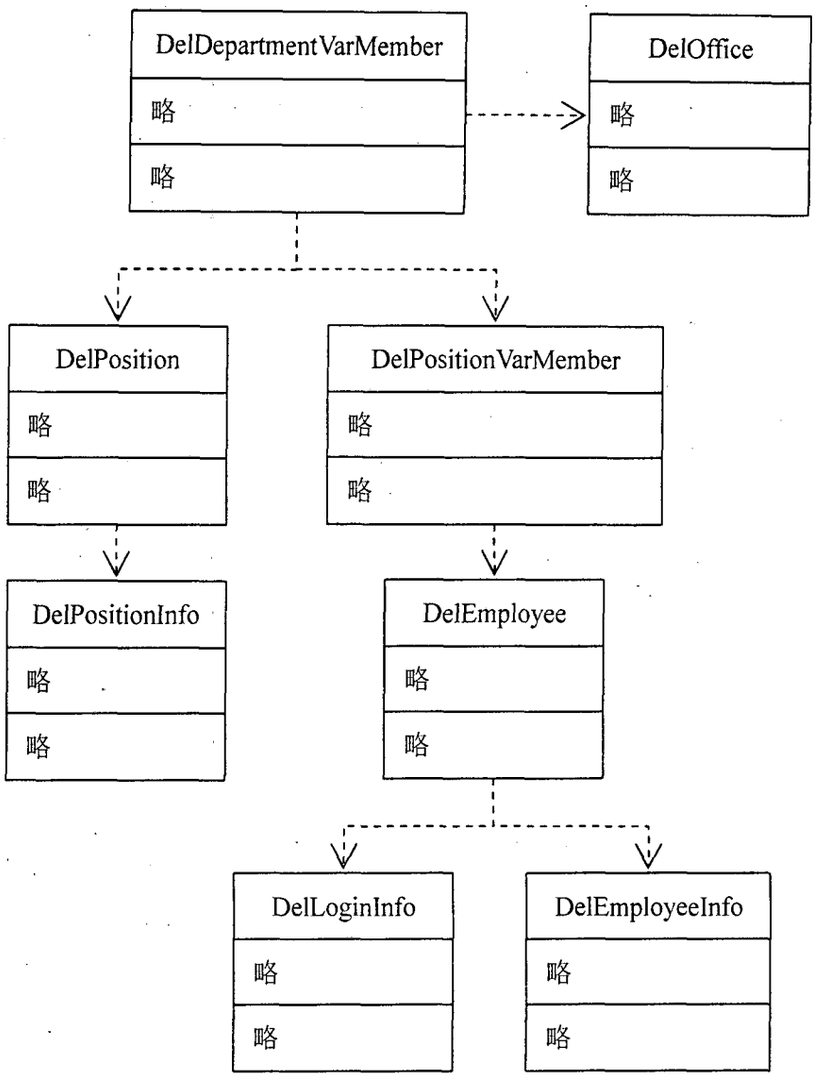

图 4.6 部门仓储的 DelVarMember 模块结构

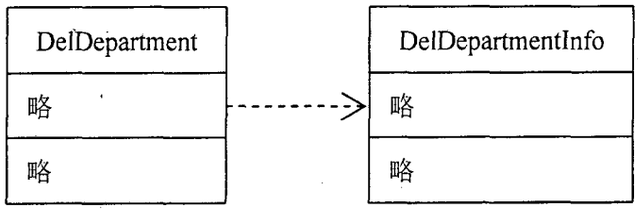

图 4.7 部门仓储的 Del 模块结构

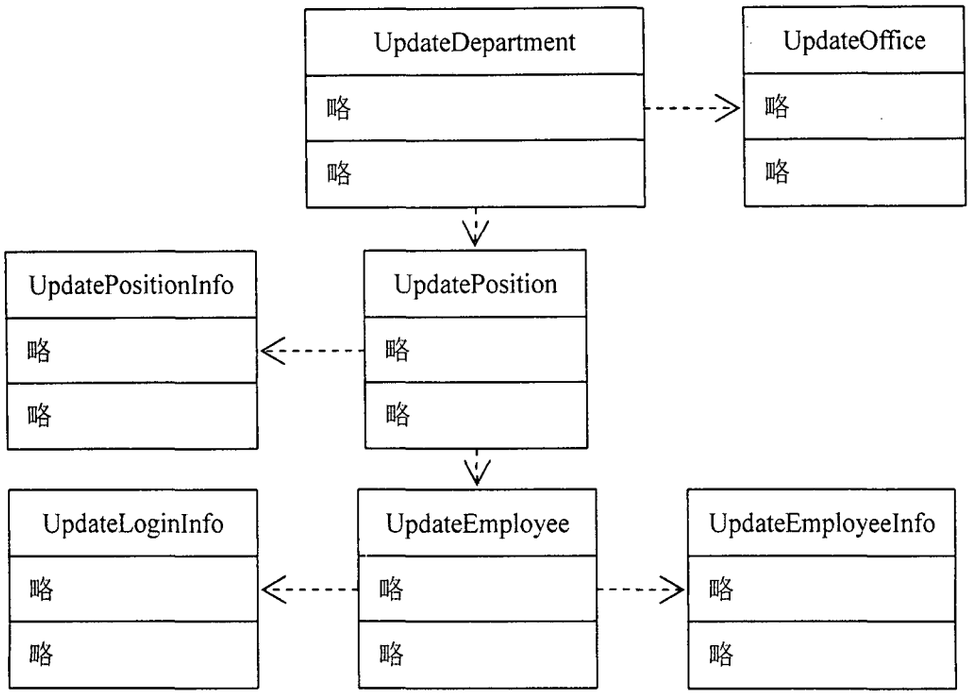

图 4.8 部门仓储的 Update 模块结构

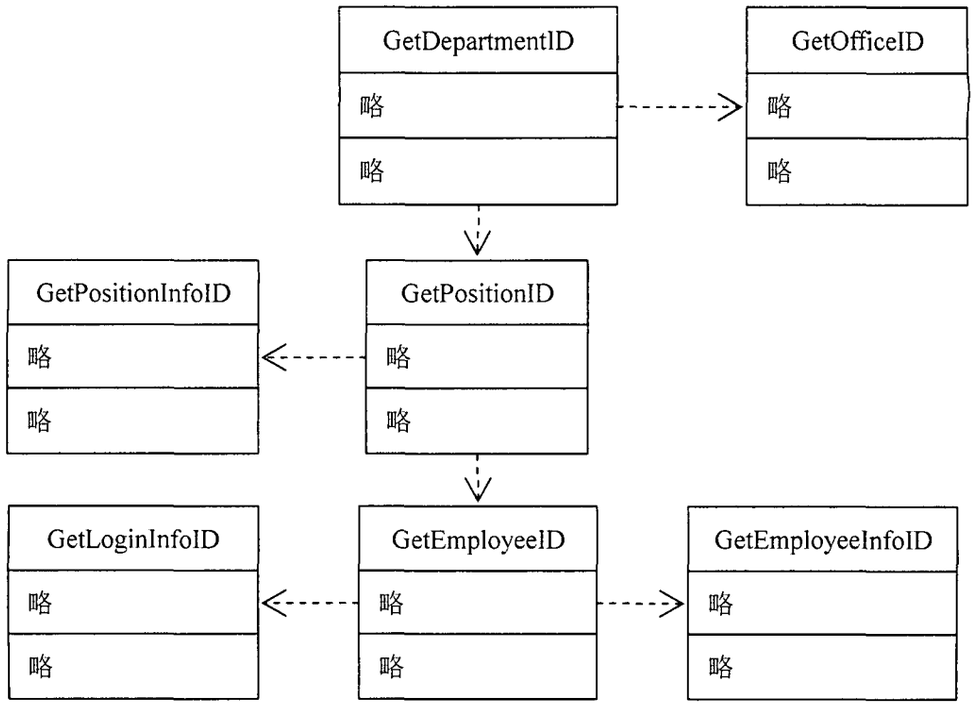

图 4.9 部门仓储的 GetID 模块结构

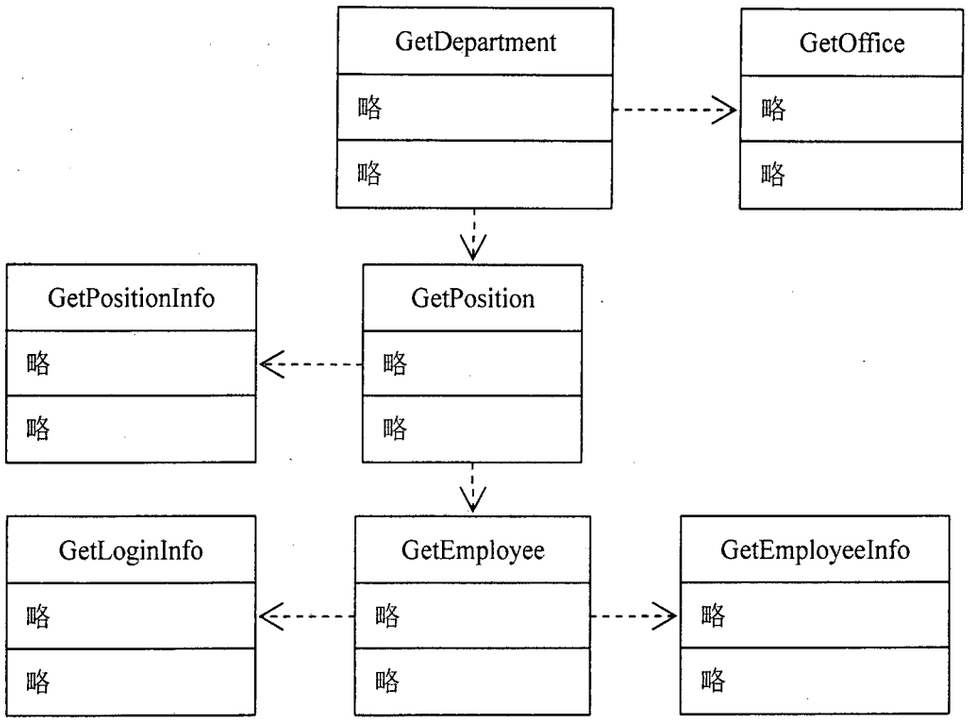

图 4.10 部门仓储的 Get 模块结构

7 个仓储模块涵盖了所有仓储功能，开发人员可以使用仓储提供的服务自由进行数据持久化操作。最为重要的是，仓储是依据聚合结构设计的，因此它提供服务的原则是“让开发人员选择，而非输入”，这是符合现实领域思想的。开发人员选择聚合仓储模块，然后输入必要值。部门仓储 Get 模块和 Delete 模块的操作代码如下所示（使用 C#语言描述）：

Get 模块：

```
Department department;
GetDepartment getDepartment = new GetDepartment();
department = getDepartment.Get(“销售部”);
```

Delete 模块：

```
DeleteDepartment deleteDepartment = new DeleteDepartment();
deleteDepartment.Del(“素质拓展部”);
```

仓储提供的这种服务方式，使得开发人员完全感受不到数据库的存在，真正以领域为中心，在领域范围内执行许可的操作，减轻了思考压力。

4.5 本章小结

本章详细介绍了领域驱动设计方法在实际系统开发当中的应用。该系统为某水泥厂综合管理系统，涵盖产品管理、销售管理、部门管理、员工管理、客户管理、财务管理等多方面内容。本章根据上一章提出的领域驱动设计框架，详细介绍了系统总体结构和领域层的设计，并通过实例给出了聚合模块、工厂模块和仓储模块的设计方案，最后例举了一段仓储服务代码展示了领域驱动设计的思想原则与优点。

第 5 章 结论与展望

5.1 结论

本文从本质上分析了领域驱动设计思想的理论，针对理论中提出的六种领域模型元素模式的抽象概念，给出了具体的实现，并从宏观上将六种领域模型元素模式联系在一起：值对象是领域内部公认的基本概念，实体是领域需要处理的具体对象，聚合则是领域模型的基本组成部分；值对象、实体、聚合，组成了一个粒度由细到粗的层次结构，构成领域模型的数据框架；领域模型的功能由服务进行宏观协调，领域对象的生命周期则由工厂和仓储进行管理；六种领域模型元素模式互相配合，最终表现为一个完整的领域模型。

作为领域驱动设计思想的实现，本文还提出了一个轻量级领域驱动设计框架，该框架对系统架构、领域模型、数据访问模块以及数据库表结构的设计方法都作出了详细规定，能够将复杂的领域逻辑变得有条理，并封装繁琐的技术细节。使用该框架进行软件开发，可以让开发人员仅仅需要面对领域进行应用层面的设计，完全忽略底层具体平台的实现细节，减轻了开发人员的压力。

为了更好的说明领域驱动设计方法在实际软件开发中的应用，本文最后介绍了一个利用领域驱动设计思想实现的水泥厂综合管理系统。该系统通过本文提出的设计框架实现，将复杂的领域对象组合成领域模型，并将底层实现封装起来，使得整个系统的结构清晰自然，也提高了系统的可维护性与可扩展性。

5.2 展望

领域驱动设计是一种指导思想，一种思想可以衍生出无数种具体实现，本文提出的领域驱动实现框架只是其中一种。实际上，思想的升华是无穷无尽的，对思想的研究也是无穷无尽的，本文提出的框架仅仅是对目前领域驱动设计思想的部分实现，在未来的工作中，该框架还可以从以下几个方面进行改进：

- 领域模块的实现：领域模块是一种动态模块，需要根据具体领域即时实现，虽然可以考虑模块重用，但与基础服务模块不同，每一个系统的

业务逻辑都会和别的系统大相径庭[37]，领域模块重用率并不理想。由于领域模块实现依据的规则都相同，因此可以考虑开发一个自动生成领域模块的工具，开发人员只需输入领域的结构，即可生成特定的领域模块，这样大大减轻了开发人员的负担，避免了枯燥无味的劳动。文献[38-40]展示了这种方式的可行性。

- 仓储的使用：虽然仓储通过服务的形式使得持久化操作得到简化，但与大多数框架一样，仓储的使用仍需要手工调用。开发人员尽管不需要了解持久化的细节，但他们依然知道数据需要经历持久化这一过程，这意味着开发人员并没有完全从数据持久化中解脱出来。理想状况下，开发人员应当感觉自己是在内存中读取数据一般，完全忽略持久化这一过程，持久化操作交由系统自动负责。为达成这个目标，可以考虑在实现领域对象时，利用对象的析构函数来完成持久化操作，使开发人员与数据持久化完全脱离。
- 数据访问模块的改进：数据访问模块的作用，是根据不同数据库与参数信息，动态生成 SQL 语句，执行持久化操作。对于一个特定领域而言，需要进行的持久化操作是确定的，具体实现时，选择的数据库也是确定的，因此可以考虑开发一个工具，根据输入的领域信息与选择的数据库，自动生成数据库结构与比 SQL 语句执行效率更高的存储过程，提高持久化操作的效率，此时，数据访问模块的功能就简化为传递参数与调用存储过程。

虽然目前由于种种原因，领域驱动设计思想在软件开发行业还未形成潮流，但这并不能掩盖它的光芒，任何一种新思想、新技术的传播都是需要时间的，随着技术的进步，随着人们对其知识体系结构的不断完善，进而推出更多更适合的实现框架，领域驱动设计思想未来一定会成为软件开发行业的主流趋势。

致 谢

光阴似箭，两年半的研究生生活就要结束了。在此，我要向我的母校南昌大学和所有关心过我的老师、同学表示衷心的感谢。

在研究生学习期间和课题研究过程中，我的导师林仲达老师给予了我极大的帮助和关心。林老师丰富的专业知识，严谨的治学态度，勤奋的工作精神，给我留下了深刻的印象，使我终身受益匪浅。在本文的写作过程中，林老师即使工作繁忙，也不忘对我进行全面、耐心、细致的指导。正因为有了林老师的帮助，本文才得以按时完成，在此，谨向林老师表示崇高的敬意和感激。

此外，我还要特别感谢我的同学刘荣华、于仕，他们在我的学习和生活上给了我悉心的指导和关怀，使我少走了很多弯路，节约了大量时间。

感谢我的父母，他们给了我鼓励和支持。同时感谢参与论文评审的各位专家和老师！

严欣喆

2010 年 12 月

参考文献

[1] A.Andrews，J.Offutt，R.Alexander．Testing Web Applications by Modeling with FSMs[J]．Software and System Modeling，2005，4(3)：326-345．

[2] 陈建峡．RUP在软件开发中的应用[J]．武汉大学学报，2005，38(4)：116-120．

[3] Graig Larman．UML和模式应用——面向对象分析与设计导论[M]．姚淑珍，李虎，等译．北京：机械工业出版社，2003．

[4] Alexander，et al．A Pattern Language[M]．Oxford：Oxford University，1977．

[5] Martin Fowler．企业应用架构模式[M]．侯捷，熊杰，等译．北京：机械工业出版社，2004．

[6] 彭晨阳．对象和数据库的天然阻抗[OL]．http://www.jdon.com/mda/oo-reltaion2.html．

[7] 彭晨阳．面向对象与领域建模[OL]．http://www.jdon.com/mda/modeling.html．

[8] Dave T，Hansson D．Agile web development with rails[M]．New York：The Pragmatic Bookshelf，2005．

[9] Nilsson J．Applying domain-driven design and patterns:with examples in C# and .NET[M]．New Jersey：Addison-Wesley Professional，2006．

[10] Wesenberg H，Olmheim J，Landre E．Using domain—driven design to evaluate commercial of the shelf software[C]．The 21th ACM SIGPLAN conference on Object oriented programming systems and applications，Oregon，2006：824-829．

[11] Landre E，Wesenberg H，Olmheim J．Architectural improvement by use of strategic level domain-driven design[C]．The 21th ACM SIGPLAN conference on Object oriented programming systems and applications，Oregon，2006：809-814．

[12] Landre E，Wesenberg H，Olmheim J．Agile enterprise software development using domain-driven design and test first[C]，The 22th ACM SIGPLAN conference on Object oriented programming systems and applications．Quebec，2007：983-993．

[13] 彭晨阳．当前Java软件开发中几种认识误区[OL]．http://www.jdon.com/mda/nlayes.html．

[14] Martin Fowler．重构——改善既有代码的设计[M]．王怀民，周斌，等译．北京：中国电力出版社，2003．

[15] Harald Wesenberg，Einar Landre，Harald Ronneberg．Using domain-driven design to evaluate commercial off-the-shelf software[C]．Companion to the 21th ACM SIGPLAN conference on object-oriented programming languages，systems，and applications，Portland，Oregon，USA，2006，824-829．

[16] 张向芳，李华，姜英伟．软件测试自动化的实施方案[J]．太原：山西科技，2006年7月第四期．

[17] Daniel J.Mosley，Bruce A.Posey．软件自动化测试[M]．邓波，黄丽娟，曹青春，等译．北京：机械工业出版社，2003．

[18] 王珊，萨师煊．数据库系统概论（第 4 版）[M]．北京：高等教育出版社，2006．
[19] Hu Y，Peng S．So we thought we know money[C]．The 22th ACM SIGPLAN conference on Object oriented programming systems and applications，Quebec，2007：971-975．
[20] Martin Fowler，Kendall Scott．UML 精粹（第 2 版）[M]．北京：清华大学出版社，2002．
[21] Metayer D．Describing Software Architecture Styles Using Graph Grammars[J]．IEEE Transactions on Software Engineering，1998，24(7)：521-533．
[22] Steve McConnell．代码大全（第 2 版）[M]．金戈, 汤凌，陈硕，张菲，等译．北京：电子工业出版社，2006．
[23] RobertC.Martin．敏捷软件开发——原则、模式与实践[M]．邓辉，译．北京：清华大学出版社，2003．
[24] Dave Thomas，Brain M.Barry．Model driven development:the case for domain oriented programming[C]．Companion to the 18th ACM SIGPLAN conference on object-oriented programming languages，systems，and applications，Anaheim，CA，USA，2003，2-7．
[25] Martin Fowler．Language Workbenches：The killer—app for domain specific languages?[OL]．(2005-01-12)．http://martinfowler.com/articles/languageWorkbench.html．
[26] Len Bass，Paul Clements，Rick Kazman．软件架构实践[M]．车立红，译．北京：清华大学出版社，2004．
[27] 汤晟，吴朝晖．一个利用模型驱动体系结构技术的分布式系统实现[J]．北京：计算机工程与应用，2003，(33)：133-135．
[28] Jeff Garland Richard Antbony．大型软件体系结构：使用 UML 实践指南[M]．叶俊民，汪望珠，译．北京：电子工业出版社，2004．
[29] Eric Evans．领域驱动设计——软件核心复杂性应对之道[M]．陈大峰, 张泽鑫，等译．北京：清华大学出版社，2006．
[30] 张海藩编著．软件工程导论（第四版）[M]．北京：清华大学出版社，2003．
[31] E.Gamma，R.Helm，R.Johnson，and Vlissides．Desigh Patterns—Elements of Reusable Object-Oriented Software[M]．Addison-Wesley，1995．
[32] 阎宏．JAVA 与模式[M]．北京：电子工业出版社，2002．
[33] 彭晨阳．状态对象：数据库的替代者[OL]．http://www.jdon.com/artichect/state.htm．
[34] 彭晨阳．面向对象建模与数据库建模两种分析设计方法的比较[OL]．http://www.jdon.com/mda/oo_relation.html．
[35] 夏昕，曹晓钢，唐勇．深入浅出 Hibernate[M]．北京：电子工业出版社，2005．
[36] Christian，Bill Evjen，Jay Glynn，等著．C#高级编程[M]．李敏波，译．北京：清华大学出版社，2006．
[37] 冯冲，江贺，冯静芳．软件体系结构理论与实践[M]．北京：人民邮电出版社，2004．
[38] Martin Fowler. Domain specific languages[OL]. http://martinfowler.com/dslwip/index.html.
[39] Microsoft Corporation．Web service software factory[OL]．(2006-12)．http://msdn2.microsoft.com/en-us/library/aa480534.aspx．
[40] 张韧．业务功能驱动的服务提炼研究[D]．上海：上海交通大学电子信息与电气工程学院，2007．

攻读学位期间的研究成果

已发表论文：

1. 林仲达，严欣喆. 面向 RFID 海量数据的数据挖掘研究[J]. 电脑知识与技术，2010 年 19 期.