附录A

关于模式

我的第一辆“好车”，是我大学毕业不久别人送给我的，那是一辆已经开了 8 年的 Peugeot(标致)轿车。这种车有时候被亲切地称为“法国的梅塞德斯”，它的工艺十分精良，性能可靠，驾驶起来也很轻松愉快。但当我获得它的时候，它已经到了开始出现故障并且需要更多维护的时候了。

Peugeot 是一家已经有着几十年历史的老公司了。它有自己的机械术语，以及自己独特的设计风格。尽管如此，各个功能部件也不是标准的，换句话说，只有 Peugeot 的专家才能对车辆进行维护。对我这样一个刚毕业的学生来说，维护的费用就成了一个潜在而又十分重要的问题。

举一个典型的例子，我让本地的一个机械师帮我检查这辆车漏油的问题。他检查了车盘后告诉我，油是“从车后大概三分之二部位的一个小箱子里漏出来的，这个箱子看起来跟前后轮之间的制动力有关”。随后他拒绝了修理这辆车的请求，并且建议我到代理处去修理，可那里离本地有 50 公里的距离。与 Ford(福特)或 Honda(本田)汽车相比，它们的机械与 Peugeot 同样复杂，但所有的机械师都能维修它们，这就是为什么这些汽车变得更方便、更便宜的原因所在。

我确实很喜欢这辆 Peugeot 车，但我绝对不会再去购买一辆同样古怪的车了。一旦这辆车被检查出一个毛病，我将为此花费很多修理费，我已经受够了 Peugeot，就把它捐给了一家慈善机构。随后我把原打算用于维修那辆 Peugeot 车的钱买了一辆破旧的老 Honda Civic(本田思域)。

若在领域开发中缺乏标准设计基础，就会导致每个领域模型以及对应的实现都变得千奇百怪、难以理解。此外，每个团队都必须重复设计轮子或(传动装置、风挡刮水器)。在面向对象设计中，任何事物都是一个对象、一种引用或者一个消息，当然这是一种有

用的抽象。但是对象设计不能有效地限制领域设计选择的范围，并且也不能支持对领域模型进行有效的讨论。

如果停留在“任何事物都是对象”，那么木匠或者建筑师在描述房子时，他们只能说“每个都是房间”。那个有电源插座和一个水池的大房间是用来做饭的，而楼上的小房间则是您的卧室。如果这样来描述一个普通的房子会需要不少篇幅。因此修建和使用房子的人都意识到需要给房间分配一种模式，每种模式都要有专门的名称，例如“厨房”。这种语言能使我们明确地讨论房子的设计。

但并非所有功能的组合都是有用的。为什么不修建一个既可以供您洗澡又可以睡觉的房间呢？这样做会更加方便些。原因是因为长期的经验已经形成了习惯，并且通常我们把“卧室”跟“浴室”分得很开。毕竟，与卧室相比起来，洗澡的地方往往会被更多的人共同使用，他们需要绝对的独处而不受其他人的干扰，即使是住在同一间卧室的人。浴室需要有专门而又昂贵的基础结构，象浴缸和盥洗盆这样只能供私人使用的设施，通常都是放在同一个房间里的，因为它们都需要有同样的基础结构(水和排水装置)。

另外一间需要专门基础结构的房间是“厨房”，您可以在这个房间里准备饭菜。跟浴室相比，厨房没有专门的私用要求。建造厨房是十分昂贵的，那是因为在一个相当大的房子里也通常只有一个厨房。这种惟一性虽然使得它的造价昂贵，但也同样方便了我们共同分享准备好的食物，这也是符合我们的饮食习惯的。

当我说想要“一个有三间卧室、两间浴室、一间厨房的房子”时，在这句短短的句子中，我已经包含了大量的信息，并且避免了很多白痴的错误，例如把厕所放在冰箱的旁边。

在每一种设计领域，例如房子、汽车、划艇以及软件，我们都是在已建立模式的基础上，根据确定的主题即席创作的。有时，我们必须创造一些完全新的东西。但基于模式上的标准元素，我们避免了把精力浪费在已有解决方案的问题上，而可以把注意力放在不同寻常的需要上。同样的，从常用的模式中进行组装，帮助我们避免了设计太特殊而导致难以理解的问题。

尽管软件领域设计不如其他设计领域那么成熟，在软件领域中的任何情况下变化可能太多，以至于没有办法提供像汽车部件或者房间那样具体的模式，但至少应该像区分螺钉和弹簧一样，仍需要具有“任何事物都是对象”的概念。

共享形式和标准化设计是由 Christopher Alexander(Alexander et al. 1977)领导的一群建筑师们在 20 世纪 70 年代提出的。他们的“模式语言”与可靠解决公共问题(比起我举的“厨房”的例子复杂多了，这些问题可能会使 Alexander 的一些读者望而却步)的设计结合在一起。目的是让建造者和用户可以用这种语言进行交流，通过模式指导他们建造

出即漂亮又设施完善的建筑，让使用这些建筑的人们感到满意。

不管架构设计者们如何想，这种模式语言已经对软件设计产生了很大的影响。在20世纪90年代软件模式被应用到很多方面，并取得了一些成功，特别是详细设计(Gamma et al. 1995)和技术架构(Buschmann et al. 1996)。最近，模式已经被用来证明基本的面向对象设计技术(Larman 1998)和企业架构(Fowler 2002, Alur et al. 2001)。模式语言现在已经成为组织软件设计思想的一种主流技术。

模式名很可能会成为在开发团队中使用的语言的术语，本书中我已经把模式名当成术语来使用了。

附录B

术 语 表

下面简要地定义了本书中所使用的术语、模式名称和其他概念。

aggregate(聚合) ——一组被视为一个整体以便于进行数据变更的对象。外部对象只能引用聚合中一个特定的成员，即聚合的根元素。在聚合内部可以应用一系列一致性规则。

analysis pattern(分析模式) ——一组用来描述业务建模中常见构造的概念。它可能只与一个领域相关，但也可以跨多个领域(Fowler 1997，p. 8)。

assertion(断言) ——用来声明程序在某个时刻应具有的正确状态，而不管程序如何达到这个状态。断言的一般用法是用来指定某个操作的结果或者某个设计元素的不变量。

Bounded Context(限界上下文) ——一个特定模型在一定范围内有效的适用性。限界上下文使小组成员能够清晰而统一地理解哪些地方应该保持一致，哪些地方可以独立开发。

client(客户程序) ——调用当前设计元素以使用其能力的程序元素。

cohesion(内聚) ——逻辑上的协定和依赖性。

command / modifier(命令/修改器) ——使系统发生某些改变(例如给变量赋值)的操作。以产生某种副作用为目的的操作。

Conceptual Contour(概念轮廓) ——领域本身的内在一致性，如果这种一致性能反映到模型中的话，可以使设计更自然地适应改变。

context(上下文) ——一个单词或语句出现的背景，它决定了单词或语句的含义。参见限界上下文。

Context Map(上下文图) ——用来描述项目中包含的限界上下文及其与模型的实际联系。

Core Domain(核心领域) ——模型中以用户目标为中心的独特部分，它使得系统与众不同且具有价值。

declarative design(声明性设计) ——一种程序设计的形式，它用精确描述的属性来控制软件的行为。一种可执行的规范。

deep model(深层模型) ——对领域专家所关注的主要问题，以及与这些问题关系最密切的知识的深入描述。深层模型不再停留在领域的表面现象和粗浅理解上。

design pattern(设计模式) ——一组相互关联的对象和类的描述。我们可以通过定制这些对象和类来解决出现于特定上下文中的一般性设计问题。

distillation(精炼) ——从混合物中分离出各种成分，以便将本质提炼出来使之更有价值、更有用的过程。在软件设计中，精炼表示对模型中的关键概念进行抽象，或者对一个大的系统进行分解并从中析取出核心领域的过程。

domain(领域) ——知识、影响或活动的范围。

domain expert(领域专家) ——软件项目的一个成员，其专业方向是应用系统的领域，而不是软件开发。领域专家具有深厚的领域知识，而不仅仅是一个普通的软件用户。

domain layer(领域层) ——在分层架构中，负责领域逻辑的设计和实现的那个部分。领域层是用软件来表达领域模型的地方。

entity(实体) ——一个对象，它不是通过属性来定义，而是通过连续性(continuity)和标识来从根本上进行定义的。

factory(工厂) ——一种封装机制，它将复杂的对象创建逻辑封装起来，使客户程序无需关心待创建对象的具体类型。

function(函数) ——一种计算并返回结果的操作，它不产生可见的副作用。

immutable(不变性) ——一旦创建就永不改变的性质。

implicit concept(隐含概念) ——对于理解模型或设计的含义不可或缺，但从未被提起过的概念。

Intention-Revealing Interface(释意接口) ——在设计中，类、方法和其他元素的名字既表达出开发人员创建它们的原本意图，又表达了它们对于客户程序开发者的用处。

invariant(不变量) ——不变量是针对某个设计元素的断言，除了一些特殊的过渡状态(例如正在执行某个方法，或者数据库事务尚未提交之时)，这个断言必须总是被满足。

iteration(迭代) ——反复对程序进行小步改进的过程。也可以表示其中的一个小步。

Large-Scale Structure(大比例结构) ——构成整个系统设计的模式的一批高层概念或(和)规则，它们为我们讨论和理解系统提供了一种写意式的语言。

Layered Architecture(分层架构) ——一种分解软件系统关注点的技术，它使领域层与其他问题隔离开来。

life cycle(生命周期) ——一个对象从创建到消亡的过程中经历的状态序列，通常对状态变迁还有一些约束以保证完整性。可能包括实体在系统和不同限界上下文之间的迁移。

model(模型) ——一个描述了领域的指定方面，并能用来解决领域相关问题的抽象系统。

Model-Driven Design(模型驱动设计) ——在模型驱动的设计中，软件元素的一些子集与模型元素密切对应。模型驱动设计也可以表示使模型和实现互相匹配的联合开发过程。

modeling paradigm(建模范型) ——将概念从领域中析取出来的特定方式，以及把这些概念用软件模拟出来的工具(例如，面向对象程序设计和逻辑程序设计)。

repository(仓储) —— 一种用来封装存储、读取和查找行为的机制，它模拟了一个对象集合。

responsibility(职责) —— 执行某些任务或了解某些信息的责任(Wirfs Brock et al. 2003, p. 3)。

service(服务) ——一种以接口形式提供的操作，在模型中独立存在，且内部没有封装的状态。

side effect(副作用) ——由一个操作引起的任何可见的状态改变，无论这种改变是在执行过程中额外产生的，还是专门对状态进行的更新。

Side-Effect-Free Function(无副作用函数) ——参见函数

Standalone Class(孤立类) ——除了系统的原始类型和基本类库之外，没有任何其他依赖的类。

stateless(无状态) ——设计元素的一种性质。无状态元素允许客户使用它的任何操作，而无需考虑它的历史。无状态元素可能会使用某些全局可见的信息，甚至可能修改那些全局信息(也就是说它可以产生副作用)，但是它不保存任何可以影响其行为的私有状态。

strategic design(战略性设计) ——在对系统建模和设计的宏观性决策。这些决策将对整个项目产生影响，因此必须由小组讨论决定。

supple design(柔性设计) ——柔性设计能够充分发挥深层模型的内在能力，使客户程序的开发人员能够清晰、灵活地编写健壮的代码并取得预期的结果。柔性设计的另一个特性也同样重要，它能借助深层模型使得设计本身更容易被实现和重塑，以便适应新的理解。

Ubiquitous Language(通用语言) ——围绕领域模型来构造的语言，所有小组成员都使用这些语言来将小组的活动与软件联系起来。

unification(统一性) ——模型内部的一致性，即每个术语都是无二义性的，规则之间也不存在矛盾。

Value Object(值对象) ——描述某些特征或属性，但不包含任何概念标识的对象

Whole Value(整值) ——建模了一个完整的概念的对象。

附录C

参考文献

[1] Alexander, C., M. Silverstein, S. Angel, S. Ishikawa, and D. Abrams. 1975. *The Oregon Experiment*. Oxford University Press.

[2] Alexander, C., S. Ishikawa, and M. Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press.

[3] Alur, D., J. Crupi, and D. Malks. 2001. *Core J2EE Patterns*. Sun Microsystems Press.

[4] Beck, K. 1997. *Smalltalk Best Practice Patterns*. Prentice Hall PTR.

[5] Beck, K. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

[6] Beck, K. 2003. *Test-Driven Development: By Example*. Addison-Wesley.

[7] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. 1996. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley.

[8] Cockburn, A. 1998. *Surviving Object-Oriented Projects: A Manager's Guide*. Addison- Wesley.

[9] Evans, E., and M. Fowler. 1997. "Specifications." Proceedings of PLoP 97 Conference.

[10] Fayad, M., and R. Johnson. 2000. *Domain-Specific Application Frameworks*. Wiley.

[11] Fowler, M. 1997. *Analysis Patterns: Reusable Object Models*. Addison-Wesley.

[12] Fowler, M. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[13] Fowler, M. 2003. *Patterns of Enterprise Application Architecture*. Addison-Wesley.

[14] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. 1995. *Design Patterns*. Addison-Wesley.

[15] Kerievsky, J. 2003. "Continuous Learning," in *Extreme Programming Perspectives*, Michele Marchesi et al. Addison-Wesley.

[16] Kerievsky, J. 2003. Web site: http://www.industriallogic.com/xp/refactoring.

[17] Larman, C. 1998. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*. Prentice Hall PTR.

[18] Merriam-Webster. 1993. *Merriam-Webster's Collegiate Dictionary*. Tenth edition. Merriam- Webster.

[19] Meyer, B. 1988. *Object-oriented Software Construction*. Prentice Hall PTR.

[20] Murray-Rust, P., H. Rzepa, and C. Leach. 1995. *Abstract 40*. Presented as a poster at the 210th ACS Meeting in Chicago on August 21, 1995. http://www.ch.ic.ac.uk/cml/

[21] Pinker, S. 1994. *The Language Instinct: How the Mind Creates Language*. HarperCollins.

[22] Succi, G. J., D. Wells, M. Marchesi, and L. Williams. 2002. *Extreme Programming Perspectives*. Pearson Education.

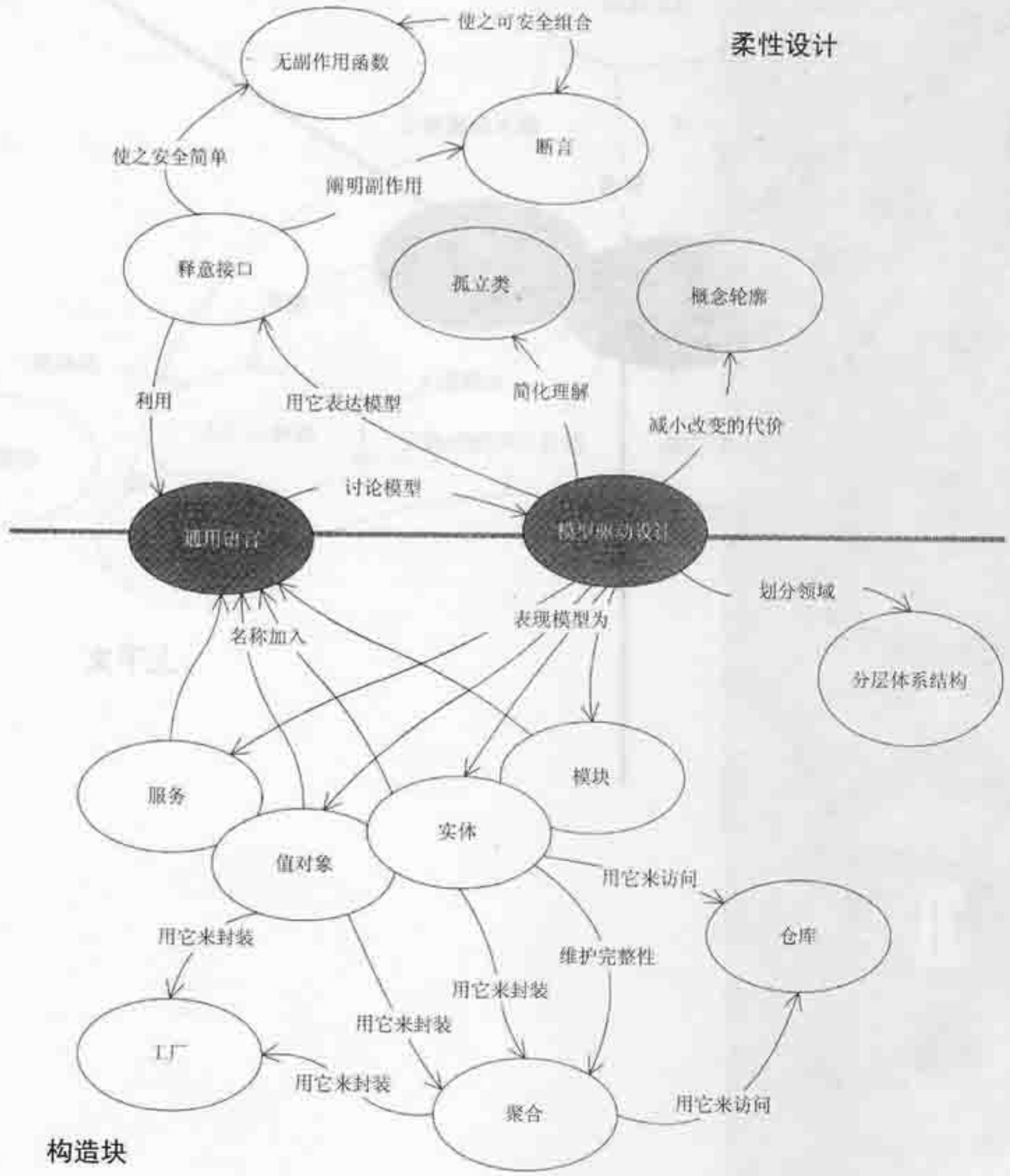
[23] Warmer, J., and A. Kleppe. 1999. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley.

[24] Wirfs-Brock, R., B. Wilkerson, and L. Wiener. 1990. *Designing Object-Oriented Software*. Prentice Hall PTR.

[25] Wirfs-Brock, R., and A. McKean. 2003. *Object Design: Roles, Responsibilities*, and Collabora ti o ns. Addison-Wesley.

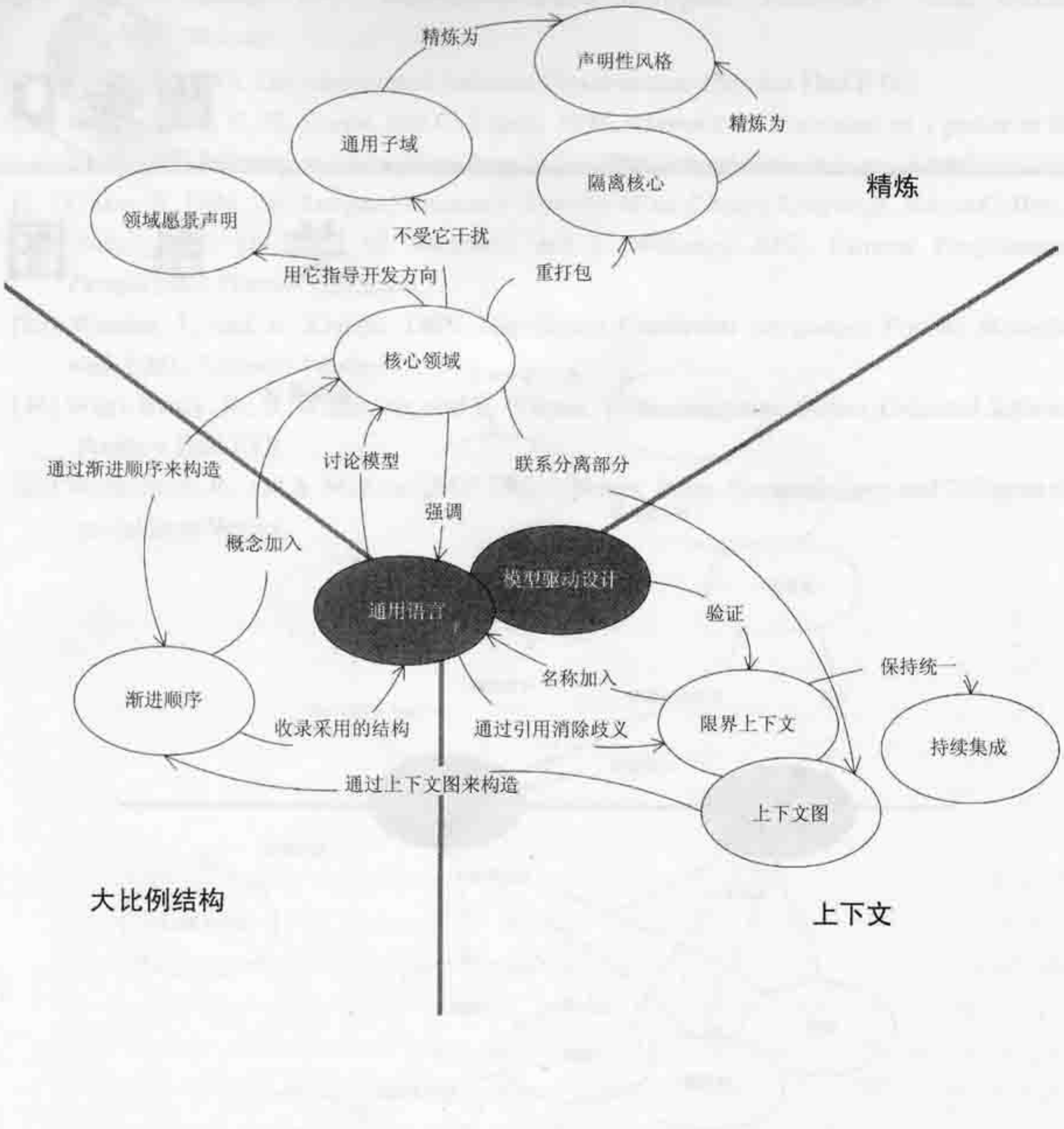
附录D

关 系 图



柔性设计

构造块


精炼为
声明性风格
精炼为
通用子域
隔离核心
精炼
领域愿景声明
不受它干扰
用它指导开发方向
重打包
核心领域
通过渐进顺序来构造
讨论模型
联系分离部分
强调
概念加入
模型驱动设计
通用语言
验证
名称加入
保持统一
渐进顺序
限界上下文
持续集成
收录采用的结构
通过引用消除歧义
通过上下文图来构造
上下文图
大比例结构
上下文

Domain-Driven DESIGN

领域建模已被业界普遍认为是软件设计成败的关键。通过领域建模，软件开发人员能够展示丰富的功能并将这些功能实现为真正满足用户需要的软件。尽管领域建模非常重要，但市面上介绍如何将有效的领域建模结合到软件开发过程中的著作却非常少。

本书就是为此目的而编写的。它向读者系统地讲述了领域驱动设计的方法，介绍了大量优秀的设计示例、技术经验以及用于处理复杂领域软件工程的基本原则。本书做到了设计和开发实践相结合，在介绍领域驱动设计的同时，还提供了大量的Java示例。

通过本书，读者将获得对领域驱动设计的总体认识，了解领域驱动设计中涉及的关键原则和术语。

面向对象的开发人员、系统分析师以及设计师在深入思考领域问题时，能够从本书中获得一定的指导，从而建立丰富而有用的领域模型，并将这些模型转化为高质量和持久的软件实现。

本书主要内容

- 分离领域
- 使用通用语言
- 管理领域对象的生命周期
- 实现建模突破
- 挖掘模型中的隐含概念
- 应用分析模式
- 将设计模式与模型相联系
- 维护模型的完整性
- 编写领域愿景声明
- 面向更深层理解的重构
- 柔性设计
- 创建插件框架
- 结合大比例结构与限界上下文

"每个有思想的软件开发人员的书架上都应该珍藏这样一本书。"

—— Kent Beck

"Eric设法收集了资深对象设计人员长期使用的一些设计过程，且在本书中分析、归纳了这些过程，并将建立领域逻辑的原则组织起来，使其系统化，以期与众多设计人员共享。本书将会让您受益匪浅！"

—— Kyle Brown，*Enterprise Java Programming with IBM WebSphere*一书的作者

上架建议：软件工程

ISBN 7-302-11576-1

9 787302 115762 >

定价：48.00元

本书合作站点：www.awprofessional.com
www.pearsoned.com

PEARSON Addison Wesley