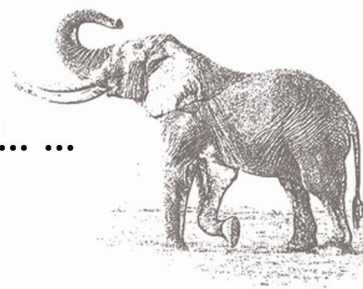


PART I

你需要了解 Prepare



何为面向对象？很多人不懂；很多人以为懂了，其实没懂。面向对象的精髓在抽象；面向对象的困难在抽象；面向对象的成功在于成功的抽象；面向对象的失败在于失败的抽象。正所谓成也抽象，败也抽象。还是打好基本功，从基本的面向对象开始吧。

抽象抽象就是抽象

为什么需要 UML

过程还是对象？这是个问题。谈到 UML，第一个绕不开的话题就是面向对象，就让我们先从基本的方法开始，逐步揭开面向对象的面纱吧。

1.1 面向过程还是面向对象



面向对象如今在软件行业是如此著名的一个术语，以至于人们以为面向对象是现代科学发展到一定程度才出现的研究成果。在很多人看来，面向过程和面向对象都是一种软件技术。例如把面向过程归纳为结构化程序设计、DFD 图、ER 模型、UC 矩阵等，而面向对象则被归纳为继承、封装、多态、复用等具体的技术。事实上，上述的所有技术都只是人们在采用不同的方法来认识和描述这个世界时所采用的工具，它们都只是表征而不是本征。让我们先来看看公认的面向对象大师，也是 UML 创始人之一的 Grady Booch 在 2004 年 IBM Developer Works Live! 大会的访谈中讲过的一段流传甚广的话。



我对面向对象编程的目标从来就不是复用。相反，对我来说，对象提供了一种处理复杂性问题的方式。这个问题可以追溯到亚里士多德：您把这个世界视为过程还是对象？在面向对象兴起运动之前，



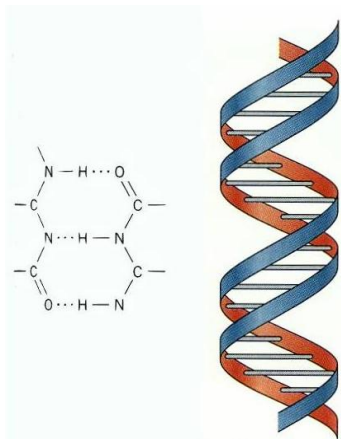
编程以过程为中心，例如结构化设计方法。然而，系统已经到达了超越其处理能力的复杂性极点。有了对象，我们能够通过提升抽象级别来构建更大的、更复杂的系统——我认为，这才是面向对象编程运动的真正胜利。

不知读者看完这段话有何感想？您心目中的面向对象是这样的吗？正如 Booch 讲到的一样，从本质上说面向过程和面向对象是一个古已有之的认识论的问题。之所以面向对象方法会兴起，是因为这种认识论能够帮助我们构造更为复杂的系统来解释越来越复杂的现实世界。认识到这一点，我们应该知道比掌握具体的技术更重要的是掌握认识论所采用的方法和分析过程。只有掌握了方法才能自如地使用工具。

作者本人认同这个世界的本质是由对象组成的，平时看上去相互无关的独立对象在不同的驱动力和规则下体现出不同的运动过程，然后这些过程便展现出了我们这个生动的世界。在面向过程的眼中，世界的一切都不是孤立的，它们相互地紧密联系在一起，缺一不可，互相影响，互相作用，并形成一个个具有严格因果律的小系统；而更多的小系统组成了更大的系统，所有小系统之间的联系也是紧密和不可分割的。

面向对象思想其实并不复杂。但是对习惯了以过程方法来认识这个世界的朋友来说完全理解和接受面向对象思想并不容易。然而如果您真的打算学习面向对象的方法，那么恐怕您得接受这个世界是分割开来的这个事实，并且相信只有特定的场景下，孤立对象之间进行了某些信息交互才表现出我们所看到的那样一个过程。在接下来的章节里，作者将透过 UML 和实例来阐述这种思维方法，希望读者能够在逐渐深入的过程中习惯和掌握从对象的角度来认识这个世界。

1.1.1 面向过程方法



面向过程方法认为我们的世界是由一个个相互关联的小系统组成的。正如左图所示的 DNA，整个人体就是由这样的小系统依据严密的逻辑组成的，环环相扣，井然有序。面向过程方法还认为每个小系统都有着明确的开始和明确的结束，开始和结束之间有着严谨的因果关系。只要我们将这个小系统中的每一个步骤和影响这个小系统走向的所有因素都分析出来，我们就能完全定义这个系统的行为。

所以如果我们要分析这个世界，并用计算机来模拟它，首要的工作是将这个过程描绘出来，把它们的因果关系都定义出来；再通过结构化的设计方法，将这些过程进行细化，形成可以控制的、范围较小的部分。通常，面向过程的分析方法是找到过程的起点，然后顺藤摸瓜，分析每一个部分，直至达到过程的终点。这个过程上的每一部分都是过程链上不可分割的一环。

让我们看看一个传统的商业分析过程。

如图 1.1 所示，计算机通过数据来记录这个过程的变迁。过程中每一步都会产生、修改或读取

一部分数据。每一个环节完成后，数据将顺着过程链传递到下一部分。当我们需要的最终结果在数据中被反映出来，即达到预期状态的时候，我们认为这个过程结束了。从图 1.1 中也可以看出，销售定单数据是这个过程的核心。为了能很好地分析这样的过程，DFD 图被广泛应用。DFD 图表达了“(从上一步)输入数据→(在这一步)功能计算→(向下一步)输出数据”这样一个基础单元。例如图中的“销售定单”单元，它读取客户请求，创建了销售定单数据；而“财务处理”单元则读取定购的商品信息，写入财务数据……直到“物流”单元将货物送到消费者手中并将数据写入销售定单后，这个过程才宣告结束。

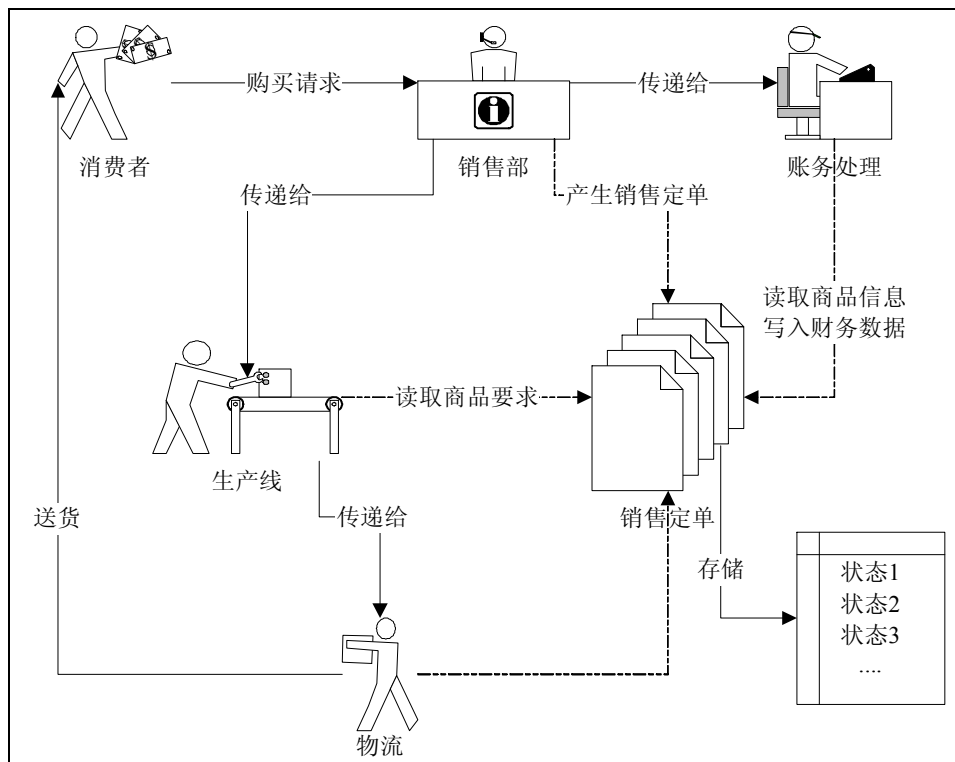


图 1.1 传统型商务

由于数据是如此重要，因此数据的正确性和完备性对系统成功与否至关重要。为了更好地管理数据，不至于让系统运行紊乱，人们通过定义主键、外键等手段将数据之间的关系描绘出来，结构化地组织它们，利用关系理论，即数据库的三大范式来保证它们的完备性和一致性。在面向过程成为主要的软件方法之后，关系型数据库得到了极大的发展，针对数据的分析方法 ER 模型也深入人心，被极为广泛地使用。

然而随着需求越来越复杂，系统越来越庞大，功能点越来越多，一份数据经常被多个过程共享，这些过程对同一份数据的创建和读取要求越来越趋于复杂和多样，经常出现相矛盾的数据需求，因此分析和设计也变得越来越困难。为了解决这个问题，IBM 在 20 世纪 70 年代提出了 UC 矩阵的



方法来求解功能和数据之间的依赖问题。这个方法将功能点和数据分别作为横纵坐标来组成一个二维矩阵，在横纵坐标的每个交叉点上标记功能点对数据的要求，即 U (Use) 和 C (Create)。标记完成后，对功能点位置和数据位置在各自坐标系上进行调整，调整的目标是尽量将 C 放在整个矩阵的对角线上，这时，各个功能点对数据的交叉依赖是最小的。最后，再把相邻的一些功能点分组，由此来获得数据交叉依赖性最小，功能聚合最紧密的子系统。

尽管 UC 矩阵是一个非常管用的方法，但面向过程的困难并没有从根本上解决。好方法只是使困难暂时得以缓解。本质的问题出在认识方法上。将世界视为过程这个方法本身蕴涵着一个前提假设，即这个过程是稳定的，这样我们才有分析的基础，所有的工作成果都依赖于对这个过程的步步分析。同时，这种步步分析的过程分析方法还导致另一个结果，即过程中的每一步都是预设好的，有着严谨的因果关系。只可惜我们这个世界从来都不是一成不变的，尤其到了信息化时代，一切都无时无刻不在发生着变化，系统所依赖的因果关系变得越来越脆弱。如今，SOA 已经提上日程，IBM 也提出了著名的口号：**On-Demand Business (按需应变的商务)**，面向过程已经面临了太多的困难，世界的复杂性和频繁变革已经不是面向过程可以轻易应付的了。

1.1.2 面向过程的困难

正如上一节所言，面向过程面临了太多的困难，甚至对某些情况束手无策。那到底是什么样的困难呢？让我们来看看图 1.2。当需求只是一个简单的过程（图 1.1）时，面向过程还可以从容应对，做出一个完整的销售过程分析；但是当需求变成一个按需应变的商务（图 1.2）时，这个过程的复杂性就已经超出了我们用一个完整过程来模拟的程度了。甚至，图 1.2 中的某些节点之间根本没有因果关系，它们只是临时组合起来表达某种商业需要。我们即使花费大力气把可能的信息组合方式一一枚举出来，还没等我们舒一口气，由于某个商业事件的变化，这些信息的组合方式又变了……图 1.2 充斥的问号反映了这种困惑。

例如，通过“商业分析”来收集和分析消费者的消费习惯，通过对细分市场的调查来了解商品需求变化。通过对这些采集来的数据进行分析 and 预测，销售策略就有可能发生变化。这个变化导致的结果是整个销售过程被颠覆。回想一下面向过程分析方法的前提和基础。当过程不再稳定，结果不再能预设的时候，面向过程方法还如何进行分析？

再例如，传统商业的销售数据可能只包含销量和利润，而到了今天，客户满意度、客户消费习惯、细分市场的变化、质量反馈……一份又一份的采样数据被包含进来。对以数据为中心的面向过程方法来说，数据的变化不但过于频繁，而且常常是结构性的颠覆。以数据为分析基础的面向过程方法该如何保持程序的稳定呢？

我们看到，面向过程的困难，本质上是因为面向过程方法将世界看作是过程化的，一个个紧密相连的小系统，构成这个系统的各个部分之间有着密不可分的因果关系。这种分析方法在需求复杂度较低的时候非常管用，如同一台照相机，将物体的反光经过镜头传导到感光胶片，再经过冲洗就能将信息复制出来。然而这个世界系统是如此的复杂和不可捉摸，就如同那个著名的蝴蝶效应，预设的过程仅仅因为一只蝴蝶轻轻扇动了一下翅膀就从此被颠覆，变得面目全非了。

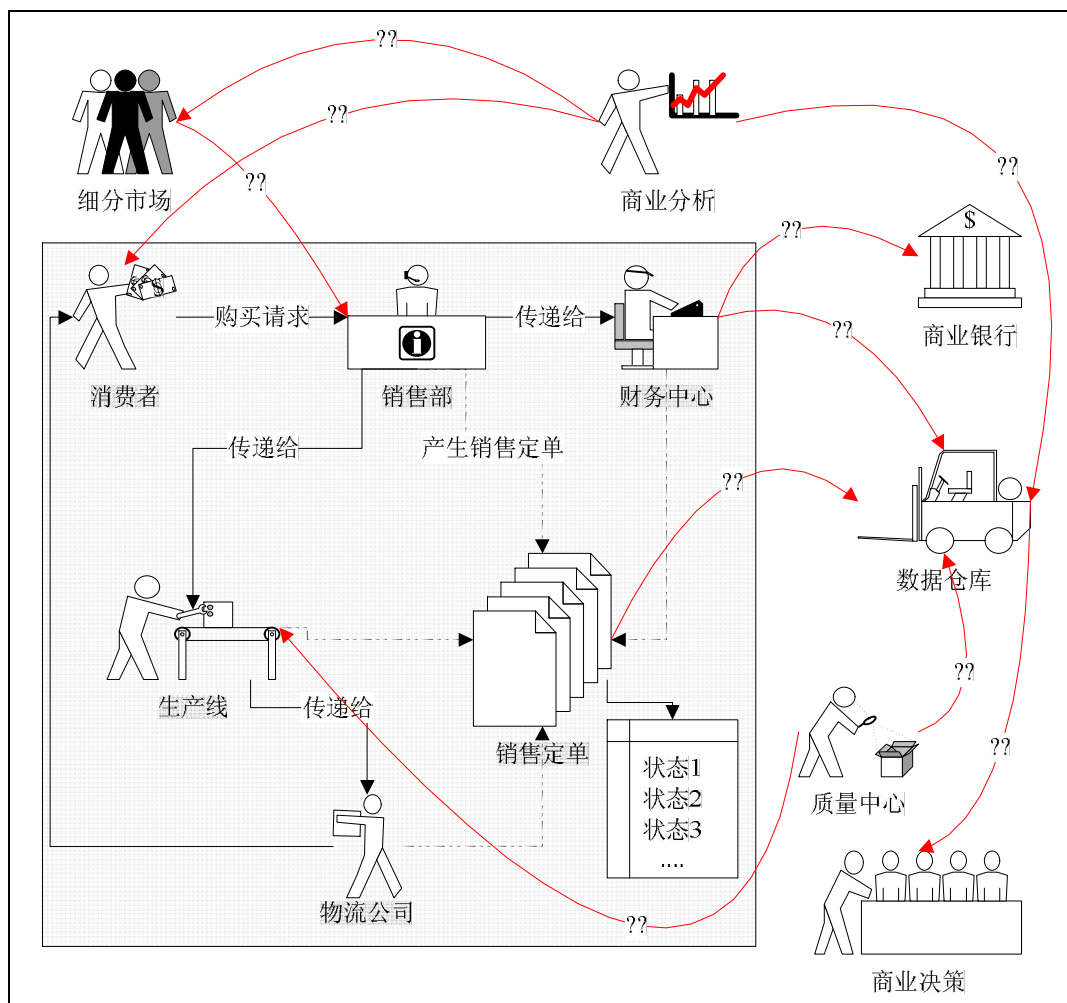


图 1.2 按需应变的商务

其实并非面向过程的方法不正确，只是因为构成一个系统的因素太多，要把所有可能的因素都考虑到，把所有因素的因果关系都分析清楚，再把这个过程模拟出来实在是太困难了。我们的精力有限，计算能力有限，只能放弃对整个过程的了解，重新寻找一个方法，能够将复杂的系统转化成一个个我们可以控制的小单元。这个方法的转换正如：如果一次成型一辆汽车太过困难，我们可以将汽车分解为很多零件，分步制造，再依据预先设计好的接口把它们安装起来，形成最终的产品。

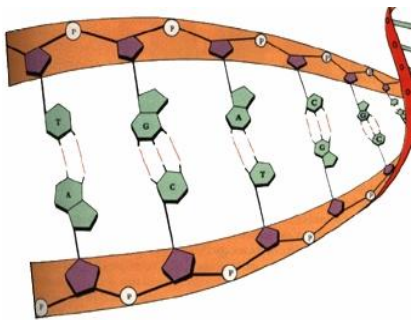
这种把复杂工程转化成标准零部件的做法，在工业界早已非常普遍，这正是一种面向对象的方法。与过程方法不同的是，汽车不再被看作一个一次成型的整体，而是被分解成了许多标准的功能部件来分步设计制造。我们在市面上看到的每一款汽车，都是基于某个商业策略，由不同的标准零部件组合而成。当市场变化、商业策略变化时，可以通过变更标准零部件来迅速生产一款新车型。

面向过程面对如今这个复杂的世界显得无能为力，面向对象又如何呢？从下一节开始我们将进



入精彩纷呈的面向对象世界，来探询面向对象方法是如何面对这个复杂世界的。

1.1.3 面向对象方法



面向对象（Object Oriented，简称 OO）方法将世界看作一个个相互独立的对象，相互之间并无因果关系，它们平时是“鸡犬之声相闻，老死不相往来”的。只有在某个外部力量的驱动下，对象之间才会依据某种规律相互传递信息。这些交互构成了这个生动世界的一个“过程”。在没有外力的情况下，对象则保持着“静止”的状态。

如左图所示，在面向对象看来，同样一个 DNA，它们的联系并非是这样紧密的。看上去浑然一体的小系统，其实是由脱氧核苷酸这一独立对象通过一定的化学键构成的。独立对象依据某个规律结合在一起，具备了特有性质和功能，然后又构成更为复杂的更大的对象，这正是面向对象的基本原理。

从微观角度说，这些独立的对象有着一系列奇妙而古怪的特性。例如，对象有着坚硬的外壳，从外部看来，除了它用来与外界交互的消息通道之外，对象内部就是一个黑匣子，什么也看不到，这称为**封装**；再例如对象可以结合在一起形成新的对象，结合后的对象具有前两者特性的总和，这称为**聚合**；对象可以繁育，产下的孩子将拥有父辈全部的本领，这称为**继承**；每个对象都有多个外貌，在不同情况下可以展现不同的外貌，但本质只有一个，这就是**接口**；而多个对象却可能长着相同的脸，但同样的这张脸背后却是不同的对象，它们有着不同的行为，这就是**多态**。

从宏观角度说，对象是“短视”的，它不知道也无法理解它所处的宏观环境，也不知道它的行为会对整个宏观环境造成怎样的影响。它只知道与它有着联系的身边的一小群伙伴，这称为**依赖**，并与小伙伴间保持着信息交流的关系，这称为**耦合**。同时对象也是“自私”的，即便在伙伴之间，每个对象也仍然顽固地保护着自己的领地，这称为**类属性**，只允许其他人通过它打开的小小窗口，这称为**方法**，进行交流，从不允许对方进入它的领地。

然而对象也喜欢群居，并且总是“物以类聚，人以群分”。这些群居的对象有着一些相似的性质，它们依靠这些相似的性质来组成一个部落。对象们寻找相似性质并组成部落的过程称为**抽象**，它们组成的部落称为**类**；部落里的每个成员既有共同的性质又有自己的个性，我们只有把特有的个性赋给部落成员才能区分它们并使它们活动起来，这称为**实例化**。

相信您一定已经被对象这些古怪的性质搞迷糊了。难道这个世界不应该有明确的因果关系吗？由着这些奇怪的不通人情的对象胡乱组合，就能形成我们这个规律的世界吗？对象或许是没有纪律的，但是一旦我们确定了一系列的规则，把符合规则要求的对象组织起来形成特定的结构，它们就能拥有某些特定的能力；给这个结构一个推动力，它们就能做出规则要求的行为。

图 1.3 展示了这样一个结果，当离散对象们被按规则组合起来以后，就能表达预期的功能。其实世界就是这样组成的。平时看上去每个对象都互无关系，然而当它们按图示规则组织起来之后，踩下刹车，汽车便乖乖停住了。

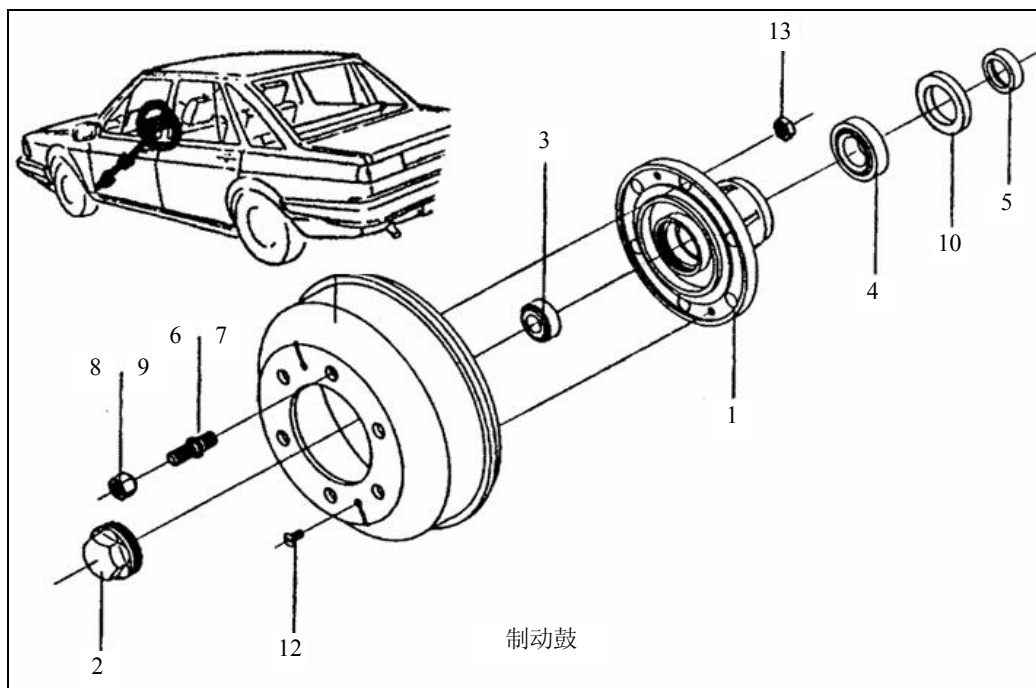


图 1.3 对象组装

从图 1.3 中可以发现一个特点，每个对象都只与有限的其他对象有关系。分析对象时不再需要动辄把整个世界拉下水，从头到尾分析一遍，我们只需要关心与它有关系的那几个对象。这使得我们在分析对象的时候需要考虑的信息量大大减少，自然的，这就减化了我们所面对问题领域的复杂程度。

从图 1.3 中还可以发现，某些零件不是特殊的只能用于制动鼓的。如螺丝和螺帽，它们还可以用于别的地方。这是面向对象的一个重要特性：**复用**。

从图 1.3 还可以读出的另一个重要的信息是，由于对象是独立于最终产品的，只要符合规则要求，这些标准零件就可以替换！我们可以采用钢制的，也可以采用合金制的；可以采用 A 工厂生产的，也可以采用 B 工厂生产的。这使得我们可以在不改变既定目标的情况下替换零件，给我们带来了极大的灵活性和扩展能力。

再扩展一下我们的视野。按照图 1.3 的规则，我们用零件组装出了一个刹车部件；相似的，按其他特定的规则，我们可以用零件组装出发动机、底盘等其他大一些的部件。然后，我们还可以用这些大一些部件来组装更大一些的东西，例如一辆完整的小汽车，当然，也可能是一部拖拉机。

无论如何，以上描述揭示了面向对象的一个非常重要的特性：**抽象层次**。以上的描述是由小及大的（或称为自底向上）的抽象过程；我们也可以反过来，由大及小（或称为自顶向下）的来抽象。例如站在汽车的抽象层次，我们会发现汽车是由变速器、发动机、底盘等大一些的部件组成的；如



果降低一点，站在发动机的抽象层次上，我们会发现发动机是由汽缸、活塞等零件组成的；而站在活塞的抽象层次，我们还会发现活塞是由拉杆、曲轴等更小的零件组成的……只要你愿意，这种抽象层次可以一直延伸下去，直到原子，夸克……

抽象层次的好处是不论在哪一个层次上，我们都只需要面对有限的复杂度和有限的对象结构，从而可以专心地了解这个层次上的对象是如何工作的；抽象层次的另一个更重要的好处是低层次的零件更换不会影响高层次的功能，设想一下更换了发动机的火花塞以后，汽车并不会因此而不能驾驶。

不知聪明的读者有没有从这个例子中悟出一点什么来。面向对象方法与面向过程方法根本的不同，就是不再把世界看作是一个紧密关联的系统，而是看成一些相互独立的离散的小零件，这些零件依据某种规则组织起来，完成一个特定的功能。原来，过程并非这个世界的本源，过程是由通过特定规则组织起来的一些对象“表现”出来的，原始的对象既独立于过程，也独立于组装规则。面向对象和面向过程的这个差别导致了整个分析设计方法的革命。分析设计从过程分析变成了对象获取，从数据结构变成了对象结构。在后续的章节里，将看到面向对象的分析设计是如何进行的。这个过程正如同组装一辆汽车，您将不会觉得有任何的难以理解。相反，一旦开始习惯了这种方法，会感到面向对象其实比面向过程更自然地表达了这个世界。

当然，世上并无完美的事情，面向对象尽管有这么一大堆的好处，它也有着其与生俱来的困难。

1.1.4 面向对象的困难

在上一节中我们看到了对象是如何按规则组装出一辆汽车的。然而细心的读者可能会提出这样一些疑问：

- 你只告诉了我们利用零件能够组装出我们需要的功能，但是，你却没有告诉我们零件是怎么来的？难道零件是从石头里蹦出来的孙悟空，突然出现的吗？符合规则的标准零件是如何设计和制造出来的？
- 经过测试，我承认现在这个结构可以完成那个特定的功能，但我还是不明白，如果我用另一些零件，换另一个组装规则，就不能完成那个特定的功能了吗？为什么是这个结构而不能是另一个？这个结构到底是怎样实现那个特定功能的呢？
- 零件是标准的，组装规则是可以变化的，这意味着我可以任意改变规则来组合它们。显然的，即使是任意的组装，它们也必然表达了某一种特定的功能。那么我随意组装出来的结构表达了什么功能呢？

上述疑问实质上体现了现实世界和对象世界的差距，即使面对简单的传统商业模式，我们仍有如下困惑：

- 对象是怎么被抽象出来的？现实世界和对象世界看上去差别是那么大，为什么要这么抽象而不是那么抽象呢？（Why）
- 对象世界由于其灵活性，可以任意组合，可是我们怎么知道某个组合就正好满足了现实世界的需求呢？什么样的组合是好的，什么样的组合是差的呢？（How）

- 抛开现实世界，对象世界是如此的难以理解。如果只给我一个对象组合，我怎么才能理解它表达了怎样的含义呢？（What）

这些困惑可以总结为图 1.4。

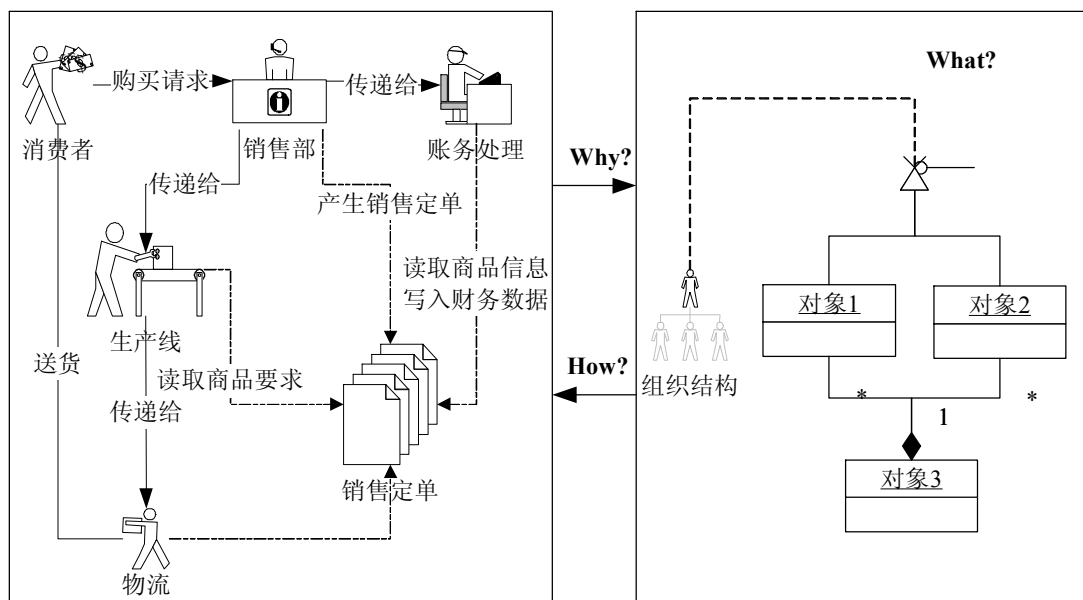


图 1.4 面向对象的困难

在实际的工作中，我们常常设计出许多类来满足某个需求。但是如果问一问为什么要这样设计，为什么是五个类而不是七个类？为什么是十个方法而不是十二个？能很好回答这个问题的人并不多，绝大部分人的回答是凭经验。经验是宝贵的，可惜经验也是靠不住的，凭经验的另一个说法是拍脑袋。从需求到设计，从现实到对象，那些类的确正如孙悟空从石头里蹦出来一样，设计师一拍脑袋就出来了。而可怜的经验不足的设计师们，在面对一个复杂需求的时候，只好不断尝试着弄出几个类来，拼一拼，凑一凑，发现解决不了问题，再重新来过……许多项目就在这样的尝试中不断受伤。

不知亲爱的读者您是否也是这样做设计的？

尽管不情愿，还是得承认很多时候我们只能通过不断测试来证明我们设计出来的那些类的确实现了需求，这往往需要在项目后期投入大量的返工成本，却不能清楚地设计阶段就证明这些类已经满足了实际需求。许多设计师在被要求验证他的设计的确满足需求的时候，常常听到的回答是我的这个设计应用了某某设计模式，这个结构很灵活，扩展性很强，它肯定能满足需求……空谈了许多却无法拿出一个实实在在的推导过程。

不知亲爱的读者您是否也有这样的经历？

而同时许多程序员，一手拿着设计师设计出来的结果，一手拿着系统分析员编写的需求说明书，



苦思冥想，就是无法把两者对上号；既搞不清楚到底设计是如何映射到需求的，也找不到两者之间的关系；他们脑子里常常盘旋着的问题是：这个类是表达了什么意思？为什么是这样的？

不知亲爱的读者是否也有这样的困惑？

如果您正被上面的这些问题困扰着，请您不要怀疑是否面向对象错了。我们把世界看作是由许多对象组成的这并没有错，只是现实世界和对象世界之间存在着一道鸿沟，这道鸿沟的名字就叫做抽象。抽象是面向对象的精髓所在，同时也是面向对象的困难所在。实际上，要想跨越这道鸿沟，我们需要：

- 一种把现实世界映射到对象世界的方法。
- 一种从对象世界描述现实世界的方法。
- 一种验证对象世界行为是否正确反映了现实世界的方法。

幸运的是，UML，准确地说是 UML 背后所代表的面向对象分析设计方法，正好架起了跨越这道鸿沟的桥梁。在下一节里，让我们带着疑问，来看看 UML 是如何解决这些问题的。



闲话：今天你 OO 了吗？

如果你的分析习惯是在调研需求时最先弄清楚有多少业务流程，先画出业务流程图，然后顺藤摸瓜，找出业务流程中每一步骤的参与部门或岗位，弄清楚在这一步参与者所做的事情和填写表单的结果，并关心用户是如何把这份表单传给到下一个环节的。那么很不幸，你还在做面向过程的事情。

如果你的分析习惯是在调研需求时最先弄清楚有多少部门，多少岗位，然后找到每一个岗位的业务代表，问他们类似的问题：你平时都做什么？这件事是谁交办的？做完了你需要通知或传达给谁吗？做这件事情你都需要填写些什么表格吗？...那么恭喜你，你已经 OO 啦！

1.2 UML 带来了什么

上一节中我们了解了面向过程和面向对象两种不同方法在描述现实世界时的不同，相信面向对象是更好的方法。但是，面向对象也有着天然的困难。本节我们来看看面向对象设计的事实标准 UML 是如何解决这些困难的。

1.2.1 什么是 UML

从 20 世纪 70 年代末期面向对象运动兴起以来，到现在为止，面向对象已经成为了软件开发的最重要的方法。上一节中我们也提出了面向对象有着诸多的困难，它的发展并不是一帆风顺的。

面向对象的兴起是从编程领域开始的。第一种面向对象语言 Smalltalk 的诞生宣告了面向对象开始进入软件领域。最初，人们只是为了改进开发效率，编写更容易管理、能够重用的代码，在编程语言中加入了封装、继承、多态等概念，以求得代码的优化。但分析和设计仍然是以结构化的面向过程方法为主。

在实践中,人们很快就发现了问题:编程需要的对象不但不能够从设计中自然而然地推导出来,而且强调连续性和过程化的结构化设计与事件驱动型的离散对象结构之间有着难以调和的矛盾。由于设计无法自然推导出对象结构,使得对象结构到底代表了什么样的含义变得模糊不清;同时,设计如何指导编程,也成为了困扰在人们心中的一大疑问。

为了解决这些困难,一批面向对象的设计方法(OOD方法)开始出现,例如 Booch86、GOOD(通用面向对象开发)、HOOD(层次化面向对象设计)、OOSE(面向对象结构设计)等。这些方法可以说是如今面向对象方法的奠基者和开拓者,它们的应用为面向对象理论的发展提供了非常重要的实践和经验。同时这些方法也是相当成功的,在不同的范围内拥有着各自的用户群。

然而,虽然解决了从设计到开发的困难,随着应用程序的进一步复杂,需求分析成为比设计更为重要的问题。这是因为人们虽然可以写出漂亮的代码,却常常被客户指责不符合需要而推翻重来。事实上如果不符合客户需求,再好的设计也等于零。于是 OOA(面向对象分析)方法开始走上了舞台,其中最为重要的方法便是 UML 的前身,即:由 Booch 创造的 Booch 方法,由 Jacobson 创造的 OOSE、Martin/Odell 方法, Rumbaugh 创造的 OMT、Shlaer/Mellor 方法。这些方法虽然各不相同,但它们的共同的理念却是非常相似的。于是三位面向对象大师决定将他们各自的方法统一起来,在 1995 年 10 月推出了第一个版本,称为“统一方法”(Unified Method 0.8)。随后,又以“统一建模语言”(Unified Modeling Language) UML 1.0 的正式名称提交到 OMG(对象管理组织),在 1997 年 1 月正式成为一种标准建模语言。之所以改名,是因为 UML 本身并没有包含软件方法,而仅是一种语言,我们将在 1.3 节统一过程简介中解释语言和方法的关系。

如上所述 UML 是一种建模用的语言,而所有的语言都是由基本词汇和语法两个部分构成的,UML 也不例外。UML 定义了一些建立模型所需要的、表达某种特定含义的基本元素,这些元素称为元模型,相当于语言中的基本词汇,例如用例、类等。另外,UML 还定义了这些元模型互相之间关系的规则,以及如何用这些元素和规则绘制图形以建立模型来映射现实世界;这些规则和图形称为表示法或视图(View),相当于语言中的语法。如同我们学习任何一种语言一样,学习 UML 无非是掌握基本词汇的含义,再掌握语法,通过语法将词汇组合起来形成一篇有意义的文章。UML 与其他自然语言和编程语言在原理上并无多大差别,无非是 UML 这种语言是用来写说明文的,用自然世界和计算机逻辑都能够理解的表达方法来说明现实世界。

然而,即使是同样的语言、同样的文字、同样的语法,有的人能够写出优美的小说和瑰丽的诗句,有的人却连一封书信都写不通顺。这种差别除了对语言掌握的功力之外,更重要的是写作人自己脑子里的思想和理念。好的文章除了语言功底,更重要的是言之有物、言之精确、言之全面,也就是作者要肚子里有货。如果以写文章来类比的话,学习 UML 只是学会了一门语言,而要写出一篇精彩的文章,却要依靠写作人对生活的感悟和升华,这两者缺一不可。因此比学会用 UML 建模本身更重要的是要理解 UML 语言背后所隐含的最佳实践。

谈到语言,我们无法回避的一个问题是沟通。如果不能用于沟通,那语言就没有意义。而要最大程度地沟通,那么最好的办法就是创造一种大家都认同的统一语言。



1.2.2 统一语言



统一的意义似乎不用多说，秦始皇历史上的一大功绩便是统一语言和度量衡。统一的目标就是形成标准。对于现代社会来说，标准被广泛应用的重要程度不亚于互联网的发明，各行各业无不纷纷制定各式各样的标准。标准使得不同地域、不同文化、不同社会、不同组织的信息能够以所有人都明白的表述和所有人都遵从的格式在人群中无障碍地流通。在我们的生活中，在街上随便买一款手机就能上网通话，随便买张影碟就能在家里的 DVD 机上播放；而在软件界，任何一种组件化开发模式背后都有一个标准在规范和指导，可以说没有标准就没有工业现代化，没有标准就没有编程组件化，这就是标准的意义；回到 UML 来说，就是统一的意义。

目前，随着软件工程的不断成熟，软件开发越来越朝着专业化和横向分工化发展。以前人们认为从需求到代码是一个紧密联系的过程，是不可分离的。一旦分开就会导致高成本和高技术风险。然而与现代工业的分工越来越细致和专业化的趋势一样，软件行业的需求、分析、设计、开发这些过程也被分离开来并专业化了。需求由专门的需求团队来做，甚至会委托给一个咨询公司；分析由专门的系统分析团队来做；设计由专门的设计团队来做……以往，开发人员是项目的中心，一个开发人员常常从需求一直做到编码；而现在，程序员只负责根据设计结果来编码，设计师只负责根据需求分析结果来设计，项目组里还有架构师、质量保证小组等许多角色，各自负担着自己的职责要求，在软件工程的约束下相互协作来完成一个项目。软件开发工作被横向分工化的一个显著的例子便是软件外包，承包商采集需求，设计团队进行设计，然后把编码工作外包给另一个公司来完成。

软件开发工作中这种将角色细分，将职责明确的做法，在提高专业化和资源效率的同时也带来了严重的沟通问题。假如承包商采用一种自己的方法来做需求，设计团队由于不熟悉这种方法，在理解需求文档的过程中就会产生误解；如果编码团队也不熟悉设计师的设计文档，很容易再次产生信息歧义。文档从一个角色传向另一个角色，从一个组织传向另一个组织的过程中如何保证信息被准确地传达和准确地理解呢？一种好办法就是大家都使用统一的或者说标准化的语言。UML 统一建模语言的意义也正在于此，它试图用统一的语言来覆盖整个软件过程，让不同的团队操着同一个口音顺畅地沟通。

统一语言的另一个意义是要让人和机器都能读懂。

好，统一的任务完成了，接下来的任务就是可读性。如果语言可读性很差，人们在理解起来同样会有困难。一门好的语言要能够让人们快速理解并留下深刻印象。

我们知道，相对文字和图形，人脑对图形的接受能力显然更强。因此，UML 采用了“可视化”的图形方式来定义语言。

1.2.3 可视化

可视化，这是一个奇怪的词，什么东西不都是可见的吗？UML 是可视化的，用文字写的文档不也是可视化的吗？在这里可视化的含义并不是指 UML 的图形是可以用眼睛看到的，可视化的含

义是指, UML 通过它的元模型和表示法, 把那些通过文字或其他表达方法很难表达清楚的, 隐晦的潜台词用简单直观的图形表达和暴露出来, 准确而直观地描述复杂的含义。把“隐晦”的变成“可视”的, 也就是把文字变成图形, 这才是 UML 可视化的真正含义。

举个例子, 有一段文字描述: 造一辆车身是红色金属漆的小轿车, 装备四个普利斯通牌的轮胎, 它是一辆四门车, 车门是加厚的, 并且前后门玻璃上贴黑色的膜。前后挡风玻璃里都装有电热丝, 后视镜是电动可调的。

这段话很简单, 初看上去简单明了, 任谁都可以一下子就看明白。这是因为汽车是我们很熟悉的事物。如果一个没有看到过汽车的人要靠这段话去真正造一辆汽车一定会觉得很多地方没有讲清楚, 少了很多信息。比方说, 它是一辆四门车这句话就有疑问, 并没有说明白车门是装在哪里的。当然读者可以说那我多加几行字不就说明白了嘛, 当然是可以的, 不过您要是真的对自己的文字功底如此有自信, 您可以试着用文字描述一下图 1.3, 看看您觉得多少文字可以讲清楚, 并且让看的人能明白。

显然如果信息点比较多, 而且相互之间有关系, 阅读文字并不容易让人理解到底描述了怎样的一个逻辑结构。如果是面对更加复杂的业务需求, 书写或阅读长达几十页的文字, 要把所有信息都关联起来并准确理解, 就更困难了。用文字表达风花雪月是很美的, 朦胧美、想象美。但是要用来说明一个结构还真不是件容易的事, 我们总不能靠朦胧和想象去做软件吧? 其实在文字还未出现以前人类就学会绘画了, 一幅图画可以表达的含义远远胜过文字描述, 上面的那段话让我们试着换一种形式来表达, 如图 1.5 所示。

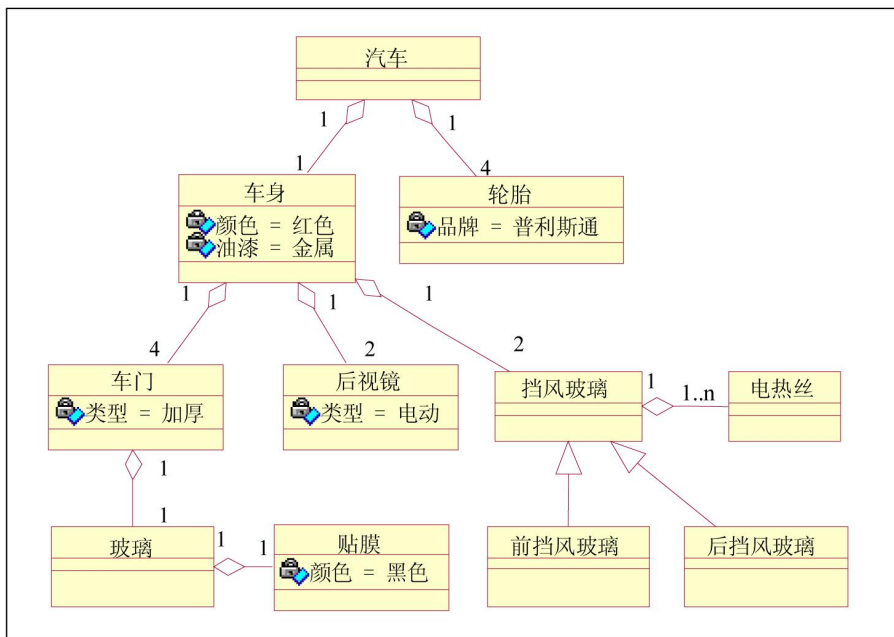


图 1.5 汽车的 UML 表述



显然图 1.5 所表达出来的含义远远超出了文字所描述的内容，除了文字描绘的各个汽车部件之外，更重要的是表达出了这些部件之间的组装关系。例如车身和轮胎是安装在汽车上的，汽车有一个车身和 4 个轮胎。车门是安装在车身上的，一个车身有 4 个车门等等。同时，图 1.5 还表达出了文字里隐晦的含义，使得文档的作者和读者之间不至于产生歧义。比如“一辆车身是红色金属漆的小轿车”这样简单的一句话，就隐含了汽车与车身的关系以及车身的属性。如果这些隐含的信息不能清楚地表达出来，那就可能会在沟通过程中产生歧义。事实上，实际的项目中由于表达和理解之间的歧义导致的返工绝不在少数。图形的优势就在这里，如果非要用一段描述性文字把隐含的意思全表达出来，并一点歧义没有，这种文字的可阅读性是值得怀疑的。想象一下有多少人愿意阅读严谨但枯燥的法律书籍呢？如果把本书中的大量图片拿走只剩下文字的话，相信读者对本书的阅读兴趣也会随着图片被一起带走的。

好，UML 统一了语言，让隐晦的含义可视化了。接下来，这种统一的可视化语言又如何来描绘现实世界并解决面向对象的困难呢？

让我们从第一步，现实世界到业务模型开始吧。

注：以下描述的解决面向对象的过程采用了 RUP 方法，实际上 UML 本身并不是一定要采用 RUP 方法的。关于这一点，在 1.3 节统一过程简介中会详细阐述语言和方法的关系。

1.2.4 从现实世界到业务模型

建立模型是人们解决现实世界问题的一种常用手段。我们通常接触到的建模是为了解决某个问题而建立的一个数学模型，通过数学计算来分析和预测，找出解决问题的办法。从理论上说，建立模型是指通过对客观事物建立一种抽象的方法，用来表征事物并获得对事物本身的理解，再把这种理解概念化，并将这些逻辑概念组织起来，形成对所观察的对象的内部结构和工作原理的便于理解的表达。模型要能够真实反映客观事物就需要有一个论证过程，使得模型建立过程是严谨的，并且结果是可追溯和验证的。对于一种软件建模方法来说，为现实世界建立逻辑模型也要是严谨的、可追溯和可验证的，除了描述清楚需求，还要能很容易地将这个模型转化为计算机也能够理解的模型。

我们所处的这个现实世界充满了丰富多彩但杂乱无章的信息，要建立一个模型并不容易。建立模型的过程是一个抽象的过程，所以要建立模型，首先要知道如何抽象现实世界。如果我们站在很高的抽象层次，以高度归纳的视角来看这个世界的运作，就会发现现实世界无论多复杂，无论是哪个行业，无论做什么业务，其本质无非是由人、事、物和规则组成的。人是一切的中心，人要做事，做事就会使用一些物并产生另一些物，同时做事需要遵循一定的规则。人驱动系统，事体现过程，物记录结果，规则是控制。建立模型的关键就是弄明白有什么人，什么人做什么事，什么事产生什么物，中间有什么规则，再把人、事、物之间的关系定义出来，一个模型也就基本成型了。

那么 UML 是不是提供了这样的元素来为现实世界建立模型呢？是的。

第一，UML 采用称之为参与者（actor）的元模型作为信息来源提供者，参与者代表了现实世界的“人”。参与者是模型信息来源的提供者，也是第一驱动者。



参与者

换句话说,要建立的模型的意义完全被参与者决定,所建立的模型也是完全为参与者服务的,参与者是整个建模过程的中心。UML 之所以这样考虑,是因为最终计算机的设计结果如果不符合客户需求,再好的设计也等于零。与其在建立计算机系统后因为不符合系统驱动者的意愿而推倒重来,还不如在一开始就从参与者的角度为将来的计算机系统规定好它必须实现的那些功能和必须遵守的参与者的意志,由驱动者来检验和决定将来的计算机系统要如何运作。

另外,在这个顾客就是上帝的时代,以参与者也就是“人”为中心还顺应了“以人为本”这一时代的要求,更容易获得客户满意度。



用例

第二, UML 采用称之为用例 (use case) 的一种元模型来表示驱动者的业务目标,也就是参与者想要做什么并且获得什么。这个业务目标就是现实世界中的“事”。而这件事是怎么做的,依据什么规则,则通过称之为业务场景 (business scenario) 和用例场景 (use case scenario) 的 UML 视图来描绘的,这些场景便是现实世界中的“规则”。最后, UML 通过称之为业务对象模型 (business object model) 的视图来说明在达成这些业务目标的过程中涉及到的事物,用逻辑概念来表示它们,并定义它们之间的关系。业务对象模型则代表了现实世界中的“物”。

人、事、物、规则就是这样被模型化的。如果您现在对这些概念和过程还不是很理解,没关系,本节里只是为了说明 UML 如何为现实世界建模,在本书的后续章节中您将详细了解到这一建模方法,这里只是简单地引用了这些 UML 名词。

UML 通过上面的元模型和视图捕获现实世界的人、事、物和规则,于是现实信息转化成了业务模型,这也是面向对象方法中的第一步。业务模型真实映射了参与者在现实世界的行为,图 1.6 展示了这种映射关系。

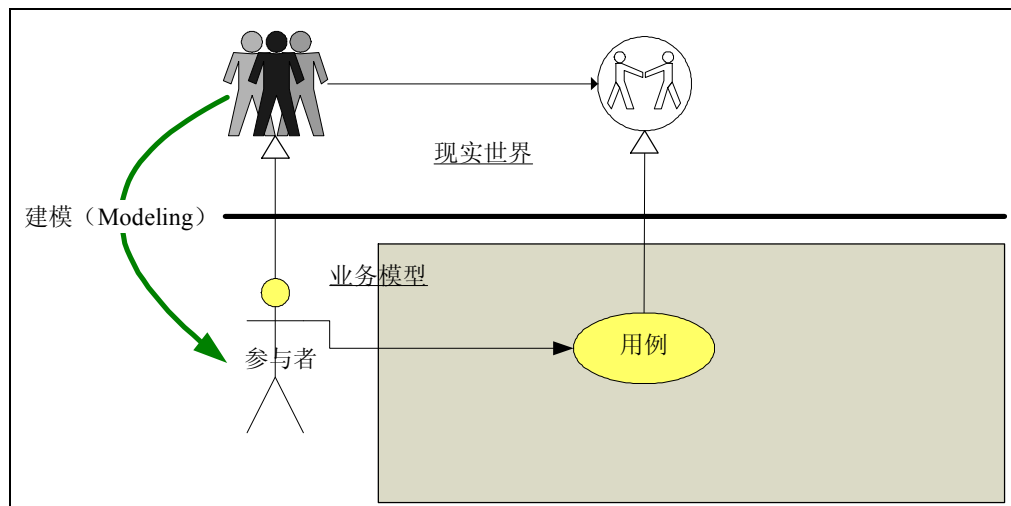


图 1.6 从现实世界到业务模型

得到业务模型仅仅是一个开始,要想将业务模型转化到计算机能理解的模型,还有一段路要走。






这其中最重要的一步便是概念模型。下一节将讨论业务模型到概念模型的转化。

1.2.5 从业务模型到概念模型

虽然上一节中现实世界被业务模型映射并且记录下来，但这只是原始需求信息，距离可执行的代码还很遥远，必须把这些内容再换成一种可以指导开发的表达方式。UML 通过称之为概念化的过程 (Conceptual) 来建立适合计算机理解和实现的模型，这个模型称为分析模型 (Analysis Model)。分析模型介于原始需求和计算机实现之间，是一种过渡模型。分析模型向上映射了原始需求，计算机的可执行代码可以通过分析模型追溯到原始需求；同时，分析模型向下为计算机实现规定了一种高层次的抽象，这种抽象是一种指导，也是一种约束，计算机实现过程非常容易遵循这种指导和约束来完成可执行代码的设计工作。

事实上分析模型在整个分析设计过程中承担了很大的职责，起到了非常重要的作用。绘制分析模型最主要的元模型有：

- 边界类 (boundary) 。边界是面向对象分析的一个非常重要的观点。从狭义上说，边界就是大家熟悉的界面，所有对计算机的操作都要通过界面进行。从广义上说，任何一件事物都分为里面和外面，外面的事物与里面的事物之间的任何交互都需要有一个边界。比如参与者与系统的交互，系统与系统之间的交互，模块与模块之间的交互等。只要是两个不同职责的簇之间的交互都需要有一个边界，换句话说，边界决定了外面能对里面做什么“事”。在后续的章节中，读者会感受到边界的重要性，边界能够决定整个分析设计的结果。
- 实体类 (entity) 。原始需求中领域模型中的业务实体映射了现实世界中参与者完成业务目标时所涉及的事物，UML 采用实体类来重新表达业务实体。实体类可以采用计算机观点在不丢失业务实体信息的条件下重新归纳和组织信息，建立逻辑关联，添加那些实际业务中不会使用到，但是执行计算机逻辑时需要的控制信息等。这些实体类可以看作是业务实体的实例化结果。
- 控制类 (control) 。边界和实体都是静态的，本身并不会动作。UML 采用控制类来表述原始需求中的动态信息，即业务或用例场景中的步骤和活动。从 UML 的观点看来，边界类和实体类之间，边界类和边界类之间，实体类和实体类之间不能够直接相互访问，它们需要通过控制类来代理访问要求。这样就把动作和物体分开了。考虑一下，实际上在现实世界中，动作和物体也是分开描述的。

读者或许在小时候都玩过一个游戏，每个同学发四张小纸条，在第一张纸条上写上 XXX 的名字，在第二张纸条上写上在什么地方，在第三张纸条上写上一个动作，在第四张纸条上写一个物体，然后将这些字条分开放在四个箱子里，再随意地从这四个箱子里各取一张纸条，就能组成很多非常搞笑的句子，例如张 XX 在公园里跳圆规之类的奇怪语句，一个班的学生常常笑得前仰后合。

游戏虽然是游戏，但说明了一个道理，只要有人、事、物和规则（定语），就能构成一个有意

义的结果，无非是是否合理而已。分析类也是应用这个道理来把业务模型概念化的。由于所有的操作都通过边界类来进行，能做什么不能做什么由边界决定，所以边界类实际上代表了原始需求中的“事”；实体类则由业务模型中的领域模型转化而来，它代表了现实世界中的“物”；控制类则体现了现实世界中的“规则”，也就是定语；再加上由参与者转化而来的系统的“用户”，这样一来，“人”也有了。有了人、事、物、规则，我们就可以像那个游戏一样把它们组合成各种各样的语句，只不过不是为了搞笑，所以不能随意组合，而是要依据业务模型中已经描绘出来的用例场景来组合这些元素，让它们表达特定的业务含义。

另外，在这个阶段，还可以对这些分析类在不同的视角上进行归纳和整理，以表达软件所要求的一些信息。例如包、组件和节点（详细内容参阅第二部分的第3章 UML 核心元素）。软件架构和框架也通常在这个阶段产生。

图 1.7 展示了从业务模型到概念模型的转化过程。

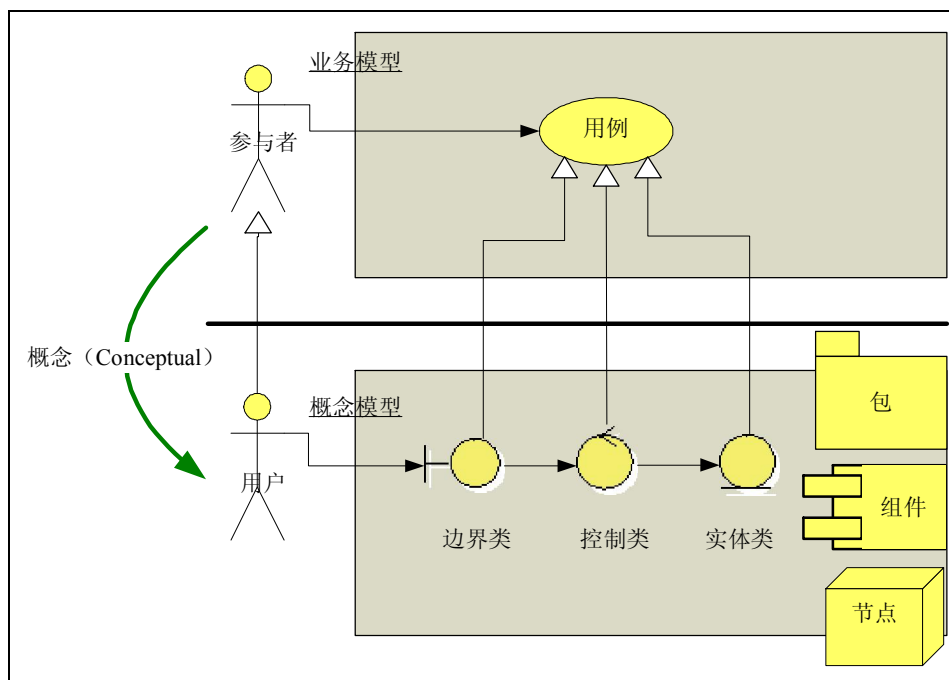


图 1.7 业务模型到概念模型

经过概念模型的转换，业务模型看起来对计算机来说可理解了。但是要得到真正可执行的计算机代码，我们还有一步要走。我们需要将概念模型实例化，即再次转化为计算机执行所需要的设计模型。下一节将讨论概念模型到设计模型的转化问题。

1.2.6 从概念模型到设计模型

上一节中建立的概念模型距离可执行代码已经非常接近了。概念模型使我们获得了软件的蓝



图，获得了建设软件所需要的所有组成内容以及建设软件所需要的所有必要细节。这就类似于我们已经在图纸上绘制出了一辆汽车所有的零部件，并且绘制出如何组装这些零部件的步骤，接下来的工作就是建造或者购买所需的零部件，并送到生产线去生产汽车。

设计模型的工作就是建造零部件，组装汽车的过程。在大多数情况下，实现类可以简单地从分析类映射而来。在设计模型中，概念模型中的边界类可以被转化为操作界面或者系统接口；控制类可以被转化为计算程序或控制程序，例如 workflow、算法体等；实体类可以转化为数据库表、XML 文档或者其他带有持久化特征的类。这个转化过程也是有章可循的，一般来说，可以遵循的规则有：

- 软件架构和框架。软件架构和框架规定了实现类必须实现的接口、必须继承的超类、必须遵守的编程规则等。例如当采用 J2EE 架构时，Home 和 Remote 接口就是必需的。
- 编程语言。各类编程语言有不同的特点，例如在实现一个界面或者一个可持久化类时，采用 C++ 还是 Java 作为开发语言会有不同的设计要求。
- 规范或中间件。如果决定采用某个规范或采用某个中间件时，实现类还要遵循规范或中间件规定的那些必需特性。

实际上，由于软件项目可以选择不同的软件架构和框架，可以选择不同的编程语言，也可以选择不同的软件规范，还可以购买不同的中间件，因此同样的概念模型会因为选择不同而得到不同的设计模型。图 1.8 展示了从概念模型到设计模型的转化过程。

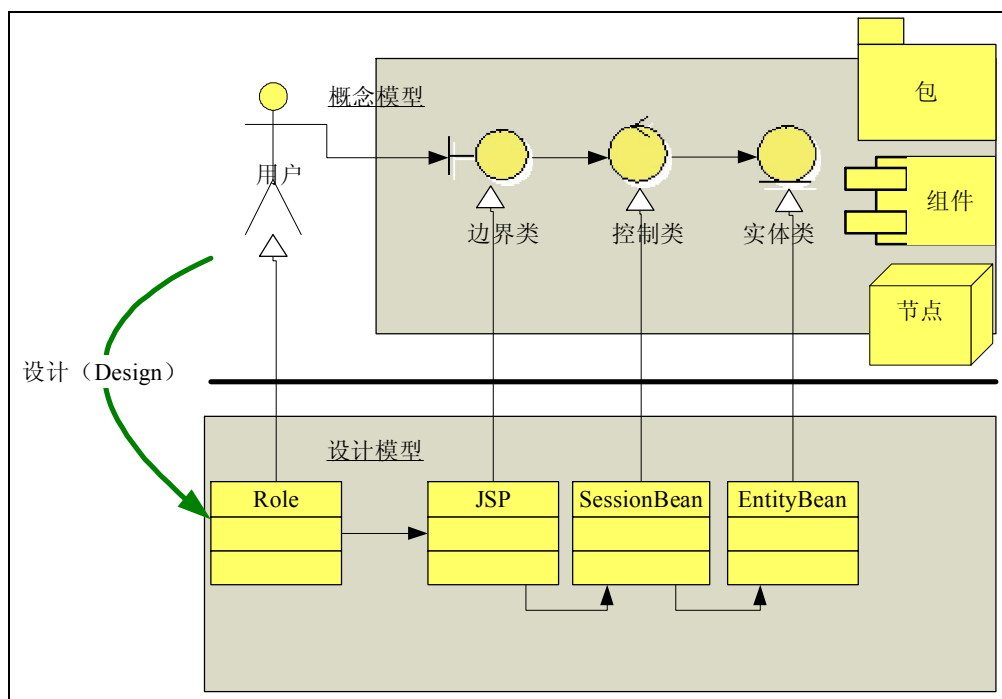


图 1.8 从概念模型到设计模型

经过上述的转化过程,我们回头来看看,UML 是否解决了我们在 1.1.4 面向对象的困难一节中提出的问题?

1.2.7 面向对象的困难解决了吗

经过上面三个模型的转化,让我们再回顾一下 1.1.4 节中所谈到的面向对象的困难。在 1.1.4 节中提到,要解决面向对象的困难我们需要这样一些方法:

- 一种把现实世界映射到对象世界的方法。
- 一种用对象世界描述现实世界的方法。
- 一种验证对象世界行为正确反映了现实世界的方法。

那 UML 是否解决了面向对象的困难或者说有没有提供出我们需要的方法呢?下面来验证一下。

1.2.7.1 从现实世界到业务模型

这是把现实世界映射到对象世界的第一步。UML 采用用例这一关键元素捕获了现实世界的人要做的事,再通过用例场景、领域模型等视图将现实世界的人、事、物、规则这些构成现实世界的元素用 UML 这种语言描述出来。而 UML 本身被设计成为一种不但适于现实世界理解,也适于对象世界理解的语言,所以用 UML 来描述现实世界这句话可以稍微换一下说法,变成:现实世界被我们用一种对象型语言描述。

这不正是我们需要的把现实世界映射到对象世界的方法吗?看来,我们找到了一种把现实世界映射到对象世界的方法。

1.2.7.2 从业务模型到概念模型

这是从对象世界来描述现实世界的方法。当业务模型用分析类来描述的时候,我们实际上已经采用了对象视角。用例所代表的现实的业务过程,被“边界”、“控制”、“实体”以及“包”、“组件”等概念替代。而这些概念是可以被计算机理解的,是抽象化了的对象。现实世界千差万别的业务,都用“边界”、“控制”、“实体”这几个固定的元素来描述,也就是说,现实具体的业务被“抽象”成几个固定的概念。同时,这些概念还可以用“包”、“组件”等这些与现实世界毫不相关的纯计算机逻辑术语包装。这说明概念模型是计算机视角,或者说是对象视角的,而且这些对象视角的分析类所描述的信息是从映射了现实世界的业务模型转化而来的。

可以说,从业务模型到概念模型这一过程,正是我们需要的一种从对象世界来描述现实世界的方法。

1.2.7.3 从概念模型到设计模型

这是验证对象世界是否正确反映了现实世界的方法。“边界”、“控制”、“实体”这些对象化的概念,虽然是计算机可以理解的,但它并不是真正的对象实例,即它们并不是可执行代码,概念模型只是纸上谈兵。真正的对象世界行为是由 Java 类、C++类、EJB、COM+等这些可执行代码构成的。然而,如果缺少了从概念模型到设计模型这个过程,Java 类也好,C++类也好,它们的行为是否正确凭什么去验证呢?图 1.8 展示的信息指出,设计类并不是像孙悟空一样从石头缝里蹦出来的,而是用某种语言在某种特定规范的约束下“实例”化分析类的结果。换句话说,设计模型是概念模



型在特定环境和条件下的“实例”化，实例化后的对象行为“执行”了概念模型描述的那些信息，因此设计模型得以通过概念模型追溯到原始需求来验证对象世界是否正确反映了现实世界。

看来，我们找到了一种验证对象世界是否正确反映了现实世界的方法。

如果把三个模型的建立过程综合起来，形成图 1.9，从中我们可以更清楚地看到面向对象的困难是如何在模型的转化过程中解决的。这一过程看来是有规律、可推导、可追溯的过程。

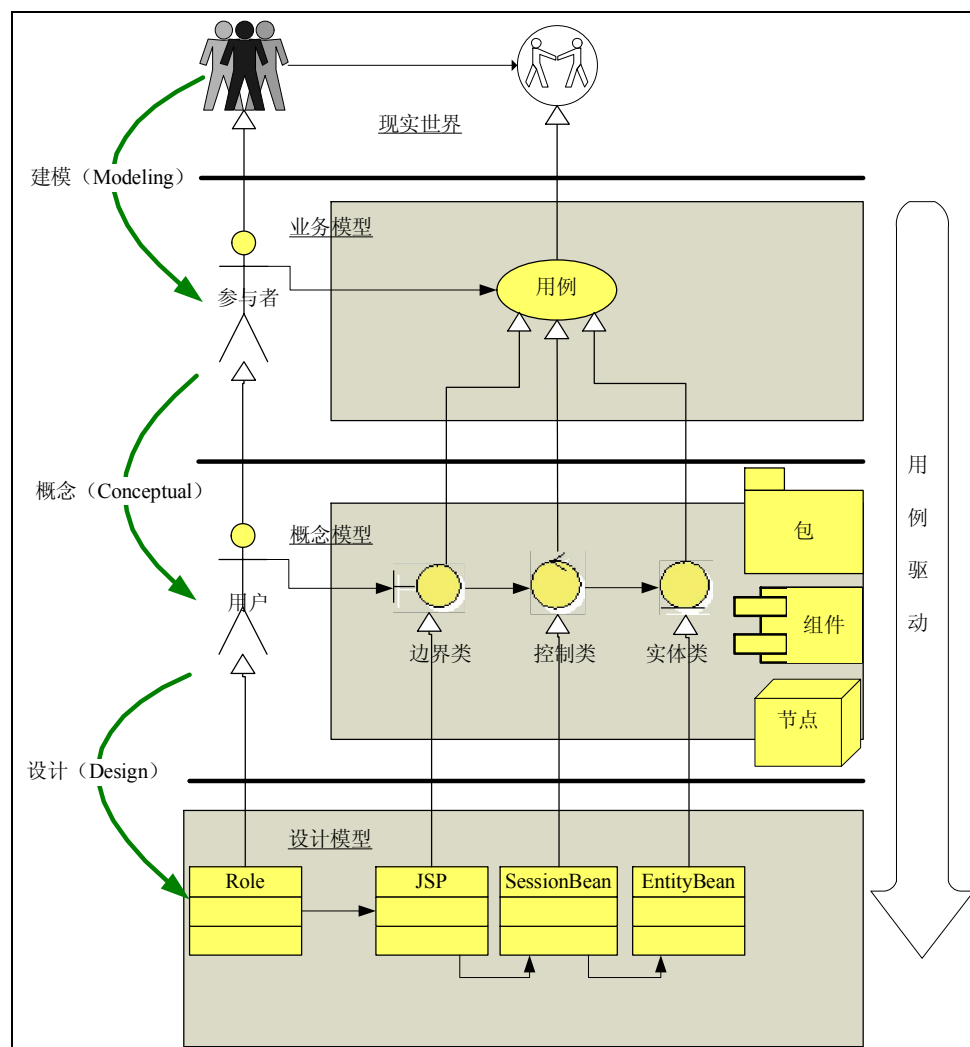


图 1.9 面向对象分析设计的完整过程

在图 1.9 中引入了另一个概念，就是用例驱动。尽管在这一章并没有讲述这个概念，但是相信读者也能从中初步地领略到用例是如何驱动整个开发过程的。请读者记住这幅图，本书后续大量的篇幅都是由用例驱动来完成整个软件过程的。

UML 作为标准的面向对象建模语言，它需要在某个建模方法的指导下进行建模工作。正如我们学会了一门语言，还需要知道文体一样。统一过程是这其中最为著名的建模方法，下一节就来初步了解一下统一过程。实际上，统一语言加上统一过程就构成了本书的主线，也就是读者将要学习到的建模方法。

1.3 统一过程简介

谈到 UML 不能不谈到统一过程，即 RUP。UML 和 RUP 师出同门，尽管目前仍有许多其他的建模方法，不过 RUP 仍然是其中对 UML 使用最为全面的，同时也是最为复杂的。本书的中心是 UML，不能够深入讨论 RUP，本节只介绍 RUP 与 UML 有关的基本知识以帮助理解本书的内容。

1.3.1 RUP 是什么

严格说起来 UML 并不是一个方法，而只是一种语言。UML 定义了基本元素，定义了语法，但是如果要做软件项目，还需要有方法的指导。正如写文章有文法，有五言律，有七言律一样，UML 也需要有方法的指导来完成一个软件项目。RUP 无疑是目前与 UML 集成和应用最好、最完整的软件方法。

RUP（Rational Unified Process）译为统一过程。统一过程并非是因为 UML 才诞生的，也不是最近才出来的软件方法，而是有着很长时间的的发展，有着很深的根源。统一过程归纳和整理了很多在实践中总结出来的软件工程的最佳实践，是一个采用了面向对象思想，使用 UML 作为软件分析设计语言，并且结合了项目管理、质量保证等许多软件工程知识综合而成的一个非常完整和庞大的软件方法。统一过程经过了三十多年发展，和统一过程本身所推崇的迭代方法一样，统一过程这个产品本身也经过了很多次的迭代和演进，才最终推出了现在这个版本。图 1.10 展示了统一过程的演进过程。

统一过程归纳和集成了软件开发活动中的最佳实践，它定义了软件开发过程中最重要的阶段和工作（四个阶段和九个核心工作流），定义了参与软件开发过程的各种角色和他们的职责，还定义了软件生产过程中产生的工件（见注），并提供了模板。最后，采用演进式软件生命周期（迭代）将工作、角色和工件串在一起，形成了统一过程。

工件：工件也称为成果物或者制品（Artifact），这与可交付物（Deliverable）是有一些差别的。当某一个或某一些工作是最终产品的一部分需要交付出去时，才被称为可交付物。而在软件生产过程中任何留下记录的事物，都可称为工件。

图 1.11 摘自统一过程的官方文档，展示了统一过程的总体概述。

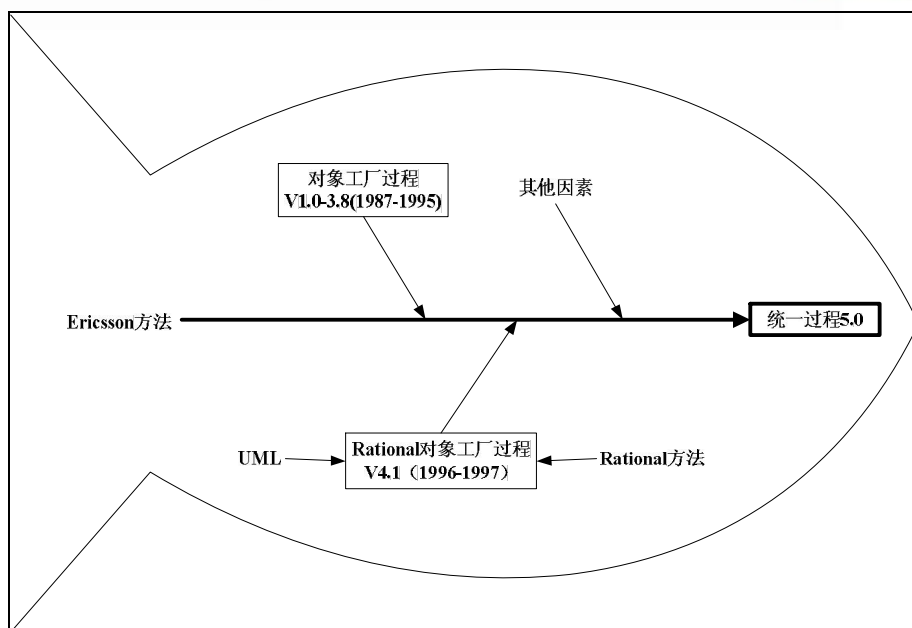


图 1.10 RUP 的历史演进过程

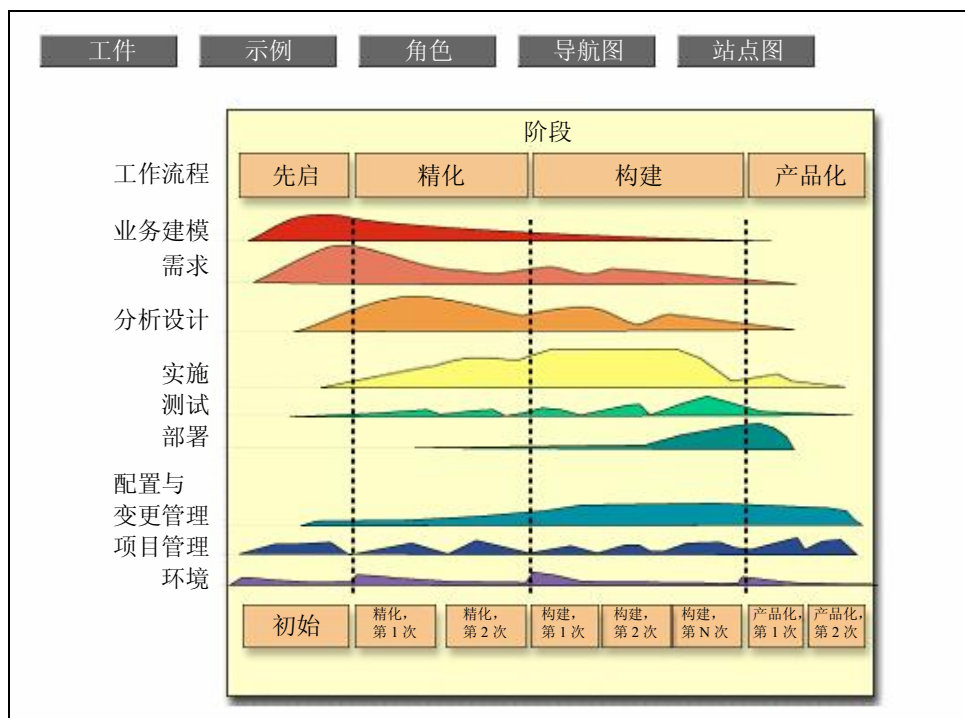


图 1.11 统一过程概述

从图 1.11 中读者可以看到, 统一过程将软件生产分为了四个阶段和九个核心 workflow, 每个 workflow 在不同的阶段有不同的工作量比重, 这些数据来自实际项目的数据统计。每个阶段都会做哪些 workflow 呢? 这由迭代计划来决定。一个软件从开始到产品推出要经过多次的演进, 每一个演进会有一个迭代计划来描述这次演进要达成的目标、要经历的阶段以及要进行的工作流。统一过程对每一个 workflow 都规定了标准的流程、参与角色和工件模板, 而在迭代计划里可以依据实际情况对这些流程、角色和模板进行裁剪。

这里只对统一过程进行了简单的介绍, 但统一过程不可避免地在这本书中占有很重要的地位。事实上, 本书中所讲述的用 UML 来分析和设计的所有方法都来自于统一过程。因此, 本书不会展开详细地阐述统一过程, 读者阅读本书的过程实际上正是在经历着统一过程, 阅读完本书, 读者应当已经掌握了最基本的统一过程方法, 尤其是业务建模、需求、分析设计这几个核心 workflow。

1.3.2 RUP 与 UML



如果说一曲美妙的乐章是作曲家根据音乐理论进行创作最后用标准的五线谱记录下来, 相信不会有什么疑问。实际上 RUP 与 UML 的关系正类似音乐理论和五线谱的关系。相信有很多读者并没有考虑过这个问题, 他们会觉得统一过程和 UML 就是一个东西, 统一过程就是 UML, UML 就是统一过程。这个错误的认识其实是因为统一过程采用了 UML 作为基本语言, 再加上统一过程和 UML 都来自三位面向对象大师的研究成果, 都出自 Rational 公司 (见注)。但是从本质上说, 统一过程和 UML 是不同的两个领域。UML 是一种语言, 用来描述软件生产过程中要产生的文档, 统一过程则是指导如何产生这些文档以及这些文档要讲述什么的方法。虽然现在统一过程是指导 UML 的方法中最著名、应用最广、可能也是最成功的一个, 但这两者却不是完全不可以分开的。

注: Rational 目前已被 IBM 收购, Rational 产品也成为 IBM 五大产品家族之一。

认识到这一点, 许多读者的一些疑惑就能解开。所有初学 UML 的朋友都会以为学习 UML 必须同时学习统一过程, 而由于统一过程本身的庞大和复杂, 成为了学习 UML 的障碍。更多的 UML 学习者不知道 UML 该怎么用, 在什么地方用, 在什么时候用, 用 UML 的目标是什么。相信初学者最大的疑惑就是这些吧?

事实上笔者自己在学习 UML 初期阶段也曾经非常迷惑而不得要领, 这么多 UML 元素, 每个都有其特定的含义, RUP 中定义了更多更复杂的流程、模板、工具……虽然读了很多资料, 却始终感觉 UML 的信息太过于分散, 不能很好地把 UML 应用到实际的项目中去。直到有一天笔者认识到 RUP 和 UML 并不是天生一体的, 它们只是软件方法和建模语言的一个完美结合, 然后更进一步认识到其实软件过程是比 UML 更重要更本质的东西; 于是转变了思维, 不再把 UML 和 RUP 混在一起造成方法和语言混淆不清, 而是站在软件过程的角度, 先了解一个软件项目是怎么做的, 再去 UML 中寻找需要的工具, 用 UML 中适合的工具把软件过程要达到的要求记录下来。



正是这一转变使笔者对 UML 的认识茅塞顿开，实际上不仅仅是 RUP 和 UML 的关系，对软件项目来说，面向对象也好，面向过程也好，UML 也好，UC 矩阵也好，这些都不是最重要的，软件项目真正的灵魂是软件过程，软件过程的需要才是这些工具和语言诞生的原因。因此建议读者在学习 UML 之前，应当先系统学习软件过程。只有掌握了软件过程，才会知道为什么要有用例，为什么要有分析模型；站在软件过程的立场，那些孤独的 UML 视图才会变得有生命力，才会知道在什么时候，在什么地方需要用什么样的 UML 图符来表达软件的观点，也才会知道 UML 的那些视图到底在软件开发过程里起到了什么作用。认识到这些，UML 的元模型和视图就不会再面目可憎，它们是一群有着强大能力的精灵，帮助你在复杂的软件工程道路上搭起一座座通向光明目标的桥梁。

不过笔者的意思并不是让读者把统一过程和 UML 完全分开学习或只学习其中一个，而是告诉读者一个学习 UML 的方法。目前为止统一过程仍然是与 UML 结合最广的方法，毕竟师出同门，有着天然的内在联系，学习统一过程能更好地理解 UML，这也是本书采用统一过程作为方法来讲述 UML 的原因。

1.3.3 RUP 与软件工程

统一过程方法是一个庞大和复杂的知识体系，它几乎囊括了软件开发这一生产过程所需要知识的方方面面。但同时，也正由于统一过程的复杂和庞大，使得它学习起来很困难；要在实际项目中实施统一过程也非常困难，统一过程也是迄今为止最重量级的软件方法。统一过程是一种追求稳定的软件方法，它追求开发稳定架构，控制变更，立足于长期战略，适用于指导大中型软件产品的开发。由于统一过程是一种重量级的方法，因此实施统一过程是高成本的，是一个组织战略的选择，而不仅仅是某一个项目的战术选择。实施统一过程不但需要在初期投入庞大的学习成本，也需要在实施过程中投入庞大的管理成本。那我们为什么要投入那么多成本来实施统一过程呢？

一方面是出于提高软件成熟度的需要。我们知道 CMM 只规定了每个成熟度等级的评估标准，但没有规定如何做才能达到这一标准。与其耗费更大的成本去摸索自己的软件过程，不如采用统一过程这种已经集成了软件活动最佳实践的成熟的软件过程。实际上能够实施统一过程大致已经相当于达到了 CMM 二级到三级的水平。

另一方面是出于提高软件技术水平和质量的需要。我们知道要生产一个好的软件产品，必须保证需求、分析、设计、质量等工作。统一过程由于集成了面向对象方法、UML 语言、核心 workflow、工件模板和过程指导等许多知识，使得软件生产工作能够利用这些成熟的指导来提高组织内整体软件认识水平和开发人员的软件素质，借此来提高软件产品的技术水平和质量。

再一方面，统一过程适于开发稳定的架构，它通过不断的演进来逐步推进软件产品，这一特点使得它特别适合于长期战略的软件产品。例如那些长期立足于做某一个行业，希望做精做深，做行业整体解决方案的组织。

但是统一过程由于太过于庞大和复杂，相对于轻量级的敏捷方法，例如著名的 XP（极限编程）方法来说，显得死板和难以实施。统一过程不但不能快速适应需求的变化，而且变更一个需求要经

历复杂的过程和很多额外的工作。对于较小的组织和项目来说,使用统一过程的确有些费力不讨好,也因此招来了许多置疑。

笔者觉得这种置疑是大可不必的,轻量级的敏捷方法和重量级的统一过程都是非常优秀的软件方法,只是它们各有各的适用范围。轻量级的 XP 方法追求在变化中用最快速的办法适应变更,用小的管理成本保障软件质量。对于中小型公司和中小型软件来说,XP 的确是非常有效的软件方法,它能大大降低管理、开发成本和技术风险。不过对于大型公司和大型项目来说,XP 就不适用了,这时 RUP 却非常适合。因为对大型项目来说,一个项目有可能经历几年甚至几十年的时间,涉及几千甚至几万人,如果没有一个稳定的架构,在朝令夕改的方式下项目是不可能顺利实施下去的。

你能想象洛克希德·马丁公司用 XP 的方法来开发 F-35 战斗机会是一个什么情形吗?没有人清楚地知道将来飞机的整体是什么样,好不容易设计出机翼来,另一个小组说我们决定改变一下气动外形,你们再重构一下吧;没有人知道最后飞机的性能怎么样,反正先造一架出来,要是摔了找找原因,改进改进,重构一下,再造一架……再摔了,没关系,咱们拥抱变更,再造就是了……

显然对这种大型产品来说 XP 方法是不可接受的,而 RUP 的稳步推进的方法却正好适合。那 XP 什么情况下适用呢?如果你是一个杂货店的老板,刚开业不知道什么样的商品受欢迎,没关系,先各进一小批货,卖上一段时间,受欢迎的货品多进,不受欢迎的不进,随时向顾客做一些调查,顾客喜欢什么商品就进什么,不断改进,最后一定会顾客盈门的。这时如果这个老板采用统一过程的方法,先做商业分析、客户关系分析、消费曲线分析……还没开业呢,估计就破产了,或者好不容易做出了一个商业策略,客户兴趣已经改变了。

另外,RUP 和 XP 也不是非此即彼的关系,比如在造 F-35 的过程中,对整体飞机来说,用 RUP 是适合的,具体到零部件倒是大可 XP 一把,先在风洞里试验试验,不符合条件就更换了再试,最终只要得到最适合的零部件就 OK 了。

上一节讲到 RUP 和 UML 是可以分离的,所以读者完全可以根据自己的实际情况来决定采用哪种软件方法,而采用哪种软件方法其实并不妨碍使用 UML 来做软件的分析 and 设计。

1.3.4 RUP 与最佳实践

如今软件产品越来越复杂,越来越庞大。长久以来,人们期望软件开发能够像其他工业产品一样,可以单独生产标准零部件,然后按照要求来组装它们,用较少的投入完成最终的软件产品。现在随着面向对象的发展,基于架构的、构件式的软件开发模式已经成为软件开发的主流,这代表着软件从“手工业”向“工业”的转化取得了重大的进步。

但是构件式开发并非想象的那么乐观,构件的生产以及构件的组装还面临着许多困难。困难一方面来自于构件并不像工业产品那样容易定义,另一方面软件构件的组装(接口+标准)也远远不像把螺丝钉拧入螺丝帽那样容易。现实世界很复杂,要把现实世界用逻辑构件复制下来首先需要非常精确的抽象,这是一项很不容易的工作;但即使是做到了精确的抽象也还不够。举个例子,对于工业产品来说,人们预先确定并接受了它的用途,没有人会要求自行车生产商让自行车飞起来,然而对软件来说,出现这种类似的要求却并不离奇。因此,除了精确的抽象,构件还需要有“自适应”



和“自我成长”的能力。

人们做了很多努力，可惜现在软件技术还离这个目标很远，但是人们已经积累了足够多的知识和经验，形成了很多针对普遍问题的解决方案。虽然这些经验还不能够让自行车飞起来，不过要生产出既能够用电力驱动也能够用人力驱动的自行车却是完全有可能的。这些知识和经验就是最佳实践。对于软件产品来说，最佳实践来自两个方面：一方面是技术类的，如设计模式；另一方面的是过程类的，如需求方法、分析方法、设计方法等。

统一过程集成了很多过程类的最佳实践，这些最佳实践中包括用例驱动、架构导向、构件化等。另外，统一过程不仅仅集成了软件过程的技术方面的内容，还集成了大量的管理方面的内容，涉及到了软件工程的方方面面。可以说是目前为止最全面、最广泛、最综合的软件体系。因此笔者建议读者认真学习统一过程。学习统一过程的目的一定要在项目中去实施，因为上一节讲到实施统一过程并不容易，而是因为学习统一过程将了解到软件的本质，对提升软件“智商”是非常有好处的。

从笔者自己的经验来看，通过学习统一过程来全面认识软件是怎么一回事是一个提升自我能力的捷径，不论读者是程序员、设计师、系统分析员、架构师、测试人员、项目经理、需求工程师、配置管理员等等，都将在统一过程中找到自己在软件生产过程中的位置，找到自己的职责，以及与其他角色的互动，进而从更高的层次和整体上提升自身的软件能力。图 1.12 展示了统一过程中包含的主要知识体系。

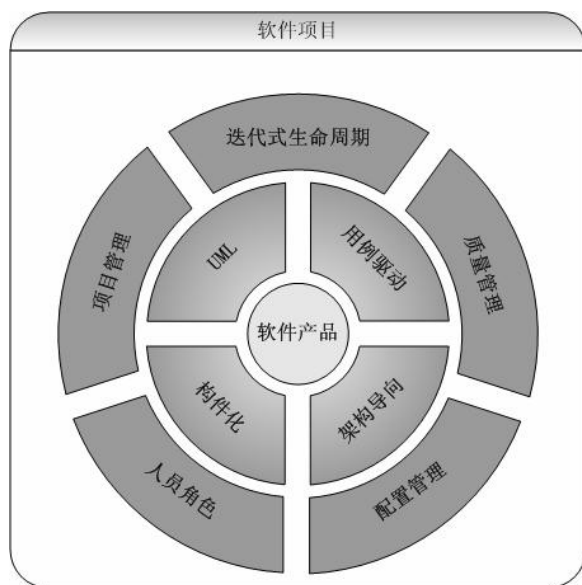


图 1.12 统一过程的最佳实践

1.3.5 RUP 与本书

统一过程在本书中占有非常重要的地位。虽然本书接下来的篇幅中并没有显式地阐述统一过

程，但统一过程却是本书的骨架和脉络。在 1.3.2 节“RUP 与 UML”中我们得知 UML 只是一种语言而已，如果缺少了统一过程，单单只讲述 UML 的话本书就变成了一本解释基础语言的“字典”。事实上，本章讲述的解决面向对象困难的方法来自统一过程，本书第二部分的整个组织结构也是按照统一过程的核心工作流与最主要的工作组织和讲述的。

统一过程学习的困难在于庞大和复杂，而 UML 学习的困难在于不知如何使用。在本书中，笔者避免显式地讲述统一过程以免引起读者在复杂的知识点中迷失，但是会隐式地用统一过程中最主要的线索串起 UML 的知识点，并用这些知识点来贯穿一个项目的绝大多数阶段。因此，读者在阅读本书的同时将潜移默化地学习到统一过程如何在项目中实施，以及 UML 如何在项目中使用的主要知识。

相信大多数 RUP 的初学者会在统一过程浩如烟海的概念、阶段、术语、流程、模板、指南、定义、工具中迷失方向。既然如此，何不干脆忘掉你正身处一片迷失森林，沿着本书中为你趟出的那条羊肠小道领略一下森林的风光吧。虽然还不足以让你了解整个森林，但是已经足以让你敢于独自走进这片森林了。