第2章 UML 介 绍

本章内容：

- UML概述
- 理解UML的3个步骤
- 软件体系结构
- 软件开发过程

统一建模语言（Unified Modeling Language，UML）是一种绘制软件蓝图的标准语言。可以用UML对软件密集型系统的制品进行可视化、详述、构造和文档化。

从企业信息系统到基于Web的分布式应用，甚至严格的实时嵌入式系统都适合于用UML来建模。它是一种富有表达力的语言，可以描述开发所需要的各种视图，然后以此为基础装配系统。虽然UML的表达力很丰富，但理解和使用它并不困难。要学习使用UML，一个有效的出发点是形成下述语言的概念模型，这要求学习三个要素：UML的基本构造块、支配这些构造块如何放置在一起的规则和运用于整个语言的一些公共机制。

UML仅仅是一种语言，并且仅仅是软件开发方法的一部分。UML是独立于过程的，但最好把它用于以用况为驱动、以体系结构为中心、迭代及增量的过程中。

2.1 UML概述

UML是一种对软件密集型系统的制品进行下述工作的语言，这些工作包括：

- 可视化
- 详述
- 构造
- 文档化

1. UML是一种语言

一种语言提供了用于交流的词汇表和在词汇表中组合词汇的规则。而一种建模语言的词汇表和规则则注重于对系统进行概念上和物理上的描述。像UML这样的建模语言正是用于软件蓝图的标准语言。

建模是为了产生对系统的理解。只用一个模型是不够的，相反，为了理解系统（除非非常微小的系统）中的各种事物，经常需要多个相互联系的模型。对于软件密集型系统，就需要这样一种语言，它贯穿于软件开发生命期，并能表达系统体系结构的各种不同视图。【在第1章中讨论建模的基本原理。】

像UML这样的语言的词汇表和规则可以告诉你如何创建或理解结构良好的模型，但它没有告诉你应该在什么时候创建什么样的模型，因为这是软件开发过程的工作。一个定义明确的过程会给出如下的指导：决定生产什么制品；由什么样的活动和人员来创建和管理这些制品，怎样采用这些制品从整体上去度量和控制项目。

2. UML是一种可视化语言

对于很多程序员来说，把实现的思想变成程序代码，其间没有什么距离可言，就是思考和编码。事实上，对有些事情的处理最好就是直接编码。文本是既省事又直接的书写表达式和算法的方式。

在这种情况下，程序员仍然要做一些建模，虽然只是在内心里这样做。程序员甚至可能在白板或餐巾纸上草拟出一些想法。然而，这其中存在几个问题。第一，别人对这些概念模型容易产生错误的理解，因为并不是每个人都使用相同的语言。一种有代表性的情况是，假设项目开发单位建立了自己的语言，如果你是外来者或是加入项目组的新人，你就难以理解该单位在做什么事情。第二，除非建立了模型（不仅仅是文字的编程语言），否则你就不能够理解软件系统中的某些事情。例如，阅读一个类层次的所有代码，虽可推断出它的含义，但不能直接领会它。类似地，在基于Web的系统中研究系统的代码，虽可推断出对象的物理分布和可能移动，但也不能直接领会它。第三，如果一个开发者删节了代码而没有写下他头脑中的模型，一旦他另寻高就，那么这些信息就会永远丢失，最好的情况也只能是通过实现而部分地重建。

用UML建模可解决第三个问题：清晰的模型有利于交流。

对有些事物最好是用文字建模，而对有些事物又最好是用图形建模。的确，在所有引人关注的系统中都有不能仅用编程语言描述就能完全阐述清楚的结构。 UML正是这样的图形化语言。这一点针对前面谈到的第二个问题。

UML只是一组图形符号。确切地讲， UML表示法中的每个符号都有明确语义。这样，一个开发者可以用UML绘制一个模型，而另一个开发者（甚至工具）可以无歧义地解释这个模型。这一点针对前面谈到的第一个问题。【UML的完整语义在《UML参考手册》一书中讨论。】

3. UML是一种可用于详细描述的语言

在此处，详细描述意味着所建的模型是精确的、无歧义的和完整的。特别是， UML适于对所有重要的分析、设计和实现决策进行详细描述，这些是软件密集型系统在开发和部署时所必需的。

4. UML是一种构造语言

UML不是一种可视化的编程语言，但用UML描述的模型可与各种编程语言直接相连。这意味着一种可能性，即可把用UML描述的模型映射成编程语言，如Java、C++和Visual Basic等，甚至映射成关系数据库的表或面向对象数据库的永久存储。对一个事物，如果表示为图形方式最为恰当，则用UML，而如果表示为文字方式最为恰当，则用编程语言。

这种映射允许进行正向工程：从UML模型到编程语言的代码生成。也可以进行逆向过程：由编程语言代码重新构造UML模型。逆向工程并不是魔术。除非你对实现中的信息编码，否则从模型到代码会丢失这些信息。逆向工程需要工具支持和人的干预。把正向代码生成和逆向工程这两种方式结合起来就可以产生双向工程，这意味着既能在图形视图下工作，又能在文字视

图下工作，这需要用工具来保持二者的一致性。【本书的第二部分和第三部分中讨论对系统的结构建模。】

除了直接映射以外，UML具有丰富的表达力，而且无歧义性，这允许模型的直接执行、系统的模拟以及对运行系统进行操纵。【本书的第四部分和第五部分讨论对系统的行为建模。】

5. UML是一种文档化语言

一个健壮的软件组织除了生产可执行的源代码之外，还要给出各种制品。这些制品包括（但不限于）：

- 需求
- 体系结构
- 设计
- 源代码
- 项目计划
- 测试
- 原型
- 发布

依靠于开发背景，有些制品做得或多或少地比另一些制品要正规些。这些制品不但是项目交付时所要求的，而且无论是在开发期间还是在交付使用后对控制、度量和理解系统也是关键的。

UML适于建立系统体系结构及其所有的细节文档。 UML还提供了用于表达需求和用于测试的语言。最终UML提供了对项目计划和发布管理的活动进行建模的语言。

6. 在何处能使用UML？

UML主要用于软件密集型系统。在下列领域中已经有效地应用了 UML：

- 企业信息系统
- 银行与财政服务
- 电信
- 运输
- 国防/航天
- 零售
- 医疗电子学
- 科学
- 基于Web的分布服务

UML不限于对软件建模。事实上，它的表达能力对非软件系统建模也是足够的。例如，立法系统的工作流程、病人保健系统的结构和行为以及硬件设计等。

2.2 UML的概念模型

为了理解UML，需要形成一个语言的概念模型，这要求学习建模的3个主要要素： UML的

基本构造块、支配这些构造块如何放在一起的规则和一些运用于整个 UML的公共机制。如果你掌握了这些思想，你就能够读懂 UML模型，并能建立一些基本模型。当你有了较丰富的应用UML的经验时，你就能够在这些概念模型之上使用更高深的语言特征进行构造。

1. UML的构造块

UML的词汇表包含3种构造块：

1）事物

2）关系

3）图

事物是对模型中最具有代表性的成分的抽象；关系把事物结合在一起；图聚集了相关的事物。

• UML中的事物

在UML中有4种事物：

1）结构事物

2）行为事物

3）分组事物

4）注释事物

这些事物是UML中基本的面向对象的构造块。用它们可以写出结构良好的模型。

• 结构事物

结构事物（*structural thing*）是UML模型中的名词。它们通常是模型的静态部分，描述概念或物理元素。共有7种结构事物。

第一，类（*class*）是对一组具有相同属性、相同操作、相同关系和相同语义的对象的描述。一个类实现了一个或多个接口。在图形上，把一个类画成一个矩形，通常矩形中写有类的名称、类的属性和类的操作，如图2-1所示。【在第4章和第9章中讨论类。】

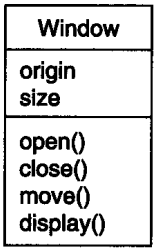

图2-1 类

第二，接口（*interface*）是描述了一个类或构件的一个服务的操作集。因此，接口描述元素的外部可见行为。一个接口可以描述一个类或构件的全部行为或部分行为。接口定义了一组操作的描述（即特征标记），而不是操作的实现。在图形上，把一个接口画成一个带有名称的圆。接口很少单独存在，而是通常依附于实现接口的类或构件，如图 2-2所示。【在第11章中讨论接口。】

第三，协作（*collaboration*）定义了一个交互，它是由一组共同工作以提供某协作行为的角

色和其他元素构成的一个群体，这些协作行为大于所有元素的各自行为的总和。因此，协作有结构、行为和维度。一个给定的类可以参与几个协作。这些协作因而表现了系统构成模式的实现。在图形上，把一个协作画成一个通常仅包含名称的虚线椭圆，如图 2-3所示。【第27章中讨论协作。】

图2-2 接口

图2-3 协作

第四，用况（*use case*）是对一组动作序列的描述，系统执行这些动作将产生一个对特定的参与者有价值而且可观察的结果。用况用于对模型中的行为事物结构化。用况是通过协作实现的。在图形上，把一个用况画成一个实线椭圆，通常仅包含它的名称，如图 2-4所示。【第16章中讨论用况。】

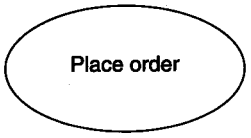

图2-4 用况

剩余的3种事物 —— 主动类、构件和节点，都和类相似，就是说它们也描述了一组具有相同属性、相同操作、相同关系和相同语义的对象。然而，这 3种事物与类的不同点也不少，而且对面向对象系统的某些方面的建模是必要的，因此对这几个术语需要单独处理。

第五，主动类（*active class*）是这样的类，其对象至少拥有一个进程或线程，因此它能够启动控制活动。主动类的对象所描述的元素的行为与其他元素的行为并发，除了这一点之外，它和类是一样的。在图形上，主动类很像类，只是它的外框是粗线，通常它包含名称、属性和操作，如图2-5所示。【第22章中讨论主动类。】

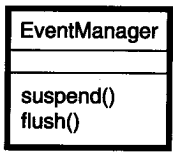

图2-5 主动类

剩下的两种元素是构件和节点，它们也是不同的。它们描述的是物理事物，而前 5种元素描

述的是概念或逻辑事物。

第六，构件（*component*）是系统中物理的、可替代的部件，它遵循且提供一组接口的实现。在一个系统中，你将遇到不同类型的部署构件，如COM+构件和Java Beans，以及在开发过程中所产生的制品构件，如源代码文件。通常构件是一个描述了一些逻辑元素（如类、接口和协作）的物理包。在图形上，把一个构件画成一个带有小方框的矩形，通常在矩形中只写该构件的名称，如图2-6所示。【在第24章中讨论构件。】

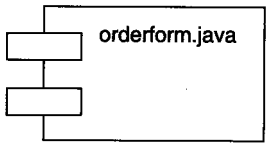

图2-6 构件

第七，节点（*node*）是在运行时存在的物理元素，它表示了一种可计算的资源，它通常至少有一些记忆能力和处理能力。一个构件集可以驻留在一个节点内，也可以从一个节点迁移到另一个节点。在图形上，把一个节点画成一个立方体，通常在立方体中只写它的名称，如图2-7所示。【在第26章中讨论节点。】

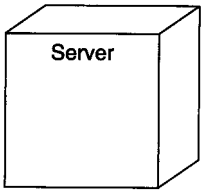

图2-7 节点

这7种元素，即类、接口、协作、用况、主动类、构件和节点，是UML模型中可以包含的基本结构事物。它们也有变体，如参与者、信号、实用程序（一种类）、进程和线程（两种主动类）、应用、文档、文件、库、页和表（一种构件）等。

• 行为事物

行为事物（*behavioral thing*）是UML模型的动态部分。它们是模型中的动词，描述了跨越时间和空间的行为。共有两类主要的行为事物。

第一，交互（*interaction*）是这样一种行为，它由在特定语境中共同完成一定任务的一组对象之间交换的消息组成。一个对象群体的行为或单个操作的行为可以用一个交互来描述。交互涉及一些其他元素，包括消息、动作序列（由一个消息所引起行为）和链（对象间的连接）。在图形上，把一个消息画成一条有向直线，通常在表示消息的线段上总有操作名，如图2-8所示。【在第16章中讨论用于模型中构造行为事物的用况；在第15章中讨论交互。】

图2-8 消息

第二，状态机（*state machine*）是这样一种行为，它描述了一个对象或一个交互在生命期内响应事件所经历的状态序列。单个类或一组类之间协作的行为可以用状态机来描述。一个状态机涉及到一些其他元素，包括状态、转换（从一个状态到另一个状态的流）、事件（触发转换的事物）和活动（对一个转换的响应）。在图形上，把一个状态画成一个圆角矩形，通常在圆角矩形中含有状态的名称及其子状态，如图 2-9所示。【在第21章中讨论状态机。】

图2-9 状态

交互和状态机这两种元素是可以包含在 UML模型中的基本行为事物。在语义上，这些元素通常与各种结构元素（主要是类、协作和对象）相关。

• 分组事物

分组事物（*grouping thing*）是UML模型的组织部分。它们是一些由模型分解成的“盒子”。在所有的分组事物中，最主要的分组事物是包。

包（*package*）是把元素组织成组的机制，这种机制具有多种用途。结构事物、行为事物甚至其他的分组事物都可以放进包内。包不像构件（仅在运行时存在），它纯粹是概念上的（即它仅在开发时存在）。在图形上，把一个包画成一个左上角带有一个小矩形的大矩形，在矩形中通常仅含有包的名称，有时还有内容，如图 2-10所示。【在第12章中讨论包。】

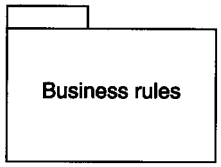

图2-10 包

包是用来组织UML模型的基本分组事物。它也有变体，如框架、模型和子系统等（它们是包的不同种类）。

• 注释事物

注释事物（*annotational thing*）是UML模型的解释部分。这些注释事物用来描述、说明和标注模型的任何元素。有一种主要的注释事物，称为注解。注解（*note*）是一个依附于一个元素或一组元素之上，对它进行约束或解释的简单符号。在图形上，把一个注解画成一个右上角是折角的矩形，其中带有文字或图形解释，如图 2-11所示。【在第6章中讨论注解。】

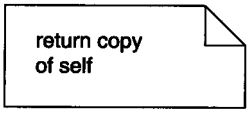

图2-11 注解

该元素是可以包含在UML模型中的基本注释事物。通常可以用注解修饰带有约束或解释的图，当然最好是把注释表示成形式或非形式化的文本。像需求（从模型的外部来描述一些想得到的行为）一样，这种元素也有变体。

• UML中的关系

在UML中有4种关系：

1）依赖

2）关联

3）泛化

4）实现

这些关系是UML的基本关系构造块，用它们可以写出结构良好的模型。

第一，依赖（*dependency*）是两个事物间的语义关系，其中一个事物（独立事物）发生变化会影响另一个事物（依赖事物）的语义。在图形上，把一个依赖画成一条可能有方向的虚线，偶尔在其上还有一个标记，如图2-12所示。【在第5章和第10章中讨论依赖。】

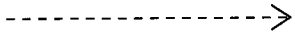

图2-12 依赖

第二，关联（*association*）是一种结构关系，它描述了一组链，链是对象之间的连接。聚合是一种特殊类型的关联，它描述了整体和部分间的结构关系。在图形上，把一个关联画成一条实线，它可能有方向，偶尔在其上还有一个标记，而且它经常还含有诸如多重性和角色名这样的修饰，如图2-13所示。【在第5章和第10章中讨论关联。】

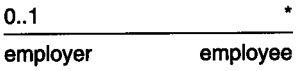

图2-13 关联

第三，泛化（*generalization*）是一种特殊/一般关系，特殊元素（子元素）的对象可替代一般元素（父元素）的对象。用这种方法，子元素共享了父元素的结构和行为。在图形上，把一个泛化关系画成一条带有空心箭头的实线，它指向父元素，如图2-14所示。【在第5章和第10章中讨论泛化。】

图2-14 泛化

第四，实现（*realization*）是类元之间的语义关系，其中的一个类元指定了由另一个类元保证执行的契约。在两种地方要遇到实现关系：一种是在接口和实现它们的类或构件之间；另一种是在用况和实现它们的协作之间。在图形上，把一个实现关系画成一条带有空心箭头的虚线，它是泛化和依赖关系两种图形的结合，如图2-15所示。【在第10章中讨论实现关系。】

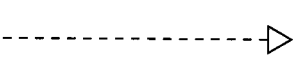

图2-15 实现

这4种元素是UML模型中可以包含的基本关系事物。它们也有变体，例如，依赖的变体有精化、跟踪、包含和延伸。

• UML中的图

图（*diagram*）是一组元素的图形表示，大多数情况下把图画成顶点（代表事物）和弧（代表关系）的连通图。为了对系统进行可视化，可以从不同的角度画图，这样图是对系统的投影。除了非常微小的系统外，图是系统组成元素的省略视图。有的元素可以出现在所有图中，有的元素可以出现在一些图中（很常见），还有的元素不能出现在图中（很罕见）。在理论上，图可以包含任何事物及其关系的组合。然而，实际上仅存在着少量的常见组合，它们要与 5种最有用的组成了软件密集型系统的体系结构的视图相一致。由于这个原因， UML包括9种这样的图：

1）类图
2）对象图
3）用况图
4）顺序图
5）协作图
6）状态图
7）活动图
8）构件图
9）实施图

类图（*class diagram*）展现了一组对象、接口、协作和它们之间的关系。在对面向对象系统的建模中所建立的最常见的图就是类图。类图给出系统的静态设计视图。包含主动类的类图给出系统的静态进程视图。【在第8章中讨论类图。】

对象图（*object diagram*）展现了一组对象以及它们之间的关系。对象图描述了在类图中所建立的事物的实例的静态快照。和类图一样，这些图给出系统的静态设计视图或静态进程视图，但它们是从真实的或原型案例的角度建立的。【在第14章中讨论对象图。】

用况图（*use case diagram*）展现了一组用况、参与者（一种特殊的类）及其它们之间的关系。用况图给出系统的静态用况视图。这些图对于系统的行为进行组织和建模是非常重要的。【在第17章中讨论用况图。】

顺序图和协作图都是交互图。交互图（*interaction diagram*）展现了一种交互，它由一组对象和它们之间的关系组成，包括在它们之间可能发送的消息。交互图专注于系统的动态视图。顺序图（*sequence diagram*）是一种强调消息的时间顺序的交互图；协作图（*collaboration diagram*）也是一种交互图，它强调收发消息的对象的结构组织。顺序图和协作图是同构的，这意味着它们是可以相互转换的。【在第18章中讨论交互图。】

状态图（*statechart diagram*）展现了一个状态机，它由状态、转换、事件和活动组成。状态图专注于系统的动态视图。它对于接口、类或协作的行为建模尤为重要，而且它强调对象行为的事件顺序，这非常有助于对反应式系统建模。【在第24章讨论状态图。】

活动图（*activity diagram*）是一种特殊的状态图，它展现了在系统内从一个活动到另一个活动的流程。活动图专注于系统的动态视图。它对于系统的功能建模特别重要，并强调对象间的

控制流程。【在第19章中讨论活动图。】

构件图（*component diagram*）展现了一组构件之间的组织和依赖。构件图专注于系统的静态实现视图。它与类图相关，通常把构件映射成一个或多个类、接口或协作。【在第29章中讨论构件图。】

实施图（*deployment diagram*）展现了对运行时处理节点以及其中的构件的配置。实施图给出了体系结构的静态实施视图。它与构件图相关，通常一个节点包含一个或多个构件。【在第30章中讨论实施图。】

并不限定仅使用这9种图，开发工具可以采用UML来提供其他种类的图，但到目前为止，这9种图在实际应用中是最常用的。

2. UML规则

不能简单地把UML的构造块按随机的方式放在一起。像任何语言一样， UML有一套规则，这些规则描述了一个结构良好的模型看起来应该像什么。一个结构良好的模型应该在语义上是前后一致的，并且与所有的相关模型协调一致。

UML有用于描述如下事物的语义规则：

- 命名　　为事物、关系和图起名
- 范围　　给一个名称以特定含义的语境
- 可见性　　怎样让其他人使用或看见名称
- 完整性　　事物如何正确、一致地相互联系
- 执行　　运行或模拟动态模型的含义是什么

在软件密集型系统的开发期间所建造的模型往往需要发展变化，并可以由许多人员以不同的方式、在不同的时间进行观察。由于这个原因，下述的情况是常见的，即开发组不仅要建造一些结构良好的模型，也要建造一些这样的模型：

- 省略　　隐藏某些元素以简化视图
- 不完全性　　可以遗漏某些的元素
- 不一致性　　不保证模型的完整性

在软件开发的生命期内，随着系统细节的展开和变动，不可避免地要出现这些不太规范的模型。UML的规则鼓励（不是强迫）你专注于最重要的分析、设计和实现问题，这些问题将促使模型随着时间的推移而具有良好的结构。

3. UML中的公共机制

通过与具有公共特征的模式取得一致性，可以使一座建筑更为简单和更为协调。一座房子可以按一定的结构模式（它定义了建筑风格）建造成维多利亚式的或法国乡村式的。对于 UML也是如此。由于在UML中有4种贯穿整个语言且一致应用的公共机制，因此使得 UML变得较为简单。这4种机制是：

1）详述
2）修饰
3）通用划分
4）扩展机制

• 详述

UML不只是一种图形语言。实际上，在它的图形表示法的每部分背后都有一个详述，这个详述提供了对构造块的语法和语义的文字叙述。例如，在一个类的图符背后就有一个详述，它提供了对该类所拥有的属性、操作（包括完整的特征标记）和行为的全面描述；在视觉上，类的图符可能仅展示了这个详述的一小部分。此外，可能存在着该类的另一个视图，其中提供了一个完全不同的部件集合，但是它仍然与该类的基本详述相一致。 UML的图形表示法用来对系统进行可视化；UML的详述用来描述系统的细节。假定把二者分开，就可能进行增量式的建模。这可以通过以下方式完成：先画图，然后再对这个模型的详述增加语义，或直接创建详述，也可能对一个已经存在的系统进行逆向工程，然后再创建作为这些详述的投影图。

UML的详述提供了一个语义底版，它包含了一个系统的各模型的所有部分，并且各部分相互联系，并保持一致。因此， UML的图只不过是对底版的简单视觉投影，每一个图展现了系统的一个特定的方面。

• 修饰

UML中的大多数元素都有唯一的和直接的图形表示符号，这些图形符号对元素的最重要的方面提供了可视化表示。例如，特意把类的符号设计得容易画出，这是因为在对面向对象系统建模中，类是最常用的元素。类的图形符号也展示了类的最重要方面，即它的名称、属性和操作。【在第6章中讨论注解和其他修饰】

对类的详述可以包含其他细节，例如，它是否是抽象类，或它的属性和操作是否可见。可以把很多这样的细节表示为图形或文字修饰，放到类的基本矩形符号上。例如，图 2-16表示的是一个带有修饰的类，图中表明这个类是一个抽象类，有两个公共操作、一个保护操作和一个私有操作。

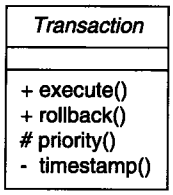

图2-16 修饰

UML表示法中的每一个元素都有一个基本符号，可以把各种修饰细节加到这个符号上。

• 通用划分

在对面向对象系统建模中，至少有两种划分方法。

第一种方法是对类和对象的划分。类是一个抽象；对象是这种抽象的一个具体形式。在UML中，可以对类和对象建立模型，如图 2-17所示。【在第13中讨论对象。】

在这个图中，有一个名称为 Customer的类，它有3个对象，分别为 Jan（它被明确地标记为 Customer的对象），:Customer（匿名的Customer对象）和Elyse（它在详述中被说明为是一种Customer对象，尽管在这里没有明确表示）。

UML的每一个构造块几乎都存在像类 /对象这样的二分法。例如，可以有用况和用况实例，

构件和构件实例，节点和节点实例等。在图形上， UML用与类同样的图形符号来表示对象，只是简单地在对象名的下边画一道线，以示与类的区别。

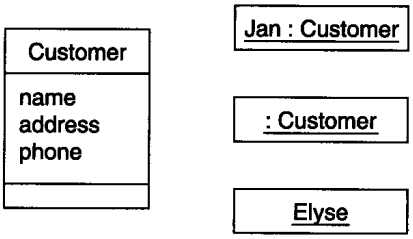

图2-17 类和对象

第二种方法是接口和实现的分离。接口声明了一个契约，而实现则表示了对该契约的具体实施，它负责如实地实现接口的完整语义。在 UML中，可以既对接口又对它们的实现建模，如图2-18所示。【在第11章中讨论接口。】

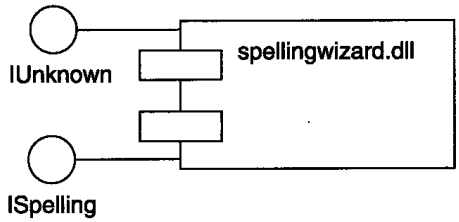

图2-18 接口和实现

在这个图中，有一个名称为 `spellingwizard.dll`的构件，它实现了接口 `IUnknown`和接口`ISpelling`。

几乎每一个UML的构造块都有像接口/实现这样的二分法。例如，用况和实现它们的协作，操作和实现它们的方法。

- 扩展机制

UML提供了一种绘制软件蓝图的标准语言，但是一种闭合的语言即使表达能力再丰富，也难以表示出各种领域中的各种模型在不同时刻所有可能的细微差别。由于这个原因， UML是可扩展的，可以以受控的方式扩展该语言。【在第6章中讨论UML的扩展机制。】UML的扩展机制包括：

- 构造型
- 标记值
- 约束

构造型（*stereotype*）扩展了UML的词汇，它允许你创造新的构造块，这个新构造块既可从现有的构造块派生，又专门针对你要解决的问题。例如，你正在使用一种编程语言，如 Java或C++，你经常要对“异常事件”建模。在这些语言里，“异常事件”就是类，只是用很特殊的方法进行了处理。通常你可能只想允许放弃和捕捉异常事件，没有其他要求。此时你可以让异常事件在你的模型中成为一等公民——即可以像对待基本构造块一样对待它们，只要用一个适当的构造型来标记它们即可。请看图 2-19中的类`Overflow`。

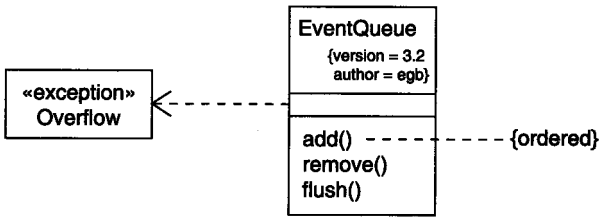

图2-19 扩展机制

标记值（*tagged value*）扩展了UML构造块的特性，允许创建详述元素的新信息。例如，如果你在制作一种经过包装的产品，随着时间的推移，它经过了多次发行，那么你经常会想要跟踪产品的版本和对产品做评论性摘要的作者。版本和作者不是UML的基本概念。通过引入新的标记值，可以把它们加到像类那样的任何构造块中去。例如，在图2-19中，在类`EventQueue`上标记了明确的版本和作者，就对该类进行了扩展。

约束（*constraint*）扩展了UML构造块的语义，它允许增加新的规则或修改现有的规则。例如，你可能想约束类`EventQueue`，以使所有的增加都按序排列。如图2-19所示，对操作`add`增加了一个约束，即`{ordered}`。

总的来说，这3种扩展机制都允许你根据项目的需要塑造和培育UML。这些机制也使得UML适合于新的软件技术（例如，很可能出现的功能更强的分布式编程语言）。可以增加新的构造块，修改已存在的构造块的详述，甚至可以改变它们的语义。当然，以受控的方式进行扩展是重要的，这可使你不偏离UML的真正目的——信息交流。

2.3 体系结构

可视化、详述、构造和文档化一个软件密集型系统，要求从几个角度去观察系统。各种人员——最终用户、分析人员、开发人员、系统集成人员、测试人员、技术资料作者和项目管理者——他们各自带着项目的不同日程，而且在项目的生命周期内各自在不同的时间、以不同的方式来看系统。系统体系结构或许是最重要的制品，它可以驾驭不同的观点，并在整个项目的生命周期内控制对系统的迭代和增量式开发。【在第1章中讨论需要从不同的角度观察复杂系统。】

体系结构是一组有关下述内容的重要决策：

- 软件系统的组织
- 对组成系统的结构元素及其接口的选择
- 如元素间的协作中所描述的那样的行为
- 将这些结构和行为元素组合到逐步增大的子系统
- 指导这种组织的体系结构风格：静态和动态元素及其他们的接口、协作和组成

软件体系结构不仅关心结构和行为，而且还关心用法、功能、性能、弹性、复用、可理解性、经济技术约束及其折衷，以及审美的考虑。

如图2-20所示，最好用5个互连的视图来描述软件密集型系统的体系结构。每一个视图是在

一个特定的方面对系统的组织和结构进行的投影。【第31章中讨论对系统的体系结构建模。】

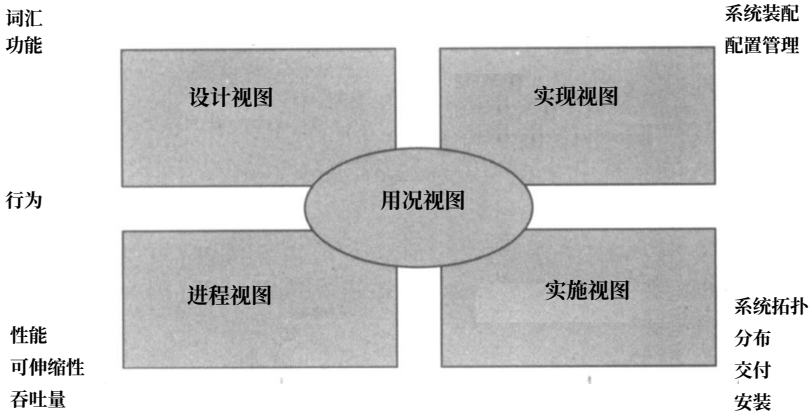

图2-20 对系统的体系结构建模

系统的用况视图（*use case view*）由专门描述可被最终用户、分析人员和测试人员看到的系统行为的用况组成。用况视图实际上没有描述软件系统的组织，而是描述了形成系统体系结构的动力。在UML中，该视图的静态方面由用况图表现；动态方面由交互图、状态图和活动图表现。

系统的设计视图（*design view*）包含了类、接口和协作，它们形成了问题及其对问题解决方案的术语词汇。这种视图主要支持系统的功能需求，即系统提供给最终用户的服务。在 UML中，该视图的静态方面由类图和对象图表现；动态方面由交互图、状态图和活动图表现。

系统的进程视图（*process view*）包含了形成系统并发与同步机制的线程和进程。该视图主要针对性能、可伸缩性和系统的吞吐量。在 UML中，对进程视图的静态方面和动态方面的表现与设计视图相同，但注重于描述线程和进程的主动类。

系统的实现视图（*implementation view*）包含了用于装配与发布物理系统的构件和文件。这种视图主要针对系统发布的配置管理，它由一些独立的构件和文件组成；这些构件和文件可以用各种方法装配，以产生运行系统。在 UML中，该视图的静态方面由构件图表现；动态方面由交互图、状态图和活动图表现。

系统的实施视图（*deployment view*）包含了形成系统硬件拓扑结构的节点（系统在其上运行）。这种视图主要描述对组成物理系统的部件的分布、交付和安装。在 UML中，该视图的静态方面由实施图表现；动态方面由交互图、状态图和活动图表现。

这5种视图中的每一种视图都可单独使用，使不同的人员能专注于他们最为关心的体系结构问题。这5种视图也可相互作用，如实施视图中的节点拥有实现视图的构件，而这些构件又表示了设计视图和进程视图中的类、接口、协作以及主动类的物理实现。 UML允许表达这5种视图中的任何一种视图，也允许表达它们之间的交互。

2.4 软件开发生命周期

UML在很大程度上是独立于过程的，这意味着它不依赖于任何特殊的软件开发生命周期。

然而，为了从UML中得到最大的收益，你应该考虑这样的过程：

- 用况驱动的
- 以体系结构为中心的
- 迭代的和增量的

【在附录C中概述了Rational 的统一过程；对本过程的更完整处理在《统一软件开发过程》一书中讨论。】

用况驱动（*use case driven*）意味着，把用况作为一种基本的制品，用于建立所要求的系统行为、验证和确认系统的体系结构、测试以及在项目组成员间进行交流。

以体系结构为中心（*architecture-centric*）意味着，要以系统的体系结构作为一种基本制品，用于在开发中对系统进行概念化、构造、管理和演化。

迭代过程（*iterative process*）是这样一种过程：它涉及到一连串可执行发布的管理。增量过程（*incremental process*）是这样一个过程：它涉及到系统体系结构的持续集成，以产生各种发布，当然每个新的发布都比上一个发布有所改善。总的来讲，迭代和增量的过程是由风险来驱动的（*risk-driven*），这意味着每个新的发布都致力于处理和降低对于项目成功影响最为显著的风险。

这种用况驱动的、以体系结构为中心的、迭代的 /增量的过程可以被分成几个阶段。一个阶段（*phase*）是过程的两个主要里程碑间的时间跨度，当达到一组明确的目标时，就完成了一定的制品，并作出了是否进行下一步的决策。如图 2-21所示，在软件的开发生命周期内有 4个阶段：初始、细化、构造和移交。在图 2-21中，按这些阶段对工作流进行了划分，并显示了它们的焦点随时间的推移而变化的程度。

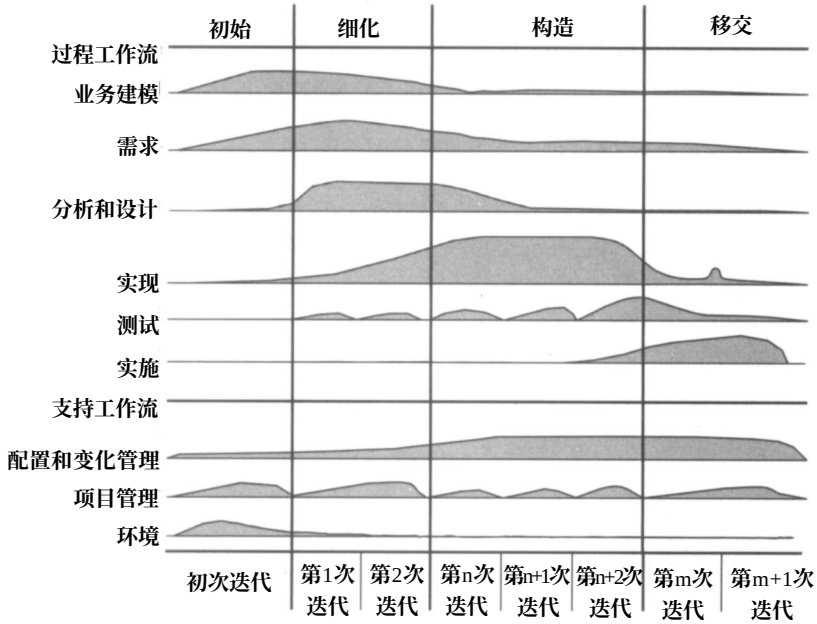

图2-21 软件开发生命周期

初始（*inception*）是这个过程的第一个阶段，此时萌发的开发想法经过培育，要达到这样一

个目标：至少要在内部奠定足够的基础，以保证能够进入到细化阶段。

细化（*elaboration*）是这个过程的第二个阶段，此时定义想象的产品和它的体系结构。在这个阶段，将明确系统需求，按重要性对其排序并划定基线。可以按一般的描述，也可以按精确的评估准则来排列系统的需求，每个需求都说明了特定的功能或非功能的行为，并为测试提供了基础。

构造（*construction*）是这个过程的第三个阶段，此时软件从可执行的体系结构基线发展到准备移交给用户。针对项目的商业需要，也要对系统的需求，特别是对系统的评价准则，不断地进行检查，并要适当地分配资源，以主动地减低项目的风险。

移交（*transition*）是这个过程的第四个阶段，此时把软件交付给用户。在这个阶段，软件开发过程很少能结束，还要继续改善系统，根除错误，增加早期发布未能实现的特性。

使这个过程与众不同，并对这 4个阶段有影响的一个元素是迭代。迭代（*iteration*）是一组明确的活动，它有导致发布的基准计划和评价准则（无论是内部的还是外部的）。这意味着软件开发生命周期有一个特征，即可持续地发布系统体系结构的可执行版本。正是强调要把体系结构作为一个重要的制品，才使得 UML注重于对系统体系结构的不同视图进行建模。