第8章 类　　图

本章内容：

- 对简单协作建模
- 对逻辑数据库模式建模
- 正向与逆向工程

类图是面向对象系统的建模中最常见的图。类图显示了一组类、接口、协作以及它们之间的关系。

类图用于对系统静态设计视图建模。其大部分涉及到对系统的词汇建模、对协作建模或对模式建模。类图也是两个相关图（构件图和实施图）的基础。

类图不仅对结构模型的可视化、详述和文档化很重要，而且对通过正向与逆向工程构造可执行的系统也很重要。

8.1　入门

当建造房屋时，要从包括像墙、楼板、门、窗、天花板和托梁的基本构造块的词汇开始。这些东西主要是结构性的（墙有高度、宽度和厚度），但也具有一些行为性（不同种类的墙支撑不同的负重，门能开关，对无支撑的楼板跨度有一些约束）。事实上，不能单独地考虑结构和行为特征，而必须在建造房屋时考虑它们如何相互作用。建造房屋的过程涉及到以所要求的独特和合意的方式装配这些事物，以满足所有的功能性和非功能性的需求。你创建的用来可视化你的房屋并向承包商详述细节的蓝图实际上是对这些事物以及它们之间的关系的图形描述。

构造软件也有许多与此相同的特性，所不同的只是对于给定的软件的流动性，你有能力从草图定义自己的基本构造块。你可以用 UML的类图对这些构造块的静态方面和它们之间的关系进行可视化，并描述其构造细节，如图 8-1所示。

8.2　术语和概念

类图（*class diagram*）是显示一组类、接口、协作以及它们之间关系的图。在图形上，类图是顶点和弧的集合。

1. 普通特性

类图是一种特殊的图，它和所有的其他图有相同的普通特性，即有名称和投影到一个模型的图形内容。类图与所有其他图的区别是它的特殊内容。【在第7章中讨论图的普通特性。】

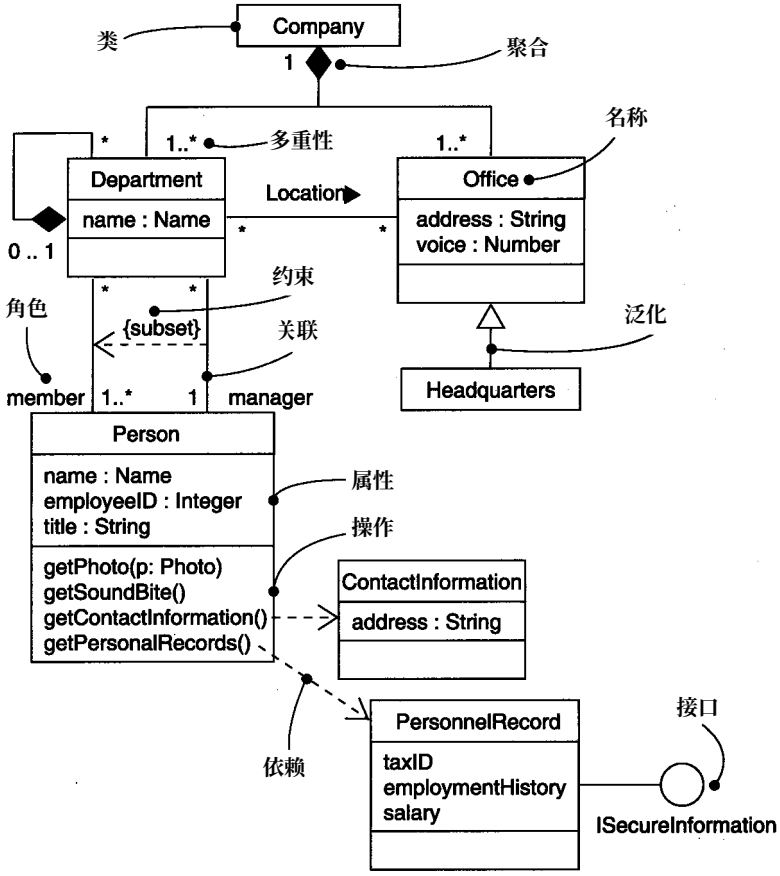

图8-1 类图

2. 内容

类图通常包含下述内容：

- **类**
- **接口**
- **协作**
- **依赖、泛化和关联关系**

像所有的其他图一样，类图可以包含注解和约束。

类图还可以含有包或子系统，二者都用于把模型元素聚集成更大的组块。有时也要在类图中放置实例，特别是在对实例的类型（可能是动态的）进行可视化时。【在第4章和第9章中讨论类；在第11章中讨论接口；在第27章中讨论协作；在第5章和第10章中讨论关系；在第12章中讨论包；在第31章中讨论子系统；在第13章中讨论实例。】

注释 构件图和实施图与类图类似，只是分别包含构件和节点，而不是类。

3. 普通用法

类图用于对系统的静态设计视图建模。这种视图主要支持系统的功能需求，即系统要提供给最终用户的服务。【在第2章中讨论设计视图。】

当对系统的静态设计视图建模时，通常以下述3种方式之一使用类图。

1）对系统的词汇建模

对系统的词汇建模涉及到作出这样的决定：哪些抽象是考虑中的系统的一部分，哪些抽象是处于系统边界之外。用类图详述这些抽象和它们的职责。【在第4章中讨论对系统的词汇建模。】

2）对简单协作建模

协作是一些共同工作的类、接口和其他元素的群体，该群体提供的一些合作行为强于所有这些元素的行为之和。例如，当对分布式系统中的事务语义建模时，不能仅仅盯着一个单独的类来推断要发生什么，而要由相互协作的一组类来实现这些语义。用类图对这组类以及它们之间的关系进行可视化和详述。【在第27章中讨论协作。】

3）对逻辑数据库模式建模

将模式看作为数据库的概念设计的蓝图。在很多领域中，要在关系数据库或面向对象数据库中存储永久信息。可以用类图对这些数据库的模式建模。【在第23章中讨论永久性。在第29章中讨论对物理数据库建模。】

8.3 普通建模技术

8.3.1 对简单协作建模

类不是单独存在的，而是要和其他的类协同工作，以实现一些强于使用单个类的语义。因此，除了捕获系统的词汇之外，也需要把注意力转移到对词汇中的这些事物协同工作的各种方式进行可视化、详述、构造和文档化。用类图描述这些协作。

创建类图就是对组成系统设计视图的一部分事物及关系进行建模。由于这个原因，每个类图在一个时间内应该注重于一个协作。

为了对协作建模，要遵循如下策略：

- 识别要建模的机制。一个机制描述了正在建模的部分系统的一些功能和行为，这些功能和行为起因于类、接口以及一些其他事物所组成的群体的相互作用。
- 对每种机制，识别参与协作的类、接口和其他协作，并识别这些事物之间的关系。
- 用脚本排演这些事物。通过这种方法，可发现模型的哪些部分被遗漏以及哪些部分有明显的语义错误。
- 要把元素和它们的内容聚集在一起。对于类，开始要做好职责的平衡，然后随着时间的推移，把它们转换成具体的属性和操作。【像此处的机制经常要与用况相结合，有关这方面的问题在第16章中讨论；脚本是贯穿用况的线索，有关这方面的问题在第15章中讨论。】

例如，图8-2描述了取自一个自治机器人实现中的一组类。该图关注包含在沿着一条路径移动机器人的机制中的类。你会发现在图中有一个抽象类（Motor），它有两个具体子类，分别是

SteeringMotor和MainMotor。这两个类都继承父类Motor的5个操作。这两个类又被显示为另一个类Driver的部分。类PathAgent与Driver有一对一的关联，与CollisionSensor有一对多的关联。虽然PathAgent被给定了系统职责，但此处没显示出任何属性和操作。

在这个系统中还包含更多的类，但这个图只关注被直接包含在移动机器人中的那些抽象。在其他的图中会看到一些同样的类。例如，尽管在此图中没有显示，但类 PathAgent至少和另外的两个类（Environment和GoalAgent）在更高层次机制上相互协作，用于管理机器人在特定时刻可能有冲突的目标。类似地，尽管在此图也没有显示，但类 CollisionSensor和Driver（和它的部分）与另一个类（FaultAgent）在一种机制中协作，该机制负责持续地检查机器人的硬件故障。通过在不同的图中关注每一个这样的协作，就从几个角度提供了可理解的系统视图。

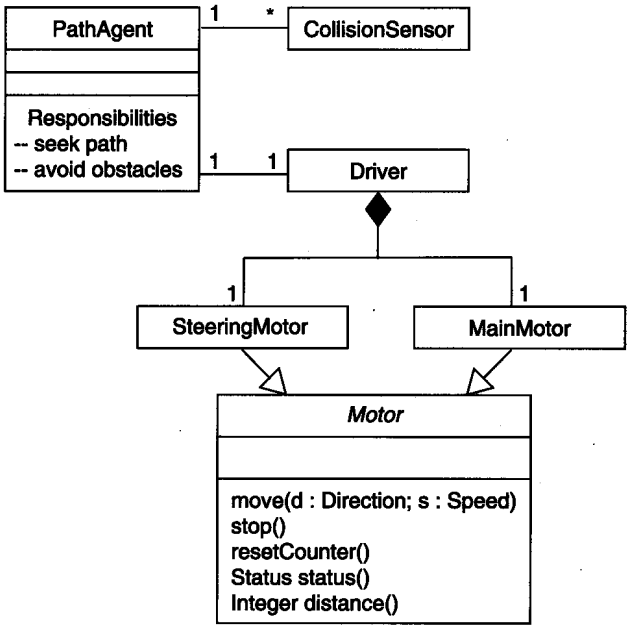

图8-2 对简单协作建图

8.3.2 对逻辑数据库模式建模

你所建模的很多系统都有永久对象，即要把这些对象存储在数据库中，以便后来检索。最经常的是用关系数据库、面向对象数据库或混合的关系 /对象数据库存储永久对象。UML很适合于对逻辑数据库模式和物理数据库本身建模。【在第23章中讨论对永久对象的分布和迁移建模；在第 29章中讨论对物理数据库的建模。】

实体-关系（E-R）图是一种用于逻辑数据库设计的通用建模工具， UML的类图是实体-关系

图的超集。传统的E-R图只针对数据，类图则进了一步，它也允许对行为建模。在物理数据库中，一般要把这些逻辑操作转换成触发器或存储的过程。

为了对模式建模，要遵循如下策略：

- 在模型中识别其状态必须超过其应用系统生命周期的类。
- 创建含有这些类的类图，并把它们标记成永久的（用 UML提供的标准标记值）。对于特定数据库细节，可以定义自己的标记值集合。
- 展开这些类的结构性细节。通常，这意味着详述属性的细节，并注重于关联和构造这些类的基数。
- 观察使物理数据库设计复杂化的公共模式，如循环关联、一对一关联和 n-元关联。必要时创建简化逻辑结构的中间抽象。
- 也要考虑这些类的行为，展开对数据存储和数据完整性来说是重要的操作。通常，为了提供更好的关注分离，与这些对象集的操纵相关的业务规则应该被封装在这些永久类的上一层。
- 如有可能，用工具来帮助把逻辑设计转换成物理设计。

注释 逻辑数据库设计超出了本书的范围。这里只是简单地指明如何用 UML对模式建模。在实际应用中，最终要对所用数据库的种类（关系型的或面向对象的）使用构造型。【在第6章中讨论构造型。】

图8-3显示了取自某学校的信息系统的一组类。该图对本书在前面给出的一个类图作了扩展，

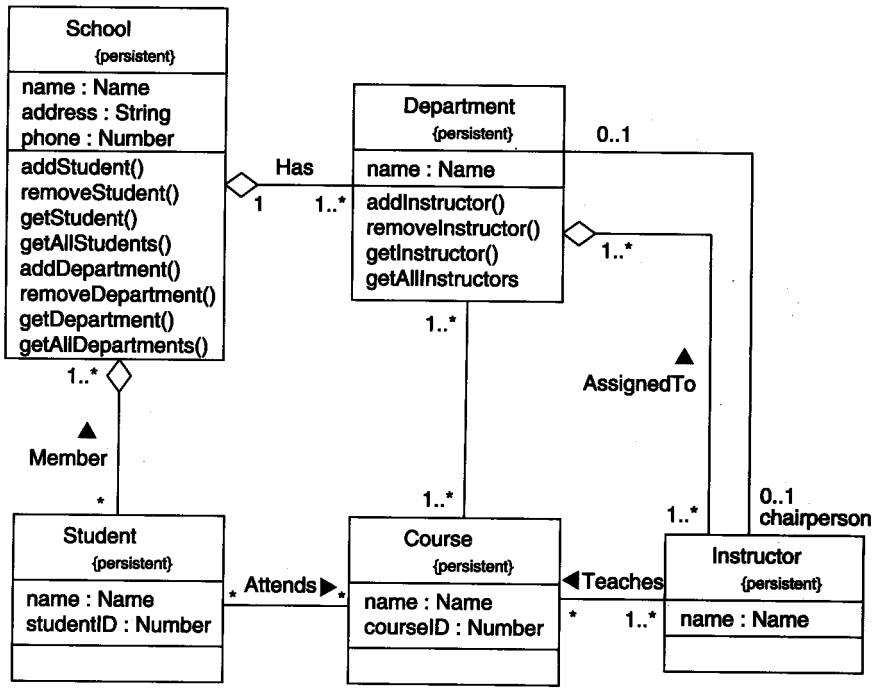

图8-3 对模式建模

你会看到，本图所显示的这些类的细节足以构造一个物理数据库。从图的左下部开始，你会发现有3个名为Student、Course和Instructor的类。Student和Course之间有一个说明学生所听课程的关联。此外，每个学生可以听的课程门数不限，听每门课程的学生人数也不限。

这6个类都被标记为永久的，表明它们的实例要存放在数据库或其他形式的永久存储系统中。该图也显示了这6个类的属性。注意，所有的属性都是简单类型的。当对模式建模时，一般要用显式的聚合而不是用属性对任何非简单类型的关系建模。【在第4章中讨论对简单类型建模；在第5章和第10章中讨论聚合。】

类School和类Department显示了几个操纵其部分对象的操作。模型中包含这些操作是因为它们对维护数据的完整性是很重要的（例如增加或移动 Department将有一些连带的影响）。对于这些类和其他的类，还有很多可以考虑的其他操作，例如，在指派学生前，查询课程的先决条件。这些操作更接近业务规则而不是用于数据库完整性，因而最好把它们放在比这个模式更高的抽象层次上。

8.3.3 正向工程和逆向工程

建模是重要的，但要记住开发组的主要产品是软件而不是图。当然，创建模型的原因是为了及时交付满足用户及业务发展目标的正确软件。因此，使创建的模型与交付的产品相匹配，并使二者保持同步的代价减少到最小（甚至取消）是很重要的。【在第1章中讨论建模的重要性。】

使用UML的某些功能，所创建的模型将永远不会映射成代码。例如，若用活动图对业务过程建模，则很多被建模的活动要涉及到人员而不是计算机。另一种情况是你所建模的系统的组成部分在你的抽象层次上只是一些硬件（虽然在另一个抽象层次上看，也可以说该硬件包含了嵌入的计算机和软件）。【在第19章中讨论活动图。】

在大多数情况下，要把所创建的模型映射成代码。 UML没有指定对任何面向对象编程语言的映射，但UML还是考虑了映射问题。特别是对类图，可以把类图的内容清楚地映射到各种工业化的面向对象的语言，如Java、C++、Smalltalk、Eiffel、Ada、ObjectPascal和Forte。UML也被设计得可映射到各种商用的基于对象的语言，如 Visual Basic。

> **注释** 对于正向和逆向工程， UML到特定的实现语言的映射已经超出了本书的范围。在实际应用中，最终要针对所用的编程语言使用构造型和标记值。【在第6章中讨论构造型和标记值。】

正向工程（*forward engineering*）是通过到实现语言的映射而把模型转换为代码的过程。由于用UML描述的模型在语义上比当前的任何面向对象编程语言都要丰富，所以正向工程将导致一定信息的损失。事实上，这是为什么除了代码之外还需要模型的主要原因。像协作这样的结构特征和交互这样的行为特征，在 UML中能被清晰地可视化，但源代码就不会如此清晰。

对类图进行正向工程，要遵循如下的策略：

- 识别映射到所选择的实现语言的规则。这是你要为整个项目或组织做的事。
- 根据所选择语言的语义，可能要限定对一些 UML特性的用法。例如， UML允许对多继承

建模，但Smalltalk仅允许单继承。可以选择禁止开发人员用多继承建模（这使得模型依赖于语言），也可以选择把这些丰富的特征转化为实现语言的惯用方法（这样使得映射更为复杂）。

- 用标记值详述目标语言。如果需要精确的控制，可以在单个类的层次上这样做。也可以在较高的层次上（如协作或包）这样做。
- 使用工具对模型进行正向工程。

图8-4是一个简单的类图，它描述了职责模式链的一个实例化。这个特殊的实例化包含3个类：Client、EventHandler和GUIEventHandler。Client和EventHandler被显示为抽象类，而GUIEventHandler是具体类。虽然在这个实例化中增加了两个私有属性，但EventHandler有这个模式（handleRequest）所期望的通常操作。【在第28章中讨论模式】

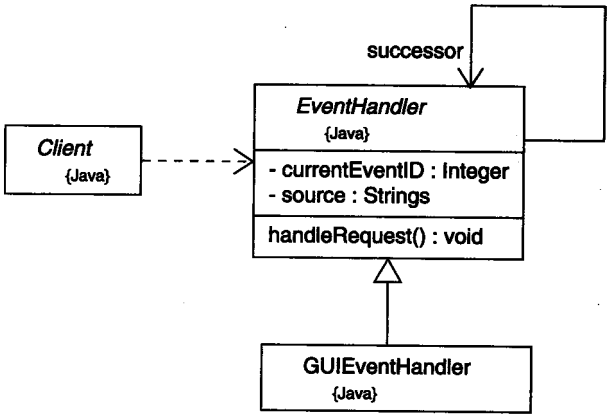

图8-4 正向工程

所有这些类中都指定了一个到Java的映射，被注解在它们的标记值中。用工具对这个图中的类进行正向工程，使之成为Java程序是简单的。对类EventHandler进行正向工程将产生如下代码：

```
public abstract class EventHandler {

  EventHandler successor;
  private Integer currentEventID;
  private String source;

  EventHandler() {}
  public void handleRequest() {}

}
```

逆向工程（reverse engineering）是通过从特定实现语言的映射而把代码转换为模型的过程。逆向工程会导致大量的多余信息，其中的一些信息属于细节层次，对于建造有用的模型来说过于详细。同时，逆向工程是不完整的。由于在正向工程中产生的代码丢失了一些信息，

所以，除非所使用的工具能对原先注释中的信息（这超出了实现语言的语义）进行编码，否则就不能再从代码创建一个完整的模型。

对类图进行逆向工程，要遵循如下的策略：

- 识别从实现语言或所选的语言进行映射的规则。这是你要为整个项目或组织做的事。
- 使用工具，指向要进行逆向工程的代码。用工具生成新的模型或修改以前进行正向工程时已有的模型。
- 使用工具，通过查询模型创建类图。例如，可以从一个或几个类开始，然后通过追踪特定的关系或其他相邻的类扩展类图。根据表达你的意图的需要，显示或隐藏类图内容的细节。

【图3-3是通过对部分Java类库进行逆向工程产生的。】

8.4 提示和技巧

在UML中创建类图时，要记住各个类图仅仅是系统静态设计视图的图形表示。不必用单个类图去表达系统设计视图的所有内容。而是用系统的所有类图共同表达系统的全部静态设计视图；单个类图仅表达系统的一个方面。

一个构造良好的类图，应满足如下的要求：

- 注重于表达系统静态设计视图的一个方面。
- 仅包含对理解该方面是必要的元素。
- 提供的细节与抽象的层次要一致，仅带有对理解系统是必要的修饰。
- 没有过分地压缩内容以致使读者对重要的语义产生误解。

当绘制类图时，要遵循如下的策略：

- 要给出一个能反映出类图的用途的名称。
- 安排各个元素，尽量减少线段交叉。
- 在空间上组织元素，使得在语义上接近的事物在物理位置上也接近。
- 为了把注意力引导到类图的重要特性上，要使用注解或颜色作为可视化提示。
- 尽量不要显示过多种类的关系。通常，每个类图应由一种关系支配。