UML for Java Programmers 中文共享版

(草稿版 Ver 0.8)

[重要说明：本文档目前是我们快速完成的草稿版，肯定存在一些翻译问题，请各位阅读过的兄弟能够帮我们指出存在的问题，这将是对我们的工作的最大支持！请发邮件给我(wanghaibo@netease.com)或发帖在我们的讨论组上，谢谢！]

原著：Robert Cecil Martin

http://uml4java.126.com

翻译日志

NO:	章节	开始日期	结束日期	参与人	文档版本号
1	第二章到第四章	2004-4-7	2004-4-10	Bob Wang	Ver 0.4
2	第五章	2004-4-16	2004-4-16	Bob Wang	Ver 0.5
3	补上第一章	2004-4-18	2004-4-18	Bob Wang	Ver 0.51
4	第七章	2004-4-19	2004-4-28	Melthaw Zhang	Ver 0.6
5	完成第六章	2004-4-19	2004-4-29	Bob Wang	Ver 0.7
6	完成第八、九章	2004-4-19	2004-5-12	Orient Sun	Ver 0.8
7	完成第十章	2004-4-19	2004-5-12	LishiFeng	Ver 0.8

非常感谢各位参与者的辛勤努力！！！

目 录

第一章　针对 Java 程序员的 UML 概述

UML（统一建模语言）是一个绘制软件概念图的图形化记法(notation)。人们可以用它绘制图形，用这些图形来表示一个计划进行的软件设计的问题域，或者用这些图来表示一个已经完成的软件实现。Fowler（译者注：著名 IT 技术作家）描述它们时分成了三种不同的层次：**概念层（Conceptual）、规格说明层（Specification）**和**实现层（Implementation）**，我们将细述后面两种。

规格说明层和实现层的图形与源代码有明显的关系，实际上，规格说明层的图是准备用来转换成成源代码的，类似地，实现层的图是打算用来描述已经存在的源代码的。在这些层次的图形，有许多规则和语义学要遵从，这些图较少有歧义，基本上都有严格的格式。

在另外一方面，概念层上的图形与源代码没有什么严格的关系，它们与人类自然语言相关。它们是用来描述有关已经存在的人类的问题领域的概念和抽象的速记。它们无须遵从严格的语义规则，因此它们的意思理解会有歧义、主题可被解释。

考虑一下，以这个句子为例：一条狗(Dog)是一只动物(Animal)。我们能够创建表示这句话的一个概念层次的 UML 图，见 Figure 1-1。

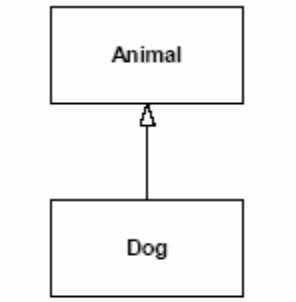


Figure 1-1
A Dog is an Animal

这个图描绘了通过泛化(generalization)关系连接起来的称为 Animal（动物）和 Dog（狗）的两个实体。一个 Animal 是一条 Dog 的泛化，一条 Dog 是一种特定的 Animal。这是所有这张图的意义了，没有什么其他意思可以从中推断出来了。我们可以声称，我们的这条叫 Sparky 的宠物狗是一只动物，我们或者可以谈到属于动物界的生物学的分类上去。因此，这张图是主题可解释的。

不过，这张图在规格说明层次和实现层次上有更明确的意思：

```
public class Animal {}
public class Dog extends Animal {}
```

这些代码定义了通过继承关系连接的 Animal 类和 Dog 类。但是这个概念模型没有涉及任何有关计算机、数据处理和程序，这个规格说明模型描述了程序的一部分。

不幸的是，这些图本身并不能说明它们描绘在哪个层次上了。无法识别出图的层次是造成程序员和系统分析员无法进行有意义的表达的根源。一个概念层次上的图没有定义源代码，也不应该去定义源代码。一个描述了某个问题的解决方法的规格说明层次上的图，它不会去寻找任何像概念层那样的问题的描述。

在本书中其余的所有图将是有关规格说明层或实现层的，并只要有可能，都会伴有对应的源代码。我们已经看完了我们最后的一个概念层次上的图了。

各种类型的图

以下是在 UML 中的主要使用的图的一个非常快速的教程，一旦你读完了它，你将能够阅读和编写那些你常用的大多数的 UML 图。其余那些将在接下来的章节中要叙述的是那些你熟练掌握 UML 的细节和体系。

UML 有三类主要的图，**静态图(static diagrams)**描述了那些不发生变化的软件元素的逻辑结构，描绘了类、对象、数据结构及其存在于它们之间的关系。**动态图(Dynamic diagrams)**展示了在运行期间的软件实体的变化，描绘了执行流程、实体改变状态的方式。**物理图(Physical diagrams)**显示了软件实体的不变化的物理结构，描绘的物理实体有源文件、库文件、字节文件、数据文件等等，以及存在于它们之间的关系。

考虑一下在 Listing 1-1 所示的代码，这个程序实现了一个基于简单的二元树算法的图。在你思考接下的图之前先自己熟悉一下这些代码。

类图(Class Diagram)

在 Figure 1-2 中的**类图**(Class Diagram)显示了在 Listing 1-1 所指的程序中的主要类及其它们之间的关系。它说明了一个 TreeMap 类有一个叫 add 和 get 的公共方法，说明了 TreeMap 持有一个到 TreeMapNode 类的实例 topName 的变量的**引用**（reference）。它也说明了每个 TreeMapNode 类持有一个到在某种容器中的其他的两个 TreeMapNode 类的实例--命名为 nodes 的引用，并且它说明了每个 TreeMapNode 实例持有到命名为 itsKey 和 itsKey 的其他两个实例变量的引用，这个 itsKey 变量持有一个实现了 Comparable 接口的实例的引用，这个 itsValue 变量简单地持有有些对象的引用。

Listing 1-1
TreeMap.java

```
public class TreeMap {
  TreeMapNode topNode = null;

  public void add(Comparable key, Object value) {
    if (topNode == null)
      topNode = new TreeMapNode(key, value);
    else
      topNode.add(key, value);
  }

  public Object get(Comparable key) {
    return topNode == null ? null : topNode.find(key);
  }
}

class TreeMapNode {
  private final static int LESS = 0;
  private final static int GREATER = 1;
  private Comparable itsKey;
  private Object itsValue;
  private TreeMapNode nodes[] = new TreeMapNode[2];

  public TreeMapNode(Comparable key, Object value) {
    itsKey = key;
    itsValue = value;
  }

  public Object find(Comparable key) {
    if (key.compareTo(itsKey) == 0) return itsValue;
    return findSubNodeForKey(selectSubNode(key), key);
  }

  private int selectSubNode(Comparable key) {
    return (key.compareTo(itsKey) < 0) ? LESS : GREATER;
  }

  private Object findSubNodeForKey(int node, Comparable key) {
    return nodes[node] == null ? null : nodes[node].find(key);
  }

  public void add(Comparable key, Object value) {
    if (key.compareTo(itsKey) == 0)
      itsValue = value;
    else
      addSubNode(selectSubNode(key), key, value);
  }
  private void addSubNode(int node, Comparable key,
                          Object value) {
    if (nodes[node] == null)
      nodes[node] = new TreeMapNode(key, value);
    else
      nodes[node].add(key, value);
  }
}
```

我们将下一章节中仔细体会类图的微妙之处。对此现在来说，你仅仅需要知道如下一些事情：

- 长方形表示类、箭头表示关系；
- 在图中所有的关系叫关联(associations)，关联是简单的数据关系，用来表示一个

对象持有一个另外一个对象的引用，或是调用另外一个对象的方法；

- 关系的名称映射到持有的引用的变量名称；
- 挨着箭头的数字通常用来说明关联持有的实例的数量，如果这个数字大于 1 说明采用了某些容器，通常是使用了一个数组；
- 类图标可以有多于一个的框格，最上面的框格永远是表示类的名称，其他的框格描述函数和变量；
- «interface»符号表示 Comparable 是一个接口(interface)；
- 大多数符号是可选的。

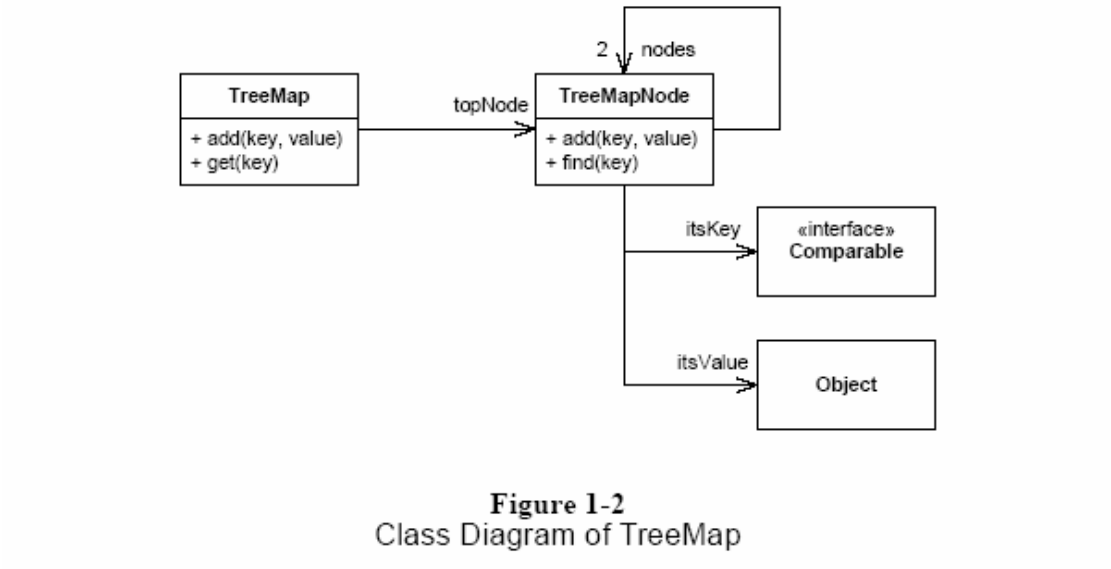


Figure 1-2
Class Diagram of TreeMap

仔细查看这个图和在 Listing 1-1 中相关的代码。注意这个关联关系是如何对应到实例变量的。例如，从 TreeMap 到 TreeMapNode 的关联被命名为 topNode，并对应到在 TreeMap 类中的 topNode 变量。

对象图(Object Diagram)

Figure 1-3 是一个对象图(Object Diagram)，它说明了系统执行期间在某一特定时刻的一组对象及其关系。你可以把它们视作内存的一个快照(snapshot)。

在这张图中，长方形图标表示对象，因为它们的名称是有下划线的，所以你能够断定它们是对象。冒号（“ :”）号后面跟着的名称是这个对象所属的对象的名称。注意，每一个对象的最下面的框格说明了对象的变量 itsKey 的值。

对象之间的关系被称为链接(links)，是从在 Figure 1-3 中的关联被派生出来的。注意，这个链接为在 nodes 数组中的两个数组单元进行了命名。

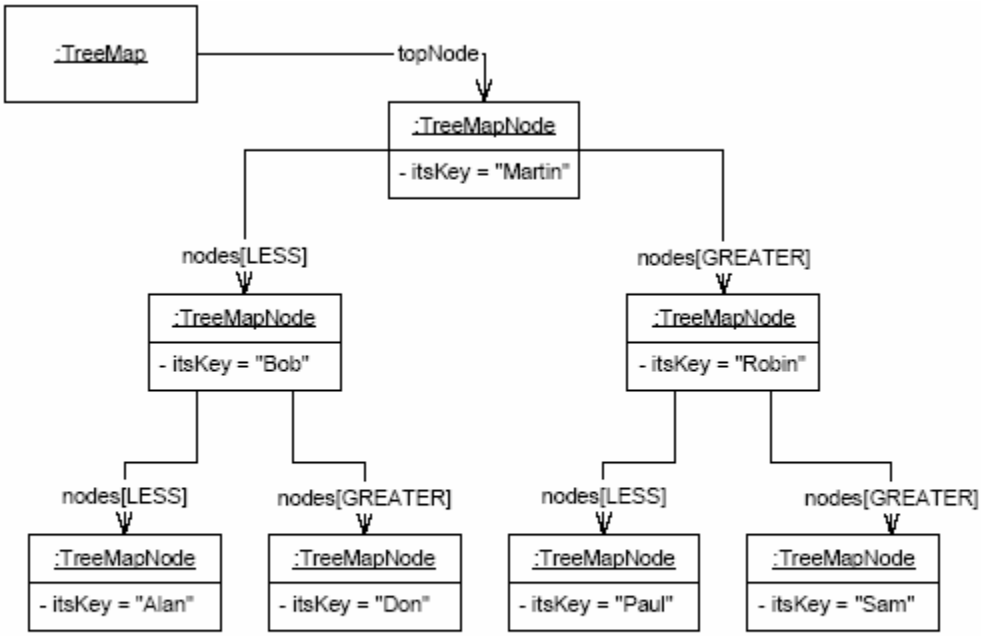


Figure 1-3
TreeMap Object Diagram

序列图(sequence diagram)

Figure 1-4 是一个序列图(sequence diagram)，它描述了 TreeMap.add 方法是如何被实现的。

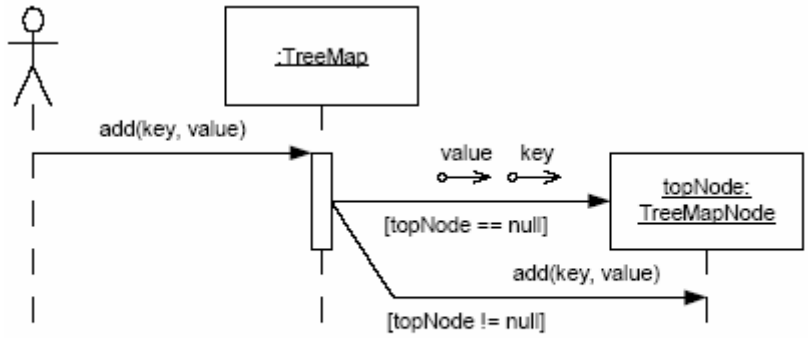


Figure 1-4
TreeMap.add

这个人样图表示一个未知的调用者，这个调用者调用在一个 TreeMap 对象上的 add 方法。如果这个 topNode 变量是空(null)，这个 TreeMap 负责创建一个新的 TreeMapNode 并将它赋值为 topNode。另外，这个 TreeMap 发送一个 add 消息给 topNode。

在方括号（“ [] ”）内的表达式称为监护(guards)，它说明了采取什么的路径。这个消息箭头终止在 TreeMapNode 图标上表示构造(construction)。这个带小圆圈的小箭头叫数据标

记(data tokens)，在这个例子中它描述了构造参数。在 TreeMap 下面的小长方形叫做活动（activation），它描述了 add 方法执行时间的长短。

协作图(collaboration diagram)

在 Figure 1-5 中所示的图称为协作图(collaboration diagram)，它描绘了在 treeNode 不是空的情况下 TreeMap.add 的状况。协作图包含了序列图包含的同样的信息，不过序列图清楚地描述了消息的先后次序，而协作图清楚地描述了对象间的关系。

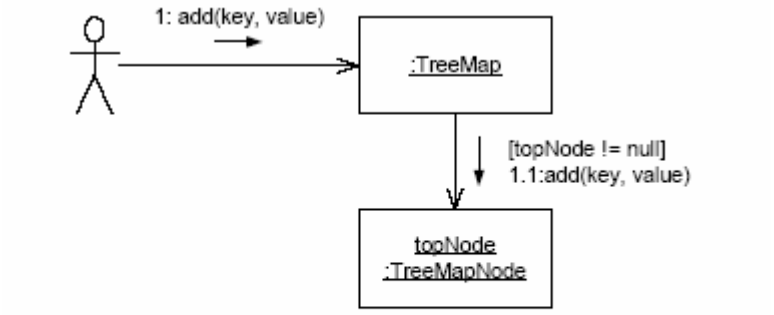


Figure 1-5
Collaboration Diagram of one case of TreeMap.add

对象通过叫链接(links)的关系被连接起来了，一个链接存在于一个对象发送消息给另外一个对象的时。在那些链接上传播的是消息本身。它们被描绘成一个小箭头。消息被标记上消息的名称、序列数和一些监护。

序列数的点结构说明了调用的继承性，TreeMap.add 函数（消息 1）调用了 TreeMapNode.add 函数（消息 1.1），因此，消息 1.1 是被消息 1 调用函数时第一个被发送的消息。

状态图（State Diagrams）

UML 有一个针对限定状态机(finite state machines)的非常容易理解的标记符号，Figure 1-6 显示了一个相当简单的标记子集。

Figure 1-6 显示了一个针对地铁十字转门（译者注：地铁验票设施）的状态机。有名为 Locked 和 UnLocked 的两个状态。两个事件能够被发送到这个机器，coin 事件意味着用户已经丢进了一个硬币到十字转门，pass 事件意味着用户已经通过了十字转门。

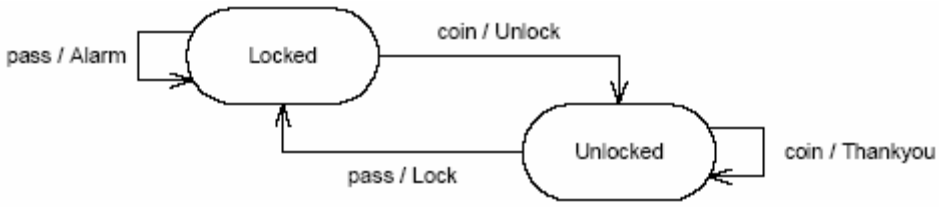


Figure 1-6
State Machine of a Subway Turnstile

这个箭头被称为转换(transitions)，它们用触发这个转换的事件名称及转换执行做了标记，当一个转换被触发时，这个系统的状态发生了改变。

我们将 Figure 1-6 翻译成如下的描述：

- 如果我们是在 Locked 状态，并且我们得到一个 coin 事件时，我们就转换到 Unlocked 状态并且我们调用这个 Unlock 函数。
- 如果我们是在 Unlocked 状态，并且我们得到了一个 pass 事件时，我们应转换到 Locked 状态并且我们调用 Lock 函数。
- 如果我们在 Unlocked 状态，并且我们得到了一个 coin 事件时，我们停留在 Unlocked 的状态并且我们调用 Thankyou 函数。
- 如果我们是在 Locked 状态，并且我们得到了一个 pass 事件时，我们停留在 Locked 状态并且调用一个 Alarm 函数。

像这种图用来描绘出系统的行为方式是极其有用的。它们给了我们机会去研究系统在一个未预料的情况下系统将如何做，像当用户丢进了一个硬币，然后因为不好的原因丢进了另外一个硬币。

小结

在本章中展示的图对于大多数情况已经足够用了。大多数程序能够在没有了解比这里展现的更多的 UML 的情况下做得很好的。

参考文献

[Fowler00]: UML Distilled, 2d. ed. Martin Fowler, Addison Wesley, 199?

第二章　使用图(Diagrams)

在我们了解 UML 的细节之前，谈论何时使用、如何使用 UML 是明智的。UML 的误用和滥用已经给相关的软件项目造成了极大的危害。

为什么用模型？

为什么工程师要建造模型(models)？为什么航天工程师要建造航天器的模型？为什么桥梁工程师要建造桥的模型？提供这些模型的目的是什么？

这些工程师建造模型来查明他们的设计是否可以正常工作。航天工程师建造好了航天器的模型，然后把他们放入风洞中了解这些航天器是否可以飞行。桥梁工程师建造桥的模型来了解桥能否接立起来。建筑工程师建造建筑的模型了解客户是否喜欢这种建筑模样。通过建立模型来验证事物是否可工作。

模型意味着它们必须是可被检验的。为了检验它，如果一个模型没有一个可用的检验标准，它是相当糟糕的。如果你不能评估一个模型，这个模型是没有价值的。

为什么航天工程师不马上建造一个飞机然后去试飞呢？为什么桥梁工程师不立即建造一座桥然后看它是否可以接立起来呢？因为航天器和桥的造价比模型昂贵多了。当模型比我们实际建造的东西划算多了的时候，我们用模型来研究设计。

为什么给软件建模？

一个 UML 图可被检验吗？它比创建、检验这个软件更划算吗？对于这两个问题，这个答案是无法像航天工程师和桥梁工程师那样清楚地了解境况。没有一个检验一个 UML 图的固定标准。我们能够观察它、评估它，然后应用原则和模式于它，但是最后的评估仍然是相当主观的。画 UML 图比编写软件花费更少，但不是重要的因素。当然，改变一个 UML 图比修改源代码容易多了，用 UML 是不是有意义呢？

如果使用 UML 没有什么意义的话，我就不会写这本书了。不过，以上举例说明 UML 的易用是误用了。当我们需要通过检验确定某些东西的时候，或是使用 UML 来检验比编码来检验更划算的时候，我们就使用 UML。

举一个例子，我有一个特定设计的主意，我需要通过我的团队中的开发人员来考虑它是

不是一个好主意去检验，因此，我在白板上画出了一个 UML 图，然后询问队友们的反馈。

我们为什么应该在编码前构造一个全面的设计？

桥梁工程师、航天工程师和建筑工程师都画设计图，为什么呢？因为画一个房子的设计图一个人就可以了，而建造它需要五个或更多人。区区十来个航天工程师能画一个飞机的设计图，而需要上千人去建造它。绘制设计图不需挖掘地基、浇注混凝土和悬挂窗户。简而言之，预先计划一个建筑物远比没有计划的情况下试图建筑它更划算。丢弃一张有错误的设计图花不了多少钱，而拆卸一栋失败的建筑物却要花不少的钱。

另外，软件中的裁剪也不是那样彻底，比编写代码更划算的编制 UML 图的裁剪也不是特别地彻底。实际上，许多项目团队在 UML 图上花费了比编写代码本身更多的时间。弃用一个图比弃用代码是不是更划算，这也不一定。因此，在编写代码前去创建一个全面的 UML 设计作为一个有价值、有效的选项，也是不一定的。

有效地使用 UML

显然，无论是桥梁工程师、航天工程师还是建筑工程师都无法为软件开发打一个比喻。我们无法愉快地使用 UML 那些用于设计图和模型的规则。因此到底我们什么时候、为什么应当 UML 呢？

人员之间传达

UML 在软件开发人员之间传达设计概念是非常方便的。在一小群开发人员中，许多事情可在一个白板上进行。如果你有一些主意需要传达给其他人员，用 UML 可能是非常好的。

UML 用于表达集中的设计思想相当不错，举一个例子，在图 Figure 2-1 中的图描述得非常清楚，我们知道 LogiServlet 实现了 Servlet 接口，并使用了 UserDatabase。很明显，LoginServlet 需要 HTTPRequest 类和 HTTPResponse 类。很容易想像一群开发者围站在一块白板前为一张这样的图讨论的情景。实际上，这张图清楚地描述了代码的结构。

另一方面，在表达算法细节时 UML 不是特别理想。考察一下在 Listing-2-1 中的这个简单的冒泡排序代码，表达这个简单的模块用 UML 不是令人非常满意。

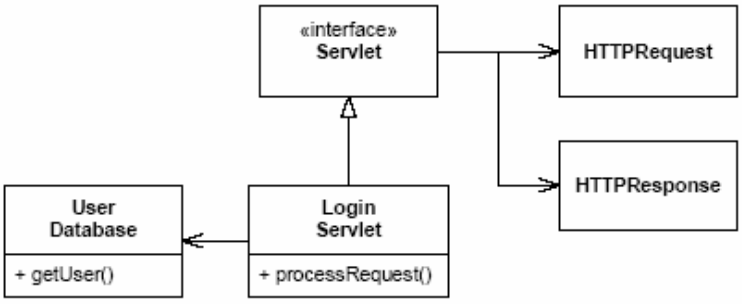


Figure 2-1
Login Servlet

Listing 2-1
Bubble Sort

```
public class BubbleSorter
{
  static int operations = 0;
  public static int sort(int [] array)
  {
    operations = 0;
    if (array.length <= 1)
      return operations;

    for (int nextToLast = array.length-2;
         nextToLast >= 0; nextToLast--)
      for (int index = 0; index <= nextToLast; index++)
        compareAndSwap(array, index);

    return operations;
  }

  private static void swap(int[] array, int index)
  {
    int temp = array[index];
    array[index] = array[index+1];
    array[index+1] = temp;
  }

  private static void compareAndSwap(int[] array, int index)
  {
    if (array[index] > array[index+1])
      swap(array, index);
    operations++;
  }
}
```

在 Figure 2-2 中的图给我们一个粗略的结构，但是它不方便、也无法反映出让人感兴趣的细节。在 Figure 2-3 中的图不比代码容易读，况且它还相当难以创建。UML 在这方面的用途留下了许多需要解决的问题。

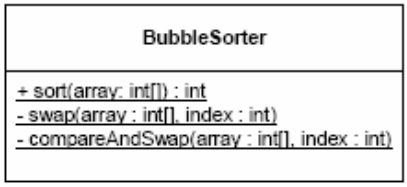


Figure 2-2
Bubble Sorter

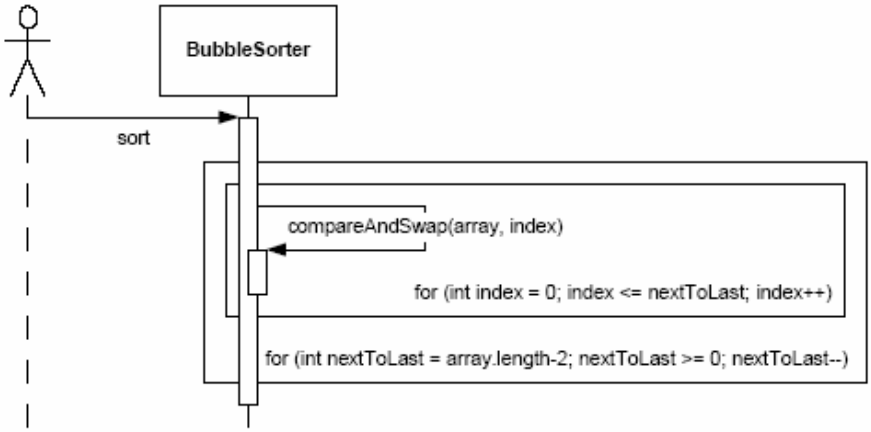


Figure 2-3
Bubble Sort Sequence Diagram

UML 在创建大型软件结构的“ 路标图 ”(roadmaps)时是比较有用，这样的“ 路标图 ”给开发人员一个快速的手段，用来发现某一个类依赖于另外哪些类，并为整个系统的结构的提供了一个参考。

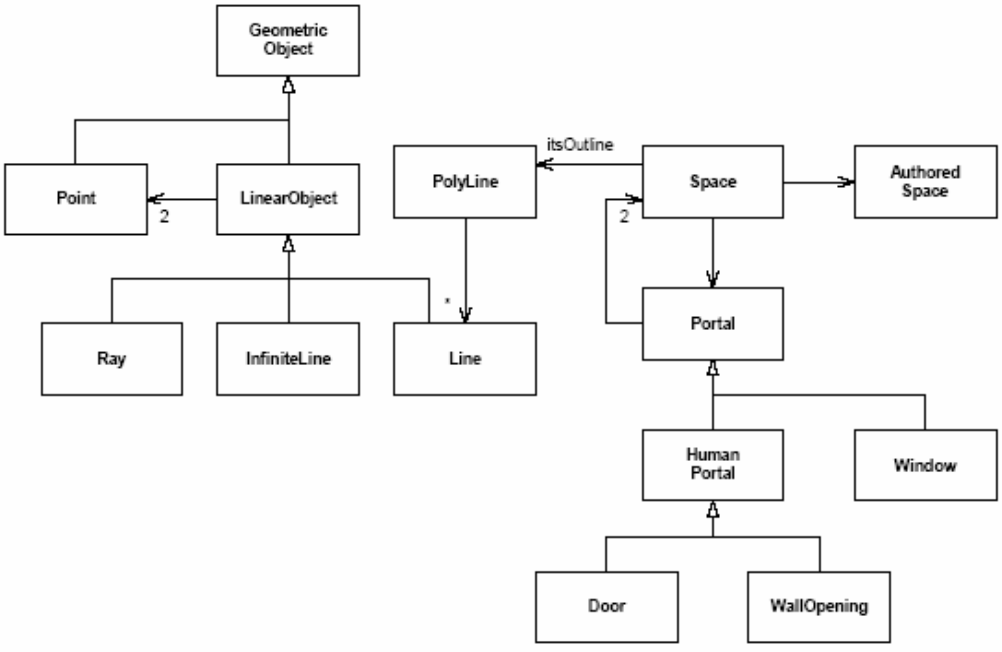


Figure 2-4
Roadmap

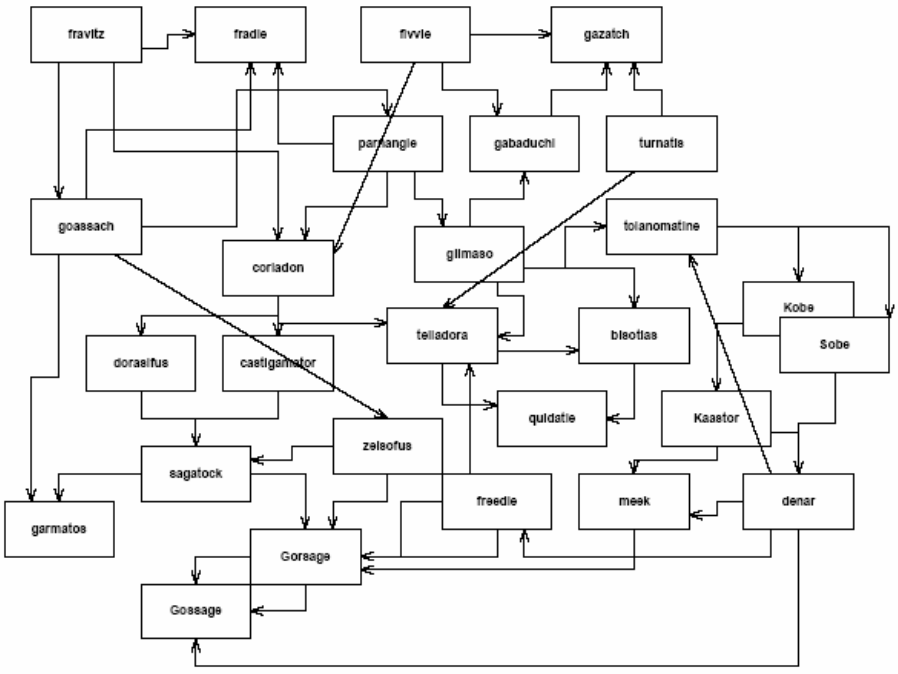
例如在 Figure 2-4 中，非常容易了解 Space 对象引用一个 Polyline 对象，而 Polyline 对象由许多由 LinearObject 对象派生出来的 Lines 对象构造成，而 LinearObject 对象则包括了两个 Points 对象。在代码中去发现这些结构是相当乏味的、累人的，而在“路标图”中去发现结构却是轻而易举的。

这种“路标图”是一种非常有用的教学工具。可是，任何一个团队成员应该可以在白板上画出这样一个图作为一个临时的布告。实际上，在五年前的开发的工作中，我从脑海里画出了有关一个系统的如上所示的图。这样的图记录了所有的开发人员必须记住的，可以用于有效地开发系统的相关知识。因此，大多数情况下，准备费力去创建和归档一大堆这们的文档基本没有什么意义，最好的方式，再说一次，就在白板上画图吧。

最后的文档

什么时候最适合创建一个设计文档呢？最好在团队所有的工作干完了，项目要结束了的时候。这样的文档真实地反映了一个团队工作结果的状况，它对接着即将开展工作的团队极有帮助。

不过，这里有一些缺陷。UML 图需要经过深思熟虑。我们不需要一个有上千页的序列图，相反，我们需要一个非常明显的描述了系统主要观点的图。没有 UML 图比一个由许多线和许多框纠缠在一起组成的混乱的图更糟糕的情况了，如图(Figure 2-5)，千万不要这样做。



保留什么，舍弃什么？

养成舍弃 UML 图的习惯吧，最好，养成不要把图建立在能长期保存的介质上习惯。在一个白板或一个草稿纸上画它们，经常擦掉白板或丢弃草稿纸。一个原则就是不使用 Case 工具或一个画图工具。这些工具只用一次即可，你的大多数 UML 图都是短命的。

但是有些图保存下来非常有用：

- 表现你的系统中一个通用设计解决方案的图
- 记录了复杂的协议，难以通过代码了解的图
- 提供了比较少涉及到的系统范围内的“路标图”的图
- 记录了比代码更易表述的设计意图的图

认真去辩认这些图是没有意义的，你一看到他们去明白了。没有必要预先建立这些图，如果你猜的话，你会猜错的。这些真正有用的图将用反复浮现出来，它们将一次次地在设计时出现在白板上或草稿纸上。最后将有一个人将最终确定的图制作一个长期保存的备份，把图放在一个每一个人能够访问得到的公共区域。

公共区域必须保持简洁和访问方便，把这些有用的图放在一个 Web 服务器上或一个网络知识库中是一个比较好的主意。不过，不要使用成千上万张图全堆积在一起。明辨出哪些图是真正有用的图和哪些是将会在下一次临时通知中被重新创建的图。保留那些非常价值、能够长期保存的图。

迭代精化

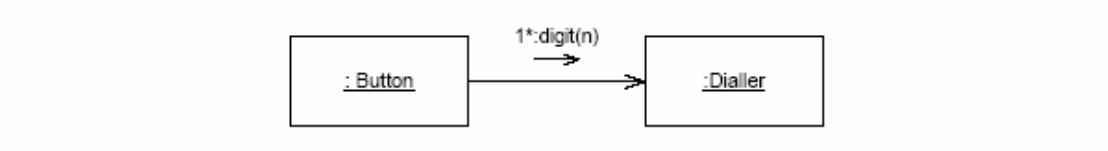
如何建立 UML 呢？是在灵光乍现时去画下他们？是不是一定要先画类图再画序列图呢？是不是我们丰富完成了部分细节之前一定要搭建好整个系统的架构呢？

答案是：绝对不是! 每一个人在一小步的细节上都是做得比较好，并且能够很好地评估它，但是无法实现一次大的跳跃。我们渴望建立非常有用的 UML 图，因此我们将在每一小步创建好他们。

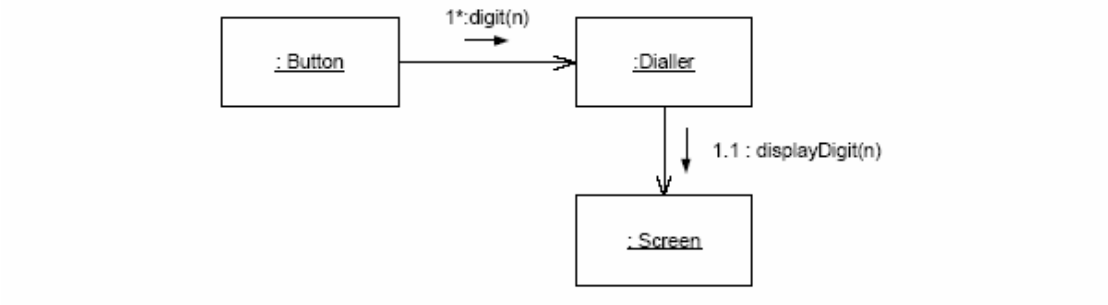
行为(Behavior)优先

我喜欢先处理行为，我想 UML 将帮助我彻底地解决上述问题，我将先开始画一个有关问题的简单序列图。想一下，举一个例子，这个控制便携式电话的软件是如何完成一个电话呼叫呢？

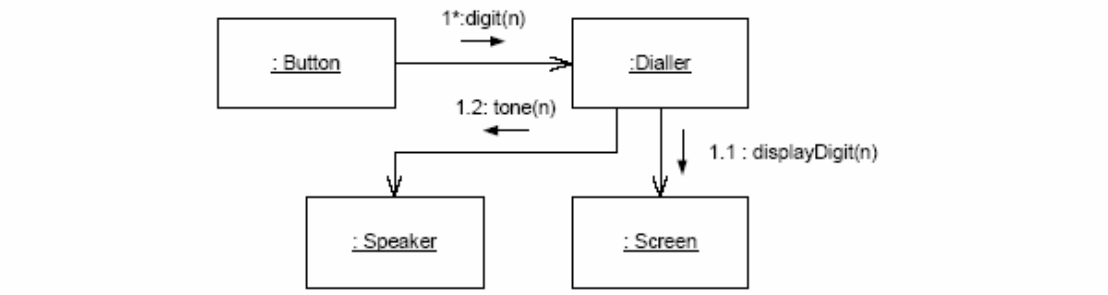
我们可以想象一下，软件检测到每一个按钮的操作，然后发一个消息给控制拨号的对象。因此，我画了一个 Button 的对象和一个 Dialler 对象，并且显示出 Button 对象发一个 digit 消息给 Dialler 对象。



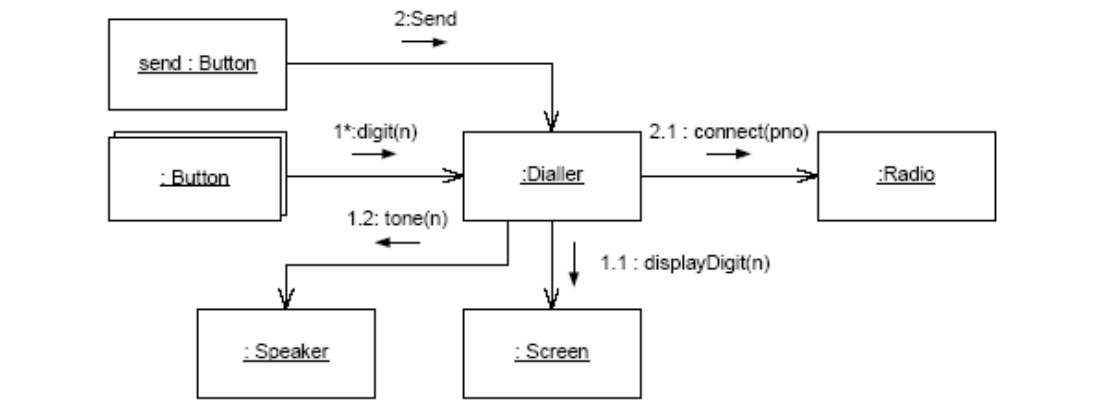
当 Dialler 接收到这个 digit 消息的时候，它将做什么呢？是的，它需要在一个屏幕上显示数字出来。因此，也许它将发送一个 displayDigit 消息给 Screen 对象。



接下来，这个 Dialler 对象最好从扬声器中发一个拨号音，因此，我们将发送一个 tone 消息给 Speaker 对象。



有时用户将按“ 发送 ”按钮要完成呼叫，这时，我们将指示便携式电话连接到无线网络，并发送出被拨打的电话号码。



当连接被建立后，这个 Radio 对象能够告诉 Screen 对象去显示出“ 使用中 ”的指示状态。这个消息将被在另外一个控制线程中被发出(它将会用序列号前面的字符来描述)。最后的协作图将如下图 Figure 2-6 所示：

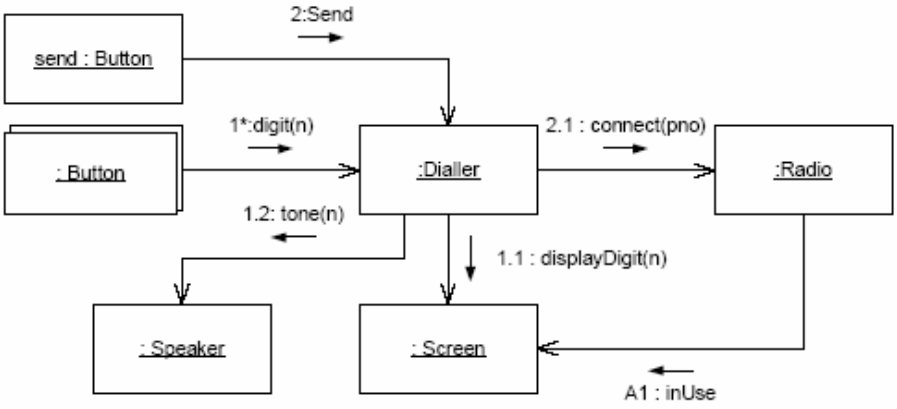


Figure 2-6
Cell Phone Collaboration Diagram

检查结构

上述的小练习展示了如何从无到有地建立一个协作图。注意，我们按照这种方法创建了对象，但是我们之前并不知道对象将会在哪里。我们只知道某些事情将会发生，对象会被创建。

但是在进一步深入探讨之前，我们需要检验一下这个协作图对代码的结构意味着什么。因此，我们将建立支持这个协作图的类图，这个类图将会为协作图中的每一个对象建立一个类(class)，建立针对协作之间联接的联系(association)。

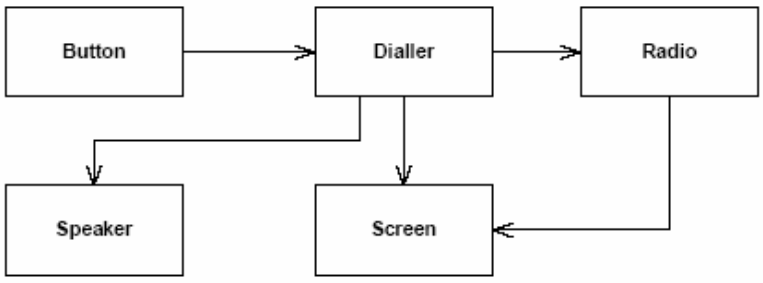


Figure 2-7
Cell Phone Class Diagram

那些熟悉 UML 的读者已经注意到我已经忽略了聚合(aggregation)和组成(composition)，这是有意的。这里有大量的时间去考虑在哪里建立这些联系(relationships)。

现在对我们来说什么是一个相关性分析的重点呢？为什么 Button 对象要依靠 Dialler 对象呢？如果你想到这里，显得有些可怕了，考虑一下如下的代码：

```
public class Button
{
  private Dialler itsDialler;
  public Button(Dialler dialler)
  {itsDialler = dialler;}
  ...
}
```

我不想有关 Button 的代码引用到 Dialler 的代码，Button 是一个我能够在不同的环境下使用的类。举个例子，我可能用 Button 类去控制开发的“ 通或断 ”，或者是一个菜单按钮，或是一个电话的控制按钮。如果我将 Button 类绑定到 Dialler 类，我就不用将 Button 的代码重用于其他的目的了。

我将通过在 Button 和 Dialler 之间插入一个接口(interface)的方法来解决这个问题。如图 Figure 2-8 所示。我们看到每一个 Button 用一个标记(token)去标识它。当 Button 类检测到实际上的按钮被按下时，它将调用 ButtonListener 接口的 buttonPressed 方法，并传送这个标记。这将打破 Button 对 Dialler 的依赖，并允许 Button 被实际使用于需要接到到按钮按下操作的任何地方。

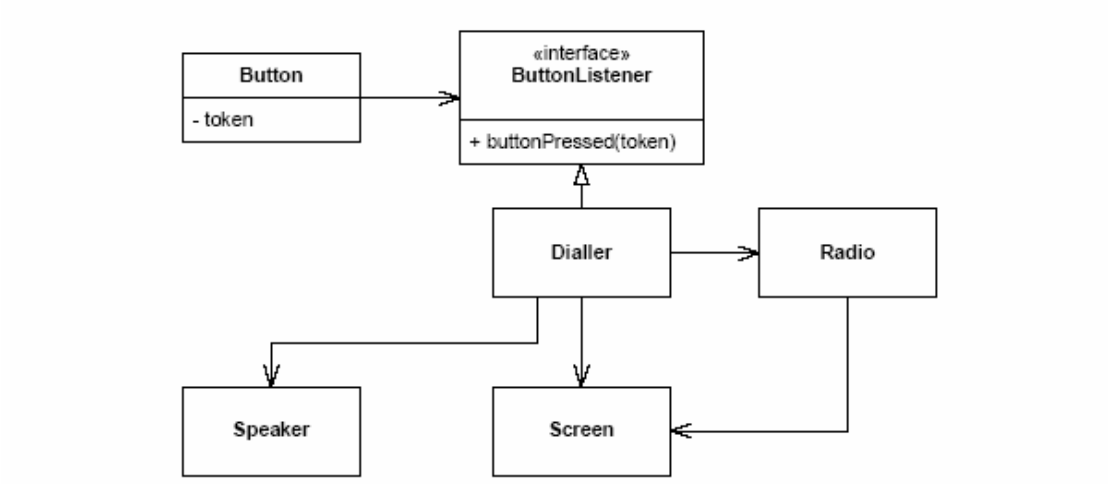


Figure 2-8
Isolating Button from Dialler

我们注意一下，上述的改变并未影响到如图 Figure 2-6 的动态图，对象还是一样的，只是这些类已经发生了变化。

不幸的是，我们已经让 Dialler 知道了有关 Button 的一些情况，为什么 Dailler 期望从 ButtonListener 得到输入？为什么它会有一个叫做 buttonPressed 的方法？这个 Dialler 通过 Button 做了什么？

我们能够通过用一批小的适配器可以解决这个问题，解除所有没有意义的标记。这个 ButtonDiallerAdapter 实现了 ButtonListener 接口。它接收 buttonPress 方法并发送 digit(n) 消息给 Dialler，这个 digit 通过这个接口传送给 Dialler。

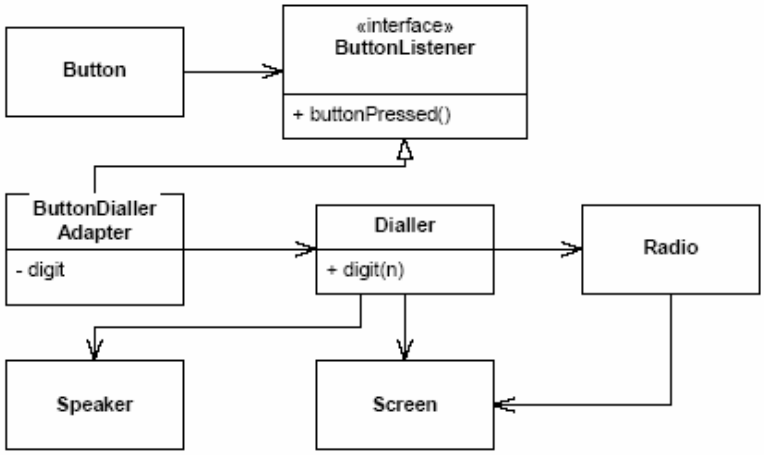


Figure 2-9
Adapting Buttons to Diallers

在脑海中想像这些代码

我们能够容易地想象出有关 ButtonDiallerAdapter 的代码。如 Listing 2-2 所示。当你做图的时候能够想象到你的代码是极其重要的。我们把图当作了解代码的一条捷径，并不是替代代码。如果你画着图但是不能想像出它代表着什么样的代码，你是正在空气中建筑着城堡。**停下来你正在做的，想想如何如能将它可以转化成代码**。不要为了图而画图，你必须时时刻刻记得，代码才是你要表现的。

Listing 2-2
ButtonDiallerAdapter.java

```
public class ButtonDiallerAdapter implements ButtonListener
{
  private int digit;
  private Dialler dialler;
  public ButtonDiallerAdapter(int digit, Dialler dialler)
  {
    this.digit = digit;
    this.dialler = dialler;
  }

  public void buttonPressed()
  {
    dialler.digit(digit);
  }
}
```

迭代精化

请注意，我们在 Figure 2-9 图中的最后的改变已经使在 Figure 2-6 中的动态模型已经无效了。这个动态模型根本不知道任何适配器，我们将改变这种情况，如 Figure 2-10 所示：

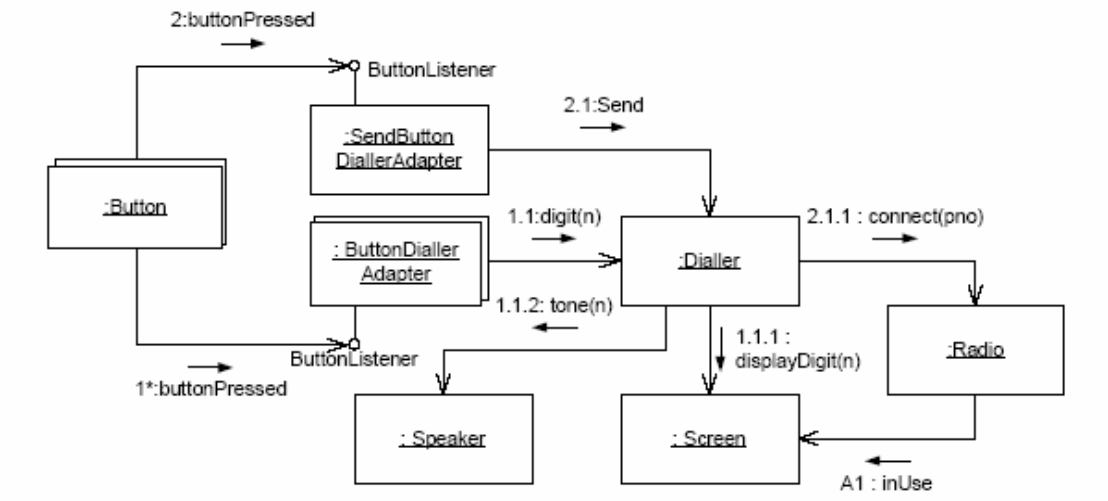


Figure 2-10
Adding Adapters to the Dynamic Model

这显示了这些图以一种迭代的方式进行了完善，你从一个动态方面的小处着手，然后研究什么是动态方面与静态关系之间的内在意义。你遵从好的设计原则去修改这些静态关系，然后回过头来去改善这些动态图。

每一步骤都是细小的，在研究静态结构的内在意义之前，我们不使用超过 5 分钟时间去了解动态图。在考虑动态行为的影响之前，我们花费不超过 5 分钟的时间去精化这个静态图。最好，我们用一个非常短的时间去完善这两张图。

记住，我们尽可能地在白板上去做这些，我们尽可能地不记录最后答案的中间结果。我们无须非常规范和非常精确。实际上，我在图上已经包括的两张图远比一般你做的更精确、更规范。使用白板的目的不是为了正确地标出图上序列号的点，而是使每个围在白板前讨论的人都理解它，最终目标是停止在白板上画而开始编写代码。

最低纲领

这本书给你展示了大量可以美化一个 UML 图的修饰符号和图标，使用它们可以让你做一个相当复杂的 UML 图，能够表现出一些令人难以置信的细节信息。不过，我建议你不要这样做。

就像我下面要讨论一样，图在同事间进行表达比较有用，它将帮你解决设计上的问题。你仅用一定数量的、需要的细节去完成你的目标。弄出一个充斥着修饰符号的图不是不可以，但是它是效能低下的。保持你的图简洁，UML 图不是源代码，也不能视作方法、变量和关系声明。

什么时候和如何画图

画 UML 图是一种非常有用的活动，它也可能成为一种浪费时间的、可怕的活动。使用 UML 的决定是一件好事，也可能成为一件坏事。它依赖你确定怎么使用、多大范围内的使用它。

什么时候画 UML 图，什么时候停止

不要制定什么都必须画图的规则，这样的规则将比不用更糟糕。项目的大量时间和精力将会被浪费在追逐那个根本没有人去读的图上。

什么时候画图：

- 当许多人一起需要同时进行开发时，这些人需要都理解一个系统的特定部分的设计结构时，开始画图。当所有的人都已经声明理解了的时候，结束画图。
- 当两个人或更多人不同意一个特定的元素如何设计的时候，你需要你的团队意见一致的时候，要找一个时间进行讨论做出决定，比如投票，或一个公正的宣告的方式进行，这时你需要画图。当决定做出来后，擦掉这些图。
- 当你需要探讨一个设计的想法时，画图能够帮你更好的地思考。当你得到了能够帮助你完成思考的代码的要点的时候，扔掉这些图。
- 当你需要向其他人或自己解释一部分代码的结构的时候，你可以画图。当你觉得其实最好看代码来进行解释的时候，停止画图。
- 当项目快要结束，顾客需要你将图与其他文档一起提供的时候，你开始画图。

什么时候停止画图：

- 不要因为觉得过程需要而画图。
- 不要因为你有一个觉得一个好的设计者必须画图，不画图反而感觉有罪的想法而去画图。好的设计者只有在需要的时候才去编写代码和画图。

- 不要在编码之前的设计阶段去为创建全面的文档而画图，这样的文档基本没有意义并且将会浪费大量的时间。
- 不要为了其他人编码而去画图。真正的软件构架师将去实现自己的设计的编码。这样他们反而简单了。

Case 工具

UML Case 工具能够有用，但是它们相当昂贵。谨慎地做出购买和部署一个 UML case 工具的决定。

- UML case 工具是不是让你画图的工作更容易？不是的，他们让画图更难了。有一个长的学习了解的周期。工具有时比白板更讨厌，白板非常容易用，开发都通常都已经非常熟悉它，至少，它没有学习周期。
- UML case 工具使大的团队协作画图更容易？在有些情况下，大多数开发者和开发的项目不是真正需要产生如此大量和复杂的图，不需要一个自动协作系统去协调他们的开发活动。在有些情况下，只有当一个手工的协作系统到了开始显得不够用地步，没有其他的选择的时候，才去考虑购买一个 UML 图的协作系统。
- 是不是 UML case 工具让代码生成更容易呢？包括建立图、生成代码、使用代码的一系列努力看起来不比一开始就写代码轻松。如果有益处，它们不是一个巨大的需求、甚至不是需要选择两种方式之一的一个因素。开发者知道如何使用一个 IDE 和修改一个文本文件。从图生成代码看起来是一个不错的主意，但是我强烈建议你在花了大量的银子之前权衡一下效能的提升。
- 那些包括了 IDE 和同时显示代码和图的 CASE 工具如何？这些工具无疑是比较 Cool 的。不过，我不认为没完没了的 UML 表现是多么的重要。实际上，当我改变代码时也改变图，或者我改变图的时候也改变代码，并没有真正帮助我多少。坦白地讲，我宁愿买一个 IDE 来尽力帮助我应付程序而非图。再说，在花大钱成交之前权衡一下效能的提升吧。

简而言之，对于购买 Case 工具你一定要三思而行。也许为你的团队配备一个昂贵的 Case 工具能够带来好处，但是购买前凭你自己的经验核验一下购买的好处。

用文档如何？

任何一个项目都离不开好的文档，没有了它，团队将迷失在代码的海洋里。从另外一方

面来说，太多错误类型的文档也是糟糕的，因为你尽是些分散精力和令人误解的纸张，你仍将迷失在代码的海洋里。

必须建立文档，但是必须谨慎地创建文档，通常选择不创建文档和创建文档是一样重要的。一个复杂的通讯协议需要创建文档，一个复杂的关系型模式需要创建文档，一个复杂的可重用的框架需要创建文档。

不过，无须上百页的 UML 文档。软件文档应该言简意赅，一个软件文档的价值通常与文档的大小成反比。

对于一个有 12 个人共同工作、编写 100 万行 Java 代码的项目团队，我想会固定保存的文档将会是 25 到 200 页，我本人宁愿是更少。这包括一些有关重要模块的高层结构的 UML 图的文档、关系型模式的 ER 图、一到两页纸描述如何建立系统、测试指令、源代码控制指令等等。

我将把这些文档放到一个 wiki 中，或有些协作编写工作中，这样团队中的每一个人都能够从他们的屏幕上访问或搜索到它们，每一个人都能够在需要的时候更新它们。

需要花一些功夫去让文档变小，这样的工作是有意义的。人们将会阅读小的文档，而不是上千页的大部头书。

用 Javadocs

Javadocs 是非常优秀的工具。创建它们，但保持它们小而精要。那些供他们使用的功能描述需要仔细编写，需要包含足够的信息用户理解。Javadocs 那些表示私有工具函数或方法不进行分发的话，将使文档更简短。

小结

一些人站在白板前能够使用 UML 他们更好地思考一个设计问题。图将会在一个极短周期内以迭代的方式创建。最好首先研究动态场景，然后确定静态结构的内在涵义。用不超过 5 分钟的、非常短的迭代周期内，同时完善动态图或静态图是相当重要的。

UML Case 工具在某些情况下能有用，但是对于大多数的开发团队来说，它可能成为一种障碍。如果你觉得你需要一个 UML case 工具，甚至一个集成了 IDE 的 UML Case 工具，首先进行效能的实验，三思而后行。

UML 是一种工具，工具本身不是最终的目的。作为一种工具，它能够帮助你思考你的

设计和在同事间进行信息传达。节制地使用它们，它们将给你极大的帮助。如果滥用它的话，它将极大地浪费你的时间。当用到 UML 的时候，尽量精简！

第三章　类(Class)图

UML 的类图允许我们去标记静态内容及其类之间的关系。在一个类图中，我们能够查看一个类的成员变量和成员函数。我们也能查看一个类是否继承自另外一个类，是否拥有对另外一个类的引用。简而言之，我们能够描绘出类之间的源代码依存关系。

这是非常有价值的，从一个图上去评估一个系统的依存结构比从代码中去评估容易多了。图让一些依存结构可视化了。我们能够了解依存回路关系，并决定如何以最佳的方式打破它。我们能够了解什么时候抽象类依赖具体类，并确定一个更改这种依存关系的策略。

基础知识

类

Figure 3-1 展现了一个类图的最简单的形式，这个名叫 Dialler 的类代表一个简单的长方形。这个图代表的东西并不比右边的代码要多。

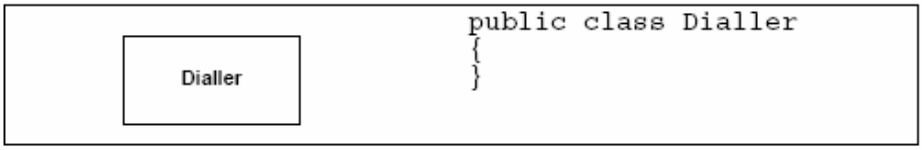


Figure 3-1
Class Icon

这是表现一个类的最常用的方法，大多数图的类有一个能够清楚表达的命名就可以了。

一个类的图像符号被细分成几个框格，最上面部分表示类的名字，第二个框格表示类的变量，第三个框格表示类的方法。Figure 3-2 展示所有的框格及对应的源代码。

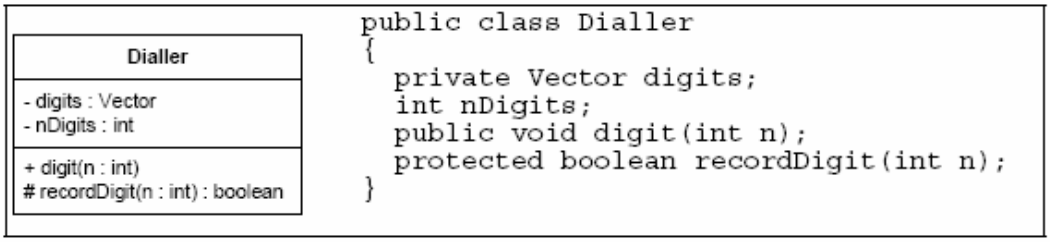


Figure 3-2

请注意在类图像符号里，在变量和函数的前面一个字符，一个“ - ”表示变量或函数是私有（private），“ # ”表示变量或函数是受保护(protected)的，“ + ”表示变量或函数是公开的(public)。

紧接在变量或参数名称的冒号（:）号之后，表示了变量的类型或一个函数的参数的类型。同样地，函数的返回值的类型是在函数后面的冒号之后反映的。

这种细节有时有用，但是用得并不是特别多。UML 图并不是声明变量和函数的地方，这种事情最好在源代码里去做。只有是在为了做图的时候才去使用这些修饰符号。

关联

类之间的关联大多用来表示变量实例持有着对其他对象的引用。举一个例子，在 Figure 3-3 中，我们看到在 Phone 和 Button 之间一个关联，这个箭头的方向告诉我们是 Phone 持有着 Button 的引用。靠近箭头的地方标有着变量实例的名称，箭头附近的数字告诉我们被引用的数量。

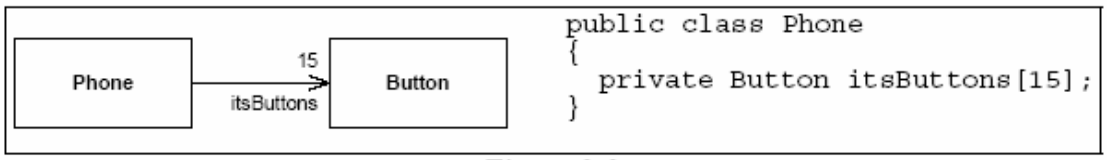


Figure 3-3

多重性

这个箭头附近的数字表示连接到其他对象的关联的数量。在以上的 Figure 3-3 中，我们看到有 15 个 button 对象被连接到 Phone 对象。在下面的 Figure 3-4 中，我们将看一种没有数量限制的情况，一个 PhoneBook 被连接到许多的 PhoneNumber 对象。这个星号(*)意味着非常多的数量。在 Java 中通常用一个 Vector、一个 List 或其他容器类型来实现。

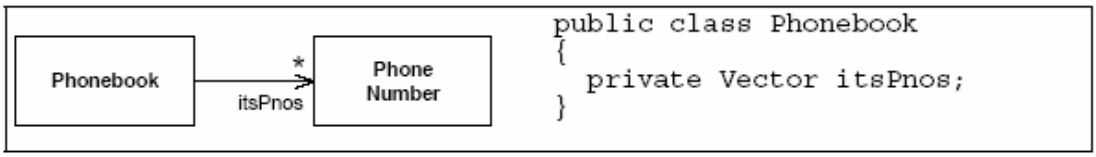


Figure 3-4

我为什么不用 HasA 呢？你也许已经注意到我避开用“ has ”的字眼。我也能够说：“ 一个 Phonebook 拥有许多的 PhoneNumbers ”，但我故意没有这样说。这个 Hasa 和 IsA 的对象对象术语的动词已经导致了许多让人遗憾的误解。我们将在第 6 章中进行探讨。现在，不要指望我用这些通用的术语。我宁可用那些在软件中发生了什么的描述术语。

继承

你不得不在 UML 中小心地使用箭头，在 Figure 3-5 中说明了为什么？正指着 Employee

的小箭头表示了继承(inheritance)。如果你画箭头不小心，可能就很难说清是继承还是关联了，尽量小心一点。我经常用垂直方向的箭头表示继承关系，用水平方向的箭头表示关联。

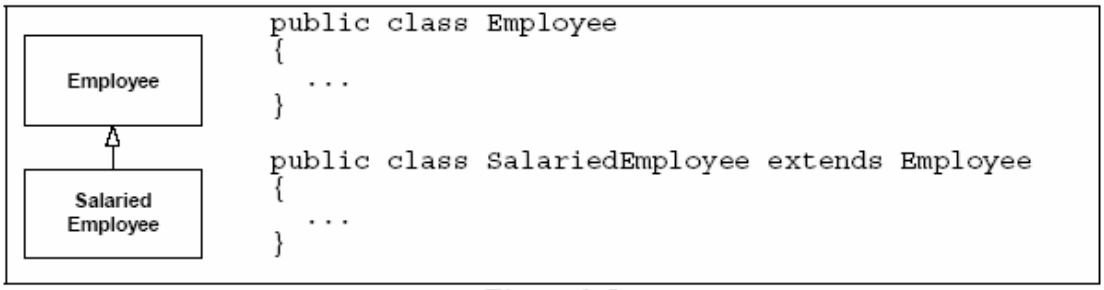


Figure 3-5

在 UML 中所有的箭头指出源代码依存关系的方向。因为是 SalariedEmployee 类引用到 Emplyee 的名称，这个箭头是指在 Employee 上，因此，在 UML 中，继承箭头是指在基类上的(base class)。

UML 有一个特殊的符号，用来说明一个 Java 类和一个 Java 接口之间不同类型的继承关系。如下 Figure 3-6 所示，是一个虚线继承箭头。在图上，你发觉我忘了虚线化那个指向接口的箭头。我建议你在白板上也忘了去虚线化那个箭头吧，画一个虚线化的箭头太耗时间了。

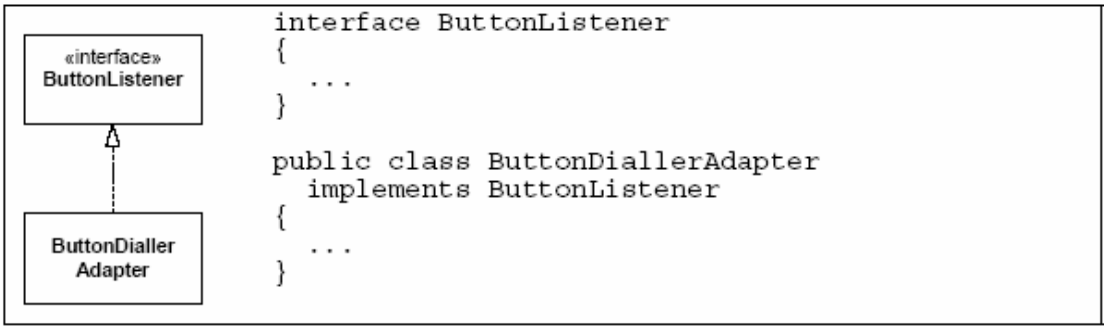


Figure 3-6

Figure 3-7 显示了另外一种传达同样信息的方式，接口能够被画成一个形状像棒棒糖的符号，并连接在实现它的类上。我们经常在 COM 的设计里发现这种类型的标记。

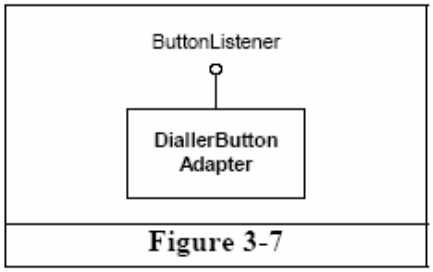


Figure 3-7

一个类图的例子

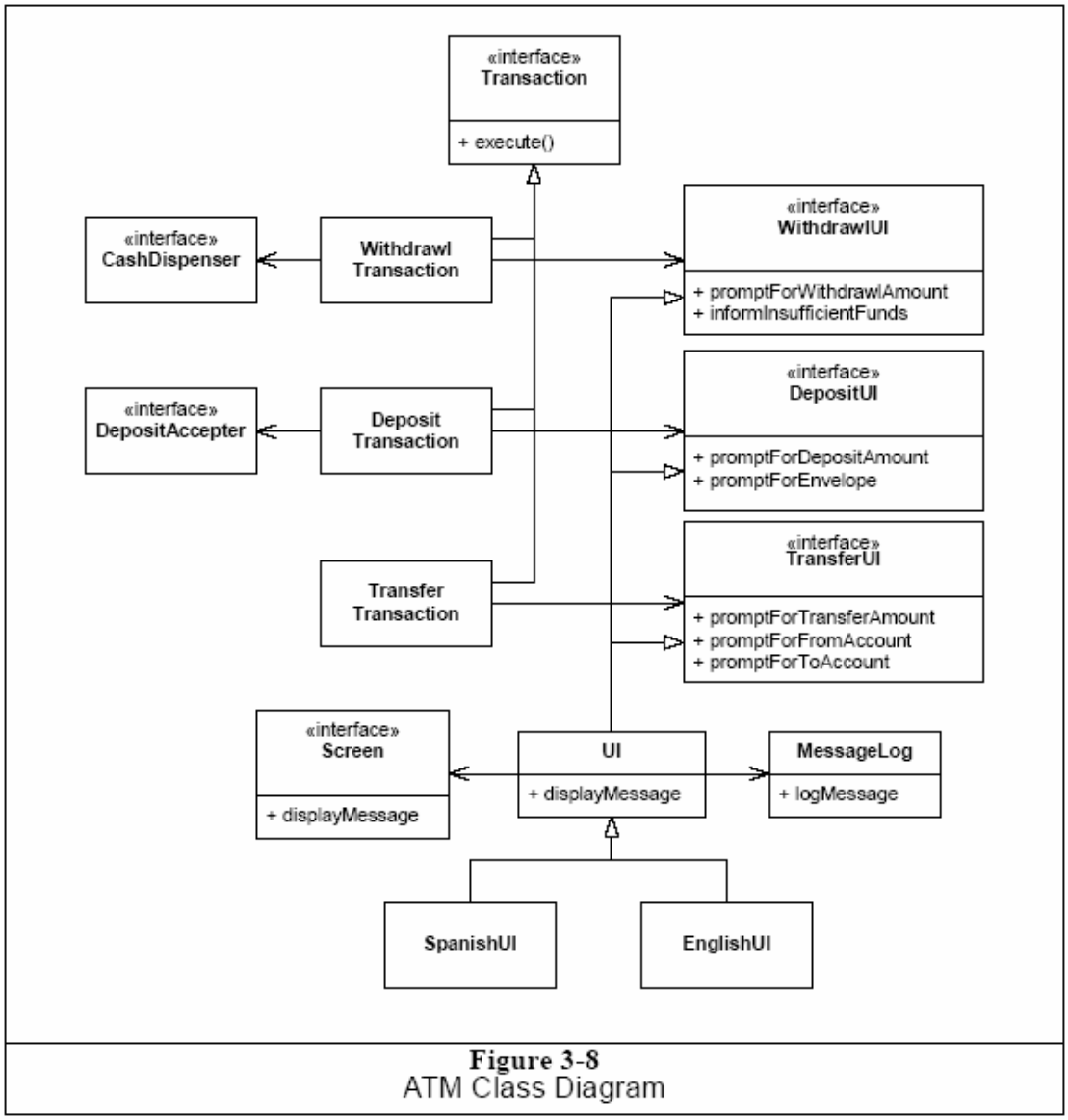


Figure 3-8
ATM Class Diagram

Figure 3-8 展示了一个简单的类图，它是一个 ATM 系统的一部分。这个图同时考虑到了那些已经显示出来的和没有显示出来的东西。注意，标记所有的接口让我很是痛苦，我觉得让我的读者了解到哪些类将会是接口、哪是类是被具体实现的相当重要。举一个例子，这张图能够马上告诉你 WithdrawTransaction 实现了 CashDispenser 接口，显然，系统中的某些类不得不实现 CashDispenser 接口，但是在这张图中，我们不需要关心这些类。

到目前为止，我并没有特别彻底地描述那些不同的 UI 接口的方法。的确，WithDrawlUI 在这里将需要两个以上的方法。PromptForAccount 或 informCashDispenserEmpty 又应该有些什么方法呢？在一个图中同时展现所有的方法将会引起混乱。通过提供一批有代表性的方法，我将给读者一些主意，那些才是真正需要的。

再次注意垂直方向的箭头表示继承关系，用水平方向的箭头表示关联的约定，这将真正帮助你区分那些大量不同类型的关系，没有一种这样的约定将很难从混乱中弄清真相。

注意我是如何将这张图分成三个不同的区域，交易及它们的动态是在左边，那些不同的 UI 接口都在右边，那些 UI 的实现都在底部。那些不同组间的连接是最小化的和整齐的。在某些情况下，三个关联都被用同一种方式指明。在另外一种情况下，三个继承关系都被合并成单独的一条线。这种分组和同一方式的连接将帮助读者更连贯地了解这张图。

当你看到图时，你应该能够了解这些代码。那些实现 UI 的代码（如 Listing 3-1 所示）是不是你期望的那些代码呢？

Listing 3-1
UI.java

```
public class UI implements
  WithdrawlUI, DepositUI, TransferUI
{
  private Screen itsScreen;
  private MessageLog itsMessageLog;

  public void displayMessage(String message)
  {
    itsMessageLog.logMessage(message);
    itsScreen.displayMessage(message);
  }
}
```

细节

有大量的细节和修饰符号能够被加到 UML 类图中去，大多数时候他们是不需要的，但是有时候，他们是很有用的。

类的构造型

类的构造型显示在一对双角括符号“ «» ”之间，经常是在类的名称上面。我们已经见过他们了，在 Figure 3-8 中的«interface»表示一个类的构造型。«interface»是能够被 java 程序员使用的两种标准的构造型之一，另外一个是«utility»。

«interface»，用这个构造型标记的类的所有的方法均是抽象的，任何一个方法都不能被实现。另外，«interface»类能有无实例的变量，仅有静态变量，这精确地对应到 java 接口，见 Figure 3-9。

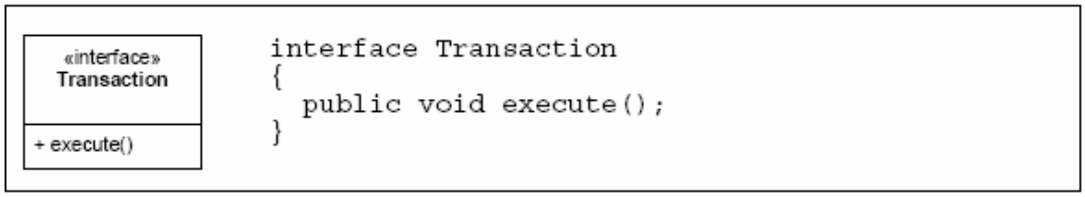


Figure 3-9

«utility»，一个标有«utility»的类的所有的方法和变量都是静态的，Booch 经常叫它们是工具类，如 Figure 3-10 所示：

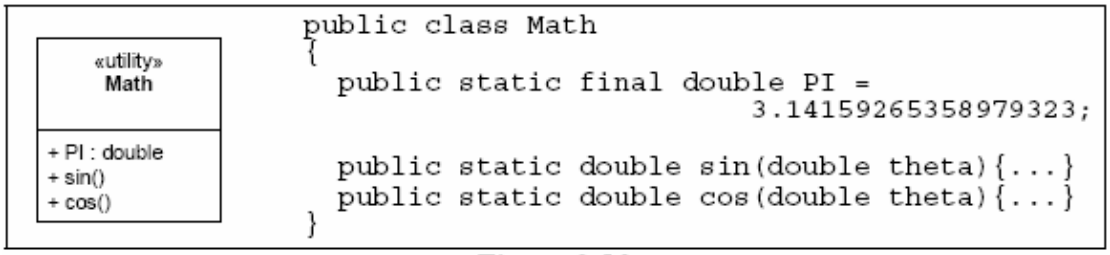


Figure 3-10

你能够创建你喜欢的自己的构造型，我经常用一些如«persistent»、«C-API»、«struct»或 «function»的构造型。你必须得确信那些阅读你的图的人知道那些构造型的涵义。

抽象类

在 UML 中，你有两种方式去表示一个类或一个方法是抽象的。你可以用斜体字或用{abstract}属性，两种选件如 Figure 3-11 所示：

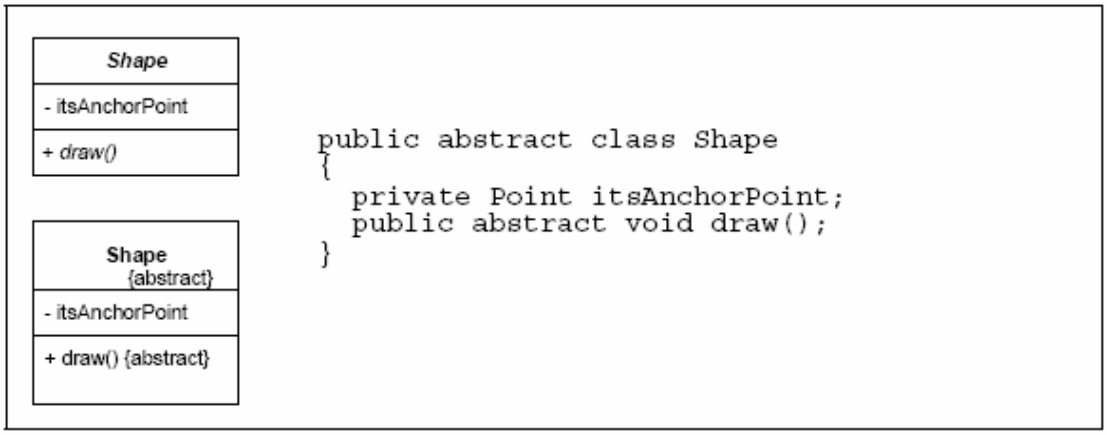


Figure 3-11

在白板上写斜体字有点困难，用这个{abstract}属性有点罗嗦。因此需要在白板去描述一个类或方法是抽象的，我用 Figure 3-12 中的约定方法，这不是一个合法的 UML 画法，但在白板上画起来比较方便。

属性

属性，比如{abstract}能够被添加到任何的类，它们通常不是类的一部分，但用来代表额外的信息。你可以随时创建自己的属性。

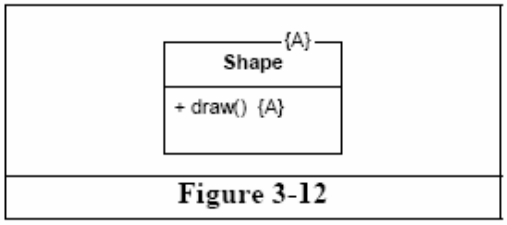


Figure 3-12

属性被写成用逗号分隔由名称、值组对的列表，如下所示：

{author=Martin, date=20020429, file=shape.java, private}

以上例子中的属性并不是 UML 的一部分，这个{abstract}属性只是一个 UML 的定义属性，对于 java 程序是比较有用的。

如果一个属性没有一个对应的值，它假设 boolean 值是真(true)，这个{abstract}和{abstract=true}是一个意思。

属性被写在正确的类名称下文，如 Figure 3-13 所示。

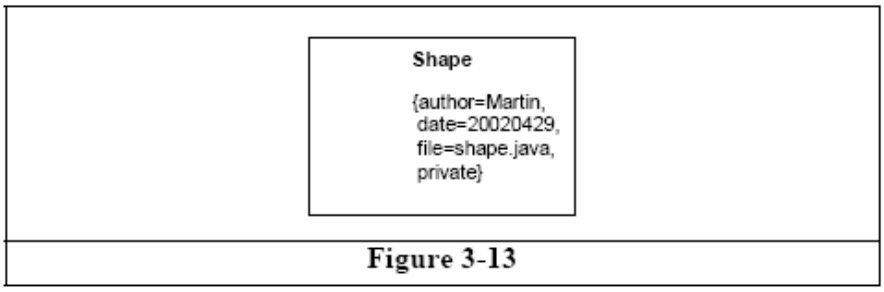


Figure 3-13

除了这个{abstract}属性，我不知道属性还有什么用。在我本人这么多年的画 UML 图的经历中，我从没有机会去为了什么而使用类属性。

聚合

聚合是关联的一种特殊形式，它意味着一种整体/部分（whole/part）的关系。Figure 3-14 展示了它如何画和如何实现。注意，在 Figure 3-14 中展示的实现是没有办法同关联进行区分的，这是一个提示。

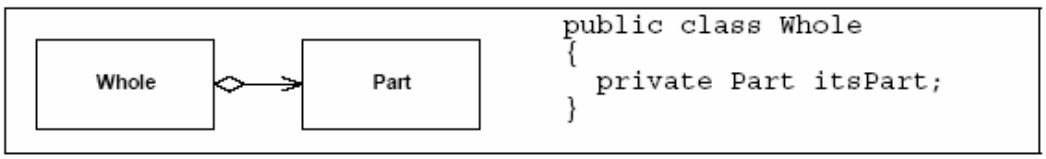


Figure 3-14

不幸的是，UML 并没有为这种关系提供一个强大的定义，由于不同的程序员和系统分析员为这种关系采用了他们自己喜欢的定义方式导致了混淆。由于这个原因，我将不使用任何这种关系，我建议大家也避免这样做。

UML 给我们的关于聚合的硬性规定是相当简单的。一个整体不能是它自己的一部分。因此，实例不能形成聚合回路，一个单独的对象不能够成为它自己的聚合，两个对象不能互相聚合，三个对象不能形成一个聚合环，如 Figure 3-15 所示：

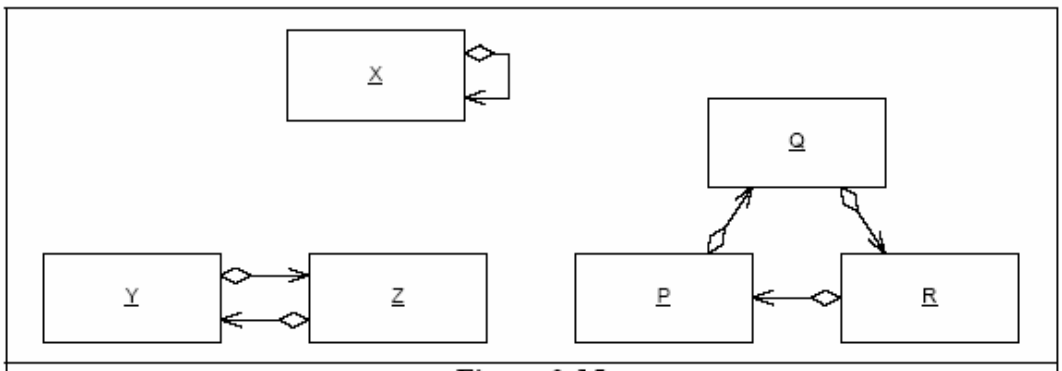


Figure 3-15
Cycles of aggregation between instances.

我并没有发现这是一个特别有用的定义，我是不是经常关心是否一个实例形成了一个有方向的生命周期图呢？不是经常关心它。因此，我发现在我画的各种图中这种关系用得最少。

组合

组合是一种特殊的聚合形式，如 Figure 3-16 所示。再次声明，它的 UML 实现还是无法从关联中区分出来。不过，这一次不是定义缺乏的原因，这次是无论是 Java 程序还是 C++程序中都很少用到这种关系，换句话说，它用得极少。

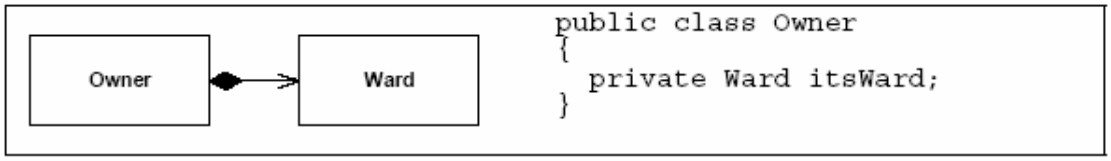


Figure 3-16

应用于聚合的规则可以同样适用于组成。这里没有实例生命周期。一个系主(owner)不能是自己的被组合的对象(ward)，不过，UML 提供了更多的定义。

- 一个被组合的对象的实例不能同时属于两个系主。在 Figure 3-17 中的对象是非法的。不过，它对应的类图并不是非法的。所有的系主均可以将一个被组合的对象的拥有关系传送到另外一个系主。
- 系主负责被组合的对象的生命周期的管理。如果系主被销毁，被组合的对象也必须

跟着一起被销毁，如果系主被复制，被组合的对象也必须跟着一起被复制。

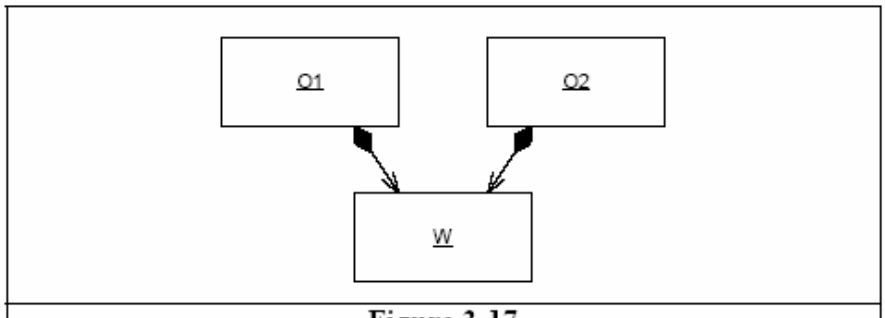


Figure 3-17
Illegal composition.

Java 的销毁由垃圾回收工具来幕后处理，因此很少需要去管理一个对象的生命周期。深度的复制不用去关心，在一个图上去描述深度复制的语义是少见的，尽管我在这里已经描述了这种 java 程序的组合关系，但是很少用到。

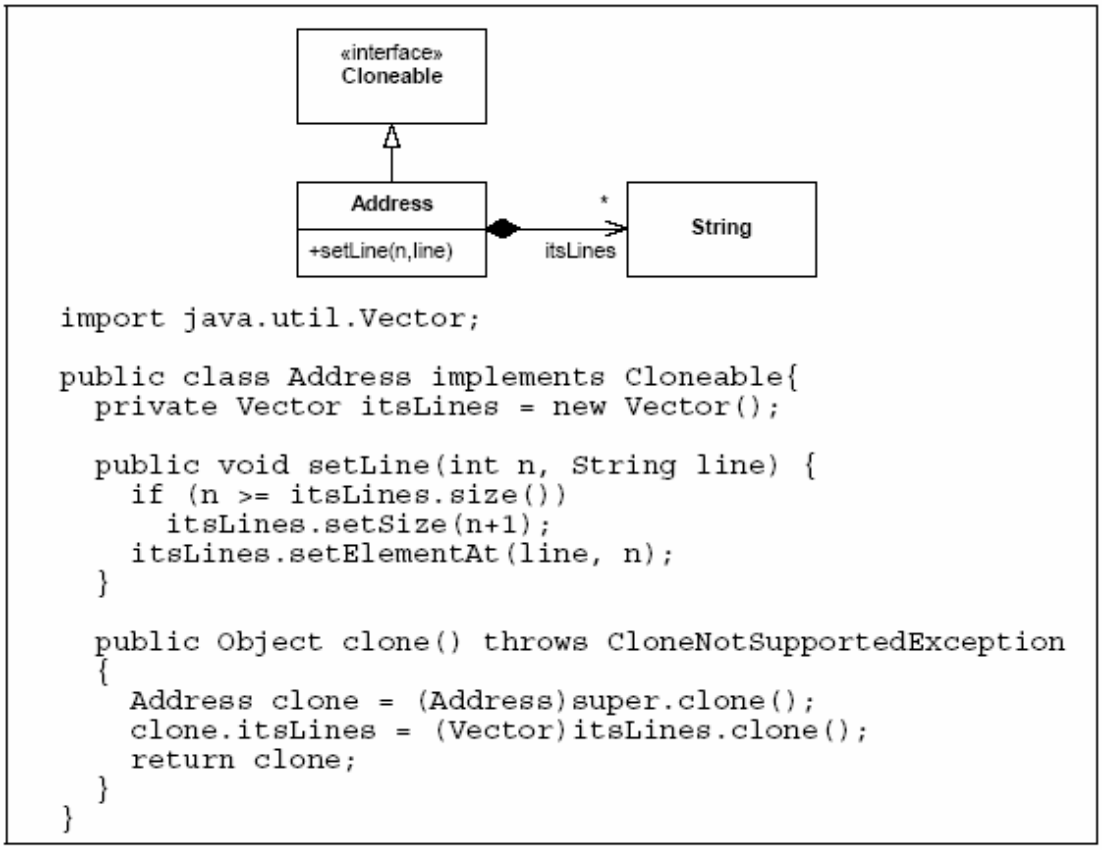
Figure 3-18 显示了组合如何用于描述深度复制。我们有一个名 Address 的类，它持有着许多的 Address 对象。每个 String 是一行地址。明显地，当你要复制这个 Address 的时候，你简单的复制将改变原先的依存关系。因此，我们需要一个深度复制，这种 Address 和 String 之间的组成关系也需要被深度复制。

多重性

对象能够持有其他对象的数组或向量，或者说它们能够持有许多同一类型的、不同实例变量的对象。在 UML 中，通过放置一个多重性(multipicity)表达式在关联线末端来表示。多重性表达性可以是一个数字、一段范围或者是它们的组合。举例如 Figure 3-19 表示了一个 BinaryTreeNode 使用了 2 的多重性表达。

这里有一些允许的形式：

- 数字　　精确的数量
- *或 0..*　0 到无数
- 0..1　　0 个或 1 个，在 Java 中经常用一个空的引用来实现
- 1..*　　1 个到无数个
- 3..5　　3 个到 5 个
- 0,2..5,9..*　　非法的



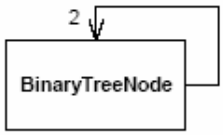

```
import java.util.Vector;

public class Address implements Cloneable{
  private Vector itsLines = new Vector();

  public void setLine(int n, String line) {
    if (n >= itsLines.size())
      itsLines.setSize(n+1);
    itsLines.setElementAt(line, n);
  }

  public Object clone() throws CloneNotSupportedException
  {
    Address clone = (Address)super.clone();
    clone.itsLines = (Vector)itsLines.clone();
    return clone;
  }
}
```

Figure 3-18
DeepCopy is implied by Composition




```
public class BinaryTreeNode
{
  private BinaryTreeNode leftNode;
  private BinaryTreeNode rightNode;
}
```

Figure 3-19
Simple multiplicity

关联构造型

关联能够通过带标记的构造型来改变它的意义。Figure 3-20 显示了我经常用到的一些，所有这些除了最后一个都是标准 UML 标记。

«creates»的构造型指明关联的目标对象是被源对象创建的。意思是说源对象创建了目标对象，然后将它传送给系统的其他对象。在这个例子中我显示一个典型的工厂(factory)模式。

«local»构造型用于源类创建了一个目标对象的实例，把它当做一个本地变量。它暗示被创建的实例不能在创建它的成员函数之外存活，因此，它不被任何实例变量持有，也以任何一种方式传递给系统的其他部分。

«parameter»构造型显示了源类获取了目的实例的访问入口，尽管是通过它的成员函数的参数的方式进行的。另外，这暗示当成员函数返回时，源对象便忘了这个目标对象，这个目标对象没有在实例变量中被保存。

«delegates»构造型不是一个标准的 UML 组成部分，它是我创建的。不过我经常用它，因此我想它最好包括在这里。当源类传递目标对象的一个成员函数调用时，能够用到它。有许多的设计模式用到了这种技术，如 PROXY、 DECORATOR 和 COMPOSITE7。因为我经常用它们，我觉得这个标记非常有用。

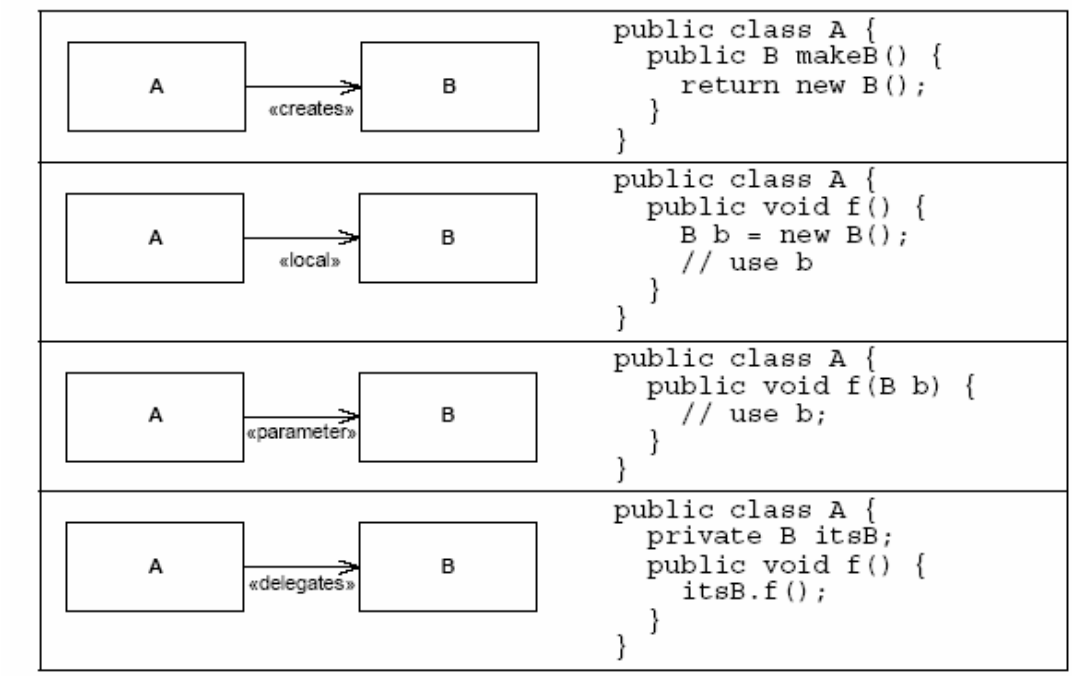


Figure 3-20

内部类

内部类在 UML 中用一个带十字圆圈的关联标记来表示。

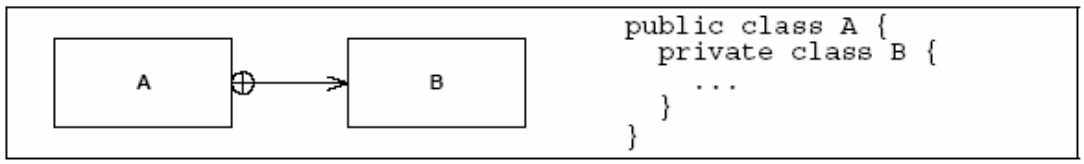


Figure 3-21

匿名内部类

Java 最有趣的特征之一是匿名内部类。当 UML 在这里还没有提供官方的支持，我发现用如 Figure 3-22 的标记能够比较好地为我工作，这种标记简洁而描述完整。匿名内部类用有一个«anonymous»构造型的嵌入类来展示，被实现的接口的名称也被给出。

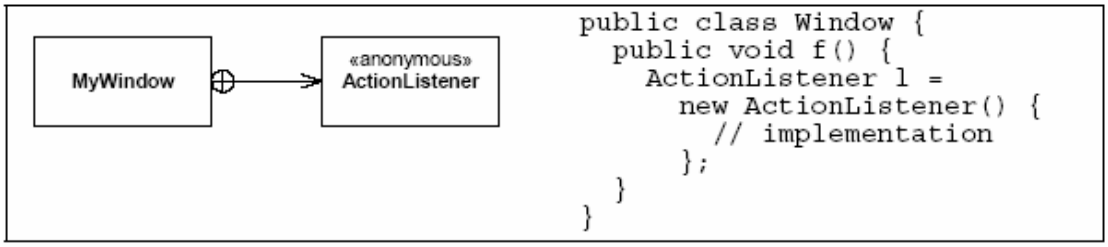


Figure 3-22

关联类

多重性关联已经告诉我们源对象能够连接到多少目标对象的实例，但是这个图并没有告诉我们，哪种类型的容器将会用到，能够用如 Figure 3-23 的关联类来描述。

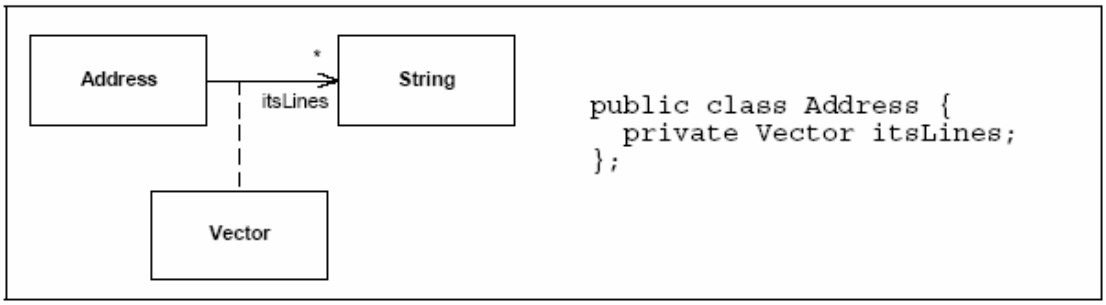


Figure 3-23
Association class.

关联类展示了一个特殊的关联如何被实现。在这张图上，它用有一个虚线连接着的关联的一个普通类来表现。对 Java 程序员而言，它能够解释这意味着源类实际上包含着一个关联类的引用，它依次包含着目标对象的引用。

关联类能够用来指明特定形式的引用，如不固定的（weak）、松散的（soft）或幻影（phantom）引用，见 Figure 3-24。

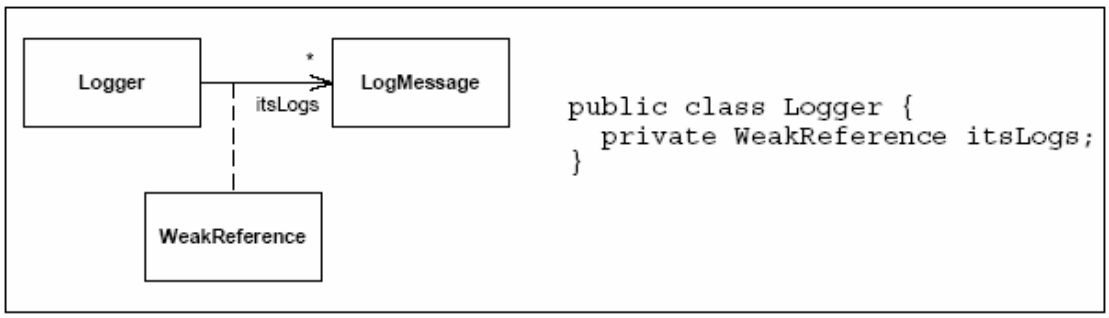


Figure 3-24

另外一方面，这种标记有点烦人，如果用如 Figure 2-25 的构造型也许更好。

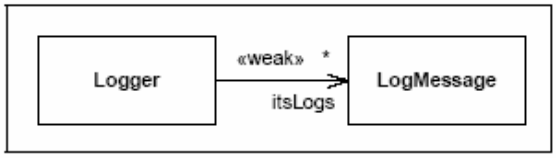


Figure 3-25

关联限定符

当关联通过某种类型的关键字或标记被实现的时候，经常用到关联限定符来替代一个普通的 Java 引用。在 Figure 3-26 中的例子显示了一个 LoginServlet 关联到一个 Employee 类。这个关联用是一个叫 empid 的变量来进行中介的，它包含了一个针对 Employee 的数据库关键字。

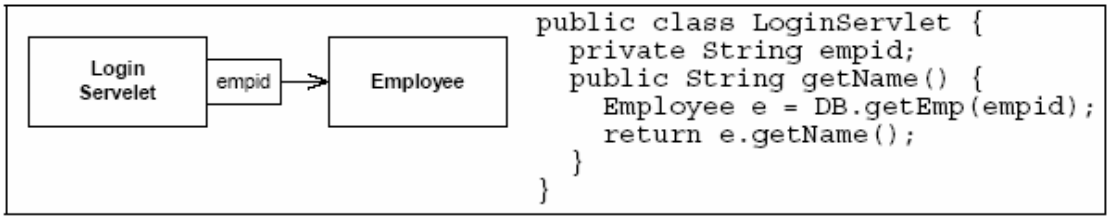


Figure 3-26

我发现这个标记使用的场合比较少，有时用来表现一个通过数据库或目录关键字来关联到其他对象的时候还是有用的，不过，那些读到这些图的人知道限定符如何用于访问到真实的对象是比较重要的，~~显然这没有什么问题~~。

小结

在 UML 中有大量的修饰、标记和其他一些玩意儿。如此多的东西，你需要花大量的时间去了解 UML 语言的规则，写出的文档任何一个人都能够理解。在本章中，我避开了有关 UML 的一些有关技巧和奥妙的特征，相反我向你展示了我用到的一些部分。我希望你我已经给你灌输的这些有价值的原则的知识。少用 UML 远比多用 UML 好。

参考文献

[Booch94]: Object Oriented Analysis and Design with Applications, Grady Booch, Benjamin Cummings, 1994

[GOF94]: Design Patterns, Gamma, Helm, Vlissides, Johnson, Addison Wesley, 1994

第四章　序列(Sequence)图

序列图是一种被使用 UML 的人使用的最常用的动态建模画法，就像你期望的那样，UML 提供了一大堆有趣的东西帮助你画出那种很难理解的图。在这一章，我们将学习这些有趣的东西，并试图说服你节制地使用它们。

我曾经给一个团队做过咨询，他们已经决定为每一个类的每一个方法建立序列图。不！千万不要那样做！这样非常浪费你的时间。只有当你迫切地需要给某个人描述一组对象的如何协作的情况的时候，或者当你要为你自己可视化协作情况的时候，你再去用序列图吧。仅把它们当做你偶尔用来磨练你自己的分析技术的工具吧，胜于把它作为你需要的文档。

基础

我第一次学习画序列图是在 1978 年，James Grenning，我一个长期的朋友和合作伙伴，当我们为一个有关通过调制解调器连接的计算机之间的复杂的通讯协议的项目工作时，他给我展示了如何画序列图。我要教授给你们的只比他教授给我的复杂那么一点，这将满足你大部分画序列图的需要了。

对象、生命线、消息

Figure 4-1 展示了一个典型的序列图，有关协作的对象被放置在图的顶部，这个人样图在左边表示一个匿名对象（参与者），它是所有进入和离开这个协作的消息的源头和汇集点。不是所有的序列图都有这样的匿名参与者，但大部分的图都有。

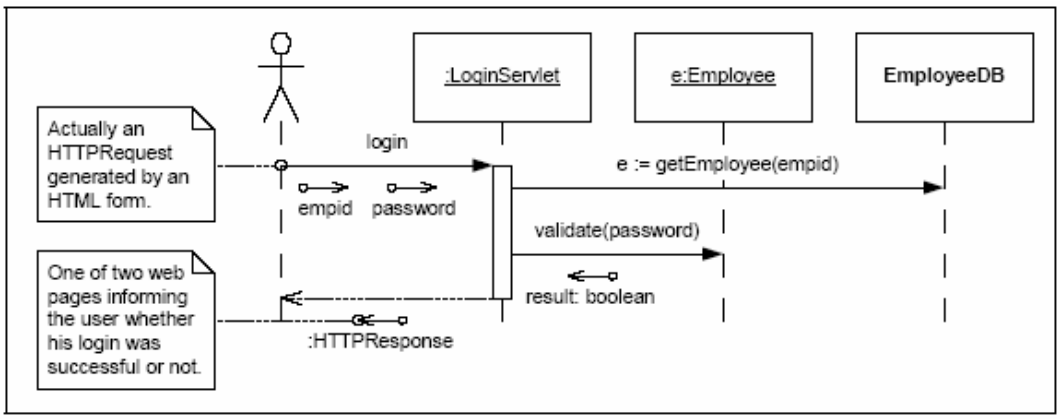


Figure 4-1
Typical Sequence Diagram

这条垂立在对象或参与者下面的虚线叫做生命线(lifelines)。一个从一个对象被发送到另

外一个对象的消息被画成一个两条生命线之间的箭头。每个消息都被标记出名称。参数被画在消息名称后面的括号里或是一个数据标记(data tokens，一个以小圈结束的箭头)之后。时间(Time)是在垂直方向，因此越是下面的消息是越晚被发送。

在 LoginServlet 对象生命线上的小框框叫做活动(Activation)，活动是可选的，大多数图都不需要它们。它们表示函数的执行时间。在这个例子中表示 login 函数执行时间长短。离开这个活动往右的两个消息被 login 方法发送出。那个没有标记的箭头表示 login 函数返回给参与者，传回一个返回值。

注意在 getEmployee 消息中的返回变量 e，它表示 getEmployee 的返回值。注意这个 Employee 对象也叫做 e，你已经猜到了，它们是同一个东西！getEmployee 的返回值是一个 Employee 对象的引用。

最后，注意那个 EmployeeDB 是一个类，而不是一个对象。这仅能说明 getEmployee 是一个静态方法，因此，我们期望的 EmployeeDB 将如 Listing 4-1 所示：

Listing 4-1
EmployeeDB.java

```
public class EmployeeDB
{
  public static Employee getEmployee(String empid)
  {
    ...
  }
  ...
}
```

创建和销毁

我们能够用 Figure 4-2 中约定的画法在一个序列图中展示一个对象的创建。一个没有标记的消息终止在一个被创建的对象上，而不是它的生命线上。我们期望 ShapeFactory 如 Listing 4-2 所示那样被实现。

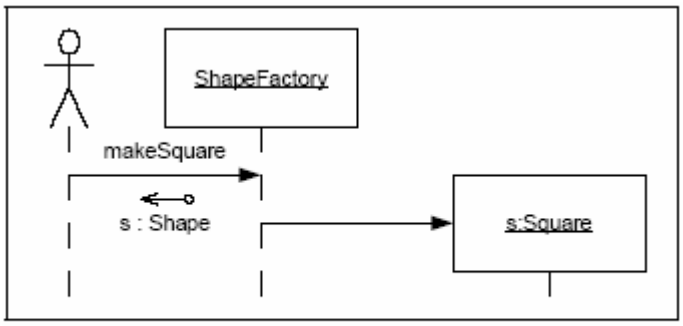


Figure 4-2
Creating an object

Listing 4-2
ShapeFactory.java

```
public class ShapeFactory
{
  public Shape makeSquare()
  {
    return new Square();
  }
}
```

在 Java 中我们不需要明确地销毁对象，垃圾回收器将为我们做所有明确地销毁。不过，有时我们也要清楚垃圾回收器已有的关于一个对象为我们做的或被关注的。

Figure 4-3 显示了我们如何在 UML 去标记它，一个对象的生命线被声明终止在一个在“X”上。消息箭头终止在这个“X”表示声明一个对象被垃圾回收器处理。

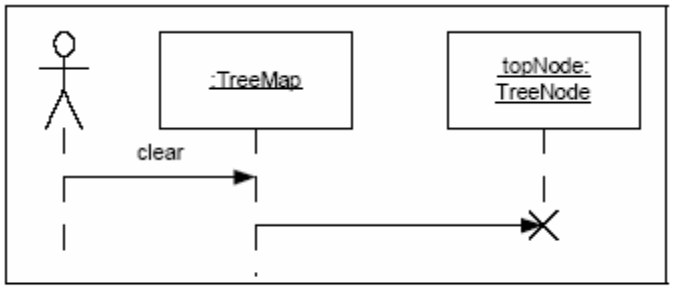


Figure 4-3
Releasing an object to the garbage collector

Listing 4-3 说明了我们从上图期望的实现，注意那个 clear 方法将 topNode 变量设置为空(nil)。因为这个 TreeMap 仅是一个持有 TreeNode 实例的一个引用的对象，它被声明给垃圾回收器去处理。

Listing 4-3
TreeMap.java

```
public class TreeMap
{
  private TreeNode topNode;
  public void clear()
  {
    topNode = nil;
  }
}
```

简单的循环

你可以通过在一个重复的消息周围画一个方框的方式在一个 UML 图中表示一个简单的循环。这个循环条件能放置在方框中的任何地方，经常被放置在左下角，见 Figure 4-4。

这是一个非常有用的标记约定，不过，在序列图中用它去捕捉算法是不明智的。序列图应当用于展现对象间的连接而不是那些七七八八的算法细节。

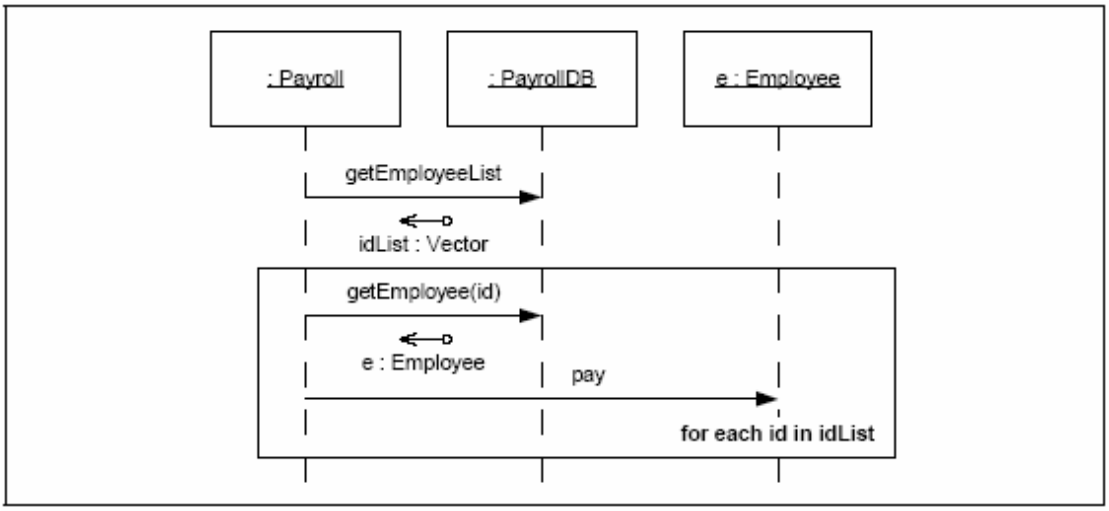


Figure 4-4
A simple loop

案例和场景

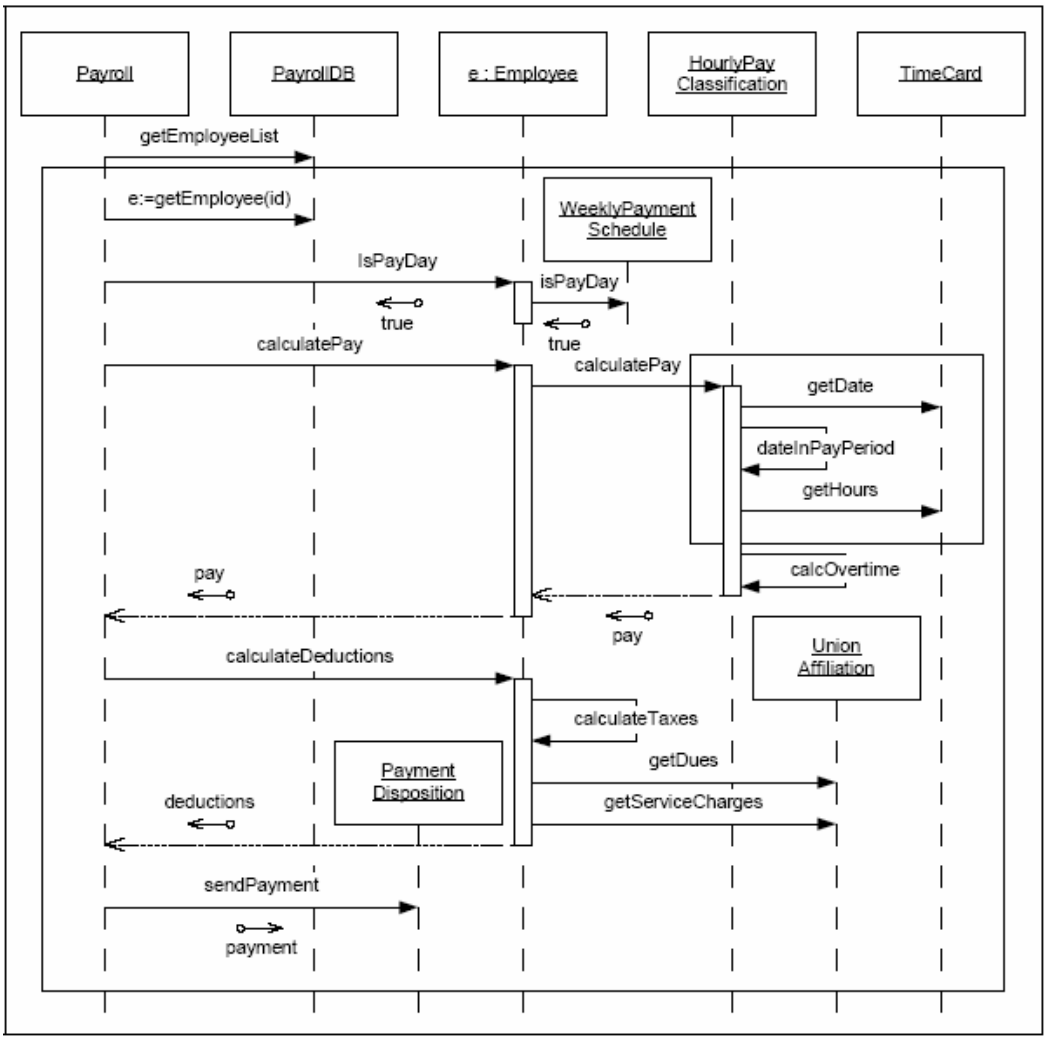


Figure 4-5
This sequence diagram is too complex

规则：不要画像 Figure 4-5 那种充斥着大量对象和消息返回的序列图，没有人能够读它们，没有人去读它们，那是极大的时间浪费。相反，去学习如何画一个更小的、能够抓住了你试图做的东西的本质的序列图。每张序列图应该只有一张纸大，留下大量的地方去用文本解释清楚。不要为了把它们放在一张纸上而想办法缩小图标。

另外，不要画成打或上百个序列图，如果你画得太多，它们将没有人去看。找出所有相关场景的共性，并将焦点放在这里。对于 UML 图来说，共性远比差异重要。用你的图去揭示共有的主题和共有的惯例。不要用它们去描述每个小细节。如果你真的需要画一个序列图去描述消息流，节制而简洁地使用它们，尽可能地少用它们。

首先，问你自己这个序列图是否真的需要？代码本身提供的信息更多、也更划算。如 Listing 4-4 中展示了这个 Payroll 类可能的情况，这个代码是完整地表达所有信息，我们并不需要一个序列图去理解它。因此可能根本不需要去画序列图，也许这个代码能很好的、足够地说明它自己。有代码足够说明它自己时，图是多余的、浪费的。

Listing 4-4
Payroll.Java

```
public class Payroll
{
  private PayrollDB itsPayrollDB;
  private PayrollDisposition itsDisposition;
  public void doPayroll()
  {
    List employeeList = itsPayrollDB.getEmployeeList();
    for (Iterator iterator = employeeList.iterator();
         iterator.hasNext();)
    {
      String id = (String) iterator.next();
      Employee e = itsPayrollDB.getEmployee(id);
      if (e.isPayDay())
      {
        double pay = e.calculatePay();
        double deductions = e.calculateDeductions();
        itsDisposition.sendPayment(pay - deductions);
      }
    }
  }
}
```

代码真是能够用于描述系统的一部分吗？实际上，这应当是每个设计者和开发者的目标。团队应该努力去创建表述清楚和可读性强的代码，使更多的代码能够描述自己，更少地需要图，最好整个项目不需要图。

第二，如果你感觉序列图真的需要，问你自己是否有办法把它们分割成许多一小组一小组的场景。举一例子，我们能够把 Figure 4-5 中的大序列图分散成能被更好读的更小的序列图。想想在 Figure 4-6 中的小序列图是多么容易理解了。

第三，想想你试图去描绘的，是不是像 Figure 4-6 那样展现了如何计算一个按时付费

的低层操作细节？或者是你准备展现一个如图 Figure 4-7 所示的系统流程的高层视图？通常，高层图比低层图更有用，它们将帮助读者更好地理解系统，它们揭示的共性比差异多。

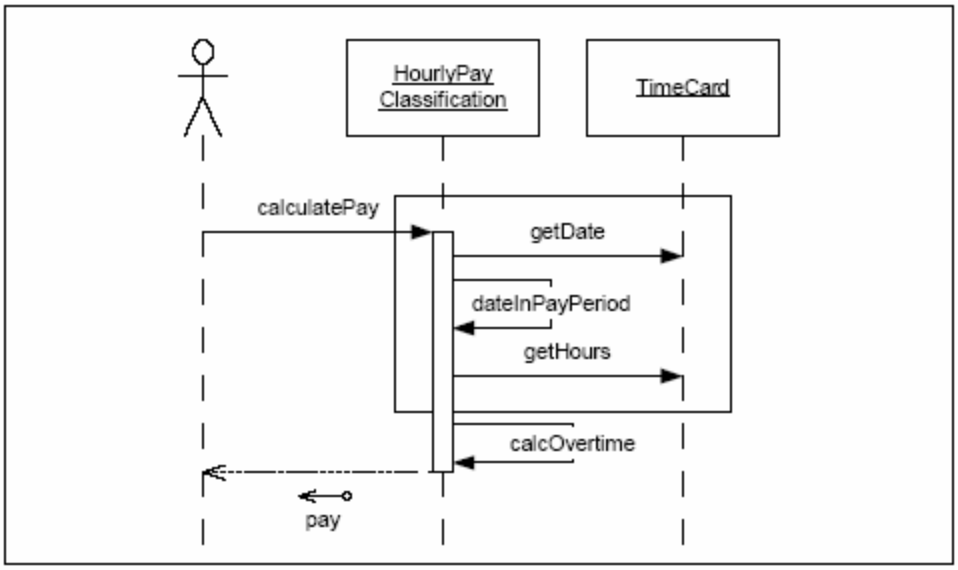


Figure 4-6
One small scenario

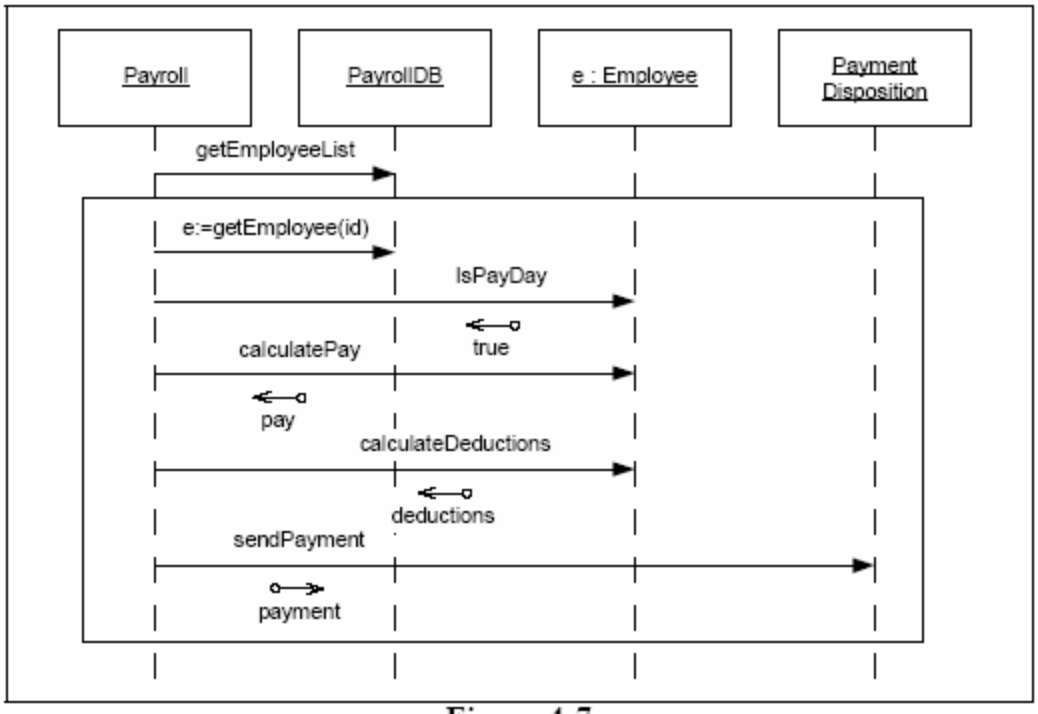


Figure 4-7
A high level view.

高级概念

循环和条件

用一张序列图去完整地详细说明一个算法是可能的。在 Figure 4-8 中，你可以看到一

个用较好的说明的的循环(loop)和 if 语句完成的薪水册的算法。

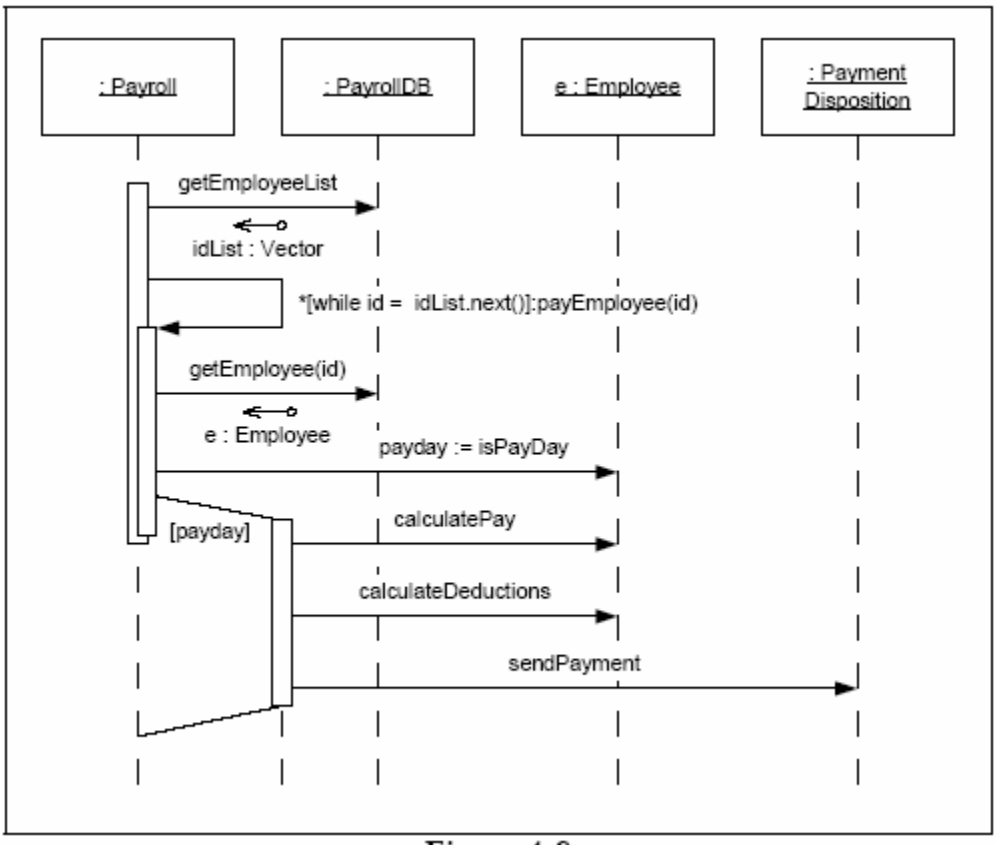


Figure 4-8

这个 payEmployee 消息使用了一个循环表达式的前缀，如下所示：

*[while id := idList.next()]

这个星号告诉我们这是一个迭代，这个消息将会重复发出直到方括号中的监视(guard)表达式的值为 false 为止。UML 没有为这个监视表达式指定一个语法规则，因此，我使用了一个 java 伪代码去提出使用一个迭代。

这个 payEmployee 消息结束在一个它相关的活动，但是有点偏差，首先，这标记了两个函数正执行在同一个对象上，因为 payEmploy 消息是重复的，第二个活动也将是重复的，所有来源于此的消息都是整个循环的一部分。

注意那个[payday]监护表达式旁边的活动，它标记了一个 if 语句。第二个活动只有当这个监视条件成立的情况下才能得到控制。因此，如果 ifPayDay 返回 true，这个 CalculatePay,CalculateDedutions 和 sendPayment 将会被执行，否则不会执行。

实际上，虽然在一个序列图中去捕获的所有的算法细节是可能的，但你不应该用这种方式去分析所有的算法。用 UML 来描述的算法最多只能算是清楚(clunky?)，在 Listing 4-4 中代码是一个更有效的表达算法的方式。

花时间的消息

通常我们不用考虑对象间的消息发送的时间，大多数 OO 语言这个时间是极短暂的。那就是为什么我们画消息的时候是水平的——他们不费时间。不过，在某些情况下，消息要费时间。我们想要穿过网络发送消息或一个系统中，控制线程被分成消息发送和消息接收。这是可能的，我们能够用如 Figure 4-9 所示的一个带角度的线来标记。

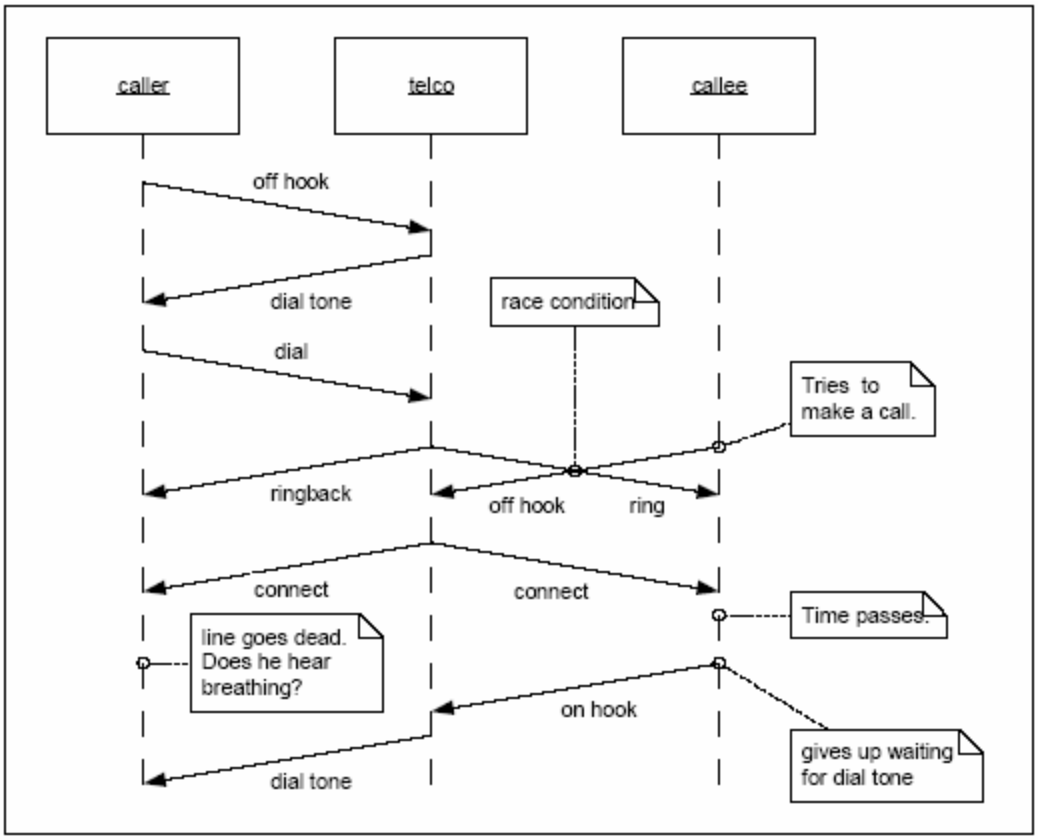


Figure 4-10
Failed Phone Call.

这张图说明了一个正在拨打的电话，在这个序列图中有三个对象，caller 表示主叫——呼叫发起人，callee 被表示被叫——呼叫接收人，telco 表示电话公司。

当主叫提起话筒时，电话机向电话公司发送一个摘机(off hook)消息，电话公司用准备拨号音（长长嘟声）响应，主叫在拨打被号码时会收到拨号音（received dial tone），电信公司发给被叫的一个振铃音(ringing)，并回给主叫一个回铃音(playing ringback)，被叫听到铃声后摘机，电信公司建立连接，当被叫说“喂”的时候，电话呼叫已经成功了。

不过，这里有其他可能的情况用来演示那些花时间的消息的图。仔细察看 Figure 4-10，注意开始是一样的，不过，当被叫的振铃响起之前，他拿起话筒准备自己打一个电话，这个主叫正被连接到了被叫，但是两个人都不知道，这个主叫在等待被叫的“喂”声，这个被叫正在等待振铃音，被叫最终失望地挂了机，主叫听到了拨号音。

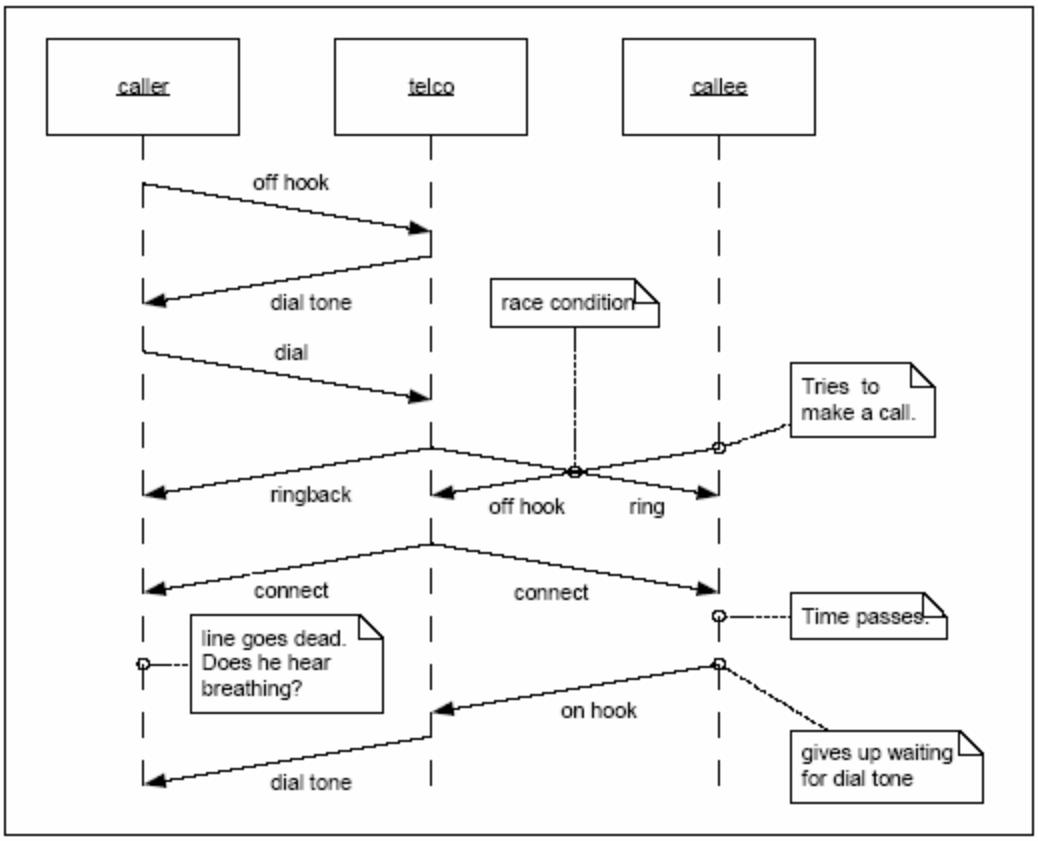


Figure 4-10
Failed Phone Call.

在 Figure 4-10 中两个箭头的交叉表示竞争条件(race condition)，竞争条件发生在两个异步实体同时调用相反的操作。在我们的例子里，电信公司发出振铃音，被叫摘机，在这时，系统的参与者有一个不同的关于系统状态的想法。主叫在想待被叫的“ 喂 ”声，电信公司在想它的工作已经做了，被叫在想等候一个拨号音。

在软件系统中，竞争条件相当被难以发现和排除的，用这种花时间的消息的图能够有效地查找和诊断它们，一旦发现它们，能够使我们向其他人进行解释。

异步消息

通常，当你发送一个消息给一个对象时，你不希望等到接收消息的对象完全执行完成后才获回控制。以这种方式传递的消息为**同步消息**。不过，在一个分布式和多线程的系统中，将消息发送后立即获回控制权，而接收消息的对象在另外一个控制线程中执行的情况是可能的。这种消息称为**异步消息(asychronous message)**。

Figure 4-11 显示一个异步消息，注意是用空心的箭头代替了实心箭头。回头看看本章所有其他的序列图，这都是被画成实心的同步消息。稍微不同的箭头有完全不同的行为表现是 UML 的精妙之处。

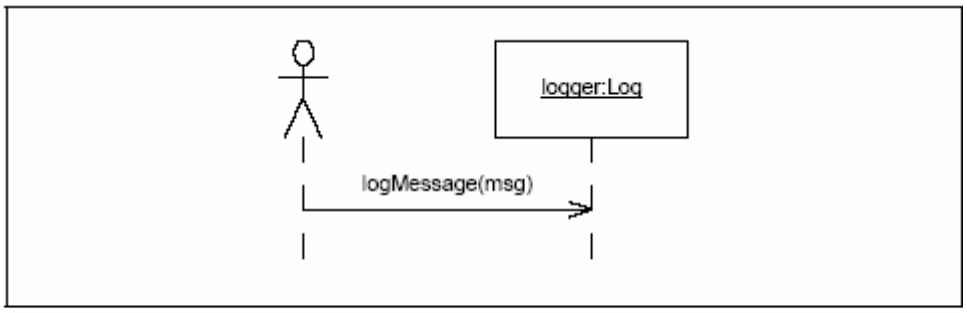


Figure 4-11
Asynchronous Message

Listing 4-5 和 listing 4-6 说明了对应于 Figure 4-11 的源代码。Listing 4-5 表示一个有关 Listing 4-6 中的 Log 类的一个单元测试。注意到 logMessage 函数在排队了一个消息后立即返回。注意，这个消息实际上被一个由构造器开始的完全不同的线程处理了。这个 LogText 类来确认 logMessage 方法是异步运行，首先检查这个消息被记录但没有执行，然后让这个处理器到另外一个线程，最后检验这个消息被处理了并从队列中移除了。

Listing 4-5
TestLog.java

```
import junit.framework.TestCase;
import junit.swingui.TestRunner;

public class TestLog extends TestCase {

  public static void main(String[] args) {
    TestRunner.main(new String[]{"TestLog"});
  }

  public TestLog(String name) {
    super(name);
  }

  public void testSend() throws Exception {
    Log l = new Log(System.out);
    l.logMessage("the message");
    assertEquals(1, l.messages());
    assertEquals(0, l.logged());
    Thread.yield();
    assertEquals(1, l.logged());
    assertEquals(0, l.messages());
    l.stop();
  }
}
```

Listing 4-6
Log.java

```
import java.util.Vector;
import java.io.PrintStream;

public class Log {
  private Vector messages = new Vector();
  private Thread t;
  private boolean running = false;
  private int logged = 0;
  PrintStream logStream;

  public Log(PrintStream stream) {
    logStream = stream;
    running = true;
    t = new Thread(
      new Runnable() {
        public void run() {
          while (running) {
            if (messages() > 0) {
              String msg;
              synchronized (messages) {
                msg = (String) messages.remove(0);
              }
              logStream.println(msg);
              logged++;
            }
          }
        }
      }
    );
    t.start();
  }

  public void logMessage(String msg) {
    synchronized (messages) {
      messages.add(msg);
    }
  }

  public int messages() {
    return messages.size();
  }

  public int logged() {
    return logged;
  }

  public void stop() throws InterruptedException {
    running = false;
    t.join();
  }
}
```

这是一种异步消息的可能实现，其他的实现也是可能的。实际上，如果调用者希望在被期望的操作被执行前就返回，我们能够这样标记一个消息是异步的。

多线程

异步消息暗示着多线程控制。我们在 UML 图中通过带一个线程标识的消息名称标记，来显示几个不同的线程控制，如 Figure 4-12。

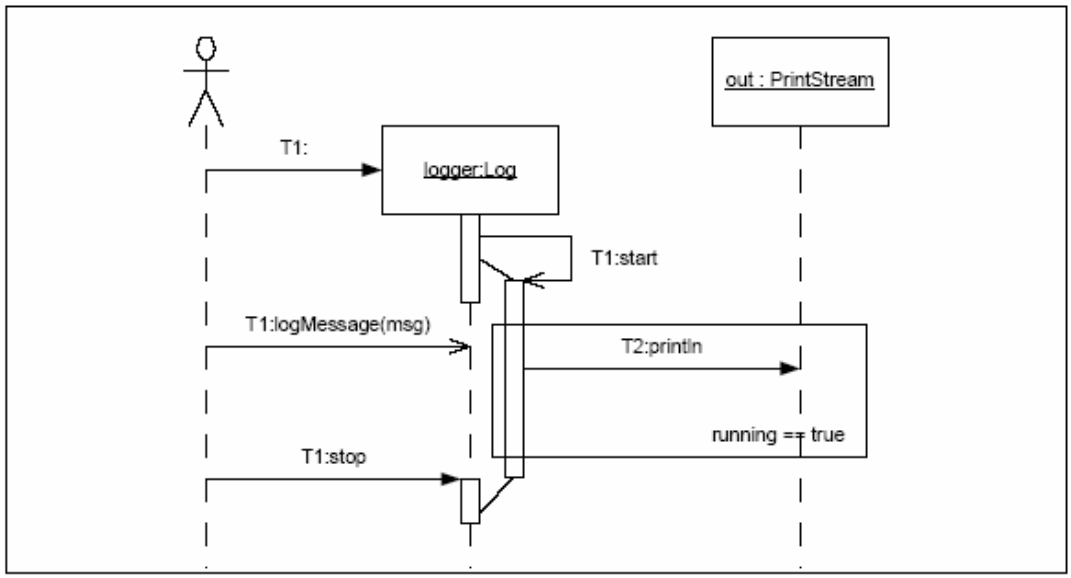


Figure 4-12
Multiple Threads of Control

注意，消息名称前置一个像 T1 的标识名称，紧跟在冒号后后面。这个标识命名了发送消息的线程。在这个图中，这个 Log 对象被线程 T1 创建并操纵。在 Log 对象中实现消息记录、运行的线程命名为 T2。

就像你看到的一样，这个线程名称并不需要对应到代码中去。上面的 Listing 4-6 并没有命名线程 T2。然而，这线程标识对于图来说是有好处的。

活动对象

有些时候，我们需要标记一个有完全不同内部线程的对象，这样的对象叫做活动对象(active objects)，他们用一个粗体框来表示，如 Figure 4-13:

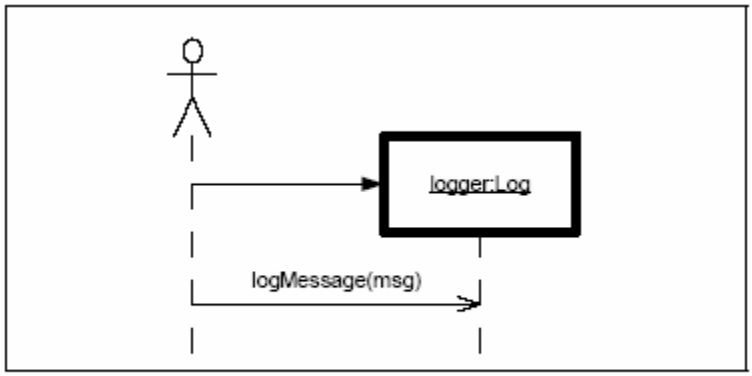


Figure 4-13
Active Object

活动对象简化了那些实例化和控制它们自己的对象，它们的方法没有限制，它们的方法可以运行在对象的线程内，或者是运行在调用者的线程。

发送消息给接口

我们的 Log 类是一种记录消息的方式。如果我们需要我们的程序能够使用不同类型的记录器，我们可能建立一个声明了 logMessage 方法的接口，然后派生出我们自己的 Log 类，所有的实现都来自于这个接口，如 Figure 4-14。

这个应用程序打算发送消息给 Logger 接口，我们不知道这个对象一个 AsynchronousLogger，我们如何在一张序列图中描绘它呢？

在 Figure 4-15 中的图是一种比较明显方法。我们为这个接口命名对象，然后处理它。这个方法看起来打破了规则，因为一个拥有一个接口实例是不可能的。不过，所有我们在这里说的是这个 logger 对象符合这个 Logger 类型，我们不能够说有什么导致了接口的实例化。

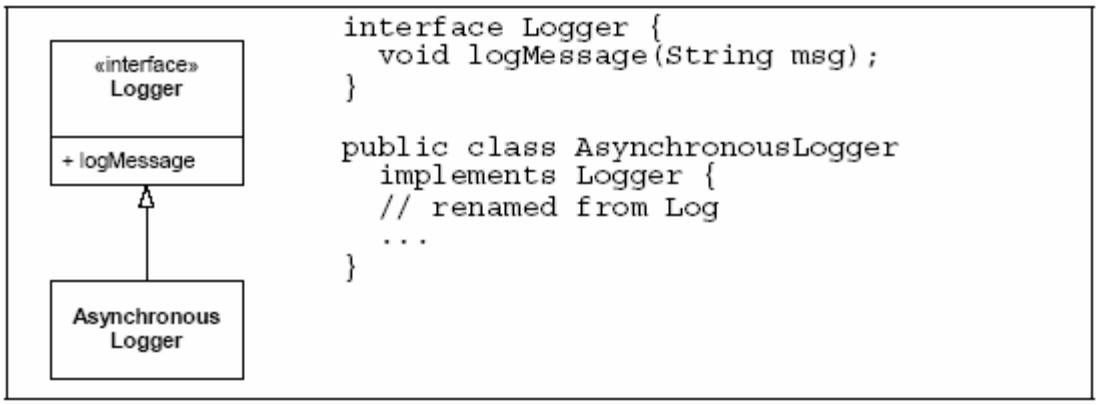


Figure 4-14

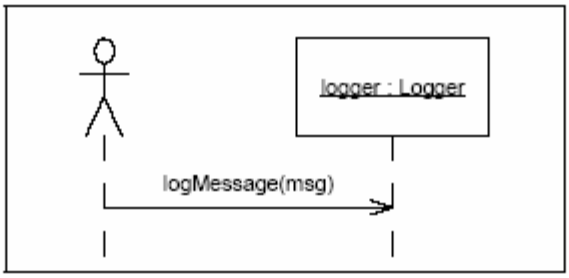


Figure 4-15
Sending to an interface

有时，虽然知道这个对象的类型，但是仍然需要表示消息被发送到一个接口。例如，我们知道我们已经建立了一个 AsychronousLogger 类，但是我们仍然要说明这个应用仅使用这 Logger 接口，Figure 4-16 说明了如何描绘的，我们将棒棒糖样的接口画在一个对象的生命线上。

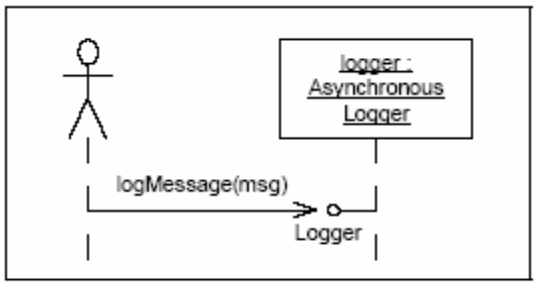


Figure 4-16

小结

就像我们已经看到的一样，在一个面向对象应用程序中，序列图是一种强大的表达消息流的形式，我们也指出它们易被乱用和易被过度使用的事实。

在白板上画出一个序列图是非常有价值的，一个 5 到 6 页纸的非常短的序列图能够描绘一个子系统的共用的交互是金子般的珍贵的。另外一方面，一个画满了上千个序列图的文档是不值得打印出来的。

在上个世纪 90 年代，有关软件开发一个最大的谬论就是开发者需要在编码前画出所有方法的序列图，这被证明是非常浪费时间的，千万不要那样做！

相反，把仅序列图当作一种打算要用的工具，在一个白板上实时地使用它们与同事进行表达，用它们在文档中去捕获核心的、突出的系统协作。

至于对序列图的关注，少一点比多一点好，你总能够在发现需要它的时候去画它。

第五章　用例(use case)

用例(Use case)是已经变得极其复杂的一个出色的想法。我一次又一次地看到有些团队翻来捣去地试图编写用例。通常他们对用例的探讨流于形式而非实质，他们乐衷于讨论有关用例的前置条件、后置条件、参与者、次要参与者等一大堆其他的东西，唯独无关用例实质。

处理用例的真正诀窍是保持用例的简单，不要担心用例的形式，在一张白纸上、或是在一个简单的字处理器里的空白页里、或者在一个空白的索引页里写下它们。不要担心所有细节的填充，细节是不重要的，除非到了最后。不要担心所有的用例捕获，无论如何那是一个不可能完成的任务。

有关用例要留意的一件事情是：明天它们将要发生变化。不管你如何坚持不懈地去捕获它们，不管你如何一丝不苟地记录这些细节，不管你如何想得如何彻底，不管你多么努力地致力于探索和分析这些需求，明天它们将要发生变化。

如果有些事情明天将要发生变化，你今天就不需要真正去捕获细节。实际上，你要尽量推迟细节的捕获，直到最后可能的时刻。

把用例当做最及时的需求！

编写用例

注意这个章节的标题，我们是编写用例而不是画它们。用例并不是图，用例是有关动作性需求的文本性描述，从一个正确的视角点去编写它们。

“ 等等！”，你说：“ 据我所知，UML 有用例图，我曾经见过它们 ”。

是的，UML 的确有用例图，并且我们将在几页纸里学习它们。不过，那些图并没有告诉你所有有关用例的内容。它们是没有任何有关用例打算去捕获的动作性需求的信息。在 UML 中的用例图捕获的完全是别的东西，我们将在适当的时候去讨论它们。

什么是用例

一个用例是有关一个系统的行为的一个描述。那个描述是从一个用户的观点编写的，这个用户告诉系统去做一些特定的事情。一个用例捕获一个事件的可视化序列，这个事件是一个系统对单个用户的激励(stimulus)的响应过程。

一个可视化事件是这个用户可以看到的事件，用例不描述任何隐藏着的行为，它们不讨

论有关系统的隐藏着的机制，它们仅描述那些用户可以看到的事情。

主要课程

通常，一个用例被分解成两个部分，第一个部分是这个主要课程，这个部分描述了系统如何响应用户的激励(stimulus)，并假设没有出现错误。

举一个例子，这里有一个销售系统的一个点的典型用例：

Check Out Item（商品付帐）：

1、 收银员用扫描器扫读产品的 UPC（通用产品代码）。

2、 货品的单价、描述、当前的小计出现在面向顾客的显示器上，这个单价和描述也出现在这个收银员的屏幕上。

3、 价格和描述被打印在收条上。

4、 系统发出一个可以听得到的“ 确认 ”声音去通知收银员 UPC 已经被正确扫读到。

那就是一个用例的主要课程！不需要更复杂的东西。实际上，如果这个用例不打算马上被实现的话，上述的小序列可能是太详细了一点。我们将不要去记录这种类型的细节，除非这些用例在近几周内会被实现。

如果你不记录一个用例的细节，那么如何去评估它呢？在没有记录细节时，你可以跟商场经理讨论这些细节，讨论将给你有关充分评估的足够信息。如果我们打算跟商场经理讨论时，为什么我们不去记录这些细节呢？因为明天这些细节将会发生变化。难道变化不会影响到评估？会的，对于大多数的用例来说，那些影响是牵一发而动全身的。过早地记录这些细节恰恰是不值得的。

如果这个时候我们还不打算记录这个用例的细节的话，那么什么是我们要去记录的呢？如果我们不写下些什么的话，我们如何知道那个用例已经存在了呢？写下这个用例的名称，在一个电子表格里或是在一个字处理软件的文档里将它们形成一个列表，最好的话，写下这些用例的名称在一个索引卡上，并维护好这堆用例卡。当它们快要被实现的时候，填充这些细节。

预备课程

有些我们正在关心的事情的细节会出现问题，在与商场经理会唔的期间，你将要详细讨论那些有问题的场景。稍后，随着用例实现的时间越来越近，你将要更多地考虑那些用例的备选方案。它们成为那些用例的主要课程的补充部分，他们可以写成如下所示：

UPC Code Not Read（UPC 代码不可读取）：

如果 UPC 代码的扫读失败，这个系统将发出“ 重扫 ”的声音告诉这个收银员再试一下。如果三次扫读仍然失败，收银员将手工录入。

No UPC Code（没有 UPC 代码）：

如果货品没有一个 UPC 代码，收银员将手工录入价格。

这些预备课程是非常有趣的，因为它们暗示出了那些商场经理刚开始时没有识别出来的用例。在这个例子中，能够手工输入 UPC 或者价格显示是必要的。

其他

参与者、次要参与者、前置条件、后置条件、等等到底怎么回事呢？那些东西到底怎么回事呢？

不要担心这些，对于大多数系统来说，你会继续工作，你将不需要知道那一堆其他的事情。会有你需要更多地了解用例的时候，这些你可以读读 Alistair Cockburn 的权威性的著作——《有效编写用例》。对于现在来说，在你学会跑之前先学会走吧，你先习惯于编写如下所述的简单的用例吧。当你精通了它们时（即你可以在一个项目中成功地使用它们），你能够非常小心地、少量地采用那些复杂的技术。但要记住，不要翻来捣去地用。

用例图

在 UML 中所有的图，用例图是最容易混淆的、也是最没有什么用的，不过系统边界图（System Boundary Diagram）除外，在这里我将花一点时间来描述它，我还是建议你们全不用它们。

系统边界图

Figure 5-1 显示了一张系统边界图，那个大长方形表示系统边界。在这个长方形里面的任何东西都是要开发的系统的一部分。在这个长方形的外面，我们看到与系统交互的参与者(actor)，参与者是为系统提供激励(stimulus)的属于系统外的实体，通常是人类用户，也可以是其他系统，甚至可以是设备，如实时的时钟。

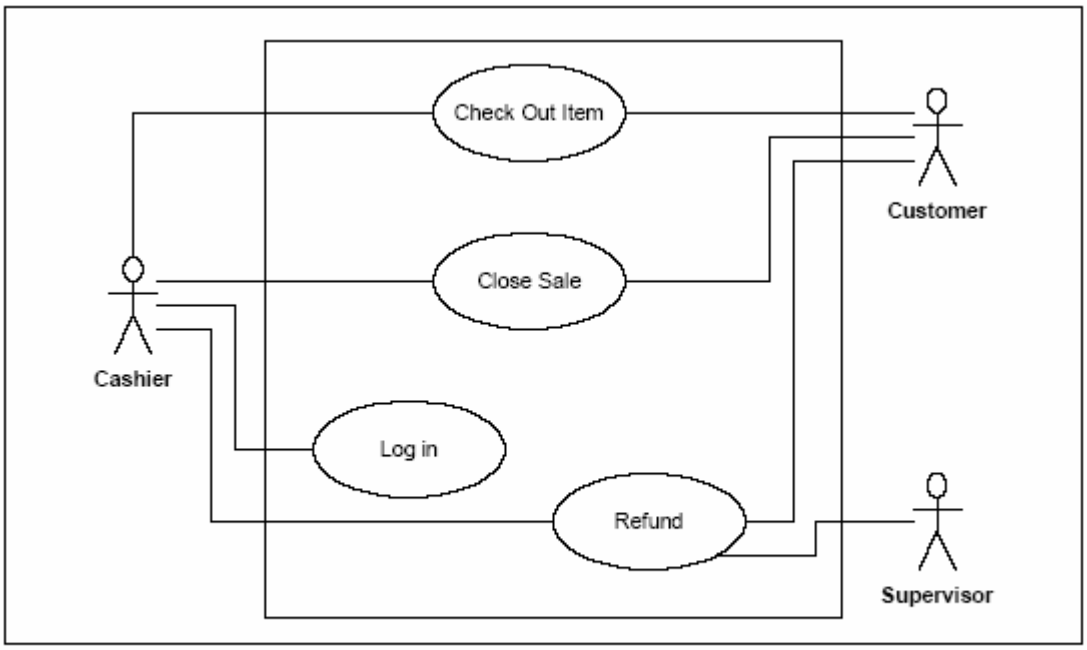


Figure 5-1
System Boundary Diagram

在这个边界长方形里面，我们可以看到用例，它们是在里面带名称的椭圆形。一根线将参与者连接到他们激励的用例上。要避免使用箭头，没有人能够知道这个箭头的方向代表着什么。

这张图几乎没有什么用。它对于 Java 程序员来说只包含了非常少的信息，不过它可以是向商场经理做讲解时一张比较好的封面页。

用例关系

用例关系属于那种“ 有时看起来是一个不错的主意 ”的东西。我建议你尽量地忽略它们。它们无论对于你的用例、还是对于系统的理解来说，都是没有什么价值的，它们将是那些到底是用“ 扩展 ”还是用“ 泛化 ”之类没完没了的争论的源头。

小结

这是比较短的一章，因为这个主题比较简单，所以篇幅是合适的。你的有关用例的态度必须是“ 简单 ”! ~~如果你一旦继续沿袭用例复杂化这条惨淡的路线，它将永远支配你的命运。使用力量、运气保持你的用例简单。~~

第六章　面向对象设计（OOD）原则

阅读有关 UML 标记方法的章节就像打开了一个巧克力糖盒子，刚吃了一些糖时就开始渴望能够来些肉食。让我们先搁下这甜蜜的 UML 标记方法一会儿，来吃一块美味的、鲜嫩的吉士汉堡包吧。

当我们察看一个 UML 图时，我们在找寻什么？我们如何评估它？什么是我们要应用的设计原则？这本章中，我们将讨论五个原则，它们将帮助我们评估一组 UML 图或一批代码是否被适当地设计。

设计质量

怎样意味着适当的设计？一个被适当地设计了的系统是容易被理解的、容易改变的和容易重用的。它表现为没有特别的开发困难，是简单的、扼要的和经济的。使用它是一件愉快的事情。相反地，一个糟糕的设计散发出像腐烂的肉般的臭味。

臭哄哄的设计

当一个程序员正忙于一个糟糕的设计时，你能够看到他查看代码时眼睛和鼻子的样子。如果他或她的面部表情仿佛像刚刚打开了一个装有已经死了 12 天的尸体的裹尸袋一般，毫无疑问，这个设计也许是相当臭的。一个糟糕的设计的臭味有许多不同的成分：

- **僵化性（Rigidity）**：系统难以被改变，因为每当你进行一个改动时，你不得不做其他没完没了的相关改动。
- **脆弱性(Fragility)**：针对系统的某一部分的改变，将导致许多系统中的其他完全不相关的部分的出现问题。
- **牢固性(immobility)**：很难将系统拆分成可被其他系统重用的组件。
- **粘滞性(Viscosity)**：这个开发环境的各个部分被像用胶袋纸和牙膏般地糊弄在一起的，进行编辑、编译和测试时折腾起来是没完没了的。
- **不必要的复杂性(Needless Complexity)**：存在着许多的精巧的代码结构，但是它们只将将来某一天非常有用，而对于现在是没有必要的。
- **不必要的重复(Needless repetition)**：代码看起来是被叫”cut”和”paste”的两个程序员编写的，系统存在着重复的代码。

- **晦涩性(Opacity)**：由于创作者的表达不够明晰，去阐明创作者的意图时表现得相当困难。

我们希望能够去除代码的这些臭味，UML 图常常能够帮助我们，因为通过考察图中的依存关系可以了解许多的问题。

依存关系管理

许多的臭味是无效管理的依存关系的结果。无效管理的依存关系导致代码的视图像一团乱麻，这种纠缠在一起的视图是术语“ 意大利式细面条般代码 ”的来源。

面向对象语言提供了有助于管理依存关系的工具，创建接口(interface)能够打破和转化一些依存关系的用法。多态性(Polymorphism)允许模块调用那些没有包含在内的的其他模块的函数。实际上，一个 OOPL 给了我们许多的方法去定型依存关系。

那么，我们如何使它定形依存关系呢？那就是接下来我要讲的一些原则。我已经编写了一些定型依存关系的原则，其更详细的叙述在[Martin2002]中，在www.objectmmentor.com上也有一些描述。以下是一个非常简要的汇总。

单一职责原则（SRP）

一个类应当只有一个改变的原因。

你可能已经读到了有关对象需要知道的一些废话，诸如如何在一个 GUI 上去画它们、如何将它们保存在磁盘上、如何将它们转化成 XML。像那样去开始学习 OO 知识，这对吗？真是可笑！类只需要知道一件事情，它们应当有一个单独的职责，要点就是当一个类需要改变时，应当只有一个原因。

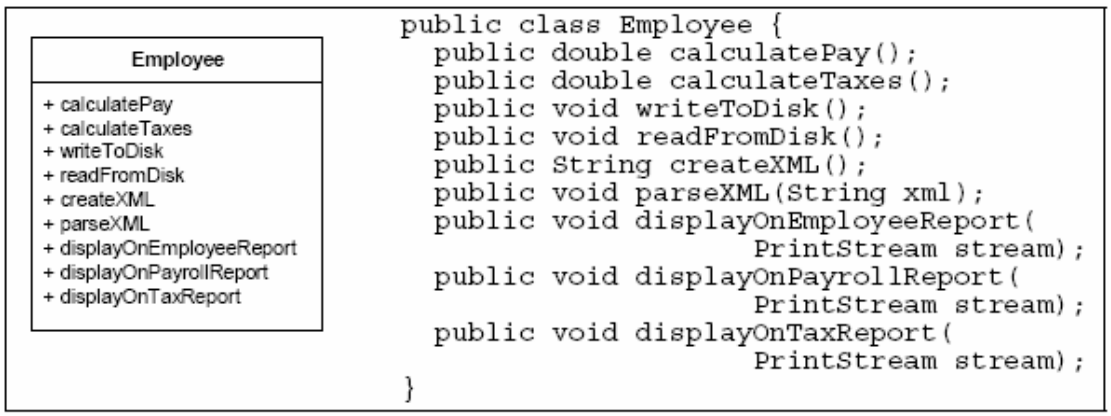


Figure 6-1
Class knows too many things.

考虑一下 Figure 6-1，这个类要处理的事情太多了。它知道如何去计算薪水和如何计算税费；知道如何在磁盘上读写它们自己；知道如何它将它们转化成 XML 或转化回来；知道如何用不同的报表打印它们自己。你能够嗅到脆弱性的臭味吗？从 SAX 变成 JDOM，你不得不改变 Employee 类。将数据库从 Access 变成 Oracle，你不得不改变 Employee 类。改变税费报表的格式，你不得不改变 Employee 类。这个设计被极为错误地连结。

实际上，我们要拆分所有的概念成为它们自己的类，每一个类有一个且只有一个改变的原因。我们将 Employee 类处理薪水和税费的计算，一个 XML 相关的类处理 Employee 实例和 XML 之间的转换，一个 EmployeeDatabase 类处理 Employee 实例和数据库之间的读写，每一个不同类型的报表有一个单独类。简而言之，我们要一个关系的拆分，一个可能的结构图如 Figure 6-2 所示：

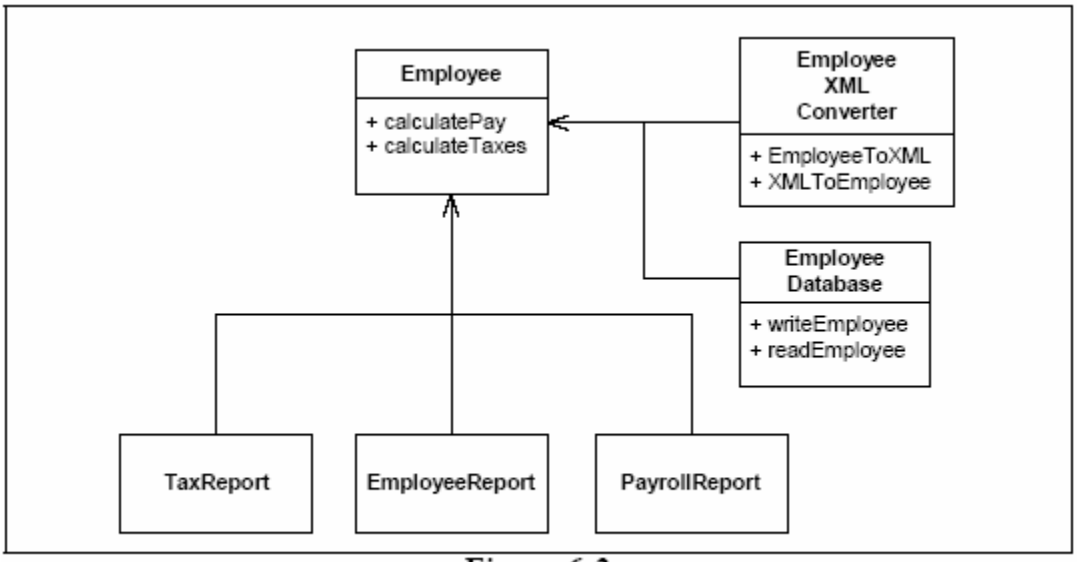


Figure 6-2
Separation of concerns.

在一个 UML 图上找出这个原则的违例是相当容易的，查看那些有一个以上主题范围的依存关系。~~写实现了一个或多个接口并赋予了一些属性的类是一种自寻死路的做法。~~例如粗心地使用一个能够将一个对象存储在磁盘上接口，能够导致一个类将商务规则和持久化问题纠结在一起。

考虑一下在 Figure 6-3 中的两个图，左边这个将 Persistable 和 Employee 紧紧地结合在一起，所有 Employee 的使用者将不可避免地依赖 Persistable。依存关系可能不太，但是的确存在在那里了，Persistable 接口的改变将潜在地影响所有 Employee 的使用者。

Figure 6-3 右边的那个分离了 Persistable 接口和 Employee 的依存关系，但仍然允许持久化。PersistableEmployee 的实例能够做为 Employee 传送到系统其他地方去，而无须留心系统有关的连结。连结的确存在，但是对大多数系统而言已经被隐藏掉了。

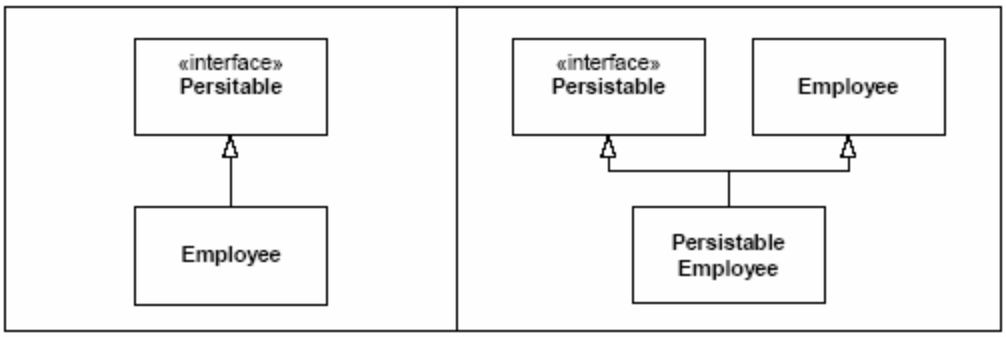


Figure 6-3
Two ways to use Serializable

开放-封闭原则（OCP）

软件实体(类、模块、函数等)应当为扩展而开放，又为修改而封闭。

这个原则有一个相当详细的定义，但是一个简单的意思是：**你应当能够改变一个模块的周边环境而无须改变模块本身**。

考虑一下 Figure 6-4 的例子，它显示了一个简单的应用程序，应用程序通过一个叫 EmployeeDB 的数据库接口(facade)来处理 Employee 对象，接口(facade)直接使用数据库的 API。这违反了 OCP，因为 EmployeeDB 类的实现的一个改变将强制要求 Employee 类进行重建。Employee 类被牢牢地与数据库的 API 绑定到了一起，任何包含了 Empoyee 类的系统必须也包含 TheDatabase API。

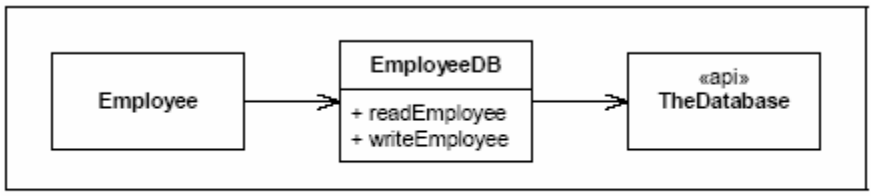


Figure 6-4
Violation of OCP

单元测试(Unit test)常常是用在我们要建立对于环境而言改变可控的地方。考虑一下这个例子，我们将如何测试 Employee，Employee 对象造成了数据库的改变。在一个测试的环境中，我们不需要真正的数据库的改变，我们也不需要仅仅为了单元测试而去创建一个虚构的数据库。相反地，我们宁愿去改变环境，这样这个测试捕获了 Employee 对数据库的所有调用，并且检验那些调用是正确的。

我们能够通过将 Employee 转换成一个**接口(interface)**来实现它，如 Figure 6-5 所示，接着我们能够创建调用真实的数据库 API 或者是支持我们的测试的派生类。这个接口将 Empolyee 从数据库 API 中分离了出来，并允许我们改变有关 Employee 的数据库环境而完

全不影响 Employee 本身。

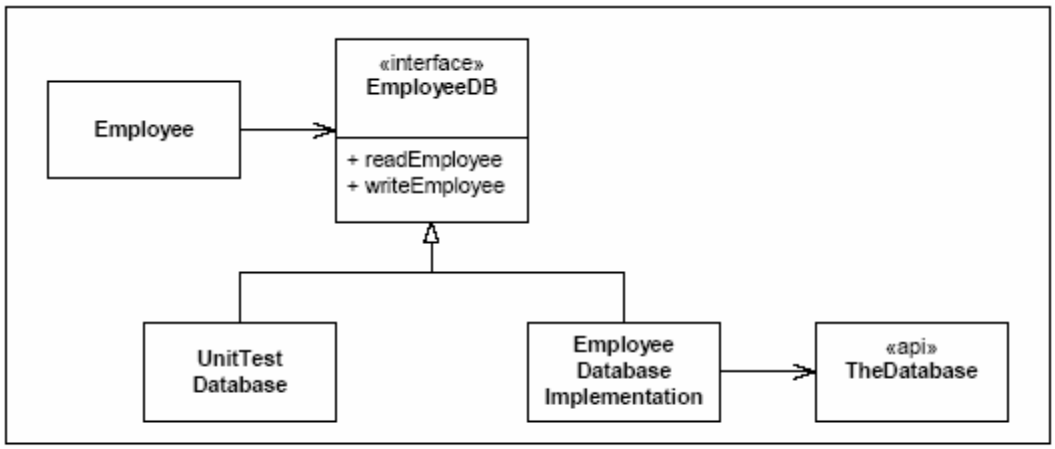


Figure 6-5
Conforming to the OCP

在系统中经常违反 OCP 的是 GUI，尽管 MODEL-VIEW-CONTROLLER 已经被我们所知道了近 30 年了，但是我们通常看起来还是不能好好地设计 GUI 系统。通常那些操纵 GUI API 的代码被紧紧地绑定到了那些管理和操纵被显示的数据的代码上。

考虑一下这个例子，一个非常简单的对话框，它显示雇员的列表。用户从列表中选择了一个雇员，然后点击”Terminate”（解雇）按钮。我们希望如果没有雇员被选择，”Terminate”按钮是被禁用掉了的。从另外一方面来说，如果我们从列表中选择了一个雇员，”Terminate”按钮将被启用，当用户点击了”Terminate”按钮，被解雇(terminated)的雇员将不被列表显示，如果没有雇员可选的时候，这个”Terminate”按钮将被禁用。

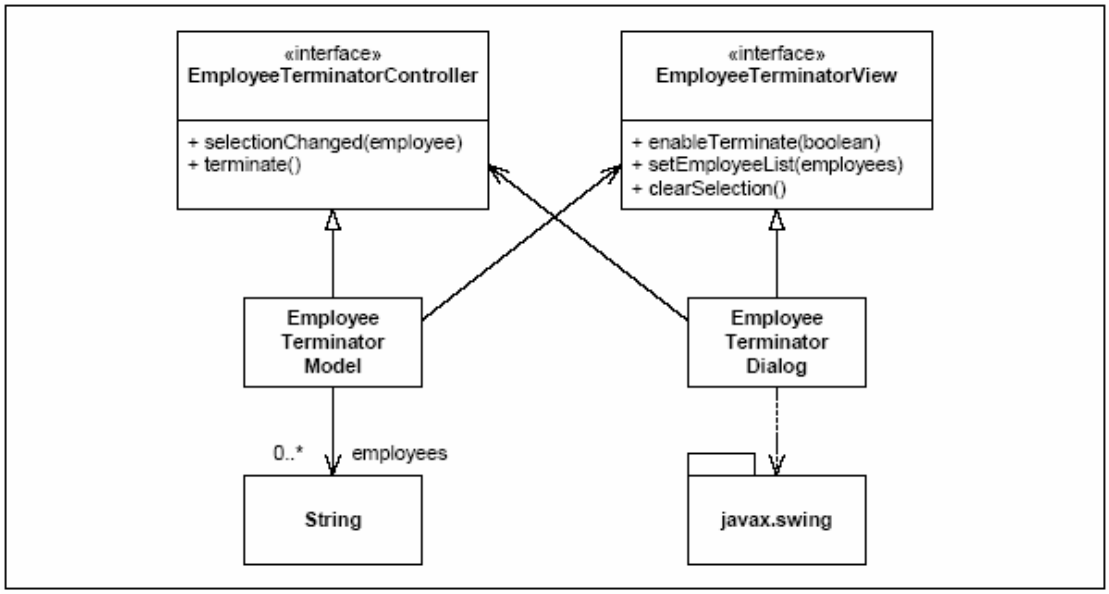


Figure 6-6
Isolating GUI from Data Manipulation

违反了 OCP 的实现将所有的行为放入一个使用 GUI API 的类中，遵循 OCP 的系统将 GUI 的操纵与数据的操纵分离。

Figure 6-6 显示了一个遵循 OCP 的系统，EmployeeTerminatorModel 管理雇员列表，当用户选择或解雇一个雇员时被通知。EmployeeTerminatorDialog 管理 GUI，它显示雇员的列表，当一个选择改变或"Terminate"按钮被按下时通知它的控制者。

EmployeeTerminatorModel 负责从列表中真正删除被选择的雇员，也负责检测解雇控制是启用还是禁用，它并不知道这个控制是被按钮的实现的。它只是简单地告诉它相关的视图是否允许用户执行解雇操作。类似地，尽管这个模型并不知道一个列表框的任何信息，它能够告诉它的视图去清除选择。

这个 EmployeeTerminatorDialog 是极弱智的，它自己不做决定，也不管理数据。这个 EmployeeTerminatorModel 操作它的字符串和使对话框能够响应操作。如果用户用对话框交互，它只是通过调用 EmployeeTerminatorController 中的方法，简单地告诉它的控制者什么操作在进行，消息被传送到接下来解释和具体操作的模型中。

Listing 6-1 到 Listing 6-4 是以上结构的 Java 实现，这两个接口并没有什么让人惊奇的。有人可能要问为什么在 EmployeeTerminatorController 中 selectionChanged 与 terminate 是分开的函数？为什么 terminate 没有接收 employee 参数？我之所以这样做是为了不让 EmployeeTerminatorDialog 去管用户点击了“ 解雇 ”按钮需要去做什么之类的事情。

Listing 6-1
EmployeeTerminatorView.java

```
import java.util.Vector;

public interface EmployeeTerminatorView {
    void enableTerminate(boolean enable);
    void setEmployeeList(Vector employees);
    void clearSelection();
}
```

Listing 6-2
EmployeeTerminatorController.java

```
public interface EmployeeTerminatorController {
    public void selectionChanged(String employee);
    public void terminate();
}
```

EmployeeTerminatorModel 是非常明晰的，在构造函数初始化中它发送雇员的列表给

视图、清除了选择并禁用了“ 解雇 ”命令。当对话框通过调用 selectionchange 方法报告了一个选择改变时，这个模型适当地启用了“ 解雇 ”按钮并保存了这个选择结果。当对话框调用“ 解雇 ”操作时，这个被选择的雇员被从列表中删除，这个被修改的列表被送回到视图，选择被清除掉了，这个“ 解雇 ”按钮被禁用了。

Listing 6-3
EmployeeTerminatorModel.java

```
import java.util.Vector;
public class EmployeeTerminatorModel
                implements EmployeeTerminatorController {
   private EmployeeTerminatorView view;
   private Vector employees;
   private String selectedEmployee;

   public void initialize(Vector employees,EmployeeTerminatorView view)
   {
      this.employees = employees;
      this.view = view;
      view.setEmployeeList(employees);
      view.clearSelection();
      view.enableTerminate(false);
   }

   // EmployeeTerminatorController interface
   public void selectionChanged(String employee) {
      view.enableTerminate(employee != null);
      selectedEmployee = employee;
   }

   public void terminate() {
      if (selectedEmployee != null)
         employees.remove(selectedEmployee);
      view.setEmployeeList(employees);
      view.clearSelection();
      view.enableTerminate(false);
   }
}
```

EmployeeTerminatorDialog 是其中最复杂的一个类，幸运的是，这种复杂性只是跟使用 GUI 各种控件有关，而跟系统的业务规则没有关系。它简单地创建了“ 解雇 ”按钮和列表框，适当地把它们组合在一起并做好显示出来的准备。EmployeeTerminatorView 的实现是很容易的，并没有什么地方让人惊奇。

Listing 6-4
EmployeeTerminatorDialog.java

```
import javax.swing.*;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;
import java.awt.*;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Vector;

public class EmployeeTerminatorDialog  implements EmployeeTerminatorView {
    private JFrame frame;
    private JList listBox;
    private JButton terminateButton;
    private EmployeeTerminatorController controller;
    private Vector employees;
    public static final String EMPLOYEE_LIST_NAME = "Employee List";
    public static final String TERMINATE_BUTTON_NAME = "Terminate";

    public void  initialize(EmployeeTerminatorController controller) {
        this.controller = controller;
        initializeEmployeeListBox();
        initializeTerminateButton();
        initializeContentPane();
    }

    private void initializeEmployeeListBox()
    {
        listBox = new JList();
        listBox.setName(EMPLOYEE_LIST_NAME);
        listBox.addListSelectionListener(new ListSelectionListener() {
           public void valueChanged(ListSelectionEvent e)
           {
             if (!e.getValueIsAdjusting())
                 controller.selectionChanged(
                   (String)listBox.getSelectedValue());
           }
         }
        );
     }

    private void initializeTerminateButton() {
        terminateButton = new JButton(TERMINATE_BUTTON_NAME);
        terminateButton.disable();
```

```
        terminateButton.setName(TERMINATE_BUTTON_NAME);
        terminateButton.addActionListener(
        new ActionListener() {
          public void actionPerformed(ActionEvent e) {
          controller.terminate();
        }
        }
        );
    }

    private void initializeContentPane() {
        frame = new JFrame("Employee List");
        frame.getContentPane().setLayout(new FlowLayout());
        frame.getContentPane().add(listBox);
        frame.getContentPane().add(terminateButton);
        frame.getContentPane().setSize(300, 600);
        frame.pack();
    }

    public Container getContentPane() {
        return frame.getContentPane();
    }

    public JFrame getFrame() {
        return frame;
    }

    // functions for EmployeeTerminatorView interface
    public void enableTerminate(boolean enable) {
        terminateButton.setEnabled(enable);
    }

    public void setEmployeeList(Vector employees) {
        this.employees = employees;
        listBox.setListData(employees);
        frame.pack();
    }

    public void clearSelection() {
        listBox.clearSelection();
    }
}
```

当有选择的变化发生时，model 对象与 dialog 对象以一种有趣的方式进行交互。dialog 通过 controller 接口向 model 报告所有的选择变化，其中包括当 model 调用 clearSelection 方法时的变化。就像你在 Figure 6-7 中看到的一样，当 model 调用在 dialog 上的 clearSelection 方法时，dialog 是通过调用 model 上的 setSelection 方法来响应的。

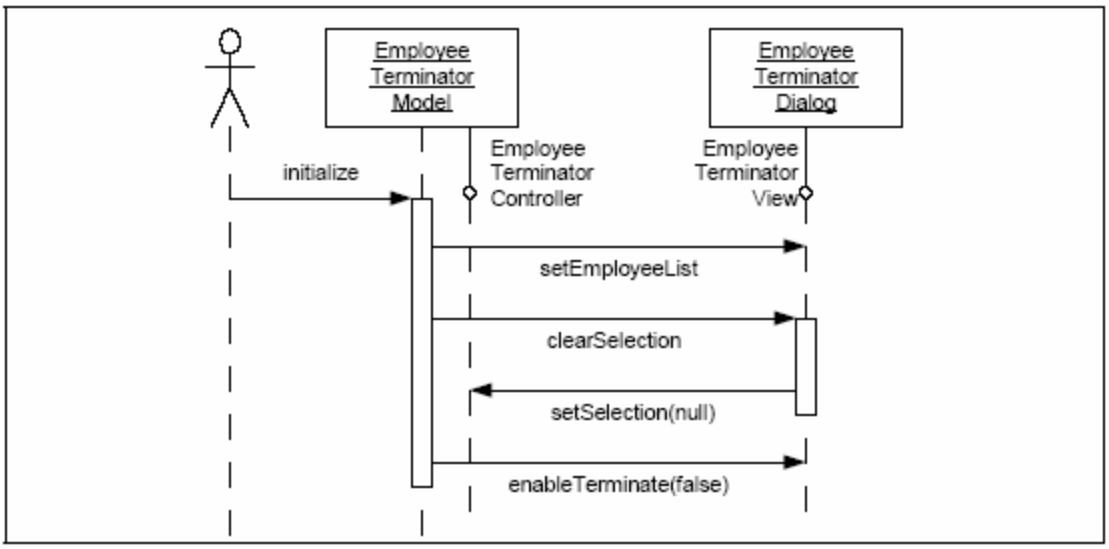


Figure 6-7
Interaction between the Model and Dialog

当你看到了 model 和 dialog 的**单元测试**(unit test)的代码时(分别在 Listing 6-5 和 Listing 6-6 中)，你就最能够清楚以上的设计是多么符合 OCP 原则了。~~无须被测试的模块知道其他相关模块的存在的情况下~~，这些单元测试就可以工作。TestEmployeeTerminatorModel 类测试 model 的功能时，这个测试假扮了一个 EmployeeTerminatorView 并捕获所有 model 可以发送到 view 的消息，来验证它们的调用是在正确的时机和携带了合适的信息。这是我们知道的有关单元测试的 SELF SHUNT 模式。

Listing 6-5
TestEmployeeTerminatorModel.java

```
import junit.framework.TestCase;
import junit.swingui.TestRunner;
import java.util.Vector;

public class TestEmployeeTerminatorModel extends TestCase
                implements EmployeeTerminatorView {
private boolean terminateEnabled = true;
private String selectedEmployee;
private Vector noEmployees = new Vector();
private Vector threeEmployees = new Vector();
private Vector employees = null;
private EmployeeTerminatorModel m;
```

```
public static void main(String[] args) {
    TestRunner.main(new String[]{"TestEmployeeTerminatorModel"});
}

public TestEmployeeTerminatorModel(String name) {
    super(name);
}

public void setUp() throws Exception {
    m = new EmployeeTerminatorModel();
    threeEmployees.add("Bob");
    threeEmployees.add("Bill");
    threeEmployees.add("Robert");
}

public void tearDown() throws Exception {
}

public void testNoEmployees() throws Exception {
    m.initialize(noEmployees, this);
    assertEquals(0, employees.size());
    assertEquals(false, terminateEnabled);
    assertEquals(null, selectedEmployee);
}

public void testThreeEmployees() throws Exception {
    m.initialize(threeEmployees, this);
    assertEquals(3, employees.size());
    assertEquals(false, terminateEnabled);
    assertEquals(null, selectedEmployee);
}

public void testSelection() throws Exception {
    m.initialize(threeEmployees, this);
    m.selectionChanged("Bob");
    assertEquals(true, terminateEnabled);
    m.selectionChanged(null);
    assertEquals(false, terminateEnabled);
}

public void testTerminate() throws Exception {
    m.initialize(threeEmployees, this);
    assertEquals(3, employees.size());
    selectedEmployee = "Bob";
```

```
        m.selectionChanged("Bob");
        m.terminate();
        assertEquals(2, employees.size());
        assertEquals(null, selectedEmployee);
        assertEquals(false, terminateEnabled);
        assert(employees.contains("Bill"));
        assert(employees.contains("Robert"));
        assert(!employees.contains("Bob"));
    }

    // EmployeeTerminatorView interface
    public void enableTerminate(boolean enable) {
        terminateEnabled = enable;
    }

    public void setEmployeeList(Vector employees) {
        this.employees = (Vector) employees.clone();
    }

    public void clearSelection() {
        selectedEmployee = null;
    }
}
```

TestEmployeeTerminatorDialog 类也使用了 SELF SHUNT 模式，它假扮了一个 TestTerminatorController，它捕获了从 dialog 发送到 controller 的消息，验证了它们的调用是在正确的时机和包含了合适的数据。这个测试的大部分的工作是检查了这个对话框的组合，它验证了列表框和按钮被正确地创建，验证了假定的功能。

Listing 6-6
TestEmployeeTerminatorDialog.java

```
import junit.framework.TestCase;
import junit.swingui.TestRunner;

import javax.swing.*;
import java.awt.*;
import java.util.HashMap;
import java.util.Vector;

public class TestEmployeeTerminatorDialog
  extends TestCase
  implements EmployeeTerminatorController
{
    private EmployeeTerminatorDialog terminator;
```

```
    private JList list;
    private JButton button;
    private Container contentPane;
    private String selectedValue = null;
    private int selectionCount = 0;
    private int terminations = 0;

    public static void main(String[] args) {
      TestRunner.main(new String[]{"TestEmployeeTerminatorDialog"});
    }

    public TestEmployeeTerminatorDialog(String name) {
      super(name);
    }

    public void setUp() throws Exception {
      terminator = new EmployeeTerminatorDialog();
      terminator.initialize(this);
      putComponentsIntoMemberVariables();
    }

    private void putComponentsIntoMemberVariables() {
      contentPane = terminator.getContentPane();
      HashMap map = new HashMap();
      for (int i = 0; i < contentPane.getComponentCount(); i++) {
        Component c = contentPane.getComponent(i);
        map.put(c.getName(), c);
      }
      list = (JList) map.get(
        EmployeeTerminatorDialog.EMPLOYEE_LIST_NAME);
      button = (JButton) map.get(
      EmployeeTerminatorDialog.TERMINATE_BUTTON_NAME);
    }

    private void putThreeEmployeesIntoTerminator() {
      Vector v = new Vector();
      v.add("Bob");
      v.add("Bill");
      v.add("Boris");
      terminator.setEmployeeList(v);
    }

    public void testCreate() throws Exception {
      assertNotNull(contentPane);
```

```
    assertEquals(2, contentPane.getComponentCount());
    assertNotNull(list);
    assertNotNull(button);
    assertEquals(false, button.isEnabled());
  }

  public void testAddOneName() throws Exception {
    Vector v = new Vector();
    v.add("Bob");
    terminator.setEmployeeList(v);
    ListModel m = list.getModel();
    assertEquals(1, m.getSize());
    assertEquals("Bob", m.getElementAt(0));
  }

  public void testAddManyNames() throws Exception {
    putThreeEmployeesIntoTerminator();
    ListModel m = list.getModel();
    assertEquals(3, m.getSize());
    assertEquals("Bob", m.getElementAt(0));
    assertEquals("Bill", m.getElementAt(1));
    assertEquals("Boris", m.getElementAt(2));
  }

  public void testEnableTerminate() throws Exception {
    terminator.enableTerminate(true);
    assertEquals(true, button.isEnabled());
    terminator.enableTerminate(false);
    assertEquals(false, button.isEnabled());
  }

  public void testClearSelection() throws Exception {
    putThreeEmployeesIntoTerminator();
    list.setSelectedIndex(1);
    assertNotNull(list.getSelectedValue());
    terminator.clearSelection();
    assertEquals(null, list.getSelectedValue());
  }

  public void testSelectionChangedCallback() throws Exception {
    putThreeEmployeesIntoTerminator();
    list.setSelectedIndex(1);
    assertEquals("Bill", selectedValue);
    assertEquals(1, selectionCount);
```

```
        list.setSelectedIndex(2);
        assertEquals("Boris", selectedValue);
        assertEquals(2, selectionCount);
      }

      public void testTerminateButtonCallback() throws Exception {
        button.doClick();
        assertEquals(1, terminations);
      }

      // implement EmployeeTerminatorController
      public void selectionChanged(String employee) {
        selectedValue = employee;
        selectionCount++;
      }

      public void terminate() {
        terminations++;
      }
  }
```

这个测试演示了这个设计遵循了 OCP 原则，因为可以改变有关 dialog 和 model 的周边环境到一个测试环境，而无须 dialog 和 model 知道它。考虑一下这个模块的伸缩性术语的意思。我们能够轻松地用一个命令行的 UI(用户界面)或一个文本菜单 UI 替代这个 dialog，这个 model 将不知道这个区别。我们能够将 model 和 dialog 使用 RMI 技术放置在不同的机器上。我们能够在不影响其他模块的情况下改变每一个模块的环境。

非常容易了解遵循 OCP 原则的机制，让我们再看看 Figure 6-6，我们看到这个我们非常熟悉的 FLIP-FLOP 模式，模块实现了一个接口并与其他模块通讯。很明显，我们能够轻松地改变这个模块的周边环境，因为它们使用了抽象的接口。实际上，遵循 OCP 原则的一个关键是使用抽象。

那么，我们应该怎样标识出帮助我们遵循 OCP 原则的抽象呢？为了地遵循 OCP 原则，通常我只是简单地在编写实际代码之前编写测试。在所有之前，使用了 test-first（测试优先）方法的单元测试被编写。每一个测试函数首先被编写，接着在模块中编写足够的代码以让测试功能通过。

出于完整性的考虑，Listing 6-7 说明了绑定了 model 和 model 在一起的代码，显示了这个对话框，我用这个模块做为最后的、手工的测试。它允许我去检验这个对话框的外观。这是我编写的真正显示对话框的仅有的代码，以前的 dialog 单元测试只是简单地验证了对话

框被正确地组合及功能正确，并没有实际地显示它。

Listing 6-7
ShowEmployeeTerminator.java

```
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.util.Vector;

public class ShowEmployeeTerminator {
    static Vector employees = new Vector();
    static EmployeeTerminatorDialog dialog;

    public static void main(String[] args) {
        initializeEmployeeVector();
        initializeDialog();
        runDialog();
    }

    private static void initializeEmployeeVector() {
        employees.add("Bob");
        employees.add("Bill");
        employees.add("Robert");
    }

    private static void initializeDialog() {
        EmployeeTerminatorModel model = new
        EmployeeTerminatorModel();
        dialog = new EmployeeTerminatorDialog();
        dialog.initialize(model);
        model.initialize(employees, dialog);
    }

    private static void runDialog() {
        dialog.getFrame().addWindowListener(
        new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
          for (int i = 0; i < employees.size(); i++) {
          String s = (String) employees.elementAt(i);
          System.out.println(s);
        }
        System.exit(0);
        }
        }
        );
        dialog.getFrame().setVisible(true);
```

```
        }
    }
```

Liskov 替换原则(LSP)

子类型(subtypes)必须是为它们的基类型(base types)可替代的。

你曾经在if语句的子句中看过一些有instanceof表达式的代码吗？虽然像这种表达式是合法的，也是较少的，其实通常它们是违反了 LSP 原则的结果，同时它们也违反了 OCP 原则。

这个 LSP 原则是指基类(base classs)的用户为了使用它们的派生类，应当无须做特别的处理。特别是，它们应当无须使用 instanceof 或向下转型(Downcast)操作。实际上，它们根本无须知道有关派生类的事情，甚至无须知道它们是否存在。

考虑一下显示在 Figure 6-8 的薪水计算应用程序，这个 Employee 类是抽象的，并且有一个名叫 calcPay 的抽象方法。非常明显，SalariedEmployee 将实现它并返回雇员的薪水，同时也能很明显地看出 HourlyEmployee 将实现它并返回根据本周时间卡汇总后的按小时的费用。

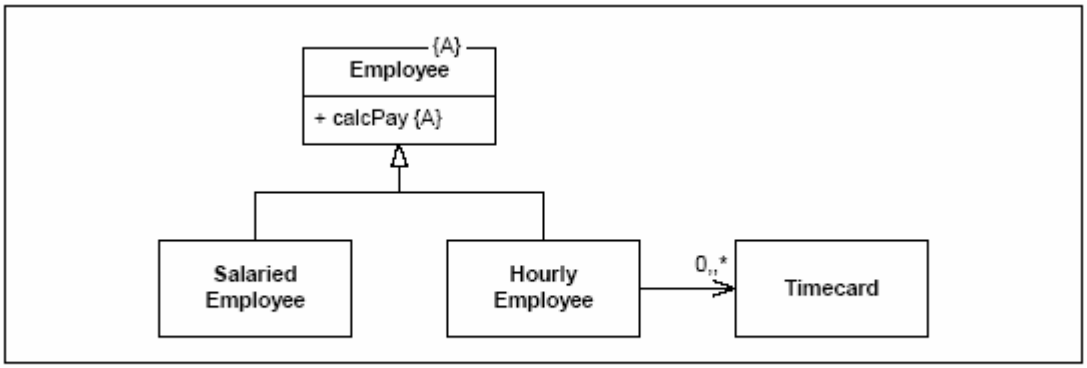


Figure 6-8
Simple Payroll Example

如果我们决定要增加一个 VolunteerEmployee 类的时候将会发生什么呢？我们将如何实现 calcPay 方法呢？初一看这好像比较简单，我们将实现返回 0 值的 calcPay 方法即可，如下所示。

```
public class VolunteerEmployee extends Employee {
  public double calcPay() {
    return 0;
  }
}
```

但是这样做对吗？在 VolunteerEmployee 中调用 calcPay 方法是任何意义吗？毕竟 calcPay 返回 0 值表示这个方法是有必要被调用的、付费是可能的。在我们打印和邮寄一个

付费总额为 0 的支票或其他类似的情况的时候，我们可能发现我们自己已经陷入了窘境。

因此，也许处理它的最好办法是抛出一个异常，表示这函数并不需要被调用。

```
public class VolunteerEmployee extends Employee {
  public double calcPay() {
    throw new UnpayableEmployeeException();
  }
}
```

乍一看，这好像是一种情有可原的做法，毕竟调用 VolunteerEmployee 的 calcPay 方法是非法的。像这种非法的情况下，异常将被抛出。

不幸的是,每一次调用 calcPay 方法将会抛出一个 UnpayableEmloyeeException 异常，因此要么异常必须被捕获，要么被调用者声明。因此，在一个派生类上的约束已经影响到了基类的用户。

更要命的是，如下所示的代码现在是非法的了。

```
    for (int i = 0; i < employees.size(); i++) {
      Employee e = (Employee) employees.elementAt(i);
      totalPay +=  e.calcPay();
    }
```

为了使它合法，我们将这个调用放入到一个 try/catch 语句块中。

```
    for (int i = 0; i < employees.size(); i++) {
      Employee e = (Employee) employees.elementAt(i);
      try {
        totalPay +=  e.calcPay();
      }
      catch (UnpayableEmployeeException e1) {
      }
    }
    return totalPay;
  }
```

这种令人讨厌，过于复杂，也让人觉得心情烦燥。我们就很想改成如下所示的代码：

```
for (int i = 0; i < employees.size(); i++) {
  Employee e = (Employee) employees.elementAt(i);
  if (!(e instanceof VolunteerEmployee))
    totalPay += e.calcPay();
}
```

但是，这种做法是相当有问题的，因为现在的代码已经假定在 Employee 这个基类上的操作已经创建了对它的一个派生类的明确的引用。

所有这些已经造成的混淆是因为我们违反了 LSP 原则。VolunteerEmployee 不是 Employee 可替换的，并且这个奇怪的异常和在 if 语句中古怪的 instanceof 子句的的结果，都是因为违反了 OCP 原则。

只要当你调用一个派生类上的函数时造成了非法使用，你就知道你正在违反了 LSP 原则，如果你使用了一个退化的派生类的方法，也就是说什么也没有实现，你也正在违反 LSP 原则。在这两种情况下，你可以说在派生类中这个函数是没有意义的，并且这是一个最后能

导致有令人生厌的异常和 instanceof 测试的 LSP 原则的违例。

那么什么是这个 VolunteerEmployee 问题的解决方法呢？Volunteers（志愿者）不是雇员，在它们上面调用 calcPay 方法是没有意义的，因此它们不应当从 Employee 派生出来，它们应当不需要使用 calcPay 函数。

依存关系倒置原则(DIP)

A．高层模块应当不依赖低层模块，它们应当依赖于抽象。

B．抽象应当不依赖于细节，细节应当依赖于抽象。

更好的描述是：**不要依赖那些容易变化的具体类**。如果你要继承一个类，从一个抽象类继承吧。如果你要持有一个类的引用，从一个抽象的类引用吧。如果你要调用一个函数，从一个抽象的函数调用吧。

一般来说，对抽象类和接口的改变远不及由它们派生出来的具体的类的改变更频繁。因此，我们宁可依赖抽象(指抽象类和接口)而不是具体的实现。遵循这个原则将减少一个变化对系统的影响。

这是不是意味着我们就不能使用 Vector 或 String 类了呢？毕竟，它们是具体的类。是不是使用它们就等同于违反了 DIP 原则呢？不是的！依赖一个不会发生改变的具体的类是相当安全的。Vector 和 String 类在下一个十年内不会发生改变（或变化不多），因此，我们能够放心地使用它们。

我们要避免去依赖一个容易变化的具体类，它们是指那些正在开发的具体的类、那些容易变化的捕获商业规则的类。我们要为那些类创建接口并依赖于那些接口。

从一个 UML 的立场来看，这个原则是非常容易被检查的。顺着一个 UML 图中的每一个箭头，确认每一个箭头指向的是一个接口或一个抽象的类。如果不是这样的、或者具体类是容易变化的，这就违反了 DIP 原则的，这个系统将对变化十分敏感。

接口隔离原则（ISP）

客户不应当依赖那些它们根本不用的方法。

你是否曾经看到过一个肥类(fat class)，一个肥类是一个有着成堆的方法的类，而在我们的系统中并不需要如此多的类，但是有时它们是不可避免。

伴随着肥类的问题除了它们过大和令人讨厌外，它们的用户极少地使用它们的方法，这

是指一个类声明了一打方法，而用户却只使用两个或三个方法。更不幸的是，当这些并不被用户调用的方法发生了变化时，它们的用户受到了影响。

举一个例子，考虑一下在 Figure 6-9 中的注册系统的教程，这个图显示了一个叫 StudentEnrollment 的类的两个客户。很明显这个 EnrollmentReportGenerator 并不调用 prepareInvoice 或 postPayment 这样的方法。我们也可以推断 AccountsReceivable 并不调用 getName 和 getDate 方法。

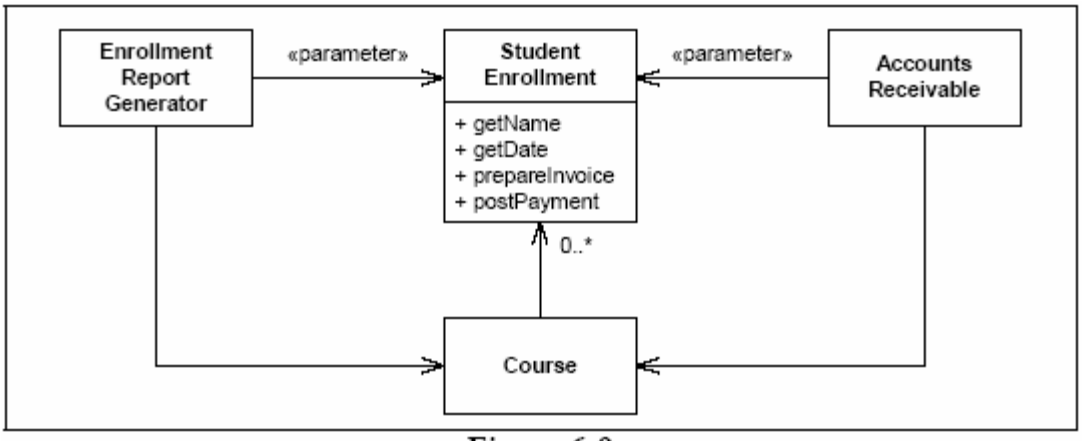


Figure 6-9
Unsegregated Enrollment System

现在，假如稍微有一些需求变化，它要求我们为 postPayment 方法增加一个新的参数。对 StudentEnrollment 声明的改变将强迫我们将 EnrollmentReportGenerator 类重新编译和重新部署。这是非常不幸的，因为 EnrollmentReportGenerator 就根本不关心 postPayment 方法。

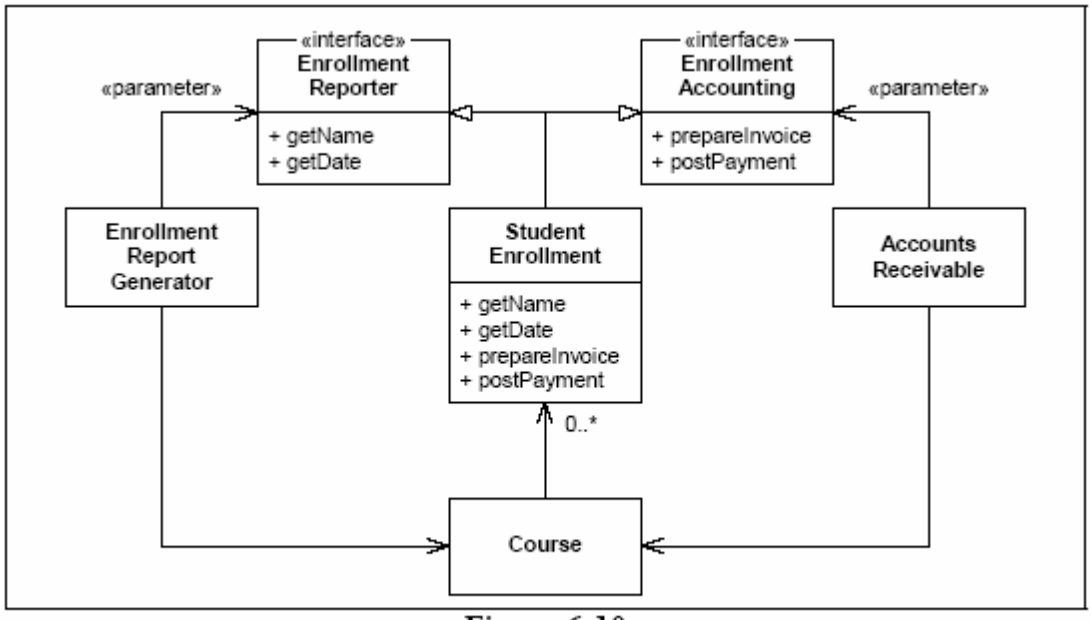


Figure 6-10
Segregated Enrollment System

通过遵循一个简单的规则，我们就能够防止这种不幸的依赖。~~从不用的方法方面保护用户，给他们一些仅有他们需要的方法的接口~~。Figure 6-10 显示了如何应用这种规则。

每一个 StudentEnrollment 对象的用户被指定了一个接口，给它们提供了它们感兴趣的方法。这从只改变用户不涉及的方法的方式来保护用户，它也从充分了解一个正在使用的对象的实现方面保护用户。

小结

五个简单的原则是：

1、 SRP－－－一个类应当只有一个发生变化的原因。

2、 OCP——应当能够改变一个类的环境，而无须改变类本身。

3、 LSP——避免造成派生类的方法非法或退化，一个基类的用户应当不需要知道这个派生类。

4、 DIP——用依赖于接口和抽象类来替代依赖容易变化的具体类。

5、 ISP——给一个对象的每一个用户一个接口，这个接口仅有用户需要的方法。

什么时候我们应当应用这些原则呢？首先一个痛苦的提示是，试图让所有的系统时时刻刻遵循所有的原则那是不明智的。你将花费很长的时间试图去设想在所有不同的环境中应用 OCP 原则，或所有不同的发生改变的源头应用 SRP，你将要为应用 ISP 原则编写大量的小接口，将为应用 DIP 原则而创建一堆没有价值的抽象。

应用这些原则的最好方式是与主动性式(proactively)相对的反应性式(reactively)的方法。当你第一次觉察到在代码里有一个结构性的问题存在的时候，或当你第一次意识到一个模块的改变被其他模块影响到了的时候，你应当看看是否能够运用这些原则去解决问题。

当然，如果你使用一个主动性式方法去应用这些原则，你也需要运用一个主动性式途径去解决系统上的将造成痛苦的各种类型的压力。如果你对痛苦有了反应，你需要坚持不懈地发现这个痛点(the sore spots)。

去搜寻这些痛点的最好方式之一是编写许多的单元测试。如果在你编写代码并传递它们之前，先编写这个单元测试，它将工作得更好，这是下一章要讨论的一个主题。

参考文献

[Feathers2001]: The 'Self'-Shunt Unit Testing Pattern, Michael Feathers, May, 2001,

http://www.objectmentor.com/resources/articles/SelfShunPtrn.pdf

[Martin2002]: The Principles, Patterns, and Practices of Agile Software Development,

Robert C. Martin, Prentice Hall, 2002

第七章　dX 实践

——感谢 Melthaw Zhang 对翻译本章的贡献！

现在我们可以将前面讲到的东西整合起来，形成一套的实践，真正让一个团队可以完成项目。类似的一组实践有时也被称作为一个过程。但是，下面我将阐述的还是轻量级的，不足以用过程来形容。可以认为它是一套简单的规则，通过应用这套规则，使一个团队的工作达到立竿见影的效果。我给这套规则起了个名字：**dX**。

迭代开发

关键字“ dX 实践 ”说白了就是短周期迭代式处理“ 所有事情 ”。这里我所指的“ 所有事情 ”包括了需求、分析、设计、实现、测试和文档等等。而所谓的短周期就是两个星期。在两个星期内要涉及到所有的事情。每个周期以计划作为开始，交付作为结束。在经过初始探索阶段后，每个周期首要交付的就是可以工作的代码，即使是最早的周期。

初始探索

我们的第一次 dX 迭代将从探索需求展开。这是唯一的不需要用代码来作为结束标记的迭代。也是唯一的可能不用两个星期就可以完成的迭代。典型的初始探索可能只有几天，有时候可能一个星期，两个星期的情况就比较少见了。

首先我们需要负责需求和负责决策的人，也就是“ 客户 ”。实际上在很多项目里，就是实实在在的客户。还有一些项目里面，“ 客户 ”这个角色也指做商业分析的人 s。

然后我们和客户坐下来一起讨论系统到底是做什么的。刚开始的几天，大家只需要讨论一些系统怎么运作，以及必须的一些功能等等。大家也不用记录太多内容，因为这个时候我们并不是要把握需求，只是要搞清楚系统的整体概况。

在讨论系统过程中，我们识别出 use-case（**用例**）。然后把每一个 use-case 的名字记在索引卡片上。这些卡片也就是我们所说的“ user story ”。这些 story（**素材**）除了 use-case 的名字以外并不包括 use case 的细节。可以在卡片写下一些其他的重要信息，但是这并不是代表要将这个一口气做完，其实我们还是在做一些把握系统的整体情况的工作。

这样的探索过程不会终止。即使我们已经进入实现阶段，甚至是已经发布了第 N 个系

统的版本之后，我们还是需要和客户坐下来讨论有什么新的需求和功能特征。

功能特征评估

现在我们要在每个卡片上写下对相应 story 的评估。这种评估纯粹就是一个数字。这个时候我们还不关心数字的单位是什么。对于每个 story 之间的比例的评估才是我们真正关心的。也就是说，一个 8 个点数的 story 的是一个 4 个点数的 story 的两倍。

最好的评估方式就是将当前的评估建立在上一个 story 的基础上。如果已知上一个 story 的评估结果是 6 个点数，你手头正好有一个新的 story，你可以给新的 story 评估为 5 个，6 个，或者 7 个点数，这些都取决于你要处理的 story 的难易程度。如果手头上没有现成的 story 的评估可以参考的话，那你可以用 “ 完美编程 ” 的天数作为标准。“ 完美编程 ” 的天数可以这样理解，从头天晚上准时上床，第二天早上起来享用了像样的早餐，去上班的路上也没有塞车，工作的时候根本没有电话骚扰，也没有什么官僚会议要参加，电脑从不死机，网络也快的不得了，你的搭档跟你合作的时候也出奇的机敏，考虑又周全，而且还很有耐心，在这样的情况下工作一天我们就称为进行了一天 “ 完美编程 ”。在这样的情况下你一天能做多少事情呢？假如你每天都这样工作，你手头上的 story 需要多少天才能完成？写下这个数字后就把这种得心应手的日子忘记吧，毕竟，现实是残酷的，这样的日子根本不会出现。太长的 story 需要进行分割。反之，太短的 story 也需要合并。如果让整个团队来完成一个 story，最好不要超过 3 到 4 天，最短也不能低于半天。如果最后得到的 story 太短往往是高估自己的实力了，而太长的 story 则可能评估太保守了。所以我们要经常合并或者拆分 story，直至达到比较准确合理的评估为止。

探究

在上一次初始探索中，我们花两到三天的时间很快地粗糙地实现两三个比较有趣的 story。然后就不管了。我们这样做的目的很简单，就是要验证一下我们的评估。举个例子，如果完成一个 7 个点数的 story 需要 5 人天的工作量的话。那么就意味着我们可以在 5 天内快速的粗略的完成 7 个点数。也许要保质保量的完成这样的工作需要 3 倍的时间。因此可以调整一下我们的评估为 15 天完成 7 个点。除一下再四舍五入后就是一天完成半个点。这个数字就是我们的初始的速度了。

计划

现在 story，评估，速度我们都有了，可以开始计划迭代了。计划要做的其实就是用我们当前的速度作为标准来算出在每次迭代里要做的 story。

发布计划

我们开始计划我们的第一次发布。典型的一次发布周期往往是六次迭代或者说是三个月。假设我们的团队有 5 个人。也就是说每次迭代以 50 人天为单位。六次迭代我们就可以完成了 300 人天的工作量。以目前 0.5 的速度来看，六次迭代后我们应该可以完成 150 个 story 点数。因此客户可以选择累计评估是 150 点数的 story。客户挑选出最重要的和最有效的 story。这样的一批 story 就是我们的发布计划了。

迭代计划

迭代开始的第一天都会创建一个迭代计划。一次迭代包含了 50 人天的工作量。按照当前的速度来看，一次迭代我们可以完成 25 个 story 点。因此客户必须从我们的发布计划里挑选出累计为 25 个点的 story 交给我们完成。这就是初始的迭代计划。

我们再把用户挑选的 story 细分成任务。任务跟 story 比起来就要小很多了。它是以 4－10 人小时为单位的。一个任务通常就是单个开发人员能够负责的简单的工作单元。可能是一个简单的对话框，也可能是单一的数据库事务处理，或者类似相关的事情。客户协助我们把这些任务进一步细分，告诉我们需求的细节。同时客户还协助我们平衡任务的优先顺序，指出各部分的轻重要次。协助我们识别重要和次要的用户接口。通过这样的协助，让我们的迭代始终保持具有很高的商业价值。

通常将 story 再细化到任务需要两三小时的时间。在这个期间我们可以把任务记在白板或者活动挂图上。接下来就是开发人员要在自己想负责的任务上签名了。

签字负责是一个很简单的过程。每个开发人员都要在他的头脑里面有一个预算。这个预算就是开发人员在迭代期间要花多少“ 人小时 ”在任务上。开发人员在某个任务上签字负责的时候，就在对这个任务进行评估了，并且将从自己的整体预算里减去这个评估的数字。开发人员就这样不断地在任务上签字负责，直到自己的预算花光为止。

我们并不会给开发人员分配任务，而是让他们主动签字负责。因为我们发现通过这种方式，开发人员总是能够找出最好的任务分配结果。同时，我们也会让每个挑选了任务的开发

人员对他的任务进行评估。

如果进展顺利，签字结束后，应该是每个任务都分配出去了，每个程序员的预算也花光了。不过这种情况很少见。通常，特别是第一次迭代，最后还会剩下很多任务，而且每个程序员的预算也花光了。如果这种情况出现了，开发人员就要找出一个可以分摊这些剩余任务的方法。也许有些人挑选任务的时候并没有尽力。也许这些剩下来的任务应该分配给某些人。开发人员应该反复斟酌，尽力负责更多的任务。

如果还有未决任务留下来，是时候告诉客户了，开发人员不可能全部完成 25 个 story 点。客户这个时候可以取消这些未决的任务，直至所有的任务都分配出去。

如果进展比较理想，大概中午的时候就可以完成这次计划了。

中点

.首先假设这些迭代我们要处理 20 个 story 点。从现在开始分析，设计，实现这些 story。具体怎么做留到下一部分再讲。现在把时间切换到周一的早上，也就是迭代的中点。周一的早上我们应该已经完成了 10 个 story 点了。将我们完成的 story 的评估点数累加起来就得到了完成的 story 点数。这样可以防止我们把每个 story 都处理一半就完事了。我们要做的是完成半数的 story。如果我们连 10 个 story 点数都没有完成，那说明我们远没有我们期望的那么快。假设我们只完成了 8 个 story 点数。那么我们有必要告诉客户：看来这次迭代要完成超过 16 个 story 点数的任务是不太可能了。只好请求客户取消一两个 story，直到 story 点数合计为 16 个点为止。

同样的，如果我们的完成超过半数，就可以请客户添加额外的 story 到这次迭代中来。举个例子，假设我们完成了 15 个 story 点数，那么这次迭代我们有可能能够累计完成 30 个 story 点数。所以需要让客户加多一些 story。.

速度反馈

无论是否完成所有的 story，周五下午就必须终止本次迭代。然后就是重新计算我们的速度。如果我们完成了 23 个 story 点，那么我们的速度就是一次迭代 23 个 story 点数。在接下来的下个周一，当我们做下一个迭代计划的时候，客户挑选的 story 的点数累加起来要为 23。

这就是我们怎么样去校准评估的。通过测出每次迭代完成的 story 点数，然后把这个点数作为下次迭代的目标。同样的，我们测出上一次发布完成的 story 点数作为下一次发布的

目标。个别情况下，我们也会用人小时作为单位，测出一次迭代的完成的任务的花掉的人小时的总数，然后把这个数字作为下次迭代的依据。

将迭代组织进管理各阶段

“ 统一过程 ”提出项目应该经历四个管理阶段。在初始阶段我们尽量决定系统的可行性和商业案例。在细化阶段我们决定系统的架构和创建一个比较可靠的实现计划。在构造阶段我们开始进行系统的实现。最后在移交阶段我们安装系统，和用户一起调试。

.刚才“ 统一过程 ”里涉及到的每个阶段都由一个或多个迭代组成，而且都会产生可工作的代码。从程序员的角度来看呢，这些阶段之间都没有什么区别。每个阶段都简单的由迭代组成，整体结构上也差不多。每个阶段都是在进行 story 的识别，评估，以及挑选一些出来实现。

一次迭代中包括了什么？

在两个星期的迭代里面我们完成了软件开发都会遇到的所有传统任务。也就是分析需求，设计解决方案，实现解决方案。只是在当前的迭代期间我们只考虑我们需要处理的 story。至于那些在以后的迭代里要处理的 story 我们并不关心。我们只关注当前的迭代。

.有人会抱怨这样做只会导致脆弱的架构，毫无弹性可言的设计，甚至大量重复的工作。实际上刚好相反，这样做可以形成最佳的架构，弹性十足的设计，和极少的重复工作。原因很简单，那就是我们始终都在处理最重要的特征。用户从商业价值的角度出发，挑选了每次迭代期间需要开发的特征。任何下一个阶段需要处理的功能特征，都没有当前我们正在处理的重要。因此只有能成为最重要的部分，才是我们感兴趣的。

在实现 story 的时候，我们可以效仿很多现有的实践经验。这些实践可以帮助我们写出清晰灵活的代码，降低出错率，创建灵活的系统，易于理解设计。帮助我们同客户交流，排除突发事件。

结对开发

首先，使用 dX 的时候，开发是结对进行的。两个开发人员用同一台机器一起工作，一起处理他们负责的任务。两个程序员都完全忙于写代码，他们的眼睛都关注着屏幕。无论谁负责敲打键盘，另一个家伙都知道他会做什么。键盘确实在他们之间频繁快速的切换。我甚

至看到过有个家伙负责敲键盘而另一个家伙负责用鼠标的情形。

. 我们一天换一次搭档。任务负责人保持不变，新的搭档会来和他一起工作。平均下来每个程序员会花一半的时间在自己负责的任务上，剩下的一半则用来帮助其他程序员。

你也许会认为这种划分方式会让生产效率降低一半，毕竟任务负责人只花了一半时间在自己的任务上。然而，这种所谓的生产效率的降低缺没有发生。实践证明结对开发是一种非常有效的开发软件的方式。

可验收测试

.在每次 dX 迭代开始的时候，一旦客户挑选要实现的 story 之后，客户将和 QA 的工作人员一起把 user story 充实到 use-case，并为之编写可执行的验收测试案例。当迭代进行到一半的时候，这些测试案例将递交给程序员。这些测试就是真正的需求文档。在这些测试案例里需求的细节才真正的文档化。一旦开发人员收到这些测试，他们就知道他们的 story 和任务到底要完成些什么。在整个开发过程中他们都持续的运行测试案例，以保证正确的通过测试。一次成功的迭代总是虎头蛇尾的，因为所有的验收测试都通过了，周五下午所有人早早的就下班回家了。

单元测试

在采纳 dX 的项目里，我们会编写单元测试，大量的单元测试。可以说代码未动，测试先行。而且还有个规则：如果有任何一个单元测试没有通过就不得编写新的产品代码。产品的每一行代码的都要通过特别的单元测试。

这种技术就是极限迭代。我们写下小到不超过 5－10 行的一段单元测试代码，然后编译这段代码，通常编译是通不过的，因为相应的产品代码还没有开始写呢。为了让这段测试代码可以成功编译，接下来我们应该开始着手编写产品代码，通常都不会超过 6 行。然后运行测试，因为产品代码还没有完成，所以测试又失败了。面对这样的情况，我们要做的就是充实我们的产品代码，让测试最终能够通过。一旦测试通过之后我们就可以添加新的测试片段，然后开始新的迭代。

这样的测试周期通常在 1－10 分钟之间，时间的长短还要取决于你具体使用的编程语言环境。这样的循环越快越好。每次循环你都必须运行所有针对你的模块而编写的测试代码。意思就是，无论现在进展到什么程度，几分钟前我们的代码总是可以运行的。这样不用到了最后要一次把非常多的分散的窗口和模块整合起来。大约一分钟前所有的测试都通过了。

.随着时间的流逝，单元测试也进行着增长式的积累，我们以每天几十个，一周上百个，一个月上千个的速度，不断地添加新的单元测试。为了便于运行，我们把这些单元测试组织在一起。由于不断的运行测试，对我们的代码的状态也就非常的自信。

测试同时也是文档的一种形式。如果你想知道怎样去调用一个特定的 API 的功能，那就可以参考测试代码。如果你想知道怎么样创建一个特定的对象，测试代码里也包括了这样的例子。测试可以说是一套最贴近系统编程任务的例子了。而且这种文档是明确的，准确的，可编译的，和可执行的。

重构

运行部分或者所有的单元测试或者验收测试套件，这种能力提供了一种简单的方法让我们可以判断我们所作的事情是否背离了系统。如果我们希望改变系统，可以放手去做，因为测试会提醒我们是否背离了重要的部分。

.也就是说我们可以用最小的代价来做变更。如果发现一个变量应该用一个更好的名字来代替，那动手修改就是了。如果一个类里面方法太多了，那就拆分开来。如果发现一个类里面有不适合的方法，那就移到适合的类里面去。只要用大量的单元测试和验收测试作靠山，我们就可以放心大胆的去修改我们认为应该修改的任何部分。

在不改变程序的行为的情况下改进内部结构就是重构 ,在 dX 实践的过程中我们总会使用这种技术——重构。每小时左右的编程结束后，接下来就是进行重构。回顾写下的代码并改进它。这种改进可以说是小碎步式的，几分钟一次，紧跟着就是测试。通过逐渐的改进系统结构，保持系统的先进性。这种规则也就是：***绝不能把烂代码留到第二天***。

开放式办公环境

最适合 dX 的环境就是开放式的办公环境，实验室也可以。把桌子和电脑摆在中间，这样方便结对编程。所有的工作都在一个房间里进行，即使是客户也不例外。我们的目标就是真正象一个团队一样的工作，成员之间有着非常频繁的交流和沟通，这样的环境下，可以快速的互相提问，得到最快的响应和建议，做到互相依靠，互看代码。

持续集成

在 dX 里面代码控制系统是开放式的，也就是说不管有没有其他人已经 check out 过某个模块，任何人都可以再次 check out。只要手脚够快，谁都可以 check in。而第二个 check

in 的就要负责合并。

这种合并的可能性导致了一种有趣的东西，那就是压力。你越是 check out 的时间越长，那你越有可能要做合并。没有人喜欢干这个活，不管你用的工具多强大的支持这种功能。因此有种无形的压力促使你频繁而快速的做 check in。这是好事情。

然而，dX 实践也有专门针对 check in 的规则。那就是任何 check in 都是以通过单元测试和验收测试作为前提的。也就是说你必须把自己所做的修改集成到整个系统中，创建并测试系统，如果成功了，你就可以 check in 了。

在一个使用 dX 的项目里面，这种情况在每一对结对的开发人员之间时刻都在上演。因此集成做到了持续性。从来不会说项目都进展到尾声了，才来一个非常庞大的终极集成。

小结

哎呀，我肯定忘了提 UML 了。嘿嘿，我才不会忘呢。同样地，我连 JAVA 都一字未提呢。主要还是因为这些工具跟 dX 实践没什么关系。UML 不是一个方法，而是一种图示的标记。JAVA 也不是一个方法，不过是一种编程的语言。只有需要的时候我们才会使用它们。它们并不是我们想要遵循的方法，而是一种使用工具。有人要问了，那么文档呢？难道我们不需要用 UML 来创建一些像样的实用的文档吗？当然需要了。但是这并不是过程的一部分。为了文档而文档一点用都没有。在 dX 里有一个规则，也就是家喻户晓的 Martin 文档第一要律：只编写那些你真正需要的有意义的文档。这就是为什么我们不认为画 UML 图是理所当然的了。不需要专门用类图来捕获所有的需求，同样，也不需要在 sequence 图里捕获所有的 use-case。只有真正紧急需要使用这些工具的时候才使用，否则就让它们一边歇着去。

.在一个使用 dX 的项目里面，你常常会看到有人在使用 UML。在互相争论不同的设计思路的时候，他们会在白板上画出草图。也会看到他们书面化一些 UML 图，其他人就按照这个来做。因为 dX 过程需要他们那样做，但是这是不为你所察觉的。当他们确定需要这样做的时候才会画这些 UML 图。

以上我阐述的 dX 实践其实源自 XP(极限编程)实践。如果你观察够仔细的话，你会发现把 XP 倒转过来就是 dX 了，哈哈。

参考文献

[Kruchten1998]: The Rational Unified Process, Philippe Kruchten, Addison Wesley,

Reading, MA, 1998.

[Fowler1999]: Refactoring, Martin Fowler, Addison Wesley, Reading, MA, 1999

[Martin1999]: RUP vs. XP, Robert C. Martin, 1999,

http://www.objectmentor.com/resources/articles/RUPvsXP.pdf

[Beck1999]: Extreme Programming Explained, Kent Beck, Addison Wesley, Reading,

MA, 1999

[Jeffries2000]: Extreme Programming Installed, Ron Jeffries, et. al, Addison Wesley, Upper Saddle River, NJ, 2000

6. [Beck1999], [Jeffries2000]

第八章　包(Packages)

——感谢 Orient Sun 对翻译本章的贡献！

对 Java 程序员来说有两类重要的包。第一种是源代码包，它由 Java 的 **package** 关键字表示；第二种包是二进制组件，用.jar 文件表示。

Java Packages

Java packages 是**名称空间**(namespaces)，它们允许程序员创建小的私有区域，以便在其中声明类。这些类的名字不会与其他包中同名的类发生冲突。

Java 编译系统常常保持生成的二进制.class 文件的目录结构与其源代码的包结构相同。这样 class A.B.C 的.class 文件将可能以 A/B/C.class 的路径存储在磁盘上。因为 Java 编译器从.class 文件而不是.java 文件中读取外来的声明，因此让编译器和运行时系统知道应用程序中包的恰当 classpath 是至关重要的。

由于这些问题的存在，所以要对系统的包结构给予足够的重视。UML 提供了相应的工具帮助处理这样问题。

Packages

在 UML 中有几种不同的方式表示一个包，Figure 8-1 显示了一种最简单的形式。包的图标仅仅是一个矩形框，在其顶部有一个标签，看起来有点像一个文件夹。完整的包名显示在矩形框中间。

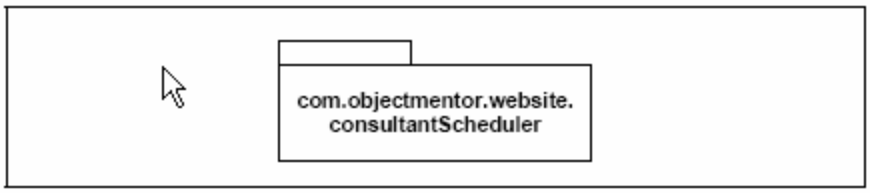


Figure 8-1
Simple UML Package

如果你喜欢也可以将包名放在矩形框的标签上，剩下大的矩形框用于列出包中定义的所有类。Figure 8-2 显示了这种包的形式。

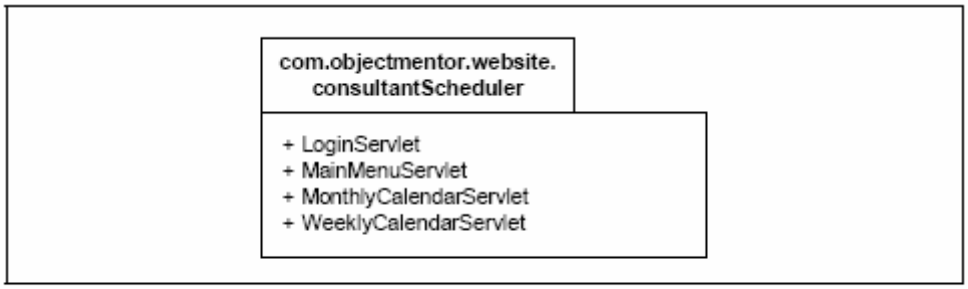


Figure 8-2
UML package showing contents.

最后，你可以用包含关系（*contains*）显示包的嵌套结构，如 Figure 8-3 所示。

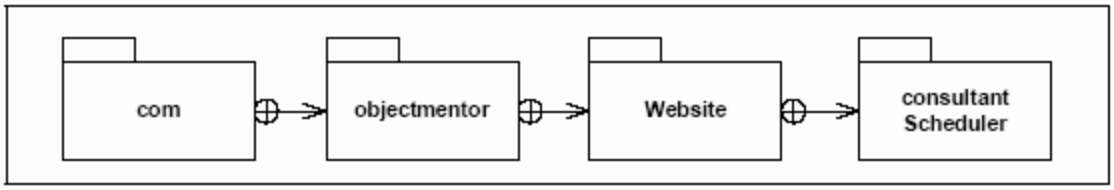


Figure 8-3

依赖（Dependencies）

在同一个包中的代码常常依赖另一个包中的代码。在 Java 中，当我们 import 一个类或者一组类时可以看到这种情况，如果我们使用类的完整限定名也可以了解到这种情况。在 UML 中我们用依赖关系（*dependency*）表示这种依赖性，如 Figure 8-4 所示。

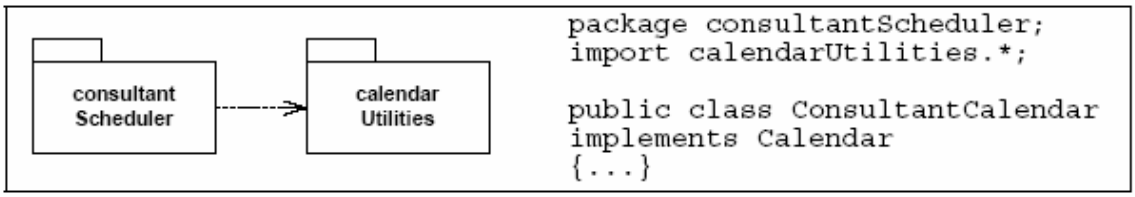


Figure 8-4
Package Dependency

请注意依赖关系不是 import 语句的结果，而是因为在 consultant-Scheduler.ConsultantCalendar 中实际用到了类 calendarUtilities.Calendar 而导致的。在 Java 中，import 并不会创建真正的依赖关系，尽管编译器在发现被引入的类或者包不存在时会发出抱怨。

二进制组件.jar 文件（Binary Components）

虽然包作为源代码分组的手段很方便，但用于二进制代码的分组就可能不太方便。我们常希望就很多小的二进制代码块打成一个.jar 文件的组件包中，这样组件就能很方便地发布到它的执行环境中。

组件在 UML 中用 Figure 8-5 的形式表示，上方的接口图标是可选的。图中表示在 CalendarRenderer 组件中有一个名为 Calendar 的接口。

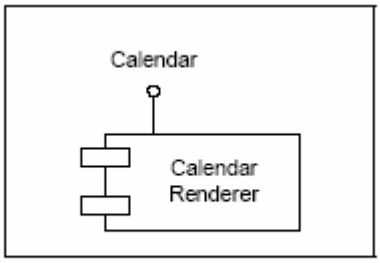


Figure 8-5
A component -- possibly a `.jar` file.

就象包一样，组件之间也存在依赖关系。事实上，一个组件常常包含一个或者多个包，所以组件之间的依赖关系常是包之间依赖关系的子集。

包设计的原则（Principles of Package Design）

经过多年的实践，我开始依靠一套简单的原则来帮助我组织大型软件应用的结构。这些原则并不是强制的规则，甚至可能不是完成正确的方法。但是，它们是简单的启发式方法，可以帮助我在进行系统划分时进行平衡。

你会发现这些原则并不会引导你进行功能分解。依据这些原则创建的包会将易变的类放在一起；会将因为各种原因需要被改变的类分离开；它们也试图将经常改变的类和不常变化的类独立开；并试图分离系统的高层结构和低层实现细节，以保持高层结构的独立。

这些原则的详细讨论包含于[PPP2002]，在此我只作简单的介绍。

发布/重用等价原则（The Release/Reuse Equivalency Principle ）(REP)

人们一般不重用单个类，他们常常需要重用一组相关的类。这些被重用的一组类应该放到一个包中，然后这个包被那些希望重用他们的开发人员发布和跟踪。

这个原则说明了将类放到包中的标准之一是：创建一个包是为了方便别人重用。

被其他人重用的包，其开发者必须更加仔细对待它。如果开发者企图对包进行修改他应该通知使用这个包的其他人。开发者也应该考虑是否对以前老版本进行一段时间的维护，这可以给重用这个包的人有机会逐步改变他的系统。这就意味着需要相当高的成本，很难应用于单个类上。这样的话，创建的可重用类就对其开发者和其他重用它的人都很方便。

因此，你能够重用的最小的东西应该要是其他人愿意努力去发布和跟踪的。也就是说重

用的粒度就是发布的粒度。

公共闭合原则（The Common Closure Principle） (CCP)

单一责任原则（SRP）告诉我们当改变一个类时应该给出一个且仅有一个原因。CCP 将这个原则扩展到包上。我们希望在一个包中的所有类应该在需要修改的情况方面是基本相似的。我们希望将修改限制于同一个包内。

大多数系统由很多包组成，有时需要成十上百个包。而这些包之间互相依赖最终产生很复杂的依赖关系。CCP 原则的目标是按预期的修改将类分组，那些因为相同原因要被修改的类被放在一个包中。这样，当要进行修改时所涉及的包会较少。

公共重用原则（The Common Reuse Principle） (CRP)

ISP 原则告诉我们为类的每个客户创建相应的接口。CRP 原则将这规则应用于包。一个有很多客户的包需要对所有用到它的包负责，对这个包上任何改变将对依赖它的所有包产生很大的冲击。因此，应该尽可能地将只被一个客户使用的包与被多个不同客户用到的包分开。当一个包中包含被多个不同客户使用到的类时，对其中某个类的修改将影响到其他没有使用该类的客户。事实上因为这个包的修改可能导致客户的包需要重新编译和发布。

非循环依赖原则（The Acyclic Dependencies Principle） (ADP)

包的循环依赖将导致编译和开发方面的问题。当存在循环依赖时，就不可能判断出哪个类或者包应该先编译，哪些应该在其后编译。

依赖性是传递的，如果包 A 依赖于包 B，包 B 又依赖于包 C，那么包 A 也就依赖于包 C。这意味着当包依赖图中存在一个循环时，在这个循环上的每个包将陷入循环依赖状态。如此复杂的交叉依赖图将很难保持包之间的独立，其结果是开发人员将不可能在对这些包进行工作时不影响其他的开发人员。

解决的办法是不要在依赖图中出现循环依赖。这个工作可以手工进行，也可以利用象 JDepend (see www.clarkware.com)之类的工具。

稳定依赖原则（The Stable Dependencies Principle） (SDP)

对有些包进行改变很容易,但另外一些包因为有多个包依赖于它所以难以进行改变。如果一个包有多个指向它的依赖，而它自己又依赖于一个易发生改变的包，其结果就导致这个

包很难改变。请看 Figure 8-6。

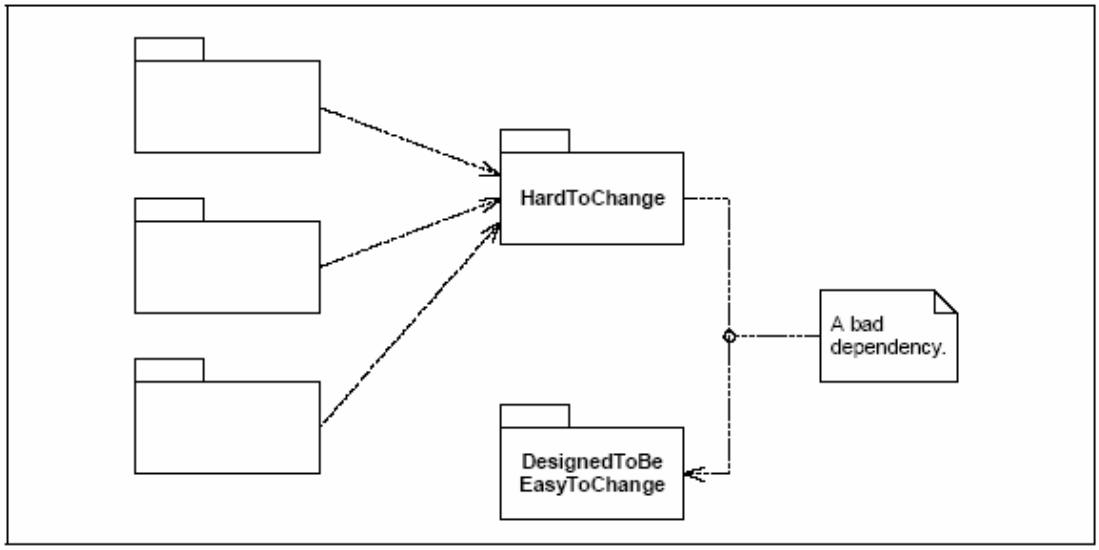


Figure 8-6
A violation of the SDP.

稳定依赖原则是说包应该不依赖于那些比它们自己更不稳定(更易改变)的包。每个包所依赖的对象比依赖于它们的包更难修改。

在[PPP2002]中讨论了一些简单的办法，开发团对可以采用这些方法来计算每个包的稳定性，以此评价依赖性是否在朝稳定性方向发展。

稳定抽象原则（The Stable Abstractions Principle） (SAP)

既然稳定的包难以改变，那么我们需要一种方法来保持它们的灵活性。OCP 原则告诉我们可以不用修改而达到扩充的目的。SAP 原则就说为了保证稳定的包易被控制，稳定包应该是抽象的。越稳定的包应该越抽象。

一个包的抽象程度取决于它包含的抽象类和接口的数量。 抽象类和接口的比例越大包就越抽象。按照 SAP 原则：一个包如果有多个指向它的依赖就是很稳定的，因而应该是抽象的。

SDP 原则和 SAP 原则相结合就成了包版本的 DIP 原则。DIP 原则告诉我们类的关系应该瞄准抽象类和接口。结合 SDP 和 SAP，可以得出：稳定性随指向它的依赖的增多而增加；抽象应该随稳定性的增加而增加，因此包的抽象性应该随指向它的依赖的增多而增加。

[PPP2002]同样提出了一套测量包抽象程度的方法，以用于控制稳定性和抽象性之间的关系。

小结

画包和组件图有多重要呢？显然它们是相当有用的图。ADP 原则告诉我们包和组件的循环依赖是有问题的，需要解决这些循环问题。画出结构图常有助于解决循环依赖问题。

包依赖图对显示包被编译的顺序也是相当有用的。用错误的顺序编译包会导致奇怪的创建问题（build problems）。依赖图可以告诉你哪些包依赖于其他包，哪些应该首先被编译。

当然，创建这些图的最好方法是从代码生成。关于包结构的图如果没有包含存在于代码中的每种依赖关系，它们对于解决循环问题和判断编译的顺序就不会特别有用。因此，利用工具判断依赖关系或者创建依赖图是个不错的想法，至少可以利用工具提供一张依赖关系的列表以帮助你创建自己的图 [2]。

---

2. see www.clarkware.com for just such a tool.

第九章　对象图（Object Diagrams）

——感谢 Orient Sun 对翻译本章的贡献！

有些时候，将系统某一特定时刻的状态表示出来是很有用的，UML 对象图就像系统运行时的一个快照(Snapshot)，显示了给定实例的对象、关系和属性值。

快照

不久前，我曾参与过一个应用程序的开发，这个应用程序允许用户在图形用户界面(GUI)上画建筑物的平面布置图。程序记录了房间、门、窗户和开了口子的墙，其数据结构如 Figure 9-1，这个图能够向你说明了可能会使用哪些类型的数据结构，但是无法告诉你在某个特定的运行时刻有什么对象和关系被实例化了。

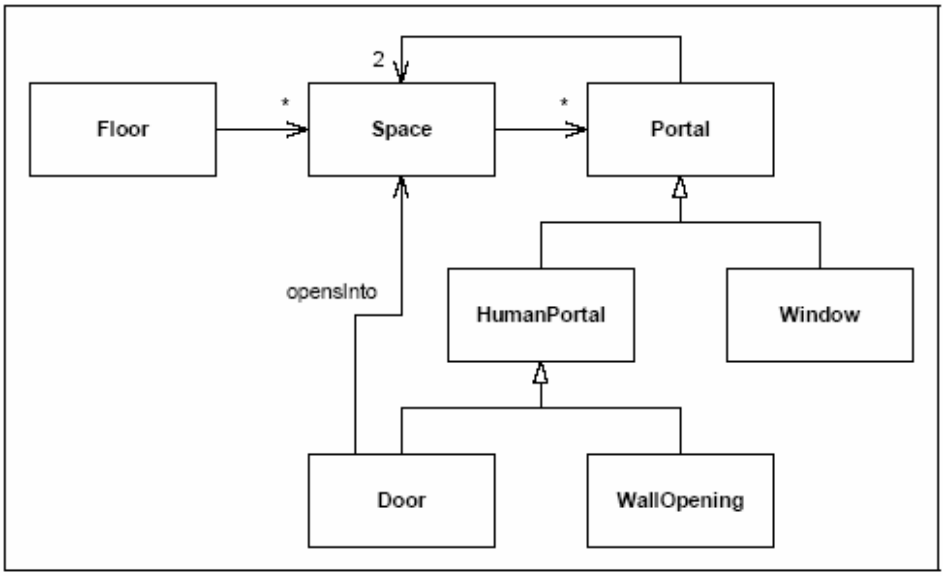


Figure 9-1
Floorplan

设想一下：我们程序的一个用户画了两个房间，一个厨房、一个餐厅，它们之间用一堵墙相连着。厨房和餐厅各有一个向外的窗户，餐厅还有一个向外开的门，门是打开着的。这个场景就可以用图 Figure 9-2 的对象图进行描叙。

Figure 9-1 显示了当时存在于系统中的对象，以及它们被哪些对象关联着。它显示厨房和餐厅是 Space 类的两个独立实例；显示了这两个房间是如何通过墙相连；显示外部空间实际上是用 Space 类另外实例表示；还显示了所有其他必须存在的对象和关系。

当你需要显示系统在某个特定时刻，或者系统处于某个特定状态时的内部结构是什么样

时，对象图是很有用的。对象图显示了设计者的意图。它显示了对象和关系被实际使用的方式，它也有助于显示系统根据不同的输入是如何变化的。

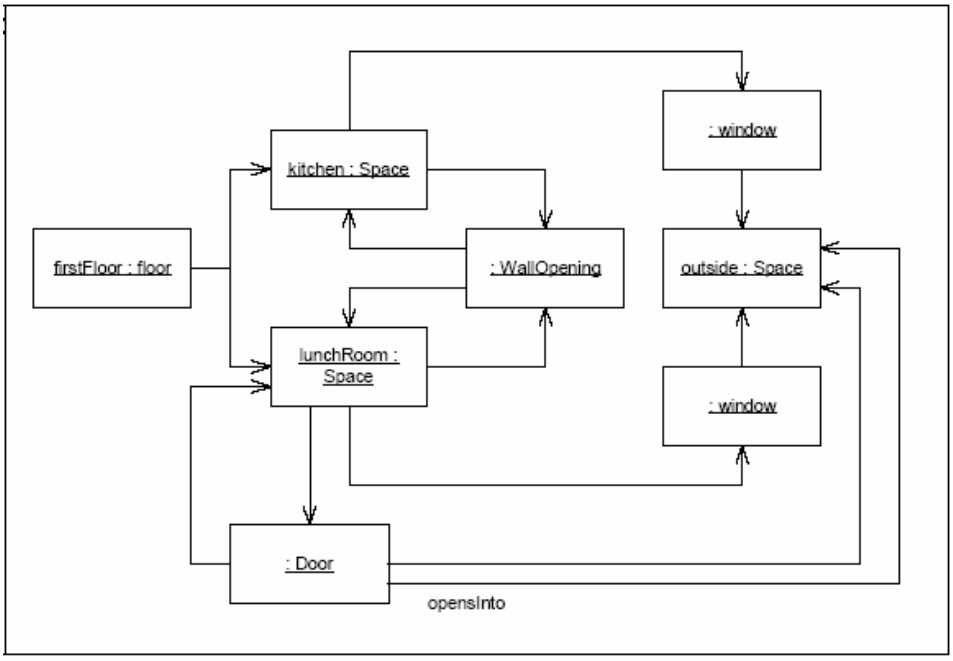


Figure 9-2
Lunch room and Kitchen

需要注意的是不要滥用对象图。在过去十年里我总共画了不到十张这样的对象图，对它们的需要也并没有频繁地增涨。我之所以将对象图包含在本书中，是因为当需要它们时，它们是必不可少的。可是，你并不是经常需要它们，为系统的每个场景，甚至每个系统都画对象图是件不可想象的事情！

主动对象（Active Objects）

对象图的另一个有用之处是在多线程系统中。考虑一下在 **Listing 9-1** 例子中的 SocketService 代码，这个程序实现一个简单的框架让你编写 socket 服务器程序而不需要处理与 socket 相连的线程和同步问题。

Listing 9-1
SocketService.java

```
import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.LinkedList;

public class SocketService {
  private ServerSocket serverSocket = null;
  private Thread serviceThread = null;
  private boolean running = false;
  private SocketServer itsService = null;
  private LinkedList threads = new LinkedList();
```

```
public class SocketService {
  private ServerSocket serverSocket = null;
  private Thread serviceThread = null;
  private boolean running = false;
  private SocketServer itsService = null;
  private LinkedList threads = new LinkedList();

  public SocketService(
         int port, SocketServer service) throws Exception {
    itsService = service;
    serverSocket = new ServerSocket(port);
    serviceThread = new Thread(
      new Runnable() {
        public void run() {
          serviceThread();
        }
      }
    );
    serviceThread.start();
  }

  public void close() throws Exception {
    running = false;
    serviceThread.interrupt();
    serverSocket.close();
    serviceThread.join();
    waitForServerThreads();
  }

  private void serviceThread() {
    running = true;
    while (running) {
      try {
        Socket s = serverSocket.accept();
        startServerThread(s);
      }
      catch (IOException e) {
      }
    }
  }

  private void startServerThread(Socket s) {
    Thread serverThread = new Thread(new ServerRunner(s));
    synchronized (threads) {
      threads.add(serverThread);
    }
    serverThread.start();
  }

  private void
  waitForServerThreads() throws InterruptedException {
    while (threads.size() > 0) {
      Thread t;
      synchronized (threads) {
        t = (Thread) threads.getFirst();
      }
      t.join();
    }
  }

  private class ServerRunner implements Runnable {
    private Socket itsSocket;

    ServerRunner(Socket s) {
      itsSocket = s;
    }
```

```
        public void run() {
          try {
            itsService.serve(itsSocket);
            synchronized (threads) {
              threads.remove(Thread.currentThread());
            }
            itsSocket.close();
          }
          catch (IOException e) {
          }
        }
      }
    }
```

以上代码的类图显示在 Figure 9-3 中，结果并不令人鼓舞。从类图上难以看出代码的意图，它可以较好地显示类和关系，但是其他一些情况就无法看出了。

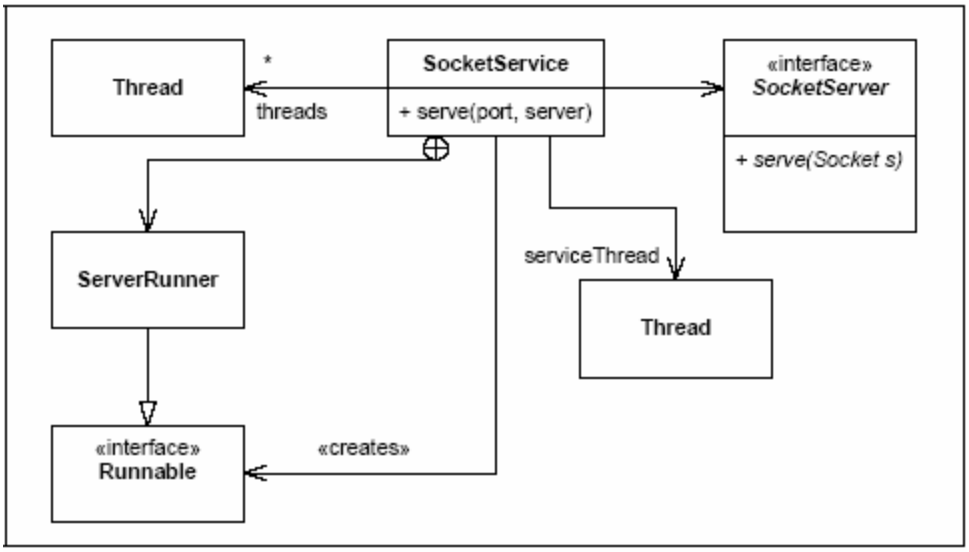


Figure 9-3
SocketService class diagram.

但是，看一下 Figure 9-4 的对象图，它比类图更好地显示了系统结构，它表明 SocketService 由 serviceThread 控制，serviceThread 运行在一个匿名内嵌类中，同时也表明由 serviceThread 负责创建所有的 ServerRunner 实例。

请注意图中环绕线程实例上的粗线条，这些有粗线条框的对象称为**主动对象**(Active Ojbect)。主动对象就像是控制线程的控制者，它们包含有控制线程的方法，如 start、stop、setPriority 等。图中的所有主动对象都是 Thread 类的实例，因为它们的线程控制处理都是派生于 Runnable 接口，所有的线程实例都持有到 Runnable 的引用。Runnable 的派生对象本身并不是主动的，因为它们不控制线程，而是由线程调用它们。

为什么在这里对象图比类图更具有表达力呢？是因为以上程序的结构是在运行时建立的，这是一种是关于对象的结构，而不是关于类的结构。

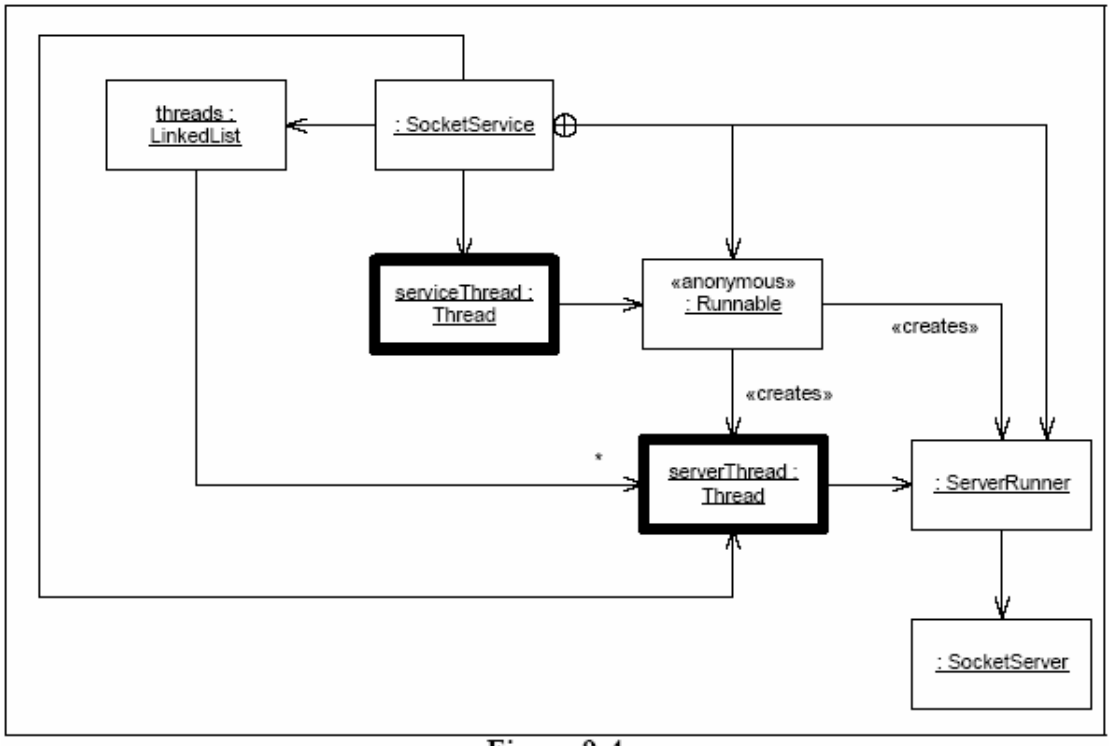


Figure 9-4
SocketService Object Diagram

小结

对象图向你显示了在特定时刻系统状态的一个快照。这是一种描叙系统的有用的方法，特别是当系统结构是动态建立而不是由它类的静态结构表示时。但是应该警惕不要画过多的对象图，很多对象图能够由类图直接推断出来，因此它们应用得比较少。

第十章　状态图(State Diagrams)

——感谢 LiShiFeng 对翻译本章的贡献！

UML 有丰富的符号，用来描述有限状态机（FSMs)。在本章节你将看到很多有用的符号。在写各种软件的时候 FSMs 是非常有用的工具。我在编写图形用户界面(GUI)、通讯协议和其它基于事件的系统中使用他们。可惜的是，我发现好多的开发人员不熟悉 FSMs 的概念，因而失去了好多使工作简单化的机会。在本章节，我将尽我的所能来说明 FSMs。

基础知识

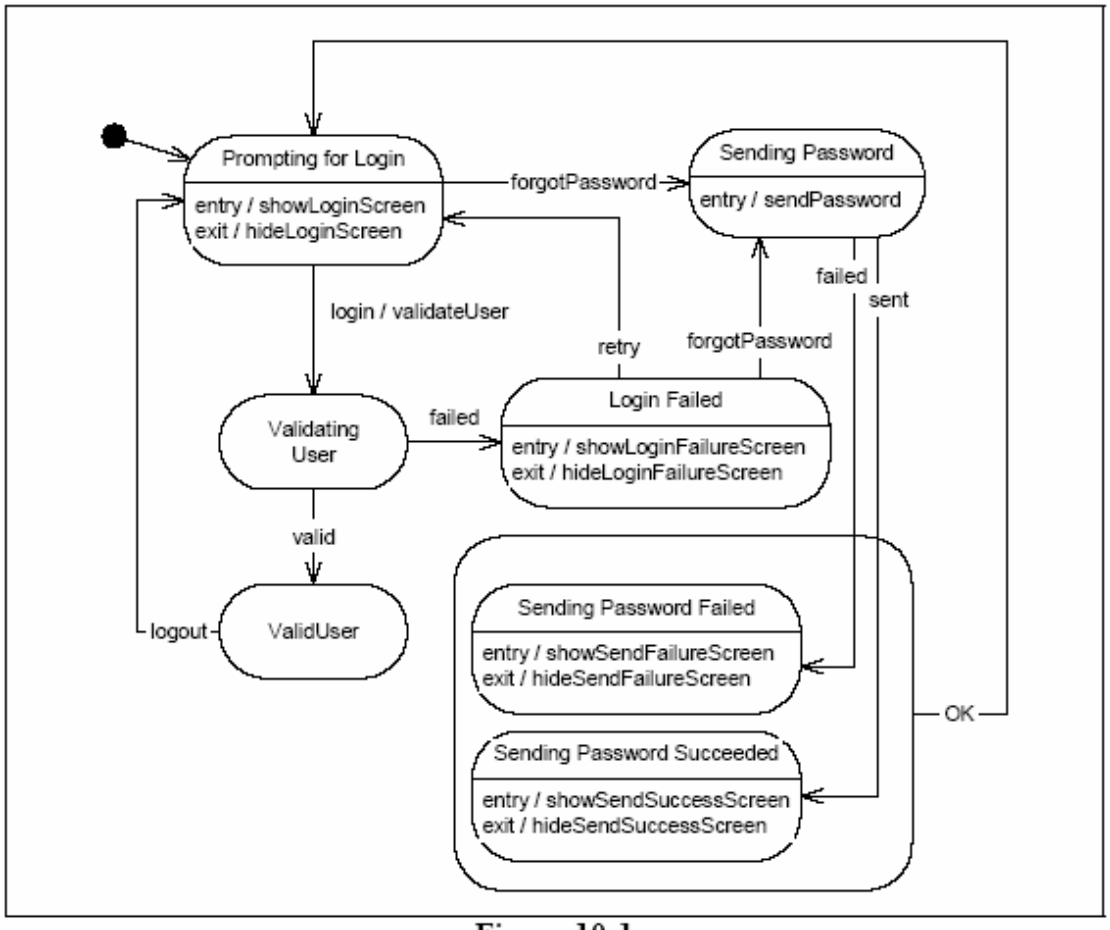


Figure 10-1
Simple Login State Machine

Figure 10-1 显示了一个简单的***状态转换图（****State Transition Diagram* (STD)）的例子。这个例子描述了如何通过有限状态机控制一个用户登录到一个系统。图中的圆形拐角矩形表示的是状态。每一个状态的名字，在状态图上面的框格中。在状态图的下面的框格中，是专

门用来告诉我们，当进入或者退出这个状态时触发的动作。例如：当我们开始进入“ Prompting for Login ”这个状态的时候，我们调用“ showLoginScreen ”动作；当我们退出“ Prompting for Login ”这个状态的时候，我们调用“ hideLoginScreen ”这个动作。

箭头表示的是状态之间的***转换***(Transitions)（译者注：箭头指向表示的是从原状态到目标状态），触发转换的每一个事件的名称都被写在标签上。当转换被触发的时候，执行的动作也记录在标签上面。例如：假如我在“ Prompting for Login ”状态中，并且获得了“ login ”这个事件，那么我们将转换到“ Validating User ”状态，并调用“ validateUser ”这个动作。

在图 Figure 10-1 的左上角有一个实心的黑色圆圈，它叫做***初始伪状态***（*initial pseudo state*）。有限状态机的生命周期，是从这个初始状态进行转换开始的。因而，我们这个例子是从“ Prompting for Login ”这个状态开始进行转换的。

我画了一个***超状态***（*superstate*）里面包括了“ Sending Password Failed ”和“ Sending Password Succeeded ”这两个状态，因为这两个状态都会与“ OK ”事件作用而转换到“ Prompting for Login ”状态，因此我不想画两个这样的箭头指向同一个状态，所以我使用了更简单的一个超状态进行描述。

这个有限状态机图示，使登录这个功能很清晰明了，它将这个过程分成一些小的功能。如果我们实现图中的所有动作，例如“ showLoginScreen ”、“ validateUser ”、和“ dsendPassword ”以及按表示逻辑的箭头连线，我确信这个 Login 的过程是可以工作的。

专用事件

在状态图的下面部分包括了一对事件和动作（译者注：事件和动作书写形式为 event / action）。图 Figure 10-2 中的 entry 和 exit 事件是标准的，但是就像在图 Figure 10-2 中显示的，如果你喜欢你也可以添加自己的事件。当 FSM 进入这个状态的时候，事件中的一个就会被触发，与此同时相应的动作就会被调用。

Figure 10-2
States in UML

在使用 UML 之前，如图 Figure 10-3 所示我通常习惯使用一个状态自己指向自己的箭

头来表示该状态的专用事件。但是，在使用 UML 中，图 Figure 10-3 有不同的含义。任何一个退出状态的转换，都将会调用 exit 动作。同样的，任何一个进入状态的转换都将会调用 entry 动作。因而，在使用 UML 中，如图 Figure 10-3 所示，一个自身的转换，不仅调用 myAction 动作，同时也调用 exit 和 entry 动作。

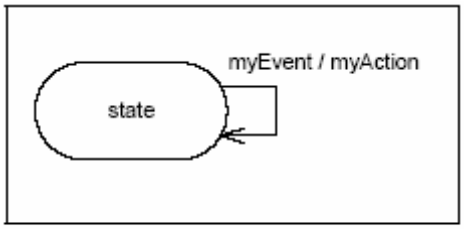


Figure 10-3
Reflexive Transition

超状态

如同你在图 Figure 10-1 中登陆的 FSM 中曾看到的那么样，当你有多个状态要用相同的方式响应某些相同事件的时候，超状态此时使用起来就方便多了。你可以画一个超状态把这些类似的状态括起来，只需要从超状态画一个转换箭头，来代替从每一个状态画转换箭头。因此，在图 Figure 10-4 中的两种表示是相同的。

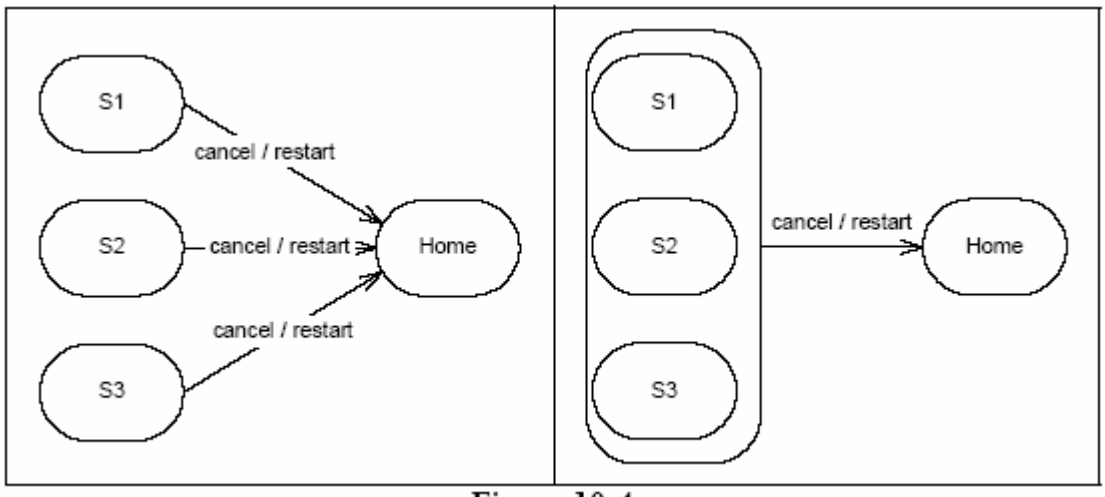


Figure 10-4

超状态的转换有可能被从子状态发出的明确转换重载。因而，从图 Figure 10-5 中可以看到 S3 的 “ pause ” 转换重载了默认的 Cancelable 这个超状态的 “ pause ” 转换。在意义上讲，超状态相当于一个基类（编者注：基类也叫父类）子状态重载他们超状态的转换，就像子类重载它们基类的方法一样。但是用这个比喻有一点夸张，不是很确切。超状态和子状态之间是没有继承的关系。

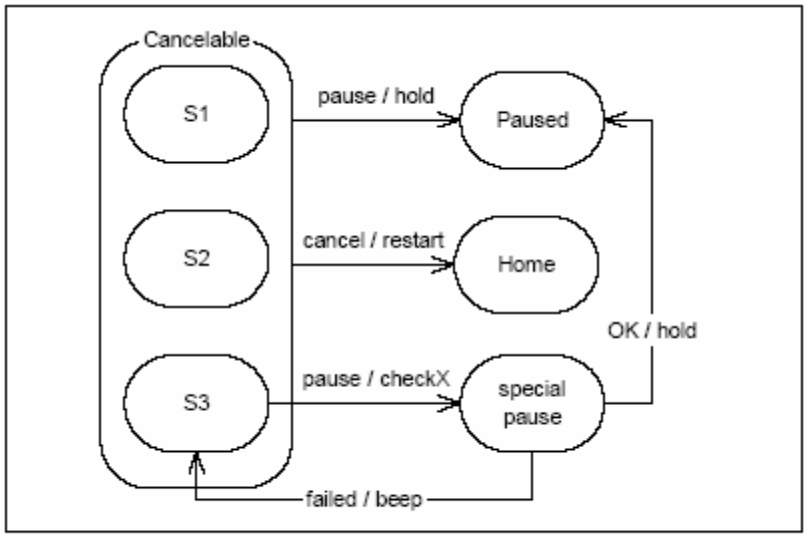


Figure 10-5
Overriding superstate transitions

超状态和一般的状态一样有 entry、exit 和专有事件。图 Figure 10-6 显示了一个 FSM 的 exit 和 entry 事件在超状态和子状态中的动作。当 FSM 的转换从“ Some State ”进入“ Sub ”，它首先调用“ enterSuper ”动作，随后调用“ enterSub ”动作。同样地，假如这个 FSM 从 Sub2 状态回到‘ Some State ”，它首先调用‘ exitSub2 ’动作，然后调用‘ exitSuper ”动作。然而，从“ Sub ”状态到“ Sub2 ”状态的“ e2 ”转换，没有退出这个超状态，所以它只调用“ exitSub ”和“ enterSub2 ”的动作。

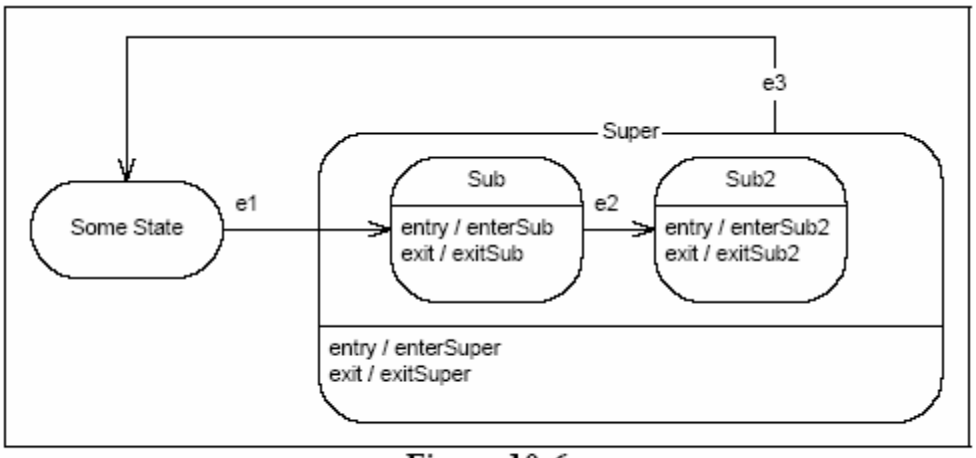


Figure 10-6
Hierarchical invocation of Entry and Exit actions.

初始伪状态和结束伪状态

如图 Figure 10-7 显示了通常在 UML 中使用的伪状态。通过初始伪状态 FMS 进入一个真实的转换过程。因为这个事件是在创建状态机，所以通过初始伪状态开始的转换，初始伪状态不需要有一个事件。不过初始伪状态，它可以有一个动作，这个动作将是在 FSM 被创

建以后，第一个被调用的动作。

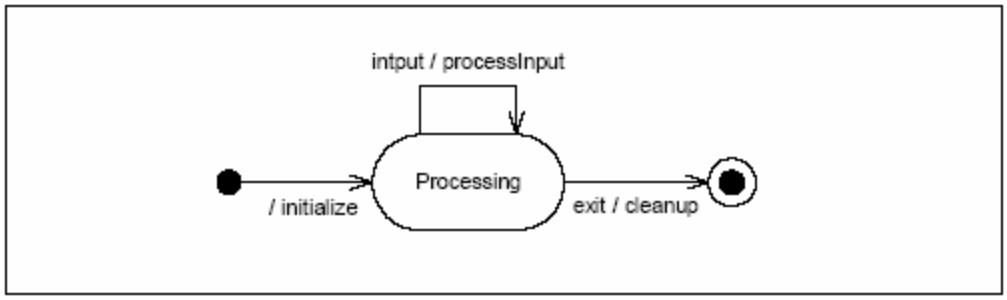


Figure 10-7
Initial and Final Pseudo states

同样的，当转换过程进入结束伪状态时，一个 FSM 就结束了。这个结束伪状态实际上不会实现。假如在转换到结束伪状态的时候，有一个动作，这个动作将会是 FSM 最后执行的一个动作。

有限状态机图的使用

我发现，象这样的图，对于子系统的状态机有很好的使用，而且也能够被人们很好的理解。另一方面，使用 FSMs 的大多数系统的行为，没有被人们很好的理解。随着时间的推移，很多系统的行为在不断的增长和变化，对于肯定要频繁变更的系统，使用图进行描述不是很有益的。布局和空间的问题影响了图表的内容。这种影响有时可能要妨碍设计师对设计的一些必要的修改。图表版面的重排，使设计师他们不能再增加一个类或者一些状态，并使他们使用一个不影响图解布局的一个低标准的解决办法。

从另一方面来说，文本则是一种可以非常灵活的处理变更的介质。布局的问题降至最低，而且总有地方可以新增一行文本。因此，对于日趋完善中的系统，我在文本中创建 STTs（*State Transition Table*s）要好于使用 STDs（*State Transition Diagrams*）。考虑一下在图 Figure 10-8 中所示的地铁十字转门（译者注：地铁验票设施），它能很简单地使用 STT 来表示，如图 Figure 10-9。

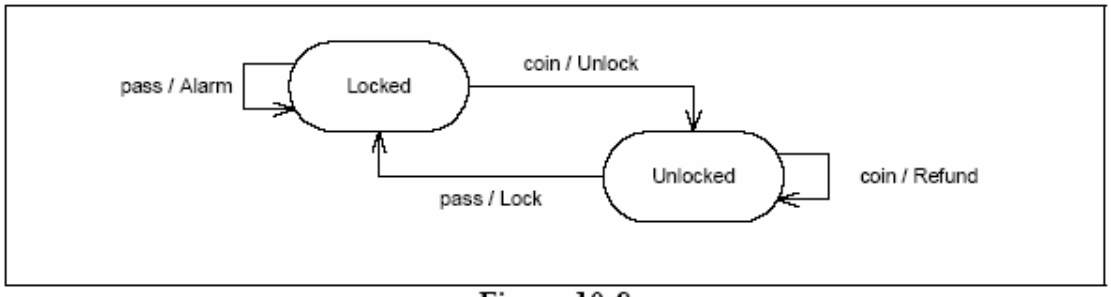


Figure 10-8
Subway Turnstile STD

Current State	Event	New State	Action
Locked	coin	Unlocked	Unlock
Locked	pass	Locked	Alarm
Unlocked	coin	Unlocked	Refund
Unlocked	pass	Locked	Lock

Figure 10-9
Subway Turnstile STT

STT 是一个有四列的简单表格。表格的每一行表示一个转换。从图上看每一个转换的箭头符号，你将会看到，表格的一行包含了箭头的两个终点。分别表示箭头的事件和动作。通过下面的模板语句来读 STT：“假如我们在 Locked 状态，并且我们得到了一个 coin 事件，那么我们就会达到 Unlocked 状态并且执行 Unlock 方法。

这个表格的内容可以很容易的转换成一个文本文件：

Locked coin Unlocked Unlock

Locked pass Locked Alarm

Unlocked coin Unlocked Refund

Unlocked pass Locked Lock

这 16 个单词包括了这个有限状态机的所有的逻辑。可以写一个简单的编译器来读取这个文件，并且自动生成代码来实现这些逻辑。

SMC

因此，在 15 年前，我写了一个简单的编译器，名字叫SMC，用来读取STT并且自动生成C++的代码来实现STT中的逻辑。随后，SMC不断的完善，现在已经可以支持不同的编程语言。SMC可以从www.objectmentor.com资源栏中免费得到可用的版本。

Turnstile 输入 SMC 的文件如 Listing 10-1 所示。大多数的语法是很容易理解的。详细的内容在 smc.txt 文件中，你可以从前面提到的 URL 中下载。FSMName 表示的是 SMC 将要产生的类的名字。Context 用来表示 SMC 所产生的类需要继承的类的名字。

由此输入所产生的这个代码的列表如 Listing 10-2 所示。它利用了状态模式。代码产生以后，这个代码不再需要进行编辑和校验。它只是实现了逻辑，使完成动作的方法在 Context 类中得以实现。

Listing 10-1
Turnstile.sm

```
Context TurnStileContext
```

```
FSMName TurnStile
Initial Locked
{
    Locked
    {
        Coin Unlocked Unlock
        Pass Locked Alarm
    }
    Unlocked
    {
        Coin Unlocked Thankyou
        Pass Locked Lock
    }
}
```

Listing 10-2
TurnStile.java (Generated)

```
public class TurnStile extends TurnStileContext {
    private State itsState;
    private static String itsVersion = "";
    private static Locked itsLockedState;
    private static Unlocked itsUnlockedState;

    public TurnStile() {
        itsLockedState = new Locked();
        itsUnlockedState = new Unlocked();
        itsState = itsLockedState;
    }

    public String getVersion() {
        return itsVersion;
    }

    public String getCurrentStateName() {
        return itsState.stateName();
    }

    public void Pass() {
        itsState.pass();
    }

    public void Coin() {
        itsState.coin();
```

```
    }

    private abstract class State {
        public abstract String stateName();
        public void pass() {
            FSMError("Pass", itsState.stateName());
        }
        public void coin() {
            FSMError("Coin", itsState.stateName());
        }
    }

    private class Locked extends State {
        public String stateName() {
            return "Locked";
        }
        public void pass() {
            Alarm();
            itsState = itsLockedState;
        }
        public void coin() {
            Unlock();
            itsState = itsUnlockedState;
        }
    }

    private class Unlocked extends State {
        public String stateName() {
            return "Unlocked";
        }
        public void pass() {
            Lock();
            itsState = itsLockedState;
        }
        public void coin() {
            Thankyou();
            itsState = itsUnlockedState;
        }
    }
}
```

使用这种方法创建和维护有限状态机，比维护图表要简单多了，并且通过生成代码也节省了大量的时间。因此，虽然图表在帮助你或者让其它人理解 FSM 方面可能是非常有用的，但是该文本形式的表格用于开发更加方便。

ICE：案例研究

几年以前，我参与了一个工作站的项目，名字叫 ICE。使用者，通过屏幕上的用户界面，完成一连串的简单的工作流程工作。这个 GUI 的逻辑如图 Figure 10-10 显示。这个图不是在项目开发中画出的。我在此画这个图只是告诉大家，如何把复杂的 FSM 用 UML 画出来。

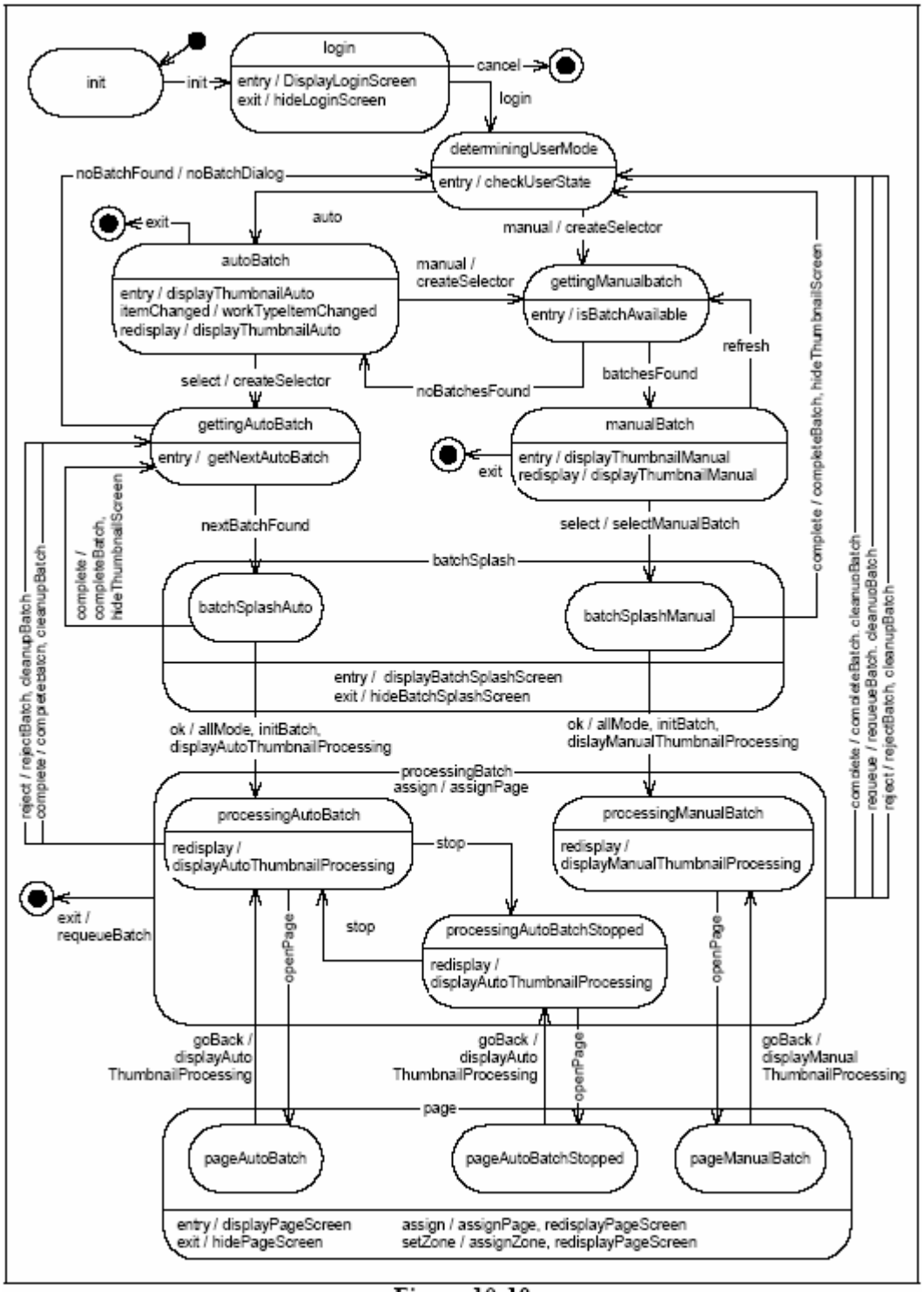


Figure 10-10
ICE FSM

这个图 SMC 输入的文件如 Listing 10-3 所示。这个源码的文件，从粗糙的一开始，一

直演变到你现在看到的完整的描述。这个文件很容易创建和维护，并且非常好的适合我们编译的程序。

Listing 10-3
ice.sm

```
Context RootFSM
Initial init
FSMName RootFSMGen
Version 042399 1528 rcm
FSMGenerator smc.generator.java.SMJavaGenerator
Pragma Package root
{
       init
       {
          init login {}
       }

       login <displayLoginScreen >hideLoginScreen
       {
          login determiningUserMode {}
          cancel end {}
       }

       determiningUserMode < { cleanupThumbnails checkUserState }
       {
          auto autoBatch {}
          manual gettingManualBatch { createSelector }
       }

       autoBatch < { setUserAuto displayThumbnailAuto }
       {
          manual gettingManualBatch { createSelector }
          select gettingAutoBatch { createSelector }
          itemChanged * workTypeItemChanged
          redisplay * displayThumbnailAuto
          exit end {}
       }

       gettingAutoBatch <getNextAutoBatch
       {
          nextBatchFound batchSplashAuto {}
          noBatchFound determiningUserMode { noBatchDialog }
       }
```

```
gettingManualBatch <isBatchAvailable
{
    batchesFound manualBatch {}
    noBatchFound autoBatch {}
}

manualBatch < { setUserManual displayThumbnailManual }
{
    auto autoBatch {}
    refresh gettingManualBatch {}
    select batchSplashManual selectManualBatch
    redisplay * displayThumbnailManual
    exit end {}
}

(processingBatch) >hideThumbnailScreen
{
    ok * {}
    cancel * {}
    complete determiningUserMode { completeBatch
                                       cleanupBatch }
    requeue determiningUserMode { requeueBatch
                                       cleanupBatch }
    reject determiningUserMode { rejectBatch
                                       cleanupBatch }
    assign * assignPage
    exit end requeueBatch
}

processingAutoBatch : processingBatch
{
    stop processingAutoBatchStopped {}
    complete gettingAutoBatch { completeBatch
                                    cleanupBatch }
    reject gettingAutoBatch { rejectBatch
                                    cleanupBatch }
    openPage pageAutoBatch {}
    redisplay *
    displayAutoThumbnailProcessing
}

processingAutoBatchStopped : processingBatch
{
    complete determiningUserMode { completeBatch
```

```
                                    cleanupBatch }
    reject determiningUserMode { rejectBatch
                                    cleanupBatch }
    openPage pageAutoBatchStopped {}
    stop processingAutoBatch {}
    redisplay *
    displayAutoThumbnailProcessing
}

processingManualBatch : processingBatch
{
    openPage pageManualBatch {}
    redisplay *
    displayManualThumbnailProcessing
}

(batchSplash) <displayBatchSplashScreen >hideBatchSplashScreen
{
}

batchSplashAuto : batchSplash
{
    ok processingAutoBatch {allMode initBatch
                displayAutoThumbnailProcessing}
    complete gettingAutoBatch {completeBatch hideThumbnailScreen}
}

batchSplashManual : batchSplash
{
    ok processingManualBatch {allMode initBatch
                               displayManualThumbnailProcessing}
    complete determiningUserMode {completeBatch hideThumbnailScreen}
}

(page) <displayPageScreen >hidePageScreen
{
    assign * {assignPage redisplayPageScreen}
    setZone * {assignZone redisplayPageScreen}
}

pageAutoBatch : page
{
    goBack processingAutoBatch
    displayAutoThumbnailProcessing
```

```
        }

        pageAutoBatchStopped : page
        {
            goBack processingAutoBatchStopped
            displayAutoThumbnailProcessing
        }

        pageManualBatch : page
        {
            goBack processingManualBatch
            displayManualThumbnailProcessing
        }
        end <exitProgram
        {
        }
    }
```

小结

有限状态机对于软件的结构化是非常有效的设计理念。UML 为表示 FSM 提供了最有效的符号。可是，在开发和维护 FSM 方面，使用文本语言比图形更加简单。