

第2章 实例研究：设计一个文档编辑器

这一章将通过设计一个称为 Lexi^① 的“所见即所得”（或“WYSIWYG”）的文档编辑器，来介绍设计模式的实际应用。我们将会看到在 Lexi 和类似应用中，设计模式是怎样解决设计问题的。在本章最后，通过这个例子的学习你将获得 8 个模式的实用经验。

图2-1是Lexi的用户界面。文档的所见即所得的表示占据了中间的大矩形区域。文档能够以不同的格式风格自由混合文本和图形。文档的周围是通常的下拉菜单和滚动条，以及一些用来跳到特定页的页码图标。

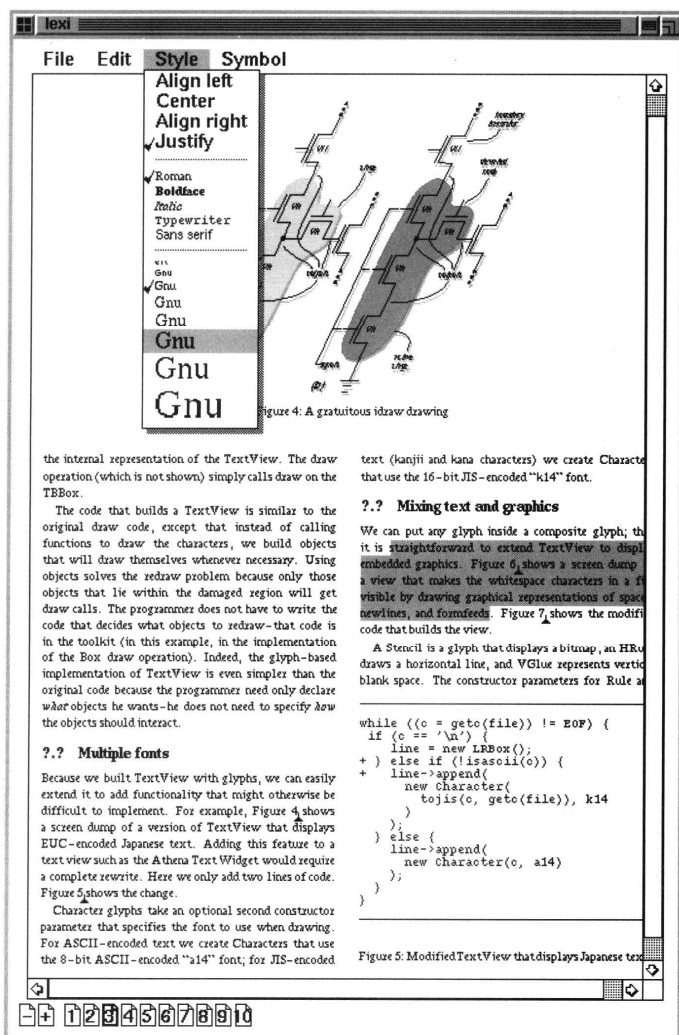


图2-1 Lexi的用户界面

① Lexi的设计是基于Calder开发的文本编辑应用Doc的。[CL92]

2.1 设计问题

我们将考察 Lexi 设计中的 7 个问题：

1) 文档结构 对文档内部表示的选择几乎影响 Lexi 设计的每个方面。所有的编辑、格式安排、显示和文本分析都涉及到这种表示。我们怎样组织这个信息会影响到应用的其他方面。

2) 格式化 Lexi 是怎样将文本和图形安排到行和列上的？哪些对象负责执行不同的格式策略？这些策略又是怎样和内部表述相互作用的？

3) 修饰用户界面 Lexi 的用户界面包括滚动条、边界和用来修饰 WYSIWYG 文档界面的阴影。这些修饰有可能随着 Lexi 用户界面的演化而发生变化。因此，在不影响应用其他方面的情况下，能自由增加和去除这些修饰就十分重要了。

4) 支持多种视感 (look-and-feel) 标准 Lexi 应不需作较大修改就能适应不同的视感标准，如 Motif 和 Presentation Manager (PM) 等。

5) 支持多种窗口系统 不同的视感标准通常是在不同的窗口系统上实现的。Lexi 的设计应尽可能的独立于窗口系统。

6) 用户操作 用户通过不同的用户界面控制 Lexi，包括按钮和下拉菜单。这些界面对应的功能分散在整个应用对象中。这里的难点在于提供一个统一的机制，既可以访问这些分散的功能，又可以对操作进行撤消 (undo)。

7) 拼写检查和连字符 Lexi 是怎样支持像检查拼写错误和决定连字符的连字点这样的分析操作的？当我们不得不添加一个新的分析操作时，我们怎样尽量少修改相关的类？

我们将在下面的各节里讨论这些设计问题。每个问题都有一组相关联的目标集合和我们怎样达到这些目标的限制条件集合。在给出特定解决方案之前，我们会详细解释设计问题的目标和限制条件。问题和其解决方案会列举一个或多个设计模式。对每个问题的讨论将在对相关设计模式的简单介绍后结束。

2.2 文档结构

从根本上来说，一个文档只是对字符、线、多边形和其他图形元素的一种安排。这些元素记录了文档的整个信息内容。然而，一个文档作者通常并不将这些元素看作图形项，而是看作文档的物理结构——行、列、图形、表和其他子结构^①。而这些子结构也有自己的子结构。

Lexi 的用户界面应该让用户直接操纵这些子结构。例如，一个用户应该能够将一个图表当作一个单元，而不是个别图形原语的一组集合。用户应该能够对表进行整体引用，而不是将表作为非结构化的一堆文本和图形。这有助于使界面简单和直观。为了使 Lexi 的实现具有类似的性质，我们选择能匹配文档物理结构的内部表示。

特别的，内部表示应支持如下几点：

- 保持文档的物理结构。即将文本和图形安排到行、列、表等。
- 可视化生成和显示文档。
- 根据显示位置来映射文档内部表示的元素。这可以使 Lexi 根据用户在可视化表示中所点击的某个东西来决定用户所引用的文档元素。

① 作者也常从逻辑结构来看文档，即看成句子、段落、节、小节和章。为了使这个例子简单，我们的文档内部表示不显式储存逻辑结构信息。但是我们描述的设计方案同样适用于表述逻辑结构信息的情况。

除了这些目标外，还有一些限制条件。首先，我们应该一致对待文本和图形。应用界面允许用户在图形中自由的嵌入文本，反之亦然。我们应该避免将图形看作文本的一种特殊情况，或将文本看作图形的特例。否则，我们最后得到的是冗余的格式和操纵机制。机制集合应该使文本和图形都能满足。

其次，我们的实现不应该过分强调内部表示中单个元素和元素组之间的差别。Lexi应该能够一致地对待简单元素和组合元素，这样就允许任意复杂的文档。例如，第5行第2列的第10个元素既可以是一个字符，也可以是一个由许多子元素组成的复杂图表。一旦我们知道这个元素能够画出自己并指定了它的区域，那么它怎样显示在页面上和它的显示位置的确定就并不困难了。

然而，为了检查拼写错误和确定连字符的连接点，需要对文本进行分析。这就与第二个限制条件产生了矛盾。我们通常并不关心一行上的元素是简单对象还是复杂对象，但是文本分析有时候依赖于被分析的对象。例如，检查多边形的拼写或以连字符连接它是没有意义的。文档内部表示设计应该考虑和权衡这个或其他潜在的彼此矛盾的限制条件。

2.2.1 递归组合

层次结构信息的表述通常是通过一种被称为递归组合 (Recursive Composition) 的技术来实现的。递归组合可以由较简单的元素逐渐建立复杂的元素，是我们通过简单图形元素构造文档的方法之一。第一步，我们将字符和图形从左到右排列形成文档的一行，然后由多行形成一列，再由多列形成一页，等等，见图2-2。

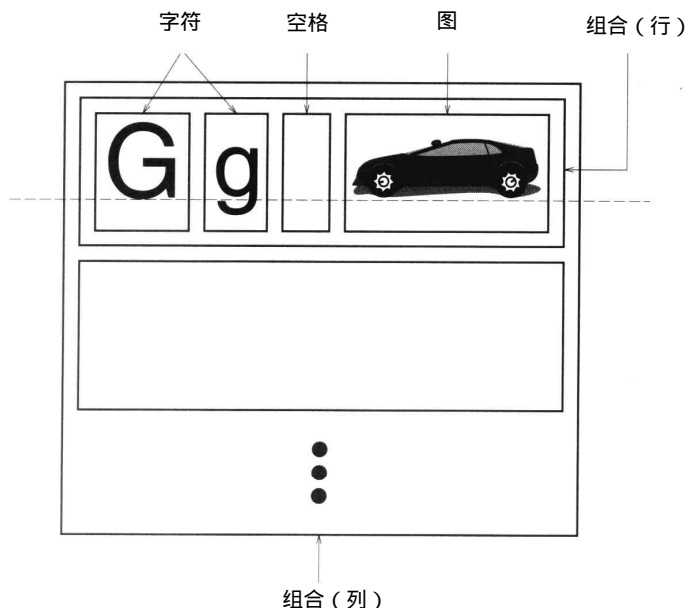


图2-2 包含正文和图形的递归组合

我们将每一个重要元素表示成一个对象，就可以描述这种物理结构。它不仅包括字符、图形等可见元素，也包括不可见的、结构化的元素，如行和列。结果就是如图2-3所示的对象结构。

通过用对象表示文档的每一个字符和图形元素，我们可以提高 Lexi最佳设计的灵活性。

我们能够在显示、格式化和互相嵌入等方面一致对待图形和文本。我们能够扩展 Lexi以支持新的字符集而不会影响其他功能。Lexi的对象结构与文档的物理结构非常相像。

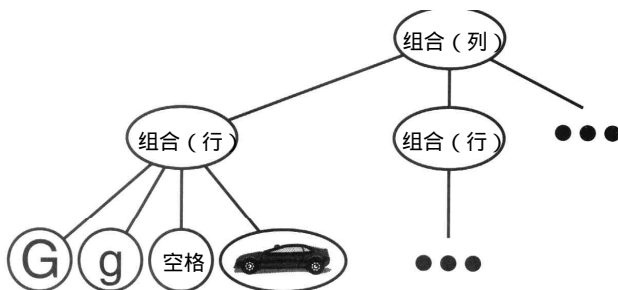


图2-3 递归组合的对象结构

这里隐含了两个重要的地方。第一个很明显，对象需要相应的类。第二个就不那么明显了，因为我们要一致性地对待这些对象，所以这些类必须有兼容的接口。在像 C++这样的语言中，可以通过继承来关联类，使得接口兼容。

2.2.2 图元

我们将为出现在文档结构中的所有对象定义一个抽象类图元(Glyph[⊖])。它的子类既定义了基本的图形元素（像字符和图像），又定义了结构元素（像行和列）。图2-4描述了Glyph类

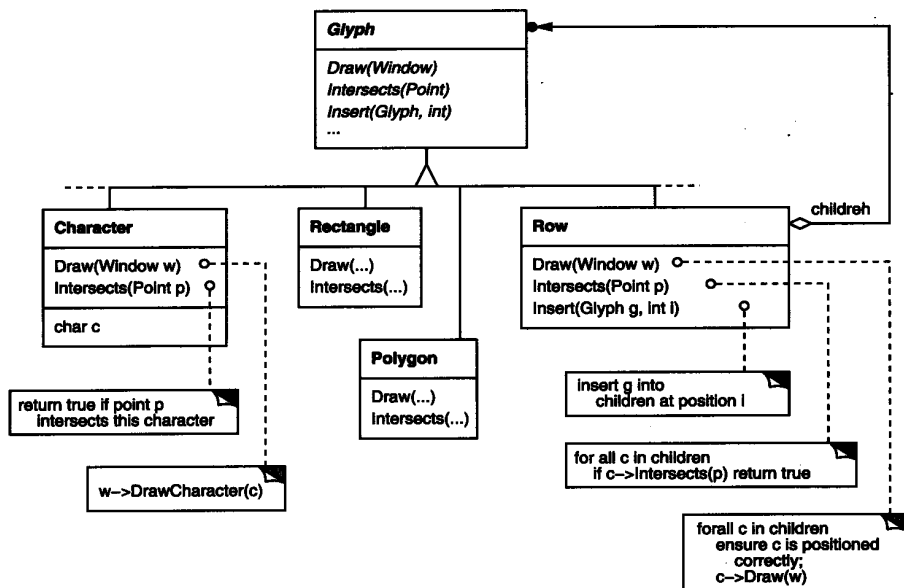


图2-4 部分Glyph类层次

⊖ Calder第一个在这种上下文使用术语“Glyph”[CL90]。大多数同时代的文档编辑器由于效率原因，并不是对一个字符就使用一个对象的。Calder在他的论文[Ca193]中论证了该方法的可行性。为了简单起见，我们将图元严格限制在类层次结构上，所以没有Calder的那么复杂。Calder的图元还能减少存储开销，形成有向无环图结构。我们也可以使用Flyweight(4.6)模式来达到相同的效果，我们将把它作为留给读者的一个练习。

层次的部分表示，表2-1以C++表示法描述了基本的 Glyph接口[⊖]。

表2-1 基本Glyph接口

Responsibility	Operations
Appearance	Virtual Void Draw (Window*) Virtual Void Bounds (Rect&)
hit detection	Virtual bool Intersects (Const Point&)
Structure	Virtual Void Insert (Glyph*, int) Virtual Void Remove (Glyph*) Virtual Glyph* Child (int) Virtual Glyph* Parent()

图元有三个基本责任，它们是 1)怎样画出自己，2)它们占用多大空间，3)它们的父图元和子图元是什么。

Glyph子类为了在窗口上表示自己，重新定义了 Draw操作。调用 Draw时，它们传递一个引用给 Window对象。Window类为了在屏幕窗口上表示文本和基本图形，定义了一些图形操作。一个Glyph的子类Rectangle可能会像下面这样重定义 Draw：

```
void Rectangle::Draw (Window* w) {
w->DrawRect(_x0, _y0, _x1, _y1);
}
```

这里的 _x0, _y0, _x1, _y1是Rectangle的数据成员，定义了矩形的对顶点。 DrawRect是 Window操作，用来在屏幕上显示矩形。

父图元通常需要知道像子图元需要占用多大空间这样的信息，以把它和其他图元安排在一行上，保证不会互相覆盖（参见图2-2）。Bounds操作返回图元占用的矩形区域，它返回的是包含该图元的最小矩形的对角顶点。 Glyph各子类重定义该操作，返回它们各自画图所用的矩形区域。

Intersects操作判断一个指定的点是否与图元相交。任何时候用户点击文档某处时， Lexi都能调用该操作确定鼠标所在的图元或图元结构。 Rectangle类重定义了该操作，用来计算矩形和给定点的相交。

因为图元可以有子图元，所以我们需要一个公共的接口来添加、删除和访问这些子图元。例如，一个行的子图元是该行上的所有图元。 Insert操作在整数Index指定的位置上插入一个图元[⊖]。 Remove操作移去一个指定的子图元。

Child操作返回给定Index的子图元（如果有的话），像行这样有子图元的图元应该内部使用 Child操作，而不是直接访问子数据结构。这样当你将数据结构由数组改为连接表时，你也不需修改像 Draw这样重复作用于各个子图元的操作。类似的， Parent操作提供一个标准的访问父图元的接口。 Lexi的图元保存一个指向其父图元的指引， Parent操作只简单的返回这个指引。

⊖ 为了使讨论简单化，我们这里特地使用最小化的接口。一个完备的接口应该包括管理颜色、字体和坐标转换等图形属性的操作，和管理更复杂子对象的操作。

⊖ 一个整数Index可能并不是指定子图元的最好方法，它依赖于图元所用的数据结构。如果图元在连接表中储存子图元，那么使用连接表指针应该更有效。我们在 2.8节讨论文档分析的时候，将会给出索引问题的更好解决方案。

2.2.3 组合模式

递归组合不仅可用来表示文档，我们还可以用它表示任何潜在复杂的、层次式的结构。Composite(4.3)模式描述了面向对象的递归组合的本质。现在是回到此模式并学习它的时候了，需要时再回头参考这个场景。

2.3 格式化

我们已经解决了文档物理结构的表示问题。接着，我们需要解决的问题是怎样构造一个特殊物理结构，该结构对应于一个恰当地格式化了文档。表示和格式化是不同的，记录文档物理结构的能力并没有告诉我们怎样得到一个特殊格式化结构。这个责任大多在于 Lexi，它必须将文本分解成行，将行分解成列等等。同时还要考虑用户的高层次的要求，例如，用户可能会指定边界宽度、缩进大小和表格形式、是否隔行显示以及其他可能的许多格式限制条件^①。Lexi的格式化算法必须考虑所有这些因素。

现在我们将“格式化”含义限制为将一个图元集合分解为若干行。下面我们可以互换使用术语“格式化”(formatting)和“分行”(linebreaking)。下面讨论的技术同样适用于将行分解为列和将列分解为页。

2.3.1 封装格式化算法

由于所有这些限制条件和许多细节问题，格式化过程不容易被自动化。这里有许多解决方法，实际上人们已经提出了各种各样具有不同能力和缺陷的格式化算法。因为 Lexi是一个所见即所得编辑器，所以一个必须考虑的重要权衡之处在于格式化的质量和格式化的速度之间的取舍。我们通常希望在不牺牲文档美观外表的前提下，能得到良好的反映速度。这种权衡受许多因素影响，而并不是所有因素在编译时刻都能确定的。例如，用户也许能忍受稍慢一点的响应速度，以换取较好的格式。这种选择也许导致了比当前算法更适用的彻底不同的格式化算法。另一个例子，更多实现驱动的权衡是在格式化速度和存储需求之间：很有可能为了缓存更多的信息而降低格式化速度。

因为格式化算法趋于复杂化，因而可以考虑将它们包含于文档结构之中，但最好是将它们彻底独立于文档结构之外。理想情况下，我们能够自由地增加一个 Glyph子类而不用考虑格式算法。反过来，增加一个格式算法不应要求修改已有的图元类。

这些特征要求我们设计的 Lexi易于改变格式化算法。最好能在运行时刻改变这个算法，如果难以实现，至少在编译时刻应该可以很方便地改变。我们可以将算法独立出来，并把它封装到对象中使其便于替代。更进一步，可以定义一个封装格式化算法的对象的类层次结构。类层次结构的根结点将定义支持许多格式化算法的接口，每个子类实现这个接口以执行特定的算法。那时就能让 Glyph子类对象自动使用给定算法对象来排列其子图元。

2.3.2 Compositor和Composition

我们为封装格式化算法的对象定义一个 Compositor类。它的接口（见表 2-2）可让

① 用户可能更关心的是文档的逻辑结构——句子、段落、小节、章节等等。相比而言，对物理结构就没有这样的兴趣了。大部分用户不在意段落中的换行发生在何处，只要该段落能正确格式化就行了。格式化列和页，也是这样的。因而用户最终只指定物理结构的高层限制条件，用来满足他们的艰难工作则由 Lexi去完成。

compositor获知何时去格式化哪些图元。它所格式化的图元是一个被称为 Composition的特定图元的各个子图元。一个 Composition在创建时得到一个 Compositor子类实例，并在必要的时候（如用户改变文档的时候）让 Compositor对它的图元作 Compose操作。图 2-5描述了 Composition类和Compositor类之间的关系。

表2-2 基本Compositor接口

责 任	操 作
格式化的内容	void SetComposition (Composition*)
何时格式化	virtual void Compose()

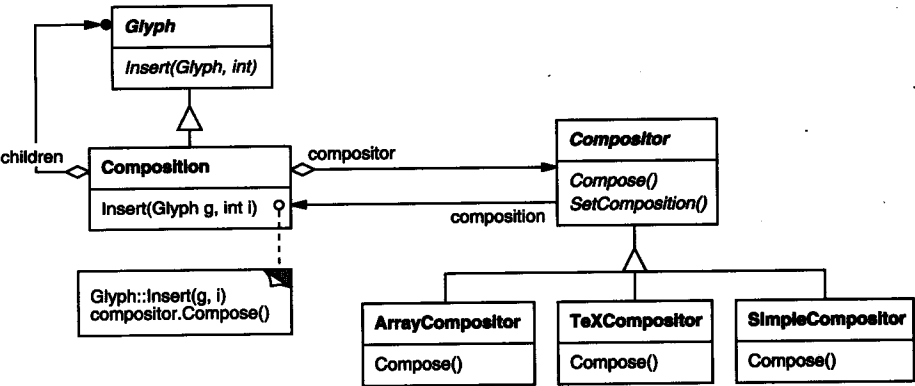


图2-5 Composition和Compositor类间的关系

一个未格式化的 Composition对象只包含组成文档基本内容的可见图元。它并不包含像行和列这样的决定文档物理结构的图元。Composition对象只在刚被创建并以待格式化的图元进行初始化后，才处于这种状态。当 Composition需要格式化时，调用它的 Compositor的 Compose操作。Compositor依次遍历Composition的各个子图元，根据分行算法插入新的行和列图元[⊖]。图2-6显示了得到的对象结构。图中由 Compositor创建和插入到对象结构中的图元

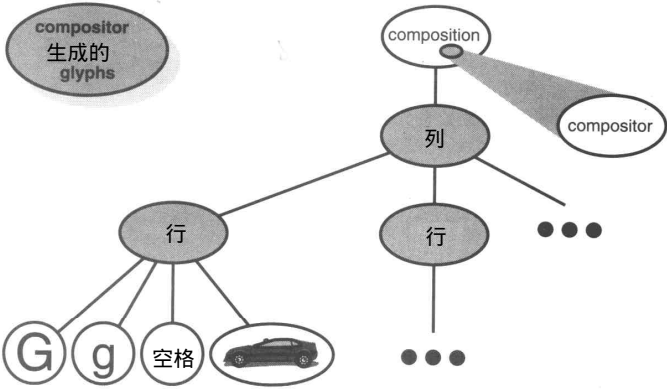


图2-6 对象结构反映Compositor制导的分行

⊖ Compositor为了计算换行必须知道字符图元的字符代码。在 2.8节，我们将会看到：怎样可以不在 Glyph接口中添加一个特定于字符的操作，而多态地获得这个信息。

以灰色背景显示。

每一个Compositor子类都能实现一个不同的分行算法。例如，一个 SimpleCompositor可以执行得很快，而不考虑像文档“色彩”这样深奥的东西。好的色彩意味着文本和空白的平滑分布。一个TeXCompositor会实现完全的T_EX算法[Knu84]，会考虑像色彩这样的东西，而以较长的格式化时间作为代价。

Compositor-Composition类的分离确保了支持文档物理结构的代码和支持不同格式化算法的代码之间的分离。我们能增加新的 Compositor子类而不触及 Glyph类，反之亦然。事实上，我们通过给Composition的基本图元接口增加一个 SetCompositor操作，即可在运行时刻改变分行算法。

2.3.3 策略模式

在对象中封装算法是 Strategy(5.9)模式的目的。模式的主要参与者是 Strategy对象（这些对象中封装了不同的算法）和它们的操作环境。其实 Compositor就是Strategy。它们封装了不同的格式算法。Composition就是Compositor策略的环境。

Strategy模式应用的关键点在于为 Strategy和它的环境设计足够通用的接口，以支持一系列的算法。你不必为了支持一个新的算法而改变 Strategy或它的环境。在我们的例子中，支持子图元访问、插入和删除操作的基本 Glyph接口就足以满足一般的用户需求，不管 Compositor子类使用何种算法，都足以支持其对文档的物理结构的修改。同样地，Compositor接口也足以支持Composition启动格式化操作。

2.4 修饰用户界面

我们针对Lexi用户界面考虑两种修饰，第一种是在文本编辑区域周围加边界以界定文本页；第二种是加滚动条让用户能看到同一页的不同部分。为了便于增加和去除这些修饰（特别是在运行时刻），我们不应该通过继承方式将它们加到用户界面。如果其他用户界面对象不知道存在这些修饰，那么我们就获得了最大的灵活性。这使我们无需改变其他的类就能增加和移去这些修饰。

2.4.1 透明围栏

从程序设计角度出发，修饰用户界面涉及到扩充已存在的代码。我们可以用继承的方式完成这种扩充，但如此运行时刻对这些修饰作重新安排则十分困难。并且同样严重的问题是，基于类继承方法通常会引起类爆炸现象。

我们可以为Composition创建一个子类BorderedComposition，用来给Composition添加边界，或者以同样方式创建子类 ScrollableComposition来添加滚动条。如果我们既想要滚动条又想要边界，则可创建BorderedScrollableComposition等等。极端情况下，我们创建一个包含各种可能修饰组合的类。但一旦修饰类型增加，它就变得无效了。

对象组合提供了一种潜在的更有效和更灵活的扩展机制，但是我们组合一些什么对象呢？既然我们知道要修饰的是已有的图元，我们就可以把修饰本身看作对象（如，类 Border的实例）。这样我们有了两个组合候选对象：图元（Glyph）和边界（Border）。下一步是决定用谁来组合谁的问题。我们可以在边界中包含图元，这给人以边界在屏幕上包围了图元的感

觉。或者，反之在图元中包含边界，但是我们必须对相应的 Glyph子类作修改以使边界对所有子类有效。在我们的第一个选择中，可以将画边界的代码完全保存在 Border类中，而独立于其他类。

Border类看起来是什么样的呢？边界有形这个事实说明它的确应该是图元，即 Border类应该是Glyph的子类。此外还有一个强制性的必须如此的原因：客户应该一致地对待图元，而不应关心图元是否有边界。当客户画一个简单的、无边界的图元时，就不必对它作修饰。如果那个图元包含于一个边界对象中，客户应该以画出前面简单图元同样的方法画出这个边界对象，而不应该特殊对待该边界对象。这暗示了 Border接口是与 Glyph接口匹配的。我们将 Border作为Glyph的子类可以保证这种关系。

我们根据这些得出了透明围栏(Transparent Enclosure)的概念。它结合了两个概念：1) 单子女(单组件)组合；2) 兼容的接口。客户通常分辨不出它们是在处理组件还是组件的围栏(即，这个组件的父组件)，特别是当围栏只是代理组件的所有操作时更是如此。但是围栏也能通过在代理操作之前或之后添加一些自己的操作来修改组件的行为。围栏也能有效地为组件添加状态。

2.4.2 MonoGlyph

我们可以将透明围栏的概念用于所有的修饰其他图元的图元。为了使这个概念具体化，我们定义 Glyph的子类MonoGlyph作为所有像Border这样“起修饰作用的图元”的抽象类(见图2-7)。MonoGlyph保存了指向一个组件的引用并且传递所有的请求给这个组件。

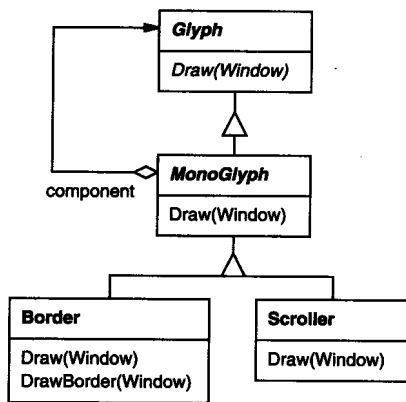


图2-7 MonoGlyph类关系

这使得MonoGlyph缺省情况下对客户完全透明。例如，MonoGlyph实现Draw操作如下：

```
void MonoGlyph::Draw (Window* w) {
    _component->Draw(w);
}
```

MonoGlyph的子类至少重新实现一个这样的传递操作，例如，Border::Draw首先激活基于组件的父类操作 MonoGlyph::Draw，让组件做部分工作——即画出边界以外的其他东西。Border::Draw通过调用私有操作 DrawBorder来画出边界。细节我们这里不赘述了：

```
void Border::Draw (Window* w) {  
    MonoGlyph::Draw(w);  
    DrawBorder(w);  
}
```

注意Border::Draw是怎样有效扩展父类操作来画出边界的。这与忽略 MonoGlyph::Draw的调用，而完全代替父类操作是截然不同的。

另一个出现在图 2-7 中的 MonoGlyph 子类是 Scroller，它根据作为修饰的两个滚动条的位置在不同的位置画出组件。当画它的组件时，它会告诉图形系统裁剪边界以外的部分，滚动出视图以外的部分是不会显示在屏幕上的。

现在我们已经有了给 Lexi 文本编辑区增加边界和滚动界面所需的一切准备。我们可以在一个 Scroller 实例中组合已存在的 Composition 实例以增加滚动界面，然后再把它组合到 Border 实例中。结果对象结构如图 2-8 所示。

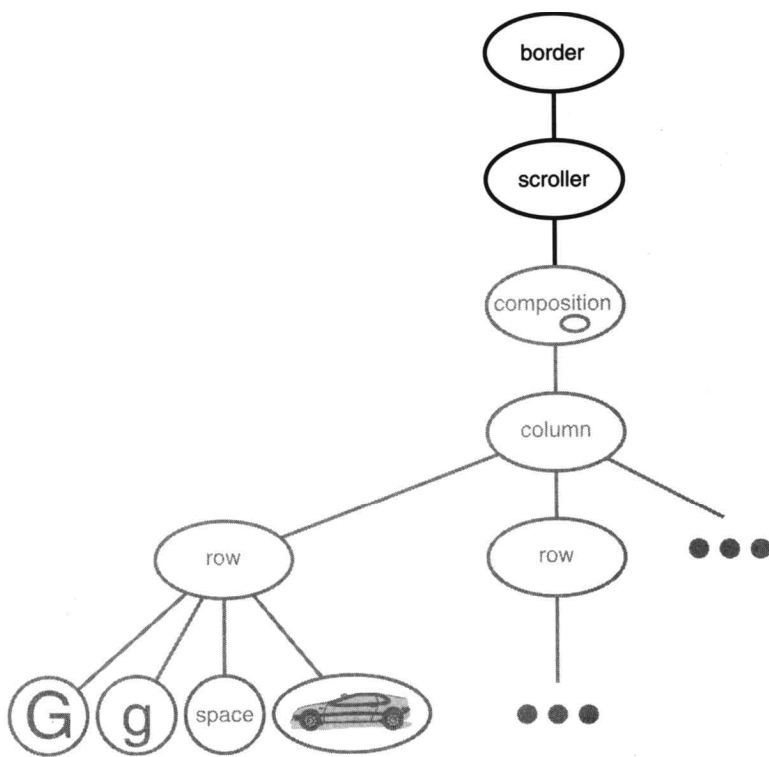


图2-8 嵌入对象结构

注意我们也可以交换组合顺序，把一个带有边界的组合放在 Scroller 实例中。这样边界可以和文本一起滚动，但我们一般不要求这么做。关键在于，透明围栏使得试验不同的选择变得很容易，使得客户和修饰代码无关。

还要注意 Border 是怎样组合一个而不是两个或多个 Glyph 对象的。这不同于我们迄今为止所定义的组合，在那些组合中父对象是允许有多个不确定的子对象的。这里讲给某物加上边界暗示了“某物”是单个的。我们可以定义同时修饰多个对象的行为，但那样我们就不得不将多种组合和修饰概念混合起来形成所谓的行修饰、列修饰等等。因为我们已经有许多类用来做这些组合，所这种行为对我们并没帮助。我们最好使用已有的类去做组合的工作，并

通过增加新类去修饰组合的结果。使修饰独立于其他组合，既可以简化修饰类又可以减少它们的数目，还可以保证我们不重复已有的组合功能。

2.4.3 Decorator模式

Decorator(4.4)模式描述了以透明围栏来支持修饰的类和对象的关系。事实上术语“修饰”的含义比我们这里讨论的更广泛。在Decorator模式中，修饰指给一个对象增加职责的事物。我们可以想到用语义动作修饰抽象语法树、用新的转换修饰有穷状态自动机或者以属性标签修饰持久对象网等例子。Decorator一般化了我们在Lexi中使用的方法，而使它具有更广泛的实用性。

2.5 支持多种视感标准

获得跨越硬件和软件平台的可移植性是系统设计的主要问题之一。将 Lexi重新定位于一个新的平台不应当要求对 Lexi进行重大的修改，否则的话就失去了重新定位 Lexi的价值。我们应当使移植尽可能地方便。

移植的一大障碍是不同视感标准之间的差异性。视感标准本是用来加强某一窗口平台上各个应用之间用户界面的一致性的。这些标准定义了应用应该怎样显示和对用户请求作出反映。虽然已有的标准彼此差别不大，但用户还是可以清楚地区分它们——一个应用程序在 Motif平台上的视感决不会与其他某个平台上的完全一样，反之亦然。一个运行于多个平台的应用程序必须符合各个平台的用户界面风格。

我们的设计目标就是使 Lexi符合多个已存在的视感标准，并且在新标准出现时要能很容易地增加对新标准的支持。我们也希望我们的设计能支持最大限度的灵活性：运行时刻可以改变Lexi的外观和感觉。

2.5.1 对象创建的抽象

我们在Lexi用户界面看到的和操作的是一个图元，它被组合于诸如行和列等不可见的图元之中。而这些不可见图元又组合了按钮、字符等可见图元，并能正确的展现它们。界面风格关于所谓的“窗口组件”(Widgets)有许多视感规则。窗口组件是关于用户界面上作为控制元素的按钮、滚动条和菜单等可视图元的另一个术语。窗口组件可以使用像字符、圆、矩形和多边形等简单图元来表示数据。

我们假定用两个窗口组件图元集合来实现多个视感标准：

1) 第一个集合是由抽象 Glyph子类构成的，对每一种窗口组件图元都有一个抽象 Glyph子类。例如，抽象子类 ScrollBar放大了基本的Glyph接口，以便增加通用的滚动操作；Button是用来增加按钮有关操作的抽象类；等等。

2) 另一个集合是与抽象子类对应的实现不同视感标准的具体的子类的集合。例如，ScrollBar可能有 MotifScrollBar和PMScrollBar两个子类以实现相应的 Motif和PM(Presentation Manager)风格的滚动条。

Lexi必须区分不同视感风格的窗口组件图元之间的差异。例如，当 Lexi需要在界面上放一个按钮时，它必须实例化一个有正确按钮风格的 Glyph子类 (MotifButton、PMButton或MacButton等)。

很明显Lexi的实现不能够直接通过调用 C++构造器来做这些工作，那会把按钮硬性编定为

一种特殊风格，而不能在运行时刻选择风格。当 Lexi要移植到其他平台时，我们还不得不进行代码搜索以改变所有这些构造器调用。并且按钮还仅仅是 Lexi用户界面上众多窗口组件之一。对特定视感类进行构造器调用会使代码混乱，产生维护困难——只要稍有遗漏，你就可能在Mac应用程序中使用了Motif的菜单。

Lexi需要一种方法来确定创建合适窗口组件所需的视感标准。我们不仅必须避免显式的构造器调用，还必须能够很容易地替换整个窗口组件集合。可以通过抽象对象创建过程来达到上述两个要求，我们将用一个例子来说明。

2.5.2 工厂类和产品类

通常我们可能使用下面的C++代码来创建一个Motif滚动条图元实例：

```
ScrollBar* sb = new MotifScrollBar;
```

但如果你想使Lexi的视感依赖性最小的话，这种代码要尽量避免。假如我们按如下方法初始化sb：

```
ScrollBar* sb = guiFactory->CreateScrollBar();
```

这里guiFactory是MotifFactory类的实例。CreateScrollBar为所需要的视感返回一个合适的ScrollBar子类的新的实例，如 MotifScrollBar。一旦跟客户相连，它就等价于直接调用一个 MotifScrollBar的构造器。但是两者有本质区别：它不像使用直接构造器那样在程序代码中提及Motif 的名字。guiFactory对象抽象了任何视感标准下的滚动条的创建过程，而不仅仅是Motif滚动条的。并且 guiFactory不局限于创建滚动条，它广泛适用于包括滚动条、按钮、输入域、菜单等窗口组件图元。

上述办法是可行的，其原因在于MotifFactory是GUIFactory的子类，而GUIFactory是定义了创建窗口组件图元公共接口的抽象类，它包含了用以实例化不同窗口组件图元的像CreateScrollBar和CreateButton这样的操作。GuiFactory的子类实现这些操作，并返回像MotifScrollBar和PMButton这样实现特定视感的图元。图2-9显示了guiFactory对象的结果类层次结构。

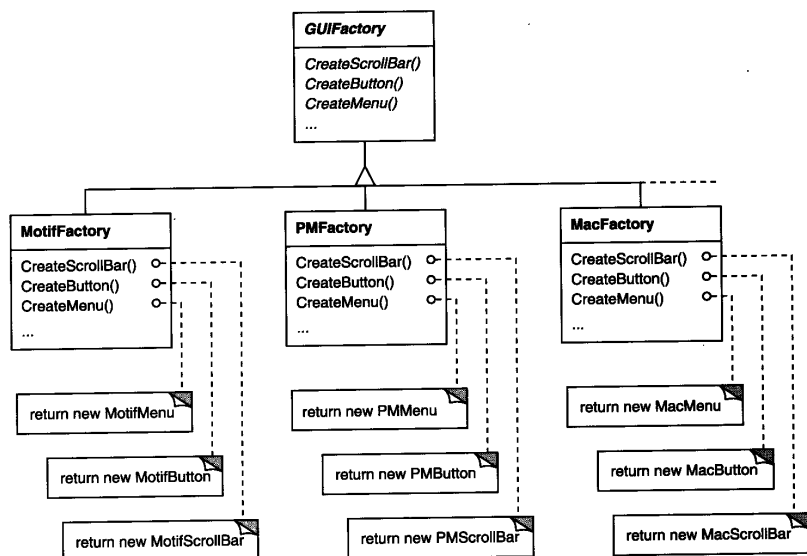


图2-9 GUIFactory类层次

我们说工厂 (Factory) 创造了产品 (Product) 对象。更进一步，工厂生产的产品是彼此相关的；这种情况下，产品是相同视感的所有窗口组件。图 2-10 显示了这样一些产品类，工厂产生窗口组件图元时要用到它们。

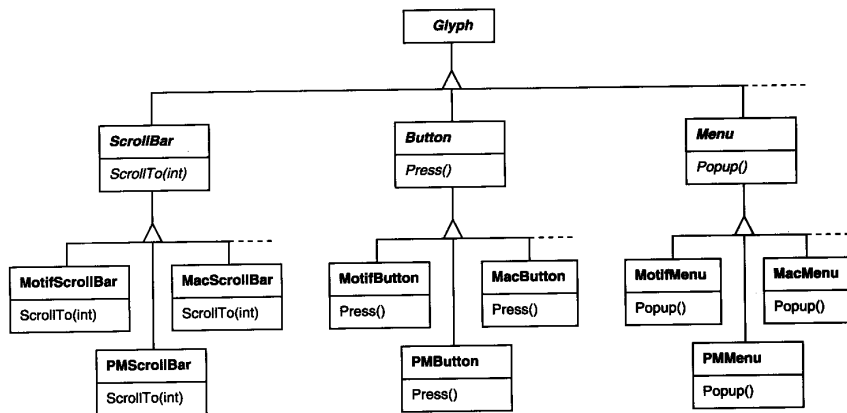


图2-10 抽象产品类和具体子类

我们要回答的最后一个问题是：GUIFactory实例是从哪儿来的？答案是哪儿方便就从哪儿来。变量guiFactory可以是全局变量、一个众所周知的类的静态成员，或者如果整个用户界面是在一个类或一个函数中创建的，它甚至可以是局部变量。甚至有一个设计模式 Singleton(3.5) 专门用来管理这样的众所周知的、只能创建一次的对象。然而，重要的是在程序中某个合适的地方来初始化 guiFactory，这要在它被用来创建窗口组件之前，而在所需的视感标准清楚确定下来之后。

如果视感在编译时刻就知道了，那么 guiFactory能够在程序开始的时候以一个新的工厂实例简单赋值来初始化：

```
GUIFactory* guiFactory = new MotifFactory;
```

如果用户能通过程序启动时候的字符串来指定视感，那么创建工厂的代码可能是：

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
// user or environment supplies this at startup

if (strcmp(styleName, "Motif") == 0) {
    guiFactory = new MotifFactory;
} else if (strcmp(styleName, "Presentation_Manager") == 0) {
    guiFactory = new PMFactory;
} else {
    guiFactory = new DefaultGUIFactory;
}
```

还有更高级的在运行时刻选择工厂的方法。例如，你可以维持一个登记表，将字符串映射给工厂对象。这允许你无需改变已有代码就能登记新的工厂子类实例，而前面的方法则要求你改变代码。并且这样你还不必将所有平台的工厂连接到应用中。这一点很重要，因为在一个不支持Motif的平台上连接一个MotifFactory是不太可能的。

但是关键还在于一旦我们给应用配置好了正确的工厂对象，它的视感从那时起就设定好了。而如果我们改变了主意，我们还能以一个不同的视感工厂重新初始化 `guiFactory`，重新构造界面。我们知道，不管怎样、何时初始化 `guiFactory`，一旦这么做了，应用就可以在不修改代码的前提下创建合适的外观。

2.5.3 Abstract Factory 模式

工厂（Factory）和产品（Product）是Abstract Factory (3.1) 模式的主要参与者。该模式描述了怎样在不直接实例化类的情况下创建一系列相关的产品对象。它最适用于产品对象的数目和种类不变，而具体产品系列之间存在不同的情况。我们通过实例化一个特定的具体工厂对象来选择产品系列，并且以后一直使用该工厂生产产品对象。我们也能够通过用一个不同的具体工厂实例来替换原来的工厂对象以改变整个产品系列。抽象工厂模式对产品系列的强调使它区别于其他只与一种产品对象有关的创建性模式。

2.6 支持多种窗口系统

视感只是众多移植问题之一。另一个移植性问题就是 Lexi所运行的窗口环境。一个平台将多个互相重叠的窗口展示在一个点阵显示器上。它管理屏幕空间和键盘、鼠标到窗口的输入通道。目前存在一些互不兼容的重要的窗口系统（如Macintosh、Presentation Manager、Windows、X等）。我们希望Lexi可以在尽可能多的窗口系统上运行，这和 Lexi要支持多个视感标准是同样的道理。

2.6.1 我们是否可以使用Abstract Factory模式

乍一看，这似乎又是一个使用 Abstract Factory模式的情况。但是对窗口系统移植的限制条件与视感的独立性条件是有极大不同的。

在使用 Abstract Factory 模式时，我们假设我们能为每一个视感标准定义具体的窗口组件类。这意味着我们能从一个抽象产品类（如 `ScrollBar`），针对一个特定标准来导出每一个具体产品（如 `MotifScrollBar`、`MacScrollBar`等）。现在假设我们已经有一些不同厂家的类层次结构，每一个类层次对应一个视感标准。当然，这些类层次不太可能有太多兼容之处。因而我们无法给每个窗口组件（滚动条、按钮、菜单等）都创建一个公共抽象产品类。而没有这些类 Abstract Factory模式无法工作。所以我们不得不根据抽象产品接口的共同集合来调整不同的窗口组件类层次结构。只有这样我们才能在我们的抽象工厂接口中定义合适的 `Create...`操作。

对窗口组件，我们通过开发我们自己的抽象和具体的产品类来解决这个问题。现在当我们试图使Lexi工作在已有窗口的系统时，我们面对的是类似的问题。即不同的窗口系统有不兼容的程序设计接口。但这次的麻烦更大些，因为我们不能实现我们自己的非标准窗口系统。

但是事情还是有挽回的余地。像视感标准一样，窗口系统的接口也并非截然不同。因为所有的窗口系统总的来说是做同一件事的。我们可对不同的窗口系统作一个统一的抽象，在对各窗口系统的实现作一些调整，使之符合公共的接口。

2.6.2 封装实现依赖关系

在2.2节中，我们介绍了用以显示一个图元或图元结构的 `Window`类。我们并没有指定这个

对象工作的窗口系统，因为事实上它并不来自于哪个特定的窗口系统。 Window类封装了窗口要各窗口系统都要做的一些事情：

- 它们提供了画基本几何图形的操作。
- 它们能变成图标或还原成窗口。
- 它们能改变自己的大小。
- 它们能够根据需要画出（或重画出）窗口内容。例如，当它们由图标还原为窗口时，或它们在屏幕空间上重叠、出界的部分重新显示时，都要重画，如表2-3所示。

表2-3 Windows类接口

责 任	操 作
窗口管理	virtual void Redraw()
	virtual void Raise()
	virtual void Lower()
	virtual void Iconify()
	virtual void Deiconify()
图形	...
	virtual void DrawLine(...)
	virtual void DrawRect(...)
	virtual void DrawPolygon(...)
	virtual void DrawText(...)
	...

Window类的窗口功能必须跨越不同的窗口系统。让我们考虑两种极端的观点：

1) 功能的交集 Window类的接口只提供所有窗口系统共有的功能。该方法的问题在于Window接口在能力上只类似于一个最小功能的窗口系统，对一些即使是大多数窗口系统都支持的高级特征，我们也无法利用。

2) 功能并集 创建一个合并了所有存在系统的功能的接口。但是这样的接口势必规模巨大，并且存在不一致的地方。此外，当某个厂商修改它的窗口系统时，我们不得不修改这个接口和Lexi，因为Lexi依赖于它。

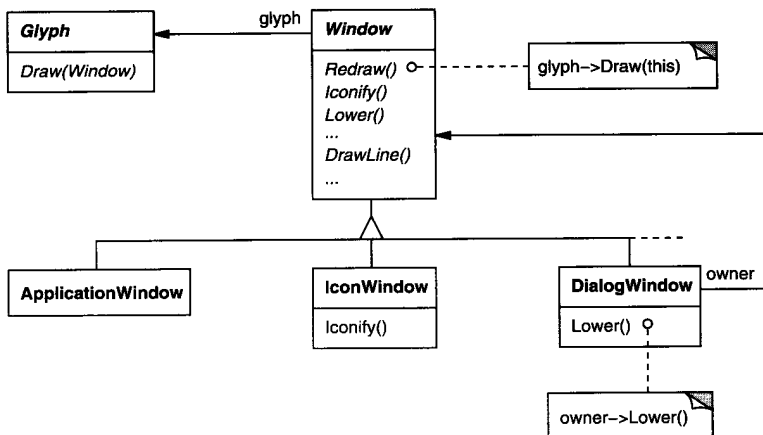
以上两种方法都不切实可行，所以我们的设计将采取折中的办法。 Window类将提供一个支持大多数窗口系统的方便的接口。因为 Lexi直接处理 Window类，所以它还必须支持Lexi的图元。这意味着 Window接口必须包括让图元可以在窗口中画出自己的基本图形操作集合。表 2-3给出了Window类中一些操作的接口。

Window是一个抽象类。其具体子类支持用户用到的不同种类的窗口。例如，应用窗口、图标和警告对话框等都是窗口，但它们在行为上稍有不同。所以我们能定义像 Application Window、IconWindow和DialogWindow这样的子类去描述这些不同之处。得到的类层次结构给了像Lexi这样的应用一个统一的窗口抽象，这种窗口层次结构不依赖于任何特定厂商的窗口系统，如下页上图所示。

现在我们已经为 Lexi定义了工作的窗口接口，那么真正与平台相关的窗口是从哪儿来的？既然我们不能实现自己的窗口系统，那么这个窗口抽象必须用目标窗口系统平台来实现。怎样实现？

一种方法是实现 Window类和它的子类的多个版本，每个版本对应一个窗口平台。当我们

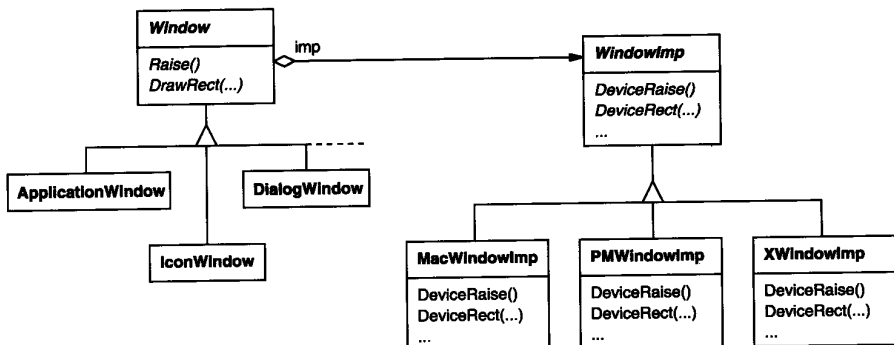
在一给定平台上建立 Lexi 时，我们选择一个相应的版本。但想象一下，维护问题实在令人头疼，我们已经保存了多个名字都是“Window”的类，而每一个类实现于一个不同的窗口系统。另一种方法是为每一个窗口层次结构中类创建特定实现的子类，但这会产生我们在试图增加修饰时遇到的同样的子类数目爆炸问题。这两种方法还都有另一个缺点：我们没有在编译以后改变所用窗口系统的灵活性。所以我们还不得不保持若干不同的可执行程序。



既然这两种方法都没有吸引力，那么我们还能做些什么呢？那就是我们在格式化和修饰时都做过的：对变化的概念进行封装。现在所变化的是窗口系统实现。如果我们能在一个对象中封装窗口系统的功能，那么我们就根据对象接口实现 **Window** 类及其子类。更进一步讲，如果那个接口能够提供我们所感兴趣的所有窗口系统的服务，那么我们无需改变 **Window** 类或其子类，也能支持不同的窗口系统。我们可以通过简单传递合适的窗口系统封装对象，来给我们想要的窗口系统设定窗口对象。我们甚至能在运行时刻设定窗口。

2.6.3 Window和WindowImp

我们将定义一个独立的 **WindowImp** 类层次来隐藏不同窗口系统的实现。**WindowImp** 是一个封装了窗口系统相关代码的对象的抽象类。为了使 Lexi 运行于一个特定的窗口系统，我们用该系统的一个 **WindowImp** 子类实例设置 **Window** 对象。下面的图显示了 **Window** 和 **WindowImp** 层次结构之间的关系。



通过在 WindowImp 类中隐藏实现，我们避免了对窗口系统的直接依赖，这可以让 Window 类层次保持相对较小并且较稳定。同时我们还能方便地扩展实现层次结构以支持新的窗口系统。

1. WindowImp 的子类

WindowImp 的子类将用户请求转变成对特定窗口系统的操作。考虑我们在 2.2 节所用的例子，我们根据 Window 实例的 DrawRect 操作定义了 Rectangle::Draw：

```
void Rectangle::Draw (Window* w) {
    w->DrawRect (_x0, _y0, _x1, _y1);
}
```

DrawRect 的缺省实现使用了 WindowImp 定义的画出矩形的抽象操作：

```
void Window::DrawRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    _imp->DeviceRect(x0, y0, x1, y1);
}
```

这里 _imp 是 Window 的成员变量，它保存了设置 Window 的 WindowImp。窗口的实现是由 _imp 所指的 WindowImp 子类的实例定义的。对于一个 XWindowImp (即 X 窗口系统的 WindowImp 子类)，DeviceRect 的实现可能如下：

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

DeviceRect 这样做是因为 XDrawRectangle (在 X 系统中画矩形的接口) 是根据矩形的左下顶点、宽度和高度定义矩形的，DeviceRect 必须根据参数值来计算这些值。首先它必须确定左下顶点 (因为 (x0,y0) 可能是矩形四个顶点中的任一个)，然后计算宽度和高度。

PMWindowImp (Presentation Manager 的 WindowImp 子类) 定义 DeviceRect 时会有所不同：

```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];

    point[0].x = left;    point[0].y = top;
    point[1].x = right;   point[1].y = top;
    point[2].x = right;   point[2].y = bottom;
    point[3].x = left;    point[3].y = bottom;

    if (
        (GpiBeginPath(_hps, 1L) == false) ||
        (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
    )
```

```

        (GpiPolyLine(_hps, 4L, point) == GPI_ERROR) ||
        (GpiEndPath(_hps) == false)
    ) {
        // report error

    } else {
        GpiStrokePath(_hps, 1L, 0L);
    }
}

```

为什么这和X版本有如此大的差别？因为PM没有像X那样显式画矩形的操作，它有一个更一般性的接口可以指定多个段（或称之为路径）的顶点、画出这些线段并且填充它们所围成的区域。

DeviceRect的PM实现很显然与X的实现有很大不同，但问题不大。WindowImp用一个可能巨大但却稳定的接口隐藏了各个窗口系统接口的差异。这使得Window子类的实现者可以将更多的精力放在窗口的抽象上，而不是窗口系统的细节。它也支持我们增加新的窗口系统，而不会搞乱Window类。

2. 用WindowImp来配置Windows

我们还没有论述的一个关键问题是：怎样用一个合适的WindowImp子类来配置一个窗口？也就是说，什么时候初始化_imp，谁知道正在使用的是哪个窗口系统（也就是哪一个WindowImp子类）？窗口在能做它所感兴趣的事情之前，都需要某种WindowImp。

这些问题的答案存在很多种可能性，但我们只关注使用Abstract Factory(3.1)模式的情形。我们可以定义一个抽象工厂类WindowSystemFactory，它提供了创建不同种与窗口系统有关的实现对象的接口：

```

class WindowSystemFactory {
public:
    virtual WindowImp* CreateWindowImp() = 0;
    virtual ColorImp* CreateColorImp() = 0;
    virtual FontImp* CreateFontImp() = 0;

    // a "Create..." operation for all window system resources
};

```

现在我们可以为每一个窗口系统定义一个具体的工厂：

```

class PMWindowSystemFactory : public WindowSystemFactory {
    virtual WindowImp* CreateWindowImp()
    { return new PMWindowImp; }
    // ...
};

class XWindowSystemFactory : public WindowSystemFactory {
    virtual WindowImp* CreateWindowImp()
    { return new XWindowImp; }
    // ...
};

```

Window基本类的构造器能使用WindowSystemFactory接口和合适的窗口系统的WindowImp来初始化成员变量_imp：

```

Window::Window () {
    _imp = windowSystemFactory->CreateWindowImp();
}

```


windowSystemFactory变量是 WindowSystemFactory 某个子类的实例，它是公共可见的，正如 guiFactory 是公共可见的定义视感的变量。windowSystemFactory 变量可用相同的方法进行初始化。

2.6.4 Bridge 模式

WindowImp 类定义了一个公共窗口系统设施的接口，但它的设计是受不同于 Window 接口的限制条件驱动的。应用程序员不直接处理 WindowImp 的接口；它们只处理 Window 对象。所以 WindowImp 的接口不必与应用程序员的客观世界视图一致，就像我们只关心 Window 类层次和接口的设计。WindowImp 的接口更能如实反映事实上提供的是什么窗口系统。它可以偏向于功能方法的交集，也可以偏向于功能方法的合集，只要是最适合各目标窗口系统即可。

要注意的是 Window 类接口是针对应用程序员的，而 WindowImp 接口是针对窗口系统的。将窗口功能分离到 Window 和 WindowImp 类层次中，这样我们可以独立实现这些接口。这些类层次的对象合作来实现 Lexi 无需修改就能运行在多窗口系统的目标。

Window 和 WindowImp 的关系是 Bridge(4.2) 模式的一个例子。Bridge 模式的目的是允许分离的类层次一起工作，即使它们是独立演化的。我们的设计准则使得我们创建了两个分离的类层次，一个支持窗口的逻辑概念，另一个描述了窗口的不同实现。Bridge 模式允许我们保持和加强我们对窗口的逻辑抽象，而不触及窗口系统相关的代码。反之也一样。

2.7 用户操作

Lexi 的一些功能可以通过文档的 WYSIWYG 表示得到。你可以敲入和删除文本，移动插入点，通过在文档上直接点、击和打字来选择文本区域。另一些功能是通过 Lexi 的下拉菜单、按钮和键盘加速键来间接得到的。这些功能包括：

- 创建一个新的文档。
- 打开、保存和打印一个已存在文档。
- 从文档中剪切选中的文本和将它粘贴回文档。
- 改变选中文本的字体和风格。
- 改变文本的格式，例如对齐格式和调整格式。
- 退出应用。

等等。

Lexi 为这些用户操作提供不同的界面。但是我们不希望一个特定的用户操作就联系一个特定的用户界面。因为我们可能希望多个用户界面对应一个操作（例如，你既可以用一个页按钮，也可以用一个菜单项来表示翻页）。你可能以后也会改变界面。

再说，这些操作是用不同的类来实现的。我们想要访问这些功能，但又不希望在用户界面类和它的实现之间建立过多依赖关系。否则的话，最终我们得到的是紧耦合的实现，它难以理解、扩充和维护。

更复杂的是我们希望 Lexi 能对大多数功能支持撤销（undo）和重做（redo）[⊖] 操作。特别地，我们希望撤销类似删除这样一不注意就会破坏数据的操作的用户。但是我们不应该试图

⊖ 即重做一个刚被撤销的操作

撤销像保存一幅画和退出应用程序这样的操作。这些操作应该不受撤销操作的影响。我们也不希望对撤销和重做的等级进行任何限制。

很明显对用户操作的支持渗透到了应用中。我们所面临的挑战在于提出一个简单、可扩充的机制来满足所有这些要求。

2.7.1 封装一个请求

从我们设计者的角度出发，一个下拉菜单仅仅是包含了其他图元的又一种图元。下拉菜单和其他有子女的图元的差别在于大多数菜单中的图元会响应鼠标点击做一些操作。

让我们假设这些做事物的图元是一个被称之为 MenuItem 的 Glyph 子类的实例，并且它们做一些事情来响应客户的一个请求^①。执行一个请求可能涉及到一个对象的一个操作或多个对象的多个操作，或其他介于这两者之间的情况。

我们可以为每个用户操作定义一个 MenuItem 的子类，然后为每个子类编码去执行请求。但这并不是正确的办法，我们并不需要为每个请求定义一个 MenuItem 子类，正如我们并不需要为每个下拉菜单的文本字符串定义一个子类。再说，这种方法将请求与特定的用户界面结合起来，很难满足从不同用户界面发来的同样的请求。

假设你既能够通过下拉菜单的菜单项，又能通过 Lexi 界面底部的页图标（对短文档可能更方便一些）来到达文档的最后一页。如果我们用继承的方法将用户请求和菜单项连接起来，那么我们必须同样对待页图标或其他类似的发送该用户请求的窗口组件。这样所生成的类的数目就是窗口组件类型的数目和请求数的乘积。

现在所缺少的是一种允许我们用菜单项所执行的请求对菜单项进行参数化的机制。这种方法可以避免子类的剧增并可获得运行时刻更大的灵活性。我们可以调用一个函数来参数化一个 MenuItem，但是由于以下的至少三个原因，这还不是很完整的解决方案：

- 1) 它还没有强调撤销/重做操作。
- 2) 很难将状态和函数联系起来。例如，一个改变字体的函数需要知道是哪一种字体。
- 3) 函数很难扩充，并且很难部分地复用它们。

以上这些表明，我们应该用对象来参数化 MenuItem，而不是函数。我们可以通过继承扩充和复用请求实现。我们也可以保存状态和实现撤销/重做功能。这里是另一个封装变化概念的例子，即封装请求。我们将在 command 对象中封装每一个请求。

2.7.2 Command 类及其子类

首先我们定义一个 Command 抽象类，以提供发送请求的接口。这个基本接口由一个抽象操作“Execute”组成。Command 的子类以不同方式实现 Execute 操作，以满足不同的请求。一些子类可以将部分或全部工作委托给其他对象。另一些子类可能完全由自己来满足请求（参见图2-11）。然而对于请求者来说，一个 Command 对象就是一个 Command 对象，它们都是一致的。

现在，MenuItem 可以保存一个封装请求的 Command 对象（如图2-12）。我们给每一个菜单项一个适合该菜单项的 Command 子类实例，就像我们为每个菜单项指定一个文本字符串。当用户选中一个特定菜单项时，菜单项只是调用它的 Command 对象的 Execute 操作去执行请求。注意按钮和其他窗口组件可以用相同的方式处理请求。

^① 从概念上讲，客户就是 Lexi 用户，但实际上客户是管理用户输入的另外一个对象（如事件发送对象）

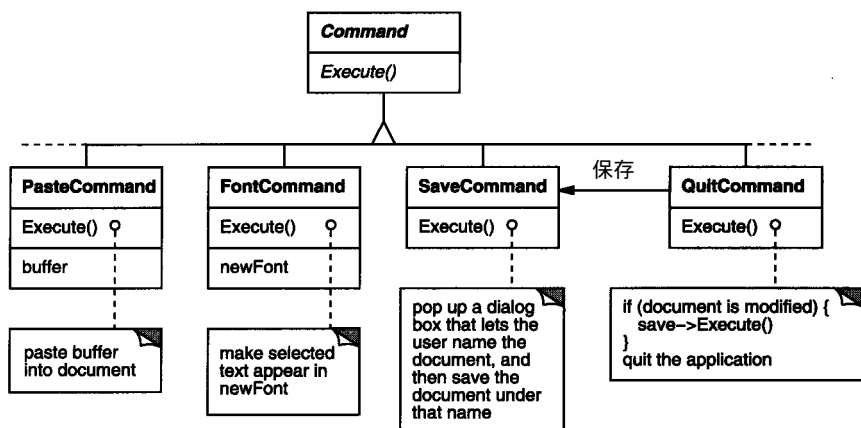


图2-11 部分Command类层次

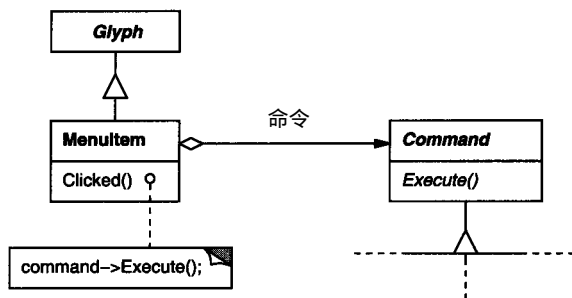


图2-12 MenuItem-Command关系

2.7.3 撤销和重做

在交互应用中撤销和重做 (Undo/redo)能力是很重要的。为了撤销和重做一个命令，我们在Command接口中增加Unexecute操作。Unexecute操作是Execute的逆操作，它使用上一次Execute操作所保存的取消信息来消除Execute操作的影响。例如，在FontCommand的例子中，Execute操作会保存改变字体的文本区域和以前的字体。FontCommand的Unexecute操作将把这个区域的文本回复为以前的字体。

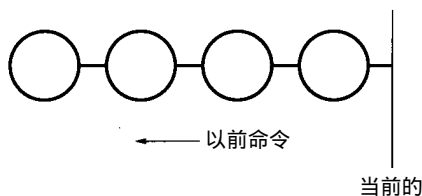
有时需要在运行时刻决定撤销和重做。如果选中文本的字体就是某个请求要修改的字体，那么这个请求是无意义的，它不会产生任何影响。假选中了一些文字，然后发一个无意义的字体改变请求。那么接下来撤销该请求会产生什么结果呢？是不是一个无意义的字体改变操作会引起撤销请求时同样做一些无意义的事？应该不是这样的。如果用户多次重复无意义的字体改变操作，他应该不必执行相同数目的撤销操作才可以返回到上一次有意义的操作。如果执行一个命令不产生任何影响，那么就不需要相应的撤销操作。

因此为了决定一个命令是否可以撤销，我们给Command接口增加了一个抽象的Reversible操作，它返回Boolean值。子类可以重定义这个操作，以根据运行情况返回true或false。

2.7.4 命令历史记录

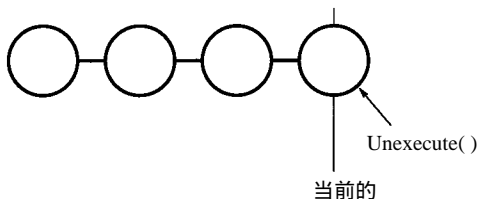
支持任意层次的撤销和重做命令的最后一步是定义一个命令历史记录 (Command

History)，或已执行的命令列表（或已被撤销的一些命令）。从概念上理解，命令的历史记录看起来有如下形状，如下图所示。

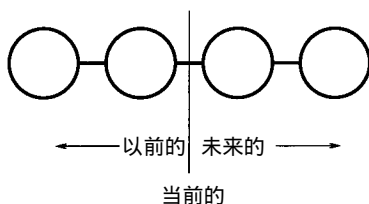


每一个圆代表一个 Command 对象。在这个例子中，用户已经发出了四条命令。最左边的命令是最先发出的，依次下来，最右边的命令是最近发出的。标有“present”的线跟踪表示最近执行（和取消）的命令。

要撤销最近命令，我们调用最右的 Command 对象的 Unexecute 操作，如下图所示。

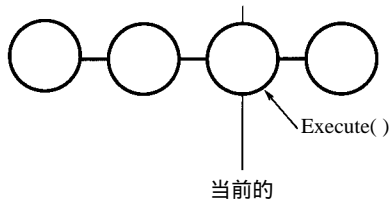


对最近命令调用 Unexecute 之后，我们将“present”线左移一个 Command 对象的距离。如果用户再次选择撤销操作，则下一个最近发送的命令以相同的方式被撤销，我们可以看到如下的状态，如下图所示。



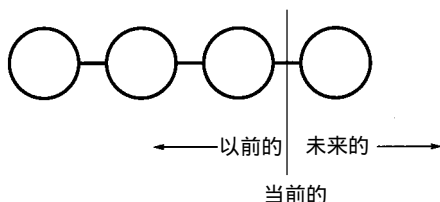
你可以看到，通过重复这个过程，我们可以进行多层次的撤销。层次数只受命令历史记录长度的限制。

要重做一个刚被撤销的命令，我们只需做上面的逆过程。在 present 线右面的命令是以后可以被重做的命令。重做刚被撤销的命令时，我们调用紧靠 present 线右边的 Command 对象的 Execute，如下图所示。



然后我们将 present 线前移，以便于接下来的重做能够调用下一个命令对象，如下图所示。

当然，如果接下来的操作不是重做而是撤销，那么 present 线左边的命令将被撤销。这样当需要从错误中恢复时，用户能有效及时地撤销和重做命令。



2.7.5 Command 模式

Lexi 的命令是 Command(5.2) 模式的应用。该模式描述了怎样封装请求，也描述了一致性的发送请求的接口，允许你配置客户端以处理不同请求。该接口保护了客户请求的实现。一个命令可以将所有或部分的请求实现委托给其他对象，也可不进行委托。这对于像 Lexi 这样必须为分散功能提供集中访问的应用来说，是相当完美的。该模式还讨论了基于基本的 Command 接口的撤销和重做机制。

2.8 拼写检查和断字处理

最后一个设计问题涉及到文本分析，这里特别指的是拼写错误的检查 and 良好格式所需的连字符点。

这里的限制条件与 2.3 节格式化设计问题的限制条件是相似的。类似于换行策略，拼写检查和连字符点的计算也存在多种方法。因此，我们能同时希望支持多个算法。一组不同算法的集合能够提供时间/空间/质量选择时的权衡，我们也希望应该能很容易加进新的算法。

我们要尽量避免将功能与文档结构紧密耦合，此时这个目标甚至比格式化设计时更重要。因为拼写检查和连字符只是我们希望 Lexi 支持的许多潜在的文本分析中的两种。不可避免的，我们可能会多次扩展 Lexi 的分析能力。我们可能会加入查找、字数统计、计算表格总值的设施、语法检查等等。但是我们并不希望在每次引入这种新功能时，都要改变 Glyph 类及其子类。

事实上这个难题可以分成两部分：1) 访问需要分析的信息，而它们是被分散在文档结构的图元中的；2) 分析这些信息。我们将这两部分分开对待。

2.8.1 访问分散的信息

许多分析要求逐字检查文本，而我们需要分析的文本是分散在图元对象的层次结构中的。为了检查在这种结构中的文本，我们需要一个知道数据结构中所包含图元对象的访问机制。一些图元可能以连接表保存它们的子图元，另一些可能用数组保存，还有一些可能使用更复杂的数据结构。我们的访问机制应该能处理所有这些可能性。

此外，更为复杂的情况是，不同分析算法将会以不同方式访问信息。大多数分析算法总是从头到尾遍历文本，但也有一些恰恰相反——例如，逆向搜索的访问顺序是从后往前的而不是从前往后。算术表达式的求值则可能需要一个中序的遍历过程。

所以我们的访问机制必须能容纳不同的数据结构，并且我们还必须支持不同的遍历方法，如前序、后序和中序。

2.8.2 封装访问和遍历

假如我们的图元的接口使用一个整型数字索引，让客户引用子图元。尽管这对以数组储存子图元的图元类来说是合理的，但对使用连接表的图元类却是低效的。图元抽象的一个重要作用就是隐藏了存储其子图元的数据结构，我们可以在不影响其他类的情况下改变图元类的数据结构。

因而，只有图元自己知道它所使用的数据结构。可以有这样的推论：图元接口不应该偏重于某个数据结构。不应该像上面这样，即数组比使用连接表更好。

我们有可能解决这个问题，并且同时支持多种遍历方式。我们可以将多个访问和遍历功能直接放到图元类中，并提供一种选择方式，这可能是通过增加一个枚举常量作为参数。类在遍历过程中传递该参数以确保所用的是同一种遍历方式，它们必须传递遍历过程中积累的任何信息。

我们可以给 Glyph 的接口增加如下的抽象操作来支持这种方法：

```
void First(Traversal kind)
void Next()
bool IsDone()
Glyph* GetCurrent()
void Insert(Glyph*)
```

First、Next 和 IsDone 操作控制遍历。First 初始化遍历过程，它根据枚举类型 Traversal 的参数值确定执行何种遍历，其值可以是 CHILDREN（只遍历图元的直接子图元）、PREORDER（以先序方式遍历整个结构）、POSTORDER 和 INORDER。Next 在遍历时前进到下一个图元。IsDone 则报告遍历是否完成。GetCurrent 代替了 Child 操作，它访问遍历的当前图元。Insert 操作代替了以前的操作，它在当前位置插入给定的图元。

一个分析可以使用如下 C++ 代码实现对 g 为根结点的图元结构作先序遍历：

```
Glyph* g;

for (g->First(PREORDER); !g->IsDone(); g->Next()) {
    Glyph* current = g->GetCurrent();

    // do some analysis
}
```

注意我们已经放弃了图元接口的数字索引。这样就不会偏重于某种数据结构。我们也使得客户不必自己实现通用的遍历方法。

但是该方法仍然有一些问题。举个例子，它在不扩展枚举值或增加新的操作的条件下，不能支持新的遍历方式。比方说，我们想要修改一下先序遍历，使它能自动跳过非文本图元。我们就不得不改变枚举类型 Traversal，使它包含 TEXTUAL_PREORDER 这样的值。

我们最好避免改变已存在的说明。把遍历机制完全放到 Glyph 类层次中，将会导致修改和扩充时不得不改变一些类。也使得复用遍历机制遍历其他对象结构时很困难，并且在一个结构不能同时进行多个遍历。

再一次提及，一个好的解决方案是封装那些变化的概念，在本例中我们指的是访问和遍历机制。我们引入一类称之为 iterators 的对象，它们的目的是定义这些机制的不同集合。我们可以通过继承来统一访问不同的数据结构和支持新的遍历方式，同时不改变图元接口或打

乱已有的图元实现。

2.8.3 Iterator类及其子类

我们使用抽象类 `Iterator` 为访问和遍历定义一个通用的接口。由具体子类，诸如 `ArrayIterator` 和 `ListIterator`，负责实现该接口以提供对数组和列表的访问；而 `PreorderIterator` 和 `PostorderIterator` 以及类似的类在指定结构上实现不同的遍历方式。每个 `Iterator` 子类有一个它所遍历的结构的引用，在创建子类实例时，需用这个引用进行初始化。图 2-13 展示了 `Iterator` 和它的若干子类之间的关系。注意，我们在 `Glyph` 类接口中增加了一个 `CreateIterator` 抽象操作以支持 `Iterator`。

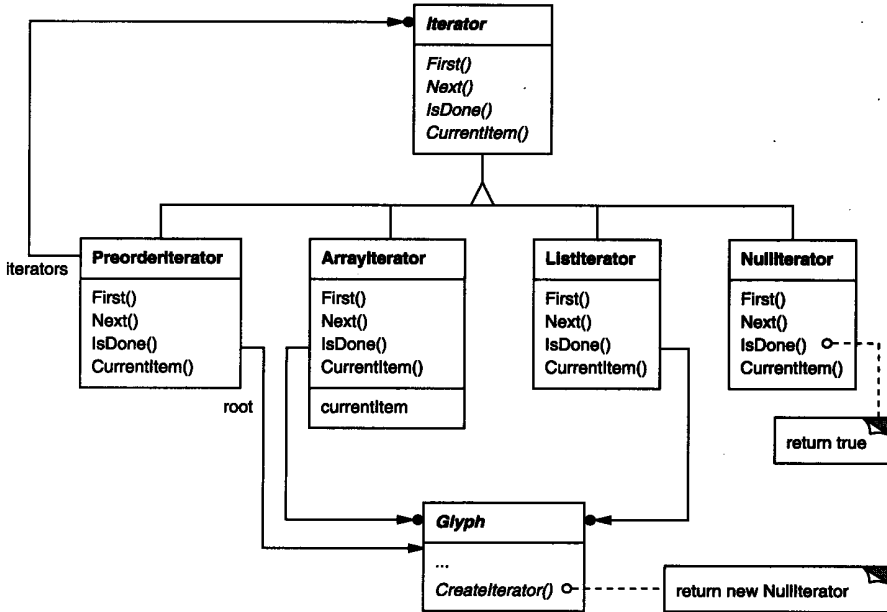


图2-13 Iterator类和它的子类

`Iterator` 接口提供 `First`、`Next` 和 `IsDone` 操作来控制遍历。`ListIterator` 类实现的 `First` 操作指向列表的第一个元素；`Next` 前进到列表的下一个元素；`IsDone` 返回列表指针是否指向列表范围以外；`CurrentItem` 返回 `iterator` 所指的图元。`ArrayIterator` 类的实现类似，只不过它是针对一个图元数组的。

现在我们无需知道具体表示也能访问一个图元结构的子女：

```
Glyph* g;
Iterator<Glyph*> i = g->CreateIterator();

for (i->First(); !i->IsDone(); i->Next()) {
    Glyph* child = i->CurrentItem();

    // do something with current child
}
```

在缺省情况下 `CreateIterator` 返回一个 `NullIterator` 实例。`NullIterator` 是一个退化的 `Iterator`，它适用于叶子图元，即没有子图元的图元。`NullIterator` 的 `IsDone` 操作总返回 `true`。

一个有子女的图元 Glyph 子类将重载 CreateIterator，返回不同 Iterator 子类的一个实例，这依赖于保存图元子女所用的结构。如果 Glyph 的行子类在一个 _children 列表中保存其子类，那么它的 CreateIterator 操作实现如下：

```
Iterator<Glyph*>* Row::CreateIterator () {
    return new ListIterator<Glyph*>(_children);
}
```

用于先序和中序遍历的 Iterator 是用各图元自身特定的 iterator 实现的。这些遍历的 Iterator 还要保存对以它们所遍历的结构根图元的引用。它们调用结构中图元的 CreateIterator，并用栈来保存返回的 Iterator。

例如，类 PreorderIterator 从根图元得到 Iterator，将它初始化为指向第一个元素，然后将它压入栈中：

```
void PreorderIterator::First () {
    Iterator<Glyph*>* i = _root->CreateIterator();

    if (i) {
        i->First();
        _iterators.RemoveAll();
        _iterators.Push(i);
    }
}
```

CurrentItem 只是调用栈顶的 Iterator 的 CurrentItem 操作：

```
Glyph* PreorderIterator::CurrentItem () const {
    return
        _iterators.Size() > 0 ?
        _iterators.Top()->CurrentItem() : 0;
}
```

Next 操作得到栈顶的 Iterator，并且让它的当前项创建一个 Iterator，尽可能遍历到图元结构的最远处（因为这是一个先序遍历）。Next 将新的 Iterator 设置到遍历中的第一个元素，再将它压栈。然后 Next 测试最近的 Iterator，如果它的 IsDone 操作返回 true，那么我们就完成了对当前子树（或叶子）的遍历。本例中，Next 弹出栈顶的 Iterator 并且重复上述过程，直到发现下一个还没完成的遍历；否则，我们就完成了对整个结构的遍历。

```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();

    i->First();
    _iterators.Push(i);

    while (
        _iterators.Size() > 0 && _iterators.Top()->IsDone()
    ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

注意 Iterator 类层次结构是怎样允许我们不改变图元类而增加新的遍历方式的——如 PreorderIterator 所示，我们只需创建 Iteraror 子类，并给它增加一个新的遍历算法即可。 Glyph

子类给客户相同的接口去访问它们的子女，并不揭示其底层的数据结构。由于 `Iterator` 保存了自己的遍历状态，所以我们可以同时执行多个遍历，甚至可以对相同的结构进行同时遍历。尽管我们在本例中的遍历是针对图元结构的，但我们没有理由不可以将像 `PreorderIterator` 这样的类参数化，使其能遍历其他类型的对象结构。我们可以使用 C++ 的模板技术来做这件事，这样我们在遍历其他结构时就能复用 `PreorderIterator` 的机制。

2.8.4 `Iterator` 模式

`Iterator(5.4)` 模式描述了那些支持访问和遍历对象结构的技术，它不仅可用于组合结构也可用于集合。该模式抽象了遍历算法，对客户隐藏了它所遍历对象的内部结构。`Iterator` 模式再一次说明了怎样封装变化的概念，有助于我们获得灵活性和复用性。尽管如此，`Iteration` 问题的复杂性还是令人吃惊的，`Iterator` 模式包含了比我们这里考虑的更多的细微差别和权衡。

2.8.5 遍历和遍历过程中的动作

现在我们有遍历图元结构的方法，可以进行检查拼写和支持连字符。这两种分析都涉及到了遍历过程中的信息累积。

首先我们要决定将分析的责任放在什么位置上。我们可以在 `Iterator` 类中作分析，将分析和遍历有机结合起来。但是如果我们能区别遍历和遍历过程中所执行动作之间的差别的话，就可以得到更多的灵活性和潜在复用性，这是因为不同的分析通常需要相同的遍历方式。因而，对于不同的分析而言，我们可以复用相同的 `Iterator` 集合。例如，先序遍历对于许多分析，包括拼写检查、连字符、向前搜索和字数统计等，都是通用的。

因此，我们应当将分析和遍历分开，那么将分析责任放到什么地方呢？我们知道有许多种分析可以做，每一种分析将在不同的遍历点做不同的事情。根据分析的种类，有些 `Glyph` 比其他的图元更具重要性。如果作拼写检查和连字符分析，我们要考虑的是字符型的图元，而不是像行和位图图形这样的图元。如果我们作颜色分割，我们要考虑的是可见的图元，而不是不可见图元。因此，不同的分析过程必然是分析不同的图元。

因而一个给定的分析必须能区别不同种类的图元。很明显的一种做法是将分析能力放到图元类本身。针对每一种分析，我们为 `Glyph` 类增加一个或多个抽象操作，并且根据它们在分析中所起作用，在 `Glyph` 子类中实现这些操作。

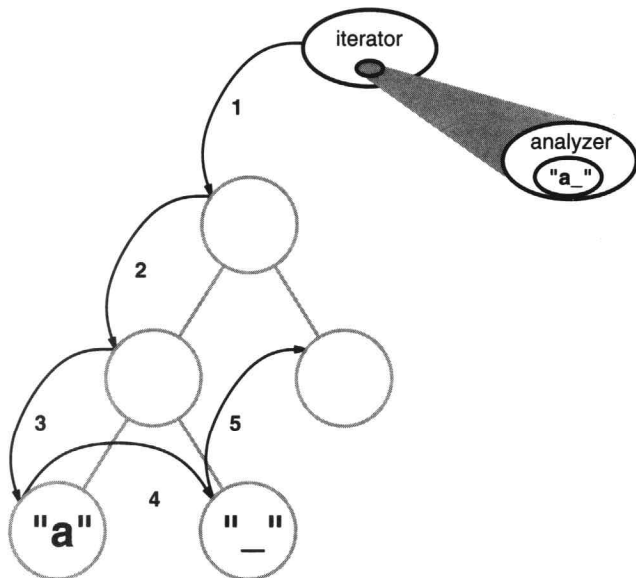
但麻烦的是我们每增加了一种新的分析，都必须改变每一个图元类。某些情况下使这个问题可以简化：有时只有部分类参与分析。又如有时大多数类都以相同方式去做分析，那么我们可以为 `Glyph` 类中的抽象操作补充一个缺省的实现。该缺省操作将包含许多通用情况。这样我们可以将修改只限于 `Glyph` 类和那些非标准子类。

然而即使缺省实现可以减少需要修改的类的数目，一个隐含的问题依然存在：随着新的分析功能的增加，`Glyph` 的接口会越来越大。众多的分析操作会逐渐模糊基本的 `Glyph` 接口，从而很难看出图元的主要目的是定义和结构化那些有外观和形状的对象——这些接口完全被淹没了。

2.8.6 封装分析

所有迹象表明，我们需要在一个独立对象中封装分析方法，就像我们以前多次做过的那

样。我们可以将一个给定的分析封装在一个类中，并把该类的实例和合适的 Iterator 结合起来使用。这个 Iterator 负责将该实例携带到所遍历结构的每一个图元中。这样分析对象可以在每个遍历点做一些分析工作。在遍历过程中，分析者积累它所感兴趣的信息（本例中指字符信息），如下图所示。



该方法的基本问题在于：分析对象怎样才能不使用类型测试或强制类型转换也能正确对待各种不同的图元。我们不希望 SpellingChecker 包含类似如下的(伪)代码：

```
void SpellingChecker::Check (Glyph* glyph) {
    Character* c;
    Row* r;
    Image* i;

    if (c = dynamic_cast<Character*>(glyph)) {
        // analyze the character
    } else if (r = dynamic_cast<Row*>(glyph)) {
        // prepare to analyze r's children
    } else if (i = dynamic_cast<Image*>(glyph)) {
        // do nothing
    }
}
```

这段代码相当拙劣。它依赖于比较高深的像类型的安全转换这样的能力，并且难以扩展。无论何时当我们改变 Glyph 类层次时，都要记住修改这个函数。事实上，这也是面向对象语言力图消除的那种代码。

我们如何避免这种不成熟的方式呢？我们在 Glyph 类中添加如下代码时会发生什么：

```
void CheckMe (SpellingChecker&)
```

我们在每一个 Glyph 子类中定义 CheckMe 如下：

```
void GlyphSubclass::CheckMe (SpellingChecker& checker) {
    checker.CheckGlyphSubclass(this);
}
```



```
}

```

这里的 GlyphSubclass 将会被图元子类的名字所代替。注意当调用 CheckMe 时，当前是哪一个特定 Glyph 子类是知道的——毕竟，我们在使用它的操作。相对应的，SpellingChecker 类的接口包含每一个 Glyph 子类的类似于 CheckGlyphSubclass 的操作^①：

```
class SpellingChecker {
public:
    SpellingChecker();

    virtual void CheckCharacter(Character*);
    virtual void CheckRow(Row*);
    virtual void CheckImage(Image*);

    // ... and so forth

    List<char*> GetMisspellings();

protected:
    virtual bool IsMisspelled(const char*);

private:
    char _currentWord[MAX_WORD_SIZE];
    List<char*> _misspellings;
};

```

SpellingChecker 的检查字符图元的操作可能像如下所示：

```
void SpellingChecker::CheckCharacter (Character* c) {
    const char ch = c->GetCharCode();

    if (isalpha(ch)) {
        // append alphabetic character to _currentWord
    } else {
        // we hit a nonalphabetic character

        if (IsMisspelled(_currentWord)) {
            // add _currentWord to _misspellings
            _misspellings.Append(strdup(_currentWord));
        }

        _currentWord[0] = '\0';
        // reset _currentWord to check next word
    }
}

```

注意我们已经在 Character 类中定义了一个特殊的 GetCharCode 操作。拼写检查者能够处理特定子类的操作，而无需类型检查或转换——这让我们可以分别对待各个对象。

CheckCharacter 将字母字符累积在 _CurrentWord 数组中。当碰到像下划线这样的非字母字符时，它使用 IsMisspelled 操作去检查 _CurrentWord 中单词的拼写^②。如果该单词拼写错误，

① 我们可以使用函数重载来给每一个这样的成员函数以相同的名字，因为它们的参数已经将它们区分开了。我们这里给它们不同名字是为了强调它们的不同性，尤其当调用它们的时候。

② IsMisspelled 实现了拼写算法，因为它独立于 Lexi 的设计，所以这里我们就不细说。我们这里通过子类 SpellingChecker 来支持不同的算法；但也可以使用 Strategy 模式来支持不同的拼写检查算法（就像在 2.3 节中格式化时所做的那样）。

CheckCharacter将它加到拼错单词的列表中。然后必须清空数组 _CurrentWord，以便检查下一个单词。当遍历结束后，你可以通过 GetMisspellings操作遍历拼写错误的单词的列表。

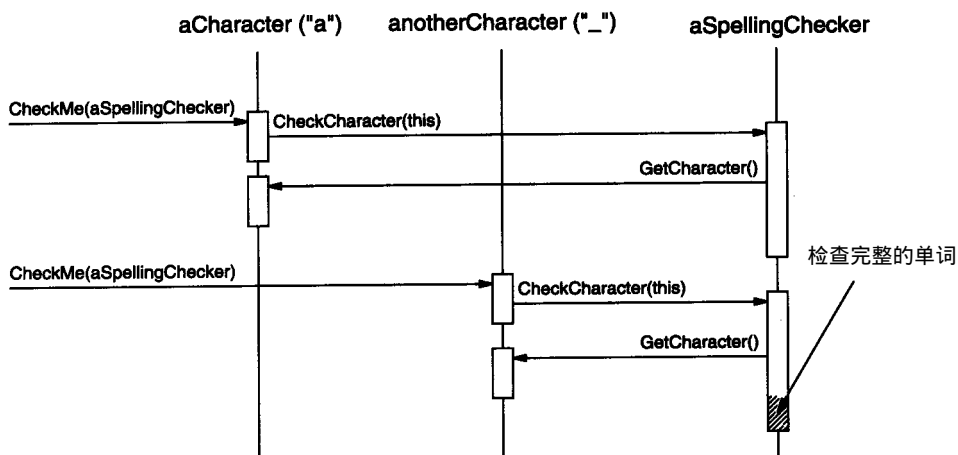
现在，我们以拼写检查器为参数调用每个图元的 CheckMe操作，来实现对图元结构的遍历。这使得拼写检查器 SpellingChecker可以有效区分每个图元，并不断推进检查器以检查下面的内容。

```
SpellingChecker spellingChecker;
Composition* c;

// ...

Glyph* g;
PreorderIterator i(c);
for (i.First(); !i.IsDone(); i.Next()) {
    g = i.CurrentItem();
    g->CheckMe(spellingChecker);
}
```

下面的交互图展示了 Character图元和SpellingChecker对象是怎样协同工作的：



这种方法适合于找出拼写错误，但怎样才能帮助我们去支持多种分析呢？看上去有点像我们每增加一种新的分析，就不得不为 Glyph及其子类增加一个类似于 CheckMe(SpellingChecker&) 的操作。如果我们坚持每一个分析对应一个独立的类的话，事实确实如此。但是没有理由说我们不能给所有分析类型一个相同的接口。应该允许我们多态使用各种分析。也就是说，我们应能够用一个有通用参数的与分析无关的操作来替代像 CheckMe(SpellingChecker&) 这样表示特定分析的操作。

2.8.7 Visitor类及其子类

我们使用术语访问者（visitor）来泛指在遍历过程中“访问”被遍历对象并做适当操作的一类对象^①。本例中我们使用一个 Visitor类来定义一个访问结构中图元的接口。

```
class Visitor {
public:
```

① “访问”只是一个比“分析”稍微通用一点的术语。它显示了我们在设计模式中所使用的术语。

```
virtual void VisitCharacter(Character*) { }
virtual void VisitRow(Row*) { }
virtual void VisitImage(Image*) { }

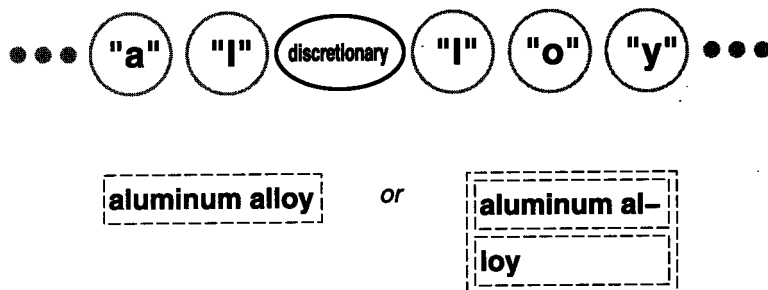
// ... and so forth
};
```

访问者的具体子类做不同的分析，例如，我们可以用一个 SpellingCheckingVisitor 子类来检查拼写；用 HyphenationVisitor 子类做连字符分析。SpellingCheckingVisitor 可以就像我们上面的 SpellingChecker 那样来实现，只是操作名要反映通用的访问者的类接口。例如，CheckCharacter 应该改成 VisitCharacter。

既然 CheckMe 对于访问者并不合适，因为访问者不检查任何东西。故我们要使用一个更加通用的名字：Accept，其参数也应该改成 Visitor&，以反映它能接受任何一个访问者这一事实。现在定义一个新的分析只需要定义一个新的 Visitor 子类——我们无需触及任何图元类。通过在 Glyph 及其子类中增加这一操作，我们就可以支持以后的所有分析方法。

我们已经看到怎样做拼写检查了。我们可以在 HyphenationVisitor 中使用类似的方法来累积文本，但一旦 HyphenationVisitor 的 VisitCharacter 操作作用于处理整个单词，它的工作方式将略有不同。它并不是检查单词的拼写错误，而是使用一个连字符算法决定单词可能的连字符点的位置（如果有的话）。然后在每一个连字符点，插入一个 Discretionary 图元。Discretionary 图元是 Glyph 子类 Discretionary 的实例。

一个 Discretionary 图元有两种可能的外观，这决定于它是否是一行的最后一个字符。如果是最后一个字符，那么 Discretionary 看起来像一个连字符；如果不是，那么 Discretionary 不显示任何东西。Discretionary 检查它的父对象（一个行对象）来判断它是否是最后的子女。Discretionary 在每次被激活画自己或计算它的边界时，都要作这个检查。格式化策略将 Discretionary 看成空格，将它们都作为行结束的标志。下图说明了一个嵌入的 Discretionary 是怎样显示的。



2.8.8 Visitor 模式

我们这里所描述的是 Visitor 模式的一个应用。前面的 Visitor 类及其子类是该模式的主要参与者。Visitor 模式记述了这样一种我们前面已使用过的技术，它允许对图元结构所作分析的数目不受限制地增加而不必改变图元类本身。访问者类的另一个优点是它不局限使用于像图元结构这样的组合者，也适用于其他任何对象结构。包括集合、列表，甚至无环有向图。再者，访问者所能访问的类之间无需通过一个公共父类关联起来。也就是说，访问者能跨越类层次结构。

在使用 Visitor 模式之前你要问自己的一个重要问题是：哪一个类层次变化得最厉害？该模式最适合于当你想对一个稳定类结构的对象做许多不同的事情的情况。增加一种新的访问者而不需要改变类结构，这对于很大的类结构是尤其重要的。但是，只要你给类结构增加了一个子类，你就不得不更新你所有访问者类的接口以包含针对那个子类的 Visit... 操作。

比如，在我们的例子中，增加一个被称为 Foo 的新 Glyph 子类，将需要改变 Visitor 及其子类，以包含一个 VisitFoo 操作。但是考虑到我们的设计限制条件，我们比较多的是为 Lexi 增加一种新的分析方法，而不是增加一种新的图元。所以 Visitor 模式是适合我们的需要的。

2.9 小结

我们在 Lexi 的设计中使用了 8 种不同的模式：

- 1) Composite (4.3) 表示文档的物理结构。
- 2) Strategy (5.9) 允许不同的格式化算法。
- 3) Decorator (4.4) 修饰用户界面。
- 4) Abstract Factory (3.1) 支持多视感标准。
- 5) Bridge (4.2) 允许多个窗口平台。
- 6) Command (5.2) 支持撤销用户操作。
- 7) Iterator (5.4) 访问和遍历对象结构。
- 8) Visitor (5.11) 允许无限扩充分析能力而又不会使文档结构的实现复杂化。

以上这些设计要点都不仅仅局限于像 Lexi 这样的文档编辑应用。事实上，很多重要的应用都可以使用这些模式处理不同的事情。一个财务分析应用可能使用 Composite 定义由多种类型子文件夹组成的投资文件夹。一个编译程序可能使用 Strategy 模式来考虑不同目标机上的寄存器分配方案。图形界面的应用可能是至少要用到 Decorator 和 Command 模式，正如本例所示。

我们已经涉及到了 Lexi 设计中的一些主要问题，但还有很多其他的问题我们没有讨论。需再次说明的是，本书描述的不仅是以上我们所用到的 8 个模式。所以在学习其余模式时，你要考虑怎样才能把它们用在 Lexi 中。最好能考虑在你自己的设计中怎样使用它们。