

## 第4章 结构型模式

结构型模式涉及到如何组合类和对象以获得更大的结构。结构型类模式采用继承机制来组合接口或实现。一个简单的例子是采用多重继承方法将两个以上的类组合成一个类，结果这个类包含了所有父类的性质。这一模式尤其有助于多个独立开发的类库协同工作。另外一个例子是类形式的 Adapter(4.1)模式。一般来说，适配器使得一个接口 (adaptee的接口) 与其他接口兼容，从而给出了多个不同接口的统一抽象。为此，类适配器对一个 adaptee类进行私有继承。这样，适配器就可以用 adaptee的接口表示它的接口。

结构型对象模式不是对接口和实现进行组合，而是描述了如何对一些对象进行组合，从而实现新功能的一些方法。因为可以在运行时刻改变对象组合关系，所以对对象组合方式具有更大的灵活性，而这种机制用静态类组合是不可能实现的。

Composite (4.3) 模式是结构型对象模式的一个实例。它描述了如何构造一个类层次式结构，这一结构由两种类型的对象（基元对象和组合对象）所对应的类构成，其中的组合对象使得你可以组合基元对象以及其他的组合对象，从而形成任意复杂的结构。在 Proxy (4.7) 模式中，proxy对象作为其他对象的一个方便的替代或占位符。它的使用可以有多种形式。例如它可以在局部空间中代表一个远程地址空间中的对象，也可以表示一个要求被加载的较大的对象，还可以用来保护对敏感对象的访问。Proxy模式还提供了对对象的一些特有性质的一定程度上的间接访问，从而它可以限制、增强或修改这些性质。

Flyweight(4.6)模式为了共享对象定义了一个结构。至少有两个原因要求对象共享：效率和一致性。Flyweight的对象共享机制主要强调对象的空间效率。使用很多对象的应用必需考虑每一个对象的开销。使用对象共享而不是进行对象复制，可以节省大量的空间资源。但是仅当这些对象没有定义与上下文相关的状态时，它们才可以被共享。Flyweight的对象没有这样的状态。任何执行任务时需要的其他一些信息仅当需要时才传递过去。由于不存在与上下文相关的状态，因此Flyweight对象可以被自由地共享。

如果说Flyweight模式说明了如何生成很多较小的对象，那么 Facade(4.5)模式则描述了如何用单个对象表示整个子系统。模式中的 facade用来表示一组对象，facade的职责是将消息转发给它所表示的对象。Bridge(4.2)模式将对象的抽象和其实现分离，从而可以独立地改变它们。

Decorator(4.4)模式描述了如何动态地为对象添加职责。Decorator模式是一种结构型模式。这一模式采用递归方式组合对象，从而允许你添加任意多的对象职责。例如，一个包含用户界面组件的Decorator对象可以将边框或阴影这样的装饰添加到该组件中，或者它可以将窗口滚动和缩放这样的功能添加的组件中。我们可以将一个Decorator对象嵌套在另外一个对象中就可以很简单地增加两个装饰，添加其他的装饰也是如此。因此，每个Decorator对象必须与其组件的接口兼容并且保证将消息传递给它。Decorator模式在转发一条信息之前或之后都可以完成它的工作（比如绘制组件的边框）。

许多结构型模式在某种程度上具有相关性，我们将在本章末讨论这些关系。

## 4.1 ADAPTER（适配器）——类对象结构型模式

### 1. 意图

将一个类的接口转换成客户希望的另外一个接口。Adapter模式使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

### 2. 别名

包装器 Wrapper。

### 3. 动机

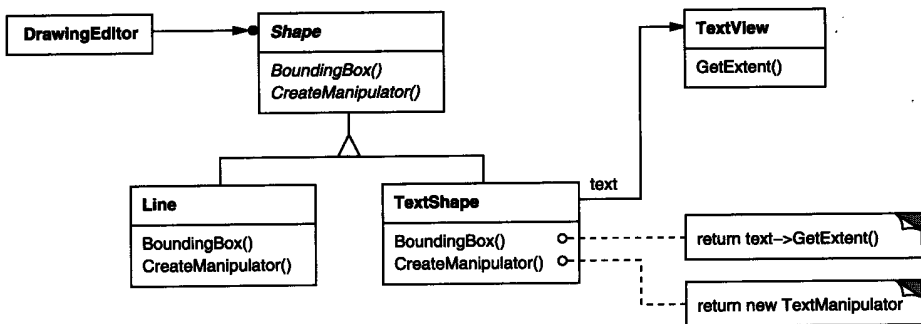
有时，为复用而设计的工具箱类不能够被复用的原因仅仅是因为它的接口与专业应用领域所需要的接口不匹配。

例如，有一个绘图编辑器，这个编辑器允许用户绘制和排列基本图元（线、多边形和正文等）生成图片和图表。这个绘图编辑器的关键抽象是图形对象。图形对象有一个可编辑的形状，并可以绘制自身。图形对象的接口由一个称为 Shape 的抽象类定义。绘图编辑器为每一种图形对象定义了一个 Shape 的子类：LineShape 类对应于直线，PolygonShape 类对应于多边形，等等。

像 LineShape 和 PolygonShape 这样的基本几何图形的类比较容易实现，这是由于它们的绘图和编辑功能本来就很有限。但是对于可以显示和编辑正文的 TextShape 子类来说，实现相当困难，因为即使是基本的正文编辑也要涉及到复杂的屏幕刷新和缓冲区管理。同时，成品的用户界面工具箱可能已经提供了一个复杂的 TextView 类用于显示和编辑正文。理想的情况是我们可以复用这个 TextView 类以实现 TextShape 类，但是工具箱的设计者当时并没有考虑 Shape 的存在，因此 TextView 和 Shape 对象不能互换。

一个应用可能会有些类具有不同的接口并且这些接口互不兼容，在这样的应用中象 TextView 这样已经存在并且不相关的类如何协同工作呢？我们可以改变 TextView 类使它兼容 Shape 类的接口，但前提是必须有这个工具箱的源代码。然而即使我们得到了这些源代码，修改 TextView 也是没有什么意义的；因为不应该仅仅为了实现一个应用，工具箱就不得不采用一些与特定领域相关的接口。

我们可以不用上面的方法，而定义一个 TextShape 类，由它来适配 TextView 的接口和 Shape 的接口。我们可以用两种方法做这件事：1) 继承 Shape 类的接口和 TextView 的实现，或 2) 将一个 TextView 实例作为 TextShape 的组成部分，并且使用 TextView 的接口实现 TextShape。这两种方法恰恰对应于 Adapter 模式的类和对象版本。我们将 TextShape 称之为适配器 Adapter。



上面的类图说明了对象适配器实例。它说明了在 Shape类中声明的 BoundingBox请求如何被转换成在 TextView类中定义的 GetExtent请求。由于 TextShape将 TextView的接口与 Shape的接口进行了匹配，因此绘图编辑器就可以复用原先并不兼容的 TextView类。

Adapter时常还要负责提供那些被匹配的类所没有提供的功能，上面的类图中说明了适配器如何实现这些职责。由于绘图编辑器允许用户交互的将每一个 Shape对象“拖动”到一个新的位置，而 TextView设计中没有这种功能。我们可以实现 TextShape类的 CreateManipulator操作，从而增加这个缺少的功能，这个操作返回相应的 Manipulator子类的一个实例。

Manipulator是一个抽象类，它所描述的对象知道如何驱动 Shape类响应相应的用户输入，例如将图形拖动到一个新的位置。对应于不同形状的图形， Manipulator有不同的子类；例如子类 TextManipulator对应于 TextShape。TextShape通过返回一个 TextManipulator实例，增加了 TextView中缺少而 Shape需要的功能。

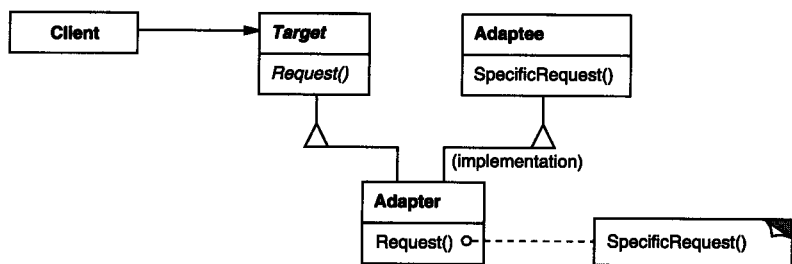
#### 4. 适用性

以下情况使用 Adapter模式

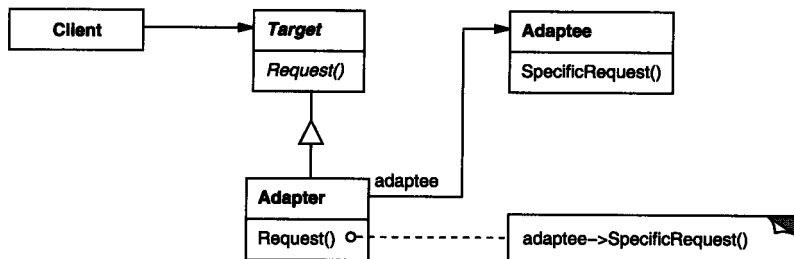
- 你想使用一个已经存在的类，而它的接口不符合你的需求。
- 你想创建一个可以复用的类，该类可以与其他不相关的类或不可预见的类（即那些接口可能不一定兼容的类）协同工作。
- （仅适用于对象 Adapter）你想使用一些已经存在的子类，但是不可能对每一个都进行子类化以匹配它们的接口。对象适配器可以适配它的父类接口。

#### 5. 结构

类适配器使用多重继承对一个接口与另一个接口进行匹配，如下图所示。



对象适配器依赖于对象组合，如下图所示。



#### 6. 参与者

- Target (Shape)
  - 定义 Client使用的与特定领域相关的接口。
- Client (DrawingEditor)

— 与符合Target接口的对象协同。

- Adaptee (TextView)

— 定义一个已经存在的接口，这个接口需要适配。

- Adapter (TextShape)

— 对Adaptee的接口与Target接口进行适配

## 7. 协作

- Client在Adapter实例上调用一些操作。接着适配器调用 Adaptee的操作实现这个请求。

## 8. 效果

类适配器和对象适配器有不同的权衡。类适配器

- 用一个具体的 Adapter类对 Adaptee和Target进行匹配。结果是当我们想要匹配一个类以及所有它的子类时，类 Adapter将不能胜任工作。
- 使得Adapter可以重定义 Adaptee的部分行为，因为 Adapter是Adaptee的一个子类。
- 仅仅引入了一个对象，并不需要额外的指针以间接得到 adaptee。

对象适配器则

- 允许一个 Adapter与多个 Adaptee——即Adaptee本身以及它的所有子类（如果有子类的话）——同时工作。Adapter也可以一次给所有的 Adaptee添加功能。
- 使得重定义 Adaptee的行为比较困难。这就需要生成 Adaptee的子类并且使得Adapter引用这个子类而不是引用Adaptee本身。

使用Adapter模式时需要考虑的其他一些因素有：

1) Adapter的匹配程度 对Adaptee的接口与Target的接口进行匹配的工作量各个 Adapter可能不一样。工作范围可能是，从简单的接口转换(例如改变操作名)到支持完全不同的操作集合。Adapter的工作量取决于Target接口与Adaptee接口的相似程度。

2) 可插入的 Adapter 当其他的类使用一个类时，如果所需的假定条件越少，这个类就更具可复用性。如果将接口匹配构建为一个类，就不需要假定对其他的类可见的是一个相同的接口。也就是说，接口匹配使得我们可以将自己的类加入到一些现有的系统中去，而这些系统对这个类的接口可能会有所不同。Object-Work/Smalltalk[Par90]使用pluggable adapter一词描述那些具有内部接口适配的类。

考虑TreeDisplay窗口组件，它可以图形化显示树状结构。如果这是一个具有特殊用途的窗口组件，仅在一个应用中使用，我们可能要求它所显示的对象有一个特殊的接口，即它们都是抽象类Tree的子类。如果我们希望使 TreeDisplay有具有良好的复用性的话（比如说，我们希望将它作为可用窗口组件工具箱的一部分），那么这种要求将是不合理的。应用程序将自己定义树结构类，而不应一定要使用我们的抽象类 Tree。不同的树结构会有不同的接口。

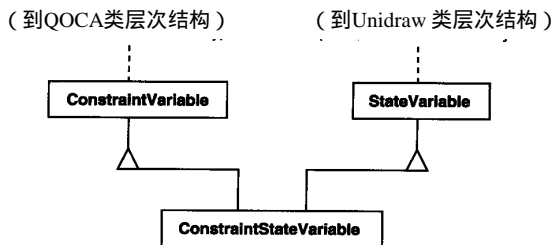
例如，在一个目录层次结构中，可以通过 GetSubdirectories操作进行访问子目录，然而在一个继承式层次结构中，相应的操作可能被称为 GetSubclasses。尽管这两种层次结构使用的接口不同，一个可复用的 TreeDisplay窗口组件必须能显示所有这两种结构。也就是说，TreeDisplay应具有接口适配的功能。

我们将在实现一节讨论在类中构建接口适配的多种方法。

3) 使用双向适配器提供透明操作 使用适配器的一个潜在问题是，它们不对所有的客户都透明。被适配的对象不再兼容 Adaptee的接口，因此并不是所有 Adaptee对象可以被使用的

地方它都可以被使用。双向适配器提供了这样的透明性。在两个不同的客户需要用不同的方式查看同一个对象时，双向适配器尤其有用。

考虑一个双向适配器，它将图形编辑框架 Unidraw [VL90] 与约束求解工具箱 QOCA [HHMV92] 集成起来。这两个系统都有一些类，这些类显式地表示变量：Unidraw 含有类 StateVariable，QOCA 中含有类 ConstraintVariable，如下图所示。为了使 Unidraw 与 QOCA 协同工作，必须首先使类 ConstraintVariable 与类 StateVariable 相匹配；而为了将 QOCA 的求解结果传递给 Unidraw，必须使 StateVariable 与 ConstraintVariable 相匹配。



这一方案中包含了一个双向适配器 ConstraintStateVariable，它是类 ConstraintVariable 与类 StateVariable 共同的子类，ConstraintStateVariable 使得两个接口互相匹配。在该例中多重继承是一个可行的解决方案，因为被适配类的接口差异较大。双向适配器与这两个被匹配的类都兼容，在这两个系统中它都可以工作。

## 9. 实现

尽管 Adapter 模式的实现方式通常简单直接，但是仍需要注意以下一些问题：

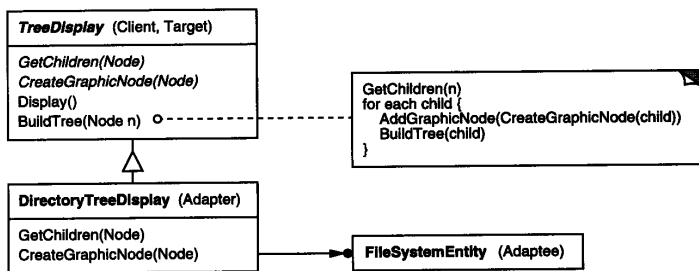
1) 使用 C++ 实现适配器类 在使用 C++ 实现适配器类时，Adapter 类应该采用公共方式继承 Target 类，并且用私有方式继承 Adaptee 类。因此，Adapter 类应该是 Target 的子类型，但不是 Adaptee 的子类型。

2) 可插入的适配器 有许多方法可以实现可插入的适配器。例如，前面描述的 TreeDisplay 窗口组件可以自动的布置和显示层次式结构，对于它有三种实现方法：

首先（这也是所有这三种实现都要做的）是为 Adaptee 找到一个“窄”接口，即可用于适配的最小操作集。因为包含较少操作的窄接口相对包含较多操作的宽接口比较容易进行匹配。对于 TreeDisplay 而言，被匹配的对象可以是任何一个层次式结构。因此最小接口集合仅包含两个操作：一个操作定义如何在层次结构中表示一个节点，另一个操作返回该节点的子节点。

对这个窄接口，有以下三个实现途径：

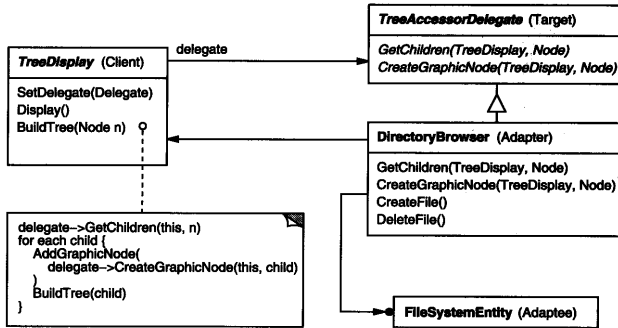
a) 使用抽象操作 在 TreeDisplay 类中定义窄 Adaptee 接口相应的抽象操作。这样就由子类来实现这些抽象操作并匹配具体的树结构的对象。例如，DirectoryTreeDisplay 子类将通过访问目录结构实现这些操作，如下图所示。





DirectoryTreeDisplay对这个窄接口加以特化，使得它的 DirectoryBrowser客户可以用它来显示目录结构。

b) 使用代理对象 在这种方法中，TreeDisplay将访问树结构的请求转发到代理对象。TreeDisplay的客户进行一些选择，并将这些选择提供给代理对象，这样客户就可以对适配加以控制，如下图所示。



例如，有一个DirectoryBrowser，它像前面一样使用TreeDisplay。DirectoryBrowser可能为匹配TreeDisplay和层次目录结构构造出一个较好的代理。在Smalltalk或Objective C这样的动态类型语言中，该方法只需要一个接口对适配器注册代理即可。然后TreeDisplay简单地将请求转发给代理对象。NEXTSTEP[Add94]大量使用这种方法以减少子类化。

在C++这样的静态类型语言中，需要一个代理的显式接口定义。我们将TreeDisplay需要的窄接口放入纯虚类TreeAccessorDelegate中，从而指定这样的接口。然后我们可以运用继承机制将这个接口融合到我们所选择的代理中——这里我们选择DirectoryBrowser。如果DirectoryBrowser没有父类我们将采用单继承，否则采用多继承。这种将类融合在一起的方法相对于引入一个新的TreeDisplay子类并单独实现它的操作的方法要容易一些。

c) 参数化的适配器 通常在Smalltalk中支持可插入适配器的方法是，用一个或多个模块对适配器进行参数化。模块构造支持无子类化的适配。一个模块可以匹配一个请求，并且适配器可以为每个请求存储一个模块。在本例中意味着，TreeDisplay存储的一个模块用来将一个节点转化成为一个GraphicNode，另外一个模块用来存取一个节点的子节点。

例如，当对一个目录层次建立TreeDisplay时，我们可以这样写：

```

directoryDisplay :=
    (TreeDisplay on: treeRoot)
        getChildrenBlock:
            [:node | node getSubdirectories]
        createGraphicNodeBlock:
            [:node | node createGraphicNode].
    
```

如果你在一个类中创建接口适配，这种方法提供了另外一种选择，它相对于子类化方法来说更方便一些。

## 10. 代码示例

对动机一节中例子，从类Shape和TextView开始，我们将给出类适配器和对象适配器实现代码的简要框架。

```

class Shape {
public:
    Shape();
    virtual void BoundingBox(
    
```

```

        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};

class TextView {
public:
    TextView();
    void GetOrigin(Coord& x, Coord& y) const;
    void GetExtent(Coord& width, Coord& height) const;
    virtual bool IsEmpty() const;
};

```

Shape假定有一个边框，这个边框由它相对的两角定义。而 TextView则由原点、宽度和高度定义。Shape同时定义了CreateManipulator操作用于创建一个Manipulator对象。当用户操作一个图形时，Manipulator对象知道如何驱动这个图形<sup>①</sup>。TextView没有等同的操作。TextShape类是这些不同接口间的适配器。

类适配器采用多重继承适配接口。类适配器的关键是用一个分支继承接口，而用另外一个分支继承接口的实现部分。通常 C++中作出这一区分的方法是：用公共方式继承接口；用私有方式继承接口的实现。下面我们按照这种常规方法定义 TextShape适配器。

```

class TextShape : public Shape, private TextView {
public:
    TextShape();

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
};

```

BoundingBox操作对TextView的接口进行转换使之匹配Shape的接口。

```

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    GetOrigin(bottom, left);
    GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

```

IsEmpty操作给出了在适配器实现过程中常用的一种方法：直接转发请求：

```

bool TextShape::IsEmpty () const {
    return TextView::IsEmpty();
}

```

最后，我们定义CreateManipulator (TextView不支持该操作)，假定我们已经实现了支持TextShape操作的类TextManipulator。

```

Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}

```

① CreateManipulator是一个Factory Method的实例。

对象适配器采用对象组合的方法将具有不同接口的类组合在一起。在该方法中，适配器 `TextShape` 维护一个指向 `TextView` 的指针。

```
class TextShape : public Shape {
public:
    TextShape(TextView*);

    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual bool IsEmpty() const;
    virtual Manipulator* CreateManipulator() const;
private:
    TextView* _text;
};
```

`TextShape` 必须在构造器中对指向 `TextView` 实例的指针进行初始化，当它自身的操作被调用时，它还必须对它的 `TextView` 对象调用相应的操作。在本例中，假设客户创建了 `TextView` 对象并且将其传递给 `TextShape` 的构造器：

```
TextShape::TextShape (TextView* t) {
    _text = t;
}

void TextShape::BoundingBox (
    Point& bottomLeft, Point& topRight
) const {
    Coord bottom, left, width, height;

    _text->GetOrigin(bottom, left);
    _text->GetExtent(width, height);

    bottomLeft = Point(bottom, left);
    topRight = Point(bottom + height, left + width);
}

bool TextShape::IsEmpty () const {
    return _text->IsEmpty();
}
```

`CreateManipulator` 的实现代码与类适配器版本的实现代码一样，因为它的实现从零开始，没有复用任何 `TextView` 已有的函数。

```
Manipulator* TextShape::CreateManipulator () const {
    return new TextManipulator(this);
}
```

将这段代码与类适配器的相应代码进行比较，可以看出编写对象适配器代码相对麻烦一些，但是它比较灵活。例如，客户仅需将 `TextView` 子类的一个实例传给 `TextShape` 类的构造函数，对象适配器版本的 `TextShape` 就同样可以与 `TextView` 子类一起很好的工作。

## 11. 已知应用

意图一节例子来自一个基于 `ET++`[WGM88] 的绘图应用程序 `ET++Draw`，`ET++Draw` 通过使用一个 `TextShape` 适配器类的方式复用了 `ET++` 中一些类，并将它们用于正文编辑。

`InterView2.6` 为诸如 `scrollbars`、`buttons` 和 `menus` 的用户界面元素定义了一个抽象类 `Interactor`[VL88]，它同时也为 `line`、`circle`、`polygon` 和 `spline` 这样的结构化图形对象定义了一个抽象类 `Graphics`。`Interactor` 和 `Graphics` 都有图形外观，但它们有着不同的接口和实现（它们没有同一个父类），因此它们并不兼容。也就是说，你不能直接将一个结构化的图形对象嵌入

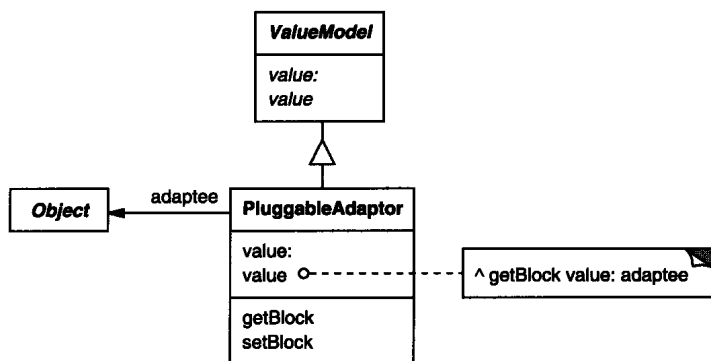


一个对话框中。

而Interview2.6定义了一个称为GraphicBlock的对象适配器，它是Interactor的子类，包含Graphic类的一个实例。GraphicBlock将Graphic类的接口与Interactor类的接口进行匹配。GraphicBlock使得一个Graphic的实例可以在Interactor结构中被显示、滚动和缩放。

可插入的适配器在ObjectWorks/Smalltalk[Par90]中很常见。标准Smalltalk为显示单个值的视图定义了一个ValueModel类。为访问这个值，ValueModel定义了一个“value”和“value:”接口。这些都是抽象方法。应用程序员用与特定领域相关的名字访问这个值，如“width”和“width:”，但为了使特定领域相关的名字与ValueModel的接口相匹配，他们不一定要生成ValueModel的子类。

而ObjectWorks/Smalltalk包含了一个ValueModel类的子类，称为PluggableAdaptor。PluggableAdaptor对象可以将其他对象与ValueModel的接口（“value”和“value:”）相匹配。它可以用模块进行参数化，以便获取和设置所期望的值。PluggableAdaptor在其内部使用这些模块以实现“value”和“value:”接口，如下图所示。为语法上方便起见，PluggableAdaptor也允许你直接传递选择器的名字（例如“width”和“width:”），它自动将这些选择器转换为相应的模块。



另外一个来自ObjectWorks/Smalltalk的例子是TableAdaptor类，它可以将一个对象序列与一个表格表示相匹配。这个表格在每行显示一个对象。客户用表格可以使用的消息集对TableAdaptor进行参数设置，从一个对象得到行属性。

在NeXT的AppKit[Add94]中，一些类使用代理对象进行接口匹配。一个例子是类NXBrowser，它可以显示层次式数据列表。NXBrowser类用一个代理对象存取并适配数据。

Mayer的“Marriage of Convenience”[Mey88]是一种形式的类适配器。Mayer描述了FixedStack类如何匹配一个Array类的实现部分和一个Stack类的接口部分。结果是一个包含一定数目项目的栈。

## 12. 相关模式

模式Bridge(4.2)的结构与对象适配器类似，但是Bridge模式的出发点不同：Bridge目的是将接口部分和实现部分分离，从而对它们可以较为容易也相对独立的加以改变。而Adapter则意味着改变一个已有对象的接口。

Decorator(4.4)模式增强了其他对象的功能而同时又不改变它的接口。因此decorator对应用程序的透明性比适配器要好。结果是decorator支持递归组合，而纯粹使用适配器是不可能实现这一点的。

模式Proxy(4.7)在不改变它的接口的条件下，为另一个对象定义了一个代理。

## 4.2 BRIDGE（桥接）——对象结构型模式

### 1. 意图

将抽象部分与它的实现部分分离，使它们都可以独立地变化。

### 2. 别名

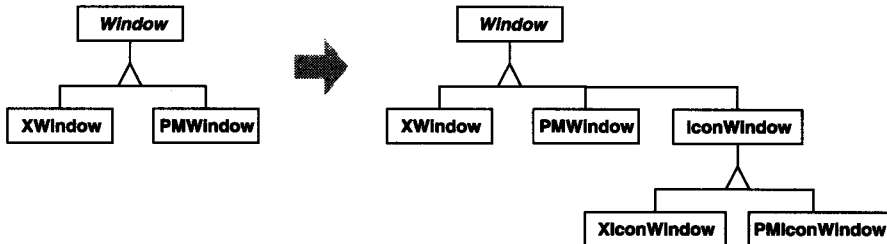
Handle/Body

### 3. 动机

当一个抽象可能有多个实现时，通常用继承来协调它们。抽象类定义对该抽象的接口，而具体的子类则用不同方式加以实现。但是此方法有时不够灵活。继承机制将抽象部分与它的实现部分固定在一起，使得难以对抽象部分和实现部分独立地进行修改、扩充和重用。

让我们考虑在一个用户界面工具箱中，一个可移植的 Window 抽象部分的实现。例如，这一抽象部分应该允许用户开发一些在 X Window System 和 IBM 的 Presentation Manager (PM) 系统中都可以使用的应用程序。运用继承机制，我们可以定义 Window 抽象类和它的两个子类 XWindow 与 PMWindow，由它们分别实现不同系统平台上的 Window 界面。但是继承机制有两个不足之处：

1) 扩展 Window 抽象使之适用于不同种类的窗口或新的系统平台很不方便。假设有 Window 的一个子类 IconWindow，它专门将 Window 抽象用于图标处理。为了使 IconWindow 支持两个系统平台，我们必须实现两个新类 XIconWindow 和 PMIconWindow，更为糟糕的是，我们不得不为每一种类型的窗口都定义两个类。而为了支持第三个系统平台我们还必须为每一种窗口定义一个新的 Window 子类，如下图所示。

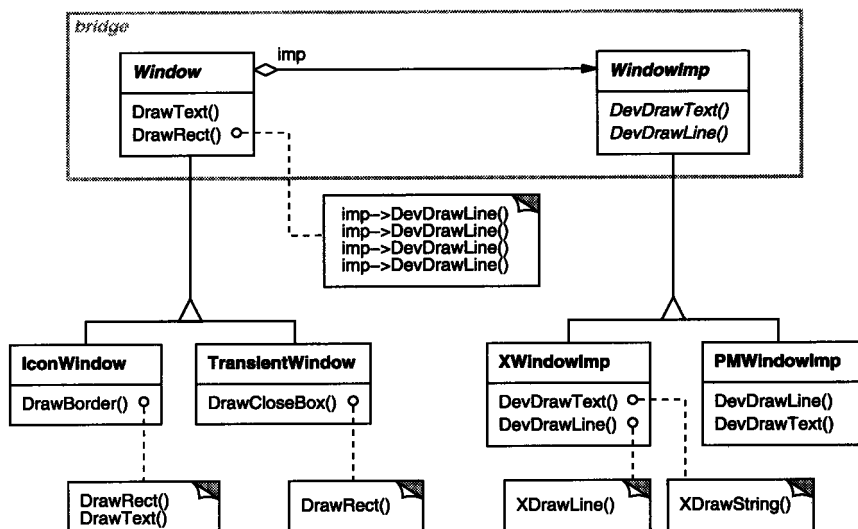


2) 继承机制使得客户代码与平台相关。每当客户创建一个窗口时，必须要实例化一个具体的类，这个类有特定的实现部分。例如，创建 Xwindow 对象会将 Window 抽象与 X Window 的实现部分绑定起来，这使得客户程序依赖于 X Window 的实现部分。这将使得很难将客户代码移植到其他平台上去。

客户在创建窗口时应该不涉及到其具体实现部分。仅仅是窗口的实现部分依赖于应用运行的平台。这样客户代码在创建窗口时就不应涉及到特定的平台。

Bridge 模式解决以上问题的方法是，将 Window 抽象和它的实现部分分别放在独立的类层次结构中。其中一个类层次结构针对窗口接口（Window、IconWindow、TransientWindow），另外一个独立的类层次结构针对平台相关的窗口实现部分，这个类层次结构的根类为 WindowImp。例如 XwindowImp 子类提供了一个基于 X Window 系统的实现，如下页上图所示。

对 Window 子类的所有操作都是用 WindowImp 接口中的抽象操作实现的。这就将窗口的抽象与系统平台相关的实现部分分离开来。因此，我们将 Window 与 WindowImp 之间的关系称之为桥接，因为它在抽象类与它的实现之间起到了桥梁作用，使它们可以独立地变化。

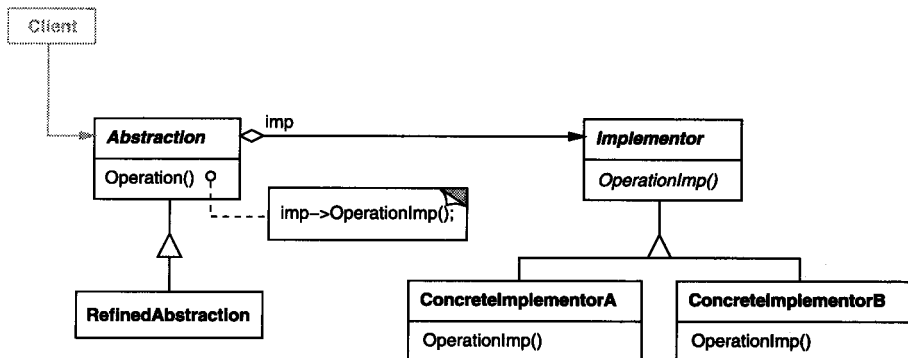


#### 4. 适用性

以下一些情况使用Bridge模式:

- 你不在希望在抽象和它的实现部分之间有一个固定的绑定关系。例如这种情况可能是因为，在程序运行时刻实现部分应可以被选择或者切换。
- 类的抽象以及它的实现都应该可以通过生成子类的方法加以扩充。这时 Bridge模式使你可以对不同的抽象接口和实现部分进行组合，并分别对它们进行扩充。
- 对一个抽象的实现部分的修改应对客户不产生影响，即客户的代码不必重新编译。
- (C++) 你想对客户完全隐藏抽象的实现部分。在 C++ 中，类的表示在类接口中是可见的。
- 正如在意图一节的第一个类图中所示的那样，有许多类要生成。这样一种类层次结构说明你必须将一个对象分解成两个部分。Rumbaugh称这种类层次结构为“嵌套的普化”(nested generalizations)。
- 你想在多个对象间共享实现(可能使用引用计数)，但同时要求客户并不知道这一点。一个简单的例子便是Coplien的String类[Cop92]，在这个类中多个对象可以共享同一个字符串表示(StringRep)。

#### 5. 结构



## 6. 参与者

- Abstraction (Window)
  - 定义抽象类的接口。
  - 维护一个指向 Implementor 类型对象的指针。
- RefinedAbstraction (IconWindow)
  - 扩充由 Abstraction 定义的接口。
- Implementor (WindowImp)
  - 定义实现类的接口，该接口不一定要与 Abstraction 的接口完全一致；事实上这两个接口可以完全不同。一般来讲，Implementor 接口仅提供基本操作，而 Abstraction 则定义了基于这些基本操作的较高层次的操作。
- ConcreteImplementor (XwindowImp, PMWindowImp)
  - 实现 Implementor 接口并定义它的具体实现。

## 7. 协作

- Abstraction 将 client 的请求转发给它的 Implementor 对象。

## 8. 效果

Bridge 模式有以下一些优点：

1) 分离接口及其实现部分 一个实现未必不变地绑定在一个接口上。抽象类的实现可以在运行时刻进行配置，一个对象甚至可以在运行时刻改变它的实现。

将 Abstraction 与 Implementor 分离有助于降低对实现部分编译时刻的依赖性，当改变一个实现类时，并不需要重新编译 Abstraction 类和它的客户程序。为了保证一个类库的不同版本之间的二进制兼容性，一定要有这个性质。

另外，接口与实现分离有助于分层，从而产生更好的结构化系统，系统的高层部分仅需知道 Abstraction 和 Implementor 即可。

2) 提高可扩展性 你可以独立地对 Abstraction 和 Implementor 层次结构进行扩充。

3) 实现细节对客户透明 你可以对客户隐藏实现细节，例如共享 Implementor 对象以及相应的引用计数机制（如果有的话）。

## 9. 实现

使用 Bridge 模式时需要注意以下一些问题：

1) 仅有一个 Implementor 在仅有一个实现的时候，没有必要创建一个抽象的 Implementor 类。这是 Bridge 模式的退化情况；在 Abstraction 与 Implementor 之间有一种一对一的关系。尽管如此，当你希望改变一个类的实现不会影响已有的客户程序时，模式的分离机制还是非常有用的——也就是说，不必重新编译它们，仅需重新连接即可。

Carolan[Car89]用“常露齿嘻笑的猫”(Cheshire Cat)描述这一分离机制。在 C++ 中，Implementor 类的类接口可以在一个私有的头文件中定义，这个文件不提供给客户。这样你就对客户彻底隐藏了一个类的实现部分。

2) 创建正确的 Implementor 对象 当存在多个 Implementor 类的时候，你应该用何种方法，在何时何处确定创建哪一个 Implementor 类呢？

如果 Abstraction 知道所有的 ConcreteImplementor 类，它就可以在它的构造器中对其中的一个类进行实例化，它可以通过传递给构造器的参数确定实例化哪一个类。例如，如果一个

collection类支持多重实现，就可以根据 collection的大小决定实例化哪一个类。链表的实现可以用于较小的collection类，而hash表则可用于较大的collection类。

另外一种方法是首先选择一个缺省的实现，然后根据需要改变这个实现。例如，如果一个collection的大小超出了一定的阈值时，它将会切换它的实现，使之更适用于表目较多的collection。

也可以代理给另一个对象，由它一次决定。在 Window/WindowImp的例子中，我们可以引入一个factory对象（参见Abstract Factory(3.1)），该对象的唯一职责就是封装系统平台的细节。这个对象知道应该为所用的平台创建何种类型的 WindowImp对象；Window仅需向它请求一个WindowImp，而它会返回正确类型的 WindowImp对象。这种方法的优点是 Abstraction 类和任何一个Implementor类直接耦合。

3) 共享Implementor对象 Coplien阐明了如何用C++中常用的Handle/Body方法在多个对象间共享一些实现[Cop92]。其中Body有一个对象引用计数器，Handle对它进行增减操作。将共享程序体赋给句柄的代码一般具有以下形式：

```
Handle& Handle::operator= (const Handle& other) {
    other._body->Ref();
    _body->Unref();

    if (_body->RefCount() == 0) {
        delete _body;
    }
    _body = other._body;

    return *this;
}
```

4) 采用多重继承机制 在C++中可以使用多重继承机制将抽象接口和它的实现部分结合起来[Mar91]。例如，一个类可以用 public方式继承 Abstraction而以 private方式继承 ConcreteImplementor。但是由于这种方法依赖于静态继承，它将实现部分与接口固定不变的绑定在一起。因此不可能使用多重继承的方法实现真正的 Bridge模式——至少用C++不行。

#### 10. 代码示例

下面的C++代码实现了意图一节中 Window/WindowImp的例子，其中 Window类为客户应用程序定义了窗口抽象类：

```
class Window {
public:
    Window(View* contents);

    // requests handled by window
    virtual void DrawContents();

    virtual void Open();
    virtual void Close();
    virtual void Iconify();
    virtual void Deiconify();

    // requests forwarded to implementation
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();
}
```

```
virtual void DrawLine(const Point&, const Point&);
virtual void DrawRect(const Point&, const Point&);
virtual void DrawPolygon(const Point[], int n);
virtual void DrawText(const char*, const Point&);
```

```
protected:
    WindowImp* GetWindowImp();
    View* GetView();
private:
    WindowImp* _imp;
    View* _contents; // the window's contents
};
```

Window维护一个对WindowImp的引用，WindowImp抽象类定义了一个对底层窗口系统的接口。

```
class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;

    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // lots more functions for drawing on windows...
protected:
    WindowImp();
};
```

Window的子类定义了应用程序可能用到的不同类型的窗口，如应用窗口、图标、对话框、临时窗口以及工具箱的移动面板等等。

例如ApplicationWindow类将实现DrawContents操作以绘制它所存储的View实例：

```
class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};

void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
```

IconWindow中存储了它所显示的图标对应的位图名...

```
class IconWindow : public Window {
public:
    // ...
    virtual void DrawContents();
private:
    const char* _bitmapName;
};
```

...并且实现DrawContents操作将这个位图绘制在窗口上：

```
void IconWindow::DrawContents() {
    WindowImp* imp = GetWindowImp();
    if (imp != 0) {
        imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
    }
}
```



我们还可以定义许多其他类型的 Window 类，例如 TransientWindow 在与客户对话时由一个窗口创建，它可能要和这个创建它的窗口进行通信；PaletteWindow 总是在其他窗口之上；IconDockWindow 拥有一些 IconWindow，并且由它负责将它们排列整齐。

Window 的操作由 WindowImp 的接口定义。例如，在调用 WindowImp 操作在窗口中绘制矩形之前，DrawRect 必须从它的两个 Point 参数中提取四个坐标值：

```
void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

具体的 WindowImp 子类可支持不同的窗口系统，XwindowImp 子类支持 X Window 窗口系统：

```
class XWindowImp : public WindowImp {
public:
    XWindowImp();

    virtual void DeviceRect(Coord, Coord, Coord, Coord);
    // remainder of public interface...

private:
    // lots of X window system-specific state, including:
    Display* _dpy;
    Drawable _winid; // window id
    GC _gc;          // window graphic context
};
```

对于 Presentation Manager (PM)，我们定义 PMWindowImp 类：

```
class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // remainder of public interface...

private:
    // lots of PM window system-specific state, including:
    HPS _hps;
};
```

这些子类用窗口系统的基本操作实现 WindowImp 操作，例如，对于 X 窗口系统这样实现 DeviceRect：

```
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);
}
```

PM 的实现部分可能象下面这样：

```
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);
}
}

```

那么一个窗口怎样得到正确的 WindowImp 子类的实例呢？在本例我们假设 Window 类具有这个职责，它的 GetWindowImp 操作负责从一个抽象工厂（参见 Abstract Factory(3.1) 模式）得到正确的实例，这个抽象工厂封装了所有窗口系统的细节。

```

WindowImp* Window::GetWindowImp () {
    if (_imp == 0) {
        _imp = WindowSystemFactory::Instance()->MakeWindowImp();
    }
    return _imp;
}

```

WindowSystemFactory::Instance() 函数返回一个抽象工厂，该工厂负责处理所有与特定窗口系统相关的对象。为简化起见，我们将它创建一个单件（Singleton），允许 Window 类直接访问这个工厂。

### 11. 已知应用

上面的 Window 实例来自于 ET++[WGM88]。在 ET++ 中，WindowImp 称为“WindowPort”，它有 XWindowPort 和 SunWindowPort 这样一些子类。Window 对象请求一个称为“WindowSystem”的抽象工厂创建相应的 Implementor 对象。WindowSystem 提供了一个接口用于创建一些与特定平台相关的对象，例如字体、光标、位图等。

ET++ 的 Window/WindowPort 设计扩展了 Bridge 模式，因为 WindowPort 保留了一个指回 Window 的指针。WindowPort 的 Implementor 类用这个指针通知 Window 对象发生了一些与 WindowPort 相关的事件：例如输入事件的到来，窗口调整大小等。

Coplien[Cop92] 和 Stroustrup[Str91] 都提及 Handle 类并给出了一些例子。这些例子集中处理一些内存管理问题，例如共享字符串表达式以及支持大小可变的对象等。我们主要关心它怎样支持对一个抽象和它的实现进行独立地扩展。

libg++[Lea88] 类库定义了一些类用于实现公共的数据结构，例如 Set、LinkedSet、HashSet、LinkedList 和 HashTable。Set 是一个抽象类，它定义了一组抽象接口，而 LinkedList 和 HashTable 则分别是链表和 hash 表的具体实现。LinkedSet 和 HashSet 是 Set 的实现者，它们桥接了 Set 和它们具体所对应的 LinkedList 和 HashTable。这是一种退化的桥接模式，因为没有抽象 Implementor 类。

NeXT's AppKit[Add94]在图象生成和显示中使用了 Bridge模式。一个图象可以有多种不同的表示方式，一个图象的最佳显示方式取决于显示设备的特性，特别是它的色彩数目和分辨率。如果没有 AppKit的帮助，每一个应用程序中应用开发者都要确定在不同的情况下应该使用哪一种实现方法。

为了减轻开发者的负担，AppKit提供了 NXImage/NXImageRep 桥接。NTImage 定义了图象处理的接口，而图象接口的实现部分则定义在独立的 NXImageRep 类层次中，这个类层次包含了多个子类，如 NXEPSImageRep, NXCachedImageRep 和 NXBitmapImageRep 等。NXImage 维护一个指针，指向一个或多个 NXImageRep 对象。如果有多个图象实现，NXImage 会选择最适合当前显示设备的图象实现。必要时 NXImage 还可以将一个实现转换成另一个实现。这个 Bridge 模式变种很有趣的地方是：NXImage 能同时存储多个 NXImageRep 实现。

## 12. 相关模式

Abstract Factory(3.1) 模式可以用来创建和配置一个特定的 Bridge 模式。

Adapter(4.1) 模式用来帮助无关的类协同工作，它通常在系统设计完成后才会被使用。然而，Bridge 模式则是在系统开始时就使用，它使得抽象接口和实现部分可以独立进行改变。

## 4.3 COMPOSITE（组合）——对象结构型模式

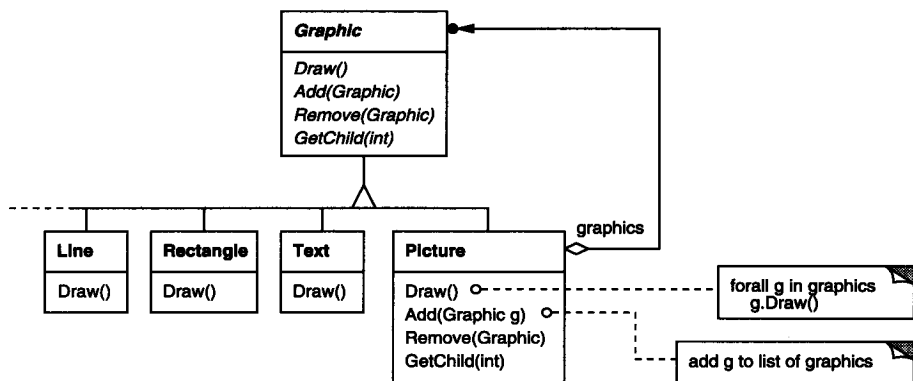
### 1. 意图

将对象组合成树形结构以表示“部分-整体”的层次结构。Composite 使得用户对单个对象和组合对象的使用具有一致性。

### 2. 动机

在绘图编辑器和图形捕捉系统这样的图形应用程序中，用户可以使用简单的组件创建复杂的图表。用户可以组合多个简单组件以形成一些较大的组件，这些组件又可以组合成更大的组件。一个简单的实现方法是为 Text 和 Line 这样的图元定义一些类，另外定义一些类作为这些图元的容器类(Container)。

然而这种方法存在一个问题：使用这些类的代码必须区别对待图元对象与容器对象，而实际上大多数情况下用户认为它们是一样的。对这些类区别使用，使得程序更加复杂。Composite 模式描述了如何使用递归组合，使得用户不必对这些类进行区别，如下图所示。



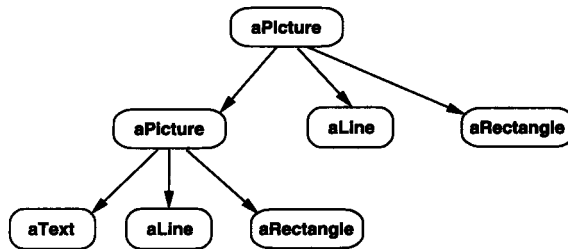
Composite 模式的关键是一个抽象类，它既可以代表图元，又可以代表图元的容器。在图形系统中的这个类就是 Graphic，它声明一些与特定图形对象相关的操作，例如 Draw。同时它

也声明了所有的组合对象共享的一些操作，例如一些操作用于访问和管理它的子部件。

子类Line、Rectangle和Text（参见前面的类图）定义了一些图元对象，这些类实现 Draw，分别用于绘制直线、矩形和正文。由于图元都没有子图形，因此它们都不执行与子类有关的操作。

Picture类定义了一个Graphic 对象的聚合。Picture 的Draw操作是通过对它的子部件调用 Draw实现的，Picture还用这种方法实现了一些与其子部件相关的操作。由于 Picture接口与 Graphic接口是一致的，因此Picture对象可以递归地组合其他 Picture对象。

下图是一个典型的由递归组合的 Graphic对象组成的组合对象结构。

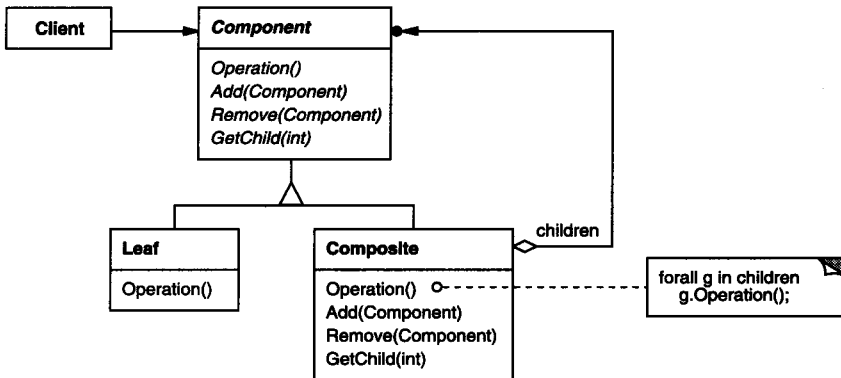


### 3. 适用性

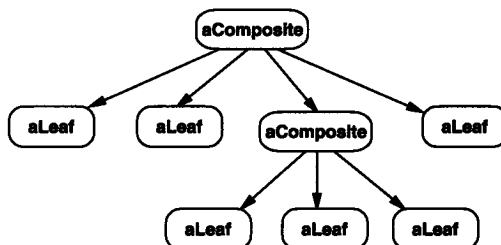
以下情况使用Composite模式：

- 你想表示对象的部分-整体层次结构。
- 你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

### 4. 结构



典型的Composite对象结构如下图所示。



## 5. 参与者

- Component (Graphic)
  - 为组合中的对象声明接口。
  - 在适当的情况下，实现所有类共有接口的缺省行为。
  - 声明一个接口用于访问和管理 Component的子组件。
  - (可选)在递归结构中定义一个接口，用于访问一个父部件，并在合适的情况下实现它。
- Leaf (Rectangle、Line、Text等)
  - 在组合中表示叶节点对象，叶节点没有子节点。
  - 在组合中定义图元对象的行为。
- Composite (Picture)
  - 定义有子部件的那些部件的行为。
  - 存储子部件。
  - 在Component接口中实现与子部件有关的操作。
- Client
  - 通过Component接口操纵组合部件的对象。

## 6. 协作

- 用户使用Component类接口与组合结构中的对象进行交互。如果接收者是一个叶节点，则直接处理请求。如果接收者是Composite，它通常将请求发送给它的子部件，在转发请求之前与/或之后可能执行一些辅助操作。

## 7. 效果

### Composite模式

- 定义了包含基本对象和组合对象的类层次结构 基本对象可以被组合成更复杂的组合对象，而这个组合对象又可以被组合，这样不断的递归下去。客户代码中，任何用到基本对象的地方都可以使用组合对象。
- 简化客户代码 客户可以一致地使用组合结构和单个对象。通常用户不知道（也不关心）处理的是一个叶节点还是一个组合组件。这就简化了客户代码，因为在定义组合的那些类中不需要写一些充斥着选择语句的函数。
- 使得更容易增加新类型的组件 新定义的Composite或Leaf子类自动地与已有的结构和客户代码一起工作，客户程序不需因新的Component类而改变。
- 使你的设计变得更加一般化 容易增加新组件也会产生一些问题，那就是很难限制组合中的组件。有时你希望一个组合只能有某些特定的组件。使用Composite时，你不能依赖类型系统施加这些约束，而必须在运行时刻进行检查。

## 8. 实现

我们在实现Composite模式时需要考虑以下几个问题：

1) 显式的父部件引用 保持从子部件到父部件的引用能简化组合结构的遍历和管理。父部件引用可以简化结构的上移和组件的删除，同时父部件引用也支持Chain of Responsibility(5.2)模式。

通常在Component类中定义父部件引用。Leaf和Composite类可以继承这个引用以及管理这个引用的那些操作。

对于父部件引用，必须维护一个不变式，即一个组合的所有子节点以这个组合为父节点，而反之该组合以这些节点为子节点。保证这一点最容易的办法是，仅当在一个组合中增加或删除一个组件时，才改变这个组件的父部件。如果能在 Composite类的Add 和Remove操作中实现这种方法，那么所有的子类都可以继承这一方法，并且将自动维护这一不变式。

2) 共享组件 共享组件是很有用的，比如它可以减少对存贮的需求。但是当有一个组件只有一个父部件时，很难共享组件。

一个可行的解决办法是为子部件存贮多个父部件，但当一个请求在结构中向上传递时，这种方法会导致多义性。Flyweight(4.6)模式讨论了如何修改设计以避免将父部件存贮在一起的方法。如果子部件可以将一些状态(或是所有的状态)存储在外部，从而不需要向父部件发送请求，那么这种方法是可行的。

3) 最大化Component接口 Composite模式的目的之一是使得用户不知道他们正在使用的具体的Leaf 和Composite类。为了达到这一目的，Composite类应为Leaf 和Composite类尽可能多定义一些公共操作。Composite类通常为这些操作提供缺省的实现，而 Leaf 和Composite子类可以对它们进行重定义。

然而，这个目标有时可能会与类层次结构设计原则相冲突，该原则规定：一个类只能定义那些对它的子类有意义的操作。有许多Component所支持的操作对Leaf类似乎没有什么意义，那么Component怎样为它们提供一个缺省的操作呢？

有时一点创造性可以使得一个看起来仅对 Composite才有意义的操作，将它移入Component类中，就会对所有的Component都适用。例如，访问子节点的接口是 Composite类的一个基本组成部分，但对 Leaf类来说并不必要。但是如果我们把一个 Leaf看成一个没有子节点的Component，就可以为在Component类中定义一个缺省的操作，用于对子节点进行访问，这个缺省的操作不返回任何一个子节点。Leaf 类可以使用缺省的实现，而Composite类则会重新实现这个操作以返回它们的子类。

管理子部件的操作比较复杂，我们将在下一项中予以讨论。

4) 声明管理子部件的操作 虽然Composite类实现了Add 和Remove操作用于管理子部件，但在Composite模式中一个重要的问题是：在 Composite类层次结构中哪一些类声明这些操作。我们是应该在Component中声明这些操作，并使这些操作对 Leaf类有意义呢，还是只应该在Composite和它的子类中声明并定义这些操作呢？

这需要在安全性和透明性之间做出权衡选择。

- 在类层次结构的根部定义子节点管理接口的方法具有良好的透明性，因为你可以一致地使用所有的组件，但是这一方法是以安全性为代价的，因为客户有可能会做一些无意义的事情，例如在Leaf 中增加和删除对象等。
- 在Composite类中定义管理子部件的方法具有良好的安全性，因为在象 C++这样的静态类型语言中，在编译时任何从Leaf 中增加或删除对象的尝试都将被发现。但是这又损失了透明性，因为Leaf 和Composite具有不同的接口。

在这一模式中，相对于安全性，我们比较强调透明性。如果你选择了安全性，有时你可能会丢失类型信息，并且不得不将一个组件转换成一个组合。这样的类型转换必定不是类型安全的。

一种办法是在Component类中声明一个操作Composite\* GetComponent()。Component提供



了一个返回空指针的缺省操作。Composite类重新定义这个操作并通过this指针返回它自身。

```
class Composite;

class Component {
public:
    //...
    virtual Composite* GetComposite() { return 0; }
};

class Composite : public Component {
public:
    void Add(Component*);
    // ...
    virtual Composite* GetComposite() { return this; }
};

class Leaf : public Component {
    // ...
};
```

GetComposite 允许你查询一个组件看它是否是一个组合，你可以对返回的组合安全地执行Add 和Remove操作。

```
Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()) {
    test->Add(new Leaf);
}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {
    test->Add(new Leaf); // will not add leaf
}
```

你可使用C++ 中的dynamic\_cast结构对Composite做相似的试验。

当然，这里的问题是我们对所有的组件的处理并不一致。在进行适当的动作之前，我们必须检测不同的类型。

提供透明性的唯一方法是在Component中定义缺省Add 和Remove操作。这又带来了新的问题：Component::Add的实现不可避免地会有失败的可能性。你可以不让Component::Add做任何事情，但这就忽略了一个很重要的问题：企图向叶节点中增加一些东西时可能会引入错误。这时Add操作会产生垃圾。你可以让Add操作删除它的参数，但可能客户并不希望这样。

如果该组件不允许有子部件，或者Remove的参数不是该组件的子节点时，通常最好使用缺省方式(可能是产生一个异常)处理Add和Remove的失败。

另一个办法是对“删除”的含义作一些改变。如果该组件有一个父部件引用，我们可重新定义Component::Remove,在它的父组件中删除掉这个组件。然而，对应的Add操作仍然没有合理的解释。

5) Component是否应该实现一个Component列表 你可能希望在Component类中将子节点集合定义为一个实例变量，而这个Component类中也声明了一些操作对子节点进行访问和管

理。但是在基类中存放子类指针，对叶节点来说会导致空间浪费，因为叶节点根本没有子节点。只有当该结构中子类数目相对较少时，才值得使用这种方法。

6) 子部件排序 许多设计指定了Composite的子部件顺序。在前面的Graphics例子中，排序可能表示了从前至后的顺序。如果Composite表示语法分析树，Composite子部件的顺序必须反映程序结构，而组合语句就是这样一些Composite的实例。

如果需要考虑子节点的顺序时，必须仔细地设计对子节点的访问和管理接口，以便管理子节点序列。Iterator模式(5.4)可以在这方面给予一些定的指导。

7) 使用高速缓冲存贮改善性能 如果你需要对组合进行频繁的遍历或查找，Composite类可以缓冲存储对它的子节点进行遍历或查找的相关信息。Composite可以缓冲存储实际结果或者仅仅是一些用于缩短遍历或查询长度的信息。例如，动机一节的例子中Picture类能高速缓冲存贮其子部件的边界框，在绘图或选择期间，当子部件在当前窗口中不可见时，这个边界框使得Picture不需要再进行绘图或选择。

一个组件发生变化时，它的父部件原先缓冲存贮的信息也变得无效。在组件知道其父部件时，这种方法最为有效。因此，如果你使用高速缓冲存贮，你需要定义一个接口来通知组合组件它们所缓冲存贮的信息无效。

8) 应该由谁删除Component 在没有垃圾回收机制的语言中，当一个Composite被销毁时，通常最好由Composite负责删除其子节点。但有一种情况除外，即Leaf对象不会改变，因此可以被共享。

9) 存贮组件最好用哪一种数据结构 Composite可使用多种数据结构存贮它们的子节点，包括连接列表、树、数组和hash表。数据结构的选择取决于效率。事实上，使用通用数据结构根本没有必要。有时对每个子节点，Composite都有一个变量与之对应，这就要求Composite的每个子类都要实现自己的管理接口。参见Interpreter(5.3)模式中的例子。

#### 9. 代码示例

计算机和立体声组合音响这样的设备经常被组装成部分 - 整体层次结构或者是容器层次结构。例如，底盘可包含驱动装置和平面板，总线含有多个插件，机柜包括底盘、总线等。这种结构可以很自然地用Composite模式进行模拟。

Equipment类为在部分 - 整体层次结构中的所有设备定义了一个接口。

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

Equipment 声明一些操作返回一个设备的属性，例如它的能量消耗和价格。子类为指定的设备实现这些操作，Equipment 还声明了一个 CreateIterator 操作，该操作为访问它的零件返回一个 Iterator（参见附录 C）。这个操作的缺省实现返回一个 NullIterator，它在空集上叠代。

Equipment 的子类包括表示磁盘驱动器、集成电路和开关的 Leaf 类：

```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

CompositeEquipment 是包含其他设备的基类，它也是 Equipment 的子类。

```
class CompositeEquipment : public Equipment {
public:
    virtual ~CompositeEquipment();

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List<Equipment*> _equipment;
};
```

CompositeEquipment 为访问和管理子设备定义了一些操作。操作 Add 和 Remove 从存储在 \_equipment 成员变量中的设备列表中插入并删除设备。操作 CreateIterator 返回一个迭代器（ListIterator 的一个实例）遍历这个列表。

NetPrice 的缺省实现使用 CreateIterator 来累加子设备的实际价格<sup>①</sup>。

```
Currency CompositeEquipment::NetPrice () {
    Iterator<Equipment*>* i = CreateIterator();
    Currency total = 0;

    for (i->First(); !i->IsDone(); i->Next()) {
        total += i->CurrentItem()->NetPrice();
    }
    delete i;
    return total;
}
```

现在我们将计算机的底盘表示为 CompositeEquipment 的子类 Chassis。Chassis 从 CompositeEquipment 继承了与子类有关的那些操作。

```
class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();
```

① 用完 Iterator 时，很容易忘记删除它。Iterator 模式描述了如何处理这类问题。

```
virtual Watt Power();
virtual Currency NetPrice();
virtual Currency DiscountPrice();
};
```

我们可用相似的方式定义其他设备容器，如 Cabinet和Bus。这样我们就得到了组装一台(非常简单)个人计算机所需的所有设备。

```
Cabinet* cabinet = new Cabinet("PC Cabinet");
Chassis* chassis = new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus* bus = new Bus("MCA Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

cout << "The net price is " << chassis->NetPrice() << endl;
```

## 10. 已知应用

几乎在所有面向对象的系统中都有 Composite 模式的应用实例。在 Smalltalk中的 Model/View/Controller[KP88]结构中，原始 View类就是一个Composite，几乎每个用户界面工具箱或框架都遵循这些步骤，其中包括 ET++ (用VObjects[WGM88])和InterViews(Style [LCI+92],Graphics[VL88]和Glyphs[CL90])。很有趣的是Model /View/Controller中的原始 View有一组子视图；换句话说，View 既是Component 类，又是Composite类。4.0版的Smalltalk-80用VisualComponent类修改了Model/View/Controller，VisualComponent类含有子类 View和CompositeView。

RTL Smalltalk 编译器框架[JML92]大量地使用了Composite模式。RTLExpression 是一个对应于语法分析树的 Component类。它有一些子类，例如 BinaryExpression，而BinaryExpression包含子RTLExpression对象。这些类为语法分析树定义了一个组合结构。RegisterTransfer是一个用于程序的中间 Single Static Assignment(SSA)形式的Component 类。RegisterTransfer的Leaf子类定义了一些不同的静态赋值形式，例如：

- 基本赋值，在两个寄存器上执行操作并且将结果放入第三个寄存器中。
- 具有源寄存器但无目标寄存器的赋值，这说明是在例程返回后使用该寄存器。
- 具有目标寄存器但无源寄存器的赋值，这说明是在例程开始之前分配目标寄存器。

另一个子类RegisterTransferSet，是一个Composite类，表示一次改变几个寄存器的赋值。

这种模式的另一个例子出现在财经应用领域，在这一领域中，一个资产组合聚合多个单个资产。为了支持复杂的资产聚合，资产组合可以用一个 Composite类实现，这个Composite类与单个资产的接口一致[BE93]。

Command ( 5.2 ) 模式描述了如何用 一个 MacroCommand Composite类组成一些Command对象，并对它们进行排序。

## 11. 相关模式

通常部件-父部件连接用于Responsibility of Chain(5.1)模式。

Decorator ( 4.4 ) 模式经常与 Composite模式一起使用。当装饰和组合一起使用时，它们通常有一个公共的父类。因此装饰必须支持具有 Add、Remove和GetChild 操作的Component

接口。

Flyweight(4.6)让你共享组件，但不再能引用他们的父部件。

Iterator(5.4)可用来遍历 Composite。

Visitor(5.11)将本来应该分布在 Composite 和 Leaf 类中的操作和行为局部化。

## 4.4 DECORATOR (装饰) ——对象结构型模式

### 1. 意图

动态地给一个对象添加一些额外的职责。就增加功能来说，Decorator 模式相比生成子类更为灵活。

### 2. 别名

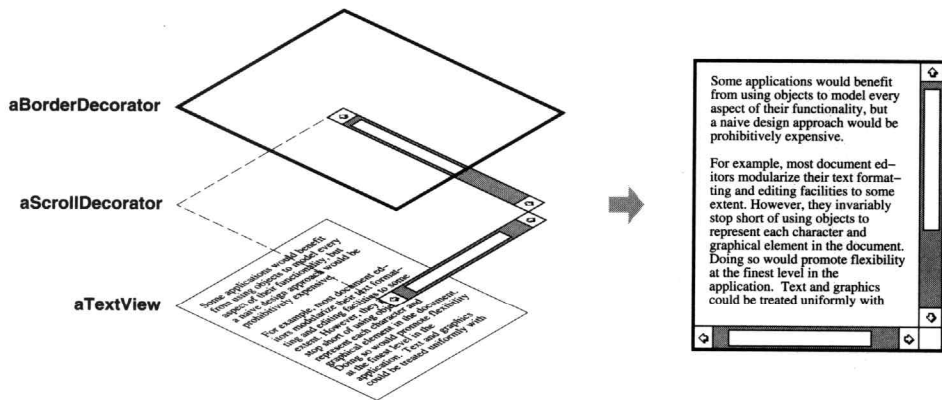
包装器 Wrapper

### 3. 动机

有时我们希望给某个对象而不是整个类添加一些功能。例如，一个图形用户界面工具箱允许你对任意一个用户界面组件添加一些特性，例如边框，或是一些行为，例如窗口滚动。

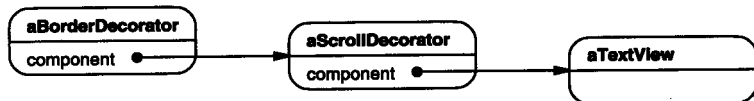
使用继承机制是添加功能的一种有效途径，从其他类继承过来的边框特性可以被多个子类的实例所使用。但这种方法不够灵活，因为边框的选择是静态的，用户不能控制对组件加边框的方式和时机。

一种较为灵活的方式是将组件嵌入另一个对象中，由这个对象添加边框。我们称这个嵌入的对象为装饰。这个装饰与它所装饰的组件接口一致，因此它对使用该组件的客户透明。它将客户请求转发给该组件，并且可能在转发前后执行一些额外的动作（例如画一个边框）。透明性使得你可以递归的嵌套多个装饰，从而可以添加任意多的功能，如下图所示。

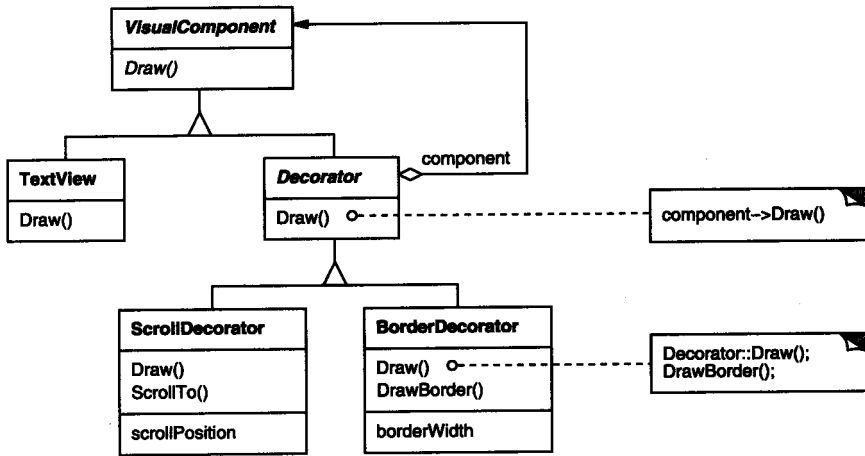


例如，假定有一个对象 TextView，它可以在窗口中显示正文。缺省的 TextView 没有滚动条，因为我们可能有时并不需要滚动条。当需要滚动条时，我们可以用 ScrollDecorator 添加滚动条。如果我们还想在 TextView 周围添加一个粗黑边框，可以使用 BorderDecorator 添加。因此只要简单地将这些装饰和 TextView 进行组合，就可以达到预期的效果。

下面的对象图展示了如何将一个 TextView 对象与 BorderDecorator 以及 ScrollDecorator 对象组装起来产生一个具有边框和滚动条的文本显示窗口。



ScrollDecorator和BorderDecorator 类是Decorator类的子类。Decorator类是一个可视组件的抽象类，用于装饰其他可视组件，如下图所示。



VisualComponent是一个描述可视对象的抽象类，它定义了绘制和事件处理的接口。注意Decorator类怎样将绘制请求简单地发送给它的组件，以及Decorator的子类如何扩展这个操作。

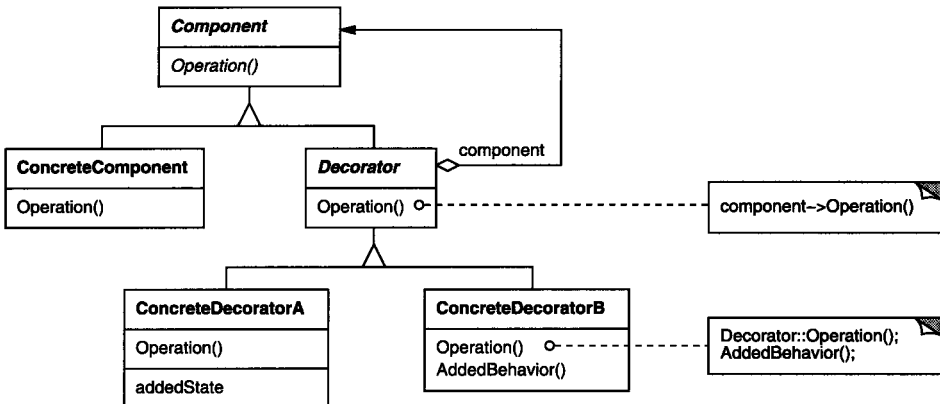
Decorator的子类为特定功能可以自由地添加一些操作。例如，如果其他对象知道界面中恰好有一个ScrollDecorator对象，这些对象就可以用ScrollDecorator对象的ScrollTo操作滚动这个界面。这个模式中有一点很重要，它使得在VisualComponent可以出现的任何地方都可以有装饰。因此，客户通常不会感觉到装饰过的组件与未装饰组件之间的差异，也不会与装饰产生任何依赖关系。

#### 4. 适用性

以下情况使用Decorator模式

- 在不影响其他对象的情况下，以动态、透明的方式给单个对象添加职责。
- 处理那些可以撤消的职责。
- 当不能采用生成子类的方法进行扩充时。一种情况是，可能有大量独立的扩展，为支持每一种组合将产生大量的子类，使得子类数目呈爆炸性增长。另一种情况可能是因为类定义被隐藏，或类定义不能用于生成子类。

#### 5. 结构





## 6. 参与者

- Component (VisualComponent)

- 定义一个对象接口，可以给这些对象动态地添加职责。

- ConcreteComponent (TextView)

- 定义一个对象，可以给这个对象添加一些职责。

- Decorator

- 维持一个指向 Component 对象的指针，并定义一个与 Component 接口一致的接口。

- ConcreteDecorator (BorderDecorator, ScrollDecorator)

- 向组件添加职责。

## 7. 协作

- Decorator 将请求转发给它的 Component 对象，并有可能在转发请求前后执行一些附加的动作。

## 8. 效果

Decorator 模式至少有两个主要优点和两个缺点：

1) 比静态继承更灵活 与对象的静态继承（多重继承）相比，Decorator 模式提供了更加灵活的向对象添加职责的方式。可以用添加和分离的方法，用装饰在运行时刻增加和删除职责。相比之下，继承机制要求为每个添加的职责创建一个新的子类（例如，BorderScrollable TextView, BorderedTextView）。这会产生许多新的类，并且会增加系统的复杂度。此外，为一个特定的 Component 类提供多个不同的 Decorator 类，这就使得你可以对一些职责进行混合和匹配。

使用 Decorator 模式可以很容易地重复添加一个特性，例如在 TextView 上添加双边框时，仅需将添加两个 BorderDecorator 即可。而两次继承 Border 类则极容易出错的。

2) 避免在层次结构高层的类有太多的特征 Decorator 模式提供了一种“即用即付”的方法来添加职责。它并不试图在一个复杂的可定制的类中支持所有可预见的特征，相反，你可以定义一个简单的类，并且用 Decorator 类给它逐渐地添加功能。可以从简单的部件组合出复杂的功能。这样，应用程序不必为不需要的特征付出代价。同时也更易于不依赖于 Decorator 所扩展（甚至是不可预知的扩展）的类而独立地定义新类型的 Decorator。扩展一个复杂类的时候，很可能会暴露与添加的职责无关的细节。

3) Decorator 与它的 Component 不一样 Decorator 是一个透明的包装。如果我们从对象标识的观点出发，一个被装饰了的组件与这个组件是有差别的，因此，使用装饰时不应该依赖对象标识。

4) 有许多小对象 采用 Decorator 模式进行系统设计往往会产生许多看上去类似的小对象，这些对象仅仅在他们相互连接的方式上有所不同，而不是它们的类或是它们的属性值有所不同。尽管对于那些了解这些系统的人来说，很容易对它们进行定制，但是很难学习这些系统，排错也很困难。

## 9. 实现

使用 Decorator 模式时应注意以下几点：

1) 接口的一致性 装饰对象的接口必须与它所装饰的 Component 的接口是一致的，因此，所有的 ConcreteDecorator 类必须有一个公共的父类（至少在 C++ 中如此）。

2) 省略抽象的Decorator类 当你仅需要添加一个职责时，没有必要定义抽象Decorator类。你常常需要处理现存的类层次结构而不是设计一个新系统，这时你可以把Decorator向Component转发请求的职责合并到ConcreteDecorator中。

3) 保持Component类的简单性 为了保证接口的一致性，组件和装饰必须有一个公共的Component父类。因此保持这个类的简单性是很重要的；即，它应集中于定义接口而不是存储数据。对数据表示的定义应延迟到子类中，否则Component类会变得过于复杂和庞大，因而难以大量使用。赋予Component太多的功能也使得，具体的子类有一些它们并不需要的功能的可能性大大增加。

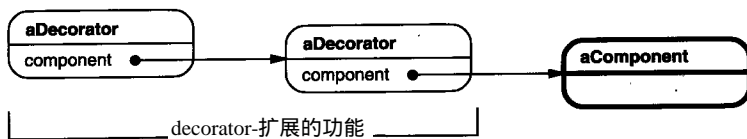
4) 改变对象外壳与改变对象内核 我们可以将Decorator看作一个对象的外壳，它可以改变这个对象的行为。另外一种方法是改变对象的内核。例如，Strategy(5.9)模式就是一个用于改变内核的很好的模式。

当Component类原本就很庞大时，使用Decorator模式代价太高，Strategy模式相对更好一些。在Strategy模式中，组件将它的一些行为转发给一个独立的策略对象，我们可以替换strategy对象，从而改变或扩充组件的功能。

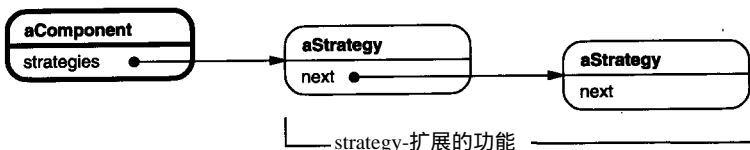
例如我们可以将组件绘制边界的功能延迟到一个独立的Border对象中，这样就可以支持不同的边界风格。这个Border对象是一个Strategy对象，它封装了边界绘制策略。我们可以将策略的数目从一个扩充为任意多个，这样产生的效果与对装饰进行递归嵌套是一样的。

在MacApp3.0[App89]和Bedrock[Sym93a]中，绘图组件（称之为“视图”）有一个“装饰”（adornner）对象列表，这些对象可用来给一个视图组件添加一些装饰，例如边框。如果给一个视图添加了一些装饰，就可以用这些装饰对这个视图进行一些额外的修饰。由于View类过于庞大，MacApp和Bedrock必须使用这种方法。仅为添加一个边框就使用一个完整的View，代价太高。

由于Decorator模式仅从外部改变组件，因此组件无需对它的装饰有任何了解；也就是说，这些装饰对该组件是透明的，如下图所示。



在Strategy模式中，component组件本身知道可能进行哪些扩充，因此它必须引用并维护相应的策略，如下图所示。



基于Strategy的方法可能需要修改component组件以适应新的扩充。另一方面，一个策略可以有自己特定的接口，而装饰的接口则必须与组件的接口一致。例如，一个绘制边框的策略仅需要定义生成边框的接口（DrawBorder, GetWidth等），这意味着即使Component类很庞大时，策略也可以很小。

MacApp和Bedrock中，这种方法不仅仅用于装饰视图，还用于增强对象的事件处理能力。在这两个系统中，每个视图维护一个“行为”对象列表，这些对象可以修改和截获事件。在已注册的行为对象被没有注册的行为有效的重定义之前，这个视图给每个已注册的对象一个处理事件的机会。可以用特殊的键盘处理支持装饰一个视图，例如，可以注册一个行为对象截获并处理键盘事件。

#### 10. 代码示例

以下 C++ 代码说明了如何实现用户接口装饰。我们假定已经存在一个 `Component` 类 `VisualComponent`。

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

    virtual void Draw();
    virtual void Resize();
    // ...
};
```

我们定义 `VisualComponent` 的一个子类 `Decorator`，我们将生成 `Decorator` 的子类以获取不同的装饰。

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
```

`Decorator` 装饰由 `_component` 实例变量引用的 `VisualComponent`，这个实例变量在构造器中被初始化。对于 `VisualComponent` 接口中定义的每一个操作，`Decorator` 类都定义了一个缺省的实现，这一实现将请求转发给 `_component`：

```
void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}
```

`Decorator` 的子类定义了特殊的装饰功能，例如，`BorderDecorator` 类为它所包含的组件添加了一个边框。`BorderDecorator` 是 `Decorator` 的子类，它重定义 `Draw` 操作用于绘制边框。同时 `BorderDecorator` 还定义了一个私有的辅助操作 `DrawBorder`，由它绘制边框。这些子类继承了 `Decorator` 类所有其他的操作。

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
```

```
};
```

```
void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

类似的可以实现 ScrollDecorator 和 DropShadowDecorator，它们给可视组件添加滚动和阴影功能。

现在我们组合这些类的实例以提供不同的装饰效果，以下代码展示了如何使用 Decorator 创建一个具有边界的可滚动 TextView。

首先我们要将一个可视组件放入窗口对象中。我们假设 Window 类为此已经提供了一个 SetContents 操作：

```
void Window::SetContents (VisualComponent* contents) {
    // ...
}
```

现在我们可以创建一个正文视图以及放入这个正文视图的窗口：

```
Window* window = new Window;
TextView* textView = new TextView;
```

TextView 是一个 VisualComponent，它可以放入窗口中：

```
window->SetContents(textView);
```

但我们想要一个有边界的和可以滚动的 TextView，因此我们在将它放入窗口之前对其进行装饰：

```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

由于 Window 通过 VisualComponent 接口访问它的内容，因此它并不知道存在该装饰。如果你需要直接与正文视图交互，例如，你想调用一些操作，而这些操作不是 VisualComponent 接口的一部分，此时你可以跟踪正文视图。依赖于组件标识的客户也应该直接引用它。

## 11. 已知应用

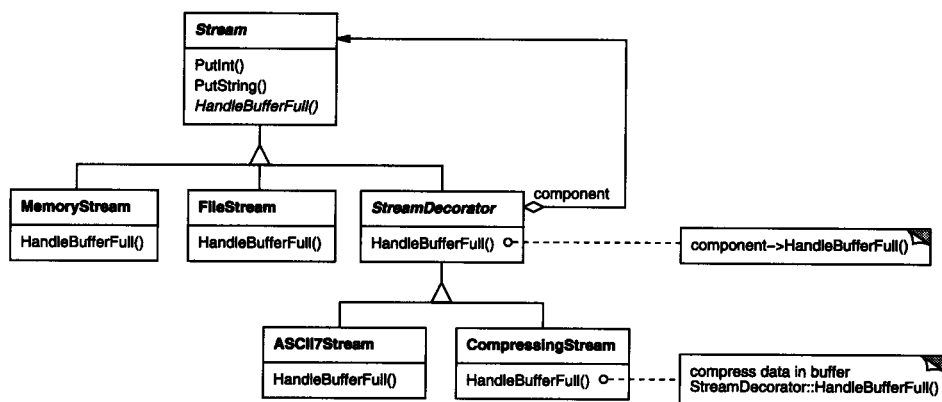
许多面向对象的用户界面工具箱使用装饰为窗口组件添加图形装饰，例如 InterViews [LVC89, LCI+92]，ET++ [WGM88] 和 ObjectWorks\Smalltalk 类库 [Par90]。一些 Decorator 模式的比较特殊的应用有 InterViews 的 DebuggingGlyph 和 ParcPlace Smalltalk 的 PassivityWrapper。DebuggingGlyph 在向它的组件转发布局请求前后，打印出调试信息。这些跟踪信息可用于分析和调试一个复杂组合中对象的布局行为。PassivityWrapper 可以允许和禁止用户与组件的交互。

但是 Decorator 模式不仅仅局限于图形用户界面，下面的例子（基于 ET++ 的 streaming 类 [WGM88]）说明了这一点。

Streams 是大多数 I/O 设备的基础抽象结构，它提供了将对象转换为字节或字符流的操作接口，使我们可以将一个对象转变为一个文件或内存中的字符串，可以在以后恢复使用。一个简单直接的方法是定义一个抽象的 Stream 类，它有两个子类 MemoryStream 与 FileStream。但假定我们还希望能够做下面一些事情：

- 用不同的压缩算法（行程编码，Lempel-Ziv等）对数据流进行压缩。
- 将流数据简化为7位ASCII码字符，这样它就可以在ASCII信道上传输。

Decorator模式提供的将这些功能添加到Stream中方法很巧妙。下面的类图给出了一个解决问题的方法。



Stream抽象类维持了一个内部缓冲区并提供一些操作（PutInt, PutString）用于将数据存入流中。一旦这个缓冲区满了，Stream就会调用抽象操作HandleBufferFull进行实际数据传输。在FileStream中重定义了这个操作，将缓冲区中的数据传送到文件中。

这里的关键类是StreamDecorator，它维持了一个指向组件流的指针并将请求转发给它，StreamDecorator子类重定义HandleBufferFull操作并且在调用StreamDecorator的HandleBufferFull操作之前执行一些额外的动作。

例如，CompressingStream子类用于压缩数据，而ASCII7Stream将数据转换成7位ASCII码。现在我们创建FileStream类，它首先将数据压缩，然后将压缩了的二进制数据转换成为7位ASCII码，我们用CompressingStream和ASCII7Stream装饰FileStream：

```
Stream* aStream = new CompressingStream(
    new ASCII7Stream(
        new FileStream("aFileName")
    )
);
aStream->PutInt(12);
aStream->PutString("aString");
```

## 12. 相关模式

Adapter(4.1)模式：Decorator模式不同于Adapter模式，因为装饰仅改变对象的职责而不改变它的接口；而适配器将给对象一个全新的接口。

Composite(4.3)模式：可以将装饰视为一个退化的、仅有一个组件的组合。然而，装饰仅给对象添加一些额外的职责——它的目的不在于对象聚集。

Strategy(5.9)模式：用一个装饰你可以改变对象的外表；而Strategy模式使得你可以改变对象的内核。这是改变对象的两种途径。

## 4.5 FACADE（外观）——对象结构型模式

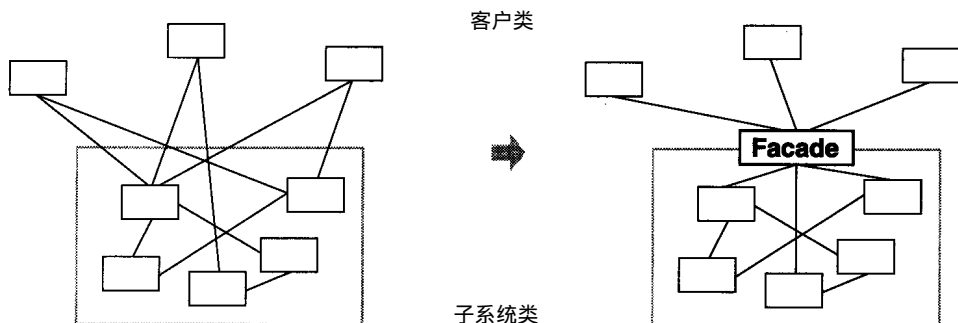
### 1. 意图

为子系统中的一组接口提供一个一致的界面，Facade模式定义了一个高层接口，这个接

口使得这一子系统更加容易使用。

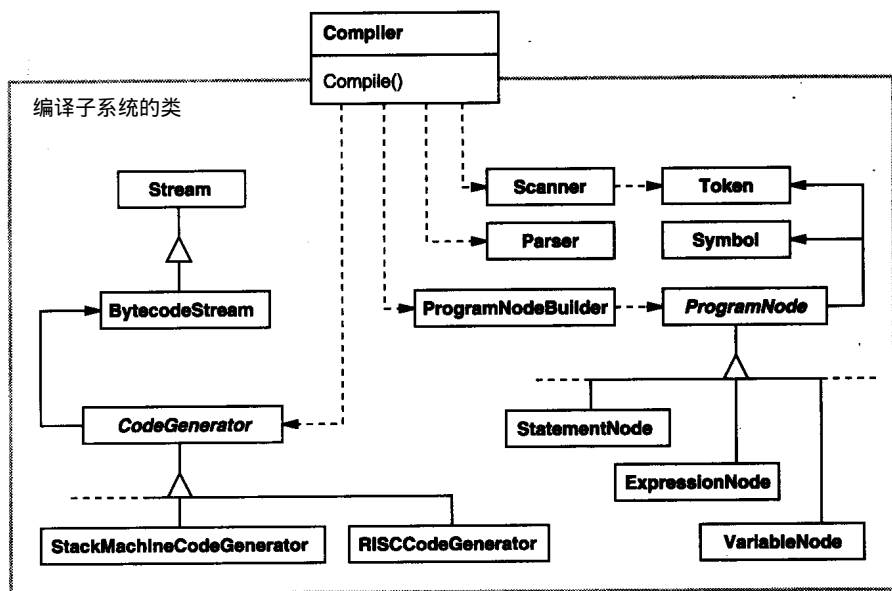
## 2. 动机

将一个系统划分成为若干个子系统有利于降低系统的复杂性。一个常见的设计目标是使子系统间的通信和相互依赖关系达到最小。达到该目标的途径之一就是引入一个外观 ( facade ) 对象，它为子系统中较一般的设施提供了一个单一而简单的界面。



例如有一个编程环境，它允许应用程序访问它的编译子系统。这个编译子系统包含了若干个类，如Scanner、Parser、ProgramNode、BytecodeStream和ProgramNodeBuilder，用于实现这一编译器。有些特殊应用程序需要直接访问这些类，但是大多数编译器的用户并不关心语法分析和代码生成这样的细节；他们只是希望编译一些代码。对这些用户，编译子系统中那些功能强大但层次较低的接口只会使他们的任务复杂化。

为了提供一个高层的接口并且对客户屏蔽这些类，编译子系统还包括一个 Compiler类。这个类定义了一个编译器功能的统一接口。Compiler类是一个外观，它给用户提供了一个单一而简单的编译子系统接口。它无需完全隐藏实现编译功能的那些类，即可将它们结合在一起。编译器的外观可方便大多数程序员使用，同时对少数懂得如何使用底层功能的人，它并不隐藏这些功能，如下图所示。



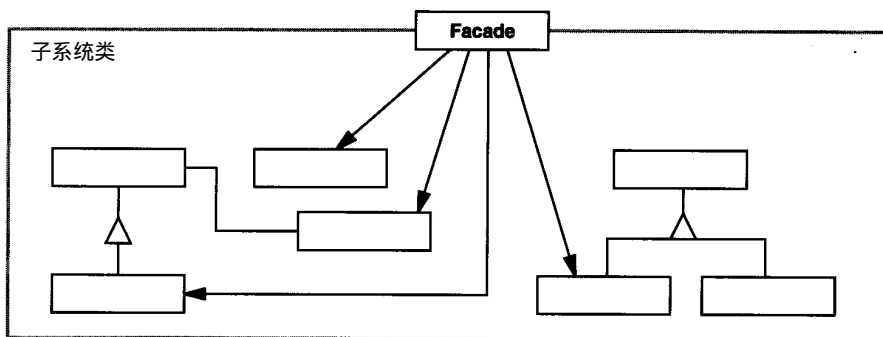


### 3. 适用性

在遇到以下情况使用 Facade 模式

- 当你要为一个复杂子系统提供一个简单接口时。子系统往往因为不断演化而变得越来越复杂。大多数模式使用时都会产生更多更小的类。这使得子系统更具可重用性，也更容易对子系统定制，但这也给那些不需要定制子系统的用户带来一些使用上的困难。Facade 可以提供一个简单的缺省视图，这一视图对大多数用户来说已经足够，而那些需要更多的可定制性的用户可以越过 facade 层。
- 客户程序与抽象类的实现部分之间存在着很大的依赖性。引入 facade 将这个子系统与客户以及其他的子系统分离，可以提高子系统的独立性和可移植性。
- 当你需要构建一个层次结构的子系统时，使用 facade 模式定义子系统中每层的入口点。如果子系统之间是相互依赖的，你可以让它们仅通过 facade 进行通讯，从而简化了它们之间的依赖关系。

### 4. 结构



### 5. 参与者

- Facade (Compiler)
  - 知道哪些子系统类负责处理请求。
  - 将客户的请求代理给适当的子系统对象。
- Subsystem classes (Scanner、Parser、ProgramNode等)
  - 实现子系统的功能。
  - 处理由 Facade 对象指派的任务。
  - 没有 facade 的任何相关信息；即没有指向 facade 的指针。

### 6. 协作

- 客户程序通过发送请求给 Facade 的方式与子系统通讯，Facade 将这些消息转发给适当的子系统对象。尽管是子系统内的有关对象在做实际工作，但 Facade 模式本身也必须将它的接口转换成子系统的接口。
- 使用 Facade 的客户程序不需要直接访问子系统对象。

### 7. 效果

Facade 模式有下面一些优点：

- 1) 它对客户屏蔽子系统组件，因而减少了客户处理的对象的数目并使得子系统使用起来更加方便。

2) 它实现了子系统与客户之间的松耦合关系，而子系统内部的功能组件往往是紧耦合的。松耦合关系使得子系统的组件变化不会影响到它的客户。 Facade模式有助于建立层次结构系统，也有助于对对象之间的依赖关系分层。 Facade模式可以消除复杂的循环依赖关系。这一点在客户程序与子系统是分别实现的时候尤为重要。

在大型软件系统中降低编译依赖性至关重要。在子系统类改变时，希望尽量减少重编译工作以节省时间。用 Facade可以降低编译依赖性，限制重要系统中较小的变化所需的重编译工作。 Facade模式同样也有利于简化系统在不同平台之间的移植过程，因为编译一个子系统一般不需要编译所有其他的子系统。

3) 如果应用需要，它并不限制它们使用子系统类。因此你可以在系统易用性和通用性之间加以选择。

## 8. 实现

使用Facade模式时需要注意以下几点：

1) 降低客户-子系统之间的耦合度 用抽象类实现Facade而它的具体子类对应于不同的子系统实现，这可以进一步降低客户与子系统的耦合度。这样，客户就可以通过抽象的 Facade类接口与子系统通讯。这种抽象耦合关系使得客户不知道它使用的是子系统的哪一个实现。

除生成子类的方法以外，另一种方法是用不同的子系统对象配置 Facade对象。为定制 facade，仅需对它的子系统对象（一个或多个）进行替换即可。

2) 公共子系统类与私有子系统类 一个子系统与一个类的相似之处是，它们都有接口并且它们都封装了一些东西——类封装了状态和操作，而子系统封装了一些类。考虑一个类的公共和私有接口是有益的，我们也可以考虑子系统的公共和私有接口。

子系统的公共接口包含所有的客户程序可以访问的类；私有接口仅用于对子系统进行扩充。当然，Facade类是公共接口的一部分，但它不是唯一的部分，子系统的其他部分通常也是公共的。例如，编译子系统中的 Parser类和Scanner类就是公共接口的一部分。

私有化子系统类确实有用，但是很少有面向对象的编程语言支持这一点。 C++和Smalltalk语言仅在传统意义下为类提供了一个全局名空间。然而，最近 C++标准化委员会在C++语言中增加了一些名字空间[Str94]，这些名字空间使得你可以仅暴露公共子系统类。

## 9. 代码示例

让我们仔细观察一下如何在一个编译子系统中使用 Facade。

编译子系统定义了一个 BytecodeStream类，它实现了一个 Bytecode对象流（stream）。Bytecode对象封装一个字节码，这个字节码可用于指定机器指令。该子系统中还定义了一个 Token类，它封装了编程语言中的标识符。

Scanner类接收字符流并产生一个标识符流，一次产生一个标识符（token）。

```
class Scanner {
public:
    Scanner(istream&);
    virtual ~Scanner();

    virtual Token& Scan();
private:
    istream& _inputStream;
};
```

用ProgramNodeBuilder，Parser类由Scanner生成的标识符构建一棵语法分析树。

```
class Parser {
public:
    Parser();
    virtual ~Parser();

    virtual void Parse(Scanner&, ProgramNodeBuilder&);
};
```

Parser回调ProgramNodeBuilder逐步建立语法分析树，这些类遵循 Builder(3.2)模式进行交互操作。

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

    virtual ProgramNode* NewVariable(
        const char* variableName
    ) const;

    virtual ProgramNode* NewAssignment(
        ProgramNode* variable, ProgramNode* expression
    ) const;

    virtual ProgramNode* NewReturnStatement(
        ProgramNode* value
    ) const;

    virtual ProgramNode* NewCondition(
        ProgramNode* condition,
        ProgramNode* truePart, ProgramNode* falsePart
    ) const;
    // ...

    ProgramNode* GetRootNode();
private:
    ProgramNode* _node;
};
```

语法分析树由 ProgramNode 子类(例如 StatementNode 和 ExpressionNode 等)的实例构成。ProgramNode 层次结构是 Composite 模式的一个应用实例。ProgramNode 定义了一个接口用于操作程序节点和它的子节点(如果有的话)。

```
class ProgramNode {
public:
    // program node manipulation
    virtual void GetSourcePosition(int& line, int& index);
    // ...

    // child manipulation
    virtual void Add(ProgramNode*);
    virtual void Remove(ProgramNode*);
    // ...

    virtual void Traverse(CodeGenerator&);
protected:
    ProgramNode();
};
```

Traverse 操作以一个 CodeGenerator 对象为参数，ProgramNode 子类使用这个对象产生机器代码，机器代码格式为 BytecodeStream 中的 ByteCode 对象。其中的 CodeGenerator 类是一个访

问者（参见 Visitor(5.11)模式）。

```
class CodeGenerator {
public:
    virtual void Visit(StatementNode*);
    virtual void Visit(ExpressionNode*);
    // ...
protected:
    CodeGenerator(BytecodeStream&);
protected:
    BytecodeStream& _output;
};
```

例如 CodeGenerator 类有两个子类 StackMachineCodeGenerator 和 RISCCodeGenerator，分别为不同的硬件体系结构生成机器代码。

ProgramNode 的每个子类在实现 Traverse 时，对它的 ProgramNode 子对象调用 Traverse。每个子类依次对它的子节点做同样的动作，这样一直递归下去。例如，ExpressionNode 像这样定义 Traverse：

```
void ExpressionNode::Traverse (CodeGenerator& cg) {
    cg.Visit(this);

    ListIterator<ProgramNode*> i(_children);

    for (i.First(); !i.IsDone(); i.Next()) {
        i.CurrentItem()->Traverse(cg);
    }
}
```

我们上述讨论的类构成了编译子系统，现在我们引入 Compiler 类，Compiler 类是一个 facade，它将所有部件集成在一起。Compiler 提供了一个简单的接口用于为特定的机器编译源代码并生成可执行代码。

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

    virtual void Compile(istream&, BytecodeStream&);
};

void Compiler::Compile (
    istream& input, BytecodeStream& output
) {
    Scanner scanner(input);
    ProgramNodeBuilder builder;
    Parser parser;

    parser.Parse(scanner, builder);

    RISCCodeGenerator generator(output);
    ProgramNode* parseTree = builder.GetRootNode();
    parseTree->Traverse(generator);
}
```

上面的实现在代码中固定了要使用的代码生成器的种类，因此程序员不需要指定目标机的结构。在仅有一种目标机的情况下，这是合理的。如果有多种目标机，我们可能希望改变 Compiler 构造函数使之能接受 CodeGenerator 为参数，这样程序员可以在实例化 Compiler 时指

定要使用的生成器。编译器的 facade 还可以对 Scanner 和 ProgramNodeBuilder 这样的其他一些参与者进行参数化以增加系统的灵活性，但是这并非 Facade 模式的主要任务，它的主要任务是为一一般情况简化接口。

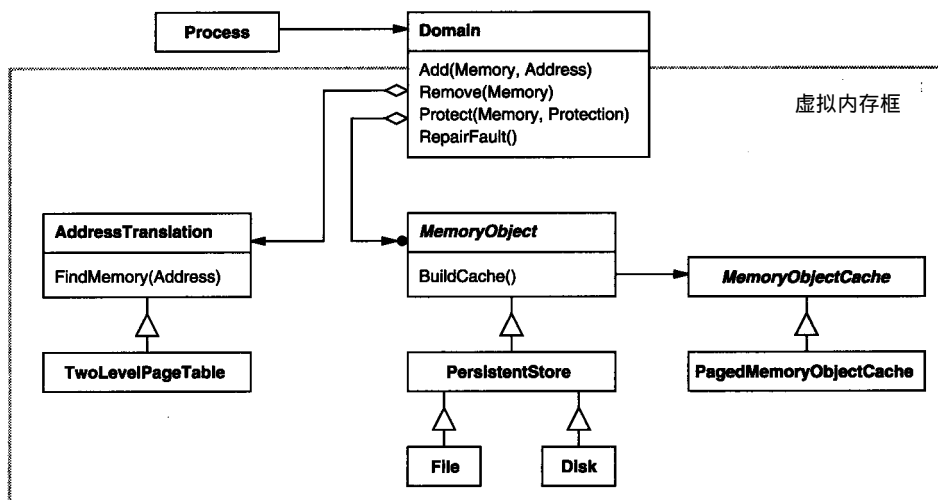
#### 10. 已知应用

在代码示例一节中的编译器例子受到了 ObjectWorks\Smalltalk 编译系统 [Par90] 的启发。

在 ET++ 应用框架 [WGM88] 中，应用程序可以有一个内置的浏览工具，用于在运行时时刻监视它的对象。这些浏览工具在一个独立的子系统中实现，这一子系统包含一个称为 ProgrammingEnvironment 的 Facade 类。这个 facade 定义了一些操作（如 InspectObject 和 InspectClass 等）用于访问这些浏览器。

ET++ 应用程序也可以不理睬这些内置的浏览功能，这时 ProgrammingEnvironment 对这些请求用空操作实现；也就是说，它们什么也不做。仅有 ETProgrammingEnvironment 子类用一些显示相应浏览器的操作实现这些请求。因此应用程序并不知道是否有内置浏览器存在，应用程序与浏览子系统的之间仅存在抽象的耦合关系。

Choices 操作系统 [CIRM93] 使用 facade 模式将多个框架组合到一起。Choices 中的关键抽象是进程 (process)、存储 (storage) 和地址空间 (address space)。每个抽象有一个相应的子系统，用框架实现，支持 Choices 系统在不同的硬件平台之间移植。其中的两个子系统有“代表”（也就是 facade），这两个代表分别是存储 (FileSystemInterface) 和地址空间 (Domain)。



例如，虚拟存储框架将 Domain 作为其 facade。一个 Domain 代表一个地址空间。它提供了虚存地址到内存对象、文件系统或后备存储设备（backing store）的偏移量之间的一个映射。Domain 支持在一个特定地址增加内存对象、删除内存对象以及处理页面错误。

正如上图所示，虚拟存储子系统内部有以下一些组件：

- MemoryObject 表示数据存储。
- MemoryObjectCache 将 MemoryObject 数据缓存在物理存储器中。MemoryObjectCache 实际上是一个 Strategy(5.9) 模式，由它定位缓存策略。
- AddressTranslation 封装了地址翻译硬件。

当发生缺页中断时，调用 RepairFault 操作，Domain 在引起缺页中断的地址处找到内存对象并将 RepairFault 操作代理给与这个内存对象相关的缓存。可以改变 Domain 的组件对 Domain 进行定制。

#### 11. 相关模式

Abstract Factory (3.1) 模式可以与 Facade 模式一起使用以提供一个接口，这一接口可用来以一种子系统独立的方式创建子系统对象。Abstract Factory 也可以代替 Facade 模式隐藏那些与平台相关的类。

Mediator (5.5) 模式与 Facade 模式的相似之处是，它抽象了一些已有的类的功能。然而，Mediator 的目的是对同事之间的任意通讯进行抽象，通常集中不属于任何单个对象的功能。Mediator 的同事对象知道中介者并与它通信，而不是直接与其他同类对象通信。相对而言，Facade 模式仅对子系统对象的接口进行抽象，从而使它们更容易使用；它并不定义新功能，子系统也不知道 facade 的存在。

通常来讲，仅需要一个 Facade 对象，因此 Facade 对象通常属于 Singleton (3.5) 模式。

## 4.6 FLYWEIGHT (享元) ——对象结构型模式

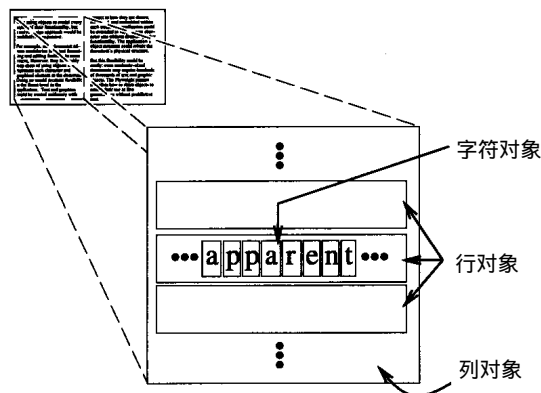
### 1. 意图

运用共享技术有效地支持大量细粒度的对象。

### 2. 动机

有些应用程序得益于在其整个设计过程中采用对象技术，但简单化的实现代价极大。

例如，大多数文档编辑器的实现都有文本格式化和编辑功能，这些功能在一定程度上是模块化的。面向对象的文档编辑器通常使用对象来表示嵌入的成分，例如表格和图形。尽管用对象来表示文档中的每个字符会极大地提高应用程序的灵活性，但是这些编辑器通常并不这样做。字符和嵌入成分可以在绘制和格式化时统一处理，从而在不影响其他功能的情况下能对应用程序进行扩展，支持新的字符集。应用程序的对象结构可以模拟文档的物理结构。下图显示了一个文档编辑器怎样使用对象来表示字符。



但这种设计的缺点在于代价太大。即使是一个中等大小的文档也可能要求成百上千的字符对象，这会耗费大量内存，产生难以接受的运行开销。所以通常并不是对每个字符都用一

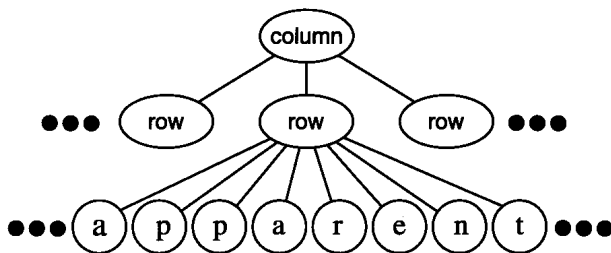


个对象来表示的。Flyweight模式描述了如何共享对象，使得可以细粒度地使用它们而无需高昂的代价。

flyweight是一个共享对象，它可以同时多个场景 (context) 中使用，并且在每个场景中 flyweight 都可以作为一个独立的对象——这一点与非共享对象的实例没有区别。flyweight 不能对它所运行的场景做出任何假设，这里的关键概念是内部状态和外部状态之间的区别。内部状态存储于 flyweight 中，它包含了独立于 flyweight 场景的信息，这些信息使得 flyweight 可以被共享。而外部状态取决于 Flyweight 场景，并根据场景而变化，因此不可共享。用户对象负责在必要的时候将外部状态传递给 Flyweight。

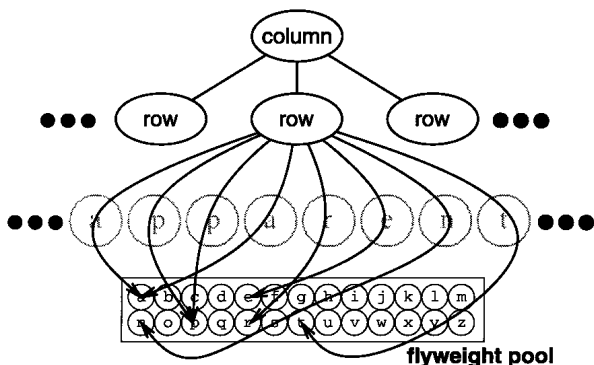
Flyweight 模式对那些通常因为数量太大而难以用对象来表示的概念或实体进行建模。例如，文档编辑器可以为字母表中的每一个字母创建一个 flyweight。每个 flyweight 存储一个字符代码，但它在文档中的位置和排版风格可以在字符出现时由正文排版算法和使用的格式化命令决定。字符代码是内部状态，而其他的信息则是外部状态。

逻辑上，文档中的给定字符每次出现都有一个对象与其对应，如下图所示。



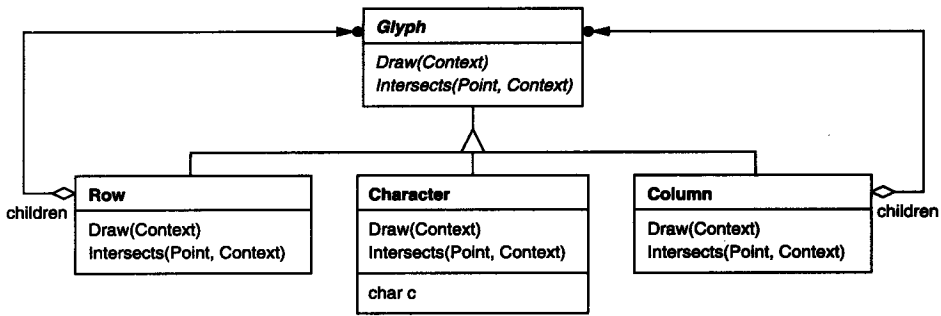
然而，物理上每个字符共享一个 flyweight 对象，而这个对象出现在文档结构中的不同地方。一个特定字符对象的每次出现都指向同一个实例，这个实例位于 flyweight 对象的共享池中。

这些对象的类结构如下图所示。Glyph 是图形对象的抽象类，其中有些对象可能是 flyweight。基于外部状态的那些操作将外部状态作为参量传递给它们。例如，Draw 和



Intersects 在执行之前，必须知道 glyph 所在的场景，如下页上图所示。

表示字母“a”的 flyweight 只存储相应的字符代码；它不需要存储字符的位置或字体。用户提供与场景相关的信息，根据此信息 flyweight 绘出它自己。例如，Row glyph 知道它的子女应该在哪儿绘制自己才能保证它们是横向排列的。因此 Row glyph 可以在绘制请求中向每一个子女传递它的位置。



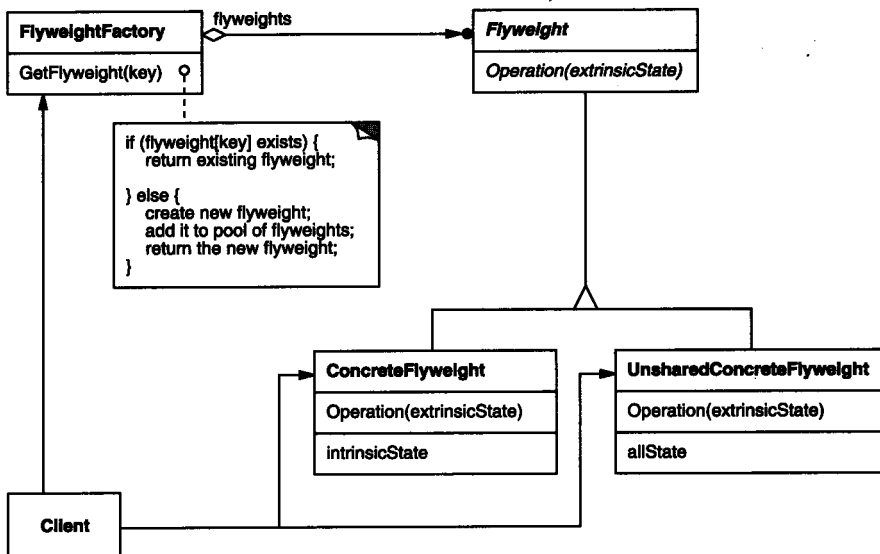
由于不同的字符对象数远小于文档中的字符数，因此，对象的总数远小于一个初次执行的程序所使用的对象数目。对于一个所有字符都使用同样的字体和颜色的文档而言，不管这个文档有多长，需要分配 100 个左右的字符对象（大约是 ASCII 字符集的数目）。由于大多数文档使用的字体颜色组合不超过 10 种，实际应用中这一数目不会明显增加。因此，对单个字符进行对象抽象是具有实际意义的。

### 3. 适用性

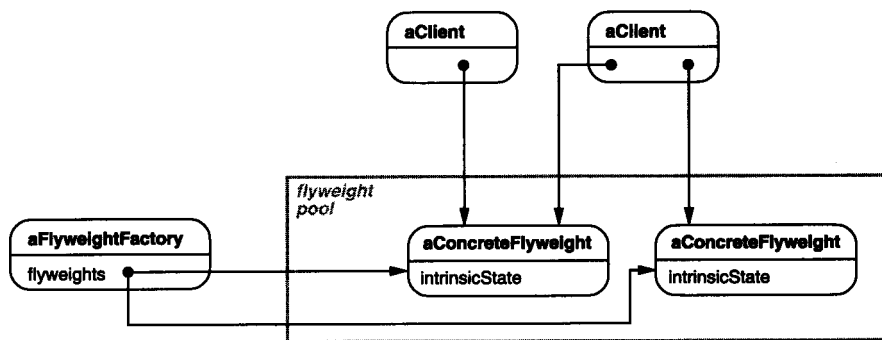
Flyweight 模式的有效性很大程度上取决于如何使用它以及在何处使用它。当以下情况都成立时使用 Flyweight 模式：

- 一个应用程序使用了大量的对象。
- 完全由于使用大量的对象，造成很大的存储开销。
- 对象的大多数状态都可变为外部状态。
- 如果删除对象的外部状态，那么可以用相对较少的共享对象取代很多组对象。
- 应用程序不依赖于对象标识。由于 Flyweight 对象可以被共享，对于概念上明显有别的对象，标识测试将返回真值。

### 4. 结构



下面的对象图说明了如何共享 flyweight。



## 5. 参与者

### • Flyweight (Glyph)

— 描述一个接口，通过这个接口 flyweight 可以接受并作用于外部状态。

### • ConcreteFlyweight(Character)

— 实现 Flyweight 接口，并为内部状态（如果有的话）增加存储空间。  
ConcreteFlyweight 对象必须是可共享的。它所存储的状态必须是内部的；即，它必须独立于 ConcreteFlyweight 对象的场景。

### • UnsharedConcreteFlyweight (Row,Column)

— 并非所有的 Flyweight 子类都需要被共享。Flyweight 接口使共享成为可能，但它并不强制共享。在 Flyweight 对象结构的某些层次，UnsharedConcreteFlyweight 对象通常将 ConcreteFlyweight 对象作为子节点（Row 和 Column 就是这样）。

### • FlyweightFactory

— 创建并管理 flyweight 对象。

— 确保合理地共享 flyweight。当用户请求一个 flyweight 时，FlyweightFactory 对象提供一个已创建的实例或者创建一个（如果不存在的话）。

### • Client

— 维持一个对 flyweight 的引用。

— 计算或存储一个（多个）flyweight 的外部状态。

## 6. 协作

• flyweight 执行时所需的状态必定是内部的或外部的。内部状态存储于 ConcreteFlyweight 对象之中；而外部对象则由 Client 对象存储或计算。当用户调用 flyweight 对象的操作时，将该状态传递给它。

• 用户不应直接对 ConcreteFlyweight 类进行实例化，而只能从 FlyweightFactory 对象得到 ConcreteFlyweight 对象，这可以保证对它们适当地进行共享。

## 7. 效果

使用 Flyweight 模式时，传输、查找和 / 或计算外部状态都会产生运行时的开销，尤其当 flyweight 原先被存储为内部状态时。然而，空间上的节省抵消了这些开销。共享的 flyweight 越多，空间节省也就越大。

存储节约由以下几个因素决定：

- 因为共享，实例总数减少的数目
- 对象内部状态的平均数目
- 外部状态是计算的还是存储的

共享的Flyweight越多，存储节约也就越多。节约量随着共享状态的增多而增大。当对象使用大量的内部及外部状态，并且外部状态是计算出来的而非存储的时候，节约量将达到最大。所以，可以用两种方法来节约存储：用共享减少内部状态的消耗，用计算时间换取对外部状态的存储。

Flyweight模式经常和Composite（4.3）模式结合起来表示一个层次式结构，这一层次式结构是一个共享叶节点的图。共享的结果是，Flyweight的叶节点不能存储指向父节点的指针。而父节点的指针将传给Flyweight作为它的外部状态的一部分。这对于该层次结构中对象之间相互通讯的方式将产生很大的影响。

#### 8. 实现

在实现Flyweight模式时，注意以下几点：

1) 删除外部状态 该模式的可用性在很大程度上取决于是否容易识别外部状态并将它从共享对象中删除。如果不同种类的外部状态和共享前对象的数目相同的话，删除外部状态不会降低存储消耗。理想的状况是，外部状态可以由一个单独的对象结构计算得到，且该结构的存储要求非常小。

例如，在我们的文档编辑器中，我们可以用一个单独的结构存储排版布局信息，而不是存储每一个字符对象的字体和类型信息，布局图保持了带有相同排版信息的字符的运行轨迹。当某字符绘制自己的时候，作为绘图遍历的副作用它接收排版信息。因为通常文档使用的字体和类型数量有限，将该信息作为外部信息来存储，要比内部存储高效得多。

2) 管理共享对象 因为对象是共享的，用户不能直接对它进行实例化，因此 FlyweightFactory可以帮助用户查找某个特定的Flyweight对象。FlyweightFactory对象经常使用关联存储帮助用户查找感兴趣的Flyweight对象。例如，在这个文档编辑器一例中的Flyweight工厂就有一个以字符代码为索引的Flyweight表。管理程序根据所给的代码返回相应的Flyweight，若不存在，则创建一个Flyweight。

共享还意味着某种形式的引用计数和垃圾回收，这样当一个Flyweight不再使用时，可以回收它的存储空间。然而，当Flyweight的数目固定而且很小的时候（例如，用于ASCII码的Flyweight），这两种操作都不必要。在这种情况下，Flyweight完全可以永久保存。

#### 9. 代码示例

回到我们文档编辑器的例子，我们可以为Flyweight的图形对象定义一个Glyph基类。逻辑上，Glyph是一些Composite类（见Composite（4.3）），它有图形化属性，并可以绘制自己。这里，我们重点讨论字体属性，但这种方法也同样适用于Glyph的其他图形属性。

```
class Glyph {
public:
    virtual ~Glyph();

    virtual void Draw(Window*, GlyphContext&);

    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont(GlyphContext&);
};
```

```

virtual void First(GlyphContext&);
virtual void Next(GlyphContext&);
virtual bool IsDone(GlyphContext&);
virtual Glyph* Current(GlyphContext&);

virtual void Insert(Glyph*, GlyphContext&);
virtual void Remove(GlyphContext&);
protected:
    Glyph();
};

```

Character 的子类存储一个字符代码：

```

class Character : public Glyph {
public:
    Character(char);

    virtual void Draw(Window*, GlyphContext&);
private:
    char _charcode;
};

```

为了避免给每一个 Glyph 的字体属性都分配存储空间，我们可以将该属性外部存储于 GlyphContext 对象中。GlyphContext 是一个外部状态的存储库，它维持 Glyph 与字体（以及其他一些可能的图形属性）之间的一种简单映射关系。对于任何操作，如果它需要知道在给定场景下 Glyph 字体，都会有一个 GlyphContext 实例作为参数传递给它。然后，该操作就可以查询 GlyphContext 以获取该场景中的字体信息了。这个场景取决于 Glyph 结构中的 Glyph 的位置。因此，当使用 Glyph 时，Glyph 子类的迭代和管理操作必须更新 GlyphContext。

```

class GlyphContext {
public:
    GlyphContext();
    virtual ~GlyphContext();

    virtual void Next(int step = 1);
    virtual void Insert(int quantity = 1);

    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);
private:
    int _index;
    BTree* _fonts;
};

```

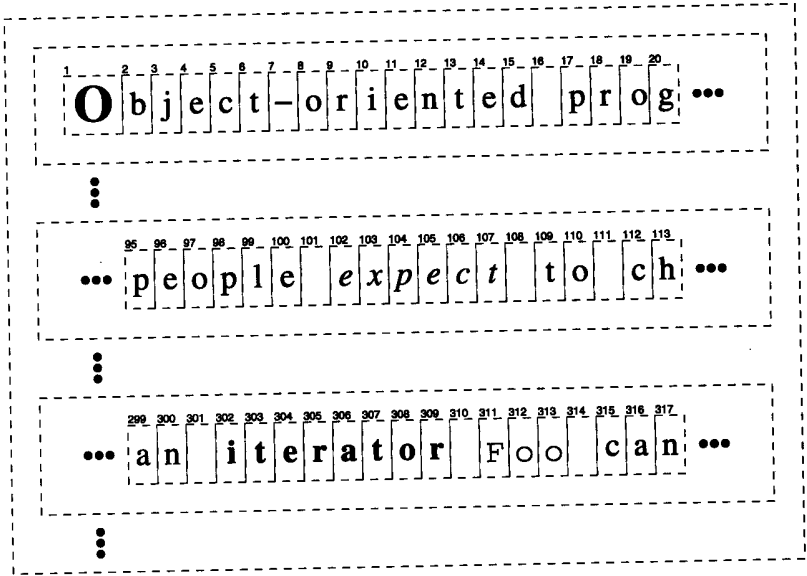
在遍历过程中，GlyphContext 必须它在 Glyph 结构中的当前位置。随着遍历的进行，GlyphContext::Next 增加 \_index 的值。Glyph 的子类（如，Row 和 Column）对 Next 操作的实现必须使得它在遍历的每一点都调用 GlyphContext::Next。

GlyphContext::GetFocus 将索引作为 Btree 结构的关键字，Btree 结构存储 glyph 到字体的映射。树中的每个节点都标有字符串的长度，而它给这个字符串字体信息。树中的叶节点指向一种字体，而内部的字符串分成了很多子字符串，每一个对应一种子节点。

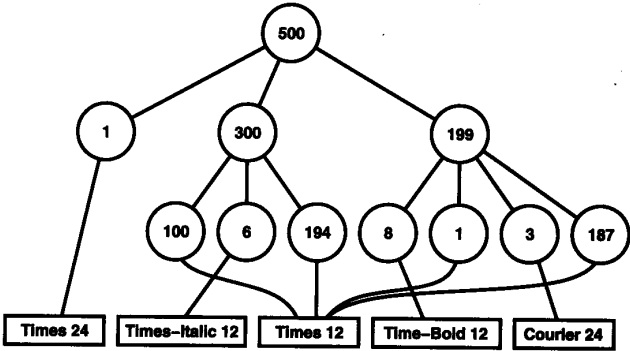
下页上图是从一个 glyph 组合中截取出来的：

字体信息的 BTree 结构可能如下：

内部节点定义 Glyph 索引的范围。当字体改变或者在 Glyph 结构中添加或删除 Glyph 时，Btree 将相应地被更新。例如，假定我们遍历到索引 102，以下代码将单词 “except” 的每个字符

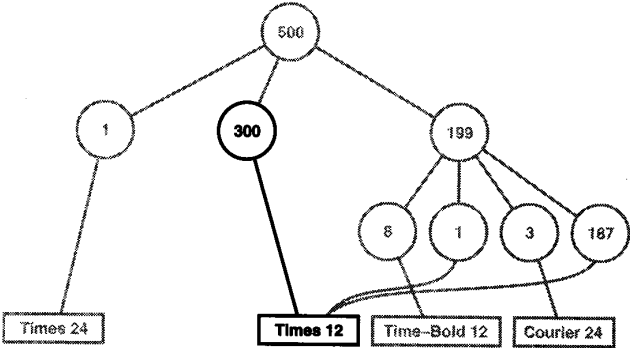


的字体设置为它周围的正文的字体（即，用Time 12字体，12-point Times Roman的一个实例）：



```
GlyphContext gc;  
Font* times12 = new Font("Times-Roman-12");  
Font* timesItalic12 = new Font("Times-Italic-12");  
// ...  
  
gc.SetFont(times12, 6);
```

新的Btree结构如下图（黑体显示变化）：

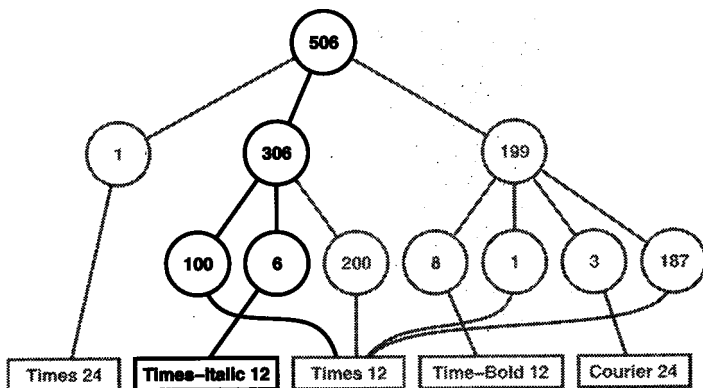




假设我们要在单词“expect”前用12-point Times Italic字体添加一个单词Don't(包括一个紧跟着的空格)。假定gc仍在索引位置102, 以下代码通知gc这个事件:

```
gc.Insert(6);
gc.SetFont(timesItalic12, 6);
```

Btree结构变为如下图所示:



当向GlyphContext查询当前Glyph的字体时, 它将向下搜寻Btree, 同时增加索引, 直至找到当前索引的字体为止。由于字体变化频率相对较低, 所以这棵树相对于Glyph结构较小。这将使得存储耗费较小, 同时也不会过多的增加查询时间。<sup>①</sup>

FlyweightFactory是我们需要的最后一个对象, 它负责创建Glyph并确保对它们进行合理共享。GlyphFactory类将实例化Character和其他类型的Glyph。我们只共享Character对象; 组合的Glyph要少得多, 并且它们的重要状态(如, 它们的子节点)必定是内部的。

```
const int NCHARCODES = 128;
```

```
class GlyphFactory {
public:
    GlyphFactory();
    virtual ~GlyphFactory();
    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    // ...
private:
    Character* _character[NCHARCODES];
};
```

\_character数组包含一些指针, 指向以字母代码为索引的Character Glyphs。该数组在构造函数中被初始化为零。

```
GlyphFactory::GlyphFactory() {
    for (int i = 0; i < NCHARCODES; ++i) {
        _character[i] = 0;
    }
}
```

CreateCharacter在字母符号数组中查找一个字符, 如果存在的话, 返回相应的Glyph。若不存在, CreateCharacter就创建一个Glyph, 将其放入数组中, 并返回它:

① 本机制中的查询时间与字体的变化频率成比例。当每一个字符的字体均不同时, 性能最差, 但通常这种情况极少。

```
Character* GlyphFactory::CreateCharacter (char c) {
    if (!_character[c]) {
        _character[c] = new Character(c);
    }

    return _character[c];
}
```

其他操作仅需在每次被调用时实例化一个新对象，因为非字符的 Glyph 不能被共享：

```
Row* GlyphFactory::CreateRow () {
    return new Row;
}

Column* GlyphFactory::CreateColumn () {
    return new Column;
}
```

我们可以忽略这些操作，让用户直接实例化非共享的 Glyph。然而，如果我们想让这些符号以后可以被共享，必须改变创建它们的客户程序代码。

## 10. 已知应用

Flyweight的概念最先是在 InterView 3.0[CL90]中提出并作为一种设计技术得到研究。它的开发者构建了一个强大的文档编辑器 Doc,作为flyweight概念的论证[CL92]。Doc使用符号对象来表示文档中的每一个字符。编辑器为每一个特定类型（定义它的图形属性）的字符创建一个 Glyph实例；所以，一个字符的内部状态包括字符代码和类型信息（类型表的索引）。<sup>①</sup>这意味着只有位置是外部状态，这就使得 Doc运行很快。文档由类 Document表示，它同时也是一个 FlyweightFactory。对Doc的测试表明共享Flyweight字符是高效的。通常，一个包含 180 000个字符的文档只要求分配大约 480个字符对象。

ET++ [WGM88]使用Flyweight来支持视觉风格独立性。<sup>②</sup>视觉风格标准影响用户界面各部分的布局（如，滚动条、按钮、菜单-统称为“窗口组件”）和它们的修饰成分（如，阴影、斜角）。widget将所有布局和绘制行为代理给一个单独的 Layout对象。改变Layout对象会改变视觉风格，即使在运行时刻也是这样。

每一个widget类都有一个Layout类与之相对应（如 ScrollbarLayout、MenubarLayout等）。使用这种方法，一个明显的问题是，使用单独的 Layout对象会使用户界面对象成倍增加，因为对每个用户界面对象，都会有一个附加的 Layout对象。为了避免这种开销，可用 Flyweight实现Layout对象。用Flyweight的效果很好，因为它们主要处理行为定义，而且很容易将一些较小的外部状态传递给它们，它们需要用这些状态来安排一个对象的位置或者对它进行绘制。

对象 Layout由Look对象创建和管理。Look类是一个 Abstract Factory（3.1），它用 GetButtonLayout和GetMenuBarLayout这样的操作检索一个特定的 Layout对象。对于每一个视觉风格标准，都有一个相应的 Look子类（如 MotifLook、OpenLook）提供相应的Layout对象。

顺便提一下，Layout对象其实是 Strategy（参见 Strategy(5.9)模式）。他们是用Flyweight实现的Strategy对象的一个例子。

## 11. 相关模式

① 在前面的代码示例一节中，类型信息是外部的，所以只有字符代码是内部状态。

② 实现视觉风格独立的另一种方法可参见 Abstract Factory(3.1)模式。

Flyweight模式通常和Composite(4.3)模式结合起来,用共享叶结点的有向无环图实现一个逻辑上的层次结构。

通常,最好用Flyweight实现State(5.8)和Strategy(5.9)对象。

## 4.7 PROXY (代理) ——对象结构型模式

### 1. 意图

为其他对象提供一种代理以控制对这个对象的访问。

### 2. 别名

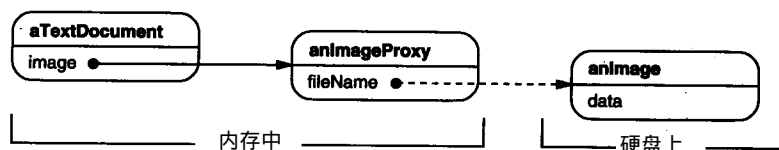
Surrogate

### 3. 动机

对一个对象进行访问控制的一个原因是为了只有在我们确实需要这个对象时才对它进行创建和初始化。我们考虑一个可以在文档中嵌入图形对象的文档编辑器。有些图形对象(如大型光栅图像)的创建开销很大。但是打开文档必须很迅速,因此我们在打开文档时应避免一次性创建所有开销很大的对象。因为并非所有这些对象在文档中都同时可见,所以也没有必要同时创建这些对象。

这一限制条件意味着,对于每一个开销很大的对象,应该根据需要进行创建,当一个图像变为可见时会产生这样的需要。但是在文档中我们用什么来代替这个图像呢?我们又如何才能隐藏根据需要创建图像这一事实,从而不会使得编辑器的实现复杂化呢?例如,这种优化不应影响绘制和格式化的代码。

问题的解决方案是使用另一个对象,即图像 Proxy,替代那个真正的图像。Proxy可以代替一个图像对象,并且在需要时负责实例化这个图像对象。



只有当文档编辑器激活图像代理的 Draw 操作以显示这个图像的时候,图像 Proxy 才创建真正的图像。Proxy 直接将随后的请求转发给这个图像对象。因此在创建这个图像以后,它必须有一个指向这个图像的引用。

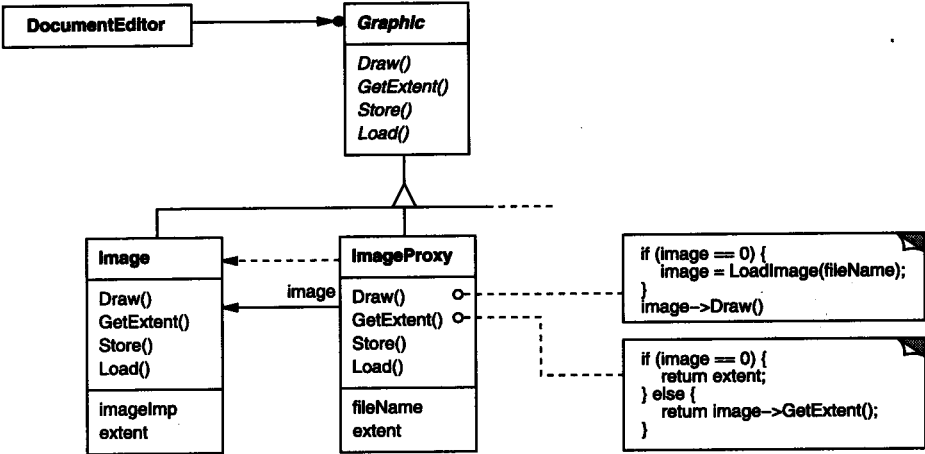
我们假设图像存储在一个独立的文件中。这样我们可以把文件名作为对实际对象的引用。Proxy 还存储了图像的尺寸(extent),即它的长和宽。有了图像尺寸,Proxy 无须真正实例化这个图像就可以响应格式化程序对图像尺寸的请求。

以下的类图更详细地阐述了这个例子。

文档编辑器通过抽象的 Graphic 类定义的接口访问嵌入的图像。ImageProxy 是那些根据需要创建的图像的类,ImageProxy 保存了文件名作为指向磁盘上的图像文件的指针。该文件名被作为一个参数传递给 ImageProxy 的构造器。

ImageProxy 还存储了这个图像的边框以及对真正的 Image 实例的指引,直到代理实例化真正的图像时,这个指引才有效。Draw 操作必须保证在向这个图像转发请求之前,它已经被实例化了。GetExtent 操作只有在图像被实例化后才向它传递请求,否则,ImageProxy 返回它存

储的图像尺寸。



4. 适用性

在需要用比较通用和复杂的对象指针代替简单的指针的时候，使用 Proxy 模式。下面是一些可以使用 Proxy 模式常见情况：

1) 远程代理 (Remote Proxy) 为一个对象在不同的地址空间提供局部代表。NEXTSTEP[Add94] 使用 NXProxy 类实现了这一目的。Coplien[Cop92] 称这种代理为“大使” (Ambassador)。

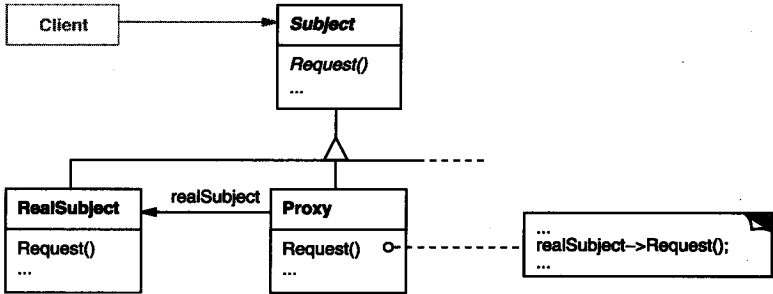
2) 虚代理 (Virtual Proxy) 根据需要创建开销很大的对象。在动机一节描述的 ImageProxy 就是这样一种代理的例子。

3) 保护代理 (Protection Proxy) 控制对原始对象的访问。保护代理用于对象应该有不同的访问权限的时候。例如，在 Choices 操作系统[CIRM93]中 KemeProxies 为操作系统对象提供了访问保护。

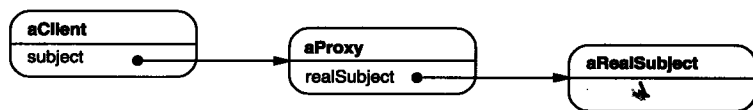
4) 智能指引 (Smart Reference) 取代了简单的指针，它在访问对象时执行一些附加操作。它的典型用途包括：

- 对指向实际对象的引用计数，这样当该对象没有引用时，可以自动释放它 (也称为 Smart Pointers[Ede92])。
- 当第一次引用一个持久对象时，将它装入内存。
- 在访问一个实际对象前，检查是否已经锁定了它，以确保其他对象不能改变它。

5. 结构



这是运行时刻一种可能的代理结构的对象图。



## 6. 参与者

### • Proxy (ImageProxy)

- 保存一个引用使得代理可以访问实体。若 RealSubject和Subject的接口相同，Proxy会引用Subject。
- 提供一个与Subject的接口相同的接口，这样代理就可以用来替代实体。
- 控制对实体的存取，并可能负责创建和删除它。
- 其他功能依赖于代理的类型：
  - Remote Proxy负责对请求及其参数进行编码，并向不同地址空间中的实体发送已编码的请求。
  - Virtual Proxy可以缓存实体的附加信息，以便延迟对它的访问。例如，动机一节中提到的ImageProxy缓存了图像实体的尺寸。
  - Protection Proxy检查调用者是否具有实现一个请求所必需的访问权限。

### • Subject (Graphic)

- 定义RealSubject 和Proxy的共用接口，这样就在任何使用 RealSubject的地方都可以使用Proxy。

### • RealSubject (Image)

- 定义Proxy所代表的实体。

## 7. 协作

- 代理根据其种类，在适当的时候向 RealSubject转发请求。

## 8. 效果

Proxy模式在访问对象时引入了一定程度的间接性。根据代理的类型，附加的间接性有多种用途：

- 1) Remote Proxy可以隐藏一个对象存在于不同地址空间的事实。
- 2) Virtual Proxy 可以进行最优化，例如根据要求创建对象。
- 3) Protection Proxies和Smart Reference都允许在访问一个对象时有一些附加的内务处理 (Housekeeping task)。

Proxy模式还可以对用户隐藏另一种称之为 copy-on-write的优化方式，该优化与根据需要创建对象有关。拷贝一个庞大而复杂的对象是一种开销很大的操作，如果这个拷贝根本没有被修改，那么这些开销就没有必要。用代理延迟这一拷贝过程，我们可以保证只有当这个对象被修改的时候才对它进行拷贝。

在实现 Copy-on-write时必须对实体进行引用计数。拷贝代理仅会增加引用计数。只有当用户请求一个修改该实体的操作时，代理才会真正的拷贝它。在这种情况下，代理还必须减少实体的引用计数。当引用的数目为零时，这个实体将被删除。

Copy-on-Write可以大幅度的降低拷贝庞大实体时的开销。

## 9. 实现

Proxy模式可以利用以下一些语言特性：

1) 重载C++中的存取运算符 C++支持重载运算符->。重载这一运算符使你可以在撤消对一个对象的引用时，执行一些附加的操作。这一点可以用于实现某些种类的代理；代理的作用就象一个指针。

下面的例子说明怎样使用这一技术实现一个称为 ImagePtr的虚代理。

```
class Image;
extern Image* LoadAnImageFile(const char*);
    // external function

class ImagePtr {
public:
    ImagePtr(const char* imageFile);
    virtual ~ImagePtr();

    virtual Image* operator->();
    virtual Image& operator*();
private:
    Image* LoadImage();
private:
    Image* _image;
    const char* _imageFile;
};

ImagePtr::ImagePtr (const char* theImageFile) {
    _imageFile = theImageFile;
    _image = 0;
}

Image* ImagePtr::LoadImage () {
    if (_image == 0) {
        _image = LoadAnImageFile(_imageFile);
    }
    return _image;
}
```

重载的->和\*运算符使用LoadImage将\_image返回给它的调用者(如果必要的话装入它)。

```
Image* ImagePtr::operator-> () {
    return LoadImage();
}

Image& ImagePtr::operator* () {
    return *LoadImage();
}
```

该方法使你能够通过 ImagePtr对象调用Image操作，而省去了把这些操作作为 ImagePtr接口的一部分的麻烦。

```
ImagePtr image = ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
// (image.operator->())->Draw(Point(50, 100))
```

请注意这里的image代理起到一个指针的作用，但并没有将它定义为一个指向 Image的指针。这意味着你不能把它当作一个真正的指向 Image的指针来使用。因此在使用此方法时用户应区别对待Image对象和Imageptr对象。

重载成员访问运算符并非对每一种代理来说都是好办法。有些代理需要清楚地知道调用了哪个操作，重载运算符的方法在这种情况下行不通。

考虑在目的一节提到的虚代理的例子，图像应该在一个特定的时刻被装载——也就是在



Draw操作被调用时——而不是在只要引用这个图像就装载它。重载访问操作符不能作出这种区分。在这种情况下我们只能人工实现每一个代理操作，向实体转发请求。

正如示例代码中所示的那样，这些操作之间非常相似。一般来说，所有的操作在向实体转发请求之前，都要检验这个要求是否合法，原始对象是否存在等。但重复写这些代码很麻烦，因此我们一般用一个预处理程序自动生成它。

2) 使用Smalltalk中的doesNotUnderstand Smalltalk提供一个hook方法可以用来自动转发请求。当用户向接受者发送一个消息，但是这个接受者没有相关方法的时候，Smalltalk调用方法doesNotUnderstand: amessage。Proxy类可以重定义doesNotUnderstand以便向它的实体转发这个消息。

为了保证一个请求真正被转发给实体，而不是无声无息的被代理所吸收，我们可以定义一个不理解任何信息的Proxy类。Smalltalk定义了一个没有任何超类的Proxy类，实现了这个目的。<sup>①</sup>

doesNotUnderstand：的主要缺点在于：大多数Smalltalk系统都有一些由虚拟机直接控制的特殊消息，而这些消息并不引起通常的方法查找。唯一一个通常用Object实现（因而可以影响代理）的符号是恒等运算符==。

如果你准备使用doesNotUnderstand：来实现Proxy的话，你必须围绕这一问题进行设计。对代理的标识并不意味着对真正实体的标识。doesNotUnderstand：另一个缺点是，它主要用作错误处理，而不是创建代理，因此一般来说它的速度不是很快。

3) Proxy并不总是需要知道实体的类型 若Proxy类能够完全通过一个抽象接口处理它的实体，则无须为每一个RealSubject类都生成一个Proxy类；Proxy可以统一处理所有的RealSubject类。但是如果Proxy要实例化RealSubjects（例如在virtual proxy中），那么它们必须知道具体的类。

另一个实现方面的问题涉及到在实例化实体以前怎样引用它。有些代理必须引用它们的实体，无论它是在硬盘上还是在内存中。这意味着它们必须使用某种独立于地址空间的对象标识符。在目的一节中，我们采用一个文件名来实现这种对象标识符。

#### 10. 代码示例

以下代码实现了两种代理：在目的一节描述的Virtual Proxy，和用doesNotUnderstand:实现的Proxy。<sup>②</sup>

1) Virtual Proxy Graphic类为图形对象定义一个接口。

```
class Graphic {
public:
    virtual ~Graphic();

    virtual void Draw(const Point& at) = 0;
    virtual void HandleMouse(Event& event) = 0;

    virtual const Point& GetExtent() = 0;

    virtual void Load(istream& from) = 0;
    virtual void Save(ostream& to) = 0;
```

① 对NEXTSTEP[Add94]中的分布式对象（尤其是类NXProxy）的实现就使用了该技术。NEXTSTEP中等价的hook方法是forward，这一实现重定义了forward方法。

② Iterator模式（5.4）描述了另一种类型的Proxy。

```
protected:
    Graphic();
};
```

Image类实现了Graphic接口用来显示图像文件。Image重定义Handlemouse操作，使得用户可以交互的调整图像的尺寸。

```
class Image : public Graphic {
public:
    Image(const char* file); // loads image from a file
    virtual ~Image();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();
    virtual void Load(istream& from);
    virtual void Save(ostream& to);
private:
    // ...
};
```

ImageProxy和Image具有相同的接口：

```
class ImageProxy : public Graphic {
public:
    ImageProxy(const char* imageFile);
    virtual ~ImageProxy();

    virtual void Draw(const Point& at);
    virtual void HandleMouse(Event& event);

    virtual const Point& GetExtent();

    virtual void Load(istream& from);
    virtual void Save(ostream& to);
protected:
    Image* GetImage();
private:
    Image* _image;
    Point _extent;
    char* _fileName;
};
```

构造函数保存了存储图像的文件名的本地拷贝，并初始化 \_extent和\_image：

```
ImageProxy::ImageProxy (const char* fileName) {
    _fileName = strdup(fileName);
    _extent = Point::Zero; // don't know extent yet
    _image = 0;
}

Image* ImageProxy::GetImage() {
    if (_image == 0) {
        _image = new Image(_fileName);
    }
    return _image;
}
```

如果可能的话，GetExtent的实现部分返回缓存的图像尺寸；否则从文件中装载图像。Draw用来装载图像，HandleMouse则向实际图像转发这个事件。

```
const Point& ImageProxy::GetExtent () {
    if (_extent == Point::Zero) {
        _extent = GetImage()->GetExtent();
    }
}
```

```

        return _extent;
    }
    void ImageProxy::Draw (const Point& at) {
        GetImage()->Draw(at);
    }

    void ImageProxy::HandleMouse (Event& event) {
        GetImage()->HandleMouse(event);
    }

```

Save操作将缓存的图像尺寸和文件名保存在一个流中。Load得到这个信息并初始化相应的成员函数。

```

void ImageProxy::Save (ostream& to) {
    to << _extent << _fileName;
}

void ImageProxy::Load (istream& from) {
    from >> _extent >> _fileName;
}

```

最后，假设我们有一个类 TextDocument 能够包含 Graphic 对象：

```

class TextDocument {
public:
    TextDocument();

    void Insert(Graphic*);
    // ...
};

```

我们可以用以下方式把 ImageProxy 插入到文本文件中。

```

TextDocument* text = new TextDocument;
// ...
text->Insert(new ImageProxy("anImageFileName"));

```

2) 使用 doesNotUnderstand 的 Proxy 在 Smalltalk 中，你可以定义超类为 nil<sup>⊖</sup> 的类，同时定义 doesNotUnderstand: 方法处理消息，这样构建一些通用的代理。

在以下程序中我们假设代理有一个 realSubject 方法，该方法返回它的实体。在 ImageProxy 中，该方法将检查是否已创建了 Image，并在必要的时候创建它，最后返回 Image。它使用 perform: withArguments: 来执行被保留在实体中的那些消息。

```

doesNotUnderstand: aMessage
    ^ self realSubject
        perform: aMessage selector
        withArguments: aMessage arguments

```

doesNotUnderstand: 的参数是 Message 的一个实例，它表示代理不能理解的消息。所以，代理在转发消息给实体之前，首先确定实体的存在性，并由此对所有的消息做出响应。

doesNotUnderstand: 的一个优点是它可以执行任意的处理过程。例如，我们可以用这样的方式生成一个 protection proxy，即指定一个可以接受的消息的集合 legalMessages，然后给这个代理定义以下方法。

```

doesNotUnderstand: aMessage
    ^ (legalMessages includes: aMessage selector)
        ifTrue: [self realSubject

```

⊖ 几乎所有的类最终均以 Object（对象）作为他们的超类。所以说这句话等于说“定义了一个类，它的超类不是 Object”。

```
perform: aMessage selector  
withArguments: aMessage arguments]  
ifFalse: [self error: 'Illegal operator']
```

这个方法在向实体转发一个消息之前，检查它的合法性。如果不是合法的，那么发送 error: 给代理，除非代理定义 error:，这将产生一个错误的无限循环。因此，error: 的定义应该同所有它用到的方法一起从 Object 类中拷贝。

#### 11. 已知应用

动机一节中 virtual proxy 的例子来自于 ET++ 的文本构建块类。

NEXTSTEP[Add94]使用代理(类 NXProxy 的实例)作为可分布对象的本地代表，当客户请求远程对象时，服务器为这些对象创建代理。收到消息后，代理对消息和它的参数进行编码，并将编码后的消息传递给远程实体。类似的，实体对所有的返回结果编码，并将它们返回给 NXProxy 对象。

McCullough [McC87] 讨论了在 Smalltalk 中用代理访问远程对象的问题。Pascoe [Pas86] 讨论了如何用“封装器”(Encapsulators) 控制方法调用的副作用以及进行访问控制。

#### 12. 相关模式

Adapter(4.1)：适配器 Adapter 为它所适配的对象提供了一个不同的接口。相反，代理提供了与它的实体相同的接口。然而，用于访问保护的代理可能会拒绝执行实体会执行的操作，因此，它的接口实际上可能只是实体接口的一个子集。

Decorator(4.4)：尽管 decorator 的实现部分与代理相似，但 decorator 的目的不一样。Decorator 为对象添加一个或多个功能，而代理则控制对对象的访问。

代理的实现与 decorator 的实现类似，但是在相似的程度上有差别。Protection Proxy 的实现可能与 decorator 的实现差不多。另一方面，Remote Proxy 不包含对实体的直接引用，而只是一个间接引用，如“主机 ID，主机上的局部地址。”Virtual Proxy 开始的时候使用一个间接引用，例如一个文件名，但最终将获取并使用一个直接引用。

## 4.8 结构型模式的讨论

你可能已经注意到了结构型模式之间的相似性，尤其是它们的参与者和协作之间的相似性。这可能是因为结构型模式依赖于同一个很小的语言机制集合构造代码和对象：单继承和多重继承机制用于基于类的模式，而对象组合机制用于对象式模式。但是这些相似性掩盖了这些模式的不同意图。在本节中，我们将对比这些结构型模式，使你对它们各自的优点有所了解。

### 4.8.1 Adapter 与 Bridge

Adapter (4.1) 模式和 Bridge (4.2) 模式具有一些共同的特征。它们都给另一对象提供了一定程度上的间接性，因而有利于系统的灵活性。它们都涉及到从自身以外的一个接口向这个对象转发请求。

这些模式的不同之处主要在于它们各自的用途。Adapter 模式主要是为了解决两个已有接口之间不匹配的问题。它不考虑这些接口是怎样实现的，也不考虑它们各自可能会如何演化。这种方式不需要对两个独立设计的类中的任一个进行重新设计，就能够使它们协同工作。另

一方面，Bridge模式则对抽象接口与它的（可能是多个）实现部分进行桥接。虽然这一模式允许你修改实现它的类，它仍然为用户提供了一个稳定的接口。Bridge模式也会在系统演化时适应新的实现。

由于这些不同点，Adapter和Bridge模式通常被用于软件生命周期的不同阶段。当你发现两个不兼容的类必须同时工作时，就有必要使用Adapter模式，其目的般是为了避免代码重复。此处耦合不可预见。相反，Bridge的使用者必须事先知道：一个抽象将有多个实现部分，并且抽象和实现两者是独立演化的。Adapter模式在类已经设计好后实施；而Bridge模式在设计类之前实施。这并不意味着Adapter模式不如Bridge模式，只是因为它们针对了不同的问题。

你可能认为facade(参见Facade(4.5))是另外一组对象的适配器。但这种解释忽视了一个事实：即Facade定义一个新的接口，而Adapter则复用原有的接口。记住，适配器使两个已有的接口协同工作，而不是定义一个全新的接口。

#### 4.8.2 Composite、Decorator与Proxy

Composite(4.3)模式和Decorator(4.4)模式具有类似的结构图，这说明它们都基于递归组合来组织可变量目的对象。这一共同点可能会使你认为，decorator对象是一个退化的composite，但这一观点没有领会Decorator模式要点。相似点仅止于递归组合，同样，这是因为这两个模式的目的不同。

Decorator旨在使你能够不需要生成子类即可给对象添加职责。这就避免了静态实现所有功能组合，从而导致子类急剧增加。Composite则有不同的目的，它旨在构造类，使多个相关的对象能够以统一的方式处理，而多重对象可以被当作一个对象来处理。它重点不在于修饰，而在于表示。

尽管它们的目的截然不同，但却具有互补性。因此Composite和Decorator模式通常协同使用。在使用这两种模式进行设计时，我们无需定义新的类，仅需将一些对象插接在一起即可构建应用。这时系统中将会有有一个抽象类，它有一些composite子类和decorator子类，还有一些实现系统的基本构建模块。此时，composites和decorator将拥有共同的接口。从Decorator模式的角度看，composite是一个ConcreteComponent。而从composite模式的角度看，decorator则是一个Leaf。当然，他们不一定要同时使用，正如我们所见，它们的目的有很大的差别。

另一种与Decorator模式结构相似的模式是Proxy(4.7)。这两种模式都描述了怎样为对象提供一定程度上的间接引用，proxy和decorator对象的实现部分都保留了指向另一个对象的指针，它们向这个对象发送请求。然而同样，它们具有不同的设计目的。

像Decorator模式一样，Proxy模式构成一个对象并为用户提供一致的接口。但与Decorator模式不同的是，Proxy模式不能动态地添加或分离性质，它也不是为递归组合而设计的。它的目的是，当直接访问一个实体不方便或不符合需要时，为这个实体提供一个替代者，例如，实体在远程设备上，访问受到限制或者实体是持久存储的。

在Proxy模式中，实体定义了关键功能，而Proxy提供（或拒绝）对它的访问。在Decorator模式中，组件仅提供了部分功能，而一个或多个Decorator负责完成其他功能。Decorator模式适用于编译时不能（至少不方便）确定对象的全部功能的情况。这种开放性使

递归组合成为 Decorator 模式中一个必不可少的部分。而在 Proxy 模式中则不是这样，因为 Proxy 模式强调一种关系（Proxy 与它的实体之间的关系），这种关系可以静态的表达。

模式间的这些差异非常重要，因为它们针对了面向对象设计过程中一些特定的经常发生问题的解决方法。但这并不意味着这些模式不能结合使用。可以设想有一个 proxy-decorator，它可以给 proxy 添加功能，或是一个 decorator-proxy 用来修饰一个远程对象。尽管这种混合可能有用（我们手边还没有现成的例子），但它们可以分割成一些有用的模式。