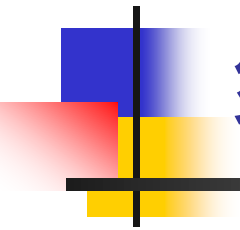
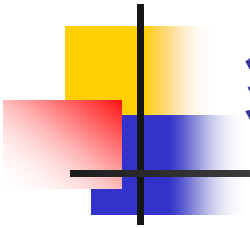
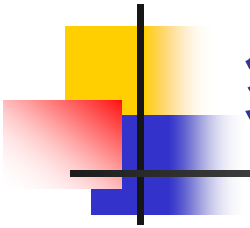


结构型模式



结构型模式

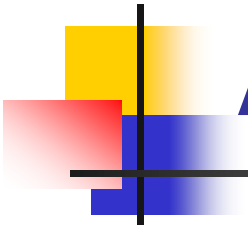
- 如何组合类和对象以获得更大的结构
- 不是对接口和实现进行组合，而是描述了如何对一些对象进行组合，从而实现新功能得一些方法
  - 动态的对象组合关系



结构型模式

- Composite模式
  - 如何构造一个类层次式结构
  - 基本对象和组合对象
- Proxy模式
  - Proxy对象作为其他对象的一个替代或占位符
- Flyweight
  - 对象共享机制
- Facade模式
  - 如何用单个对象表示整个子系统
- Decorator模式
  - 如何动态地为对象添加职责

4.1 Adapter适配器模式

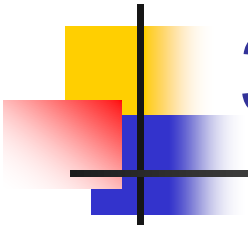


Adapter模式

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模式使得原本因为接口不兼容而不能一起工作的那些类可以一起工作。

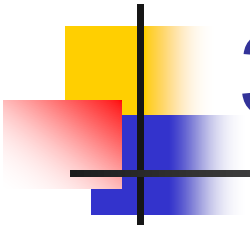
2. 别名

   包装器 Wrapper



3. 动机

- 问题：
  - 通用工具箱类应用于具体专业应用领域时，其接口可能与需要的接口不匹配
  - e.g.
    - 专业应用：绘图编辑器，其中使用TextShape类（Shape的子类）
    - 成品用户工具箱提供的：TextView类

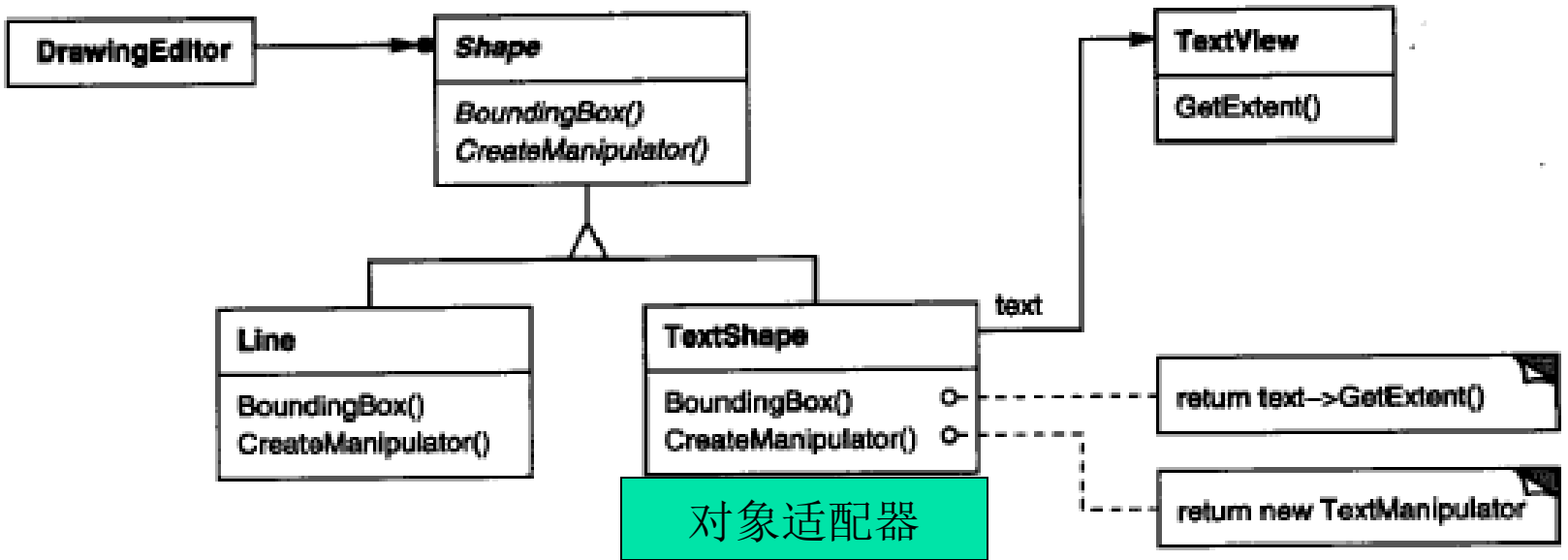


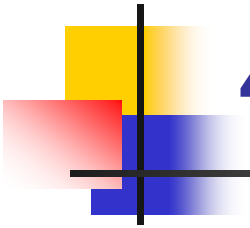
3. 动机

- 解决方式
  - 方式1：继承Shape类的接口和TextView的实现
    - Adapter模式的类版本

3. 动机

- 解决方式
  - 方式2：将TextView实例作为TextShape的组成部分，使用TextView接口实现TextShape
    - Adapter模式的对象版本



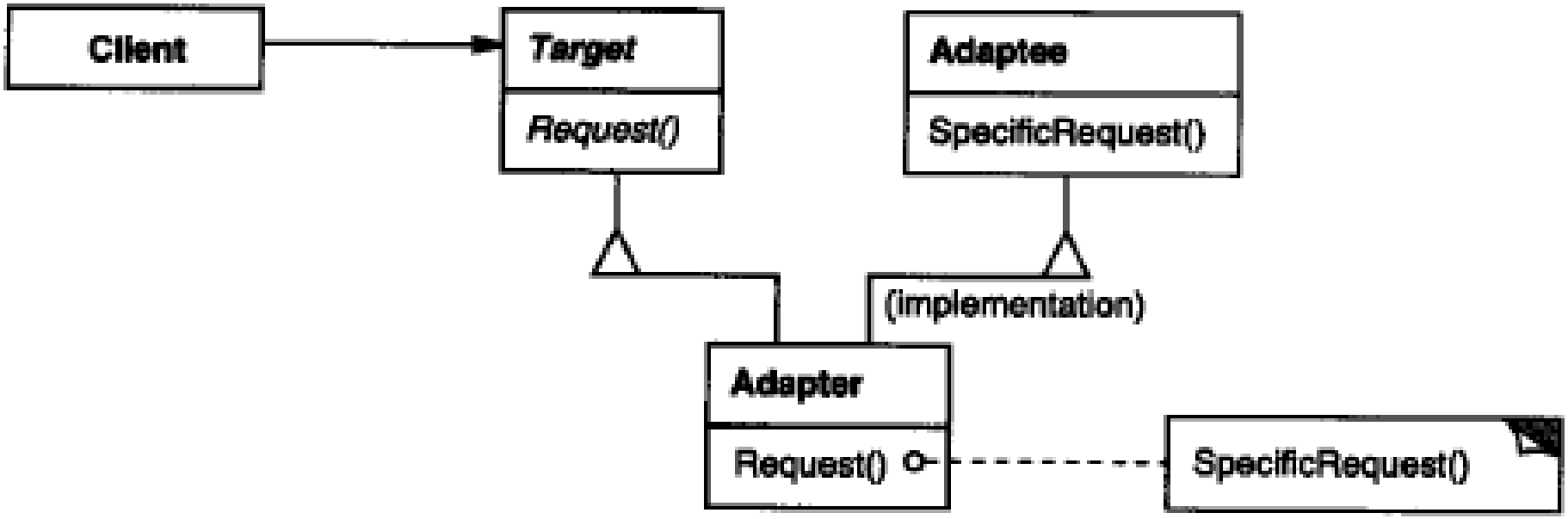


4. 适用性

- 想使用一个已经存在的类，而它的接口不符合需求
- 想创建一个可复用的类，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）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。

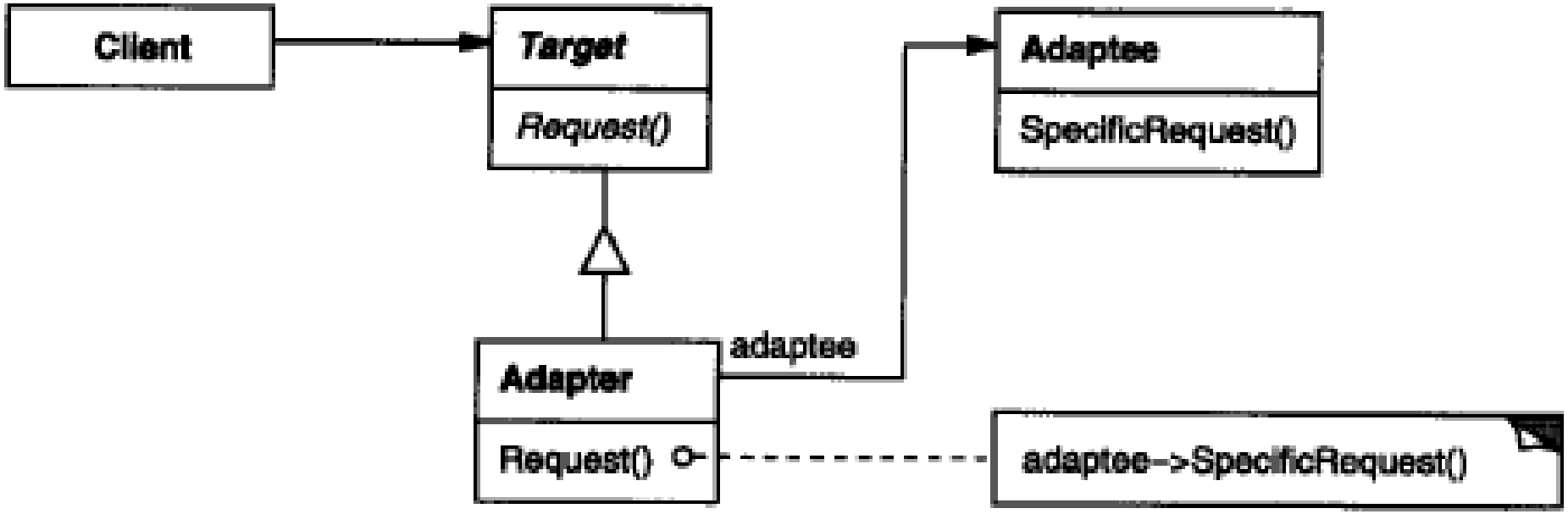
5. 结构

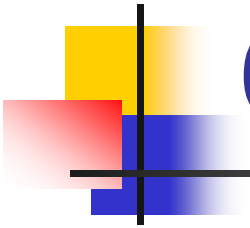
- 类Adapter



5. 结构

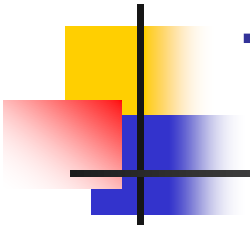
- 对象Adapter





6. 参与者

- Target(Shape)
  - 定义Client使用的接口，与特定领域相关
- 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. 协同

- Client在Adapter实例上调用一些操作
- 然后适配器Adapter调用Adaptee的操作去实现这个请求

8. 效果

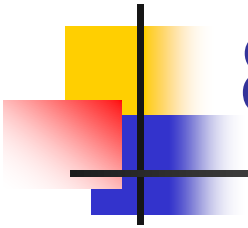
类适配器与对象适配器的区别

- 类适配器
  - 用一个具体的Adapter类对Adaptee和Target进行匹配
    - 不能处理：匹配一个类以及其所有子类
  - Adapter可以重定义Adaptee的部分行为
  - 仅引入一个对象，不需要额外指针以得到Adaptee

8. 效果

类适配器与对象适配器的区别

- 对象适配器
  - 运行一个Adapter和多个Adaptee同时工作
    - 即Adaptee本身以及它的所有子类
    - Adapter可以一次为所有的Adaptee添加功能
  - 重定义Adaptee的行为比较困难
    - 需生成Adaptee的子类
    - Adapter引用该子类

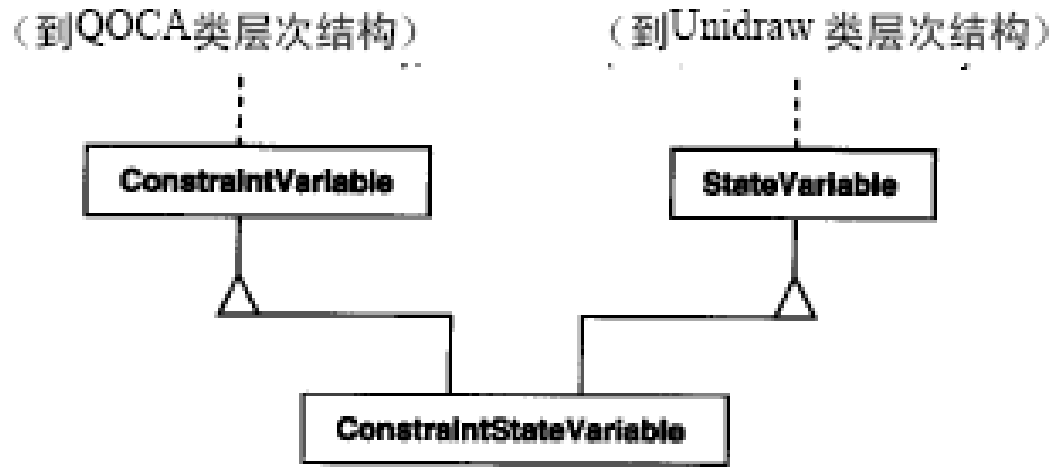


8. 效果

- 其他需要考虑因素
  - Adapter的匹配程度
    - Adapter的工作量取决于Target接口和Adaptee接口的相似程度
  - Pluggable Adapter ： 具有内部接口匹配的类
    - e.g. TreeDisplay窗口组件可以适配：
      - 目录层次结构：使用GetSubDirectories操作访问子目录
      - 继承式层次接口：GetSubclasses

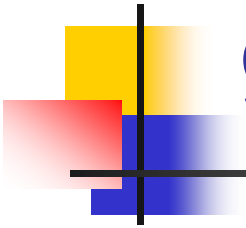
8. 效果

- 使用双向适配器提供透明操作
  - 适配器不对所有客户都透明
    - 被适配的对象不再兼容Adaptee接口
    - 因此不是所有Adaptee对象可以被使用的地方，都可以使用被适配的对象



双向适配器ConstraintStateVariable：
使得两个接口互相匹配

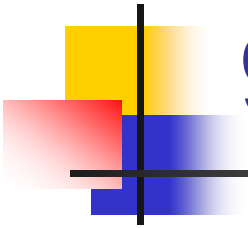
可行的解决方案：多重继承



9. 实现

1. 使用C++实现类适配器

- Adapter类
  - 公共方式集成Target类
  - 私有方式集成Adaptee类



9. 实现

2. Pluggable Adapter

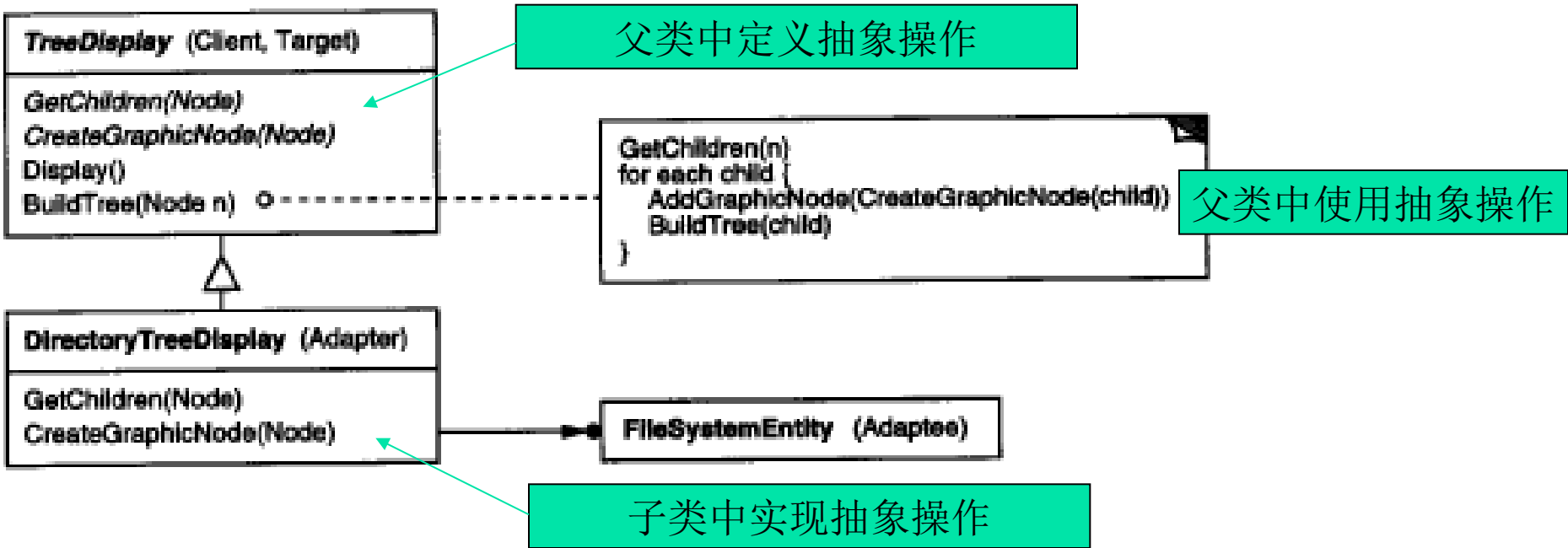
- 为Adaptee找到一个“窄”接口
  - 即可用于适配的最小操作集
  - e.g TreeDisplay的最小接口集合
    - 一个操作定义如何在层次接口中表示一个节点
    - 另一个操作返回该节点的子节点

9. 实现

2. Pluggable Adapter

- “窄”接口实现途径

a) 使用抽象操作

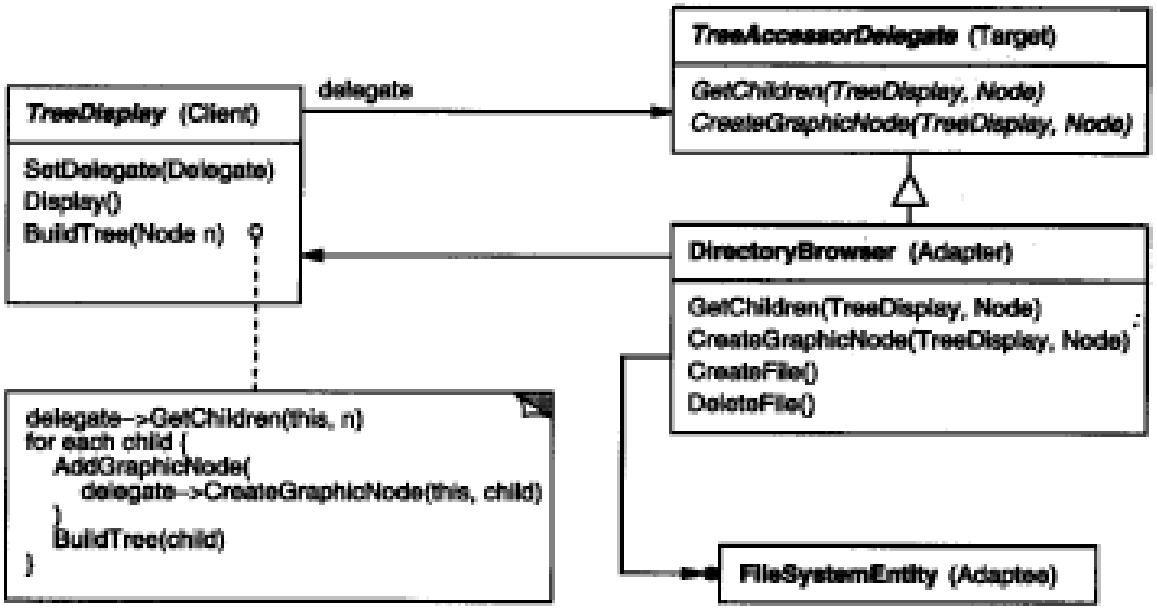


9. 实现

2. Pluggable Adapter

- “窄”接口实现途径

b) 使用代理对象



代理对象父类
其接口即为“窄”接口

子类

10. 代码示例

- 需要适配的接口

```
class Shape {
public:
    Shape();
    virtual void BoundingBox(
        Point& bottomLeft, Point& topRight
    ) const;
    virtual Manipulator* CreateManipulator() const;
};
```

Shape

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

TextView

10. 代码示例

- 类适配器方式： 用多重继承适配接口

公共方式继承接口，私有方式继承实现

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

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

10. 代码示例

- 类适配器方式

对TextView接口进行转换以匹配Shape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);
}
```

直接转发请求

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

实现TextView不支持的操作

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

10. 代码示例

- 对象适配器方式

```
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;
};
```

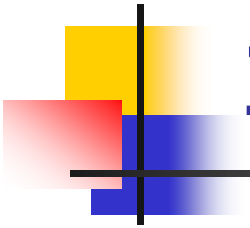
维护一个指向TextView的指针

将请求传递给TextView对象

```
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();
}
```

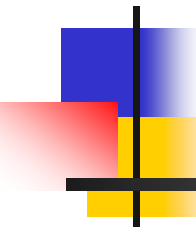
没有使用到TextView对象的代码

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



12. 相关模式

- Bridge模式 VS 对象适配器
  - Bridge的目的：接口部分与实现部分分离
  - Adapter：改变一个已有对象的接口
- Decorator
  - 增强其他对象的功能而不改变其接口
  - 对应用程序的透明性比Adapter要好
  - Decorator支持递归组合，而Adapter难于实现
- Proxy模式
  - 在不改变接口的条件下，为另一个对象定义了一个代理



4.2 Bridge模式

Bridge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

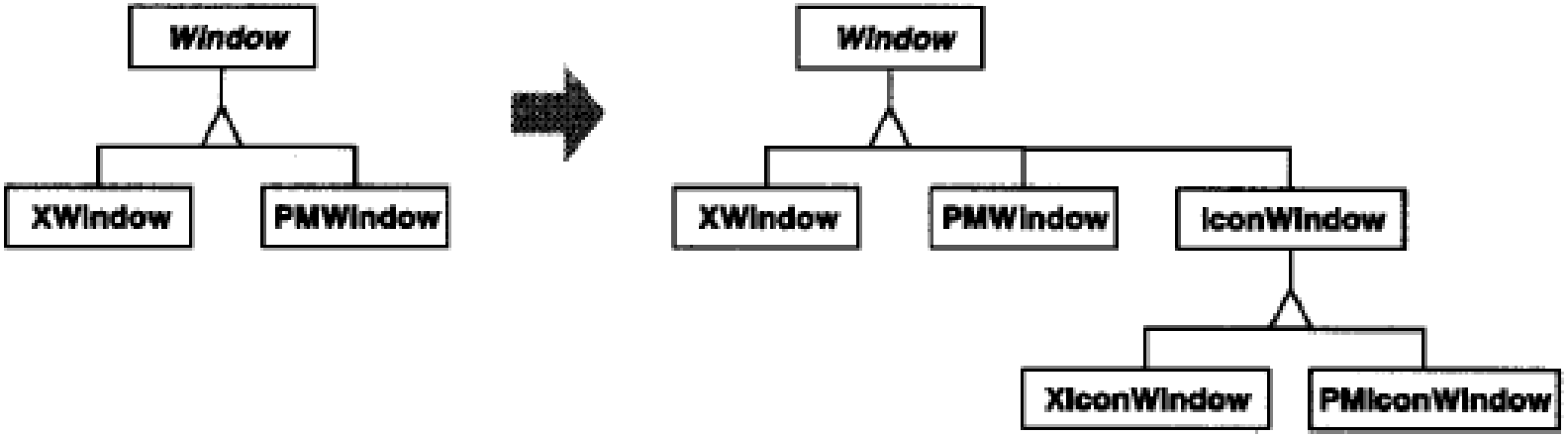
Handle / Body

句柄 / 实体

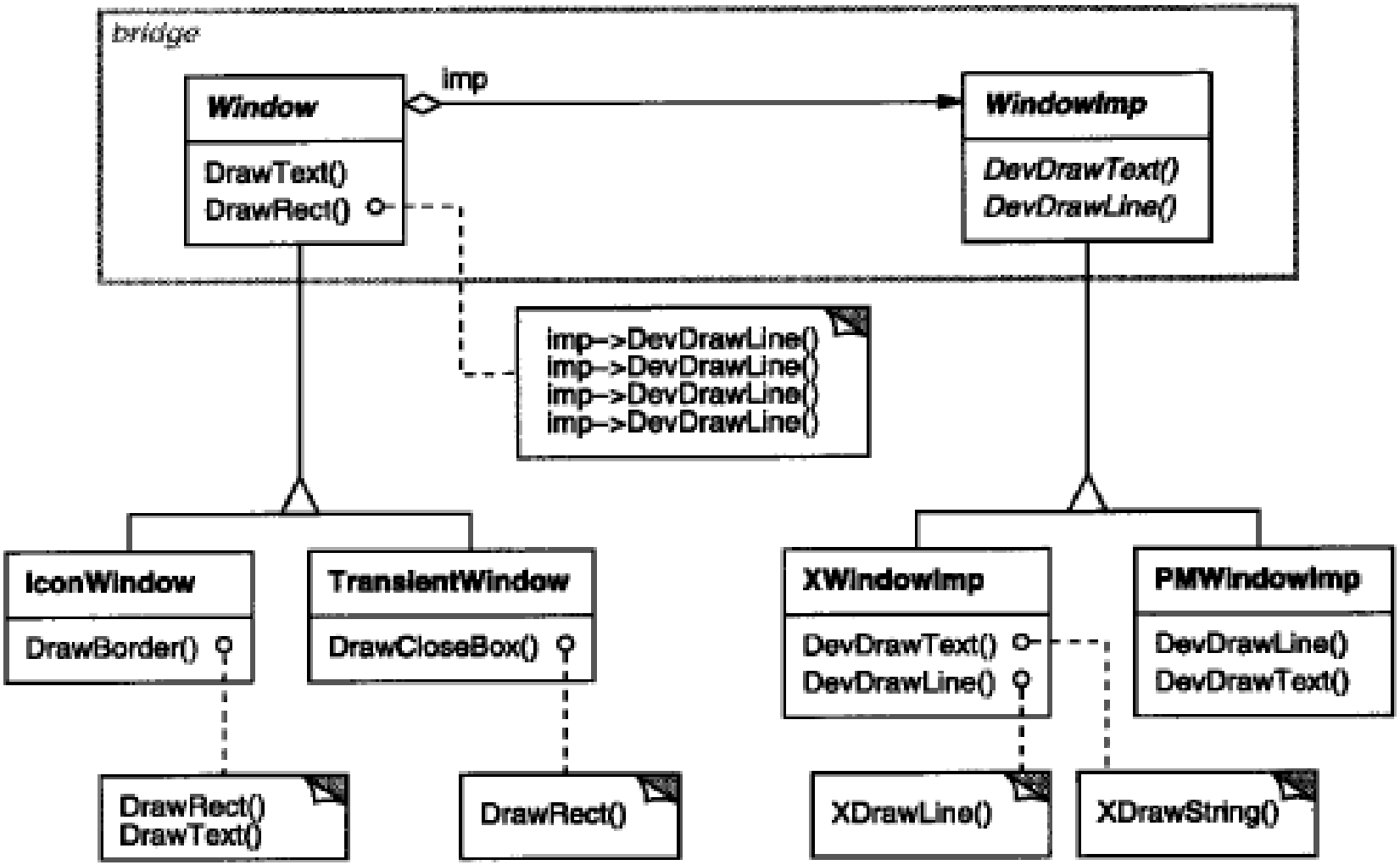
3. 动机

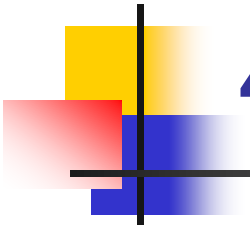
- 用户界面工具箱中，可移植的Window抽象部分的实现
  - 使用继承机制存在问题

1. 使用于新平台不方便
2. 使得客户代码与平台相关



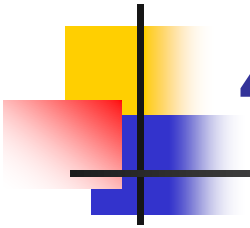
3. 动机


bridge
Window
DrawText()
DrawRect()
imp
WindowImp
DevDrawText()
DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
imp->DevDrawLine()
IconWindow
DrawBorder()
TransientWindow
DrawCloseBox()
XWindowImp
DevDrawText()
DevDrawLine()
PMWindowImp
DevDrawLine()
DevDrawText()
DrawRect()
DrawText()
DrawRect()
XDrawLine()
XDrawString()



4. 适用性

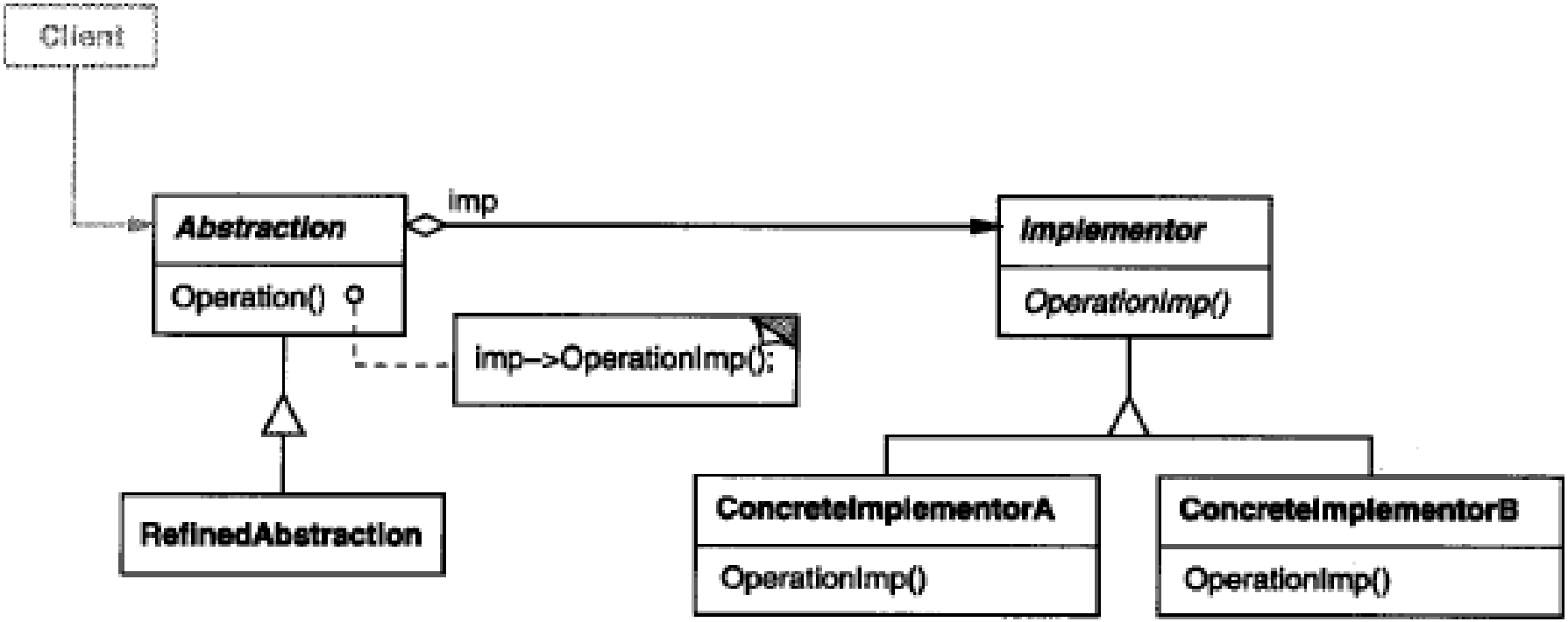
- 不希望抽象和其实现部分之间有一个固定的绑定关系
  - e.g. 希望动态切换实现部分
- 类的抽象和其实现都可以通过生成子类的方法进行扩充
  - 两部分可以独立扩充
- 对一个抽象的实现部分进行修改，客户代码不需要重新编译

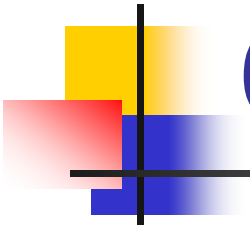


4. 适用性

- (c++)想对Client完全隐藏抽象的实现部分
- 避免过于复杂的类层次结构
- 多个抽象 VS 1个实现

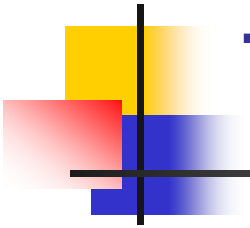
5. 结构


Client
Abstraction
Operation()
imp
Implementor
OperationImp()
imp->OperationImp();
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()



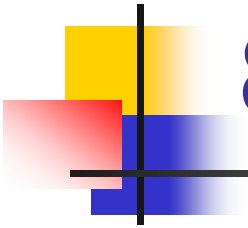
6. 参与者

- 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
  - Implementor接口仅提供基本操作，而Abstraction定义了基于这些基本操作的较高层次的操作
- ConcreteImplementor (XwindowImp, …)
  - 实现Implementor接口，定义其具体实现



7. 协作

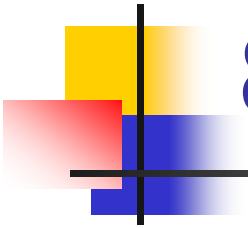
- Abstraction将Client的请求转发给它的Implementor对象



8. 效果

1) 分离接口及其实现部分

- 一个对象可以在运行时刻改变它的实现
- 改变一个实现类时，不需要重新编译Abstraction类和它的客户程序
  - 用途：保证一个类库的不同版本之间的二进制兼容性
- 有助于分层，从而产生更好的结构化系统
  - 系统高层部分仅需要知道Abstraction和Implementor即可



8. 效果

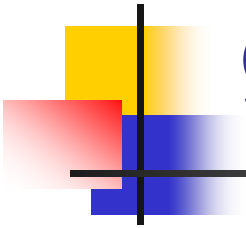
2) 提高可扩充性

独立对Abstraction和Implementor层次接口进行扩充

3) 实现细节对客户透明

- 对客户隐藏实现细节

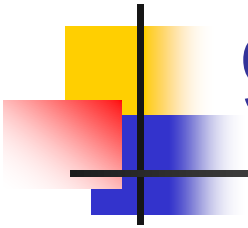
  e.g. 共享Implementor对象，相应的引用计数机制



9. 实现

1）仅有一个Implementor

- 此时没有必要创建一个抽象的Implementor类
- Bridge模式的退化



9. 实现

2）创建正确的Implementor对象

- 存在多个Implementor类时，How, When, Where 创建一个合适的Implementor类
  - 由Abstraction决定

    e.g. Collection类根据大小选择链表/hash表的实现
  - 首先选择一个缺省的实现，然后根据需要改变

    e.g. Collection类大小超过阀值时切换实现
  - 代理给另一个对象决定

    e.g. Window/WindowImp中引入factory对象

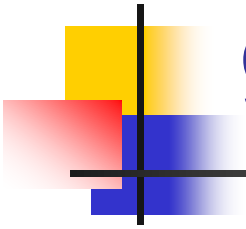
9. 实现

3）共享Implementor对象

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

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

    return *this;
}
```

Handle/Body方法在多个对象间共享实现

引用计数器



9. 实现

4）采用多重继承机制

- C++中可使用多重继承将抽象接口和实现部分结合
  - Public方式继承Abstraction
  - Private继承ConcreteImplementor
- 问题：
  - 依赖于静态继承，将实现部分与接口固定绑定
- 结论：

  不能用多重继承方法实现真正的Bridge模式

10. 代码示例

```
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
};
```

对WindowImp的引用

Window类：抽象窗口类

10. 代码示例

```
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();
};
```

WindowImp抽象类

10. 代码示例

- Window子类

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

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

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


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

10. 代码示例

- 具体WindowImp子类

```
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
};
```

```
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;
};
```

10. 代码示例

- 不同的子类实现WindowImp操作

```
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);
}
```

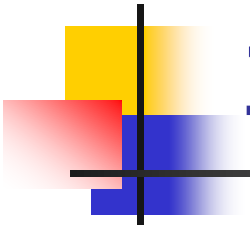
```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    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);
    }
}
```

10. 代码示例

- 一个Window对象如何获取正确的WindowImp对象

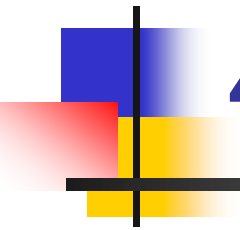
```
WindowImp* Window::GetWindowImp () {
    if (_imp == 0) {
        _imp = WindowSystemFactory::Instance()->MakeWindowImp();
    }
    return _imp;
}
```

抽象工厂， 单件

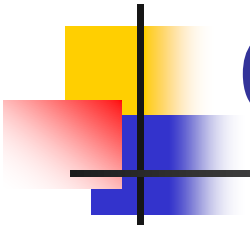


12. 相关模式

- Abstraction Factory可以用于Bridge模式
- Adapter与Bridge
  - Adapter往往是针对已有对象的
  - Bridge则是在系统开始时就被使用的



4.3 Composite 组合模式



Composite模式

1. 意图

- 将对象组合成树形结构以表示“部分-整体”的层次接口
- 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

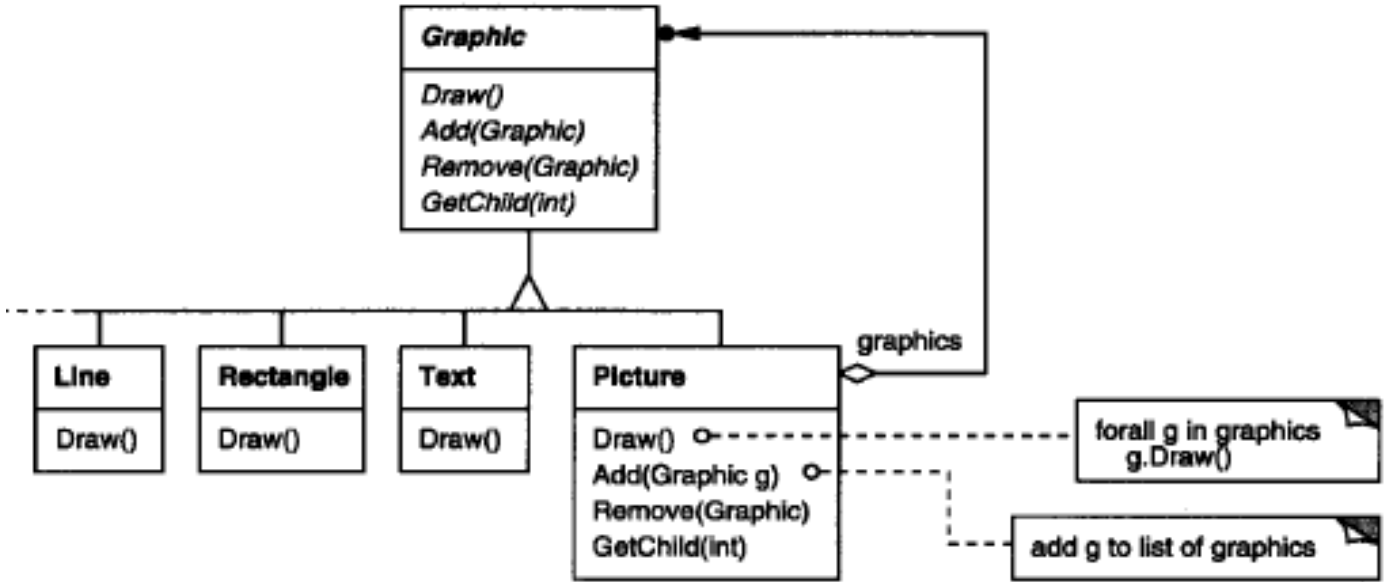
问题：

用户可以使用简单的组件创建复杂的图表，即用户可以组合多个简单组件以形成一个较大的组件。

解决方案1：定义基本的图元类，定义容器类

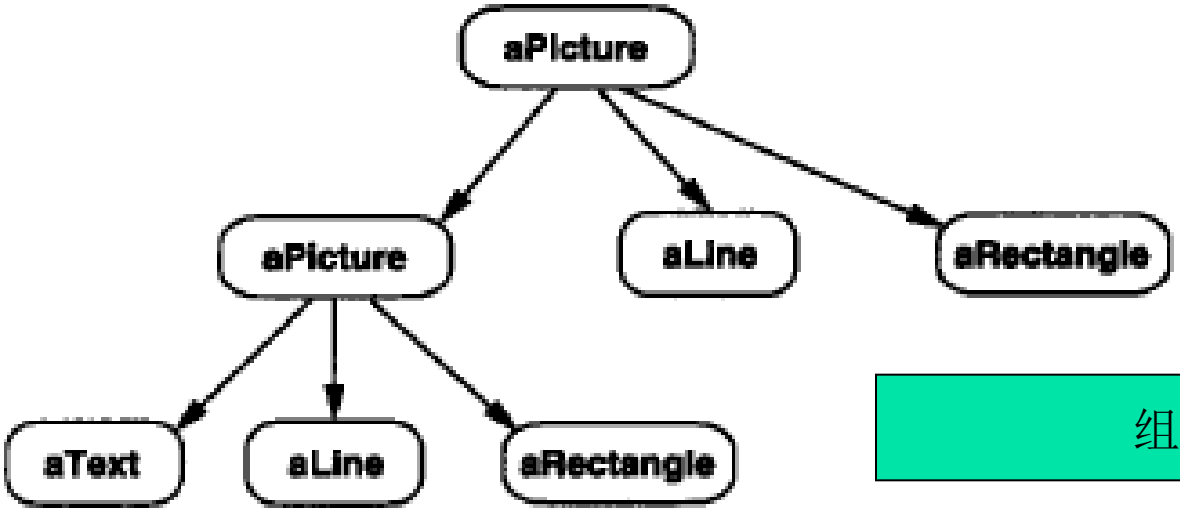
问题：使用这些类的代码必须区别对待图元对象和容器对象

解决方案2：Composite模式

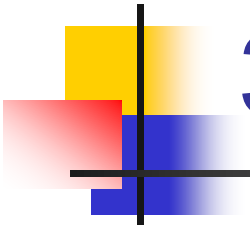


2. 动机

- Composite模式的关键
  - 存在一个抽象类，它即可以代表图元，又可以代表图元的容器。



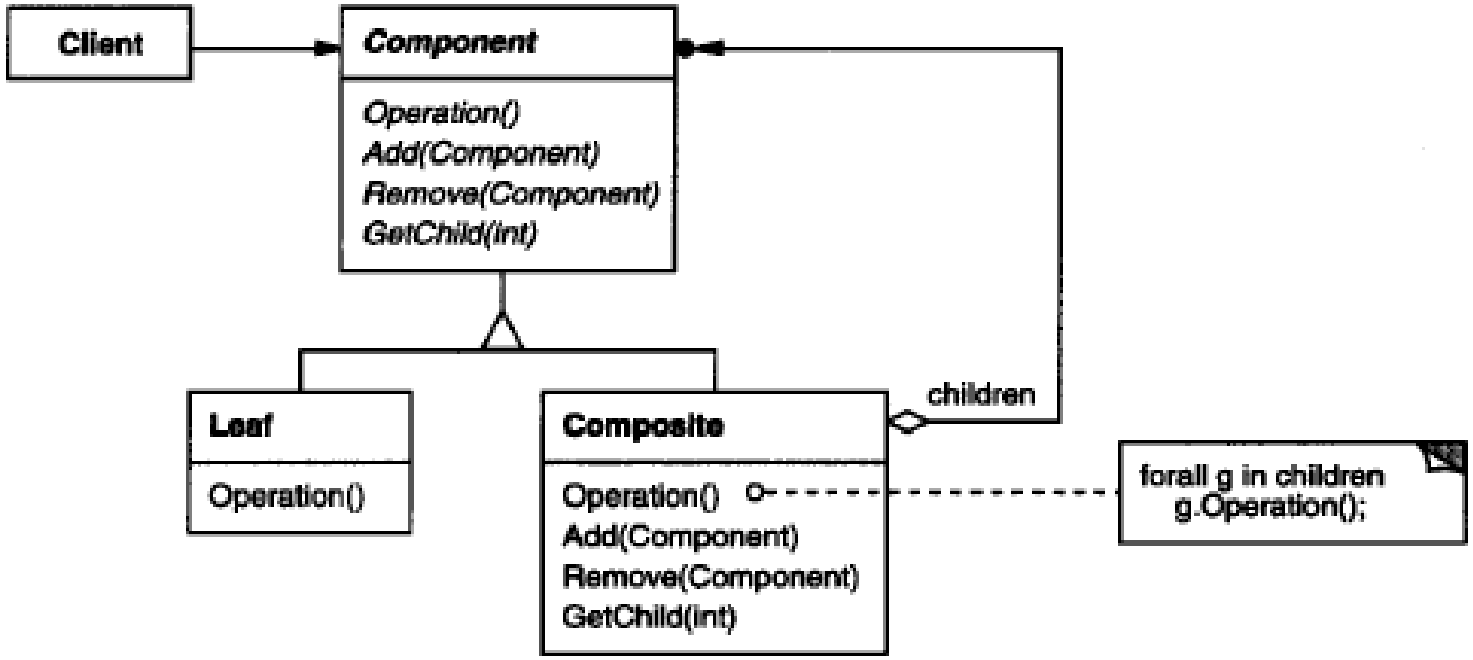
组合对象结构



3.适用性

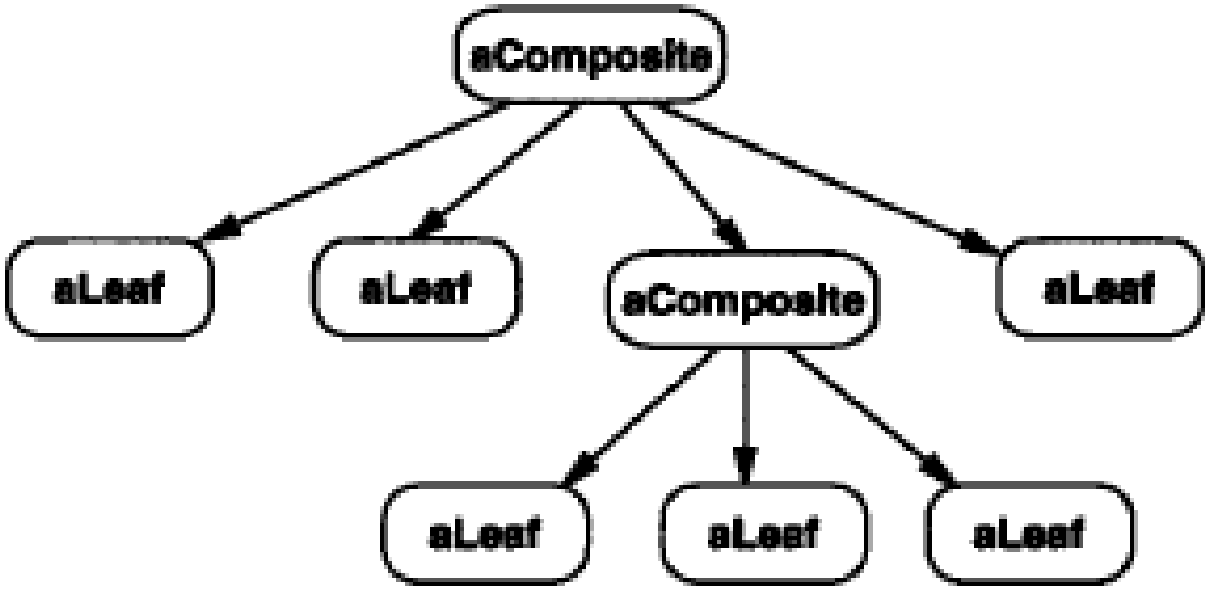
- 想表示对象的部分-整体层次结构
- 希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象

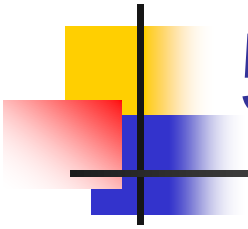
4. 结构


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
forall g in children
g.Operation();

4. 结构

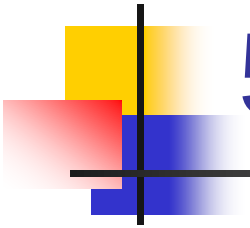
- 典型的Composite对象结构





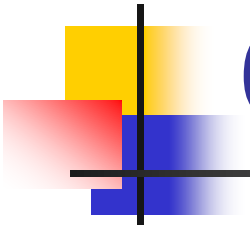
5. 参与者

- 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
    - 即可以设置 “父指针”



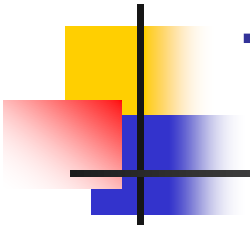
5. 参与者

- Leaf (Rectangle、Line、Text等)
  - 在组合中表示叶节点对象，叶节点没有子节点
    - e.g. 在组合中表示图元对象的行为
    - 即： 基本元素
- 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接口操作组合对象



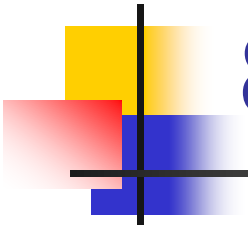
6. 协作

- 用户使用Component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，在转发请求前/后可能执行一些辅助操作



7. 效果

- 定义了包含基本对象和组合对象的类层次结构
  - 组合递归
- 简化客户代码
  - 对组合对象不需要特殊处理
- 使得更容易增加新类型的组件
  - 添加新的Component类不会影响Client
- 使你的设计更加一般化
  - 问题: 如果对组合有限制的话，需要额外的工作（Composite模式本身不支持这种限制）

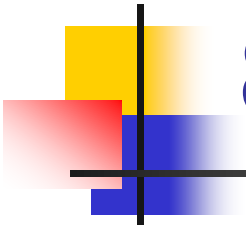


8. 实现

1) 显式的父部件引用： 父指针

- 优点：简化组合结构的遍历和管理
  - 简化结构的上移和组件的删除
  - 父部件引用可支持Chain of Responsibility模式
- 实现：在Component类中定义父部件引用
  - 不变式：

    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。
  - 仅当一个组件中增加或删除一个组件A时，才改变A的父指针



8. 实现

2) 共享组件

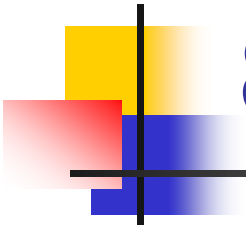
- 解决方法：在被共享的子部件中存储多个父指针
- 问题：如果一个请求需要向上传递，则会导致多义性
- 解决： Flyweight模式

8. 实现

3) 最大化Component接口

- 目标是一致性对待基本对象和组合对象，所以接口要覆盖基本对象和组合对象
- 即：最大化接口

- 问题：“一个类只能定义那些对它的子类有意义的操作” 与此原则矛盾
  - e.g. 许多Component的操作对Leaf没有意义
  - 可以将基本对象视为组合对象的特例（无子节点的对象）



8. 实现

4) 声明管理子部件的操作

- add/remove等操作放在component还是composite层次上
  - Component层次上，透明性好，但是存在安全性问题
    - e.g. 在Leaf上增加和删除对象
  - Composite层次上，安全性好，但损失了透明性

- Composite模式中，更强调透明性

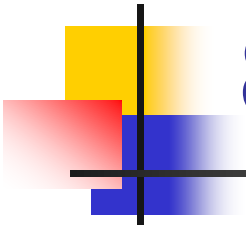
8. 实现

4) 声明管理子部件的操作

```
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 {
    // ...
};
```

```
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
}
```

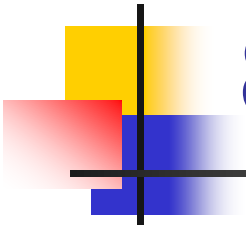
通过GetComposite操作判断是否为组合对象
只有组合对象才可以进行Add/Remove，从而保证安全性
此方法并不是完全透明的



8. 实现

5) Component是否应该实现一个Component列表

- 即：Component中有存放子指针的列表
- 此方式对叶节点会导致空间浪费
- 适用情况：
  - 当该接口中子类数目相对较少时



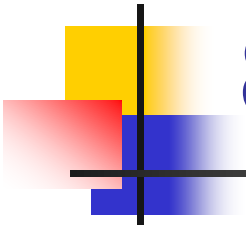
8. 实现

6) 子部件排序

- 需要考虑子节点顺序时，必须设计对子节点的访问和管理接口
- Iterator模式

7) 使用高速缓冲存储改善性能

- 需要对组合对象进行频繁遍历或查找时，可以在Composite对象中缓冲存储相关信息
- 此时，需要定义一个接口：
  - 子组件变化时，通知父组件其缓存信息无效



8. 实现

8) 应该由谁删除Component

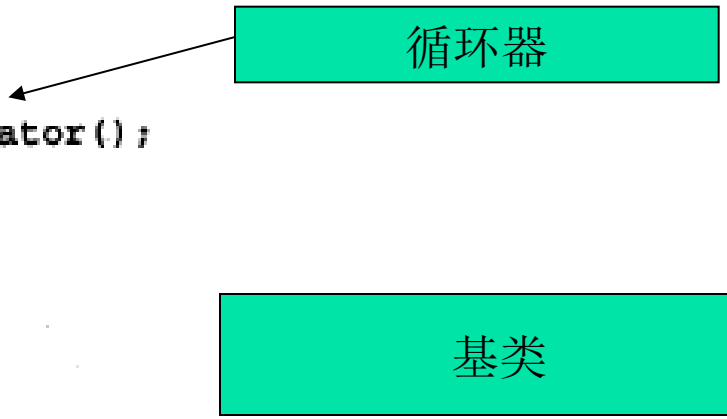
- 父节点删除时负责删除子节点
- 如果子节点用于共享，则不进行删除操作

9) 存储组件最好用哪一种数据结构

- 不同的Composite可以使用不同的数据结构
  - e.g. 列表、树、数组、hash表等

9. 代码示例

```
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;
};
```



9. 代码示例

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

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

Leaf

9. 代码示例

```
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;
};
```

Composite类

NetPrice的缺省实现

```
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;
}
```

9. 代码示例

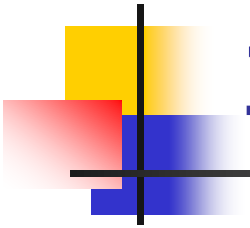
```
class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();
    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();
};
```

Composite子类

9. 代码示例

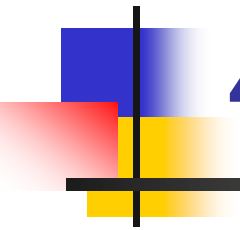
Client：

```
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;
```



11.相关模式

- 部件-父部件连接用于Responsibility of Chain模式
- Decorator模式通常与Composite模式一起使用
- Flyweight模式用于共享组件
  - 不能引用他们的父部件
- Itertor用于遍历Composite
- Visitor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装饰模式

Decorator模式

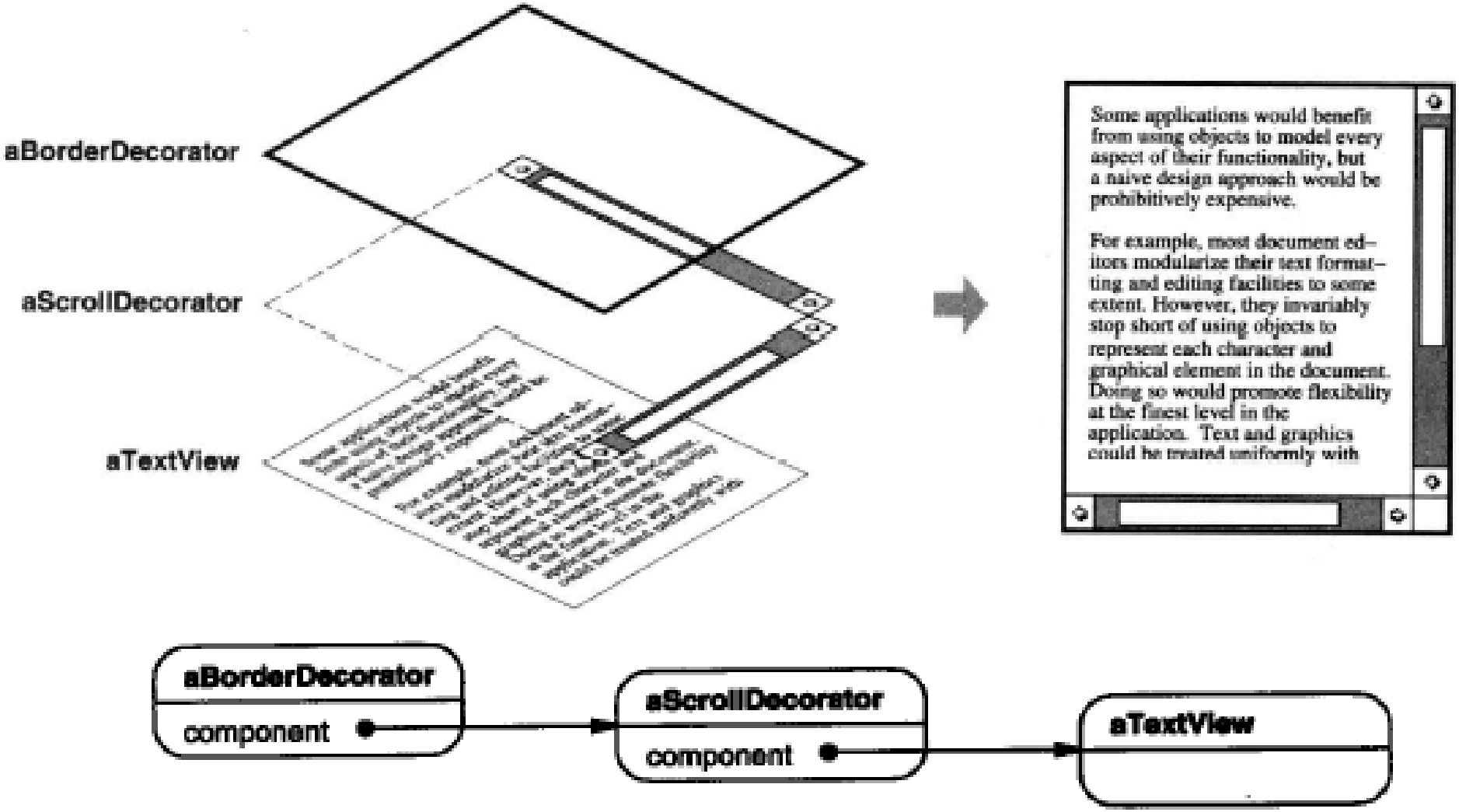
1. 意图

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

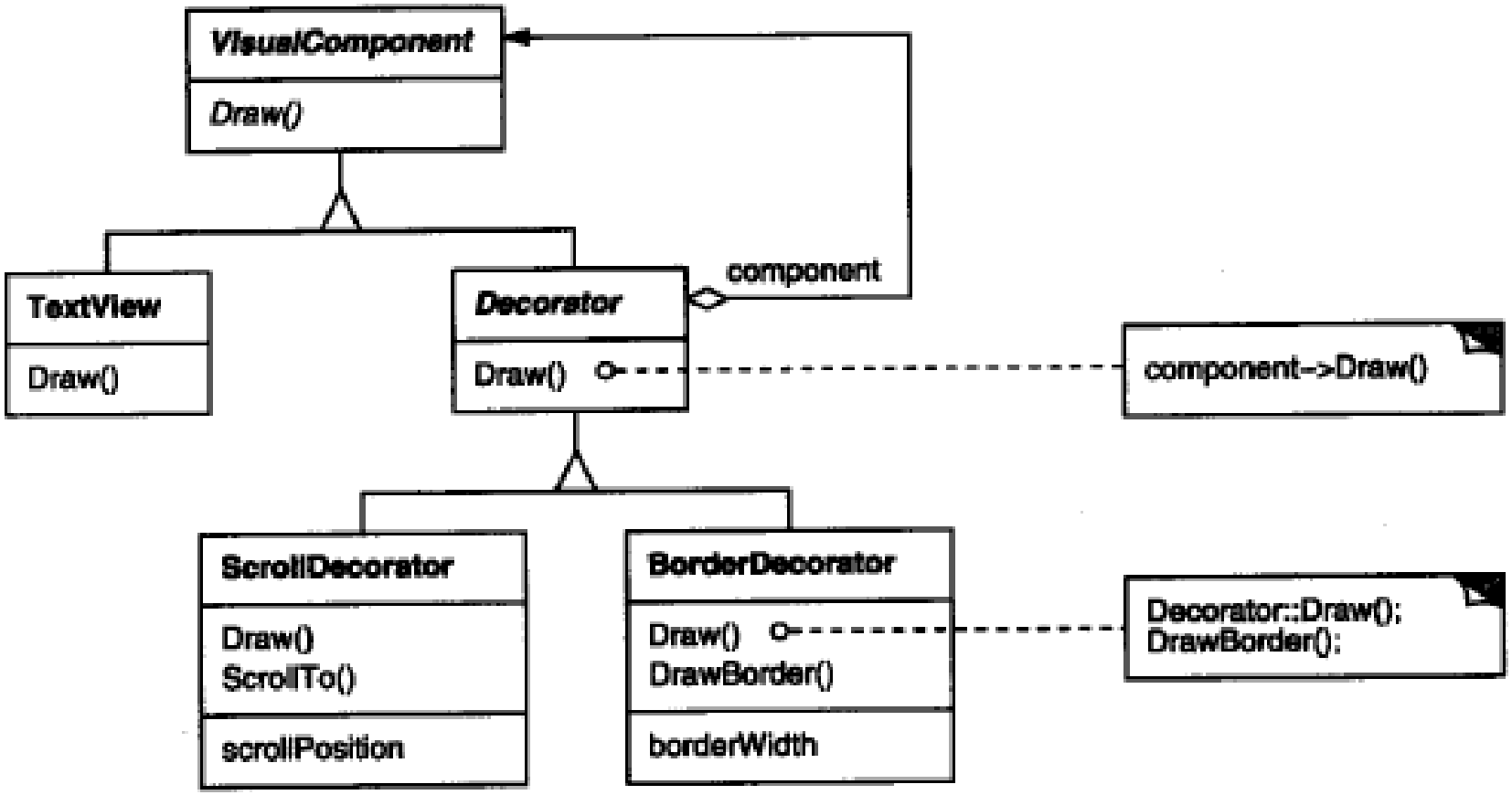
2. 别名

包装器 Wrapper

3. 动机


aBorderDecorator
aScrollDecorator
aTextView
Some applications would benefit from using objects to model every aspect of their functionality, but a naive design approach would be prohibitively expensive.
For example, most document editors modularize their text formatting and editing facilities to some extent. However, they invariably stop short of using objects to represent each character and graphical element in the document. Doing so would promote flexibility at the finest level in the application. Text and graphics could be treated uniformly with
aBorderDecorator
component
aScrollDecorator
component
aTextView

3. 动机

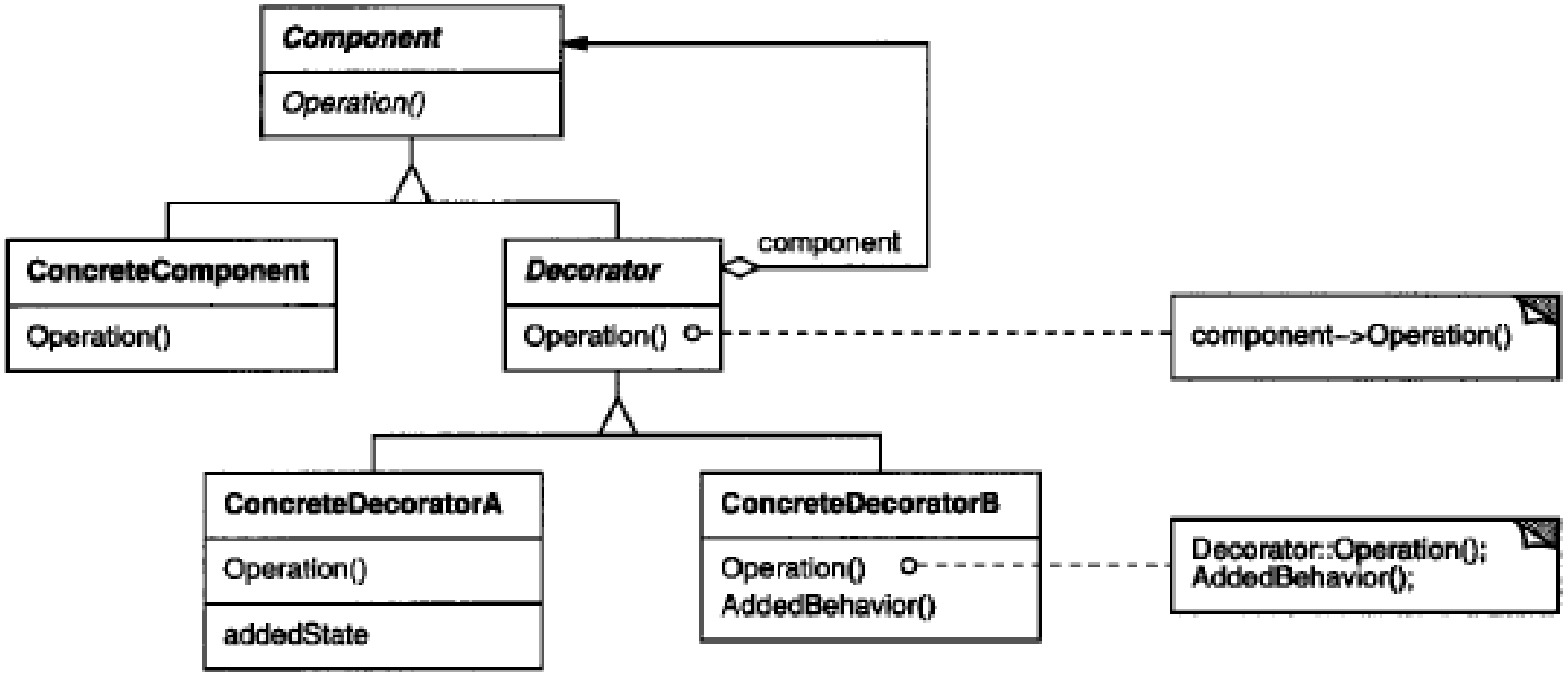

VisualComponent
Draw()
TextView
Draw()
Decorator
Draw()
component
component->Draw()
ScrollDecorator
Draw()
ScrollTo()
scrollPosition
BorderDecorator
Draw()
DrawBorder()
borderWidth
Decorator::Draw();
DrawBorder();

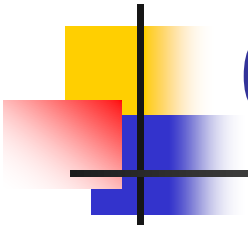
4.适用性

- 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。
- 处理那些可以撤销的职责
- 不能采用生成子类的方法进行扩充时
  - 子类爆炸问题

  或
  - 因为类定义被隐藏，或类定义不能用于生成子类

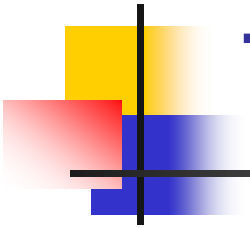
5. 结构


Component
Operation()
ConcreteComponent
Operation()
Decorator
Operation()
component
component->Operation()
ConcreteDecoratorA
Operation()
addedState
ConcreteDecoratorB
Operation()
AddedBehavior()
Decorator::Operation();
AddedBehavior();



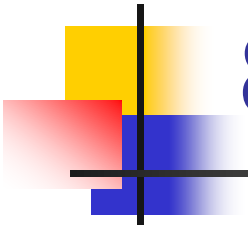
6. 参与者

- 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)
  - 具体对象
- Decorator
  - 维持一个指向Component对象的指针
  - 定义与Component接口一致的接口
- ConcreteDecorator (BorderDecorator, …)
  - 所添加的职责（具体的）



7. 协作

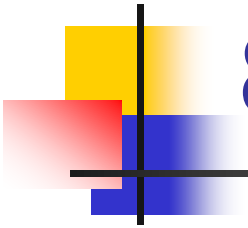
- Decorator将请求转发给它的Component对象
- 有可能在转发请求前/后执行一些附加的动作



8.效果

1) 比静态继承更灵活

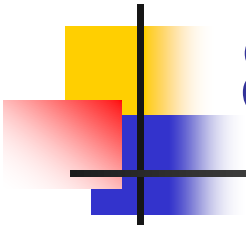
- 静态继承会引入许多类，增加系统复杂度
- Decorator可以在运行时增加和删除职责
- Decorator可以很容易重复添加特性
  - e.g. 在TextView上添加双边框



8.效果

2) 避免在层次结构高层的类有太多的特征

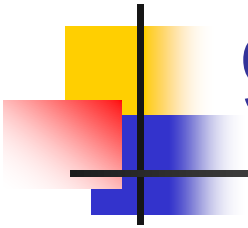
- 使用“即用即付”的方法来添加职责
- 在ConcreteComponent上不需要太多的特征



8.效果

3) Decorator与它的Component不一样

4)有许多小对象



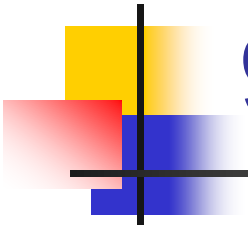
9. 实现

1) 接口的一致性

- Decorator对象和所装饰的Component的接口一致
- 则所有的ConcreteComponent类必须有一个公共的父类

2)省略抽象的Decorator类

- 仅需要一个抽象职责时，不需要定义抽象Decorator类



9. 实现

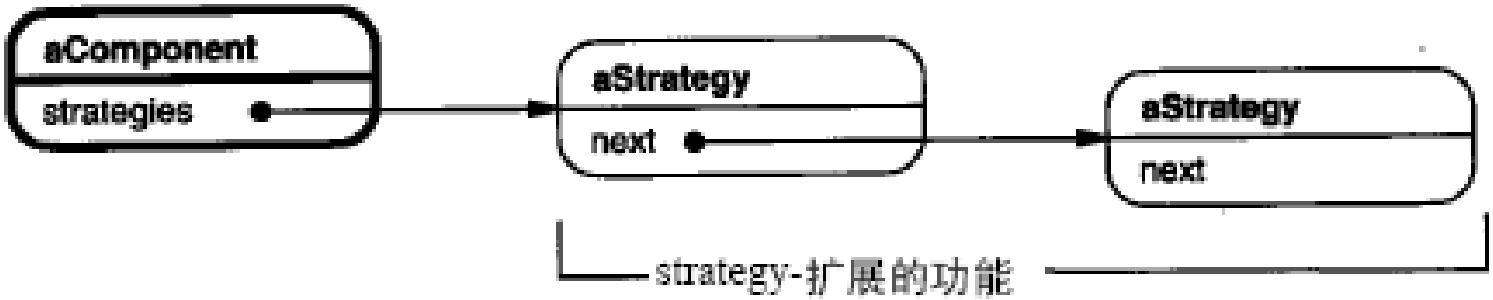
3) 保持Component类的简单性

- 组件和装饰有一个公共的Component父类，因此必须保持这个父类的简单性
- 即：该父类应集中于定义接口而不是存储数据

9. 实现

4) 改变对象外壳与改变对象内核

- Decorator模式：改变对象外壳
- Strategy模式：改变对象内核



10. 代码示例

Component类

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

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

抽象Decorator类

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


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

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

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

10. 代码示例

```
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);
}
```

具体Decorator类

10. 代码示例

- Client

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

window类提供的操作

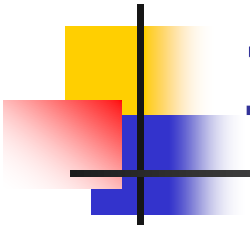
```
Window* window = new Window;
TextView* textView = new TextView;
```

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

在Window中放置一个textView对象

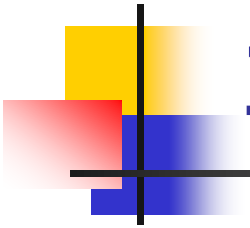
```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

在Window中放置一个有边框、滚动条的textView对象



12. 相关模式

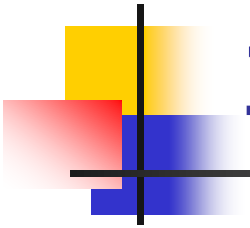
- Decorator 与 Adapter模式
  - Decorator仅改变对象的职责而不改变它的接口
  - Adapter给对象一个全新的接口
- Composite模式
  - 可以将Decorator视为一个退化的、仅有一个组件的Composite
    - 即：Decorator视为Composite的特殊情况
  - 但Decorator目的是为对象添加职责，而不是对象聚集



12. 相关模式

- Decorator与Strategy
  - Decorator用于改变对象的外表
  - Strategy模式用于改变对象的内核

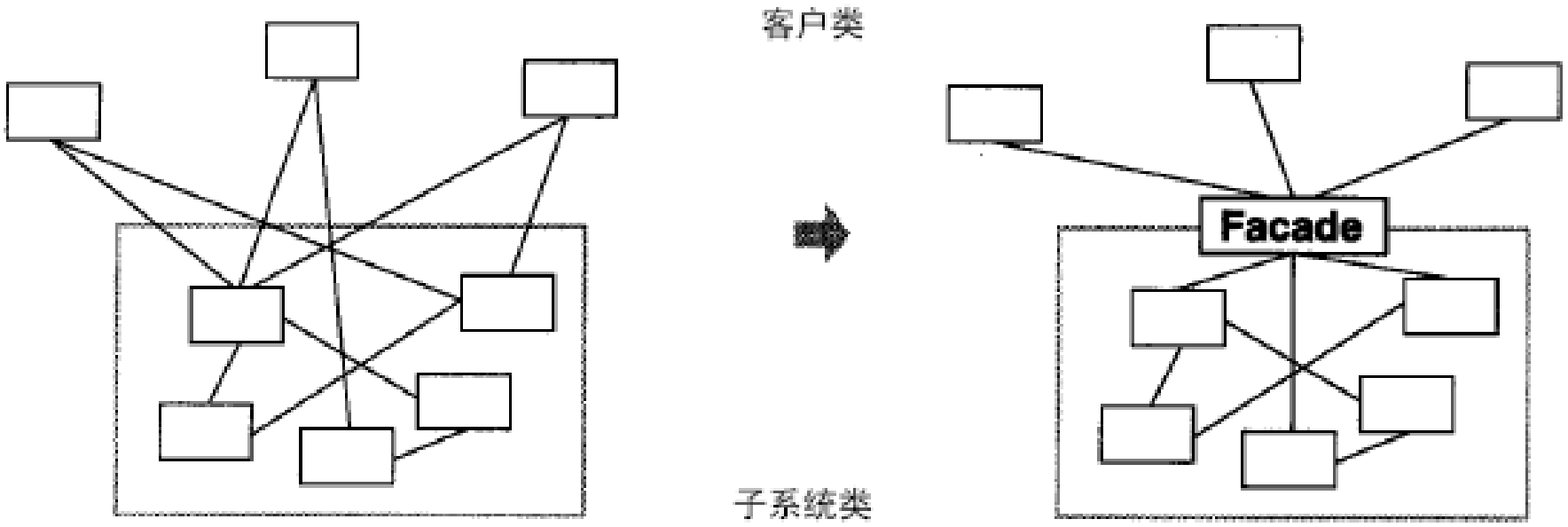
4.5 Facade 外观模式



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

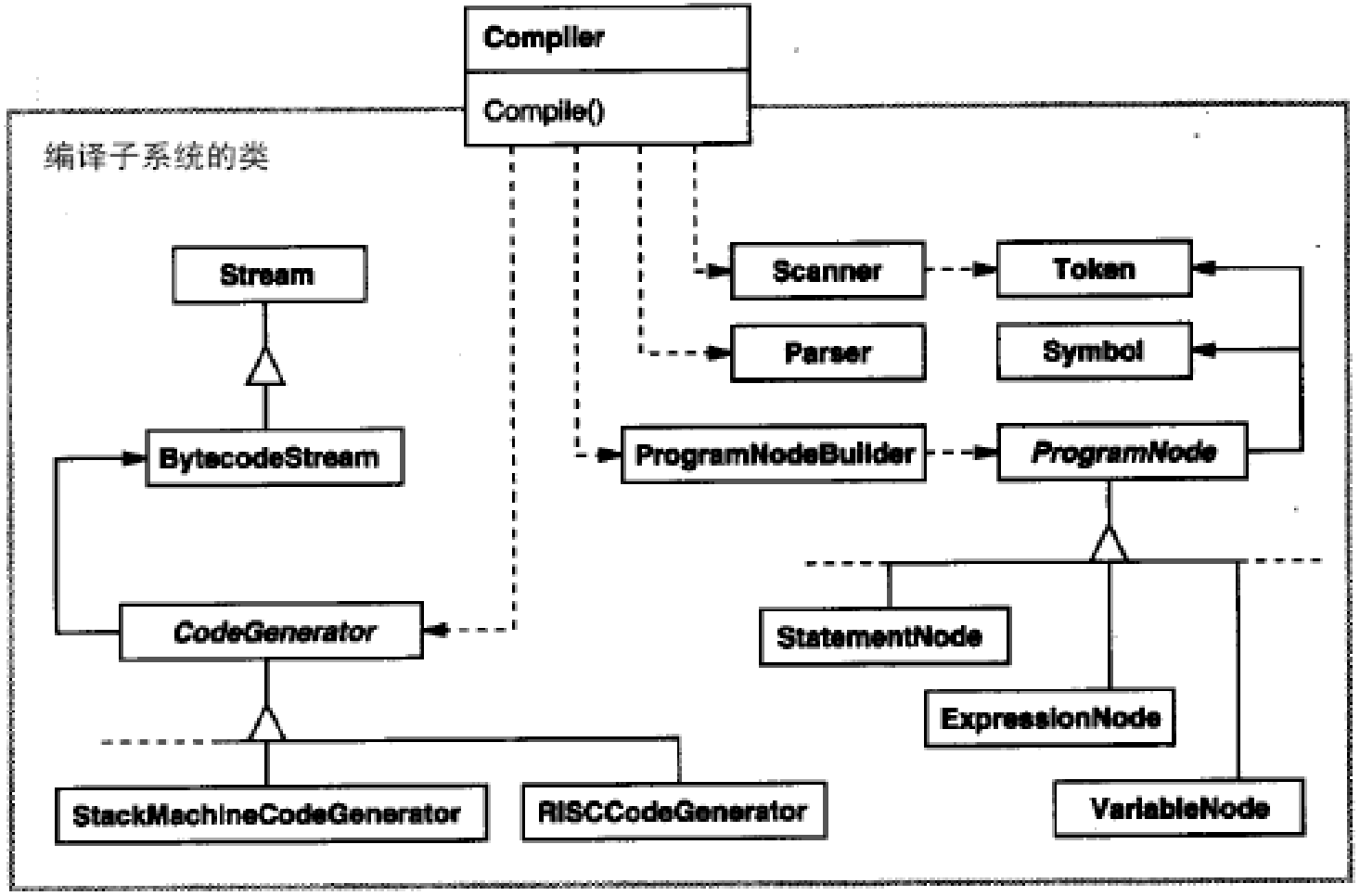
2. 动机



目标：使子系统间的通信和相互依赖关系达到最小
解决途径：引入一个外观（facade）对象，为子系统中提供一个单一而简单的界面

2. 动机

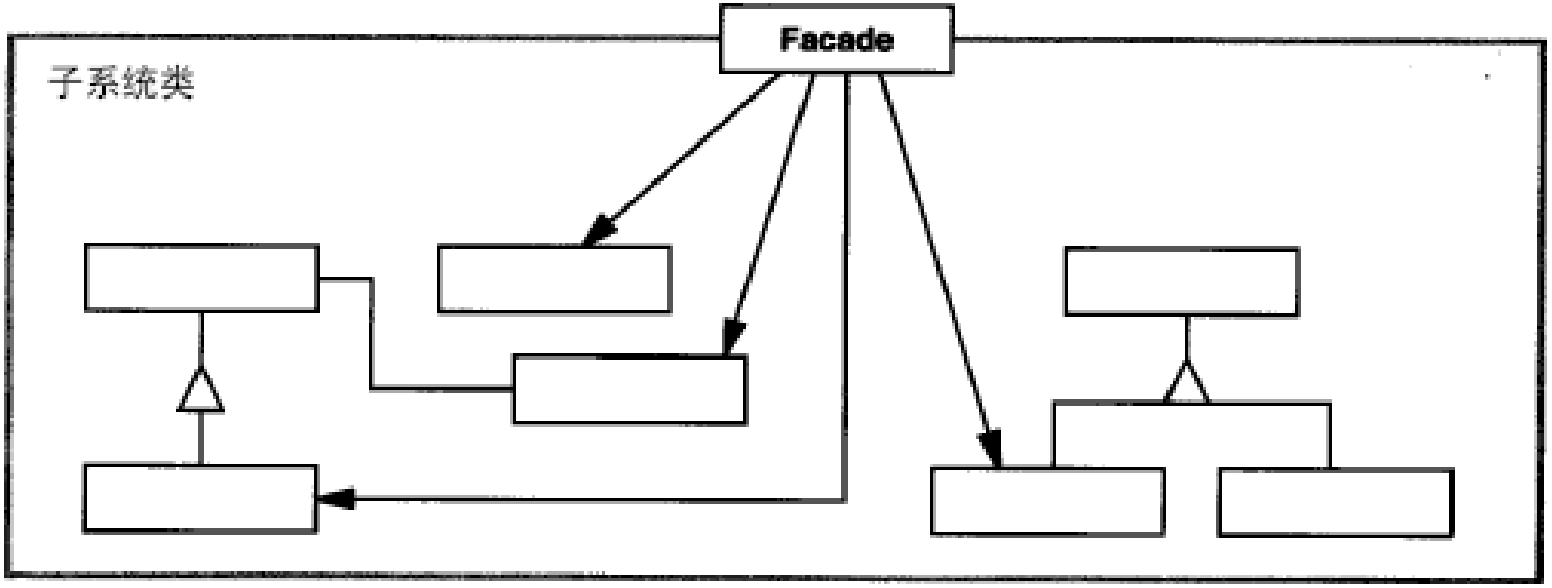
通过一个外观类来对内部子系统接口进行包装

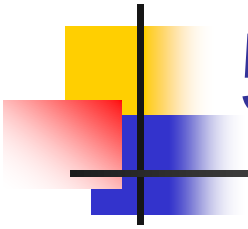


3. 适用性

- 需要为一个复杂子系统提供一个简单接口时
  - 大部分用户往往不想了解子系统内部复杂接口
- 客户程序与抽象类的实现之间存在很大依赖性
  - 引入facade将这个子系统与客户及其他子系统分离，提高子系统的独立性和可移植性
- 需要构造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
  - 简化了子系统间的依赖关系

4. 结构





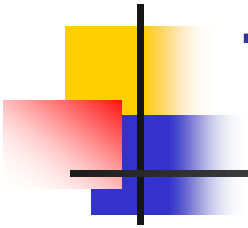
5.参与者

- Facade (Compile)：负责接口转换
  - 知道哪些子系统类负责处理请求
  - 间客户的请求代理给适当的子系统对象
- Subsytem classes (scanner、parser…)
  - 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

- 处理Client请求
  - Client发送请求给Facade
  - Facade将这些消息转发给适当的子系统对象
- 使用Facade的客户程序不需要直接访问子系统对象

由Facade进行隔断(Client 与 子系统对象)



7. 效果

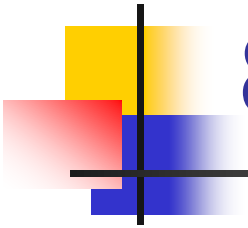
1) 对客户屏蔽子系统组件

2)实现了子系统和客户之间的松耦合关系

- 相对于引入了一个新的层次

3)如果应用需要，它并不限制它们使用子系统类

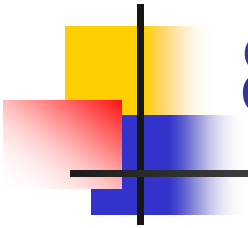
- Client还是可以去直接访问子系统类



8.实现

1) 降低客户-子系统之间的耦合度

- 方法1：
  - 使用抽象facade类，用facade的具体子类对应于不同的子系统实现
  - 这样可以替换子系统实现
- 方法2：
  - 用不同的子系统对象配置Facade对象



8.实现

2) 公共子系统类于私有子系统类

- 将子系统视为一个“类”，考虑其公共和私有接口
  - 子系统的公共接口是所有Client程序可以访问的类；
  - 子系统的私有接口仅用于对子系统进行扩充
- Facade类是公共接口的一部分，但不是唯一的部分

9.代码示例

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

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


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

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

子系统类Scanner

子系统类Parser

9.代码示例

使用Builder模式建立语法分析树：

Director：Parser

Builder: ProgramNodeBuilder

```
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;
};
```

9.代码示例

- 语法分析树： Composite模式

```
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();
};
```

9.代码示例

- 对语法分析树遍历产生代码：Visitor模式

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

CodeGenerator类具有子类：
StackMachineCodeGenerator
RISCCodeGenerator

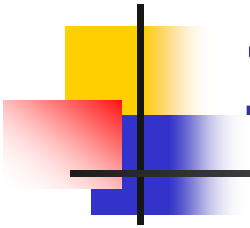
```
void ExpressionNode::Traverse (CodeGenerator& cg) {
    cg.Visit(this);

    ListIterator<ProgramNode*> i(_children);

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

9.代码示例

- Compiler类: facade

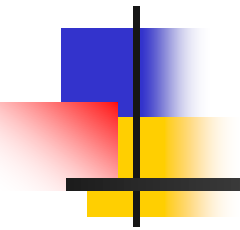
```
class Compiler {
public:
    Compiler();

    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);
}
```

这里固定了使用的CodeGenerator.
可以修改为在Compiler构造函数中参数指定

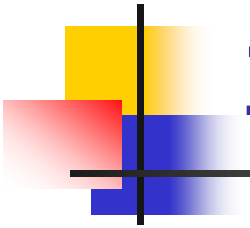


11. 相关模式

- Abstract Factory模式与Facade模式
  - 两者一起作用以提供一个接口
- Mediator模式与Facade模式
  - 相似：抽象了一些已有类的功能
  - 区别：
    - Mediator对Agent间的交互进行抽象
      - Agent间的交互 → Agent与Mediator的交互
    - Facade对子系统对象的接口进行抽象
      - 不定义新功能，子系统也不知道facade的存在
- Singleton模式
  - 仅需要一个Facade对象时



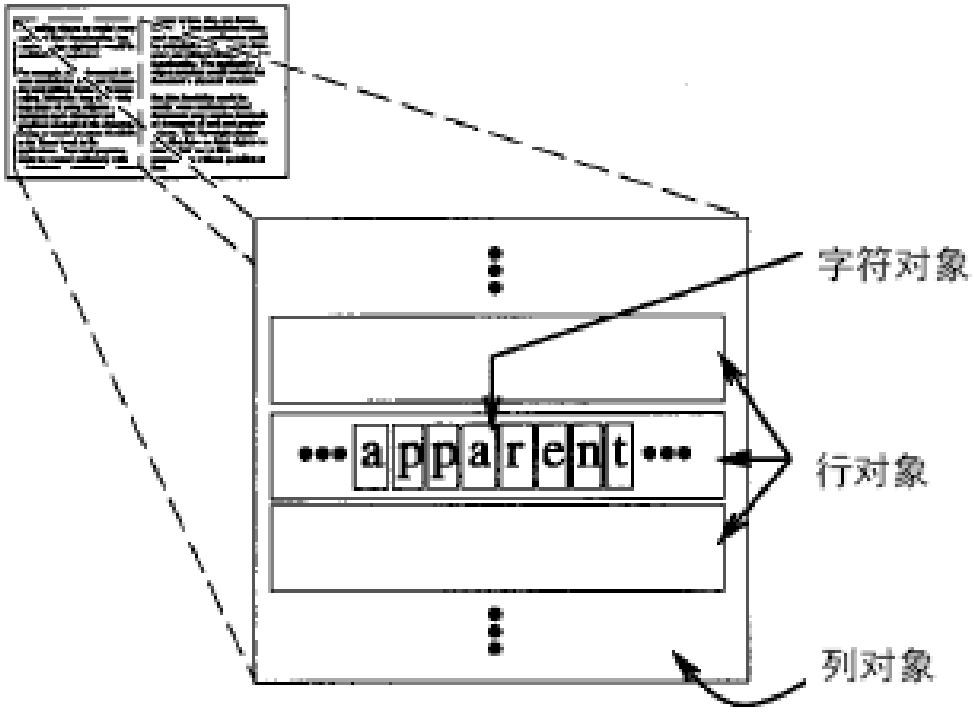
4.6 Flyweight模式
享元



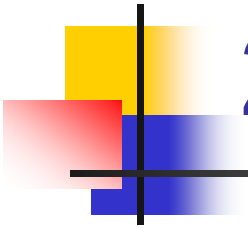
1. 意图

- 运用共享技术有效地支持大量细粒度的对象

2. 动机



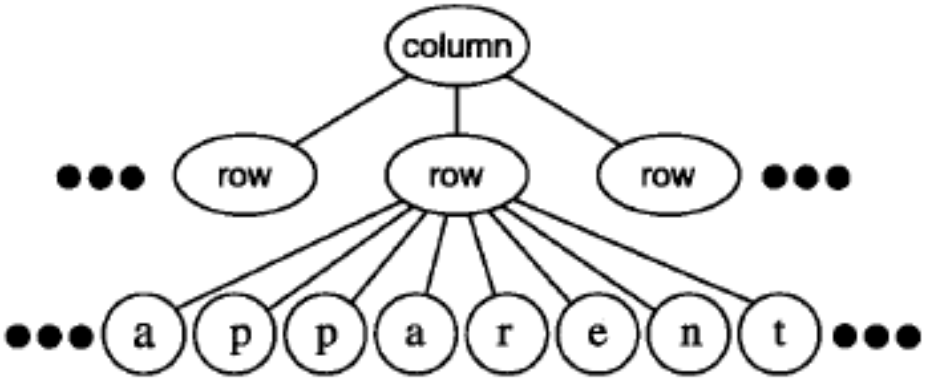
使用对象来表示字符使得设计简单，但是代价太大

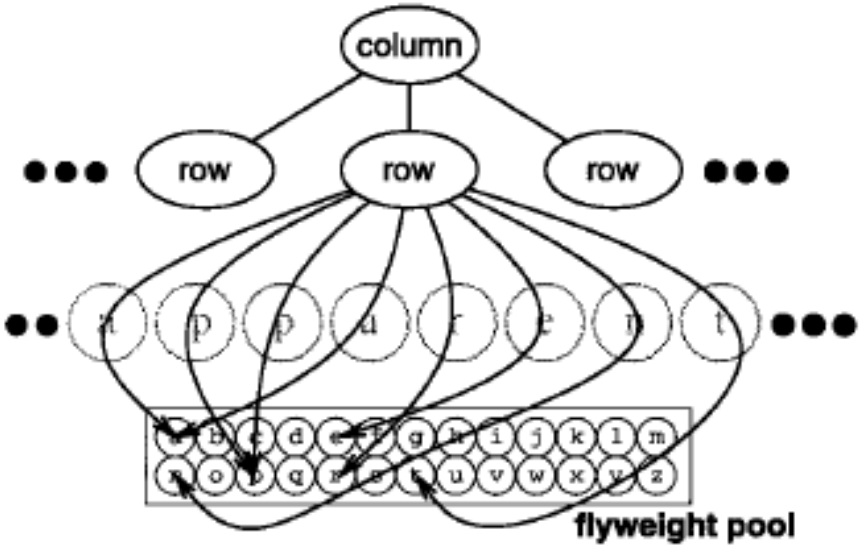


2. 动机

- Flyweight：共享对象
  - 同时在多个Context中使用，每个Context中都被视为一个独立的对象
  - 内部状态与外部状态
    - 内部状态独立于Context，可以共享
    - 外部状态依赖于Context，不能共享
      - Client需要在必要时将外部状态传递给Flyweight

2. 动机

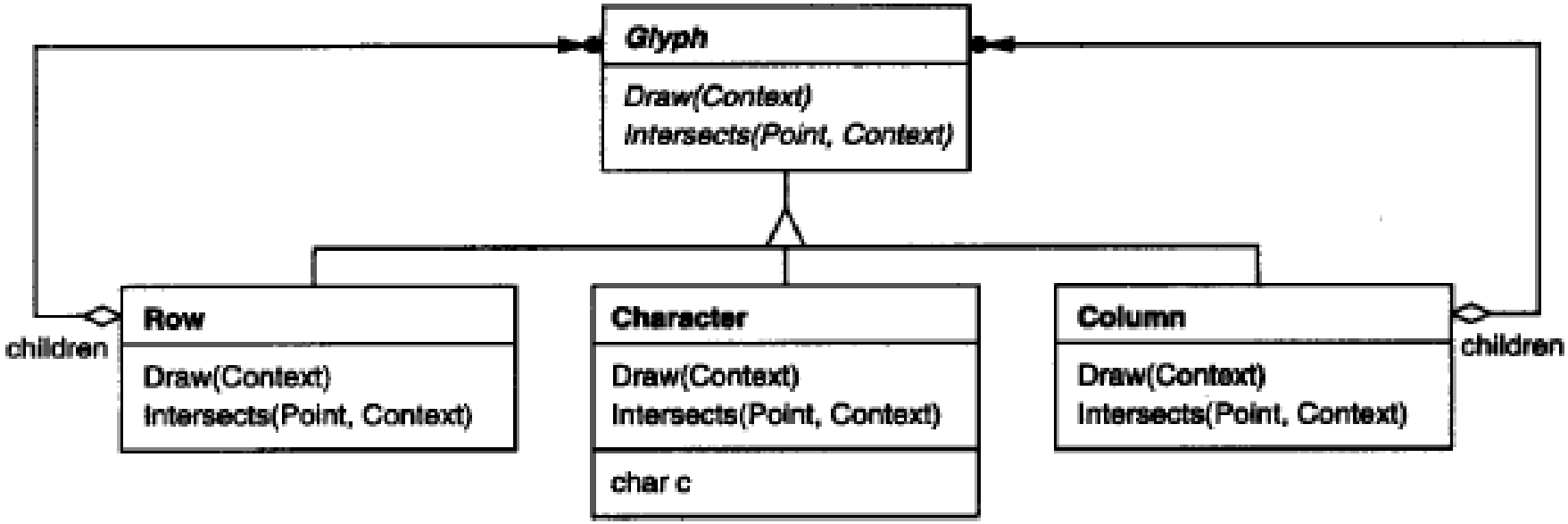




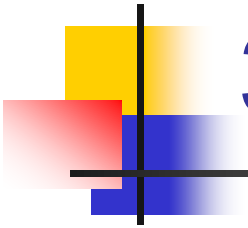
表示字母“a”的flyweight只存储相应的字符代码（内部状态）；

用户提供Context相关信息，如字符的位置或字体（外部状态），根据此信息flyweight绘制自己。

2. 动机



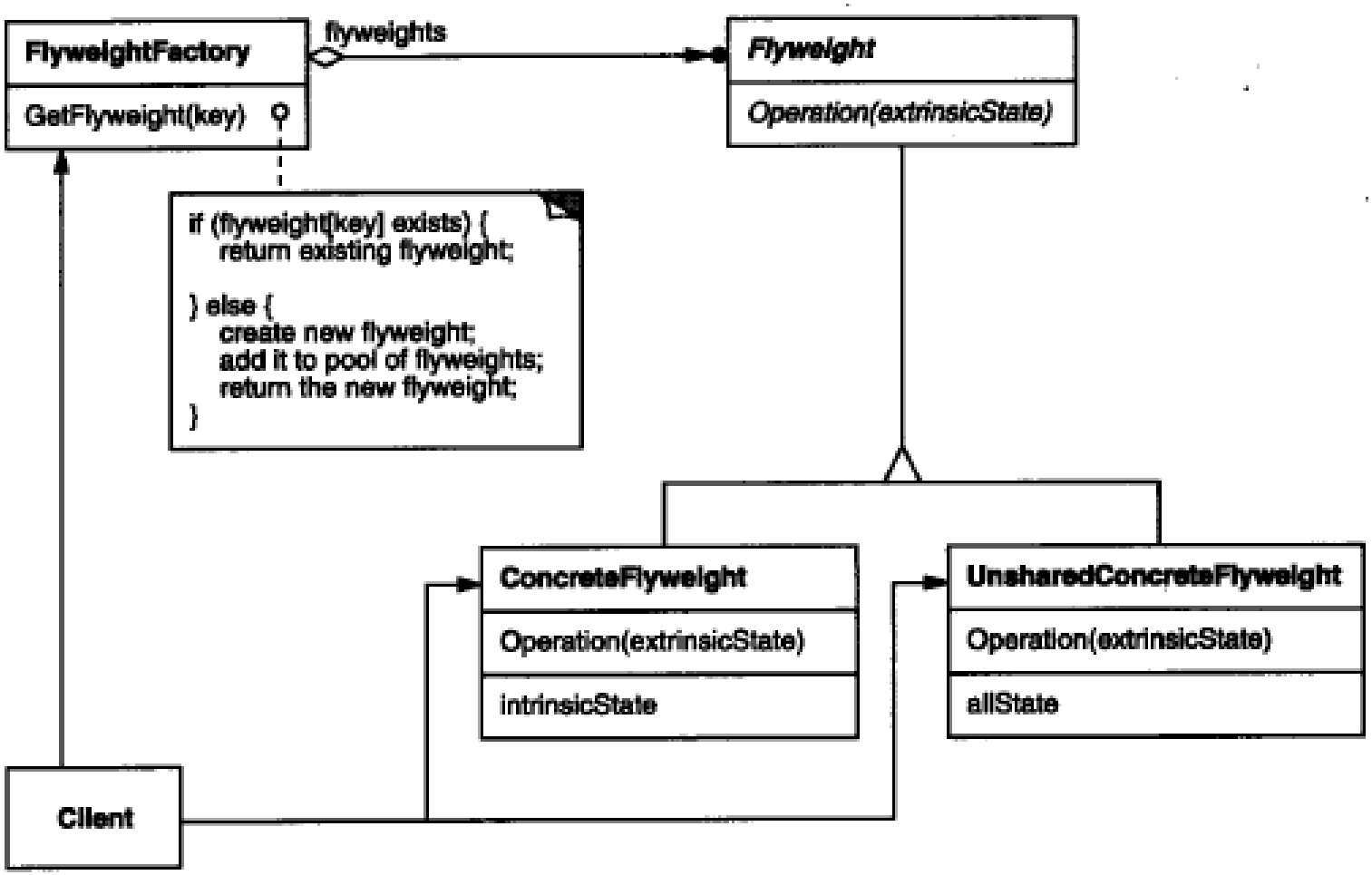
进行操作时，传入上下文Context



3. 适用性

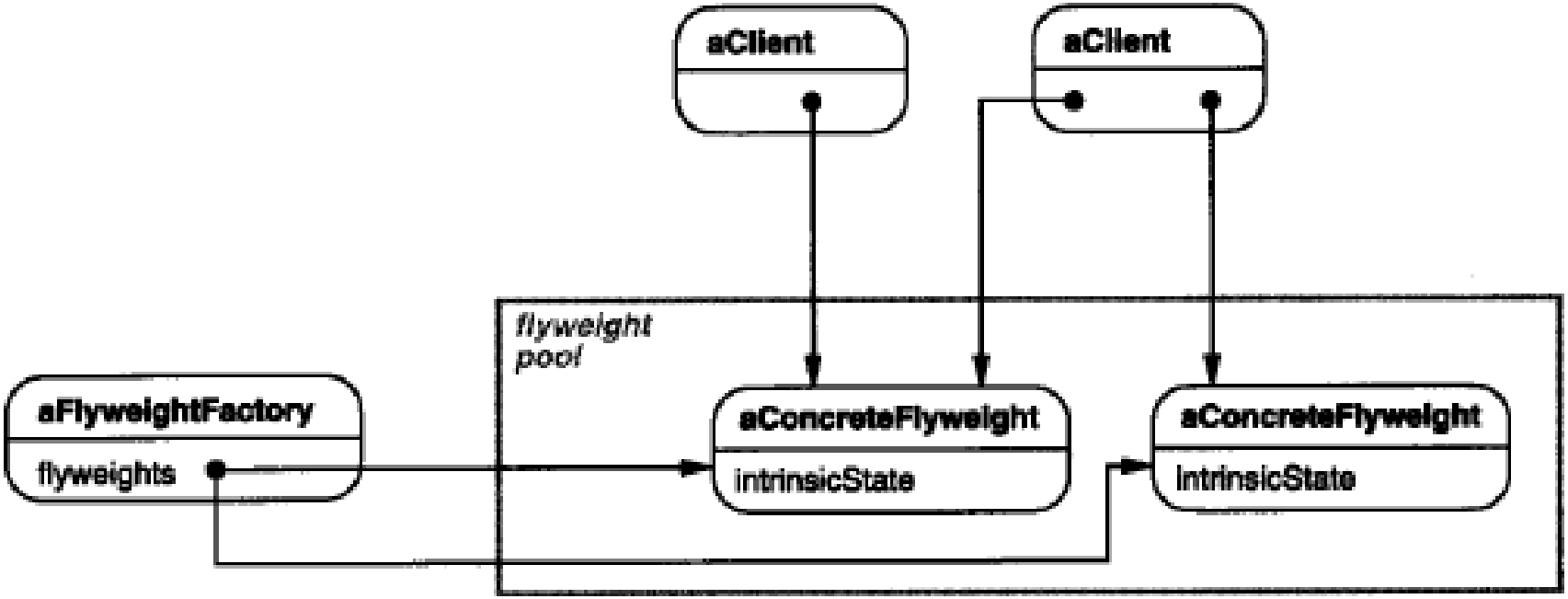
- 以下情况都成立时使用Flyweight模式
  - 一个应用程序使用了大量的对象
  - 完全由于使用大量的对象，造成很大的存储开销
  - 对象的大多数状态都可变为外部状态
  - 如果删除对象的外部状态，则可以用相对较少的共享对象取代多组对象
  - 应用程序不依赖对象标识

4. 结构


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

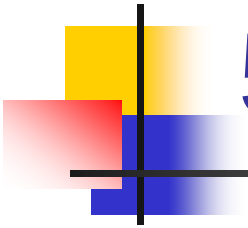
4. 结构

- 对象图


aClient
aClient
flyweight
pool
aFlyweightFactory
flyweights
aConcreteFlyweight
intrinsicState
aConcreteFlyweight
intrinsicState

5. 参与者

- Flyweight (Glyph)
  - 描述一个接口
  - 通过此接口flyweight可以接受并作用于外部状态
- ConcreteFlyweight (Charactor)
  - 实现Flyweight接口
  - 为内部状态增加存储空间
  - 注： ConcreteFlyweight对象中存储的状态必须是内部的



5. 参与者

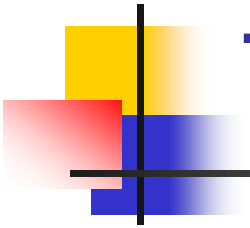
- UnsharedConcreteFlyweight
  (Row,Column)
  - Flyweight接口使共享成为可能，但是并不强制共享
  - UnsharedConcreteFlyweight对象通常将ConcreteFlyweight对象作为子节点

5. 参与者

- FlyweightFactory
  - 创建并管理flyweight对象
  - 确保合理地共享flyweight对象
    - 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
- Client
  - 维持一个对flyweight的引用
  - 计算或存储一个（多个）flyweight对象的外部状态

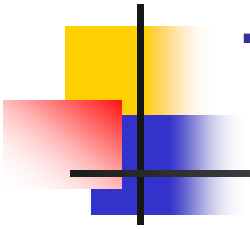
6. 协作

- 状态划分
  - flyweight执行时所需的状态必定是内部或外部的
  - 内部状态存储在ConcreteFlyweight对象之中
  - 外部状态由Client对象存储或计算
    - 当用户调用Flyweight对象的操作时，传递外部状态
- 用户不能直接对ConcreteFlyweight类进行实例化
  - 只能从FlyweightFactory对象得到ConcreteFlyweight对象



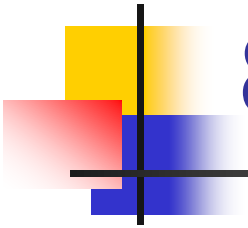
7. 效果

- 运算开销与存储节省
  - 使用Flyweight模式会产生额外运算开销
    e.g. 传输、查找 和/或 计算外部状态
  - 使用Flyweight模式会带来空间上的节省
- 存储节约取决于：
  - 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
- 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



7. 效果

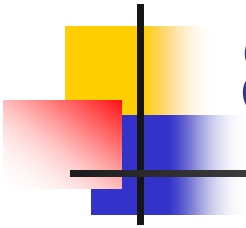
- Flyweight模式经常同Composite模式结合使用
  - 表示一个层次式结构：共享叶节点的图
  - 共享的结果：
    - 叶节点不能存储指向父节点的指针
    - 父节点指针视为Flyweight对象的外部状态
  - 对该层次结构中对象间相互通讯方式产生很大影响



8. 实现

1) 删除外部状态

- 使用Flyweight模式需要：
  - 识别外部状态并将它从共享对象中删除
- 理想状态：
  - 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



8. 实现

2) 管理共享对象

- 由于对象共享，所以不能直接进行实例化。
- FlyweightFactory对象使用关联存储帮助用户查找需要的Flyweight对象
- 方式1：
  - 某种形式的引用计数和垃圾回收
- 方式2：
  - Flyweight对象永久保存
  - Flyweight的数目固定并且很小的时候

9.代码示例

```
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();
};
```

Glyph类

9.代码示例

- Character类

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

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

9.代码示例

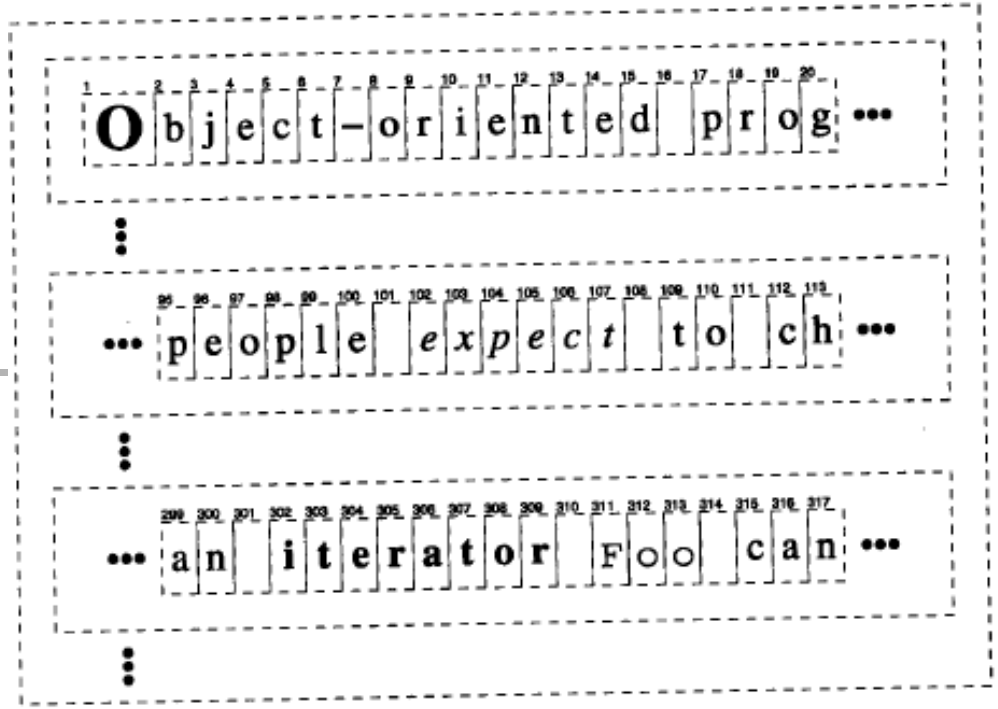
维持Glyph和字体间的映射关系

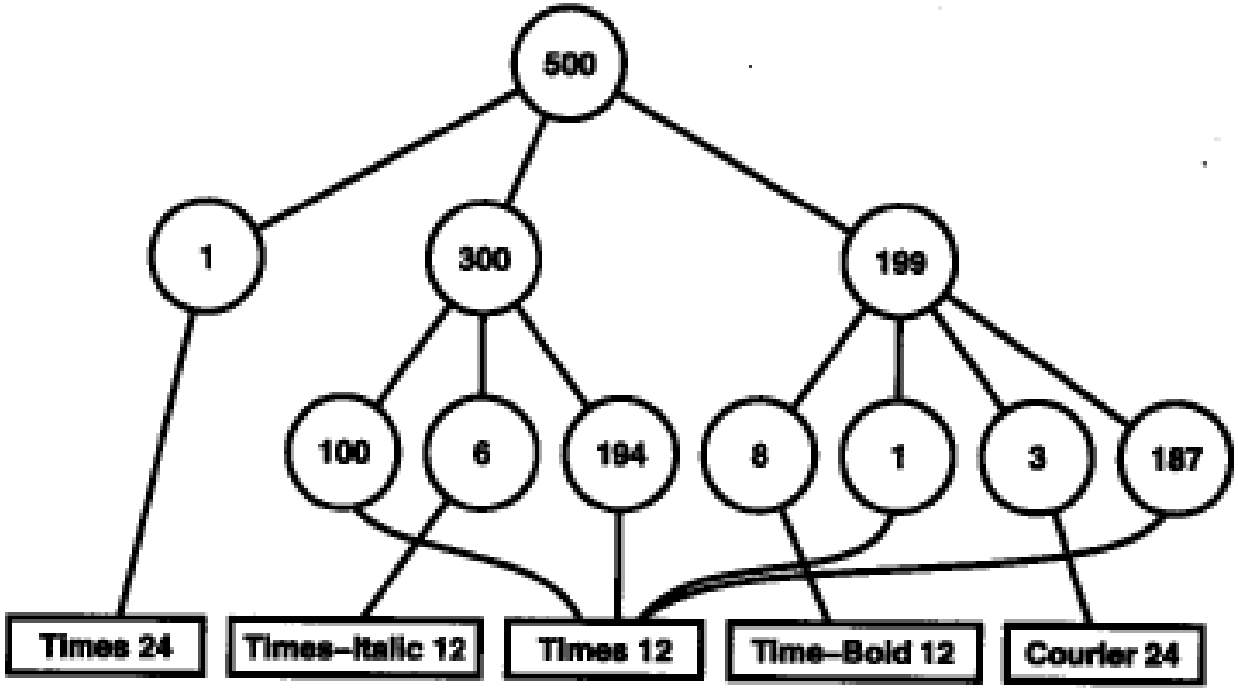
- GlyphContext ： 外部状态

```
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;
};
```

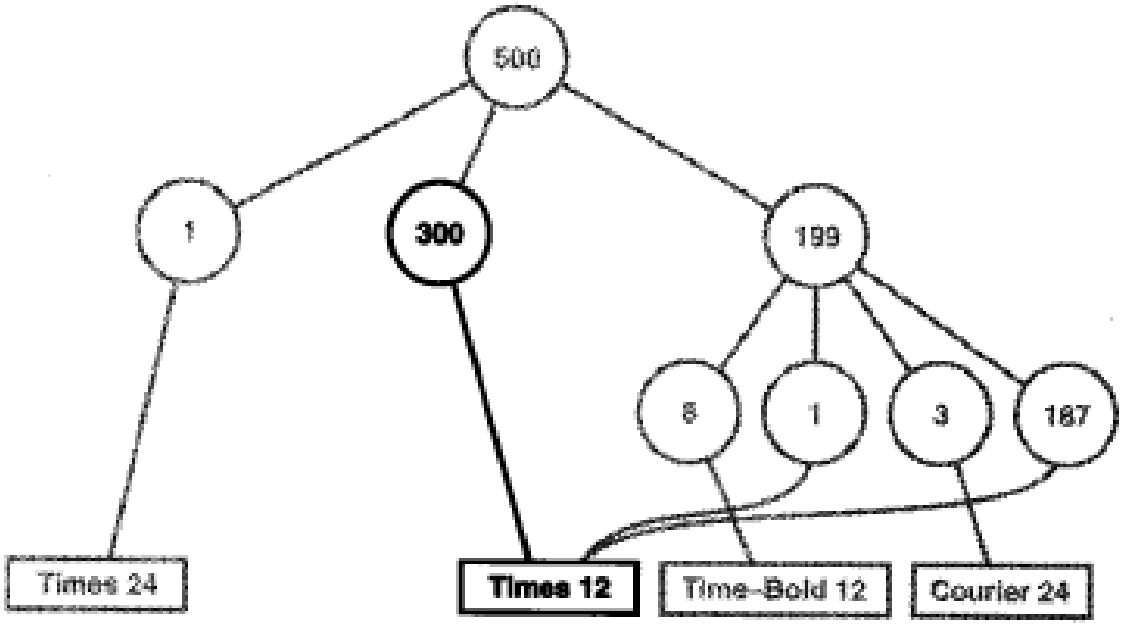
字体信息的BTree结构

9.代码示例


Object-oriented prog
people expect to ch
an iterator Foo can


500
1
300
199
100
6
194
8
1
3
187
Times 24
Times–Italic 12
Times 12
Time–Bold 12
Courier 24

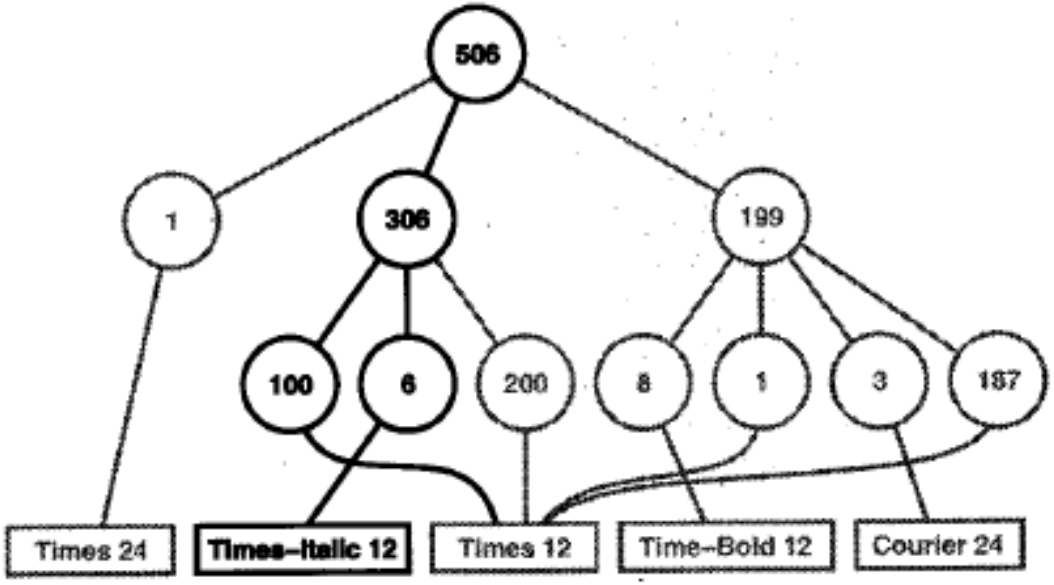
9.代码示例



修改“expect”的字体

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

gc.SetFont(times12, 6);
```

9.代码示例



在”except”前插入一个单词“Don’t”
字体12-point Times Italic

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

9.代码示例

GlyphFactory对象

```
const int NCHARCODES = 128;

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

9.代码示例

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

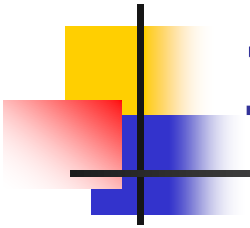
初始化_character数组

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

    return _character[c];
}
```

从_character数组中查找

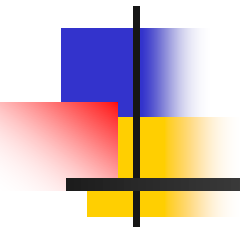
```
Row* GlyphFactory::CreateRow () {
    return new Row;
}

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

直接生成新对象

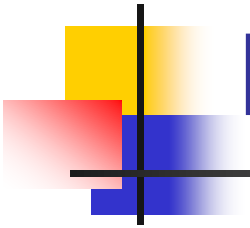


11.相关模式

- Flyweight与Composite模式
  - 用共享叶节点的有向无环图来实现一个逻辑上的层次结构
- Flyweight可用于实现State和Strategy对象



4.7 Proxy模式
代理



Proxy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

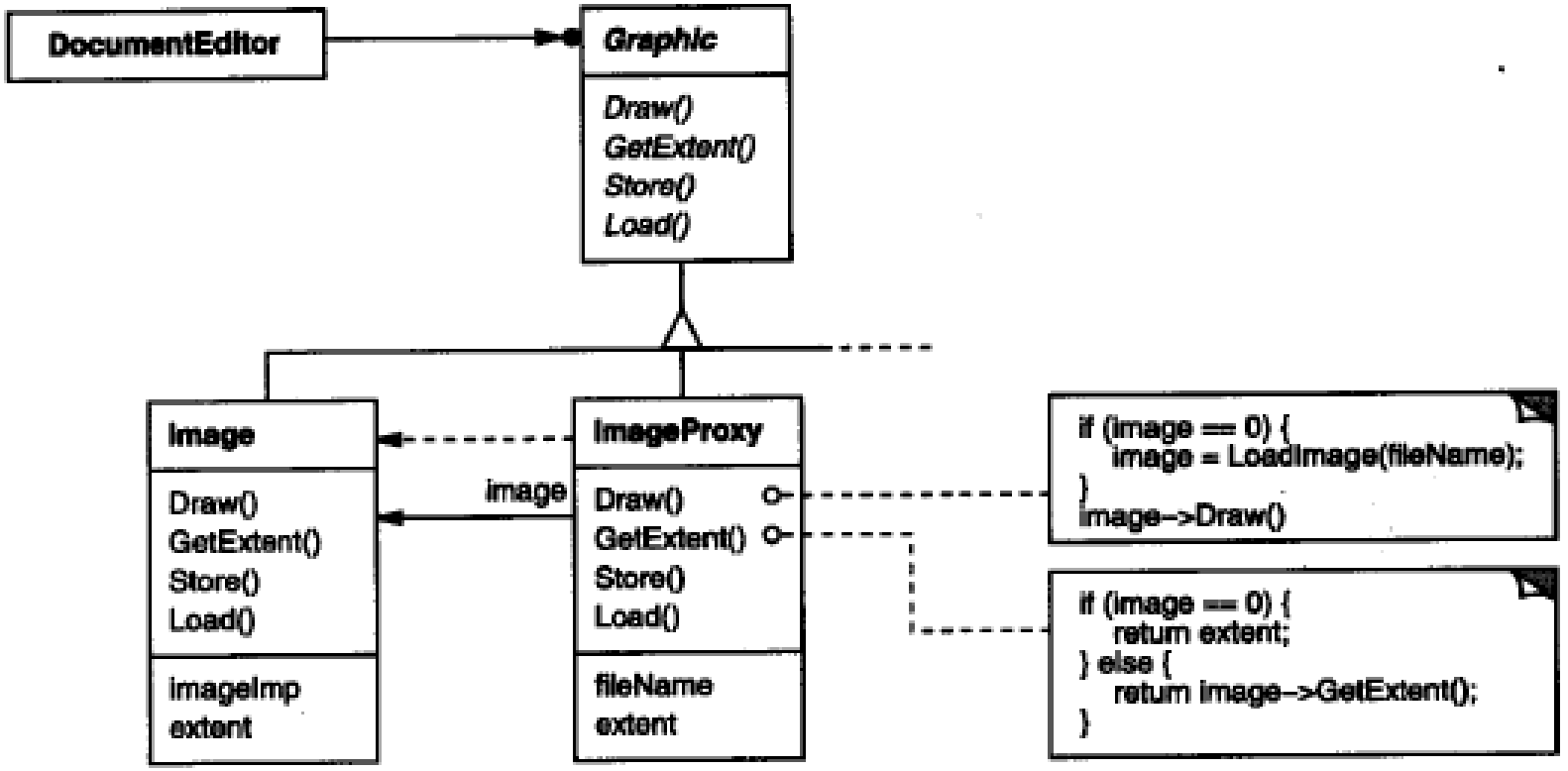
   Surrogate

3. 动机

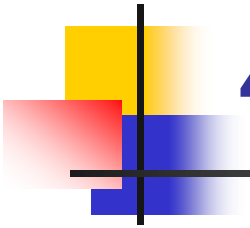
- 文档编辑器中
  - 使用“占位符”对象处理图像对象

- 文档编辑器激活图像代理的Draw操作时
  - Proxy对象才创建真正的图像对象
  - 然后将随后的请求转发给图像对象

3. 动机

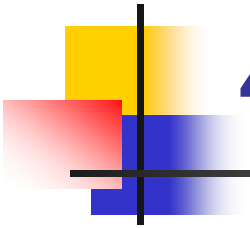


Proxy中存储了图像的尺寸（extend）



4.适用性

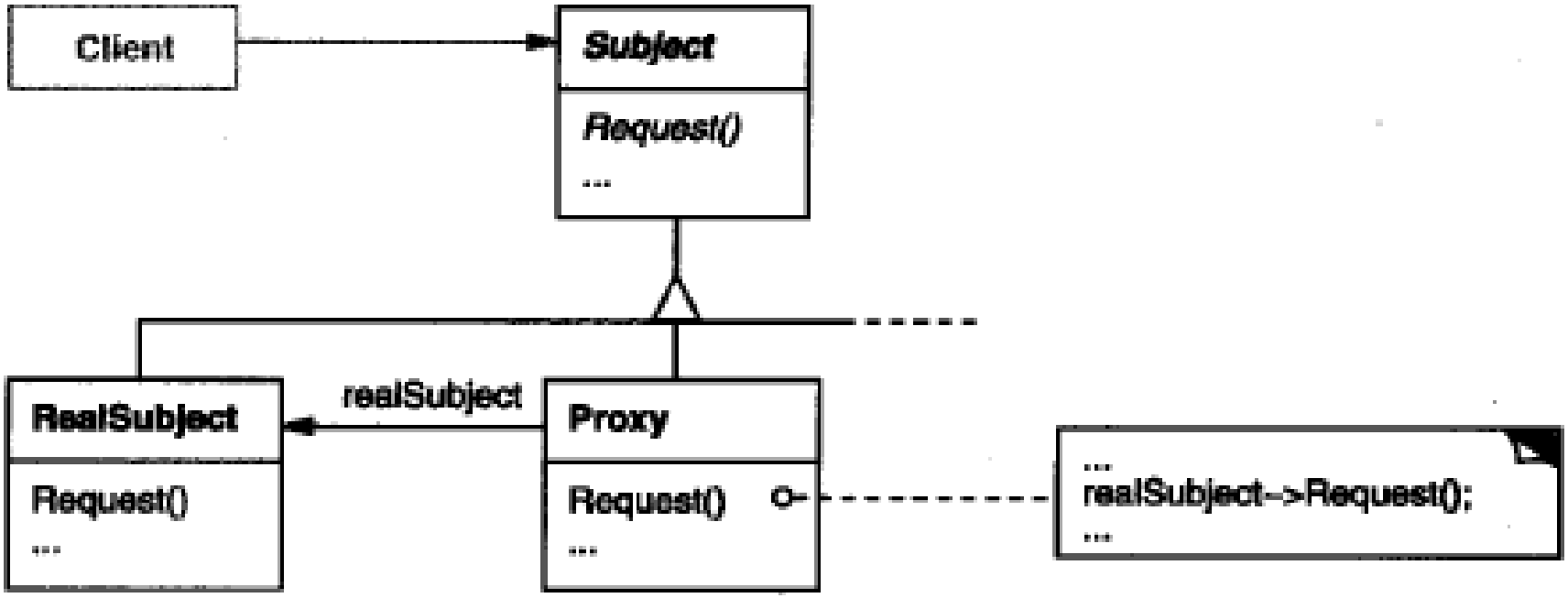
- 远程代理(Remote Proxy)
  - 为一个对象在不同的地址空间提供局部代表
- 虚代理(Virtual Proxy)
  - 根据需要创建开销很大的对象
  - e.g. ImageProxy
- 保护代理(Protection Proxy)
  - 控制对原始对象的访问
  - 保护代理用于对象应该有不同的访问权限时



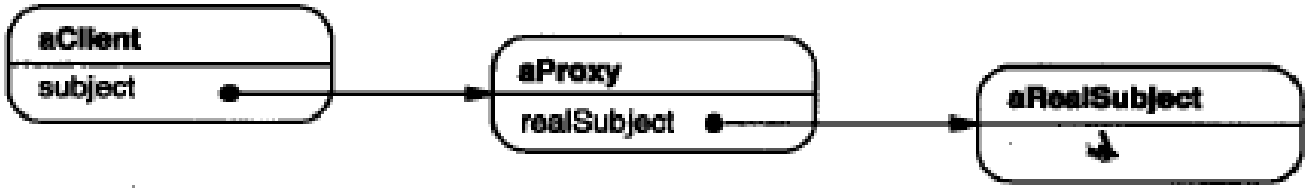
4.适用性

- 智能指引(Smart Reference)
  - 取代简单的指针，在访问对象时执行一些附加操作
  - 典型用途：
    - 对指向实际对象的引用计数。当该对象没有引用时，自动释放(Smart Pointers)
    - 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，以确保其他对象不能改变它

5.结构



运行时刻一种可能的代理结构的对象图

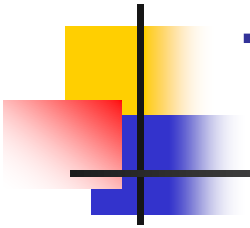


6. 参与者

- Proxy (ImageProxy)
  - 保存一个引用使得代理可以访问实体
  - 提供一个与Subject的接口相同的接口，使得代理可以用来替代实体
  - 控制对实体的存取，并可能负责创建和删除它
  - 其他功能：
    - Remote Proxy：对请求及其参数进行编码，先实体发送已编码的请求
    - Virtual Proxy：缓存实体的附加信息，以延迟对它的访问
    - Protection Proxy：检查调用者是否具有需要的访问权限

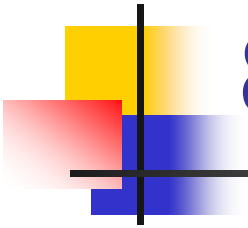
6. 参与者

- Subject (Graphic)
  - 定义RealSubject 和 Proxy的公用接口
- RealSubject (Image)
  - 定义Proxy所代表的实体



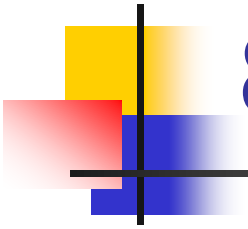
7. 协作

- 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
  - Remote Proxy隐藏一个对象存在不同地址空间的事实
    - 远程
  - Virtual Proxy可以进行最优化，例如根据要求创建对象
    - 延迟创建对象
  - Protection Proxy和Smart Reference运行访问一个对象时有一些附加的内务处理
    - 附加操作



8. 效果

- Proxy模式可以对用户隐藏“copy-on-write”的优化方式
  - 对一个负责对象进行copy操作，代价很大
  - 如果copy版本上没有被修改，则不需要产生一份copy
  - 用代理延迟拷贝过程，从而保证：

    只有当这个对象被修改时候才对它进行拷贝
  - 此时需要对实体进行计数管理

9. 实现

1)重载C++中的存取运算符

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;
}
```

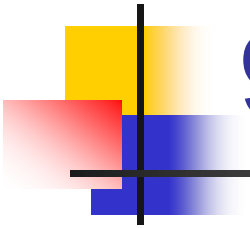
9. 实现

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

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

重载运算符

Client的使用

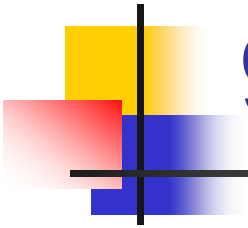
```
ImagePtr image = ImagePtr("anImageFileName");
image->Draw(Point(50, 100));
    // (image.operator->())->Draw(Point(50, 100))
```



9. 实现

1)重载C++中的存取运算符

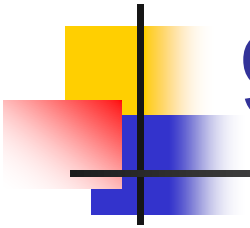
- 不适用的情况
  - 代理需要清楚地知道调用了哪个操作时
  - e.g. 动机中的例子



9. 实现

2) Proxy并不总是需要知道实体的类型

- 适用情况
  - Proxy可以通过Subject接口去访问RealSubject时
  - 此时，Proxy可以统一处理所有RealSubject类
- 不适用情况
  - Proxy要实例化RealSubject(e.g. Virtual Proxy)



9. 实现

- 实例化实体之前怎样引用它
  - 有些代理必须引用实体
    - 无论实体在硬盘还是在内存中
    - 因此，必须使用某种独立于地址空间的对象标识符
  - e.g.
    - 动机中，采用文件名作为对象标识符

10. 代码示例

- Subject

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

10. 代码示例

- RealSubject

```
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:
    // ...
};
```

10. 代码示例

- Proxy类

```
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;
};
```

接口定义

10. 代码示例

- Proxy类

```
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;
}
```

10. 代码示例

- Proxy类

```
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);
}
```

缓存图像尺寸

10. 代码示例

- Proxy类

Save: 将缓存的图像尺寸和文件名保存在一个流中

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

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

Load: 得到这个消息并初始化相应的成员函数

10. 代码示例

- Client使用

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

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


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

12.相关模式

- Adapter
  - Adapter为所适配的对象提供一个不同的接口
  - Proxy提供了与它的实体相同的接口
    - 可能是实体接口的一个子集
- Decorator ： 目的不同
  - Decorator为对象添加一个或多个功能
  - Proxy控制对对象的访问
  - Proxy实现与Decorator类似
    - Protection Proxy：基本相同
    - Remote Proxy：不包含对实体的直接引用，而只是一个间接引用(主机ID，主机上的局部地址)
    - Virtual Proxy：开始时使用一个简单引用，最终获取并使用一个直接引用

4.8 结构型模式的讨论

4.8.1 Adapter 与 Bridge

- 相同点
  - 给另一对象提供了一定程度上的间接性，从而有利于系统的灵活性
  - 都涉及从自身以外的一个接口向这个对象转发请求

4.8.1 Adapter 与 Bridge

- 不同之处：各自的用途
  - Adapter：解决两个已有接口间不匹配的问题
    - 不考虑这些接口是怎样实现的
    - 不考虑它们各自可能会如何演化
    - 不需要对两个独立设计的类中的任一个进行重新设计
  - Bridge：对抽象接口和它的(可能是多个)实现 部分进行桥接

4.8.1 Adapter 与 Bridge

- 应用场合：软件生命期的不同阶段
  - Adapter模式
    - 发现两个不兼容的类必须同时工作时
    - 目的：避免代码重复
    - 此处耦合不可预见
  - Bridge
    - 使用者事先知道“一个抽象有多个实现部分，而且抽象和实现两者是独立演化的”
  - Adapter在类已经设计后实施；Bridge在设计类之前实施

4.8.2 Composite、Decorator与Proxy

- Composite与Decorator
  - 相似点：递归组合
  - 不同点：目的不同
    - Decorator：不需要生成子类即可给对象添加职责
    - Composite
      - 使得多个相关对象能够以统一的方式处理
      - 多重对象可以被当作一个对象处理
      - 重点不在修饰，而在于表示
  - 两者目的不同，但具有互补性

4.8.2 Composite、Decorator与Proxy

- Decorator与Proxy
  - 相同点：
    - 都描述了怎样为对象提供一定程度上的间接引用
    - 实现时，都保留了指向另一个对象的指针
  - 不同点：设计目的不同

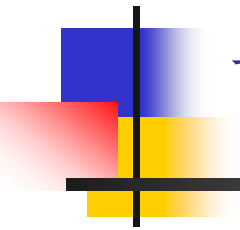
    1) Proxy
      - 不能动态添加和分离性质
      - 不是为递归组合而设计的
      - 目的是: 当直接访问一个实体不方便或不符合需要时，为这个实体提供一个替代者

4.8.2 Composite、Decorator与Proxy

- Decorator与Proxy
  - 不同点：设计目的不同

    2) 功能划分
    - Proxy中:实体定义关键功能，Proxy提供（或拒绝）对它的访问
    - Decorator: 组件仅提供部分功能，而Decorator负责完成其他功能

    3) 适用情况
    - Decorator适用于编译时不能确定对象的全部功能的情况
    - Proxy强调的关系(Proxy与其实体之间)可以静态表达



谢谢！