设计模式
Design Patterns

1.1 什么是设计模式

什么是设计模式

As an element of language , a pattern is an *instruction* , which shows *how* this spatial configuration can be used , over and over again to resolve the given system of forces , wherever the context makes it relevant .

The pattern is , in short , at the same time *a thing* , which happens in the world , and *the rule* which tells us how *to create* that thing , and *when* we must create it . It's *both a process and a thing* , both a description of a thing which is alive , and a description of the process which will generate that thing .

Christopher Alexander

The Timeless Way of Building

模式的基本要素

- 模式名称（pattern name）
- 问题（problem）
  - 描述应该在何时使用模式
- 解决方案（solution）
  - 设计的组成成分
  - 它们之间的相互关系及各自的职责和协作方式
- 效果（consequences）
  - 模式应用的效果及使用模式应权衡的问题

设计模式的抽象层次

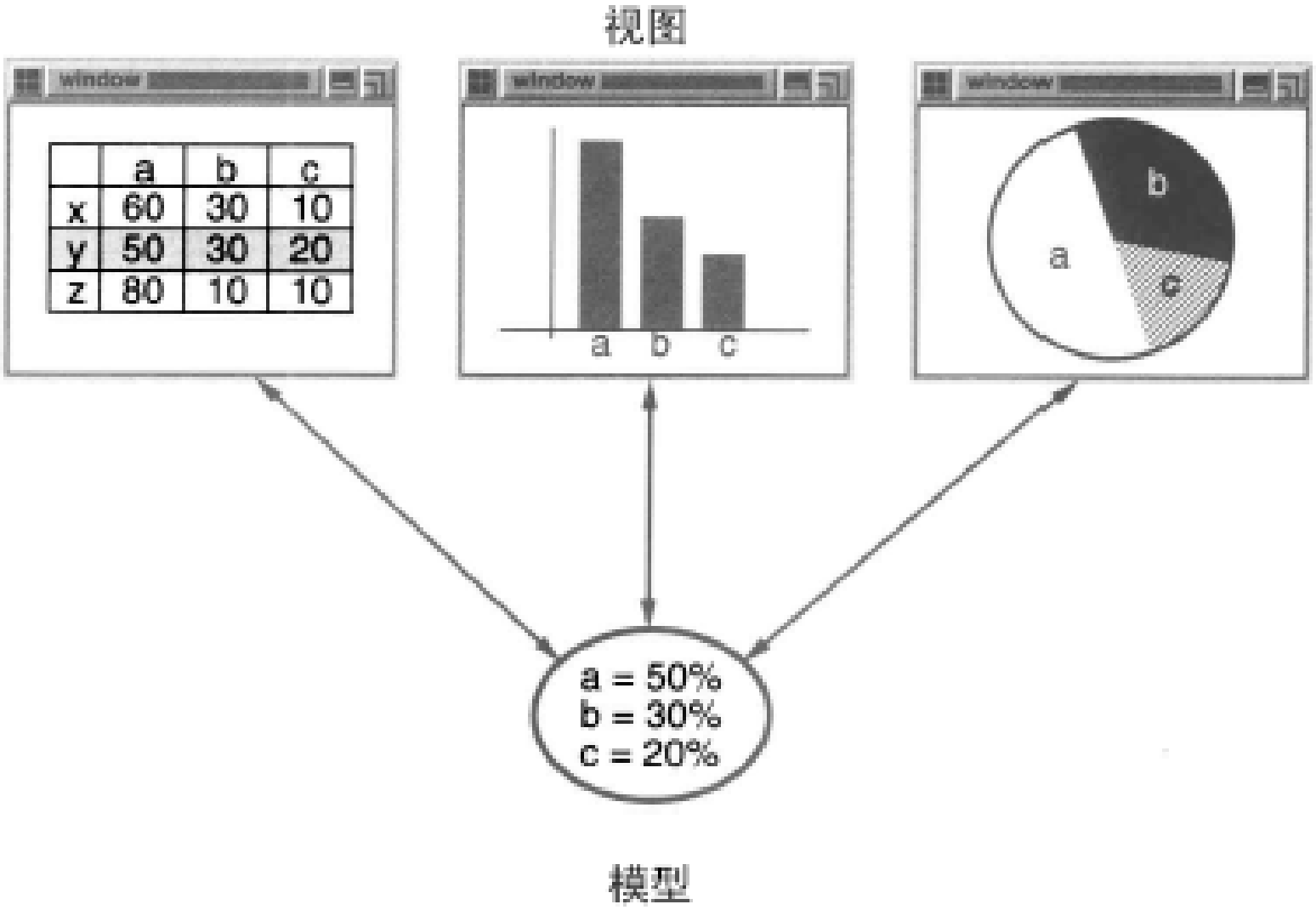
- 抽象层次
  - 对被用来在特定场景下解决一般设计问题的类和相互通信的对象的描述
- 程序设计语言的选择

1.2 Smalktalk MVC中的设计模式

MVC : Model / View / Controller

- MVC
  - 模型Model : 应用对象
  - 视图View : 屏幕上的表示
  - 控制器Controller : 用户界面对用户输入的相应
- 通过“订购 / 通知” 协议分离视图和模型

MVC


视图
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
window
a b c
window
a
b
c
a = 50%
b = 30%
c = 20%
模型

涉及的模式

- Observer模式
  - 适用问题
    - 将对象分离
    - 使得一个对象的改变能影响另一些对象
    - 这个对象并不需要知道那些被影响的对象的细节
- Composite模式
  - 适用问题
    - 将一些对象划为一组，并将该组对象当作一个对象来使用

涉及的模式

- Strategy模式
  - View-Controller关系是Strategy模式的一个例子
    - View使用Controller子类对象实现特定的响应策略
    - 通过改变View的Controller来改变View对用户输入的响应
  - Strategy模式适用问题
    - 需要静态或动态地替换一个算法
    - 有很多不同的算法
    - 算法中包含需要封装的复杂的数据结构

涉及的模式

- 其他
  - Factory Method
    - 用于指定View的缺省控制器
  - Decorator
    - 用于增加视图滚动

1.3描述设计模式

怎样描述设计模式

- 模式名和分类
- 意图
- 别名
- 动机
- 适用性
- 结构
- 参与者
- 协作
- 效果
- 实现
- 代码示例
- 已知应用
- 相关模式

1.4设计模式编目

23种设计模式（创建）

- Abstract Factory
  - 提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类
- Builder
  - 将一个复杂对象的创建与它的表示分离，使得同样的创建过程可以创建不同的表示
- Factory Method
  - 定义一个用于创建对象的接口，让子类决定将哪一个类实例化
  - 使一个类的实例化延迟到其子类

23种设计模式（创建）

- Prototype
  - 用原型实例指定创建对象的种类，并通过拷贝这个原型来创建新的对象
- Singleton
  - 保证一个类仅有一个实例，并提供一个访问它的全局访问点

23种设计模式（结构）

- Adapter
  - 将一个类的结构转换成客户希望的另外一个接口
  - 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
- Bridge
  - 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
- Composite
  - 将对象组合成树形接口以表示“部分-整体”的层次结构
  - 使得客户对单个对象和复合对象的使用具有一致性

23种设计模式（结构）

- Decorator
  - 动态地给一个对象添加一些额外的职责
  - 就扩展功能而言，此模式比生成子类方式更为灵活
- Facade
  - 为子系统中的一组接口提供一个一致的界面
  - 此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

23种设计模式（结构）

- Flyweight
  - 运用共享技术有效地支持大量细粒度的对象
- Proxy
  - 为其他对象提供一个代理以控制对这个对象的访问

23种设计模式（行为）

- Chain of Responsibility
  - 解耦：请求的发送者 ←→ 请求的接收者
- Command
  - 将一个请求封装为一个对象，使得可以用不同的请求对客户进行参数化
  - 对请求排队或记录请求日志，支持撤销操作

23种设计模式（行为）

- Interpreter
  - 给定一个语言，定义文法表示，定义解释器
  - 该解释器使用该表示来解释语言中的句子
- Iterator
  - 提供一种方法顺序访问一个聚合对象中各个元素，而无需暴露该对象的内部表示

23种设计模式（行为）

- Mediator
  - 用一个中介对象来封装一系列的对象交互
  - 中介者使各对象不需要显示地相互引用（松散耦合），而且可以独立地改变它们之间的交互
- Memento
  - 在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态
  - 便于将对象恢复到保存的状态

23种设计模式（行为）

- Observer
  - 定义对象间一对多的依赖关系
  - 当一个对象状态发生变化时，所以依赖于它的对象都将得到通知并自动刷新
- State
  - 允许一个对象在其内部状态改变时改变它的行为
  - 对象看起来似乎修改了它所属的类

23种设计模式（行为）

- Strategy
  - 定义一系列的算法，对其进行封装，使其可以互相替换
  - 使得算法的变化可以独立于使用它的客户
- Template Method
  - 定义一个操作中的算法骨架，将一些步骤延迟到子类中
  - 使得子类可以不改变一个算法的结构即可重定义该算法中某些特定步骤

23种设计模式（行为）

- Visitor
  - 表示一个作用于某对象结构中的各元素的操作。
  - 使得可以在不改变各元素的类的前提下定义作用于这些元素的新操作

设计模式所支持的设计的可变方面

目的	设计模式	可变的方面
创建	Abstract Factory	产品对象家族
	Builder	如何创建一个组合对象
	Factory Method	被实例化的子类
	Prototype	被实例化的类
	Singleton	一个类的唯一实例

设计模式所支持的设计的可变方面

目的	设计模式	可变的方面
结构	Adapter	对象的接口
	Bridge	对象的实现
	Composite	一个对象的结构和组成
	Decorator	对象的职责，不生成子类
	Facade	一个子系统的接口
	Flyweight	对象存储开销
	Proxy	如何访问一个对象，对象的位置

设计模式所支持的设计的可变方面

目的	设计模式	可变的方面
行为	Chain of Responsibility	满足一个请求的对象
	Command	何时、怎样满足一个请求
	Interpreter	一个语言的文法及解释
	Iterator	如何遍历、访问一个聚合的各元素
	Mediator	对象间怎样交互、和谁交互
	Memento	一个对象中哪些私有信息存放在该对象之外，以及在什么时候进行存储

设计模式所支持的设计的可变方面

目的	设计模式	可变的方面
行为	Observer	多个对象依赖于另外一个对象，而这些对象又如何保持一致
	State	对象的状态
	Strategy	算法
	Template Method	算法中的某些步骤
	Visitor	某些可作用于一个（组）对象上的操作，但不修改这些对象的类

Ch2 : 实例研究
设计一个文档编辑器

Lexi

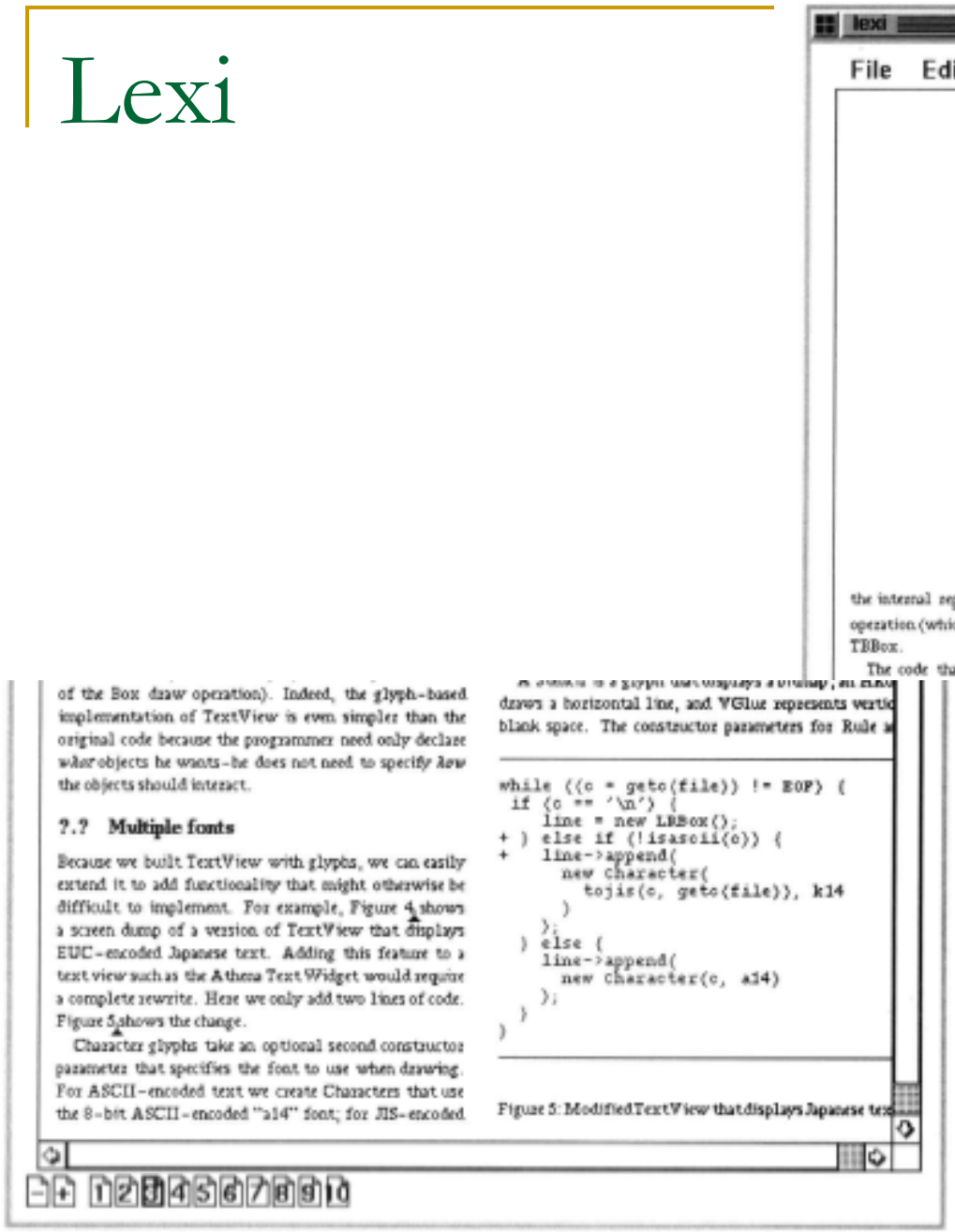


图2-1 Lexi的用户界面

2.1 设计问题

- 文档结构
- 格式化
- 修饰用户界面
- 支持多种视感（look-and-feel）标准
- 支持多种窗口系统
- 用户操作
- 拼写检查和连字符

2.2文档结构

- 数据的内部表示
  - 是决定其他设计的关键
  - 也是往往需要首先确定的
- 内部表示应该支持
  - 保持文档的物理结构
  - 可视化生成和显示文档
  - 根据显示位置来映射文档内部表示的元素
    - （设计内部数据表示时需要考虑外部界面的要求）

2.2文档结构

- 限制条件
  - 一致对待文本和图形
    - 避免将一个视为另一个的特例
  - 一致地对待简单元素和组合元素
    - 不能过分强调单个元素和元素组之间的差别
  - 对文本进行分析
    - 与其他限制条件存在矛盾

2.2.1递归组合
Recursive Composition

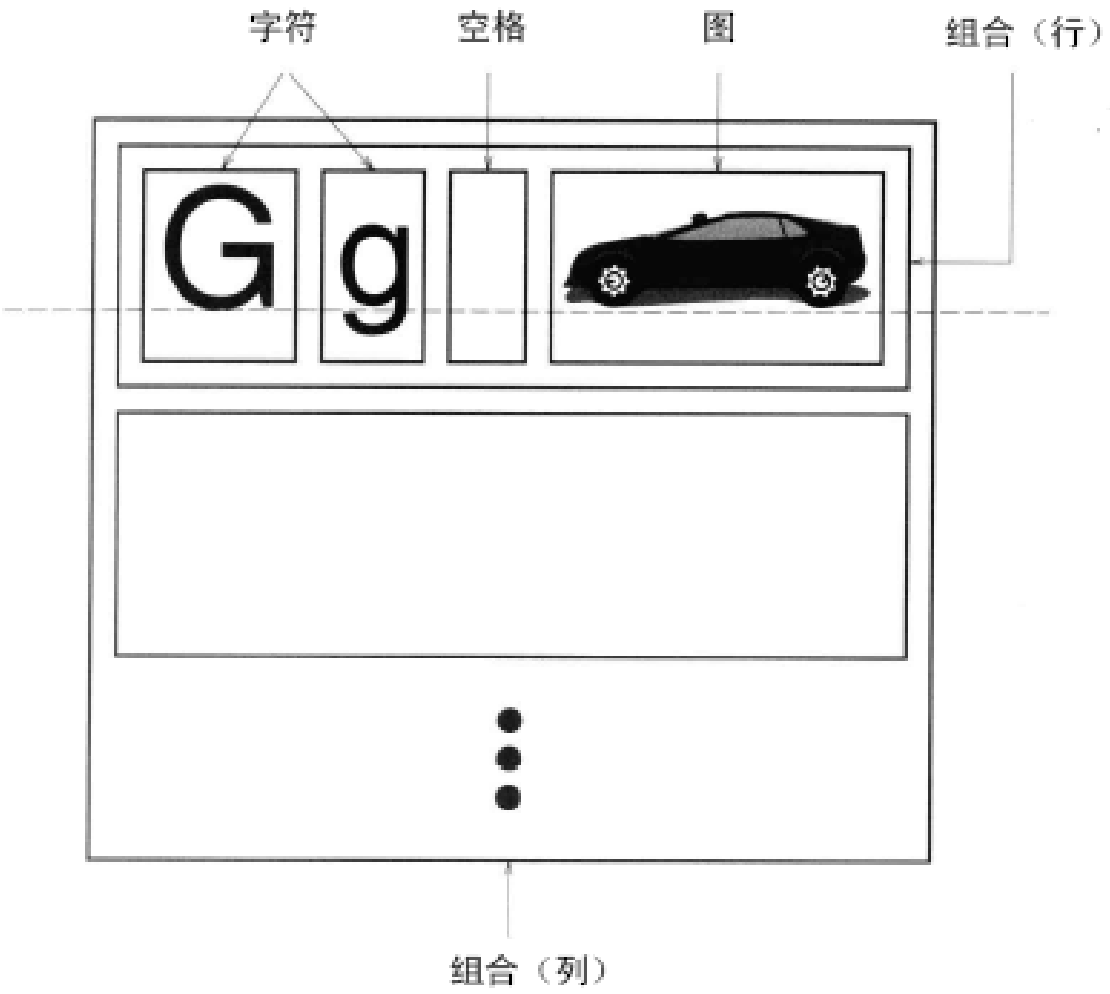


图2-2 包含正文和图形的递归组合

2.2.1递归组合

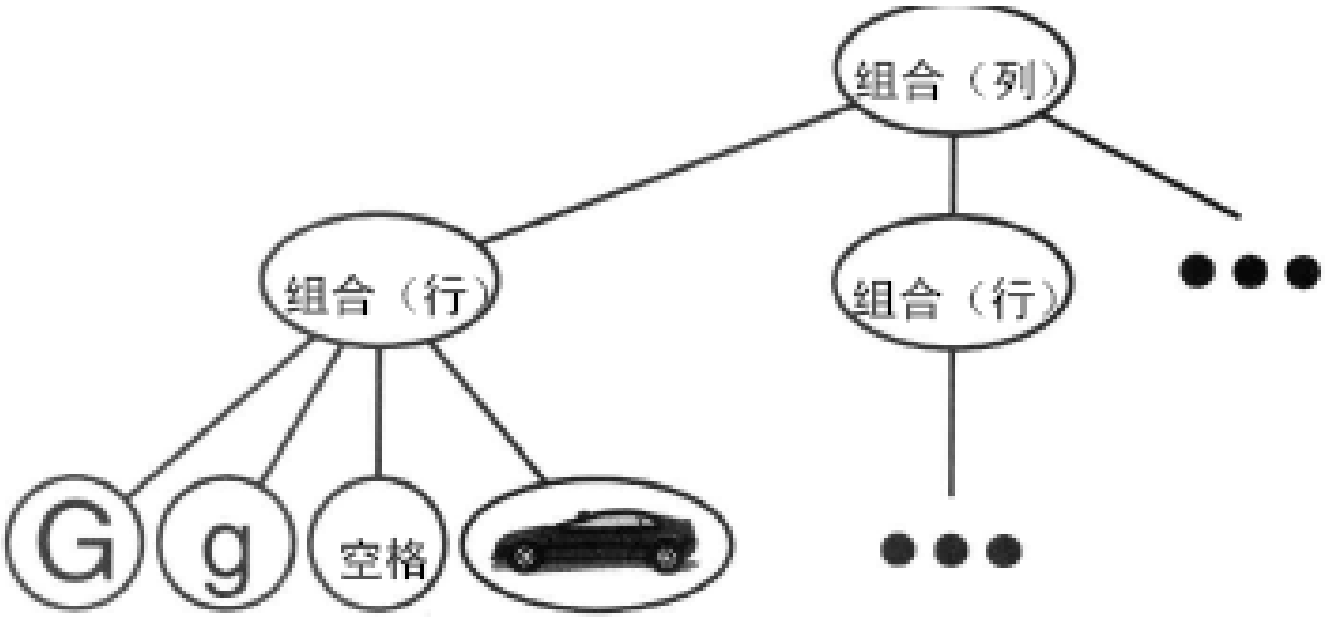


图2-3　递归组合的对象结构

1. 用类来表示对象
2. 类具有统一的操作接口

2.2.2图元

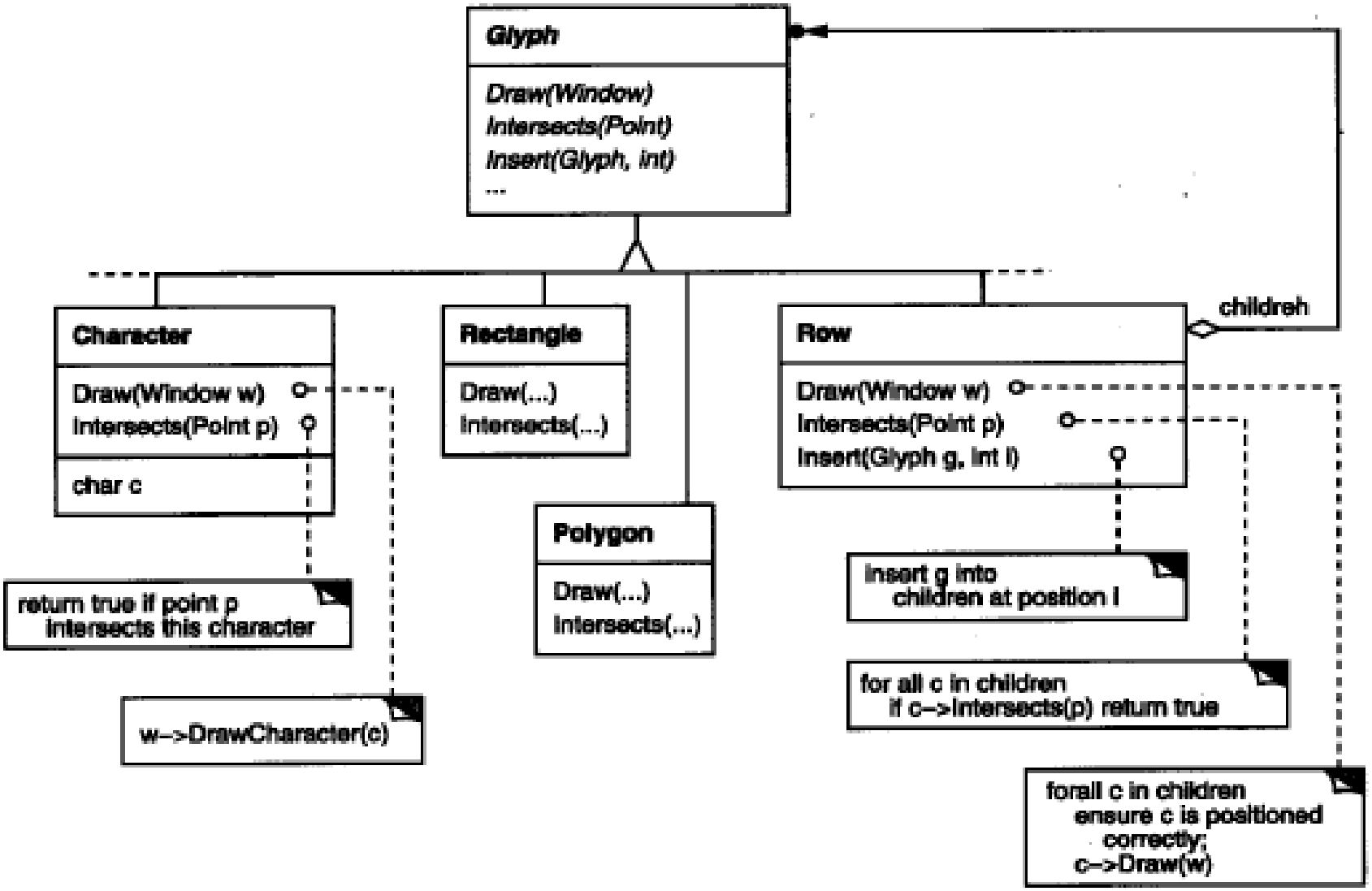


图2-4 部分Glyph类层次

2.2.2图元

- 基本Glyph接口

表2-1　基本Glyph接口

Responsibility		Operations
Appearance	绘制	Virtual Void Draw (Window*)
	占用空间	Virtual Void Bounds (Rect&)
hit detection		Virtual bool Intersects (Const Point&)
Structure	对子图元操作	Virtual Void Insert (Glyph*, int)
		Virtual Void Remove (Glyph*)
	父子图元	Virtual Glyph* Child (int)
		Virtual Glyph* Parent()

2.2.3 组合模式

- Composite模式
  - 描述了面向对象的递归组合的本质

2.3 格式化

- 格式化
  - 将一个图元集合分解为若干行
  - 格式化(formatting)　分行(linebreaking)

2.3.1封装格式化算法

- 格式化算法的复杂性
  - 将算法独立于文档结构之外
    - 自由增加Glyph子类而不考虑格式算法
    - 增加格式算法不要求修改已有的图元类
- 将算法独立出来，封装到对象中
  - 定义一个封装格式化算法的对象的类层次结构

2.3.2 Compositor与Composition

- Compositor ：封装了格式化算法的对象
- Composition：被格式化的图元

表2-2　基本Compositor接口

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

2.3.2 Compositor与Composition

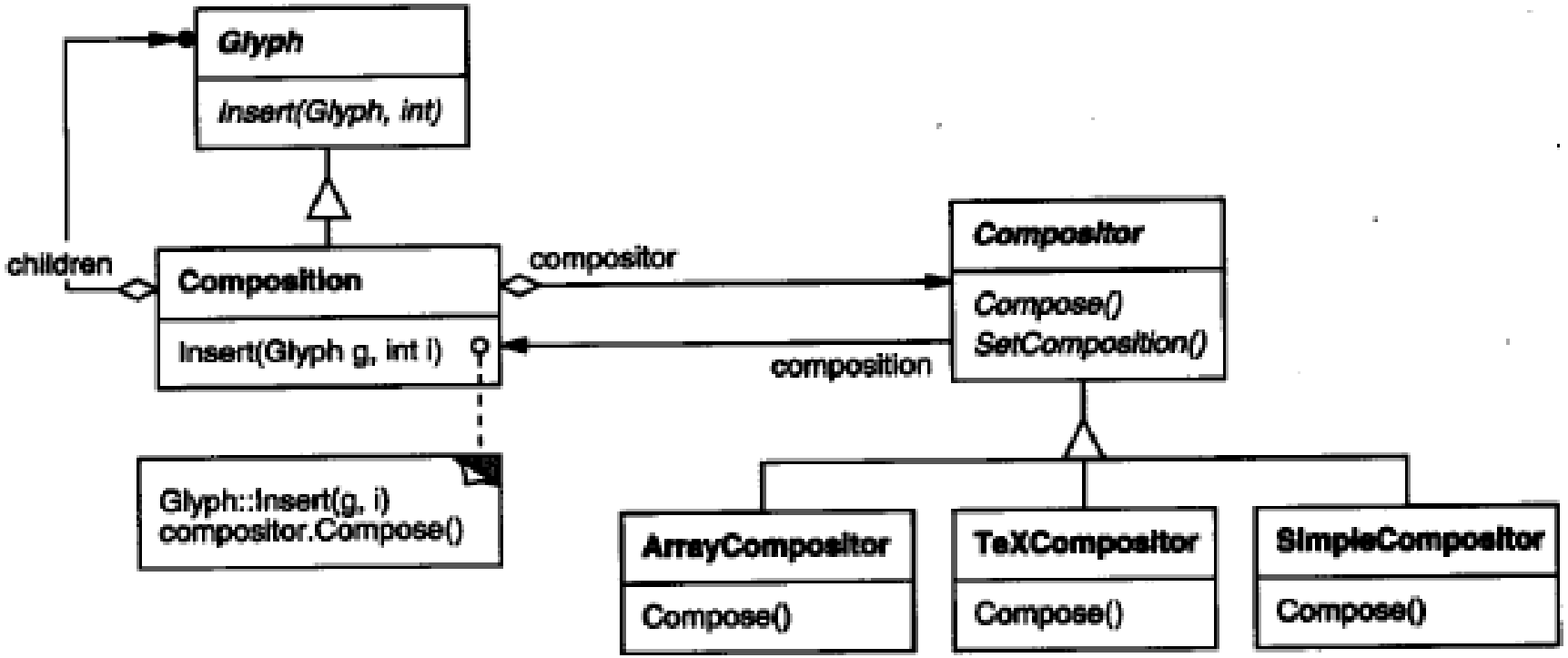


图2-5 Composition和Compositor类间的关系

2.3.2 Compositor与Composition

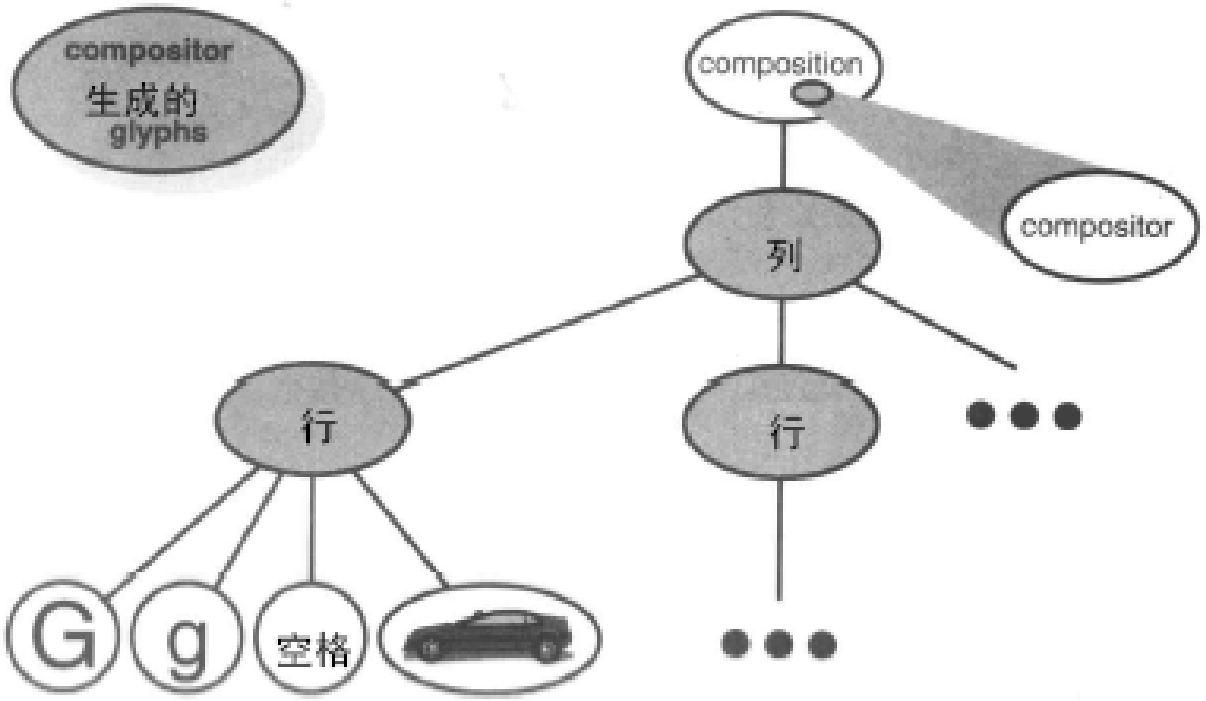


图2-6 对象结构反映Compositor制导的分行

通过Compositor-Composition类的分离确保了Composition（支持文档物理结构的代码）和Compositor（支持不同格式化算法的代码）间的分离

2.3.3策略模式

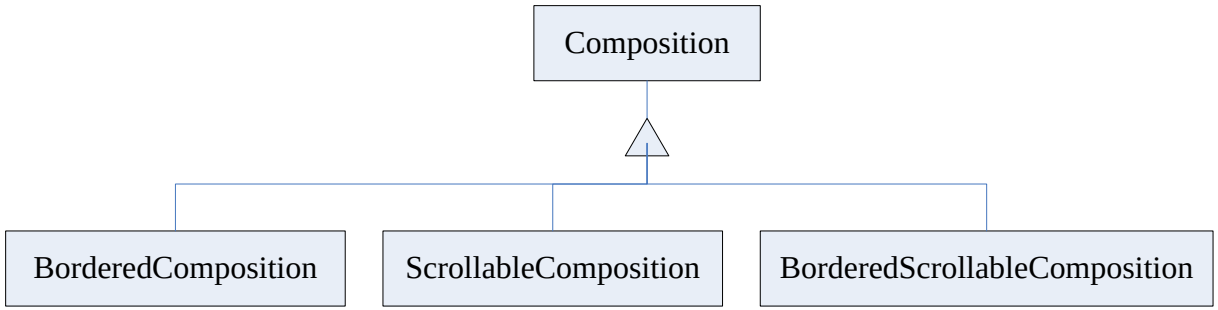
- Strategy模式
  - 目的：在对象中封装算法
  - 构成
    - Stragey对象：封装了不同的算法　Compositor
    - 操作环境：　Composition
  - 关键点
    - 为Strategy和它的环境设计足够通用的接口，以支持一系列的算法

2.4 修饰用户界面

- 修饰用户界面
  - 两种修饰
    - 边界
    - 滚动条
  - 便于增加和去除修饰
  - 用户界面对象不知道修饰的存在

2.4.1透明围栏

- 修饰用户界面解决方案1
  - 扩展子类 → 子类爆炸问题



2.4.1透明围栏

- 修饰用户界面解决方案2： 对象组合
  - 装饰对象（e.g. Border） VS 图元（Glyph）
    - 在边界中包含图元
    - 在图元中包含边界 → 需要修改已存在的代码

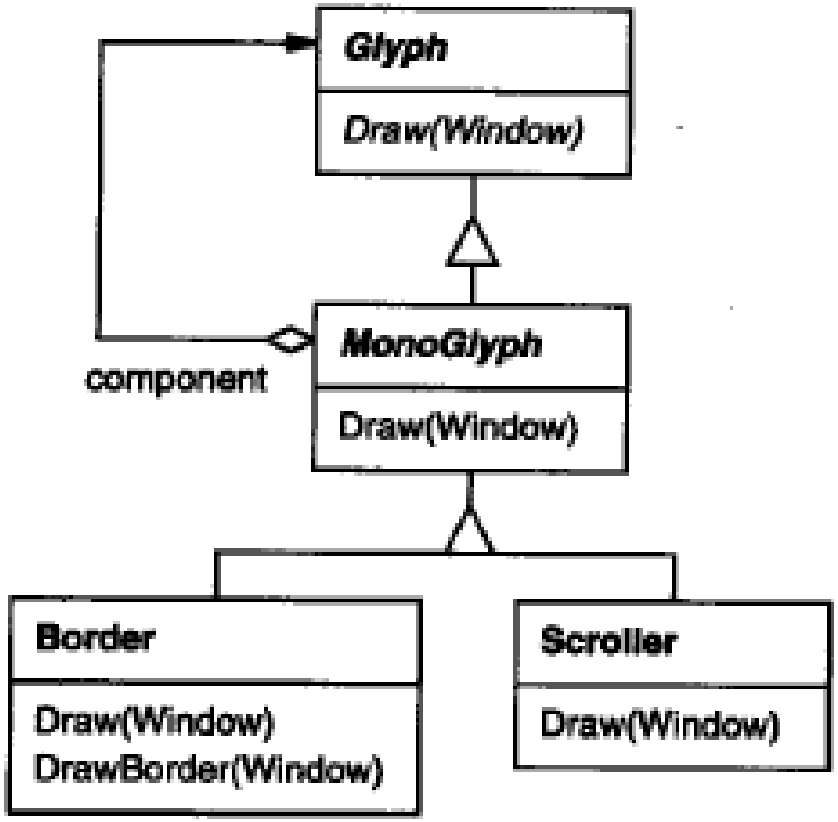
2.4.1透明围栏

- Border类的设计： Glyph的子类
  - 原因：
    1. 有形状
    2. 用户应该一致地对待图元

- 透明围栏（Transparent Enclosure）
  - 单子女（单组件）的组合
    一个Border（父组件）修饰一个图元（子组件）
  - 兼容的接口
    客户不能分辨是处理组件还是围栏

2.4.2 MonoGlyph

- MonoGlphy




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

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

图2-7 MonoGlyph类关系

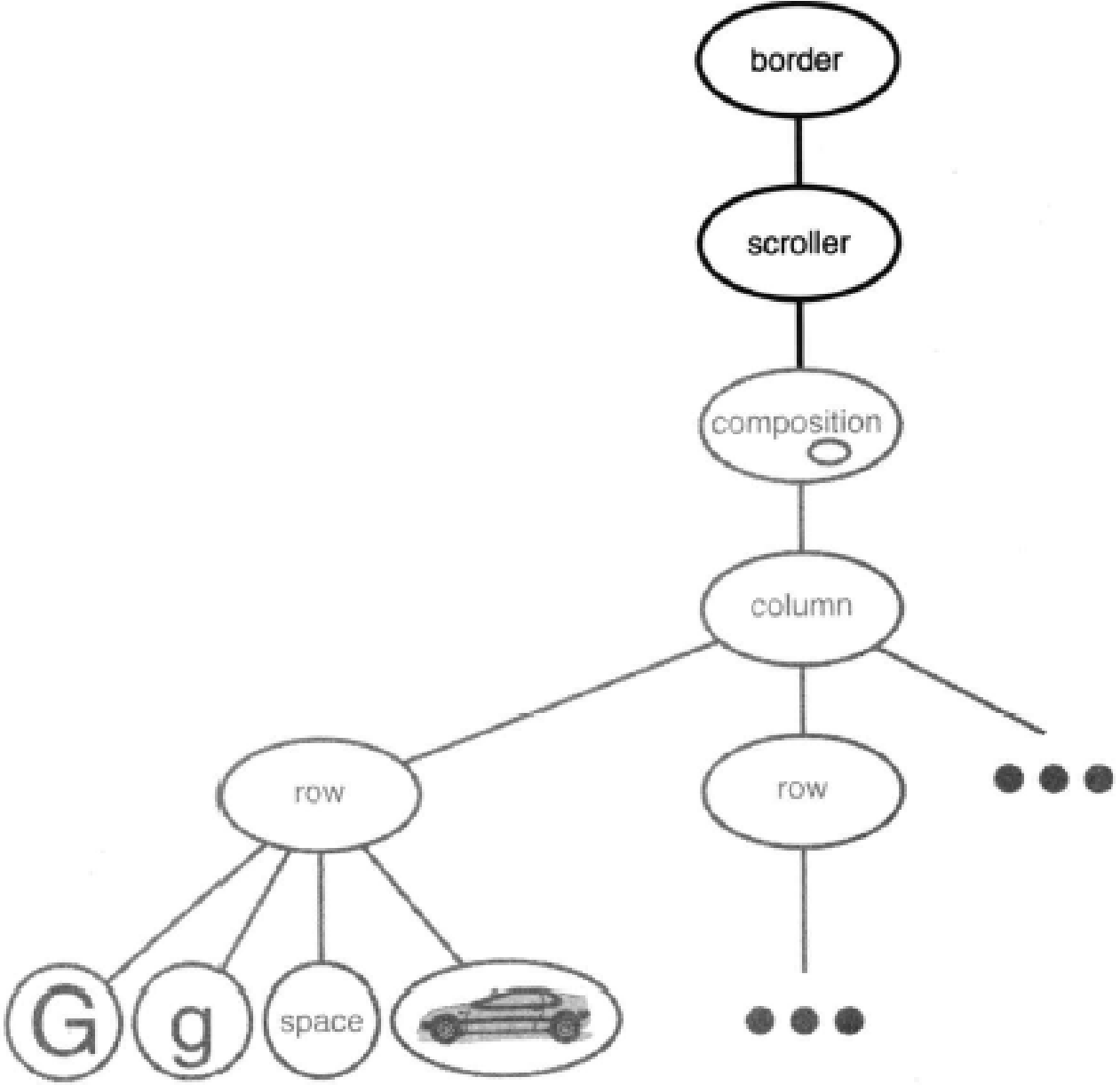


图2-8 嵌入对象结构

1. 可以交换组合顺序 （scroller / border）
2. Border/scroller只有一个子对象

2.4.3 Decorator模式

- 修饰
  - 给一个对象增加职责

2.5 支持多种视感标准

- 不同视感标准之间的差异
  - 跨平台的可移植性的一大障碍
  - 设计目标：
    - Lexi符合多个已经存在的视感标准
    - 新标准出现时能容易地支持
    - 支持最大限度灵活性： 运行时刻可以进行改变

2.5.1 对象创建的抽象

- 界面风格 → 对窗口组件（Widgets）的规则
- 用两个窗口组件图元集来实现多个视感标准
  - 抽象类集合
    - Glyph的子类
    - 对每一种窗口组件图元定义一个抽象Glyph子类
      - e.g. ScrollBar
  - 具体类集合
    - 实现不同视感标准
    - e.g. ScrollBar → MotifScrollBar / PMScrollBar
- 需要在合适的时间创建合适的对象
  - 抽象对象创建过程
    1. 避免显式的构造器调用
    2. 能很容易替换整个窗口组件集合

2.5.2 工厂类和产品类

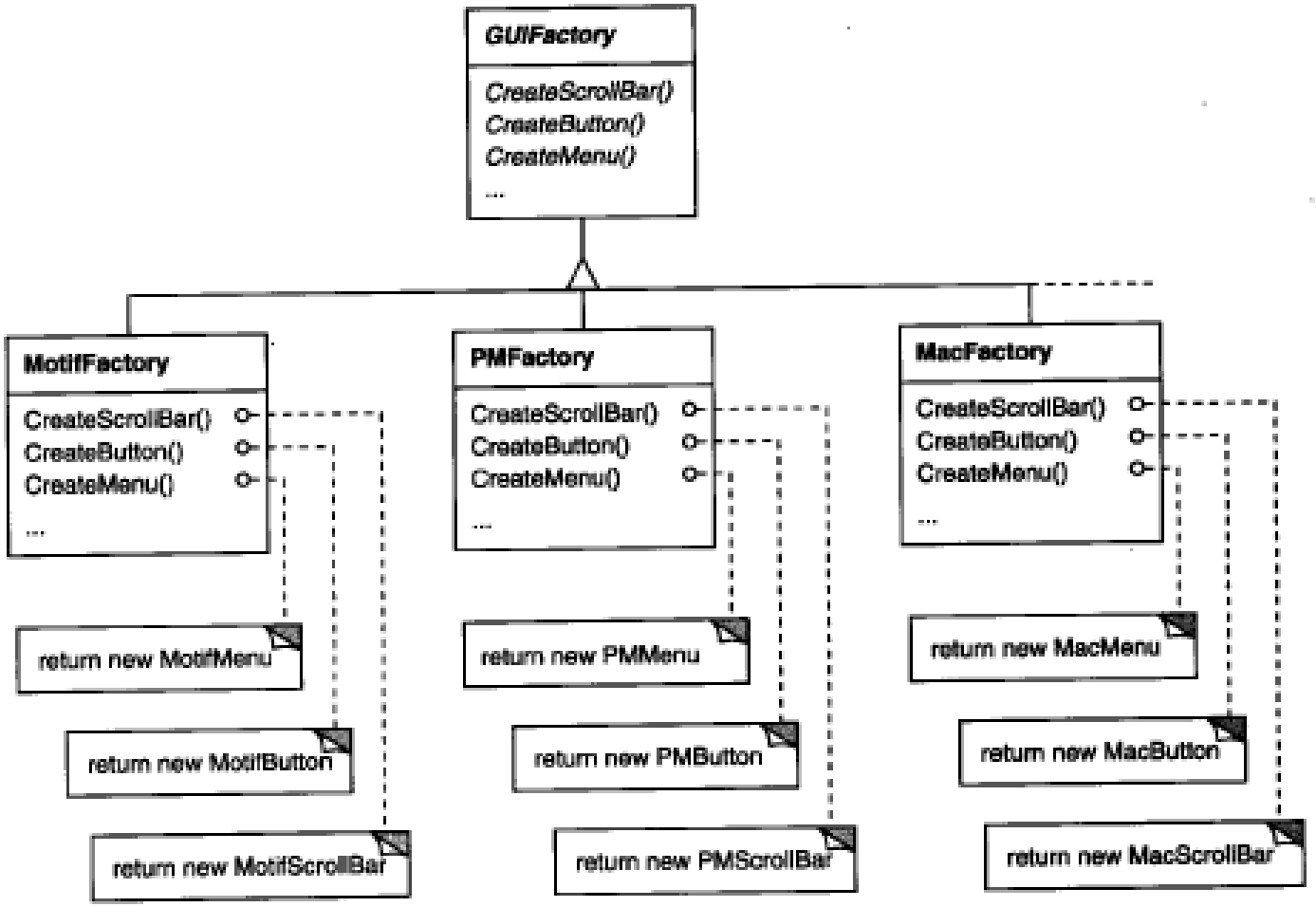


图2-9 GUIFactory类层次

ScrollBar* sb = new MotifScrollBar();

→

ScrollBar* sb = guiFactory->CreateScrollBar();

2.5.2 工厂类和产品类

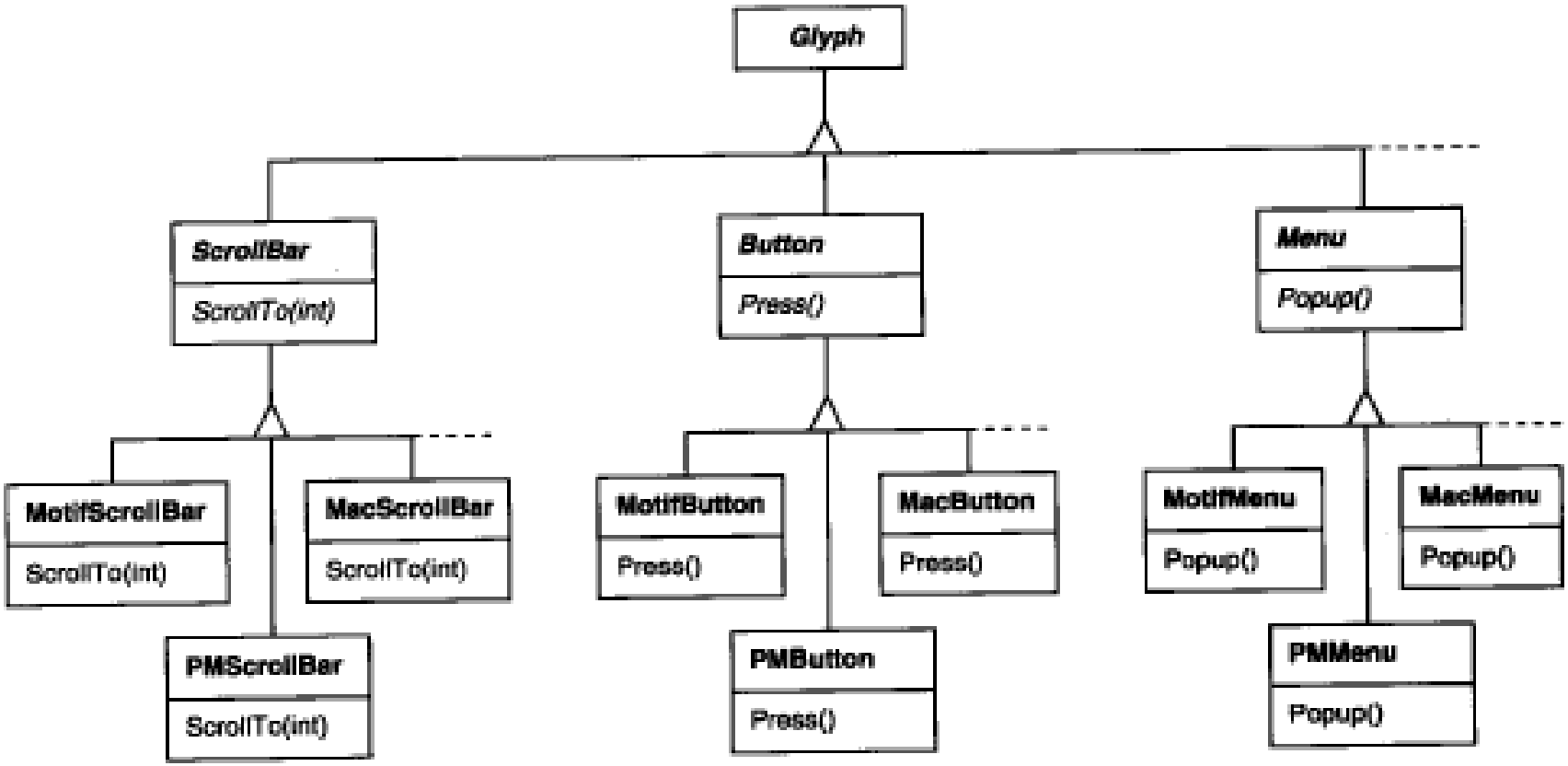


图2-10 抽象产品类和具体子类

2.5.2 工厂类和产品类

- 问题：GUIFactory实例如何创建？
  - 编译时确定

    GUIFactory* guiFactory = new MotifFactory;
  - 程序启动时的字符串

```
GUIFactory* guiFactory;
const char* styleName = getenv("LOOK_AND_FEEL");
if (strcmp(styleName, "Motif") == 0) {
    guiFactory = new MotifFactory;
} else {
    guiFactory = new DefulyGUIFactory;
}
```

增加新的工厂时需要改代码

通过维护登记表（记录需要检查的字符串）可以加入新的工厂，而不用改变代码

2.5.3Abstract Factory模式

- Abstract Factory模式
  - 主要参与者：工厂（Factory）和产品（Product）
  - 在不直接实例化类的情况下创建一系列的产品对象
  - 适用于
    - 产品对象的数目和种类不变
    - 具体产品系列之间存在不同
  - 使用方式
    - 通过实例化一个特定的具体工厂对象来选择产品系列
    - 用一个不同的具体工厂实例（替换原来的工厂对象）来改变整个产品系列
  - 产品系列

2.6 支持多种窗口系统

- 移植性问题：窗口环境
  - 目标：

    使得Lexi可以在尽可能多的窗口系统上运行

2.6.1 是否可以使用Abstract Factory模式

- Abstract Factory模式的基础
  - 为所有的产品类创建一个公共的抽象产品类
- 问题：
  - Lexi在已有窗口系统中，无法完成公共的抽象产品类的提取
  - 解决：
    - 为不同的窗口系统做一个统一的抽象

2.6.2封装实现依赖关系

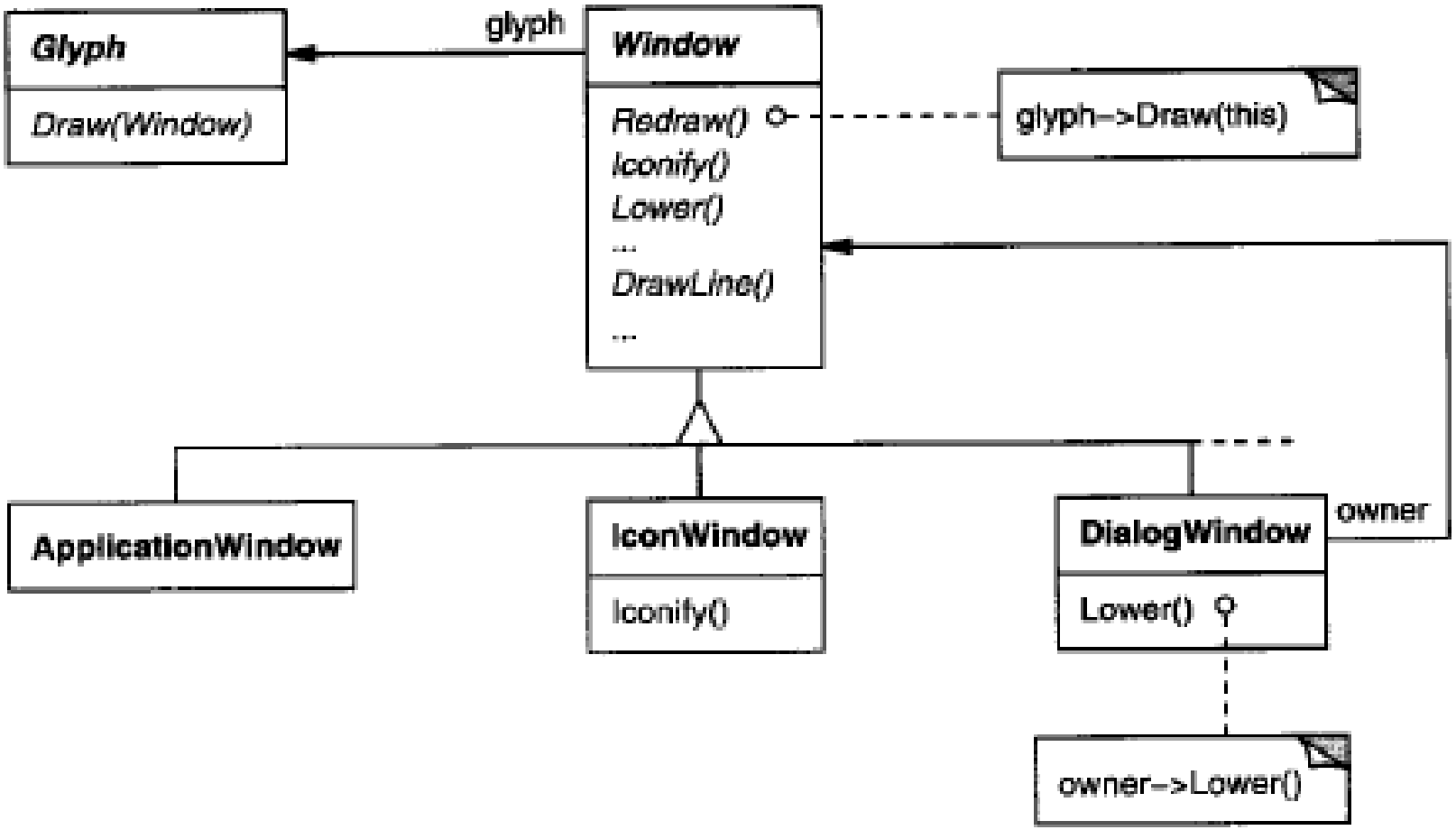
- Windows类

表2-3　Windows类接口

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

Windows类提供一个支持大多数窗口系统的方便的接口

2.6.2封装实现依赖关系



Window : 抽象类
子类: 支持用户用到的不同种类的窗口

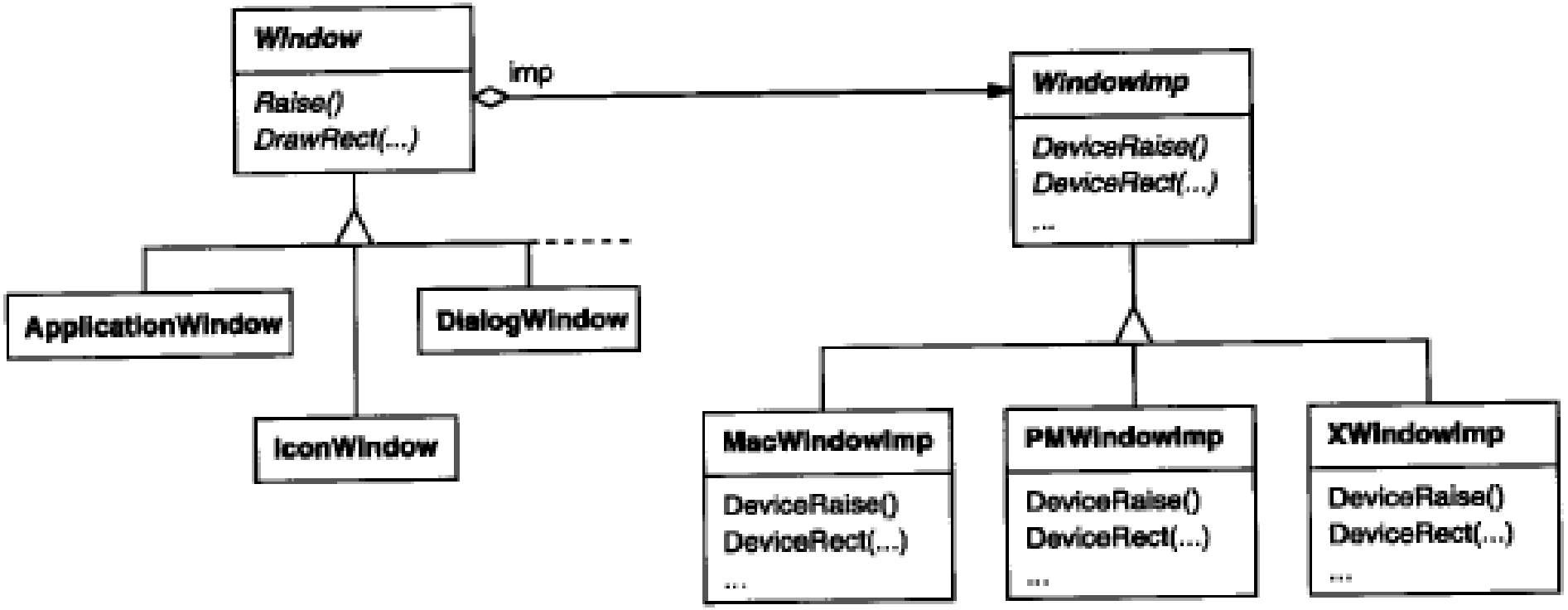
2.6.2封装实现依赖关系

- 如何实现与平台相关的窗口？
  - 方式1：
    - 实现Window类和其子类的多个版本

    e.g. Window → PlatAWindow , PlatBWindow

    IconWindow → PlatAIconWindow,PlatBIconWindow
    - 维护问题
  - 方式2：
    - 在每个窗口层次中创建特定的子类
    - 子类爆炸问题
  - 解决方式
    - 对变化的概念进行封装
    - 在一个对象中封装窗口系统的功能，然后根据此接口实现Window类及其子类

2.6.3 Window和WindowImp



WindowImp：封装了窗口系统相关代码的抽象类

2.6.3 Window和WindowImp

- 此方式的优点
  - 避免了对窗口系统的直接依赖
    - 使得Window类层次保持相当较小并且较稳定
  - 能针对新的窗口系统进行扩展

2.6.3 Window和WindowImp

1. WindowImp的子类

将用户请求转变为对特定窗口系统的操作

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

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

```
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);
}
```

2.6.3 Window和WindowImp

2. 用WindowImp来配置Windows

- 问题：
  - 如何用一个合适的WindowImp子类来配置一个窗口 ？
  - 即：_imp什么时候初始化？ 初始化成什么具体子类？
- 解决方式： Abstract Factory模式
  - 抽象工厂类WindowSystemFactory

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

2.6.4 Bridge模式

- Window与WindowImp： 功能分离
  - Window类接口针对应用程序员
  - WindowImp接口针对窗口系统

- Bridge模式
  - 目的：运行分离的类层次一起工作
  - 两个分离的类层次
    - e.g. 一个支持窗口的逻辑概念，另一个描述了窗口的不同实现
  - Bridge模式运行我们保持和加强对窗口的逻辑抽象（Window），而不触及窗口系统相关的代码（WindowImp）

2.7 用户操作

- 用户操作：e.g. 新建、保存、退出…
- 问题：
  - 允许多个用户界面对应一个操作
  - 用户界面类和操作实现之间是松耦合的
  - 能对大多数功能支持Undo和Redo操作
    - e.g. 可以对Delete操作进行撤销；对保存操作不存在撤销
- 需要一个简单、可扩充的机制

2.7.1 封装一个请求

- MenuItem图元
  - Glyph的子类
  - 响应Client的请求 （一个请求处理可能涉及多个操作）
- 不能将用户操作设计为MenuItem的子类（为什么？）
- 封装请求
  - 在Command对象中封装每一个请求

2.7.2 Command类及其子类

- Command类层次

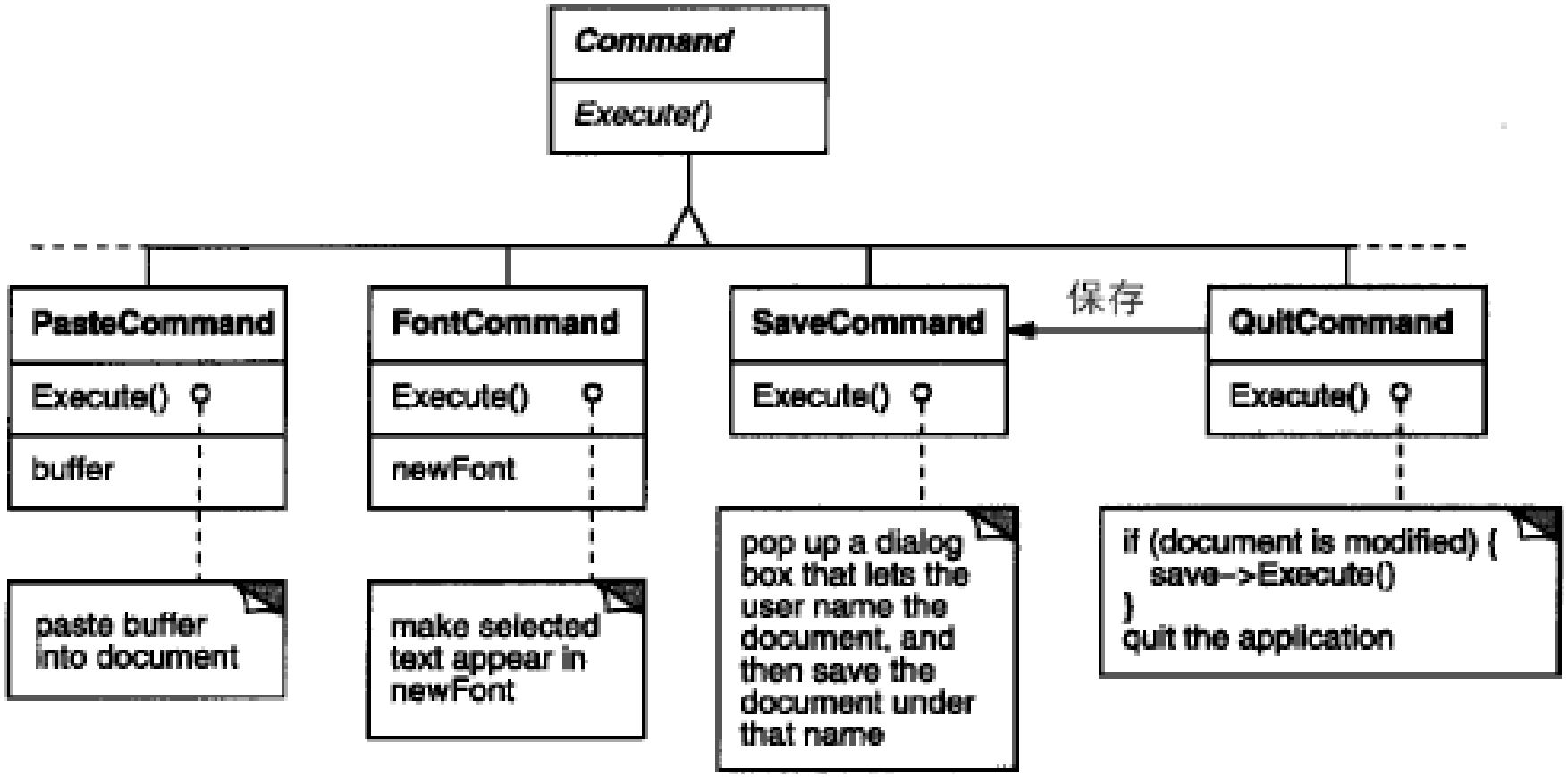


图2-11 部分Command类层次

2.7.2 Command类及其子类

- MenuItem与Command

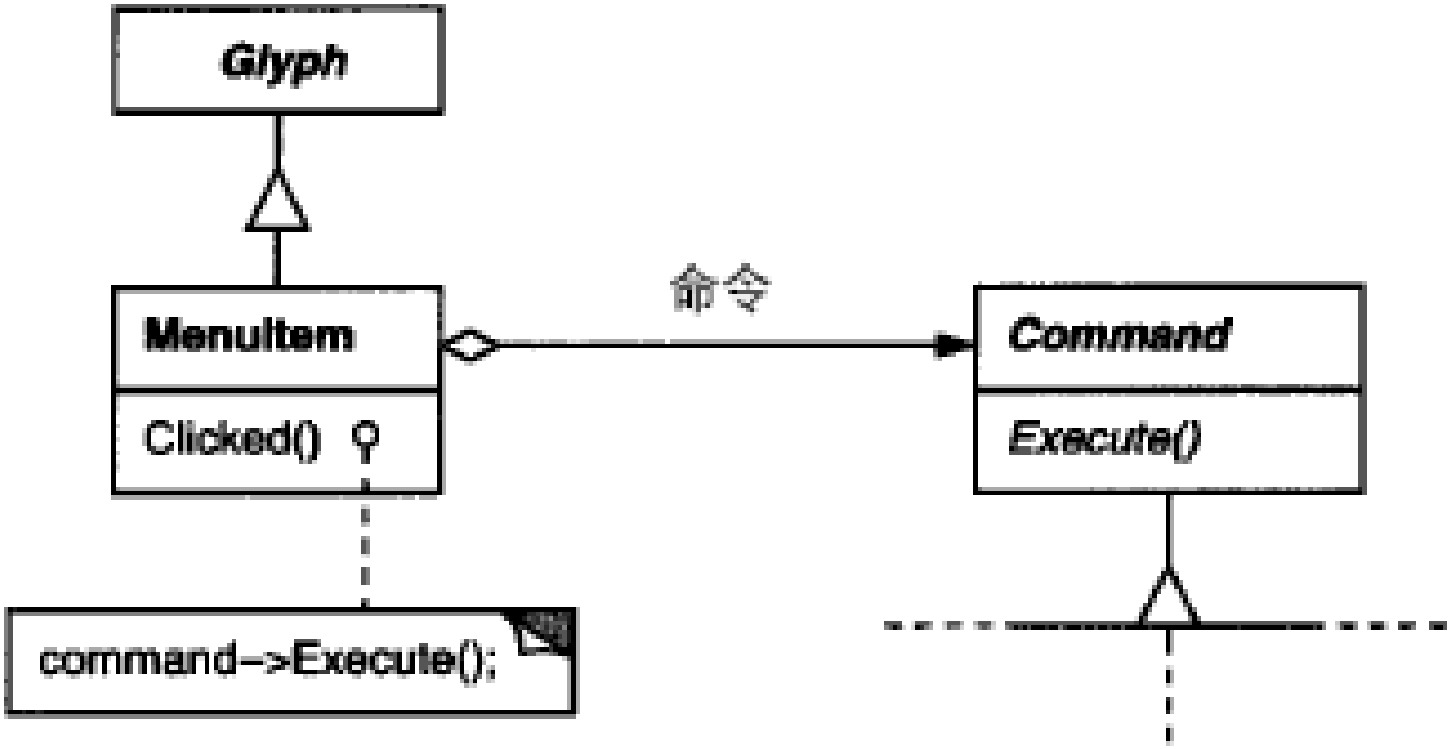
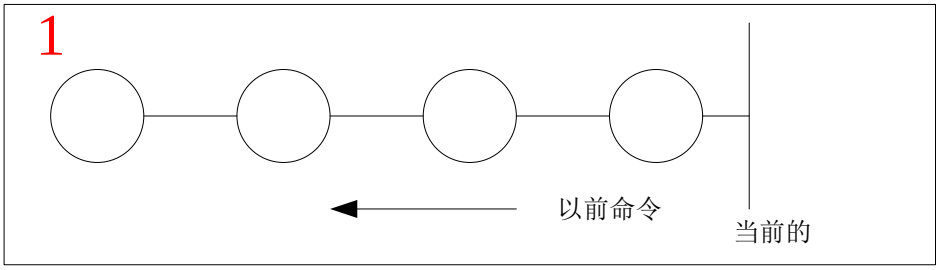


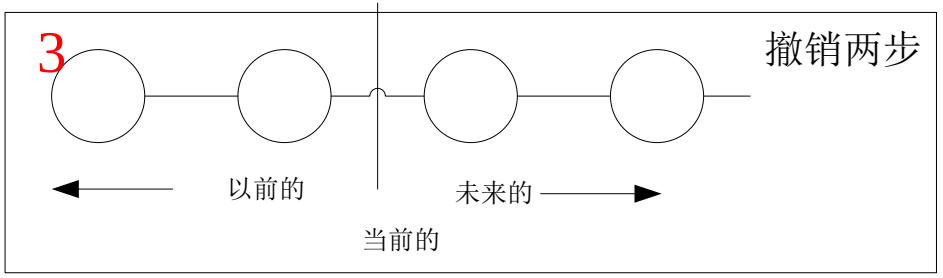
图2-12 MenuItem-Command关系

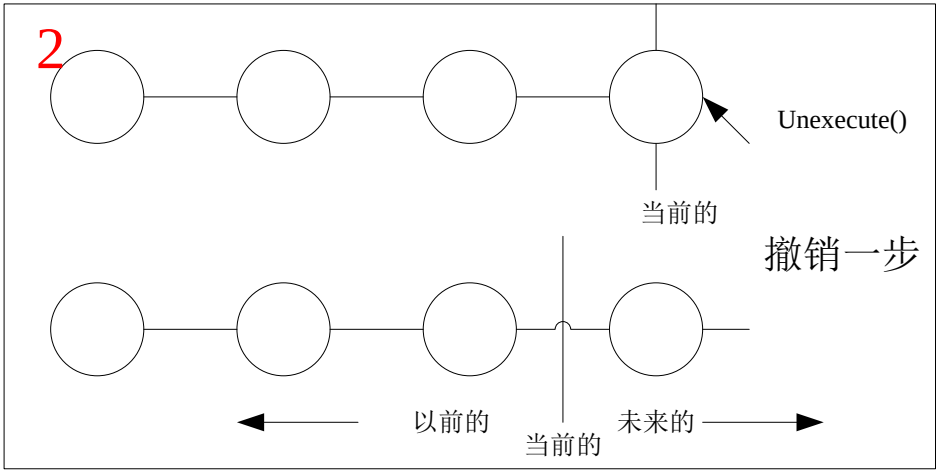
2.7.3撤销与重做

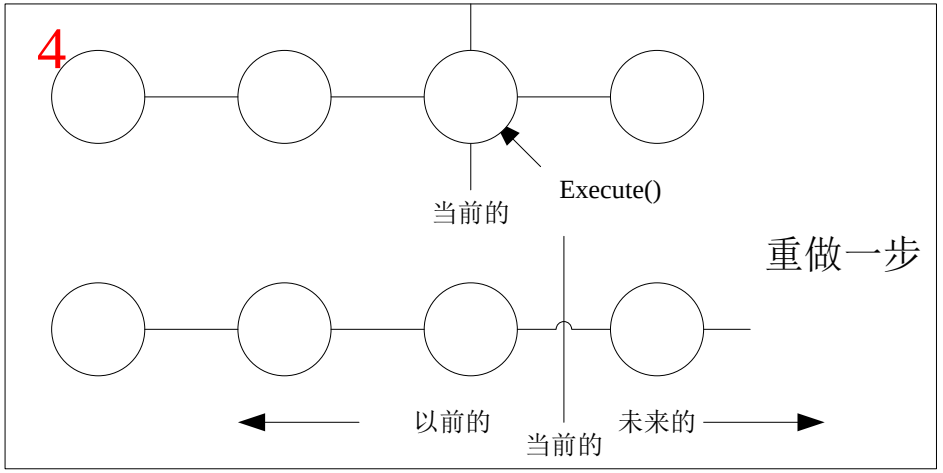
- 撤销操作
  - Command接口中增加Unexecute操作
- 判断一个命令是否可以撤销
  - Command接口中增加一个抽象的Reversible操作
  - 子类根据运行情况返回true或false

2.7.4 命令历史记录
Command History









2.7.5 Command模式

- Command模式
  - 描述了怎样封装请求，描述了一致性的发送请求的接口，允许配置客户端以处理不同请求
  - 保护了客户请求的实现
  - 基于Command接口的撤销和重做机制

2.8 拼写检查和断字处理

- 文本分析
  - 拼写错误的检查
  - 连字符号的计算
- 限制条件
  - 支持拼写检查和连字符号的计算的多种算法，便于增加新的算法
  - 避免功能与文档接口紧耦合

    1）访问需要分析的信息

    这些信息分散在文档结构的图元中

    2）分析这些信息

2.8.1 访问分散的信息处理

- 问题
  - 需要分析的文本分散在图元对象中
  - 不同图元的数据结构不唯一
  - 不同的分析算法会以不同的方式访问数据

    e.g. 前序、中序、后序

- 结论

  访问机制必须能够：

  1. 容纳不同的数据结构
  2. 支持不同的遍历方法

2.8.2 封装访问和遍历

- 解决方式1
  - 图元接口提供一个数字索引，使得客户引用子图元
  - 适合于以数组方式存储子图元的图元类，不适合使用连接表的图元类。
- 图元接口不应该偏重于某个数据结构

2.8.2 封装访问和遍历

- 解决方式2：
  - Glyph接口定义下列抽象操作

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

  - 使用示例

```
Glyph* g;
for (g->First(PREORDER); !g->IsDone(); g->Next()) {
    Glyph* current = g->GetCurrent();
    …   //处理过程
}
```

2.8.2 封装访问和遍历

- 进一步分析
  - 存在问题：
    - 要支持新的遍历方式，则需要扩展枚举值或增加新的操作
    - 即：将遍历机制完全放到Glyph类层次中，会导致变动和扩充时带来不必要的修改
  - 解决方式： 封装那些变化的概念
    - 引入Iterators对象
    - 不改变图元接口或打乱已有的图元实现

2.8.3Iterator类及其子类

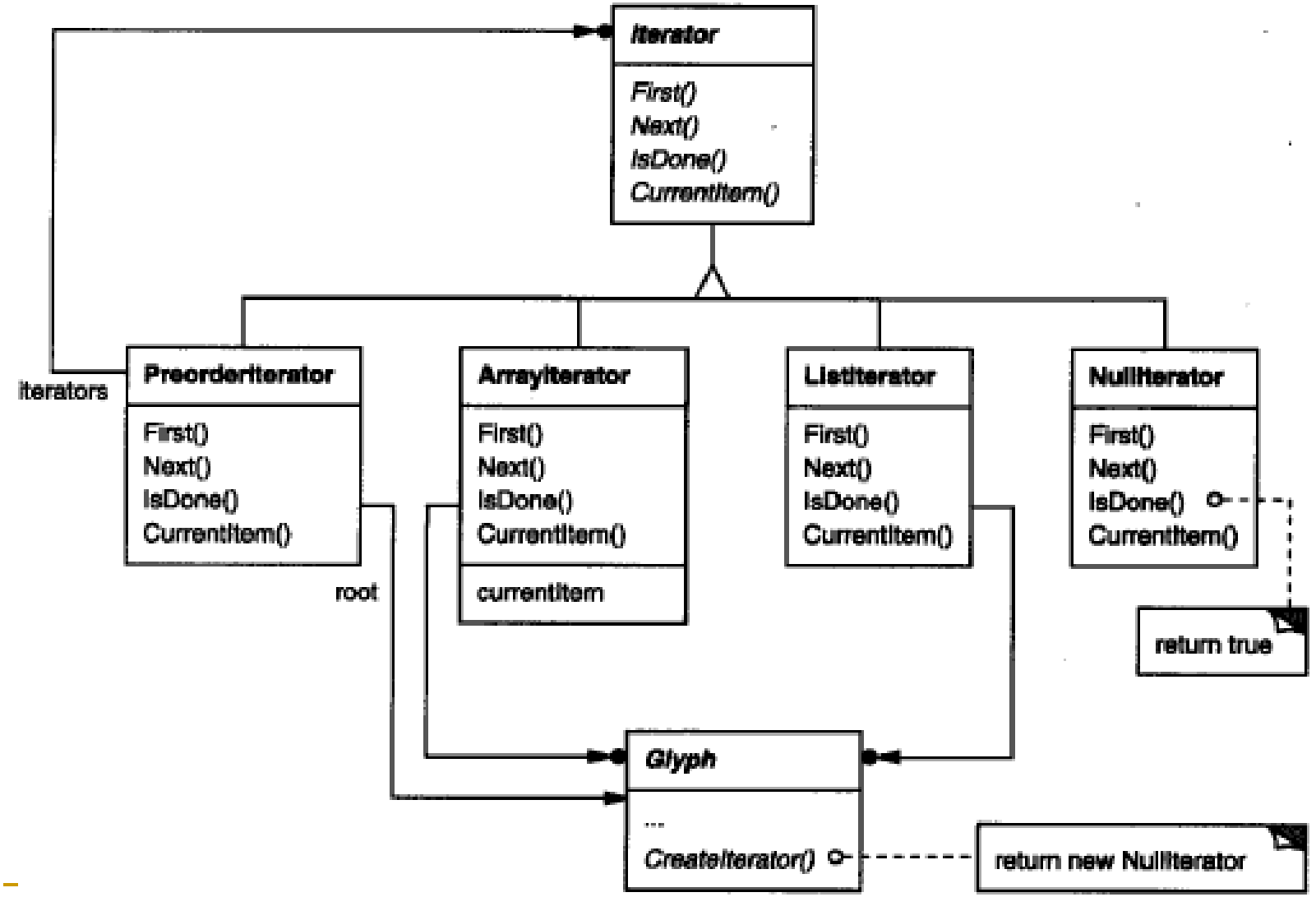


图2-13 Iterator类和它的子类

2.8.3Iterator类及其子类

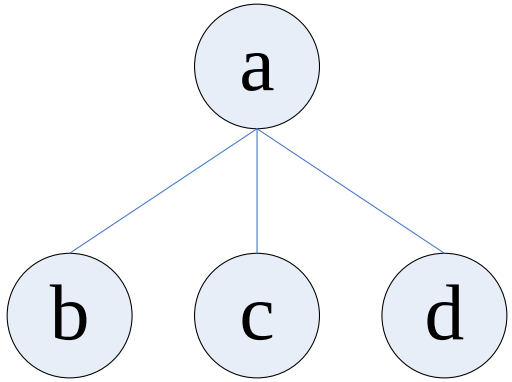
- 使用Iterator来遍历子图元

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

for ( i->First(); !i->IsDone(); i->Next()) {
    Glyph* child = i->CurrentItem();
    … // 进行处理
}
```

2.8.3Iterator类及其子类

- PreorderIterator 先序遍历　First

```
void PreorderIterator::First() {
    Iterator<Glyph*>* I = _root->CreateIterator();
    if (i) {
        i->First();
        _iterators.RemoveAll();
        _iterators.Push(i);
    }
}
```



1.对iterator_a设置First
2. iterator_a压栈

2.8.3Iterator类及其子类

- PreorderIterator 先序遍历 CurrentItem

```
//调用栈顶Iterator的CurrentItem操作
Glyph* PreorderIterator::CurrentItem() const {
    return
        _iterators.Size() > 0 ?
        _iterators.Top() -> CurrentItem() : 0;
}
```

取得iterator_a的CurrentItem (b)

2.8.3Iterator类及其子类

- PreorderIterator 先序遍历 Next

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

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

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

取得iterator_b

iterator_b设为First
压入iterator_b

弹出iterator_b

iterator_a的下一个

2.8.3Iterator类及其子类

- Iterator类层次结构
  - 允许不改变图元类而增加新的遍历方式
    - e.g PreorderIterator
- 通过使用C++模板，使得在遍历其他结构时能复用PreorderIterator机制

2.8.4 Iterator模式

- Iterator模式
  - 描述了支持访问和遍历对象结构的技术
  - 可用于组合结构，也可用于集合
  - 抽象了遍历算法，对客户隐藏了它所遍历对象的内部结构

2.8.5 遍历和遍历过程中的动作

- 问题：
  - 分析的责任放在什么位置

    1. 放在Iterator类中　　→ 不合适

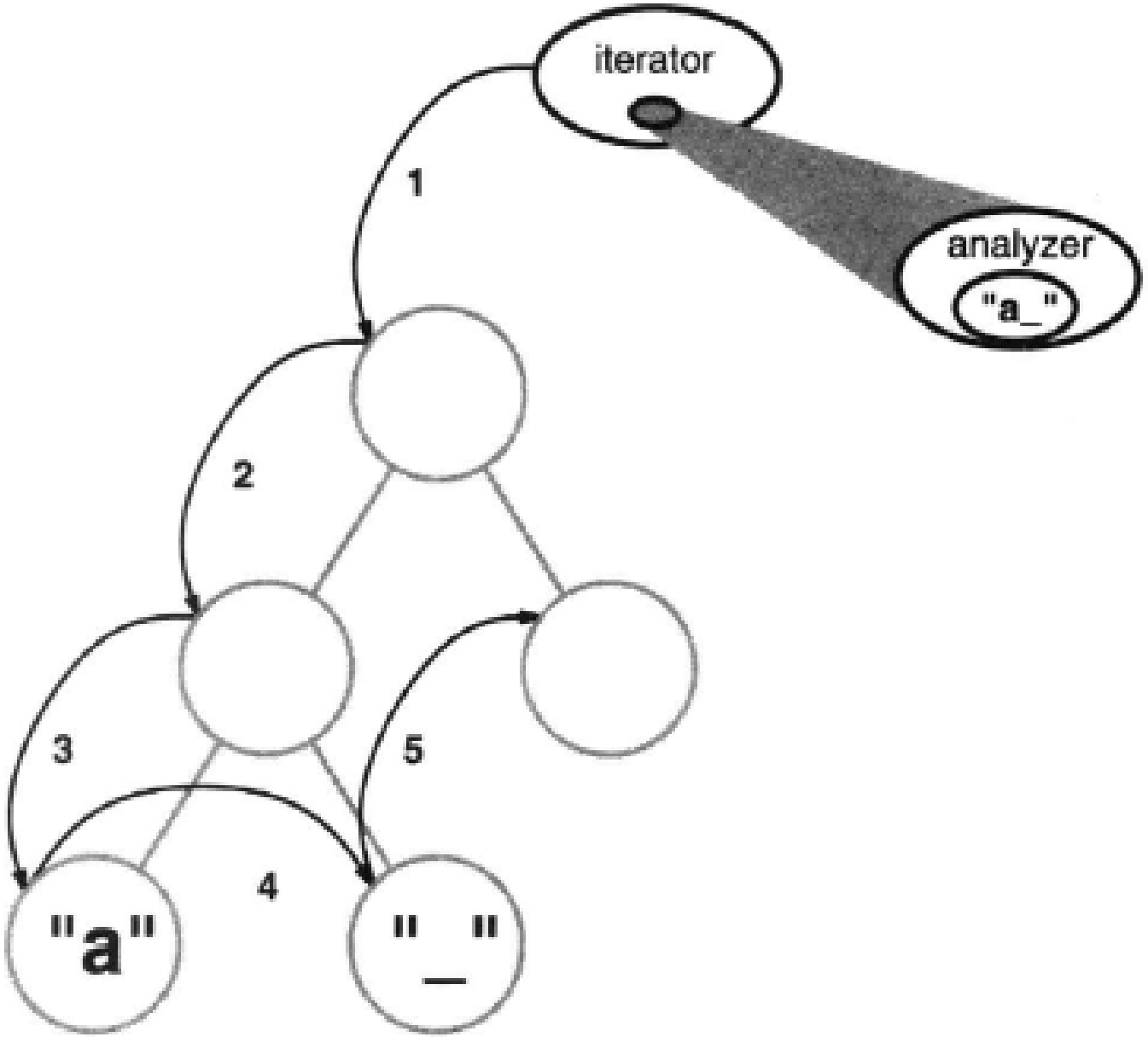
       不同的分析通常需要同样的遍历方式

    2. 将分析能力放在图元类中　→ 不合适

       功能增加的时候破坏了原有类的定义

    3. 在一个独立对象中封装分析方法

2.8.6 封装分析



2.8.6 封装分析

- 基本问题：
  - 在不使用类型测试或强制类型转换情况下也能正确对待各种不同的图元

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

    if (c = dynamic_cast<Character*>(glyph)) {
        // analyze the character

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

2.8.6 封装分析

```
//在图元子类中定义CheckMe
void GlyphSubclass::CheckMe (SpellingChecker& checker) {
    checker.CheckGlyphSubclass(this);
}
```

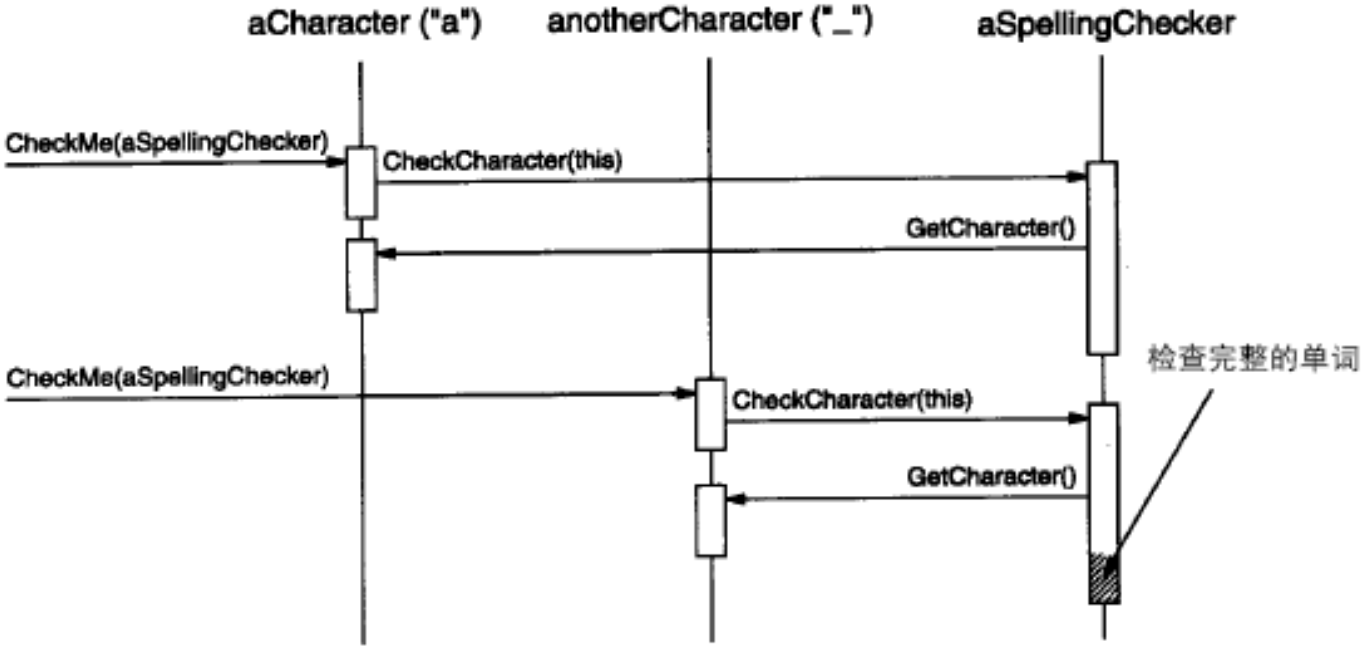
```
class SpellingChecker {
public:
    SpellingChecker();

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

    // ... and so forth

    List<char*>& GetMisspellings();

protected:
    virtual bool IsMisspelled(const char*);

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

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

    if (isalpha(ch)) {
        // append alphabetic character to _currentWord

    } else {
        // we hit a nonalphabetic character

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

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

2.8.6 封装分析

```
SpellingChecker spellingChecker;
Composition* c;

// ...

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

进行拼写检查的实际调用过程



2.8.7 Visitor类及其子类

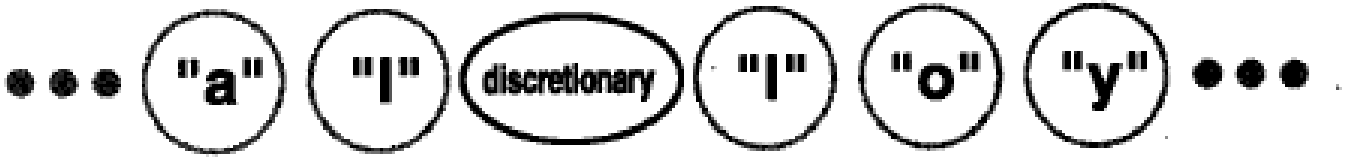
- 访问者（Visitor）
  - 遍历过程中“访问”被遍历对象并进行适当操作的对象

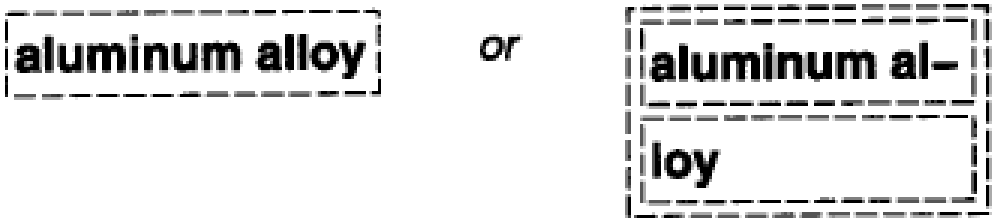
```
class Visitor {
public:

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

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

前面的SpellingChecker可以实现为Vistor的一个子类

2.8.7 Visitor类及其子类

- 连字符计算
  - HyphenationVisitor
    - 处理整个单词，在可能的连字符位置，插入一个Discretionary图元





2.8.8 Visitor模式

- Visitor模式
  - 允许对图元结构所作分析的数目不受限制地增加而不必改变图元类本身
  - 不局限使用图元结构的组合，也适用其他任何对象结构
  - Visitor所访问的类不需要一个公共父类，即Visitor可以跨越类层次结构
- 使用时注意
  - Visitor模式适合于对一个稳定类结构的对象做不同的事情
  - 即
    - 增加新的Visitor不需要改变类结构
    - 增加一个新的子类（被访问对象），则需要更新所有Visitor类接口
      - e.g. 增加一个Col子类，则Visitor接口需要加入VisitCol操作