CS10102302
设计模式

赵君峤 长聘副教授

计算机科学与技术学院

同济大学

面向对象设计原则

一个好的系统设计应该具有如下性质：

可扩展性、灵活性、可插入性。

—— Peter Code， 1999

- 可扩展性：容易添加新的功能
- 灵活性：代码修改能够平稳地发生
- 可插入性：容易将一个类抽出去，同时将另一个具有同样接口的类加进来

面向对象设计原则

- 单一职责原则
- 开放封闭原则
- Liskov替换原则
- 接口分离原则
- 依赖倒置原则

单一职责原则

单一职责原则（Single Responsibility Principle，SRP）

There should never be more than one reason for a class to change.

R. Martin, 1996

说明:

- 让一个类有且只有一种类型责任，当这个类需要承当其他类型的责任的时候，就需要分解这个类。
- 该原则可以看作是高内聚、低耦合在面向对象原则上的引申，将职责定义为引起变化的原因，以提高内聚性来减少引起变化的原因。

单一职责原则

举例：下面的接口设计有问题吗?

```
interface Modem {
    public void dial(String pno);
    public void hangup();
    public void send(Char c);
    public char recv();
}
```

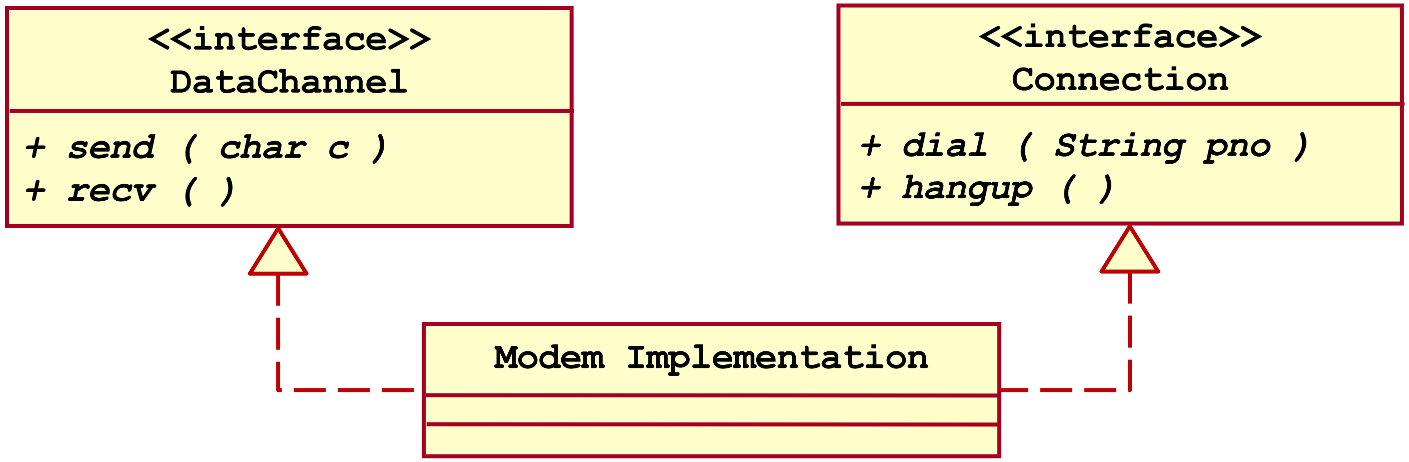
接口职责:

- 连接管理：dial 和 hangup 进行调制解调器的连接处理
- 数据通信：send 和 recv 进行数据通信

单一职责原则

Modem接口是否应该分离?

- 依赖于应用程序变化的方式，它是否影响连接函数的特征?
- 数据通信是经常变化的；连接管理是相对稳定的



单一职责原则

总结：

- 职责的划分：一个类的一个职责是引起该类变化的一个原因
- 单一职责原则分离了变与不变
  - ✓ 对象的细粒度：便于复用
  - ✓ 对象的单一性：利于稳定
- 不要创建功能齐全的类

单一职责原则是一个最简单的原则，也是最难正确应用的一个原则，因为我们会自然地将职责结合在一起。

Example

```
• class AgeCalculator {
•     private val currentYear = Calendar.getInstance().get(Calendar.YEAR)

•     fun calculate(birthYear: Int): String {
•         return (currentYear - birthYear).toString()
•     }

•     fun isValid(birthYear: Int): Boolean {
•         return currentYear > birthYear
•     }
• }
```

Correction

```
• class AgeCalculator {
•     fun calculate(birthYear: Int): String {
•         val currentYear = Calendar.getInstance().get(Calendar.YEAR)
•         return (currentYear - birthYear).toString()
•     }
• }
• class AgeValidator {
•     fun isValid(birthYear: Int): Boolean {
•         val currentYear = Calendar.getInstance().get(Calendar.YEAR)
•         return currentYear > birthYear
•     }
• }
```

开放封闭原则

开放封闭原则（Open-Closed Principle，OCP）

Software entities should be open for extension, but closed for modification.

B. Meyer, 1988 / quoted by R. Martin, 1996

- Be open for extension
  模块的行为是可扩展的，即可以改变模块的功能。
- Be closed for modification
  对模块行为进行扩展时，不需要改动源代码或二进制代码。
- 问题：怎样可以在不改动源码的情况下改变模块行为？

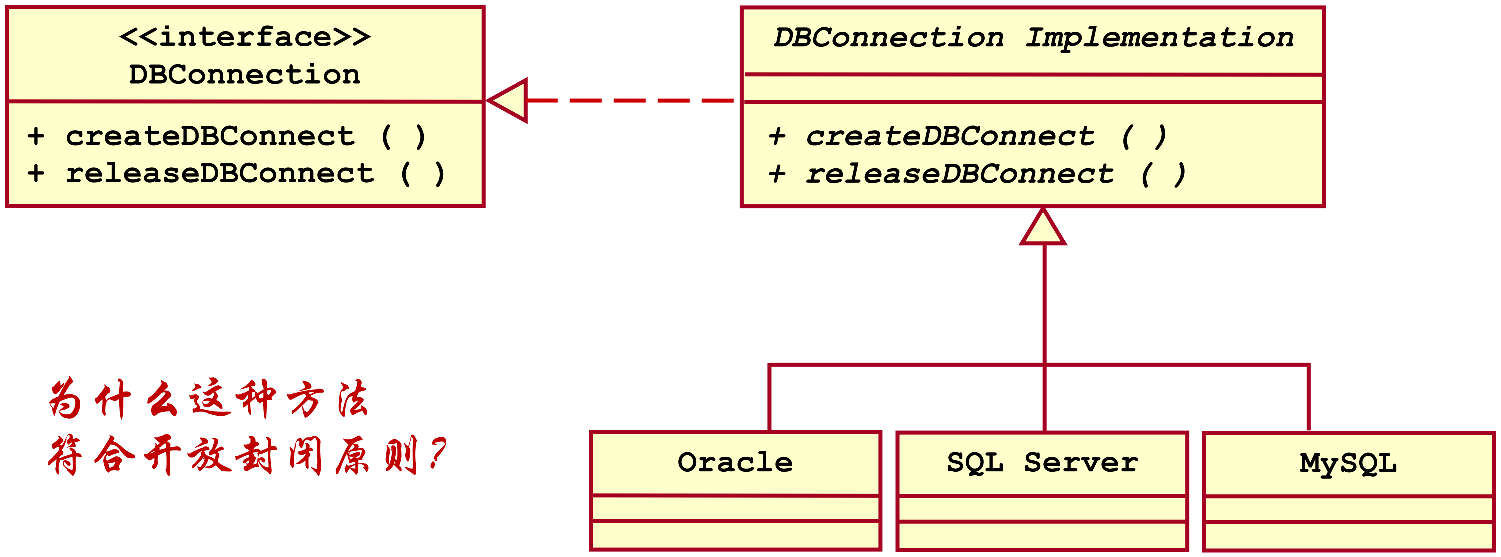
开放封闭原则

举例：如要增加新的数据库类型（如MySQL）怎么处理?

```
bool createDBConnect(...)
{
    ...
    switch (dbType)
        {
         case ORACLE:
             建立 Oracle 连接;
                 break;
         case SQLSERVER:
             建立 SQL Server 连接;
                 break;
    }
    ...
}
```

开放封闭原则

数据库连接的另一种修改方法：



为什么这种方法
符合开放封闭原则？

开放封闭原则

开放封闭原则：在设计一个软件模块（类、方法）时，应该在不修改原有代码（修改封闭）的基础上扩展其功能（扩张开放）。

实现方式

- 将那些不变的部分加以抽象成不变的接口，这些不变的接口可以应对未来的扩展；
- 接口的最小功能设计：原有的接口要么可以应对未来的扩展，不足的部分可以通过定义新的接口来实现；
- 模块之间的调用通过抽象接口来实现，即使实现层发生变化也无需修改调用方的代码。

开放封闭原则

关键是抽象

- 创建一个抽象基类，描述一组任意个可能行为，而具体的行为由派生类实现。
- 软件实体依赖于一个固定的抽象基类，这样它对于更改可以是关闭的，通过从这个抽象基类派生，也可以扩展该实体的行为。

程序不可能是100%封闭的

- 无论模块多么封闭，都会存在一些无法封闭的变化。
- 设计人员应该策略地对待这个问题，即先预测出最有可能发生变化的部分，再构造抽象来隔离这些变化。

开放封闭原则

- **开放封闭原则是面向对象设计的核心所在，遵循这个原则可以带来面向对象技术的巨大好处，即可维护性、可扩展性、可复用性和灵活性。**

- **几乎所有的设计模式都是对不同的可变性进行封装，从而使系统在不同角度上达到开放封闭原则的要求。**

- 开发人员**应该仅对程序中呈现出频繁变化的那些部分做出抽象**，然而切忌对应用程序中的每个部分都刻意地进行抽象，拒绝不成熟的抽象和抽象本身一样重要。

Example

- enum class ShoesBrand {
-     NIKE, ADIDAS
- }
- class ShoesFactory {
- fun getShoesPrice(shoesBrand: ShoesBrand): Int {
-         return which (shoesBrand) {
-             case: ShoesBrand.NIKE -> 200
-             case: ShoesBrand.ADIDAS -> 150
-         }
-     }
- }
- ShoesFactory shoes
  shoes.getShoesPrice(ShoesBrand.NIKE).toString()

Correction

- abstract class ShoesBrand {
-     abstract fun getShoesPrice(): Int
- }
- class NikeShoes : ShoesBrand() {
-     override fun getShoesPrice() = 200
- }
- class AdidasShoes : ShoesBrand() {
-     override fun getShoesPrice() = 150
- }
- ShoesBrand shoes = NikeShoes()
- shoes.getShoesPrice().toString()

Liskov替换原则

Liskov替换原则（Liskov Substitution Principle，LSP）

Inheritance should ensure that any property proved about supertype objects also holds for subtype objects.

B. Liskov, 1987

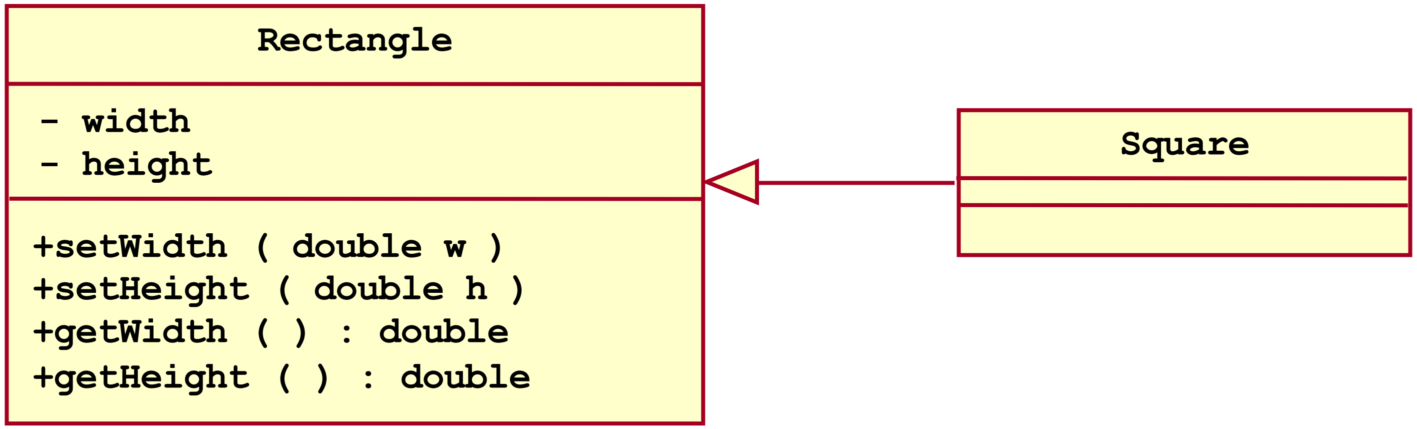
Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it.

R. Martin, 1996

理解：子类型必须能够完全替换其父类型。

Liskov替换原则

举例：正方形应该从矩形继承吗?



你是否编写过类似的程序？有什么问题吗?

Liskov替换原则

覆写 Square 类成员函数

```
void Square::setWidth(double w)
{
    Rectangle::setWidth(w);
    Rectangle::setHeight(w);
}

void Square::setHeight(double h)
{
    Rectangle::setWidth(h);
    Rectangle::setHeight(h);
}
```

考虑下面的测试函数：

```
void g(Rectangle& r)
{
    r.setWidth(5);
    r.setHeight(4);
    assert(r.Area() == 20);
}
```

为什么不对?　　原因：从行为方式来看，Square 不是 Rectangle。

Liskov替换原则

契约式设计（Design by Contract）

- 契约式设计把类及其客户之间的关系看作是一个正式的协议，明确各方的权利和义务。
- 契约是通过每个方法声明的前置条件（preconditions）和后置条件（postconditions）来指定的。
- 一个方法得以执行，其前置条件必须为真；执行完毕后，该方法应保证后置条件为真。
- 规则：**在重新声明派生类函数时，只能使用相等或者更弱的前置条件来替换原始前置条件，只能使用相等或者更强的后置条件来替换原始后置条件。**

Liskov替换原则

Square符合契约式设计规则吗?

- `Rectangle::setWidth(double w)`的后置条件
  `assert((width == w) && (height == oldHeight))`

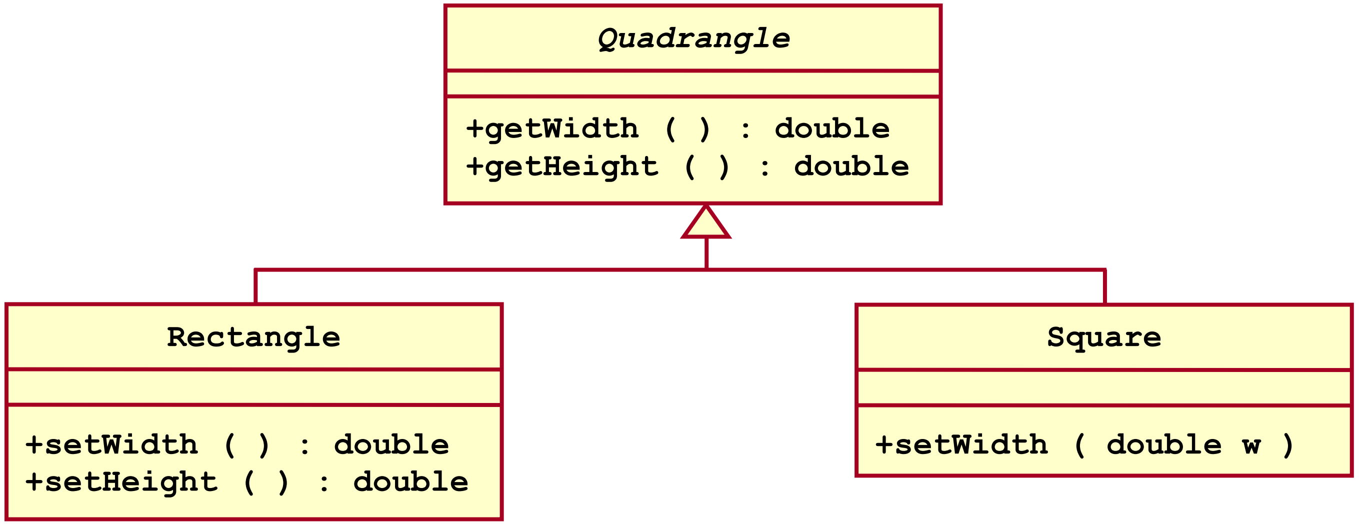
- `Square::setWidth(double w)`的后置条件
  `assert((width == w) && (height == w))`

- 由此可见，`Square::setWidth(double w)`的后置条件不能遵从`Rectangle::setWidth(double w)`后置条件的所有约束，故前者的后置条件比后者的弱。

如何解决以上问题?

Liskov替换原则

重构方法：创建一个新的抽象类C，将其作为两个具体类的超类，将A、B的共同行为移动到C中来解决问题。



Liskov替换原则

- LSP可以保证系统或子系统有良好的扩展性。只有子类能够完全替换父类，才能保证系统或子系统在运行期内识别子类接口，因而使得系统或子系统具有良好的扩展性。

- LSP有利于实现契约式编程。契约式编程有利于系统的分析和设计，即定义好系统的接口，然后在编码时实现这些接口即可。在父类里定义好子类需要实现的功能，而子类只要实现这些功能即可。

- LSP是保证OCP的重要原则。它们在实现方法上有个共同点，使用中间接口层来达到类对象的低耦合，即抽象耦合。

接口分离原则

接口分离原则（Interface Segregation Principle，ISP）

Clients should not be forced to depend upon interfaces that they do not use.

R. Martin, 1996

- 胖类会导致其客户程序之间产生不正常且有害的耦合关系。因此，客户程序应仅依赖于它们实际调用的方法。
- 将胖类的接口分解为多个特定于客户程序的接口，每个特定的接口仅声明其特定客户程序调用的那些函数，胖类则继承所有特定于客户程序的接口并实现之。

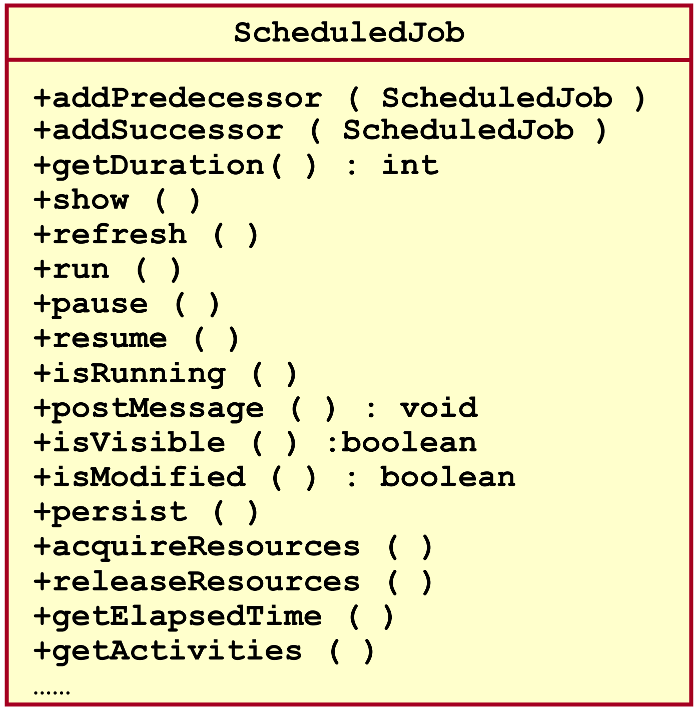
接口分离原则

举例：一个庞大的类

存在问题：

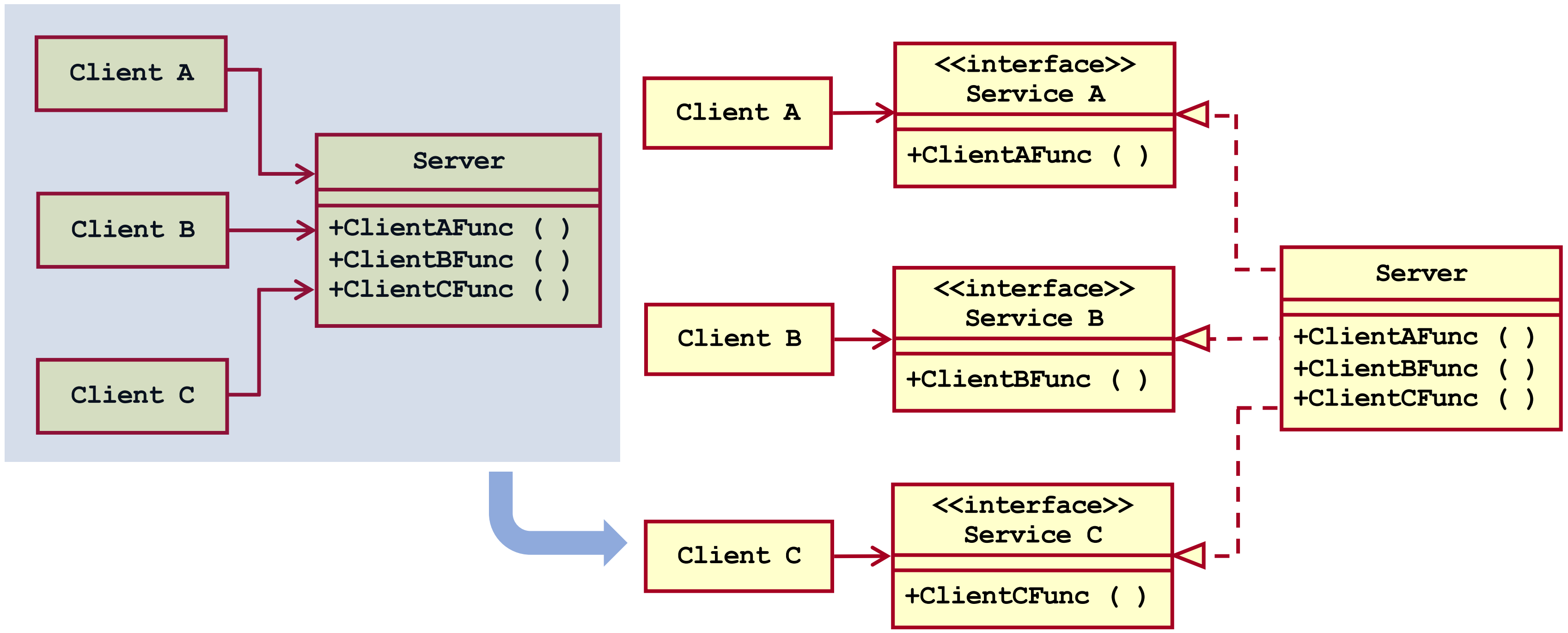
- 弄清楚需要修改什么以及修改会影响到什么是很困难的
- 可能会遇到多个开发人员同时进行不同修改的情况，这样会导致任务调度冲突
- 测试如此庞大的类是非常痛苦的

如何处理这种庞大的类？



```
ScheduledJob

+addPredecessor ( ScheduledJob )
+addSuccessor ( ScheduledJob )
+getDuration( ) : int
+show ( )
+refresh ( )
+run ( )
+pause ( )
+resume ( )
+isRunning ( )
+postMessage ( ) : void
+isVisible ( ) :boolean
+isModified ( ) : boolean
+persist ( )
+acquireResources ( )
+releaseResources ( )
+getElapsedTime ( )
+getActivities ( )
……
```

接口分离原则


Client A
Client B
Client C
Server
+ClientAFunc ( )
+ClientBFunc ( )
+ClientCFunc ( )
Client A
Client B
Client C
<<interface>>
Service A
+ClientAFunc ( )
<<interface>>
Service B
+ClientBFunc ( )
<<interface>>
Service C
+ClientCFunc ( )
Server
+ClientAFunc ( )
+ClientBFunc ( )
+ClientCFunc ( )

依赖倒置原则

常见的设计问题

- 很难添加新功能，因为每一处改动都会影响其他很多部分。
- 当对某一处进行修改，系统中看似无关的其他部分出现了问题。
- 很难在别的应用程序中重用某个模块，因为不能将它从现有的应用程序中独立提取出来。

什么原因?

- 耦合关系：“高层模块”过分依赖于“低层模块”

一个良好的设计应该是系统的每一部分都是可替换的。

依赖倒置原则

依赖倒置原则（Dependency Inversion Principle，DIP）

I. High-level modules should not depend on low-level modules.
Both should depend on abstractions.
II. Abstractions should not depend on details.
Details should depend on abstractions.

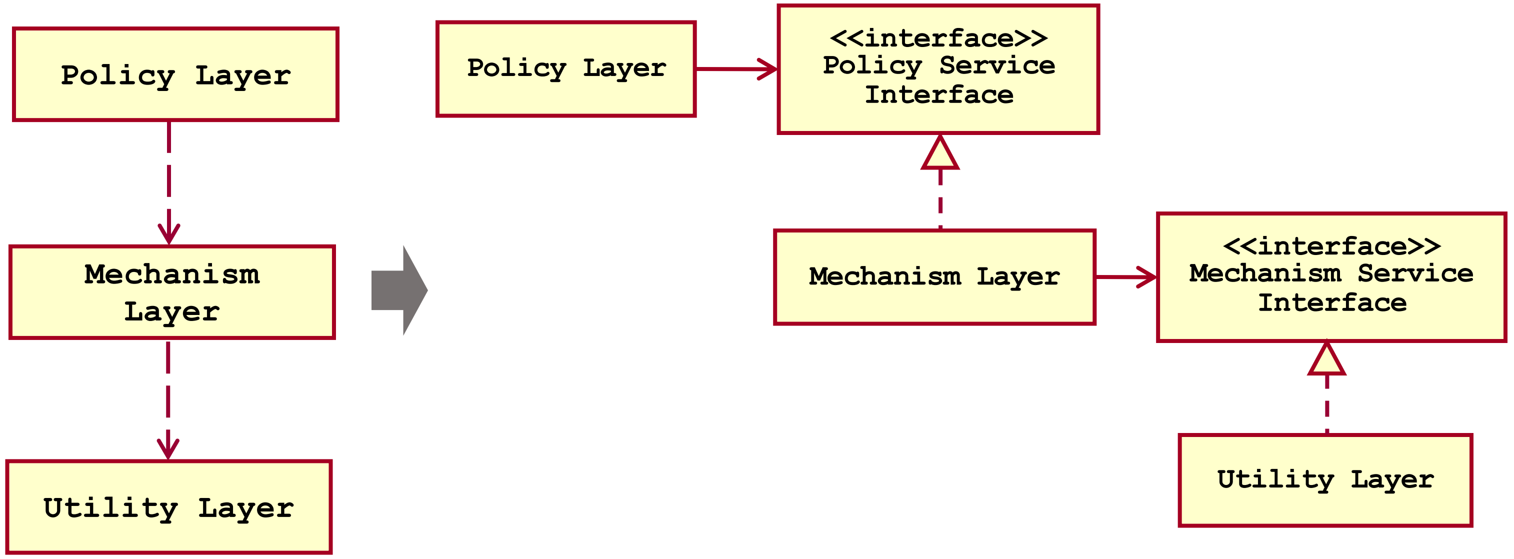
R. Martin, 1996

依赖于抽象

- 任何变量都不应该持有一个指向具体类的指针或引用
- 任何类都不应该从具体类派生
- 任何方法都不应该覆写它的任何基类中已经实现的方法

依赖倒置原则

层次化：应避免较高层次直接使用较低层次。



依赖倒置原则

总结:

- 面向接口编程，不要针对实现编程
- 抽象不应该依赖于细节，细节应该依赖于抽象
- 高层模块和低层模块以及客户端模块和服务模块等都应该依赖于接口，而不是具体实现
- 从问题的具体细节中分离出抽象，以抽象方式对类进行耦合

依赖倒置原则有利于高层模块的复用，其正确应用对于创建可重用的框架是必须的。

设计模式

回顾学过的数据结构

- Trees, Stacks, Queues
- 它们给软件开发带来了什么?

问题：在软件设计中是否存在一些可重用的解决方案?

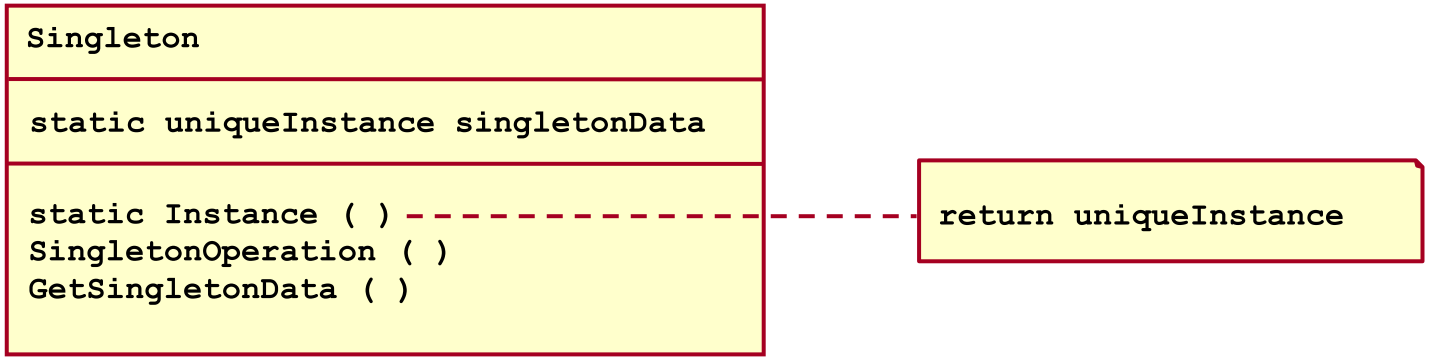
答案是肯定的

- 计模式使我们可以重用已经成功的经验
- Pattern = Documented experience

比如： 单例模式

单件模式（Singleton）

- 用于确保整个应用程序中只有一个类实例且这个实例所占资源在整个应用程序中是共享时的程序设计方法
- 适用于需要控制一个类的实例数量且调用者可以从一个公共的访问点进行访问



设计模式

设计模式描述了软件系统设计过程中常见问题的一些解决方案，它是从大量的成功实践中总结出来的且被广泛公认的实践和知识。

设计模式的好处

- 使人们可以简便地重用已有的良好设计
- 提供了一套可供开发人员交流的语言
- 提升了人们看待问题的抽象程度
- 帮助设计人员更快更好地完成系统设计
- 模式是经过考验的思想，具有更好的可靠性和扩展性

设计模式

设计模式不是万能的

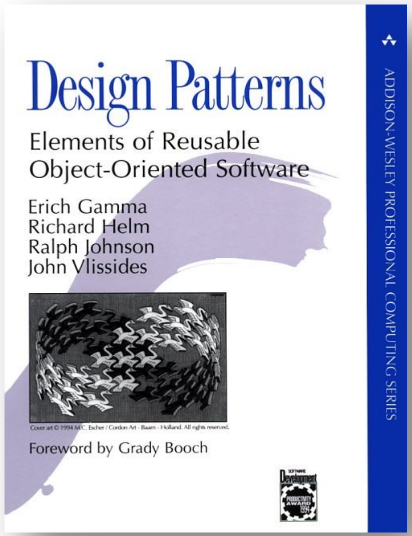
- 模式可以解决大多数问题，但不可能解决遇到的所有问题
- 应用一种模式一般会“有得有失”，切记不可盲目应用
- 滥用设计模式可能会造成过度设计，反而得不偿失

设计模式是有难度和风险的

- 一个好的设计模式是众多优秀软件设计师集体智慧的结晶
- 在设计过程中引入模式的成本是很高的
- 设计模式只适合于经验丰富的开发人员使用

Gamma 等人的设计模式

Gamma等四人提出的设计模式具有重大影响


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Gang of Four:
Gamma, Helm, Johnson, Vlissides

Gamma 等人的设计模式

创建型模式

- 创建型模式描述了实例化对象的相关技术，使用继承来改变被实例化的类，而一个对象创建型模式将实例化委托给另一个对象。

结构型模式

- 结构型模式描述了在软件系统中组织类和对象的常用方法，采用继承机制来组合接口或实现。

行为模式

- 行为模式负责分配对象的职责，使用继承机制在类件分配行为。

Gamma 等人的设计模式

创建型模式

Factory Method
Abstract Factory
Builder
Prototype
Singleton

结构性模式

Adapter
Bridge
Composite
Decorator
Façade
Flyweight
Proxy

行为模式

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

Example: A Text Editor

- Describe a text editor using patterns
  - A running example

- Introduces several important patterns

- Gives an overall flavor of pattern culture

Note: This example is from the book "Design Patterns: Elements of Reusable Object-Oriented Software", Gamma, et al. : GoF book

Text Editor Requirements

- A WYSIWYG editor (“Lexi”)
- Text and graphics can be freely mixed
- Graphical user interface
    - Toolbars, scrollbars, etc.
- Traversal operations: spell-checking, hyphenation
- Simple enough for one lecture!

Problem 1: Document Structure

A document is represented by its physical structure:

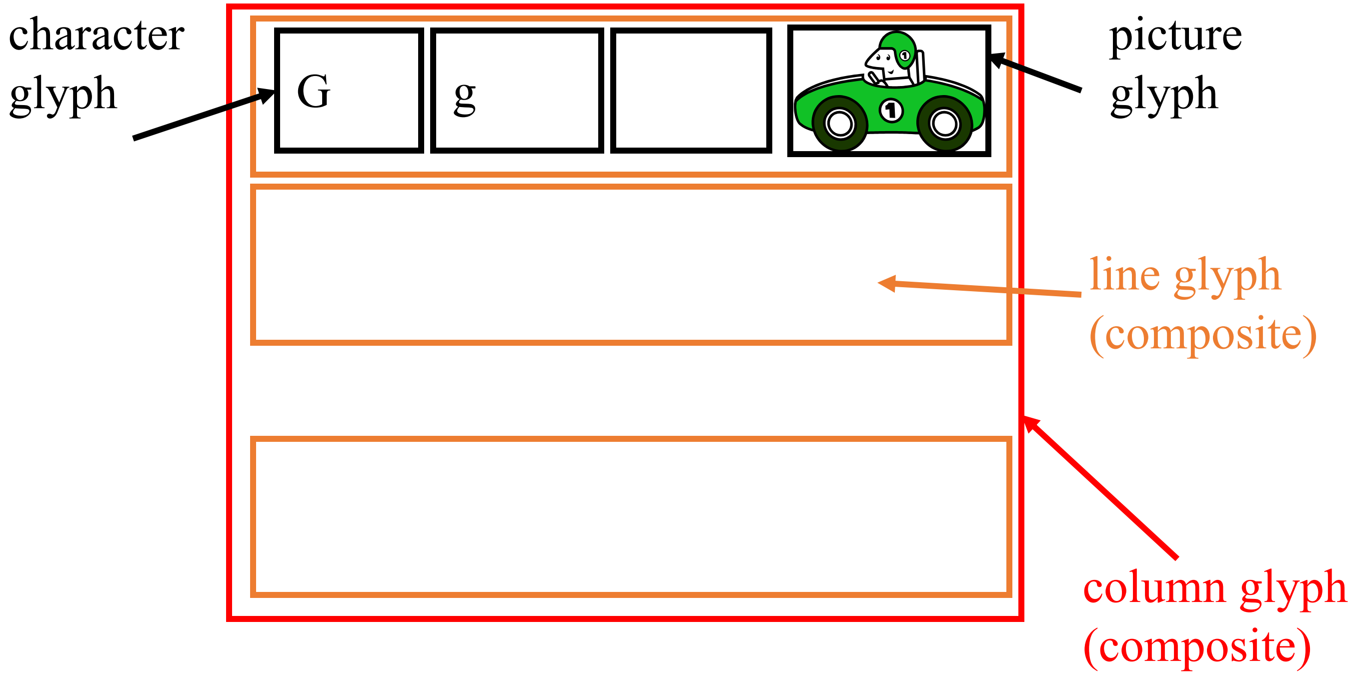
- Primitive *glyphs:* characters, rectangles, circles, pictures, . . .
- Lines: sequence of glyphs
- Columns: A sequence of lines
- Pages: A sequence of columns
- Documents: A sequence of pages

- Treat text and graphics uniformly
  - Embed text within graphics and vice versa
- No distinction between a single element or a group of elements
  - Arbitrarily complex documents

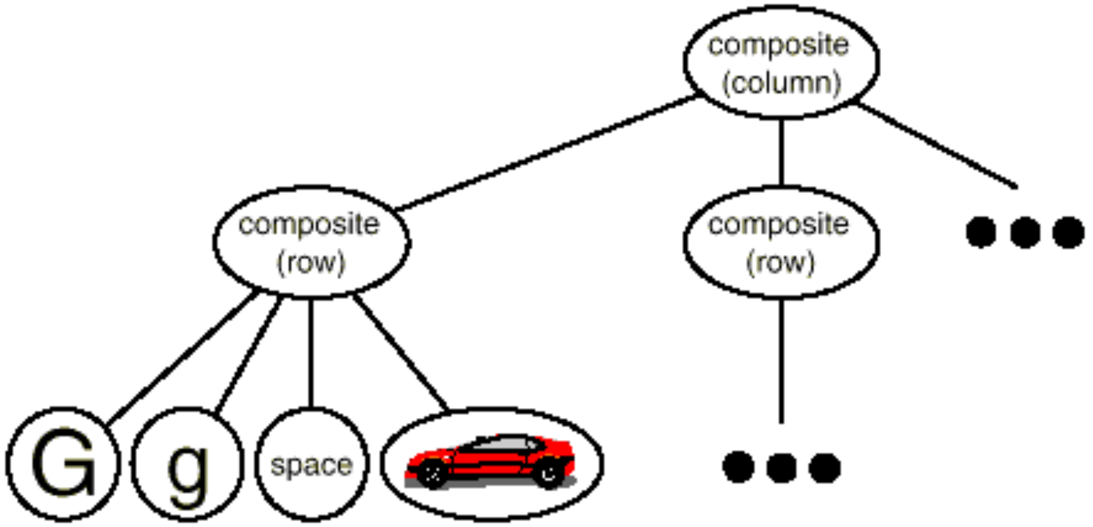
A Design

- Classes for Character, Circle, Line, Column, Page, …
  - Not so good
  - A lot of code duplication
- One (abstract) class of Glyph
  - Each element realized by a subclass of Glyph
  - All elements present the same interface
    - How to draw
    - Mouse hit detection
    - …
  - Makes extending the class easy
  - Treats all elements uniformly
- RECURSIVE COMPOSITION

Example of Hierarchical Composition



Logical Object Structure



Java Code

```
abstract class Glyph {
      List children;
      int ox, oy, width, height;
    abstract void draw();
    boolean intersects(int x,int y) {
                return (x >= ox) && (x < ox+width)
                    && (y >= oy) && (y < oy+height);
    }

    void insert(Glyph g) {
                children.add(g);
    }
}
```

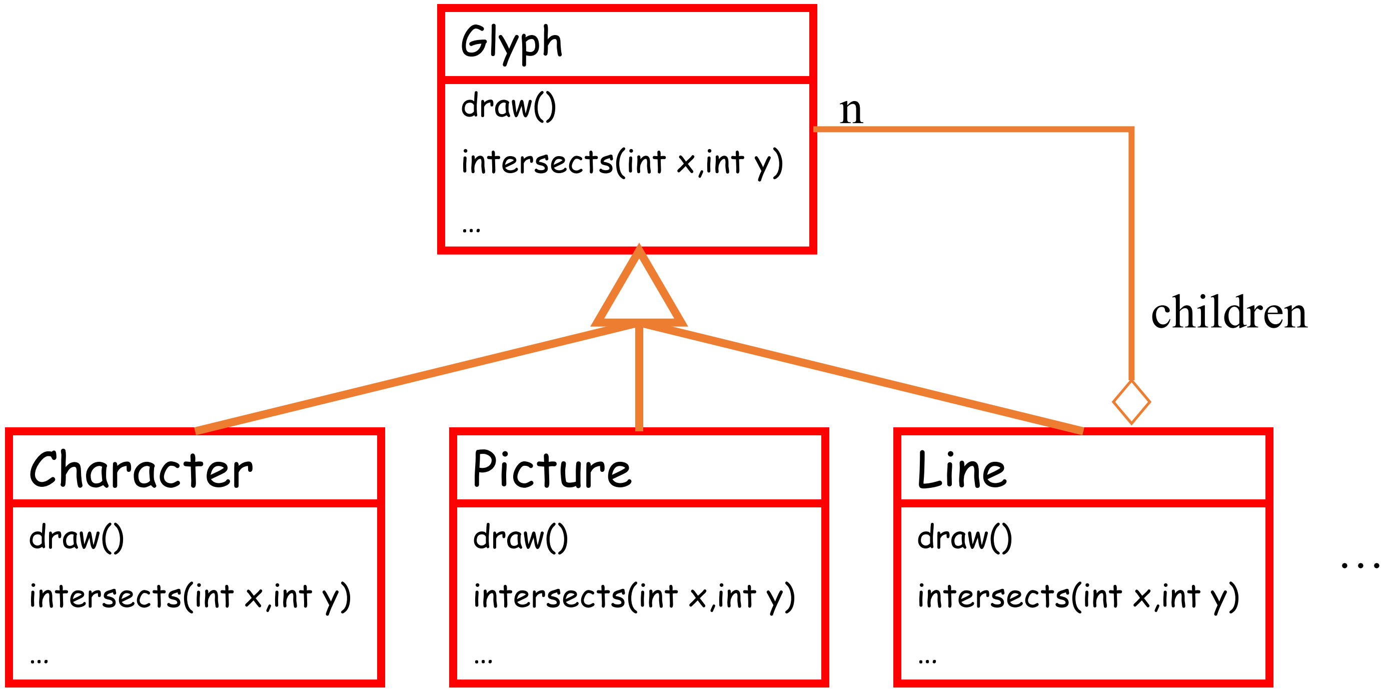
```
class Character extends Glyph {
    char c;
    // other attributes
    public Character(char c) {
                this.c = c;
                // set other attributes
    }
    void draw() {
                …
    }
    boolean intersects(int x, int y) {
                …
    }
}
```

Java Code

```
class Picture extends Glyph {
      File pictureFile;

      public Picture(File
      pictureFile){
        this.pictureFile =
      pictureFile;
      }

      void draw() {
        // draw picture
      }

}
```

```
class Line extends Glyph {
    ArrayList children;

    public Line(){
                children = new ArrayList();
    }

    void draw() {
                for (g : children)
                    g.draw();
    }

}
```

Diagram


Glyph
draw()
intersects(int x,int y)
…
n
children
Character
draw()
intersects(int x,int y)
…
Picture
draw()
intersects(int x,int y)
…
Line
draw()
intersects(int x,int y)
…
…

Composites

- This is the *composite* pattern
  - Composes objects into tree structure
  - Lets clients treat individual objects and composition of objects uniformly
  - Easier to add new kinds of components
- The GoF says you use the Composite design pattern to "**Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.**"

Problem 2: Enhancing the User Interface

- We will want to decorate elements of the UI
  - Add borders
  - Scrollbars
  - Etc.

- How do we incorporate this into the physical structure?

A Design

- Object behavior can be extended using inheritance
- Not so good
  - Major drawback: inheritance structure is static
- Subclass elements of Glyph
  - BorderedComposition
  - ScrolledComposition
  - BorderedAndScrolledComposition
  - ScrolledAndBorderedComposition
  - …
- Leads to an explosion of classes

Decorators

- Want to have a number of decorations (e.g., Border, ScrollBar, Menu) that we can mix independently

```
x = new ScrollBar(new Border(new Character(c)))
```

  - We have n decorators and 2^n combinations

Transparent Enclosure

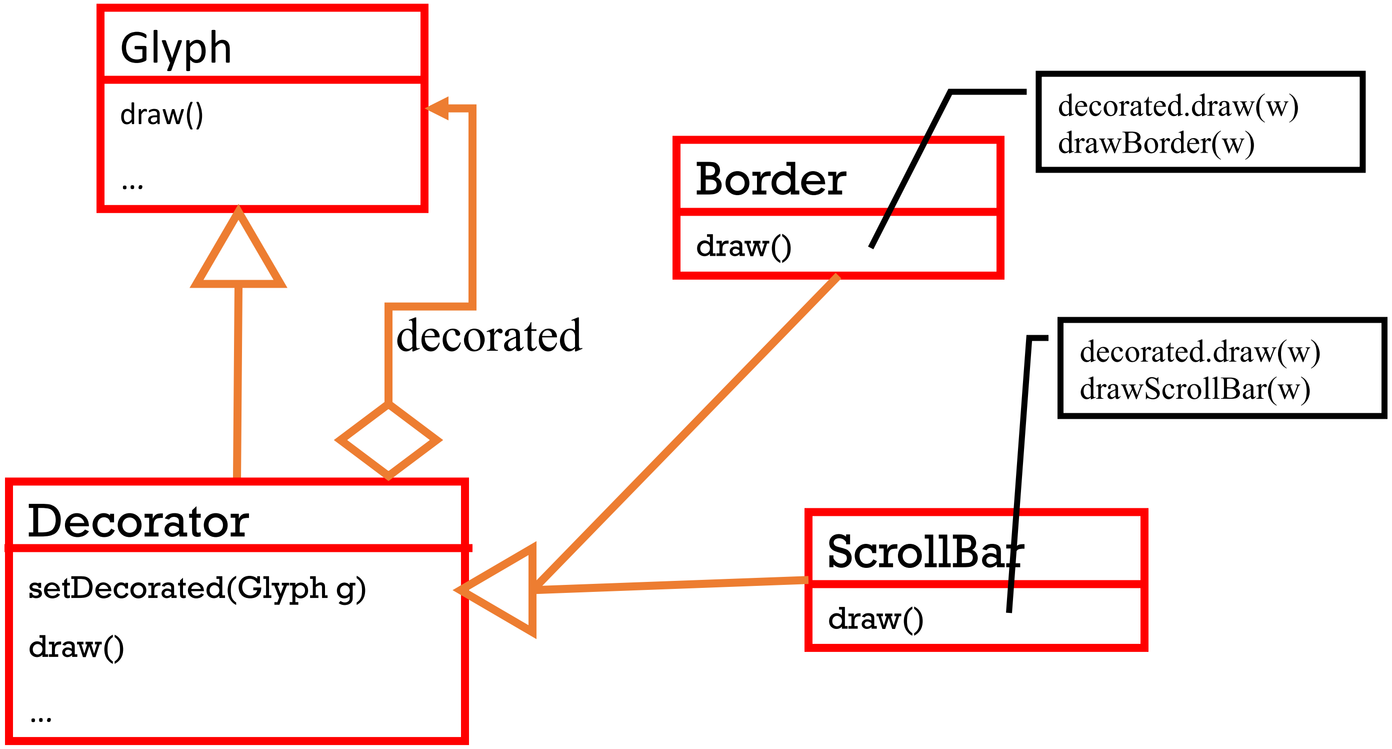
- Define Decorator
    - Implements Glyph
    - Has one member decorated of type Glyph
    - Border, ScrollBar, Menu extend Decorator

Java Code

```
abstract class Decorator extends
    Glyph {
    Glyph decorated;

    void setDecorated(Glyph d) {
            decorated = d;
    }
}
```

```
class ScrollBar extends Decorator {
    public ScrollBar(Glyph decorated) {
            setDecorated(decorated);
            …
    }

    void draw() {
            decorated.draw();
            drawScrollBar();
    }

    void drawScrollBar(){
            // draw scroll bar
    }
}
```

Diagram



Glyph
draw()
...
decorated
Decorator
setDecorated(Glyph g)
draw()
...
Border
draw()
decorated.draw(w)
drawBorder(w)
ScrollBar
draw()
decorated.draw(w)
drawScrollBar(w)

Decorators

- This is the *decorator* pattern
- The formal definition of the Decorator pattern from the GoF book says you can, "**Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality**."
- A way of adding responsibilities to an object
- Commonly extending a composite
  - As in this example

Problem 3: Supporting Look-and-Feel Standards

- Different look-and-feel standards
  - Appearance of scrollbars, menus, etc.

- We want the editor to support them all
  - What do we write in code like

```
ScrollBar* scr = new ?
```

Possible Designs

```
ScrollBar scr;
if (style == MOTIF)
   scr = new MotifScrollBar()
else if (style == MacScrollBar)
   scr = new MacScrollBar()
else if (style == …)
      ….
```

- will have similar conditionals for menus, borders, etc.

Abstract Object Creation

- **Encapsulate what varies in a class**
- Here object creation varies
  - Want to create different menu, scrollbar, etc
  - Depending on current look-and-feel

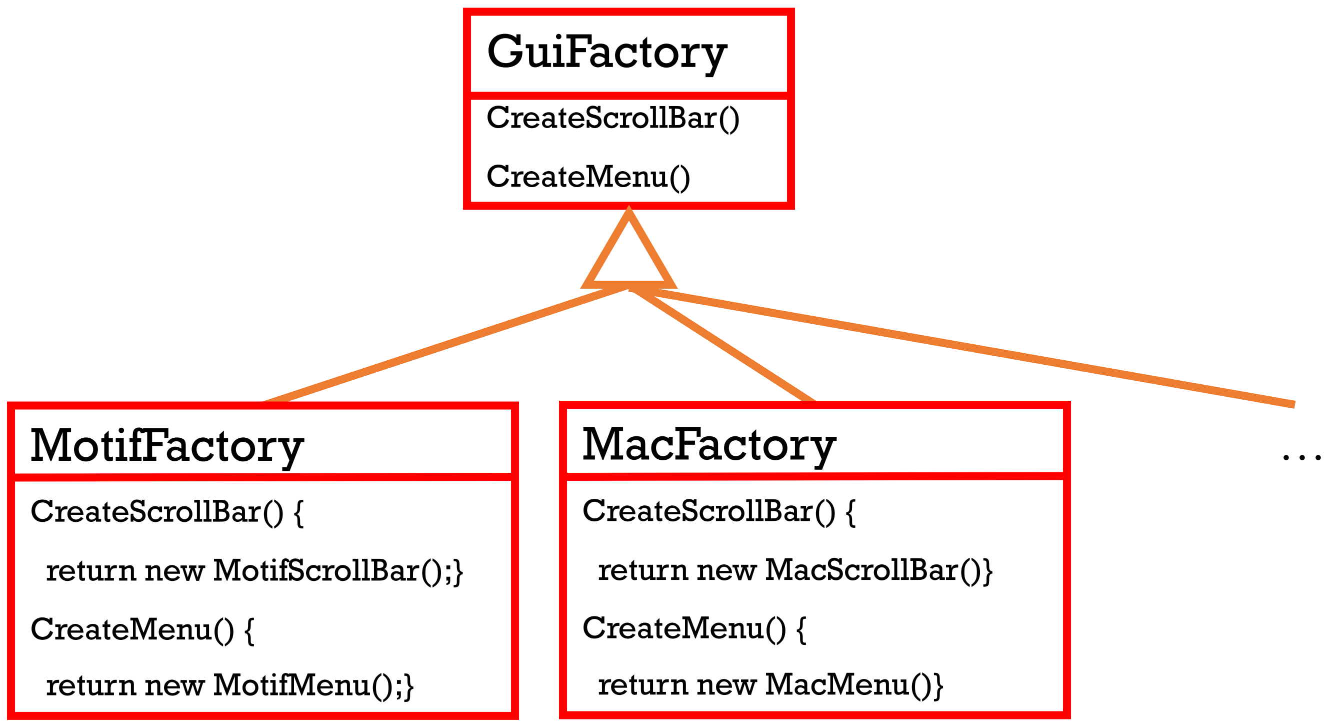
- Define a GUIFactory class
  - One method to create each look-and-feel dependent object
  - One GUIFactory object for each look-and-feel
  - Created itself using conditionals

Java Code

```
abstract class GuiFactory {
    abstract ScrollBar CreateScrollBar();
    abstract Menu CreateMenu();
    …
}
class MotifFactory extends GuiFactory
    {
    ScrollBar CreateScrollBar() {
                return new
    MotifScrollBar();
    }
    Menu CreateMenu() {
                return new MotifMenu();
    }
}
```

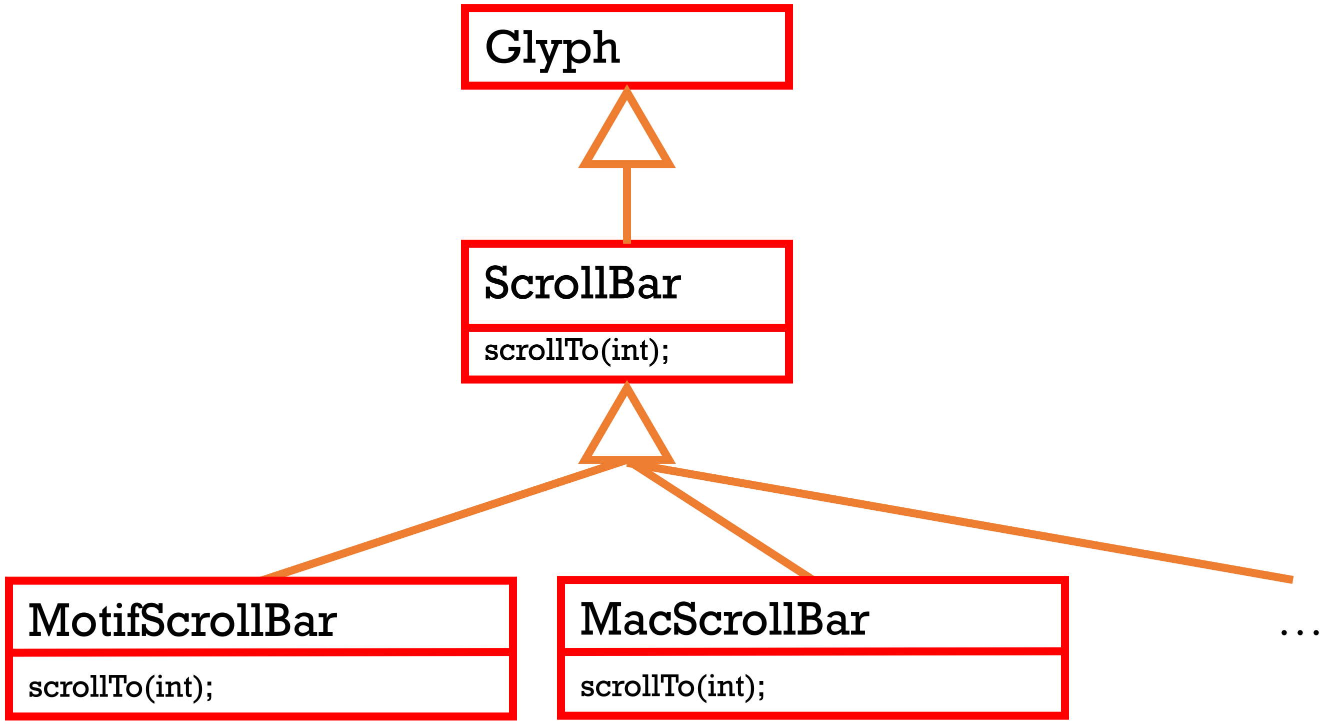
```
GuiFactory factory;
if (style==MOTIF)
    factory = new MotifFactory();
else if (style==MAC)
    factory = new MacFactory();
else if (style==…)
    …


ScrollBar scr =
    factory.CreateScrollBar();
```

Diagram



GuiFactory
CreateScrollBar()
CreateMenu()
MotifFactory
CreateScrollBar() {
return new MotifScrollBar();}
CreateMenu() {
return new MotifMenu();}
MacFactory
CreateScrollBar() {
return new MacScrollBar()}
CreateMenu() {
return new MacMenu()}
. . .

Abstract Products



Factories

- This is the *abstract factory* pattern
- According to the GoF book, the Factory Method design pattern should "**Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory method lets a class defer instantiation to subclasses**."
- A class which
  - Abstracts the creation of a family of objects
  - Different instances provide alternative implementations of that family
- Note
  - The "current" factory is still a global variable
  - The factory can be changed even at runtime

Problem 4: Spell-Checking

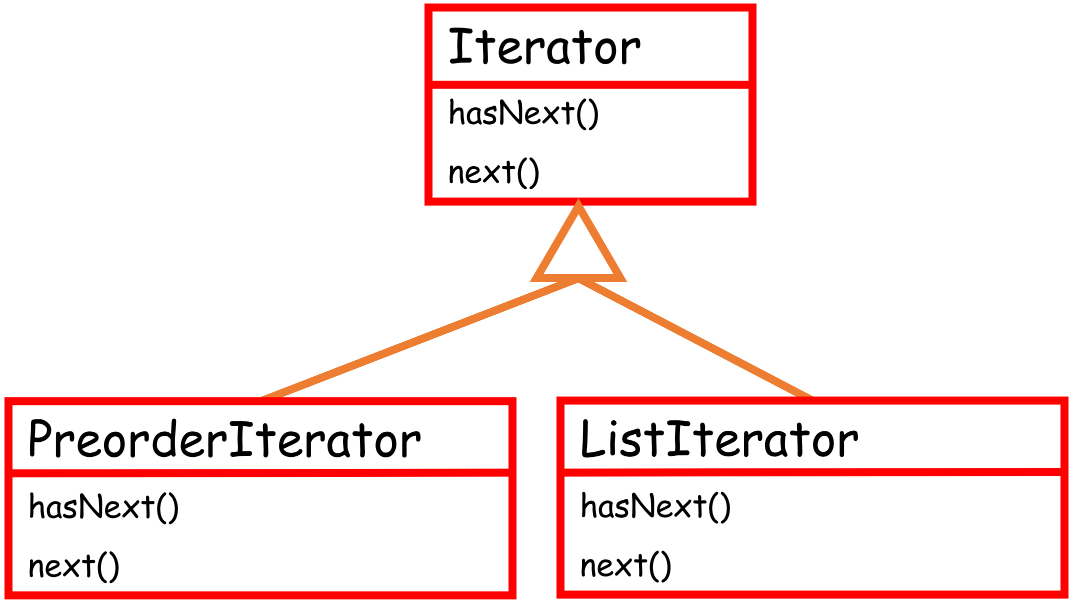
- Considerations
  - Spell-checking requires traversing the document
    - Need to see every glyph, in order
    - Information we need is scattered all over the document
  - There may be other analyses we want to perform
    - E.g., grammar analysis

One Possibility

- Iterators
    - Hide the structure of a container from clients
    - A method for
        - pointing to the first element
        - advancing to the next element and getting the current element
        - testing for termination

```
Iterator i = composition.getIterator();
while (i.hasNext()) {
   Glyph g = i.next();
   do something with Glyph  g;
}
```

Diagram


Iterator
hasNext()
next()
PreorderIterator
hasNext()
next()
ListIterator
hasNext()
next()

Notes

- Iterators work well if we don’t need to know the type of the elements being iterated over
  - E.g., send kill message to all processes in a queue
- Not a good fit for spell-checking
  - Uglyx
  - Change body whenever the class hierarchy of Glyph changes

```
Iterator i = composition.getIterator();
while (i.hasNext()) {
   Glyph g = i.next();
   if (g instanceof Character) {
            // analyze the character
   } else if (g instanceof Line) {
            // prepare to analyze children
   of
            // row
   } else if (g instanceof Picture) {
            // do nothing
   } else if (…) …
}
```

Visitors

- The visitor pattern is more general
  - Iterators provide traversal of containers
  - Visitors allow
    - Traversal
    - <u>And type-specific actions</u>

- The idea
  - Separate traversal from the action
  - **Have a “do it” method for each element type**
    - Can be overridden in a particular traversal

Java Code

```
abstract class Glyph {
    abstract void accept(Visitor vis);
    …
}

class Character extends Glyph {
    …
    void accept(Visitor vis) {
            vis.visitChar (this);
    }
}

class Line extends Glyph {
    …
    void accept(Visitor vis) {
            vis.visitLine(this);
    }
}
```

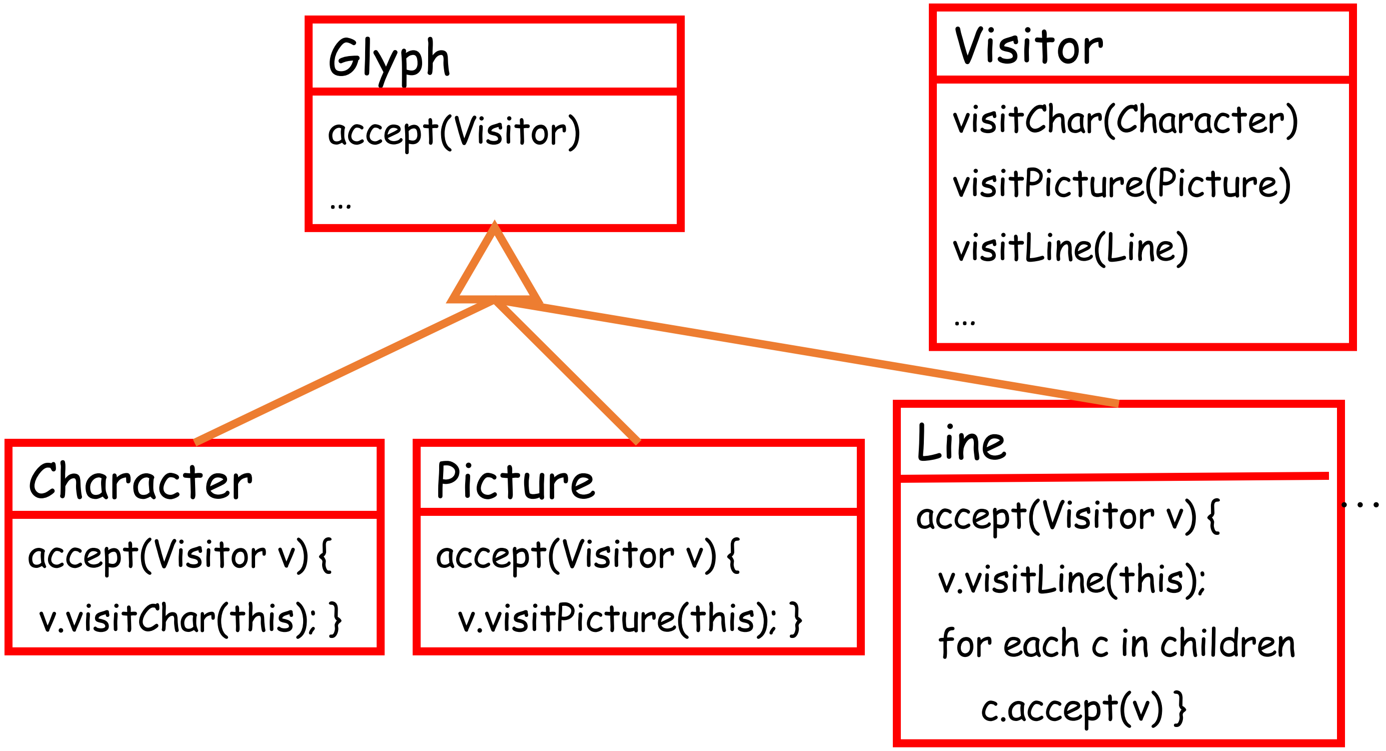
```
abstract class Visitor {
    abstract void visitChar (Character c);
    abstract void visitLine(Line l);
    abstract void visitPicture(Picture p);
    …
}

class SpellChecker extends Visitor {
    void visitChar (Character c) {
            // analyze character}
    void visitLine(Line l) {
            // process children }
    void visitPicture(Picture p) {
            // do nothing }
    …
}
```

Java Code

```
SpellChecker checker = new
      SpellChecker();
Iterator i =
      composition.getIterator();
while (i.hasNext()) {
      Glyph g = i.next();
      g.accept(checker);
}
```

```
abstract class Visitor {
    abstract void visitChar (Character c);
    abstract void visitLine(Line l);
    abstract void visitPicture(Picture p);
    …
}

class SpellChecker extends Visitor {
    void visitChar (Character c) {
                // analyze character}
    void visitLine(Line l) {
                // process children }
    void visitPicture(Picture p) {
                // do nothing }
    …
}
```

Diagram


Glyph
accept(Visitor)
…
Visitor
visitChar(Character)
visitPicture(Picture)
visitLine(Line)
…
Character
accept(Visitor v) {
v.visitChar(this); }
Picture
accept(Visitor v) {
v.visitPicture(this); }
Line
accept(Visitor v) {
v.visitLine(this);
for each c in children
c.accept(v) }
…

Visitor Pattern

- According to the GoF book, the Visitor design pattern should “**Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.**”
- Semantic analysis of an abstract syntax tree

Problem 5: Formatting

- A particular physical structure for a document
  - Decisions about layout
  - Must deal with e.g., line breaking

- Design issues
  - Layout is complicated
  - No best algorithm
    - Many alternatives, simple to complex

Formatting Examples

We've settled on a way to represent the document's physical structure. Next, we need to figure out how to construct a particular physical structure, one that corresponds to a properly formatted document.

We've settled on a way to represent the document's physical structure. Next, we need to figure out how to construct a particular physical structure, one that corresponds to a properly formatted document.

A Design

- Add a *format* method to each Glyph class
- Not so good
- Problems
  - Can’t modify the algorithm without modifying Glyph
  - Can’t easily add new formatting algorithms

The Core Issue

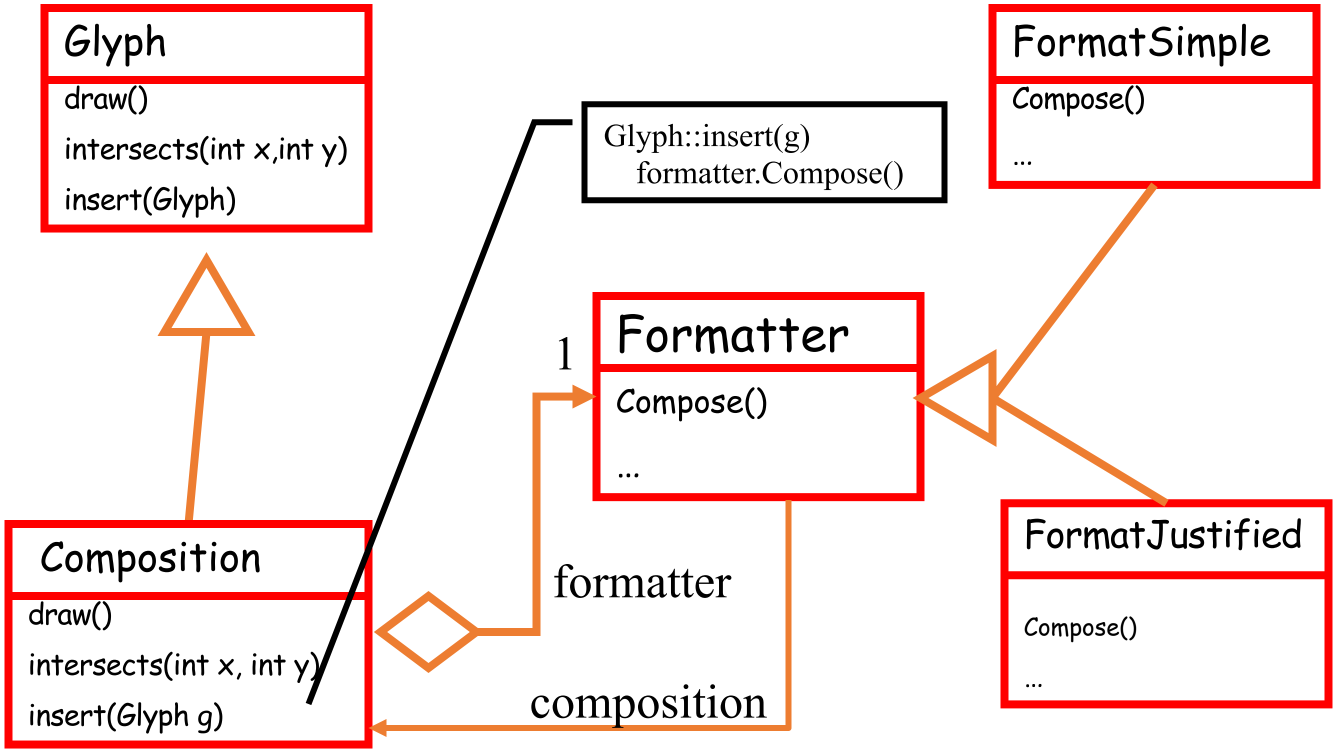
- Formatting is complex
  - We don’t want that complexity to pollute Glyph
  - We may want to change the formatting method
- Encapsulate formatting behind an interface
  - Each formatting algorithm an instance
  - Glyph only deals with the interface

Java Code

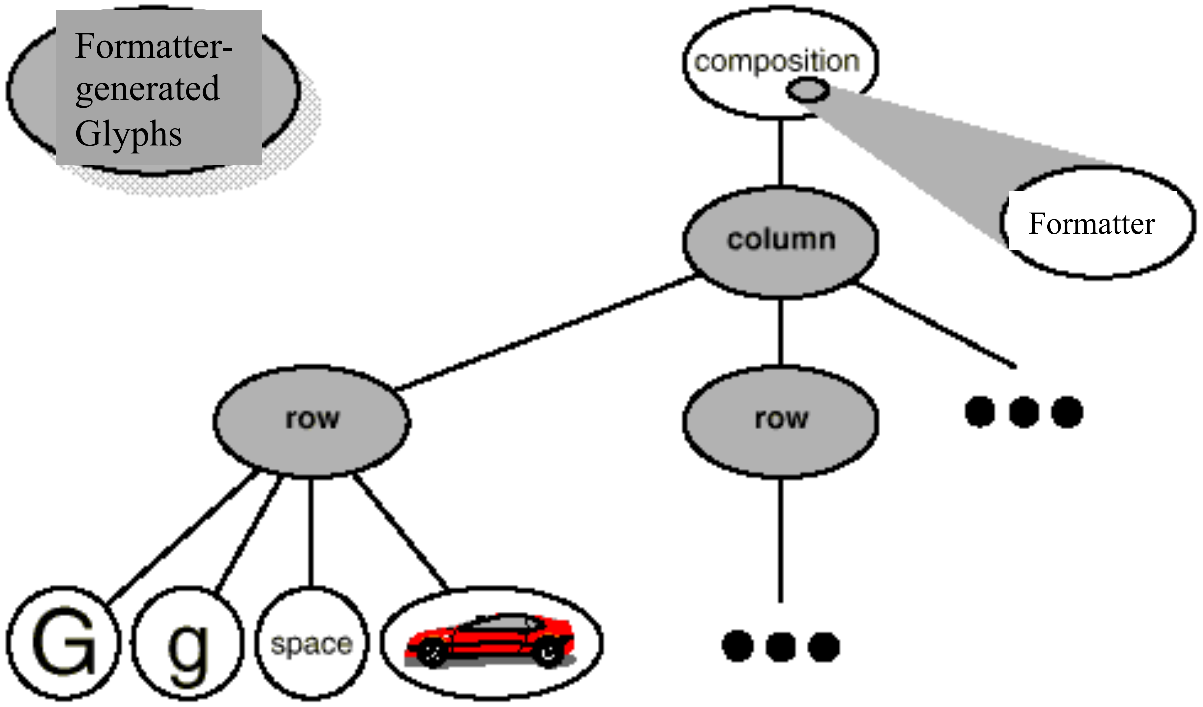
```
abstract class Composition extends Glyph {
    Formatter formatter;

    void setFormatter(Formatter f){
                formatter = f;
                formatter.setComposition(this);
    }

    void insert(Glyph g) {
                children.add(g);
                formatter.Compose();
    }
}
```

```
abstract class Formatter  {
    Composition composition

    void setComposition(Composition c){
                composition = c;
    }

    abstract void Compose();
}

class FormatSimple extends Formatter {
    void Compose() {
      // implement your formatting algorithm
    }
}
```

Diagram


Glyph
draw()
intersects(int x,int y)
insert(Glyph)
Glyph::insert(g)
formatter.Compose()
FormatSimple
Compose()
...
Formatter
Compose()
...
1
FormatJustified
Compose()
...
Composition
draw()
intersects(int x, int y)
insert(Glyph g)
formatter
composition

Formatter


Formatter-generated Glyphs
composition
Formatter
column
row
row
G
g
space

Strategies

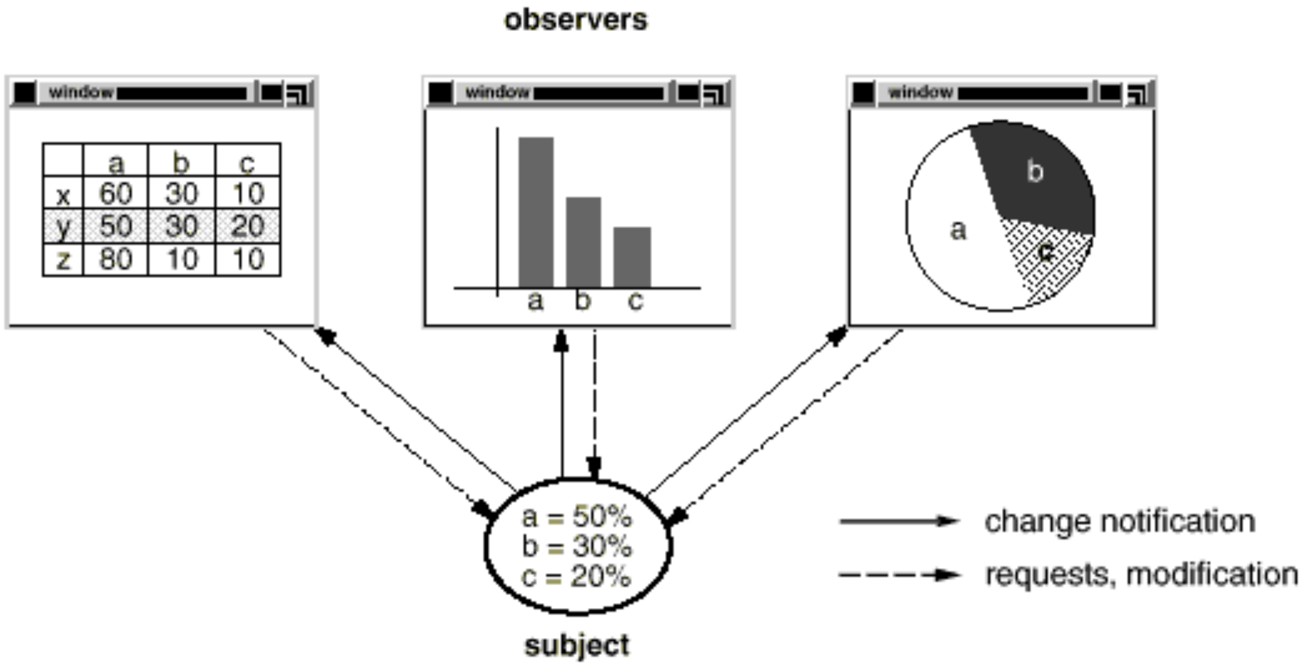
- This is the *strategy* pattern
  - Isolates variations in algorithms we might use
  - Formatter is the strategy, Composition is context
- The GoF book says the Strategy design pattern should: "**Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.**"
- General principle

encapsulate variation

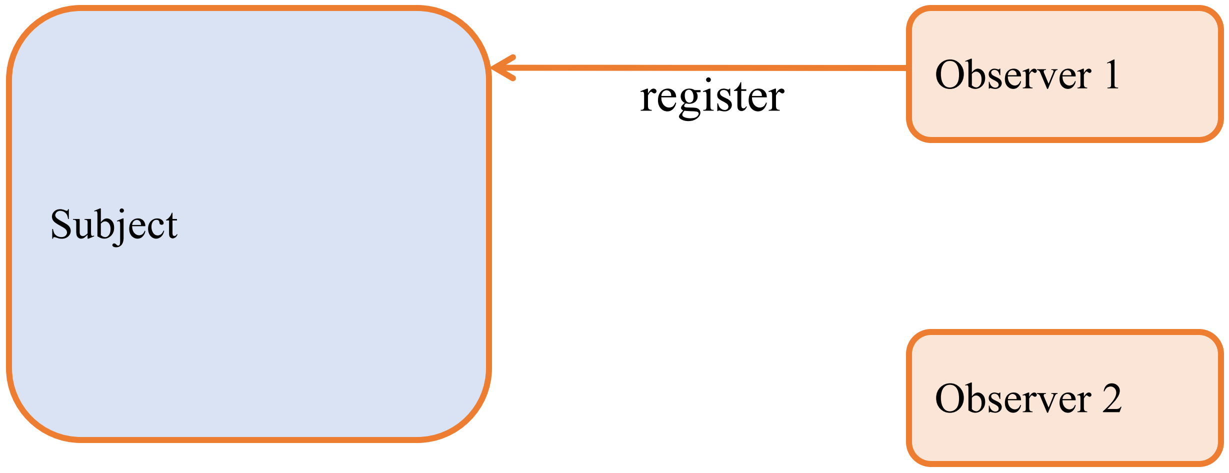
- In OO languages, this means defining abstract classes for things that are likely to change

Problem: one-to-many-dependency

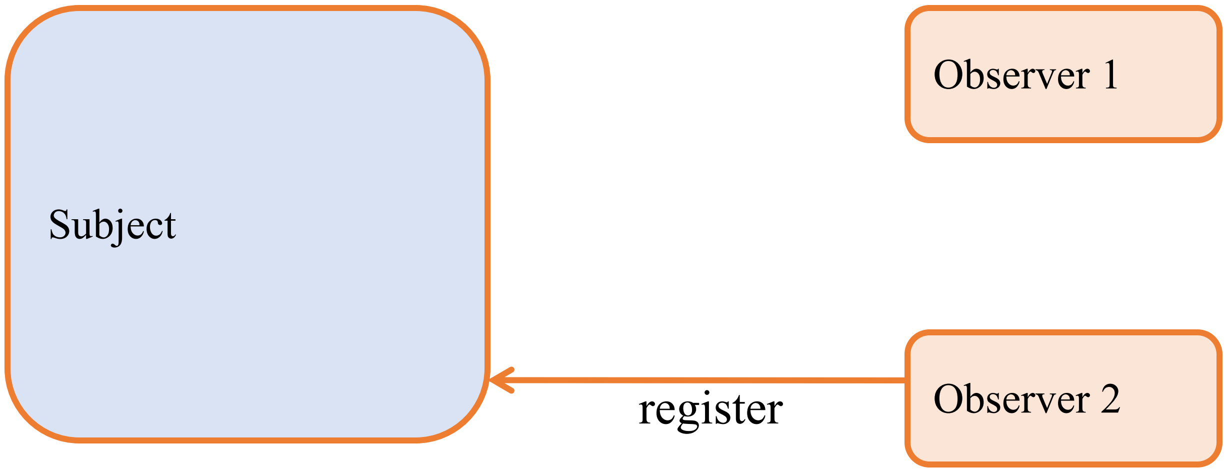
- Many objects are dependent on object o
- If o changes state, notify and update all dependent objects



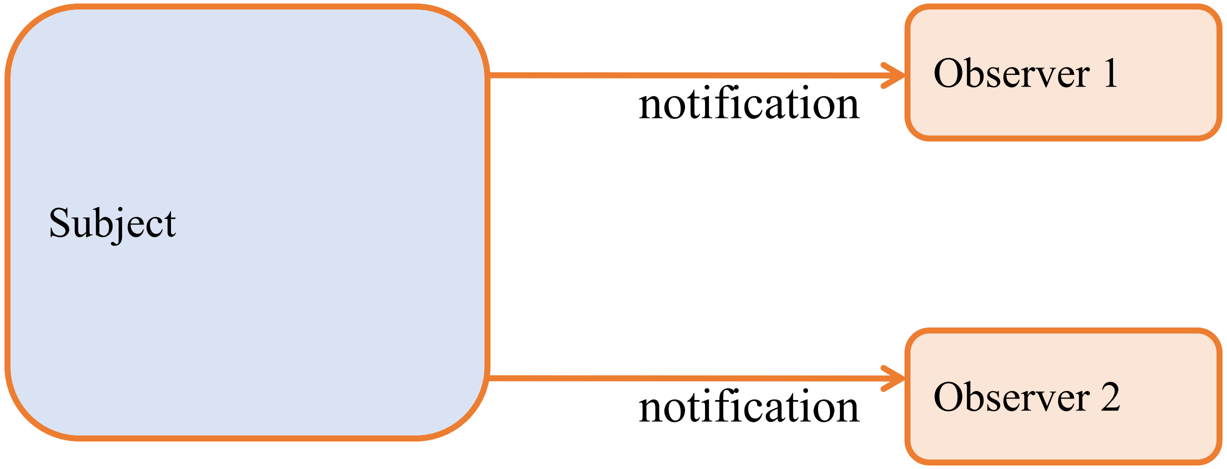
Observer Pattern



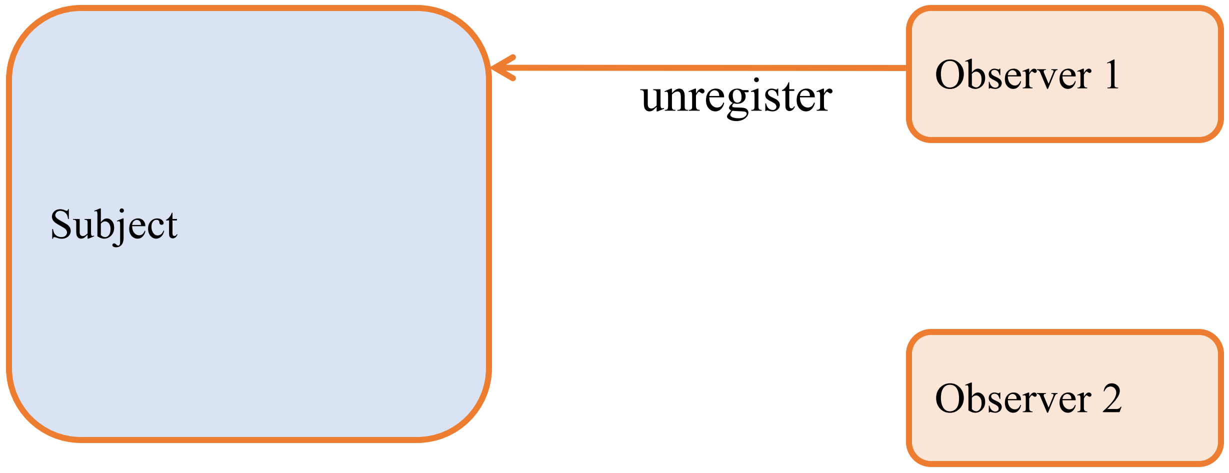
Observer Pattern



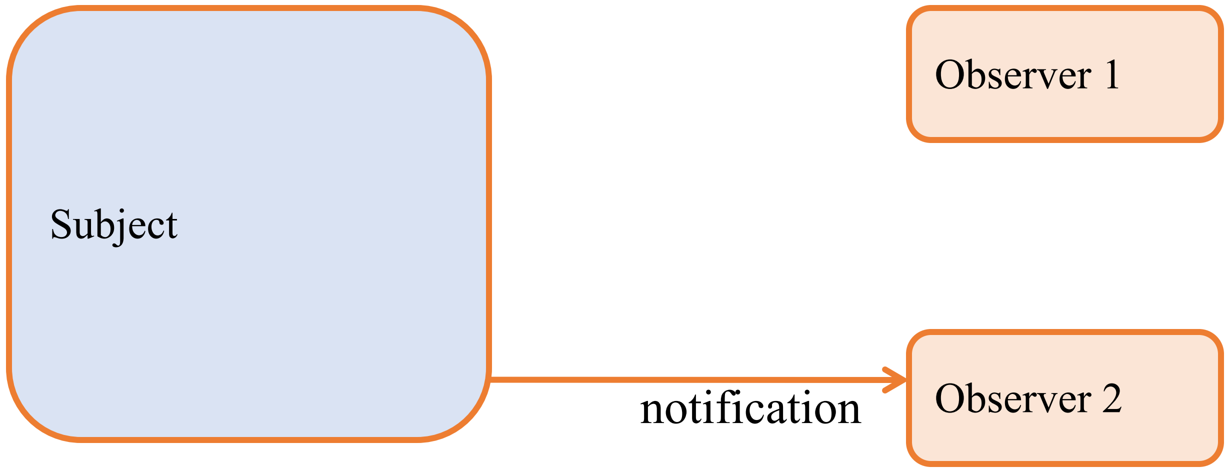
Observer Pattern



Observer Pattern



Observer Pattern



Java Code

```
class Subject {
    Vector observers = new Vector();
    void registerObserver(Observer o) {
            observers.add(o);
    }
    void removeObserver(Observer o){
            observer.remove(o);
    }
    void notifyObservers() {
            for (int i=0;i<observers.size();i++){
              Observer o=observers.get(i);
              o.update(this);
            }
    }
}
```

```
abstract class Observer {
    abstract void update(Subject s);
}

Class ClockTimer extends Subject {
    // timer state

    void tick() {
            // update timer state
            notifyObservers();
    }
}
```

Java Code

```
class PrintClock extends Observer {
    ClockTimet timer;
    public PrintClock(ClockTimer t) {
            this.timer = t;
            t.registerObserver(this);
    }
    void update(Subject s) {
            if (s == timer) {
              // get time from timer
              // and print time
            }
    }
}
```

```
abstract class Observer {
    abstract void update(Subject s);
}

Class ClockTimer extends Subject {
    // timer state

    void tick() {
            // update timer state
            notifyObservers();
    }
}
```

Observer Pattern

- According to the GoF book, the Observer design pattern should “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically”
- A subject may have any number of dependent observers.
- All observers are notified whenever the subject undergoes a change in state.
- This kind of interaction is also known as **publish-subscribe.**
- **The subject is the publisher of notifications.**

Design Patterns Philosophy

- *Program to an interface and not to an implementation*
- *Encapsulate variation*
- *Favor object composition over inheritance*

Design Patterns

- A good idea
  - Simple
  - Describe useful “micro-architectures”
  - Capture common organizations of classes/objects
  - Give us a richer vocabulary of design

- Relatively few patterns of real generality
- https://www.runoob.com/design-pattern

Reference

- Yale Software engineering, CPSC 439/539

谢谢大家！

THANKS