CS10102302
设计模式

赵君峤 副教授

计算机科学与技术系 电子与信息工程学院

同济大学

回顾软件设计

- SOLID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

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?

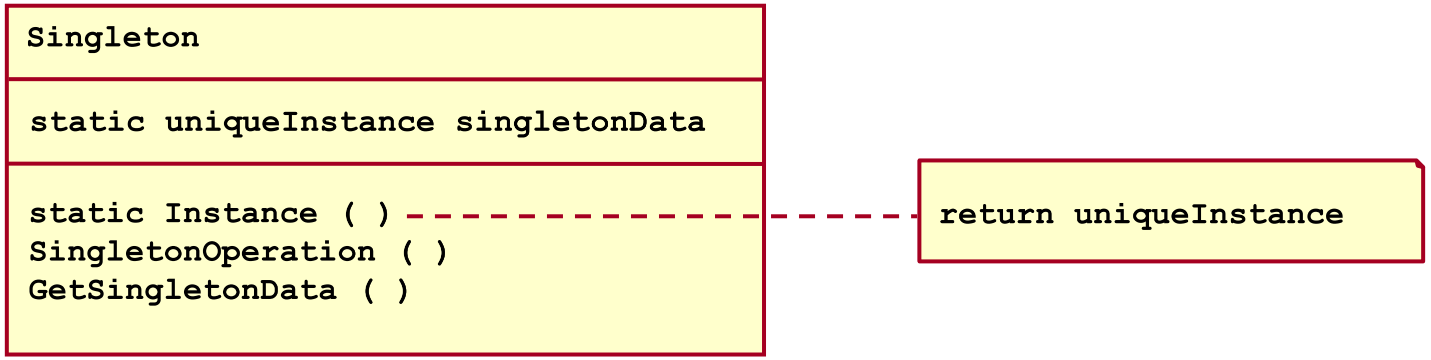
答案是肯定的

- 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

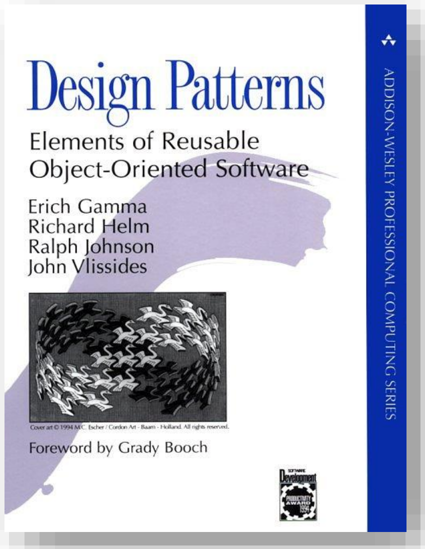
- 模式可以解决大多数问题，但不可能解决遇到的所有问题
- 应用一种模式一般会“有得有失”，切记不可盲目应用
- 滥用设计模式可能会造成过度设计，反而得不偿失

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

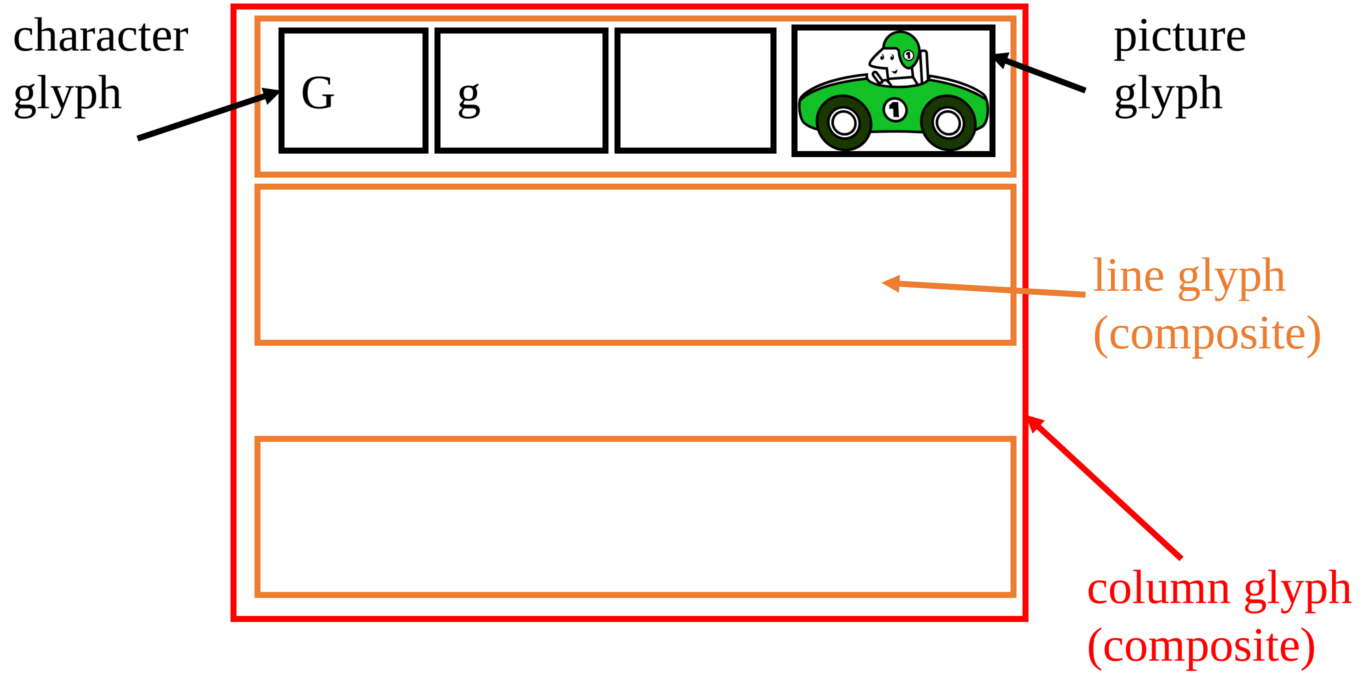
A Design

- Classes for Character, Circle, Line, Column, Page, …
  - Not so good
  - A lot of code duplication

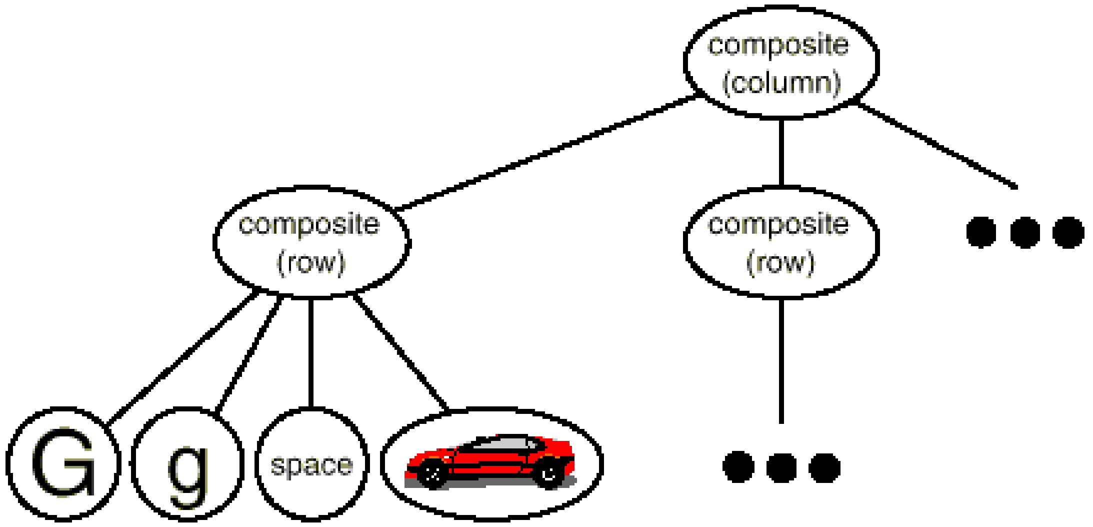
- One (abstract) class of Glyph
  - Each element realized by a subclass of Glyph
  - All elements present the same interface
    - How to draw
    - Mouse hit detection
    - …
  - Makes extending the class easy
  - Treats all elements uniformly

- 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);
    }
}
```

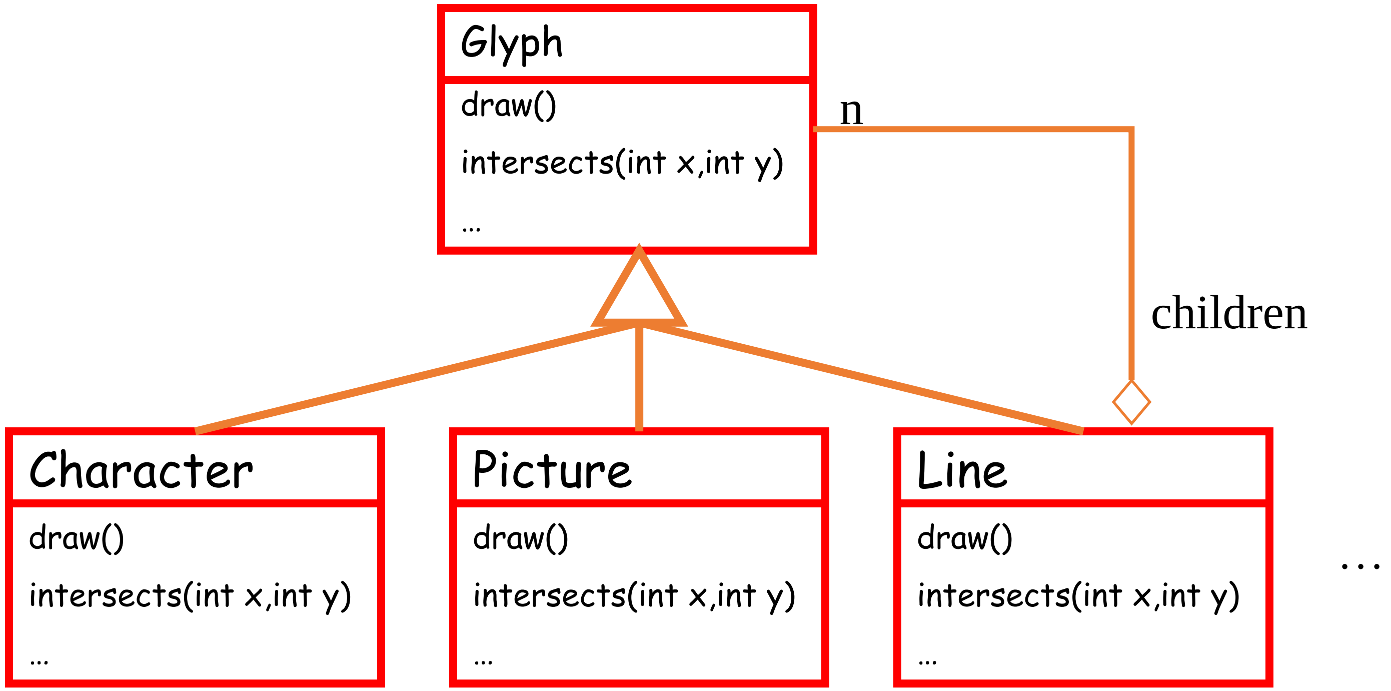
```
class Character extends Glyph {
    char c;
    // other attributes
    public Character(char c) {
                this.c = c;
                // set other attributes
    }
    void draw() {
                …
    }
    boolean intersects(int x, int y) {
                …
    }
}
```

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

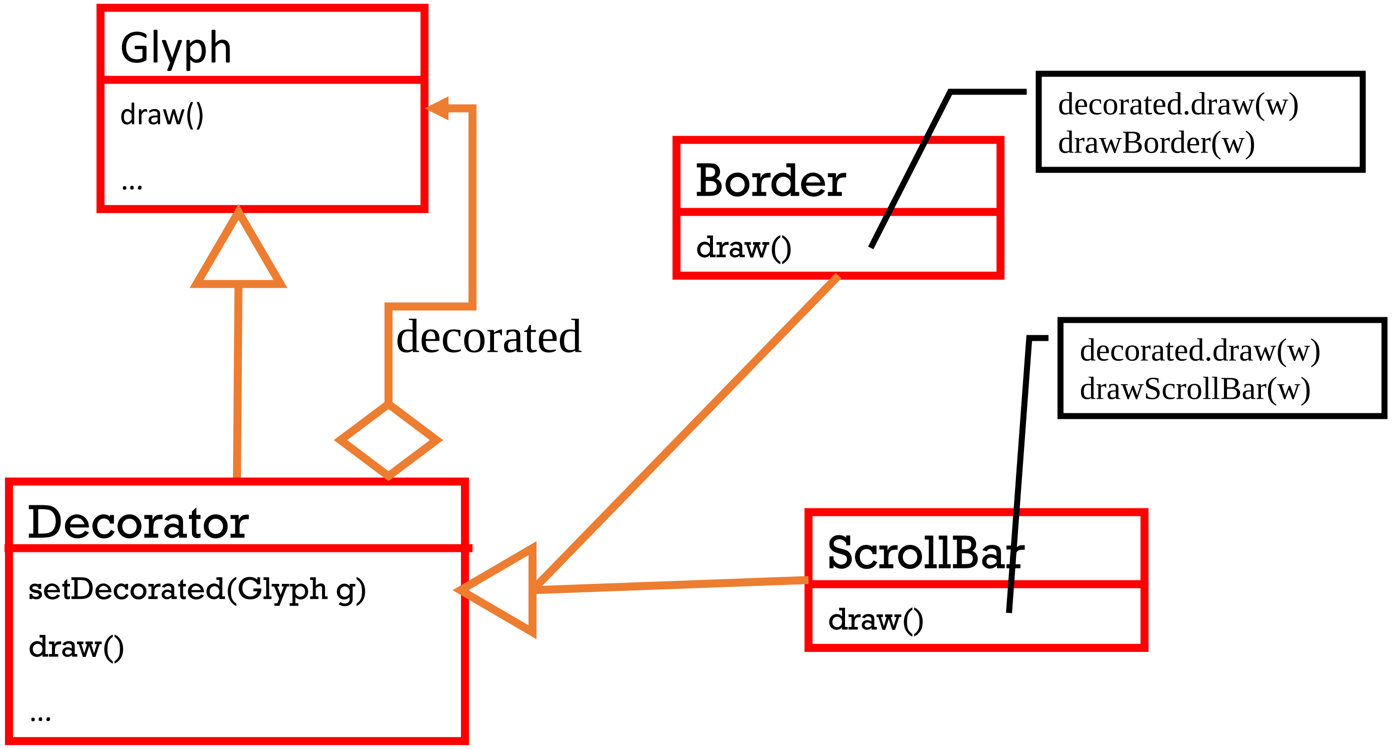
- Define Decorator
  - Implements Glyph
  - Has one member decorated of type Glyph
  - Border, ScrollBar, Menu extend Decorator

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

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

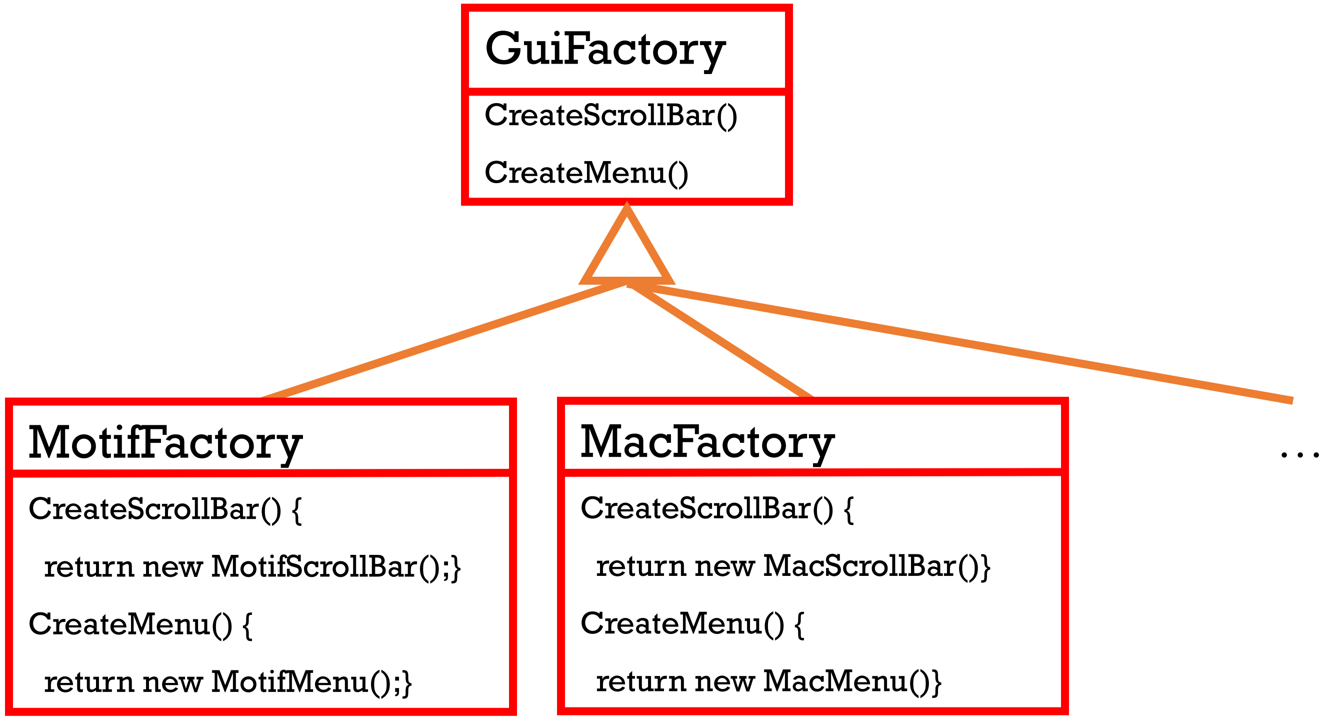
Java Code

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

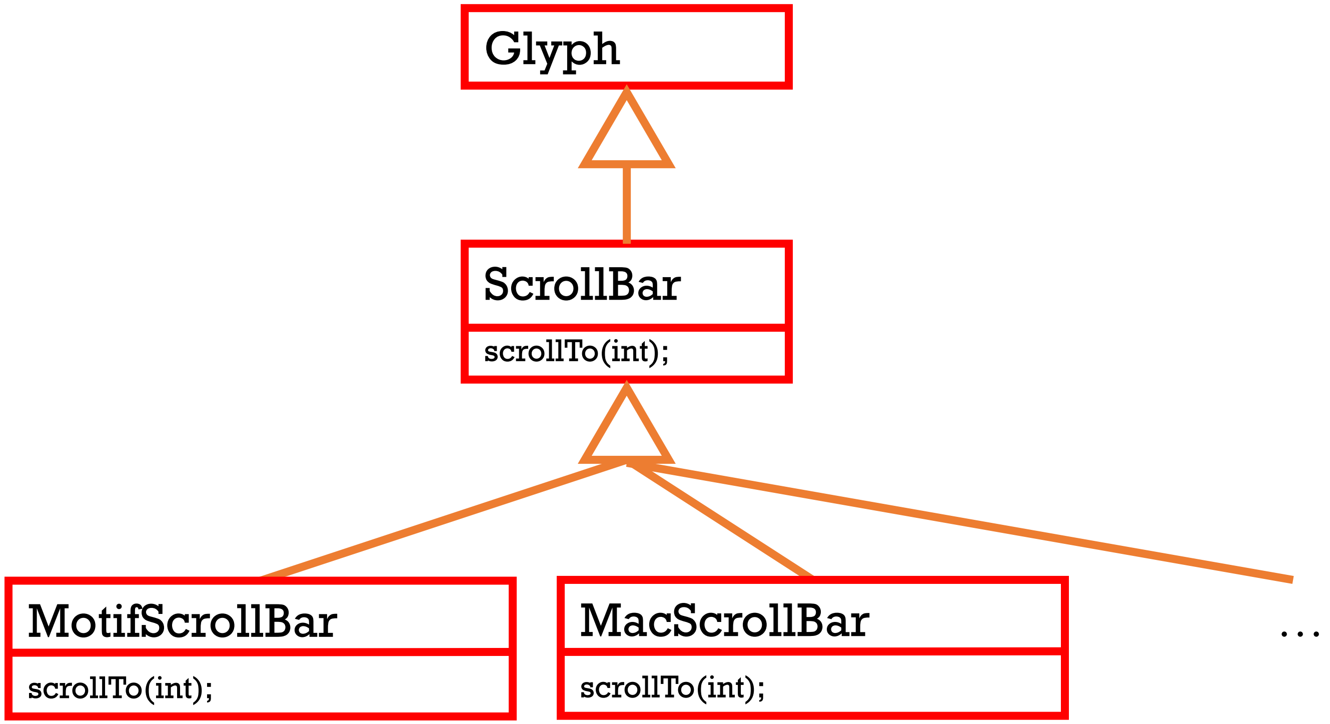
```
GuiFactory factory;
if (style==MOTIF)
    factory = new MotifFactory();
else if (style==MAC)
    factory = new MacFactory();
else if (style==…)
    …
```

```
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

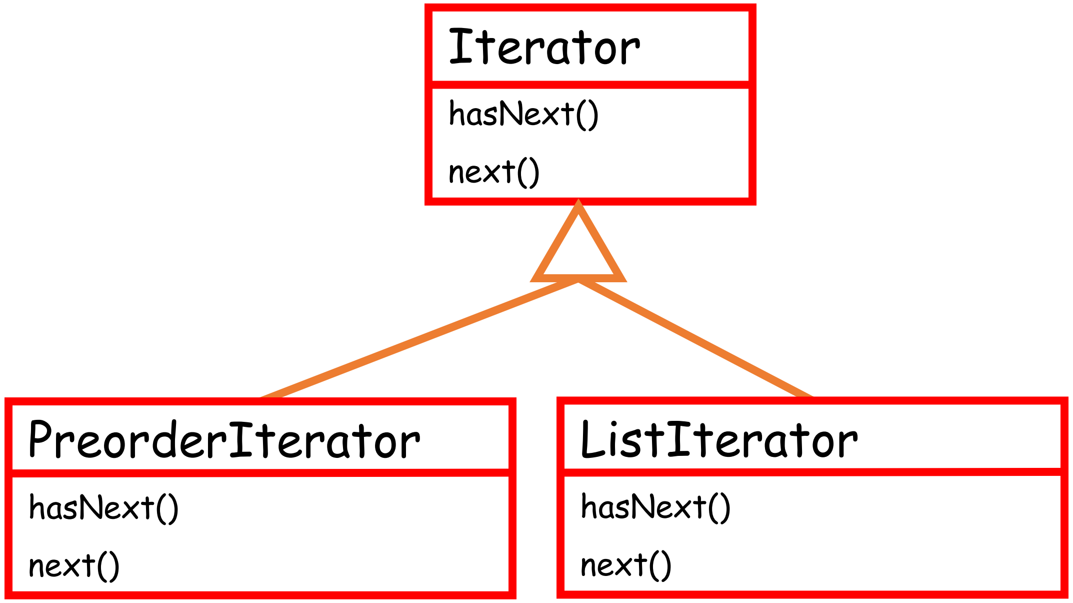
- Considerations
  - Spell-checking requires traversing the document
    - Need to see every glyph, in order
    - Information we need is scattered all over the document
  - There may be other analyses we want to perform
    - E.g., grammar analysis

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
  - Ugly
  - 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);
    }
}
```

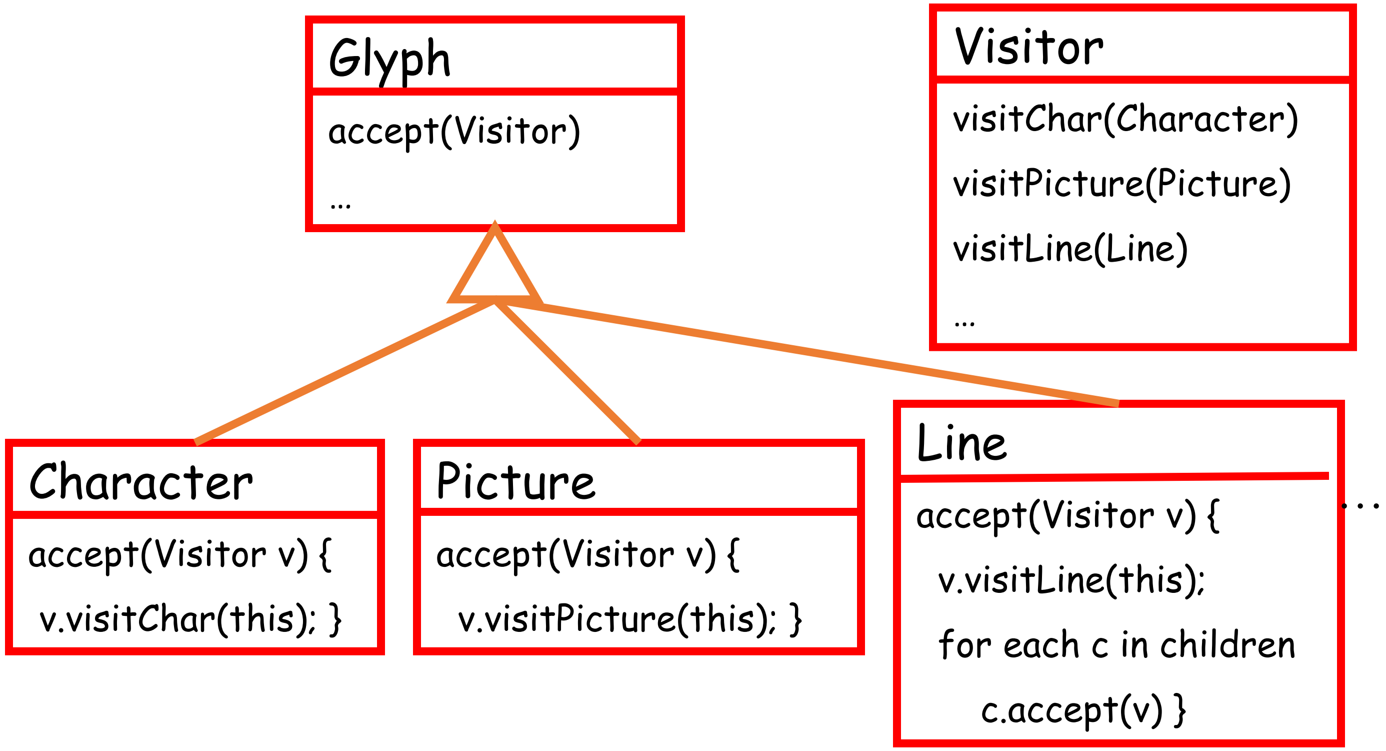
```
abstract class Visitor {
    abstract void visitChar (Character c);
    abstract void visitLine(Line l);
    abstract void visitPicture(Picture p);
    …
}

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

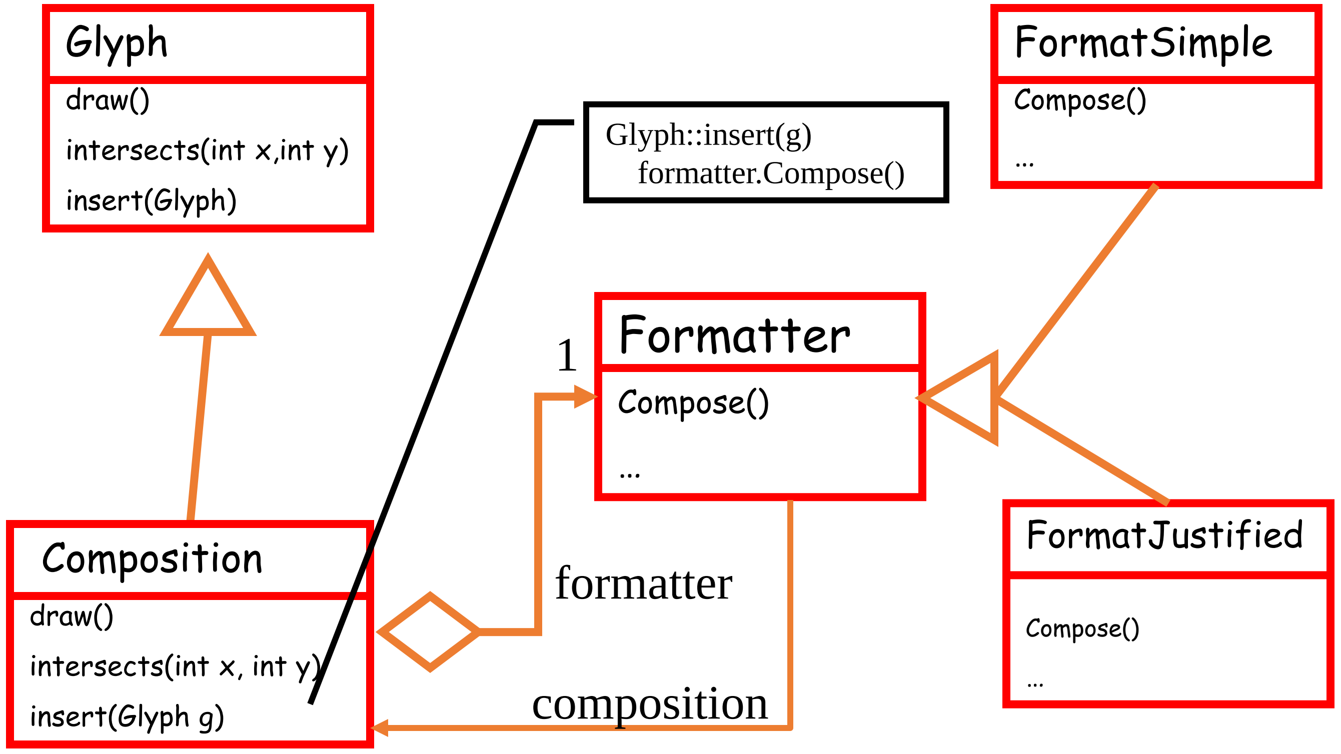
- Formatting is complex
  - We don’t want that complexity to pollute Glyph
  - We may want to change the formatting method
- Encapsulate formatting behind an interface
  - Each formatting algorithm an instance
  - Glyph only deals with the interface

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();
    }
}
```

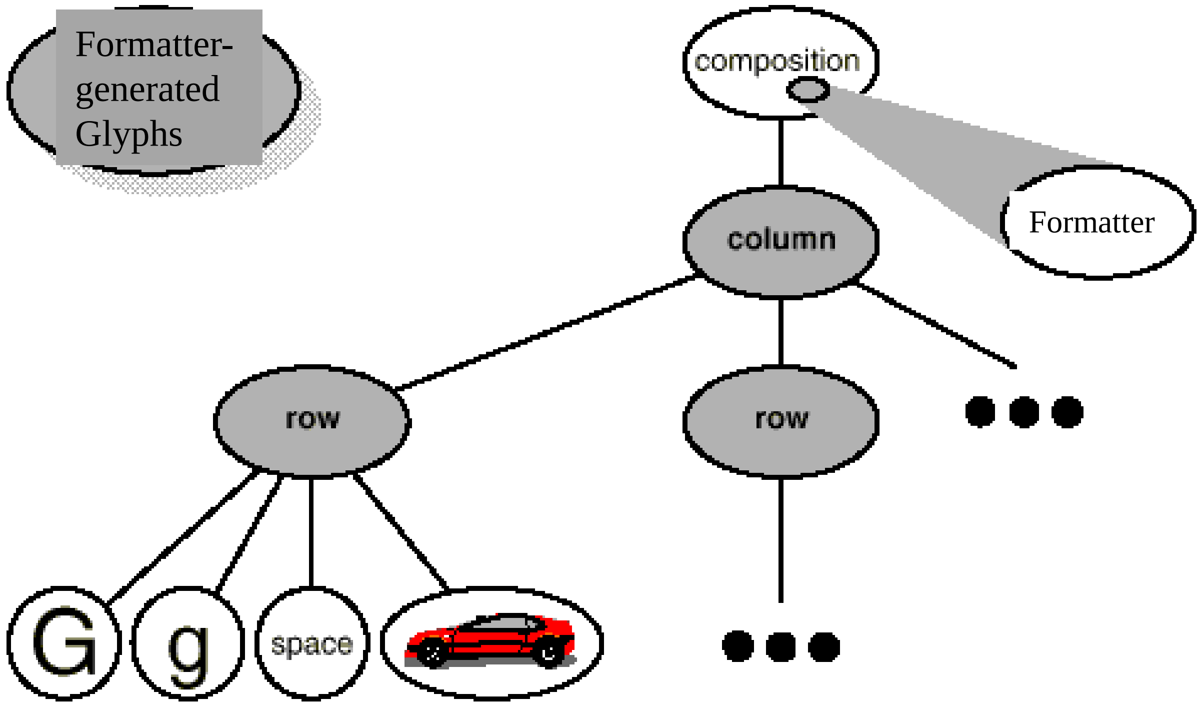
```
abstract class Formatter  {
    Composition composition

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

Formatter


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

Strategies

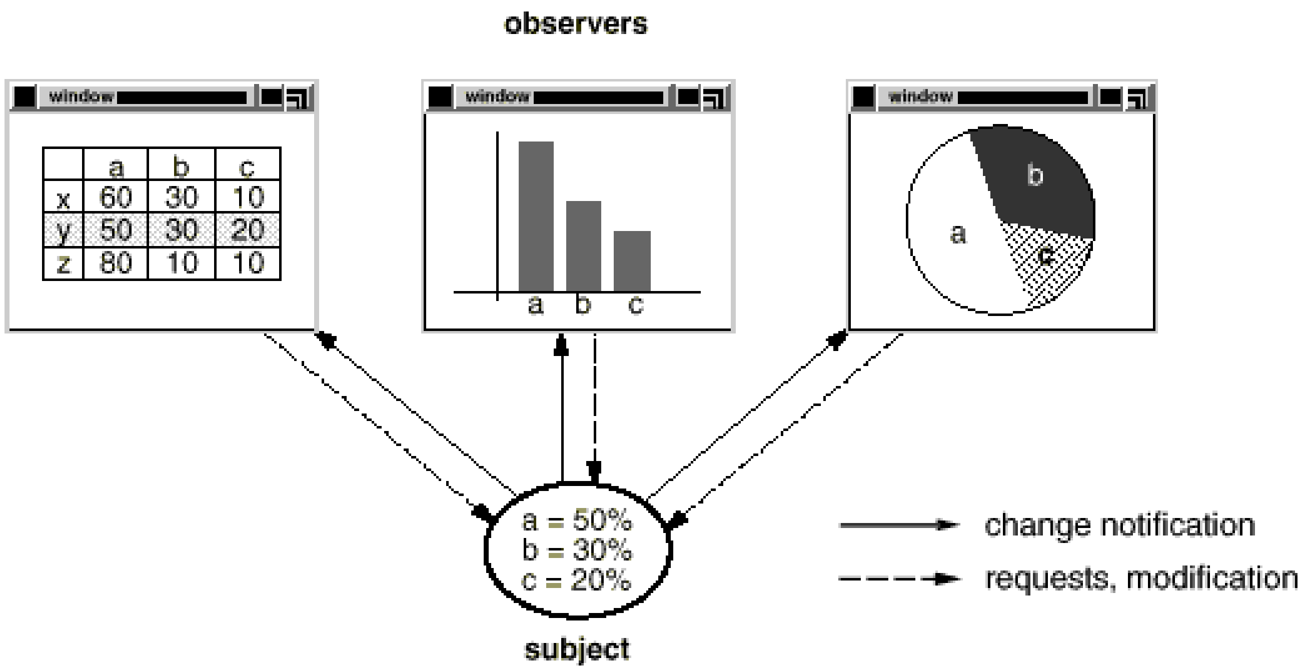
- This is the *strategy* pattern
  - Isolates variations in algorithms we might use
  - Formatter is the strategy, Composition is context
- The GoF book says the Strategy design pattern should: "**Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm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

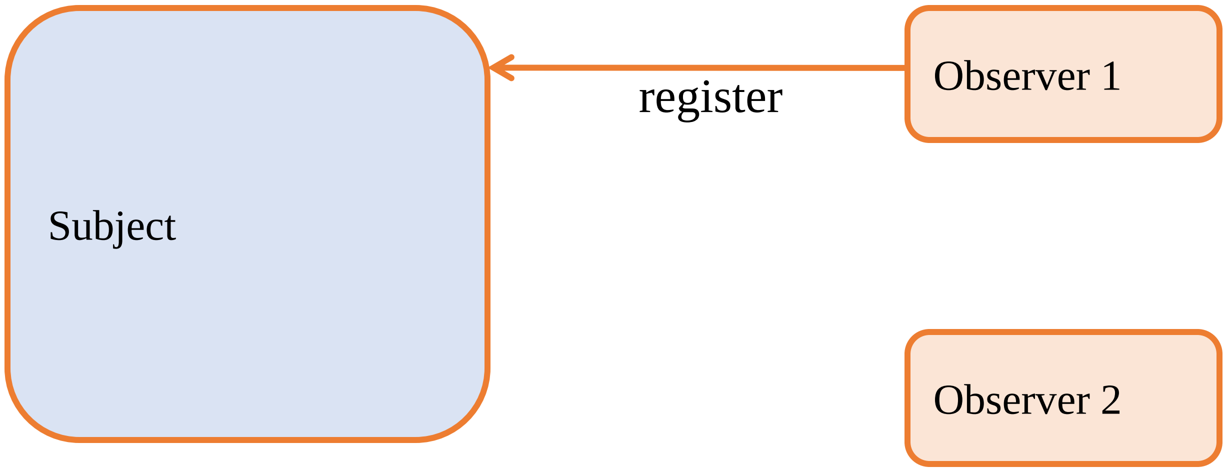
Problem: one-to-many-dependency

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

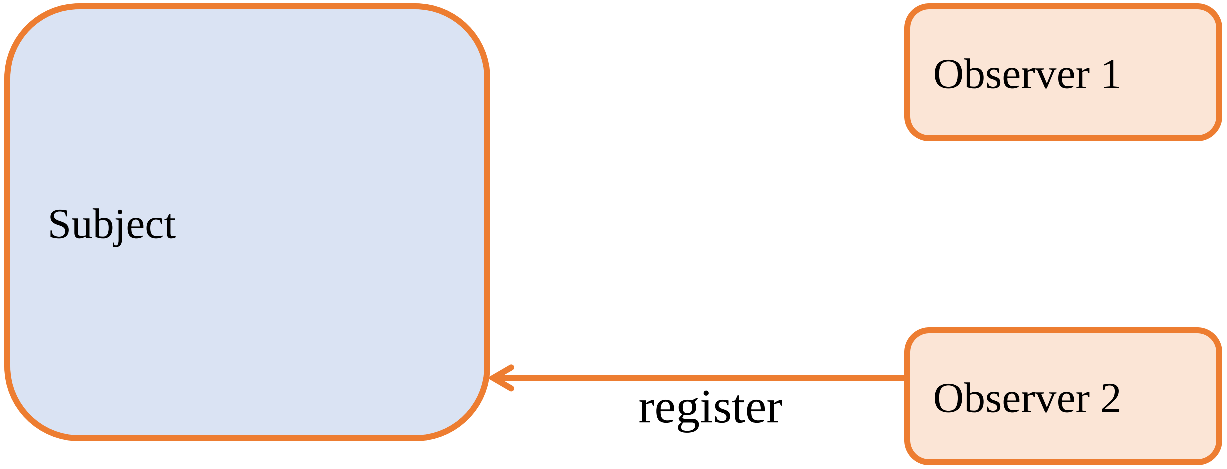


	a	b	c
x	60	30	10
y	50	30	20
z	80	10	10

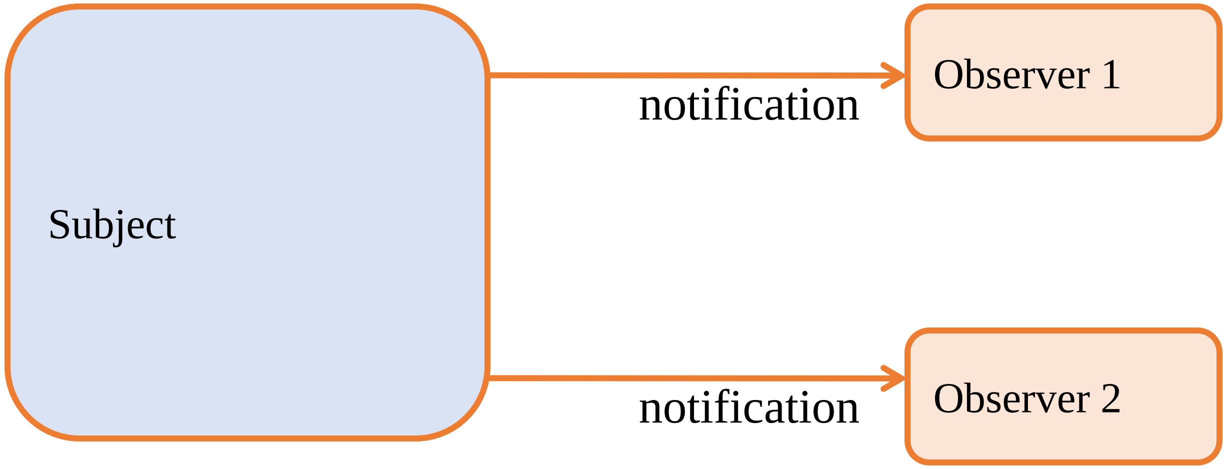
Observer Pattern



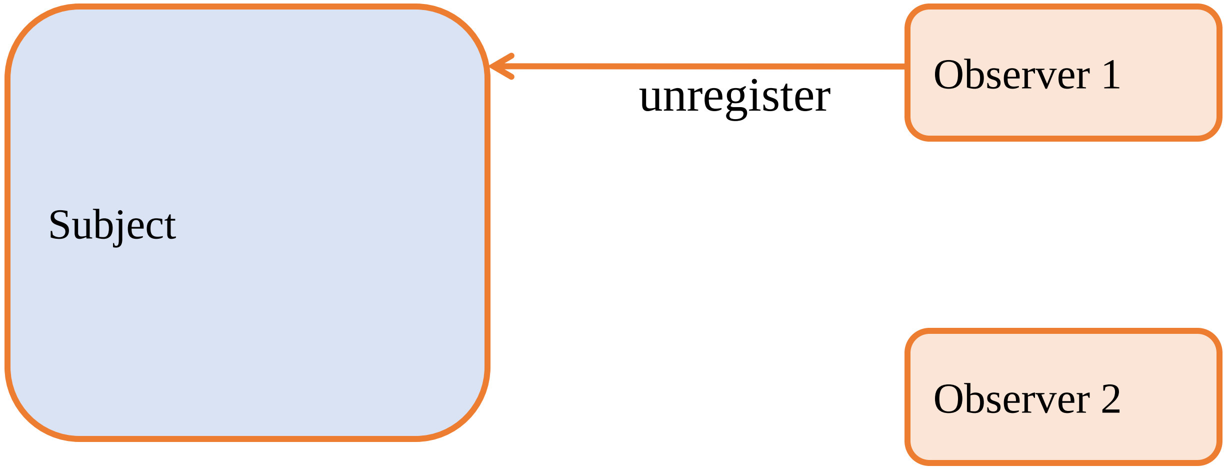
Observer Pattern



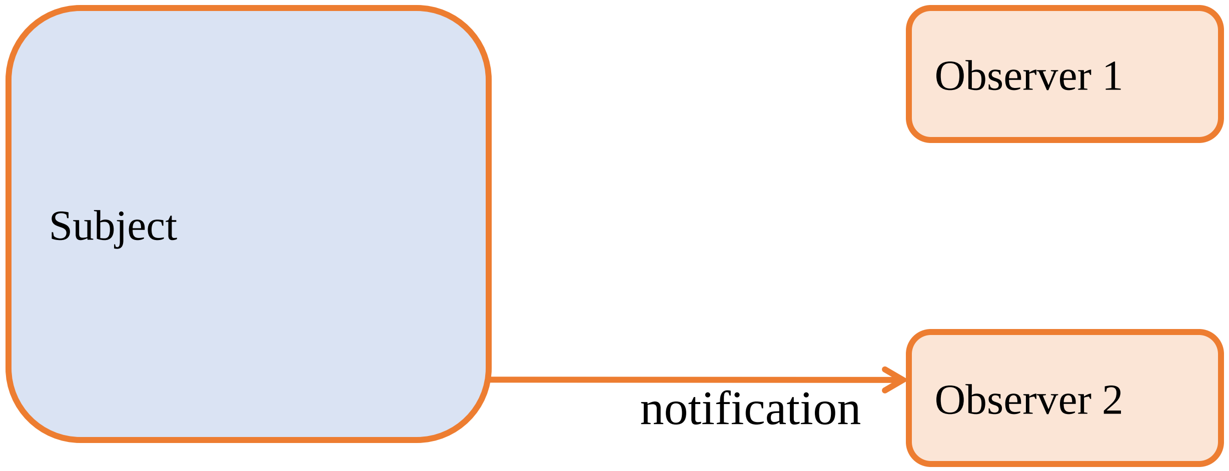
Observer Pattern



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

Reference

- 清华大学国家级精品课程 《软件工程》主讲人 刘强 副教授 刘璘 副教授
  - https://www.icourses.cn/sCourse/course_3016.html
  - https://www.xuetangx.com/course/THU08091000367/5883555?channel=learn_title
- Yale Software engineering, CPSC 439/539

谢谢大家！

THANKS