CS2001
软件工程

11. 软件设计
—设计框架

课件来源 复旦大学软件学院

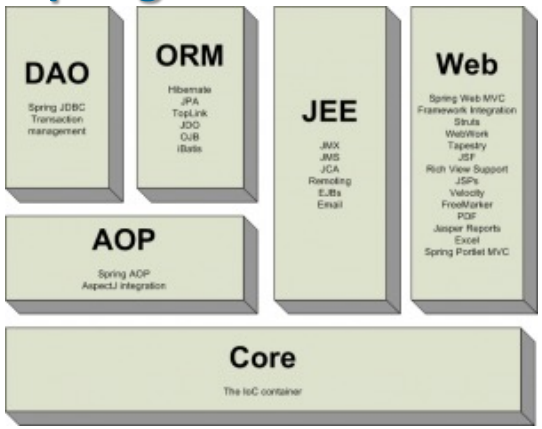
设计框架（Framework）

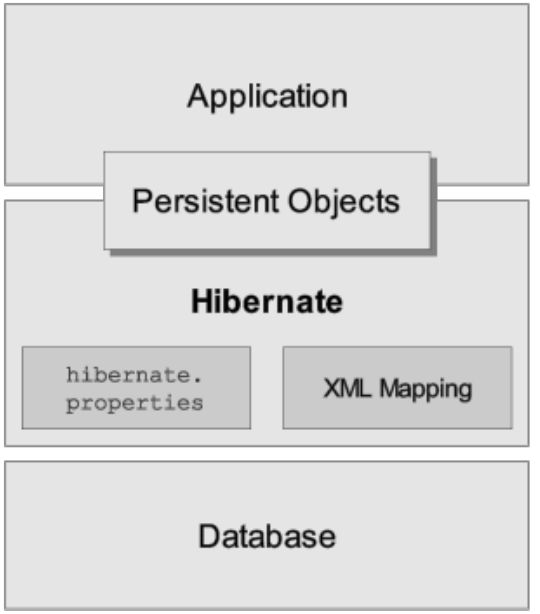
- 设计模式不足以让我们获得一个完整设计方案
- 有时需要一个包含特定实现的骨架

  “*reusable mini-architecture* that provides the generic structure and behavior for a family of software abstractions, along with a context … which specifies their collaboration and use within a given domain.” [Amb98]
- 框架是带有一组插入点（plug point、hook或slot）的骨架
  - ✓ 经过适配调整后可以支持不同的特定问题需要
  - ✓ 插入点上可以集成问题特定的类或功能

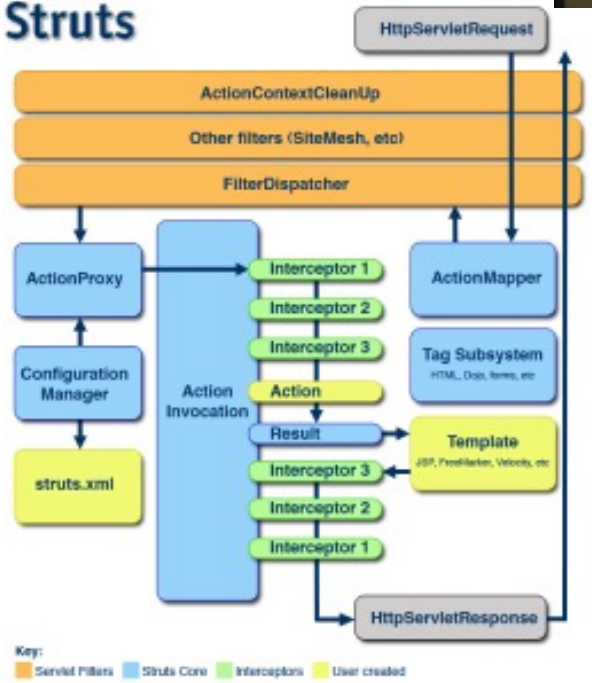
框架

Spring

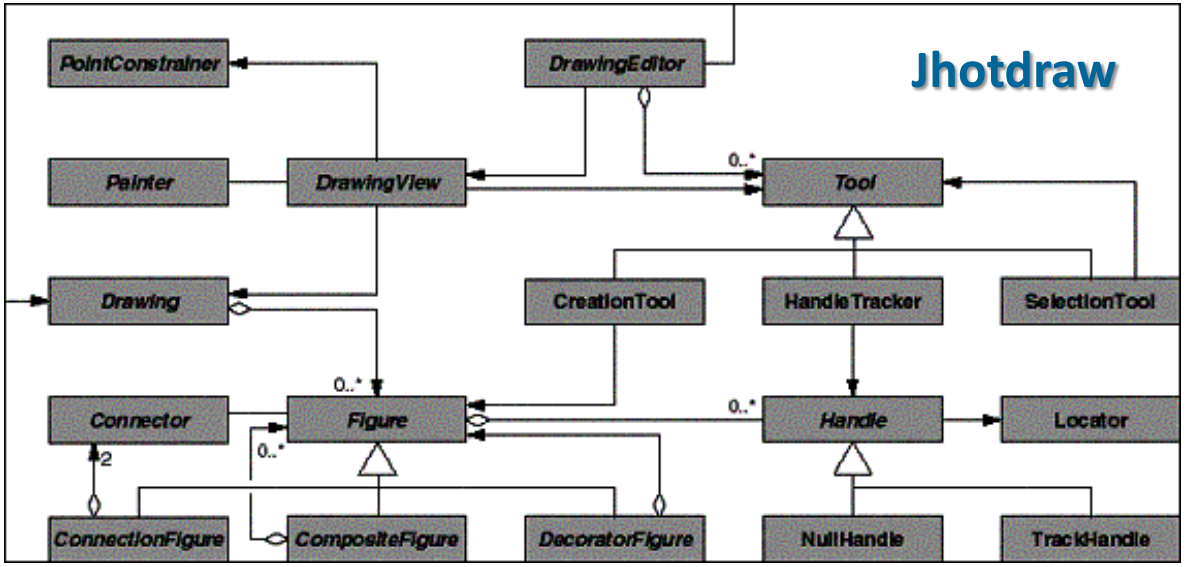




Struts



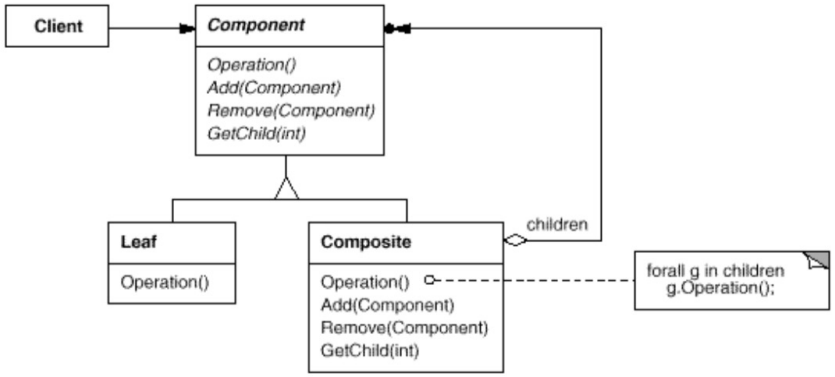
Jhotdraw



构件、模式、框架

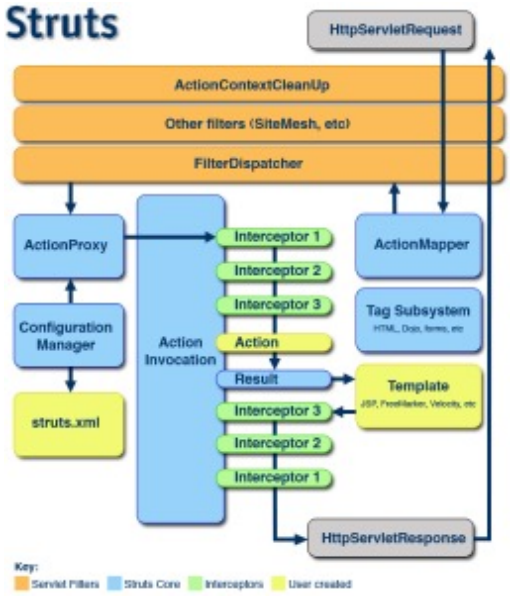
软件构件

（如图片处理、打印构件）



设计模式

（如组合模式、观察者模式）



设计框架

（如Struts、JhotDraw）

课堂讨论：构件、模式与框架

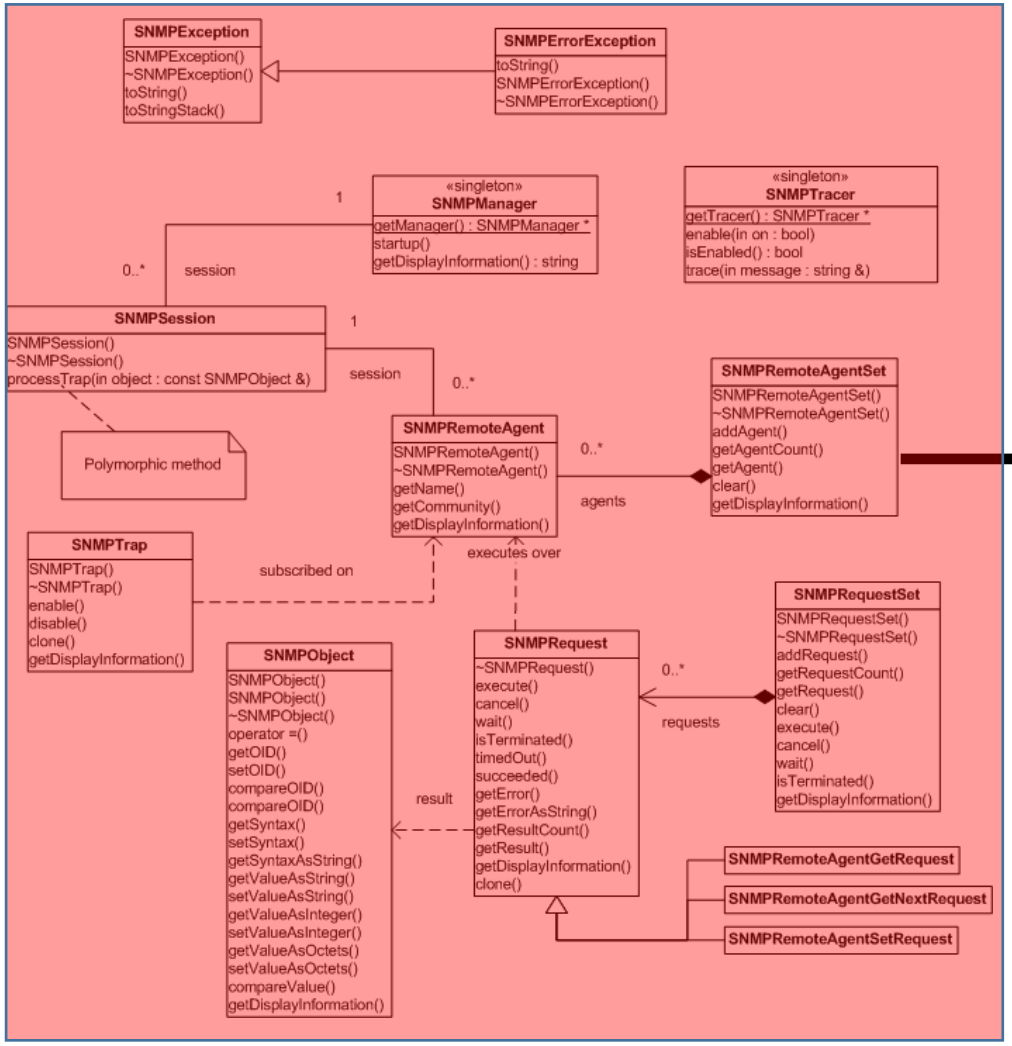
课堂讨论：构件、模式与框架都可以被复用，但它们在复用的内容以及方式上有什么区别？

软件构件：针对特定功能提供可复用的实现，开发人员在自身已有的实现方案基础上调用构件实现局部功能

设计模式：一种抽象的设计思想，往往体现为参考设计方案（如用UML图表示），本身并没有代码实现，需要针对具体问题、参考其设计思想进行实现

软件框架：本身包含相对完整的设计以及核心实现，提供扩展和定制能力，开发人员针对特定应用的实现通过扩展点插入框架中，一般由框架来调用形成完整的应用实现

软件构件复用示意（关键词：调用）

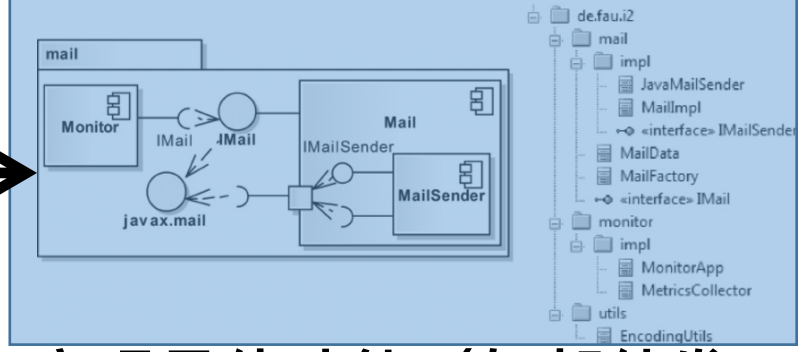


特定应用实现

被复用的部分

特定应用实现的部分

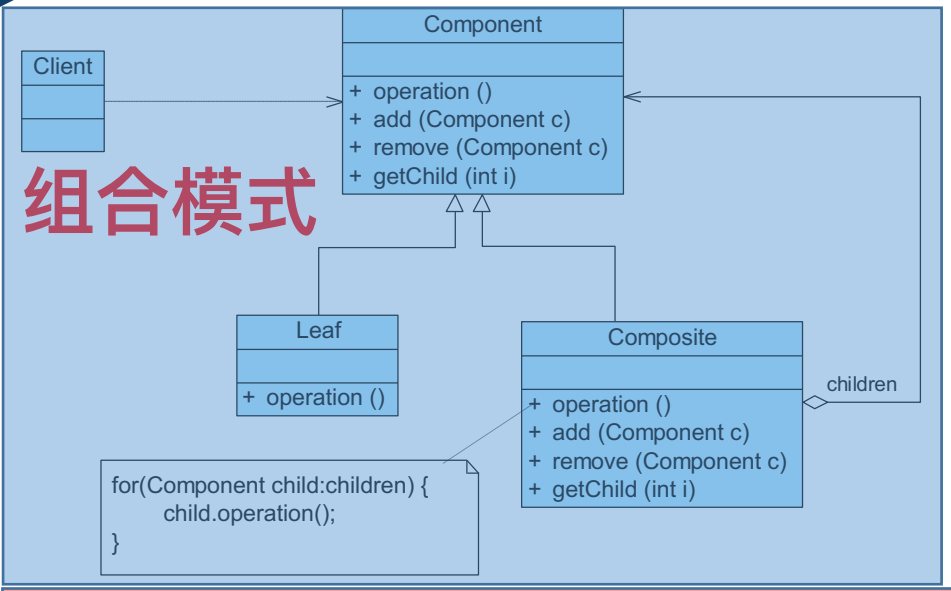
可复用的软件构件



实现具体功能（如邮件发送），有自己的实现并通过接口提供访问

特定应用实现调用软件构件实现局部功能，构件对系统整体设计影响不大

设计模式复用示意（关键词：实例化）

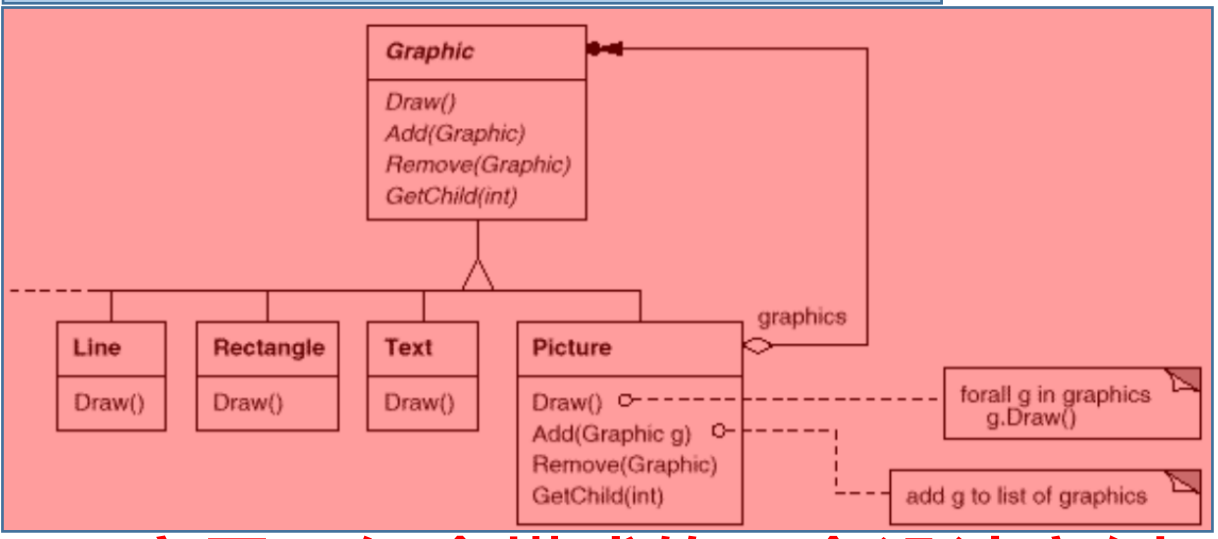


被复用的部分

特定应用实现的部分

反映的是抽象的设计思想，且只关注于某个局部设计问题，其中的类都是与特定应用无关的抽象角色，没有实现代码

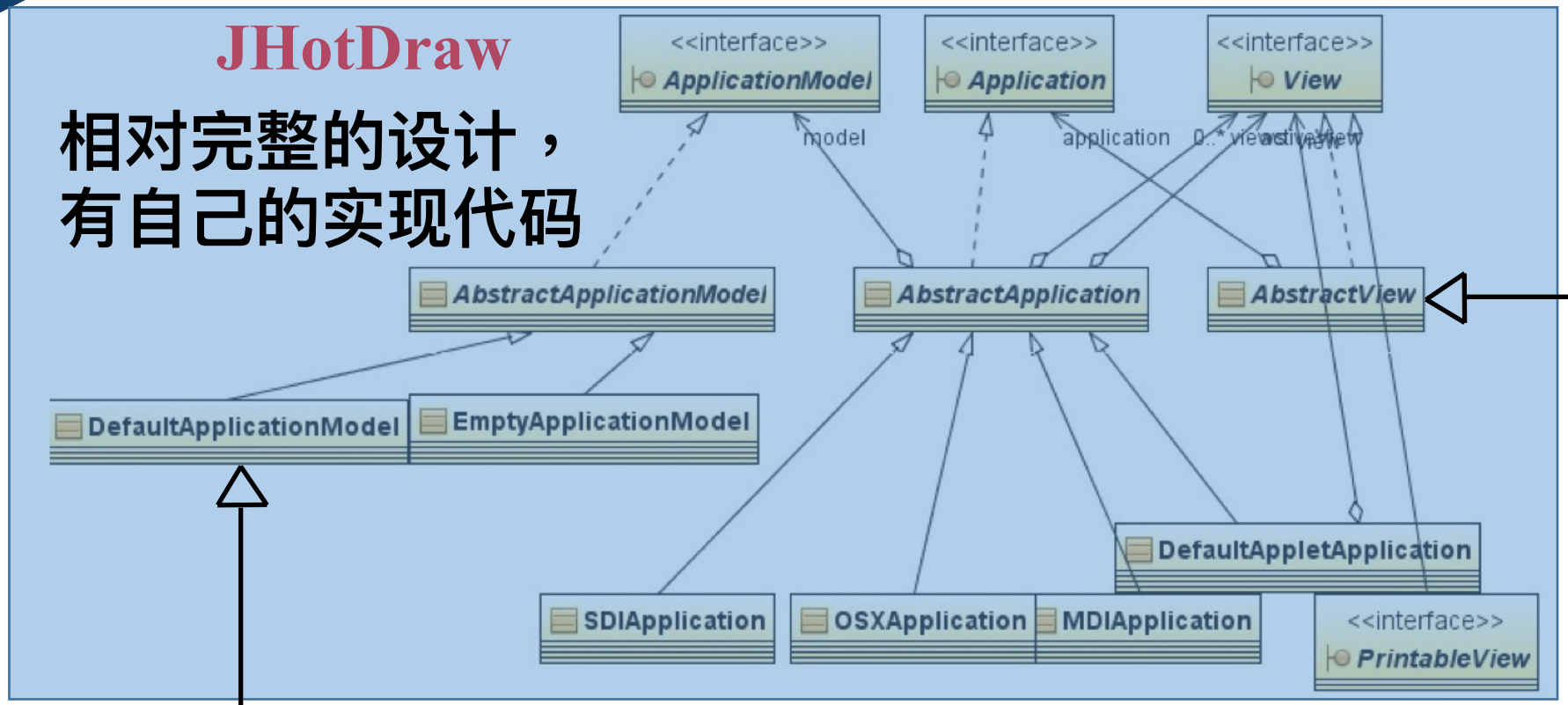
特定应用实现对设计模式中的角色及其关系进行实例化，得到特定用的设计和实现



基于设计模式的实例化实现，其中的类都是针对特定应用的设计类，存在对应的实现代码

应用了组合模式的一个设计实例

软件框架复用示意（关键词：扩展）



MyApplicationModel

特定应用通过框架的扩展点（如抽象、接口）扩展特定应用的实现，最终于框架实现融为一体

MyView

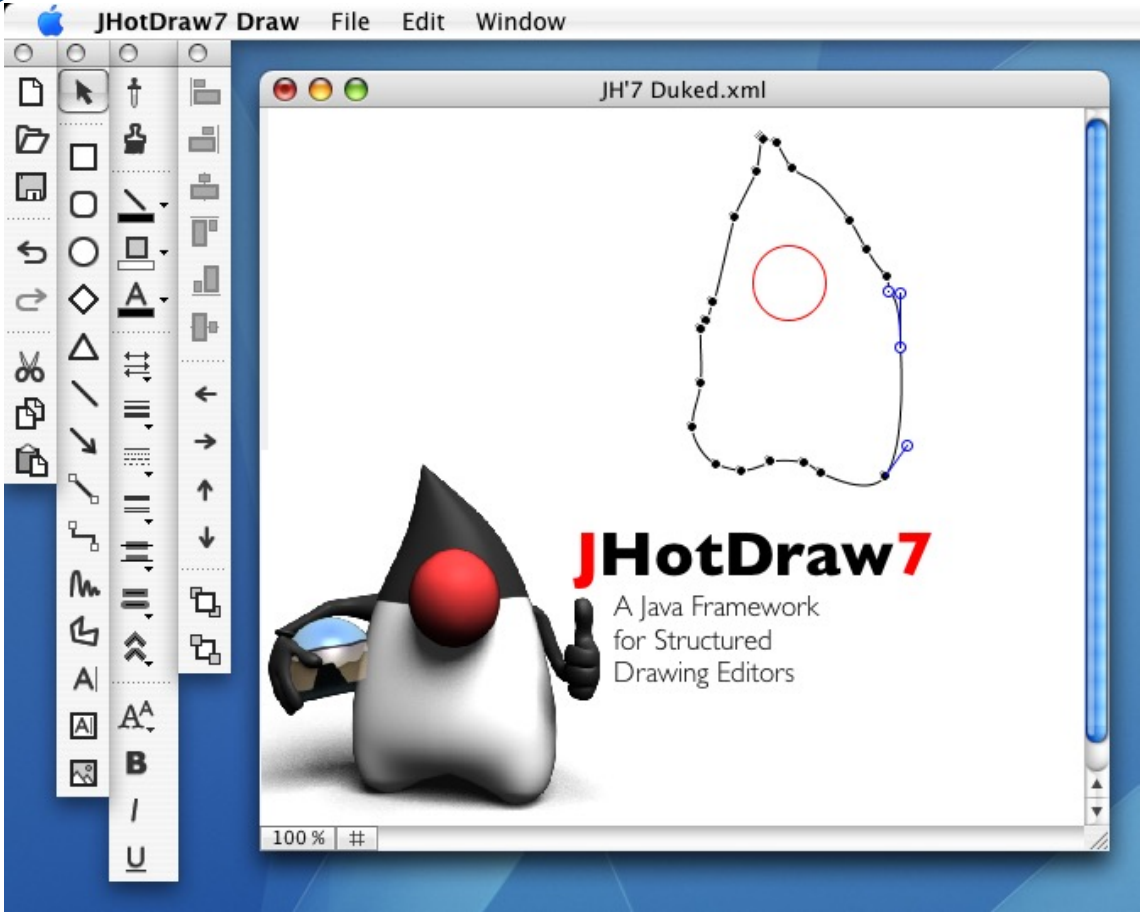
被复用的部分

特定应用实现的部分

设计框架案例分析：JHotDraw

- 基于LGPL协议的开源项目
  - ✓ LGPL协议：允许商业软件通过类库引用方式使用LGPL类库而不需要开源商业软件的代码
- 项目网址：http://www.jhotdraw.org
- Java GUI框架
  - ✓ 支持基于Java的图形编辑器开发
  - ✓ 在Swing框架基础上提供了图形编辑特性和功能
- 可扩展：以插件的形式开发各种图形编辑应用
- 内部包含丰富的设计模式实例

JHotDraw设计特性



遵循MVC架构模式

为插件化开发提供了一套精巧的设计结构

内部包含很多设计模式的具体实例

本身是一个可以直接使用的图形编辑应用，同时支持插件化扩展和定制

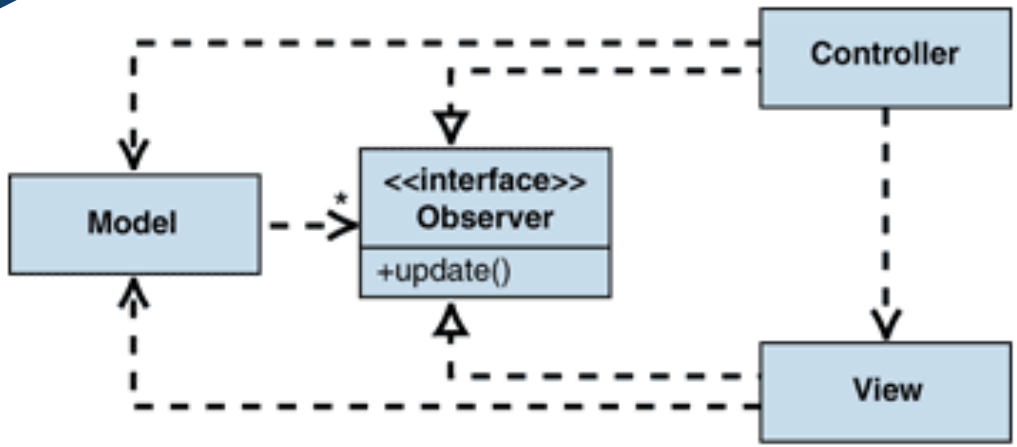
JHotDraw插件应用实例

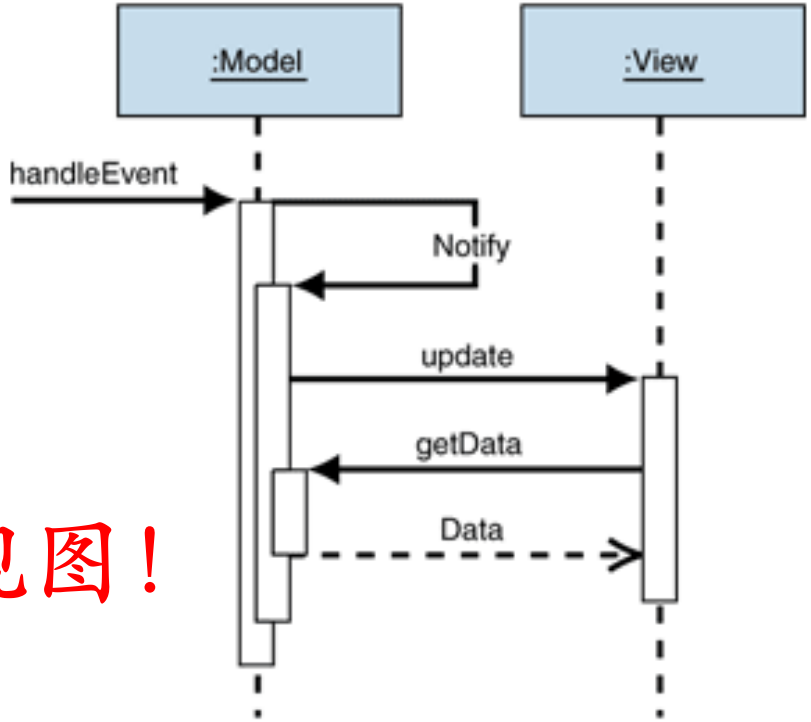
- Draw
- Net
- PERT
- SVG
- Teddy

课堂讨论：JHotDraw扩展方式

课堂讨论：根据JHotDraw应用的基本功能以及这些插件应用的特性，讨论插件应用需要在哪些方面进行扩展和定制？

MVC架构模式



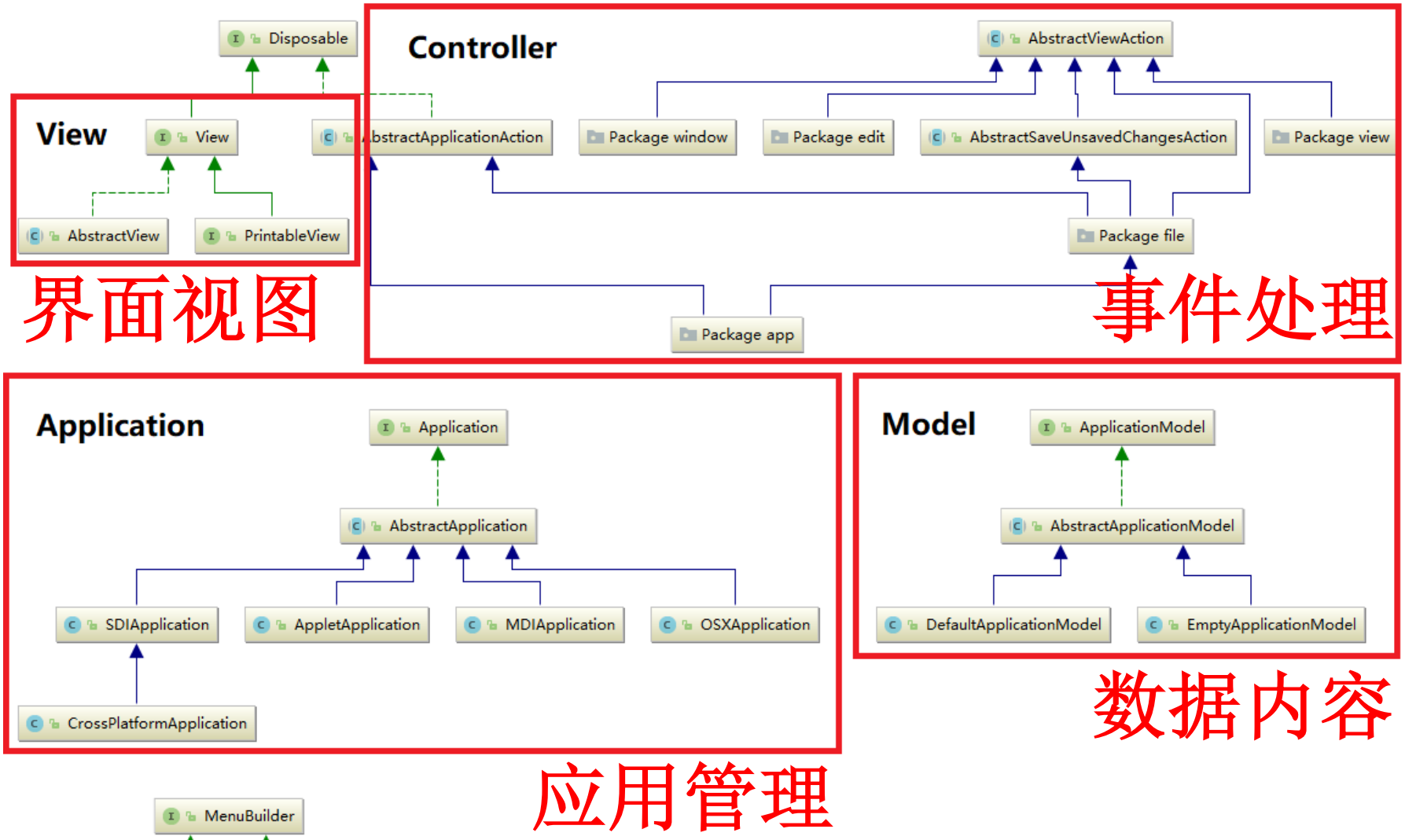


模型不依赖于控制器或视图！

模型、视图、控制器三者角色分离

- 模型保存需要展示和修改的数据，同时处理数据更新事件，数据发生修改后通知观察者（控制器和视图）
- 视图之间相互独立、不发生关联，收到更新通知后从模型那里读取数据进行视图更新
- 控制器根据模型数据的变化对视图进行控制（如设置显示属性）

JHotDraw的MVC架构

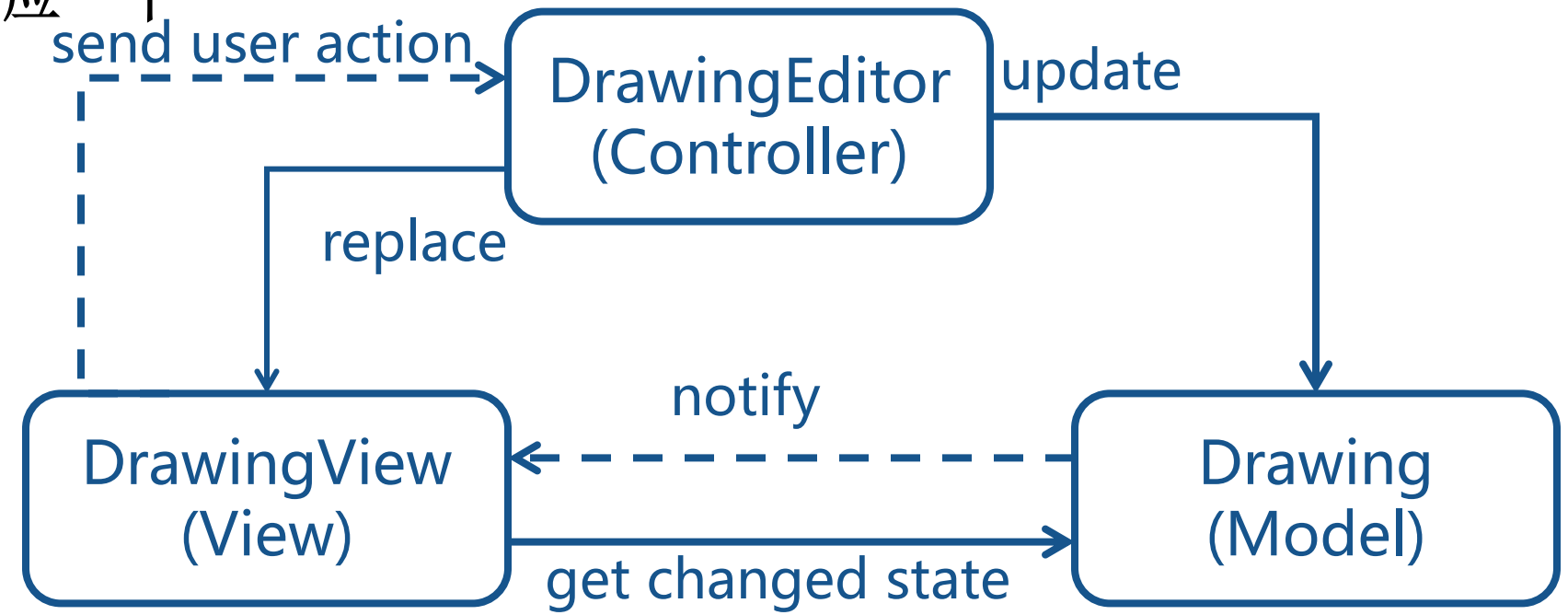


JHotDraw的MVC架构

Drawing（模型）：包含各种图形（**Figure**）的容器，其内容可以显示在多个视图上

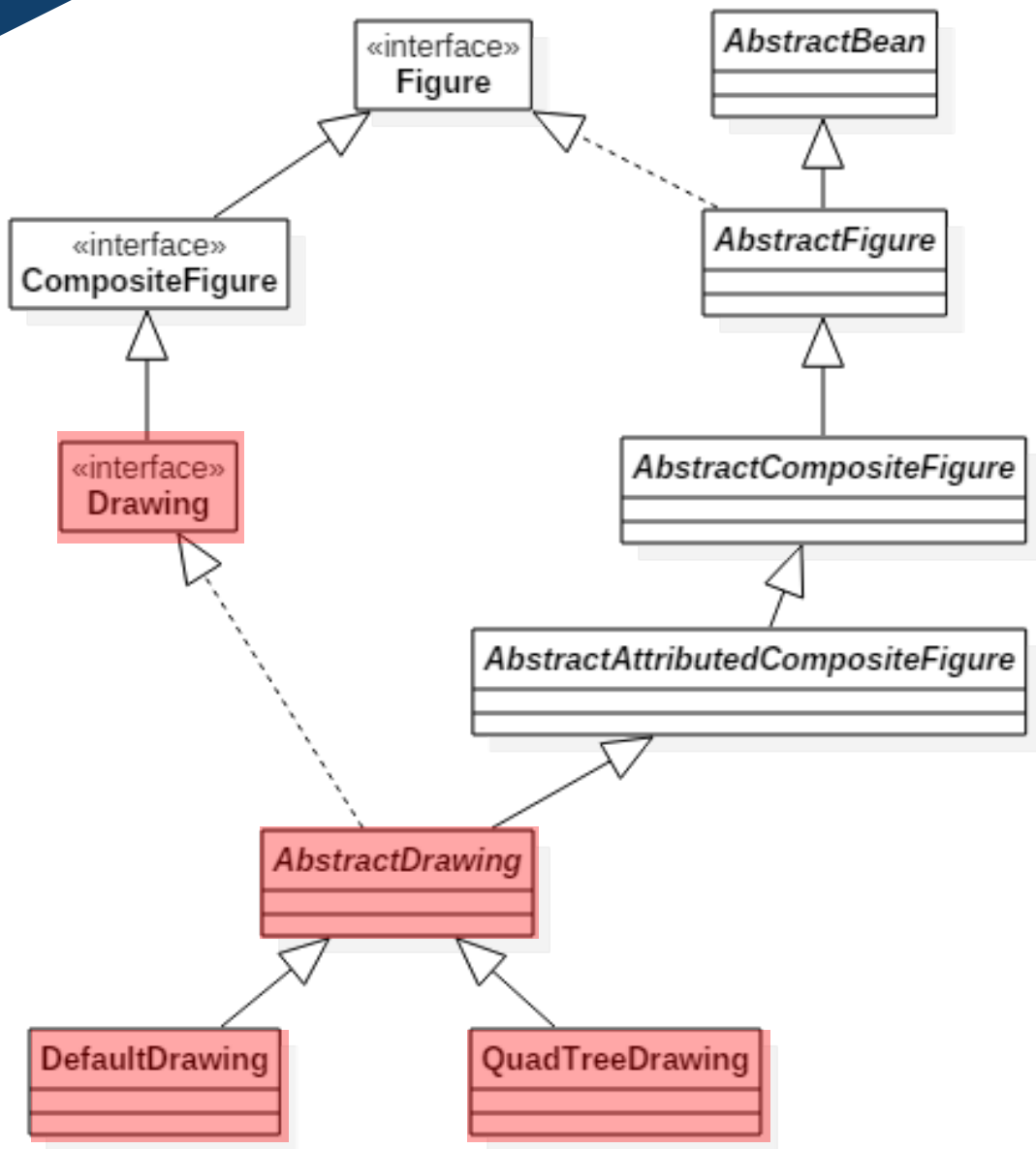
DrawingView（视图）：在**JComponent**上绘制模型内容，同时只能容纳一个模型内容

DrawingEditor（控制器）：协调画图工具和视图，每个文档窗口对应一个



模型、视图、控制器都是可扩展接口

Drawing（模型）继承结构



Figure定义了图形的一些特征和属性，如可见性、层级、边界、位置等

Drawing接口提供了一些额外的特性，包括持久化定义、undo/redo、快速搜索所包含的图形内容等

扩展应用开发者需要了解

Drawing：加入JHotDraw整体MVC结构的切入点

AbstractDrawing：插件应用模型的扩展点

DefaultDrawing：可直接使用的简单图形模型表示

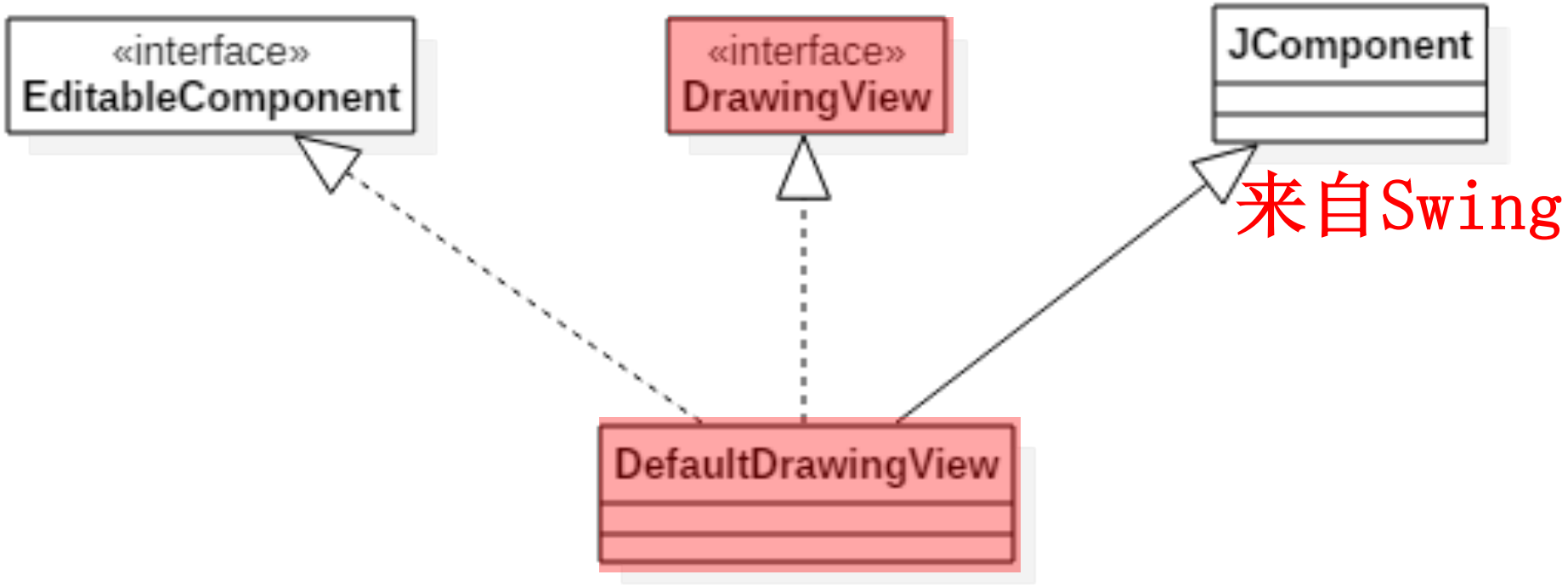
QuadTreeDrawing：更加复杂的图形模型表示

DefaultDrawing实现代码示例

```
public void draw(Graphics2D g) {
        synchronized (getLock()) {
            ensureSorted();
            ArrayList<Figure> toDraw = new
ArrayList<Figure>(getChildren().size());
            Rectangle clipRect = g.getClipBounds();
            for (Figure f : getChildren()) {
                if (f.getDrawingArea().intersects(clipRect)) {
                    toDraw.add(f);
                }
            }
            draw(g, toDraw);
        }
    }
```

收集重绘区域内的图形准备绘制（会在视图层被调用从而完成真正的绘制）

DrawingView（视图）继承结构



扩展应用开发者需要了解

DrawingView：加入JHotDraw整体MVC结构的切入点

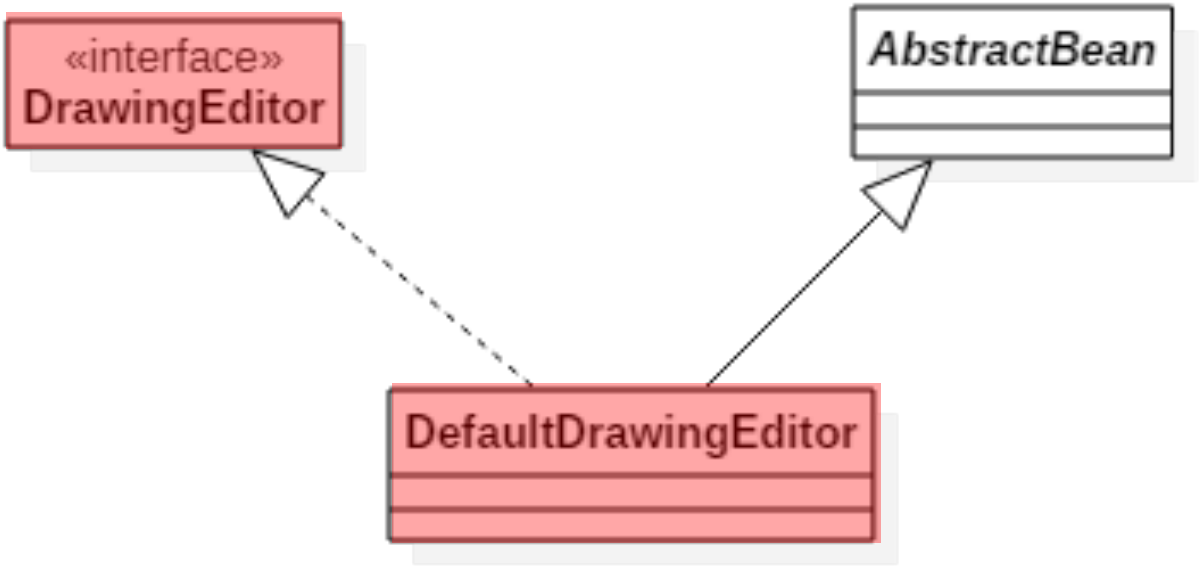
DefaultDrawingView：可直接使用的简单视图呈现

DefaultDrawingView实现代码示例

```
public void paintComponent(Graphics gr) {
        Graphics2D g = (Graphics2D) gr;
        setViewRenderingHints(g);
        drawBackground(g);
        drawCanvas(g);
        drawConstrainer(g);
        if (isDrawingDoubleBuffered()) {
            …
        } else {
            drawDrawing(g);
        }
        drawHandles(g);
        drawTool(g);
    }
```

具体绘制一个视图，其中drawDrawing()会调用之前Drawing中定义的draw()方法

DrawingEditor（控制器）继承结构



扩展应用开发者需要了解

DrawingEditor：加入JHotDraw整体MVC结构的切入点

DefaultDrawingEditor：可直接使用的简单视图呈现

DefaultDrawingEditor实现代码示例

```
public void setTool(Tool newValue) {
        …
      tool = newValue;
        if (tool != null) {
            tool.activate(this);
            for (DrawingView v : views) {
                v.addMouseListener(tool);
                v.addMouseMotionListener(tool);
                v.addKeyListener(tool);
            }
            tool.addToolListener(toolHandler);
        }
        …
}
```

具体管理视图的事件监听器及事件处理

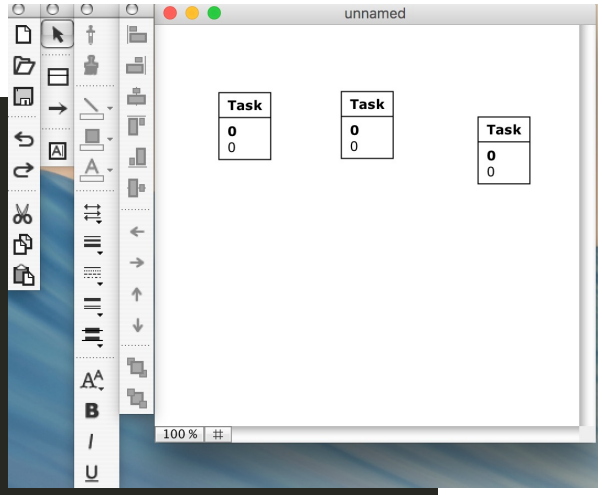
JHotDraw的主要模块

- 整个JHotDraw应用框架的API都位于org.jhotdraw.app包之中
- 其中包括模型、视图、控制器三个方面的内容
- 应用管理（Application）封装了控制应用生命周期的方法，包括
  - ✓ Init() 初始化程序
  - ✓ launch() 启动程序
  - ✓ Start() 开始运行
  - ✓ Stop() 停止运行
  - ✓ Destroy() 退出程序

JHotDraw应用扩展示例：简单日记本

- 定义Model：封装程序名称、版本、版权
- 定义View：包括文本输入框、保存按钮、打开按钮
- 定义Controller：用于保存和打开日记的控制方法
- 使用model.setView方法将View添加到Model
- 使用application.setModel方法将Model添加到Application
- 启动程序，调用Application.launch()方法启动程序

JHotDraw插件应用扩展实例（PERT）




```
public static void main(String[] args) {
    Application app;
    String os = System.getProperty("os.name").toLowerCase();
    if (os.startsWith("mac")) {
        app = new OSXApplication();
    } else if (os.startsWith("win")) {
      //  app = new DefaultMDIApplication();
        app = new SDIApplication();
    } else {
        app = new SDIApplication();
    }

    DefaultApplicationModel model = new PertApplicationModel();
    model.setName("JHotDraw Pert");
    model.setVersion(Main.class.getPackage().getImplementationVersion());
    model.setCopyright("Copyright 2006-2010 (c) by the authors of JHotDraw and all it
            "This software is licensed under LGPL and Creative Commons 3.0 Attributio
    model.setViewClassName("org.jhotdraw.samples.pert.PertView");
    app.setModel(model);
    app.launch(args);
}
```

扩展时无需了解内部实现机制，只需要继承DefaultApplicationModel，并将其注入Application

课堂讨论：

课堂讨论：模型（model）在设置对应的视图（view）实现时通过字符串传入的方式有什么好处？

```
model.setName("JHotDraw Draw");
model.setVersion(Main.class.getPackage().getImplementationVersion());
model.setCopyright("Copyright 2006-2009 (c) by the authors of JHotDraw a
        "This software is licensed under LGPL or Creative Commons 3.0 A
model.setViewClassName("org.jhotdraw.samples.draw.DrawView");
app.setModel(model);
app.launch(args);
```

课堂讨论：

课堂讨论：根据JHotDraw的特性，想象一下其中会应用哪些设计模式？

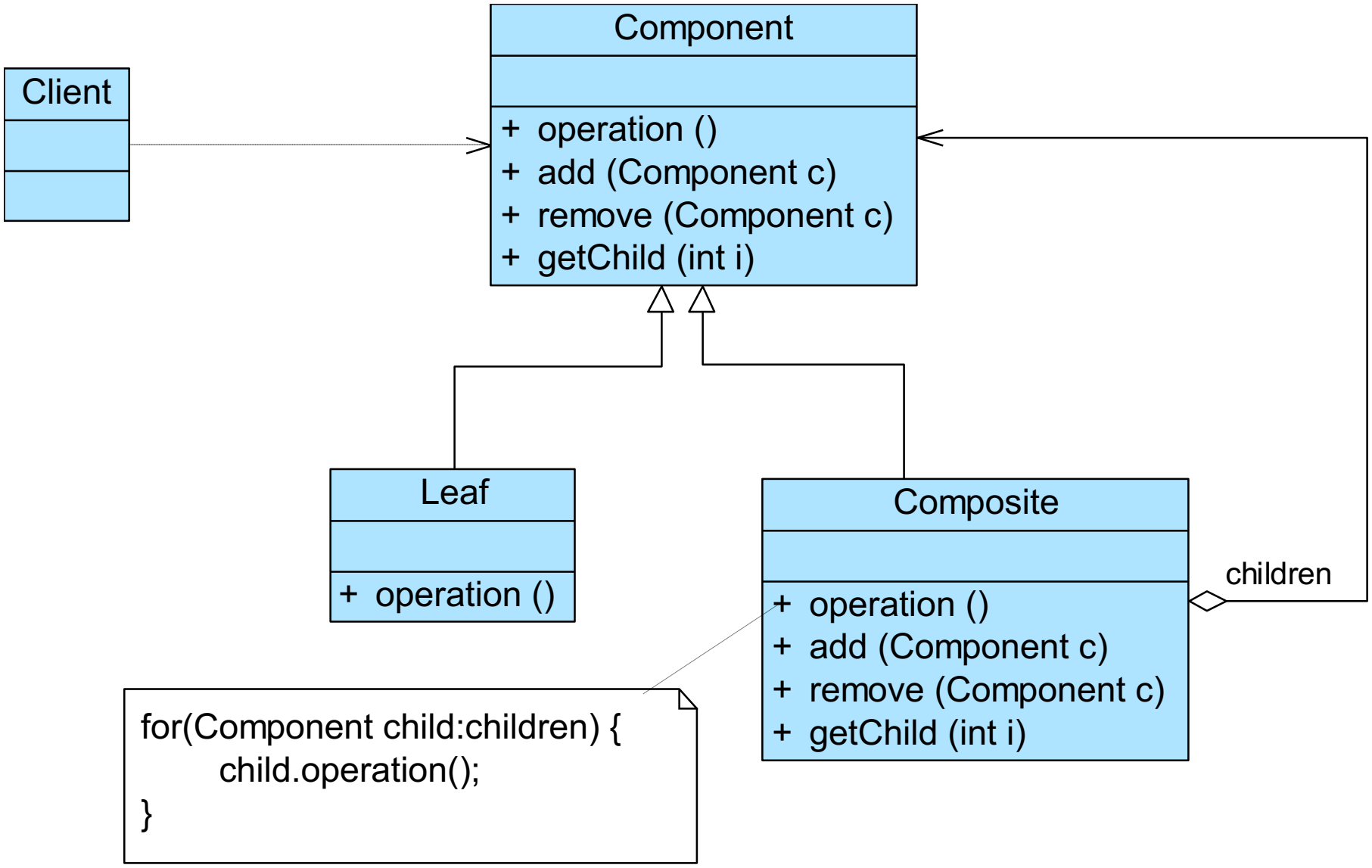
?

JHotDraw中的设计模式实例

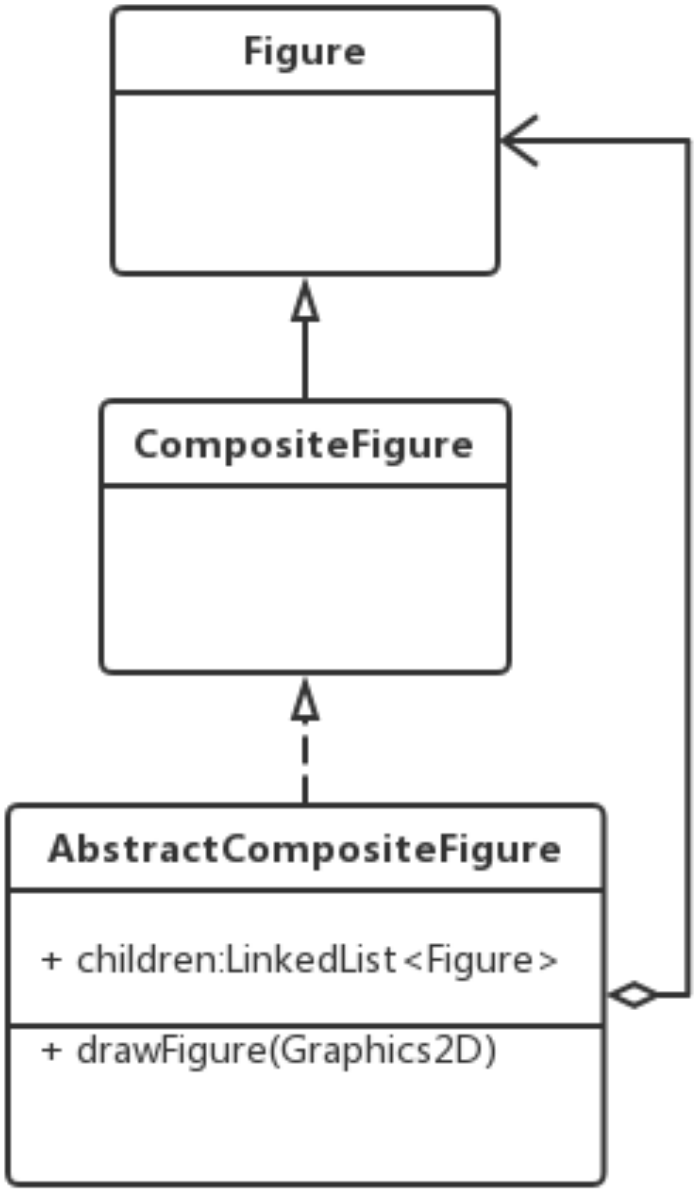
- 组合模式
- 观察者模式
- 策略模式

共性：都是以设计抽象为基本手段，都很好地实践了开闭原则。

组合模式


Component
+ operation ()
+ add (Component c)
+ remove (Component c)
+ getChild (int i)
Client
Leaf
+ operation ()
Composite
+ operation ()
+ add (Component c)
+ remove (Component c)
+ getChild (int i)
children
for(Component child:children) {
child.operation();
}

JHotDraw中的组合模式应用



AbstractCompositeFigure 使用**List<Figure>**属性表示对于子图形的包含关系（组合关系），在**drawFigure()**方法中对**List<Figure>**进行遍历（扮演客户端代码的角色）。

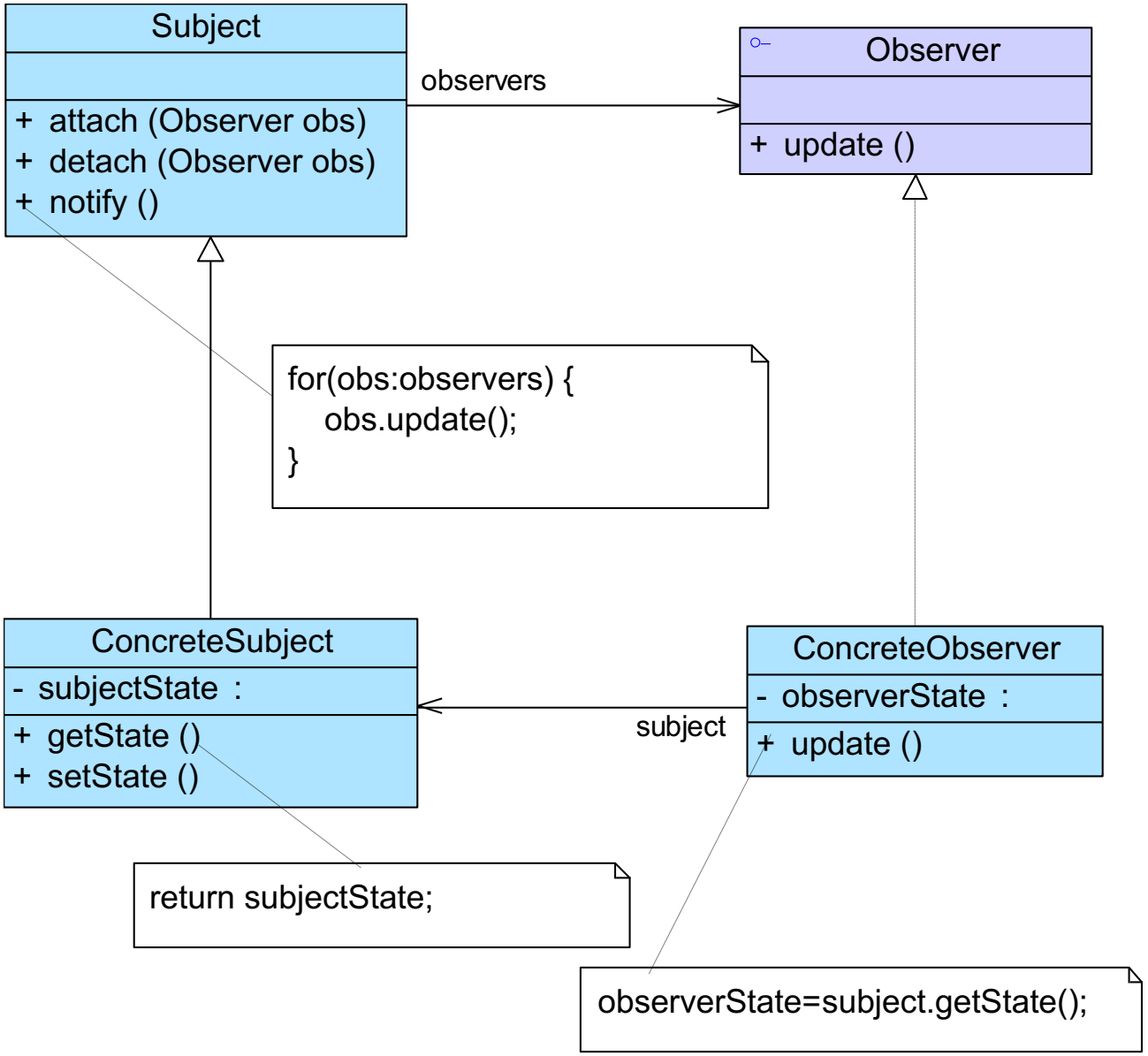
优点：为树形结构的面向对象实现提供了一种灵活的设计方案

1）清楚地定义分层次的复杂对象

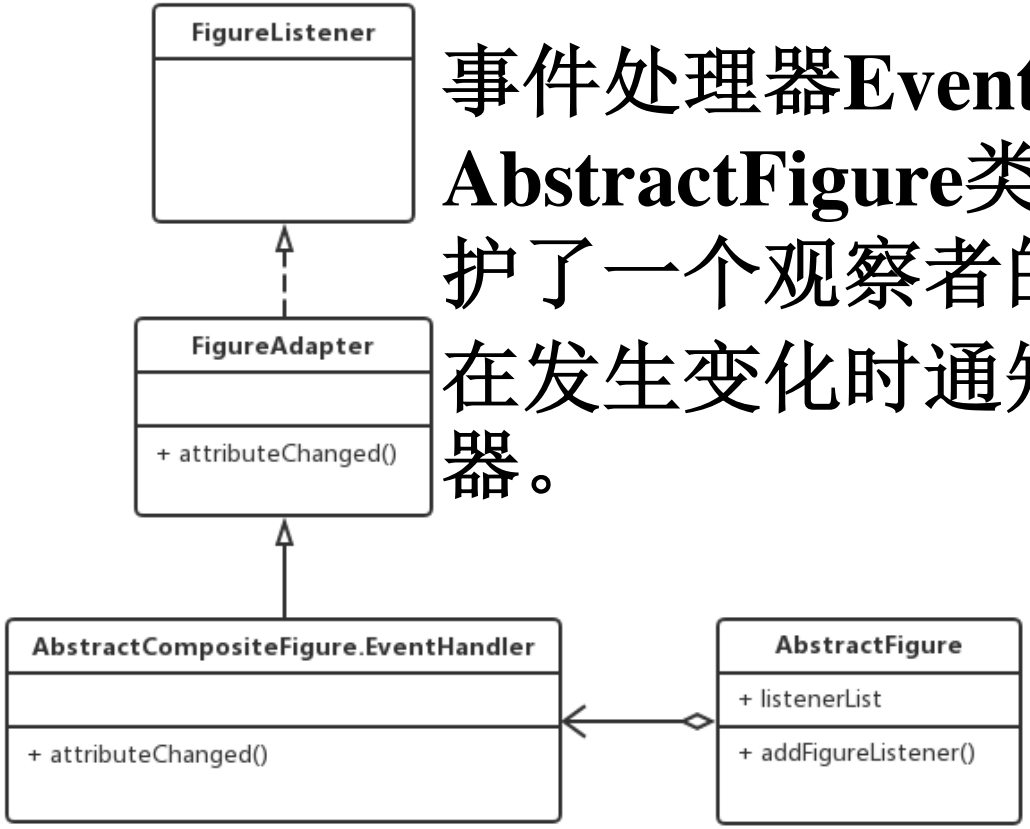
2）客户端可以一致地处理复合对象和原子对象

3）方便扩展新的复合或原子对象

观察者模式


Subject
+ attach (Observer obs)
+ detach (Observer obs)
+ notify ()
observers
Observer
+ update ()
for(obs:observers) {
obs.update();
}
ConcreteSubject
- subjectState :
+ getState ()
+ setState ()
subject
ConcreteObserver
- observerState :
+ update ()
return subjectState;
observerState=subject.getState();

JHotDraw中的观察者模式应用



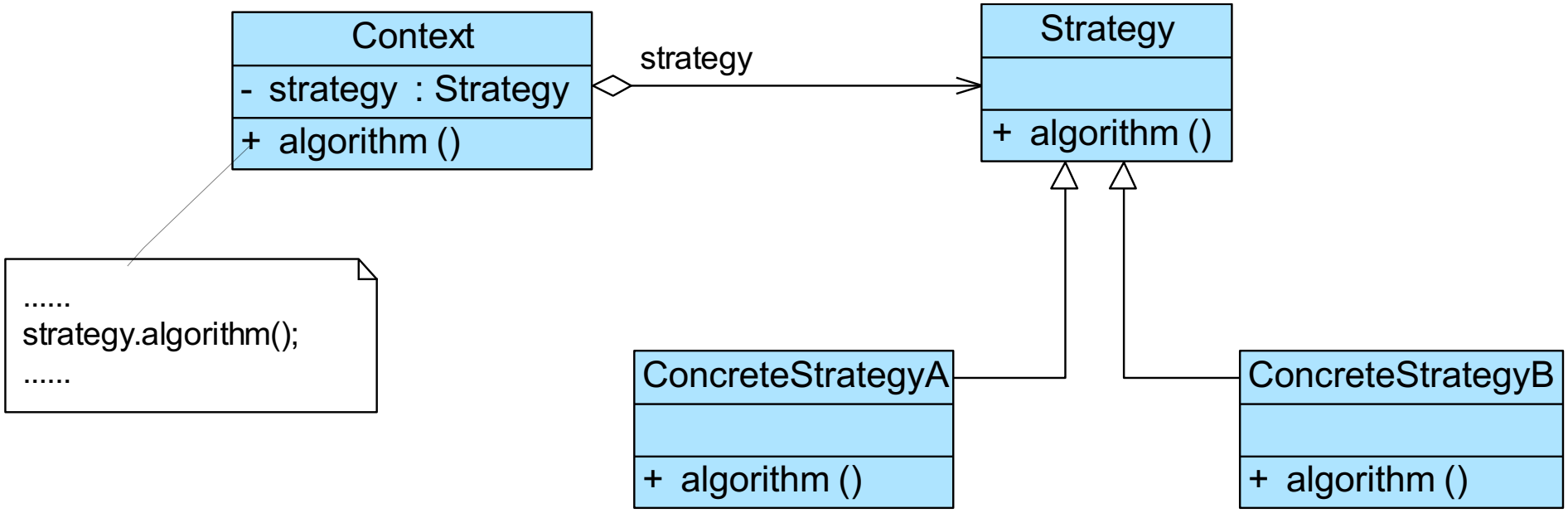
事件处理器EventHandler类是观察者；AbstractFigure类是被观察者，内部维护了一个观察者的列表（listenerList），在发生变化时通知当前注册的事件处理器。

优点：表示层（观察者）和数据逻辑层（被观察者）相分离

1）被观察者和观察者之间建立一个抽象的耦合，被观察者不依赖于具体的观察者

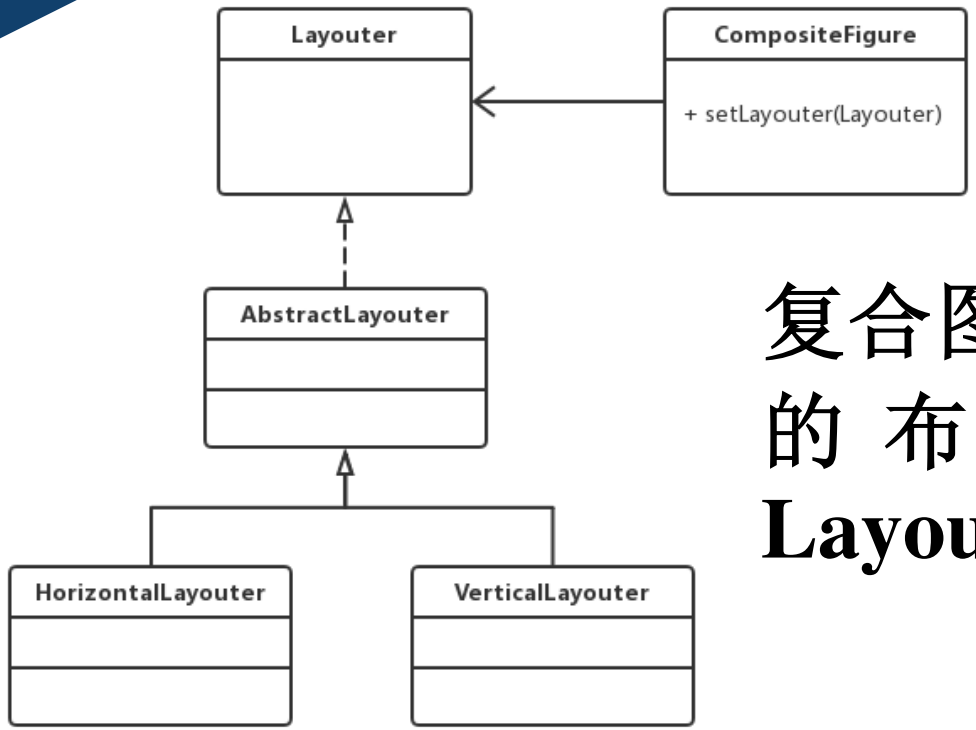
2）被观察者对观察者进行广播通信，简化了一对多的系统设计

策略模式



对算法的行为（具体处理策略）进行抽象，使得客户端代码不再依赖于具体的算法而是依赖于抽象的策略，在此基础上可以方便地对算法族进行管理。

JHotDraw中的策略模式应用



复合图形（CompositeFigure）的布局依赖于抽象的Layouter。

优点：将算法与其使用环境分离

1）算法可以独立被复用

2）避免了客户端代码中的多重条件选择语句

3）可灵活实现布局算法替换，还可以通过继承AbstractLayouter增加新的布局算法。

JHotDraw中的策略模式应用

```java
public interface Layouter {

    /**
     * Calculate the layout for the figure and all its subelements. The
     * layout is not actually performed but just its dimensions are calculated.
     *
     * @param anchor start point for the layout
     * @param lead minimum lead point for the layout
     */
    public Rectangle2D.Double calculateLayout(CompositeFigure compositeFigure, Point2D.Double anchor, Point2D.Double lead);

    /**
     * Method which lays out a figure. It is called by the figure
     * if a layout task is to be performed. Implementing classes
     * specify a certain layout algorithm in this method.
     *
     * @param anchor start point for the layout
     * @param lead minimum lead point for the layout
     */
    public Rectangle2D.Double layout(CompositeFigure compositeFigure, Point2D.Double anchor, Point2D.Double lead);
}
```

Layouter接口的定义

JHotDraw中的策略模式应用

```
public Dimension2DDouble getPreferredSize() {
    if (this.layouter != null) {
        Rectangle2D.Double r = layouter.calculateLayout(this, getStartPoint(), getEndPoint());
        return new Dimension2DDouble(r.width, r.height);
    } else {
        return super.getPreferredSize();
    }
}

public void layout() {
    // Note: We increase and below decrease the changing depth here,
    //       because we want to ignore change events from our children
    //       why we lay them out.
    changingDepth++;
    for (Figure child : getChildren()) {
        if (child instanceof CompositeFigure) {
            CompositeFigure cf = (CompositeFigure) child;
            cf.layout();
        }
    }
    changingDepth--;
    if (getLayouter() != null) {
        Rectangle2D.Double bounds = getBounds();
        Point2D.Double p = new Point2D.Double(bounds.x, bounds.y);
        Rectangle2D.Double r = getLayouter().layout(
                this, p, p);
        setBounds(new Point2D.Double(r.x, r.y), new Point2D.Double(r.x + r.width, r.y + r.height));
        invalidate();
    }
}
```

CompositeFigure类中应用了Layouter算法的两个方法

阅读建议

- 《代码大全》5.3

快速阅读后整理问题
在QQ群中提出并讨论

CS2001
软件工程

End

11. 软件设计
—设计框架