第11章 基于模式的设计

11.1 设计模式

- 设计模式：
  - 是用来描述问题以及解决方案的规范性方法，在某种程度上允许软件工程界获取设计知识，使解决方案能够得到重用。
  - 每个模式都描述了在我们所处环境内反复出现的问题，然后描述该问题的核心解决方案，这样你就可以几百万次地重复使用该解决方案，而不必用同样的方式重复工作两次。

11.1 设计模式（续）

- 设计描述可以描述为“表示上下文、问题和解决方案”三者之间关系的三部分规则。
- 只有适合问题所处环境的解决方案才是有效的。

- Coplien用以下的方式描述了有效的设计模式的特点：
  - 设计模式可以解决问题，不只是抽象的原则或策略
  - 设计模式是已经得到验证的概念，借助于踪迹记录捕捉解决方案。
  - 解决方案并不明显，试图从基本原理导出方案，最好的模式会间接产生问题的解决方案。
  - 设计模式描述关系，不止描述模块，且描述系统结构和机制。
  - 模式具有显著的人性化元素（将人工干预降到最少），改善人的生活质量，具有美学和实用性。

11.1.1 模式的种类

- 无生产力(nongenerative)的模式
  - 只描述问题和背景，但不提供明确的解决方案。
- 有生产力(generative)的模式
  - 识别能描述系统中重要的和可重复方面的模式
- 有时我们称之为生产性，“就是连续应用几种模式，每种模式都封装自己的问题和影响因素，更大的解决方案是以一些较小的解决方案的形式间接地表现出来的”

11.1.1 模式的种类（续）

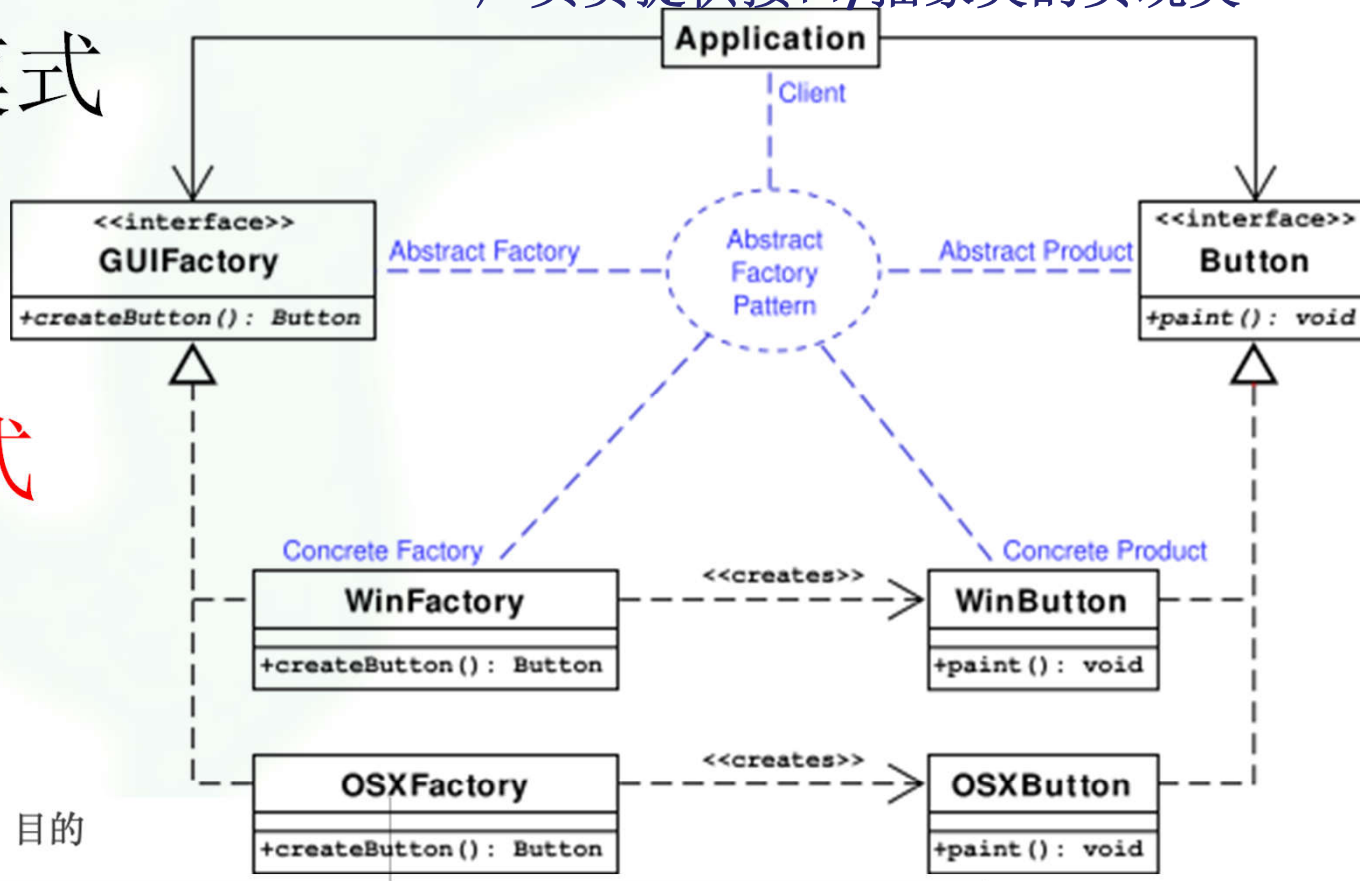
1. 体系结构模式
   - 描述了很多可以用结构化方法解决的设计问题
2. 数据模式
   - 描述了重现的面向数据的问题以及涌来解决这些问题的数据建模解决方案
3. 构件模式
   - 涉及与开发子系统和构件相关的问题、它们之间相互通信的方式以及它们在一个较大的体系中的位置。
4. 界面设计模式
   - 描述公共用户界面问题及具有影响因素的解决方案。
5. WebApp模式
   - 解决WebApp时遇到的问题，而且往往包括很多前面提到的一些其他模式。
6. 移动模式
   - 描述在开发移动平台的解决方案时碰到的问题。

11.1.1 模式的种类（续）

高层只引用接口/抽象类，抽象工厂负责提供接口/抽象类的实现类

- 一、创建型模式

1. 抽象工厂模式



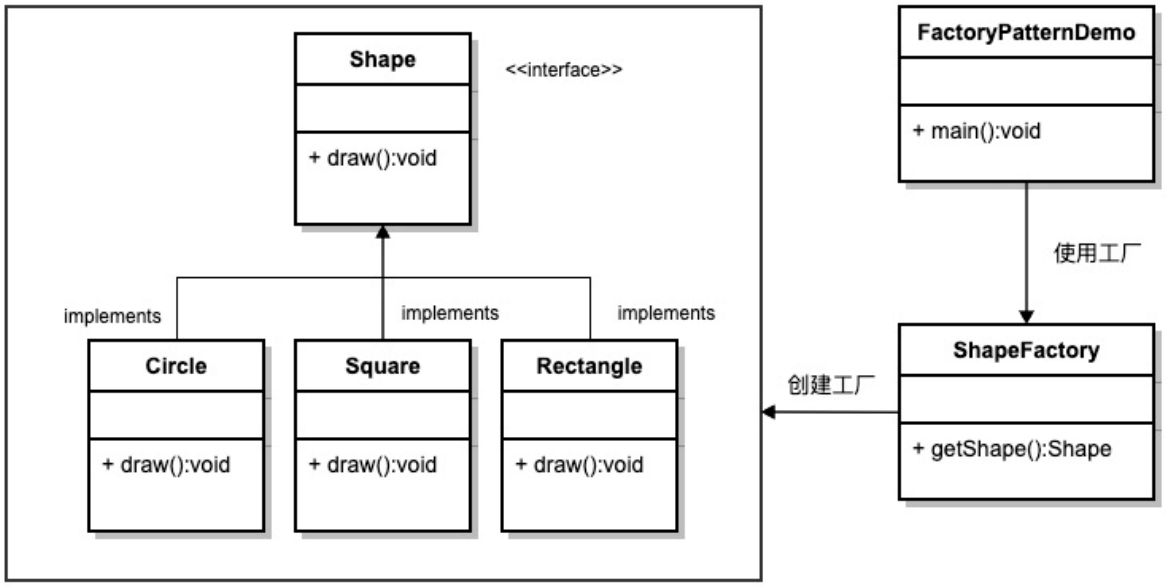
名称	目的	
抽象工厂（Abstract factory）	• 提供一个接口，用于创建相关或从属对象的族，而不需要指定它们的具体类。通过选择具体的工厂，就选定了一族对象。	• 它提供了多个工厂方法。

使用场景： 1、QQ 换皮肤，一整套一起换。2、生成不同操作系统的程序。

一、创造型模式

• 2.工厂方法模式

- 定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。

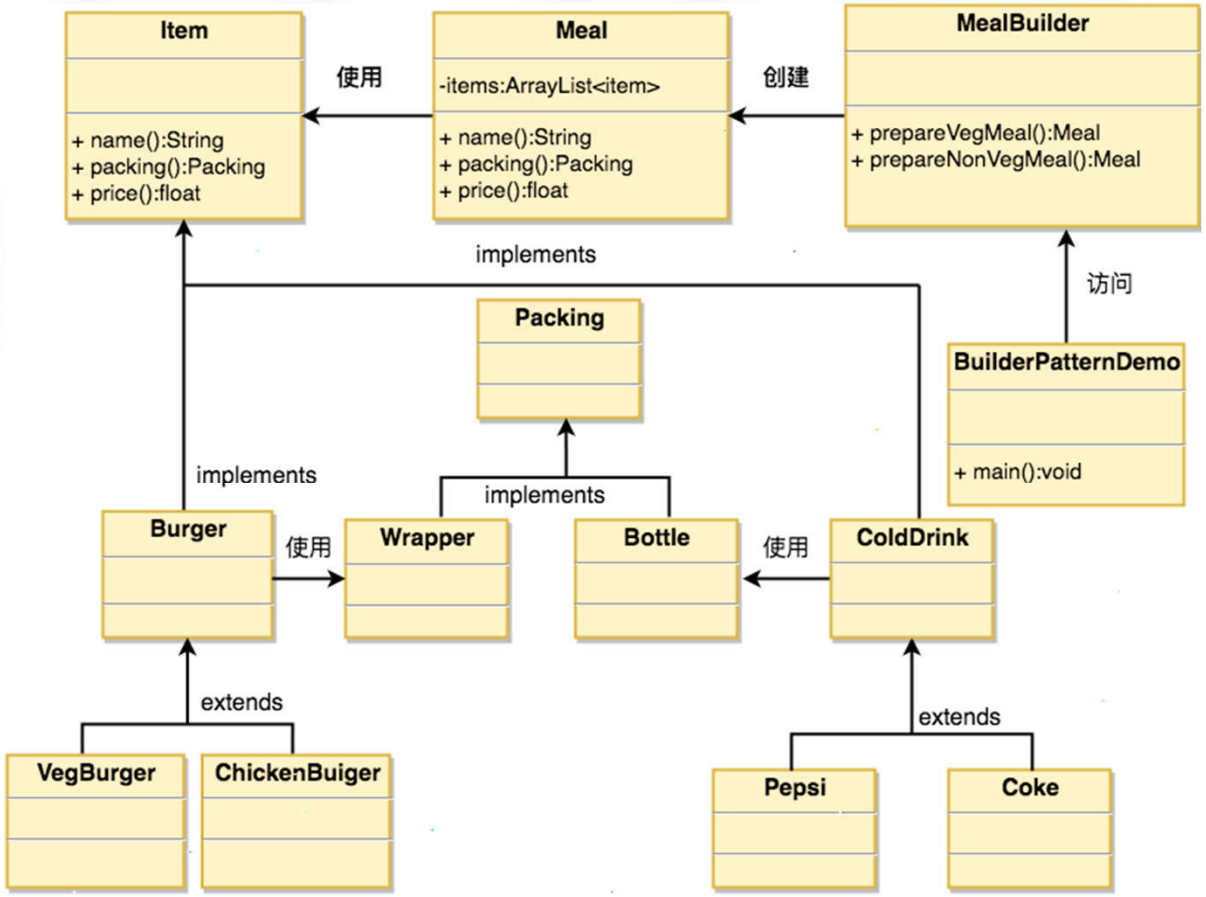


FactoryPatternDemo 类使用 *ShapeFactory* 来获取 *Shape* 对象。它将向 *ShapeFactory* 传递信息（*CIRCLE / RECTANGLE / SQUARE*），以便获取它所需对象的类型。

一、创造型模式

• 3.生成器模式

- 将一个复杂对象的构建与其表示相分离，使得同样的构建过程可以创建不同的表示；主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作

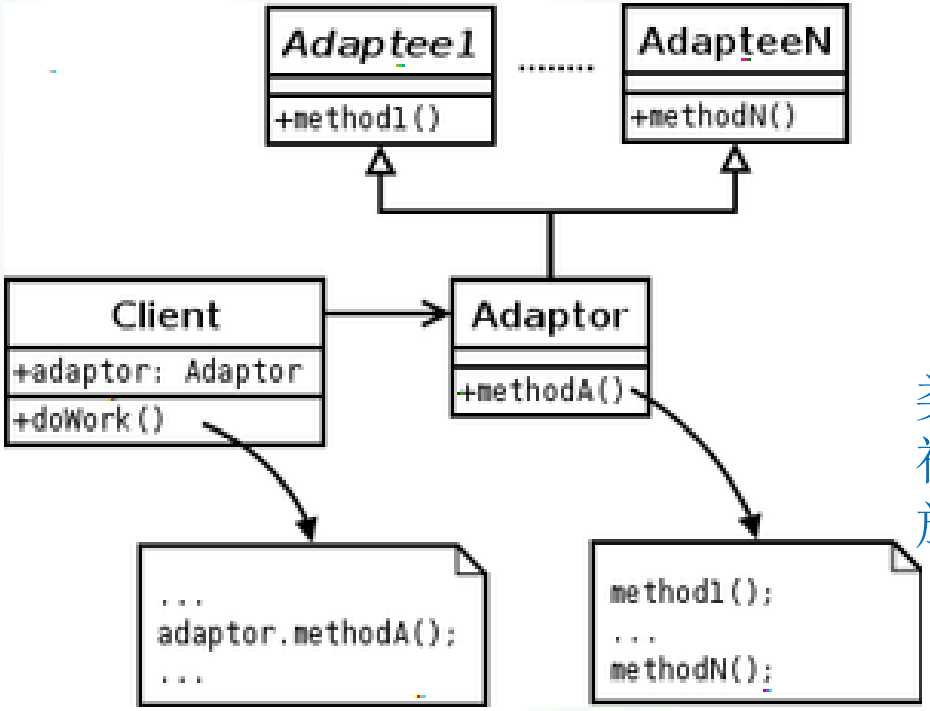


应用实例： 去肯德基，汉堡、可乐、薯条、炸鸡翅等是不变的，而其组合是经常变化的，生成出所谓的"套餐"。

二、结构型模式

• 1.适配器模式

- 将类的接口转换为客户机期望的其他接口。适配器允许类一起工作，否则由于接口不兼容而无法一起工作

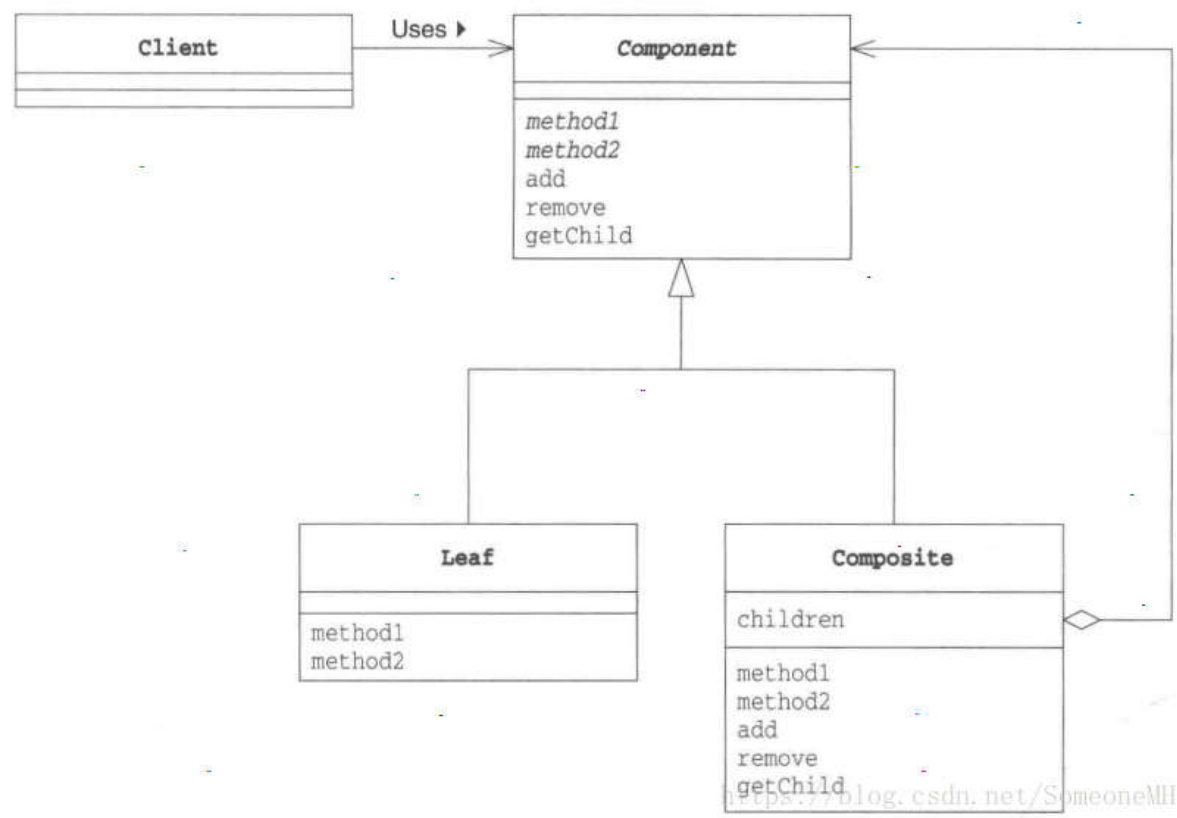


类1可以播放音乐，类2可以播放视频，类3集成类1和类2既可以播放音乐也可以播放视频。

二、结构型模式

- 2.复合模式

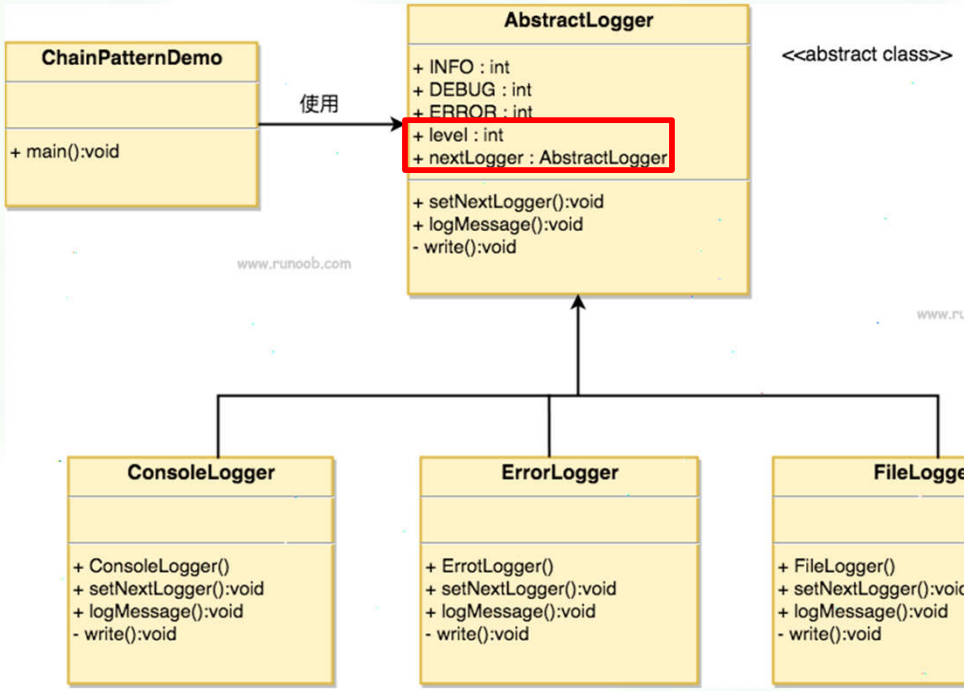
- 一个具有同样接口的处理对象的树结构模式。



三、行为型模式

• 1.责任链模式

- 通过给多个对象处理请求的机会，避免将请求的发送者与其接收者耦合。链接接收对象并沿链传递请求，直到对象处理它。

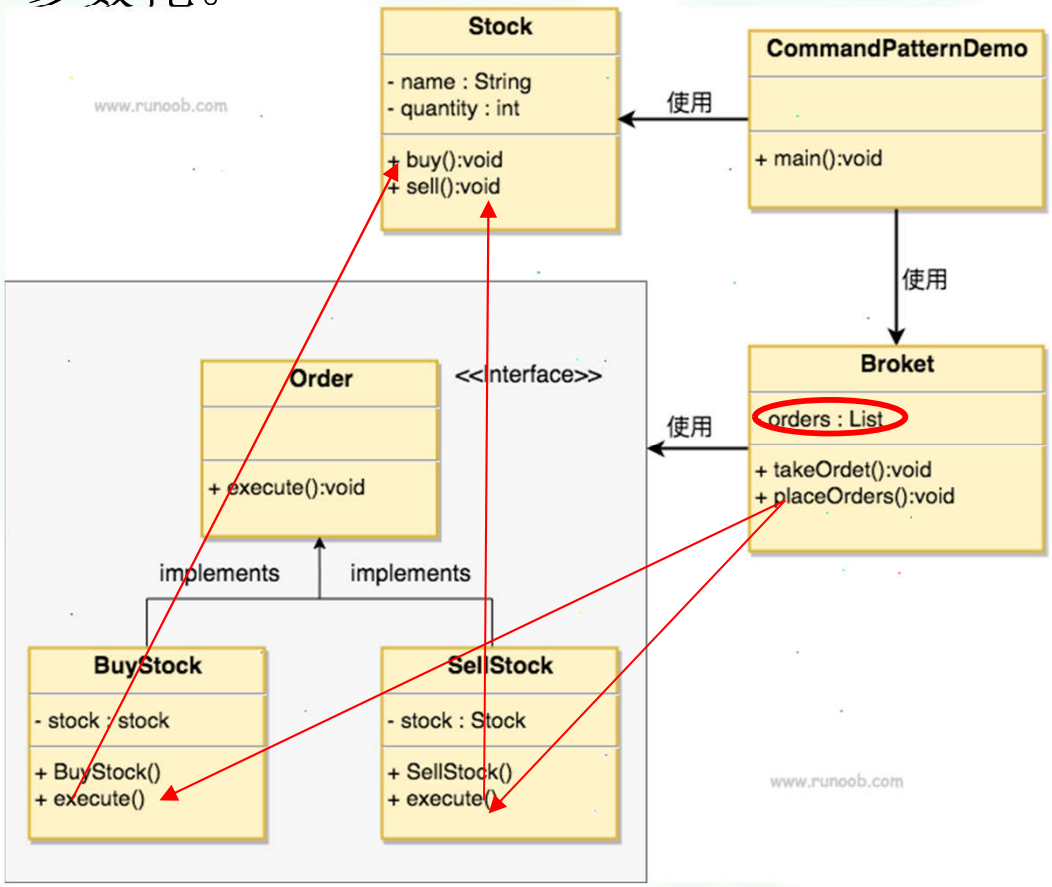


我们创建抽象类 *AbstractLogger*，带有详细的日志记录级别。然后我们创建三种类型的记录器，都扩展了 *AbstractLogger*。每个记录器消息的级别是否属于自己的级别，如果是则相应地打印出来，否则将不打印并把消息传给下一个记录器。

三、行为型模式

• 2.命令模式

- 将一个请求封装成一个对象，从而使您可以用不同的请求对客户进行参数化。



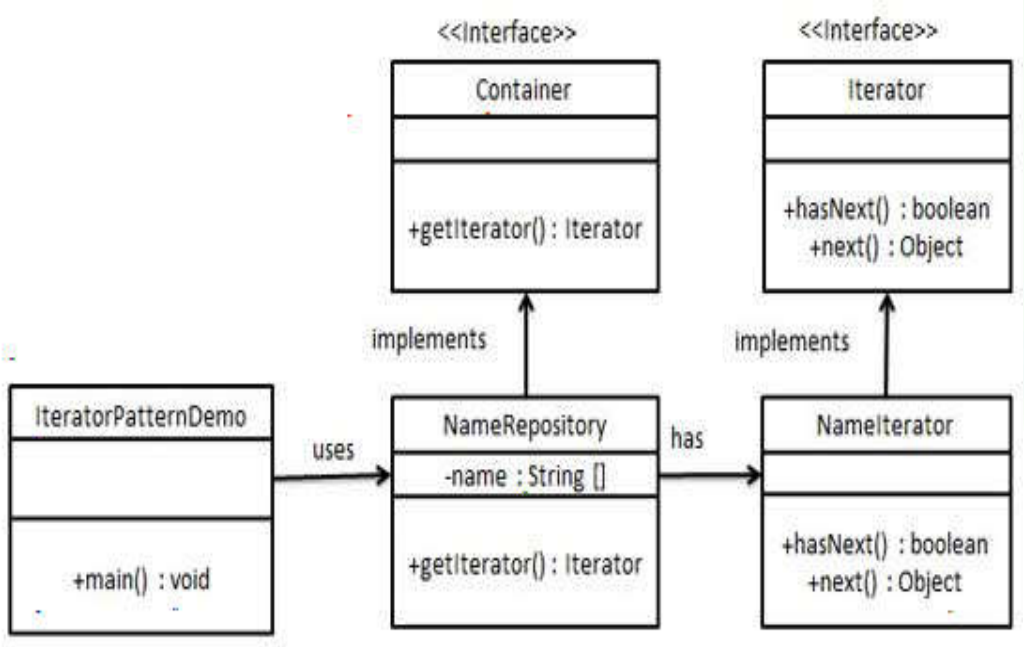
我们首先创建作为命令的接口 *Order*，然后创建作为请求的 *Stock* 类。实体命令类 *BuyStock* 和 *SellStock*，实现了 *Order* 接口，将执行实际的命令处理。创建作为调用对象的类 *Broket*，它接受订单并能下订单。

Broket 对象使用命令模式，基于命令的类型确定哪个对象执行哪个命令。*CommandPatternDemo* 类使用 *Broket* 类来演示命令模式。

三、行为型模式

- 3.迭代器模式

- 用于按顺序访问一个聚集对象的元素，而不必暴露其底层表示。



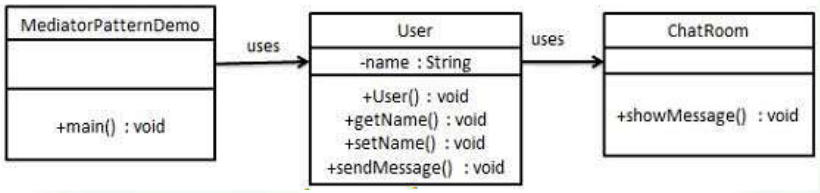
我们将创建一个叙述导航方法的 *Iterator* 接口和一个返回迭代器的 *Container* 接口。实现了 *Container* 接口的实体类将负责实现 *Iterator* 接口。

IteratorPatternDemo，我们的演示类使用实体类 *NamesRepository* 来打印 *NamesRepository* 中存储为集合的 *Names*。

三、行为型模式

- **4.中介者模式**

- 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式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。



多个用户可以向聊天室发送消息，聊天室向所有的用户显示消息。创建两个类 *ChatRoom* 和 *User*。*User* 对象使用 *ChatRoom* 方法来分享他们的消息。

ChatRoom.java

```
import java.util.Date;

public class ChatRoom {
   public static void showMessage(User user, String message){
      System.out.println(new Date().toString()
         + " [" + user.getName() +"] : " + message);
   }
}
```

User.java

```
public class User {
   private String name;

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }

   public User(String name){
      this.name  = name;
   }

   public void sendMessage(String message){
      ChatRoom.showMessage(this,message);
   }
}
```

使用 *User* 对象来显示他们之间的通信。

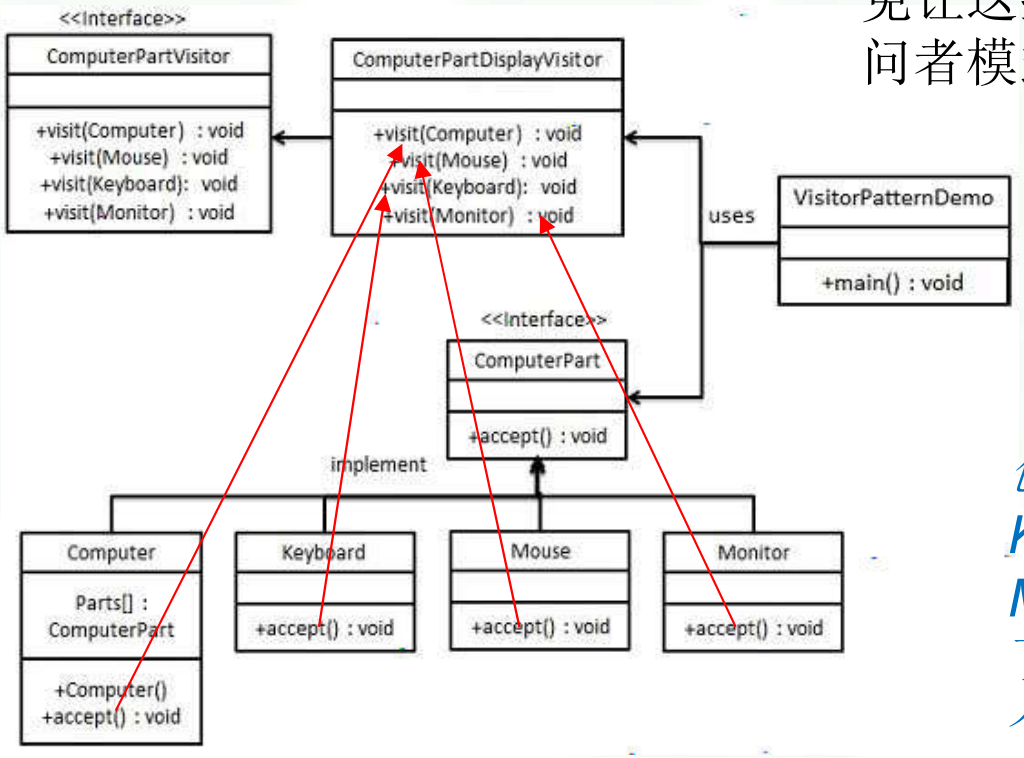
MediatorPatternDemo.java

```
public class MediatorPatternDemo {
   public static void main(String[] args) {
      User robert = new User("Robert");
      User john = new User("John");

      robert.sendMessage("Hi! John!");
      john.sendMessage("Hello! Robert!");
   }
}
```

三、行为型模式

• 5.访问者模式

主要将数据结构与数据操作分离。主要解决：稳定的数据结构和易变的操作耦合问题。何时使用：需要对一个对象结构中的对象进行很多不同的并且不相关的操作，而需要避免让这些操作“污染”这些对象的类，使用访问者模式将这些封装到类中。



创建一个接受操作的ComputerPart 接口。Keyboard、Mouse、Monitor 和 Computer 是实现了 ComputerPart 接口的实体类。定义另一个接口 ComputerPartVisitor，它定义了访问者类的操作。Computer 使用实体访问者来执行相应的动作。

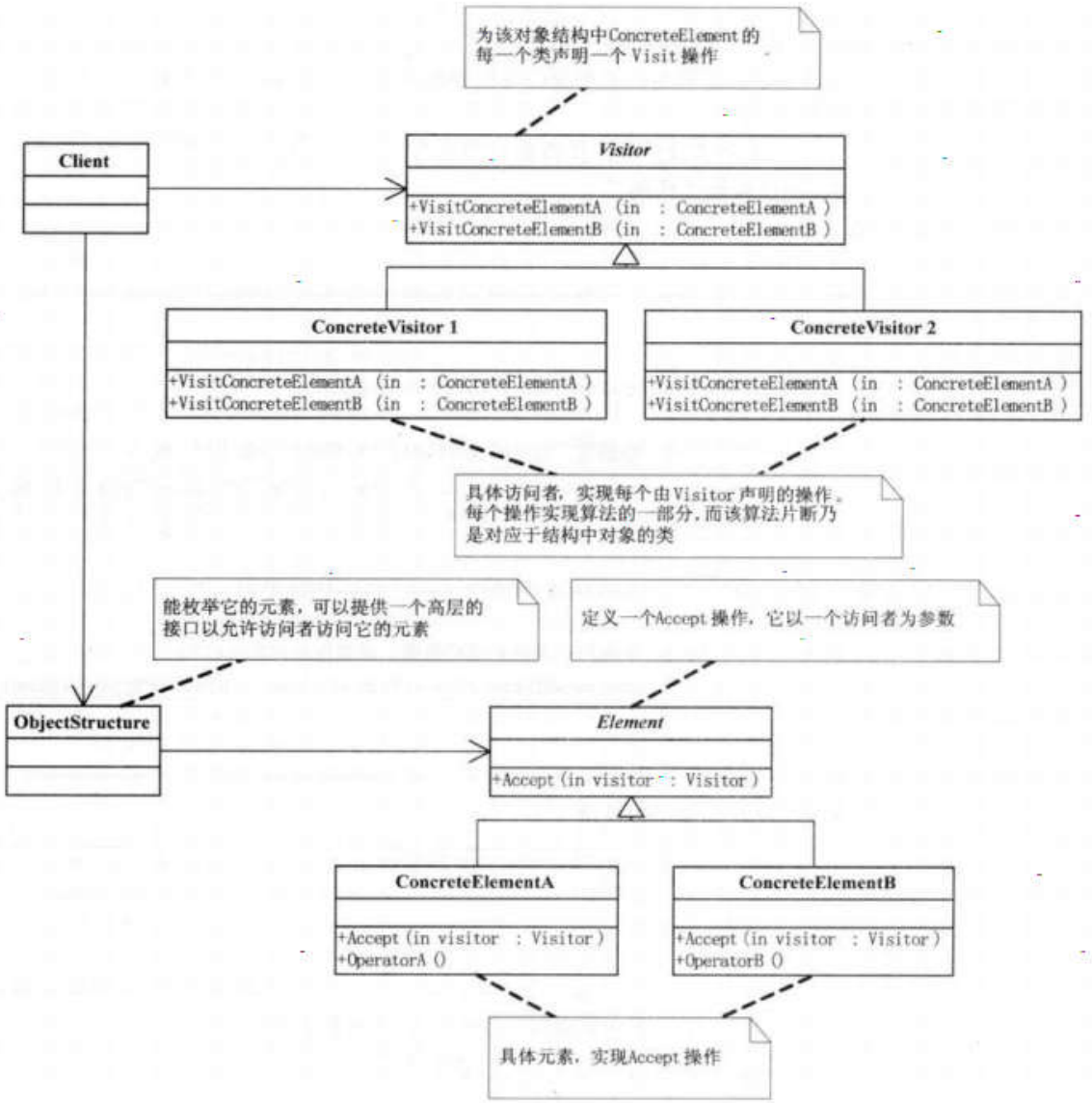
https://www.runoob.com/design-pattern/visitor-pattern.html

```
public class Keyboard implements ComputerPart {

    @Override
    public void accept(ComputerPartVisitor computerPartVisi
        computerPartVisitor.visit(this);
    }
}
```

```
public class Computer implements ComputerPart {

    ComputerPart[] parts;

    public Computer(){
        parts = new ComputerPart[] {new Mouse(), new Keyboard(), new Monitor()};
    }


    @Override
    public void accept(ComputerPartVisitor computerPartVisitor) {
        for (int i = 0; i < parts.length; i++) {
            parts[i].accept(computerPartVisitor);
        }
        computerPartVisitor.visit(this);
    }
}
```

访问者模式（Visitor），表示一个作用于某对象结构中的各元素的操作。它使你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。[DP]

访问者模式（Visitor）结构图

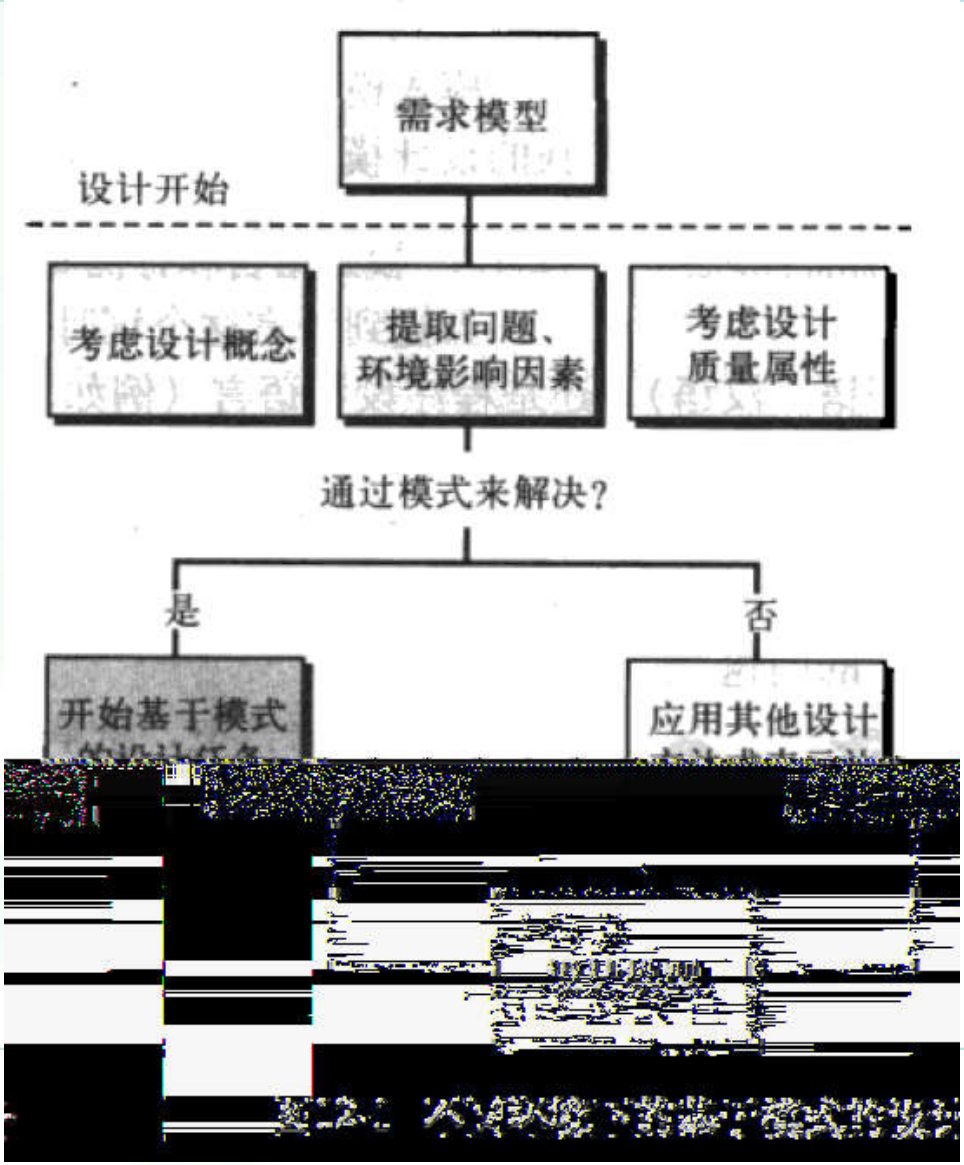


- **ObjectStructure结构对象角色**：对象结构，对象结构是一个抽象表述，具体点可以理解为一个具有容器性质或者复合对象特性的类，它会含有一组元素（Element），并且可以迭代这些元素，供访问者访问。

11.2 基于模式的软件设计

- 在整个设计过程中，（当模式符合设计要求时）应该利用一切机会来寻找现成的设计模式，而不是去创造新模式。

11.2.1 不同环境下基于模式的设计



11.2.2 用模式思考

- 为了使设计者用模式思考，Shalloway和Trott提出了下面的方法：

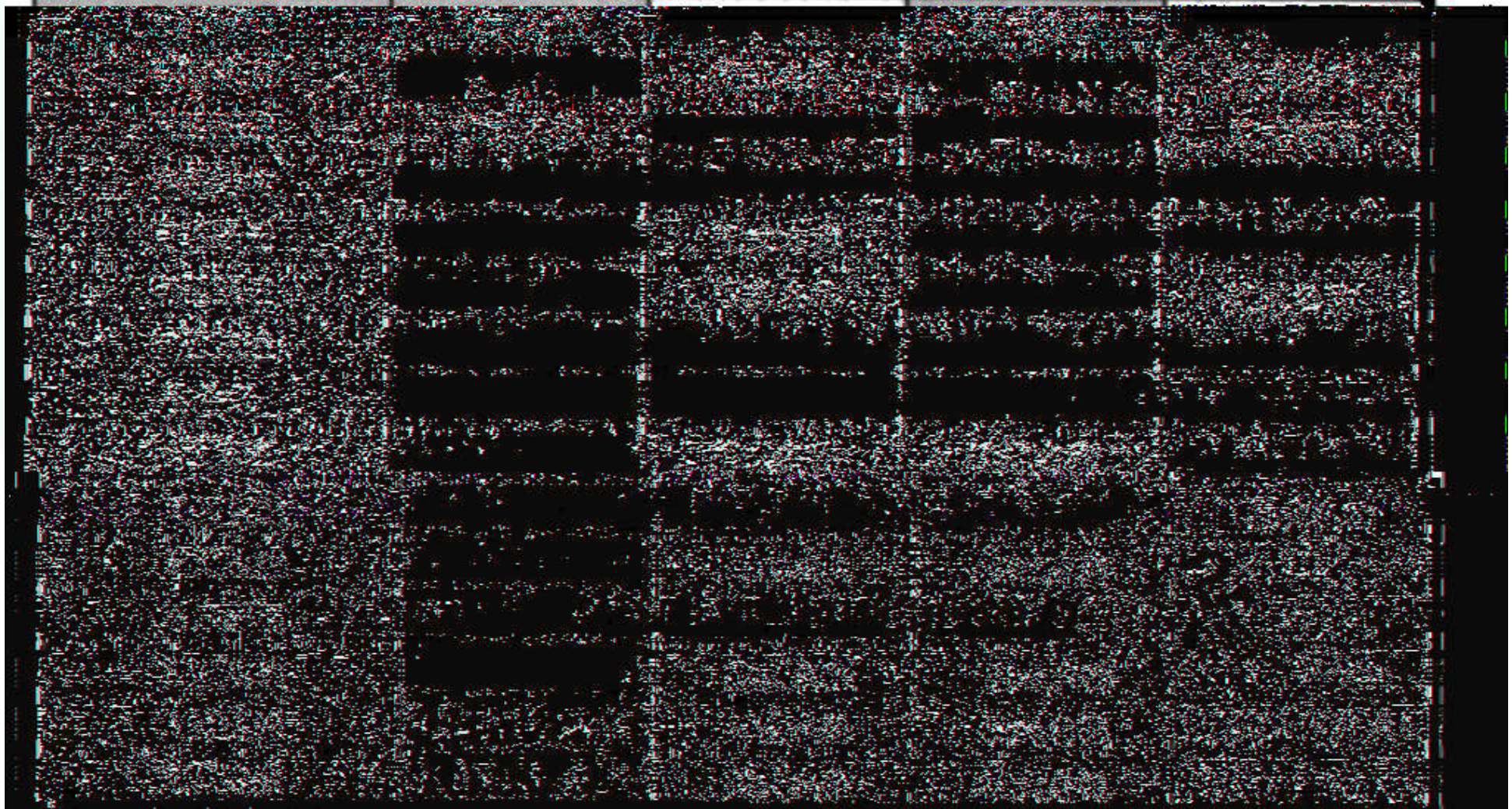
1. 保证理解全局。
2. 检查全局，提取在此抽象层上表示的模式。
3. 从“全局”模式开始设计，为将来的设计工作建立环境或架构。
4. “在环境的内部工作”是在更低的抽象层上寻找有助于设计方案的模式。
5. 重复步骤1～4，直到完成完整的设计。
6. 通过使每个模式适应将要构建的软件细节对设计进行优化。
7. 理解每个模式不是独立的实体是很重要的。

11.2.3 设计任务

- 使用基于模式的设计思想时，实施的设计任务如下：
  - 检查需求模型，并开发问题的层次结构
  - 针对问题域，确定是否已经开发了可靠的模式语言
  - 从宏观问题开始设计，确定是否存在使用的一个或多个体系结构模式
  - 利用为体系结构模式提供的协作，检查子系统或构件问题，并查找合适的模式解决这些问题。
  - 重复步骤2～4，直到所有的宏观问题都得到了解决
  - 如果已经分离出了用户界面设计（几乎总是这种情况），那么为了找到合适的模式，查找多个用户界面设计模式库
  - 当从模式推倒到设计后，使用设计质量标准作为参考，对设计进行优化。

11.2.4 建立模式组织表

	数据库	应用	实现	基础设施
数据/内容				



11.2.5 常见设计错误

- 在某些情况下，由于没有足够的时间去理解根本问题、问题所在环境及影响因素，所以，可能会选择看似正确但并不适合解决方案的模式。
- 一旦选择错误的模式就会决绝正式自己的错误，并强行适应模式。

可以避免错误吗？

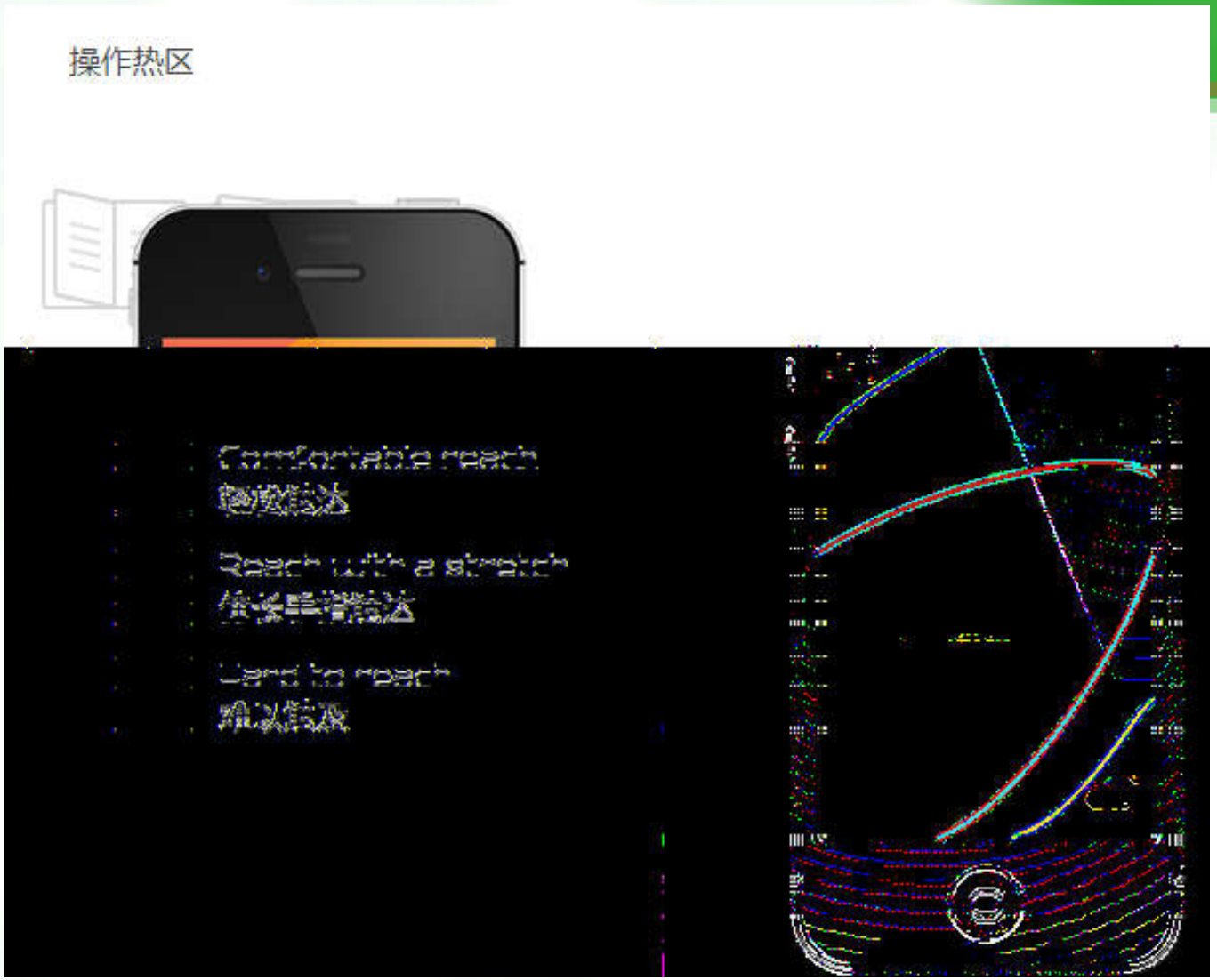
----工作评审

11.2.6 WebApp设计模式

- 设计焦点：标识着设计模式的哪个方面是相关的
- 可以按以下设计焦点的级别对WebApp模式进行分类：
  - 信息体系结构模式
  - 导航模式
  - 交互模式
  - 表示模式
  - 功能模式

11.2.7 移动App模式

- 从本质上说，移动APP都是关于界面的。在许多情况下，移动用户界面模式在各种不同的应用程序类别中表示为一组“最佳的”屏幕图像。
- 典型的例子：
  - 登陆屏幕
  - 地图
  - 漂浮选单
  - 注册流程
  - 自定义选项卡导航
  - 邀请



移动APP的常见的几种导航形式 http://www.jianshu.com/p/aafd9d25bfc3

详细设计说明书举例

- 详细设计说明书模版
- 详细设计说明书举例
- 详细设计说明书举例2
- 详细设计说明书举例3