程序设计模式作业四

2020年3月8日　20:48

设计模式概述作业

作业要求：

1、设计模式有哪些优点？

2、请查阅相关资料，了解在JDK类库设计中使用了哪些设计模式，在何处使用了何种模式？（至少2个）

3、除了设计模式之外，目前有不少人在从事“反模式”的研究，请查阅相关资料，了解何谓“反模式”，以及研究“反模式”的意义。

- 学生笔记及作业整理在Github
  https://github.com/baobaotql/CCNU_DesignPattern

- **Q1：设计模式有哪些优点？**
- A1：

  首先我们要明确为什么要使用设计模式？根据我们已有的软件工程学科基础，我们可以明确到设计模式是要:

  i. 解决面临一般问题的解决方案
  ii. 保证代码得到可复用性
  iii. 让代码更加的规范和清晰化

  设计模式分为三种类型:

  i. 创建型模式（在创建对象时封装一部分业务逻辑，而不是简单的new）
     - 工厂模式
     - 抽象工厂模式
     - 单例模式
     - 建造者模式
     - 原型模式
  ii. 结构型模式（更关注与对象和类之间的组合关系）
     - 适配器模式
     - 桥接模式
     - 过滤器模式
     - 组合模式
     - 装饰器模式
     - 外观模式
     - 享元模式
     - 代理模式
  iii. 行为型模式（更关注对象之间的通信关系）
     - 适配器模式
     - 桥接模式
     - 过滤器模式
     - 组合模式
     - 装饰器模式
     - 外观模式
     - 享元模式
     - 代理模式

共有二十三中设计模式.常用的模式有简单工厂模式 / 策略模式 / 装饰模式 / 代理模式 / 工厂方法模式 / 原型模式 / 模板方法模式

下面我们主要说明的是以上的常见模式的优缺点:

- 简单工厂模式
  - 优点:
    i. 工厂类是整个模式的关键。包含了必要的逻辑判断，根据外界给定的信息，决定究竟应该创建哪个具体类的对象。
    ii. 通过使用工厂类，外界可以从直接创建具体产品对象的尴尬局面摆脱出来，仅仅需要负责“消费”对象就可以了。而不必管这些对象究竟如何创建及如何组织的。明确了各自的职责和权利，有利于整个软件体系结构的优化。

  - 缺点:
    i. 由于工厂类集中了所有实例的创建逻辑，违反了高内聚责任分配原则，将全部创建逻辑集中到了一个工厂类中，它所能创建的类只能是事先考虑到的，如果需要添加新的类，则就需要改变工厂类了。
    ii. 当系统中的具体产品类不断增多时候，可能会出现要求工厂类根据不同条件创建不同实例的需求。这种对条件的判断和对具体产品类型的判断交错在一起，很难避免模块功能的蔓延，对系统的维护和扩展非常不利。
    iii. 这些缺点在工厂方法模式中得到了一定的克服。

- 策略模式
  - 优点:
    i. 每个算法单独封装，减少了算法和算法调用者的耦合。
    ii. 合理使用继承有助于提取出算法中的公共部分。
    iii. 简化了单元测试。

  - 缺点
    i. 策略模式只适用于客户端知道所有的算法或行为的情况。
    ii. 策略模式造成很多的策略类，每个具体策略类都会产生一个新类。不过可以使用享元模式来减少对象的数量。

- 装饰模式
  - 优点:
    i. Decorator模式与继承关系的目的都是要扩展对象的功能，但是Decorator可以提供比继承更多的灵活性。
    ii. 通过使用不同的具体装饰类以及这些装饰类的排列组合，设计师可以创造出很多不同行为的组合。

  - 缺点:
    i. 这种比继承更加灵活机动的特性，也同时意味着更加多的复杂性。
    ii. 装饰模式会导致设计中出现许多小类，如果过度使用，会使程序变得很复杂。
    iii. 装饰模式是针对抽象组件（Component）类型编程。但是，如果你要针对具体组件编程时，就应该重新思考你的应用架构，以及装饰者是否

合适。当然也可以改变Component接口，增加新的公开的行为，实现“半透明”的装饰者模式。在实际项目中要做出最佳选择。

- 代理模式
  - 优点:
    i. 职责清晰，真实的角色就是实现实际的业务逻辑，不用关心其他非本职责的事务，通过后期的代理完成一件完成事务，附带的结果就是编程简洁清晰。
    ii. 代理对象可以在客户端和目标对象之间起到中介的作用，这样起到了中介的作用和保护了目标对象的作用。
    高扩展性

  - 缺点:
    i. 在客户端和目标对象增加一个代理对象，会造成请求处理速度变慢。
    ii. 增加了系统的复杂度。

- 工厂方法模式
  - 优点:
    i. 良好的封装性，代码结构清晰，减少模块间的耦合。
    ii. 工厂方法模式的扩展性非常优秀。
    iii. 屏蔽产品类。
    iv. 工厂方法模式是典型的解耦框架。

  - 缺点:
    i. 使用者必须知道相应工厂的存在。
    ii. 每次增加一个产品时，都需要增加一个具体类和对象实现工厂，是的系统中类的个数成倍增加，在一定程度上增加了系统的复杂度，同时也增加了系统具体类的依赖。

- 原型模式
  - 优点:
    i. 隐藏了对象创建的细节。
    ii. 提高了性能。
    iii. 不用重新初始化，动态获得对象运行时的状态。

  - 缺点:
    i. 适用性不是很广。
    ii. 每一个类必须配备一个克隆方法。
    iii. 配备克隆方法需要对类的功能进行通盘考虑，这对于全新的类不是很难，但对于已有的类不一定很容易，特别当一个类引用不支持串行化的间接对象，或者引用含有循环结构的时候。

- 模板方法模式
  - 优点:
    i. 提高代码复用性。
    ii. 帮助子类摆脱重复的不变行为

- 缺点:

i. 考虑不全面统一出现问题

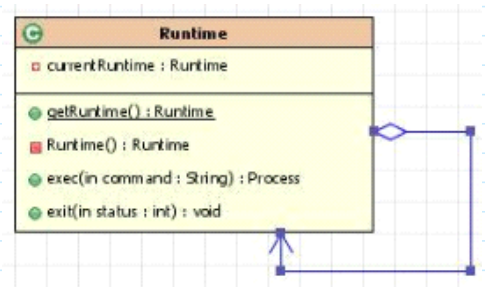
- **A2：请查阅相关资料，了解在JDK类库设计中使用了哪些设计模式，在何处使用了何种模式？（至少2个）**

  例举设计模式使用场景的例子已经标亮

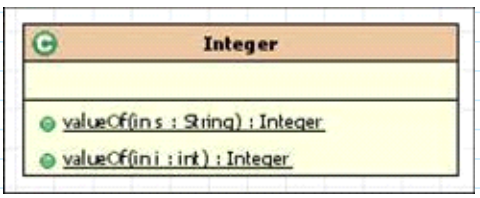
  Q2:

  - Singleton（单例）
    - 作用：

      保证类只有一个实例；提供一个全局访问点
    - JDK中体现：
      1) Runtime
      2) NumberFormat
    - 类图：



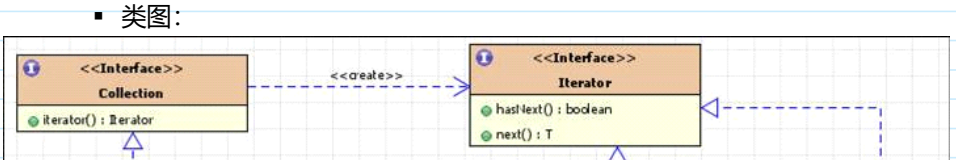
  - Factory（静态工厂）
    - 作用：
      1) 代替构造函数创建对象
      2) 方法名比构造函数清晰
    - JDK中体现：
      1) Integer.valueOf
      2) Class.forName
    - 类图：

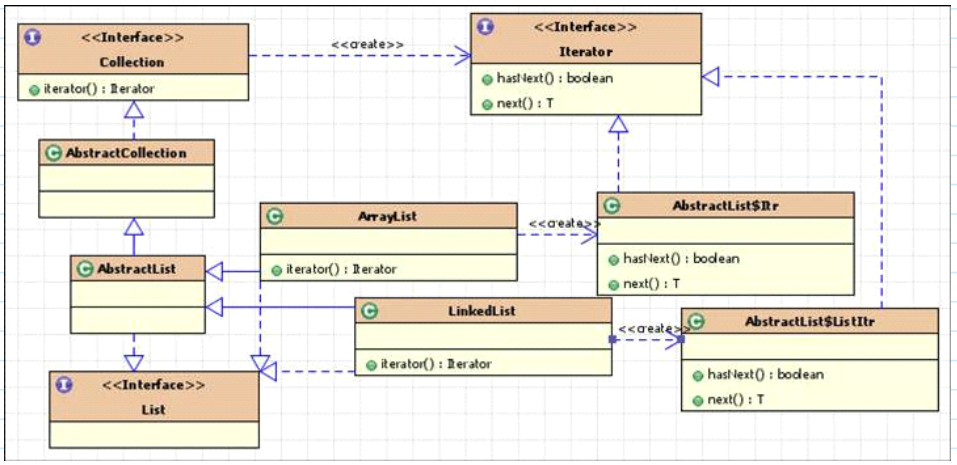


  - Factory Method（工厂方法）
    - 作用：

      子类决定哪一个类实例化
    - 使用场景：
      1) 日志记录器
      2) 数据库访问
    - JDK中体现：

      Collection.iterator方法
    - 类图：

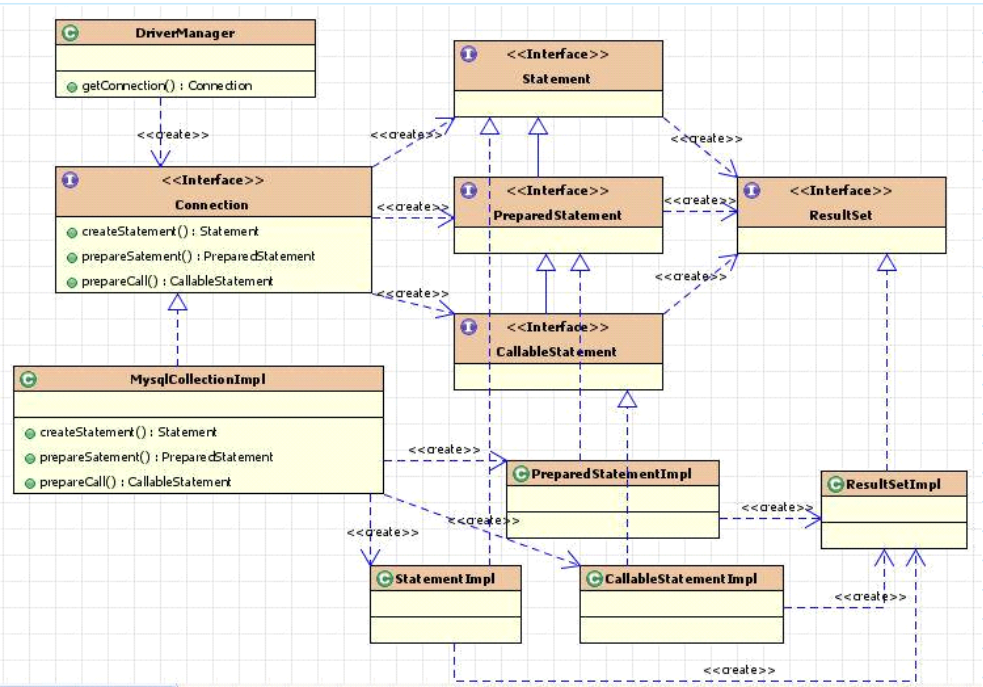




- Abstract Factory（抽象工厂）
  - 作用：

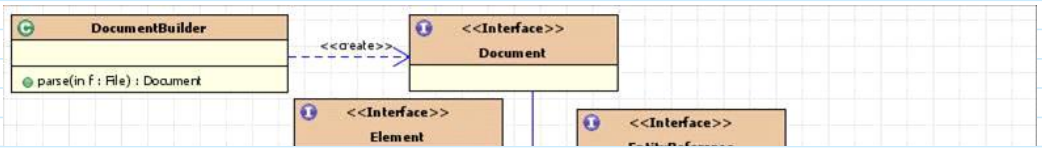
    创建某一种类的对象
  - 使用场景：

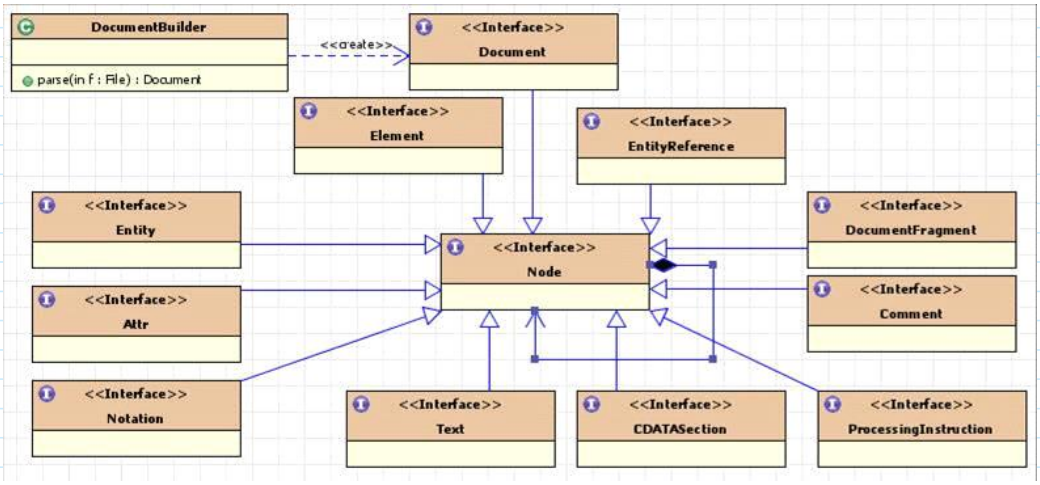
    当系统的对象是一个系列的时候使用抽象工厂模式
  - JDK中体现：
    1) java.sql包
    2) UIManager（swing外观）
  - 类图：



- Builder（构造者）
  - 作用：
    1) 将构造逻辑提到单独的类中
    2) 分离类的构造逻辑和表现
  - JDK中体现：

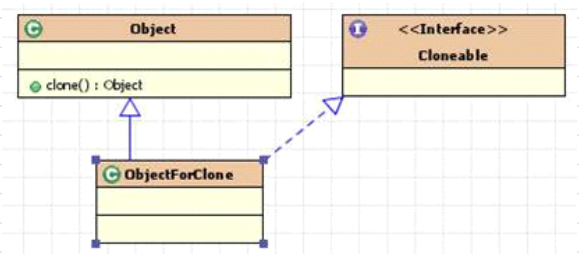
    DocumentBuilder(org.w3c.dom)
  - 类图：





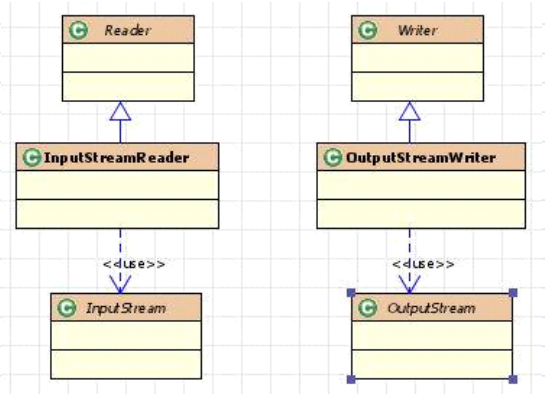
- Prototype（原型）
  - 作用：
    1) 复制对象
    2) 浅复制、深复制
  - 使用场景：
    1) 当一个对象可能各个调用者都需要修改其值的时候
    2) 当需要实例化的一些对象是在类中指定的，动态加载，反射
    3) 类的初始化需要消耗很多资源，需要提高性能和安全性时
    4) 当不需要调用构造函数时
    5) 通过new产生对象，需要繁琐的数据准备和访问权限
    6) 解耦，不能让调用时都返回同一个对象，这样耦合度很高
  - JDK中体现：

    Object.clone；Cloneable
  - 类图：



- Adapter（适配器）
  - 作用：

    使不兼容的接口相容
  - 使用场景：
    1) 系统需要使用现有的类，而此类的接口并不是所需要的
    2) 有需要的修改一个正在运行的接口
  - JDK中体现：
    1) java.io.InputStreamReader(InputStream)
    2) java.io.OutputStreamWriter(OutputStream)
  - 类图：

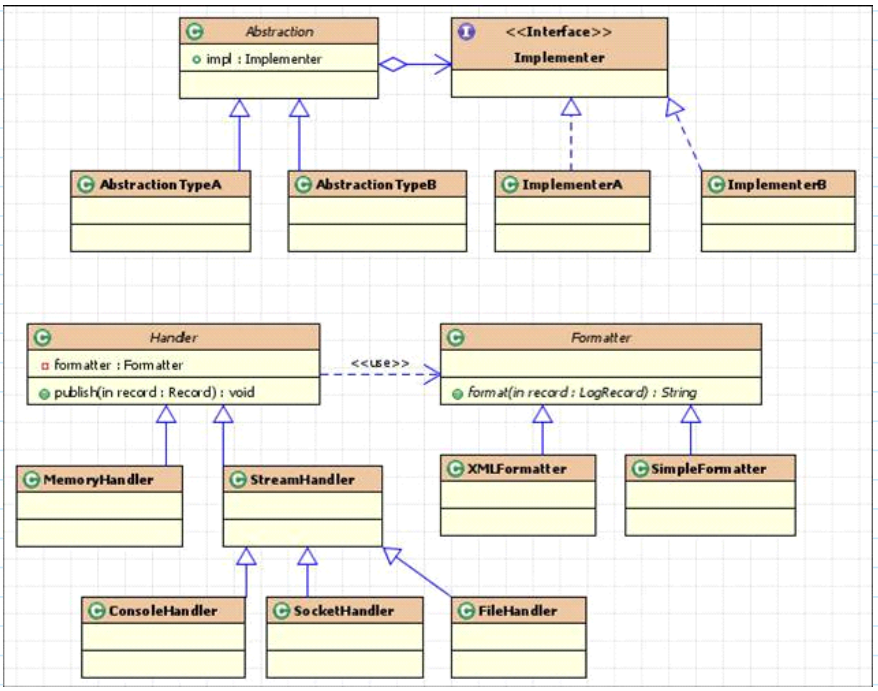


- Bridge（桥接）
  - 作用：

    将抽象部分与其实现部分分离，使它们都可以独立地变化
  - 使用场景：

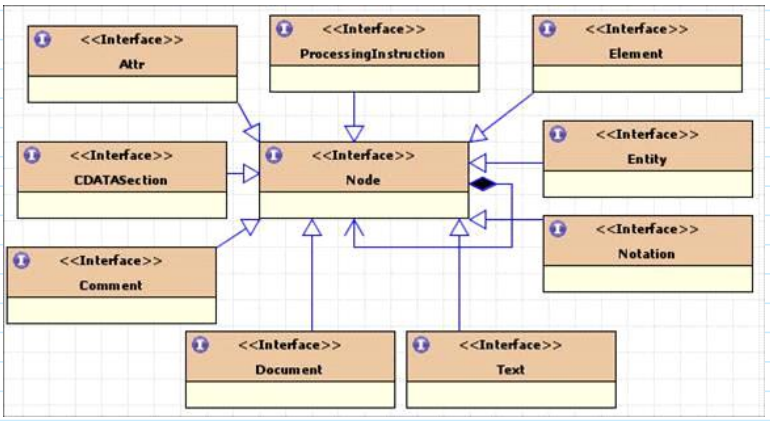
    在对象可能会有多种变化的情况下，需要特定的解耦
  - JDK中体现：

    java.util.logging中的Handler和Formatter
  - 类图：



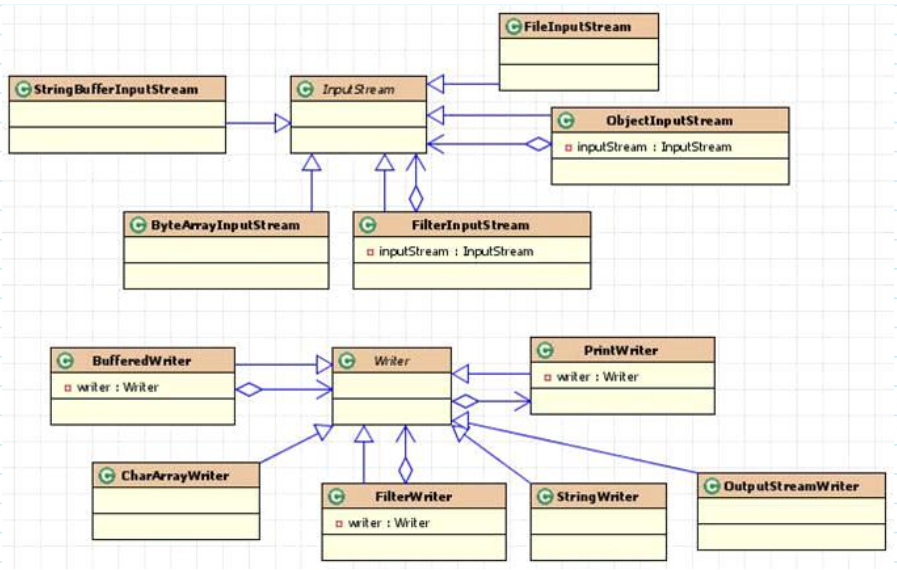
- Composite（组合）
  - 作用：

    一致地对待组合对象和独立对象
  - JDK中体现：
    1) org.w3c.dom
    2) javax.swing.JComponent#add(Component)
  - 类图：



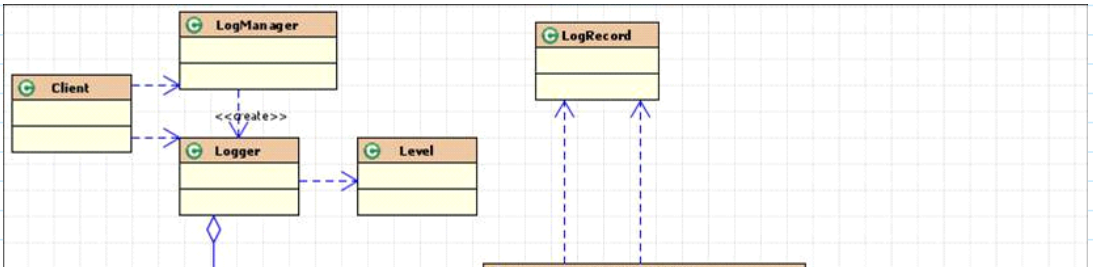
- Decorator（装饰器）
  - 作用：

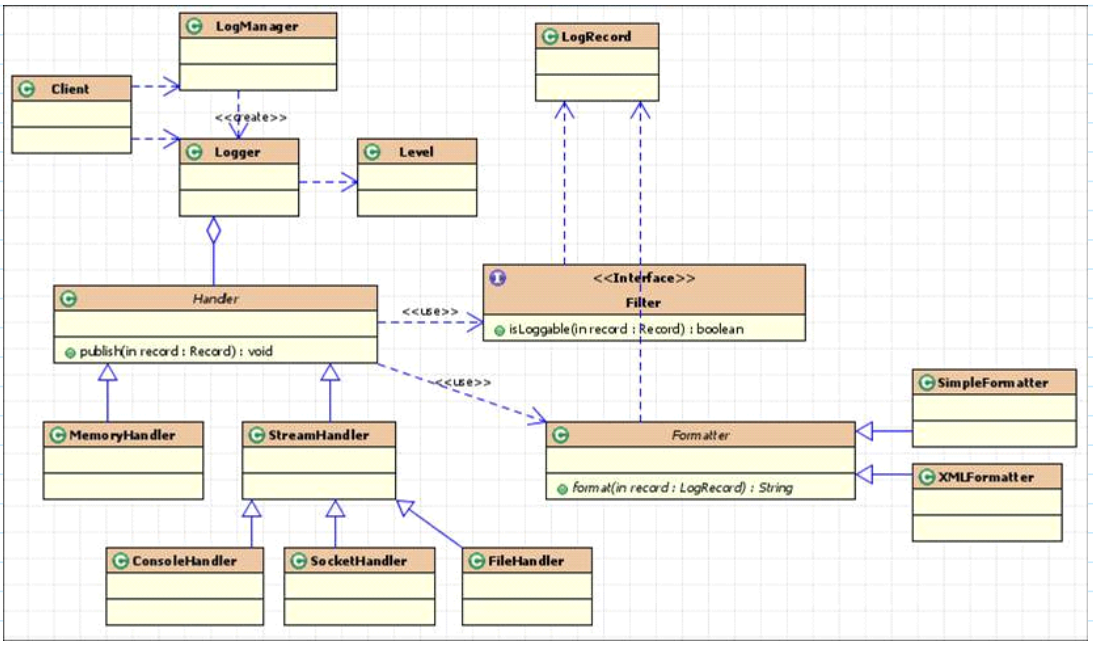
    为类添加新的功能；防止类继承带来的爆炸式增长
  - 使用场景
    1) 不想增加子类的时候使用扩展类
    2) 需要给类动态添加功能
  - JDK中体现：
    1) java.io包
    2) java.util.Collections#synchronizedList(List)
  - 类图：



- Facade（外观）
  - 作用：
    1) 封装一组交互类，一致地对外提供接口
    2) 封装子系统，简化子系统调用
  - JDK中体现：

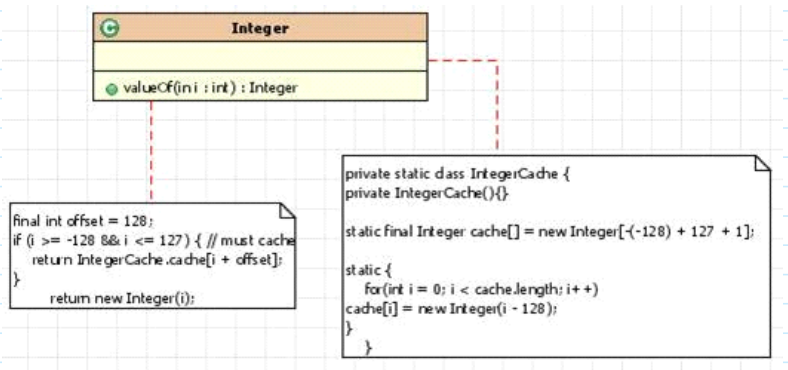
    java.util.logging包
  - 类图：





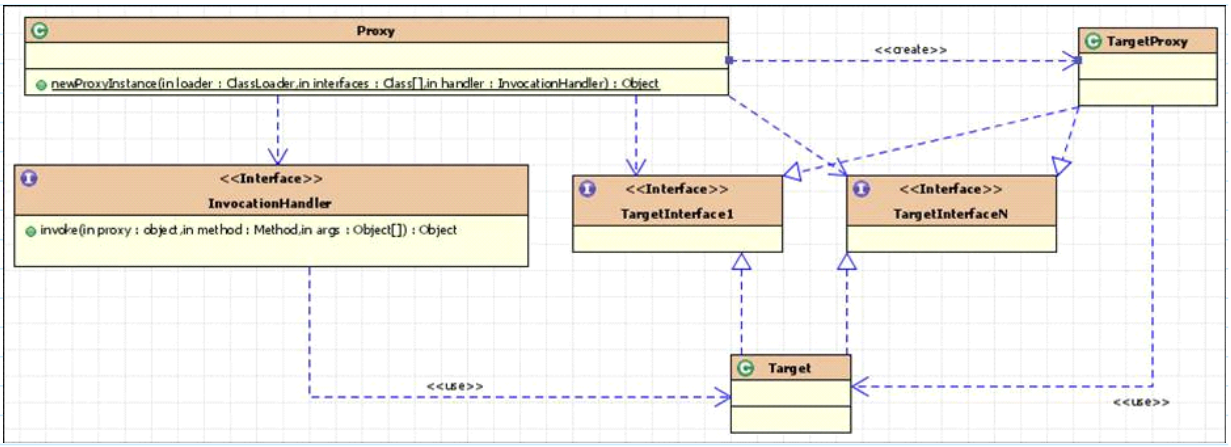
- Flyweight（享元）
  - 作用：

    共享对象，节省内存
  - 使用场景：
    1) 在访问一个类时做一些控制
    2) 远程、保护、虚拟、cache代理
  - JDK中体现：
    1) Integer.valueOf(int i)；Character.valueOf(char c)
    2) String常量池
  - 类图：



- Proxy（代理）
  - 作用：
    1) 透明调用被代理对象，无须知道复杂实现细节
    2) 增加被代理类的功能
  - JDK中体现：

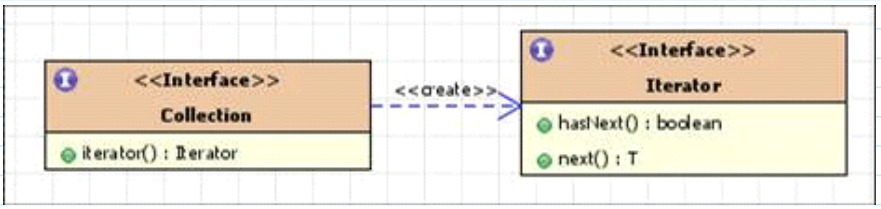
    动态代理；RMI
  - 类图：



- Iterator（迭代器）
  - 作用：

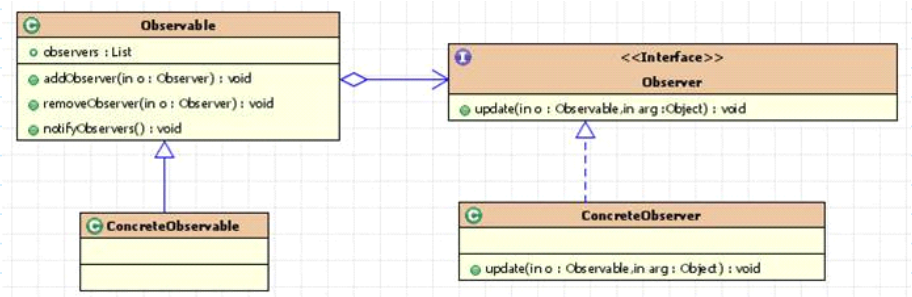
    将集合的迭代和集合本身分离
  - JDK中体现

    Iterator、Enumeration接口
  - 类图：



- Observer（观察者）
  - 作用：

    通知对象状态改变
  - JDK中体现：
    1) java.util.Observer,Observable
    2) Swing中的Listener
  - 类图：



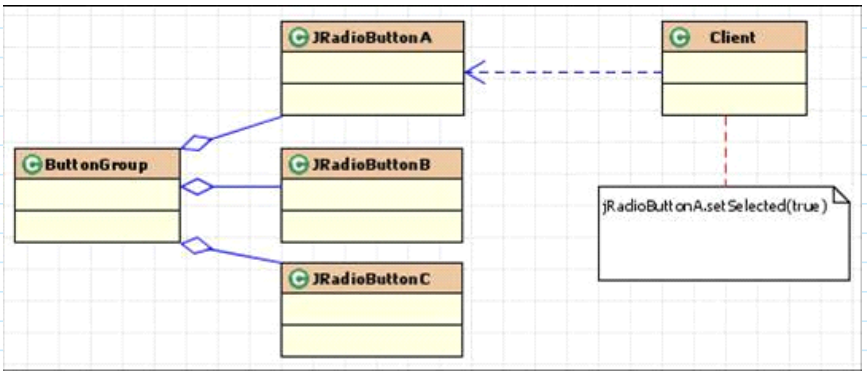
- Mediator（协调者）
  - 作用：

    用于协调多个类的操作
  - 使用场景：

    多个类相互耦合
  - JDK中体现：

    Swing的ButtonGroup

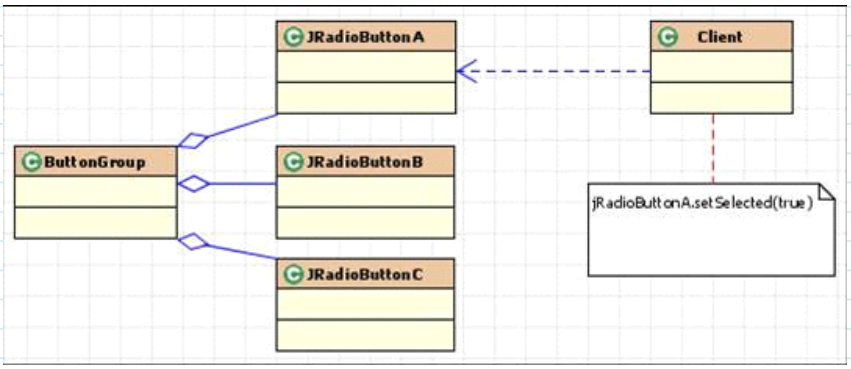
  类图：



- Template method（模板方法）
  - 作用：

    定义算法的结构，子类只实现不同的部分
  - JDK中体现：

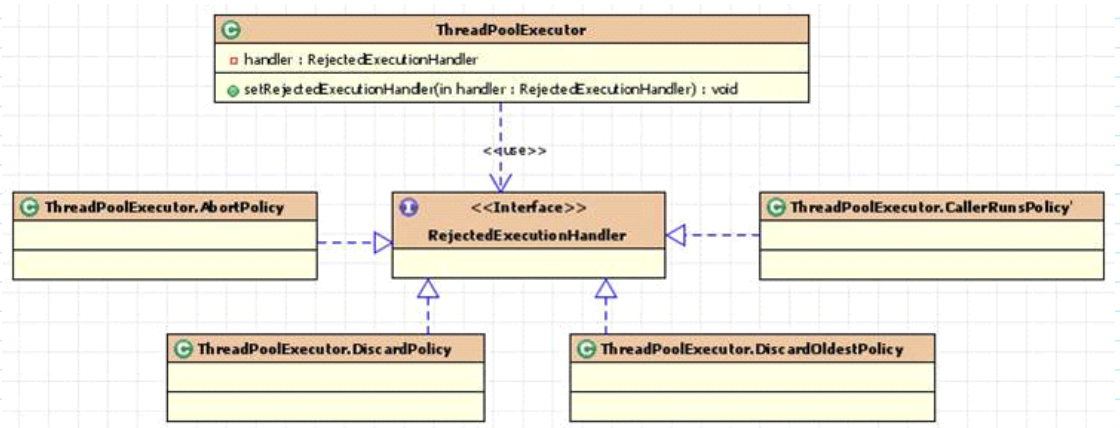
    ThreadPoolExecutor.Worker
  - 类图：



- Strategy（策略）
  - 作用：

    提供不同的算法
  - JDK中的体现：

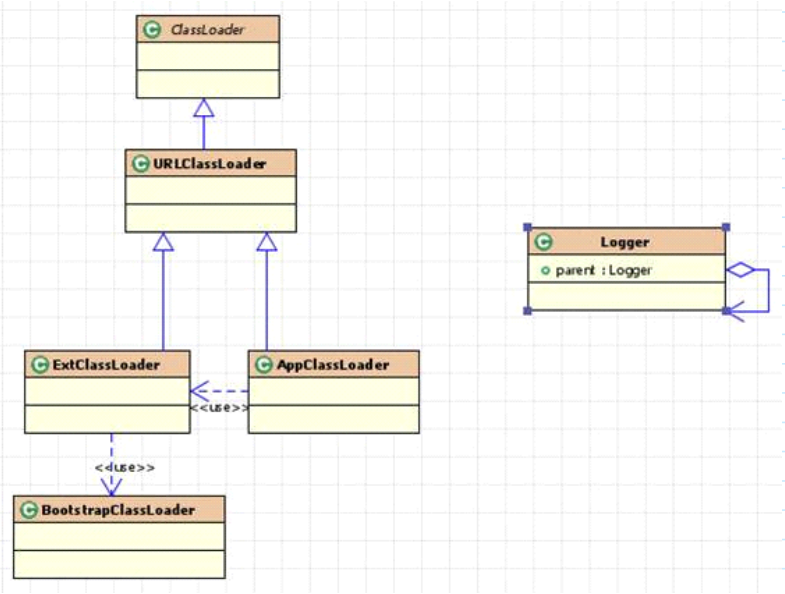
    ThreadPoolExecutor中的四种拒绝策略
  - 类图：



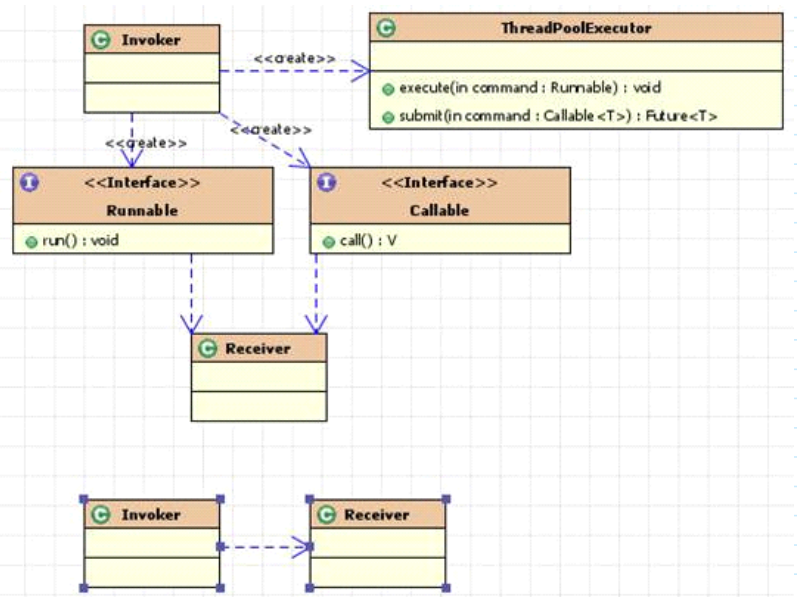
- Chain of Responsibility（责任链）
  - 作用：

    请求会被链上的对象处理，但是客户端不知道请求会被哪些对象处理

- 使用场景：
  1）js中的冒泡事件
  2）spring，jsp，servlet中的拦截器
  3）日志等级
- JDK中体现：
  1）java.util.logging.Logger会将log委托给parent logger
  2）ClassLoader的委托模型
- 类图：



- Command（命令）
  - 作用：
    封装操作，使接口一致
    将调用者和接收者在空间和时间上解耦合
  - JDK中体现：
    Runnable；Callable；ThreadPoolExecutor
  - 类图：

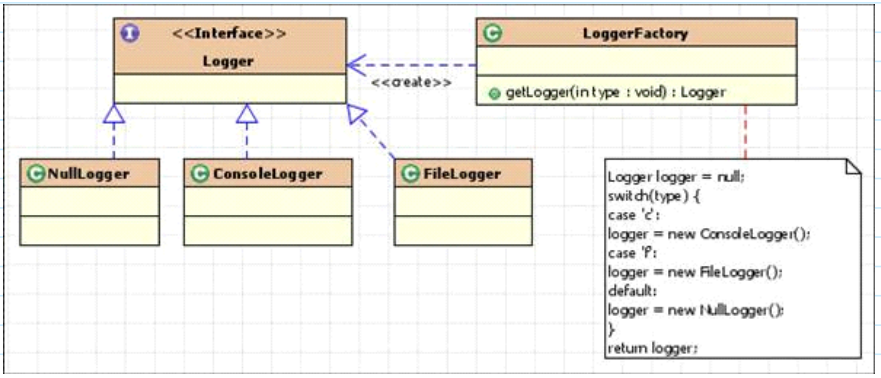


- Null Object（空对象）
  - 作用：
    不需每次判空，对待空值，如同对待一个相同接口的对象

- JDK中体现：

  Collections.EMPTY_LIST

- 类图：



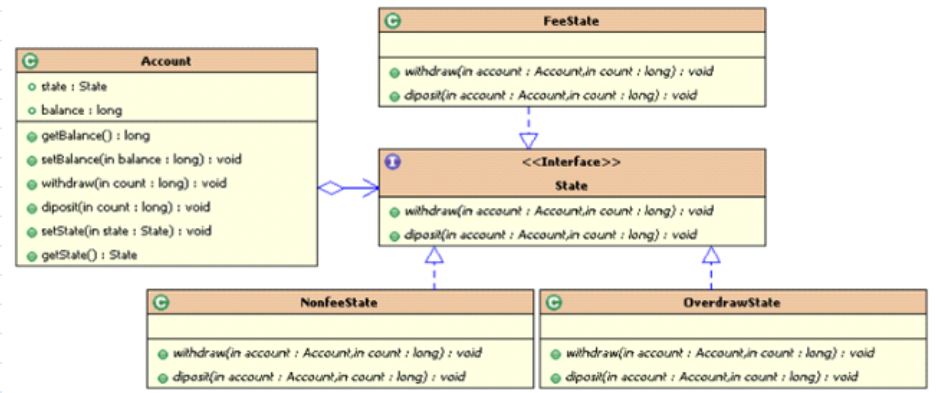
- State（状态）
  - 作用：

    将主对象和其状态分离，状态对象负责主对象的状态转换，使主对象类功能减轻

  - JDK中体现：

    未发现

  - 类图：



- Visitor（访问者）
  - 作用：

    异构的类间添加聚合操作；搜集聚合数据

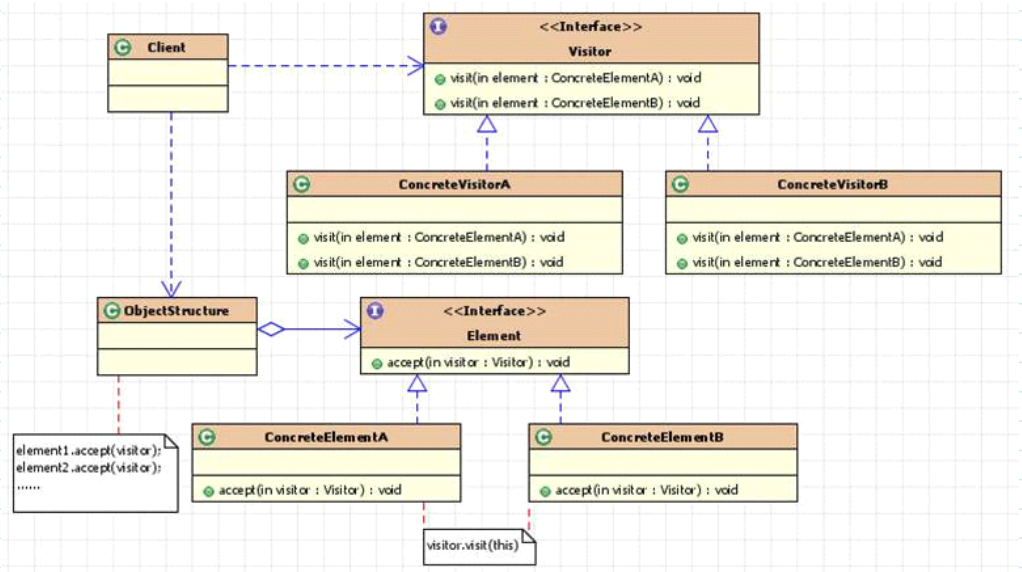
  - JDK中的体现：

    未发现

  - 使用场景：

    对象结构中对应的类很少改变，但需要在对象接口上定义新的操作

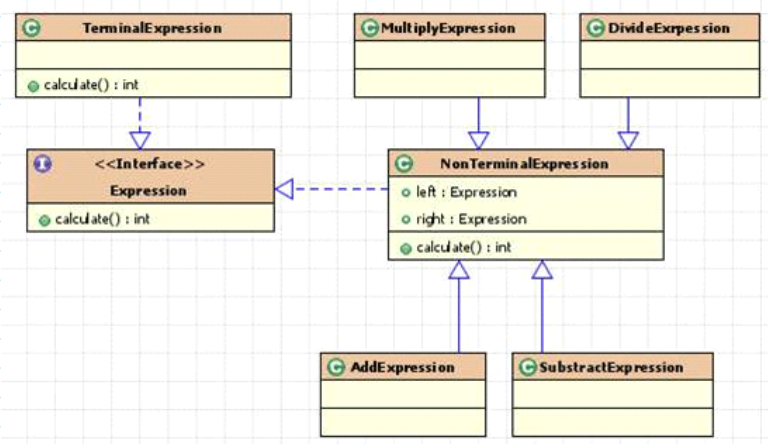
  - 类图：



- Interpreter（解释器）
  - 作用：

    用一组类代表某一规则
  - 使用场景：
    1) 编译器
    2) 表达式计算
  - JDK中体现：

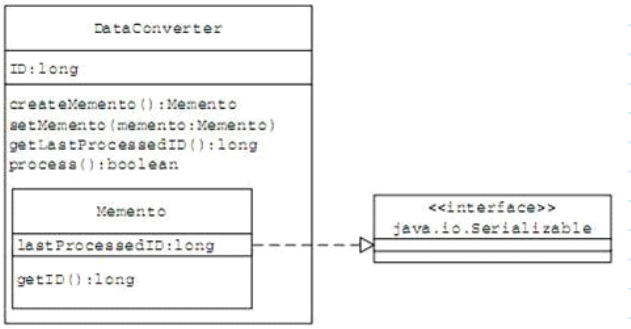
    java.util.regex.Pattern
  - 类图：四则运算



- Memento（备忘录）
  - 作用：

    保持对象状态，需要时可恢复
  - JDK中体现：

    未发现
  - 使用场景：
    1) 需要记录一个对象的内部状态时
    2) 提供一个可回滚的场景
  - 类图：



- Q2：**除了设计模式之外，目前有不少人在从事“反模式”的研究，请查阅相关资料，了解何谓“反模式”，以及研究“反模式”的意义。**

  A2：

  在了解反模式之前当然先要谈谈何为设计模式，因为两者是紧密联系在一起的。从我个人的理解认为，设计模式是一种在前人的设计经验上总结出来的对于一些普遍存在的问题提供的通用的解决方案。这些设计模式已经经过了长时间的实际应用和验证，被证实是有效可行的解决方案。

  我们来谈什么是反模式。很多人对反模式有一个理解误区，有人认为反模式是由于将通常使用的设计模式用在了错误的地方，也有人认为反模式只是一种坏习惯。简单的来说，反模式是指在对经常面对的问题经常使用的低效，不良，或者有待优化的设计模式。

  研究反模式的意义在于解决问题的带有共同性的不良方法。它们已经经过研究并分类，以防止日后重蹈覆辙，并能在研发尚未投产时辨认出来。