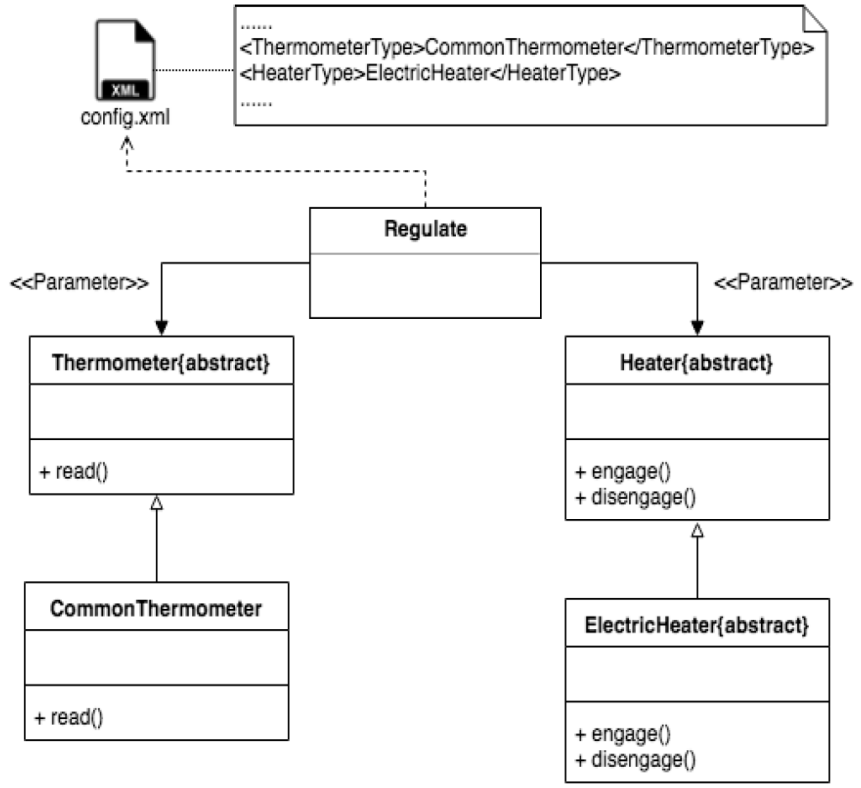
第五次作业

1. 请分析一下上述的软件设计违背了哪些设计原则，并给出你的改进设计。（考虑后续该调节软件可能会使用其它类型的温度计或者其他类型的加热器。）



- **违背开放-封闭原则（OCP）：**

  软件实体应当对扩展开放，对修改关闭。
  原因：如需增加其他类型的温度计和加热器，要修改Regulate的代码。

- **违背依赖倒置原则（DIP）：**

  高层模块不应该依赖低层模块，应该依赖抽象。
  原因： Regulate直接依赖了具体类，而非依赖于抽象。

第五次作业

2. 结合面向对象设计原则分析正方形是否为长方形的子类？

根据LSP原则（所有引用基类的地方必须能透明地使用其子类的对象），正方形不能作为长方形的子类。一个简单的情形：对四边形依次设置长和宽然后求面积。
对正方形而言，函数setWidth(W)满足后置条件width=height=W，函数setHeight(H)满足后置条件width=height=H，若正方形继承长方形并重写上述函数，则会导致结果area=H*H≠W*H，从而不满足LSP原则。

【答题内容】

1. 正方形不是长方形的子类

根据里氏替换原则，所有引用基类的地方必须能透明地使用其子类的对象。若正方形是长方形的子类，则在引用长方形对象的地方，如果改为引用正方形对象，则程序行为没有变化，即 dbc 契约的后置条件没有发生变化：假设长方形对象有设置长和宽的方法，若只是调用设置长的方法，则后置条件为长变化但宽没有变化；另一方面，正方形的长和宽一致，所以调用设置长的方法，则后置条件为长和宽都会发生变化。后置条件不一致，程序行为发生变化，故正方形不是长方形的子类。

第五次作业

3.有人将面向对象设计原则简单归结为三条：（1）封装变化点；（2）对接口进行编程；（3）多使用组合，而不是继承。请结合本章所学内容，谈谈对这三条归纳的理解

（1）封装变化点可以理解为开放封闭原则，找到系统中的可变因素并封装取来，尽量在不修改原有代码的情况下进行扩展。
（2）对接口进行编程可以理解为依赖倒置原则，尽量使用抽象类进行编程，将具体类写在配置文件中，从而创建可重用的框架。
（3）多使用组合而非继承可以理解为合成复用原则，优先使用对象组合，在运行时动态配置组件的功能，防止类层次规模的爆炸性增长。

（1）封装变化点：根据 OCP 开放-封闭原则，要求对扩展开放，对修改封闭，即软件实 体应尽量在不修改原有代码的情况下进行扩展。而一切的关键便是抽象化，将系统的可变因素封装起来，将变化部分和稳定部分隔离，有助于增加复用性，并降低系统耦合性
（2）对接口进行编程：根据 DIP 依赖倒置原则，高层模块不应该依赖底层模块，都应该依赖抽象，而高层次抽象一般是通过接口编程，而将具体类写在配置文件中避免直接实现；另一方面客户端程序并不需要了解具体实现只需要了解接口中的声明方法
（3）多使用组合，而不是编程：根据 CARP 合成/聚合复用原则，优先使用对象组合，而不是继承来达到复用的目的。因为继承的实现是静态的不能再运行时改变，且超类发生变化子类不得不改变，同时必须要求白箱复用，无法解决黑箱复用的情况。而合成/聚合复用可以动态进行，依赖少，支持黑箱复用，同时可以作为复用手段应用于几乎任何环境

第七次作业

1. **java.lang.Math类和java.lang.StrictMath类是否是单例模式？**

   都不是单例模式（只能有一个实例；自己创建自己的唯一实例；为其他对象提供这一实例）它们的构造函数是private类型的，但没有为外界提供可访问的自身的实例，外界只能调用它们的静态方法和静态常量。

```java
public final class Math {

    /**
     * Don't let anyone instantiate this class.
     */
    private Math() {}

    /**
     * The {@code double} value that is closer than any oth
     * <i>e</i>, the base of the natural logarithms.
     */
    public static final double E = 2.7182818284590452354;

    /**
     * The {@code double} value that is closer than any oth
     * <i>pi</i>, the ratio of the circumference of a circl
     * diameter.
     */
    public static final double PI = 3.14159265358979323846;
```

```java
public final class StrictMath {

    /**
     * Don't let anyone instantiate this class.
     */
    private StrictMath() {}

    /**
     * The {@code double} value that is closer than any othe
     * <i>e</i>, the base of the natural logarithms.
     */
    public static final double E = 2.7182818284590452354;

    /**
     * The {@code double} value that is closer than any othe
     * <i>pi</i>, the ratio of the circumference of a circle
     * diameter.
     */
    public static final double PI = 3.14159265358979323846;
```

第七次作业

2.单例模式课上讲了三种实现方式，分别是饿汉式，懒汉式，以及静态内部类实现方式：
(1) 对于懒汉式实现方式，请列举可以保证线程安全的两种方法；
(2) 请比较三种实现方式的差异；
(3) 请分析上述三种实现方式下，单例实例进入内存空间的时间点。

(1)改为饿汉式/synchronized锁同步/同步代码块/双检查锁Double Check Locking
(2)饿汉模式：无须考虑多个线程同时访问的问题；调用速度和反应时间优于懒汉式单例；资源利用效率不及懒汉式单例；系统加载时间可能会比较长。懒汉模式：实现了延迟加载；必须处理好多个线程同时访问的问题；需通过双重检查锁定等机制进行控制，将导致系统性能受到一定影响。静态内部类：定义内部类调用其静态私有方法获取类实例。
(3)饿汉模式：第一次加载时；懒汉模式：第一次被访问时；静态内部类：内部类被创建时

第七次作业

3. 请尝试设计实现一个多例类Multiton用户可以自行指定实例个数，给出实现代码以及设计思路。

```java
class Multiton {

    private final static String CONF_PATH = "conf.xml";
    private static int INSTANCE_NUM;
    private static Multiton[] multitons;

    static {
        Document doc = null;
        try {
            doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse(CONF_PATH);
        } catch (SAXException | IOException | ParserConfigurationException e) {
            e.printStackTrace();
        }
        assert doc != null;
        Element root = doc.getDocumentElement();
        Element multitonClass = (Element) root.getElementsByTagName("Multiton").item(0);
        Element multitonNum = (Element) multitonClass.getElementsByTagName("instance_num").item(0);
        INSTANCE_NUM = Integer.parseInt(multitonNum.getTextContent());
        multitons = new Multiton[INSTANCE_NUM];
    }

    private Multiton() {}

    public static Multiton getInstance() {
        int seed = random();
        Multiton multiton = multitons[seed];
        if (multiton == null) {
            synchronized (Multiton.class) {
                if (multitons[seed]  == null) {
                    multiton = new Multiton();
                    multitons[seed] = multiton;
                }
            }
        }
        return multiton;
    }

    private static int random() {
        return (int) (Math.random() * INSTANCE_NUM);
    }

    public static void main(String[] args) {
        System.out.println(getInstance());
    }

}
```

配置文件设置个数

double check

```java
    // 实例化N个对象实例
    static {
        // 添加Multipleton对象实例
        for (int i = 0; i < N; i++) {
            list.add(new Multipleton(i));
        }
    }

    /**
     * 随机获得 实例对象
     */
    public static Multipleton getRandomInstance() {
        // 获得随机数字
        int num = (int) (Math.random() * N);
        // 获得list中的对象实例
        return list.get(num);
    }
```

第七周、第九周课后作业

第七周课后作业

- **Factory Method模式和Abstract Factory模式的区别在哪？一般哪些情况下适合用前者，哪些情况下适合用后者？**

区别：

- Factory Method模式：定义一个创建对象的接口，使一个类的实例化（创建对象实例）延迟到子类。
- Abstract Factory模式：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

适用情况：

- Factory Method：明确地计划不同条件下创建不同实例时。
  - 当一个类不知道它所必须创建的对象的类的时候；
  - 当一个类希望由它的子类来指定它所创建的对象的时候；
  - 当类将创建对象的职责委托给多个子类中的某一个，并且你希望将哪一个子类是代理者这一信息局部化的时候。
- Abstract Factory：系统的产品有多于一个的产品族，而系统只消费其中某一族的产品时。
  - 一个系统要独立于它的产品的创建、组合和表示时
  - 一个系统要由多个产品系列中的一个来配置时
  - 当你要强调一系列相关的产品对象的设计以便进行联合使用时
  - 当你提供一个产品类库，而只想显示它们的接口而不是实现时

工厂方法模式在增加一个具体产品的时候，都要增加对应的工厂。
但是抽象工厂模式只有在新增一 个类型的具体产品时才需要新增工厂。
也就是说，工厂方法模式的一个工厂只能创建一个具体产品。 而抽象工厂模式的一个工厂可以创建属于一类类型的多种具体产品。

第七周课后作业

- **假设有一销售管理系统其中包括一个客户类Customer，在客户类中包含一个名为客户地址的成员变量，客户地址类型为Address，两个类的原始定义如下：**

```
Class Customer{
    String name; //or string name; in C++
    int age;
    bool gender;
    Address address; // or Address *address; in C++
    //getter, setter 函数省略
}
```

```
Class Address{
    String street1; //or string street1; in C++
    String city;
    String province;
    String country;
    String postcode;
    //getter, setter 函数省略
}
```

第七周课后作业

- **请尝试使用Java中的Cloneable接口以及Serializable接口实现浅拷贝以及深拷贝的原型模式以对Customer进行拷贝。对实现的两种原型模式，客户端测试代码中比较拷贝前后Customer对象是否相同， 拷贝前后的Address对象是否相同。**

```
@Override
public Customer clone(){
    Object newObj = null;
    try {
        newObj = super.clone();
    }
    catch(CloneNotSupportedException e){
        System.out.println("不支持复制！");
        return null;
    }
    return (Customer)newObj;
}
```

1. 浅拷贝

2. 深拷贝

重写clone()方法

```
@Override
protected Object clone() throws CloneNotSupportedException {
    Customer a = (Customer) super.clone();
    a.address = (Address) address.clone();
    return a;
}
```

序列化

```
public Customer deepClone() throws IOException, ClassNotFoundException{
    ByteArrayOutputStream data = new ByteArrayOutputStream();
    ObjectOutputStream objectWriter = new ObjectOutputStream(data);
    objectWriter.writeObject(this);

    ObjectInputStream objectReader = new ObjectInputStream(new ByteArrayInputStream(data.toByteArray()));
    return (Customer)objectReader.readObject();
}
```

第七周课后作业

- **另尝试不采用Java中的接口实现深拷贝以及浅拷贝的原型模式，客户端测试代码与上述要求相同。**

1. 浅拷贝

```
public Customer deepClone() throws Exception {
    Customer c = new Customer();
    c.setaddress(this.address);
    c.setage(this.age);
    c.setgender(this.gender);
    c.setname(this.name);
    return c;
}
```

2. 深拷贝

```
public Customer deepClone() throws Exception {
    Customer c = new Customer();
    Address ad = new Address();
    ad.setcity(this.address.city);
    ad.setcountry(this.address.country);
    ad.setpostcode(this.address.postcode);
    ad.setprovice(this.address.province);
    ad.setstreet1(this.address.street1);

    c.setaddress(ad);
    c.setage(this.age);
    c.setgender(this.gender);
    c.setname(this.name);
    return c;
}
```

第九周课后作业

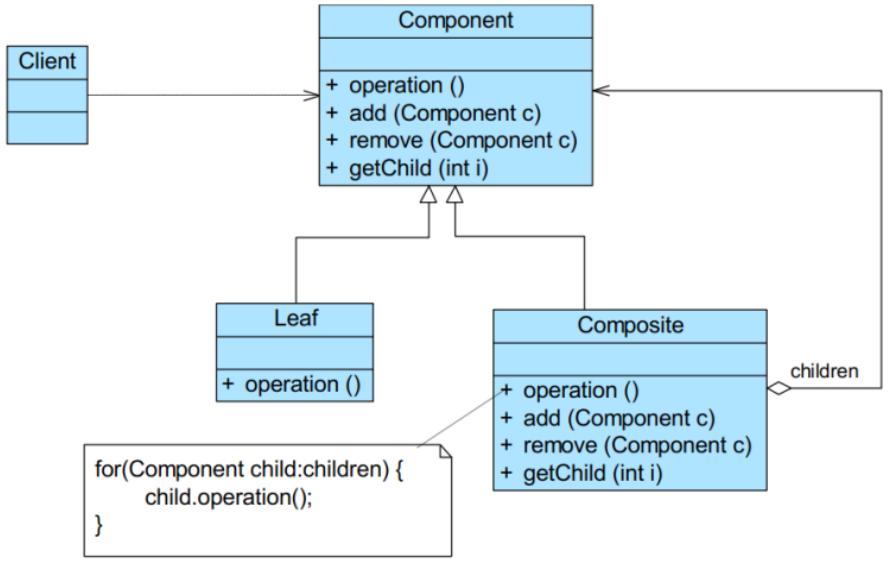
- **如何在Composite模式中避免环状引用？**
-

思路：记录下每个组件从根节点开始的路径，因为要出现环状引用，在一条路径上，某个对象就必然会出现两次。因此只要每个对象在整个路径上只是出现了一次，那么就不会出现环状引用。

①记录下每个组件从根节点开始的路径，在这条路径上，某个对象出现两次，就会构成环状引用。

②先在Component中设置一个字段，专门用来记录从根结点开始到Component本身的路径。

③当Composite对象的添加子组件方法中，先检测要添加的子组件是否出现在上面所说的路径中，如果出现环状引用，则抛出异常

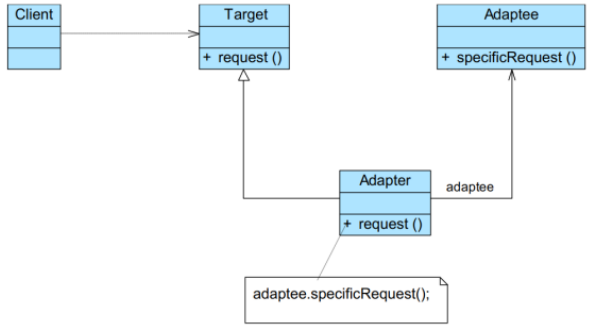


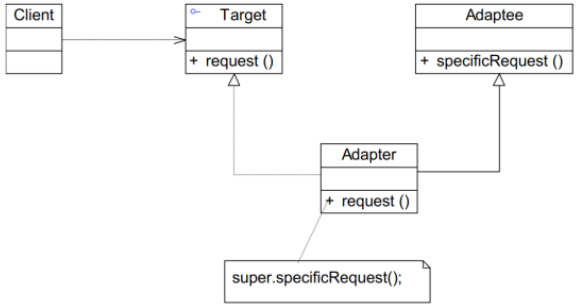
第九周课后作业

- 在对象适配器中，一个适配器能否适配多个适配者？如果能，应该如何实现？如果不能，请说明原因？如果是类适配器呢？

（1）在对象适配器中，适配器与适配者之间是关联关系，一个适配器能够对应多个适配者类，只需要在该适配器类中定义对多个适配者对象的引用即可；

（2）在类适配器中，适配器与适配者是继承关系，一个适配器能否适配多个适配者类取决于该编程语言是否支持多重类继承，例如 C++语言支持多重类继承则可以适配多个适配者，而 Java、C#等语言不支持多重类继承 则不能适配多个适配者





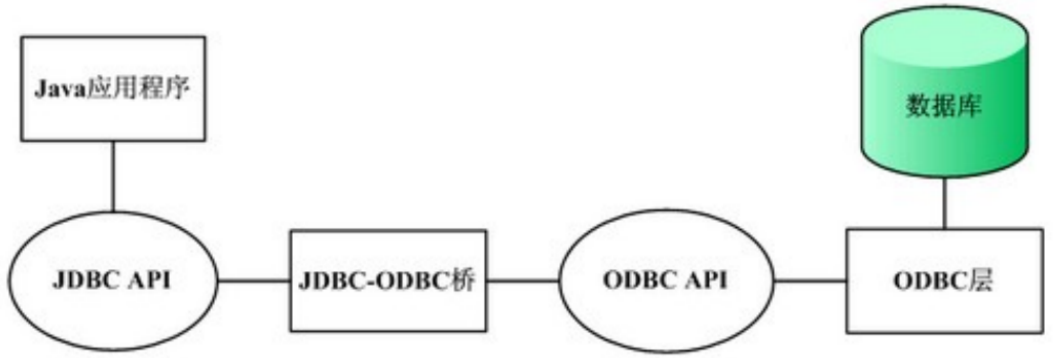
第九周课后作业

- **JDBC/ODBC桥梁是Bridge模式吗？**

  是桥梁模式。

但是JDBC-ODBC桥梁不是桥梁模式，而是适配器模式

假如**原先的数据库是基于ODBC数据库驱动的，没有对应的JDBC驱动程序**，可以使用JDBC-ODBC桥进行数据库访问。JDBC-ODBC桥是一个**JDBC驱动程序**，完成了从JDBC操作到ODBC操作之间的转换工作，允许JDBC驱动程序被用作ODBC的驱动程序。



第九周课后作业

- **JDBC/ODBC桥梁是Bridge模式吗？**

  是桥梁模式。

是桥接模式，JDBC连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不动，原因就是JDBC提供了统一接口，每个数据库提供各自的实现，用数据库驱动程序进行桥接。

其中 `java.sql.Driver` 为抽象桥类，而驱动包如 `com.mysql.jdbc.Driver` 是具体的实现桥接类，而 `Connection` 是被桥接的对象。这里的两个维度是：

数据库类型的不同（驱动不同）

数据库的连接信息不同（URL，username,password）