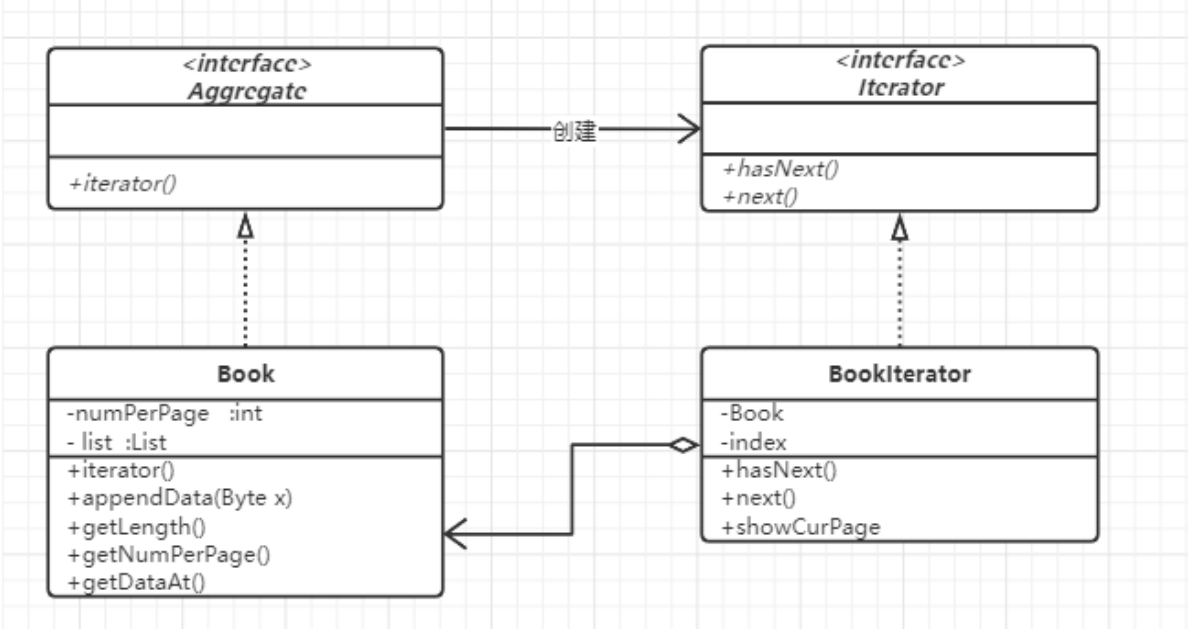
1. 迭代器模式

设计一个逐页迭代器，每次可返回指定个数（一页）元素，并将该迭代器用于对数据进行分页处理。绘制对应的类图并编程模拟实现

令实现集合接口的类名称叫Book（类比现实生活中的书把字符分页了）、Book在创建时可以指定一页拥有的数据。BookIterator以此为依据来分页、这个BookIterator可以翻到下一页，还可以返回Book当前页面的数据。




```
package iterator_moudle;

public interface Iterator {
    public boolean hasNext();
    public Object next();
}
```

```
package iterator_moudle;

public interface Aggregate {
    Iterator iterator();
}
```

```
package iterator_moudle;

import java.util.ArrayList;
import java.util.List;
```

```java
public class Book implements Aggregate{
    private int numPerPage;
    private List list;

    public Book(int numPerPage){
        this.numPerPage=numPerPage;
        this.list=new ArrayList<Byte>();
    }

    public void appendData(Byte x){
        this.list.add(x);
    }

    public Byte getDataAt(int index){
        return (Byte)this.list.get(index);
    }

    public  int getLength() {
        return this.list.size();
    }

    public int getNumPerPage(){
        return this.numPerPage;
    }

    @Override
    public Iterator iterator() {
        return new BookIterator(this);
    }
}
```

```java
package iterator_moudle;

import static java.lang.Math.min;

public class BookIterator implements Iterator {
    private Book book;
    private int index;
    public BookIterator(Book book) {
        this.book=book;
        this.index=0;
    }

    @Override
    public boolean hasNext() {
        if(index*book.getNumPerPage()<book.getLength()){
            return true;
        }else {
            return false;
        }
    }

    @Override
```

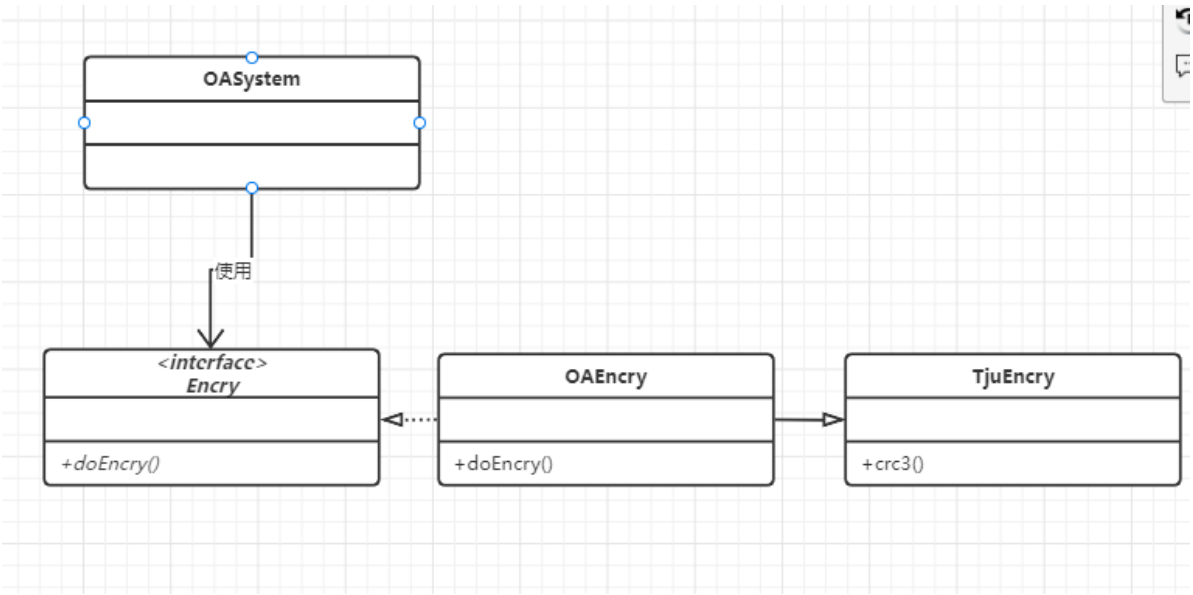
```
    public Object next() {
        Object res=(Object)showCurPage();
        index++;
        return res;
    }

    public String showCurPage(){
        StringBuffer sb=new StringBuffer();
        int start=index*book.getNumPerPage();
        int end=min((index+1)*book.getNumPerPage(),book.getLength());
        for(int i=start;i<end;i++){
            sb.append(book.getDataAt(i));
        }
        return sb.toString();
    }

}
```

2.适配器模式

某 OA 系统需要提供一个加密模块，将用户机密信息（例如口令、邮箱等）加密之后再存储在数据库中，系统已经定义好了**数据库操作类**。为了提高开发效率，现需要重用**已有的加密算法**，这些算法封装在一些由第三方提供的类中，有些甚至没有源代码。试使用适配器模式设计该加密模块，实现在不修改现有类的基础上重用第三方加密方法。要求绘制相应的类图并编程模拟实现，需要提供对象适配器和类适配器两套实现方案。

在此例中，已有的加密算法模块（命名为tjuEncry）是Adaptee，假设它有一个加密算法CRC3。target是OA系统要用的加密算法（Encry），它有一个方法。Adapter是OAEncry

类适配器



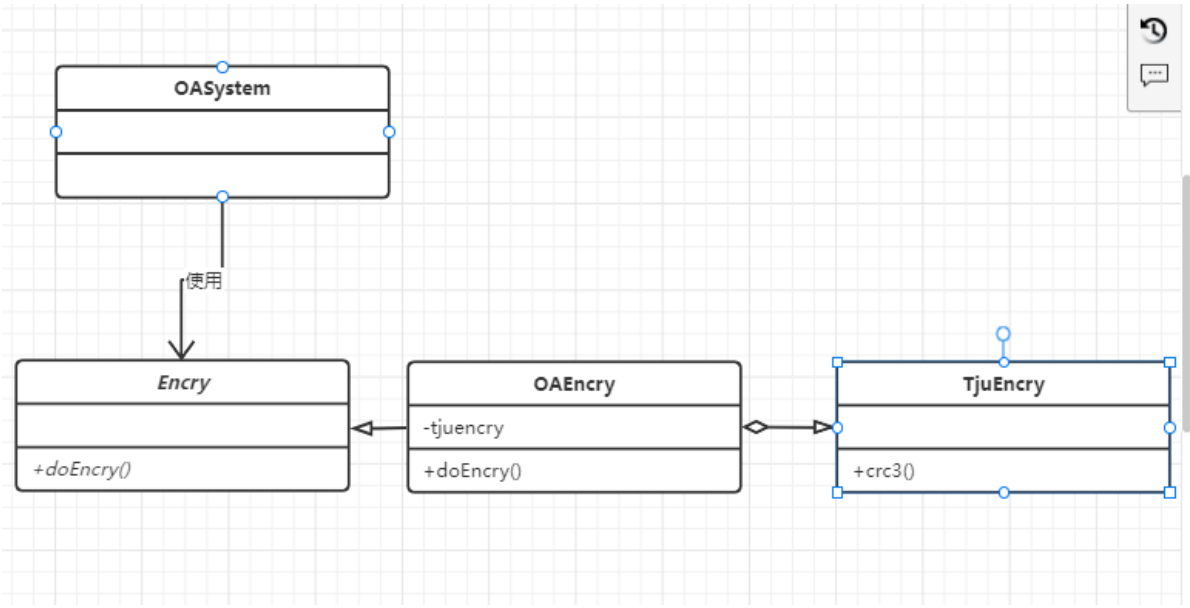
```java
package adapter_moudle;

public class OAsystem {

    public void dosomething(String x){
        Encry enc=new OAEncry();
        String encrydata=enc.doEncry(x);
        System.out.println(encrydata);
    }
}
```

```java
package adapter_moudle;

public interface Encry {
    public String doEncry(String x);
}
```

```java
package adapter_moudle;

public class OAEncry extends TjuEncry implements Encry  {
    @Override
    public String doEncry(String x) {
        return this.crc3(x);
    }
}
```

```java
package adapter_moudle;

public class TjuEncry {
    public String crc3(String x){
        return "**"+x+"**";
    }
}
```

对象适配器




```
package adapter_moudle.objectadapter;

public class OAsystem {

    public void dosomething(String x){
        Encry enc=new OAEncry();
        String encrydata=enc.doEncry(x);
        System.out.println(encrydata);
    }
}
```

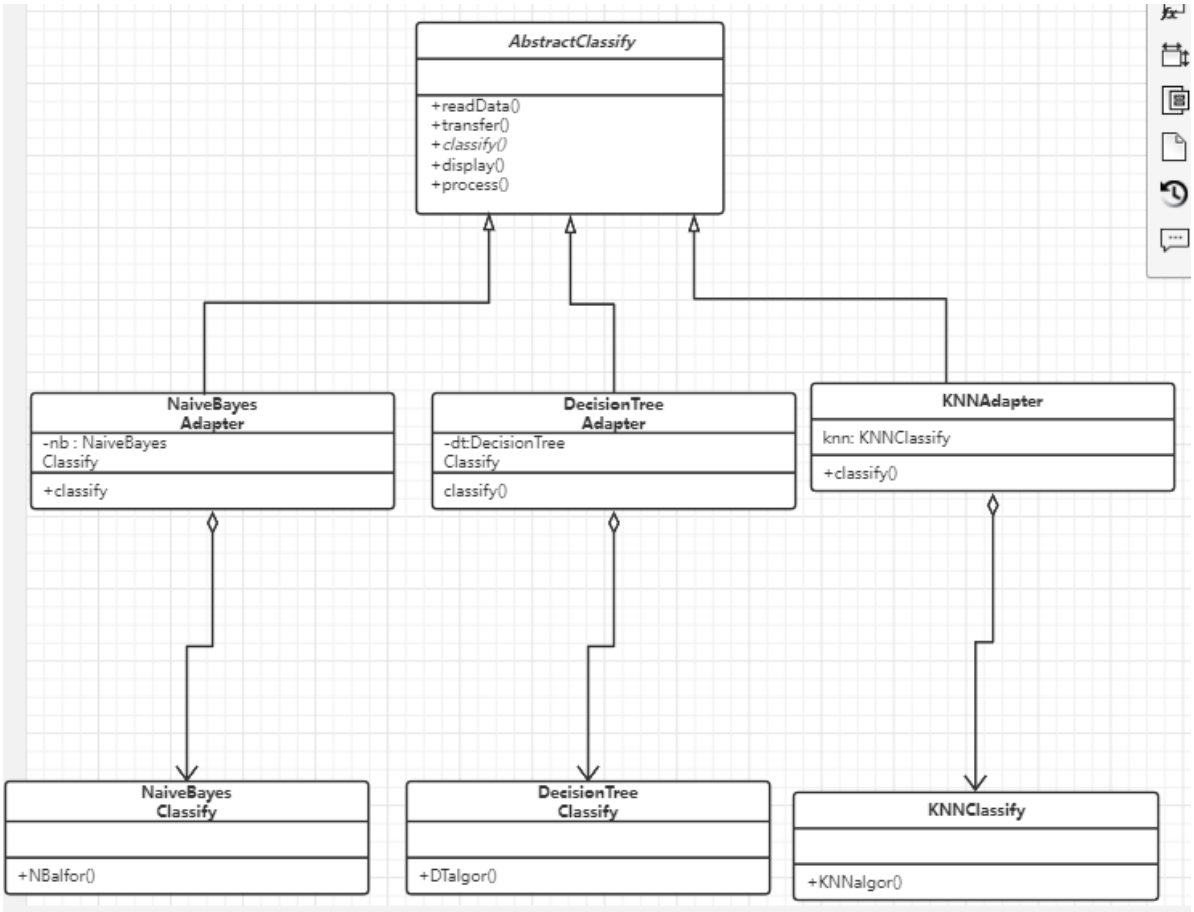
```
package adapter_moudle.objectadapter;

public abstract class Encry {
    public abstract String doEncry(String x);
}
```

```
package adapter_moudle.objectadapter;

public class OAEncry extends Encry{
    private TjuEncry tjuencry;
    public OAEncry(){
        tjuencry=new TjuEncry();
    }

    @Override
    public String doEncry(String x) {
        return tjuencry.crc3(x);
    }
}
```

```java
package adapter_moudle.objectadapter;

public class TjuEncry {
    public String crc3(String x){
        return "**"+x+"**";
    }
}
```

3. 模板方式模式和适配器模式

在某数据挖掘工具的数据分类模块中，数据处理流程包括 4 个步骤，分别是：①读取数据；②转换数据格式；③调用数据分类算法；④显示数据分类结果。对于不同的分类算法而言，第①步、第②步和第④步是相同的，主要区别在于第③步。第③步将调用算法库中已有的分类算法实现，例如朴素贝叶斯分类（Naive Bayes）算法、决策树（Decision Tree）算法、 K 最近邻（K - Nearest Neighbor , KNN）算法等。现采用模板方法模式和适配器模式设计该数据分类模块，绘制对应的类图并编程模拟实现。




```java
package template_moudle;

abstract class AbstractClassify {
    public void readData(){
        //do read
    }

    public void transfer(){
        //do transfer
    }
```

```
    public void display(){
        //do display
    }

    public abstract void classify();

    public void process(){
        readData();
        transfer();
        classify();
        display();
    }

}
```

```
package template_moudle;

public class NaiveBayesAdapter extends AbstractClassify{
    NaiveBayesClassify nb;

    public NaiveBayesAdapter(){
        nb=new NaiveBayesClassify();
    }

    @Override
    public void classify() {
        nb.NBalgor();
    }
}
```

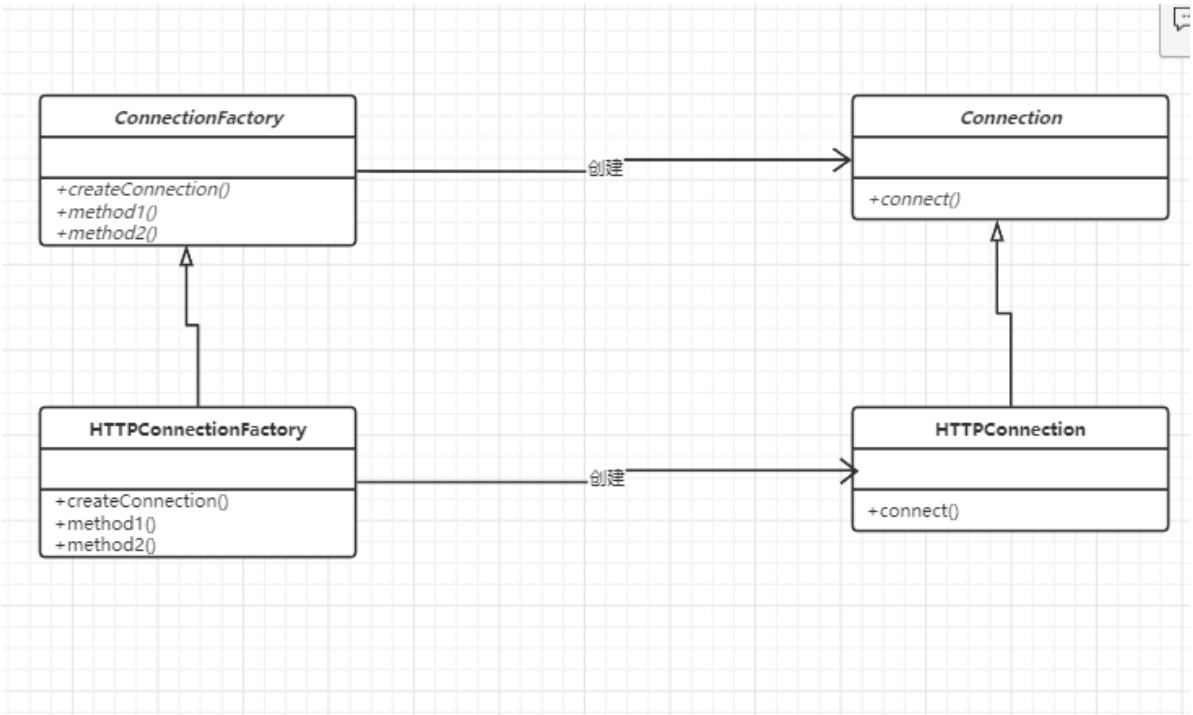
```
package template_moudle;

public class NaiveBayesClassify {
    public void NBalgor(){
        //do naive ba yes classify
    }
}
```

4.工厂方法模式

在某网络管理软件中，需要为不同的网络协议提供不同的连接类，例如针对POP3 协议的连接类 POP3Connection、针对 IMAP 协议的连接类 IMAPConnection 、针对HTTP协议的连接类 HTTPConnection 等。由于网络连接对象的创建过程较为复杂，需要将其创建过程封装到专门的类中，该软件还将支持更多类型的网络协议。现采用工厂方法模式进行设计，绘制类图并编程模拟实现。

Product是Connection、ConcreteProduct是具体的POP3Connection或IMAPConnection 等等。

以HTTPConnection 为例，其它的不再赘述。




```java
package factory_moudle;

public abstract class ConnectionFactory {
    public abstract Connection method1(String msg);
    public abstract void method2(Connection con);
    public Connection createConnection(String msg){
        Connection con=method1(msg);
        method2(con);
        return con;
    }
}
```

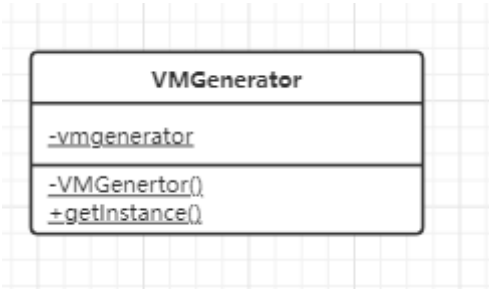
```java
package factory_moudle;

public abstract class Connection {
    public abstract void connect();
}
```

```java
package factory_moudle;

public class HTTPConnectionFactory extends ConnectionFactory{
    @Override
    public Connection method1(String msg) {
        Connection con =new HTTPConnection();
        //do something ;
        return con;
    }

    @Override
    public void method2(Connection con) {
        //do something;
    }
}
```

```
package factory_moudle;

public class HTTPConnection extends Connection{

    @Override
    public void connect() {
        //do connect;
    }
}
```

5.单例模式

某 Web 性能测试软件中包含一个虚拟用户生成器（Virtual User Generator）。为了避免生成的虚拟用户数量不一致，该测试软件在工作时只允许启动唯一一个虚拟用户生成器。采用单例模式设计该虚拟用户生成器，绘制类图并分别使用饿汉式单例、双重检测锁和IoDH 三种方式编程模拟实现。



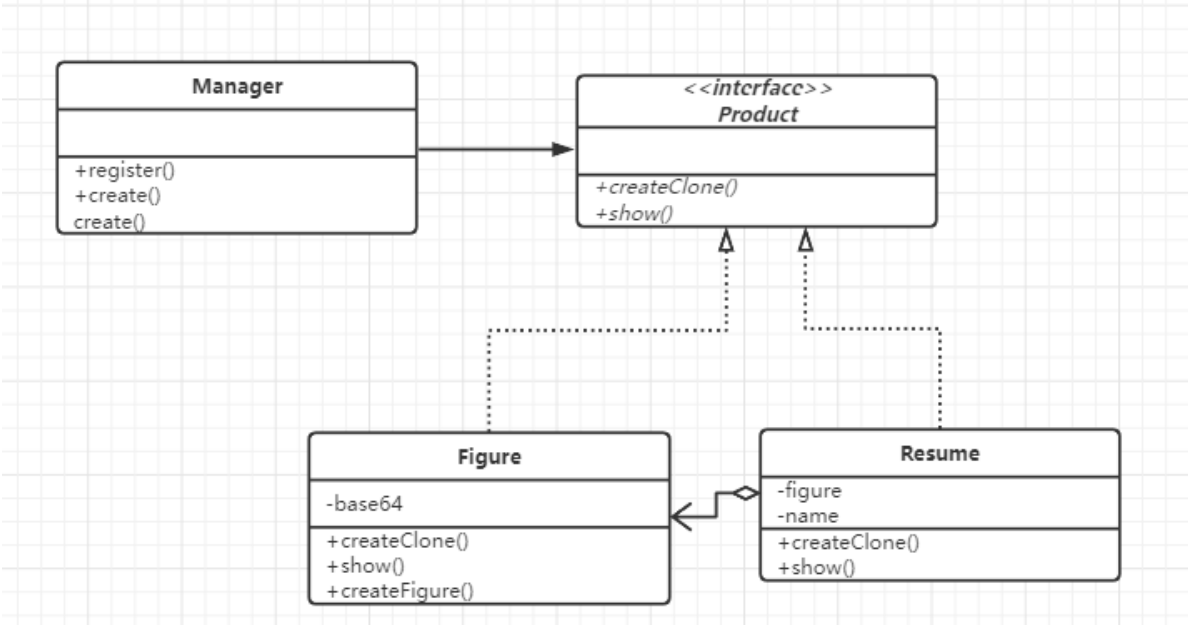

```
package singleton_moudle.hungry;

public class VMGenerator {
    private static VMGenerator vmgenerator=new VMGenerator();
    private VMGenerator(){
        System.out.println("创建一个虚拟用户生成器");
    }
    public VMGenerator getInstance(){
        return vmgenerator;
    }

}
```

```
package singleton_moudle.doublechecked;

public class VMGenerator {
    private static volatile VMGenerator vmGenerator;
    private VMGenerator() {
        System.out.println("生成了一个虚拟用户");
    }
    public static VMGenerator getInstance(){
        if(vmGenerator == null){
            synchronized (VMGenerator.class){
                if(vmGenerator == null){
                    vmGenerator = new VMGenerator();
                }
            }
        }
        return vmGenerator;
```

```
    }
}
```

6.原型模式

在某在线招聘网站中，用户可以创建一个简历模板。针对不同的工作岗位，可以复制该简历模板并进行适当修改后，生成一份新的简历。在复制简历时，用户可以选择是否复制简历中的照片：如果选择“是”，则照片将一同被复制，用户对新简历中的照片进行修改不会影响到简历模板中的照片，对模板进行修改也不会影响到新简历；如果选择“否”，则直接引用简历模板中的照片，修改简历模板中的照片将导致新简历中的照片一同修改，反之亦然。现采用原型模式设计该简历复制功能并提供浅克隆和深克隆两套实现方案，绘制对应的类图并编程模拟实现




```
package prototype;

import java.util.Scanner;

public class Figure implements Product{
    private Byte[] base64;
    @Override
    public Product createClone() {
        Product p=null;
        System.out.println("是否复制简历中的照片?");
        Scanner reader = new Scanner(System.in);
        String ans=reader.next();
        if(ans.equals("是")){
            int length=base64.length;
            Figure f=new Figure();
            for(int i=0;i<length;i++){
                f.base64[i]=base64[i];
            }
            p=f;
        }else{
            try {
```

```
                p=(Product)clone();
            } catch (CloneNotSupportedException e) {
                throw new RuntimeException(e);
            }
        }
        return p;
    }

    @Override
    public void show() {
        System.out.println(base64.toString());
    }
}
```

```
package prototype;

public interface Product extends Cloneable{
    public Product createClone();
    public void show();
}
```

```
package prototype;

public class Resume implements Product{
    private String name;
    private Figure figure;
    @Override
    public Product createClone() {
        Product p = null;
        try {
            p=(Product)clone();

        } catch (CloneNotSupportedException e) {
            throw new RuntimeException(e);
        }
        return p;
    }

    @Override
    public void show() {
        System.out.println(name);
        figure.show();
    }
}
```