天津大学

设计模式（2）实验报告

学　　院　智算学部

专　　业　软件工程

学　　号　3019213043

姓　　名　刘京宗

一、实验目的

1. 结合实例，熟练绘制设计模式结构图。
2. 结合实例，熟练使用 Java 语言实现设计模式。
3. 通过本实验，理解每一种设计模式的模式动机，掌握模式结构，学习如何使用代码实现这些设计模式。

二、实验要求

1. 结合实例，绘制设计模式的结构图。
2. 使用 Java 语言实现设计模式实例，代码运行正确。

三、实验内容

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。

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。

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。

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。

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。

6. 原型模式

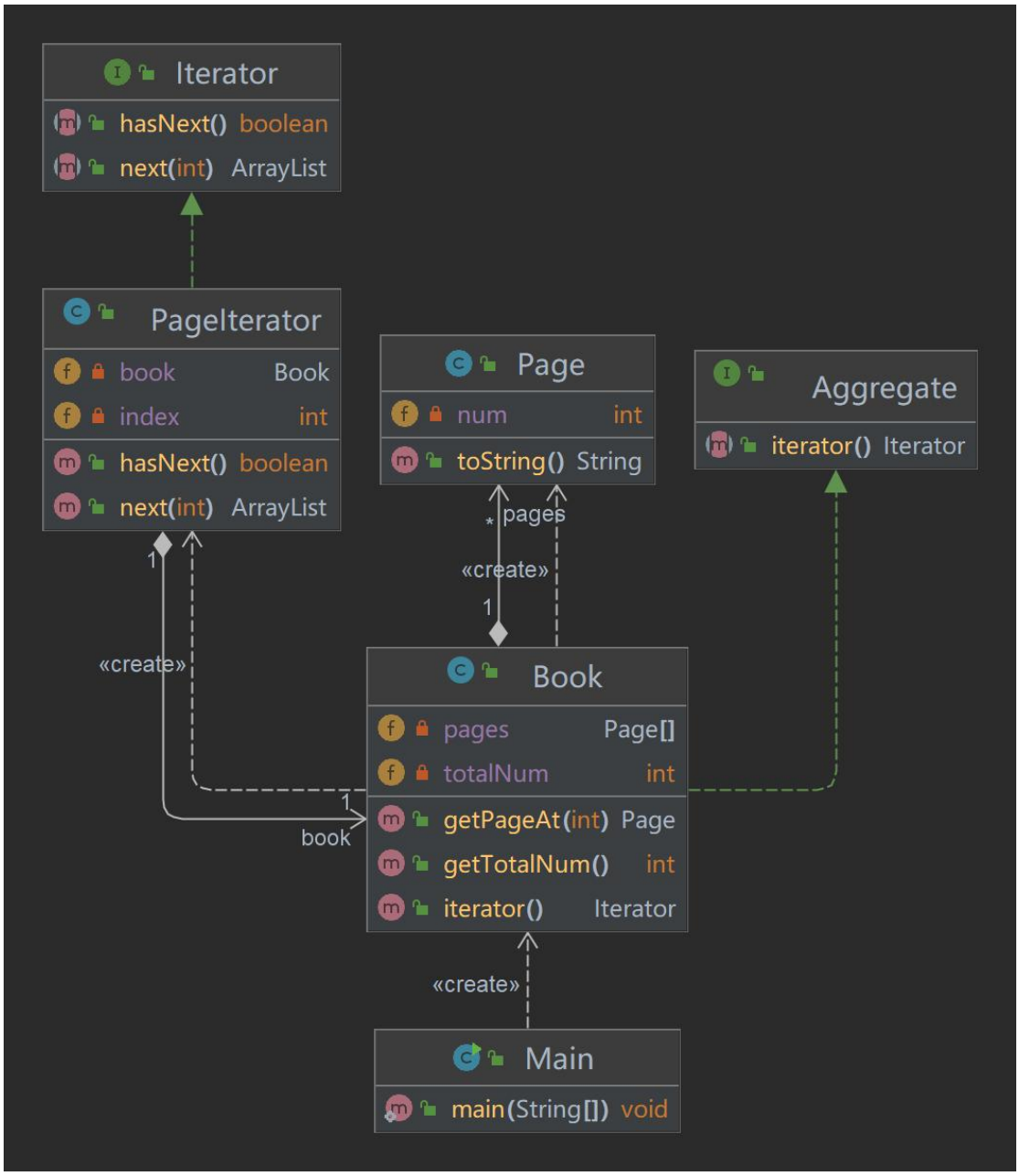
在某在线招聘网站中，用户可以创建一个简历模板。针对不同的工作岗位，可以复制该 简历模板并进行适当修改后，生成一份新的简历。在复制简历时，用户可以选择是否复制简历中的照片：如果选择“是”，则照片将一同被复制，用户对新简历中的照片进行修改不会 影响到简历模板中的照片，对模板进行修改也不会影响到新简历；如果选择“否”，则直接 引用简历模板中的照片，修改简历模板中的照片将导致新简历中的照片一同修改，反之亦然。

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。

四、实验结果

需要提供设计模式实例的结构图（类图）和实现代码。

4.1 迭代器模式



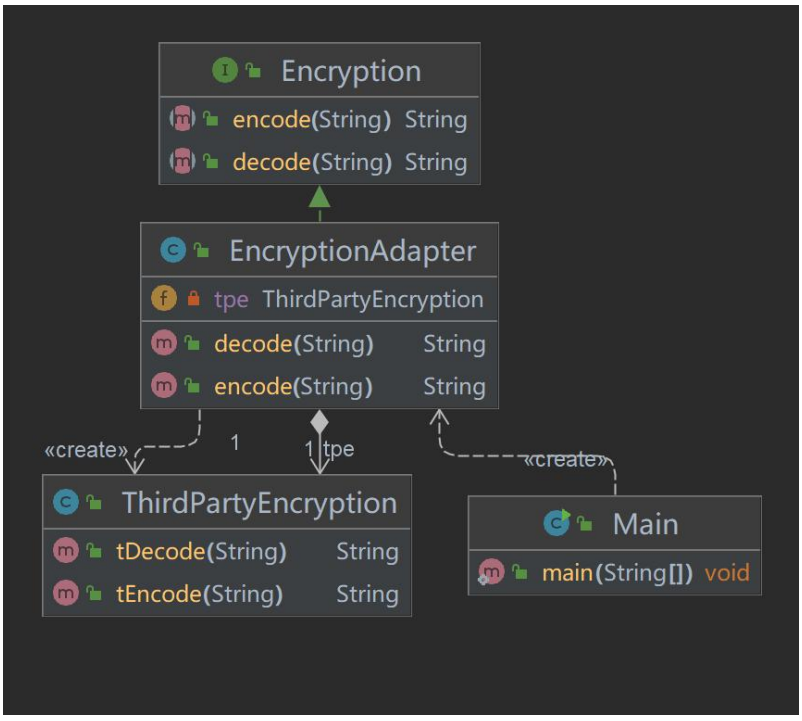

```
public interface Aggregate {
    public abstract Iterator iterator();
}
public class Book implements Aggregate {
    private Page[] pages;
    private int totalNum;

    public Book(int totalNum) {
        this.totalNum = totalNum;
        pages = new Page[totalNum];
        for (int i = 0; i < totalNum; i++) pages[i] = new Page(i);
```

```
    }

    public Page getPageAt(int index) {
        return pages[index];
    }

    public int getTotalNum() {
        return totalNum;
    }

    @Override
    public Iterator iterator() {
        return new PageIterator(this);
    }
}
public interface Iterator {
    public abstract boolean hasNext();
    public abstract ArrayList next(int pageSize);
}
public class Page {
    private int num;

    public Page(int num) {
        this.num = num;
    }

    public String toString() {
        return "第" + String.valueOf(num) + "页";
    }
}
public class PageIterator implements Iterator {
    private Book book;
    private int index;

    public PageIterator(Book book) {
        this.book = book;
    }

    @Override
    public boolean hasNext() {
        if (index < this.book.getTotalNum()) return true;
        return false;
    }
```

```
    @Override
    public ArrayList next(int pageSize) {
        ArrayList<Page> list = new ArrayList<>();
        for (int i = 0; i < pageSize; i++) {
            list.add(book.getPageAt(index));
            index++;
            if (index == book.getTotalNum()) break;
        }
        return list;
    }
}
public class Main {
    public static void main(String[] args){
        Aggregate book=new Book(24);
        Iterator it= book.iterator();
        int pageSize=5;
        while (it.hasNext()){
            ArrayList pages=it.next(pageSize);
            for(Object p:pages) System.out.println(p);
            System.out.println("--------------------");
        }
    }
}
```

4.2 适配器模式

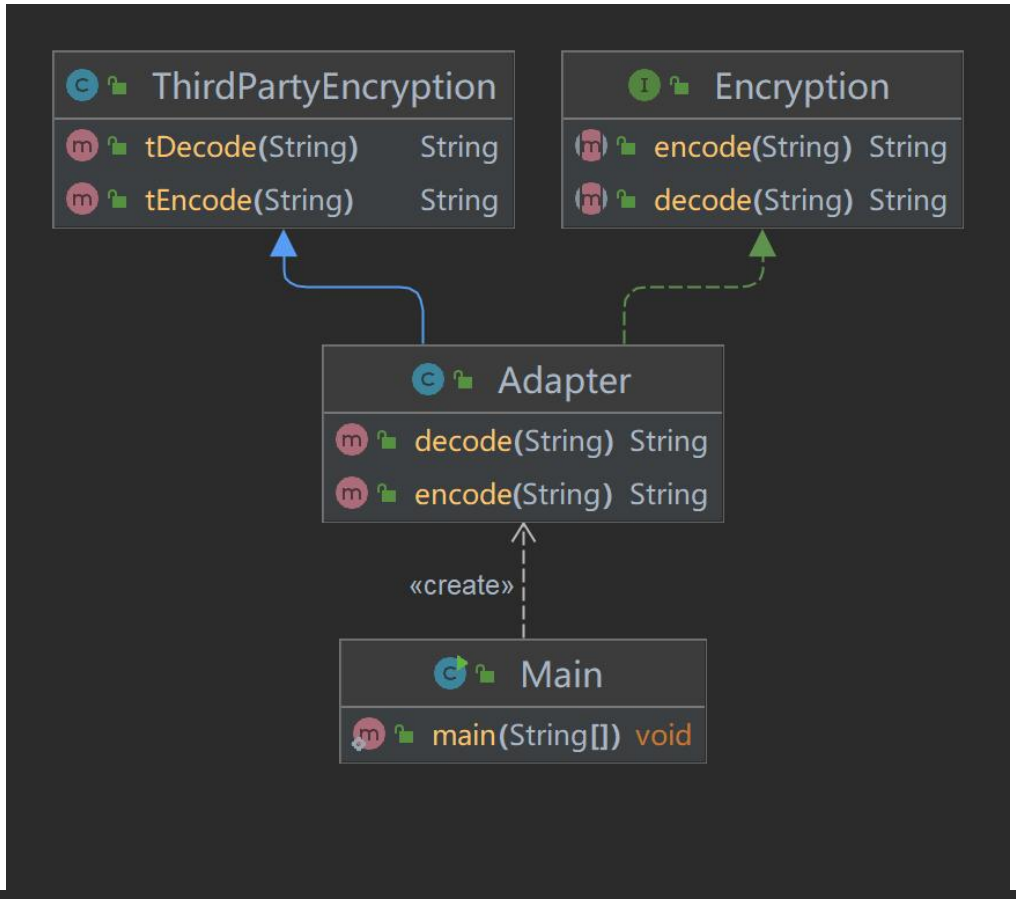
对象适配器：



```java
public interface Encryption {
    public String encode(String string);

    public String decode(String string);
}
public class EncryptionAdapter implements Encryption {

    // 组合一个第三方的加密类
    private ThirdPartyEncryption tpe;

    public EncryptionAdapter() {
        tpe = new ThirdPartyEncryption();
    }

    @Override
    public String encode(String string) {
        // 调用第三方的加密方法
        return tpe.tEncode(string);
    }

    @Override
    public String decode(String string) {
        // 调用第三方的解密方法
        return tpe.tDecode(string);
    }

}
public class ThirdPartyEncryption {
    public String tEncode(String string) {
        return "加密之后的" + string;
    }

    public String tDecode(String string) {
        return "解密之后的" + string;
    }
}
public class Main {

    public static void main(String[] args) {

        Encryption encryption = new EncryptionAdapter();
        String encodeString = encryption.encode("password");
        String decodeString = encryption.decode("password");
        System.out.println(encodeString);
```

```
        System.out.println(decodeString);
    }

}
```

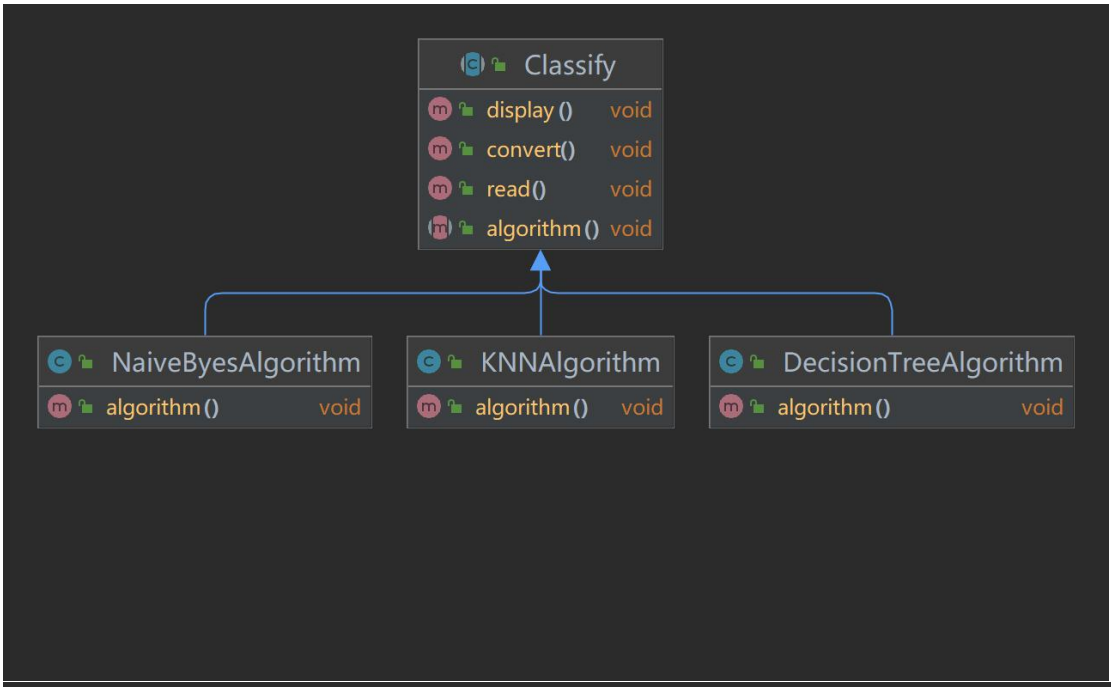
类适配器：




```
public class Adapter extends ThirdPartyEncryption implements Encryption {

    @Override
    public String encode(String string) {
        return tEncode(string);
    }

    @Override
    public String decode(String string) {
        return tDecode(string);
    }

}
```

类适配器模式和对象适配器模式最大的区别在于适配器和适配者之间的关系不同，对象适配器模式中适配器和适配者之间是关联关系，而类适配器模式中适配器和适配者是继承关系。

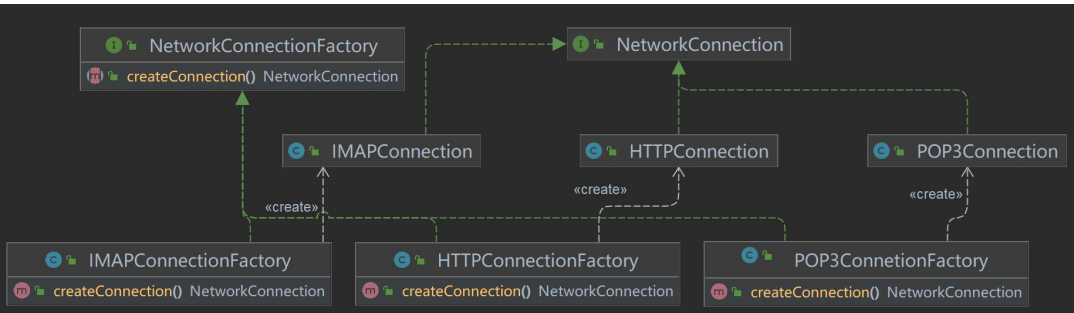
4.3 模板方式模式和适配器模式




```
public abstract class Classify {
    public void read() {
    }

    public void convert() {
    }

    public void display() {
    }

    abstract public void algorithm();

}
public class DecisionTreeAlgorithm extends Classify {
    public void algorithm() {
    }
}
public class KNNAlgorithm extends Classify {
    public void algorithm() {

    }
}

public class NaiveByesAlgorithm extends Classify{
    public void algorithm(){
```

```
    }
}
```

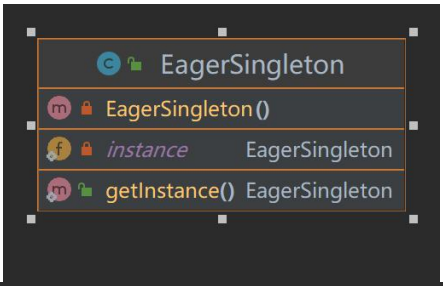
4.4 工厂方法模式




```
public interface NetworkConnection {
}
public interface NetworkConnectionFactory {
    NetworkConnection createConnection();
}
public class HTTPConnection implements NetworkConnection{
}
public class IMAPConnection implements NetworkConnection{
}
public class POP3Connection implements NetworkConnection{
}
public class HTTPConnectionFactory implements NetworkConnectionFactory {
    @Override
    public NetworkConnection createConnection() {
        System.out.println("创建 HTTP 连接");
        return new HTTPConnection();
    }
}
public class IMAPConnectionFactory implements NetworkConnectionFactory {
    @Override
    public NetworkConnection createConnection() {
        System.out.println("创建 IMAP 连接");
        return new IMAPConnection();
    }
}
```

```
public class POP3ConnetionFactory implements NetworkConnectionFactory {
    @Override
    public NetworkConnection createConnection() {
        System.out.println("创建 POP3 连接");
        return new POP3Connection();
    }
}
```

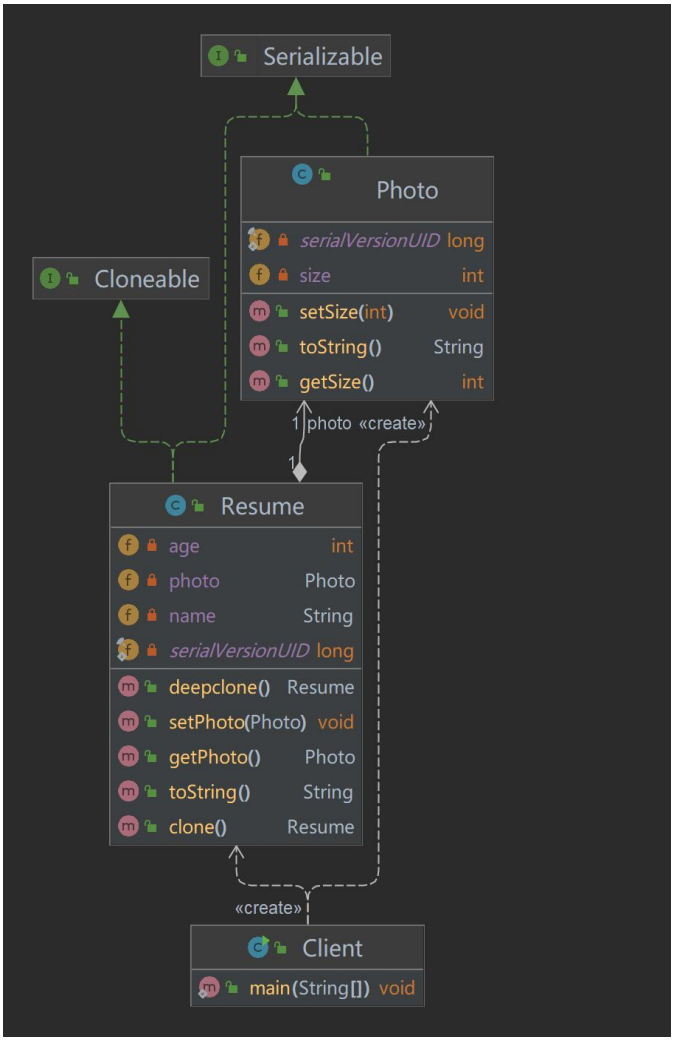
4.5. 单例模式




```
public class EagerSingleton {
    //静态私有成员变量
    private static EagerSingleton instance = new EagerSingleton();

    //私有构造方法
    private EagerSingleton() {
        System.out.println("启动唯一一个虚拟用户生成器，生成一个虚拟用户！！！");
    }

    //静态公有工厂方法，返回唯一的实例
    public static EagerSingleton getInstance() {
        return instance;
    }
}
public class DoubleCheckedLocking {
    //静态私有成员变量
    private static volatile DoubleCheckedLocking instance = null;

    //私有构造方法
    private DoubleCheckedLocking() {
        System.out.println("启动唯一一个虚拟用户生成器，生成了一个虚拟用户！！！");
    }

    //静态公有工厂方法，返回唯一的实例
    public static DoubleCheckedLocking getInstance() {
```

```
            //第一重判断
            if (instance == null) {
                //锁定代码块
                synchronized (DoubleCheckedLocking.class) {
                    //第二重判断
                    if (instance == null) {
                        instance = new DoubleCheckedLocking(); //创建单例实例
                    }
                }
            }
            return instance;
        }
}
ublic class IoDHSingleton {
        private IoDHSingleton() {

        }

        private static class HolderClass {
            private final static IoDHSingleton instance = new IoDHSingleton();
        }

        public static IoDHSingleton getInstance() {
            return HolderClass.instance;
        }
}
```

6. 原型模式




```java
public class Resume implements Cloneable, Serializable {

    private static final long serialVersionUID = 4455666L;

    private String name;
    private int age;
    private Photo photo;

    public Resume(String name, int age, Photo photo) {
        super();
        this.name = name;
        this.age = age;
        this.photo = photo;
    }
```

```
public Photo getPhoto() {
	return photo;
}

public void setPhoto(Photo photo) {
	this.photo = photo;
}

@Override
public String toString() {
	return "Resume [name=" + name + ", age=" + age + ", photo=" + photo.toString() + "]";
}

public Resume clone() {
	Object obj = null;
	try {
		obj = super.clone();
	} catch (CloneNotSupportedException e) {
		// TODO: handle exception
		System.err.println("Not supported clonerable");
		;
	}
	return (Resume) obj;
}

public Resume deepclone() {
	Object obj = null;
	try {
		ByteArrayOutputStream baos = new ByteArrayOutputStream();
		ObjectOutputStream oos = new ObjectOutputStream(baos);
		oos.writeObject(this);

		ByteArrayInputStream bais = new ByteArrayInputStream(baos.toByteArray());
		ObjectInputStream ois = new ObjectInputStream(bais);
		obj = ois.readObject();
	} catch (IOException e) {
		// TODO: handle exception
		System.err.println(e.getMessage());
	} catch (ClassNotFoundException e) {
		// TODO Auto-generated catch block
		e.printStackTrace();
	}

	return (Resume) obj;
```

```
    }
}
public class Photo implements Serializable {

    private static final long serialVersionUID = 4455555666L;

    private int size;

    public Photo(int size) {
        super();
        this.size = size;
    }

    public int getSize() {
        return size;
    }

    public void setSize(int size) {
        this.size = size;
    }

    @Override
    public String toString() {
        return "Photo [size=" + size + "]";
    }
}
public class Client {
    public static void main(String[] args) {
        Photo photo = new Photo(16);
        Resume resume = new Resume("ysa", 22, photo);
        //浅拷贝
        Resume resume1 = resume.clone();
        photo.setSize(32);
        System.out.println("浅拷贝： ");
        System.out.println("resume:" + resume);
        System.out.println("resume1:" + resume1);
        //深拷贝
        Resume resume2 = resume.deepclone();
        photo.setSize(64);
        System.out.println("深拷贝： ");
        System.out.println("resume:" + resume);
        System.out.println("resume2:" + resume2);

        //选择深拷贝还是浅拷贝复制照片
```

```
        Resume newResume = new Resume("ljz", 25, new Photo(64));
        Scanner scanner = new Scanner(System.in);
        String str = scanner.next();
        if ("y".equalsIgnoreCase(str)) {
            newResume = resume.deepclone();
            System.out.println("deepclone");
        } else if ("n".equalsIgnoreCase(str)) {
            newResume = resume.clone();
            System.out.println("clone");
        }
    }
}
```

五、实验小结

通过这次实验，我熟悉了迭代器模式、适配器模式、模板方式模式、工厂方法模式、单例模式 、原型模式这些设计模式，收益匪浅。