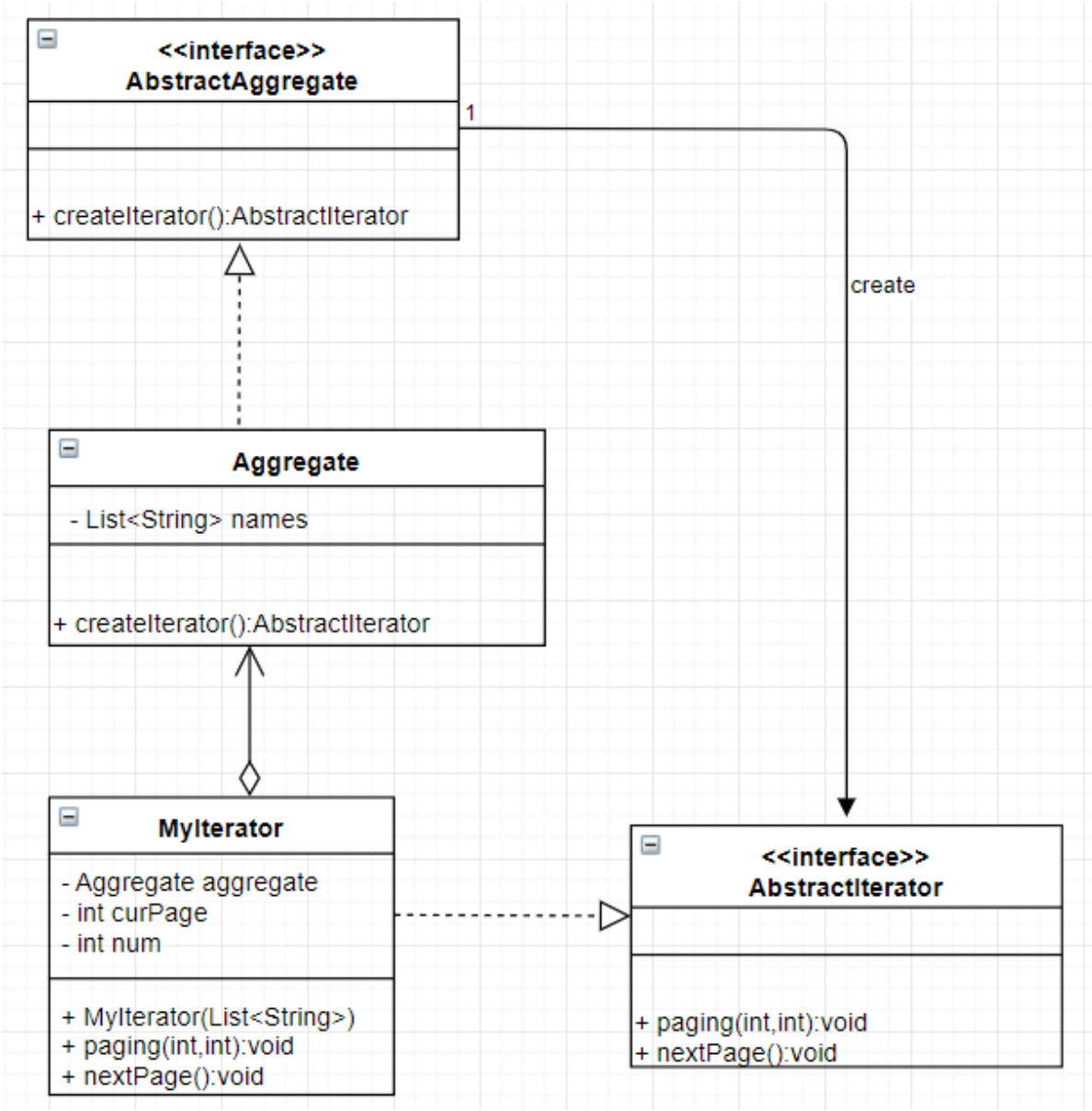
设计模式实验2

1.


<<interface>>
AbstractAggregate
+ createIterator():AbstractIterator
1
create
Aggregate
- List<String> names
+ createIterator():AbstractIterator
MyIterator
- Aggregate aggregate
- int curPage
- int num
+ MyIterator(List<String>)
+ paging(int,int):void
+ nextPage():void
<<interface>>
AbstractIterator
+ paging(int,int):void
+ nextPage():void

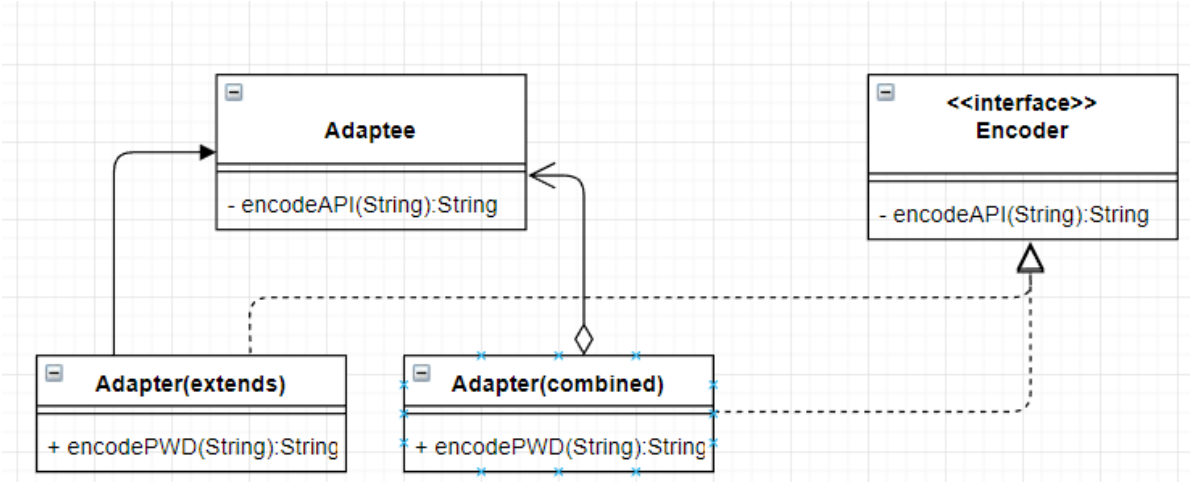
```java
package H1;

public interface AbstractAggregate {
    1 implementation
    AbstractIterator createIterator();
}
```

```java
public interface AbstractIterator {
    1 implementation
    public void paging(int a, int b);

    1 implementation
    public void nextPage();

    1 implementation
    public void previousPage();

}
```

```java
import java.util.ArrayList;

1 usage
public class Aggregate implements AbstractAggregate{
    3 usages
    private ArrayList<String> names;

    public Aggregate() {
        names = new ArrayList<String>();
        for (int i = 0; i < 100; i++) {
            names.add(i, element: "用户" + (i + 1));
        }
    }

    @Override
    public AbstractIterator createIterator() { return new MyIterator(names); }

}
```

```
class MyIterator implements AbstractIterator {
    private Aggregate aggregate;
    8 usages
    private int currentPage;  //当前页号
    8 usages
    private int num;  //每页个数

    8 usages
    private List<String> names;

    1 usage
    public MyIterator(List<String> names) { this.names = names; }

    //分页输出的方法  a表示每页的数量  ，b表示页号
    @Override
    public void paging(int a, int b) {
        num = a;
        currentPage = b;
        if (names.size() % a == 0 && b >= (names.size() / a + 1)) {
            System.out.println("您查看的页内容为空!");
        } else if (names.size() % a != 0 && b >= (names.size() / a + 2)) {
            System.out.println("您查看的页内容为空!");
        } else {
            try {
                for (int i = ((b - 1) * a); i < (((b - 1) * a) + a); i++) {
                    System.out.println(names.get(i));
                }
            } catch (Exception e) {
                System.out.println("我是有底线的————————————————————");
            }
        }
    }

  @Override
  public void nextPage() {
      try {
          for (int i = num * currentPage; i < num * currentPage + num; i++) {
              System.out.println(names.get(i));
          }
          currentPage += 1;
      } catch (Exception e) {...}

  }
```

2.




```java
public class Adaptee {
    1 usage
    public String encodeAPI(String password){
        return password.toUpperCase();
    }
}
```

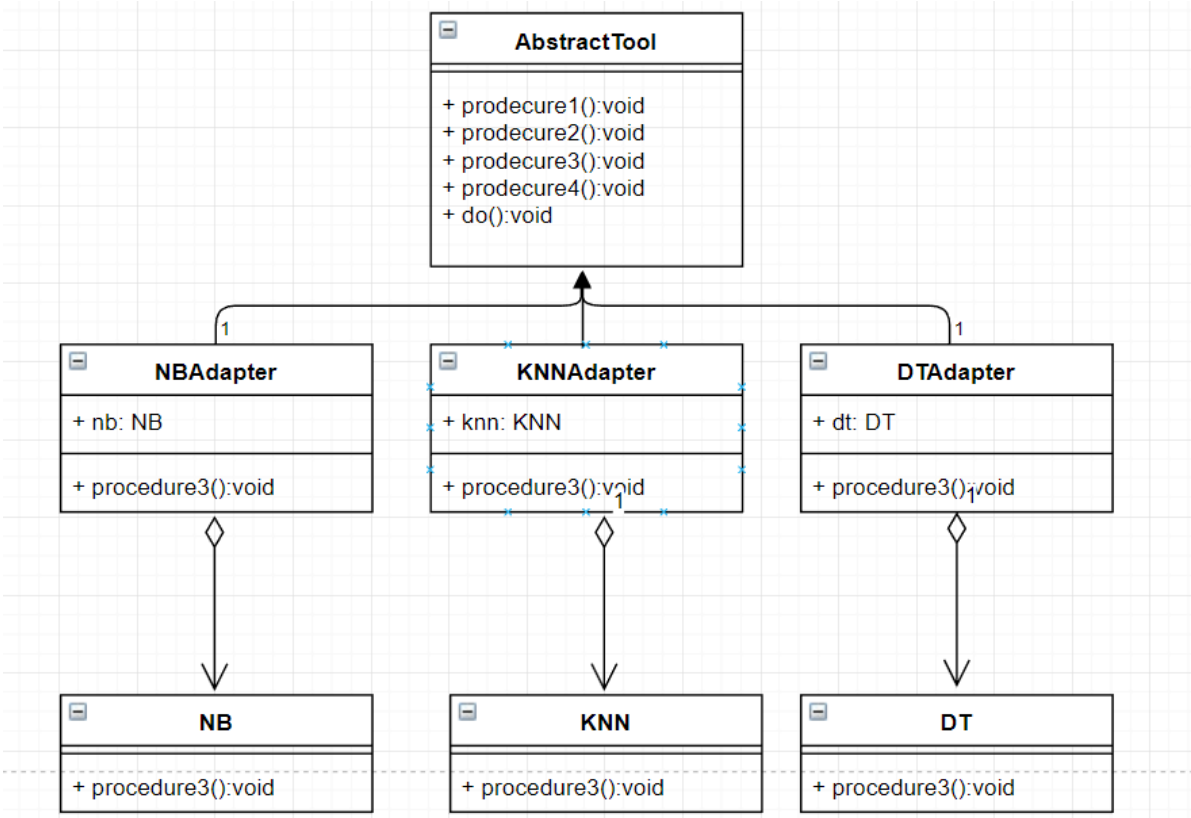
```java
public class Adapter_combined implements Encoder{
    2 usages
    private Adaptee adaptee;

    public Adapter_combined(Adaptee adaptee) { this.adaptee = adaptee; }

    1 usage
    @Override
    public String encodePwd(String password) { return adaptee.encodeAPI(password); }

}
```

```java
public class Adapter_extends extends Adaptee implements Encoder{
    1 usage
    @Override
    public String encodePwd(String password) {
        return this.encodePwd(password);
    }
}
```

```java
public interface Encoder {
    1 usage  2 implementations
    String encodePwd(String password);
}
```

3.

使用适配器委托模式和模板模式




```java
public abstract class AbstractTool {
    1 usage
    public final void procedure1(){}
    1 usage
    public final void procedure2(){}
    1 usage
    public final void procedure4(){}

    1 usage  3 implementations
    public abstract void procedure3();

    3 usages
    public final void doSomething()
    {
        this.procedure1();
        this.procedure2();
        this.procedure3();
        this.procedure4();
    }
}
```

```java
public class DT {
    1 usage
    public DT() {
    }
    1 usage
    public void Procedure3()
    {
        System.out.println("DT procedure3");
    }
}
```

```java
public class DTAdapter extends AbstractTool{
    1 usage
    DT dt = new DT();

    1 usage
    public DTAdapter()
    {}
    1 usage
    @Override
    public void procedure3() { this.dt.Procedure3(); }
}
```

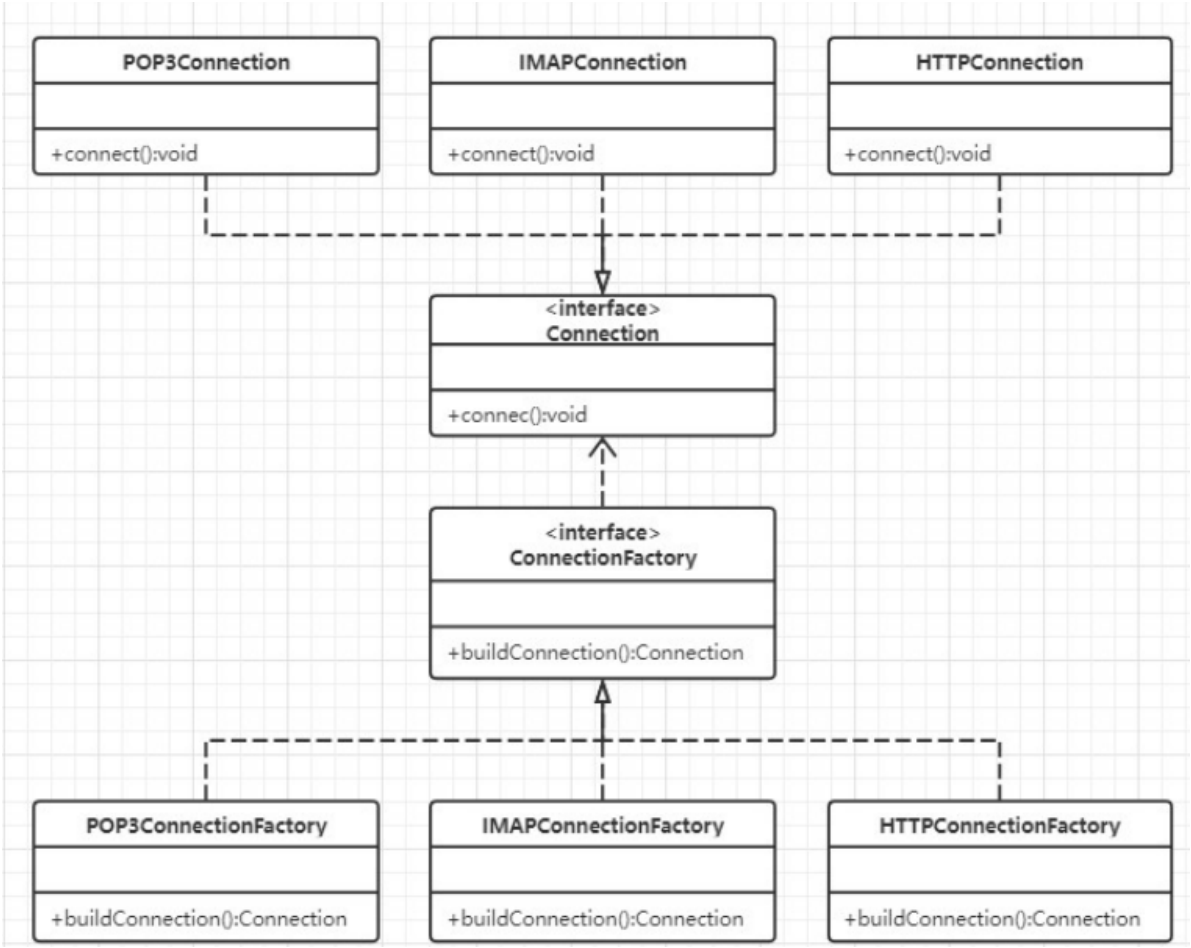
```java
public class KNN {
    1 usage
    public KNN() {
    }
    1 usage
    public void Procedure3()
    {
        System.out.println("KNN procedure3");
    }
}
```

```java
public class KNNAdapter extends AbstractTool{
    1 usage
    KNN knn = new KNN();

    1 usage
    @Override
    public void procedure3() { this.knn.Procedure3(); }
}
```

```java
public class NB {
    1 usage
    public NB() {
    }
    1 usage
    public void Procedure3()
    {
        System.out.println("NB procedure3");
    }
}
```

```java
public class NBAdapter extends AbstractTool{
    1 usage
    NB nb = new NB();
    1 usage
    @Override
    public void procedure3() { this.nb.Procedure3(); }
}
```

4.




```java
public interface Connection {
    3 implementations
    public void getConnect();
}
```

```java
public interface ConnectionFactory {
    3 implementations
    public Connection createConnection();
}
```

```java
public class HTTP implements Connection{
    @Override
    public void getConnect() {
        System.out.println("HTTP connected");
    }
}
```

```java
public class HTTPFactory implements ConnectionFactory{
    @Override
    public Connection createConnection() {
        return new HTTP();
    }
}
```

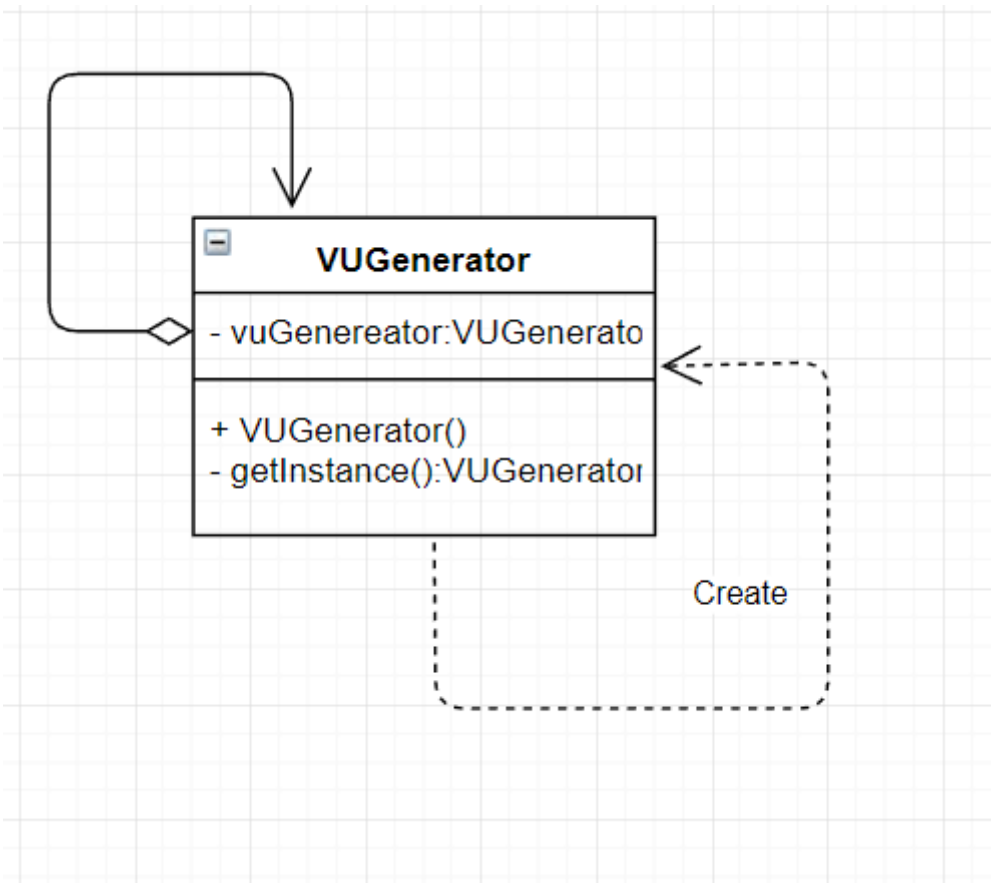
```java
public class IMAP implements Connection{
    @Override
    public void getConnect() {
        System.out.println("IMAP connected");
    }
}
```

```java
public class IMAPFactory implements ConnectionFactory{
    @Override
    public Connection createConnection() {
        return new IMAP();
    }
}
```

```java
public class POP3 implements Connection{
    @Override
    public void getConnect() {
        System.out.println("POP3 connected");
    }
}
```

```java
public class POP3Factory implements ConnectionFactory{
    @Override
    public Connection createConnection() {
        return new POP3();
    }
}
```

5.

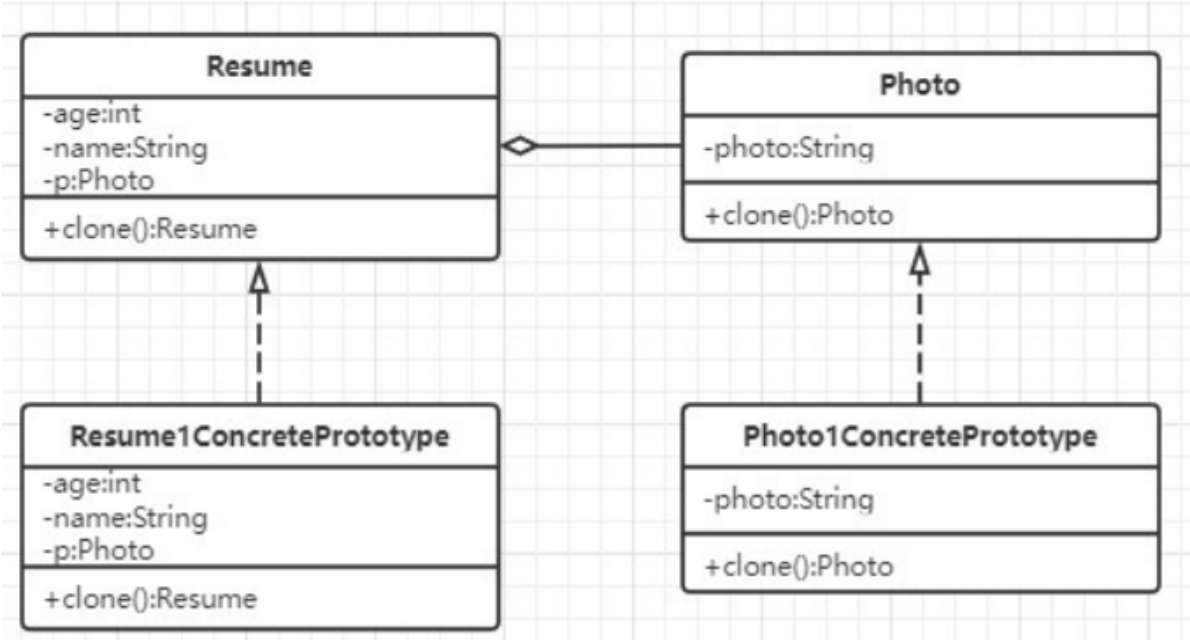



```
public class VUGenerator {
    1 usage
    private static final VUGenerator vug = new VUGenerator();
    1 usage
    private VUGenerator(){}
    public static VUGenerator getInstance()
    {
        return vug;
    }
}
```

```java
public class VUGenerator2 {
    4 usages
    private volatile static VUGenerator2 vuGenerator2 = null;
    1 usage
    private VUGenerator2(){}

    public static VUGenerator2 getInstance() {
        if(vuGenerator2 == null)
        {
            synchronized (VUGenerator2.class){
                if(vuGenerator2 == null)
                    vuGenerator2 = new VUGenerator2();
            }
        }
        return vuGenerator2;
    }
}
```

```java
public class VUGenerator3 {
    1 usage
    private VUGenerator3() {
    }

    1 usage
    private static class HolderClass{
        1 usage
        private final static VUGenerator3 vuG = new VUGenerator3();
    }
    public static VUGenerator3 getInstance() { return HolderClass.vuG; }
}
```

6.



```java
public class Client {
    public static void main(String[] args) {
        Photo photo=new Photo( size: 16);
        Resume resume=new Resume( name: "ysa",  age: 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( name: "zly",  age: 25, new Photo( size: 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");
        }
    }
}
```

```
7 usages
public class Photo implements Serializable {

    private static final long serialVersionUID = 4455555666L;

    4 usages
    private int size;

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

    public int getSize() { return size; }

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

    @Override
    public String toString() { return "Photo [size=" + size + "]"; }
}
```

```
public class Resume implements Cloneable, Serializable {

    private static final long serialVersionUID = 4455666L;

    2 usages
    private String name;
    2 usages
    private int age;
    4 usages
    private Photo photo;

    2 usages
    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() + "]";
    }
```

```java
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;
}
```

2 usages

```java
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;
}
```

实验总结

1.Iterator模式：

这个模式试图隐藏集合的内部表示，又同时可以使用户依次访问集合中的元素。现在STL和Java的跌代器就是应用这个模式的结果。

adapter模式：

Adapter模式的目的就是将一个类的接口转换为用户希望的接口，使得由于接口不兼容而不能一起工作的各个类可以一起工作。例如在一个软件里面可能使用了以前一个版本的类库。不幸的是这个类库的效率极高却和现在的接口不兼容，为了继续复用这个类库我们就可以使用Adapter模式，在原来的类库和现在的接口中间实现一个适配器，使得我们可以用现在的结构调用以前的类库。

模板方法模式：

Template Method模式的意图是：“定义一个操作中的骨架，而将一些步骤延迟到子类中。这使得子类可以不改变一个算法的结构即可以重定义该算法的某些特定步骤。这一模式和Strategy模式似乎和相似，但是他们的关注点不同。策略模式主要用于算法的替换，但是模板方法模式主要用于算法中特定步骤地替换。一个应用模板方法模式的例子是数据库操作。对于数据库操作可以有很多中，比如查询、更新。查询又可以分为连接数据库、发送请求命令、解析结果等等步骤。对于不同的数据库，比如Oracle和SQL2000，它们连接数据库、命令格式可能有所不同，但是就查询和更新着两个操作来说它们的步骤是相同的。这个时候，我们可以应用模板方法模式，为查询、更新操作建立一个抽象的算法，具体的步骤交给子类来实现。如果对于策略模式，我们替换的将是查询和更新着两个操作。

工厂方法模式：

这一模式的关键是掌握“何时应用这一模式”，事实上我觉得这也是所有设计模式的关键。一个已知的应用就是MFC中关于Document和Frame之间的关系。通常在生成一个多文档程序时，VC会为你创建一个Frame类和Document类，你的Frame类可以用来相应OnFileNew消息，然后创建一个Document对象。但是对于Windows的消息系统来说，它并不知道用户程序中创建的Document类有什么特性，对于它来说，它所看到是CFrame对象和CDocument对象。Factory Method模式可以保证其他对象不需要知道具体对象的类型而管理这些对象，这一模式通常用于制定框架。

单例模式：

虽然我们可以提供一个全局访问点，但实际上这个模式也可以应用到局部。应用这个模式一个好处就是可以“按需分配”，同时也封装了对象的获取过程。
这个模式在实现过程中可以进行变化，例如在Instance()方法上添加参数Boolean bAlloc，用于指定当实例不存在的时候是否进行创建。这样做是考虑到，有些时候我们获得实例的目的不是为了修改，而是为了读取。这个时候，返回一个空实例和返回一个没有被修改过的实例在逻辑上是相同的。例如，这个对象是一个数组时，一个“空数组”和一个“空白的数组”是相同的。

原型模式：

原型模式（Prototype Pattern）是用于创建重复的对象，同时又能保证性能。这种类型的设计模式属于创建型模式，它提供了一种创建对象的最佳方式。

应用实例： 1、细胞分裂。 2、JAVA 中的 Object clone() 方法。3、Java序列化与反序列化可以实现深度克隆