设计模式实验 1-4

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（NaiveBayes）算法、决策树（DecisionTree）算法、K 最近邻（K-NearestNeighbor,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。

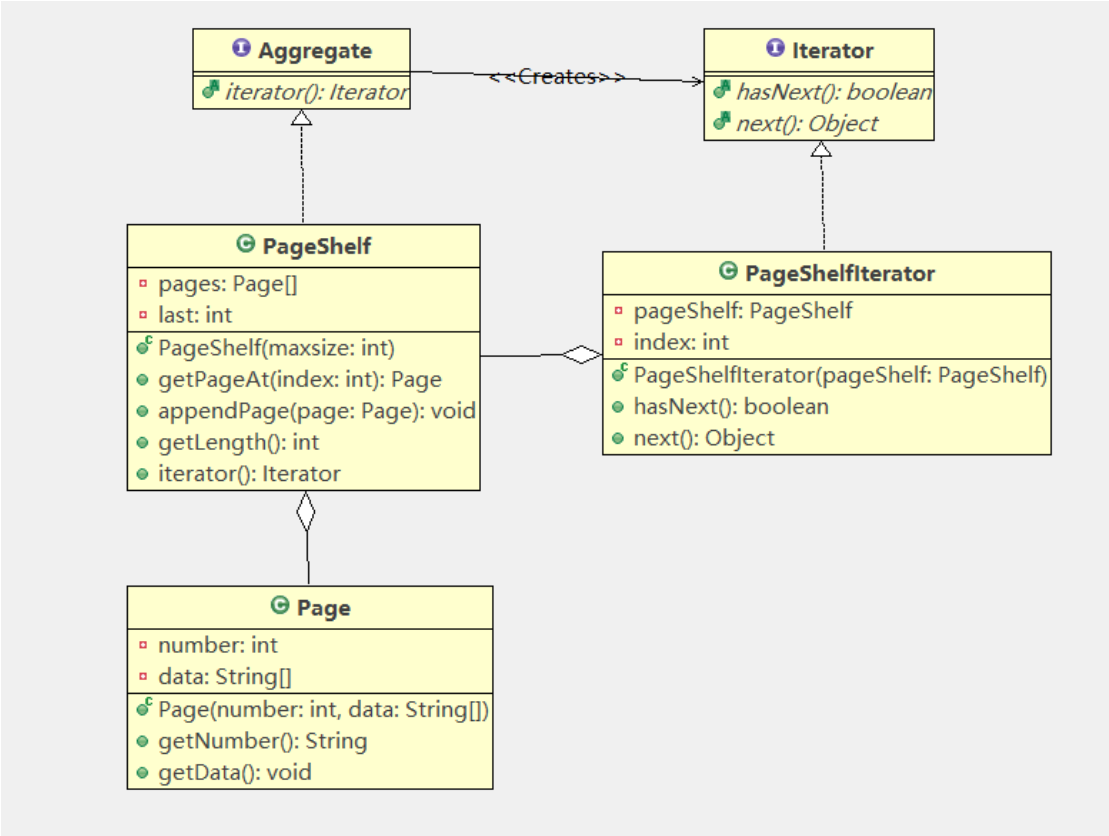
4. **工厂方法模式**：在某网络管理软件中，需要为不同的网络协议提供不同的连

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。

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。

1. 迭代器模式



代码实现：

```
public interface Aggregate {

    public abstract Iterator iterator();
}

public interface Iterator {

    public abstract boolean hasNext();
```

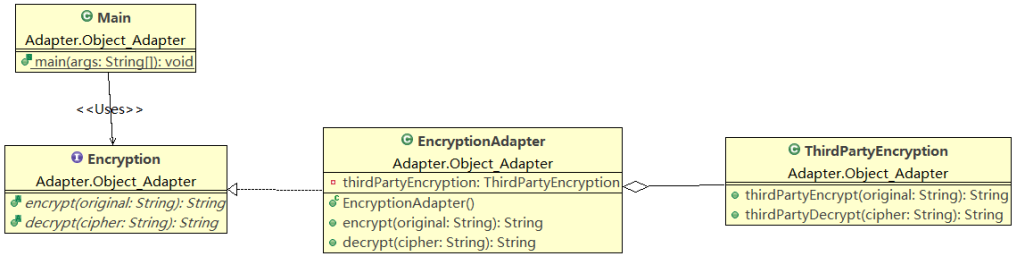
```
    public abstract Object next();
}

public class Page {

    private int number;
    private String[] data;

    public Page(int number, String[] data) {
        this.number = number;
        this.data = data;
    }

    public String getNumber() {
        return "This is Page " + number + ".";
    }

    public void getData() {
        for(int i = 0; i < data.length; i++) {
            System.out.println(data[i]);
        }
    }

}

public class PageShelf implements Aggregate{

    private Page[] pages;
    private int last = 0;

    public PageShelf (int maxsize) {
        this.pages = new Page[maxsize];
    }

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

    public void appendPage(Page page) {
        this.pages[last] = page;
        last++;
    }

    public int getLength() {
```

```
        return last;
    }

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

}

public class PageShelfIterator implements Iterator{

    private PageShelf pageShelf;
    private int index;

    public PageShelfIterator(PageShelf pageShelf) {
        this.pageShelf = pageShelf;
        this.index = 0;
    }

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

    @Override
    public Object next() {
        Page page = pageShelf.getPageAt(index);
        index++;
        return page;
    }

}

public class Main {

    public static void main(String[] args) {

        PageShelf pageShelf = new PageShelf(100);
```

```
        // the data number of one page can contain
        int pageSize = 3;
        int pageNum = 1;

        // the original data
        String[] originalArray = {"a", "b", "c", "d", "e", "f", "g",
"h"};

        // split the origin data into size of pages
        List<String[]> splitArrays = splitArray(originalArray,
pageSize);

        for (String[] arr : splitArrays) {
            pageShelf.appendPage(new Page(pageNum, arr));
            pageNum++;
        }

        Iterator it = pageShelf.iterator();
        while (it.hasNext()) {
            Page page = (Page)it.next();
            System.out.println(page.getNumber());
            page.getData();
        }

    }

    public static List<String[]> splitArray(String[] originalArray, int
pageSize) {
        List<String[]> result = new ArrayList<>();
        int length = originalArray.length;

        for (int i = 0; i < length; i += pageSize) {
            int end = Math.min(i + pageSize, length);
            String[] subArray = Arrays.copyOfRange(originalArray, i,
end);
            result.add(subArray);
        }

        return result;
    }
}
```

2. 适配器模式

对象适配器（组合+委托）

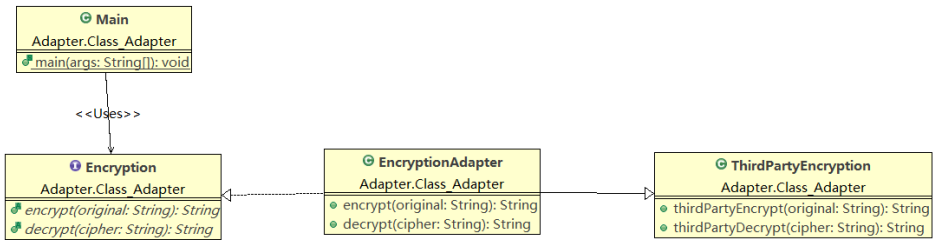


代码实现：

```
public interface Encryption {

    public abstract String encrypt(String original);
    public abstract String decrypt(String cipher);

}

public class ThirdPartyEncryption {

    public String thirdPartyEncrypt(String original) {
        return "The Encrypt Result of " + original + " through Third
Party Encrypt" ;

    }

    public String thirdPartyDecrypt(String cipher) {
        return "The Decrypt Result of " + cipher + " through Third Party
Decrypt" ;
    }

}

public class EncryptionAdapter implements Encryption{

    private ThirdPartyEncryption thirdPartyEncryption;

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

    @Override
```

```
    public String encrypt(String original) {
        return thirdPartyEncryption.thirdPartyEncrypt(original);
    }

    @Override
    public String decrypt(String cipher) {
        return thirdPartyEncryption.thirdPartyDecrypt(cipher);
    }

}

public class Main {

    public static void main(String[] args) {
        Encryption encryption = new EncryptionAdapter();
        System.out.println(encryption.encrypt("'This is the string of original data'"));
        System.out.println(encryption.decrypt("'This is the string of cipher'"));
    }

}
```

类适配器（继承）

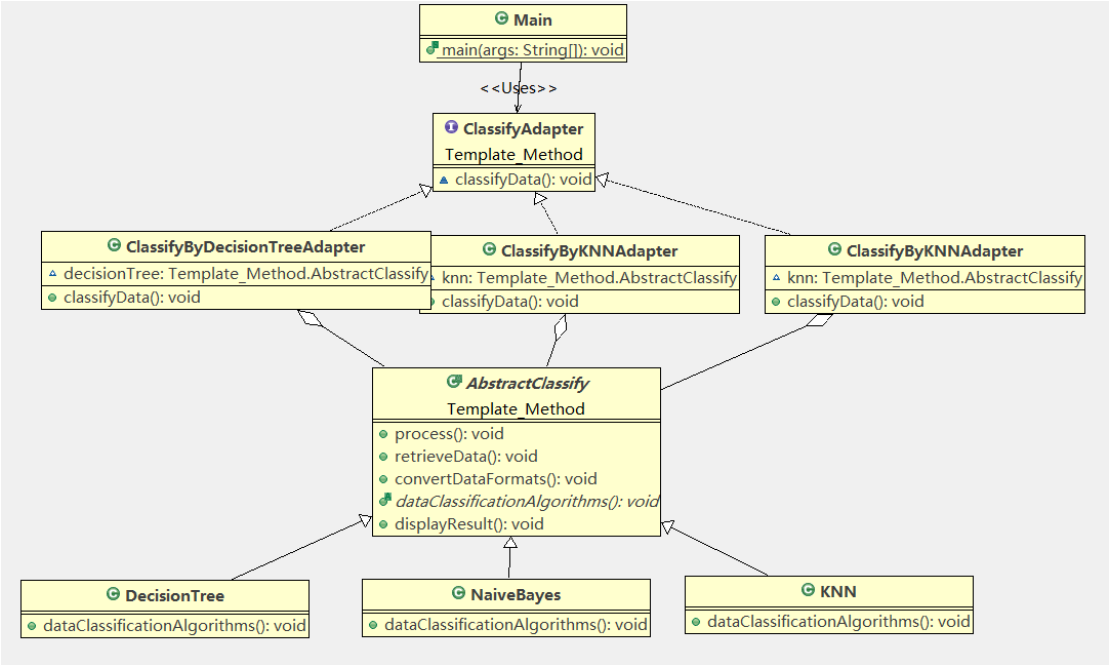


代码实现：

```
public interface Encryption {

    public abstract String encrypt(String original);
    public abstract String decrypt(String cipher);

}

public class ThirdPartyEncryption {
```

```
    public String thirdPartyEncrypt(String original) {
        return "The Encrypt Result of " + original + " through Third
Party Encrypt" ;

    }

    public String thirdPartyDecrypt(String cipher) {
        return "The Decrypt Result of " + cipher + " through Third Party
Decrypt" ;
    }

}

public class EncryptionAdapter extends ThirdPartyEncryption implements
Encryption {



    @Override
    public String encrypt(String original) {
        return thirdPartyEncrypt(original);
    }

    @Override
    public String decrypt(String cipher) {
        return thirdPartyDecrypt(cipher);
    }

}

public class Main {

    public static void main(String[] args) {
        Encryption encryption = new EncryptionAdapter();
        System.out.println(encryption.encrypt("'This is the string of
original data'"));
        System.out.println(encryption.decrypt("'This is the string of
cipher'"));
    }

}
```

3. 模板方式模式和适配器模式



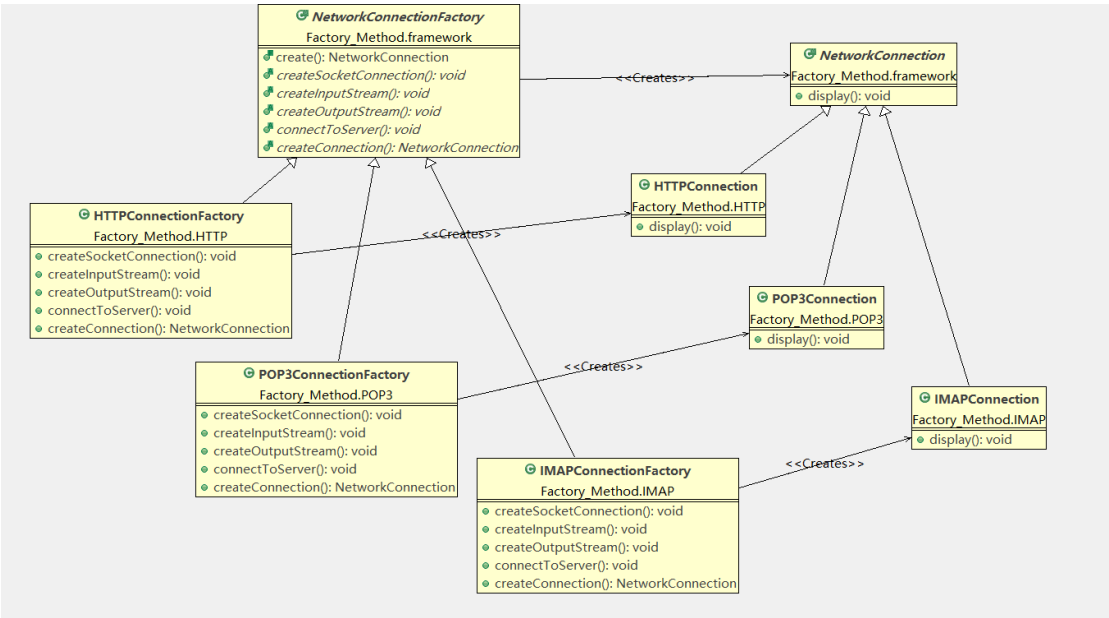
代码实现：

```
public abstract class AbstractClassify {

    public final void process() {
        retrieveData();
        convertDataFormats();
        dataClassificationAlgorithms();
        displayResult();
    }

    public void retrieveData() {
        System.out.println("Data have been retrieved");
    }

    public void convertDataFormats() {
        System.out.println("The Format of Data have been converted");
    }

    public abstract void dataClassificationAlgorithms();

    public void displayResult() {
        System.out.println("Displaying the result");
    }

}
```

```
public class NaiveBayes extends AbstractClassify{

    @Override
    public void dataClassificationAlgorithms() {
        System.out.println("Apply NaiveBayes to do Data Mining");

    }

}

public class DecisionTree extends AbstractClassify{

    @Override
    public void dataClassificationAlgorithms() {
        System.out.println("Apply DecisionTree to do Data Mining");
    }

}

public class KNN extends AbstractClassify{

    @Override
    public void dataClassificationAlgorithms() {
        System.out.println("Apply KNN to do Data Mining");

    }


}

public interface ClassifyAdapter {

    void classifyData();

}

public class ClassifyByNaiveBayesAdapter implements ClassifyAdapter{

    AbstractClassify naiveBayes = new NaiveBayes();

    @Override
    public void classifyData() {
        naiveBayes.process();
```

```
    }

}

public class ClassifyByDecisionTreeAdapter implements ClassifyAdapter{

    AbstractClassify decisionTree = new DecisionTree();

    @Override
    public void classifyData() {
        decisionTree.process();

    }

}

public class ClassifyByKNNAdapter implements ClassifyAdapter{

    AbstractClassify knn= new KNN();

    @Override
    public void classifyData() {
        knn.process();

    }

}

public class Main {

    public static void main(String[] args) {
        ClassifyAdapter[] algorithms = {
                new ClassifyByNaiveBayesAdapter(),
                new ClassifyByDecisionTreeAdapter(),
                new ClassifyByKNNAdapter()
            };

            for (ClassifyAdapter algorithm : algorithms) {
                algorithm.classifyData();
                System.out.println("");
            }

    }
}
```

4. 工厂方法模式



代码实现：

```
public abstract class NetworkConnectionFactory {

    public final NetworkConnection create() {
        createSocketConnection();
        createInputStream();
        createOutputStream();
        connectToServer();
        NetworkConnection networkConnection = createConnection();

        return networkConnection;


    }

    public abstract void createSocketConnection();

    public abstract void createInputStream();

    public abstract void createOutputStream();

    public abstract void connectToServer();

    public abstract NetworkConnection createConnection();


}
```

```java
public abstract class NetworkConnection {

    public void display() {

    }

}

public class POP3ConnectionFactory extends NetworkConnectionFactory{

    @Override
    public void createSocketConnection() {
        // TODO Auto-generated method stub

    }

    @Override
    public void createInputStream() {
        // TODO Auto-generated method stub

    }

    @Override
    public void createOutputStream() {
        // TODO Auto-generated method stub

    }

    @Override
    public void connectToServer() {
        // TODO Auto-generated method stub

    }

    @Override
    public NetworkConnection createConnection() {
        // TODO Auto-generated method stub
        return new POP3Connection();
    }

}

public class POP3Connection extends NetworkConnection{
```

```
    public void display() {
        System.out.println("POP3Connection has been created");
    }


}

public class IMAPConnectionFactory extends NetworkConnectionFactory{

    @Override
    public void createSocketConnection() {
        // TODO Auto-generated method stub

    }

    @Override
    public void createInputStream() {
        // TODO Auto-generated method stub

    }

    @Override
    public void createOutputStream() {
        // TODO Auto-generated method stub

    }

    @Override
    public void connectToServer() {
        // TODO Auto-generated method stub

    }

    @Override
    public NetworkConnection createConnection() {
        // TODO Auto-generated method stub
        return new IMAPConnection();
    }

}

public class IMAPConnection extends NetworkConnection{
```

```
    public void display() {
        System.out.println("IMAPConnection has been created");
    }


}

public class HTTPConnectionFactory extends NetworkConnectionFactory{

    @Override
    public void createSocketConnection() {
        // TODO Auto-generated method stub

    }

    @Override
    public void createInputStream() {
        // TODO Auto-generated method stub

    }

    @Override
    public void createOutputStream() {
        // TODO Auto-generated method stub

    }

    @Override
    public void connectToServer() {
        // TODO Auto-generated method stub

    }

    @Override
    public NetworkConnection createConnection() {
        // TODO Auto-generated method stub
        return new HTTPConnection();
    }

}

public class HTTPConnection extends NetworkConnection{

    public void display() {
```

```
        System.out.println("HTTPConnection has been created");
    }


}

public class Main {

    public static void main(String[] args) {
        NetworkConnectionFactory[] factories = {
                new POP3ConnectionFactory(),
                new IMAPConnectionFactory(),
                new HTTPConnectionFactory()
            };

            for (NetworkConnectionFactory factory : factories) {
                NetworkConnection networkConnection = factory.create();
                networkConnection.display();
            }
    }

}
```

五、实验小结

请总结本次实验的体会，包括学会了什么、遇到哪些问题、如何解决这些问题以及存在哪些有待改进的地方。

通过实际编写代码，我加强了对迭代器、适配器、模板方法和工厂方法四种设计模式的认识，了解了这四种设计模式在实际运用中的作用和意义。

迭代器模式中，遇到的问题：　如何管理分页逻辑？

解决方法：　在迭代器中维护当前页数和每页元素数，确保按照分页逻辑遍历数据。

工厂方法模式中，遇到的问题：　如何管理多种不同的连接类？

解决方法：　定义一个通用的连接接口，并为每种协议创建一个具体的连接类。