

深圳大学实验报告

课程名称： 软件体系结构与设计模式

实验项目名称： 实验 3 结构型设计模式分析与应用

学院： 计算机与软件学院

专业： 软件工程

指导教师： 毛斐巧

报告人： 郑彦薇 学号： 2020151022 班级： 软件工程 01 班

实验时间： 2023 年 4.19,4.26,5.3,5.10,5.17（周三）

实验报告提交时间： 2023/5/12

教务部制

一、 实验目的与要求：

1. 对 GoF 23 个设计模式中的结构型设计模式的 UML 图形表示，进行结构分析，发现与总结出反复出现的 UML 图形结构。
2. 结合实例，熟练绘制常见的结构型设计模式结构图。
3. 理解每一种结构型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、实验内容


1. 模式分析


以下为 GoF 23 个设计模式的快速游览：

ABOUT DESIGN PATTERNS

This Design Patterns refcard provides a quick reference to the original 23 Gang of Four design patterns, as listed in the book *Design Patterns: Elements of Reusable Object-Oriented Software*. Each pattern includes class diagrams, explanation, usage information, and a real world example.

 **Creational Patterns:** Used to construct objects such that they can be decoupled from their implementing system.

 **Structural Patterns:** Used to form large object structures between many disparate objects.

 **Behavioral Patterns:** Used to manage algorithms, relationships, and responsibilities between objects.

Object Scope: Deals with object relationships that can be changed at runtime.

Class Scope: Deals with class relationships that can be changed at compile time.

 Abstract Factory	 Decorator	 Prototype
 Adapter	 Facade	 Proxy
 Bridge	 Factory Method	 Observer
 Builder	 Flyweight	 Singleton
 Chain of Responsibility	 Interpreter	 State
 Command	 Iterator	 Strategy
 Composite	 Mediator	 Template Method
	 Memento	 Visitor

图 1 GoF 23 个设计模式的快速浏览

图 1. GoF(Gang of Four)设计模式的三大类：创建型设计模式（Creational Patterns）、结构型设计模式（Structural Patterns）、行为设计模式（Behavioral Patterns）。Object Scope 所标记的设计模式可用于**运行时**改变对象关系，Class Scope 所标记的设计模式不能用于运行时，而只能用于**编译时**。

我们以结构型设计模式为主要研究内容，下面列举这 7 种结构型设计模式。我们暂不分析它们的应用场景，而只是研究 UML 图的结构。

UML 图的结构主要表现为：继承（抽象）、关联、组合或聚合的三种关系，我们

来看看这三种关系的各种可能的配合，可以变化出哪些可能的具体形式。

1.1 关联后抽象

ADAPTER Class and Object Structural

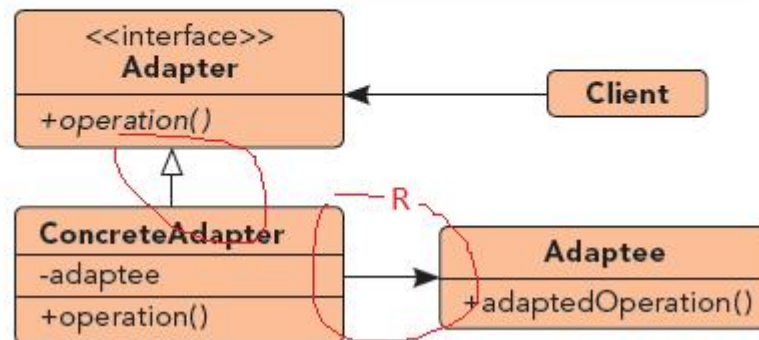


图2 Adapter（适配器）设计模式

采用的 UML 结构为：**关联后抽象**。将带关联的类 **ConcreteAdapter** 为接口 **Adapter**。其中可运行时改变的地方由“R”标记，即由 **ConcreteAdapter** 管理的 **adaptee** 成员变量可在**运行时**更改。

1.2 关联的集合

FACADE Object Structural

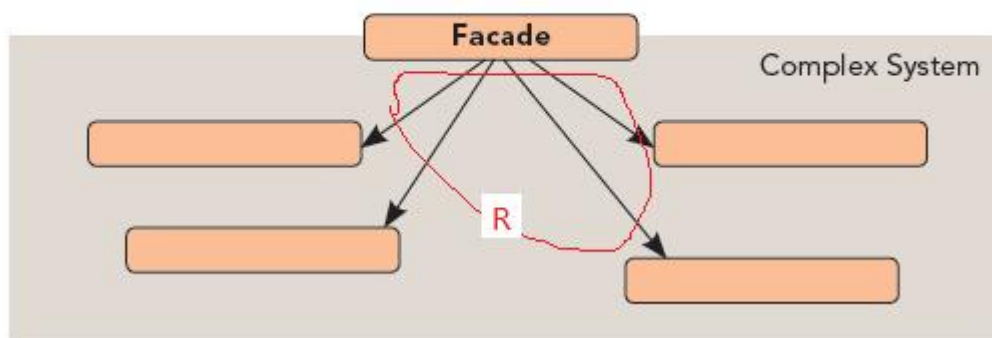


图3 Façade（门面）设计模式。采用的 UML 结构为：**关联集合**

1.3 组合接口

BRIDGE

Object Structural

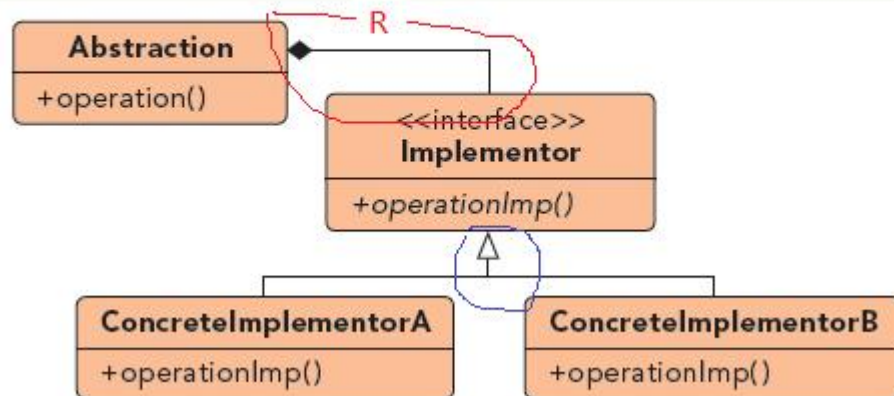


图4 Bridge（桥）设计模式。采用的UML结构为：[组合接口](#)

1.4 递归聚合接口

DECORATOR

Object Structural

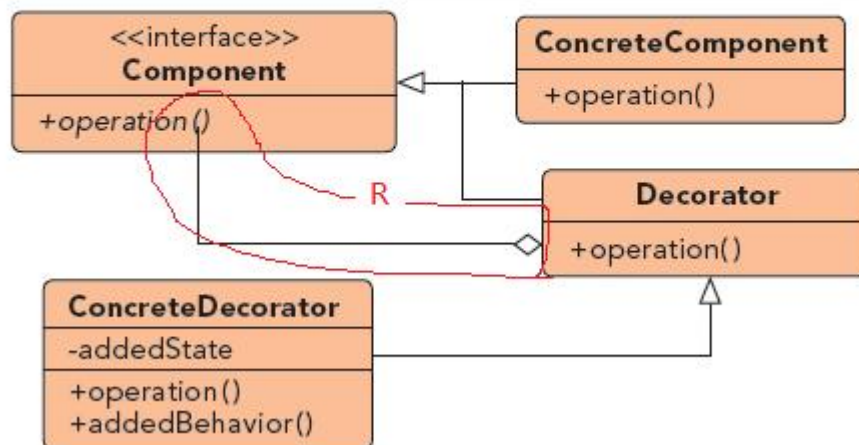


图5 Decorator（装饰器）设计模式。采用的UML结构为：[递归聚合接口](#)

回答问题 Q1: 下面的设计模式属于 1.1~1.4 中的哪一种，并给出说明。

PROXY

Object Structural

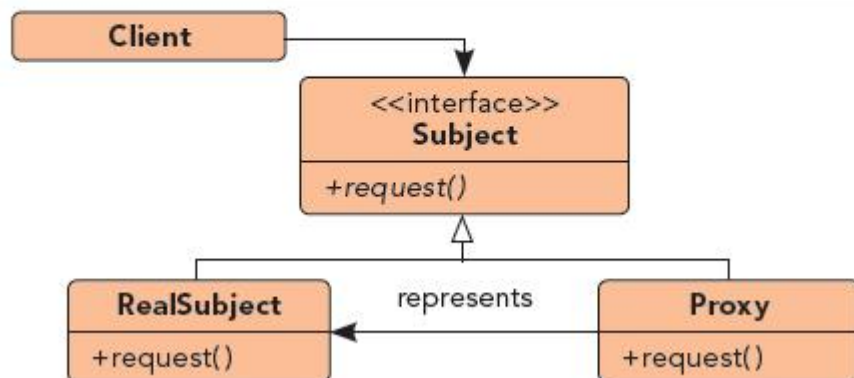


图 6 Proxy（代理）设计模式

COMPOSITE

Object Structural

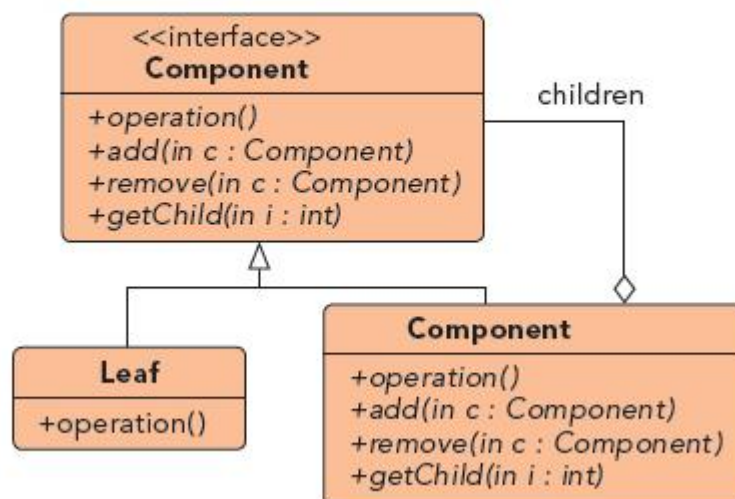


图 7 Composite（组合）设计模式

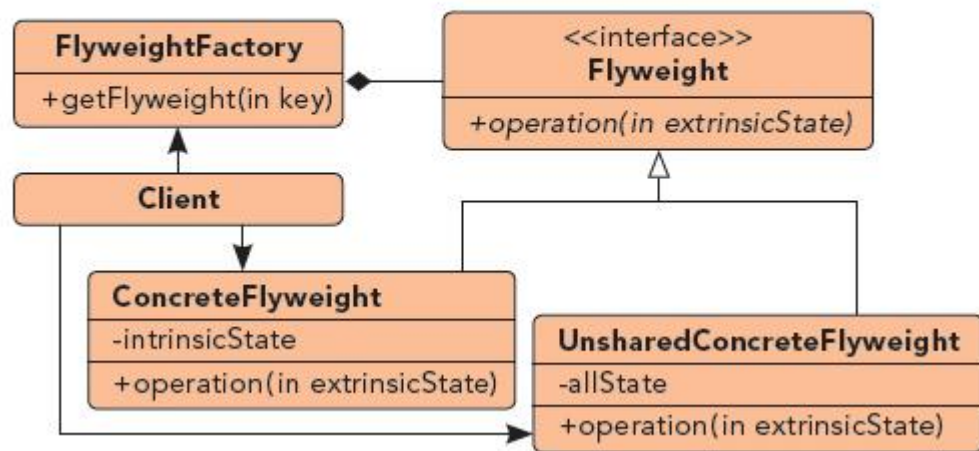


图 8 Flyweight（享元）设计模式

2. 在 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. 某手机美图 APP 软件支持多种不同的图像格式，例如 JPG、GIF、BMP 等常用图像格式，同时提供了多种不同的滤镜对图像进行处理，例如木刻滤镜(Cutout)、模糊滤镜(Blur)、锐化滤镜(Sharpen)、纹理滤镜(Texture)等。现采用桥接模式设计该 APP 软件，使得该软件能够为多种图像格式提供一系列图像处理滤镜，同时还能够很方便地增加新的图像格式和滤镜，绘制对应的类图并编程模拟实现。

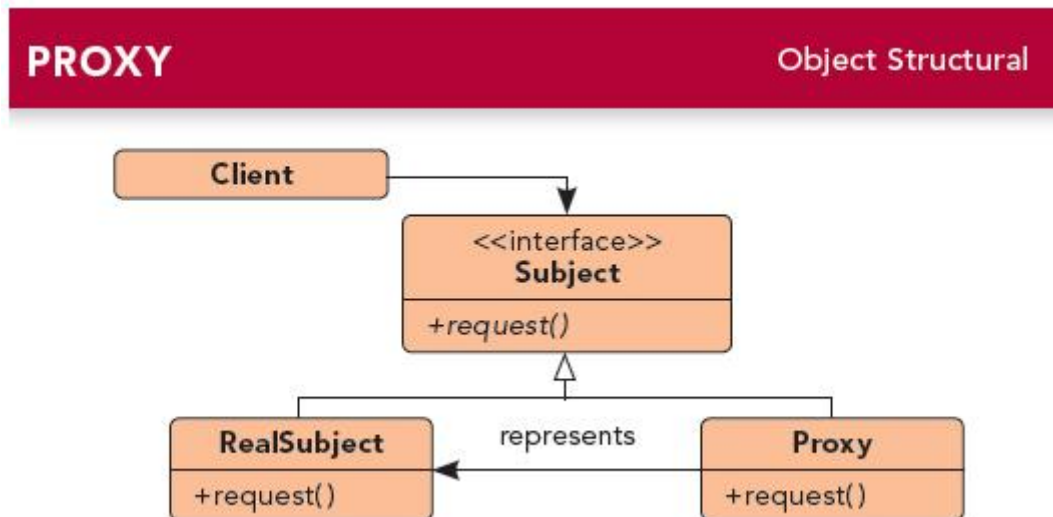
4. 某软件公司为新开发的智能手机控制与管理软件提供了一键备份功能，通过该功能可以将原本存储在手机中的通讯录、短信、照片、歌曲等资料，一次性全部拷贝到移动存储介质（例如 MMC 卡或 SD 卡）中。在实现过程中需要与多个已有的类进行交互，例如通讯录管理类、短信管理类等。为了降低系统的耦合度，试使用外观模式来设计并编程模拟实现该一键备份功能。

三、理解和分析报告、实践过程及结果等

1. 问题 Q1 的回答：

下面的设计模式属于 1.1~1.4 中的哪一种，并给出说明。

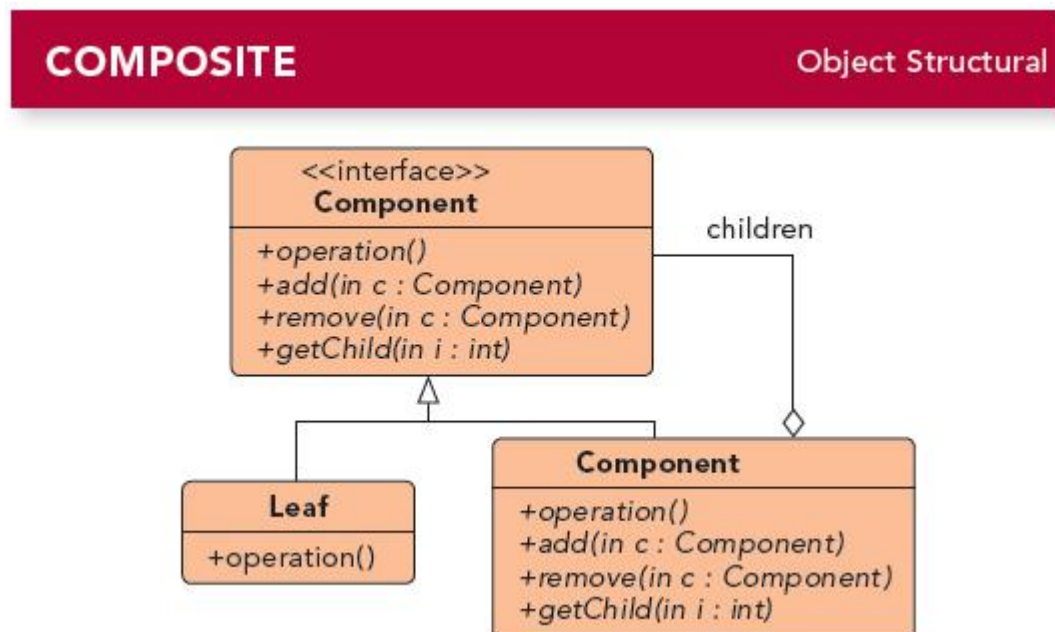
1.1. PROXY（代理）设计模式



这种设计模式采用的 uml 结构属于关联后抽象。

在上图所示的代理模式 uml 结构图中，可以看到 Subject 表示一个抽象类或者接口，在这个接口中定义了客户端 Client 需要调用的方法 request，而实际的方法实现是在 RealSubject 中。Proxy 继承了 Subject 抽象类（或实现了 Subject 接口），但在 Proxy 中并不是真正的实现，而是生成一个对 RealSubject 对象的引用 represents，再通过调用 represents.request() 来实现 Subject 中要求的方法即 request 方法。可以看出该种设计模式采用的 uml 结构是关联后抽象。

1.2. COMPOSITE（组合）设计模式

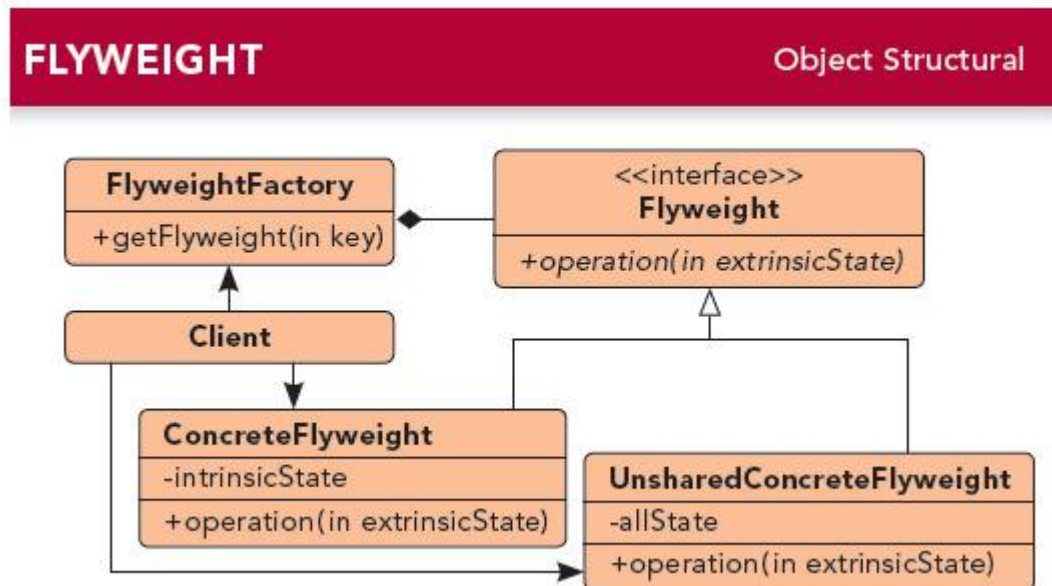


这种设计模式采用的 uml 结构属于递归聚合接口。

在上图所示的组合设计模式 uml 结构图中，interface Component 是组合中对象声明接口，它

能够在适当情况下实现所有类共有的接口默认行为，用于访问和管理 Component 子部件，它可以是抽象类或者接口。Leaf 在组合中表示叶子节点，Component 是非叶子节点，用于存储子部件，在 Component 接口中实现子部件的相关操作。该组合模式创建了对象组的属性结构，将对象组合成树状结构以表示“整体-部分”的层次关系，使得用户对单个对象和组合对象的访问具有一致性。递归聚合是把一个类或接口作为自身成员的聚合关系，一个对象可以包含多个实现同一接口的对象，这些对象又可以包含更多实现同一接口的对象，于是可以看出，该种设计模式采用的 uml 结构是递归聚合接口。

1.3. FLYWEIGHT（享元）设计模式



这种设计模式采用的 uml 结构是组合接口。

在上图所示的享元设计模式的 uml 结构图中，存在 Factory 模式的对象构造工厂 FlyweightFactory，当客户需要一个对象的时候就会向该工厂发出请求对象那个的消息 getFlyweight() 消息，FlyweightFactory 拥有一个管理和存储对象的仓库，getFlyweight 就会遍历对象池中的对象。Flyweight 是一个接口或抽象类，它定义了享元对象的方法和属性。ConcreteFlyweight 是实现 Flyweight 接口的具体类，它包含一些独立的状态，但共享的状态是从 Flyweight 接口中继承的。UnsharedConcreteFlyweight 是非共享的 flyweight，可以多次实例化。上图中接口 Flyweight 包含 ConcreteFlyweight 和 UnsharedConcreteFlyweight 两个具体类，属于组合接口。

2. 给出两套实现方案的结构图（类图）和实现代码。

该题目所需要的加密模块，包含数据库操作类、用户类、测试类以及加密方法。其中，数据库操作类是目标抽象类，加密模块是该类的抽象方法。可以对相应模块的现有实现设计如下：用户类：

```

public class User {
    // 定义属性值命令 token 和邮箱 mail
    private String token;
}

```

```

    private String mail;
//    属性的 set 方法和 get 方法
    public void setToken(String token){
        this.token = token;
    }
    public String getToken(){
        return token;
    }
    public void setMail(String mail){
        this.mail = mail;
    }
    public String getMail(){
        return mail;
    }
//    定义 toString 方法增加输出结果可读性
    public String toString(){
        return "User{token=" + token + ",mail=" + mail + "}";
    }
}

```

数据库操作类：

```

public class DBUtil {
    public void save(User user){
        System.out.println("用户机密信息： " + user.toString() + "已存储到数据库");
    }
}

```

加密类：

```

public class Encrypt {
    public String encryption(String str){
        return "这是加密后的"+str; // 物理加密
    }
}

```

测试类：

```

public class Test {
    public static void main(String[] args){
        User user = new User();
        user.setToken("1234");
        user.setMail("666666@qq.com");

        DBUtil dbUtil = new DBUtil();
        dbUtil.save(user);
    }
}

```

运行测试类 Test.java，可以得到运行结果为：

```
用户机密信息: User{token=1234,mail=666666@qq.com}已存储到数据库  
进程已结束,退出代码0
```

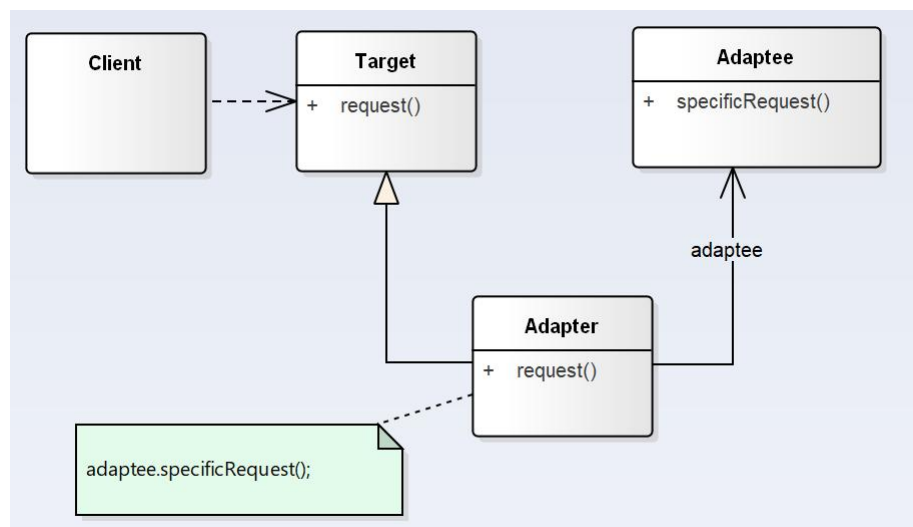
在上述现有实现中，存在数据库操作类和加密模块这两种不兼容的结构协同工作，引入适配器可以协调这种不兼容。适配器模式是指将一个类的接口转换成客户期望的另一个接口，使原本接口不兼容的类可以一起工作，从而实现对用户机密信息的加密。

适配器的角色定义包括：目标抽象类 **Target**，类定义客户所需接口，可以是一个抽象类或接口，也可以是具体类；适配器类 **Adapter**，可以调用另一个接口，作为转换器对适配器类 **Adaptee** 和目标抽象类 **Target** 进行适配，是适配器模式的核心；适配者类 **Adaptee** 是被适配的角色。

适配器模式可以分为对象适配器模式和类适配器模式。

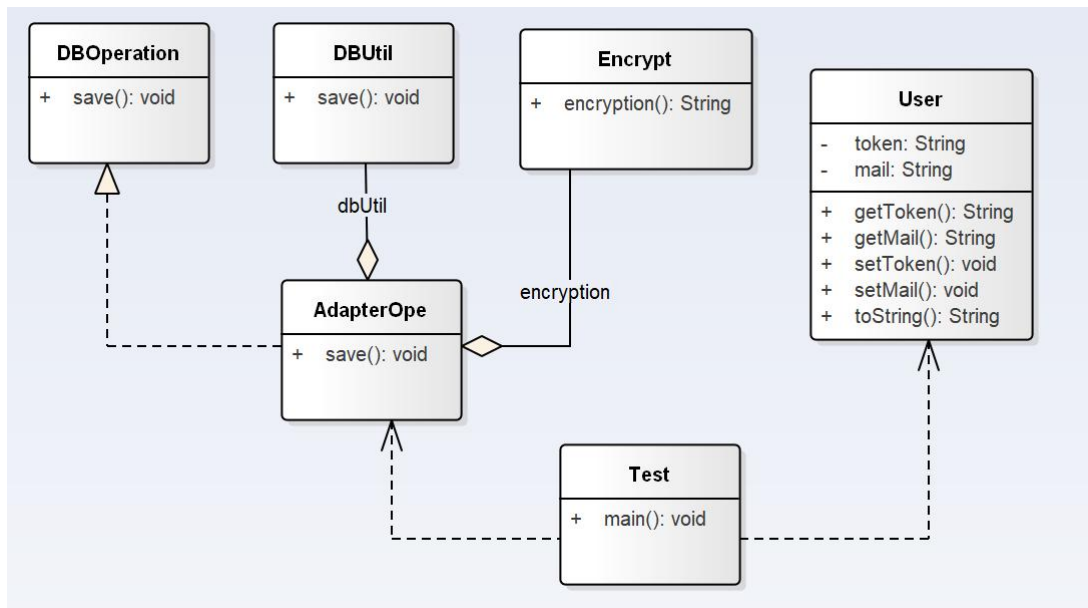
2.1. 对象适配器

通过资料查询，对象适配器模式结构图可以用下图表示：



在对象适配器中，适配器 **Adapter** 和适配者 **Adaptee** 之间是关联关系。

要对该加密模块进行重用实现，对象适配器解决方案下的类图为：



编程实现：

用户类、数据库类沿用现有实现中已定义的类，创建一个目的类接口 DBOperation：

```

public interface DBOperation {
    void save(User user);
}
  
```

定义适配器类：

```

// AdapterOpe 实现 DBOperation 接口
public class AdapterOpe implements DBOperation{
    // 维持一个对适配器对象的引用
    private final Encrypt encrypt;
    private final DBUtil dbUtil;

    public AdapterOpe(){
        encrypt = new Encrypt();
        dbUtil = new DBUtil();
    }

    // 在 save 方法中将 User 对象加密后传递给适配器类 DBUtil 的 save 方法，实现对 DBUtil
    类的适配
    public void save(User user){
        // 创建一个 User 对象，对测试类中设置的 token 和 mail 进行加密，获得加密后的 token
        和 mail
        User encryptUser = new User();
        encryptUser.setToken(encrypt.encryption(user.getToken()));
        encryptUser.setMail(encrypt.encryption(user.getMail()));
        dbUtil.save(encryptUser);
    }
}
  
```

重写测试类，创建适配器对象，调用其中的 save 方法，实现信息加密

```

public class Test {
    public static void main(String[] args){
        User user = new User();
        user.setToken("1234");
        user.setMail("666666@qq.com");
        //        DBUtil dbUtil = new DBUtil();
        //        dbUtil.save(user);

        //        重写
        DBOperation dbOperation = new AdapterOpe();
        dbOperation.save(user);
    }
}

```

运行测试类得到加密后的信息：

```

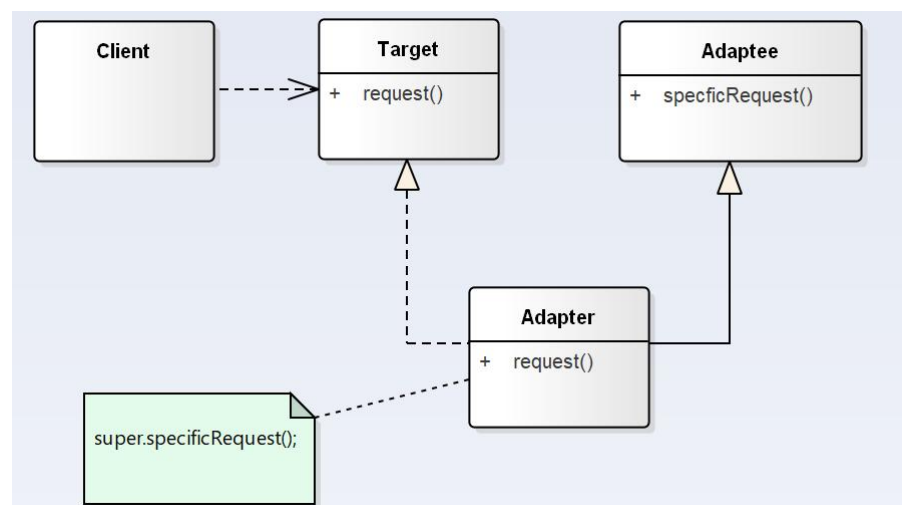
用户机密信息: User{token=这是加密后的1234,mail=这是加密后的666666@qq.com}已存储到数据库

进程已结束,退出代码0

```

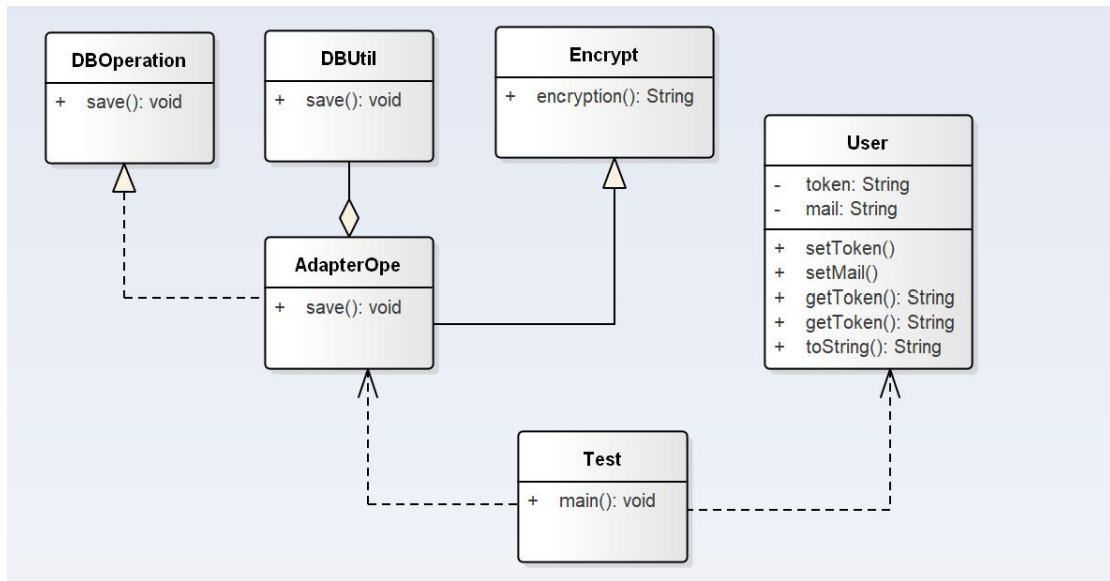
2.2. 类适配器

通过资料查询，类适配器模式结构图可以用下图表示：



在类适配器中，适配器 Adapter 和适配者 Adaptee 之间是继承（实现）关系。类适配器模式实现与对象适配器实现的区别在于适配器类直接继承加密类，并实现数据操作接口，加密类提供加密方法 encryption，数据操作类提供 save 方法。通过适配器实现 DBOperation 接口并适配 DBUtil 的功能，使客户端能够通过 DBOperation 接口对 DBUtil 类进行访问，不用直接调用 DBUtil 类。

对该加密模块进行重用，类适配器解决方案下的类图为：



编程实现：

定义类适配器对象：

```

public class AdapterOpe1 extends Encrypt implements DBOperation{
    private final DBUtil dbUtil; // 适配器 DBUtil 对象
    public AdapterOpe1(){
        this.dbUtil = new DBUtil();
    }
    public void save(User user){
        User encryptUser = new User();
        // 调用 Encrypt 中的加密方法，对 token 和 mail 进行加密
        encryptUser.setToken(super.encryption(user.getToken()));
        encryptUser.setMail(super.encryption(user.getMail()));
        // 转发调用适配器类 DBUtil 的 save 方法
        dbUtil.save(encryptUser);
    }
}
  
```

运行测试类得到加密后的信息：

```

用户机密信息: User{token=这是加密后的1234,mail=这是加密后的6666666@qq
.com}已存储到数据库

进程已结束,退出代码0
  
```

3. 给出类图和编程模拟实现代码。

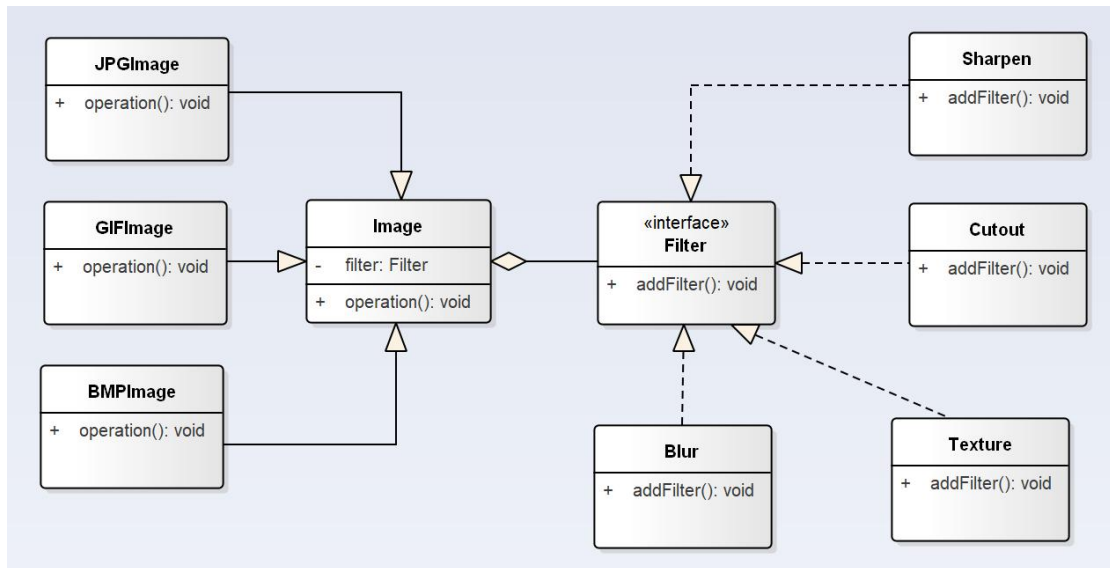
桥接模式是一种结构型设计模式，它将一个大类拆分成抽象和实现两个独立的层次结构，能够在两个层次结构中分别进行修改而不影响到另一个层次结构。

3.1. 解决思路

该问题中，对图像的处理包括图像格式处理和滤镜处理，可以将这两个部分看成两个独立的

层次结构，图像格式作为抽象层次结构，滤镜作为实现层次结构。定义一个抽象类 `Image`，包含一个 `Filter` 对象，表示该种图像格式可以使用的图像处理滤镜。然后根据不同的图像格式定义具体的 `Image` 类，能够实现自己的处理逻辑和滤镜使用方法 `operation`，当需要增加新的图像格式时，只需要定义一个具体类继承 `Image` 抽象类。定义 `Filter` 接口，包含一个 `add` 方法，可以添加新的 `Filter`。然后可以定义多个具体的实现类，对应不同的滤镜，当需要增加新的图像滤镜时，只需要实现 `Filter` 接口，并实现自己的处理逻辑即可。

3.2. 类图



3.3. 编程实现

滤镜接口类：

```
public interface Filter {

    public void addFilter();

}
```

滤镜具体实现类 Blur，其他格式同理：

```
public class Blur implements Filter{

    @Override
    public void addFilter() {
        System.out.println("add Blur Filter");
    }

}
```

Image 抽象类：

```
public abstract class Image {

    protected Filter filter;
    // 构造函数，传入 filter 对象初始化 Image 对象的 filter 属性
    public Image(Filter filter){
        super();
        this.filter = filter;
    }

}
```

```
// 抽象方法，不同格式定义自己的滤镜使用方法
public abstract void operation();
}
```

具体的图像类 JPGImage，其他格式同理：

```
public class JPGImage extends Image{
    // 调用父类的构造方法，将传入的 filter 对象传递给父类
    public JPGImage(Filter filter){
        super(filter);
    }
    public void operation(){
        filter.addFilter();
    }
}
```

客户端：

```
public class Client {
    public static void main(String[] args){
        // 创建一个 BMP 格式的图像并使用 Blur 滤镜
        Filter filter = new Blur();
        Image image = new BMPImage(filter);
        image.operation();
    }
}
```

3.4 运行程序，得到如下所示结果：

```
add Blur Filter

进程已结束,退出代码0
```

4. 给出类图和编程模拟实现代码。

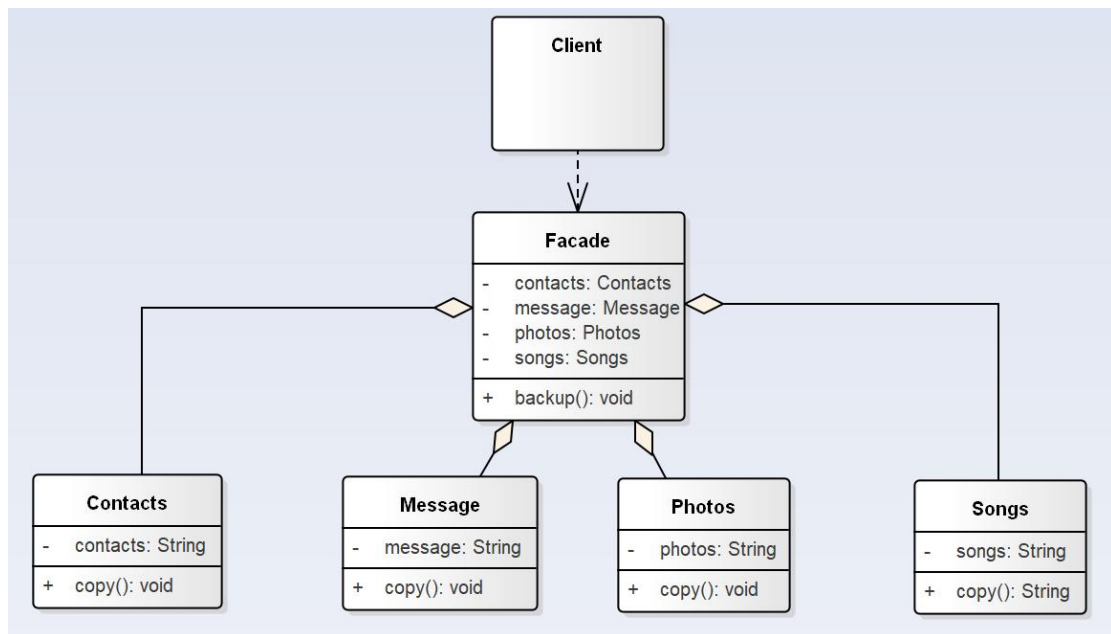
外观模式：又称门面模式，是一种结构型设计模式。其主要思路是为一组复杂的子系统提供一个更简单易用的接口，以降低其使用门槛和复杂度，同时也能减少系统之间的耦合度。

4.1. 解决思路

在该问题中，可以使用外观模式将备份功能的实现细节与多个已有的类进行解耦，提供一个简单易用的接口给用户，使得用户可以一次性备份所有需要备份的资料到移动存储介质中。首先定义一个备份管理类 Facade，封装备份功能的实现细节，与多个子系统类进行交互。定义子系统类可以包括通讯录类、短信类、照片类以及歌曲类。还需要定义一个客户端类，提供用户界面并能够调用备份管理类的接口实现备份。

在备份管理类中，需要定义备份功能接口，包括备份以上信息的多个备份方法。实现该功能接口，能够调用各个子系统的备份方法实现备份功能。在客户端类中，实例化备份管理类，并调用其备份功能接口来实现一键备份功能。

4.2. 类图



4.3. 编程模拟

子系统类通讯类（其他子系统类同理）：

```
public class Contacts {
    private String contacts;
    public Contacts(String contacts){
        this.contacts = contacts;
    }
    public void copy(){
        System.out.println(contacts + "已完成备份");
    }
}
```

备份管理类 Facade:

```
public class Facade {
    private Contacts contacts;
    private Messages messages;
    private Photos photos;
    private Songs songs;
    public Facade(){
        this.contacts = new Contacts("通讯录");
        this.messages = new Messages("短信");
        this.photos = new Photos("照片");
        this.songs = new Songs("音乐");
    }
    // 调用子系统的备份方法
    public void backup(){
        contacts.copy();
    }
}
```

```
        messages.copy();  
        photos.copy();  
        songs.copy();  
    }  
}
```

客户端:

```
public class Client {  
    public static void main(String[] args){  
        Facade facade = new Facade();  
        facade.backup();  
    }  
}
```

4.4. 运行程序，得到结果如下图所示

```
通讯录已完成备份  
短信已完成备份  
照片已完成备份  
音乐已完成备份  
  
进程已结束,退出代码0
```

四、实验总结与体会

1. 通过本次实验，学习了结构型设计模式，能够通过其 uml 结构图对其结构类型做出简单判断。
2. 通过实际应用，对适配器设计模式、桥接模式和外观模式有了一定的了解，能够进行应用，解决实际问题。
3. 结构型设计模式主要用于解决对象和类之间的组合问题，能够帮助我们构建更灵活的、可扩展的、易于维护的系统，提高软件的开发效率和质量。

五、成绩评定及评语

1.指导老师批阅意见:

2.成绩评定:

指导教师签字:毛斐巧
2023 年 月 日