Adapter Design Pattern

> The tutorial is modified from last version (2019)
>
> Designers:
>
> ZHU Yueming and PAN Chao
>
> *Only UML diagram is referenced*

Experimental Objective

Learn how to refactor source code to adapter design pattern and practice how to use multiple adapters.

Introduce of source code

1. Requirement introduction

- The `FileOperateInterfaceV1` (version 1) is an interface that includes three methods (`readStaffFile`, `writeStaffFile`, `printStaffFile`), and there is a concrete class of `FileOperateInterfaceV1` (`FileOperate`) implements those methods.

```
public interface FileOperateInterfaceV1 {
    List<StaffModel> readStaffFile();

    void printStaffFile(List<StaffModel> list);

    void writeStaffFile(List<StaffModel> list);
}
```

- How to use it:
  1) Enter 1 to read the staff file
  2) Enter 2 to print staff information
  3) Enter 3 to write file
  4) Enter 0 to end

```
Please select operation: 1.readFile 2.listFile 3. writeFile:
1
finish read
2
StaffModel{nam
StaffModel{nam
StaffModel{nam
StaffModel{name
StaffModel{name
StaffModel{nam
StaffModel{name
StaffModel{nam
StaffModel{nam
3
finish writing
0
```

2. Update the Original Interface

If we want to update the original interface to a new version (`FileOperateInterfaceV2`) by rename those methods named "`readStaffFile`", "`printStaffFile`" to "`readAllStaff`" and "`listAllStaff`" respectively, and adding another two methods including `WriteByName` and `WriteByRoom`, how can you accomplish this work?

```
public interface FileOperateInterfaceV2 {
    List<StaffModel> readAllStaff();

    void listAllStaff(List<StaffModel> list);

    void writeByName(List<StaffModel> list);

    void writeByRoom(List<StaffModel> list);
}
```

Solution 1: Design a concrete class of the new interface (`FileOperateInterfaceV2`).

Problems:

- It will overwrite the code we had already implemented.

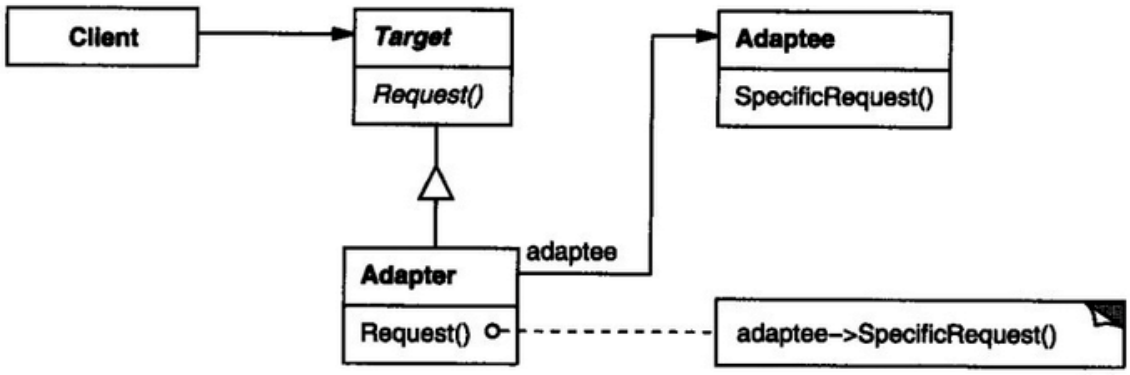
Solution 2: Modify the source code in concrete class of the original interface to match the new interface?

Problems:

- l It may cause other applications that rely on version 1 do not work properly.
- l Possibly, the programmer couldn't get the source code of version 1.

Adapter Design Pattern

We can design a new concrete class to implement the interface of Version 2, in which we can not only reuse the implement code of original concrete class (`FileOperate`), but also redesign those new features. The concrete class is adapter, and the original concrete class (`FileOperate`) is adaptee



Sample code:

```
public interface InterfaceVersion1 {
    public void function();
}

public interface InterfaceVersion2 {
    public void renamedFunction();
    public void newFunction();
}

public class Adapter implements InterfaceVersion2 {
    private InterfaceVersion1 adaptee;

    public Adapter(InterfaceVersion1 adaptee){
        this.adaptee=adaptee;
    }

    @Override
    public void renamedFunction(){
        adaptee.function();
    }

    @Override
    public void newFunction(){

    }
}
```

Task 1: Create a new package named **adapter**, copy the original code in it, and then create a class named **Adapter** to implement **FileOperateInterfaceV2**.

The Client part would be:

```
public static void main(String[] args) {
        List<StaffModel> list = new ArrayList<>();
        FileOperateInterfaceV1 adaptee = new FileOperate();
        FileOperateInterfaceV2 fileOperator = new Adapter(adaptee);
        Scanner input = new Scanner(System.in);
        System.out.println("Please select operation: 1.readFile 2.listFile
3.writeByName 4.writeByRoom :");
        int op = 0;
        do {
```

```
            try {
                op = input.nextInt();
                switch (op) {
                    case 1:
                        list = fileOperator.readAllStaff();
                        break;
                    case 2:
                        fileOperator.listAllStaff(list);
                        break;
                    case 3:
                        fileOperator.writeByName(list);
                        break;
                    case 4:
                        fileOperator.writeByRoom(list);
                        break;
                }
            } catch (InputMismatchException e) {
                System.out.println("Exception:" + e);
                input.nextLine();
            }
        } while (op != 0);

        input.close();
    }
```

Multiple Adaptees

The adapter can adapt to multiple `adaptees`, which means when adding a new target method, the adapter needs to reuse the defined code by passing a new instance of corresponding concrete class, and the class is the new `adaptee`.

Task 2: Create a new package named **mutiAdapter**, in which the interface of version 2 (**FileOperateInterfaceV2**) is not only required to adapt the methods in **FileOperateInterfaceV1**, but also required to adapt the methods in **ManageStaffInterface**.

Hints: In **Adapter**, you can define two `adaptees`, the one is an instance of `FileOperateInterfaceV1`, and the other is an instance of `ManageStaffInterface`.

`FileOperateInterfaceV2`:

```
public interface FileOperateInterfaceV2 {
    List<StaffModel> readAllStaff();

    void listAllStaff(List<StaffModel> list);

    void writeByName(List<StaffModel> list);

    void writeByRoom(List<StaffModel> list);

    void addNewStaff(List<StaffModel> list);

    void removeStaffByName(List<StaffModel> list);

}
```

`ManageStaffInterface`:

```
public interface ManageStaffInterface {
    void addingStaff(List<StaffModel> list);

    void removeStaff(List<StaffModel> list);
}
```

`ManageStaff` :It is the implement class of `ManageStaffInterface`

```
public class ManageStaff implements ManageStaffInterface {
    private Scanner input = new Scanner(System.in);

    private String inputNoEmptyLine(String tip) {
        System.out.println("Input " + tip + ":");
        String s;
        do {
            s = input.nextLine();
        } while (s.trim().length() == 0);

        return s;
    }

    @Override
    public void addingStaff(List<StaffModel> list) {
        System.out.println("Input a new Staff Info:");
        StaffModel s = new StaffModel(
                inputNoEmptyLine("name"),
                inputNoEmptyLine("title"),
                inputNoEmptyLine("email"),
                inputNoEmptyLine("room"),
                inputNoEmptyLine("link"));
        list.add(s);
        System.out.println("adding successfully");
    }

    @Override
    public void removeStaff(List<StaffModel> list) {
        System.out.println("Input staff name to remove:");
        String name = input.next();
        for (StaffModel e : list) {
            if (e.getName().equals(name)) {
                list.remove(e);
                break;
            }
        }
        System.out.println("success to remove " + name);
    }
}
```