组长：魏泽弘

成员：杨顺杰，高时玉，李纯洋，龚云基

违反

一、违反开闭原则和单一职责原则

1.代码

（1）修改前代码：

Agent类

```
package agent;

import message.Message;

public class Agent {
    private Message message;

    public void Buy(){}

    public void Sell(){}

    public void Purchase(){}

    private Agent(){}
    private static class SingletonInstance{//静态内部类
        private static final Agent INSTANCE = new Agent();
    }

    public static Agent getInstance(){
        return SingletonInstance.INSTANCE;
    }
    public String call(){
        return "代理商";
    }
    public Message getMessage() {
        return message;
    }
    public void setMessage(Message message) {
        this.message = message;
    }


}

public class AngetStuff{

    private  Agent ag =new Agent();

    public void Agent(){
        switch (ag.opeartion){
            case "Buy":
                ag.Buy();
                break;
            case "Sell":
                ag.Sell();
                break;
            case "Purchase":
                ag.Purchase();
                break;
        }
    }
}
```

（2）修改后的代码：

Agent类

```
package agent;

import message.Message;

public class Agent {
    private Message message;

    private Agent() {
    }

    public static Agent getInstance() {
        return Agent.SingletonInstance.INSTANCE;
    }

    public String call() {
        return "代理商";
    }

    public Message getMessage() {
        return this.message;
    }

    public void setMessage(Message message) {
        this.message = message;
    }

    private static class SingletonInstance {
        private static final Agent INSTANCE = new Agent();

        private SingletonInstance() {
        }
    }
}
```

AgentProcess接口：

```
package agent;

public interface AgentProcess {
    void Process();
}
```

BuyProcess类：

```
package agent;

public class BuyProcess implements AgentProcess {
    public BuyProcess() {
    }

    public void Process() {
        System.out.println("Process Buy");
    }
}
```

SellProcess类：

```
package agent;

public class SellProcess implements AgentProcess {
    public SellProcess() {
    }

    public void Process() {
        System.out.println("Process Buy");
    }
}
```

PurchaseProcess类：

```
package agent;

public class PurchaseProcess implements AgentProcess {
    public PurchaseProcess() {
    }

    public void Process() {
        System.out.println("Process Purchase");
    }
}
```

2.违反原因

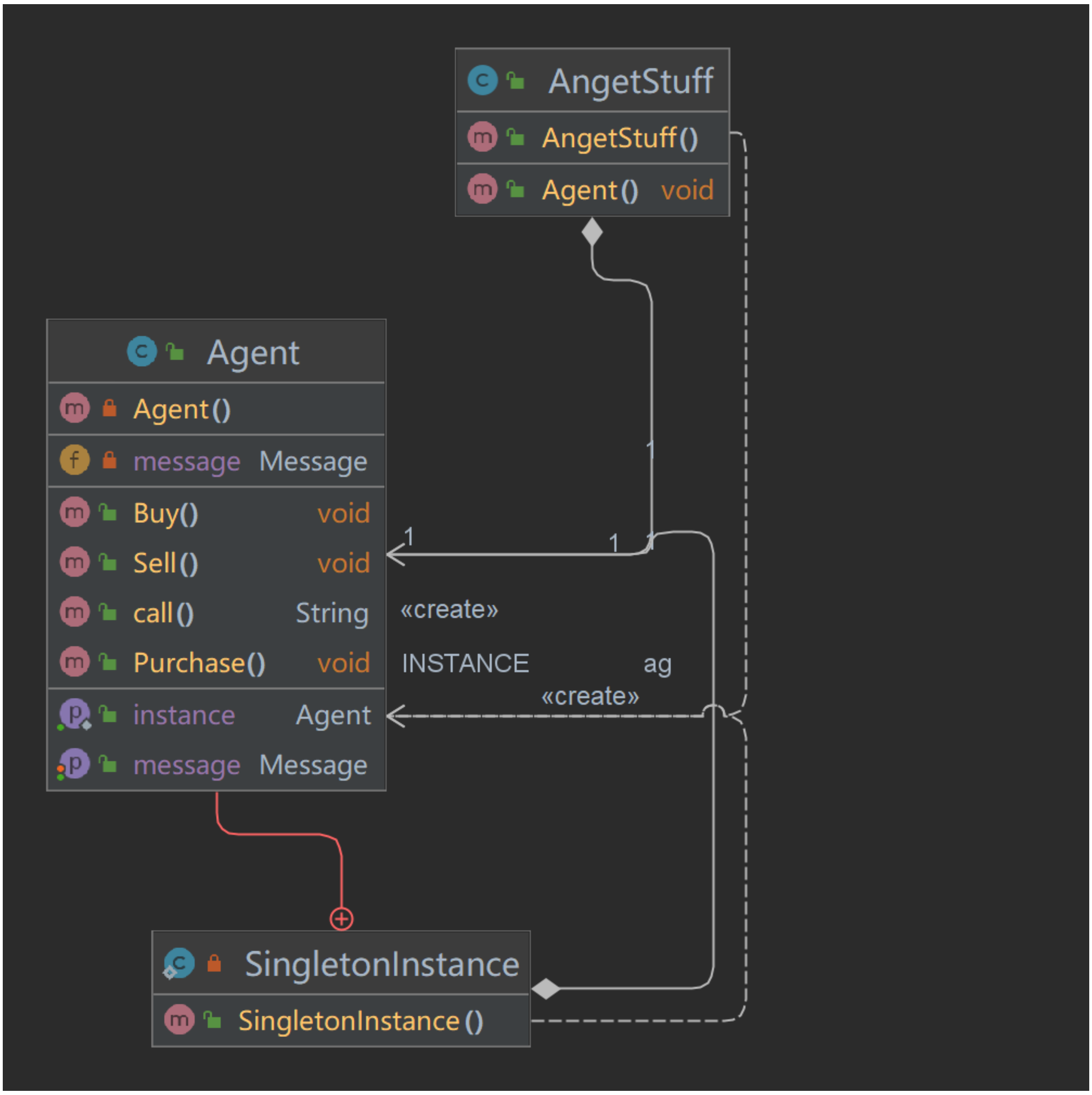
此处违反开闭原则和单一职责原则，目前代码中就只有购买，售卖和进货三个功能，将来如果业务增加了，比如转让，签收功能等，就必须要修改Agent类。我们分析上述设计就能发现不能把所有功能封装在一个类里面，不仅违反单一职责原则，而且当有新的需求发生，必须修改现有代码则违反了开放封闭原则。

3.修改方法

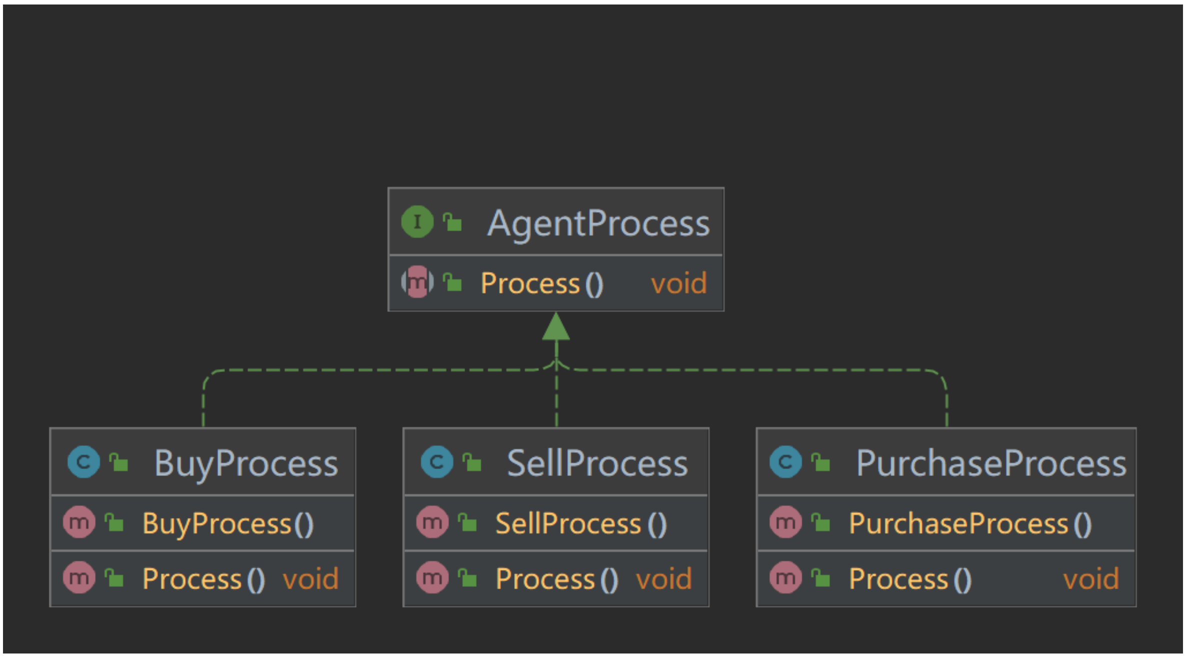
为了解决这一问题，我们将业务功能抽象为接口，当其依赖于固定的抽象时，对修改就是封闭的，而通过继承和多态继承，从抽象体中扩展出新的实现，就是对扩展的开放。

4.类图

（1）修改前类图：



（2）修改后类图：



二、违反了最少知识原则

1.代码

（1）修改代码前：

```
public class AgentFrame extends JFrame implements ActionListener {
    private static final long serialVersionUID = 1L;
    public JLabel label = new JLabel("1代理商选择进货              2  可乐公司供货
             3  顾客购买商品");
    public JButton b1 = new JButton("确认");
    public JButton b2 = new JButton("发货&提醒");
    public JButton b3 = new JButton("确定");
    public JPanel panel = new JPanel();
    public JTextArea tf1 = new JTextArea();
    public JCheckBox cb1 = new JCheckBox("可口可乐");
    public JCheckBox cb2 = new JCheckBox("百事可乐");
    public JRadioButton rb1 = new JRadioButton("门店自取");
    public JRadioButton rb2 = new JRadioButton("快递送达");
    public JScrollPane sp = new JScrollPane();
    ...
    }
```

（2）修改代码后：

```
public class AgentFrame extends JFrame implements ActionListener {
    private static final long serialVersionUID = 1L;
    private JLabel label = new JLabel("1代理商选择进货              2  可乐公司供货
             3  顾客购买商品");
    private JButton b1 = new JButton("确认");
    private JButton b2 = new JButton("发货&提醒");
    private JButton b3 = new JButton("确定");
    private JPanel panel = new JPanel();
    private JTextArea tf1 = new JTextArea();
    private JCheckBox cb1 = new JCheckBox("可口可乐");
    private JCheckBox cb2 = new JCheckBox("百事可乐");
    private JRadioButton rb1 = new JRadioButton("门店自取");
    private JRadioButton rb2 = new JRadioButton("快递送达");
    private JScrollPane sp = new JScrollPane();
    ...
    }
```

2.违反原因

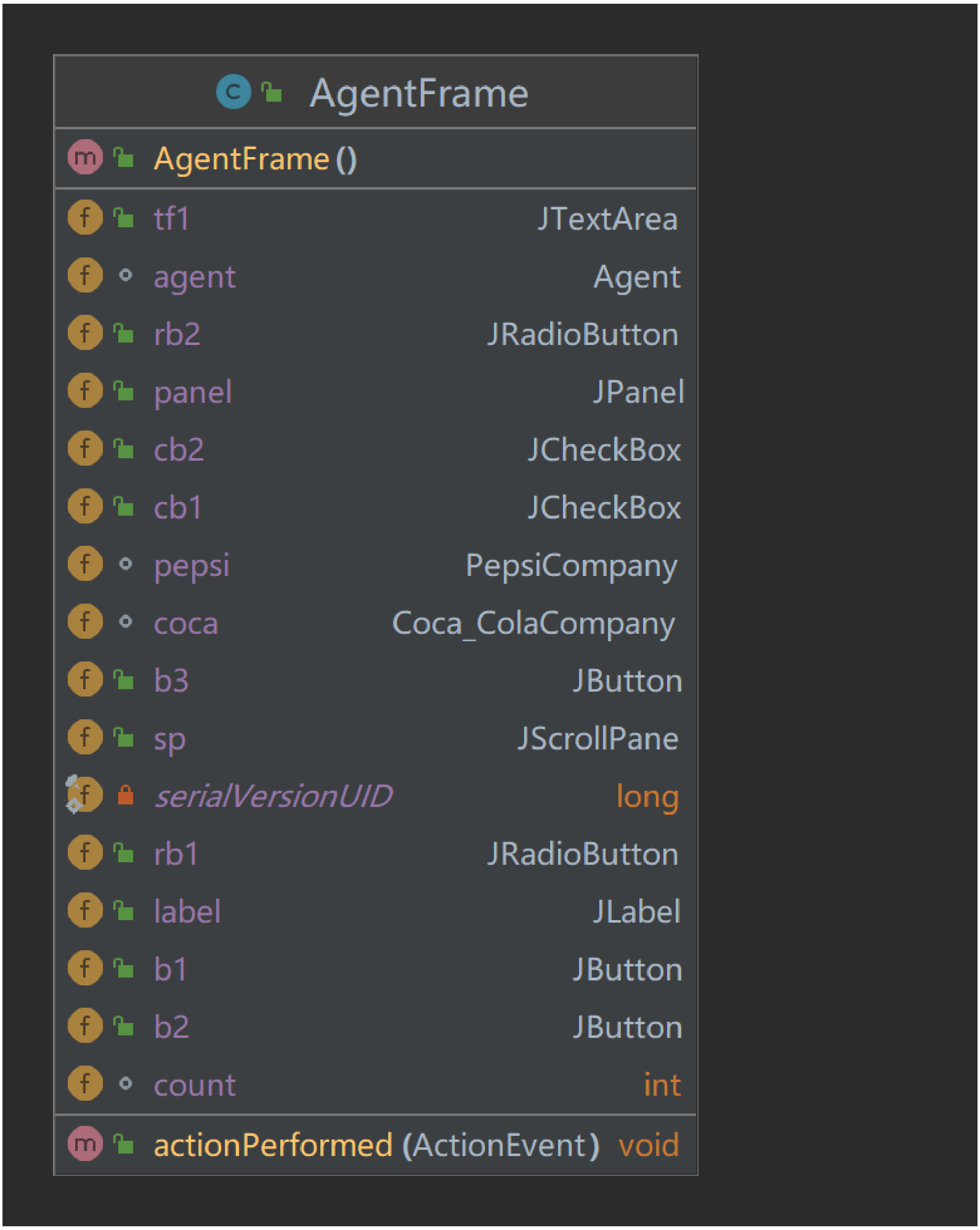
此处违反了最少知识原则。JLabel, JButton, JPanel, JTextArea, JCheckBox, JRadioButton, JScrollPane对象都是public，访问权限设置不合理，耦合度高，暴露了属性成员，没有做到对其他对象有尽可能少的了解，违反了最少知识原则。

3.修改方法

将public修改为private，使一个对象对其他对象保持尽可能最少的了解，降低对象之间的耦合度，没有暴露属性成员，符合最少知识原则。

4.类图

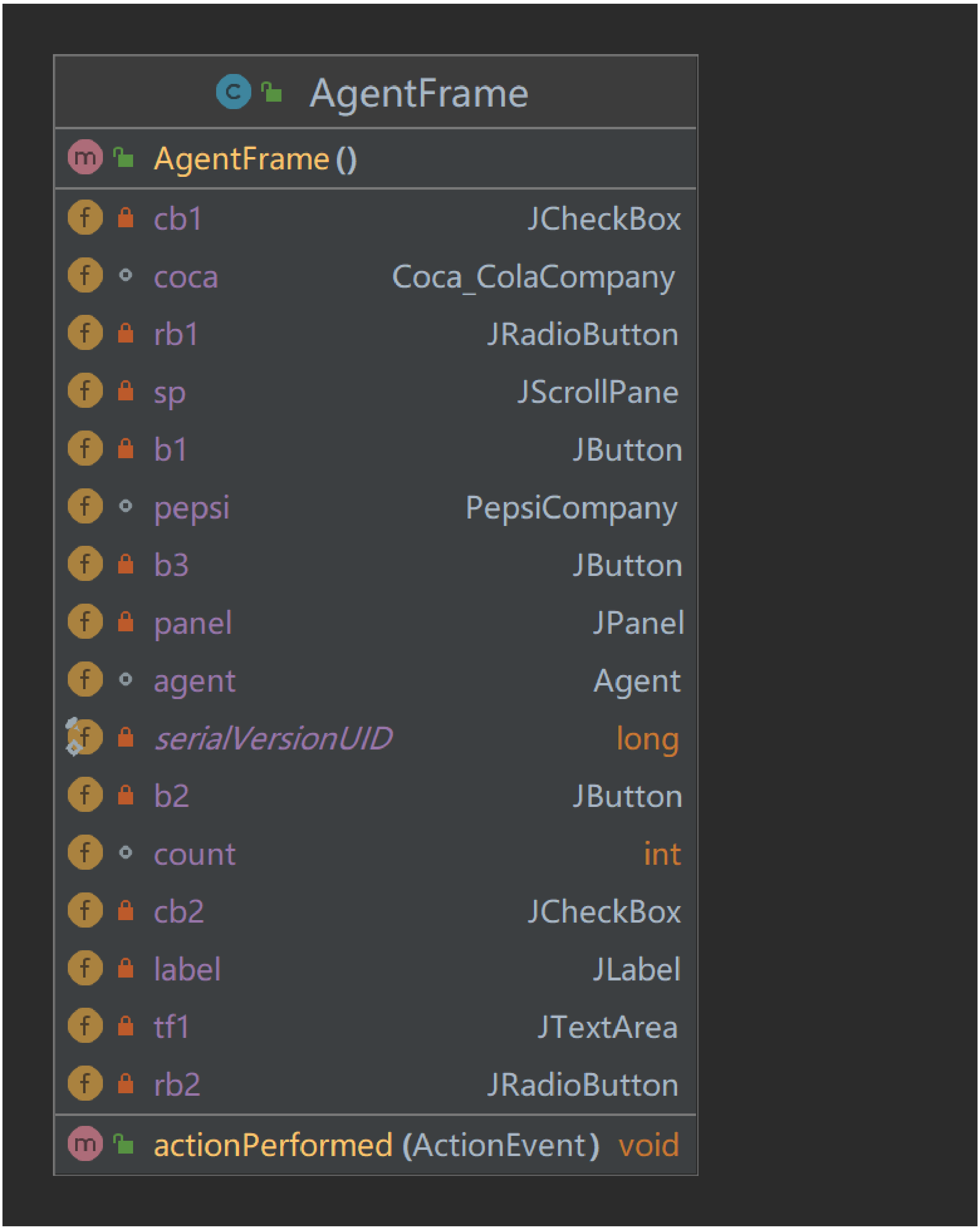
（1）修改前类图：



AgentFrame

AgentFrame ()	
tf1	JTextArea
agent	Agent
rb2	JRadioButton
panel	JPanel
cb2	JCheckBox
cb1	JCheckBox
pepsi	PepsiCompany
coca	Coca_ColaCompany
b3	JButton
sp	JScrollPane
serialVersionUID	long
rb1	JRadioButton
label	JLabel
b1	JButton
b2	JButton
count	int
actionPerformed (ActionEvent)	void

（2）修改后类图：



AgentFrame

AgentFrame ()	
cb1	JCheckBox
coca	Coca_ColaCompany
rb1	JRadioButton
sp	JScrollPane
b1	JButton
pepsi	PepsiCompany
b3	JButton
panel	JPanel
agent	Agent
serialVersionUID	long
b2	JButton
count	int
cb2	JCheckBox
label	JLabel
tf1	JTextArea
rb2	JRadioButton
actionPerformed (ActionEvent)	void

三、违反依赖倒置原则

1.代码

（1）修改前代码：

OrderBuilder类

```
package customer;

public class OrderBuidler {
    private String name;
    private String goods;
    private String address;


    public OrderBuidler(String name, String goods, String address) {
        this.name = name;
        this.goods = goods;
        this.address = address;
    }

    public Order create() {
        return new Order(this);
    }

    public String getName() {
        return this.name;
    }

    public String getGoods() {
        return this.goods;
    }

    public String getAddress() {
        return this.address;
    }
}
```

Order类

```
package customer;

public class Order {
    private String name;
    private String goods;
    private String address;

    public Order(OrderBuidler builder) {
        this.name = builder.getName();
        this.goods = builder.getGoods();
        this.address = builder.getAddress();
    }

    public void createOrder() {
        System.out.println("----订单已生成----");
        if (this.name != null) {
            System.out.println("顾客姓名：" + this.name);
        }

        System.out.println("购买的商品：" + this.goods);
        System.out.println("地址" + this.address);
    }

    public String getName() {
        return this.name;
    }

    public String getGoods() {
        return this.goods;
    }
    public String getAddress() {
        return this.address;
    }
}
```

（2）修改后代码：

IOrder

```
package customer;

public interface IOrder {
    String getName();

    String getGoods();

    String getAddress();
}
```

Order

```
package customer;

public class Order {
    private customer.IOrder iOrder;
    private java.lang.String name;
    private java.lang.String goods;
    private java.lang.String address;

    public Order(customer.IOrder iOrder) { /* compiled code */ }

    public void createOrder() { /* compiled code */ }

    public java.lang.String getName() { /* compiled code */ }

    public java.lang.String getGoods() { /* compiled code */ }

    public java.lang.String getAddress() { /* compiled code */ }
}
```

OrderBuildler

```
package customer;

public class OrderBuidler implements customer.IOrder {
    private java.lang.String name;
    private java.lang.String goods;
    private java.lang.String address;

    public OrderBuidler(java.lang.String name, java.lang.String goods,
java.lang.String address) { /* compiled code */ }

    public customer.Order create() { /* compiled code */ }

    public java.lang.String getName() { /* compiled code */ }
    public java.lang.String getGoods() { /* compiled code */ }

    public java.lang.String getAddress() { /* compiled code */ }
}
```

2.违反原因

此处违反了依赖倒置原则。OrderBuilder类需要使用Order方法，Order类也需要新建OrderBuilder对象，二者存在循环依赖，都发生了直接依赖，没有依赖于抽象，故违反了依赖倒置原则。

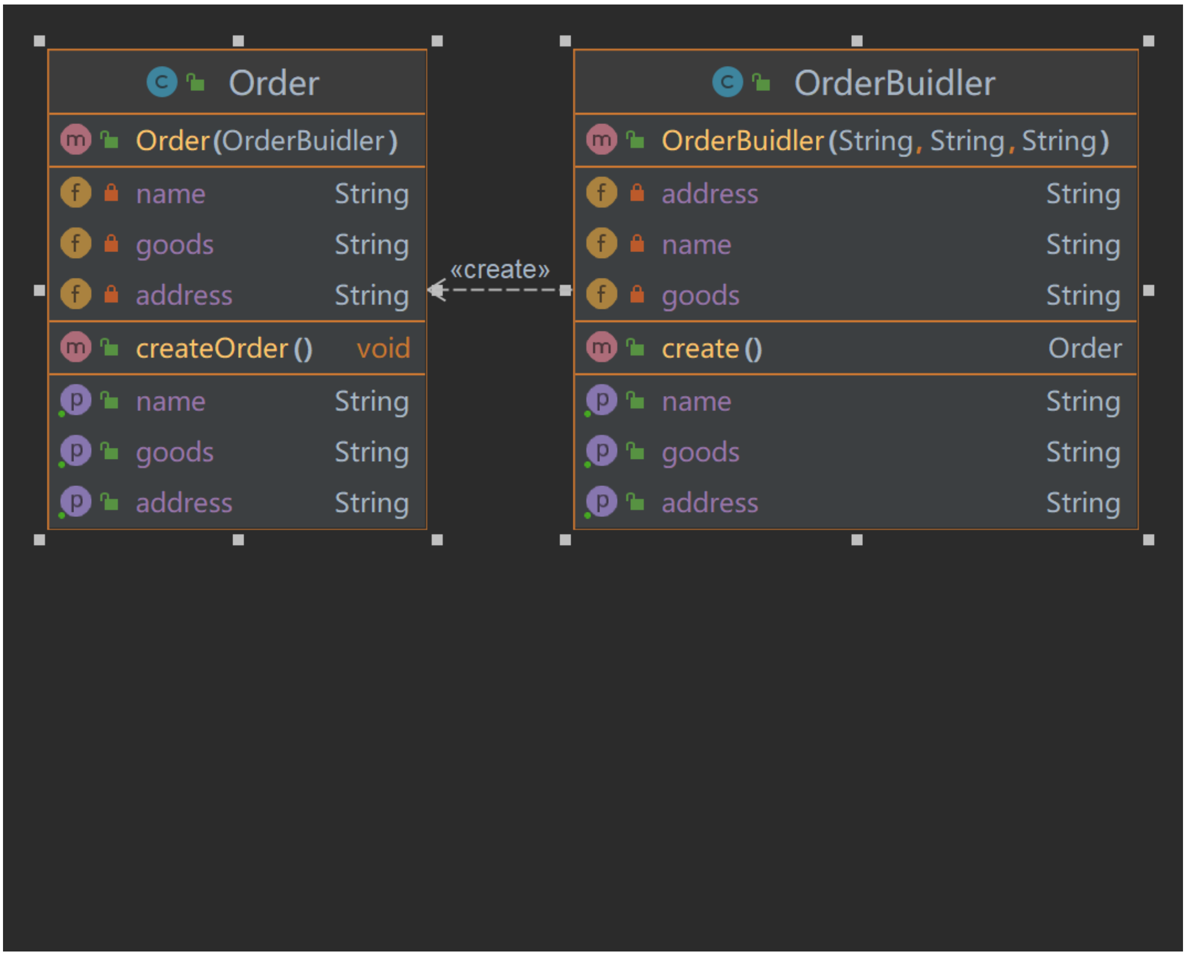
3.修改方法

运用回调方法，抽象IOrder接口，封装getName(), getGoods(), getAddress()方法。该接口与Order类处于同一层次，OrderBuilder类实现该接口，使原有Order类对OrderBuilder类的依赖转变为Order类对IOrder的依赖。

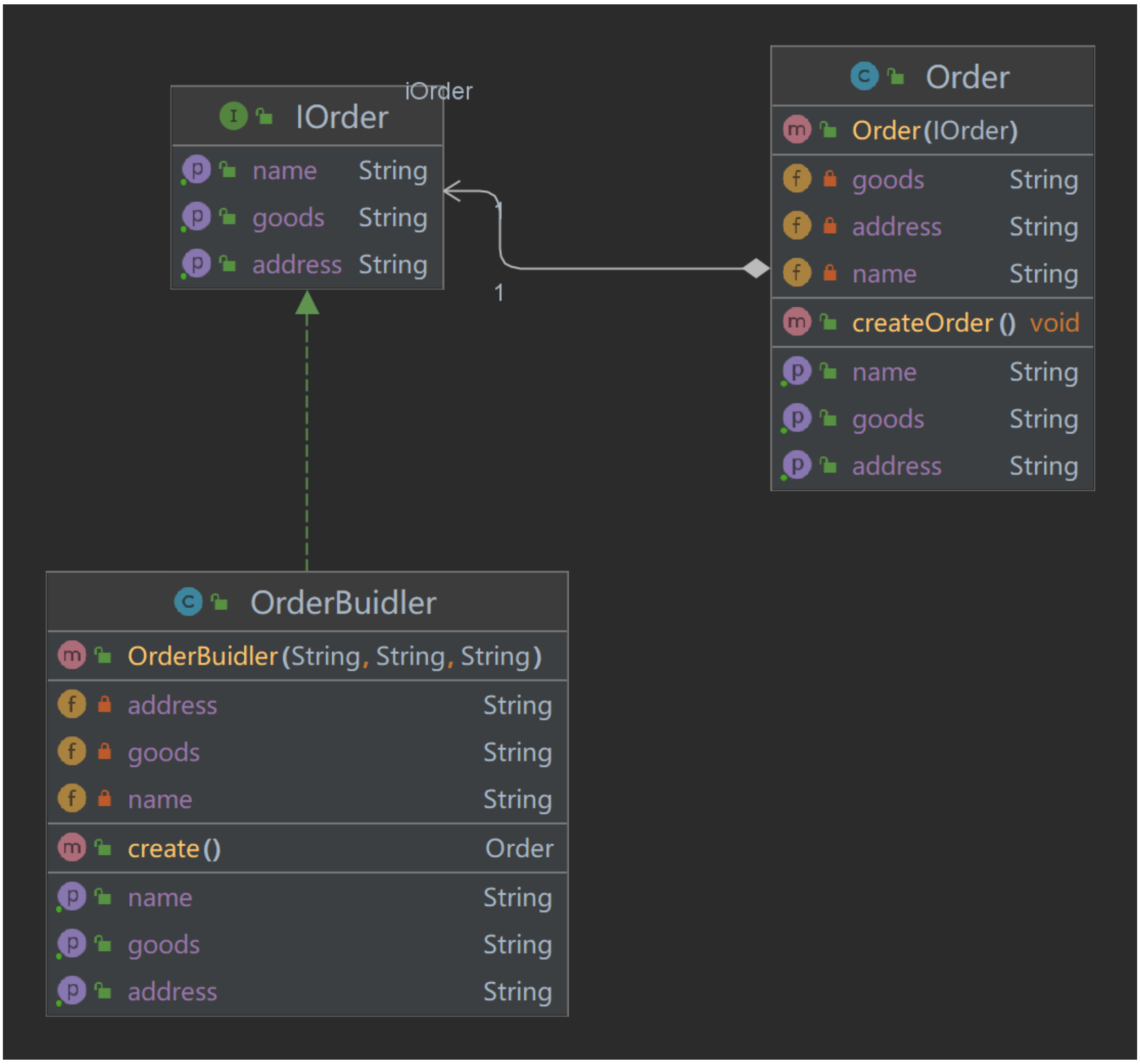
修改后上层模块不依赖于底层模块，而依赖于抽象接口，降低了耦合，符合依赖倒置原则。

4.类图

（1）修改前类图：



（2）修改后类图：



四、违反ISP接口分离原则

1.代码

（1）修改前代码：

DeliverGoods接口

```
package agent;

public interface DeliverGoods {
    String sendCoke(String var1);

    String selectMethod();
}
```

DeliverMethod类

```
package agent;

public class DeliverMethod implements DeliverGoods {
    boolean flag;

    public DeliverMethod(boolean flag) {
        this.flag = flag;
    }

    public String selectMethod() {
        if (this.flag) {
            DeliverGoods express = new Express(this.flag);
            return express.sendCoke("快递公司代理");
        } else {
            DeliverGoods tradition = new Tradition(this.flag);
            return tradition.sendCoke("门店自取");
        }
    }

    public String sendCoke(String method) {
        return null;
    }
}
```

Express类

```
package agent;

public class Express implements DeliverGoods {
    boolean flag;
    String m = null;

    public Express(boolean flag) {
        this.flag = flag;
    }

    public String sendCoke(String method) {
        if (this.flag) {
            this.m = method;
        }

        return "您的订单已生成，已经通过" + this.m + "方式为您发货，请注意查收！";
    }

    public String selectMethod() {
        return null;
    }
}
```

Tradition类

```
package agent;

public class Tradition implements DeliverGoods {
    boolean flag;
    String m = null;

    public Tradition(boolean flag) {
        this.flag = flag;
    }

    public String sendCoke(String method) {
        if (!this.flag) {
            this.m = method;
        }

        return "您的订单已生成，请持订单到线下各大" + this.m + "，谢谢！";
    }

    public String selectMethod() {
        return null;
    }
}
```

（2）修改后代码：

DeliverGoods接口

```
package agent;

public interface DeliverGoods {
    String sendCoke(String var1);
}
```

DeliverMethod类

```
package agent;

public class DeliverMethod {
    boolean flag;

    public DeliverMethod(boolean flag) {
        this.flag = flag;
    }

    public String selectMethod() {
        if (this.flag) {
            DeliverGoods express = new Express(this.flag);
            return express.sendCoke("快递公司代理");
        } else {
            DeliverGoods tradition = new Tradition(this.flag);
            return tradition.sendCoke("门店自取");
        }
    }
}
```

Express类

```
package agent;

public class Express implements DeliverGoods {
    boolean flag;
    String m = null;

    public Express(boolean flag) {
        this.flag = flag;
    }
    public String sendCoke(String method) {
        if (this.flag) {
            this.m = method;
        }

        return "您的订单已生成，已经通过" + this.m + "方式为您发货，请注意查收！";
    }
}
```

Tradition类

```
package agent;

public class Tradition implements DeliverGoods {
    boolean flag;
    String m = null;

    public Tradition(boolean flag) {
        this.flag = flag;
    }

    public String sendCoke(String method) {
        if (!this.flag) {
            this.m = method;
        }

        return "您的订单已生成，请持订单到线下各大" + this.m + "，谢谢！";
    }
}
```

2.违反原因

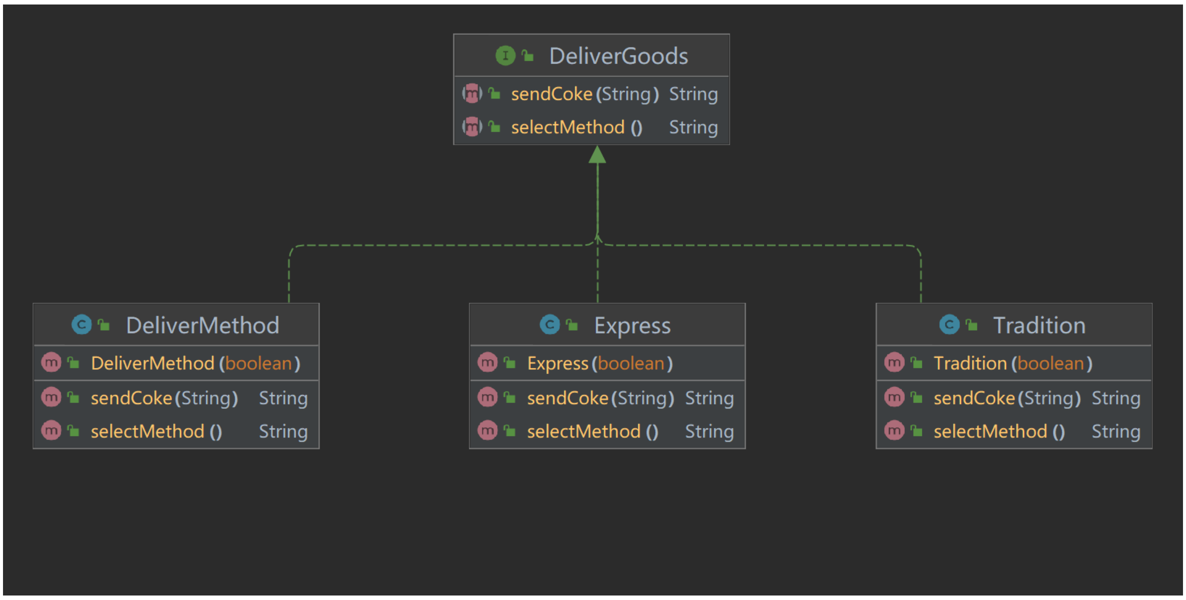
因为接口DeliverGoods中包含了“判断发货方式”和“发货”两个职责，因此实现该接口的类也必须实现与本类无关的方法。在原先的代码中，实现该接口的三个类DeliverMethod，Express，和Tradition中均有空方法体存在。

3.修改方法

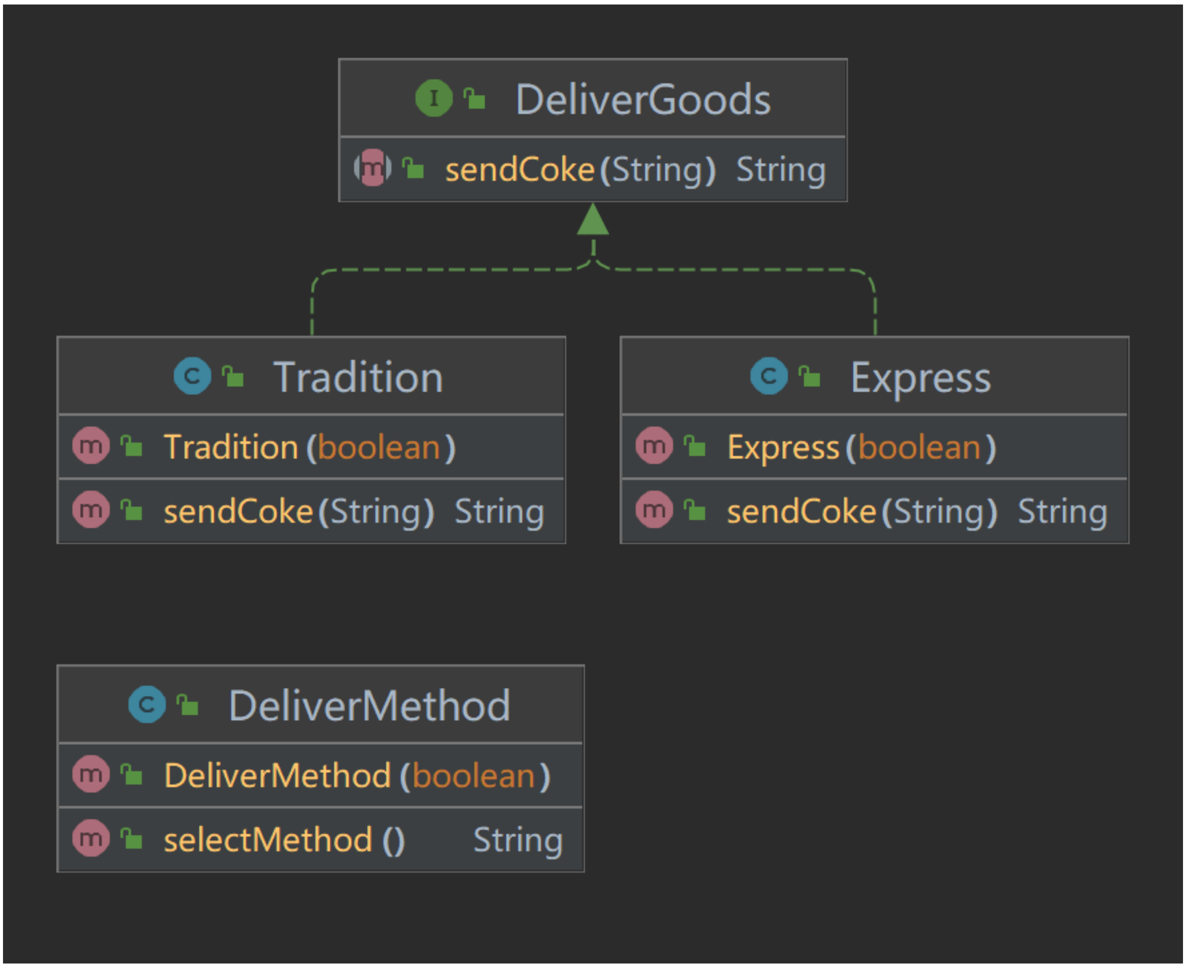
将接口DeliverGoods的职责进行拆分，使DeliverMethod不再实现DeliverGoods接口

4.类图

（1）修改前类图：



（2）修改后类图：



符合

一、符合开闭原则

1.代码

DeliverMethod类

```
package agent;
```

```
public class DeliverMethod {
    boolean flag;

    public DeliverMethod(boolean flag) {
        this.flag = flag;
    }

    public String selectMethod() {
        if (this.flag) {
            DeliverGoods express = new Express(this.flag);
            return express.sendCoke("快递公司代理");
        } else {
            DeliverGoods tradition = new Tradition(this.flag);
            return tradition.sendCoke("门店自取");
        }
    }
}
```

DeliverGoods接口

```
package agent;

public interface DeliverGoods {
    String sendCoke(String var1);

    String selectMethod();
}
```

Express类

```
package agent;

public class Express implements DeliverGoods {
    boolean flag;
    String m = null;

    public Express(boolean flag) {
        this.flag = flag;
    }

    public String sendCoke(String method) {
        if (this.flag) {
            this.m = method;
        }

        return "您的订单已生成，已经通过" + this.m + "方式为您发货，请注意查收！";
    }

    public String selectMethod() {
        return null;
    }
}
```

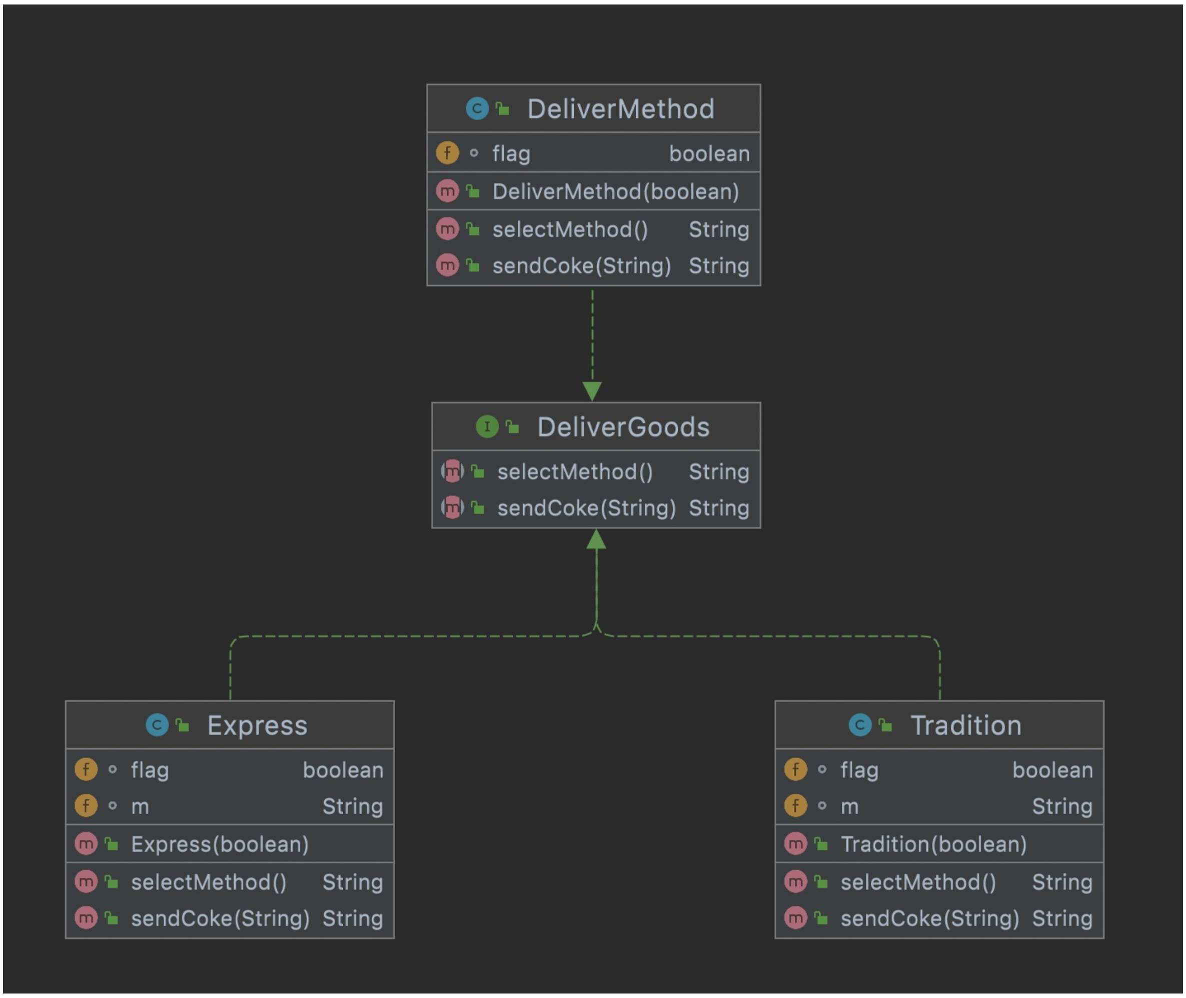
Tradition类

```java
package agent;

public class Tradition implements DeliverGoods {
    boolean flag;
    String m = null;

    public Tradition(boolean flag) {
        this.flag = flag;
    }

    public String sendCoke(String method) {
        if (!this.flag) {
            this.m = method;
        }

        return "您的订单已生成，请持订单到线下各大" + this.m + "，谢谢！";
    }

    public String selectMethod() {
        return null;
    }
}
```

2.符合原因

DeliverMethod针对抽象货物运输类DeliverGoods进行编程，DeliverGoods类是接口，不需要对DeliverMethod类进行修改，只需要在客户端来决定使用哪种运输方式。DeliverGoods是系统中相对稳定的抽象层，将不同的实现行为移至具体的实现层中完成。如果需要修改系统的行为，无须对抽象层进行任何改动，只需要增加新的具体类来实现新的业务功能即可，实现在不修改已有代码的基础上扩展系统的功能，达到开闭原则的要求。

3.类图



二、符合依赖倒置原则

1.代码

DeliverMethod类

```
package agent;

public class DeliverMethod {
    boolean flag;

    public DeliverMethod(boolean flag) {
        this.flag = flag;
    }

    public String selectMethod() {
        if (this.flag) {
            DeliverGoods express = new Express(this.flag);
            return express.sendCoke("快递公司代理");
        } else {
            DeliverGoods tradition = new Tradition(this.flag);
            return tradition.sendCoke("门店自取");
        }
    }
}
```

DeliverGoods接口

```
package agent;

public interface DeliverGoods {
    String sendCoke(String var1);

    String selectMethod();
}
```

Express类

```
package agent;

public class Express implements DeliverGoods {
    boolean flag;
    String m = null;

    public Express(boolean flag) {
        this.flag = flag;
    }

    public String sendCoke(String method) {
        if (this.flag) {
            this.m = method;
        }

        return "您的订单已生成，已经通过" + this.m + "方式为您发货，请注意查收！";
    }

    public String selectMethod() {
        return null;
    }
}
```

Tradition类

```
package agent;

public class Tradition implements DeliverGoods {
    boolean flag;
    String m = null;

    public Tradition(boolean flag) {
        this.flag = flag;
    }

    public String sendCoke(String method) {
        if (!this.flag) {
            this.m = method;
        }

        return "您的订单已生成，请持订单到线下各大" + this.m + "，谢谢！";
    }

    public String selectMethod() {
        return null;
    }
}
```

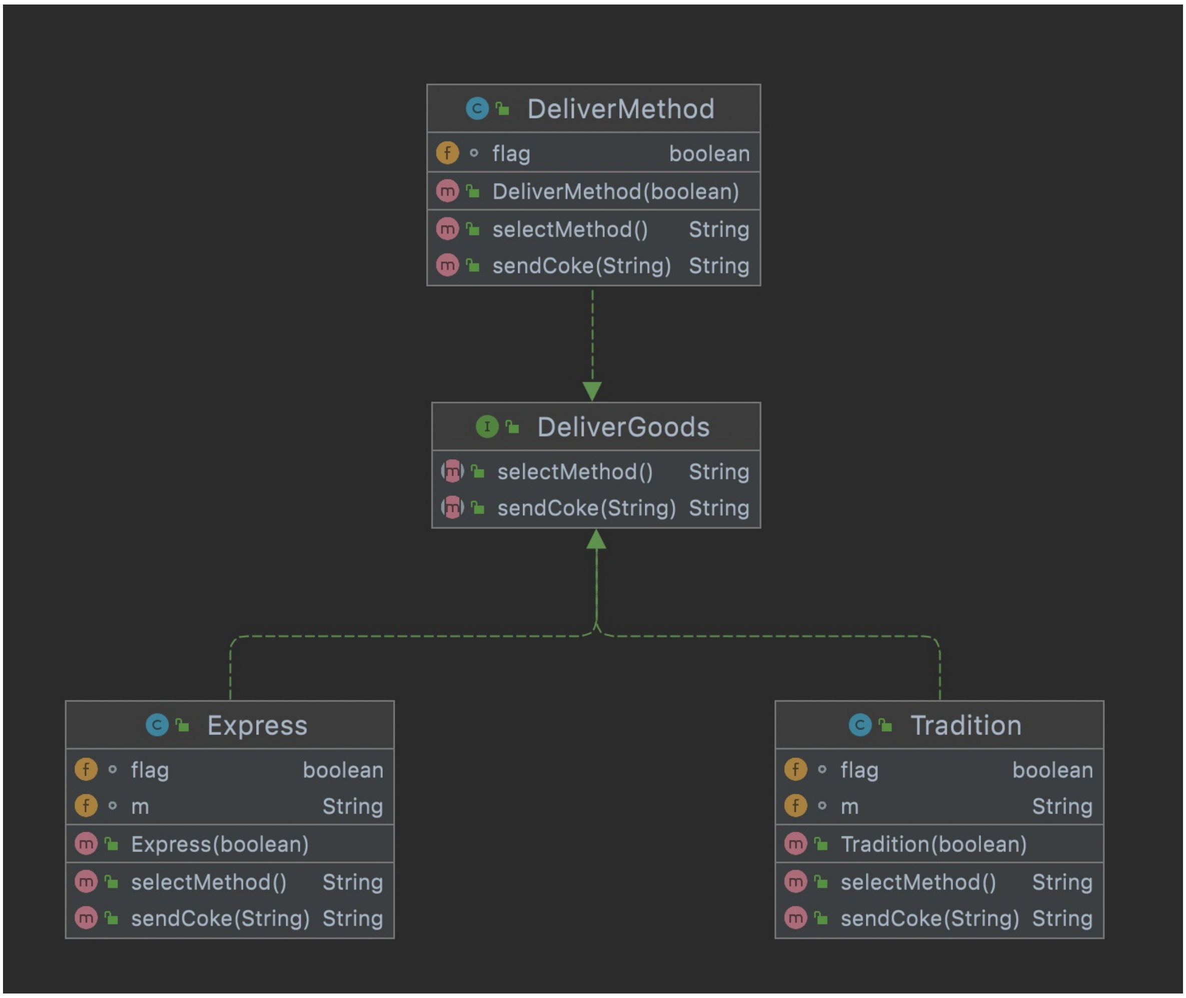
2.符合原因

在程序代码中传递参数时或在关联关系中，代码引用了层次高的抽象层类DeliverGoods，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，而没有用DeliverGoods来做这些事情。这样一来，如果系统行为发生变化，只需要对抽象层进行扩展，并修改配置文件，而无须修改原有系统的源代码，在不修改的情况下来扩展系统的功能，也满足了开闭原则的要求。

Coke类是父类，表示可乐；

Coca_Cola表示可口可乐，Pepsi表示百事可乐，这两个类是两个不同的可乐品牌。

3.类图



三、符合里氏代换原则

1.代码

Coca_Cola类

```java
package coke;

public class Coca_Cola implements Coke {
    public Coca_Cola() {
    }

    public String produce() {
        String str = "可口可乐正在生产...";
        return str;
    }
}
```

Coke接口

```java
package coke;

public interface Coke {
    String produce();
}
```

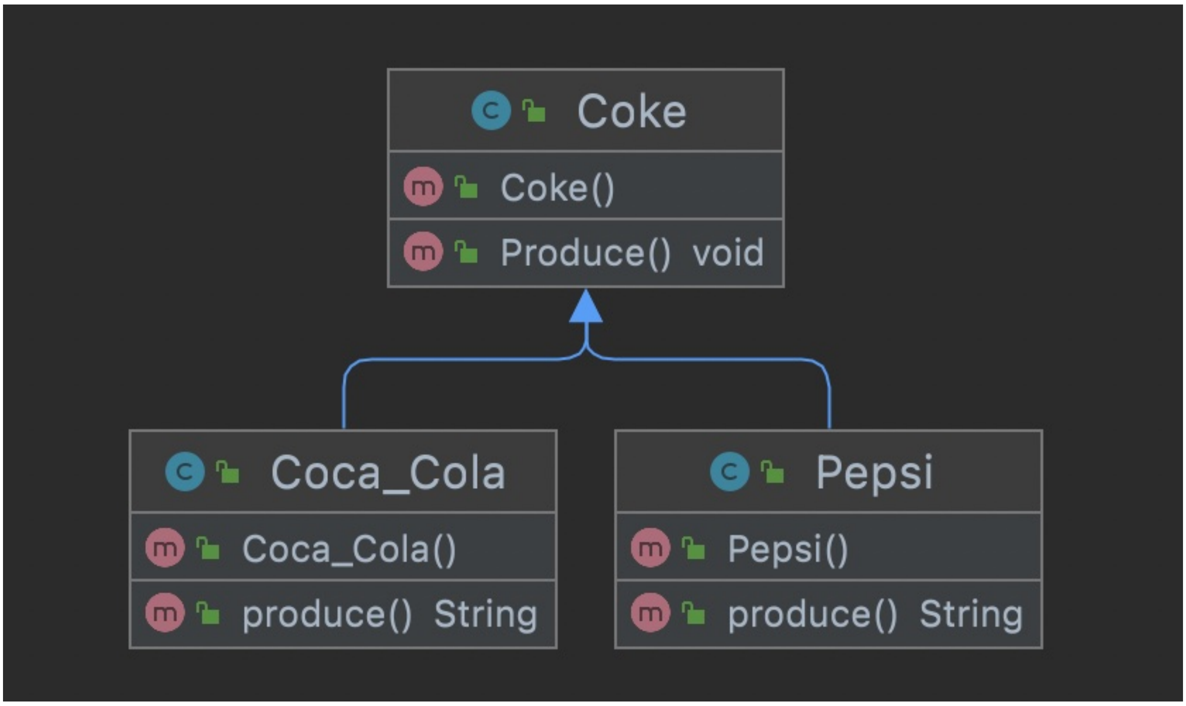
Pepsi类

```java
package coke;

public class Pepsi implements Coke {
    public Pepsi() {
    }

    public String produce() {
        String str = "百事可乐正在生产...";
        return str;
    }
}
```

2.符合原因

Coca_Cola类和Pepsi类继承了Coke类的Produce()方法，Coke是基类，定义了一些基本的方法，在子类使用时可以降低代码的重复率，在编程是可以先针对基类编程，在真正使用到子类是在确定具体子类。

3.类图


Coke
Coke()
Produce() void
Coca_Cola
Coca_Cola()
produce() String
Pepsi
Pepsi()
produce() String