LECTURE 11

Visitor, Observer and Interpreter

Visitor(訪問者模式)

Intent

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

Motivation

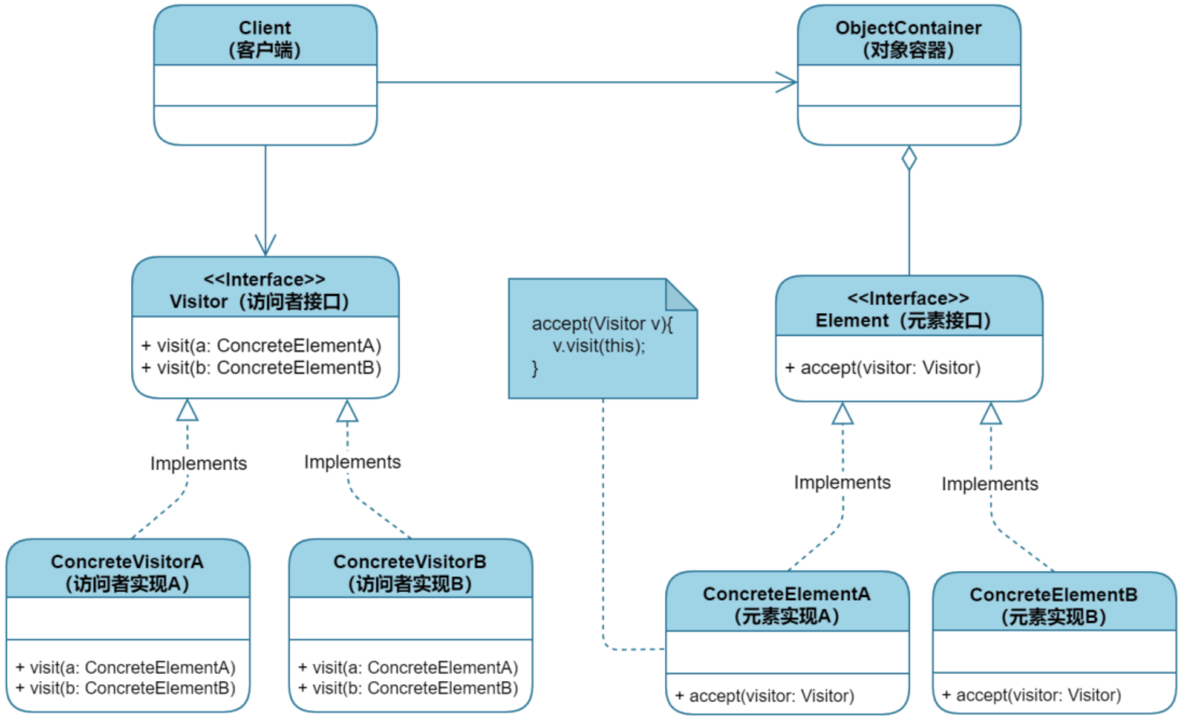
With the Visitor pattern, you define two class hierarchies: one for the elements being operated on (the Node hierarchy) and one for the visitors that define operations on the elements (the NodeVisitor hierarchy). You create a new operation by adding a new subclass to the visitor class hierarchy.

Applicability

Use the Visitor pattern when

a. an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
b. many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
c. the classes defining the object structure rarely change, but you often want to define new operations over the structure.

* Class Diagram



Participants

Visitor

- declares a Visit operation for each class of ConcreteElement in the object structure.

ConcreteVisitor

- implements each operation declared by Visitor. Each operation implements a fragment of the algorithm defined for the corresponding

Element

- defines an Accept operation that takes a visitor as an argument.

Participants

ConcreteElement

- implements an Accept operation that takes a visitor as an argument.

ObjectStructure

- can enumerate its elements.
- may provide a high-level interface to allow the visitor to visit its elements.
- may either be a composite (see Composite (183)) or a collection such as a list or a set.

Collaborations

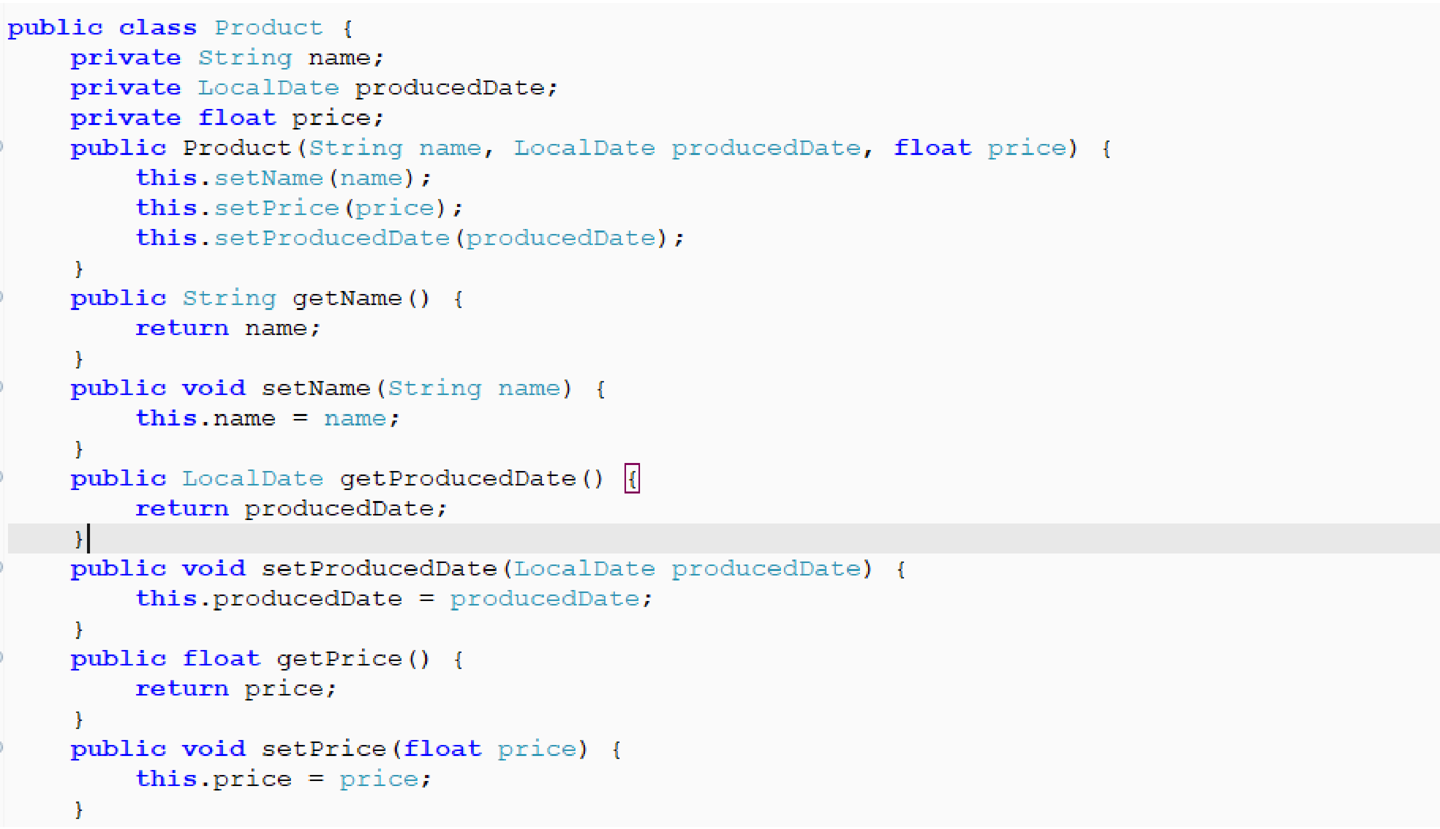
- A client that uses the Visitor pattern must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor.
- When an element is visited, it calls the Visitor operation that corresponds to its class. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary.

Consequences

The Mediator pattern has the following benefits and drawbacks:

a. Visitor makes adding new operations easy.
b. A visitor gathers related operations and separates unrelated ones.
c. Adding new ConcreteElement classes is hard.
d. Visiting across class hierarchies.
e. Accumulating state.
f. Breaking encapsulation.

Course: Soft Design



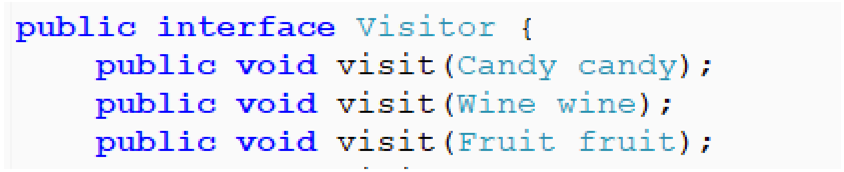
```java
public class Product {
    private String name;
    private LocalDate producedDate;
    private float price;
    public Product(String name, LocalDate producedDate, float price) {
        this.setName(name);
        this.setPrice(price);
        this.setProducedDate(producedDate);
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public LocalDate getProducedDate() {
        return producedDate;
    }
    public void setProducedDate(LocalDate producedDate) {
        this.producedDate = producedDate;
    }
    public float getPrice() {
        return price;
    }
    public void setPrice(float price) {
        this.price = price;
    }
```

```
public class Candy extends Product {
    public Candy(String name, LocalDate producedDate, float price) {
        super(name,producedDate,price);
    }
}
```

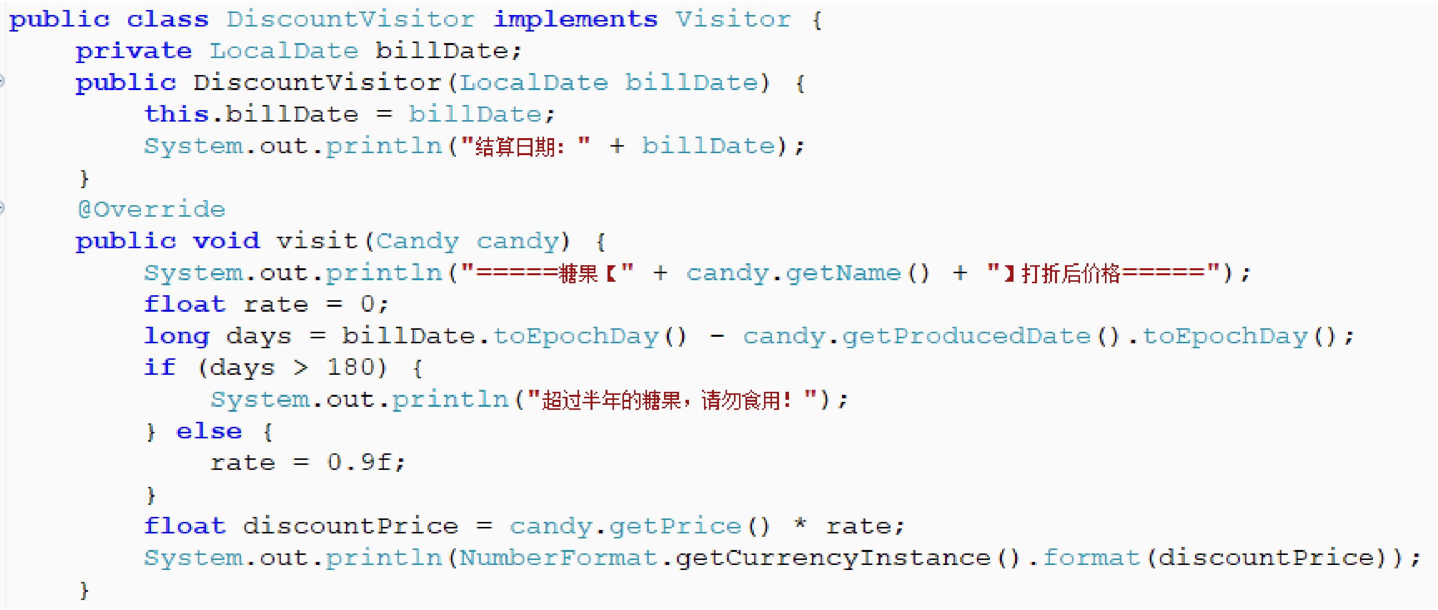
```
public class Wine extends Product{
    public Wine(String name, LocalDate producedDate, float price) {
        super(name,producedDate,price);
    }
}
```

```
public class Fruit extends Product{
    private float weight;
    public Fruit(String name, LocalDate producedDate, float price, float weight) {
        super(name,producedDate,price);
        this.setWeight(weight);
    }
    public float getWeight() {
        return weight;
    }
    public void setWeight(float weight) {
        this.weight = weight;
    }

}
```

Course: Soft Design



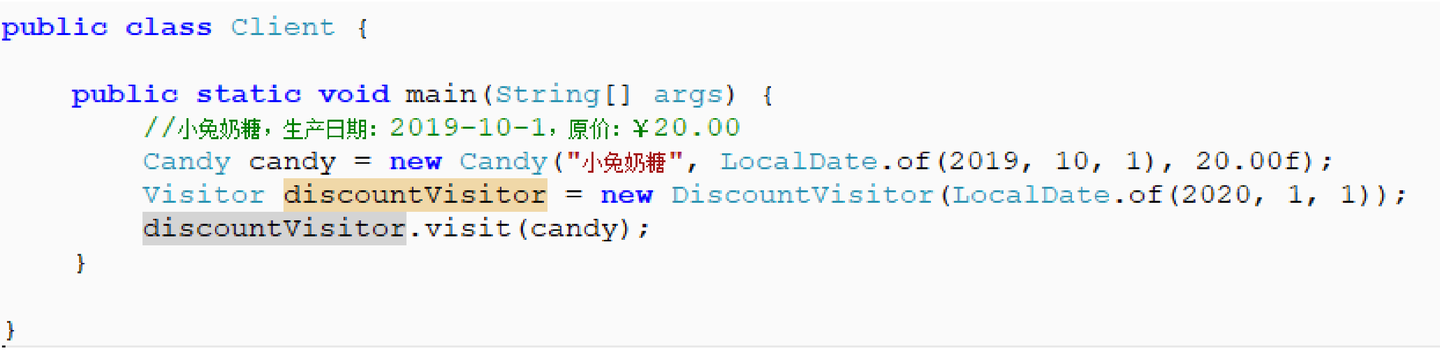
```
public interface Visitor {
    public void visit(Candy candy);
    public void visit(Wine wine);
    public void visit(Fruit fruit);
```



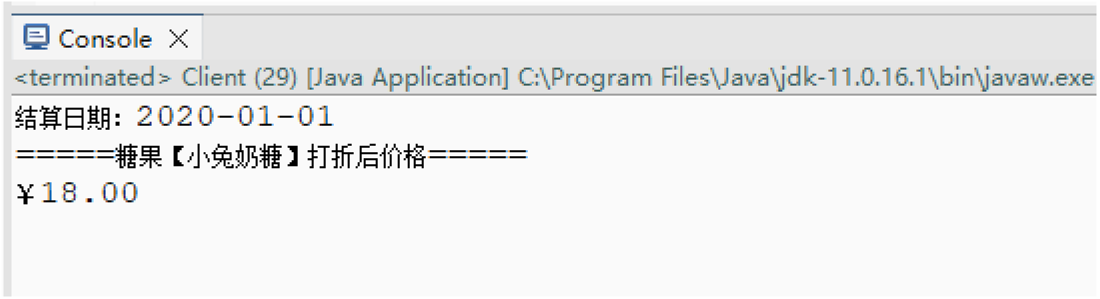
```
public class DiscountVisitor implements Visitor {
    private LocalDate billDate;
    public DiscountVisitor(LocalDate billDate) {
        this.billDate = billDate;
        System.out.println("结算日期: " + billDate);
    }
    @Override
    public void visit(Candy candy) {
        System.out.println("=====糖果【" + candy.getName() + "】打折后价格=====");
        float rate = 0;
        long days = billDate.toEpochDay() - candy.getProducedDate().toEpochDay();
        if (days > 180) {
            System.out.println("超过半年的糖果，请勿食用！");
        } else {
            rate = 0.9f;
        }
        float discountPrice = candy.getPrice() * rate;
        System.out.println(NumberFormat.getCurrencyInstance().format(discountPrice));
    }
```

Course: Soft Design

```
@Override
public void visit(Wine wine) {
    System.out.println("=====酒【" + wine.getName() + "】无折扣价格=====");
    System.out.println(
            NumberFormat.getCurrencyInstance().format(wine.getPrice())
            );

}
@Override
public void visit(Fruit fruit) {
    System.out.println("=====水果【" + fruit.getName() + "】打折后价格=====");
    float rate = 0;
    long days = billDate.toEpochDay() - fruit.getProducedDate().toEpochDay();
    if (days > 7) {
        System.out.println("¥0.00元（超过7天的水果，请勿食用！）");
    } else if (days > 3) {
        rate = 0.5f;
    } else {
        rate = 1;
    }
    float discountPrice = fruit.getPrice() * fruit.getWeight() * rate;
    System.out.println(NumberFormat.getCurrencyInstance().format(discountPrice));

}
```



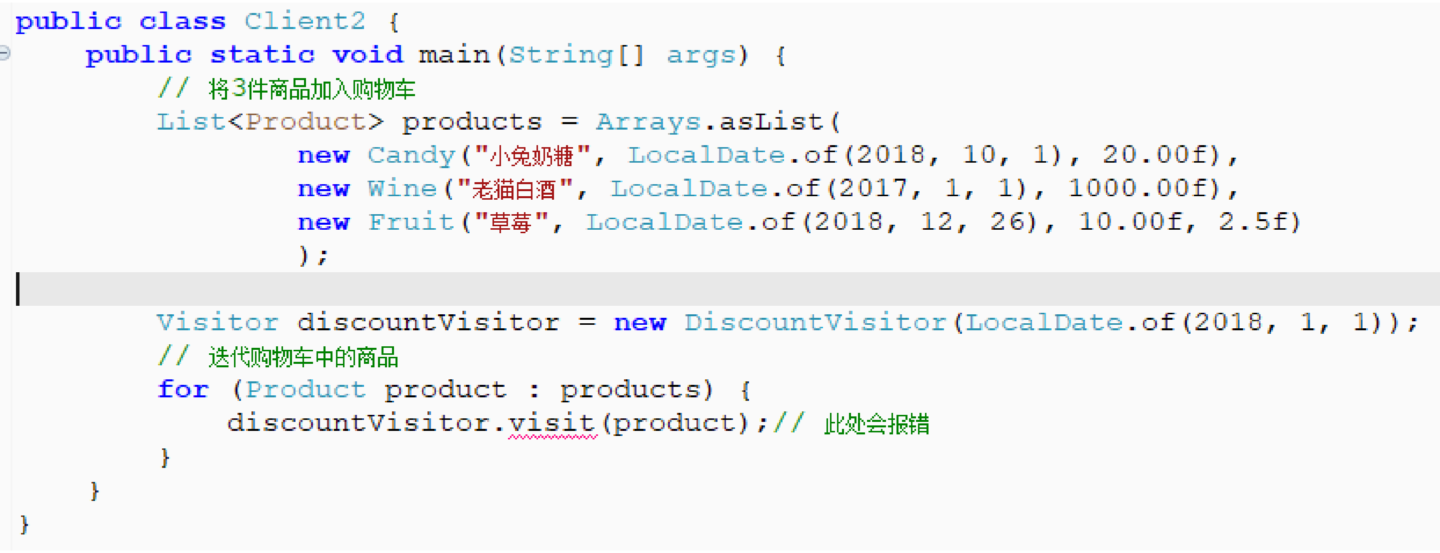
```
public class Client {

    public static void main(String[] args) {
        //小兔奶糖，生产日期：2019-10-1，原价：￥20.00
        Candy candy = new Candy("小兔奶糖", LocalDate.of(2019, 10, 1), 20.00f);
        Visitor discountVisitor = new DiscountVisitor(LocalDate.of(2020, 1, 1));
        discountVisitor.visit(candy);
    }

}
```



Console

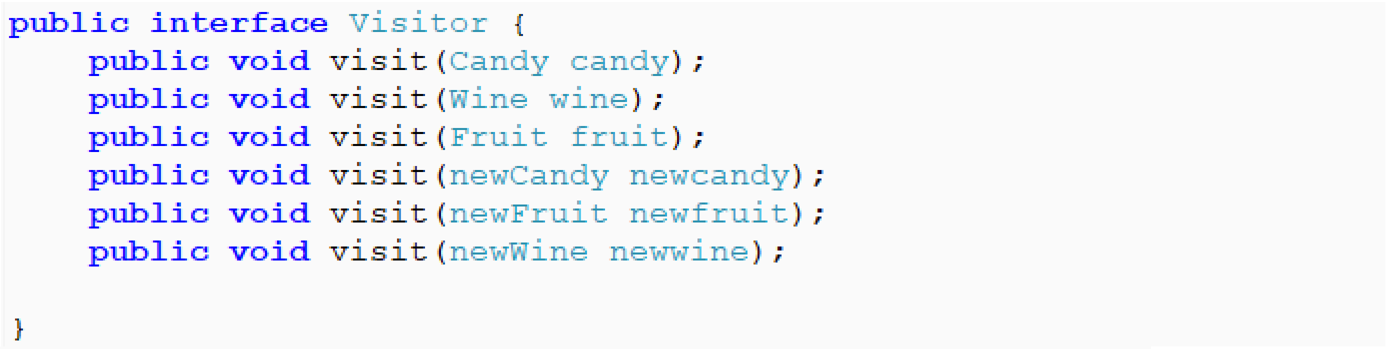
```
<terminated> Client (29) [Java Application] C:\Program Files\Java\jdk-11.0.16.1\bin\javaw.exe
结算日期：2020-01-01
=====糖果【小兔奶糖】打折后价格=====
￥18.00
```

Course: Soft Design

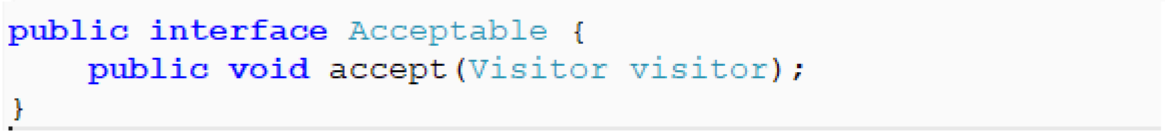


```
public class Client2 {
    public static void main(String[] args) {
        // 将3件商品加入购物车
        List<Product> products = Arrays.asList(
                new Candy("小兔奶糖", LocalDate.of(2018, 10, 1), 20.00f),
                new Wine("老猫白酒", LocalDate.of(2017, 1, 1), 1000.00f),
                new Fruit("草莓", LocalDate.of(2018, 12, 26), 10.00f, 2.5f)
                );

        Visitor discountVisitor = new DiscountVisitor(LocalDate.of(2018, 1, 1));
        // 迭代购物车中的商品
        for (Product product : products) {
            discountVisitor.visit(product);// 此处会报错
        }
    }
}
```

Course: Soft Design

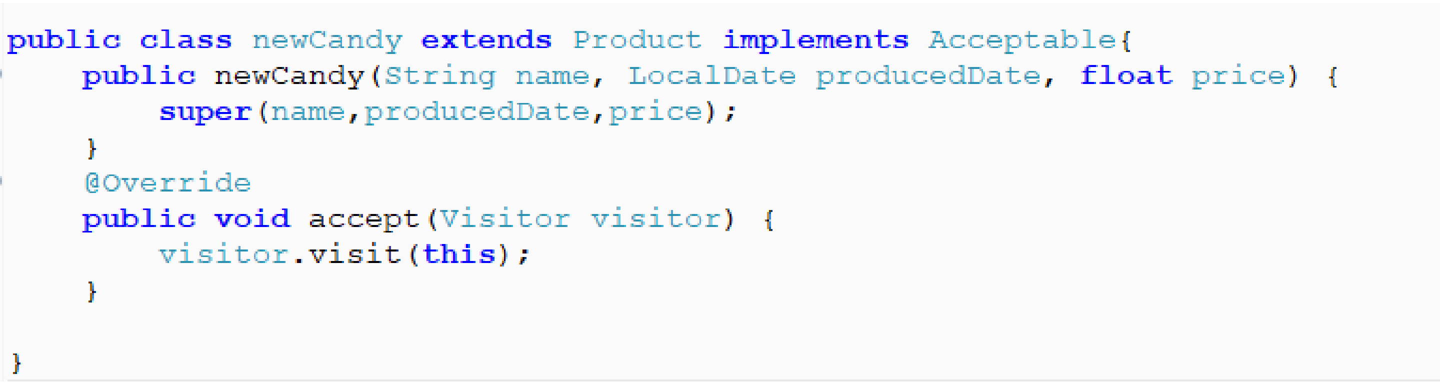


```
public interface Visitor {
    public void visit(Candy candy);
    public void visit(Wine wine);
    public void visit(Fruit fruit);
    public void visit(newCandy newcandy);
    public void visit(newFruit newfruit);
    public void visit(newWine newwine);

}
```

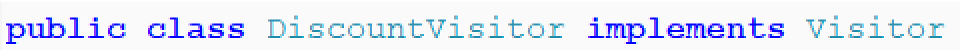


```
public interface Acceptable {
    public void accept(Visitor visitor);
}
```

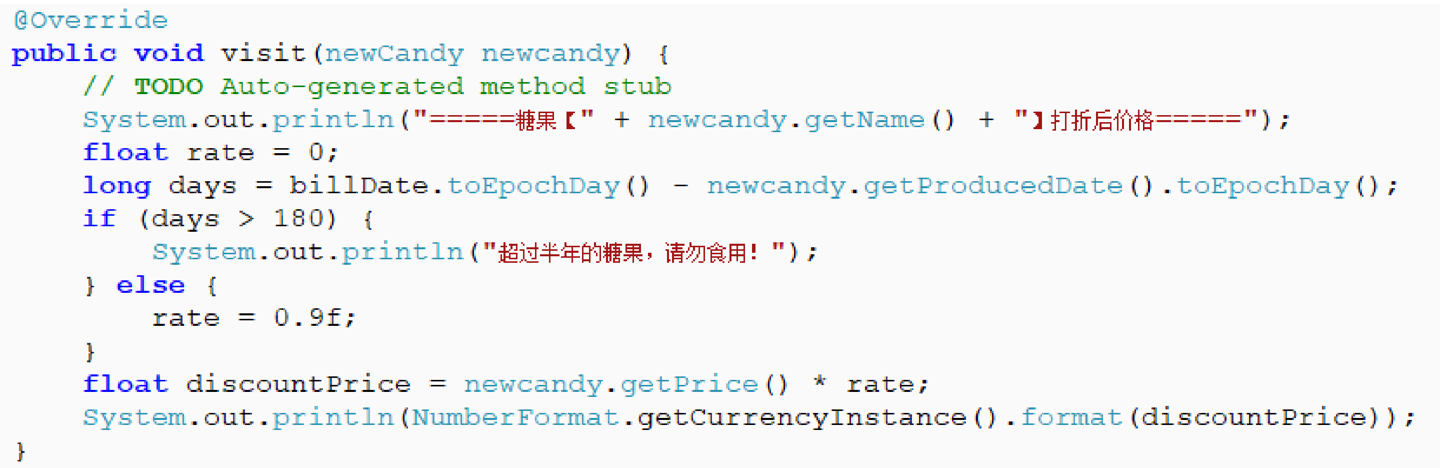
Course: Soft Design



```java
public class newCandy extends Product implements Acceptable{
    public newCandy(String name, LocalDate producedDate, float price) {
        super(name,producedDate,price);
    }
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

}
```

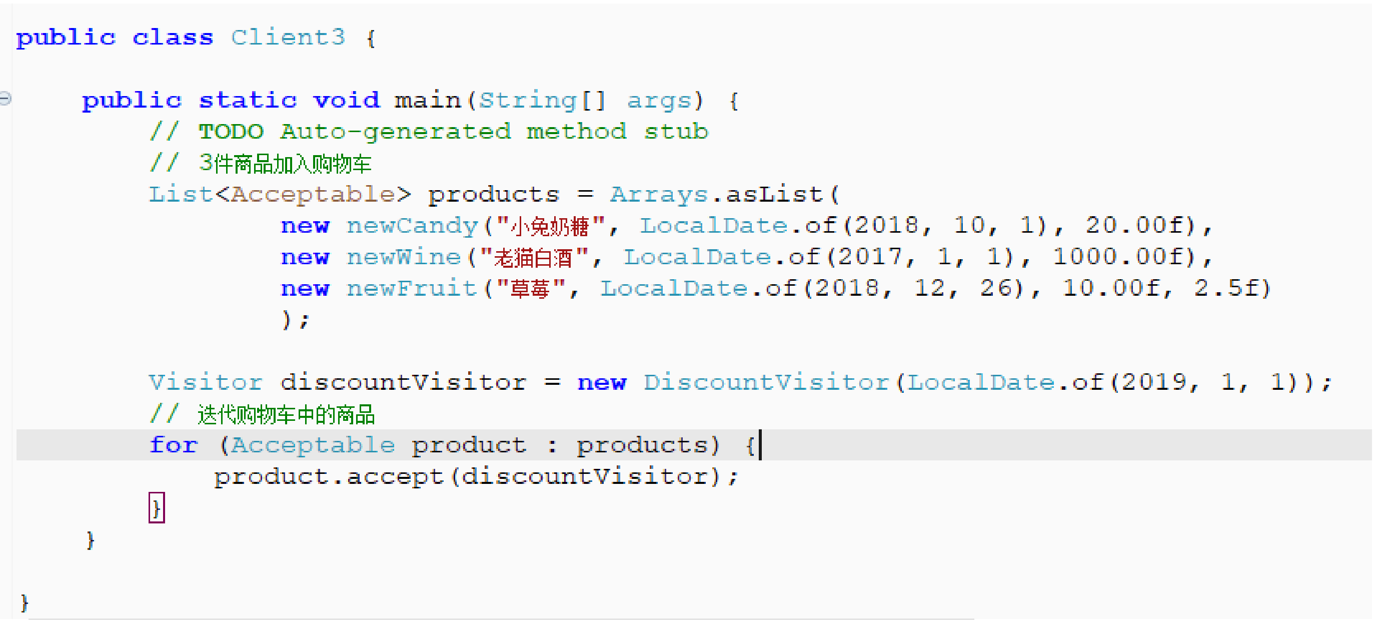


```java
public class DiscountVisitor implements Visitor
```

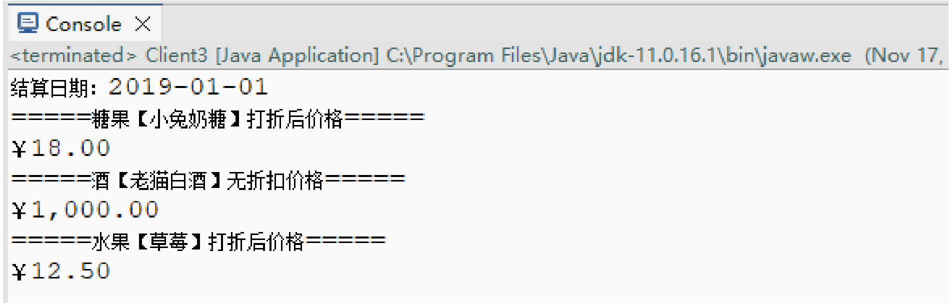


```java
@Override
public void visit(newCandy newcandy) {
    // TODO Auto-generated method stub
    System.out.println("=====糖果【" + newcandy.getName() + "】打折后价格=====");
    float rate = 0;
    long days = billDate.toEpochDay() - newcandy.getProducedDate().toEpochDay();
    if (days > 180) {
        System.out.println("超过半年的糖果，请勿食用！");
    } else {
        rate = 0.9f;
    }
    float discountPrice = newcandy.getPrice() * rate;
    System.out.println(NumberFormat.getCurrencyInstance().format(discountPrice));
}
```

Course: Soft Design



```
public class Client3 {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // 3件商品加入购物车
        List<Acceptable> products = Arrays.asList(
                new newCandy("小兔奶糖", LocalDate.of(2018, 10, 1), 20.00f),
                new newWine("老猫白酒", LocalDate.of(2017, 1, 1), 1000.00f),
                new newFruit("草莓", LocalDate.of(2018, 12, 26), 10.00f, 2.5f)
                );

        Visitor discountVisitor = new DiscountVisitor(LocalDate.of(2019, 1, 1));
        // 迭代购物车中的商品
        for (Acceptable product : products) {
            product.accept(discountVisitor);
        }
    }

}
```



Console ✕

<terminated> Client3 [Java Application] C:\Program Files\Java\jdk-11.0.16.1\bin\javaw.exe (Nov 17,

```
结算日期: 2019-01-01
=====糖果【小兔奶糖】打折后价格=====
¥18.00
=====酒【老猫白酒】无折扣价格=====
¥1,000.00
=====水果【草莓】打折后价格=====
¥12.50
```

Observer(觀察者模式)

Intent

Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Also known as

Dependents, Publish-Subscribe

Motivation

A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state.

Applicability

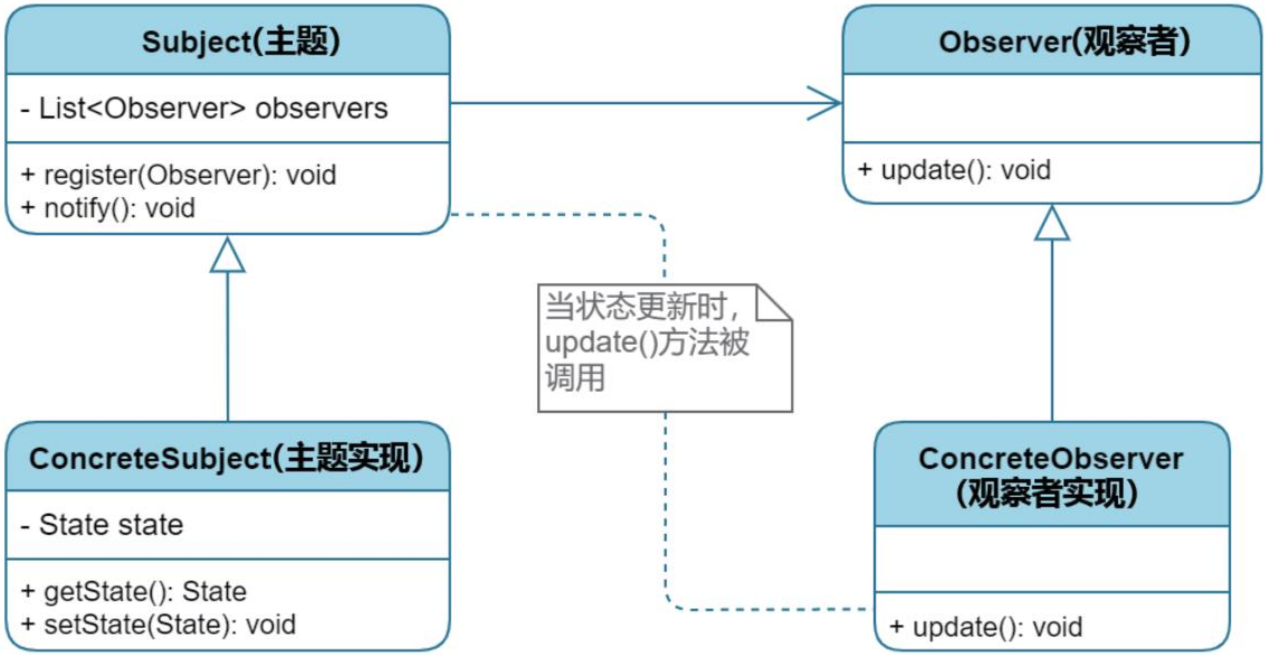
Use the Observer pattern in any of the following situations:

a. When an abstraction has two aspects, one dependent on the other.

b. When a change to one object requires changing others, and you don't know how many objects need to be changed.

c. When an object should be able to notify other objects without making assumptions about who these objects are.

* Class Diagram


Subject(主题)
- List<Observer> observers
+ register(Observer): void
+ notify(): void
Observer(观察者)
+ update(): void
当状态更新时，update()方法被调用
ConcreteSubject(主题实现)
- State state
+ getState(): State
+ setState(State): void
ConcreteObserver (观察者实现)
+ update(): void

Participants

Subject

- knows its observers. Any number of Observer objects may observe a subject.
- provides an interface for attaching and detaching Observer objects.

Observer

- defines an updating interface for objects that should be notified of changes in a subject.

Participants

ConcreteSubject

- stores state of interest to ConcreteObserver objects.
- sends a notification to its observers when its state changes.

ConcreteObserver

- maintains a reference to a ConcreteSubject object.
- stores state that should stay consistent with the subject's.
- implements the Observer updating interface to keep its state consistent with the subject's.

Collaborations

- ConcreteSubject notifies its observers whenever a change occurs that could make its observers' state inconsistent with its own.
- After being informed of a change in the concrete subject, a ConcreteObserver object may query the subject for information.

Consequences

Further benefits and liabilities of the Observer pattern include the following:

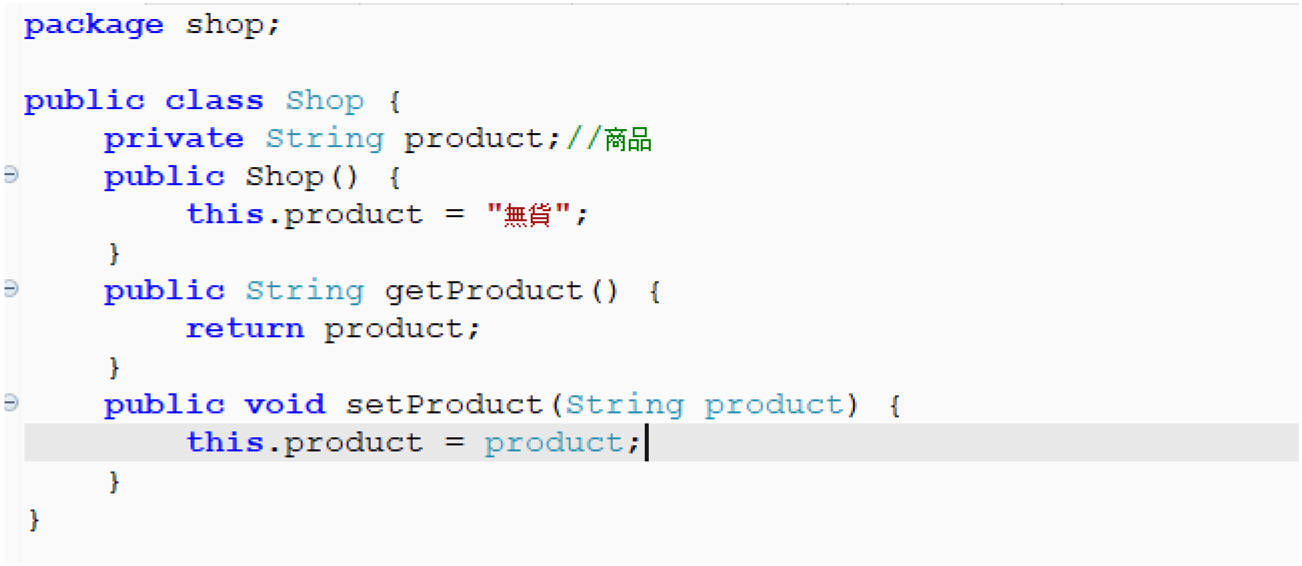
a. Abstract coupling between Subject and Observer.
b. Support for broadcast communication.
c. Unexpected updates.

Implementation

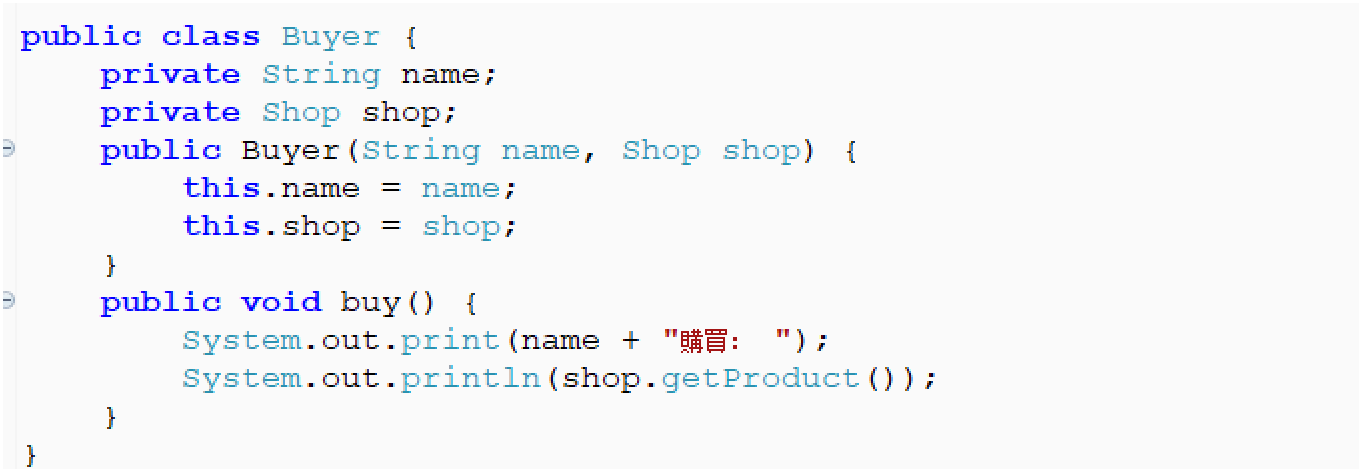
Several issues related to the implementation of the dependency mechanism are discussed in this section.

1. Mapping subjects to their observers.
2. Observing more than one subject.
3. Who triggers the update?
4. Dangling references to deleted subjects.
5. Making sure Subject state is self-consistent before notification.
6. Avoiding observer-specific update protocols: the push and pull models.
7. Specifying modifications of interest explicitly.
8. Encapsulating complex update semantics.
9. Combining the Subject and Observer classes.

Course: Soft Design

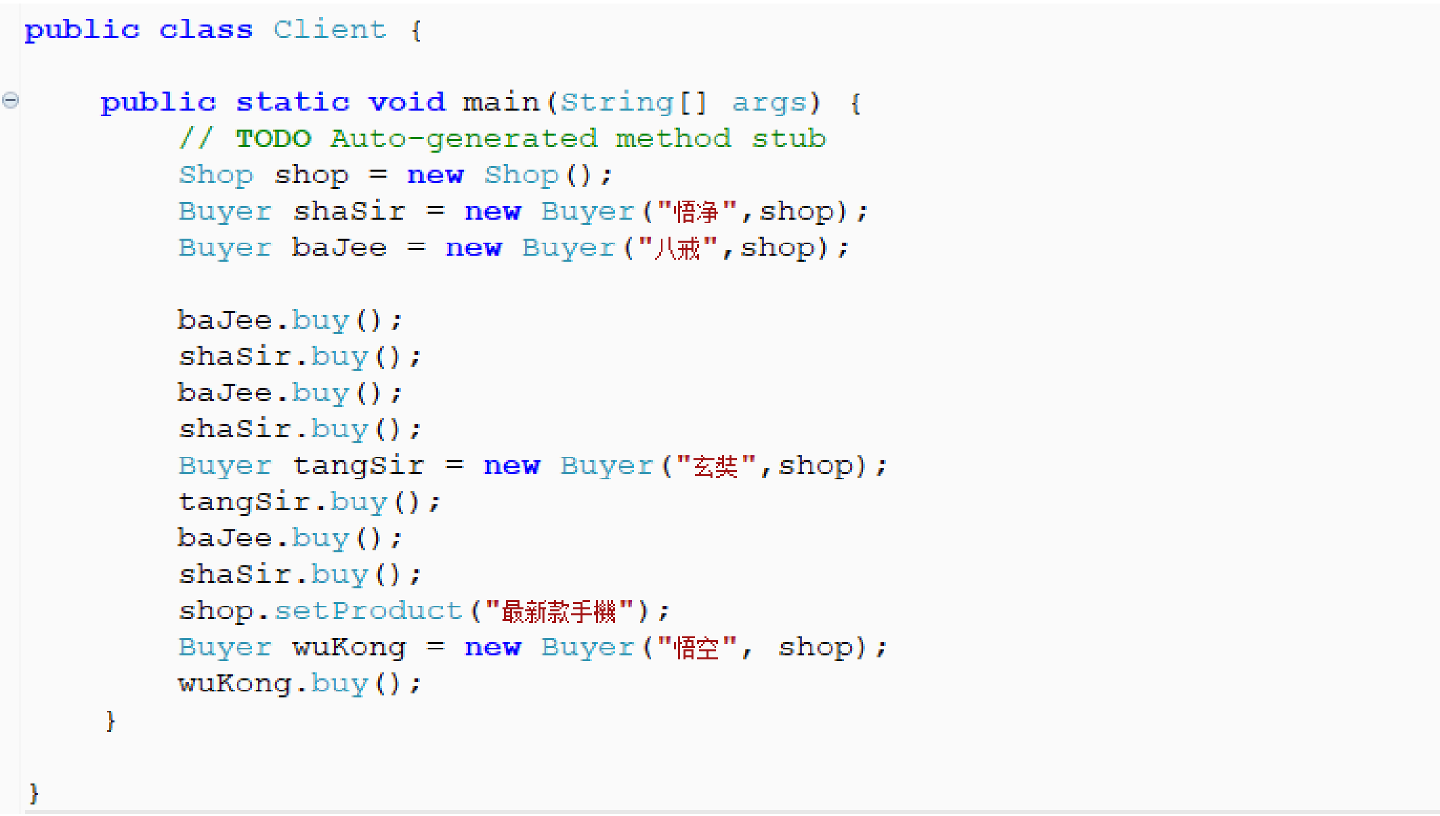


```java
package shop;

public class Shop {
    private String product;//商品
    public Shop() {
        this.product = "無貨";
    }
    public String getProduct() {
        return product;
    }
    public void setProduct(String product) {
        this.product = product;
    }
}
```



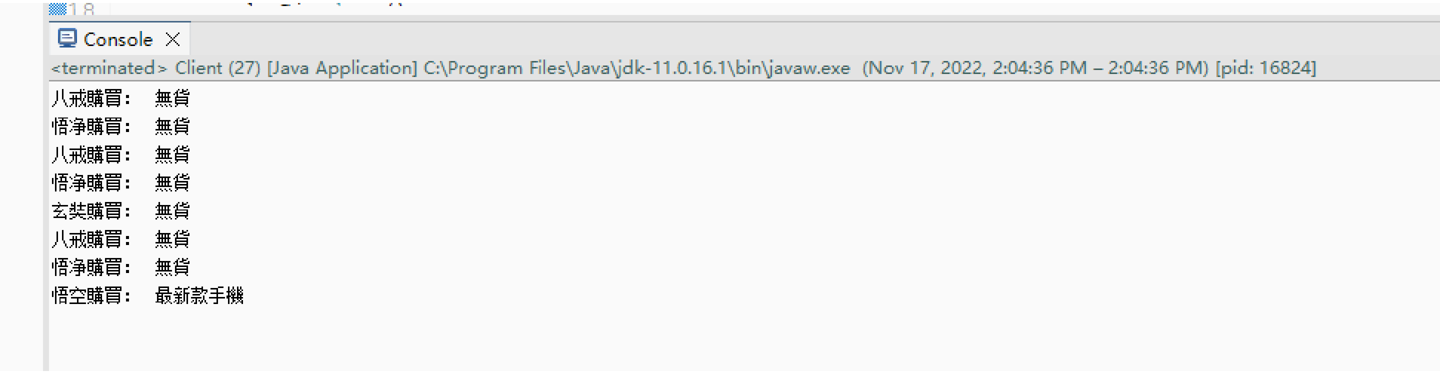
```java
public class Buyer {
    private String name;
    private Shop shop;
    public Buyer(String name, Shop shop) {
        this.name = name;
        this.shop = shop;
    }
    public void buy() {
        System.out.print(name + "購買:  ");
        System.out.println(shop.getProduct());
    }
}
```

Course: Soft Design



```java
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Shop shop = new Shop();
        Buyer shaSir = new Buyer("悟净",shop);
        Buyer baJee = new Buyer("八戒",shop);

        baJee.buy();
        shaSir.buy();
        baJee.buy();
        shaSir.buy();
        Buyer tangSir = new Buyer("玄奘",shop);
        tangSir.buy();
        baJee.buy();
        shaSir.buy();
        shop.setProduct("最新款手機");
        Buyer wuKong = new Buyer("悟空", shop);
        wuKong.buy();
    }

}
```

Course: Soft Design



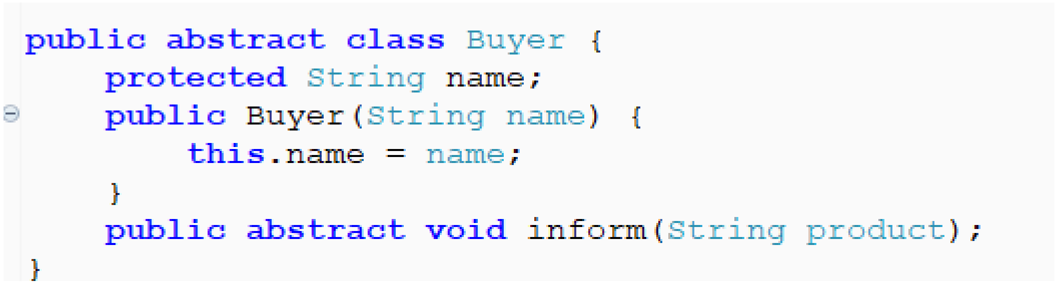
Console

```
<terminated> Client (27) [Java Application] C:\Program Files\Java\jdk-11.0.16.1\bin\javaw.exe  (Nov 17, 2022, 2:04:36 PM – 2:04:36 PM) [pid: 16824]
八戒購買:   無貨
悟淨購買:   無貨
八戒購買:   無貨
悟淨購買:   無貨
玄奘購買:   無貨
八戒購買:   無貨
悟淨購買:   無貨
悟空購買:   最新款手機
```

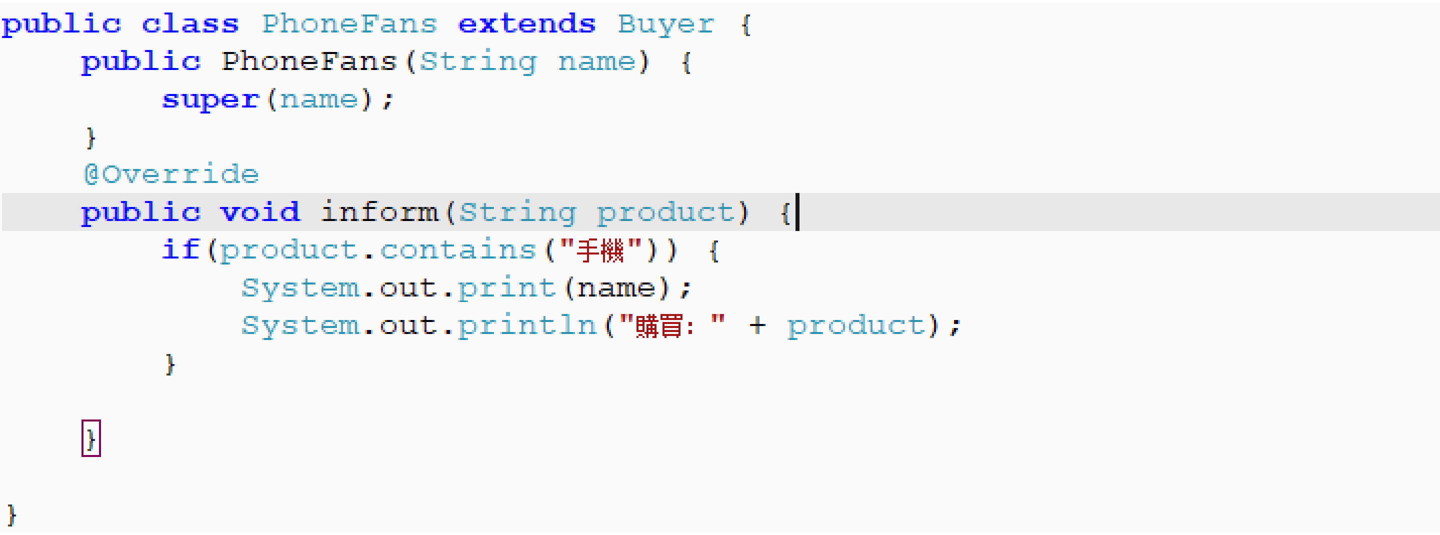
Course: Soft Design

```java
public class SmartShop {
    private String product;
    private List<Buyer> buyers;
    public SmartShop() {
        this.product = "無商品";
        this.buyers = new ArrayList<>();
    }
    public void register(Buyer buyer) {
        this.buyers.add(buyer);
    }
    public String getProduct() {
        return product;
    }
    public void setProduct(String product) {
        this.product = product;
        notifyBuyers();
    }
    public void notifyBuyers() {
        buyers.stream().forEach(b->b.inform(this.getProduct()));
    }
}
```

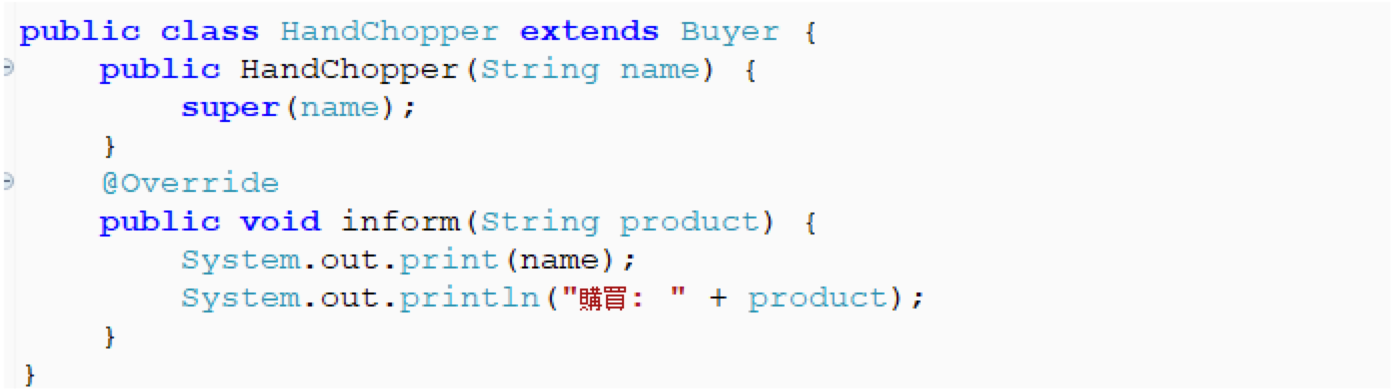
Course: Soft Design



```java
public abstract class Buyer {
    protected String name;
    public Buyer(String name) {
        this.name = name;
    }
    public abstract void inform(String product);
}
```

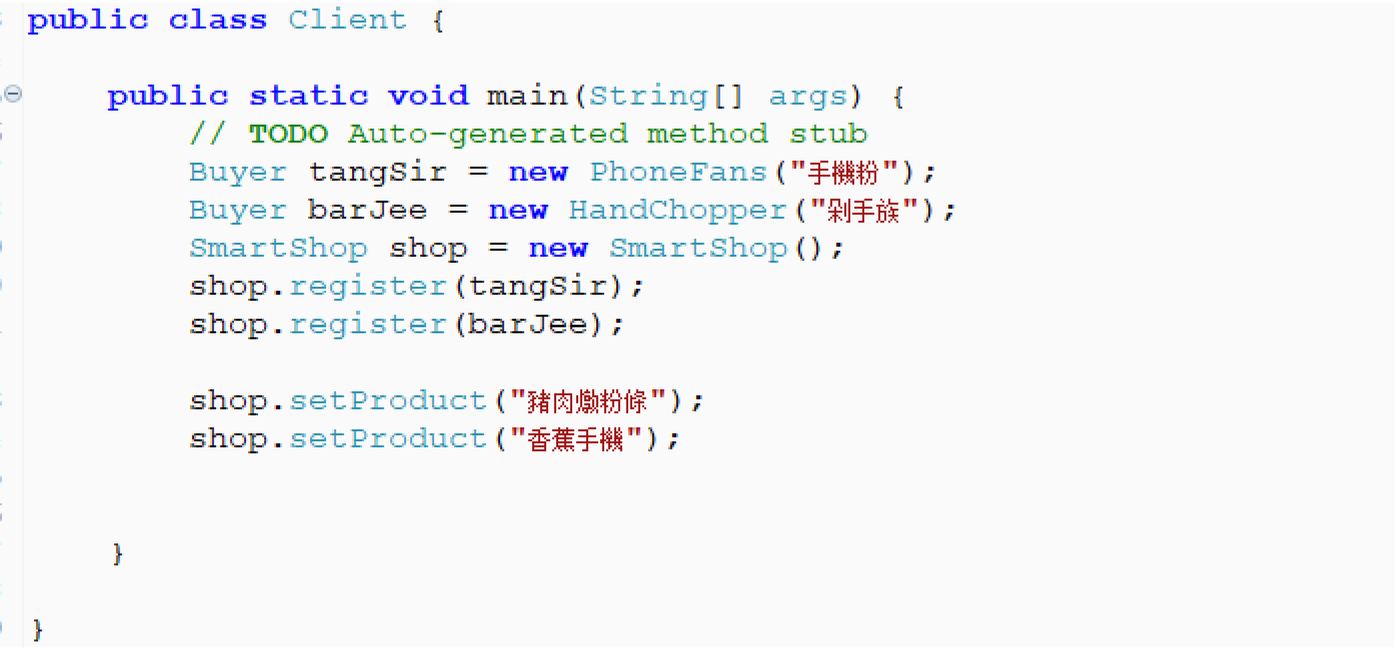


```java
public class PhoneFans extends Buyer {
    public PhoneFans(String name) {
        super(name);
    }
    @Override
    public void inform(String product) {
        if(product.contains("手機")) {
            System.out.print(name);
            System.out.println("購買: " + product);
        }

    }

}
```

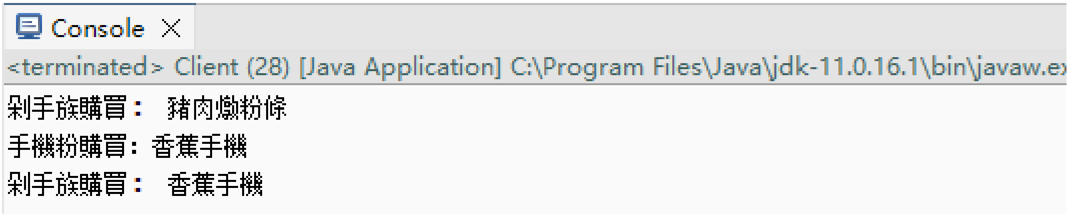


```java
public class HandChopper extends Buyer {
    public HandChopper(String name) {
        super(name);
    }
    @Override
    public void inform(String product) {
        System.out.print(name);
        System.out.println("購買: " + product);
    }
}
```

Course: Soft Design



```
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        Buyer tangSir = new PhoneFans("手機粉");
        Buyer barJee = new HandChopper("剁手族");
        SmartShop shop = new SmartShop();
        shop.register(tangSir);
        shop.register(barJee);

        shop.setProduct("豬肉燉粉條");
        shop.setProduct("香蕉手機");


    }

}
```



```
Console ✕
<terminated> Client (28) [Java Application] C:\Program Files\Java\jdk-11.0.16.1\bin\javaw.ex
剁手族購買：  豬肉燉粉條
手機粉購買: 香蕉手機
剁手族購買：  香蕉手機
```

Interpreter (解釋器模式)

Intent

Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

Motivation

If a particular kind of problem occurs often enough, then it might be worthwhile to express instances of the problem as sentences in a simple language.

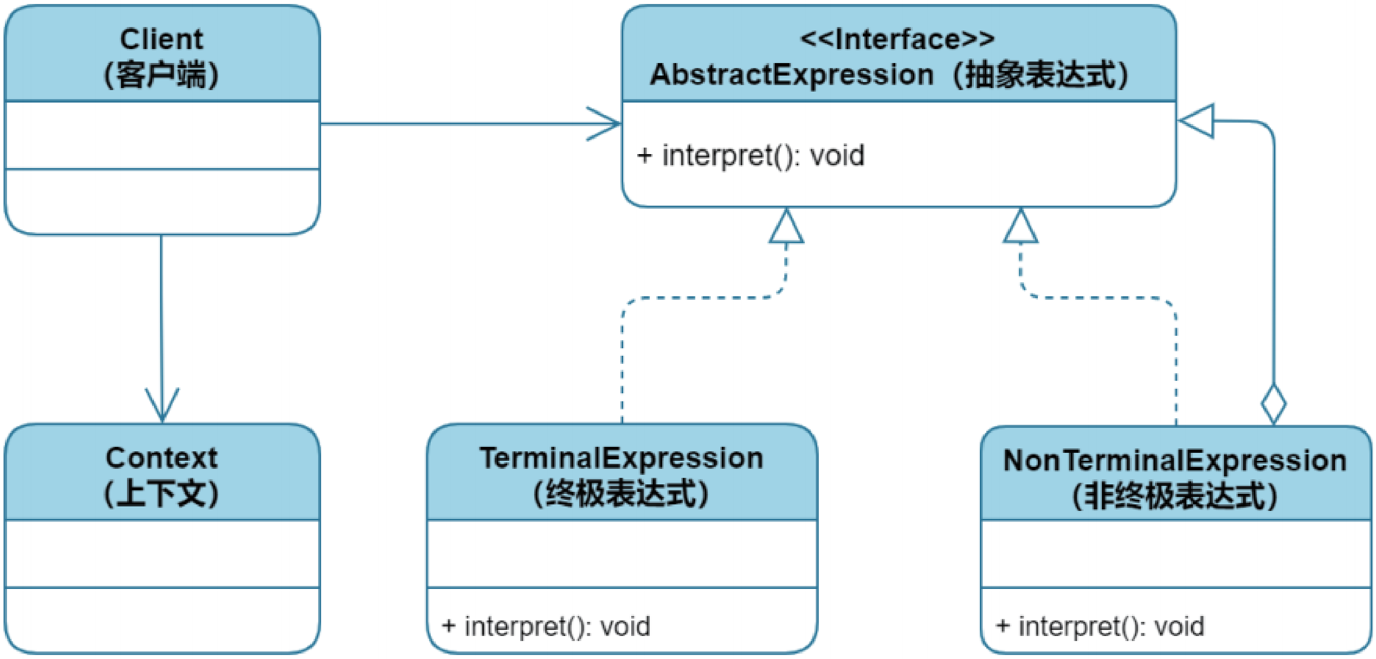
Applicability

The Interpreter pattern works best when

the grammar is simple. For complex grammars, the class hierarchy for the grammar becomes large and unmanageable.

efficiency is not a critical concern. The most efficient interpreters are usually not implemented by interpreting parse trees directly but by first translating them into another form.

* Class Diagram



Participants

AbstractExpression

- declares an abstract Interpret operation that is common to all nodes in the abstract syntax tree.

TerminalExpression

- implements an Interpret operation associated with terminal symbols in the grammar.
- an instance is required for every terminal symbol in a sentence.

NonterminalExpression

- one such class is required for every rule R ::= R1 R2 ... Rn in the grammar.
- maintains instance variables of typeAbstractExpression for each of the symbols R1 through Rn.
- implements an Interpret operation for nonterminal symbols in the grammar. Interpret typically calls itself recursively on the variables representing R1 through Rn.

Context

- contains information that's global to the interpreter.

Clinet

- builds (or is given) an abstract syntax tree representing a particular sentence in the language that the grammar defines.request.
- invokes the Interpret operation.

Collaborations

- The client builds (or is given) the sentence as an abstract syntax tree of NonterminalExpression and TerminalExpression instances.
- Each NonterminalExpression node defines Interpret in terms of Interpret on each subexpression.
- The Interpret operations at each node use the context to store and access the state of the interpreter.

Consequences

The Interpreter pattern has the following benefits and liabilities:

- It's easy to change and extend the grammar.
- Implementing the grammar is easy, too.
- Complex grammars are hard to maintain. The Interpreter pattern defines at least one class for every rule in the grammar (grammar rules defined using BNF may require multiple classes).
- Adding new ways to interpret expressions.

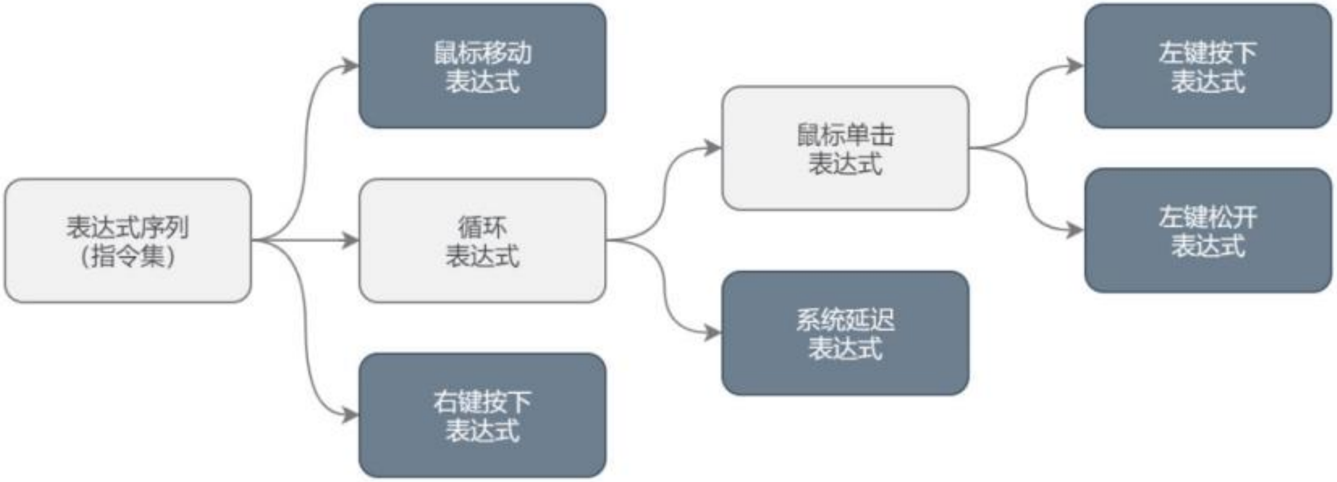
Implementation

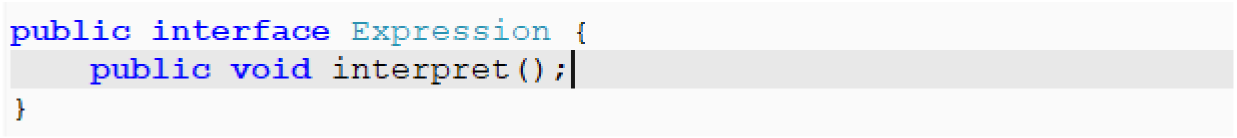
The following issues are specific to Interpreter:

- Creating the abstract syntax tree.
- Defining the Interpret operation.
- Sharing terminal symbols with the Flyweight pattern.

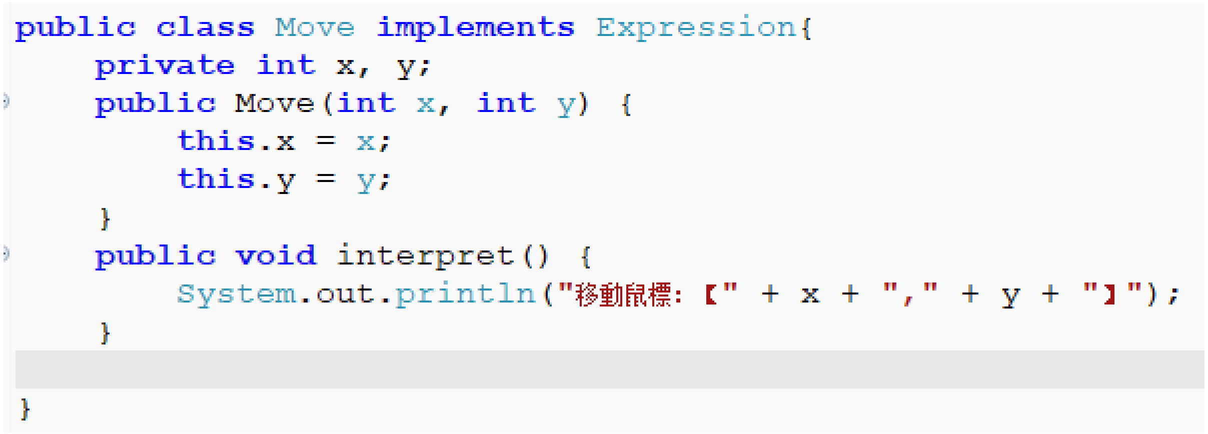
Course: Soft Design

```
* BEGIN             // 脚本开始
* MOVE 500,600;     // 鼠标指针移动到坐标(500, 600)
*   BEGIN LOOP 5    // 开始循环5次
*       LEFT_CLICK; // 循环体内单击左键
*       DELAY 1;    // 每次延迟1秒
*   END;            // 循环体结束
* RIGHT_DOWN;       // 按下右键
* DELAY 7200;       // 延迟2小时
* END;              // 脚本结束
```



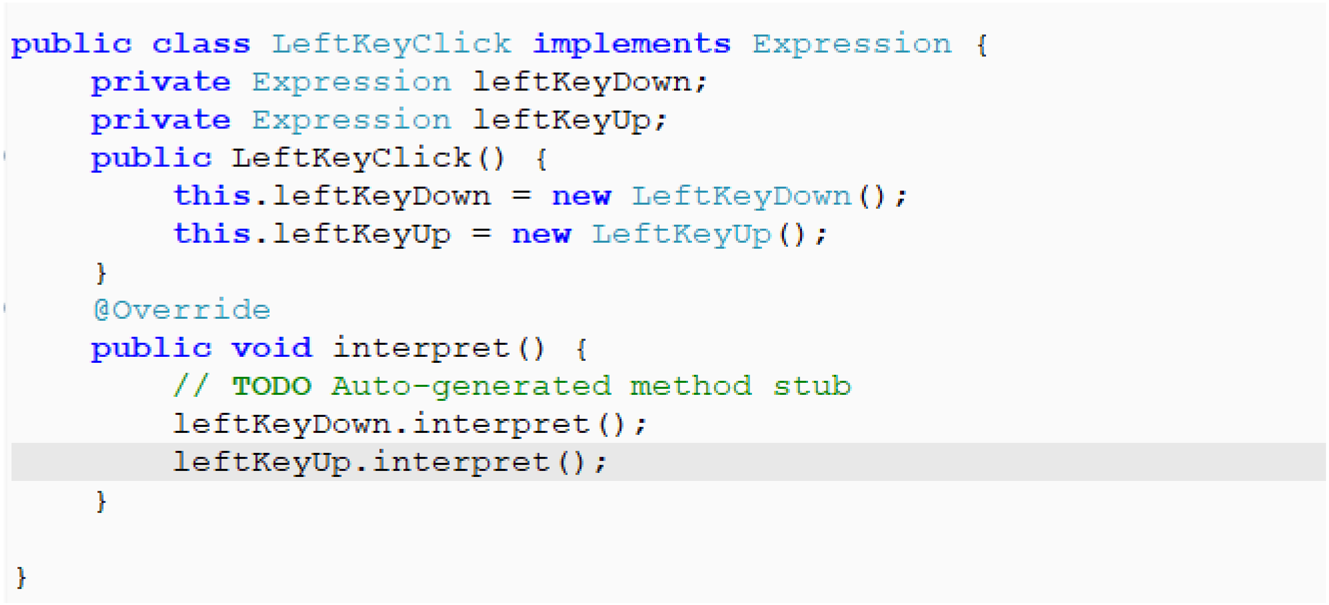


```
public interface Expression {
    public void interpret();
}
```

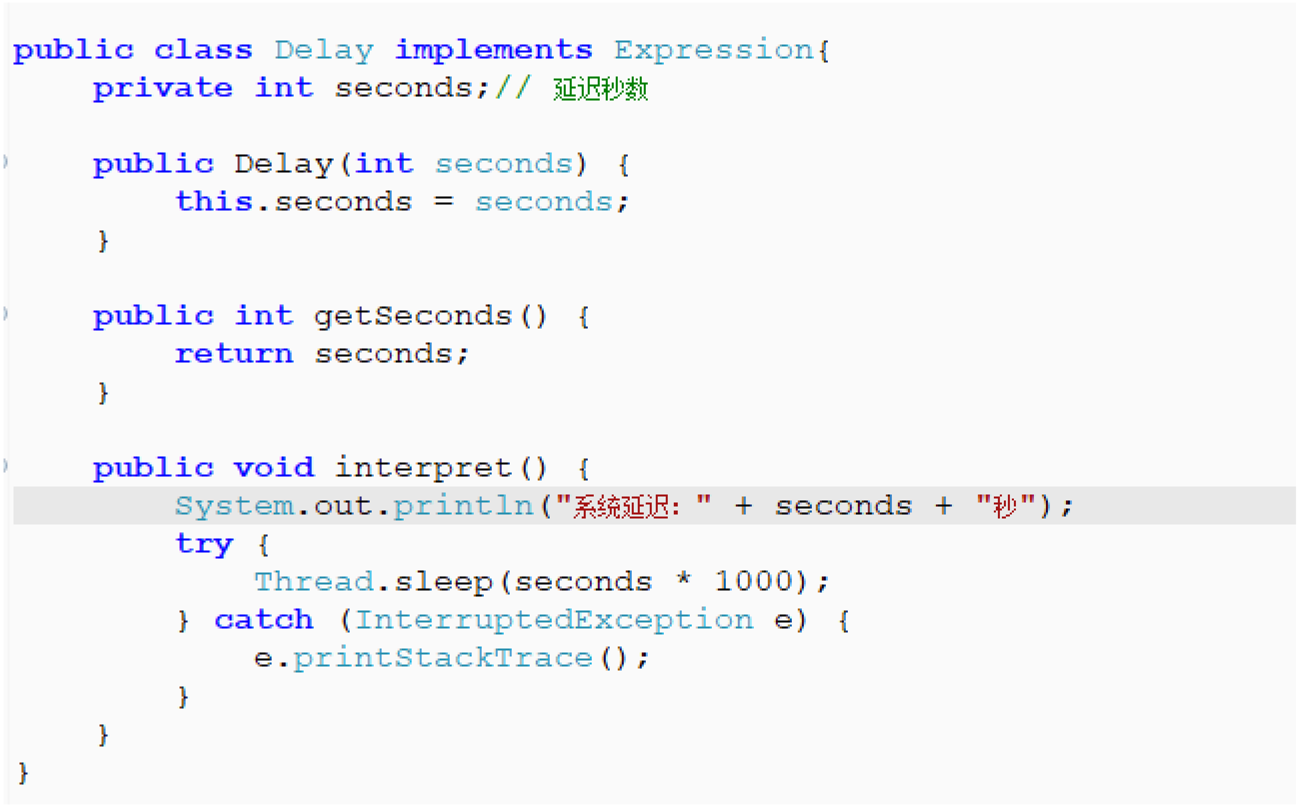


```
public class Move implements Expression{
    private int x, y;
    public Move(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public void interpret() {
        System.out.println("移動鼠標: 【" + x + "," + y + "】");
    }

}
```

Course: Soft Design

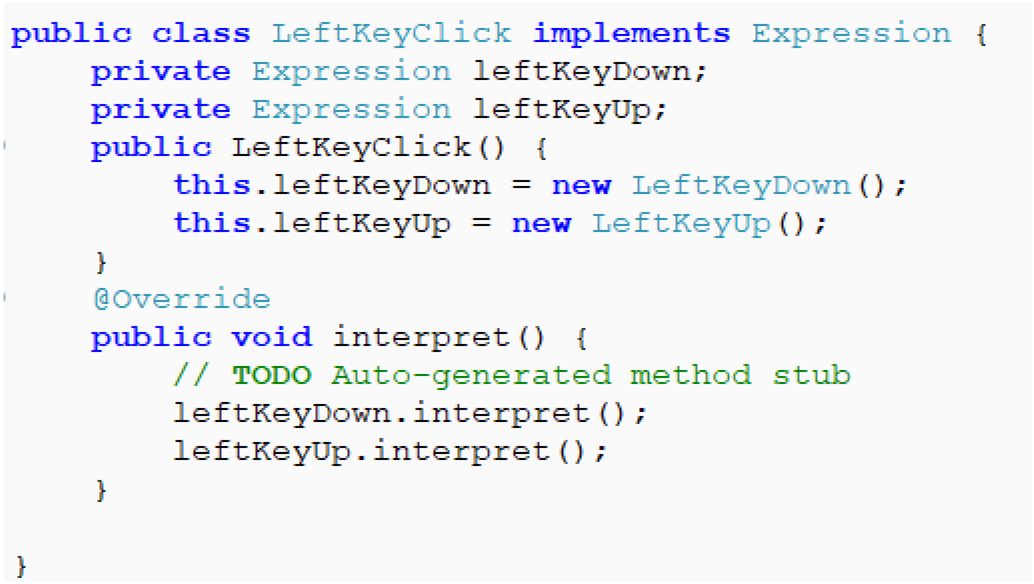


```java
public class LeftKeyClick implements Expression {
    private Expression leftKeyDown;
    private Expression leftKeyUp;
    public LeftKeyClick() {
        this.leftKeyDown = new LeftKeyDown();
        this.leftKeyUp = new LeftKeyUp();
    }
    @Override
    public void interpret() {
        // TODO Auto-generated method stub
        leftKeyDown.interpret();
        leftKeyUp.interpret();
    }

}
```



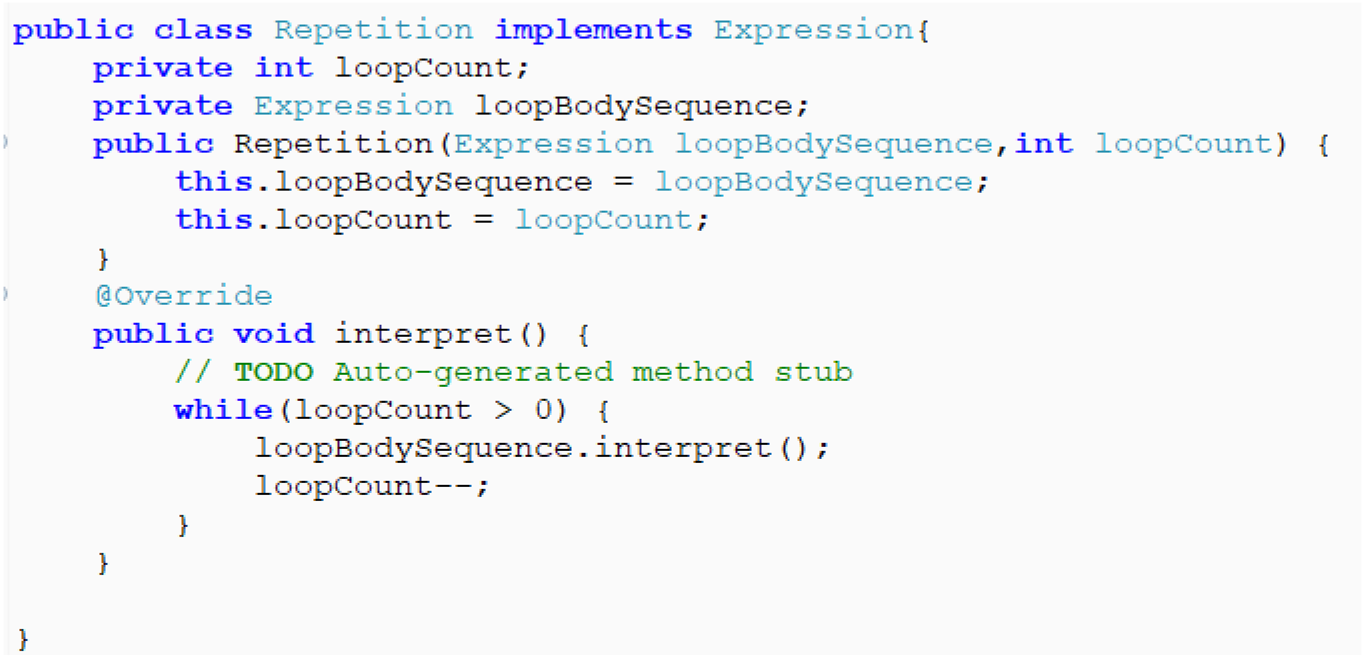
```
public class Delay implements Expression{
    private int seconds;// 延迟秒数

    public Delay(int seconds) {
        this.seconds = seconds;
    }

    public int getSeconds() {
        return seconds;
    }

    public void interpret() {
        System.out.println("系统延迟: " + seconds + "秒");
        try {
            Thread.sleep(seconds * 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Course: Soft Design



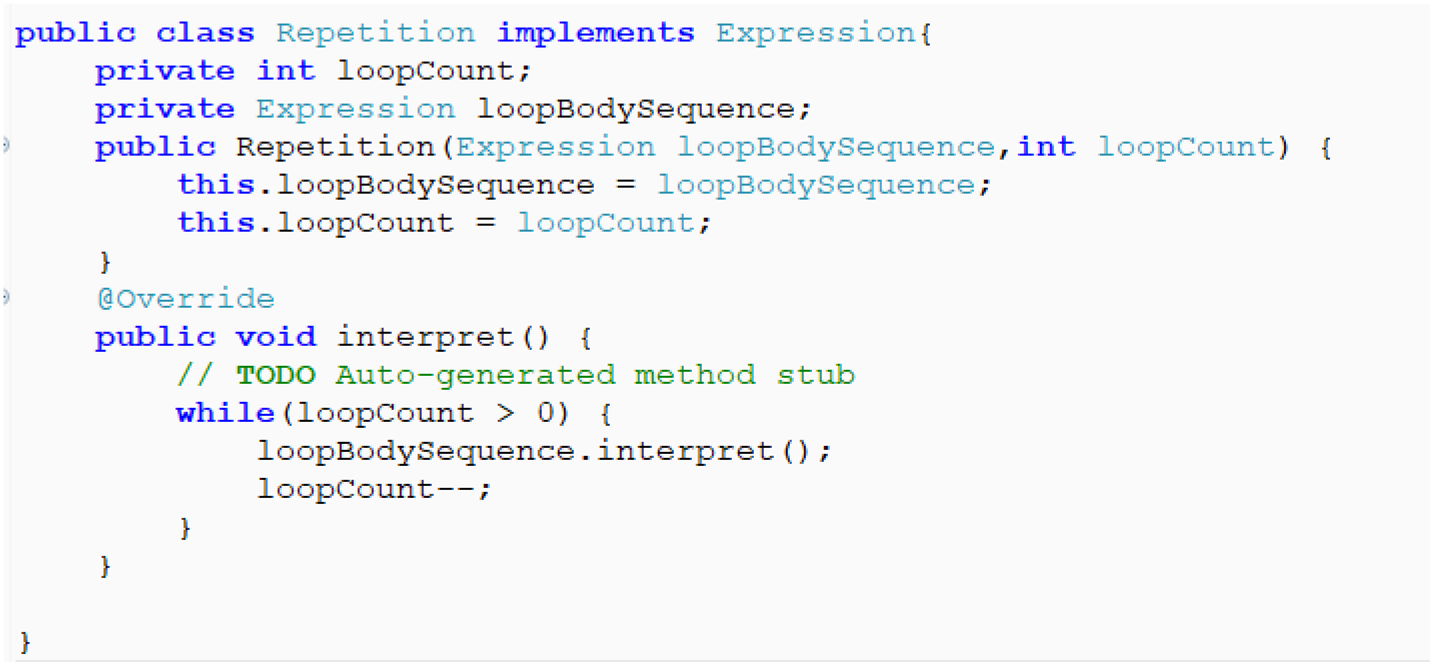
```
public class LeftKeyClick implements Expression {
    private Expression leftKeyDown;
    private Expression leftKeyUp;
    public LeftKeyClick() {
        this.leftKeyDown = new LeftKeyDown();
        this.leftKeyUp = new LeftKeyUp();
    }
    @Override
    public void interpret() {
        // TODO Auto-generated method stub
        leftKeyDown.interpret();
        leftKeyUp.interpret();
    }

}
```

Course: Soft Design



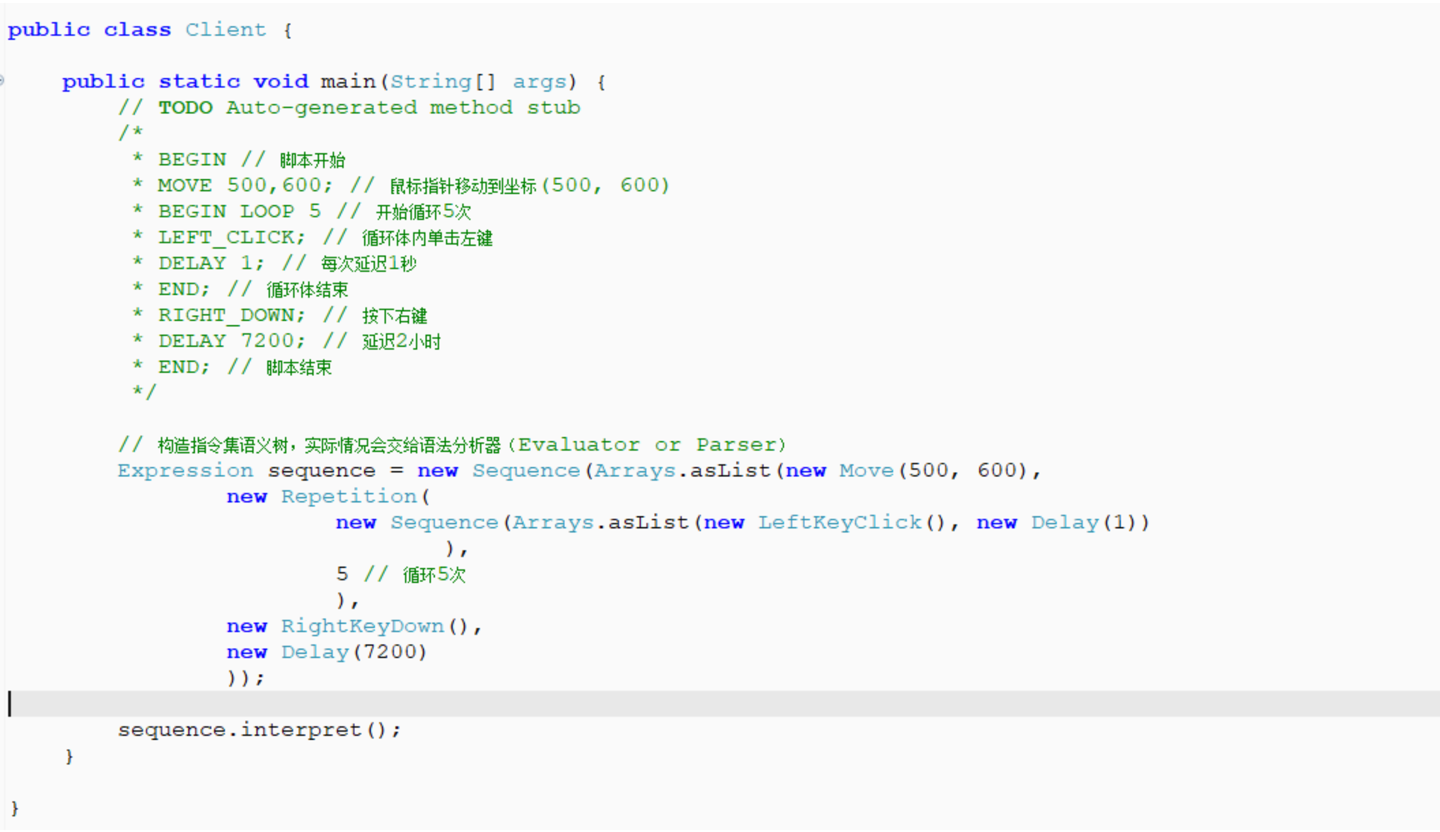
```
public class Repetition implements Expression{
    private int loopCount;
    private Expression loopBodySequence;
    public Repetition(Expression loopBodySequence,int loopCount) {
        this.loopBodySequence = loopBodySequence;
        this.loopCount = loopCount;
    }
    @Override
    public void interpret() {
        // TODO Auto-generated method stub
        while(loopCount > 0) {
            loopBodySequence.interpret();
            loopCount--;
        }
    }

}
```

Course: Soft Design



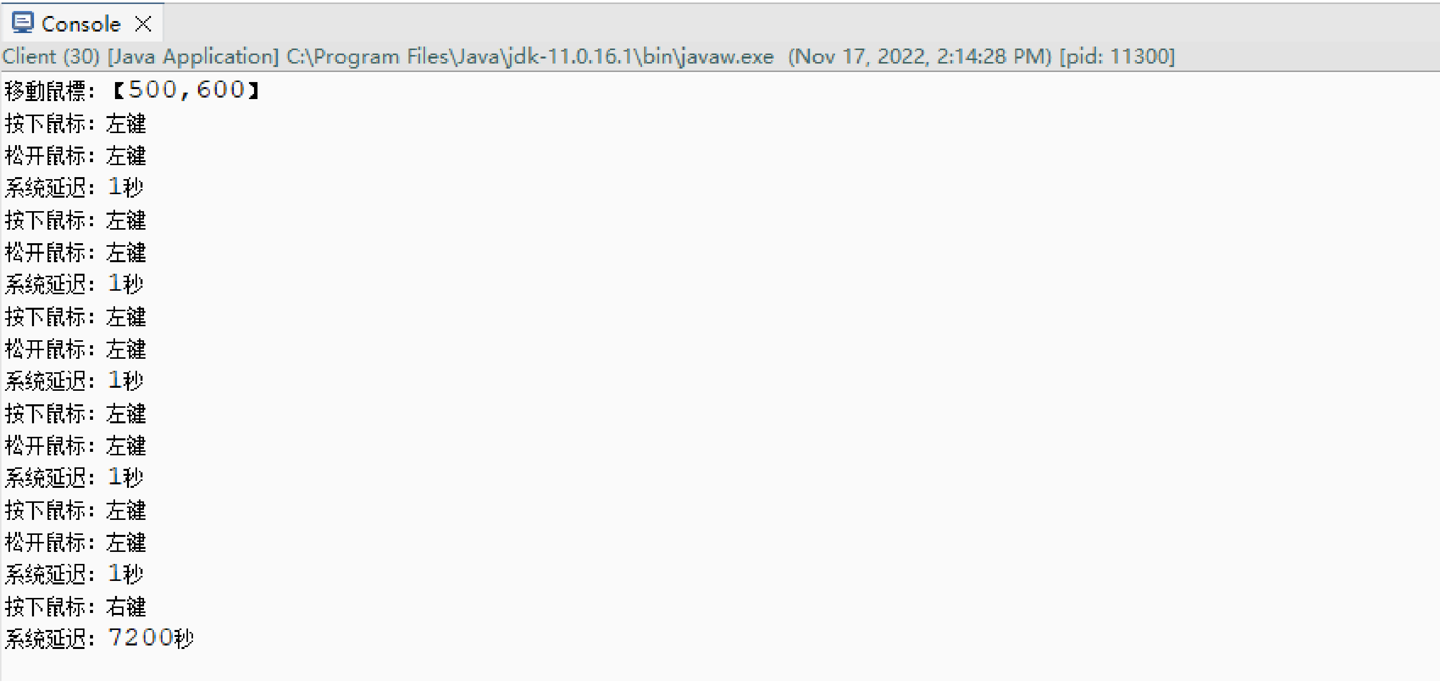
```
public class Repetition implements Expression{
    private int loopCount;
    private Expression loopBodySequence;
    public Repetition(Expression loopBodySequence,int loopCount) {
        this.loopBodySequence = loopBodySequence;
        this.loopCount = loopCount;
    }
    @Override
    public void interpret() {
        // TODO Auto-generated method stub
        while(loopCount > 0) {
            loopBodySequence.interpret();
            loopCount--;
        }
    }

}
```

Course: Soft Design



```
public class Client {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        /*
         * BEGIN // 脚本开始
         * MOVE 500,600; // 鼠标指针移动到坐标(500, 600)
         * BEGIN LOOP 5 // 开始循环5次
         * LEFT_CLICK; // 循环体内单击左键
         * DELAY 1; // 每次延迟1秒
         * END; // 循环体结束
         * RIGHT_DOWN; // 按下右键
         * DELAY 7200; // 延迟2小时
         * END; // 脚本结束
         */

        // 构造指令集语义树，实际情况会交给语法分析器(Evaluator or Parser)
        Expression sequence = new Sequence(Arrays.asList(new Move(500, 600),
                new Repetition(
                        new Sequence(Arrays.asList(new LeftKeyClick(), new Delay(1))
                                ),
                        5 // 循环5次
                        ),
                new RightKeyDown(),
                new Delay(7200)
                ));

        sequence.interpret();
    }

}
```

Course: Soft Design



Console

```
Client (30) [Java Application] C:\Program Files\Java\jdk-11.0.16.1\bin\javaw.exe  (Nov 17, 2022, 2:14:28 PM) [pid: 11300]
移動鼠標：【500，600】
按下鼠标：左鍵
松开鼠标：左鍵
系统延迟：1秒
按下鼠标：左鍵
松开鼠标：左鍵
系统延迟：1秒
按下鼠标：左鍵
松开鼠标：左鍵
系统延迟：1秒
按下鼠标：左鍵
松开鼠标：左鍵
系统延迟：1秒
按下鼠标：左鍵
松开鼠标：左鍵
系统延迟：1秒
按下鼠标：右鍵
系统延迟：7200秒
```