实物类面向对象设计

文泰来 老师

扫描二维码关注微信/微博 获取最新面试题及权威解答

微信: ninechapter

知乎专栏：http://zhuanlan.zhihu.com/jiuzhang

微博: http://www.weibo.com/ninechapter

官网: www.jiuzhang.com

扫一扫 不怀孕

课程大纲

- 实物类OOD题型
- 实物类OOD解题思路
- Vending machine
- Coffee maker
- Kindle

实物类OOD题型

- Vending machine
- Jukebox
- CD Player
- Coffee maker
- Kindle

实物类OOD题型

- 频率：中高

实物类OOD题型

- 频率：中高
- 难度：中低

实物类OOD解题技巧

- 考虑对于实物的输入输出

实物类OOD解题技巧

- 考虑对于实物的输入输出

例子：Coffee maker

实物类OOD解题技巧

- 考虑对于实物的输入输出

例子：Coffee maker

实物类OOD解题技巧

- 考虑对于实物的输入输出

例子：Coffee maker

实物类OOD解题技巧

- Design pattern的运用

实物类OOD解题技巧

- Design pattern的运用

State pattern Decorate pattern Factory pattern

Vending Machine

- Can you design a vending machine?

Clarify

- What
- How

Clarify

- What

关键字： Vending machine

Clarify

- What

关键字： Vending machine

Clarify

- What

关键字： Vending machine

Clarify

- What

关键字： Vending machine, Payment, Item

Clarify

- 关键字： Vending machine

Clarify

- 关键字： Vending machine

厂家，重量，颜色...

Clarify

- 关键字： Vending machine

厂家，重量，颜色...

VendingMachine
- String manufacture
+ String getManufacture()

Clarify

- 关键字： Vending machine

大小： Vending machine的大小是否有限制？

Clarify

- 关键字： Item

Clarify

- 关键字： Item

What items does this vending machine sell?

Clarify

- 关键字： Item

What items does this vending machine sell?

Naïve design approach: each item matches a class

Clarify

- 关键字： Item

What to do when an item sold out?

Clarify

- 关键字： Item

What to do when an item sold out?

Design: Might need to support refill use case

Clarify

- 关键字： Payment

Clarify

- 关键字： Payment

What are the supported payment methods?

Clarify

- Payment

- Coin

- Paper money

- Credit card

Clarify

- Payment

- Coin/Paper money : 知道当前收了多少钱，找零

- Credit card: 直接当前Item的价格

Clarify

- 对于本题：

- 假设Vending machine的大小没有限制
- 假设目前只卖三种产品：Coke, Sprite和Mountain Dew
- 假设目前只接受硬币

Clarify

- How

Clarify

- How

Clarify

- How to select item to purchase?

Design: selectItem(?)

Clarify

- 对于本题：

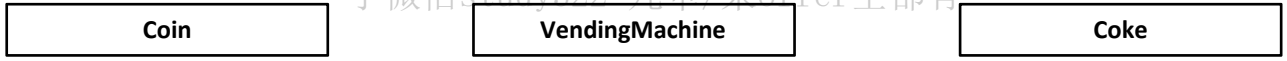
- 假设输入一个input代表一种Item (e.g. A1 -> Coke)

Core Object

VendingMachine

Core Object

Core Object

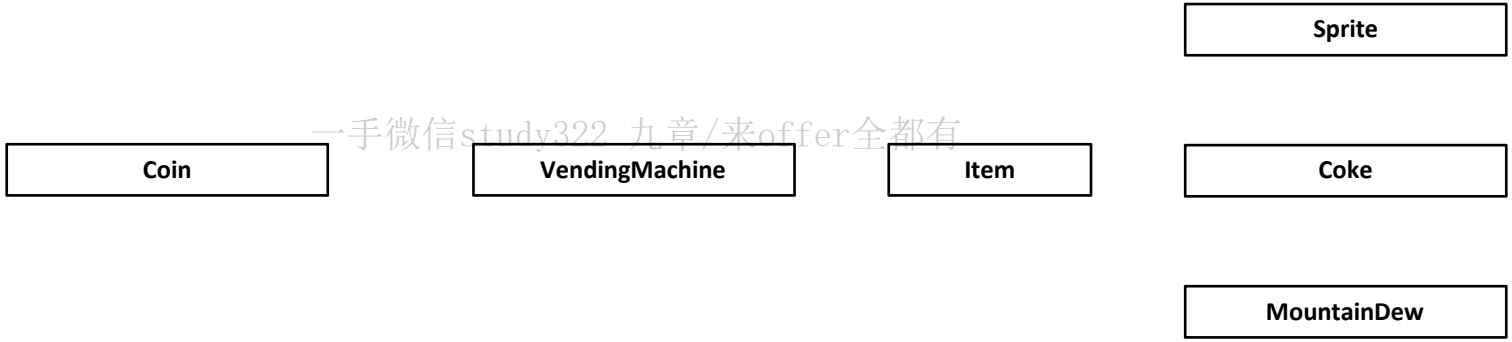


Core Object

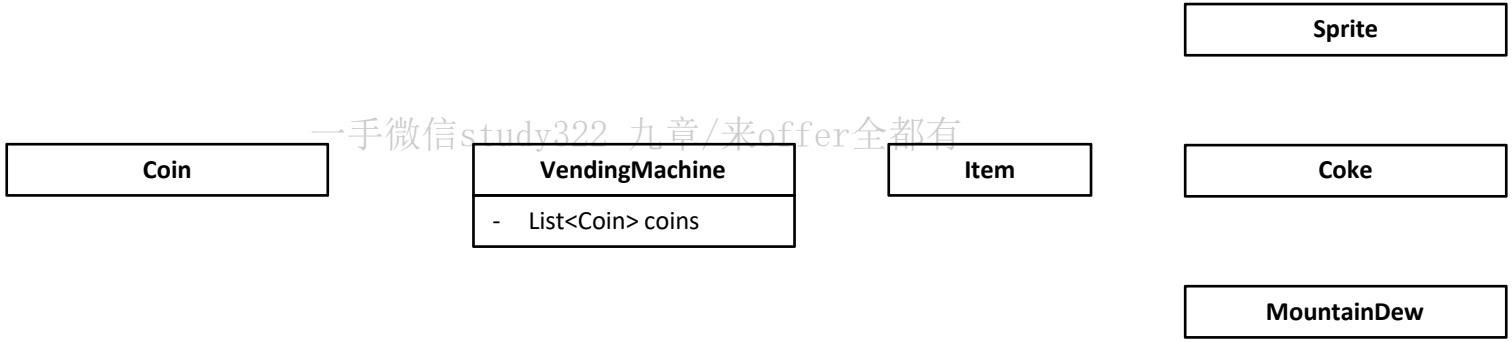
Core Object

Core Object

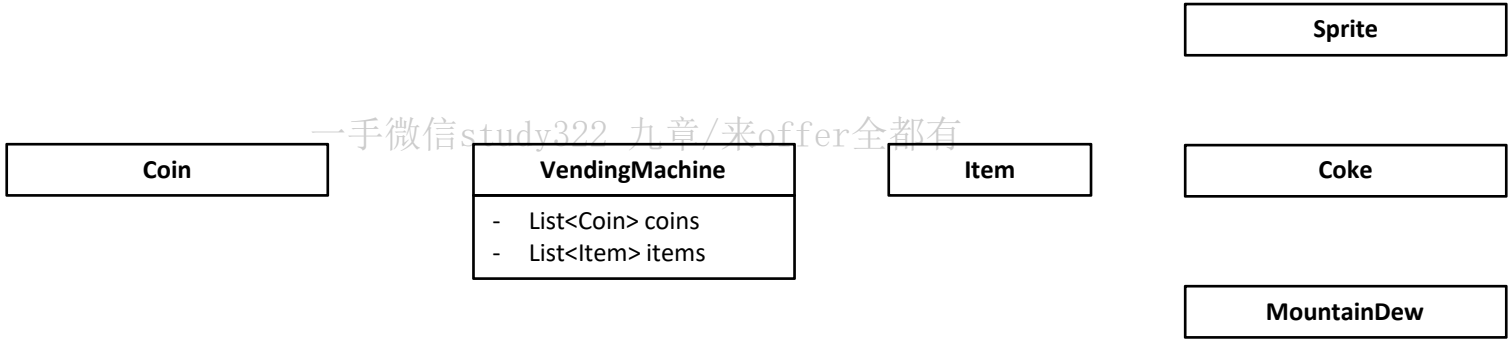
Core Object



Core Object



Core Object



Use cases

Vending machine

Use cases

Vending machine:

- Select item

Use cases

Vending machine:

- Select item
- Insert coin

Use cases

Vending machine:

- Select item
- Insert coin
- Execute transaction

Use cases

Vending machine:

- Select item
- Insert coin
- Execute transaction
- Cancel transaction

Use cases

Vending machine:

- Select item
- Insert coin
- Execute transaction
- Cancel transaction
- Refill items

Classes

Classes

Use case: Select item

- Vending machine takes an external input, shows the price of that item

Classes

Classes

Classes

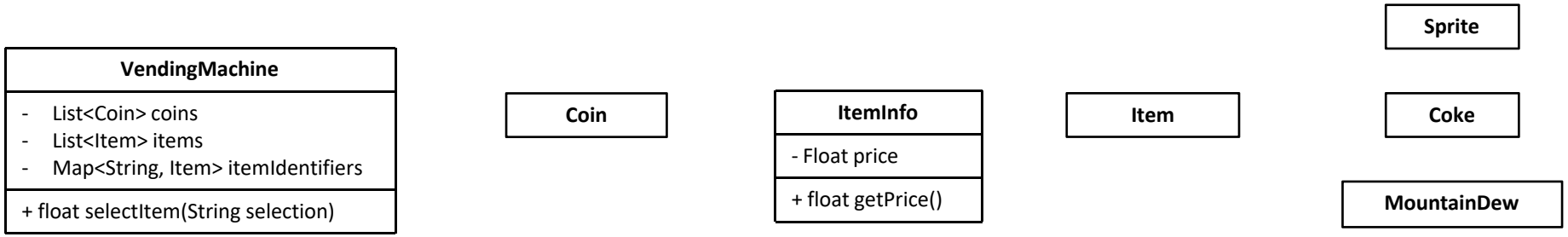
Challenge

“A1” -> 

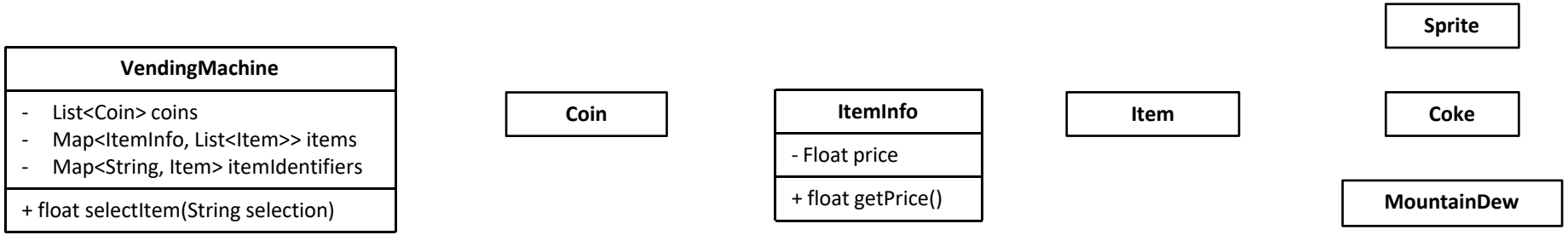
Vending machine:

Classes

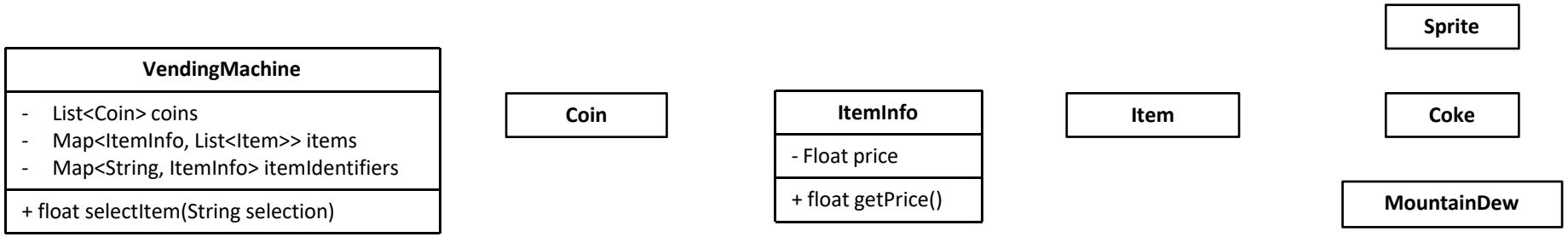
Classes



Classes



Classes

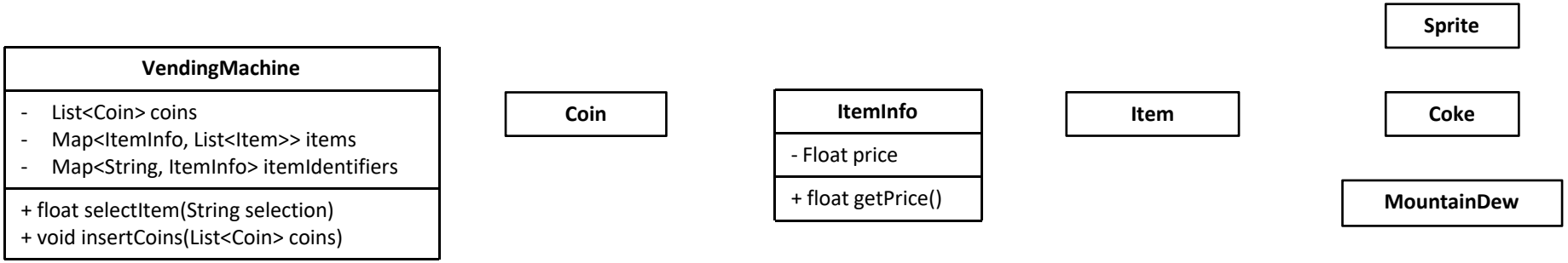


Classes

Use case: Insert coin

- Insert a list of coins into vending machine

Classes



Classes

Use case: Execute transaction

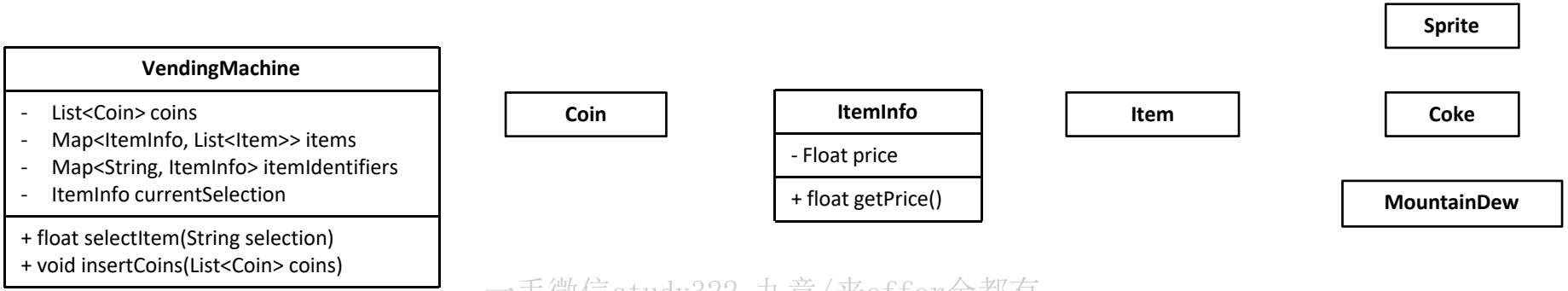
- Get the current selected item
- Compare the item price and inserted coins
- If not enough money, throw an exception
- Else, return the item purchased
- Refund if any

Classes

Use case: Execute transaction

- Get the current selected item

Classes

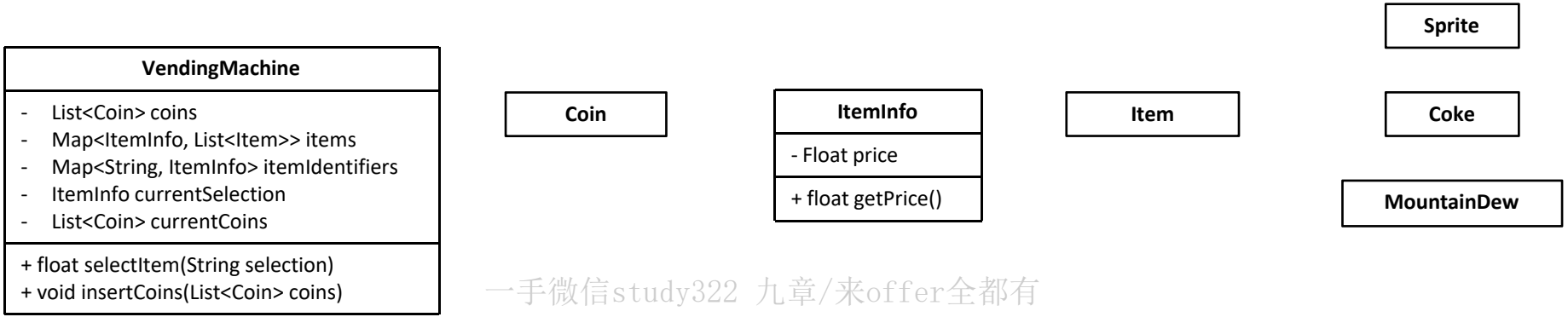


Classes

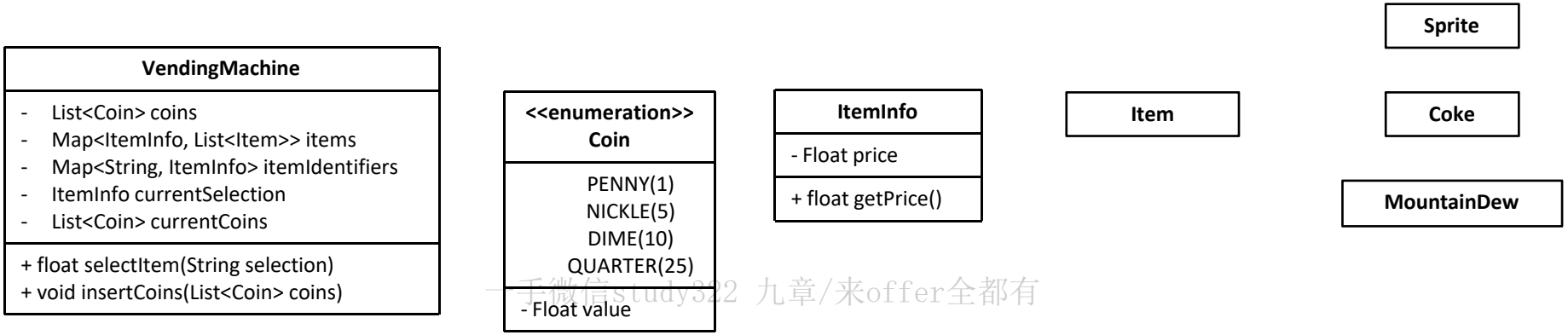
Use case: Execute transaction

- Get the current selected item
- Compare the item price and inserted coins

Classes



Classes

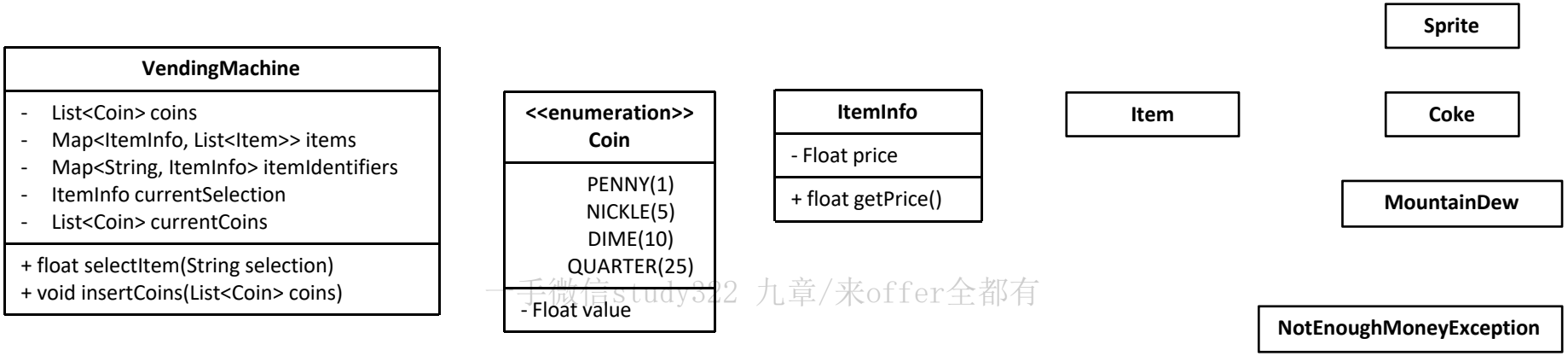


Classes

Use case: Execute transaction

- Get the current selected item
- Compare the item price and inserted coins
- If not enough money, throw an exception

Classes

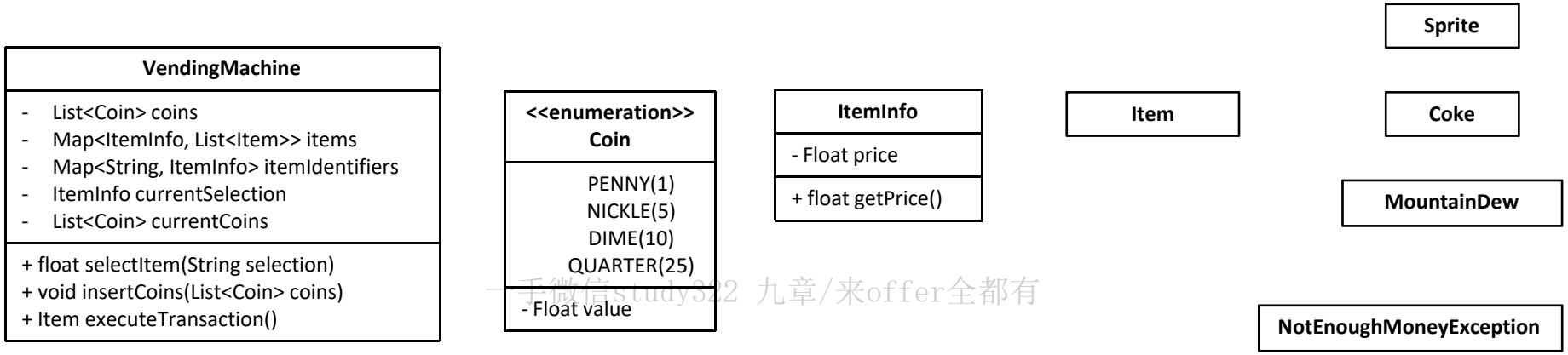


Classes

Use case: Execute transaction

- Get the current selected item
- Compare the item price and inserted coins
- If not enough money, throw an exception
- Else, return the item purchased

Classes

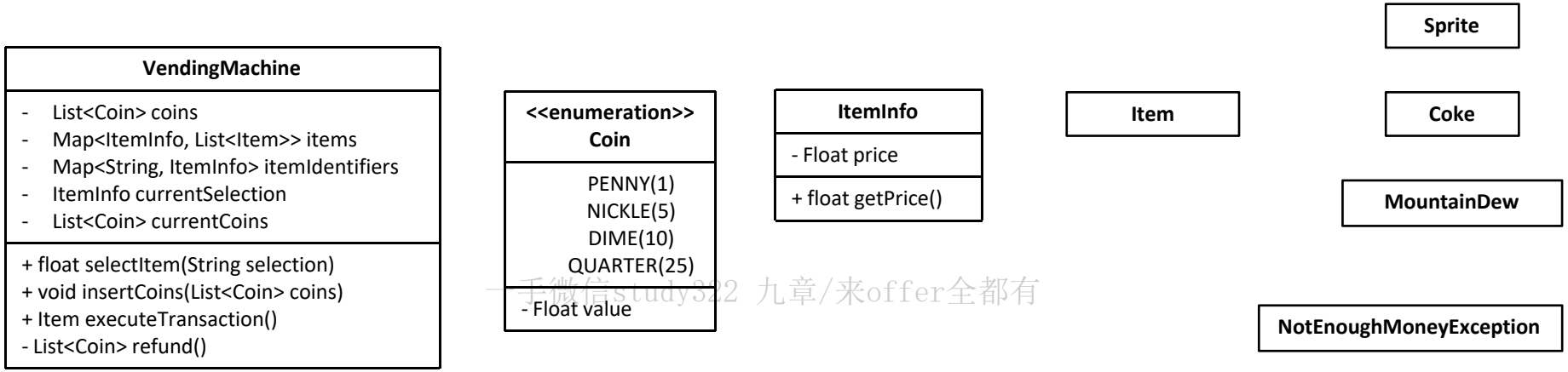


Classes

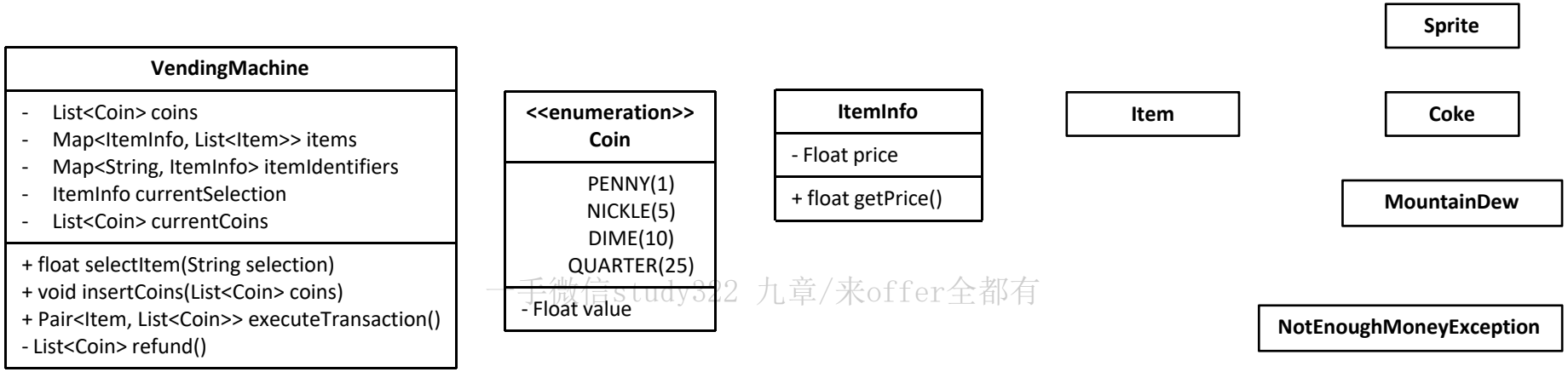
Use case: Execute transaction

- Get the current selected item
- Compare the item price and inserted coins
- If not enough money, throw an exception
- Else, return the item purchased
- Refund if any

Classes



Classes



Coin change

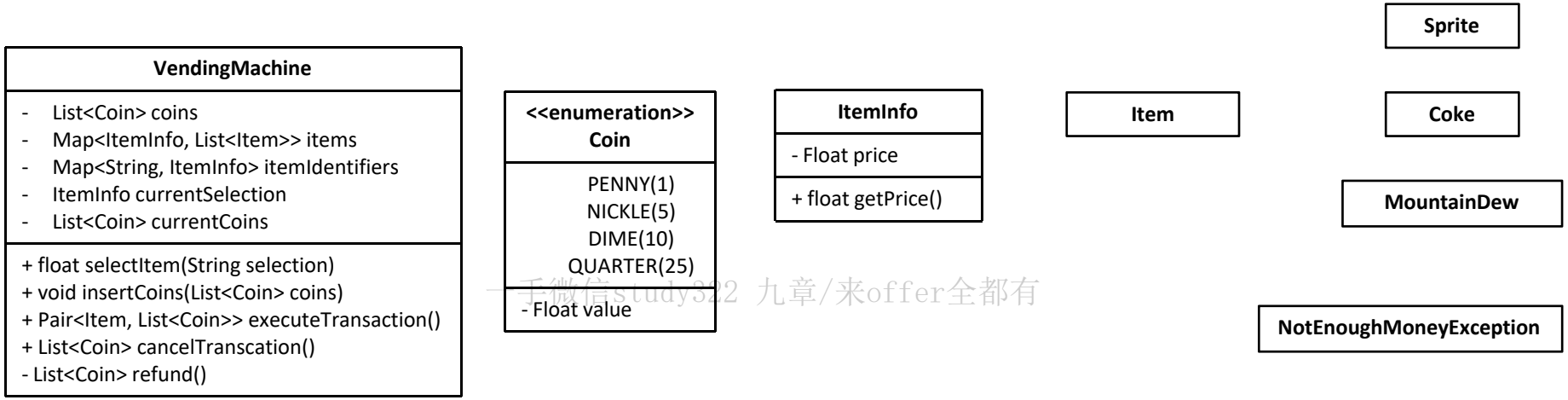
- http://www.cnblogs.com/grandyang/p/4840713.html

Classes

Use case: Cancel transaction

- Return the current coins that has been inserted

Classes

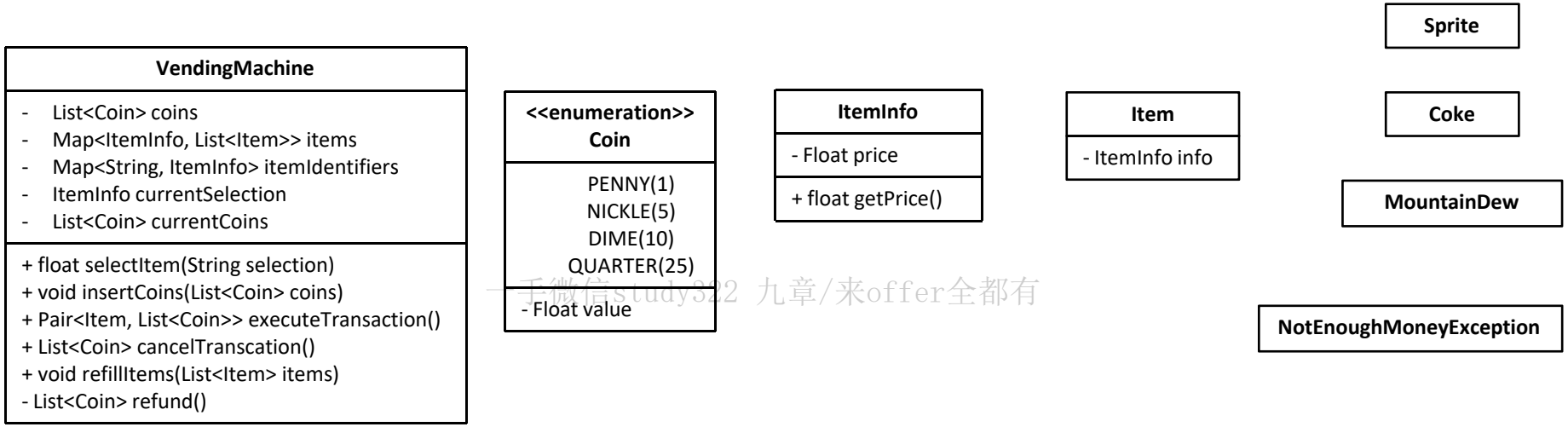


Classes

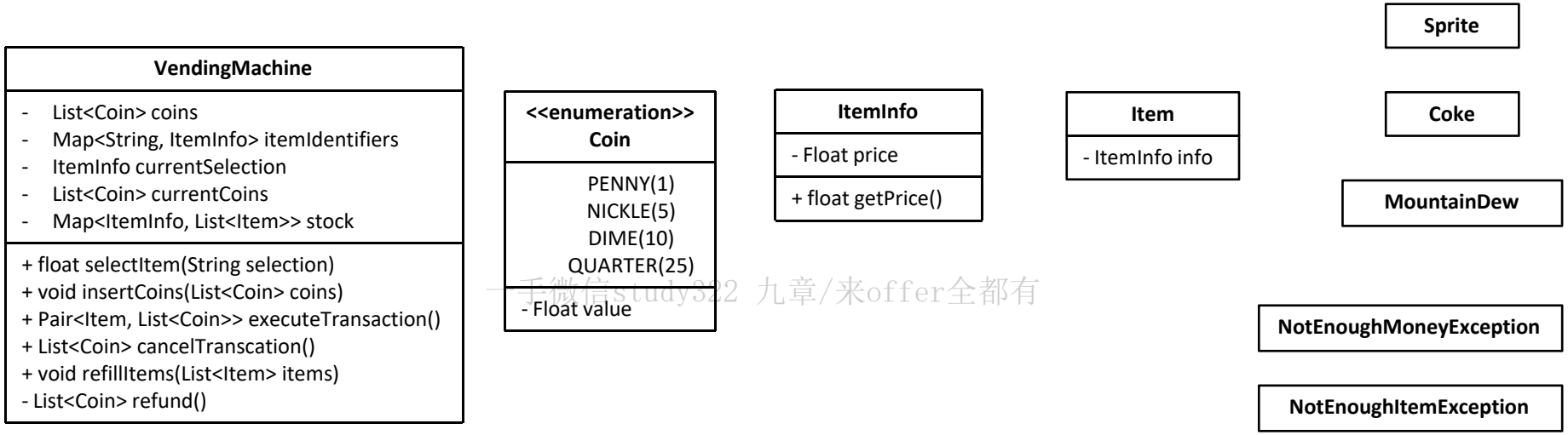
Use case: Refill items

- Refill items on top of current stock

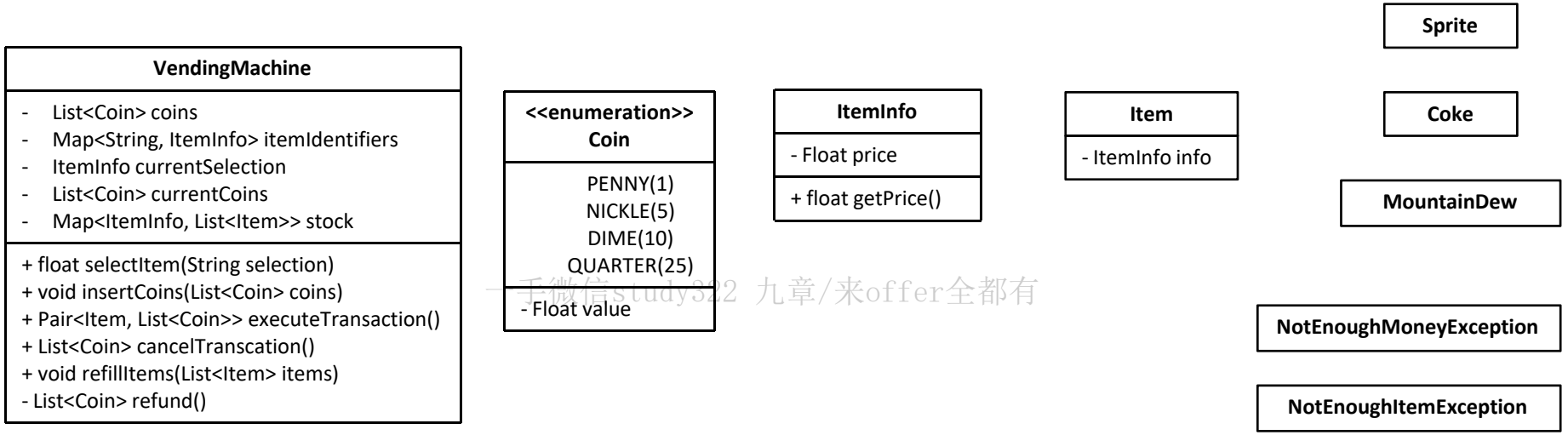
Classes



Classes



Classes - Final view



Good practice

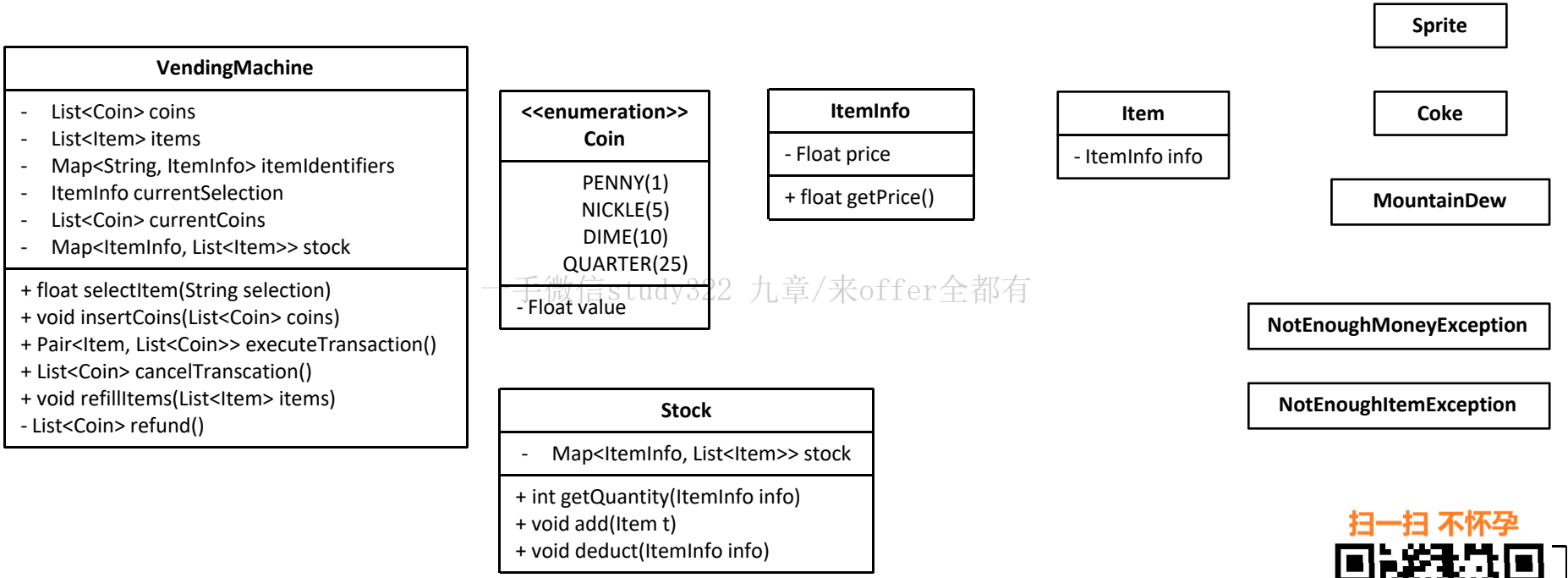
```
stock = new HashMap<ItemInfo, List<Item>>();
```

```
public void refillItem(List<Item> items)
{
    for(Item item : items)
    {
        ItemInfo info = item.getInfo();
        List<Item> itemsInStock = stock.get(info);
        itemsInStock.add(item);
        stock.put(info, itemsInStock);
    }
}
```

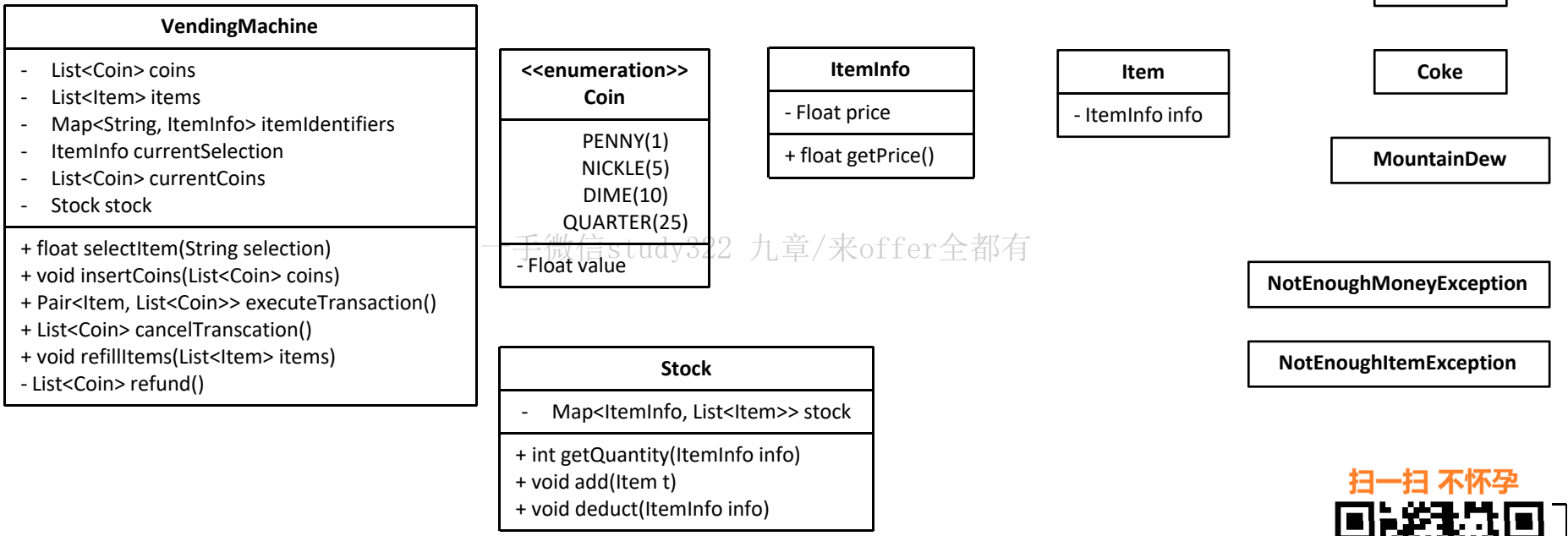

Good practice

```
class Stock
{
    private HashMap<ItemInfo, List<Item>> stock;

    public void add(Item item)
    {
        ItemInfo info = item.getInfo();
        List<Item> itemsInStock = stock.get(info);
        itemsInStock.add(item);
        stock.put(info, itemsInStock);
    }
}

stock = new Stock();

public void refillItem(List<Item> items)
{
    for(Item item : items)
    {
        stock.add(item);
    }
}
```


Good practice



Good practice



Challenge

- For these use cases:

- Select item
- Insert coin
- Execute transaction
- Cancel transaction

Challenge

- For these use cases:

- Select item
- Insert coin
- Execute transaction
- Cancel transaction

What will happen if some item has been selected?

Challenge

- For these use cases:

- Select item : throws a selections has already been made
- Insert coin
- Execute transaction
- Cancel transaction

What will happen if some item has been selected?

Challenge

- For these use cases:

- Select item : throws a selections has already been made
- Insert coin : update current inserted value
- Execute transaction
- Cancel transaction

What will happen if some item has been selected?

Challenge

- For these use cases:

- Select item : throws a selections has already been made
- Insert coin : update current inserted value
- Execute transaction : Get selected item if money is enough
- Cancel transaction

What will happen if some item has been selected?

Challenge

- For these use cases:

- Select item : throws a selections has already been made
- Insert coin : update current inserted value
- Execute transaction : Get selected item if money is enough
- Cancel transaction : return money and empty selected item

What will happen if some item has been selected?

Challenge

- For these use cases:

- Select item
- Insert coin
- Execute transaction
- Cancel transaction

What will happen if none item has been selected?

Challenge

- For these use cases:

- Select item : make a selection
- Insert coin
- Execute transaction
- Cancel transaction

What will happen if none item has been selected?

Challenge

- For these use cases:

- Select item : make a selection
- Insert coin : throws to ask user make a selection first
- Execute transaction
- Cancel transaction

What will happen if none item has been selected?

Challenge

- For these use cases:

- Select item : make a selection
- Insert coin : throws to ask user make a selection first
- Execute transaction : throws to ask user to make a selection first
- Cancel transaction

What will happen if none item has been selected?

Challenge

- For these use cases:

- Select item : make a selection
- Insert coin : throws to ask user make a selection first
- Execute transaction : throws to ask user to make a selection first
- Cancel transaction : maybe not doing anything or throw

What will happen if none item has been selected?

Challenge

- Insert coin

```
public void insertCoin(List<Coin> coins)
{
    if(selectedItem == null)
    {
        throw new Exception("You need to make a selection first");
    }
    else if(selectedItem != null)
    {
        currentCoins.add(coins);
    }
}
```

Challenge

- 我们刚刚考虑了HAS_SELECTION 和 NO_SELECTION 的情况

Challenge

- 我们刚刚考虑了HAS_SELECTION 和 NO_SELECTION 的情况

- 那么对于：

- COINS_INSERTED
- VENDING

应该怎么办？

Challenge

```
public void insertCoin(List<Coin> coins)
{
    if(selectedItem == null)
    {
        throw new Exception("You need to make a selection first");
    }
    else if(selectedItem != null)
    {
        currentCoins.add(coins);
    }
    else if(VENDING)
    {
        throw new Exception("Be patient, item is coming out, dont need to pay once more");
    }
    ...
}
```


Challenge

- State Design Pattern

Challenge

- State Design Pattern

States:

- HAS_SELECTION
- NO_SELECTION
- COINS_INSERTED
-VENDING

Challenge

- State Design Pattern

State related actions:

- select item
- insert coin
- execute transaction
- cancel transaction

Classes

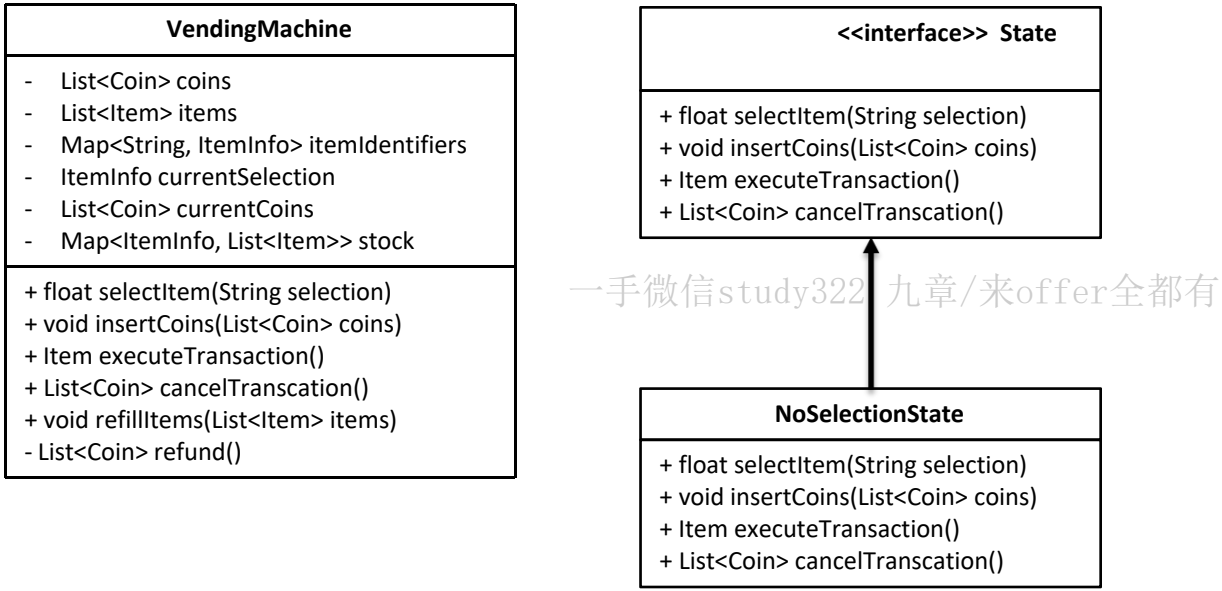
VendingMachine
- List<Coin> coins - List<Item> items - Map<String, ItemInfo> itemIdentifiers - ItemInfo currentSelection - List<Coin> currentCoins - Map<ItemInfo, List<Item>> stock
+ float selectItem(String selection) + void insertCoins(List<Coin> coins) + Item executeTransaction() + List<Coin> cancelTranscation() + void refillItems(List<Item> items) - List<Coin> refund()

Classes

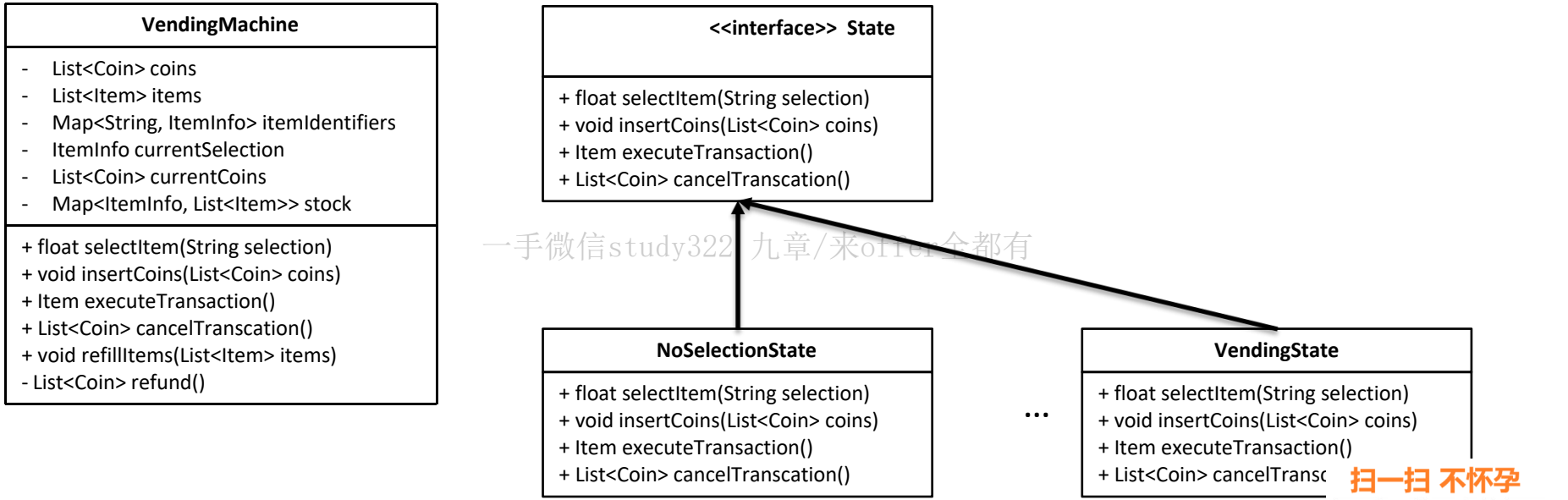
VendingMachine
- List<Coin> coins - List<Item> items - Map<String, ItemInfo> itemIdentifiers - ItemInfo currentSelection - List<Coin> currentCoins - Map<ItemInfo, List<Item>> stock
+ float selectItem(String selection) + void insertCoins(List<Coin> coins) + Item executeTransaction() + List<Coin> cancelTranscation() + void refillItems(List<Item> items) - List<Coin> refund()

<<interface>> State
+ float selectItem(String selection) + void insertCoins(List<Coin> coins) + Item executeTransaction() + List<Coin> cancelTranscation()

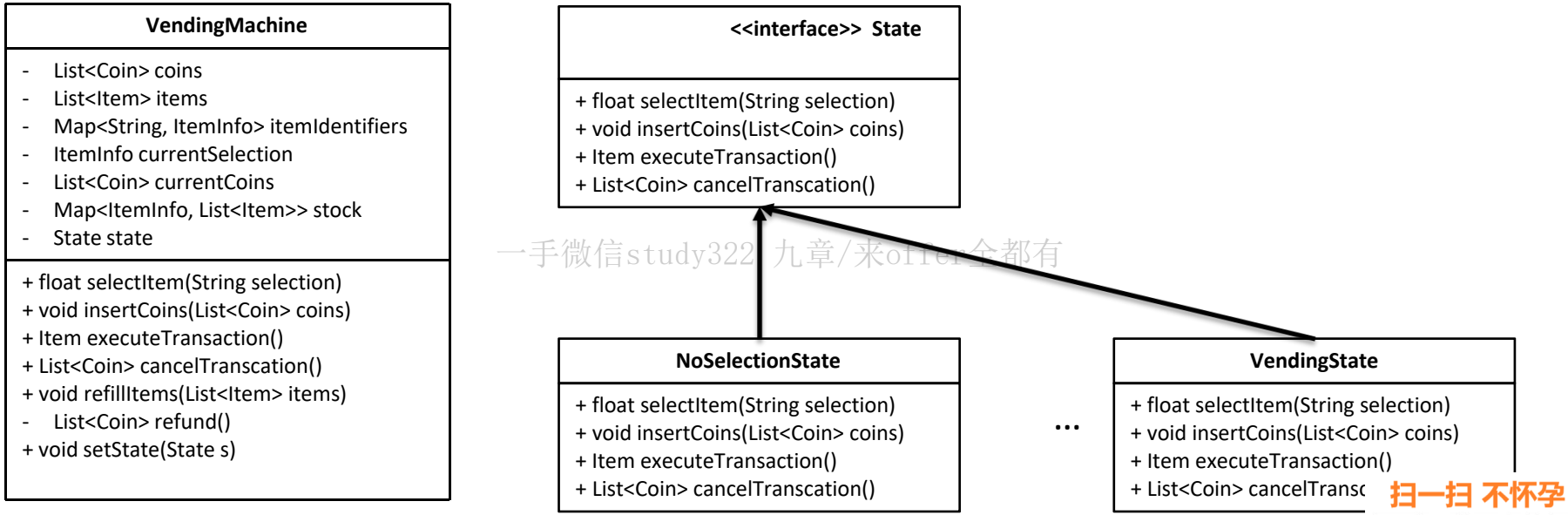
Classes



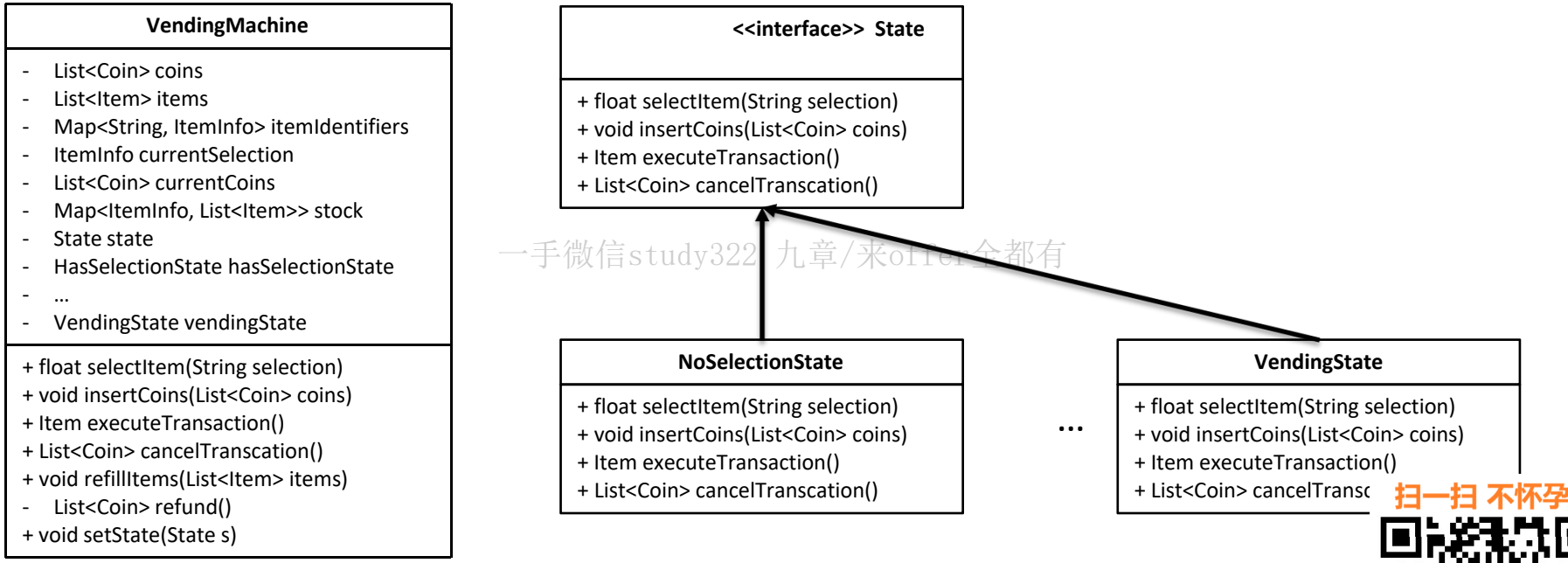
Classes



Classes



Classes



Vending machine

```
public interface State {
    public void selectItem(String selection);
    public void insertMoney(int value);
    public void executeTransaction();
    public int cancelTransaction();
}
```


Vending machine

```java
public class VendingMachine {

    private AbstractState state;
    private NoSelectionState noSelectionState;
    private HasSelectionState hasSelectionState;
    private InsertedMoneyState insertedMoneyState;


    public VendingMachine()
    {

        noSelectionState = new NoSelectionState(this);
        hasSelectionState = new HasSelectionState(this);
        insertedMoneyState = new InsertedMoneyState(this);
        state = noSelectionState;

    }

    public void changeToNoSelectionState()
    {
        state = noSelectionState;
    }

    public void changeToHasSelectionState()
    {
        state = hasSelectionState;
    }

    public void changeToInsertedMoneyState()
    {
        state = insertedMoneyState;
    }

    public void selectItem(String selection)
    {
        state.selectItem(selection);
    }

    public void addMoney(int value)
    {
        state.insertMoney(value);
    }

    public void executeTransaction()
    {
        state.executeTransaction();
    }

    public int cancelTransaction()
    {
        return state.cancelTransaction();
    }
}
```

```java
public class NoSelectionState implements AbstractState{

    VendingMachine vendingMachine;

    public NoSelectionState(VendingMachine vendingMachine) {
        this.vendingMachine = vendingMachine;
    }

    @Override
    public void selectItem(String selection) {
        // TODO Auto-generated method stub
        vendingMachine.setSelectedItem(selection);
        vendingMachine.changeToHasSelectionState();
    }

    @Override
    public void insertMoney(int value) {
        // TODO Auto-generated method stub
        System.out.println("Please make a selection first");
    }

    @Override
    public void executeTransaction() {
        // TODO Auto-generated method stub
        System.out.println("Please make a selection first");
    }

    @Override
    public int cancelTransaction() {
        // TODO Auto-generated method stub
        System.out.println("Please make a selec
        return 0;
    }

}
```


Real life object

- 难度不大

- 从Input / Output 考虑
- 继承关系
- 考虑Exception
- Design pattern if possible

Coffee maker

Clarify

- What

关键字： Coffee maker

Clarify

- What

关键字： Coffee maker

Clarify

Clarify

- Input

Clarify

- Output

Clarify

- 对于本题

Input: Coffee packs

Output: Expresso

Clarify

- How

Clarify

- How

Clarify

- What are the functions that out coffee maker supports?

Clarify

- 对于本题：

- Brew

Core object

CoffeeMaker

Core object

CoffeePack

CoffeeMaker

Core object

CoffeePack

CoffeeMaker

Expresso

Use cases

- Coffee maker

- Brew

Classes

CoffeePack

CoffeeMaker

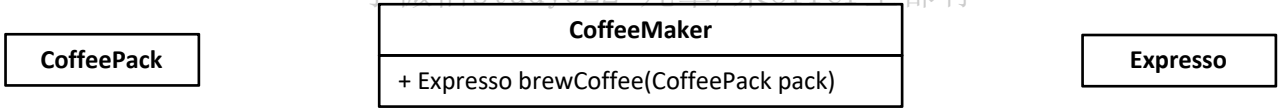
Expresso

Classes

- Use case: Brew

Coffee machine expected to use a coffee pack to get expresso coffee

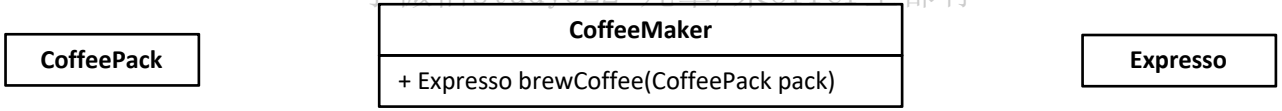
Classes



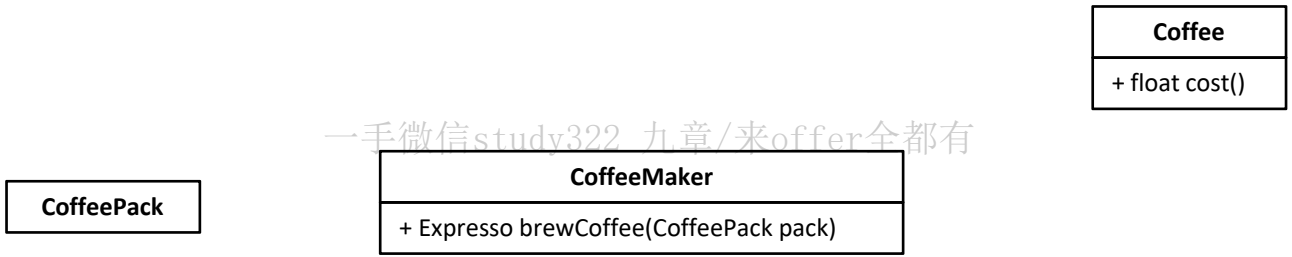
Challenge

如果需要能制作出多种咖啡(价格不同)，需要怎么做？

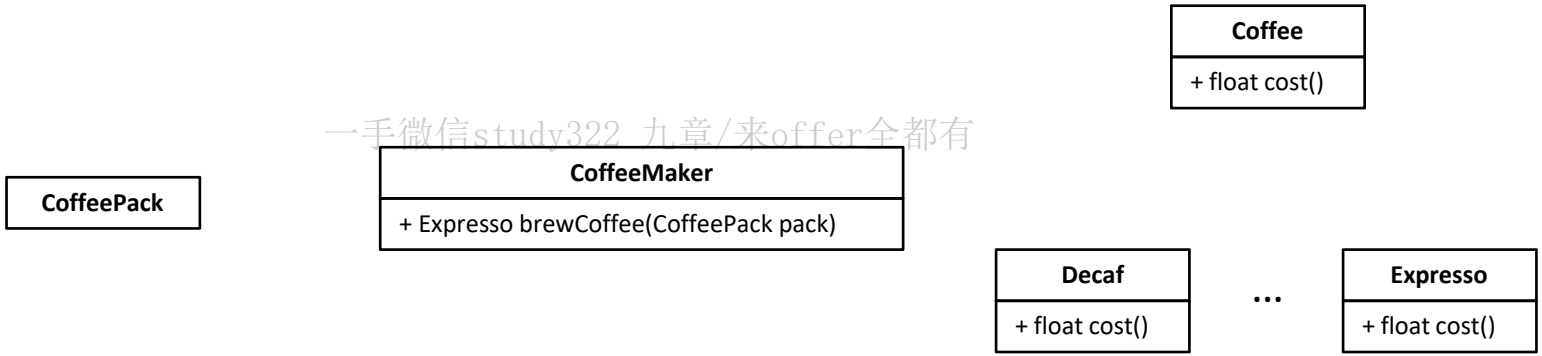
Classes



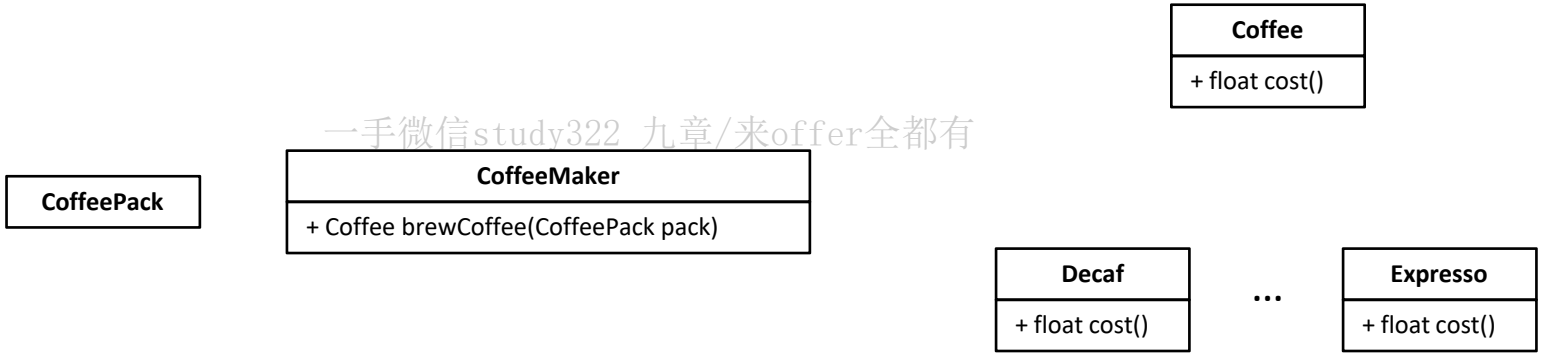
继承



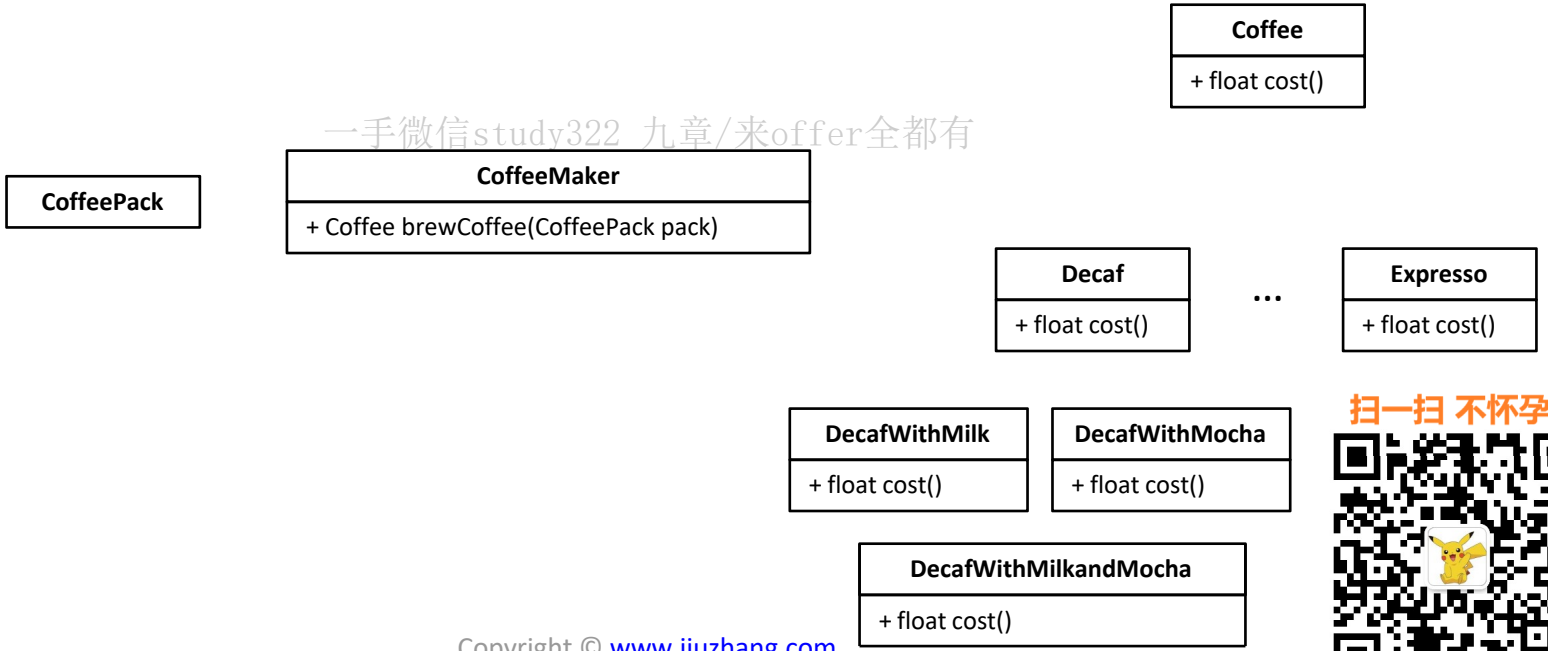
继承



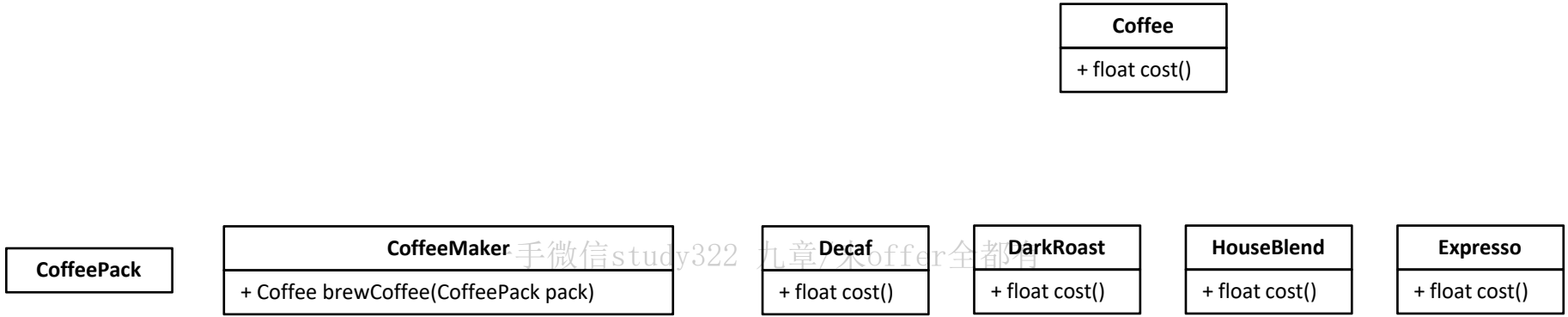
继承



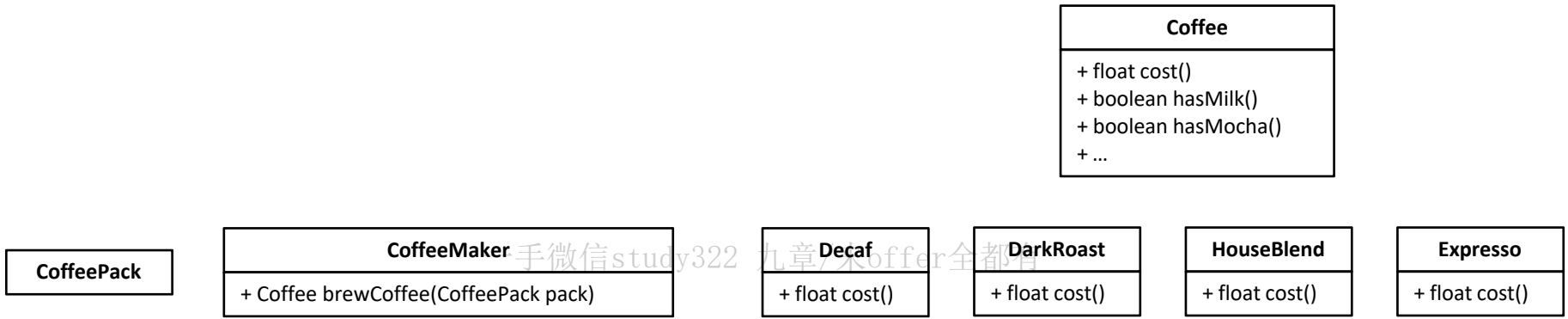
继承



另一种继承



另一种继承



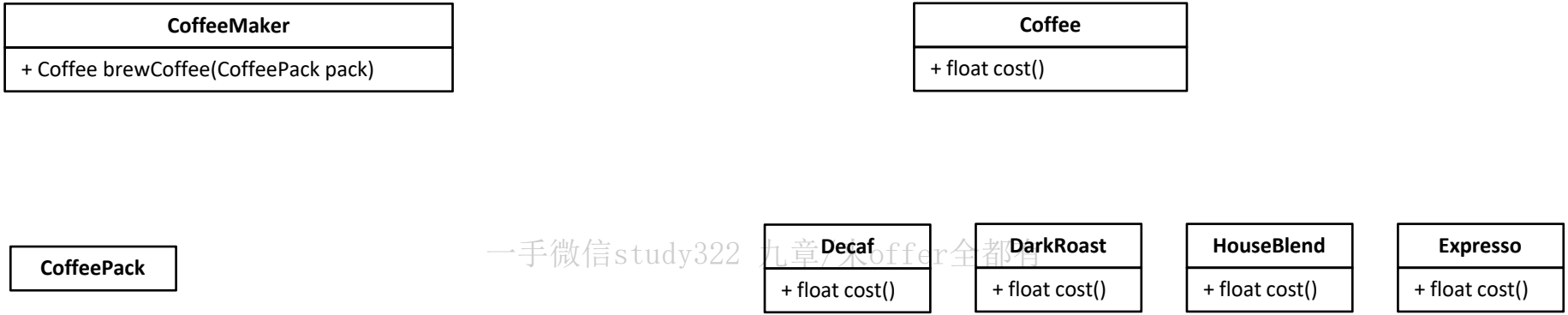
另一种继承

```
public float cost()
{
    if(hasMilk())
    {
        cost += 0.5;
    }

    if(hasMocha())
    {
        cost += 0.5;
    }

    if(hasSoy())
    {
        cost += 0.5;
    }

    ...

    return cost;
}
```

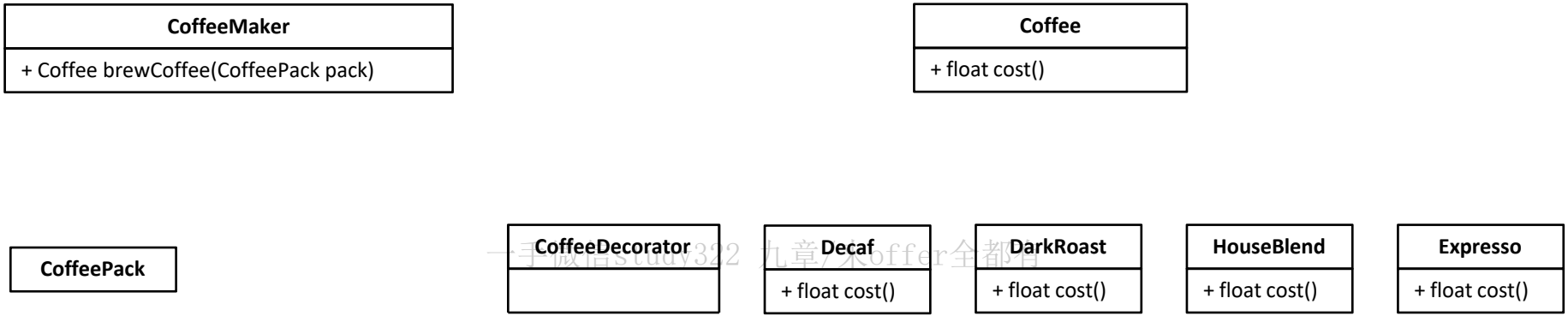

Decorator Design Pattern

- Decorator pattern allows a user to add new functionality to an existing object without altering its structure. This type of design pattern comes under structural pattern as this pattern acts as a wrapper to existing class.

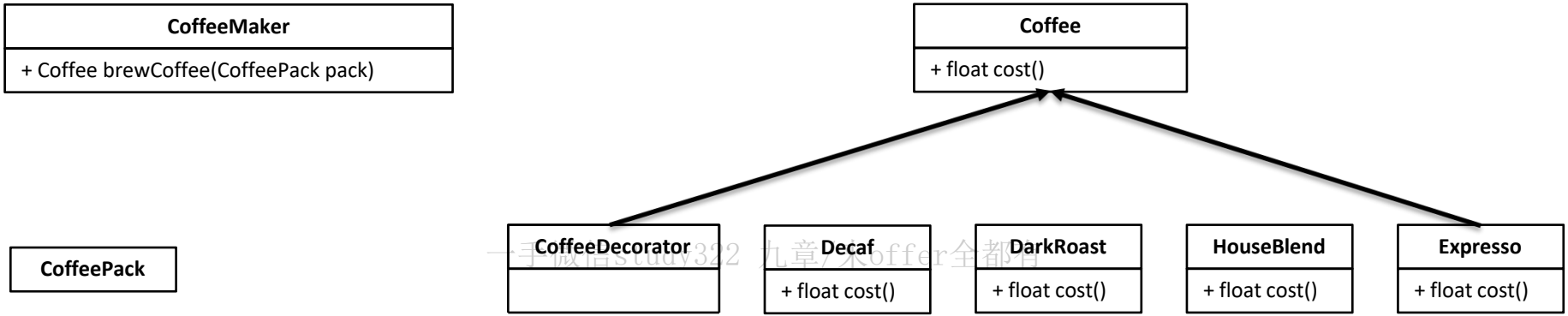
Decorator



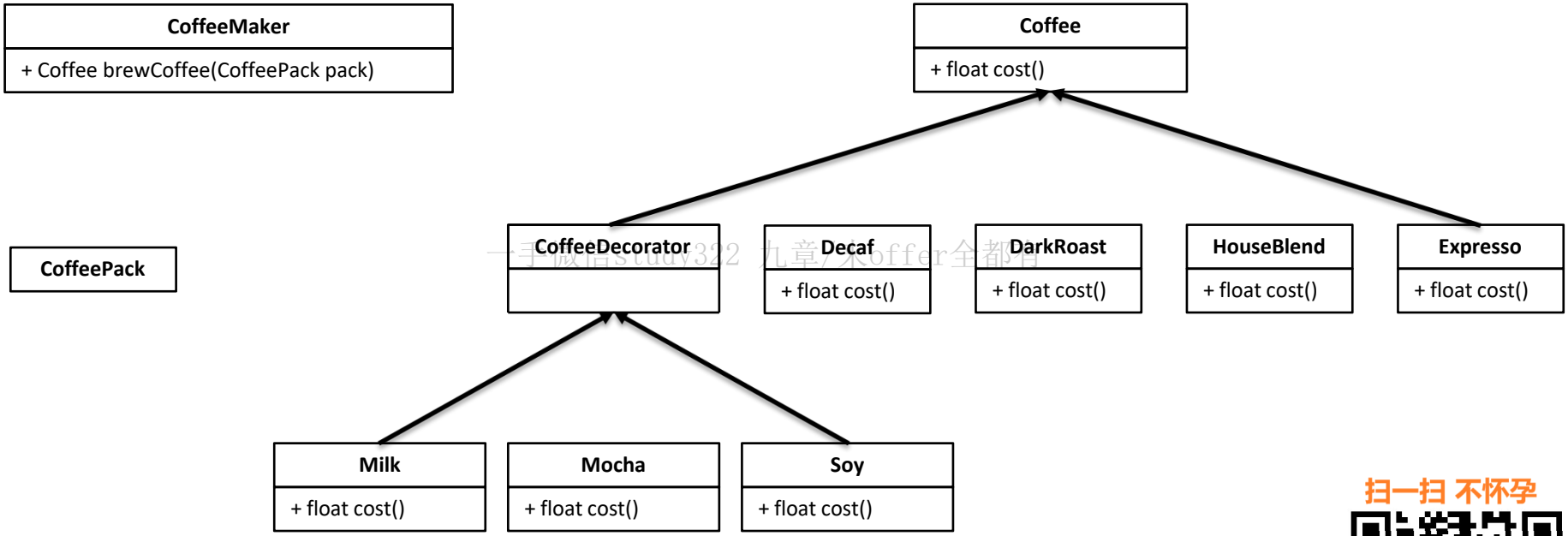
Decorator



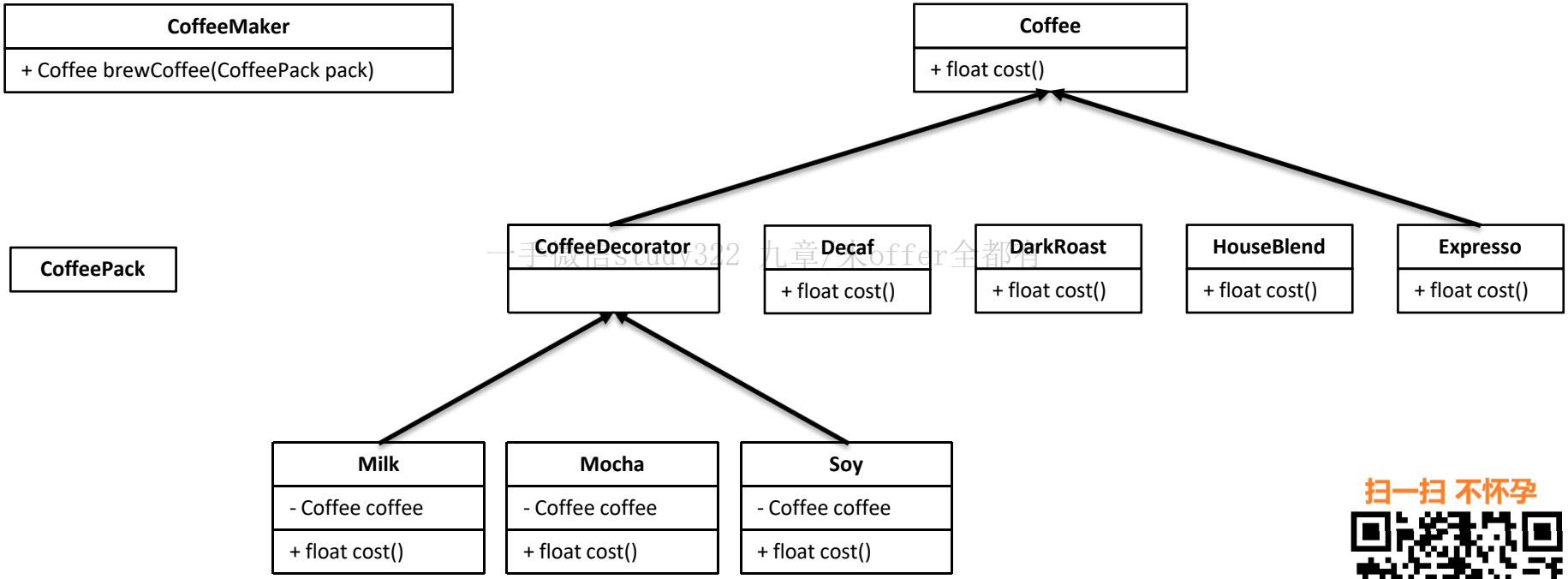
Decorator



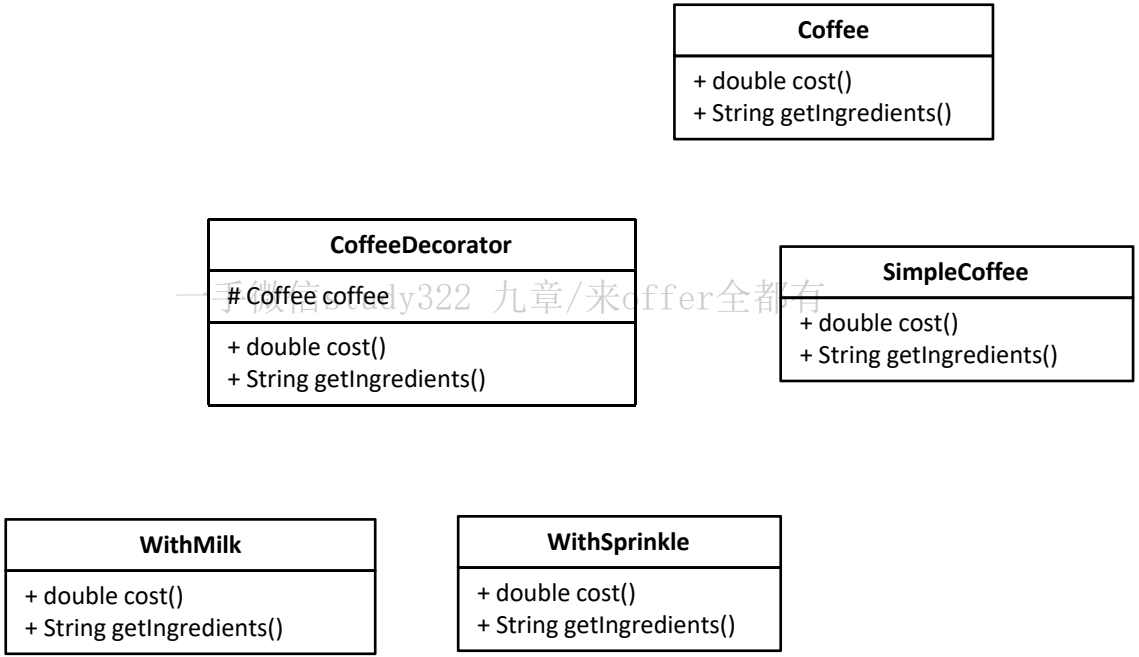
Decorator



Decorator



Decorator



Decorator

```java
// The interface Coffee defines the functionality of Coffee implemented by decorator
public interface Coffee {
    public double getCost(); // Returns the cost of the coffee
    public String getIngredients(); // Returns the ingredients of the coffee
}

// Extension of a simple coffee without any extra ingredients
public class SimpleCoffee implements Coffee {
    @Override
    public double getCost() {
        return 1;
    }

    @Override
    public String getIngredients() {
        return "Coffee";
    }
}
```

Decorator

```
// Abstract decorator class - note that it implements Coffee interface
public abstract class CoffeeDecorator implements Coffee {
    protected final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    public double getCost() { // Implementing methods of the interface
        return decoratedCoffee.getCost();
    }

    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}
```


Decorator

```
// Decorator WithMilk mixes milk into coffee.
// Note it extends CoffeeDecorator.
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

    public double getCost() { // Overriding methods defined in the abstract superclass
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ", Milk";
    }
}

// Decorator WithSprinkles mixes sprinkles onto coffee.
// Note it extends CoffeeDecorator.
class WithSprinkles extends CoffeeDecorator {
    public WithSprinkles(Coffee c) {
        super(c);
    }

    public double getCost() {
        return super.getCost() + 0.2;
    }

    public String getIngredients() {
        return super.getIngredients() + ", Sprinkles";
    }
}
```


Decorator

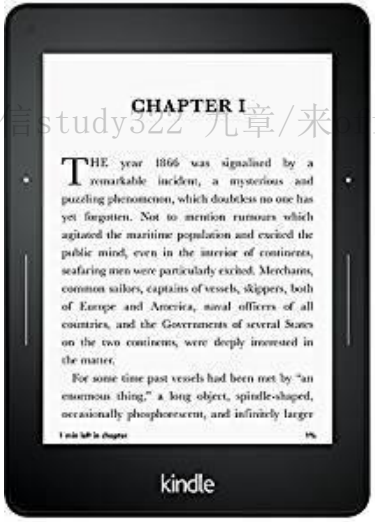
```
public class Main {
    public static void printInfo(Coffee c) {
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());
    }

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
}
```


The output of this program is given below:

```
Cost: 1.0; Ingredients: Coffee
Cost: 1.5; Ingredients: Coffee, Milk
Cost: 1.7; Ingredients: Coffee, Milk, Sprinkles
```


Kindle

Can you design Kindle?



Clarify

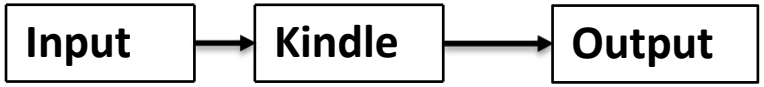
- What

关键字： Kindle

Clarify

- What

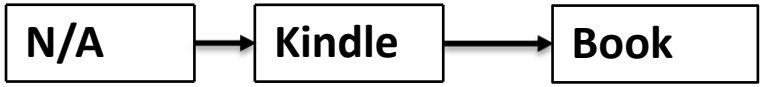
关键字： Kindle



Clarify

- What

关键字： Kindle



Clarify

- What

关键字： Kindle, Book

Clarify

关键字： Kindle

Clarify

关键字： Kindle

需不需要设计不同版本？

Clarify

关键字： Kindle

需不需要设计不同版本？

- Design: get price

Clarify

关键字： Kindle

需不需要设计不同版本？

- Design: get price
- Design: Memory difference

Clarify

关键字： Book

Clarify

关键字： Book

- 支持哪些格式的电子书？

Clarify

- 对于本题：

- 不需要考虑不同的版本
- 不需要考虑内存和书的大小
- 支持pdf, epub 和 mobi三种格式

Clarify

How

Clarify

如何获取电子书？

Clarify

如何获取电子书？

- 是否支持Upload
- 是否支持Download

Clarify

如何获取电子书？

- Upload
- Download

对于付费的电子书，提供哪些支付功能？

Clarify

如何获取电子书？

- Upload
- Download

对于付费的电子书，提供哪些支付功能？

Payment -> Strategy design pattern

Clarify

- 对于本题：支持上传，下载
- 对于本题：不需要考虑付费

Clarify

- Who

- N/A

Core Object

Kindle

Core Object

Kindle

Book

Core Object

Use cases

- Kindle

Use cases

- Kindle

- Upload book

Use cases

- Kindle

- Upload book
- Download book

Use cases

- Kindle

- Upload book
- Download book
- Read book

Use cases

- Kindle

- Upload book
- Download book
- Read book
- Remove book

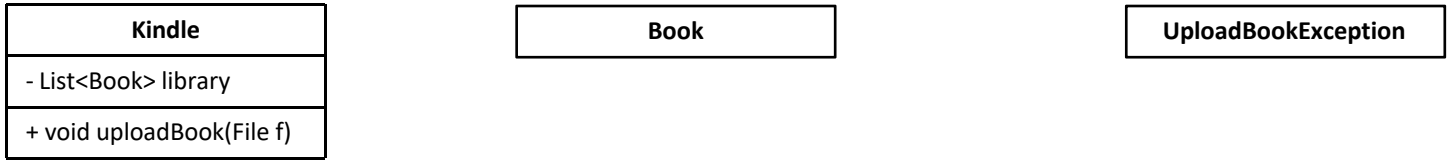
Classes

Upload book

- Upload a file to kindle and store as a book

Classes

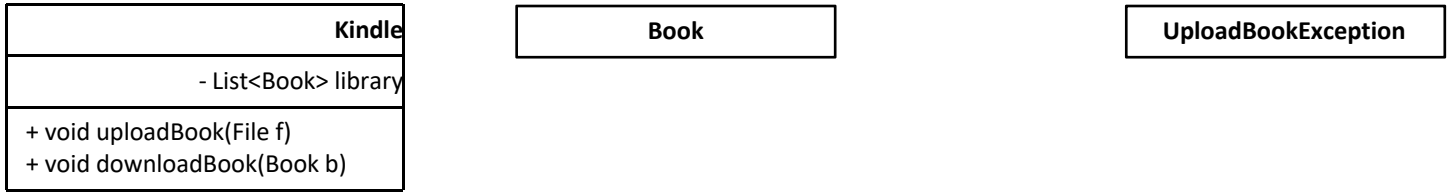
Classes



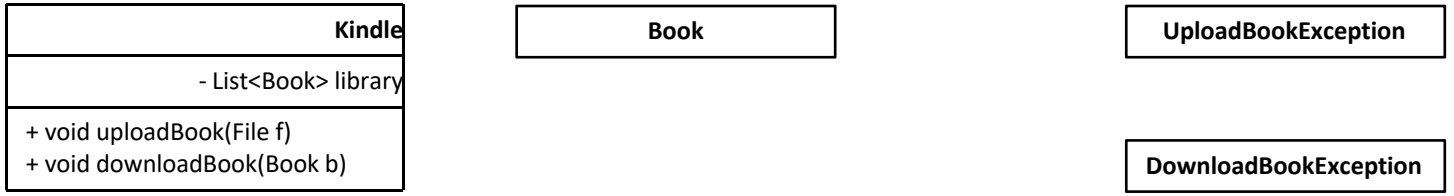
Download book

- Download a book and put in library

Classes



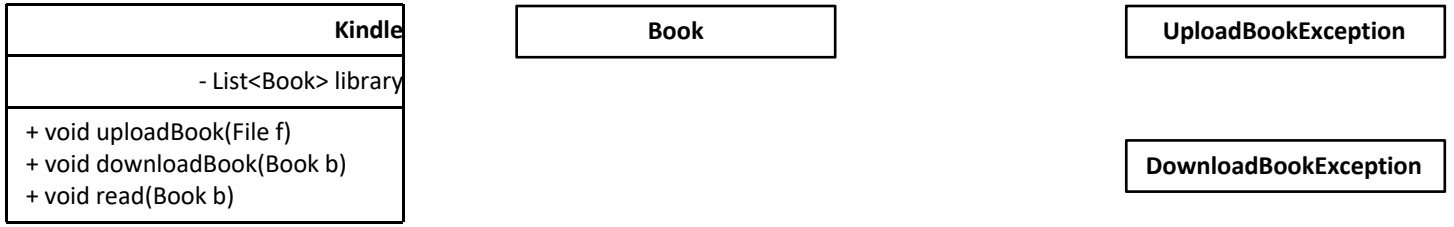
Classes



Read book

- Select a book and display it

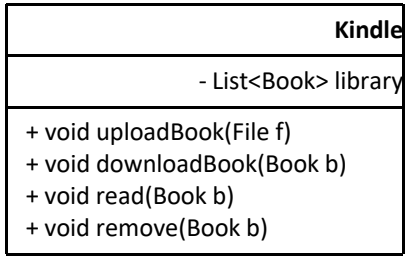
Classes



Remove book

- Remove a book from library

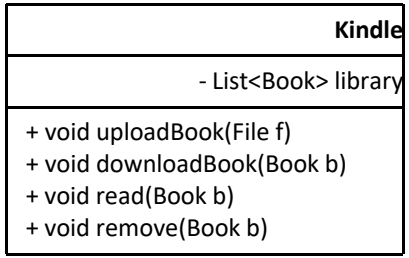
Classes

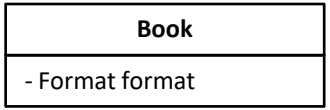


Challenge

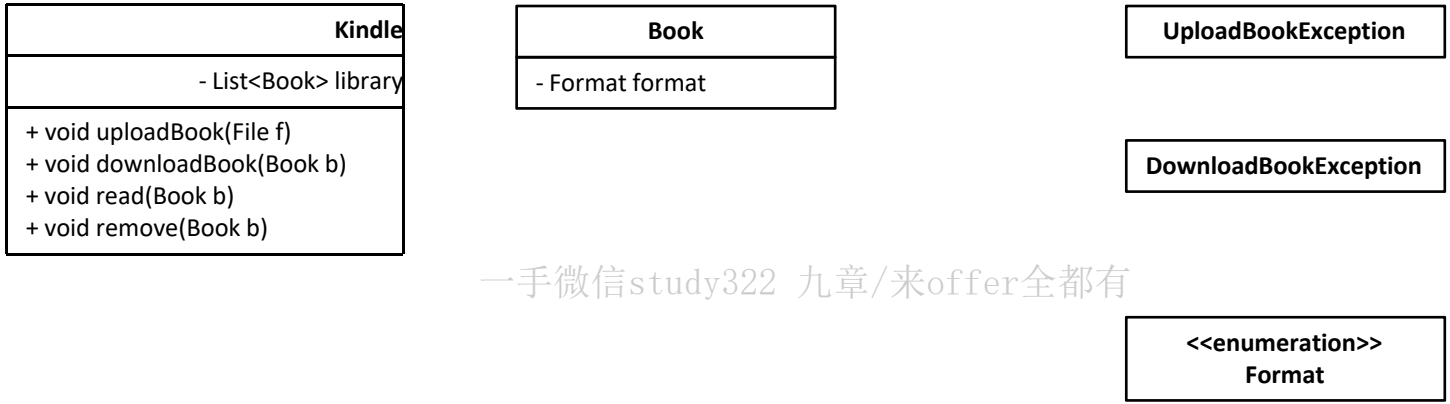
- What about different book format?

Classes

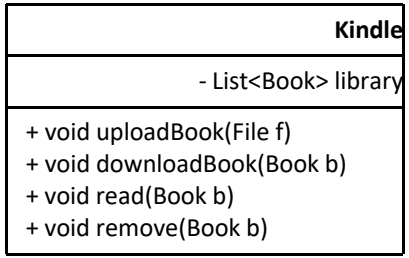


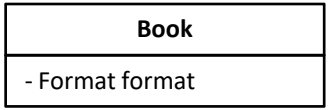


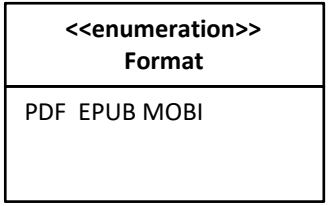
Classes



Classes



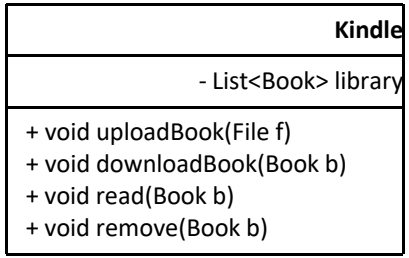


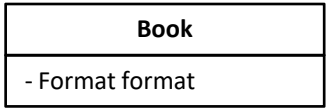


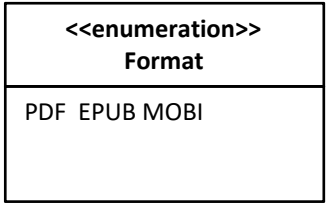
Challenge

- How would read book work?

Classes







Challenge

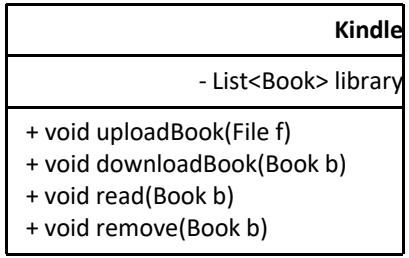
- How would read book work?

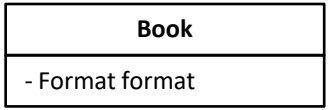
```
public void read(Book book)
{
    if(book.getFormat == Format.PDF)
    {
        PDFReader reader = new PDFReader(book);
        reader.display();
    }
    else if(book.getFormat == Format.MOBI)
    {
        MOBIReader reader = new MOBIReader(book);
        reader.display();
    }
    else if(book.getFormat == Format.EPUB)
    {
        EPUBReader reader = new EPUBReader(book);
        reader.display();
    }
}
```

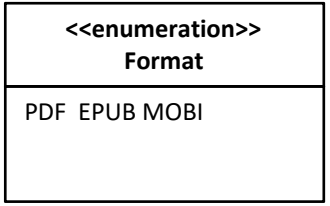

Challenge

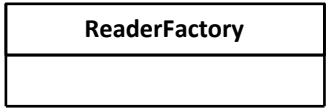
- Solution: Factory design pattern

Factory design pattern

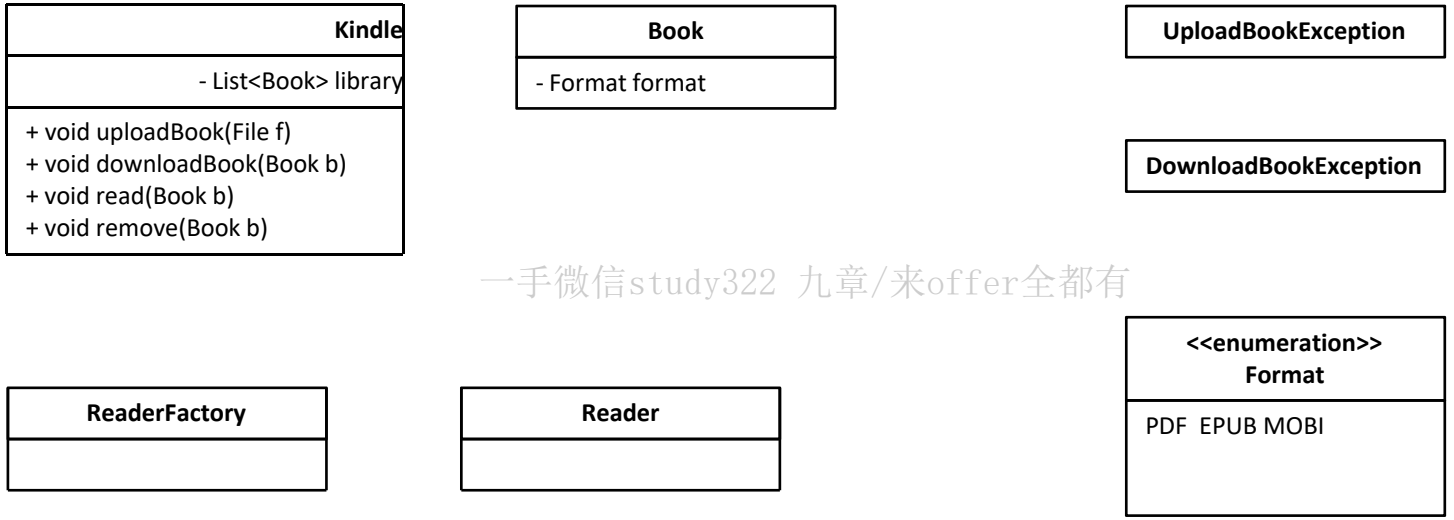




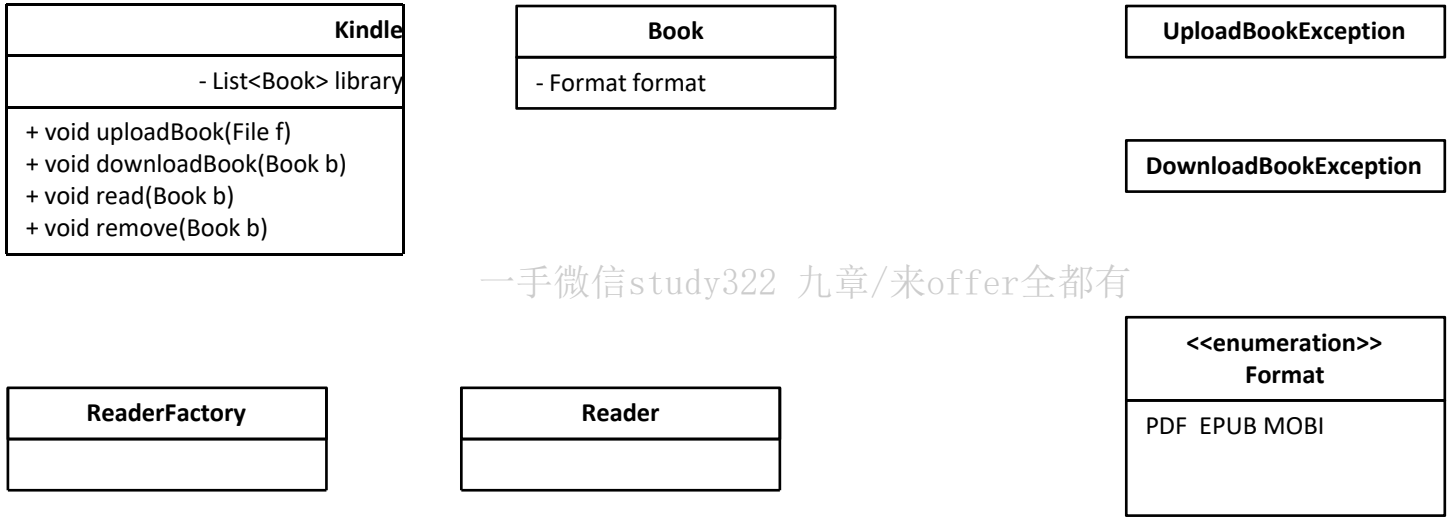




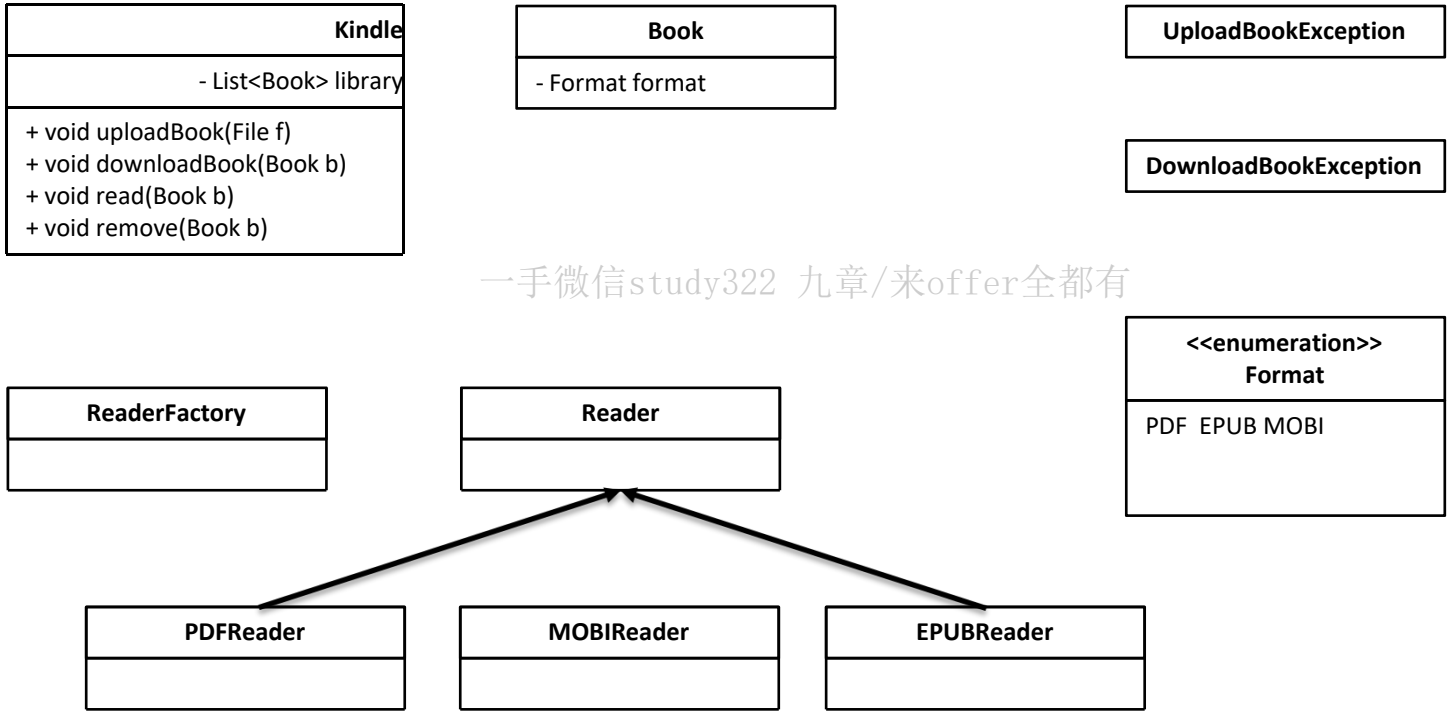
Factory design pattern



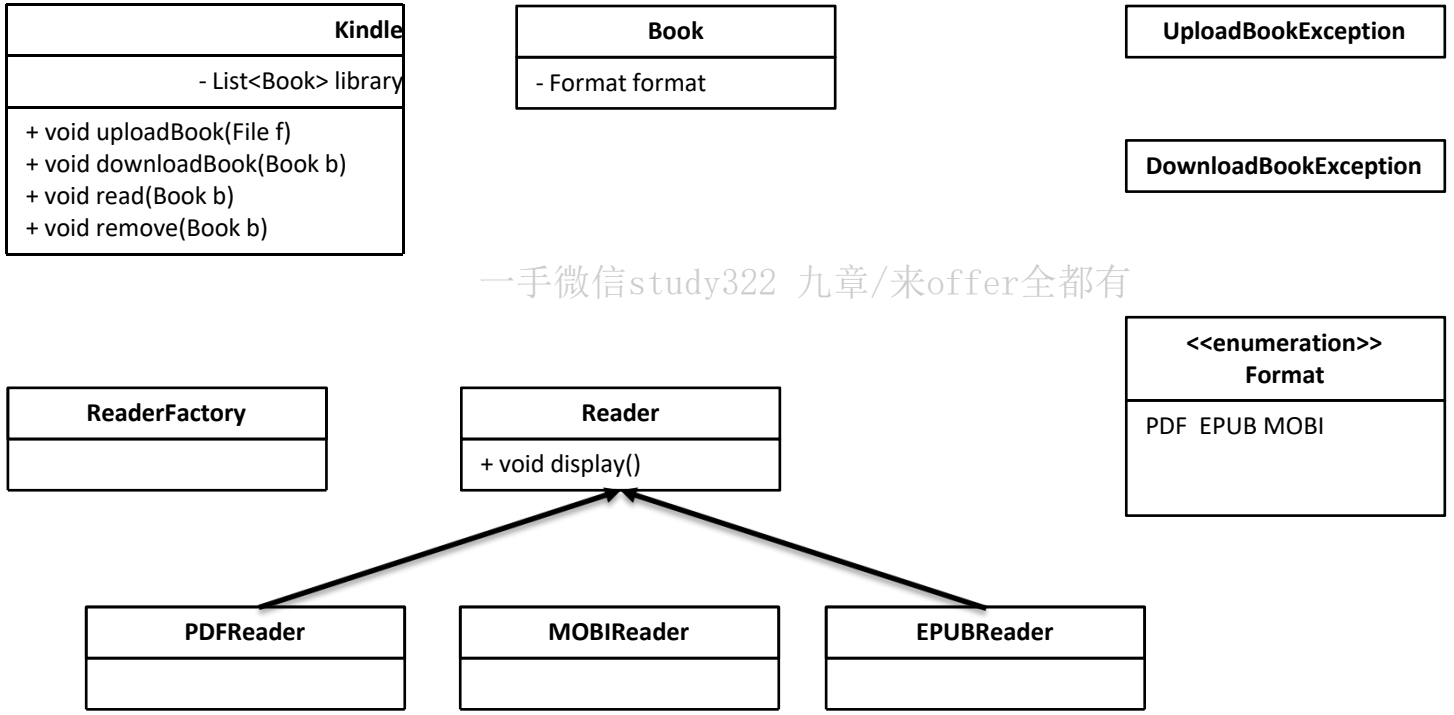
Factory design pattern



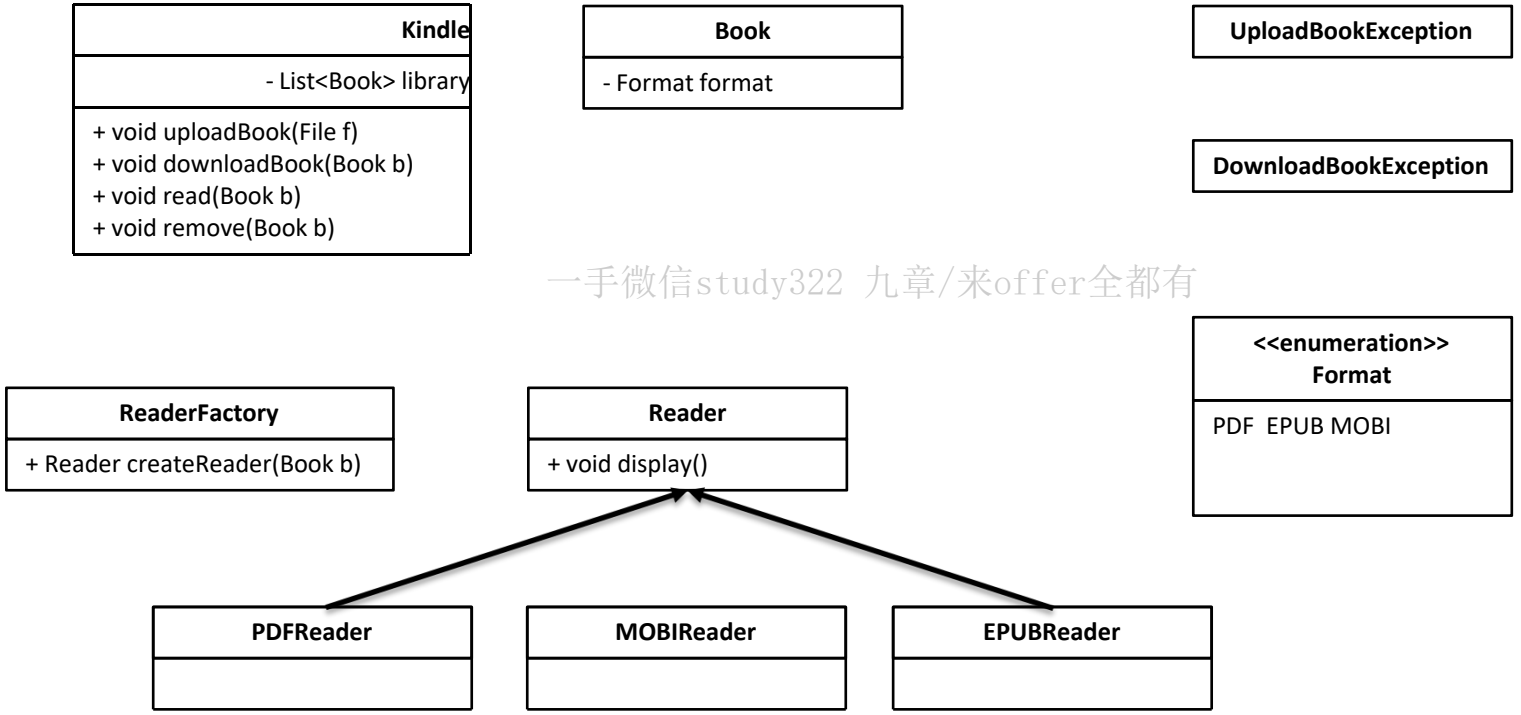
Factory design pattern



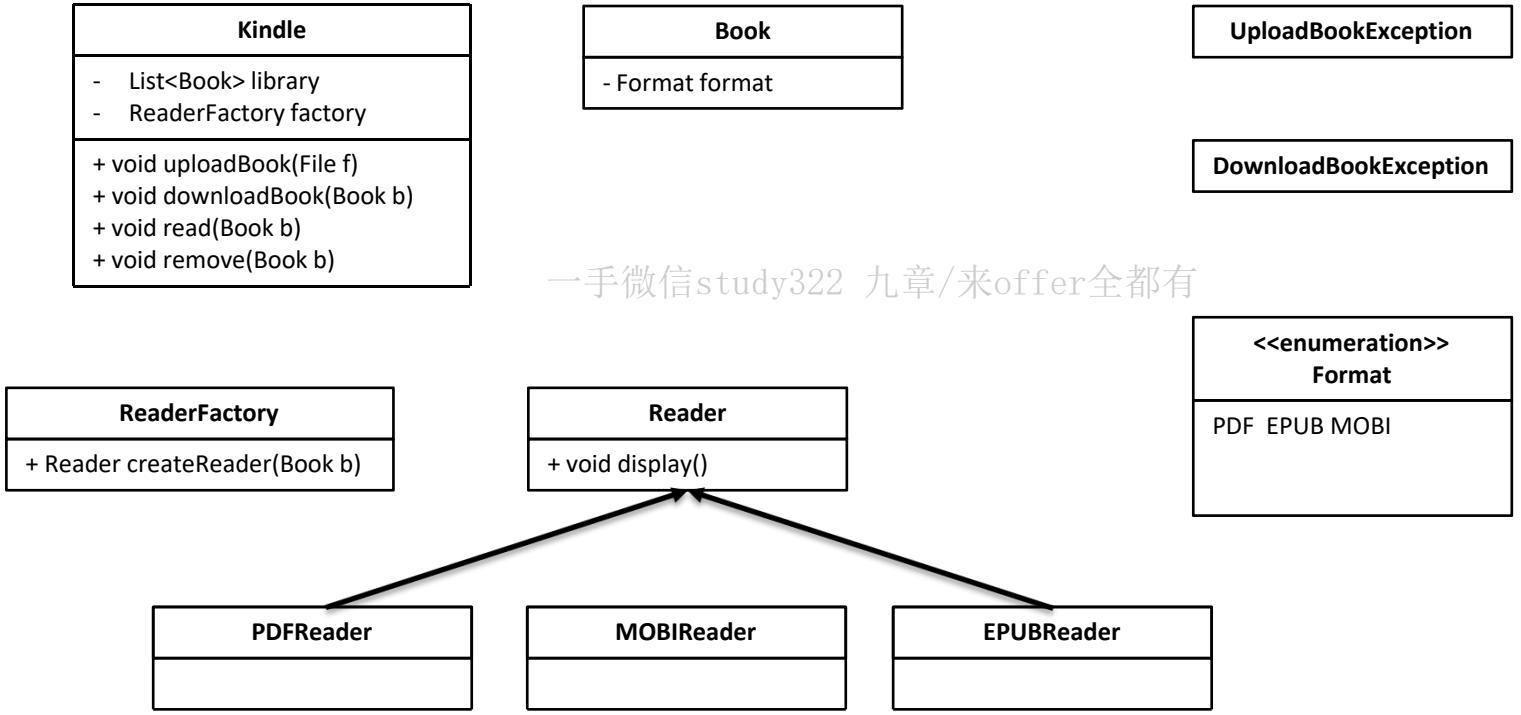
Factory design pattern



Factory design pattern



Factory design pattern



Simple factory

```
public Reader createReader(Book book)
{
    if(book.getFormat == Format.PDF)
    {
        return new PDFReader(book);
    }
    else if(book.getFormat == Format.MOBI)
    {
        return new MOBIReader(book);
    }
    else if(book.getFormat == Format.EPUB)
    {
        return new EPUBReader(book);
    }
    retrun null;
}
```

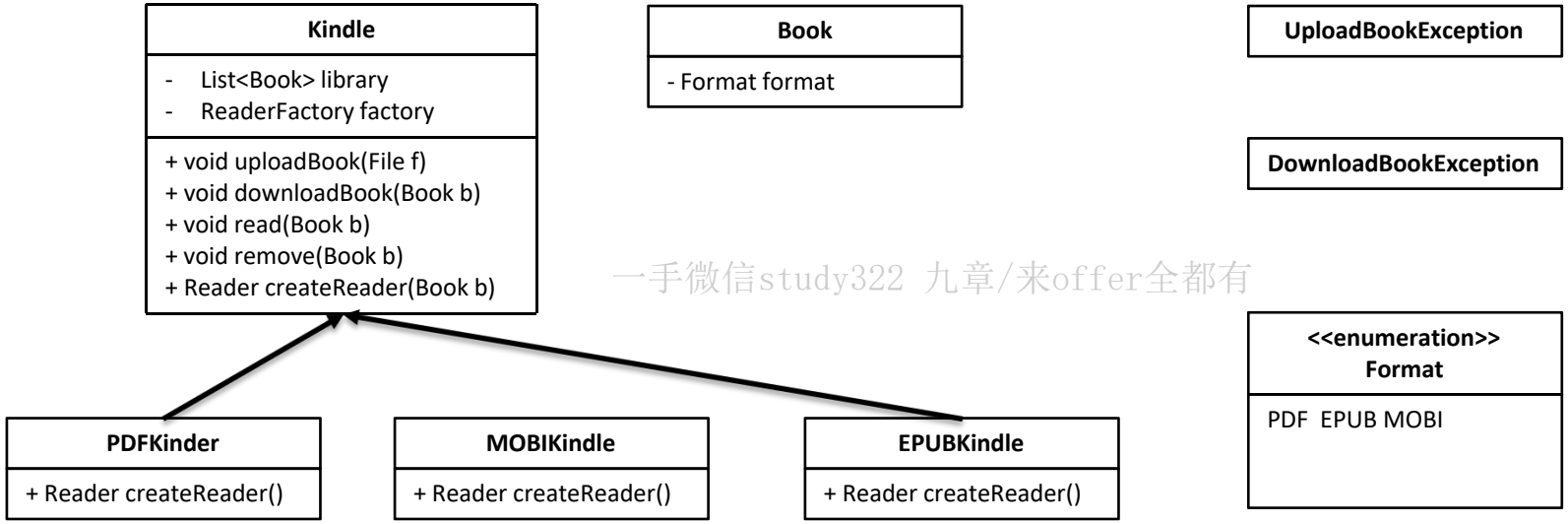


```
Reader reader = factory.createReader(book);
reader.display();
```

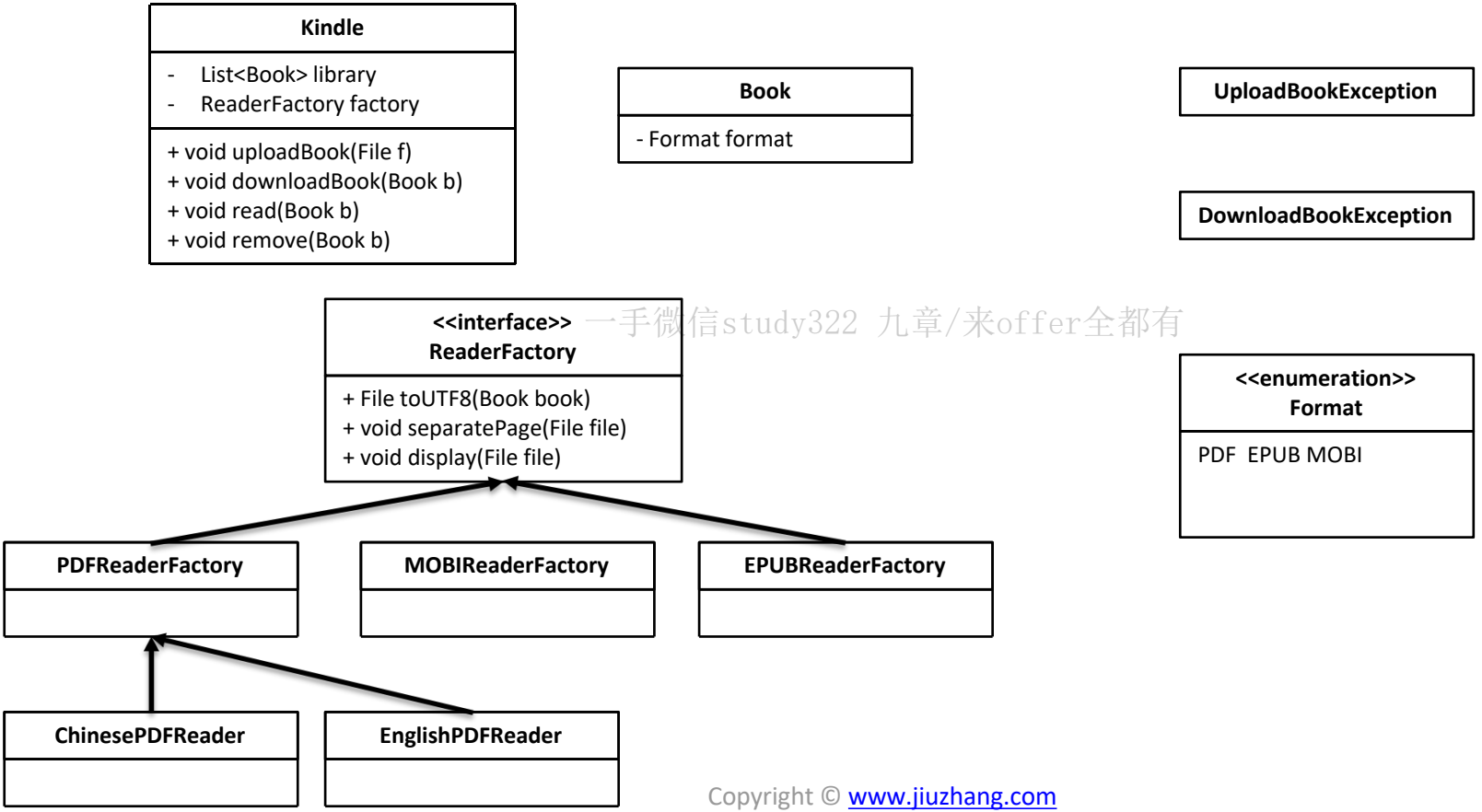

Factory design pattern

- Factory method
- Abstract factory

Factory method



Abstract factory



Strategy v.s. Factory

Strategy v.s. Factory

42

Strategy is about behavior. Factory is about creation/instatation.

Suppose you have an algorithm, to calculate a discount percentage. You can have 2 implementations of that algorithm; one for regular customers, and one for extra-ordinary good customers.
You can use a strategy DP for this implementation: you create an interface, and 2 classes that implement that interface. In one class, you implement the regular discount-calculation algorithm, in the other class you implement the 'good customers' algorithm.

Then, you can use a factory pattern to instantiate the class that you want. The factory method thus instantiates either the regular customer-discount algorithm, or the other implementation.

In short: the factory method instantiates the correct class; the strategy implementation contains the algorithm that must be executed.

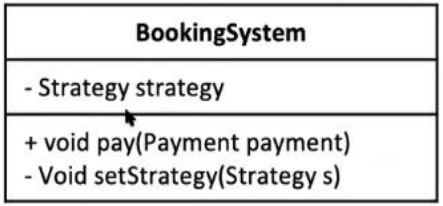
share improve this answer

answered Mar 21 '11 at 8:

Frederik Gheyse

45.8k ● 8 ● 78 ●

Strategy v.s. Factory




```
String account = payment.getAccount();
String password = payment.getPassword();
```

```
String cardId = payment.getCardId();
String name = payment.getName();
String cvv = payment.getCvv();
```

Strategy v.s. Factory

```
public class StrategyFactory
{
    public Strategy createStrategy(Payment payment)
    {
        if(payment.getMethod().equals("paypal"))
        {
            strategy = new PaypalStrategy();
        }
        else if(payment.getMethod().equals("credit card"))
        {
            strategy = new CreditCardStrategy();
        }
    }
}

public void pay(Payment payment)
{
    strategy = createStrategy(payment);
    strategy.processPayment(payment);
}
```

```
public interface Strategy
{
    public void processPayment(Payment payment);
}

public class PaypalStrategy implements Strategy
{
    public void processPayment(Payment payment)
    {
        // get paypal account
        // get paypal password
        // ...
    }
}
```


Kindle in Action

OOD 面向对象设计


```
// Use cases:
// - Upload Ebook
// - Read Ebook
// - Download Ebook
// - Delete Ebook
```


OOD 面向对象设计


```
// Upload Ebook
// Steps:
//  - Check if book is valid, and kindle has storage for it
//  - Add book to libaray
void uploadEbook(Book book) throws UploadFailureException
{

}
```



```
// Download Ebook
// Steps:
//  - Check if book is valid, and kindle has storage for it
//  - Add book to libaray
void downloadEbook(Book book) throws DownloadFailureException
{

}
```


OOD 面向对象设计


```
25 // Delete Ebook
26 // Steps:
27 //  - remove from library
28 void deleteEbook(Book book) throws DeleteFailureException
29 {
30
31 }
```



```
33 // Read Ebook
34 // Steps:
35 //  - Check if book format can be supported
36 //  - Finder approperite reader for Ebook
37 void read(Book book) throws ReadFailureException
38 {
39
40 }
```


OOD 面向对象设计


```
7  class Book
8  {
9      // getters
10     ...
11
12     // setters
13     ...
14
15     private int size;
16     private Format format;
17 }
```


OOD 面向对象设计


```
19 class kindle
20 {
21   private int availableStorage;
22   private Set<Format> supportedFormats;
23   private List<Book> library;
24
25   // Upload Ebook
26   // Steps:
27   //   - Check if book is valid, and kindle has storage for it
28   //   - Add book to libaray
29   void uploadEbook(Book book) throws UploadFailureException
30   {
31     if (availableStorage <= book.getSize())
32     {
33       throw new UploadFailureException("storage full");
34     }
35     if (!supportedFormats.contains(book.getFormat()))
36     {
37       throw new UploadFailureException("Unsupported format");
38     }
39     library.add(book);
40   }
41
```


OOD 面向对象设计


```
42  // Download Ebook
43  // Steps:
44  //  - Check if book is valid, and kindle has storage for it
45  //  - Add book to libaray
46  void downloadEbook(Book book) throws DownloadFailureException
47  {
48      // same as upload
49  }
```


```
51  // Delete Ebook
52  // Steps:
53  //   - remove from library
54  void deleteEbook(Book book) throws DeleteFailureException
55  {
56      // remove from library
57  }
```


OOD 面向对象设计


```
61  // Read Ebook
62  // Steps:
63  //   - Check if book format can be supported
64  //   - Finder approperite reader for Ebook
65  void read(Book book) throws ReadFailureException
66  {
67    if (!supportedFormats.contains(book.getFormat()))
68    {
69      throw new ReadFailureException("Unsupported format");
70    }
71    Reader reader = readerFactory.create(book);
72    reader.read(book);
73  }
74 }
```


OOD 面向对象设计


```
19 class kindle
20 {
21     private int availableStorage;
22     private Set<Format> supportedFormats;
23     private List<Book> library;
24     private ReaderFactory readerFactory;
25
```


OOD 面向对象设计


```
76 enum Format
77 {
78   PDF,
79   EPUB
80 }
81
82 abstract Class Reader
83 {
84   private Format format;
85   public abstract void read(Book book)
86   {
87   }
88 }
89
90 class PDFReader extends Reader
91 {
92   public void read(Book book)
93   {
94     // read PDF
95   }
96 }
97
98 Class ReaderFactory
99 {
100  public Reader create(Book book) throws UnknownFormatException
101  {
102    switch(book.getFormat())
103    {
104      case PDF:
105        return new PDFReader();
106      ...
107      default:
108        throw new UnknownFormatException("Unkown format");
109    }
110  }
111 }
```

Recap

- 常见的实物类面向对象设计

Recap

- 常见的实物类面向对象设计
- Input -> 题目主体 -> Output
- State design pattern
- Decorate design pattern
- Factory design pattern

Q & A

扫描二维码关注微信/微博
获取最新面试题及权威解答

微信: ninechapter
知乎专栏：http://zhuanlan.zhihu.com/jiuzhang
微博: http://www.weibo.com/ninechapter
官网: www.jiuzhang.com

扫一扫 不怀孕