Favour Skeletal Interface in java

# Skeletal Interface

The skeletal interface is a design, by which we can use the benefits of interface and abstract class together.

In Java Collection API has adopted this kind of design, AbstractSet , AbstractMap etc. are the example of Skeletal interface. Also ***Josua Bloch*** mention Skeletal interface into his book “***Effective java***”

In this article, we will see how we can efficiently design our system so it can use the feature of interface and Abstract class both.

**Let try to understand it by a problem.**

Suppose we want to create different types of Vending machines say candy vending, Soft drink vending etc. To get products from vending we need to start the vending machine then choose the product, pay for the same and collect it.

After that Vending machine should be stopped.

**First Approach:**

We can create a vending machine interface then for different product type we will create a concrete implementation of vending machine.

**Code**:

**package** com.example.skeletal;

**public** **interface** Ivending {

**void** start();

**void** chooseProduct();

**void** stop();

**void** process();

}

**package** com.example.skeletal;

**public** **class** CandyVending **implements** Ivending{

@Override

**public** **void** start() {

System.***out***.println("Start Vending machine");

}

@Override

**public** **void** chooseProduct() {

System.***out***.println("Produce diiferent candies");

System.***out***.println("Choose a type of candy");

System.***out***.println("pay for candy");

System.***out***.println("collect candy");

}

@Override

**public** **void** stop() {

System.***out***.println("Stop Vending machine");

}

@Override

**public** **void** process() {

start();

chooseProduct();

stop();

}

}

**package** com.example.skeletal;

**public** **class** DrinkVending **implements** Ivending{

@Override

**public** **void** start() {

System.***out***.println("Start Vending machine");

}

@Override

**public** **void** chooseProduct() {

System.***out***.println("Produce diiferent soft drinks");

System.***out***.println("Choose a type of soft drinks");

System.***out***.println("pay for drinks");

System.***out***.println("collect drinks");

}

@Override

**public** **void** stop() {

System.***out***.println("stop Vending machine");

}

@Override

**public** **void** process() {

start();

chooseProduct();

stop();

}

}

**package** com.example.skeletal;

**public** **class** VendingManager {

**public** **static** **void** main(String[] args) {

Ivending candy = **new** CandyVending();

Ivending drink = **new** DrinkVending();

candy.process();

drink.process();

}

}

Output :

Start Vending machine

Produce diiferent candies

Choose a type of candy

pay for candy

collect candy

Stop Vending machine

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Start Vending machine

Produce diiferent soft drinks

Choose a type of soft drinks

pay for drinks

collect drinks

stop Vending machine

For simplicity, I do not divide each step as an individual method. In chooseProduct() I merge some steps.

Although it looks goods, the above code has some problems, if we see the codes carefully we can see There is a lot of duplicate codes. start(), stop() and process() methods do same thing in

Each concrete implementation.

**Code duplication increases 3 times when the number of concrete implementation increases.**

We can create a utility class and put common code into the same but that will break the single responsibility principal. Can introduce Shotgun surgery code smell.

**Disadvantage of Interface:**

As the interface is a contract and does not contain method body, each implementation has to fulfill the contract and provide an implementation of all methods. May some of the methods duplicate across the concrete implementation.

**Approach 2:**

We can overcome it through Abstract class.

Code:

**package** com.example.skeletal;

**public** **abstract** **class** AbstractVending {

**public** **void** start()

{

System.***out***.println("Start Vending machine");

}

**public** **abstract** **void** chooseProduct();

**public** **void** stop()

{

System.***out***.println("Stop Vending machine");

}

**public** **void** process()

{

start();

chooseProduct();

stop();

}

}

**package** com.example.skeletal;

**public** **class** CandyVending **extends** AbstractVending{ //implements Ivending{

@Override

**public** **void** chooseProduct() {

System.***out***.println("Produce diiferent candies");

System.***out***.println("Choose a type of candy");

System.***out***.println("pay for candy");

System.***out***.println("collect candy");

}

}

**package** com.example.skeletal;

**public** **class** DrinkVending **extends** AbstractVending{ //implements Ivending{

@Override

**public** **void** chooseProduct() {

System.***out***.println("Produce diiferent soft drinks");

System.***out***.println("Choose a type of soft drinks");

System.***out***.println("pay for drinks");

System.***out***.println("collect drinks");

}

}

**package** com.example.skeletal;

**public** **class** VendingManager {

**public** **static** **void** main(String[] args) {

AbstractVending candy = **new** CandyVending();

AbstractVending drink = **new** DrinkVending();

candy.process();

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

drink.process();

}

}

Here, I provide common code implementation in to the abstract class. And Candyvending and DrinkVending extends this AbstractVending. This implementation gets rid of duplicate code but adds a new problem.

As, Candyvending and DrinkVending extends the Abstractclass we can’I have provision to extends another class or does not support Multiple inheritance.

Say I want to add VendingServicing class which will clean and check the vending machine.

In this scenario I can’t extend VendingServicing as I have already extends AbstractVending. One thing I can do, create a compostion but again we have to pass vending Machine into it which will strongly couple VendingServicing and Vending machine.

**Disadvantage of Abstract class:**

We can’t support Multiple inheritance due to diamond problem.

So It will be good if we can use advantages of both interface and Abstract class.

**We call it Abstract Interface or Skeletal interface.**

**To achieve Skeletal interface,**

Step 1: Create an interface.

Step 2: Create an Abstract class implement that interface and provide the implementation of Common method.

Step 3. In subclass create a private inner class which extends the Abstract class, Now this class can extend class implement any interfaces while using the common method by delegation call to Abstract class.

Let check the code,

**package** com.example.skeletal;

**public** **interface** Ivending {

**void** start();

**void** chooseProduct();

**void** stop();

**void** process();

}

**package** com.example.skeletal;

**public** **abstract** **class** AbstractVending **implements** Ivending{

**public** **void** start()

{

System.***out***.println("Start Vending machine");

}

//public abstract void chooseProduct();

**public** **void** stop()

{

System.***out***.println("Stop Vending machine");

}

**public** **void** process()

{

start();

chooseProduct();

stop();

}

}

**package** com.example.skeletal;

**public** **class** CandyVending **implements** Ivending{

**private** **class** AbstractVendingDelegator **extends** AbstractVending

{

@Override

**public** **void** chooseProduct() {

System.***out***.println("Produce diiferent candies");

System.***out***.println("Choose a type of candy");

System.***out***.println("pay for candy");

System.***out***.println("collect candy");

}

}

AbstractVendingDelegator delegator = **new** AbstractVendingDelegator();

@Override

**public** **void** start() {

delegator.start();

}

@Override

**public** **void** chooseProduct() {

delegator.chooseProduct();

}

@Override

**public** **void** stop() {

delegator.stop();

}

@Override

**public** **void** process() {

delegator.process();

}

}

**package** com.example.skeletal;

**package** com.example.skeletal;

**public** **class** DrinkVending **extends** VendingService **implements** Ivending {

**private** **class** AbstractVendingDelegator **extends** AbstractVending

{

@Override

**public** **void** chooseProduct() {

System.***out***.println("Produce diiferent soft drinks");

System.***out***.println("Choose a type of soft drinks");

System.***out***.println("pay for drinks");

System.***out***.println("collect drinks");

}

}

AbstractVendingDelegator delegator = **new** AbstractVendingDelegator();

@Override

**public** **void** start() {

delegator.start();

}

@Override

**public** **void** chooseProduct() {

delegator.chooseProduct();

}

@Override

**public** **void** stop() {

delegator.stop();

}

@Override

**public** **void** process() {

delegator.process();

}

}

**package** com.example.skeletal;

**public** **class** VendingManager {

**public** **static** **void** main(String[] args) {

Ivending candy = **new** CandyVending();

Ivending drink = **new** DrinkVending();

//AbstractVending candy = new CandyVending();

//AbstractVending drink = new DrinkVending();

candy.process();

System.***out***.println("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*");

drink.process();

**if**(drink **instanceof** VendingService)

{

VendingService vs = (VendingService)drink;

vs.service();

}

}

}

Please look the above design I create an interface then create an abstract class where I define all common implementation, then for each subclass implement a delegator class

And using that delegator we forward the call to AbstractVending.

**Benefits of Skeletal Interface:**

1. A subclass can extend other class like DrinkVending.

2. Get rid of duplicate code by delegation call to Abstract class.

3. If a subclass needs a new implementation of interface it can do so.

**Conclusion**: when your interface has some common method always create an Abstract class

And in a subclass, you can use this subclass as a delegator. Always try to use Skeletal Interface