REFACTORING

SOLUTIONS

home / solutions / refactoring

#solutions #java #refactoring #oop

DOWNLOAD AND IMPORT

Code for these solutions can be downloaded and imported into IntelliJ from here: refactoring-solution.zip

EXERCISE 1

1. DUPLICATE CODE

Methods **isElegibleForFreeDelivery()** and **printOrder()** use the same code to calculate the order total.

Solution:

- Use Extract Method to create a new **getTotal()** method.
- Use Inline Temp to simplify the resulting code.

2. DATA CLASS

Data Classes are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes. Sometimes this means that we can do without the class, other times that there are things that the class isn't, but should be, doing.

In this case, maybe it should be the class **OrderLine** that should calculate its total value.

Solution:

- Use Encapsulate Field to hide the fields and replace any code accessing them with getters and setters.
- Use Extract Method to create a new **getTotal(OrderLine line)** method.
- Use Move Method to move it to the **OrderLine** class.

3. FEATURE ENVY

Class **Order** seems more interested in whats going on in class **OrderLine** than in itself. For example, in this line it accesses data from the **OrderLine** class three times:

```
printBuffer.append(line.getProduct().getName() + "(x" +
line.getQuantity() + "): " + line.getTotal() + "\n");
```

Solution: Use Extract Method to create a new **getLineString(Line line)** method, then use Move Method to move it to the **OrderLine**.

As an extra, you could use Rename Method to rename the method to **toString()** making it override the **Object.toString()** method and replace the call with just the object reference:

```
System.out.println(line);
```

4. UNECESSARY IF STATEMENT

Not a smell *per se*, but a pet peave. We don't need the **if** in method **isElegibleForFreeDelivery**.

Solution: Just return the result of the comparison.

EXERCISE 2

---

1. SWITCH STATEMENTS

Repeated **"switch"** throughout the **Shape** class could be replaced with polymorphism.

Solution:

Use Replace Type Code with Subclasses

IntelliJ doesn't do this refactoring automatically but it can help:

- Create classes **Rectangle** and **Circle** extending **Shape**.
- Add the correct **constructor** to each of these classes (Alt+Enter will help).
- Pull Method Dow: **getArea**, **getPerimeter and** draw (and **delete** unneeded code in each class).
- Pull Field Down: **width**, **height** (to Rectangle) and **radius** (to Circle).
- Pull Method Up: **getArea**, **getPerimeter and** draw (and make them abstract).
- Replace Shape with **Circle** and **Rectangle** in tests.
- Delete **Type** and **type**.

EXERCISE 3

---

1. LONG METHOD

Bit of a reach, but the **applyDiscount** method is unnecessarily long.

Solution:

Use Replace Nested Conditional with Guard Clauses.

Or just ignore it as this method will be simplified later on...

2. SWITCH STATEMENTS

"If" in **Discount** class could be replaced with polymorphism.

Solution:

Use Replace Type Code with Subclasses.

IntelliJ doesn't do this refactoring automatically but it can help:

- Create classes **FixedDiscount** and **PercentageDiscount** extending **Discount**
- Add the correct **constructor** to each of these classes (Alt+Enter will help)
- Pull Method Down: **applyDiscount** (and **delete** unneeded code in each class)
- Pull Field Down: **fixed** (to FixedDiscount) and **percentage** (to PercentageRectangle).
- Pull Method Up: **applyDiscount** (and make it abstract)
- Replace Discount with **FixedDiscount** and **PercentageDiscount** in tests

3. SWITCH STATEMENTS AGAIN (WITH TESTING FOR NULL)

- The *getTotal()* method tests for null. This is a potential bug trap as we might accidentally forget to test somewhere in the code.
- Also it makes the code longer, harder to read and **ugly**.

Solution:

Refactoring **"switch"** statements in which one of the conditional options is *null* is a special case.

Use Introduce Null Object:

- Create a subclass called **NoDiscount** extending **Discount**
- Add default behavior to **NoDiscount** (always returns price).
- When creating a **SimpleOrder** set the discount to **NoDiscount**.
- Remove the test for *null* in **getTotal()**.

EXERCISE 4

---

1. REFUSED BEQUEST

Client extends **Worker** but doesn't implement all of its super-class interface (clients don't have logins).

Solution:

One way would be to transform inheritance into delegation but in this case it is easier to:

- Extract **Worker** super-class called **Person** and pull-up methods **getName** and **getPhone**, and fields **name** and **phone**.
- Make **Client** extend **Person**

EXERCISE 5

---

1. LONG METHOD && COMMENTS

Method **execute()** in class **Turtle** is unnecessarily long.

Solution:

Use Extract Method three times. Comments will help you with choosing the right method names and won't be necessary anymore.

2. PRIMITIVE OBSESSION && COMMENTS

A **direction** is not a char. A direction is a direction. Stop **obsessing** with **primitive** types.

Solution:

Use Replace Data Value with Object and replace the char with a class called **TurtleDirection**.

3. PRIMITIVE OBSESSION && COMMENTS

A **command** is also not a char.

Solution:

Use Replace Data Value with Object and replace the char with a class called **TurtleCommand**.

4. SWITCH STATEMENTS

Methods **moveForward()**, **rotateLeft()** and **rotateRight()** have a lot of **"ifs"** and that is always *fishy*.

Solution:

- Use Replace Type Code with Subclasses to generate **four** Turtle classes (one for each direction: **TurtleNorth**, ...). First create the four classes, then use Push Down Method, fix the code in each subclass and finally Pull Up Method making them abstract.
- Use Replace Inheritance with Delegation so that the four new classes are no longer sub-classes of the original Turtle class, but only have an association with it.
- Make these four classes implement the same interface using Extract Interface. Call the interface **TurtleState**.
- Make so that the **Turtle** class has one of these classes as a field (it can replace the direction field) and delegates to it any calls to methods: **moveForward()**, **rotateLeft()** and **rotateRight()**. Delete the methods from **Turtle** and make the class not abstract.

There a few other kinks that you will have to sort out.

Congratulations:

You just rediscovered the State pattern, but in this particular case, this pattern is clearly **overkill**. Not all smells are indicative of bad code., but in this case, maybe you can figure out a smart solution without using refactoring or complicated patterns.

EXERCISE 6

1. DATA CLUMPS

The attributes **locationLatitude**, **locationLongitude** and **locationName** are frequently used together. Bunches of data that hang around together really ought to be made into their own object.

Solution:

Use Extract Class ↗ on the fields to turn this clump into an object. Use Introduce Parameter Object ↗ or Preserve Whole Object ↗ in the constructor.

2. LONG METHOD ↗ && COMMENTS ↗

Method **isNextAppraisalOverdue()** in class **Tree** is unnecessarily long.

Solution:

Use Extract Method ↗ two times. The comments won't be necessary anymore, as the names of the extracted methods will help to understand the intent of the code.