Refactoring

What

★★☆☆ Can explain refactoring

The first version of the code you write may not be of production quality. It is OK to first concentrate on making the code work, rather than worry over the quality of the code, as long as you improve the quality later. This process of **improving a program's internal structure in small steps without modifying its external behavior is called *refactoring*.**

- **Refactoring is not rewriting**: Discarding poorly-written code entirely and re-writing it from scratch is not refactoring because refactoring needs to be done in small steps.
- **Refactoring is not bug fixing**: By definition, refactoring is different from bug fixing or any other modifications that alter the external behavior (e.g. adding a feature) of the component in concern.

💡 Improving code structure can have many secondary benefits: e.g.

- hidden bugs become easier to spot
- improve performance (sometimes, simpler code runs faster than complex code because simpler code is easier for the compiler to optimize).

Given below are two common refactorings (more).

Refactoring Name: **Consolidate Duplicate Conditional Fragments**

Situation: The same fragment of code is in all branches of a conditional expression.

Method: Move it outside of the expression.

📦 Example:

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
} else {
    total = price * 0.98;
    send();
}
```

→

```
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

Refactoring Name: **Extract Method**

Situation: You have a code fragment that can be grouped together.

Method: Turn the fragment into a method whose name explains the purpose of the method.

📦 Example:

```
void printOwing() {
    printBanner();

    // print details
    System.out.println("name:   " + name);
    System.out.println("amount  " + getOutstanding());
}
```

↓

```java
void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}

void printDetails(double outstanding) {
    System.out.println("name:   " + name);
    System.out.println("amount  " + outstanding);
}
```

Some IDEs have builtin support for basic refactorings such as automatically renaming a variable/method/class in all places it has been used.

Refactoring, even if done with the aid of an IDE, may still result in regressions. Therefore, each small refactoring should be followed by regression testing.

Exercises

How

★★★☆ Can apply some basic refactoring

Given below are some more commonly used refactorings. A more comprehensive list is available at refactoring-catalog.

1. Consolidate Conditional Expression
2. Decompose Conditional
3. Inline Method
4. Remove Double Negative
5. Replace Magic Literal
6. Replace Nested Conditional with Guard Clauses
7. Replace Parameter with Explicit Methods
8. Reverse Conditional
9. Split Loop
10. Split Temporary Variable

When

★★★☆ Can decide when to apply a given refactoring

One way to identify refactoring opportunities is by *code smells*.

> A *code smell* is a surface indication that usually corresponds to a deeper problem in the system. First, a smell is by definition something that's quick to spot. Second, smells don't always indicate a problem.
> --adapted from https://martinfowler.com/bliki/CodeSmell.html

An example (from the same source as above) is the code smell *data class* i.e., a class with all data and no behavior. When you encounter the such a class, you can explore if refactoring it to move the corresponding behavior into that class is appropriate. Some more examples:

- Long Method
- Large Class
- Primitive Obsession
- Temporary Field
- Shotgun Surgery

Periodic refactoring is a good way to pay off the *technical debt* a code base has accumulated.

> Software systems are prone to the build up of **cruft** - deficiencies in internal quality that make it harder than it would ideally be to modify and extend the system further.Technical Debt is a metaphor, coined by Ward Cunningham, that frames how to think about dealing with this cruft, thinking of it like a financial debt. The extra effort that it takes to add new features is the interest paid on the debt.
> --https://martinfowler.com/bliki/TechnicalDebt.html

While it is important to refactor frequently so as to avoid the accumulation of 'messy' code (aka technical debt), an important question is how much refactoring is *too much* refactoring? **It is too much refactoring when the benefits no longer justify the cost.** The costs and the benefits depend on the context. That is why some refactorings are 'opposites' of each other (e.g. *extract method* vs *inline method*).

Resources

- A list of code smells, symptoms, and remedies: https://refactoring.guru/refactoring/smells

Exercises