Java Refactoring

Table of contents

What is refactoring? 1
Why refactor? 1
Good practice 2
When to refactor 2
Example: making it easier to add a feature 2
Example: making it easier to understand 4
Code smells 5

Introduced in 2026-04-02-refactoring[1] (Lecture, Week 6). Practised in week9-lab-refactoring[2].

What is refactoring?

> A disciplined technique for continuously restructuring an existing body of code, altering its internal structure *without changing its external behaviour*.

Two halves make it a *technique* rather than a mood: **continuous** (not a one-off clean-up) and **behaviour-preserving** (the outside of the system cannot tell you did it). See also refactoring.guru.

Why refactor?

Over time, as features are added: quick fixes ("hacks") accumulate, the design becomes messy, and code becomes harder to change.

[1] courses/csse2002/2026-04-02-refactoring.pdf
[2] courses/csse2002/week9-lab-refactoring.pdf

Good practice

- **Don't** refactor and add functionality at the same time — keep the two activities separate.
- Have good tests *before* refactoring, so you'll know immediately if you've broken something (see java-testing[3], java-junit[4]). This is what puts `Refactor` where it is in the development cycle, drawn in 2026-04-02-refactoring[5]: you only ever enter it from a passing test.
- Take short, deliberate steps: move a member variable, split a method, rename a variable. Refactoring is usually many small, localised changes that add up to a larger-scale change — small steps + testing after each one avoids prolonged debugging.
- Refactor early, refactor often.

When to refactor

There are two triggers: refactor when it would make the next change **easier to add**, and refactor when it would make the code **easier to understand**. Code smells (below) are a third: symptoms that point at where one of the first two applies.

Example: making it easier to add a feature

A `Vehicle` class using a `type` string and a chain of `if`/`else` branches is fragile to extend. Adding a scooter means adding the *same* `else if (type.equals("scooter")) { return ..; }` block **three times over** — one for each place in the codebase that switches on `type`. Adding one vehicle is not one edit, it is an edit everywhere the string is tested, and nothing tells you when you have found them all.

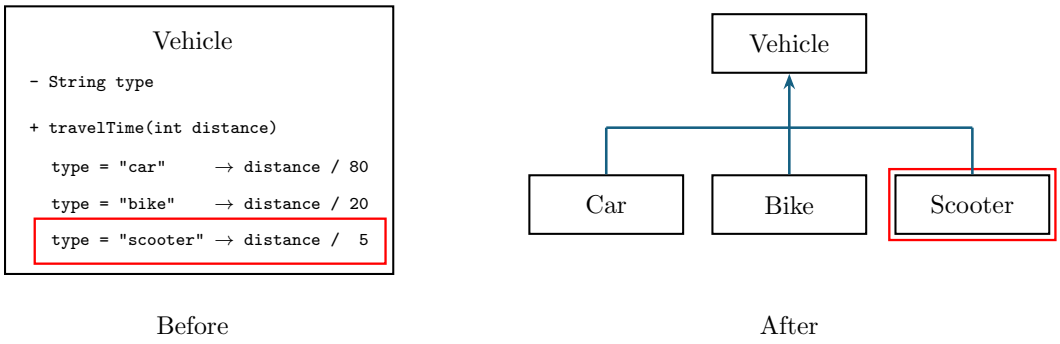
```
public class Vehicle {
    private String type;

    public Vehicle(String type) { this.type = type; }

    public int travelTime(int distance) {
        if (type.equals("car")) {
            return distance / 80;
        } else if (type.equals("bike")) {
            return distance / 20;
        }
        return distance;
    }
}
```

[3] courses/csse2002/java-testing.pdf
[4] courses/csse2002/java-junit.pdf
[5] courses/csse2002/2026-04-02-refactoring.pdf

Refactored to use inheritance/polymorphism (see java-polymorphism[6]), adding a `Scooter` is just one new class — no existing code needs editing:

```
abstract class Vehicle {
    public abstract int travelTime(int distance);
}
class Car extends Vehicle {
    @Override
    public int travelTime(int distance) { return distance / 80; }
}
class Bike extends Vehicle {
    @Override
    public int travelTime(int distance) { return distance / 20; }
}
class Scooter extends Vehicle {
    @Override
    public int travelTime(int distance) { return distance / 5; }
}
```

The refactoring has moved *where* a new vehicle type lands. Before, it lands in the middle of an existing method (and in every other method that branches on `type`); after, it lands in a new leaf of the hierarchy that nothing existing has to know about:



Before After

Figure 1: Adding a scooter, before and after. Left: one class, and the new type is another branch spliced into an existing method — the red outline marks the edit. Right: an abstract Vehicle with one subclass per type, joined by a connector with a single arrowhead into the parent. The new type is a whole new box that no existing box points at, again outlined in red.

What has *not* changed: `travelTime(int distance)` still exists and still returns the same answers for cars and bikes. That is what makes this a refactoring rather than a rewrite. The

[6] courses/csse2002/java-polymorphism.pdf

`Vehicle` abstraction being open to new subclasses without being edited is the Open-Closed Principle arriving early — see java-solid-principles[7].

Example: making it easier to understand

The second trigger. Here nothing about the structure changes — the same three branches run in the same order — but the conditions are replaced by named boolean-returning methods, so the reader sees the *intent* of a branch instead of re-deriving it from raw comparisons every time:

```
// Before
private int temperature;
private boolean isRaining;
public void provideWeatherAdvice() {
    if (temperature > 25 && !isRaining) {
        wearSunscreen();
        hydrate();
    } else if (temperature <= 25 && !isRaining) {
        play();
    } else if (isRaining) {
        equipUmbrella();
    }
}
```

```
// After
private int temperature;
private boolean isRaining;
public void provideWeatherAdvice() {
    if (isHotAndSunny()) {
        wearSunscreen();
        hydrate();
    } else if (isPleasant()) {
        play();
    } else if (isRaining) {
        equipUmbrella();
    }
}
```

`isRaining` is left as it was, because a field already named for the condition it tests reads fine — only the two *compound* conditions needed a name. The bodies of the extracted methods (`isHotAndSunny()`, `isPleasant()`) are the conditions they replaced.

[7] courses/csse2002/java-solid-principles.pdf

Code smells

A “code smell” is a surface indication that usually corresponds to a deeper problem in the design. Three are worked here.

Bad naming

Single-letter and cryptic names force the reader to trace the logic before they can know what it is *for*. The two snippets compile to the same thing; only the second one tells you it counts even numbers:

```
// Before
int r = 0;
for (int x : y) {
    if (g(x)) {
        r++;
    }
}
return r;
```

```
// After
int evenCount = 0;
for (int number : numbers) {
    if (isEven(number)) {
        evenCount++;
    }
}
return evenCount;
```

Every name is renamed, including the loop variable and the collection — `r → evenCount`, `x → number`, `y → numbers`, `g → isEven`.

Duplication

Near-identical blocks repeated for different data. Two arrays, two copy-pasted loops; a fix to one has to be remembered for the other:

```
// Before
int[] first = new int[]{1, 2, 3};
int[] second = new int[]{4, 5, 6};

int firstSum = 0;
for (int i = 0; i < 3; i++) {
```

```
    firstSum += first[i];
}
int firstAverage = firstSum / 3;

int secondSum = 0;
for (int i = 0; i < 3; i++) {
    secondSum += second[i];
}
int secondAverage = secondSum / 3;
```

```
// After
int[] first = new int[]{1, 2, 3};
int[] second = new int[]{4, 5, 6};

int firstAverage = average(first);
int secondAverage = average(second);

int average(int[] numbers) {
    int sum = 0;
    for (int i = 0; i < 3; i++) {
        sum += numbers[i];
    }
    return sum / 3;
}
```

The duplicated loop now exists once, and the two call sites read as what they mean. The extracted `average` still hardcodes the length `3` in both the loop bound and the divisor, so it silently only works for three-element arrays — `numbers.length` in both places is the obvious next step. Extracting the duplication and *then* generalising the extracted method is exactly the "short, deliberate steps" style above.

Feature envy

A method that spends most of its time reaching into *another* class's data rather than its own. `User.checkout()` touches `cart` twice and `User`'s own state never — the logic is envious of `ShoppingCart`:

```
// Before
class ShoppingCart {
    public List<Item> getItems() {...}
    public int getVoucher() {...}
}

class User {
```

```
    private ShoppingCart cart;

    public double checkout() {
        double totalPrice = 0.0;
        for (Item item : cart.getItems()) {
            totalPrice += item.getPrice();
        }
        return totalPrice - cart.getVoucher();
    }
}
```

```
// After
class ShoppingCart {
    public List<Item> getItems() {...}
    public int getVoucher() {...}

    public double checkout() {
        double totalPrice = 0.0;
        for (Item item : getItems()) {
            totalPrice += item.getPrice();
        }
        return totalPrice - getVoucher();
    }
}

class User {
    private ShoppingCart cart;
}
```

Moving `checkout()` onto `ShoppingCart` turns two cross-class calls into two calls on `this` — the giveaway is that the `cart.` qualifier simply disappears:

Two things improve at once: `ShoppingCart` gets more cohesive (the checkout logic now sits with the data it operates on) and `User` gets less coupled to it (one link instead of three). Both are the subject of java-cohesion-and-coupling[8].

Others

Many other code smells exist. Further reading:

- Martin Fowler, *Refactoring: Improving the Design of Existing Code* (2nd edition, 2019)
- Robert Martin, *Clean Code: A Handbook of Agile Software Craftsmanship* (2008)
- John Ousterhout, *A Philosophy of Software Design* (2018)
- and the catalog of refactorings at refactoring.com.

[8] courses/csse2002/java-cohesion-and-coupling.pdf

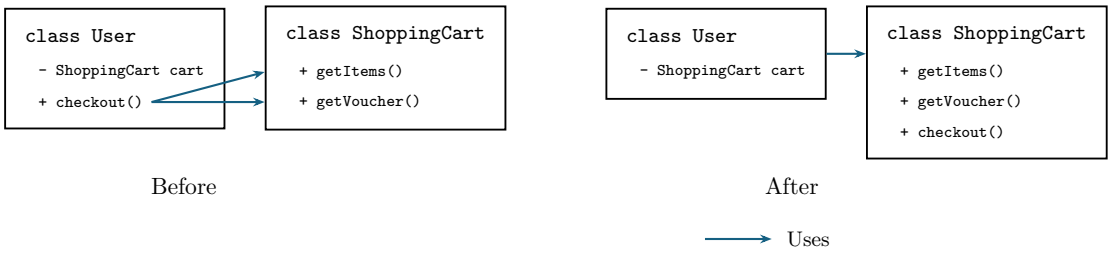


Figure 2: Feature envy, before and after. Before: User.checkout() reaches across the class boundary into two of ShoppingCart's members. After: checkout() sits with the data it uses, and User's only link to ShoppingCart is the field it holds. Each box is a class with its members listed as '+ m()'. A solid arrow means 'uses' and points from the depender to the thing depended upon; it may land on an individual member row.