Refactoring

Table of contents

Setup 1
Refactoring in the small 2
Refactoring in the large 8

Practical for 2026-04-30-generics-in-java[1] (Week 9). Extends java-refactoring[2] (Week 6) with a full worked refactoring of a Connect 4 implementation, plus a "refactoring in the large" exercise using java-cohesion-and-coupling[3] and java-solid-principles[4] (SRP, DIP).

Setup

The provided `.zip` contains:

```
resources/oTowerWin.txt
resources/simpleWin.txt
resources/towerWin.txt
src/library/Library.java
src/Connect4.java
test/Connect4Test.java
```

`Connect4` has one method, `play`, implementing (most of) the gameplay; the three `resources/*.txt` files are recorded games the tests replay. `Library` is a separate library-management system, used for the second half. Set up version control *before* refactoring (see java-version-control[5]), so each refactoring step is a separate commit that can be reverted if it breaks something.

[1] courses/csse2002/2026-04-30-generics-in-java.pdf
[2] courses/csse2002/java-refactoring.pdf
[3] courses/csse2002/java-cohesion-and-coupling.pdf
[4] courses/csse2002/java-solid-principles.pdf
[5] courses/csse2002/java-version-control.pdf

The running program prints the seven column headings `0123456`, then the board, then prompts the current player (`Player O: 6`, `Player X: 0`, …) for the column to drop into, and finally prints `Player X wins`. That console transcript is the *external behaviour* the refactoring must not change.

Refactoring in the small

Goal: make an existing, working class more **readable and understandable** without changing its behaviour — the unit tests must keep passing after every change. "In the small" refactorings touch one method/class's internals, not the overall class structure.

Checklist:

1. Code adheres to the course style guide.
2. Complex operations are explained.
3. Logical blocks of code are grouped with whitespace.
4. Methods have been decomposed such that each method has one clear purpose.
5. Appropriate data structures are used.

> **Working**
>
> Each item below demonstrates *one* type of refactoring; none is fully refactored or perfectly styled, and the list is a sample rather than an exhaustive treatment.
> **Style guide fixes** (vertical and horizontal whitespace) — trivial with a linter like Checkstyle:

```
// Before
int x=0;
while (x < 7){out.print(x); x++;}

// After
int x = 0;
while (x < 7) {
    out.print(x);
    x++;
}
```

> **Bad naming** — replace cryptic names with meaningful descriptions:

```java
// Before
public static void play(Scanner i, PrintWriter out) {
    int[] n = new int[7 * 6];
    int[] m = new int[7 * 6];
    boolean t = false;
}

// After
public static void play(Scanner in, PrintWriter out) {
    int[] boardX = new int[7 * 6];
    int[] boardO = new int[7 * 6];
    boolean isTurnX = false;
}
```

`while` loops that execute a fixed number of times — replace with `for`:

```java
// Before
int x = 0;
while (x < 7) { out.print(x); x++; }
x = 0;
while (x < 42) { ... x++; }

// After
for (int x = 0; x < 7; x++) { out.print(x); }
for (int x = 0; x < 42; x++) { ... }
```

Magic numbers — replace with named constants:

```java
private static final int ROW_SIZE = 7;
private static final int COL_SIZE = 6;

for (int i = 0; i < ROW_SIZE; i++) { out.print(i); }
for (int i = 0; i < ROW_SIZE * COL_SIZE; i++) { ... }
```

Decomposition — even code that isn't duplicated can be pulled into a helper method if it has a single, nameable purpose (here, printing the board):

```
private static void printBoard(int[] boardX, int[] boardO,
                               PrintWriter out) {
    for (int i = 0; i < ROW_SIZE; i++) { out.print(i); }
    for (int i = 0; i < ROW_SIZE * COL_SIZE; i++) { ... }
}

public static void play(Scanner in, PrintWriter out) {
    ...
    printBoard(boardX, boardO, out);
    ...
}
```

Comments — explain complex lines or the purpose of a block, not what is already obvious from the code:

```
for (int s = 0; s < 42; s++) {
    boolean nf = true;
    boolean mf = true;
    // Check horizontal win
    for (int q = 0; q < 4; q++) {
        if ((s + q) / 7 != s / 7) {
            // Found row out of bounds, not a win
            nf = false;
            break;
        }
        if (boardX[s + q] == 0) {
            // Found position that isn't an X, not a win
            nf = false;
        }
    }
    ...
}
```

Duplication — repeated blocks that do the same thing become helper methods whose differences are the parameters. The original win check repeats a near-identical horizontal/vertical scan for *both* players, and is refactored in two stages.
Stage 1 — extract a `checkWin(int[] playerBoard)` helper parameterised on which player's board to check, removing the per-player duplication:

```java
private static boolean checkWin(int[] playerBoard) {
    boolean win = true;
    // horizontal win
    for (int q = 0; q < 4; q++) {
        if ((s + q) / 7 != s / 7) { return false; }     // out of bounds
        if (playerBoard[s + q] == 0) { win = false; }   // not the player's
         ↪ piece
    }
    if (win) { return true; }

    win = true;
    // vertical win
    for (int q = 0; q < 4; q++) {
        if (s + (q * 7) >= 42) { return false; }
        if (playerBoard[s + (q * 7)] == 0) { win = false; }
    }
    return win;
}

// in play():
for (int s = 0; s < 42; s++) {          // check win
    if (checkWin(boardX)) { out.println("Player X wins"); exit = true; }
    if (checkWin(boardO)) { out.println("Player O wins"); exit = true; }
}
```

Stage 2 — split `checkWin` again into `checkHorizontalWin`/`checkVerticalWin`, each returning as soon as a check fails. This is mostly about comprehension, but it also removes the need to store a `win` variable at all:

```java
private static boolean checkHorizontalWin(int[] playerBoard) {
    for (int q = 0; q < 4; q++) {
        if ((s + q) / 7 != s / 7) { return false; }
        if (playerBoard[s + q] == 0) { return false; }
    }
    return true;
}

private static boolean checkVerticalWin(int[] playerBoard) {
    for (int q = 0; q < 4; q++) {
        if (s + (q * 7) >= 42) { return false; }
        if (playerBoard[s + (q * 7)] == 0) { return false; }
    }
    return true;
}

private static boolean checkWin(int[] playerBoard) {
    return checkHorizontalWin(playerBoard) || checkVerticalWin(playerBoard);
}
```

Data structures — the flat 42-slot array representing the board is clumsy. Three things to try (partly a matter of personal preference):

1. Change the flat array of 42 slots to a nested array of 6 rows with 7 slots (`int[] boardX = new int[7 * 6];` becomes `int[][] boardX = new int[7][6];`). This breaks every existing use of the board, so a lot of the class has to be (gradually) rewritten.
2. Combine the two per-player arrays into one board — `1` at a position means X, `2` means O. This avoids the two data structures accidentally getting out of sync.
3. Turn the array into a `boolean[]`. Since `boardX` and `board0` only ever store `1` or `0`, they are effectively boolean arrays already.

Why the win check indexes the way it does

The two guards in `checkWin` are the least obvious lines in the class, and they are exactly what the "comments" refactoring above was for: both exist because the board is *flat*, so a run of four can silently walk off the edge of one row and onto the next, or off the end of the array.

This is also the clearest argument for the "data structures" refactoring: with `int[][] boardX = new int[7][6]`, the horizontal guard becomes a plain column-index bound and the `/7` arithmetic disappears — which is why it is worth the rewrite it forces.

`int[] boardX = new int[7 * 6];` — one flat array, 7 slots per row, 6 rows

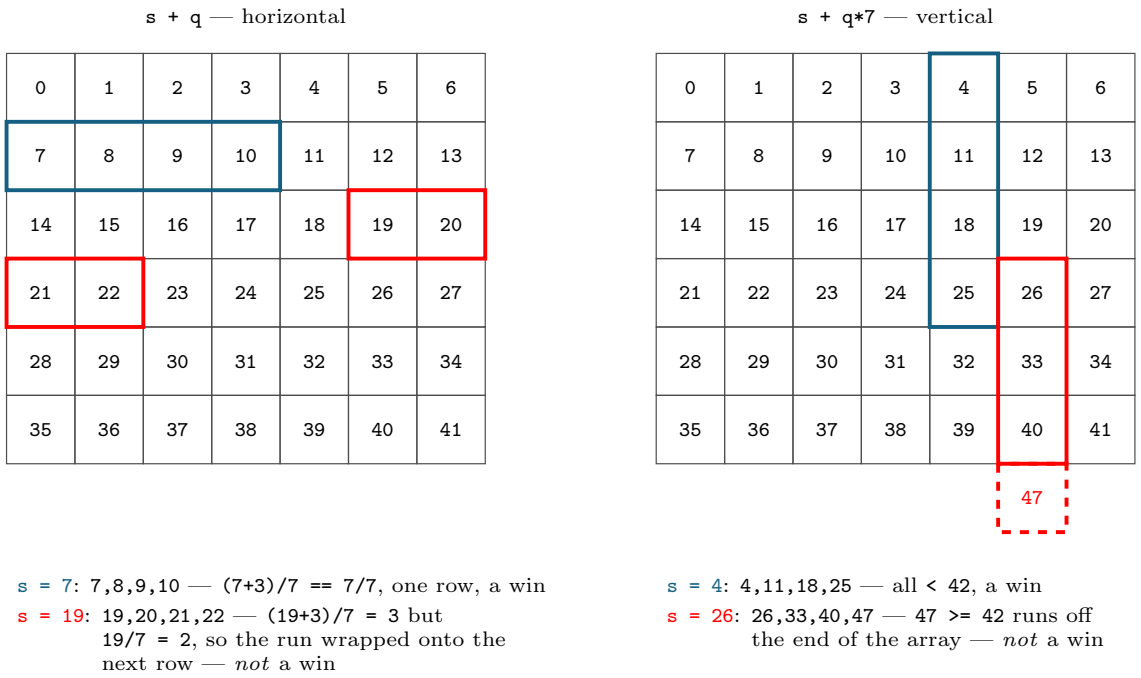

Figure 1: The two bounds checks in the win scan, drawn on the flat board. With 7 slots per row, a horizontal run is `s, s+1, s+2, s+3` and stays on one row only while `(s+q)/7 == s/7` — the left grid shows a run that satisfies it and one that wraps onto the next row. A vertical run is `s, s+7, s+14, s+21`, which needs no row test but must stay inside the array, hence `s + q*7 >= 42` — the right grid shows the run that falls off the end. The four example values of `s` exercise each branch.

Refactoring in the large

Goal: given a reasonably well-*styled* class that has **low cohesion** and too many responsibilities, split it into smaller, more cohesive classes with clear responsibilities — to make the code reusable and extensible, not just readable. Starting point: a library management system implemented entirely in the `Library` class.

Considerations when doing this:

- Each class serves one single-minded purpose (high cohesion).
- Each class has only one *reasonable* reason to change (SRP, see java-solid-principles[6]) — e.g. a change of classification system, a change of bookshelf rendering, or a change of customer management process should each land on one class.
- The interface of each class is designed to have reasonable programmatic interaction.
- Each component depends on abstractions such that each sub-component can be substituted or extended later (DIP, see java-solid-principles[7]).

For this exercise, don't worry about "over-engineering" — make the refactored system as flexible as possible.

> **Working**
>
> A possible split of `Library` (these are the basic classes — a real system could expand on this considerably):
>
> - **`Book`** — keeps track of data related to a single book (eg. title, author, id).
> - **`BookShelf`** — keeps track of all the `Book`s in the library and their availability status.
> - **`Borrower`** — keeps track of data related to a borrower (eg. name, id) and the books they're currently borrowing.
> - **`BorrowingSystem`** — handles borrowing and returning; the "bridge" that passes data between `BookShelf` and `Borrower`.

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

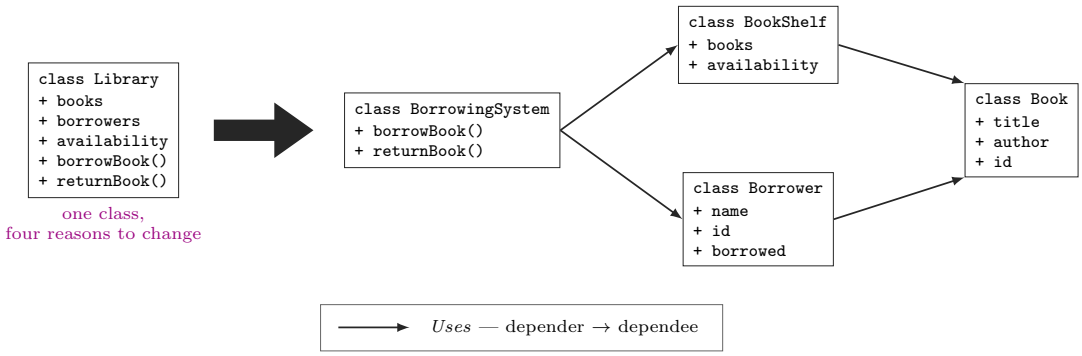

Figure 2: The `Library` monolith and the four-class split, with the dependencies that make `BorrowingSystem` a bridge: it is the only class that knows about both `BookShelf` and `Borrower`, so neither of those has to know about the other. A solid line with a solid arrowhead means *Uses* and points from depender to dependee; the block arrow means 'becomes'. All four classes are concrete and no interface is drawn, although the DIP consideration asks for dependencies on *abstractions*.