Refactoring

Table of contents

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

Starting point: a `Connect4` class (one `play` method implementing most of the game), a small `Library` helper, and a test suite. Set up version control *before* refactoring, so each refactoring step can be committed separately and reverted if it breaks something.

Refactoring in the small

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

> **i** Working
>
> **Style guide fixes** (whitespace) — trivial with a linter like Checkstyle:

[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

```
// 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 ones:

```
// Before
int[] n = new int[7 * 6];
int[] m = new int[7 * 6];
boolean t = false;

// After
int[] boardX = new int[7 * 6];
int[] boardO = new int[7 * 6];
boolean isTurnX = false;
```

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

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

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

Magic numbers — replace with named constants:

```
private static final int ROW_SIZE = 7;
private static final int COL_SIZE = 6;
// for (int i = 0; i < ROW_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); }
    // ...
}
```

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

```
// 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 — the original win-checking logic repeats a near-identical horizontal/vertical check for *both* players. This is refactored in two stages:
Stage 1 — extract a `checkWin(int[] playerBoard)` helper parameterised on which player's board to check, removing the player-specific duplication:

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

    win = true;
    for (int q = 0; q < 4; q++) {                   // vertical
        if (s + (q * 7) >= 42) { return false; }
        if (playerBoard[s + (q * 7)] == 0) win = false;
    }
    return win;
}
// if (checkWin(boardX)) { out.println("Player X wins"); exit = true; }
// if (checkWin(boardO)) { out.println("Player O wins"); exit = true; }
```

Stage 2 — further split `checkWin` into `checkHorizontalWin`/`checkVerticalWin`, each returning as soon as a check fails (removing the need for the `win` bookkeeping variable entirely):

```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 options (a matter of preference):

1. A nested array of 6 rows × 7 columns instead of a flat 42-slot array (`int[][] boardX = new int[7][6]`) — but this breaks all existing indexing code, so needs a gradual rewrite.
2. Combine `boardX`/`board0` into a single board (`1` = X, `2` = O), avoiding the risk of the two arrays getting out of sync.
3. Since `boardX`/`board0` only ever store `1` or `0`, make them `boolean[]` instead of `int[]`.

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 monolithic `Library` class handling everything.

Considerations when doing this:

- Each class should serve one single-minded purpose (high cohesion).
- Each class should have only one reasonable reason to change (SRP, see java-solid-principles[5]) — e.g. changing the classification system shouldn't require touching bookshelf-rendering code.

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

- Each class's interface should be designed for reasonable programmatic interaction (not too large — see Interface Segregation in java-solid-principles[6]).
- Components should depend on abstractions so sub-components can be substituted/extended later (DIP, see java-solid-principles[7]).

Working

A possible split of `Library` (not exhaustive — a real system could expand on this considerably):

- **`Book`** — data for a single book (title, author, id).
- **`BookShelf`** — tracks all `Book`s in the library and their availability status.
- **`Borrower`** — data for a borrower (name, id) and the books they're currently borrowing.
- **`BorrowingSystem`** — the "bridge" that handles borrowing/returning, passing data between `BookShelf` and `Borrower` rather than either of those two classes depending directly on each other.

Working

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