# Refactoring

## Table of contents


Outline . . . . . 1
Where refactoring sits in the development loop . . . . . 1
Refactoring . . . . . 3
Cohesion and coupling . . . . . 3
SOLID principles . . . . . 3
Key takeaways . . . . . 3
Homework . . . . . 3


See csse2002[1] for course logistics.

Week 6 is one lecture in three parts: **Refactoring**, **Cohesion and Coupling** (Part 02), and **SOLID Principles** (Part 03).

### Outline

1. Good Practice Refactoring, When to Refactor, Code Smells
2. Cohesion and Coupling
3. SOLID Principles

### Where refactoring sits in the development loop

Refactoring is one step in the development cycle, and the cycle is the argument for the whole topic: refactoring is a step you take **only from a passing test**, and it is a step you take **every time round**, not a clean-up project scheduled for later.

See java-testing[2] and java-junit[3] for the test half of that loop.

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/java-testing.pdf
[3] courses/csse2002/java-junit.pdf

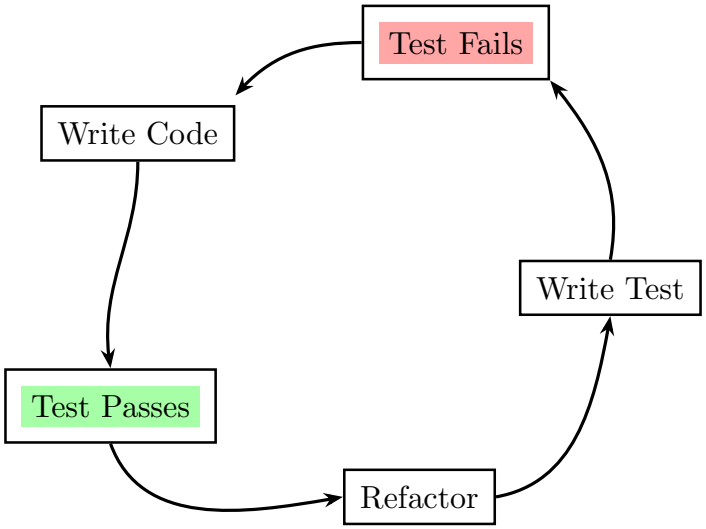


Figure 1: The development cycle. Write Test → Test Fails → Write Code → Test Passes → Refactor → back to Write Test. Refactoring is entered only from a green test and exited into writing the next test, which is why 'have good tests before you begin' is the second rule of good practice. The red and green washes mark the two test verdicts.

## Refactoring

See java-refactoring[4] — what refactoring is, why it's needed, good practice, the two triggers (making a feature easier to *add*, and making code easier to *understand*), and code smells (bad naming, duplication, feature envy). Practised further in week9-lab-refactoring[5].

## Cohesion and coupling

See java-cohesion-and-coupling[6] — modularization, the levels of coupling (content → data) and cohesion (coincidental → functional), four worked coupling examples and two worked cohesion examples, and why high cohesion + low coupling is the goal. Practised further in week10-tutorial-coupling-and-cohesion[7].

## SOLID principles

See java-solid-principles[8] — why large software becomes rigid/fragile, and the first two SOLID principles: Single Responsibility and Open-Closed. The remaining three (Liskov Substitution, Interface Segregation, Dependency Inversion) are covered in 2026-04-16-solid-principles-ii[9].

## Key takeaways

- Refactoring is a **continuous** process, not a one-off tidy-up.
- We need **highly cohesive** and **loosely coupled** classes.
- SOLID principles are **guidelines, not rules**:
  - a class should have only one reason to change;
  - components should be open for extension but closed for modification.

## Homework

- **SRP and memory management** — check whether SRP improves memory management *indirectly*, by improving the overall design structure. Think about object creation, immutability and garbage collection. See java-mutability[10].

[4] courses/csse2002/java-refactoring.pdf
[5] courses/csse2002/week9-lab-refactoring.pdf
[6] courses/csse2002/java-cohesion-and-coupling.pdf
[7] courses/csse2002/week10-tutorial-coupling-and-cohesion.pdf
[8] courses/csse2002/java-solid-principles.pdf
[9] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[10] courses/csse2002/java-mutability.pdf