CSSE2002 — Week 6 Notes

Table of contents

Refactoring

See csse2002[1] for course logistics.

Today's outline

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

Refactoring

See java-refactoring[2] — what refactoring is, why it's needed, good practice, and code smells (bad naming, duplication, feature envy).

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/java-refactoring.pdf

Cohesion and coupling

See java-cohesion-and-coupling[3] — modularization, the levels of coupling (content → data) and cohesion (coincidental → functional), and why high cohesion + low coupling is the goal.

SOLID principles

See java-solid-principles[4] — why large software becomes rigid/fragile, and the first two SOLID principles: Single Responsibility and Open-Closed (the rest are covered in a later lecture).

Reference material

Java Cohesion and Coupling

Introduced in 2026-04-02-refactoring[5] (Lecture, Week 6, Part 2).

Modularization

The process of dividing a large system (a monolith) into smaller, more manageable pieces that can be worked on and tested independently before being reassembled into the final product. This raises a question: what if the resulting modules are highly dependent on each other?

Coupling

The degree of interdependence between different modules, classes, or components of a system.

- **High coupling** — strong interconnections, where a change in one module can cascade through others.
- **Low coupling** — greater independence and isolation between modules.

[3] courses/csse2002/java-cohesion-and-coupling.pdf
[4] courses/csse2002/java-solid-principles.pdf
[5] courses/csse2002/2026-04-02-refactoring.pdf

Levels of coupling (tightly → loosely coupled)

1. **Content** — one class directly manipulates another's data (e.g. public fields instead of private).
2. **Common** — sharing access to the same global data/variables.
3. **External** — sharing something imposed by an external source, e.g. a data format.
4. **Control** — one class controls what happens in another by passing it information/instructions.
5. **Stamp** — sharing a data structure, but each class only needs access to a select part of it.
6. **Data** — sharing data through means such as parameter passing in methods.
7. **None** — no dependency at all (most loosely coupled).

Worked classification examples

Classifying real code against the levels above (my own reasoned classification, following the definitions above — the lecture posed these as open discussion questions without a printed answer key):

```
public void sinh(int x) {...}
public void cosh(int x) {...}
public void tanh(int x) {
    return sinh(x) / cosh(x);
}
```

`tanh` only interacts with `sinh`/`cosh` by passing/receiving simple parameters — **data coupling**.

```
public class R { int x, y, z; char a, b, c; }
public static void compute(R r) {
    return r.x * r.y; // only touches x and y
}
```

`compute` is handed the whole `R` object but only needs two of its six fields — **stamp coupling**.

```
public static void compute(R r) {
    return r.x * r.y * r.z / r.a + r.b + r.c; // uses every field
}
```

Here every field of `R` is actually used, so passing the whole object is fully justified — this is just **data coupling** via a composite value, not stamp coupling (contrast with the previous example, which only used part of the structure).

```
public static runReport(String name, int age, String phone, Date birth, String
 ↪  address) {...}
```

Five separate simple parameters — still **data coupling**, though a long parameter list like this is itself often a separate code smell (see java-refactoring[6]) worth considering bundling into an object.

Cohesion

The degree of interrelatedness and focus among the elements *within* a module, class, or component.

- **High cohesion** — elements are closely related and contribute collectively to one specific functionality.
- **Low cohesion** — elements are less focused, serving multiple unrelated purposes.

Levels of cohesion (low → high)

1. **Coincidental** — elements grouped arbitrarily, with no clear relationship.
2. **Temporal** — elements grouped because they execute at the same time/phase.
3. **Procedural** — elements grouped by their involvement in a specific sequence of steps.
4. **Communicational** — elements work together to manipulate a shared data structure.
5. **Sequential** — elements organised in a linear sequence, where one's output is the next's input.
6. **Functional** — elements grouped around a single, specific functionality or task (most cohesive).

Worked example

```
public void performTasks(int[] numbers, String text) {
    // Task 1: sum the numbers
    int sum = 0;
    for (int num : numbers) { sum += num; }
    System.out.println("Sum of numbers: " + sum);

    // Task 2: reverse the text
    StringBuilder reversedText = new StringBuilder();
    for (int i = text.length() - 1; i >= 0; i--) {
     ↪  reversedText.append(text.charAt(i)); }
    System.out.println("Reversed text: " + reversedText);
}
```

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

Summing numbers and reversing text share no real purpose — they're only in the same method because they happen to run one after another. This is low (coincidental, or at best temporal) cohesion, and a sign `performTasks` should be split into two focused methods.

Is this class cohesive?

> A `Car` class contains: Fuel, Steering, Speed, Route planner, Public holiday calculator.

Fuel/Steering/Speed are all core to a car's own physical behaviour, but a route planner and (especially) a public-holiday calculator are unrelated concerns bolted onto the class — this is **low cohesion**, and a sign the class is trying to do too much (a "God class"); the unrelated pieces should be split into their own classes.

Considerations

- Classes should be **highly cohesive** — a single, easily understood concept.
- Classes should have **loose coupling** to each other — this reduces overall system complexity.

Good modularization (high cohesion, low coupling) groups tightly-interconnected elements together into the same module and minimises connections *between* modules, unlike bad modularization, where connections are scattered across module boundaries with no clear grouping.

Java Refactoring

Introduced in 2026-04-02-refactoring[7] (Lecture, Week 6).

What is refactoring?

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

Why refactor?

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

[7] courses/csse2002/2026-04-02-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[8]).
- 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.

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"` case means editing `travelTime` *and* remembering every other place that branches on `type`:

```
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;
    }
}
```

Refactored to use inheritance/polymorphism (see java-polymorphism[9]), 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; }
}
```

[8] courses/csse2002/java-testing.pdf
[9] courses/csse2002/java-polymorphism.pdf

```
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; }
}
```

Example: making it easier to understand

Extracting named boolean-returning methods (`isHotAndSunny()`, `isPleasant()`) out of inline compound conditions makes the *intent* of a branch obvious at a glance, instead of making the reader re-derive it from raw comparisons every time:

```
// Before
if (temperature > 25 && !isRaining) { ... }
else if (temperature <= 25 && !isRaining) { ... }
else if (isRaining) { ... }

// After
if (isHotAndSunny()) { ... }
else if (isPleasant()) { ... }
else if (isRaining) { ... }
```

Code smells

A "code smell" is a surface indication that usually corresponds to a deeper problem in the design.

Bad naming — single-letter/cryptic names (`r`, `x`, `y`, `g(x)`) force the reader to trace logic to understand intent; descriptive names (`evenCount`, `number`, `isEven(number)`) make code self-documenting.

Duplication — near-identical blocks repeated for different data (e.g. computing an average for two separate arrays with two copy-pasted loops) should be extracted into a single shared method (`average(int[] numbers)`), so a bug fix or improvement only needs to happen once.

Feature envy — a method that spends most of its time reaching into *another* class's data (`cart.getItems()`, `cart.getVoucher()`) rather than its own is a sign the method's logic actually belongs on the class it's "envious" of. Moving `checkout()` onto `ShoppingCart` itself

(next to `getItems`/`getVoucher`) removes the envy and reduces coupling between `User` and `ShoppingCart` (see java-cohesion-and-coupling[10]).

Many other code smells exist — see Martin Fowler's *Refactoring* (2nd ed., 2019), Robert Martin's *Clean Code* (2008), John Ousterhout's *A Philosophy of Software Design* (2018), and the refactoring.com catalog.

Java SOLID Principles

Introduced in 2026-04-02-refactoring[11] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[12] (Lecture, Week 7 — Liskov Substitution, Interface Segregation, Dependency Inversion).

Why SOLID?

Large software tends to become:

- **Rigid** — difficult to change; even small changes require modifications across many parts of the codebase.
- **Fragile** — likely to break when changed, sometimes in parts of the codebase that appear unrelated to the change.

The SOLID principles (S-ingle responsibility, O-pen-closed, L-iskov substitution, I-nterface segregation, D-ependency inversion) are a set of guidelines — not rules — to help prevent rigidity and fragility. Many of the individual ideas predate the acronym; Robert Martin combined them in *Design Principles and Design Patterns* (2000).

Single Responsibility Principle (SRP)

> There should never be more than one reason for a class to change.

Closely related to java-cohesion-and-coupling[13]'s cohesion: cohesion is about how *similar* a class's functionality is, while SRP is about the *reasons for change*. A "responsibility" is a reason for change — if you can think of more than one motive to change a class, it has more than one responsibility.

[10] courses/csse2002/java-cohesion-and-coupling.pdf
[11] courses/csse2002/2026-04-02-refactoring.pdf
[12] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[13] courses/csse2002/java-cohesion-and-coupling.pdf

```java
class Student {
    private String name;
    private List<Course> courses;
    private double gpa;

    public void enrollInCourse(Course course) { courses.add(course); }
    public void calculateGPA() { /* ... */ }
    public void printTranscript() { /* ... */ }
}
```

This `Student` class has (at least) three reasons to change: how enrolment works, how GPA is calculated, and how transcripts are generated. Splitting each concern into its own class gives each one a single responsibility:

```java
class Student {
    private String name;
    private List<Course> courses;
    public void enrollInCourse(Course course) { courses.add(course); }
    public List<Course> getCourses() { return courses; }
}
class GPACalculator {
    public double calculateGPA(List<Course> courses) { /* ... */ }
}
class TranscriptPrinter {
    public void printTranscript(Student student) { /* ... */ }
}
```

Open-Closed Principle (OCP)

> Components should be **open for extension** but **closed for modification**.

Introduced by Bertrand Meyer in *Object Oriented Software Construction* (1988): "if the open-closed principle is applied well, then further changes are achieved by adding new code, not by changing old code that already works".

- **Open for extension** — a class's behaviour can be extended to support changing requirements.
- **Closed for modification** — no one may change the behaviour of an existing class; it's closed once it's available for other modules to depend on.

How do we keep a class closed for modification? Information hiding (see java-encapsulation[14]).

[14] courses/csse2002/java-encapsulation.pdf

Example: pluggable grading systems

A university needs to support multiple grading systems (letter grades, numeric grades). A first attempt branches on a `gradeType` string inside `GPACalculator` — every new grading system means *editing* this method (violates OCP):

```
class GPACalculator {
    public double calculateGPA(List<Course> courses, String gradeType) {
        double totalPoints = 0;
        if (gradeType.equals("Letter")) {
            for (Course course : courses) {
                totalPoints += convertLetterGradeToPoints(course.getLetterGrade());
            }
        } else if (gradeType.equals("Numeric")) {
            for (Course course : courses) {
                totalPoints += course.getNumericGrade();
            }
        }
        return totalPoints / courses.size();
    }
}
```

Extracting the grading logic behind an interface makes `GPACalculator` open for extension (new `GradingSystem` implementations) without ever touching its own code again:

```
interface GradingSystem {
    double getGradePoints(Course course);
}
class LetterGradingSystem implements GradingSystem {
    @Override
    public double getGradePoints(Course course) { /* convert letter grade to points
     ↪  */ return 0; }
}
class NumericGradingSystem implements GradingSystem {
    @Override
    public double getGradePoints(Course course) { return course.getNumericGrade(); }
}

class GPACalculator {
    private GradingSystem gradingSystem;
    public GPACalculator(GradingSystem gradingSystem) { this.gradingSystem =
     ↪  gradingSystem; }

    public double calculateGPA(List<Course> courses) {
        double totalPoints = 0;
        for (Course course : courses) {
            totalPoints += gradingSystem.getGradePoints(course);
```

```
        }
        return totalPoints / courses.size();
    }
}
```

`GPACalculator` now depends on the `GradingSystem` *abstraction* rather than calculating grade points itself — new grading systems (e.g. pass/fail) can be added just by writing a new class that implements `GradingSystem`, with zero changes to `GPACalculator`'s own code.

Liskov Substitution Principle (LSP)

> Subclasses should be substitutable for their parent classes.

Defined by Barbara Liskov in a 1987 keynote: an instance of a subclass type can be used wherever an instance of a superclass type is expected, *without breaking the correctness of the program*:

```
List<Animal> animals = new ArrayList<>();
animals.add(new Cat());
animals.add(new Mouse());
for (Animal animal : animals) {
    animal.eat(new Food());
}
```

Example violation

```
class Student {
    protected List<Course> courses;
    public double calculateGPA() {
        double totalPoints = 0;
        for (Course course : courses) {
            totalPoints += course.getGradePoints();
        }
        return totalPoints / courses.size();
    }
}
class PostgraduateStudent extends Student {
    @Override
    public double calculateGPA() {
        throw new UnsupportedOperationException("Postgraduates have a different GPA
        ↪  system.");
    }
}
```

Code written against `Student` (e.g. calling `calculateGPA()` on every student in a list) will blow up the moment a `PostgraduateStudent` is substituted in — `PostgraduateStudent` is not actually usable wherever a `Student` is expected, so it violates LSP.

Substitution with contracts

Even when a subclass overrides a method with a different implementation, it must still honour the parent's contract (see java-specification[15]):

```
class Parent {
    /**
     * @requires x > 0
     * @ensures \result > 'A' && \result < 'Z'
     */
    char f(int x);
}
```

Preconditions — suppose an overriding `Child.f()` can also handle negative numbers. Changing its precondition from `x > 0` to `x != 0` is fine: every input `Parent.f()` accepted (`x > 0`) is still accepted by `Child.f()` (`x != 0`), since $x > 0 \Rightarrow x \neq 0$ — so the precondition became *weaker* (less strict), not stronger. **Preconditions in a subclass must be no stronger than the superclass's — they may be weaker or stay the same.**

Postconditions — suppose `Child.f()` only ever returns `'K'`, `'L'`, or `'M'`. A postcondition of `\result > 'J' && \result < 'N'` is fine: every result still satisfies the parent's promise (`\result > 'J' && \result < 'N' \Rightarrow \result > 'A' && \result < 'Z'`), so the postcondition became *stronger* (more restrictive), which is allowed. **Postconditions in a subclass must be no weaker than the superclass's — they may be stronger or stay the same.**

> A subclass must not strengthen preconditions or weaken postconditions — it should accept everything the parent accepts, and still guarantee everything the parent guarantees.

Worked example

```
class One {
    /**
     * @require row != null && col != null && 0 < val && val < 100
     * @ensure \result is the list of all of the positive
     *     integers n < val that appear in either row or col
     */
```

[15] courses/csse2002/java-specification.pdf

```
    public List<Integer> calc(int[] row, int[] col, int val) { ... }
}
class Two extends One {
    /**
     * @require row != null && col != null && 0 < val &&
     *     val < 100 && row only contains positive integers
     * @ensure \result is the list of all of the positive
     *     integers n < val that appear in either row or col
     */
    @Override
    public List<Integer> calc(int[] row, int[] col, int val) { ... }
}
```

`Two`'s precondition adds an extra requirement (`row only contains positive integers`) that `One` never required — so `One`'s precondition does *not* imply `Two`'s: there are inputs `One` would accept (a `row` containing a negative number) that `Two` rejects. This **strengthens** the precondition and violates LSP: code written against `One` that happens to pass a negative-containing `row` would break if a `Two` were substituted in.

Homework variant: if `Two`'s postcondition is instead changed to return integers appearing in **both** `row` and `col` (rather than *either*), does this satisfy LSP? Reasoning it through: the guarantee changes from a union to an intersection, which for most inputs returns strictly *fewer* elements than `One` promised — that's a **weaker** postcondition (some outputs a caller was guaranteed under `One` are no longer guaranteed under `Two`), which also violates LSP.

Interface Segregation Principle (ISP)

> Many client-specific interfaces are better than one general-purpose interface.

Shrink interfaces to minimize *incidental dependencies* — large interfaces should be split into smaller ones, so implementing/using classes only need to be concerned about the methods that actually interest them.

```
class Z {
    public void a() {...}
    public void b() {...}
    public void c() {...}
    public void d() {...}
    public void e() {...}
}
// A only uses a() and b(), B only uses c(), C only uses d() and e() ...
// ...but all three classes still depend on the whole of Z.
```

This mirrors java-cohesion-and-coupling[16]'s stamp coupling at the interface level: despite each client using only a small subset of `Z`, every client depends on *all* of `Z` — so a change to any part of `Z` risks affecting (and forcing a recompile/redeploy of) every client, even ones that never used the changed part. Splitting `Z`'s surface into focused interfaces removes those incidental dependencies:

```
interface AI { void a(); void b(); }
interface BI { void c(); }
interface CI { void d(); void e(); }
// Z implements AI, BI, CI; A depends only on AI, B only on BI, C only on CI.
```

Dependency Inversion Principle (DIP)

> Depend upon abstractions. Do not depend upon concretions.

Entities must depend on abstractions, not concretions: high-level modules must not depend on low-level modules — both should depend on abstractions. If `A` depends directly on the concrete class `B`, changes to `B`'s implementation can propagate to `A`; if `A` instead depends only on an abstraction (`BSpec`) that `B` implements, changes inside `B` are far less likely to affect `A`, as long as `BSpec` itself stays the same.

Even a small step towards this helps — programming against the `List` interface instead of the concrete `ArrayList` type means `Student` no longer depends on which list implementation the caller chose:

```
class Student {
    private List<Course> courses; // not ArrayList<Course>
    public Student(List<Course> courses) { this.courses = courses; }
    public double calculateTotalGradePoints() { ... }
}
```

What makes a good dependency? (stability and exposure)

A class `A` has a **dependency** on class `B` if `A` refers to `B` in code (an `import B;`, or, in the same package, simply referring to `B` anywhere). Not all dependencies are equally risky — two properties determine how good/bad one is:

- **Stability** — how likely the dependency is to change. `ArrayList` is a good dependency because it's very unlikely to change; treat classes *you* write as unstable until proven otherwise, and generally assume interfaces are more stable than concrete classes (unless they're poorly designed and change often).

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

- **Exposure** — how much of the depending class is entangled with it. A dependency only used in a constructor is better than one used across every method, since less of the class would need to change if that dependency changed.

Two forms of DIP

Minimising concrete dependencies (the simple form) — change a field's compile-time type from a concrete class to whatever interface it already implements, wherever one already exists:

```
- HardwoodFloor placedOn;
+ Floor placedOn;
```

This only helps where an abstraction already exists, though. The **proper form of DIP** goes further:

> Remove dependencies on low-level components from high-level components — make *both* depend on an abstraction.

"High-level" vs "low-level" isn't a strict distinction: high-level components implement application logic (e.g. a browser's back/forward navigation history); low-level components do the grunt work that logic depends on (e.g. actually requesting a webpage's data). The goal is for high-level components to talk to low-level ones *exclusively* through an abstraction, so a low-level component changing doesn't force the high-level component depending on it to change too. Applied to a `Desk` class depending on a concrete `LogitechKeyboard` with no existing interface:

1. **Create an abstraction** of the low-level component: `interface Keyboard { void type(char key); ... }`.
2. **Modify the low-level component** to implement it: `class LogitechKeyboard implements Keyboard`.
3. **Minimise concrete dependencies** in the high-level component: `Keyboard keyboard = new LogitechKeyboard();`.

Dependency Injection

Dependency Injection (DI) is a design pattern that implements DIP by injecting a class's dependencies into it (e.g. via the constructor) instead of having the class create them internally, promoting loose coupling (see java-cohesion-and-coupling[17]):

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

```java
// Before: AlertService is tightly coupled to EmailSender, and creates its own
↪  dependency.
class AlertService {
    public void sendAlert(String msg) {
        EmailSender sender = new EmailSender();
        sender.send(msg);
    }
}
```

```java
// After: AlertService depends on the MessageSender abstraction, injected via the
↪  constructor.
interface MessageSender {
    void send(String msg);
}
class EmailSender implements MessageSender {
    @Override
    public void send(String msg) { System.out.println("Sending EMAIL: " + msg); }
}
class SmsSender implements MessageSender {
    @Override
    public void send(String msg) { System.out.println("Sending SMS: " + msg); }
}
class AlertService {
    private MessageSender sender;
    public AlertService(MessageSender sender) { this.sender = sender; }
    public void sendAlert(String msg) { sender.send(msg); }
}
```

`AlertService` can now send alerts via email, SMS, or any future `MessageSender` implementation, without ever changing its own code.

A constructor isn't the only injection point — a setter is appropriate when the dependency is likely to change over the object's lifetime, e.g. `public changeSender(MessageSender sender) { this.sender = sender; }`. Eventually *something* (typically an entry-point method like `main`) has to construct the concrete dependencies before injecting them; DI frameworks let you defer that construction to runtime via the framework's own configuration instead of in code, but unless a project needs multiple *levels* of injected dependencies, good design alone can avoid the extra conceptual overhead of adopting one.

Summary

- **S**ingle Responsibility — one reason to change per class.
- **O**pen-Closed — open for extension, closed for modification.

- **L**iskov Substitution — subclasses substitutable for their parents; no stronger preconditions, no weaker postconditions.
- **I**nterface Segregation — many small, client-specific interfaces beat one large general-purpose interface.
- **D**ependency Inversion — depend on abstractions, not concretions; inject dependencies rather than constructing them internally.