CSSE2002 — Week 6 Notes

Table of contents

Refactoring **1**
- Outline . . . 2
- Where refactoring sits in the development loop . . . 2
- Refactoring . . . 3
- Cohesion and coupling . . . 3
- SOLID principles . . . 3
- Key takeaways . . . 3
- Homework . . . 3

Reference material **4**
- Java Cohesion and Coupling . . . 4
- Java JUnit . . . 10
- Java Mutability . . . 16
- Java Refactoring . . . 19
- Java SOLID Principles . . . 27
- Java Testing . . . 38

Refactoring

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).

[1] courses/csse2002/csse2002.pdf

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.

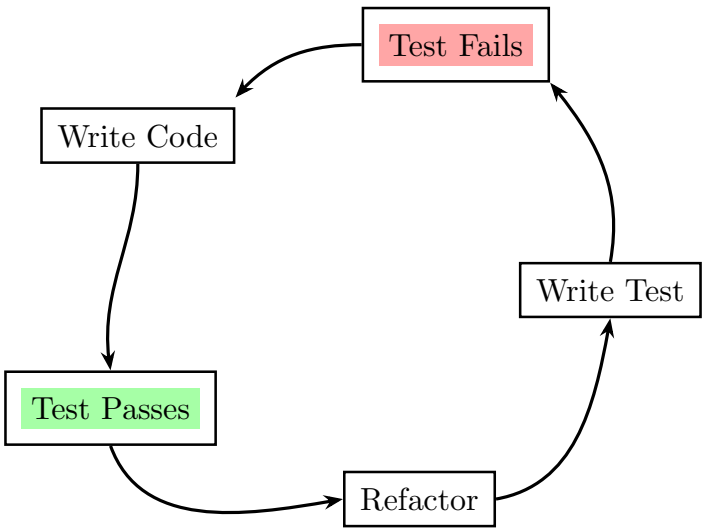

Figure 1: The development cycle. Write Test → Test Fails → Write Code → Test Passes → Refactor → back to Write Test. Refactoring is entered only from a green test and exited into writing the next test, which is why 'have good tests before you begin' is the second rule of good practice. The red and green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

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

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

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

Reference material

Java Cohesion and Coupling

Introduced in 2026-04-02-refactoring[11] (Lecture, Week 6, Part 2). Practised in week10-tutorial-coupling-and-cohesion[12].

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 the question everything below answers: **what if the resulting modules are highly dependent on each other?** Cutting a monolith into pieces buys nothing if every piece still has to know about every other one. The left panel of the coupling figure below shows exactly that failure: six modules tangled together with arrows in all directions.

Two measures separate a real decomposition from a nominal one: **coupling** (how much the modules depend on *each other*) and **cohesion** (how well the contents of *one* module belong together).

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.

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).

[11] courses/csse2002/2026-04-02-refactoring.pdf
[12] courses/csse2002/week10-tutorial-coupling-and-cohesion.pdf

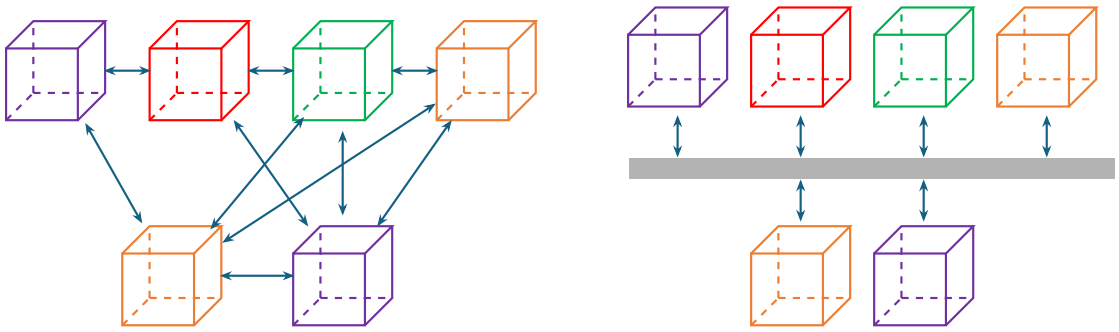

Figure 2: High versus low coupling. Left: every module talks directly to several others, so a change in any one can reach any other. Right: the same six modules each talk only to a shared interface — the grey bar — so nothing propagates further than that bar. A module is a wireframe cube, and its colour only tells the modules apart; a dependency is a teal double-headed arrow. The tangle of arrows on the left is schematic — the count is the point, not which cube joins which.

The list is ordered, and the order is the assessable part: the same two classes can be joined in any of these ways, and the further down the list you can get, the less a change in one can force a change in the other.

Worked classification examples

Four snippets, each with the same question: what type of coupling do these methods have? Each answer is reasoned from the definitions above.

```
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
}
```

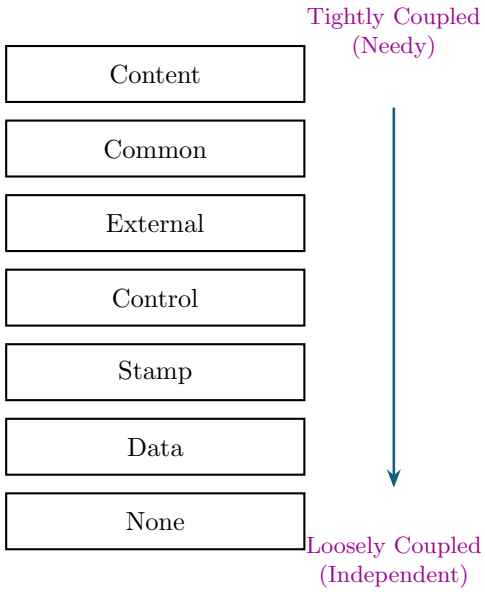

Figure 3: The coupling-levels ladder. Most tightly coupled at the top, least at the bottom, with the arrow running down from tightly coupled (needy) to loosely coupled (independent).

`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[13]) worth considering bundling into an object.

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

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.

Asked of a class, cohesion comes down to two questions: **how well do the parts of the class (state and methods) fit together?** and **do they all contribute to a single, clear purpose?**

> High cohesion is preferable. (Low cohesion usually indicates that a class should be split.)

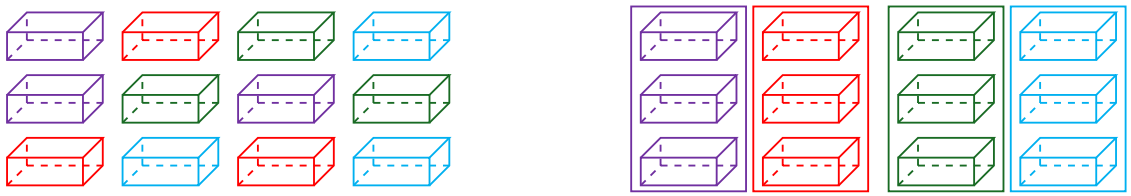

Figure 4: Low versus high cohesion. The same twelve elements in both panels: on the left the four kinds are scattered across the grid, on the right each kind is gathered into one module. Colour means kind only, and each high-cohesion group is enclosed by a rectangle in that group's own colour. This figure says nothing about coupling — it is only about what sits inside one module. The good case is again on the right.

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

```java
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);
}
```

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. The figure below makes this precise, because both panels are the *same system* — the same ten elements and the same fifteen dependencies — grouped into modules two different ways:

Read off the figure, the two measures are the same count seen from opposite sides: a dependency that stays inside a module is cohesion, and a dependency that leaves one is coupling. Maximising the first and minimising the second is one decision — *where to draw the boundaries* — which is why the goal is always stated as a pair.

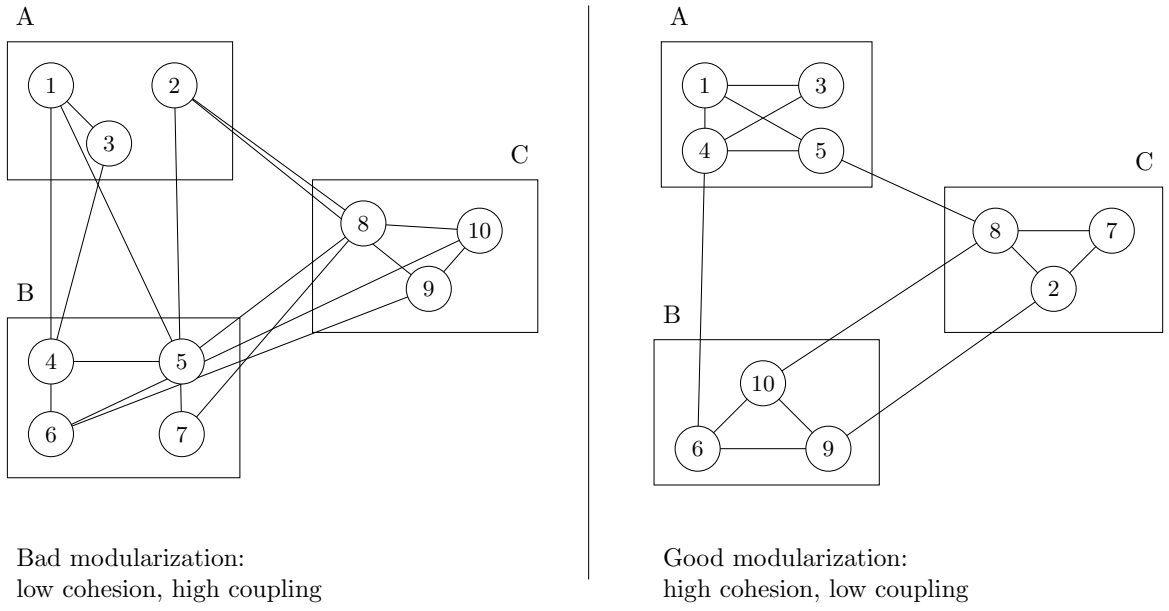

Figure 5: Bad versus good modularization. Both panels draw the same ten elements and the same fifteen dependencies; only the assignment of elements to modules A, B and C differs. Left: 5 dependencies fall inside a module and 10 cross a boundary. Right: 11 fall inside and only 4 cross. Nothing was rewritten to achieve that — the code is identical, the grouping is not.

Java JUnit

Introduced in 2026-03-26-testing[14] (Lecture, Week 5). See java-testing[15] for the broader testing concepts JUnit is used to implement.

What is JUnit?

An open-source test automation framework for Java — one of many *xUnit* frameworks (JUnit, NUnit, CPPUnit, PyUnit…). This course uses **JUnit 4** (http://junit.org/junit4, and the cookbook at https://junit.org/junit4/cookbook.html). Depending on how software is structured, unit-testing frameworks like JUnit can also be used for other kinds of automated testing, but "unit testing" is sometimes *incorrectly* used to mean any automated test.

Writing tests

Tests are defined in classes — **one test class per real class**. `@Test` marks a method as a test JUnit should run; test methods **take no parameters and return `void`**, and should be named for what they check.

The full class and its test class:

```
public class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

[14] courses/csse2002/2026-03-26-testing.pdf
[15] courses/csse2002/java-testing.pdf

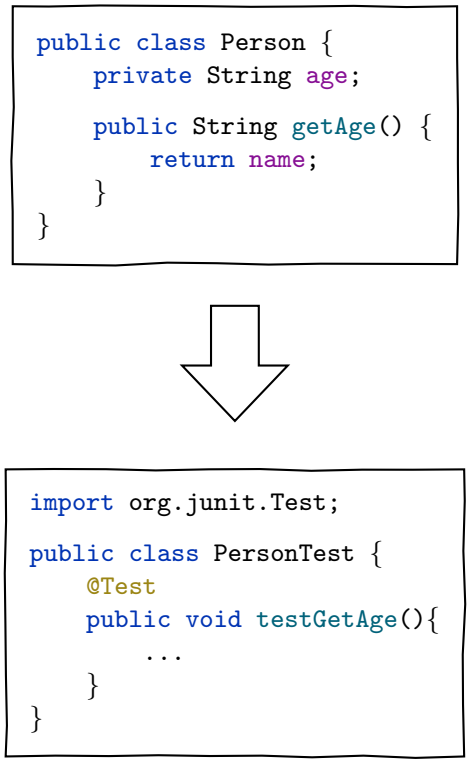

Figure 6: One test class per real class: `Person` produces `PersonTest`, which imports `org.junit.Test` and carries one `@Test`-annotated method per thing being checked. The hollow block arrow pointing down means produces/translates-to. The `Person` snippet does not compile as drawn: `getAge()` returns `name`, which the snippet never declares, and the field is typed `String age`. The full `Person` below types it `int` and is the version that compiles.

```java
import org.junit.Test;
import static org.junit.Assert.assertEquals;

public class PersonTest {

    @Test
    public void TestGetAge() {
        Person p1 = new Person("Jack", 10);
        int age = p1.getAge();
        assertEquals(10, age);
    }

    @Test
    public void TestGetName() {
        Person p1 = new Person("Jack", 10);
        String name = p1.getName();
        assertEquals("Jack", name);
    }
}
```

The test class uses two imports: `org.junit.Test` for the annotation, and a **static** import of `org.junit.Assert.assertEquals` so the assertion can be called bare rather than as `Assert.assertEquals(...)`.

`@Before` and `@After`

Each test method should be independent, but you may want common things set up before each test (e.g. object creation):

```java
public class PersonTest {

    private Person person;

    @Before
    public void setUp() {
        person = new Person("Jack", 10);
    }

    @After
    public void tearDown() {
        person = null;
    }
    // ...
}
```

`@Before` will be run before **each** test; `@After` will be run after **each** test. Every test method gets a freshly built fixture rather than whatever the previous test left behind, which is how the tests are kept independent.

`Assert`

Some static methods in `Assert`:

- `assertEquals` — asserts that two objects are equal.
- `assertArrayEquals` — asserts that two object arrays are equal.
- `assertFalse` / `assertTrue` — asserts that a condition is false or true.
- `assertSame` / `assertNotSame` — asserts that two objects refer to the same object or not.

The difference between `assertSame` and `assertEquals` is reference identity vs `.equals`: two distinct `Person` objects with the same fields can be `assertEquals`-equal (if `equals` is overridden) but are never `assertSame`.

Checking for exceptions

Don't catch the exception — the test runner does. Tell the test runner which exception it should catch, via `@Test(expected = ...)`:

```
@Test(expected = EOFException.class)
public void testExceptions() throws EOFException {
    callThatShouldThrowEOFException();
}
```

`.class` is required after the name of the class. The test **passes** if that exception is thrown and **fails** if it is not, or if a different one is. See java-exceptions[16].

Black box tests in JUnit

Write the tests purely from the specification/Javadoc, without looking at the implementation, then use smoke tests, boundary tests and equivalence classes (see java-testing[17]) to decide what to check. The hard constraint is that **tests must not violate preconditions**. A test that breaks `@requires` tells you nothing, because the method is entitled to do anything at all in that case.

[16] courses/csse2002/java-exceptions.pdf
[17] courses/csse2002/java-testing.pdf

White box tests in JUnit

Design tests based on the **internal logic**. You try to cover branches, loops and edge cases, and you may aim for a code coverage target (e.g. 100%).

The worked example is a discount calculator with two nested levels of branching:

```
public class DiscountCalculator {

    public double calculateDiscount(double amount, boolean isMember) {
        if (amount <= 0) {
            return 0;
        }

        if (isMember) {
            if (amount > 100) {
                return amount * 0.2;
            } else {
                return amount * 0.1;
            }
        } else {
            if (amount > 100) {
                return amount * 0.1;
            } else {
                return 0;
            }
        }
    }
}
```

The five tests, one per leaf:

```
public class DiscountCalculatorTest {

    DiscountCalculator calc = new DiscountCalculator();

    @Test
    public void testAmountLessOrEqualZero() {
        assertEquals(0.0, calc.calculateDiscount(0, true));
        assertEquals(0.0, calc.calculateDiscount(-10, false));
    }

    @Test
    public void testMemberAbove100() {
        assertEquals(40.0, calc.calculateDiscount(200, true));
    }
```

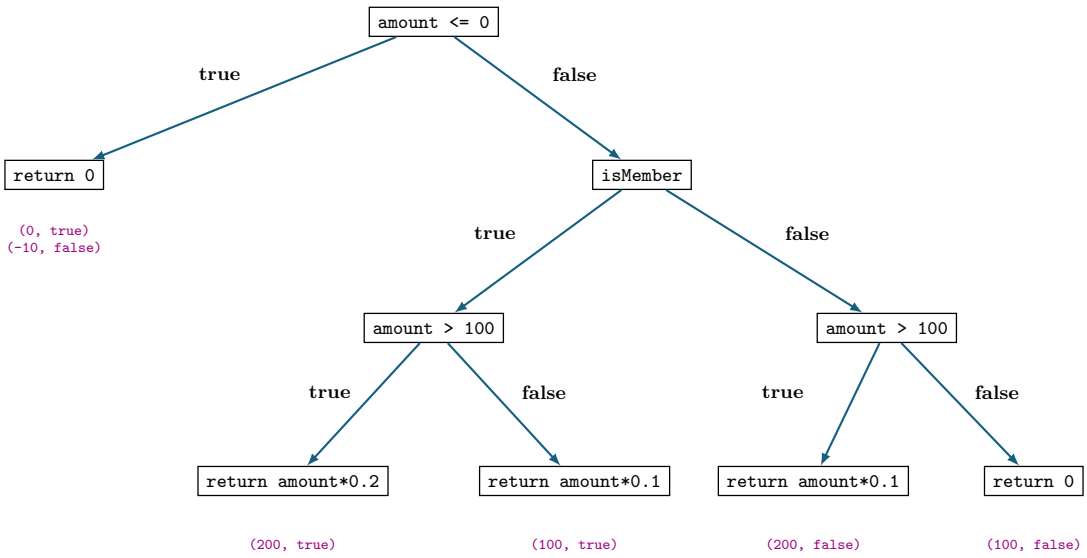

Figure 7: Why `DiscountCalculator` needs exactly five tests: the method as a decision tree, one leaf per test. Each box is a condition from the method or the `return` its branch reaches, so the expected values in the tests can be read off the leaves. The magenta `(amount, isMember)` pair under each leaf is the argument list of the test that reaches that leaf. The early return on `amount <= 0` is the one leaf covered by two assertions in a single test, which is why it carries two pairs.

```
    @Test
    public void testMemberBelowOrEqual100() {
        assertEquals(10.0, calc.calculateDiscount(100, true));
    }

    @Test
    public void testNonMemberAbove100() {
        assertEquals(20.0, calc.calculateDiscount(200, false));
    }

    @Test
    public void testNonMemberBelowOrEqual100() {
        assertEquals(0.0, calc.calculateDiscount(100, false));
    }
}
```

Two things worth reading off this example:

- The tests are **named for the branch they cover**, not for the method (`testMemberAbove100`, not `testCalculateDiscount3`), which is what naming a test for what it checks looks like in practice.
- `amount = 100` is the argument in two of the five tests. It is the boundary of `amount > 100`, so it is both a white-box branch probe and a black-box boundary test.

> **i** assertEquals on doubles
>
> The tests above call `assertEquals(0.0, ...)` on `double` values. JUnit 4's two-argument `assertEquals(double, double)` is deprecated precisely because floating-point equality is unreliable; the three-argument `assertEquals(expected, actual, delta)` is the form that compiles without a deprecation warning. The code above will warn if you type it in.

Java Mutability

Introduced in 2026-03-12-object-oriented-programming-ii[18] (Lecture, Week 3). Builds on java-encapsulation[19].

Mutable vs immutable classes

Objects in Java can either change their state after creation or remain fixed forever.

Two types of class design:

[18] courses/csse2002/2026-03-12-object-oriented-programming-ii.pdf
[19] courses/csse2002/java-encapsulation.pdf

- **Mutable class** — object state *can* change after creation.
- **Immutable class** — object state *cannot* change after creation.

Mutable

State is reachable through setters, so it can change at any point in the object's life:

```
public class Person {
    private String name;
    private int age;

    public void setName(String name) { this.name = name; }
    public void setAge(int age) { this.age = age; }
}
```

```
class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Alice");
        p.setAge(20);

        p.setAge(30);   // state changed
    }
}
```

This object can change at any time.

Immutable

The immutable version changes four things at once: the class is `final`, both fields are `final`, the values arrive through the constructor, and there are no setters — only getters.

```
public final class Person {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public int getAge() { return age; }
}
```

```java
class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Alice", 20);
        Person p2 = new Person("Bob", 30);
        System.out.println(p1.getName() + " " + p1.getAge());
    }
}
```

Changing a value now means constructing a new object, not mutating the old one.

Why `final` on the class

A subclass can introduce mutability (**Effective Java (2nd ed.), Item 16, page 78**). If the class is subclassable, a subclass can add mutable state or override methods to return values that vary, so callers can no longer rely on "an instance of this type never changes". Marking the class `final` is what forecloses that.

Arrays and lists break naive immutability

Even if a class looks immutable, it may not be.

`final` on an array field only prevents **reassigning the reference**. It does nothing to protect the array's *contents*, and a getter that returns the field hands the caller a live handle on the object's own state:

```java
public final class Person {
    private final String name;
    private final String[] phoneNumbers;

    public Person(String name) {
        this.name = name;
        this.phoneNumbers = new String[]{"111", "222"};
    }

    public String getName() { return name; }

    public String[] getPhoneNumbers() { return phoneNumbers; }
}
```

```java
class Main {
    public static void main(String[] args) {
        Person person = new Person("John");
        person.getPhoneNumbers()[0] = "999999";
```

```
        System.out.println(person.getPhoneNumbers()[0]);   // 999999
    }
}
```

Every field is `private final` and there is not a setter in sight, yet the object's state changed. The rule:

- `final` for arrays means the **reference** cannot change, not the contents.
- `final` prevents *reassigning* the array:

```
this.phoneNumbers = new String[]{"123"}; // NOT allowed if already assigned
```

Defensive copying

To protect mutable objects, create copies — on the way in *and* on the way out, so neither the constructor's caller nor the getter's caller keeps a handle on the field (**Effective Java, Item 50: "Make defensive copies when needed"**):

```
public final class Person {
    private final String name;
    private final String[] phoneNumbers;

    public Person(String Name, String[] phoneNumbers) {
        this.name = Name;
        this.phoneNumbers = phoneNumbers.clone();   // defensive copy
    }

    public String[] getPhoneNumbers() {
        return phoneNumbers.clone();                // defensive copy
    }
}
```

`clone()` on an array is a shallow copy — it duplicates the row of references, not the objects they point at. That is enough here because `String` is itself immutable, but a `Person[]` of mutable `Person` objects would still leak.

Java Refactoring

Introduced in 2026-04-02-refactoring[20] (Lecture, Week 6). Practised in week9-lab-refactoring[21].

[20] courses/csse2002/2026-04-02-refactoring.pdf
[21] courses/csse2002/week9-lab-refactoring.pdf

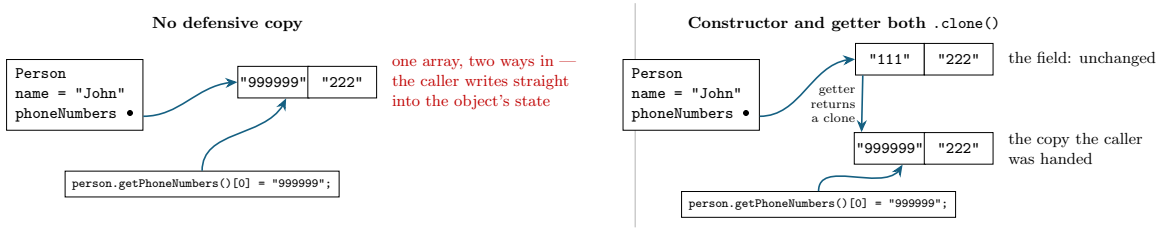

Figure 8: Why a class whose getter returns its array field is not immutable, and what `.clone()` changes. Left: the field and the getter's return value are the same array, so the caller's write lands in the object's state. Right: the getter hands out a copy, so the write lands somewhere the object does not care about.

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.

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[22], java-junit[23]). This is what puts `Refactor` where it is in the development cycle, drawn in 2026-04-02-refactoring[24]: 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.

[22] courses/csse2002/java-testing.pdf
[23] courses/csse2002/java-junit.pdf
[24] courses/csse2002/2026-04-02-refactoring.pdf

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;
    }
}
```

Refactored to use inheritance/polymorphism (see java-polymorphism[25]), 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; }
}
```

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

```java
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:

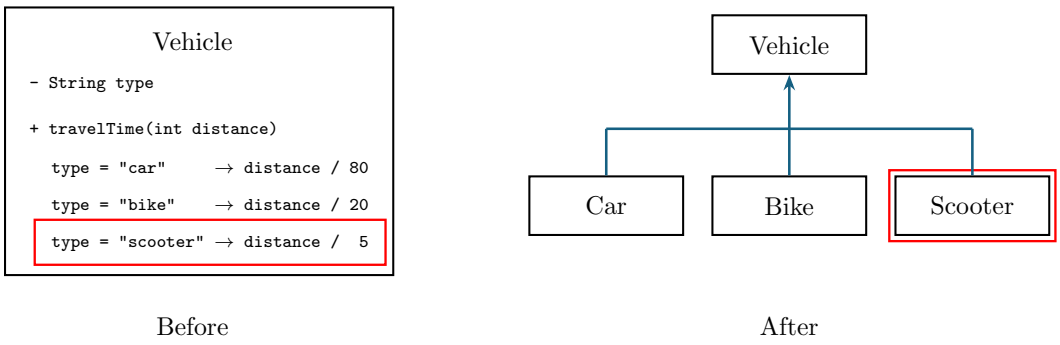

Figure 9: Adding a scooter, before and after. Left: one class, and the new type is another branch spliced into an existing method — the red outline marks the edit. Right: an abstract Vehicle with one subclass per type, joined by a connector with a single arrowhead into the parent. The new type is a whole 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 `Vehicle` abstraction being open to new subclasses without being edited is the Open-Closed Principle arriving early — see java-solid-principles[26].

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:

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

```
// 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.

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();
    }
}
```

```java
// 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:

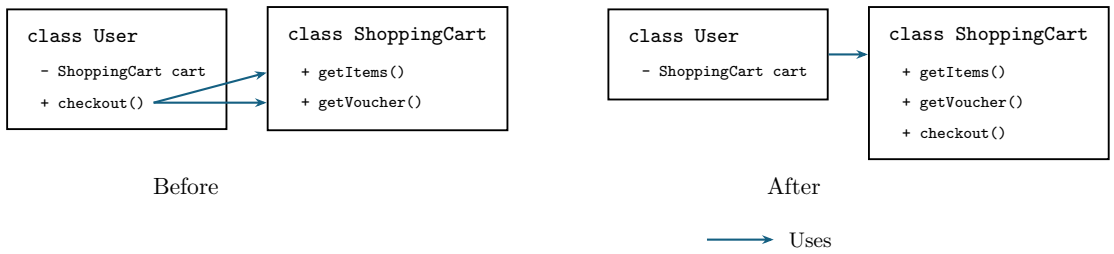

Figure 10: Feature envy, before and after. Before: User.checkout() reaches across the class boundary into two of ShoppingCart's members. After: checkout() sits with the data it uses, and User's only link to ShoppingCart is the field it holds. Each box is a class with its members listed as '+ m()'. A solid arrow means 'uses' and points from the depender to the thing depended upon; it may land on an individual member row.

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[27].

Others

Many other code smells exist. Further reading:

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

- 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.

Java SOLID Principles

Introduced in 2026-04-02-refactoring[28] (Lecture, Week 6, Part 3 — Single Responsibility and Open-Closed) and continued in 2026-04-16-solid-principles-ii[29] (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).

Week 6 covers the first two: **Single Responsibility** and **Open-closed**.

Single Responsibility Principle (SRP)

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

Closely related to java-cohesion-and-coupling[30]'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.

[28] courses/csse2002/2026-04-02-refactoring.pdf
[29] courses/csse2002/2026-04-16-solid-principles-ii.pdf
[30] courses/csse2002/java-cohesion-and-coupling.pdf

```
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 class has three reasons to change:

1. Changing how to enrol to courses.
2. Changing how overall GPAs are calculated.
3. Changing how transcripts are generated.

Splitting each concern into its own class gives each one a single responsibility:

```
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) { /* ... */ }
}
```

Two details of the split are worth noticing. The `gpa` field is gone: it was a *cached result* of one of the three responsibilities, so it leaves with the responsibility, and `GPACalculator.calculateGPA` returns a value instead of assigning one. And `Student` gains `getCourses()`, because the two extracted classes now need the data from outside — the responsibilities separate cleanly, but they do not become independent.

The same shape, stated more briefly:

```
// Before                          // After
class Report {                     class ReportCalculator { }
    void calculateData() {}        class ReportSaver { }
    void saveToFile() {}           class ReportPrinter { }
    void printReport() {}
}
```

Calculating, saving and printing are three different reasons to change one `Report` class — a new output format, a new storage backend and a new formula are unrelated events — so they become three classes.

Homework. Check whether SRP improves memory management *indirectly*, by improving overall design structure — think about object creation, immutability (see java-mutability[31]) and garbage collection.

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.

A class is kept closed for modification by information hiding (see java-encapsulation[32]).

Example: pluggable grading systems

The example picks up the `GPACalculator` extracted under SRP above, which assumes a single grading scheme:

```
class GPACalculator {
    public double calculateGPA(List<Course> courses) {
        double totalPoints = 0;
        for (Course course : courses) {
            totalPoints += course.getGradePoints();
        }
        return totalPoints / courses.size();
    }
}
```

Now a requirement arrives: the university needs to handle different grading systems, such as letter grades (A, B, C) and numerical grades (90, 80, 70). A first attempt branches on a `gradeType` string:

[31] courses/csse2002/java-mutability.pdf
[32] courses/csse2002/java-encapsulation.pdf

```java
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();
    }

    private double convertLetterGradeToPoints(String letterGrade) {
        // Convert letter grade to GPA points logic
        return 0;
    }
}
```

Three things carry the problem: the added `String gradeType` parameter, and the two string literals `"Letter"` and `"Numeric"`. They are every place that has to be *modified* the next time a grading system is added, in a class that was supposed to be finished. The class is open for extension only by editing it, which is exactly what "closed for modification" forbids. (A `gradeType` string also reintroduces the `Vehicle`-style type switch from java-refactoring[33].)

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

```java
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;
```

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

```
    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();
    }
}
```

The caller picks the grading system and hands it to the calculator:

```
public static void main(String[] args) {
    List<Course> letterGradeCourses = new ArrayList<>();
    letterGradeCourses.add(new Course("CS", "A"));

    GradingSystem letterGradingSystem = new LetterGradingSystem();
    GPACalculator gpaCalculator = new GPACalculator(letterGradingSystem);
    System.out.println("GPA (Letter Grades): " +
     ↪  gpaCalculator.calculateGPA(letterGradeCourses));
}
```

What the refactoring bought:

- `GPACalculator` depends on the abstraction `GradingSystem` instead of calculating grade points itself — which makes it open for extension (new grading systems) but closed for modification.
- More grading systems can be added without changing the core logic of `GPACalculator`.
- The interface guarantees every grading system implements the same method (`getGradePoints()`), making it easy to switch between them.
- `main` demonstrates using either `LetterGradingSystem` or `NumericGradingSystem` by passing a different grading system to the `GPACalculator`.

Two arrows into an abstraction that nobody edits is also how the class stays closed: the calculator is closed for modification because the *only* thing it can see of a grading system is the interface. That is information hiding used structurally, and it is the same move as the Dependency Inversion Principle below; the constructor parameter here is dependency injection.

Liskov Substitution Principle (LSP)

> Subclasses should be substitutable for their parent classes.

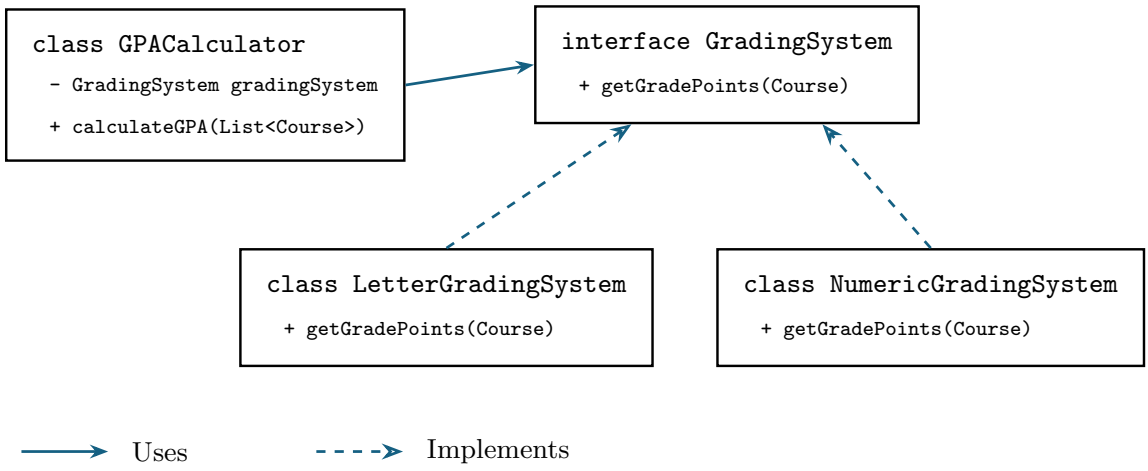

Figure 11: The open-closed refactoring of GPACalculator. GPACalculator uses only the GradingSystem interface; each concrete grading system implements it. A new scheme — pass/fail, say — is a fourth box on the bottom row with a fourth dashed arrow, and no existing box changes: that is 'open for extension, closed for modification' as a picture. Each label carries the Java keyword and members are listed as '+ m()'. A solid line with a solid head means 'uses', a dashed line with an open head means 'implements', and every arrow points from the depender to the thing depended upon. This is not UML, which would draw realisation as a hollow triangle.

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[34]):

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

```
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
     */
    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[35]'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.
```

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

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).
- **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[36]):

```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); }
```

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

```
}
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.

Java Testing

Introduced in 2026-03-26-testing[37] (Lecture, Week 5).

[37] courses/csse2002/2026-03-26-testing.pdf

Levels of testing

1. **Unit testing** — check that each "unit" in the project behaves correctly.
2. **Integration testing** — check that components work together and that the interfaces between components work as expected.
3. **System testing** — does the system as a whole work correctly?
4. **(User) acceptance testing** — do users of the system agree that the system does what it's supposed to do?

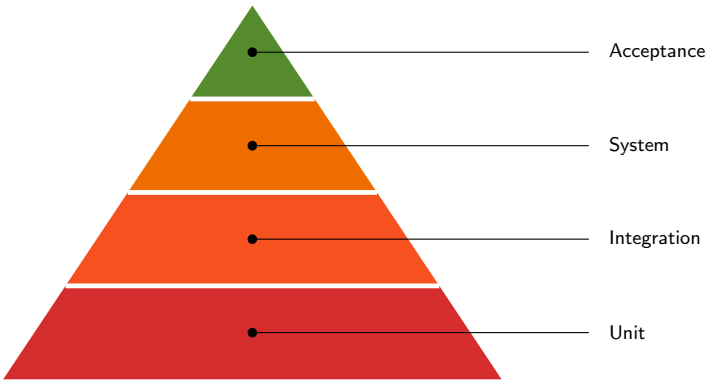

Figure 12: The four levels of testing as a pyramid. Unit tests form the wide base, acceptance tests the narrow apex. In the usual test-pyramid reading, the width of a band stands for how many tests of that kind a project has.

Test frameworks

Libraries that make writing and running tests easier. **xUnit** is the prevalent family — JUnit, NUnit, CPPUnit, PyUnit, … — and this course uses **JUnit 4** (http://junit.org/junit4); see java-junit[38].

Depending on how the software is structured, a unit-testing framework can also be driven for other kinds of automated testing. A caveat: *"unit testing" is sometimes incorrectly used to mean any automated test.*

Regression testing

During development, testing helps answer (at least) two questions:

- Does the new stuff work?

[38] courses/csse2002/java-junit.pdf

- Have we broken things which used to work — have we **regressed**?

Regression testing ensures that old features are still working as they should when new features are introduced.

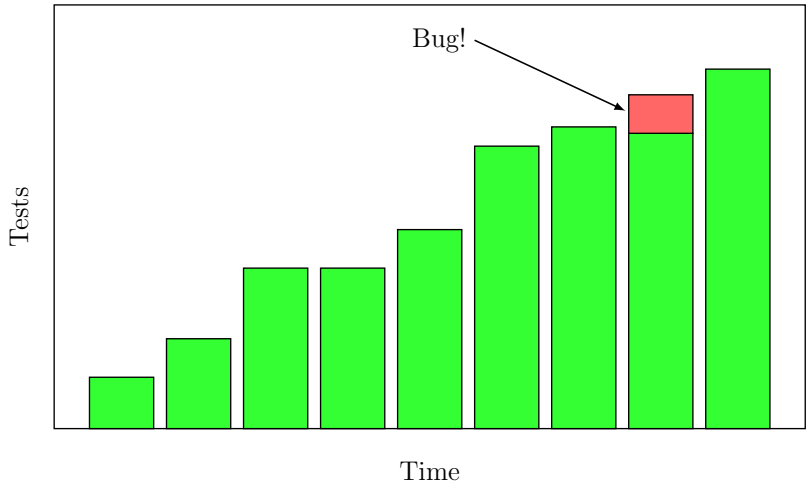

Figure 13: A regression (chart after browserstack.com/guide/regression-testing). Nine bars on *Tests*/*Time* axes: read each bar as a build, and its height as the size of the test suite growing as features are added. The red top slice on the second-to-last bar is the regression, and the arrow labelled Bug! points at it.

Black box vs white/glass box testing

- **Black box** — the software has inputs and outputs to test, but the implementation is unknown to you; you test according to the *specification* (see java-specification[39]) — what it's supposed to do.
- **White/glass box** — testing designed *with* knowledge of the internal implementation, so tests can pay special attention to cases where the implementation is complicated.

Black box techniques

Black box testing works from the Javadoc alone. The running example is a bus, and the *specification is all you get*:

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

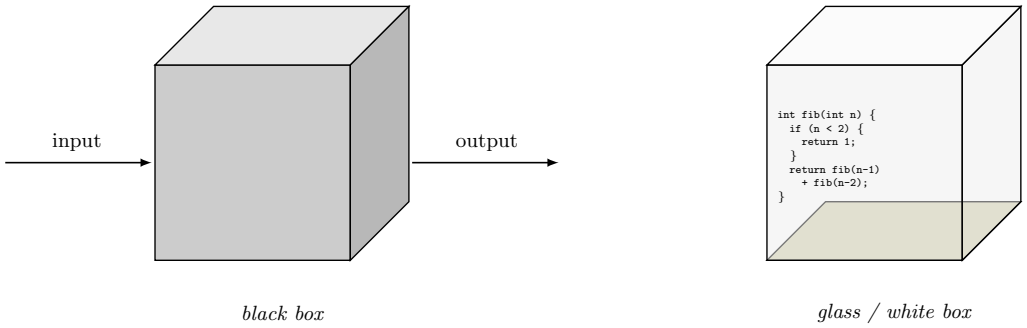

Figure 14: The two testing stances. Both are the same grey cube. The black box is opaque and shows only its input and output arrows. The glass box is translucent, so the implementation inside, a recursive `fib`, can be seen through its walls.

```
/**
 * Calculate how many passengers are currently on the bus.
 *
 * @returns The number of passengers on the bus.
 */
public int getCurrentPassengers() {...}

/**
 * Calculate the average number of passengers on the bus between stops.
 *
 * @returns The average passengers on the bus.
 */
public int getAverageCount() {...}
```

> **Note**
>
> The tag is written `@returns` here and `@return` in the specification tag table from the exceptions lecture. The course uses both spellings — see csse2002-diagram-notation[40], "Specification tag spelling".

Smoke tests — test the common-case functionality. Usually the *first* test to be performed ("can it run?"), used to determine whether further, more rigorous testing is required. The name comes from electronic hardware testing: plug in a new board, turn on the power, and if you see smoke coming from the board, turn the power off — you don't have to do any more testing. (From *Lessons Learned in Software Testing.*)

Boundary tests — investigate extremes and corner cases, i.e. where bugs often occur:

40 courses/csse2002/csse2002-diagram-notation.pdf

Input type	Try
Whole number	0, -1, minimum value, maximum value
Floating point	0, -1, `NaN`, infinity
Collection (array, list, set, etc.)	empty, one element, large
Reference type	`null`
Resource (file, link)	non-existent resource

Equivalence classes — groups of inputs that the system treats the same way. For `getCurrentPassengers()` on a bus whose maximum capacity is 80, build the table in two passes: first the classes, then the boundary tests for each class.

Class	Values	Boundary tests
Valid – Empty	0	-1, 0, 1
Valid – Normal	1 – 79 (any number within the range)	Covered by above and below
Valid – Full capacity	80	79, 80, 81
Invalid	–	-1, 81

Minimal boundary set: `-1, 0, 1, 79, 80, 81`. The "Normal" class contributes no probes of its own because its edges are already the outer probes of the classes on either side.

The black-box method

The method in six points:

1. Black box testing: no knowledge of implementation, only specification.
2. JUnit is a test framework.
3. Tests must not violate preconditions.
4. Start with smoke tests.
5. Expand test suite with boundary tests.
6. Stop when confidence is high.

Point 3: a test that breaks `@requires` proves nothing, because the method is free to do anything at all once its precondition is violated. Point 6: testing is never exhaustive, so you stop on judgement rather than on a completeness proof. The loop argument below makes the same admission concrete.

White box technique: code coverage

With tests *and* an implementation, code coverage measures how much of the code is actually tested: of all the different ways your program could run, how many are covered by your tests. There are three levels, in this order:

1. **Statement coverage** — each statement is executed.
2. **Branch coverage** — each branch is tested for the `true` case and the `false` case.
3. **Path coverage** — every 'path' through the code is traversed.

A worked example:

```
public void register(int x) {
    if (x > 0) {
        positives += 1;
    }
    if (x % 2 == 0) {
        evens += 1;
    }
}
```

- **Statement coverage:** any odd positive value, e.g. `x = 1`, gives a partial run: `if (x > 0)`, `positives += 1` and the `if (x % 2 == 0)` test are executed while `evens += 1` is missed. `x = 2` alone executes every statement, since it is positive *and* even so both `if` bodies run.
- **Branch coverage:** `x = 2` makes both conditions `true` and `x = -1` makes both `false`, so `x = 2` and `x = -1` together give each condition both outcomes, `true, false`. Two tests.
- **Path coverage:** all four combinations of the two conditions, so four tests. The four inputs are in the table below.

Written out, the four paths and their inputs are:

Path	x
`x > 0 && x % 2 == 0`	2
`x > 0 && !(x % 2 == 0)`	1
`!(x > 0) && x % 2 == 0`	-2
`!(x > 0) && !(x % 2 == 0)`	-1

The three levels run from weakest to strongest, and this exercise makes that concrete: the same `x = 2` that gives full *statement* coverage misses three of the four paths. That is why "100% statement coverage" is a weak claim.

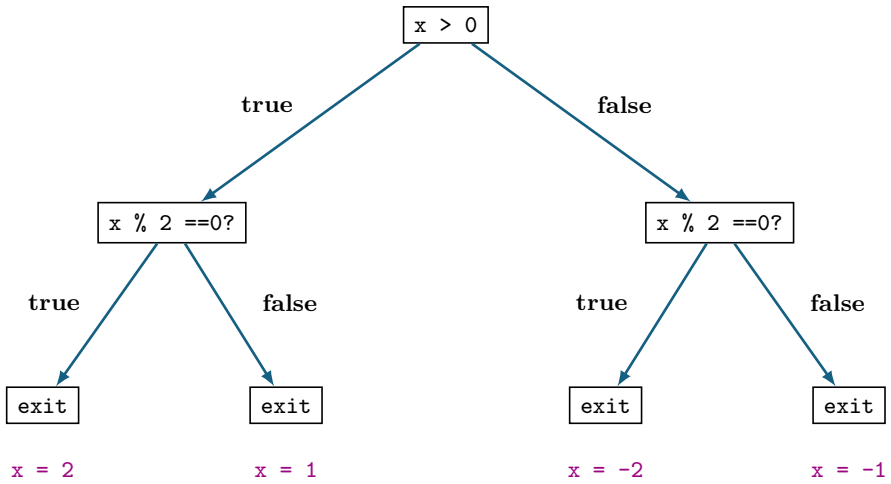

Figure 15: Path coverage for `register`. Each internal box is a condition, each teal arrow an outcome labelled true/false, each leaf one complete path to `exit` — four leaves, so four tests. The magenta input under each leaf is a value of `x` that drives that path.

Loops make exhaustive path coverage infeasible

```
for (int i = 0; i < 100; i++) {
    if (f(i, k)) {
        j++;
    }
}
```

There are 2^{100} paths through this program — 1,267,650,600,228,229,401,496,703,205,376 of them. At a billion tests per second that is 40.2 trillion years, roughly 2900 times the age of the universe.

So path coverage is **approximated**. The common approximation is that **2 or more iterations through a loop, or recursive calls, are equivalent**. This is *engineers' induction*: "One, two, three — that's good enough for me." For `for (init; test; iterate) { body }` the three cases to test are:

- **0 iterations:** `init`; `test` (**fails**); `nextStatement`;
- **1 iteration:** `init`; `test`; `body`; `iterate`; `test` (**fails**); `nextStatement`;
- **2 iterations:** `init`; `test`; `body`; `iterate`; `test`; `body`; `iterate`; `test` (**fails**); `nextStatement`;

Using JUnit for white-box testing

Design tests based on internal logic. You try to cover **branches**, **loops** and **edge cases**, and you may aim for a code coverage target (e.g. 100%). A worked white-box example — `DiscountCalculator`, the five tests that cover it, and its branch tree — is in java-junit[41].

Test Driven Development (TDD)

The TDD approach originates from the Agile manifesto principles and Extreme Programming. In TDD, developers create small test cases for every feature based on their initial understanding. The primary intention is to **modify or write new code only if the tests fail** — this prevents duplication of test scripts. Use the tests to determine when code is "ready".

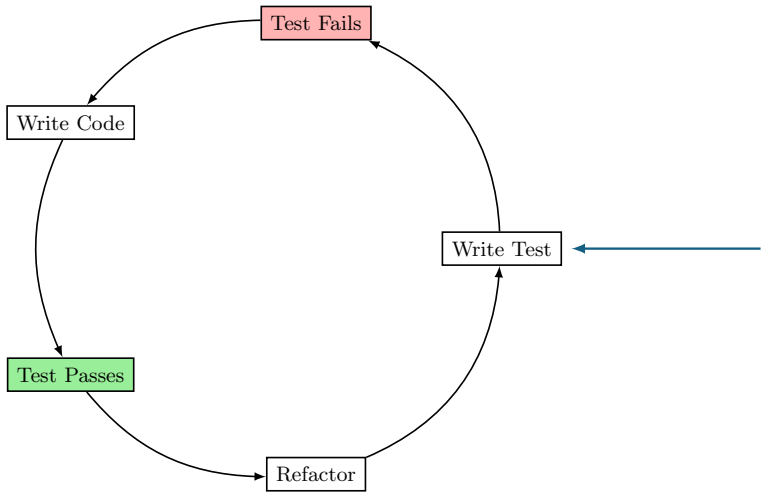

Figure 16: The TDD cycle. Read it counter-clockwise: write a test, watch it fail (pink), write code, watch it pass (green), refactor, and go round again. The teal arrow on the right points at *Write Test* and marks the entry point of the loop: the test comes before the code.

The *Refactor* step connects to java-refactoring[42]: the green test suite is what makes it safe to restructure code without changing its behaviour.

[41] courses/csse2002/java-junit.pdf
[42] courses/csse2002/java-refactoring.pdf