Coupling and Cohesion

Table of contents

Coupling 1
Cohesion — `Customer` 5
Cohesion — `Employee` 6

Applied class for Week 10 (no lecture this week). More practice applying java-cohesion-and-coupling[1] (Week 6).

Coupling

> **Class coupling**: the strength of the connection or dependence between classes — to what extent does this class depend on other classes? How many methods are called on how many other classes? Can another object influence the flow of control in this object?

Assume the following classes are all in separate files in the same package:

```java
public class X {
    public int num = 5;
    protected Z z;

    public X(Z z) { this.z = z; }

    public void doThis() { sayHello(); }

    public void sayHello() { System.out.println("Hello"); }
}
```

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

```java
public class Y extends X {
    public Y(Z z) { super(z); }

    public void doThat() { this.z.sayHello(); }

    @Override
    public void sayHello() { super.sayHello(); }
}
```

```java
public class Z {
    private X x = new X();

    public void setNum(int num) { x.num = num; }

    public void sayHello() { System.out.println("Hello"); }
}
```

These classes are tightly coupled, to a degree that is possibly unrealistic — every pair of them is connected. Identify the points of coupling between: (0) `X` and `Y`; (1) `X` and `Z`; (2) `Y` and `Z` — you don't need to name the type/level of coupling (see java-cohesion-and-coupling[2] for the ladder from content coupling down to none), just where it occurs.

As written, `Z`'s `private X x = new X()` would not compile, because `X` declares only `X(Z z)` and so has no no-argument constructor. The exercise is about locating coupling, not about compiling the listing.

Working

Class	Coupling	Implication
`X`	stores a `Z` object	`X` is coupled to `Z`
`X`	constructor takes a `Z`	`X` is coupled to `Z`
`X`	`doThis()` just calls `sayHello()`, which is overridden in `Y`	forwarding behaviour
`Y`	constructor takes a `Z`	`Y` is coupled to `Z`
`Y`	constructor calls `super(z)`	`Y` is coupled to `X`
`Y`	`doThat()` accesses the protected `this.z`	`Y` is coupled to `Z` and `X`
`Y`	`sayHello()` calls `X.sayHello()` via `super`	`Y` is coupled to `X`

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

`Z`	stores an `X` object	`Z` is coupled to `X`
`Z`	`setNum()` accesses the public `X.num`	`Z` is coupled to `X`

Drawn as a dependency graph, the table says that **every** pair of these three classes is joined, and that `X` and `Z` are joined in *both* directions — a dependency cycle, so neither can be understood, tested or reused without the other:

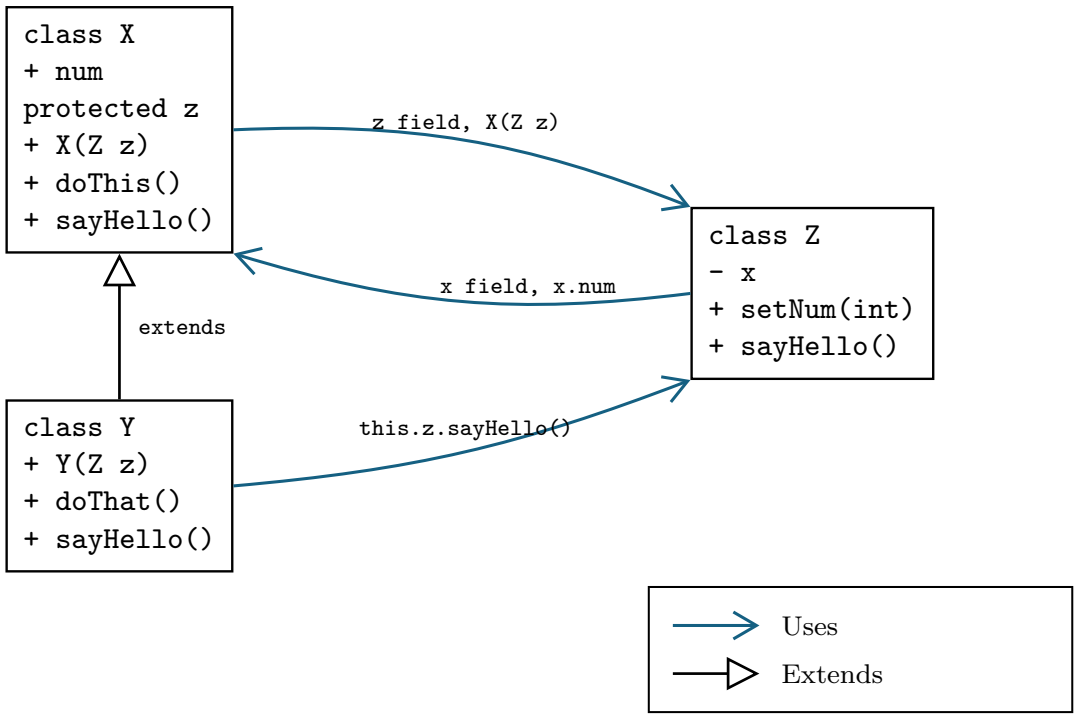

Figure 1: The coupling points of Task 0-2 collected into a dependency graph. The table's nine rows become four edges: X and Z point at each other, Y points at Z, and one extends edge carries every X-Y row at once (super(z), super.sayHello(), and doThis() forwarding to a sayHello() that Y overrides). A solid teal arrow means 'uses' and points from the depender to the thing it depends on. The black line with an unfilled triangle head is the generalisation arrow, with the superclass above. Members carry the visibility prefixes `-` (private) and `+` (public), and X's protected field is spelled out in words.

Two points the table makes that Figure 1 makes visible at once:

- The `X`–`Z` pair is a **cycle**. `X` holds a `Z` and takes one in its constructor; `Z` holds an `X` and writes straight into its public `num` field.
- `Y`'s single line `this.z.sayHello()` reaches a `Z` through a field it did not declare, so it couples `Y` to `Z` *and* to `X` at the same time — which is why the table gives that one row two implications.

Cohesion — `Customer`

> **Class cohesion**: how well components support a central purpose — how focused the components of a unit are. How well do the parts of the class (state and methods) fit together? Do they all contribute to a single, clear purpose?

```java
public class Customer {
    private String name;
    private String streetAddress;
    private String suburb;
    private String postCode;
    private List<Item> orders; // the products that have been ordered

    public Customer(String name, String streetAddress, String suburb, String
     ↪  postCode) {
        this.name = name;
        this.streetAddress = streetAddress;
        this.suburb = suburb;
        this.postCode = postCode;
        this.orders = new ArrayList<>();
    }

    public String getName() { return name; }
    public String getMailingAddress() { return streetAddress + suburb + postCode; }
    public void newOrder(List<Item> order) { orders.addAll(order); }
    public List<Item> getAllOrder() { return orders; }
}
```

Does `Customer` exhibit high or low cohesion? Justify your answer.

> **Working**

Low cohesion:

- `name` is only used once, in a single getter.
- `streetAddress`/`suburb`/`postCode` are only used once, in a single getter — and these aren't unique to a `Customer`, so belong in a separate class.
- `orders` (with its getter and add method) isn't cohesive with the rest of `Customer`'s representation, and could live in a dedicated `Order`-related class.

The grouping of member variables looks coincidental, and the methods have no clear relationship to each other's functionality — the state and methods don't contribute to a single, clear purpose.

If `Customer` doesn't have high cohesion, design replacement classes with higher cohesion.

> **Working**
>
> `Customer` is really trying to be at least three concepts at once: a customer, a mailing address, and a customer's order history. Extract a postal-address abstraction (that `Customer` stores an instance of), and an order-history abstraction (stored by `Customer`, or managed separately).

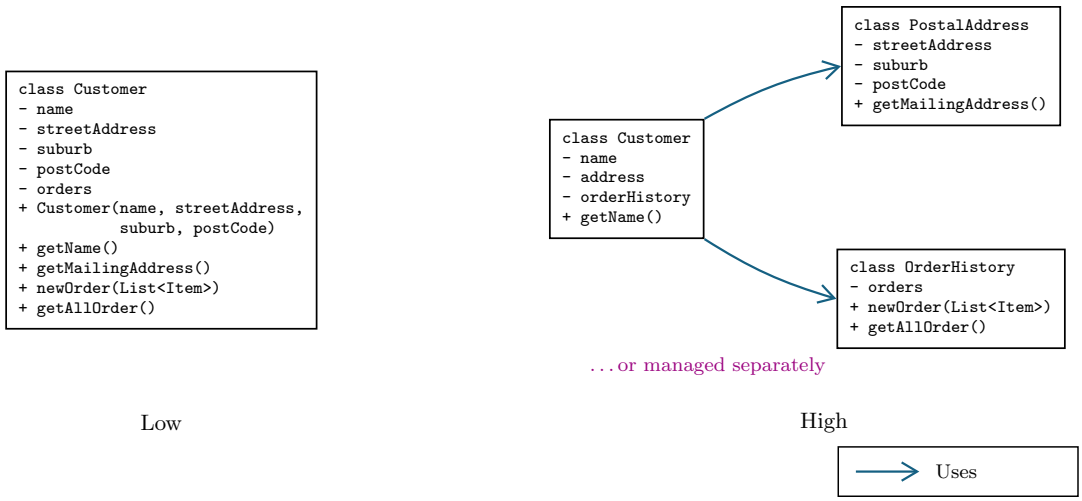

Figure 2: Customer as written, and split into its three concepts: a customer, a mailing address (PostalAddress) and a customer's order history (OrderHistory). The solid teal arrow means 'uses' and should not be read as UML aggregation or composition. The magenta note says that the order history may equally be managed outside Customer. The weaker design is on the left and the stronger on the right, each labelled with one word beneath.

The point of Figure 2 is that nothing is deleted and no behaviour changes — the same five fields and four methods are still there. All that moves is *which class owns them*, and with it the number of reasons any one class has to change.

Cohesion — `Employee`

```
public class Employee {
    private String firstName;
    private String surname;
```

```
    private String homeAddress;
    private String suburb;
    private String postCode;
    private String currentRole;
    private int currentRoleSecurityLevel;
    private int hourlyWage;

    public Employee(String fName, String lName, String homeAddress,
                    String suburb, String postCode, int hourlyWage) {
        this.firstName = fName;
        this.surname = lName;
        this.homeAddress = homeAddress;
        this.suburb = suburb;
        this.postCode = postCode;
        this.hourlyWage = hourlyWage;
    }

    public String getName() { return surname + ", " + firstName; }
    public String getMailingAddress() {
        return String.format("%s%n%s%n%s", homeAddress, suburb, postCode);
    }
    public void setRole(String newRole, int securityLevel) {
        currentRole = newRole;
        currentRoleSecurityLevel = securityLevel;
    }
    public String getCurrentRole() { return currentRole; }
    public boolean accessAllowed(int requiredSecurityLevel) {
        return currentRoleSecurityLevel >= requiredSecurityLevel;
    }
    public int getPay(int hoursWorked) { return hourlyWage * hoursWorked; }
    public void setHourlyWage(int newWage) { hourlyWage = newWage; }
}
```

Does `Employee` exhibit high or low cohesion? Justify your answer.

> **i** Working
>
> Low cohesion:
>
> - `firstName`/`surname` are only set in the constructor and returned by a single getter — better as a dedicated personal-details class, or simply a single `name` string.
> - The address fields are assigned once and used in a single getter — they appear arbitrarily grouped in this class and could live in a separate object.
> - The role fields (`currentRole`, `currentRoleSecurityLevel`) are used across several methods, but never interact with the name or address fields — this functionality could move to a `Role` class, shrinking `Employee`'s constructor and letting other

objects reuse `Role`.

`Employee` is really trying to wrap the functionality of at least two other objects (address, role) inside itself; since the grouping of state appears coincidental rather than purposeful, `Employee` has low cohesion.
"The grouping appears coincidental" is a claim you can check mechanically: tabulate which method reads or writes which field, and see whether the marks form one connected mass or several disjoint blocks.

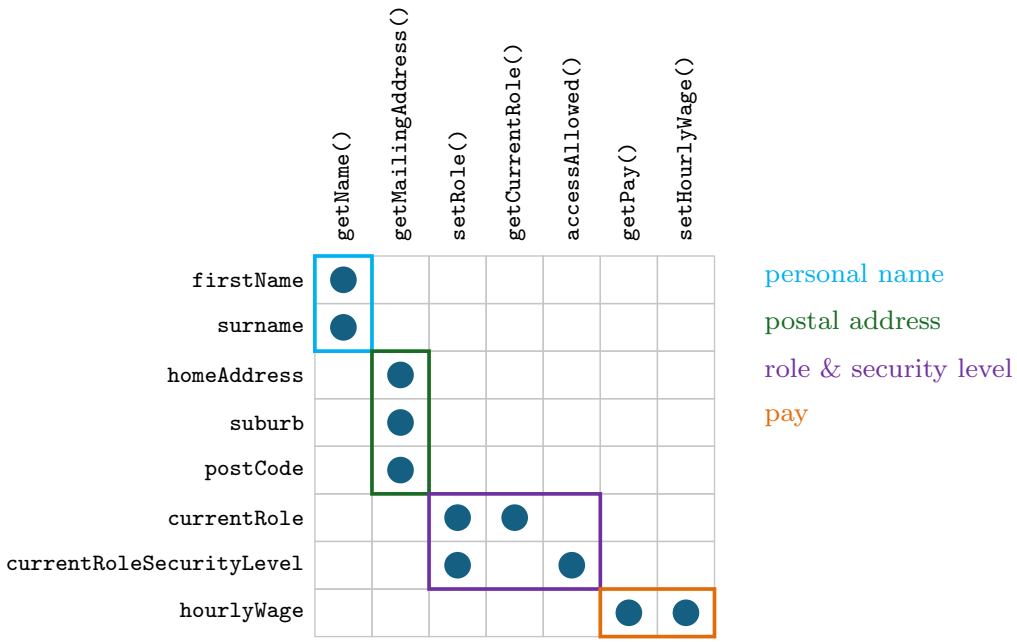

Figure 3: Which Employee method touches which Employee field. The constructor is deliberately left out: it assigns every field by definition, so it says nothing about cohesion. The result is four disjoint blocks with no mark outside them: no method in this class touches fields from two different blocks, which is exactly what 'the grouping of state appears coincidental' means, and exactly why each block can be lifted out into a class of its own. Block outline colours carry group identity only.

Compare the same test applied to a cohesive class: there the marks would spread across the whole grid, because the methods would share the state they operate on. Here they do not overlap at all — Figure 3 is four small classes drawn inside one large one.