Dependency Inversion

Table of contents

Applied class for 2026-05-14-lambdas-streams-and-events[1] (Week 11). More practice applying Dependency Inversion and Dependency Injection (see java-solid-principles[2]).

Dependencies

> A class `A` has a **dependency** on class `B` if `A` refers to `B` in code — an `import B;`, or (if in the same package) simply referring to `B` anywhere in the code.

```
import furniture.Lamp;

class Desk {
    HardwoodFloor placedOn;
    List<Junk> items = new ArrayList<>();
    LogitechKeyboard keyboard = new LogitechKeyboard();

    public Desk() {
        placedOn = new HardwoodFloor();
        items.add(new Book("Pragmatic Programmer"));
        items.add(new Monitor());
    }

    public void rotate(Angle angle) {
        if (angle == Angle.LEFT) {
            placedOn.scratch("clockwise");
        }
```

[1] courses/csse2002/2026-05-14-lambdas-streams-and-events.pdf
[2] courses/csse2002/java-solid-principles.pdf

```
        // implementation
    }

    public boolean clean() {
        // implementation
    }
}
```

List the classes `Desk` depends upon.

> **i** Working
>
> `Lamp`, `HardwoodFloor`, `Junk`, `List`, `ArrayList`, `Book`, `Monitor`, `Angle`, `LogitechKeyboard`.

Not all dependencies are equally bad. Two properties determine how good/bad a dependency is:

- **Stability** — how likely the dependency is to change. `ArrayList` is a good dependency because it's very unlikely to change; in general, treat classes *you* write as unstable until proven otherwise. Interfaces are usually assumed more stable than concrete classes (unless they're poorly designed and change often).
- **Exposure** — how much of the class uses the dependency. `Book` (used only in the constructor) is a better dependency than `HardwoodFloor` (used in any method) simply because less of the class is entangled with it.

Roughly order `Desk`'s dependencies from best to worst.

> **i** Working
>
> 1. `List` (interface, low exposure)
> 2. `ArrayList`
> 3. `Lamp` (assuming it isn't actually used anywhere — the `import` is unused)
> 4. `Book`
> 5. `Monitor`
> 6. `Angle` — likely an `enum`, so reasonably assumed stable
> 7. `LogitechKeyboard`
> 8. `HardwoodFloor`
> 9. `Lamp` (assuming it actually *is* used somewhere, contradicting the "unused import" assumption above)

Dependency Inversion

> **Dependency Inversion Principle (DIP)**: depend on abstractions, not concretions (implementations).

Minimising concrete dependencies (the simple form)

The simplest step towards DIP: change a field's compile-time type from a concrete class to whatever interface it implements (assuming the interface exposes everything the field is used for):

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

```
public class Warrior {
    private BronzeSword weapon = new BronzeSword();
    private BronzeShield shield = new BronzeShield();

    public void attack(Opponent opponent) { weapon.use(opponent); }
    public void defend(Attack attackType) { shield.block(attackType); }
}
```

Given a hierarchy `Weapon` (interface) ← `Sword` ← `BronzeSword`/`GoldSword`, and a standalone concrete `BronzeShield` (no interface):

Minimise concrete dependencies for `Warrior`.

> Working
>
> ```
> - private BronzeSword weapon = new BronzeSword();
> + private Weapon weapon = new BronzeSword();
> ```
>
> (`BronzeShield` can't be minimised this way yet — there's no interface for it, see the proper form below.)

The proper form of DIP

Minimising concrete dependencies only helps where an abstraction already exists. The *proper* form of DIP is:

> 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 the high-level logic depends on (e.g. actually requesting a webpage's data). The goal is for high-level components to talk to low-level components *exclusively* through an abstraction, so a low-level component changing doesn't force a change to the high-level component depending on it.

Applying this to `Desk`'s `LogitechKeyboard` dependency (`LogitechKeyboard` has no existing interface to fall back on):

1. **Create an abstraction** of the low-level component:

```
interface Keyboard {
    void type(char key);
    // include other ways to interface with the low-level component;
    // multiple interfaces can be used if it has multiple responsibilities
}
```

2. **Modify the low-level component** to depend on (implement) the abstraction:

```
- class LogitechKeyboard {
+ class LogitechKeyboard implements Keyboard {
```

3. **Minimise concrete dependencies** in the high-level component:

```
- LogitechKeyboard keyboard = new LogitechKeyboard();
+ Keyboard keyboard = new LogitechKeyboard();
```

Apply the proper form of DIP to `Warrior`'s `BronzeShield` dependency.

> Working

```
interface Shield {
    void block(Attack attack);
}
```

```
- class BronzeShield {
+ class BronzeShield implements Shield {
```

```
- private BronzeShield shield = new BronzeShield();
+ private Shield shield = new BronzeShield();
```

Dependency Injection

Even after minimising concrete dependencies, `Desk` still depends on the concrete types when *constructing* them. **Dependency Injection (DI)** completes DIP: provide concrete instances to a class as parameters, rather than having the class instantiate them itself:

```
- public Desk() {
-     placedOn = new HardwoodFloor();
+ public Desk(Floor floor) {
+     placedOn = floor;
```

`Desk` no longer needs to change to support a different floor type — the caller just passes a different `Floor` implementation in. A constructor isn't the only injection point — a setter is appropriate when the dependency is likely to change over the object's lifetime:

```
public changeKeyboard(Keyboard keyboard) {
    this.keyboard = keyboard;
}
```

Modify `Warrior` so its concrete types are dependency injected.

> Working

```
public class Warrior {
    private Weapon weapon;
    private Shield shield;

    public Warrior(Weapon weapon, Shield shield) {
        this.weapon = weapon;
        this.shield = shield;
    }

    // it's quite likely a warrior will change their weapon
    public void equip(Weapon weapon) { this.weapon = weapon; }

    // overloading gives a nice consistent interface for this action
    public void equip(Shield shield) { this.shield = shield; }

    public void attack(Opponent opponent) { weapon.use(opponent); }
    public void defend(Attack attackType) { shield.block(attackType); }
}
```

`weapon`/`shield` are injected via the constructor (initial equipment) *and* can be swapped later via the overloaded `equip(...)` setters (since it's plausible a warrior changes weapon

> or shield mid-game) — this is setter injection layered on top of constructor injection.

Dependency injection frameworks

Something (typically an entry-point method, e.g. `main`) still has to construct the concrete dependencies before injecting them. DI frameworks let you defer that construction to runtime, specifying which concrete classes to build via the framework's own configuration rather than in code. Unless a project needs multiple *levels* of injected dependencies, good design alone can avoid the extra conceptual overhead of adopting a DI framework.