Dependency Inversion

Table of contents

Dependencies 1
Dependency Inversion 3
Dependency Injection 7

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], where the principle itself is stated).

(The handout is headed *Week 10*. It is the week 11 applied class.)

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) {
```

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

```
        if (angle == Angle.LEFT) {
            placedOn.scratch("clockwise");
        }
        // 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`
> 2. `ArrayList`
> 3. `Lamp` (assuming that it isn't used anywhere)
> 4. `Book`
> 5. `Monitor`
> 6. `Angle`
> 7. `LogitechKeyboard`
> 8. `HardwoodFloor`
> 9. `Lamp` (assuming that it can be used anywhere)
>
> One caveat: since `Angle` appears to be an `enum` type, we may reasonably assume it is

> stable and may consider it a better dependency.

`Lamp` appears twice because it is ranked under both assumptions: the `import` is there but no use of `Lamp` is visible in the listing, so its exposure is either nil (best) or unbounded (worst). The caveat makes `Angle` a better dependency, which would move it up from sixth.

Dependency Inversion

Dependency inversion is a tool for building software that is more flexible and amenable to future changes.

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

To apply it to `Desk` you first need to know which of its dependencies already have an abstraction behind them. In the diagram, a dashed box indicates an interface.

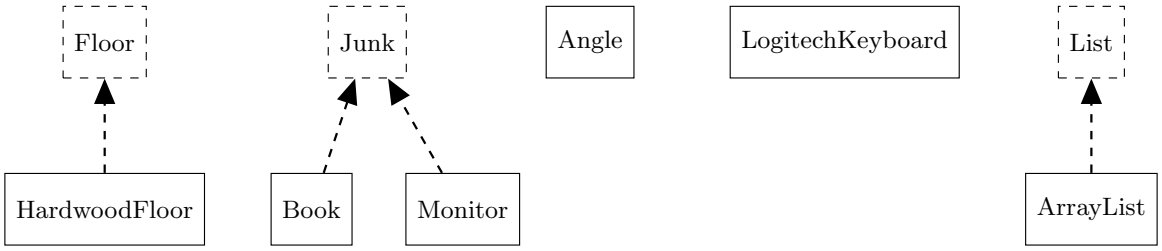


Figure 1: The structure behind Desk's dependencies. It carries eight of the nine dependencies listed above - Lamp is absent - plus the new Floor interface. A dashed box indicates an interface. A dashed line with a solid filled arrowhead runs from a concrete class up to the interface it implements.

Read off the diagram: `HardwoodFloor`, `Book`, `Monitor` and `ArrayList` each already sit under an interface, so the simple form of the DIP applies to them immediately. `Angle` and `LogitechKeyboard` sit alone, with no abstraction to switch to. `LogitechKeyboard` is the one fixed below, with the proper form of the DIP.

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); }
}
```

Minimise concrete dependencies for `Warrior`, based on the class diagram below.

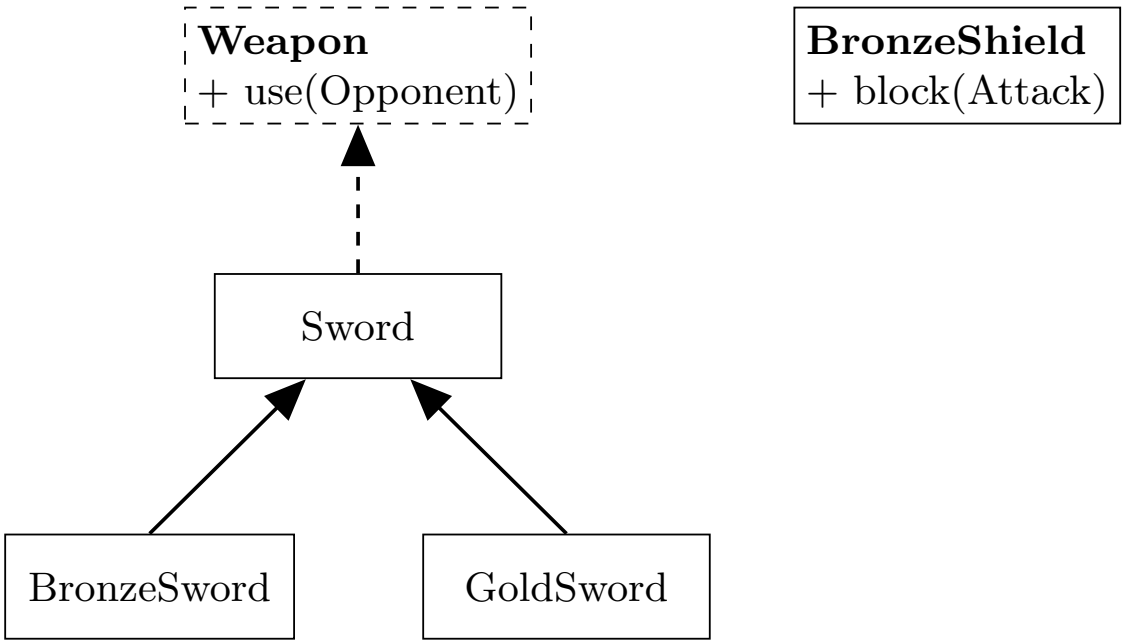


Figure 2: The class diagram for Task 2: dashed box = interface (Weapon), solid box = concrete class, dashed arrow = implements, solid arrow = extends, arrowheads solid-filled and the supertype above. Weapon and BronzeShield are the only two boxes that list a member, on the line below the name. Those two members are the only two methods Warrior actually calls.

`BronzeSword` has `Sword` above it and `Weapon` above that; `BronzeShield` stands alone, with no interface to retreat to.

Working

```
- private BronzeSword weapon = new BronzeSword();
+ private Weapon weapon = new BronzeSword();
```

The solution touches only `weapon`; `shield` has to wait for 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:

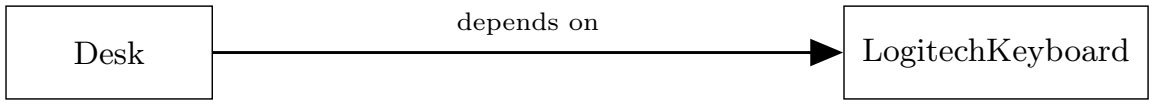
```
interface Keyboard {
    void type(char key);
    // include other ways to interface with the low-level component
    // multiple interfaces can be used if the low-level component
    // has multiple responsibilities
}
```

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();
```



Before: *the high-level class names the low-level one*



After: *both ends point at the abstraction — the right-hand edge has reversed*

Figure 3: What is inverted in dependency inversion, for the Desk-to-LogitechKeyboard example. A dashed box is an interface and a dashed arrow with a filled head means implements. A depends-on edge is a solid line with a solid head, pointing from the depender to what it depends upon.

The name of the principle is in what those three steps do to the arrows:

`Desk` still depends on something, and `LogitechKeyboard` is still the thing that runs, yet the edge that used to run *from* the high-level component *to* the low-level one now runs the other way — from the low-level component up to the shared abstraction. That reversal is the "inversion".

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 that a warrior will change their weapon
    public void equip(Weapon weapon) { this.weapon = weapon; }

    // overloading gives us 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` and `shield` are injected via the constructor *and* can be swapped later through the overloaded `equip(...)` setters — 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.