Dependency Injection Made Simple

What we'll talk about:

Dependencies

Dependency Injection pattern

Dependency Inversion principle

Unit tests

What is a Dependency?

A depends on B

Examples

→ **Image Loader** depends on **HTTP**

Examples

→ **Image Loader** depends on **HTTP**

→ **REST** depends on **HTTP**

Examples

→ **Image Loader** depends on **HTTP**

→ **REST** depends on **HTTP**

→ **REST** depends on **(de)serializer**

Examples

→ **Image Loader** depends on **HTTP**

→ **REST** depends on **HTTP**

→ **REST** depends on **(de)serializer**

→ **Sign up form** depends on **all of the above**

What is Dependency Injection?

No dependency injection:

```
public class Example {
    private final Dependency dependency;

    public Example() {
        dependency = new Dependency();
    }
}
```

No dependency injection:

```
// public class Example {
//     private final Dependency dependency;

    public Example() {
        dependency = new Dependency();
    }
// }
```

With dependency injection:

```
public class Example {
    private final Dependency dependency;

    public Example(Dependency dependency) {
        this.dependency = dependency;
    }
}
```

With dependency injection:

```
// public class Example {
//     private final Dependency dependency;

    public Example(Dependency dependency) {
        this.dependency = dependency;
    }
//}
```

```
public Example() {
    dependency = new Dependency();
}

// vs

public Example(Dependency dependency) {
    this.dependency = dependency;
}
```

Method injection

```
public class Example {
    private final Dependency dependency;

    public Example() { }

    public void setDependency(Dependency dependency) {
        this.dependency = dependency;
    }
}
```

But why?

Share dependencies

Configure dependencies externally

```
Context wrappedContext =
    new ContextThemeWrapper(context, R.style.MyTheme);

View view = new View(wrappedContext);
```

AppCompat

Dependency Inversion principle

Dependency Inversion principle

1. High-level modules should not depend on low-level modules. Both should depend on abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

Dependency Inversion principle

1. ~~High-level modules should not depend on low-level modules. Both should depend on abstractions.~~
2. ~~Abstractions should not depend on details. Details should depend on abstractions.~~

A depends on B

A is coupled to B

Copy

Read Keyboard

Write Printer

```
void copy() {
    char c;
    while ((c = readKeyboard()) != -1) {
      writePrinter(c);
    }
}
```

What if you want to read from disk?

```
enum InputDevice {
    KEYBOARD, DISK
}

void copy(InputDevice input) {
    char c;
    while (true) {
        if (input == InputDevice.KEYBOARD) {
            c = readKeyboard();
        } else if (input == InputDevice.DISK) {
            c = readDisk();
        } else {
            throw new IllegalArgumentException("Bad input device!");
        }

        if (c == -1) return;

        writePrinter(c);
    }
}
```

What if you want to write to the monitor?

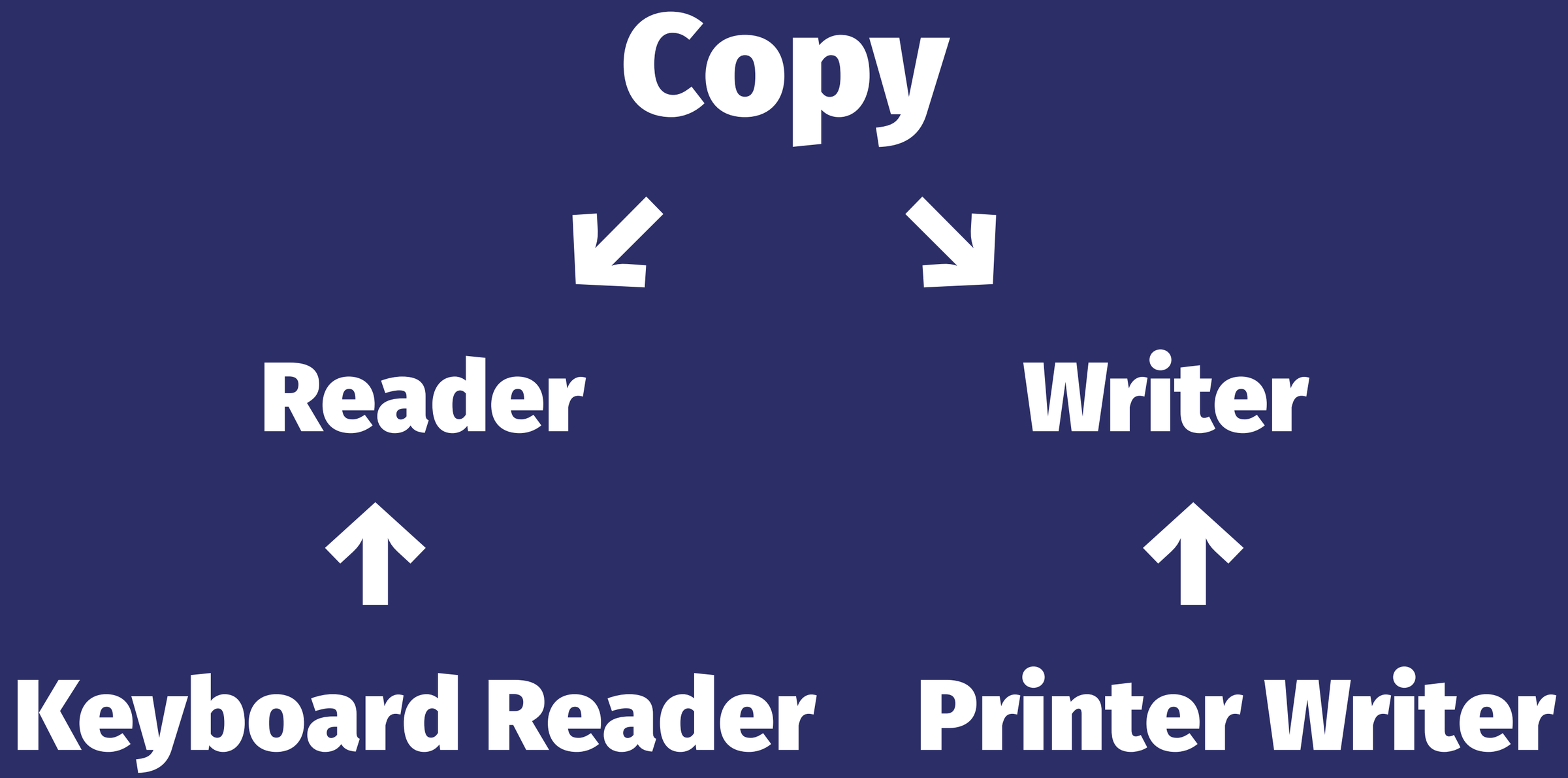
```
enum OutputDevice {
    PRINTER, MONITOR
}

void copy(OutputDevice outputDevice) {
    char c;
    while ((c = readKeyboard()) != -1) {
        if (outputDevice == OutputDevice.PRINTER) {
            writePrinter(c);
        } else if (outputDevice == OutputDevice.MONITOR) {
            writeMonitor(c);
        } else {
            throw new IllegalArgumentException("Bad output device!");
        }
    }
}
```

What do we test in the read/write modules?

How do we test copy?

Dependency Inversion to the rescue


Copy
Reader
Writer
Keyboard Reader
Printer Writer

```
interface Reader {
    char read();
}

interface Writer {
    void write(char c);
}

void copy(Reader reader, Writer writer) {
    char c;
    while ((c = reader.read()) != -1) {
        writer.write(c);
    }
}
```

But why?

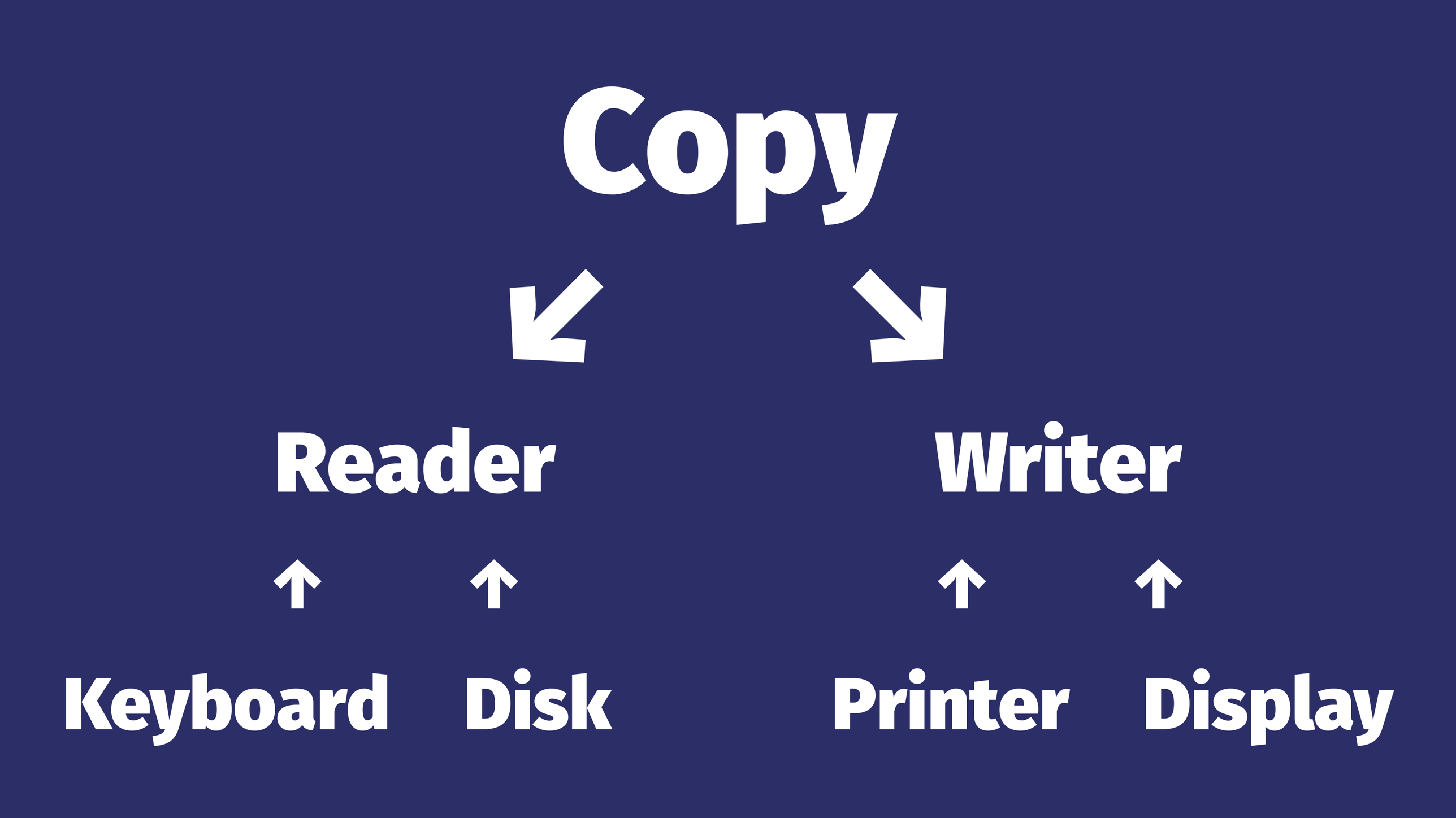
But why?

→ Share dependencies

But why?

→ Share dependencies

→ Configure dependencies externally


Copy
Reader
Writer
Keyboard
Disk
Printer
Display

But why?

→ Configure dependencies externally

→ Share dependencies

→ Clean separation of modules

But why?

→ Configure dependencies externally

→ Share dependencies

→ Clean separation of modules

→ Easier testing

```
@Test
public void testCopy() {
    TestReader reader = new TestReader("Hello, world!");
    TestWriter writer = new TestWriter();
    copy(reader, writer);
    assertEquals("Hello, world!", writer.getWritten());
}
```

But why?

→ Configure dependencies externally

→ Share dependencies

→ Clean separation of modules

→ Easier testing

→ Easier debugging / development

Why not?

→ More verbose

→ More complex

Dependency Inversion

Is about how you structure the code to take advantage of contracts in the middle

Dependency Inversion

Is about how you structure the code to take advantage of contracts in the middle

Dependency Injection

Is how you get those dependencies to those locations

Q&A

egor.neliuba@yopeso.com

github.com/egor-n