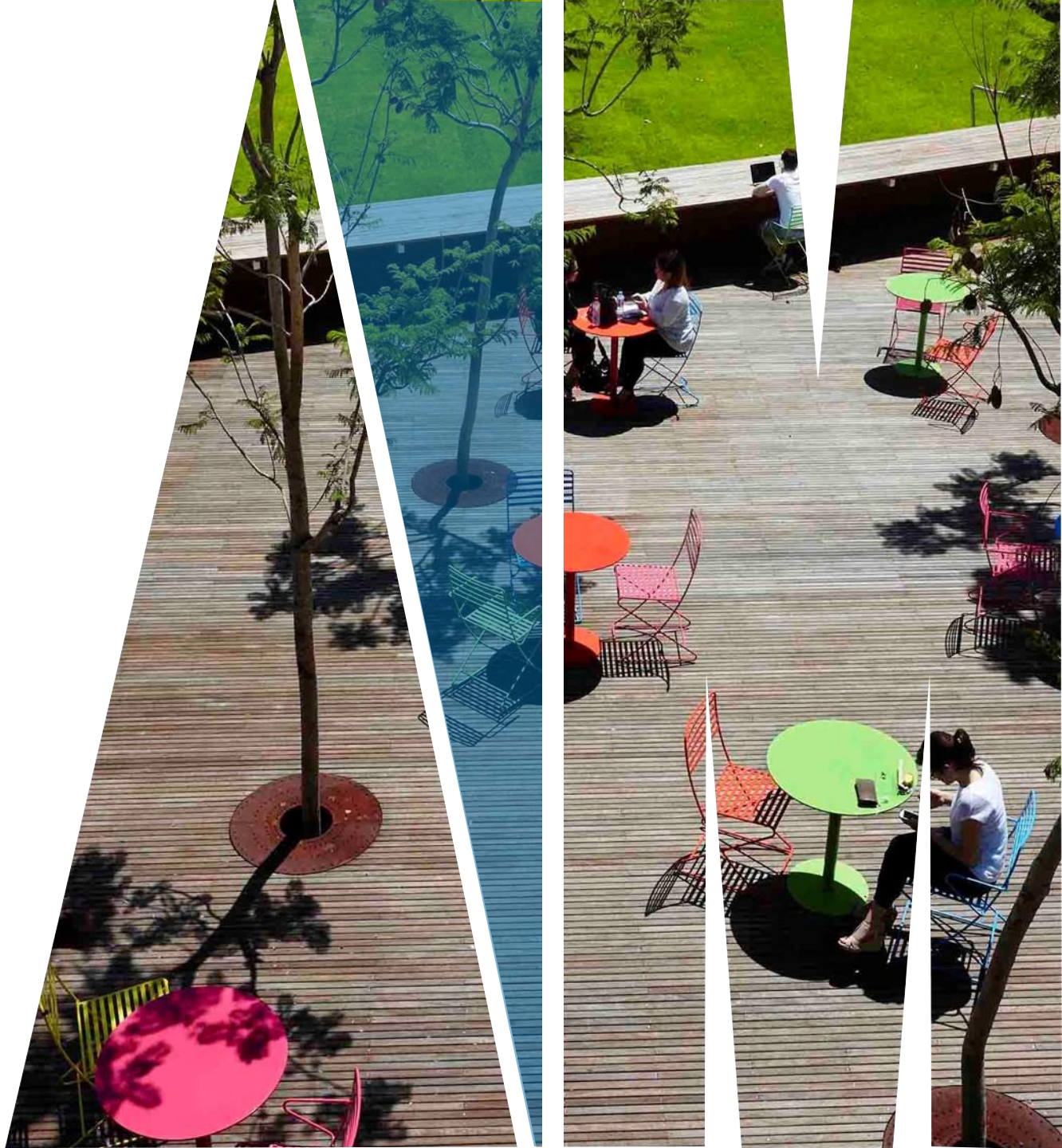
MONASH University

FIT2099 Object-Oriented Design and Implementation

Dependency injection



MONASH University

Outline

What is dependency injection?

Dependency injection and SOLID

Types of dependency injection

- constructor
- field (aka setter)
- interface

WHERE WE ARE?

Over the last few weeks, we have been looking at design principles and techniques that can make software easier to build and maintain

- **Abstraction** and **encapsulation**
  - break your code up into small, mostly independent modules
  - keep them simple
- **SOLID principles**
  - simplify interactions within and between code modules

Today, we look at **dependency injection**, a programming technique that takes these ideas further

DEPENDENCY INJECTION
STEP-BY-STEP

Classes often require references to other classes.

For example, a **Car** class might need a reference to an **Engine** class.

The Car class is dependent on having an instance of the Engine class to run.

EXAMPLE **WITHOUT** DEPENDENCY INJECTION

```
class Car {
    private Engine engine = new Engine();
    public void start() {
        engine.start();
    }
}
```

EXAMPLE **WITHOUT** DEPENDENCY INJECTION

A.The class **constructs** the dependency it needs.

```
class Car {
    private Engine engine = new Engine();
    public void start() {
        engine.start();
    }
}

class MyApp {
    public static void main(String[] args) {
        Car car = new Car();
        car.start();
    }
}
```

EXAMPLE **WITH**
DEPENDENCY INJECTION

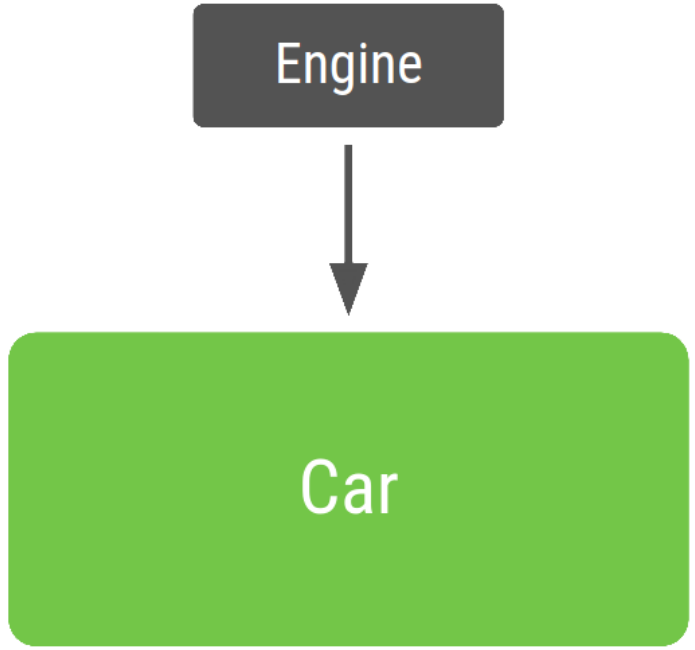
B. The class gets the dependency it needs **supplied to it.**

```
class Car {
    private final Engine engine;
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }
}
```

EXAMPLE **WITH**
DEPENDENCY INJECTION

B. The class gets the dependency it needs **supplied to it.**

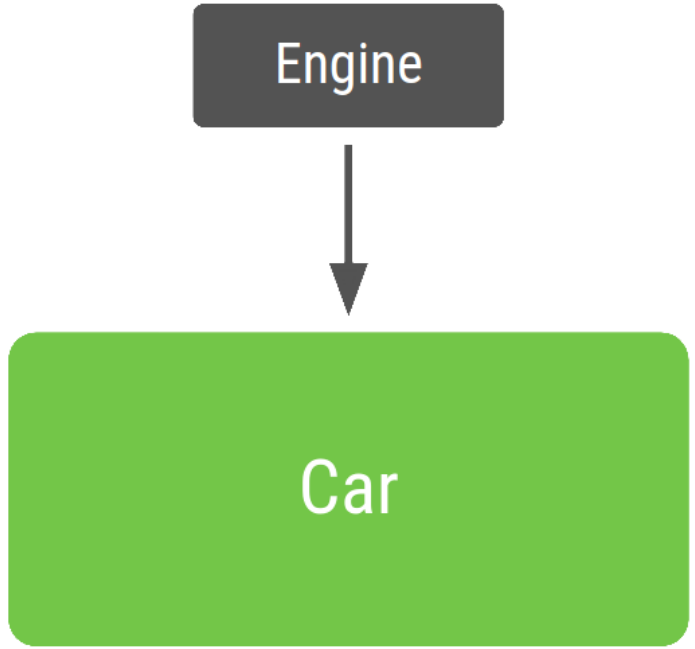
```
class Car {
    private final Engine engine;
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }
}
```



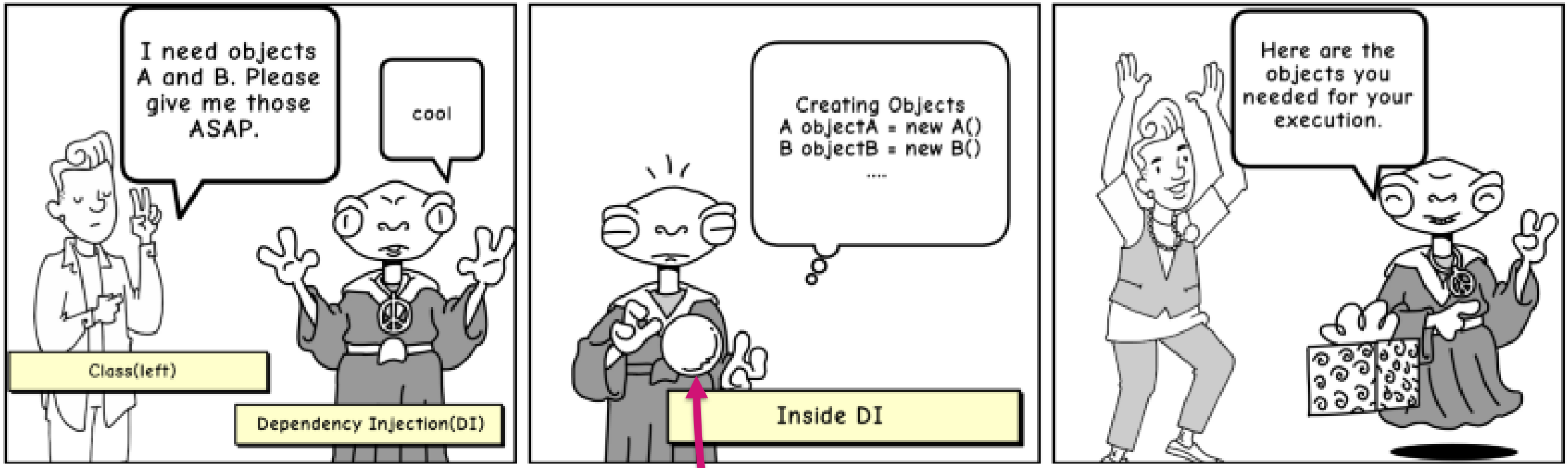
EXAMPLE **WITH**
DEPENDENCY INJECTION

B. The class gets the dependency it needs **supplied to it.**

```
class Car {
    private final Engine engine;
    public Car(Engine engine) {
        this.engine = engine;
    }

    public void start() {
        engine.start();
    }
}

class MyApp {
    public static void main(String[] args) {
        Engine engine = new Engine();
        Car car = new Car(engine);
        car.start();
    }
}
```



WHAT IS DEPENDENCY INJECTION?



Pay attention to this

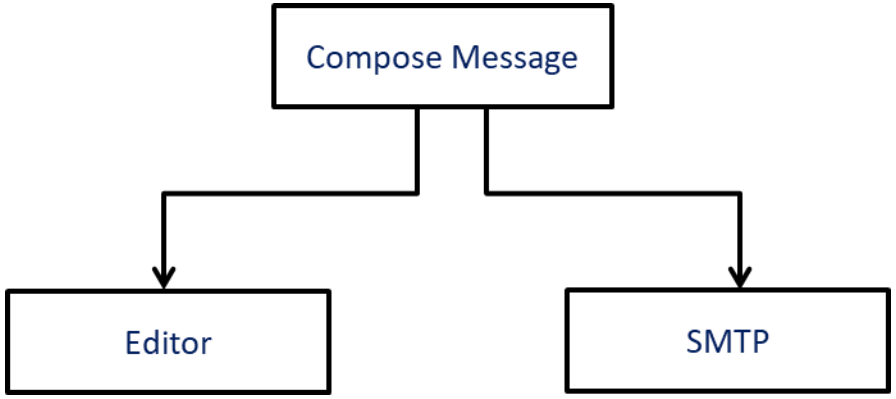
BENEFITS OF
DEPENDENCY INJECTION

Implementing dependency injection provides you with the following advantages:

- **Reusability** of code
- Ease of **refactoring**
- Ease of **testing**

WITHOUT
DEPENDENCY INVERSION

First, let's review dependency ***inversion***…

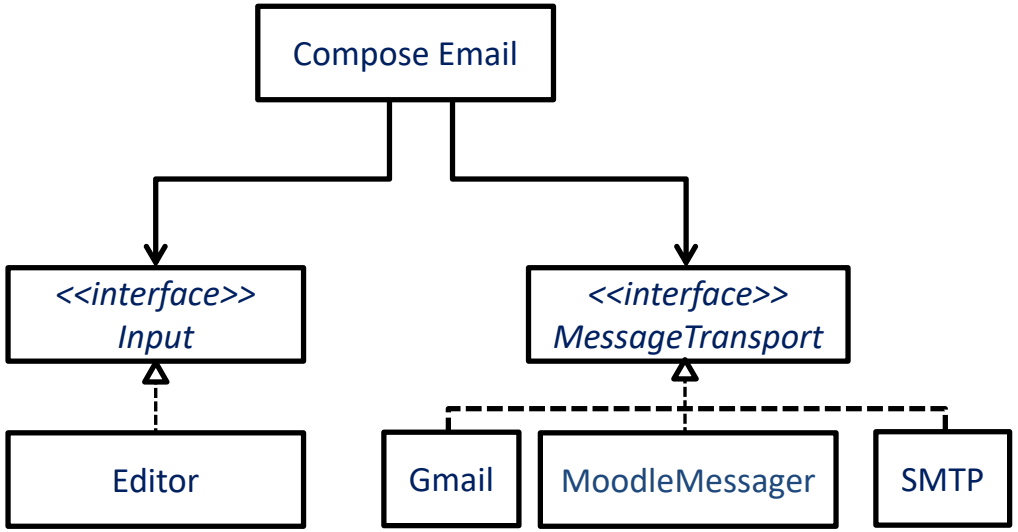


Standard top-down design makes high-level business logic depend directly on low-level components – this makes it **harder to *change* those low-level decisions** when you need to add new capabilities to the system.

WITH

DEPENDENCY INVERSION

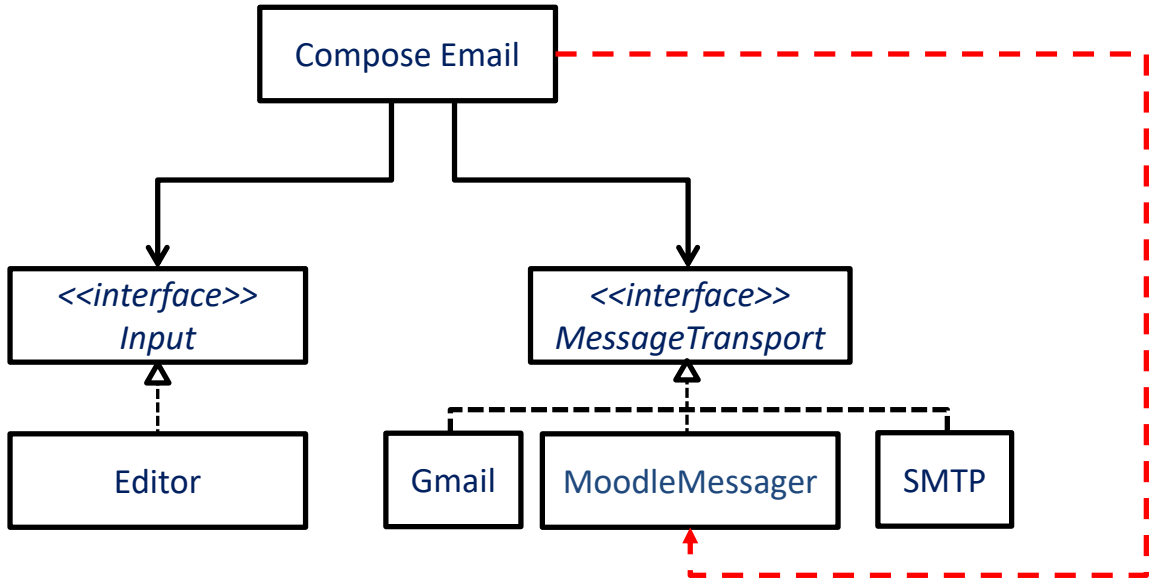
First, let's review dependency ***inversion***…



Dependency inversion introduces an **abstraction layer** – if the client code (e.g. **Compose Email**) uses predefined interfaces then new services can be easily **plugged in**

WITH
DEPENDENCY INVERSION

First, let's review dependency ***inversion***…



But something still needs to ***create*** **those concrete subclasses**… and that's **usually the client, so there's still a dependency (but not an association)**

A

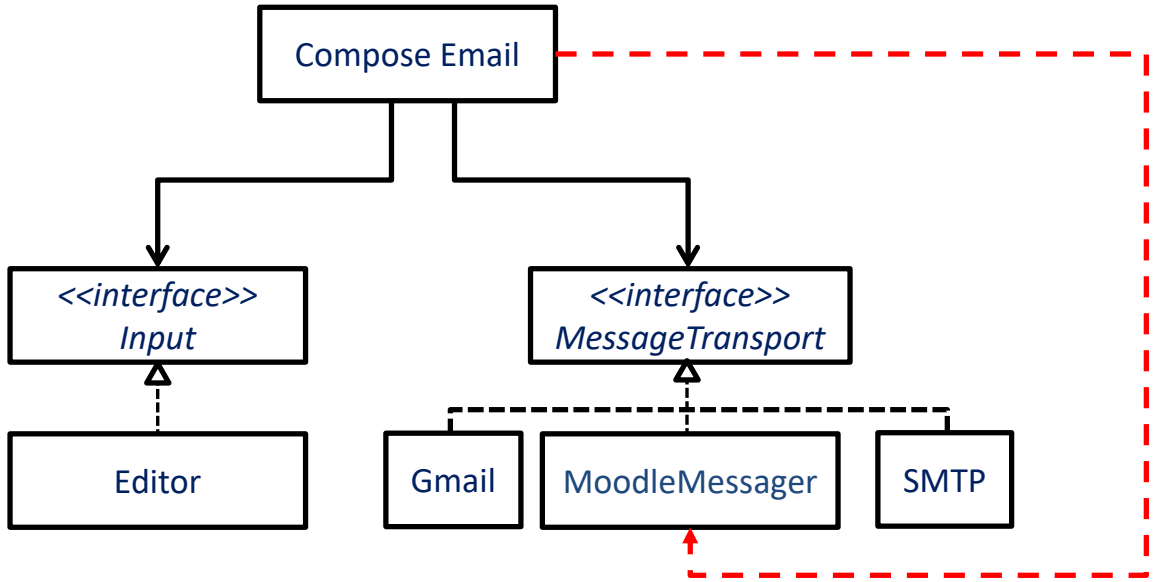
DESIGNER'S WISHLIST

Let's think about how a design would look in a perfect world…

- high level and low level components should depend on **abstractions** ✓ (**Dependency Inversion Principle**)
- should be easy to **replace** components without any impact on other components ✗
- should be easy to **test** components in isolation ✗
- dependency **inversion** doesn't do this on its own!
- *dependency **injection*** helps…

WITH
DEPENDENCY INVERSION

First, let's review dependency ***inversion***…



But something still needs to ***create*** **those concrete subclasses**… and that's **usually the client, so there's still a dependency (but not an association)**

HOW IS DEPENDENCY INJECTION IMPLEMENTED?

The inverted code might look something like this:

```
public class ComposeMessage {
    private MessageTransport mt;
    public ComposeMessage() {
        this.mt = new MoodleMessager();
        //…
    }
}
```

Still depends on the concrete subclass's constructor

The extent of this dependency isn't huge, but its presence means that you can't change the message transport layer without editing ComposeMessage… and potentially introducing a bug. ☹

SOLUTION:
DEPENDENCY INJECTION

Core idea: instead of having the high-level module **create** the low-level module, get it **passed in by an external class**

- e.g. **whatever creates the high-level module**
- or maybe some kind of **configuration module**

Then, the high-level module does not need to know ***anything*** about the low-level module

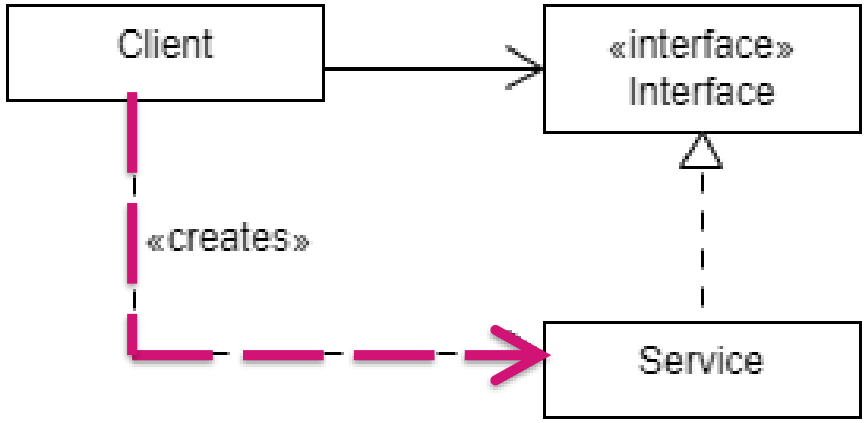
DEPENDENCY INJECTION
TERMINOLOGY

If you like to learn about new programming concepts on the internet, there are several terms you will need to know

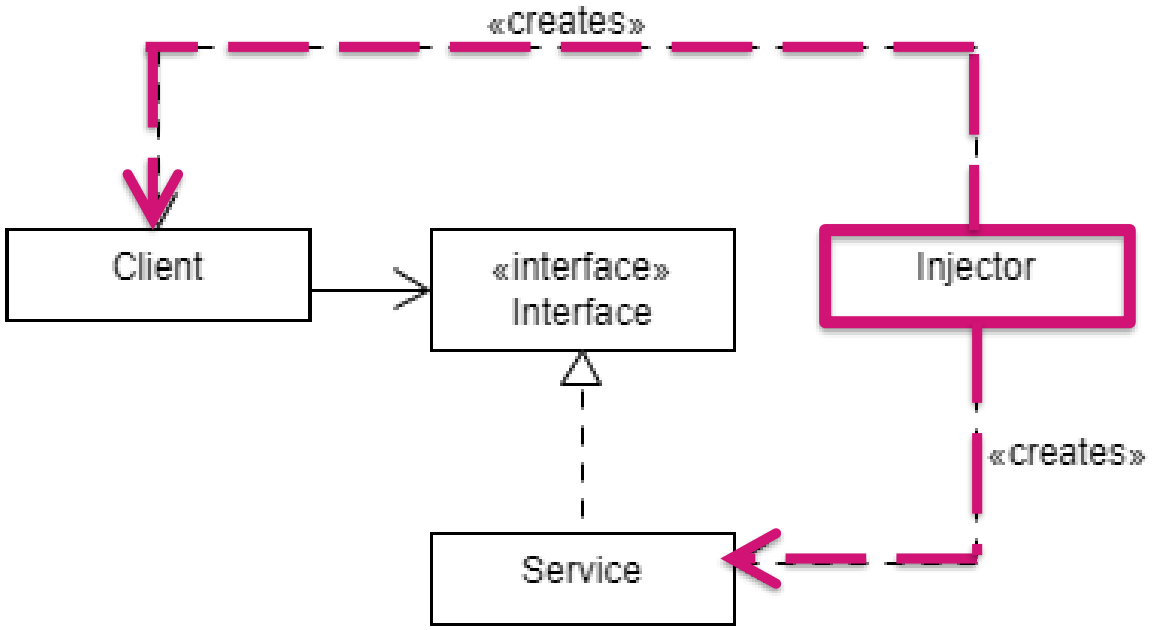
- **client**: the class that is using the interface (e.g. ComposeMessage)
- **service**: the low-level module that is injected into the client (e.g. MoodleMessager)
- **interface**: the abstract interface that defines the methods that can be called in the service (e.g. MessageTransport)
- injector: **the external module that gives the concrete service to the client**

COMPARISON OF
CLASS DIAGRAMS

Dependency ***inversion***



Dependency ***injection***



DEPENDENCY INJECTION VIA THE CONSTRUCTOR

The ComposeMessage class under dependency injection is still simple

```
public class ComposeMessage {
    private MessageTransport mt;
    public ComposeMessage(MessageTransport theMt) {
        this.mt = theMt;
        //…
    }
}
```

This example uses ***constructor injection*** but there are other kinds

TYPES OF DEPENDENCY INJECTION

Constructor injection: an instance of the service is passed into the client's constructor

- the injector must be the class that instantiates the client

Setter injection: the client has a concrete setter that the injector can use to pass in the service instance

- can be used at any time; allows you to change the service of a running client
- but requires a public setter, might not be good for information hiding

Interface injection: the client implements an interface that allows the injector to pass in the service instance

- ends up being like setter injection but you can choose what your setter is called/are not restricted to a single setter

THE SETTER INJECTION

Injector uses a setter to pass in the service

Note connascence of execution: object isn't configured until setMt() has been called

```
public class ComposeMessage{
    private MessageTransport mt;
    public void setMt(MessageTransport theMt){
        this.mt = theMt;
    }
    public ComposeMessage() {
        // the constructor
    }
```

TYPES OF DEPENDENCY INJECTION

Constructor injection: an instance of the service is passed into the client's constructor

- the injector must be the class that instantiates the client

Setter injection: the client has a concrete setter that the injector can use to pass in the service instance

- can be used at any time; allows you to change the service of a running client
- but requires a public setter, might not be good for information hiding

Interface injection: the client implements an interface that allows the injector to pass in the service instance

- ends up being like setter injection but you can choose what your setter is called/are not restricted to a single setter

THE INTERFACE INJECTION

```
public class ComposeMessage implements InjectMt{
    private MessageTransport mt;

    public void injectMt(MessageTransport theMt){
        this.mt = theMt;
    }
    public ComposeMessage() {
        // the constructor
    }
}
```

Still got some **connascence of execution** here, but more control over interface

TYPES OF DEPENDENCY INJECTION

Constructor injection: an instance of the service is passed into the client's constructor

- the injector must be the class that instantiates the client

Setter injection: the client has a concrete setter that the injector can use to pass in the service instance

- can be used at any time; allows you to change the service of a running client
- but requires a public setter, might not be good for information hiding

Interface injection: the client implements an interface that allows the injector to pass in the service instance

- ends up being like setter injection but you can choose what your setter is called/are not restricted to a single setter

WHICH TYPE OF INJECTION TO USE?

All types have good and bad points

Which is best depends on your circumstances

Often, your choice will be constrained by your **dependency injection framework**

- we'll talk about those later…

DEPENDENCY INJECTION AND TESTING

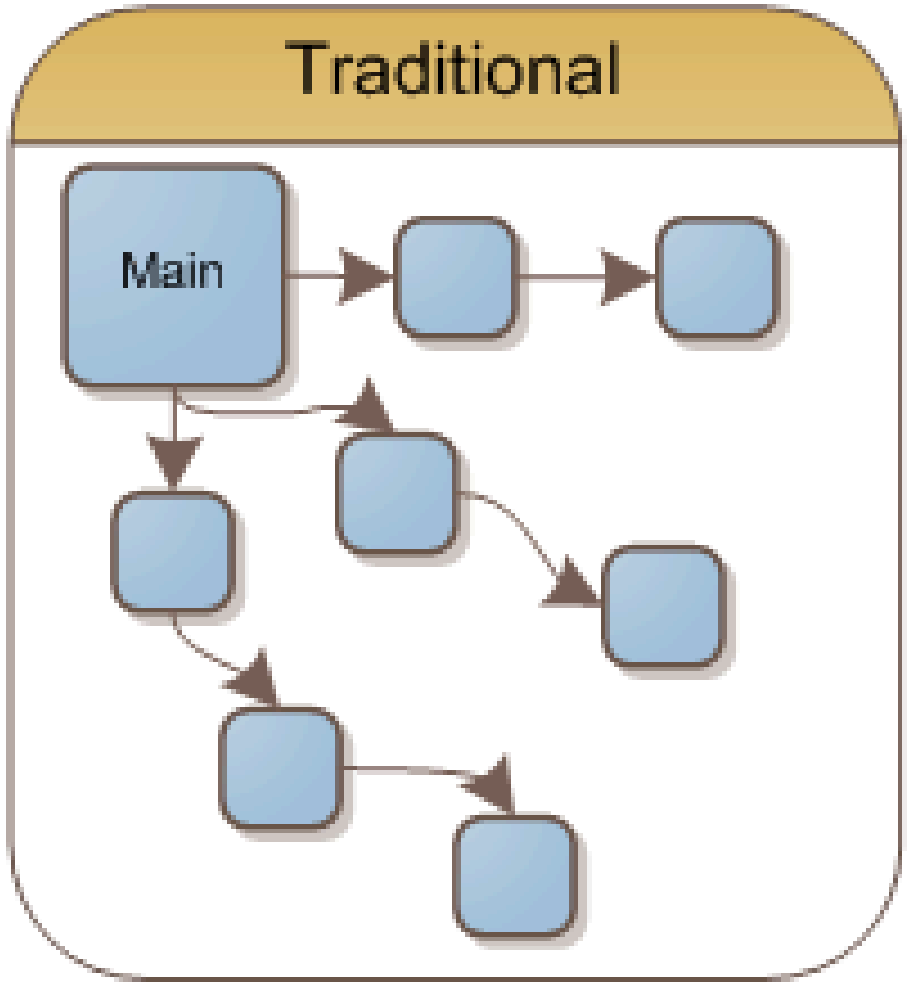
Suppose you have an ordinary program that does something **complicated**

You want to ensure that each of your classes works properly

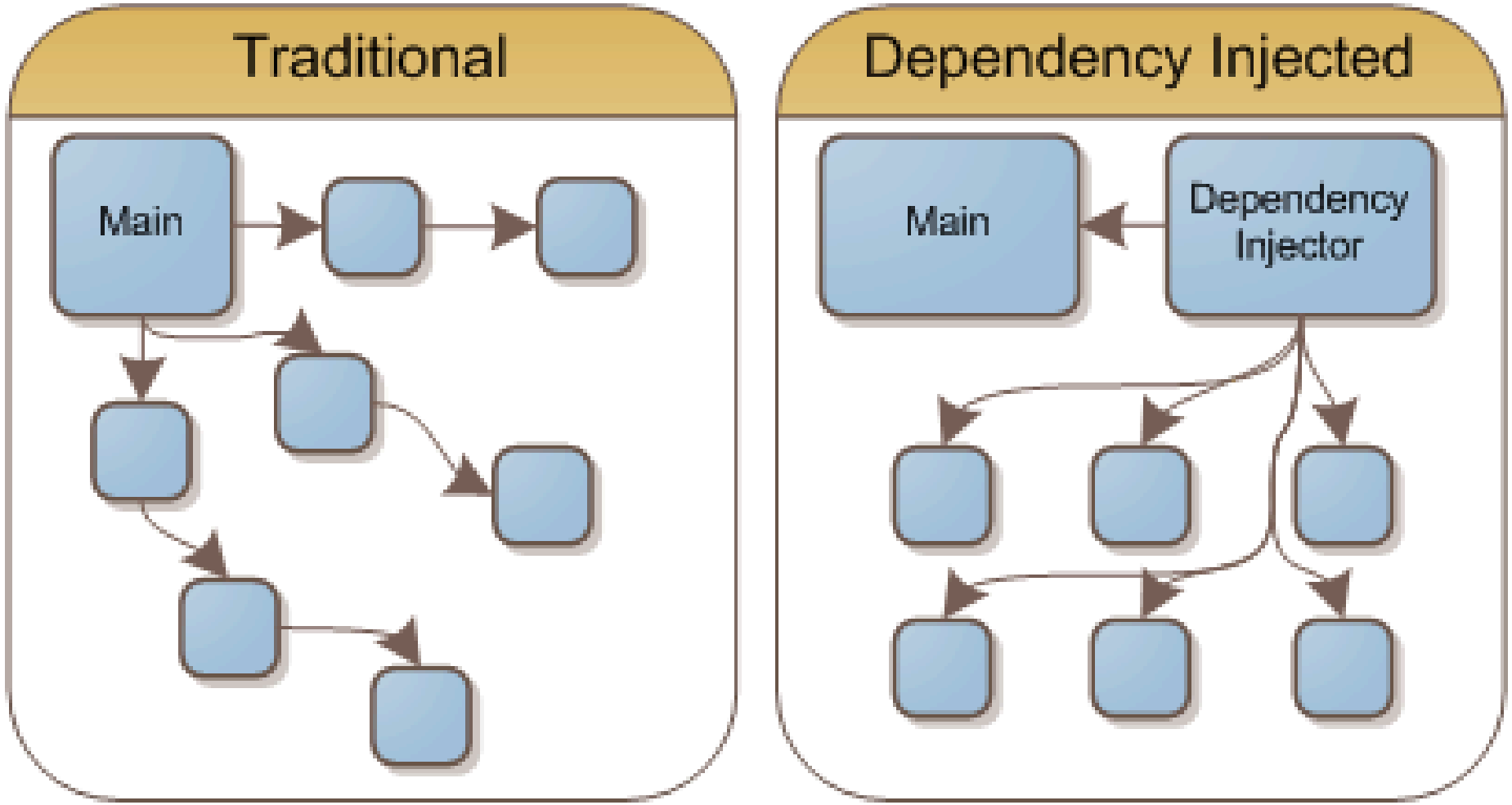
Can use unit testing for this, e.g. with **Junit**

- but what if some classes use a service that is slow?
- what if some classes use a resource that's not always available, such as the internet?

DEPENDENCY INJECTION AND TESTING



DEPENDENCY INJECTION AND TESTING



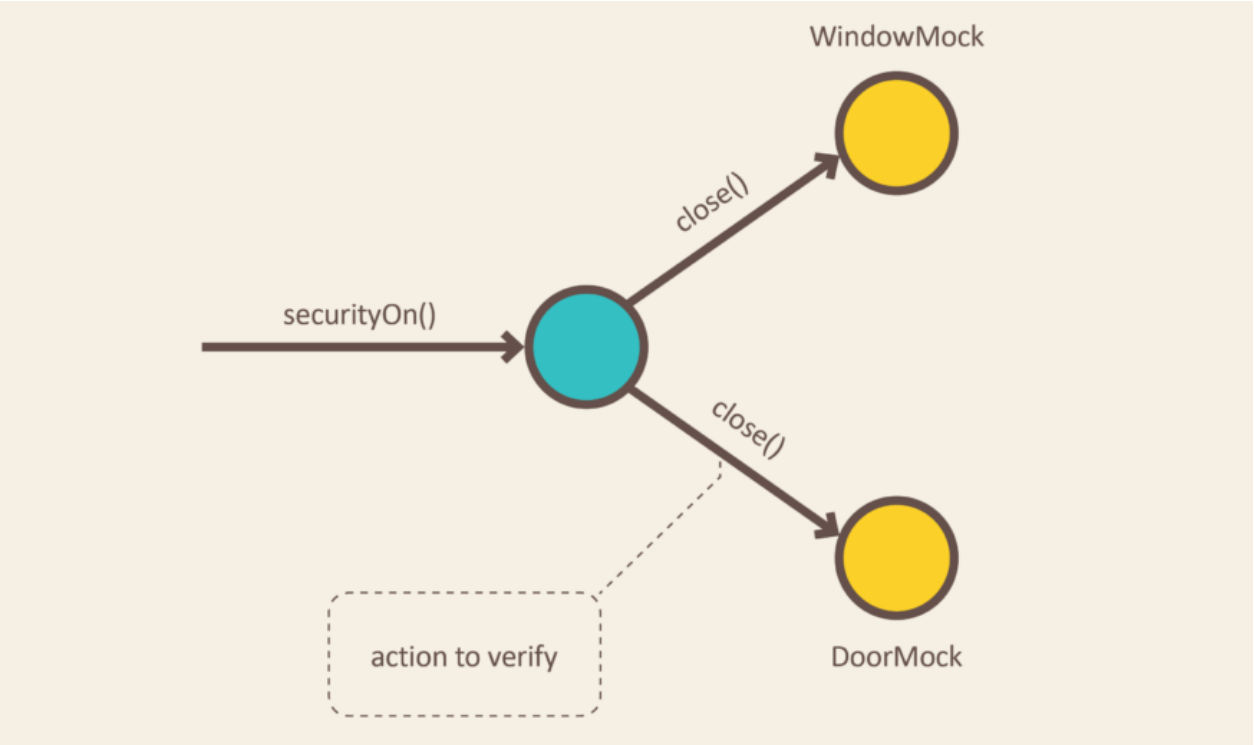
DEPENDENCY INJECTION AND TESTING

If you're using dependency injection, you can replace those services with **mocks**

- these are stub classes that **pretend to provide the service**
- actually, they provide **preprogrammed** responses
- may also store or log the methods called in them, and their parameters
- can use this **to verify** that the class under test is using services appropriately

DEPENDENCY INJECTION AND TESTING

Dependency injection is the usual way in which objects should be created. Dependencies become **visible** in the constructors and other methods. These dependencies can therefore be **easily replaced during testing with mock objects**. This can be configured either in code or via a configuration file.



DEPENDENCY INJECTION AND TESTING

Big advantage of dependency injection is that it lets you **remove *all* dependencies** that the client has on the concrete service

This means that you can substitute the service for a mock one when you're testing

- so if your service does something slow, or relies on a resource that might not be there (e.g. internet), you can replace it with a test version that doesn't

DEPENDENCY INJECTION
FRAMEWORKS

Many software packages do dependency injection for you

- typically, **they supply the injector**
- Java examples include **Spring** and **Guice**

These are ***everywhere*** in industry

Often manage other aspects of your system as well

- e.g. **persistence** (talking to back end databases); binding to REST APIs; connection to web frameworks etc.

A detailed description is beyond the scope of this unit

- but now you know enough about the background to understand the documentation if you want to research dependency injection frameworks

Summary

What is dependency injection?

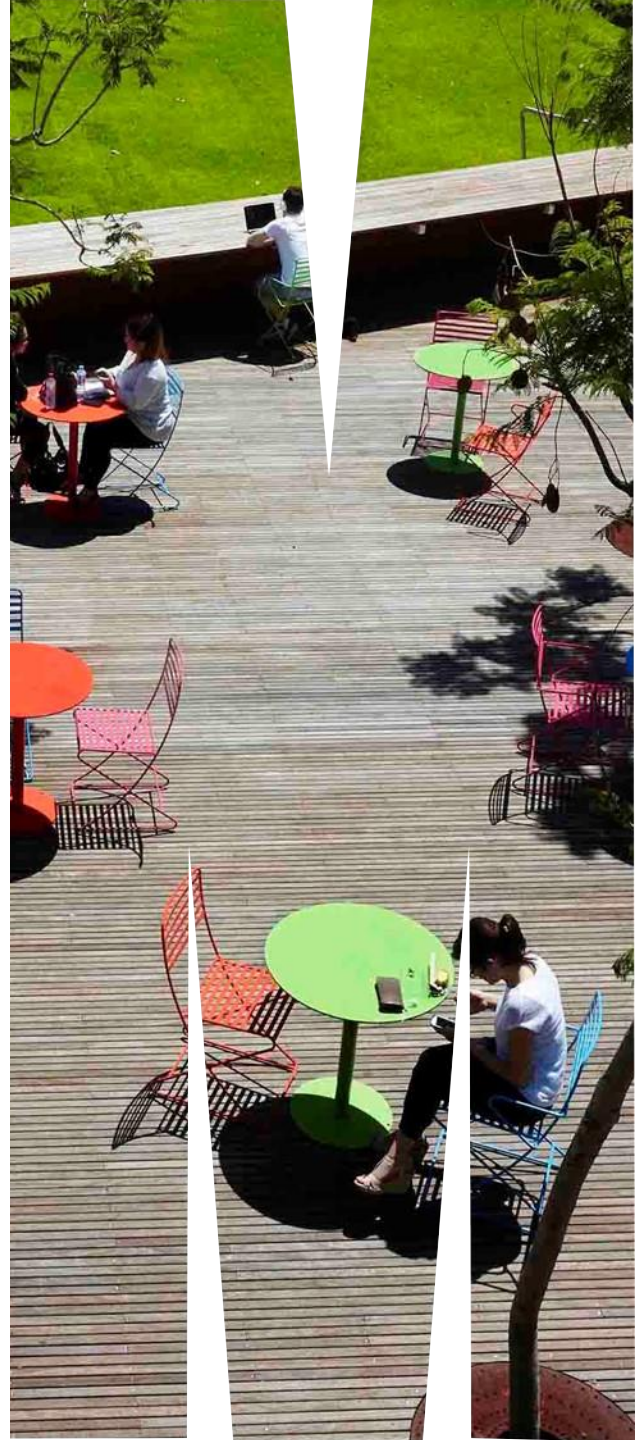
Dependency injection and SOLID

Types of dependency injection

- constructor
- field (aka setter)
- interface

MONASH University

Thanks



MONASH University