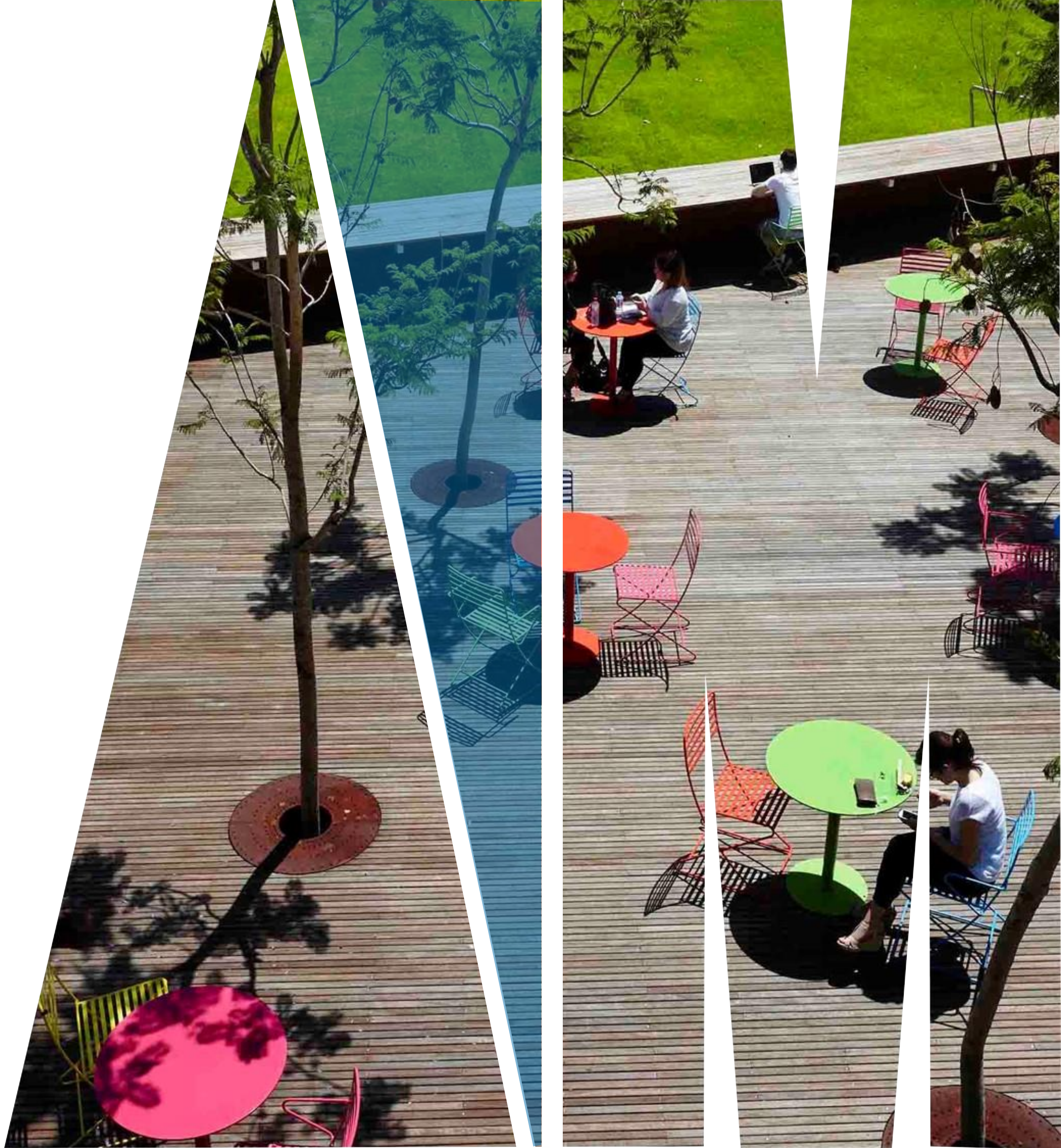
MONASH University

FIT2099 Object-Oriented Design and Implementation

SOLID principles
(Part 1)



Outline

S	Single Responsibility Principle
O	Open Closed Principle
L	Liskov Substitution Principle
I	Interface Segregation
D	Dependency Inversion Principle

QUICK
RECAP

We have been looking at ways to think about the **quality of a design**

does it support **all the functionality** that users need?

are its components **too tightly coupled**, so that it becomes hard to modify?

Today we look at an influential set of design principles: **SOLID**

Which are ultimately aimed at Reducing Dependencies!

ReD

THE
SOLID PRINCIPLES

First articulated by Robert C. Martin in 2000

- paper called **Design Principles and Design Patterns**
- available to you on Moodle

Guidelines rather than laws

Five principles:

- the **S**ingle Responsibility Principle
- the **O**pen-Closed Principle
- the **L**iskov Substitution Principle (we mentioned this one already)
- the **I**nterface Segregation Principle
- the **D**ependency Inversion Principle (covered more in depth in future lessons)

We will look at each of these

- we will return to some in future lectures too

THE
SINGLE RESPONSIBILITY PRINCIPLE (SRP)

A class should have one, and only one, reason to change.
-- Robert C. Martin

Each class should have **one responsibility**

- it **shouldn't** take on extra responsibilities
- ideally it should contain **all functionality needed** to support that responsibility

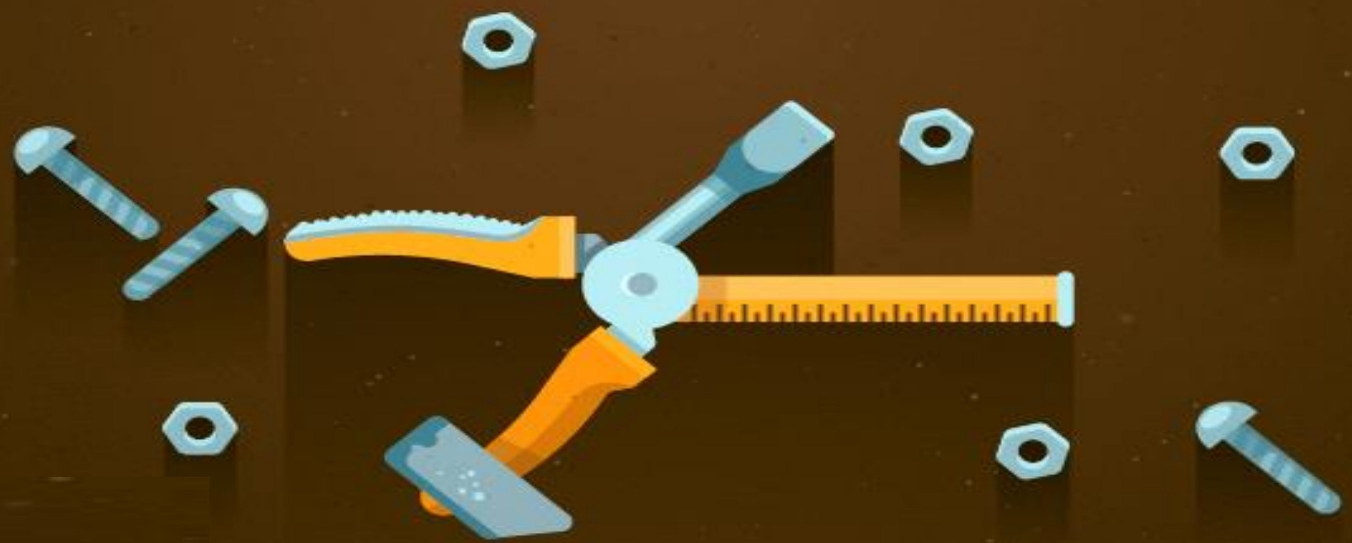
Single Responsibility Principle (SRP)

A single responsibility per tool

Single Responsibility Principle (SRP)

A single responsibility per tool

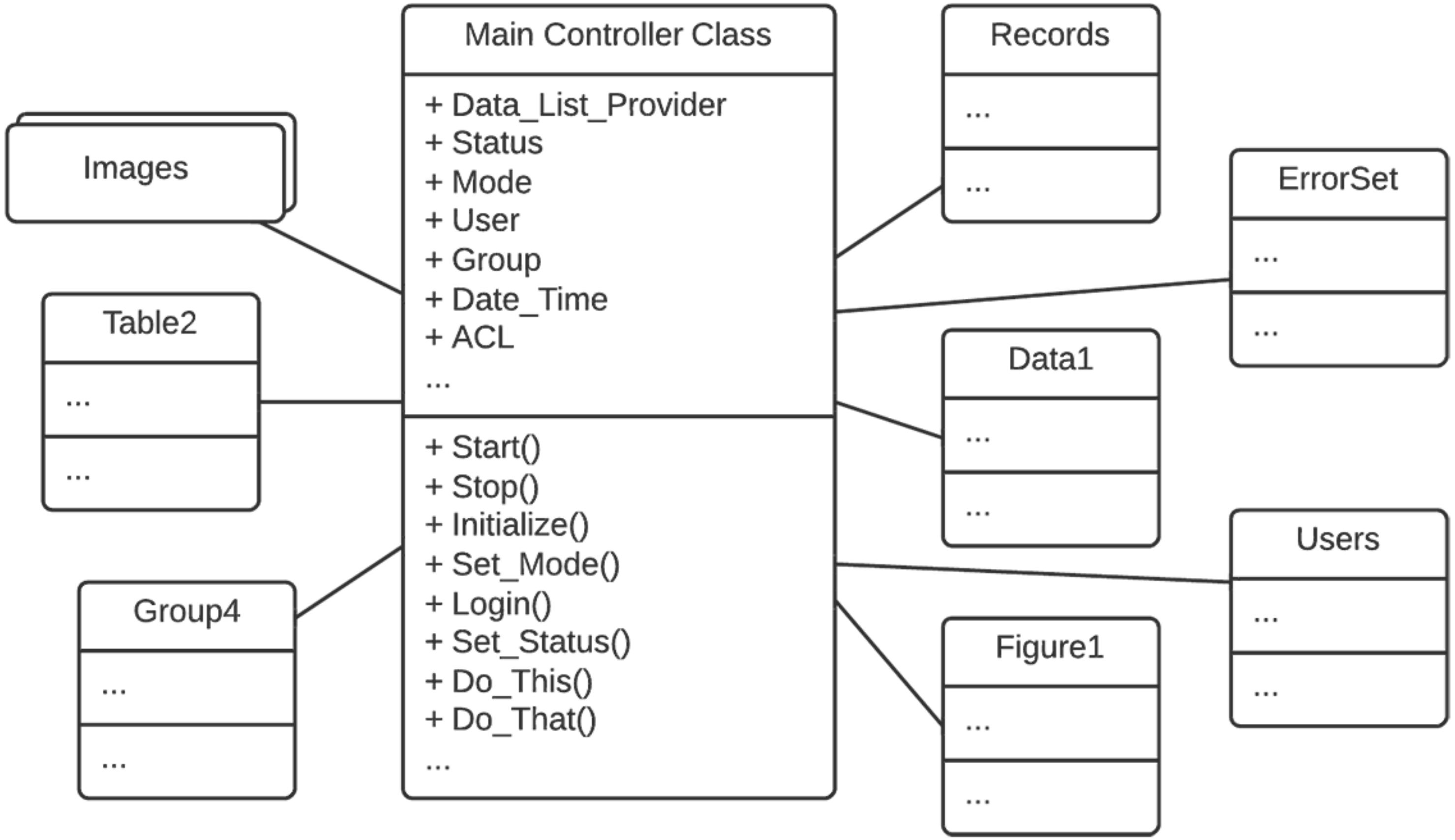
A tool with multiple responsibilities



BREACHING SRP:
GOD CLASSES

Common problem: classes that try to do too much

- e.g. **one big class** has all the methods
- **other classes just contain data**



THE
GOD CLASSES

Often seen in designs by **people uncomfortable with OOD**

- separating data from algorithms that work on it reflects **procedural rather than OO design** practice
- or **trying to simplify the class diagram** at the expense of the classes

Beginners may feel it's easier if only one file has to change

- but it's *not* easy to make changes in a big file!

Can also occur when you're **bridging** between OO and non-OO components of a system

- e.g. HTML or SQL

God classes have *lots* of reasons to change, so are **hard to maintain**

WHAT ABOUT THE OPPOSITE?:
OVERAPPLYING SRP

It is possible to take the idea of "single responsibility" too far

SRP encourages you to **split classes up**

– no other principle tells you to lump them together

Classes can become too small

– this will overcomplicate your design

Martin did not mean for you to do this!

– he was trying to address the common tendency to make classes too large

Another way Martin explained the SRP:

> *Gather together the things that <u>change</u> for **the same reasons**.*
> *Separate those things that <u>change</u> for **different reasons.***

SRP AND

REASON OF CHANGE

*Gather together the things that <u>change</u> for **the same reasons**.*
*Separate those things that <u>change</u> for **different reasons.***

```
public class Employee {
  public string Name { get; set; }
  public string Address { get; set; }
  ...
  public void ComputePay() { ... }
  public void ReportHours() { ...}
}
```

The **ComputePay()** behavior is under the responsibility of the **finance people** and the **ReportHours()** behavior is under the responsibility of the **operational people**.

It is wise to declare these methods in different dedicated modules/classes

WHAT DOES THE SRP
OFFERS TO YOU?

Good application of the SRP means that your classes are **cohesive**

- if responsibility needs to change, all of the pieces you need are right there
- no excess features getting in the way; they will have been separated into other classes

This means your system is easier to **maintain and extend**

LACK OF COHESION OF METHODS (LCOM) METRIC

The single responsibility principle states that a class should not have more than one reason to change. Such a class is cohesive.

$$LOCM = 1 - \frac{\sum_{f \in F} |M_f|}{|M| \times |F|}$$

M = static and instance methods in the class,

F = instance fields in the class,

M_f = methods accessing field f, and

$|S|$ = cardinality of set S.

In a class that is utterly cohesive, every method accesses every instance field

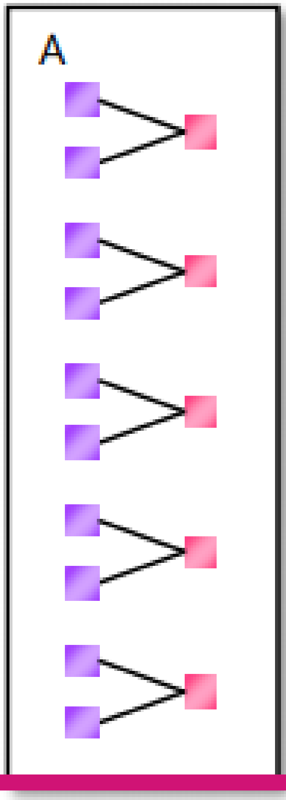
$$\sum |M_f| = |M| \times |F|$$

so LOCM = 0.

A high LCOM value generally pinpoints a poorly cohesive class.

Types where LCOM > 0.8 and $|F|$ > 10 and $|M|$ >10 might be problematic. However, it is very hard to avoid such non-cohesive types.

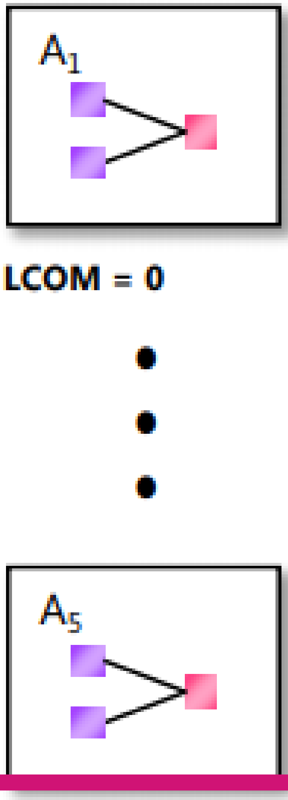
NON-COHESIVE CLASS



LCOM = 0.8

One class with five fields, each with a getter and setter.

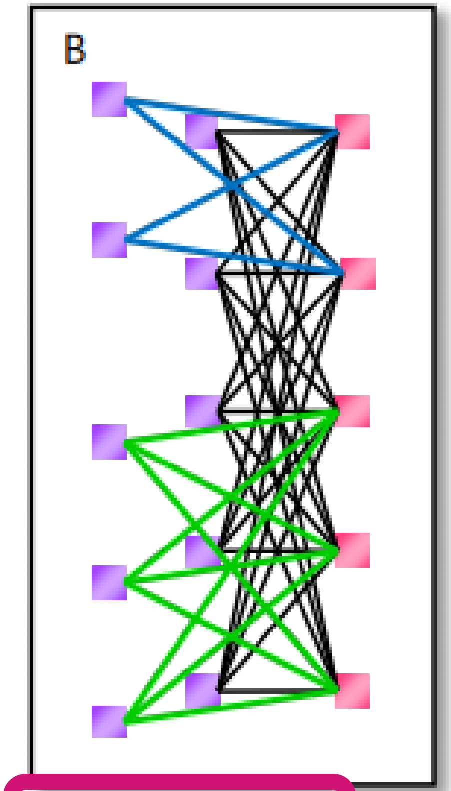
FIVE VERY COHESIVE CLASSES



LCOM = 0

Five classes, each with one field and a getter and setter.

ONE RELATIVELY COHESIVE CLASS

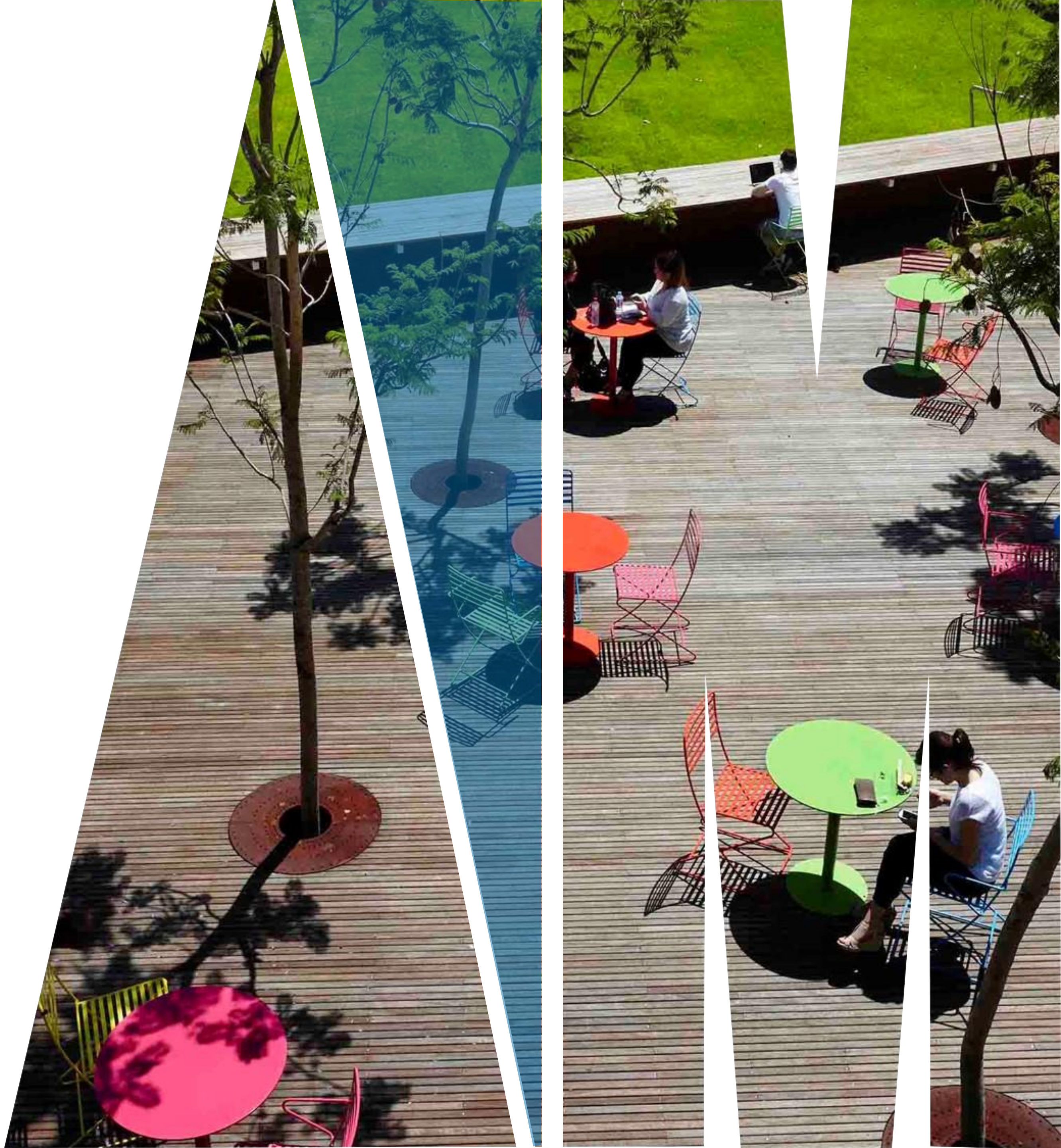


LCOM = 0.24

Five constructors each set five fields (black); two getters that access two fields (blue); and three getters that access three fields (green).

MONASH University

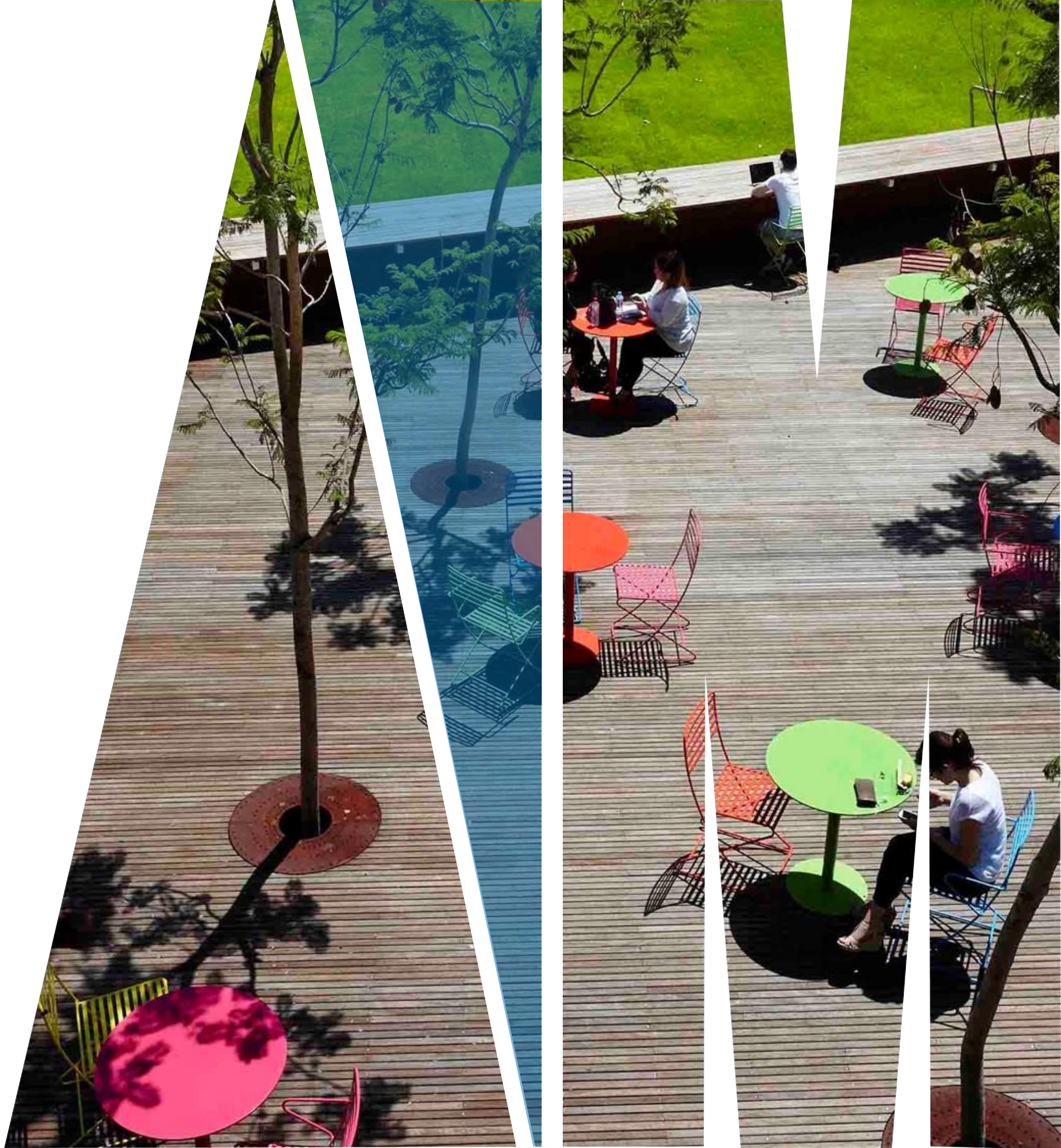
Thanks



MONASH University

FIT2099 Object-Oriented Design and Implementation

SOLID principles (Part 2: OCP and LSP)



MONASH University

THE
OPEN/CLOSED PRINCIPLE (OCP)

Software entities (classes, modules, functions, etc.) should be
open for extension, but closed for modification.
-- Robert C. Martin

Initially, sounds contradictory

- don’t you have to modify something to extend it?

Martin is talking about what should be easy and what should be hard when you’re **adding functionality** to your software

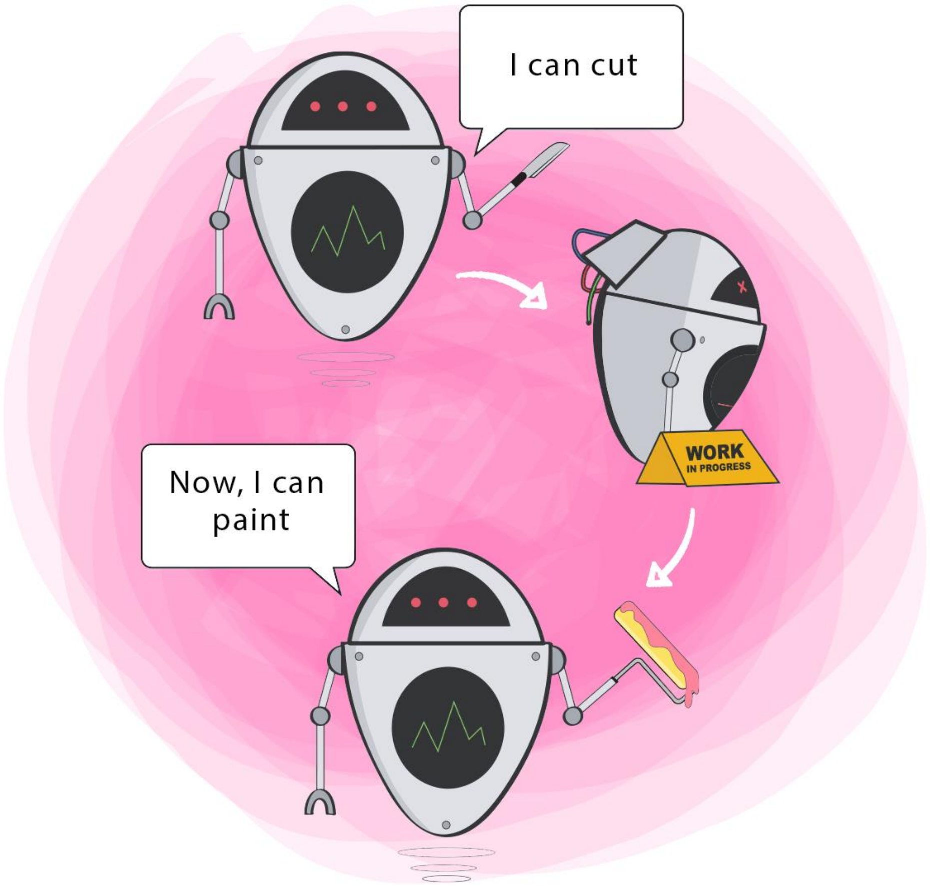
OPEN
FOR EXTENSION

We want to be able to **add features** to our program

- so we want to be able to make our code modules support new functionality
- we want to be able to **add new methods easily**

OPEN
FOR EXTENSION

We want to be able to **add features** to our program



MONASH University

CLOSED
FOR MODIFICATION

We ***don't*** want the way we currently use the code module to change

- e.g. methods change name or signature
- this could easily break existing client code

Changes to a module's interface can be **expensive**

- need to go back and change **client code** to match it

This makes it **harder to add features**

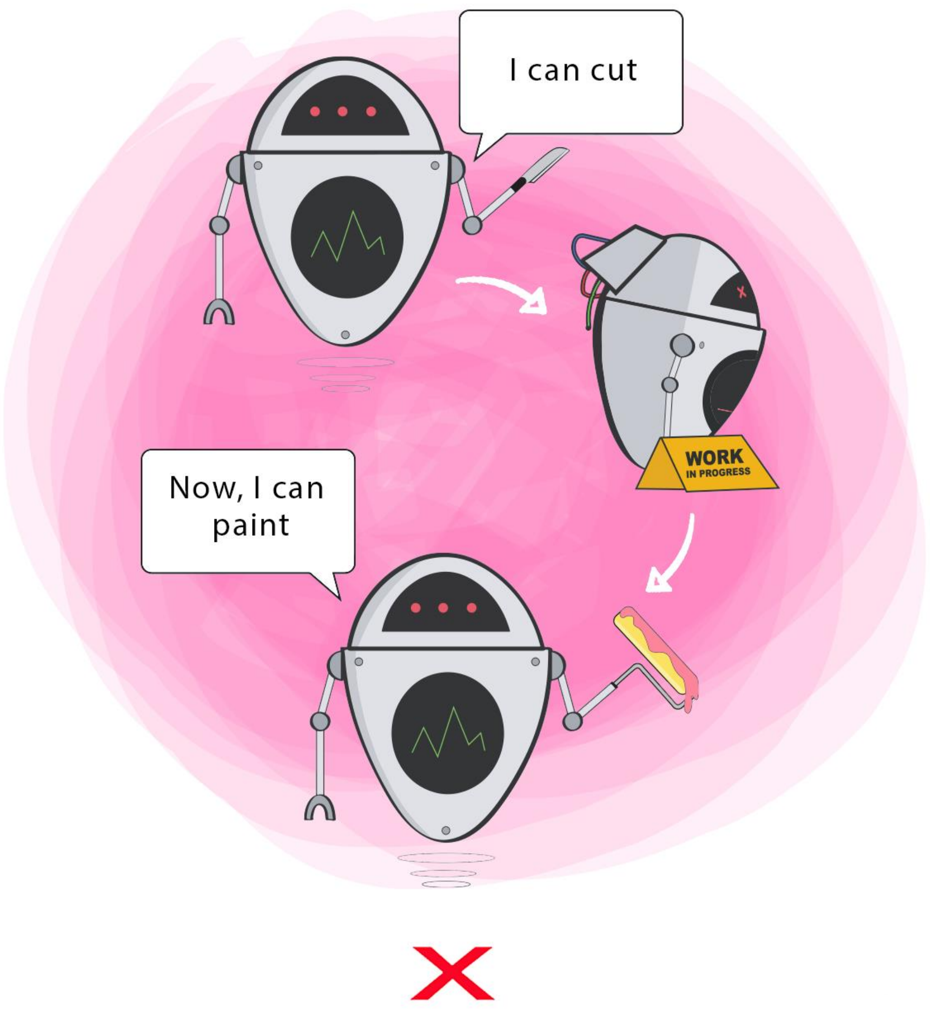
So we want to be able to add features, but in a way that **doesn't change** the way we use existing code.

You do not need to change the mixer's engine!

When changing attachments

OPEN
FOR EXTENSION

We want to be able to **add features** to our program



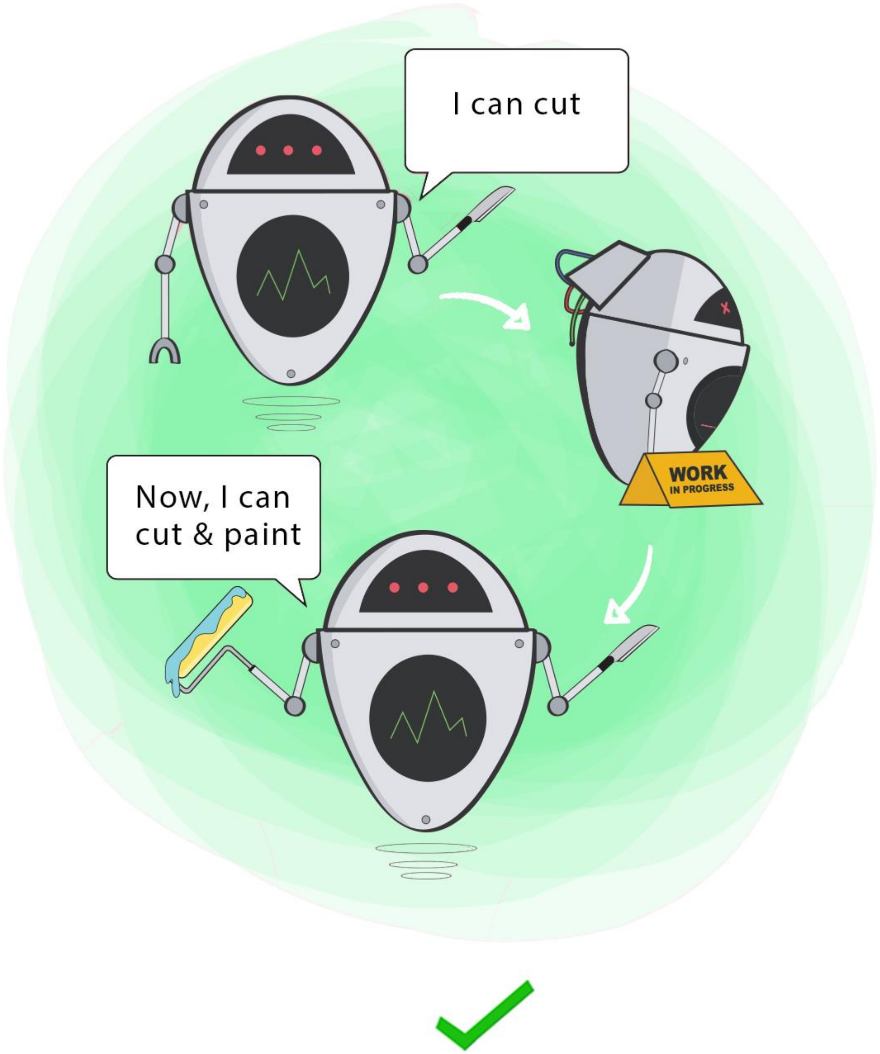
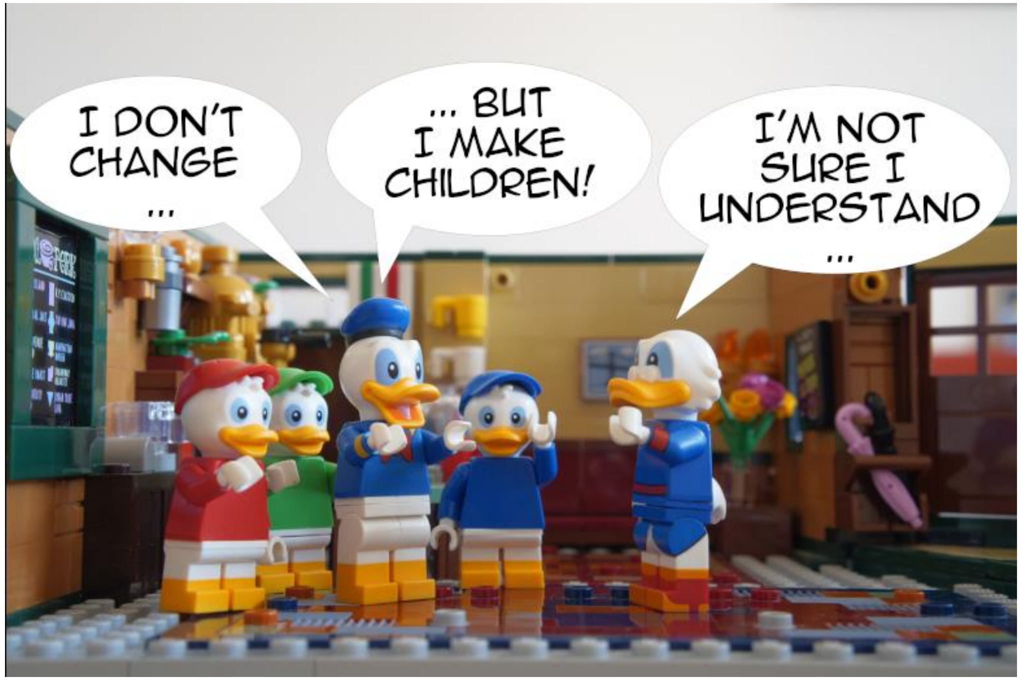


Illustration by [Ugonna Thelma]

BENEFITS OF
OCP

If your program is architected in a way that breaks the OCP, then **adding new features will often break old ones.**

Following OCP can make your code **easier to extend** and **maintain.** This is often achieved using **abstraction** (abstract classes or interfaces)



BUT

This idea can be argued as creating too many abstract classes can also make the system more complex.

OCP IN JAVA

In Java, you can use **interfaces** to support the OCP

- **define them well** and you won't have to modify them
- a class can **implement an interface**
- can also **add any extra methods** it needs
- can even implement **other interfaces**
- we will come back to this idea in future lectures

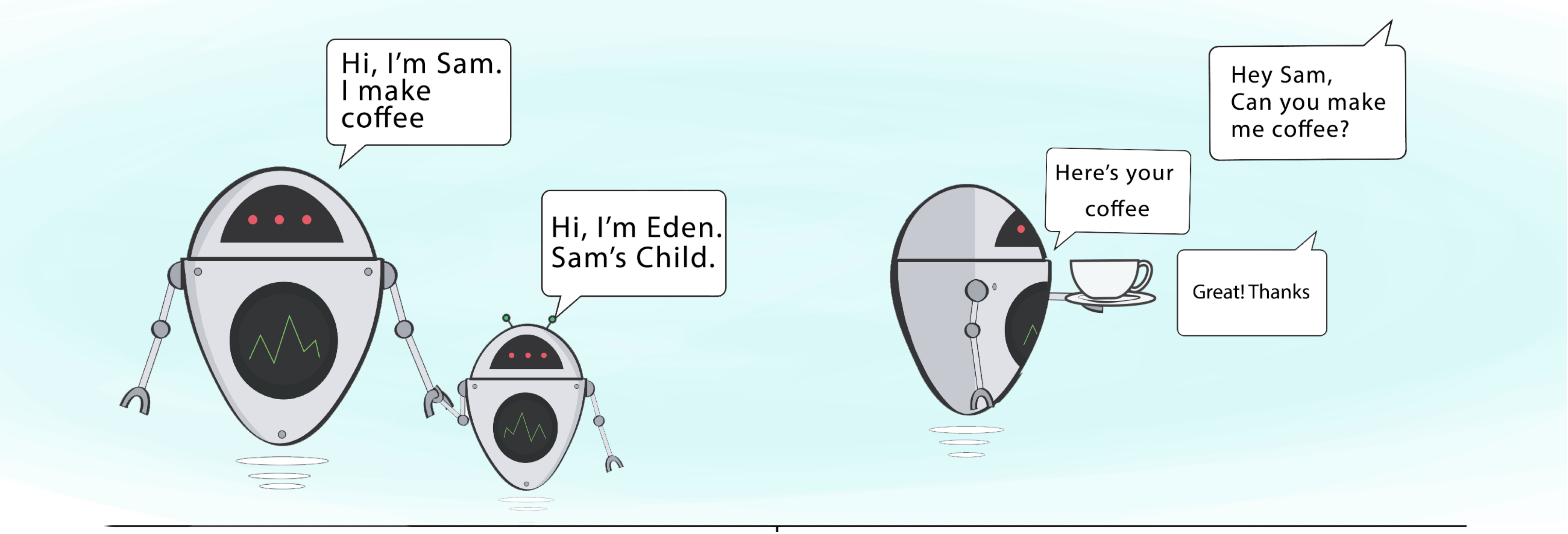
THE
LISKOV SUBSTITUTION PRINCIPLE (LSP)

Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.
-- Barbara Liskov

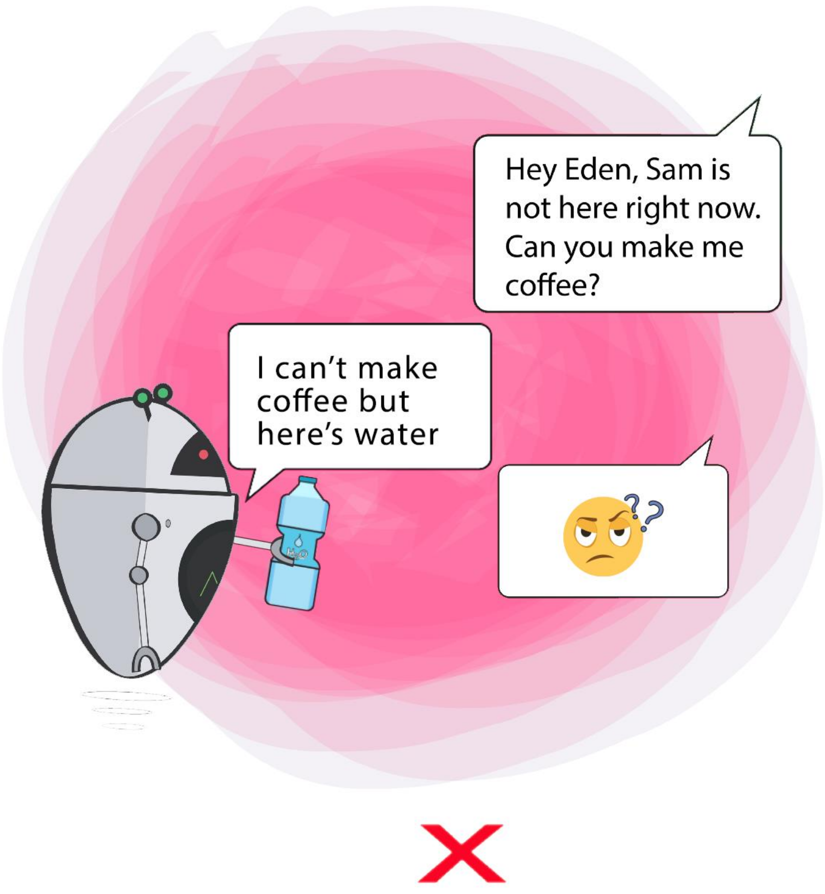
We have seen this before, when we looked at design by contract (DbC)

- essentially means that **you should always be able to use an instance of a subclass when the code is expecting an instance of the base class**
- and the software shouldn't break if you do so

THE
LISKOV SUBSTITUTION PRINCIPLE (LSP)

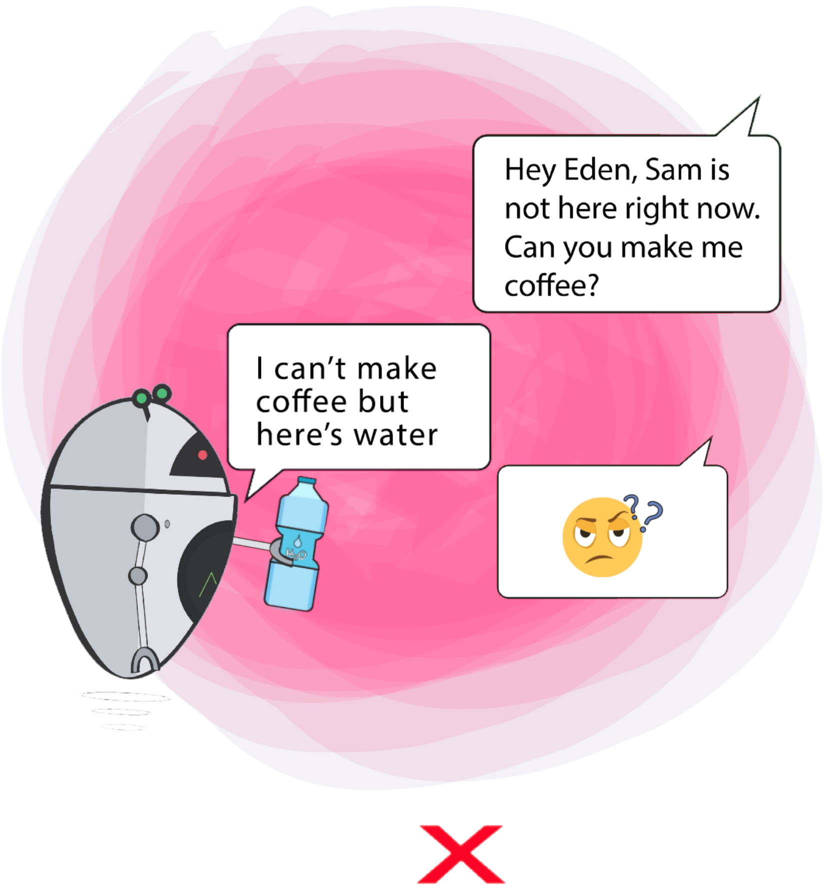


THE
LISKOV SUBSTITUTION PRINCIPLE (LSP)



THE

LISKOV SUBSTITUTION PRINCIPLE (LSP)



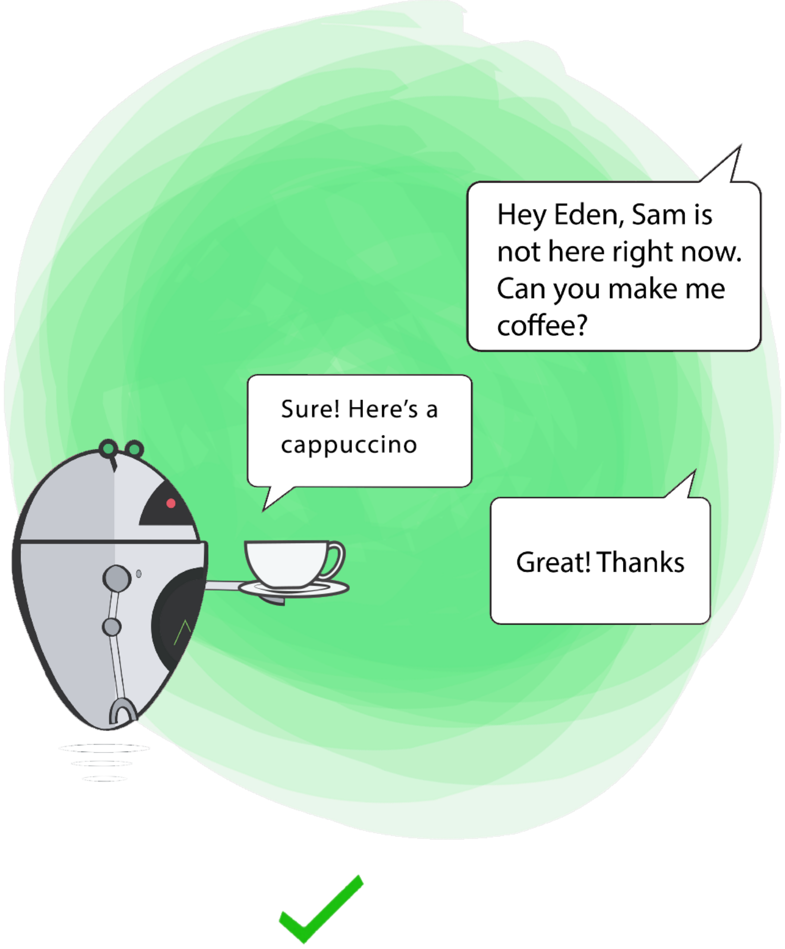


Illustration by [Ugonna Thelma]

FORMAL DEFINITION OF THE LISKOV SUBSTITUTION PRINCIPLE (LSP)

If **B** is a subclass of **A**, you should be able to **put a B** in anywhere the program expects an **A**

- so, for example: **A myA = new B();**

This is true even if A is an **abstract class or interface**

The Java compiler knows that **all methods in A exist in B** too

- so there's nothing you can do with an A that the B won't support
- no reason not to allow **B to act in place of A**

BREAKING THE LSP

LSP is broken if a subclass *can't* do everything its base class can do

This is hard to do in Java, but not impossible:

```java
public void someMethod() {
    throw new MethodNotFoundError();
}
```

In other languages such as C++, it's easy to break LSP

- C++ allows **private inheritance**
- public methods in base class are *private* in the subclass

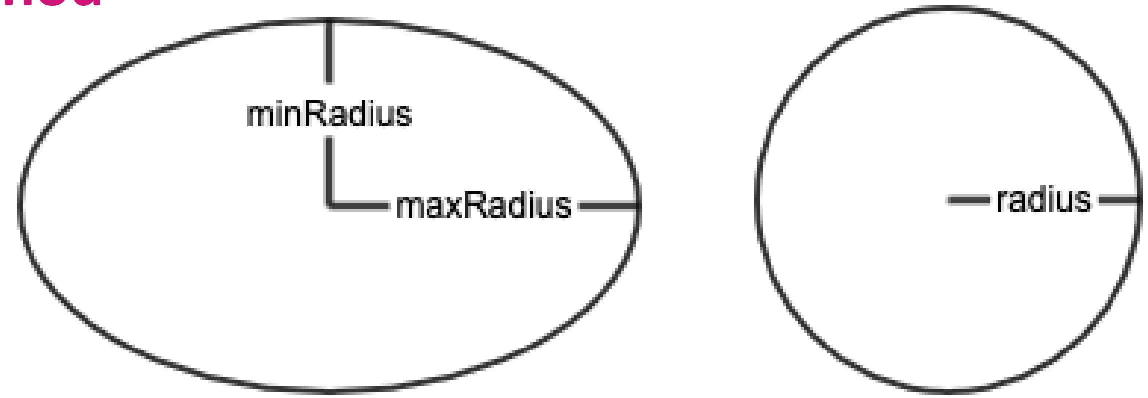
THE
CIRCLE-ELIPSE PROBLEM

Classic example: imagine you have an Ellipse class and you want to make a Circle class

1. Obviously, a circle "is-a" ellipse (this is a mathematical fact!)
2. However, if you stretch an ellipse, it's still an ellipse

   so Ellipse.stretch() is a reasonable method
3. But if you stretch a circle, it's not a circle any more

   so **Circle.stretch() is *not* a reasonable method**
4. So Circle can't inherit Ellipse



RESOLUTION TO THE CIRCLE-ELIPSE PROBLEM

- Could have Circle inherit Ellipse and not implement stretch()?
  - but this **breaks LSP** and therefore polymorphism
- Could have Ellipse inherit Circle?
  - but **an ellipse *isn't* a circle**
- Could have both extend a shared superclass, e.g. Shape or ConicSection
- Which answer is best **depends on the circumstances**

WHAT DOES THE LSP OFFER?

In a word, **polymorphism**

WHAT DOES THE LSP OFFER?

In a word, **polymorphism.**

Can add subclasses without needing to change client code that uses the base class

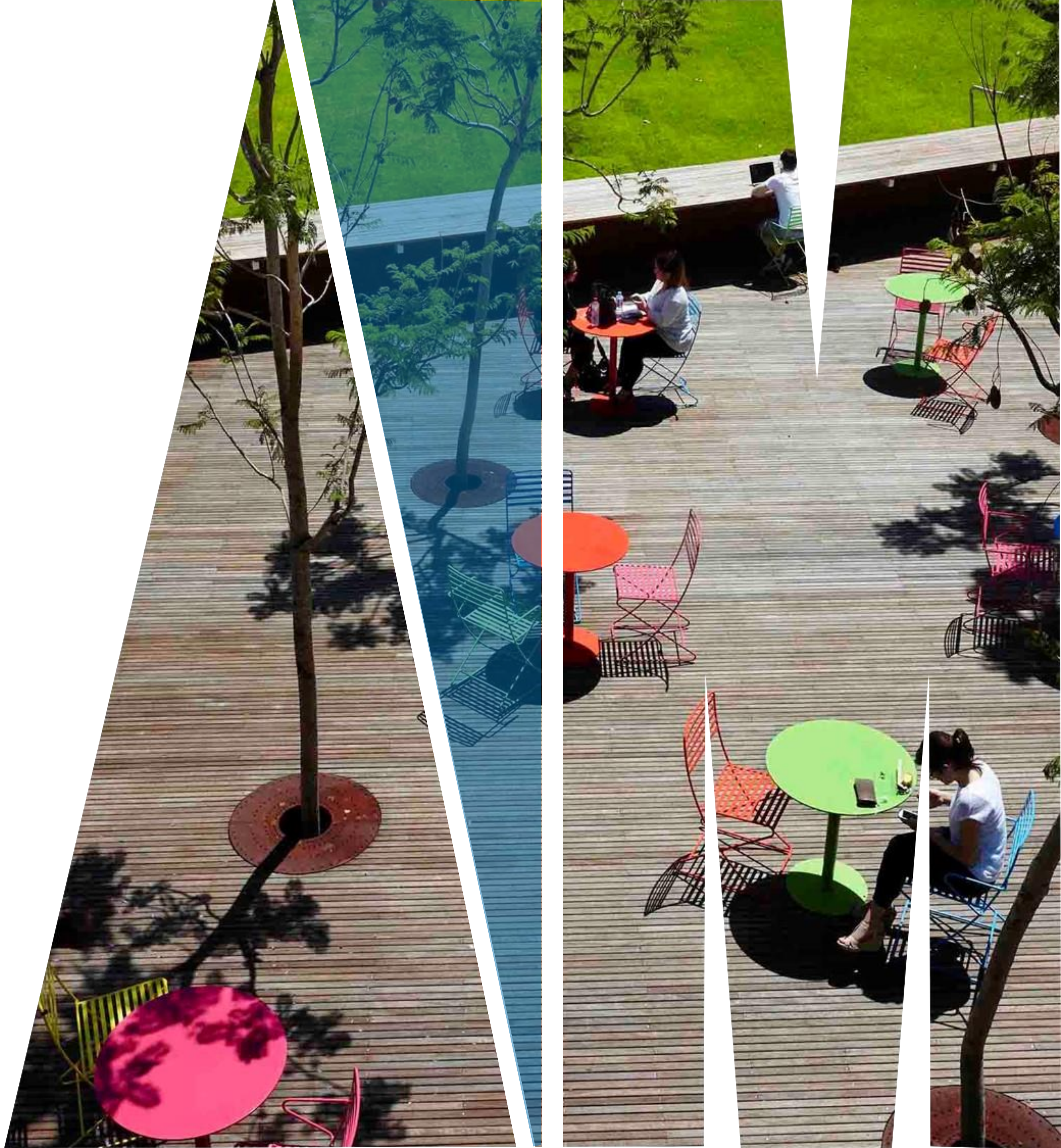
May need to change the line that instantiates the object but no other code needs to change

- as far as it's concerned, it thinks it's still using a base class object.

This makes code **much easier to extend.**

FIT2099 Object-Oriented Design and Implementation

SOLID principles
(Part 3: ISP and DIP)



THE
INTERFACE SEGREGATION PRINCIPLE (ISP)

Clients should not be forced to depend upon interfaces that they do not use.
-- Robert C. Martin

This seems obvious, but is surprisingly hard to do in practice

- your abstractions start out nice and clean, but it is hard to keep them that way over time

THE

INTERFACE SEGREGATION PRINCIPLE (ISP)

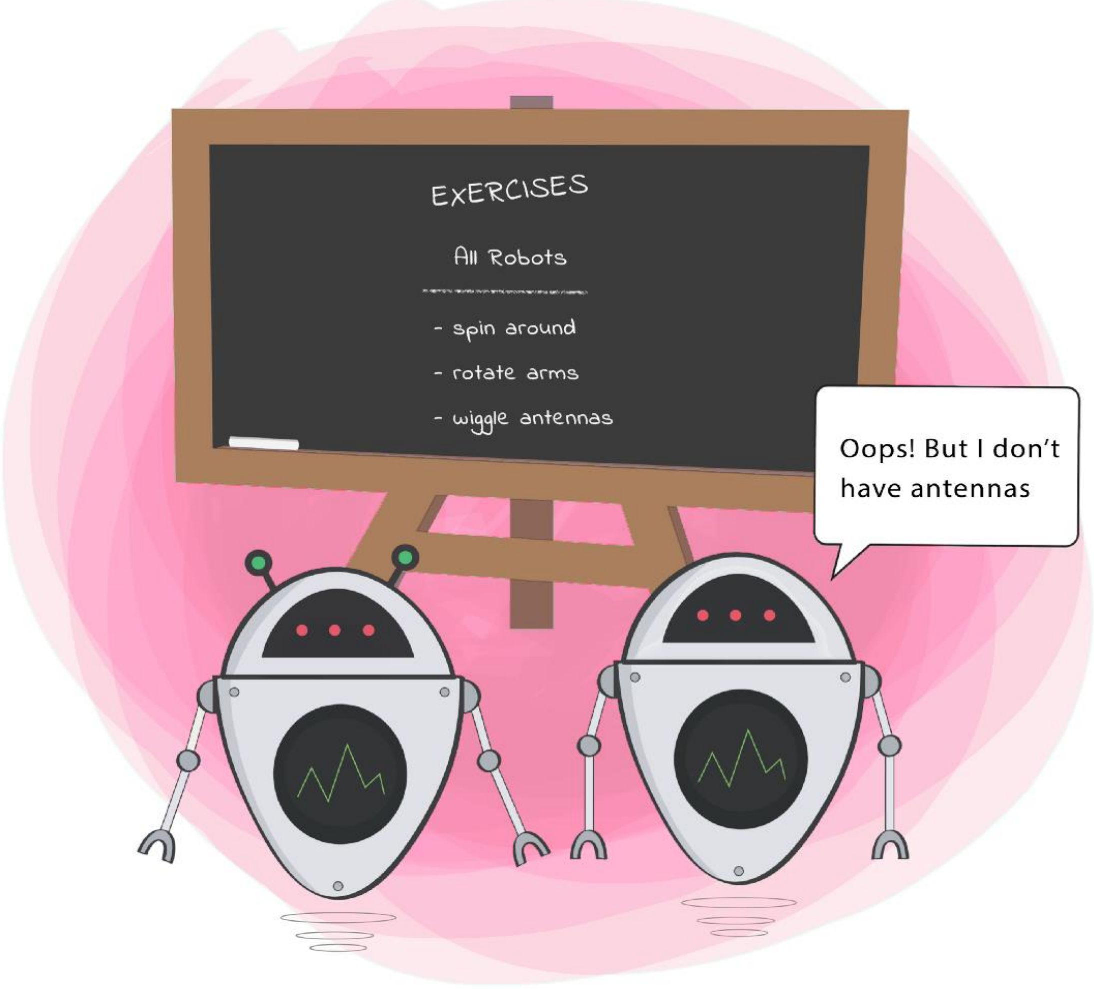
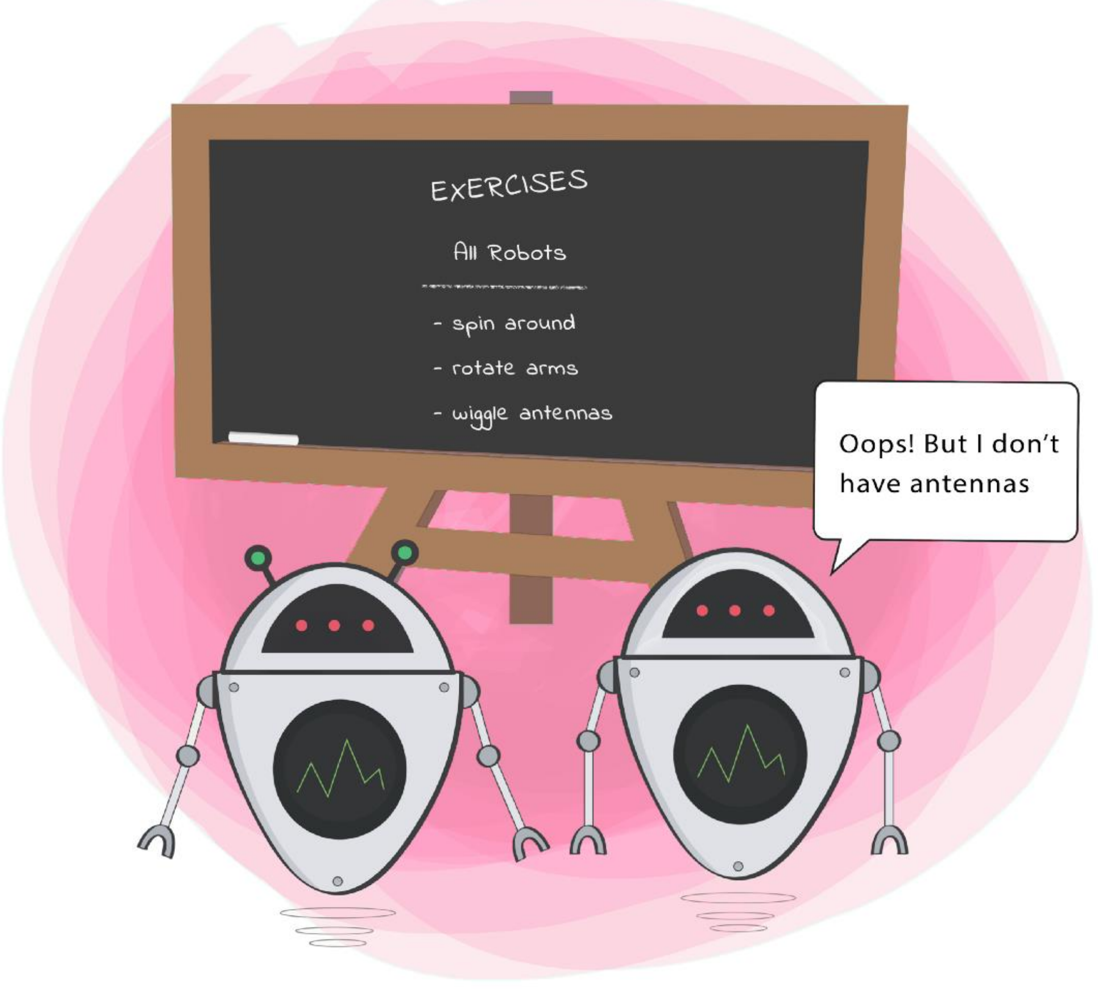


Illustration by [Ugonna Thelma]

THE
INTERFACE SEGREGATION PRINCIPLE (ISP)



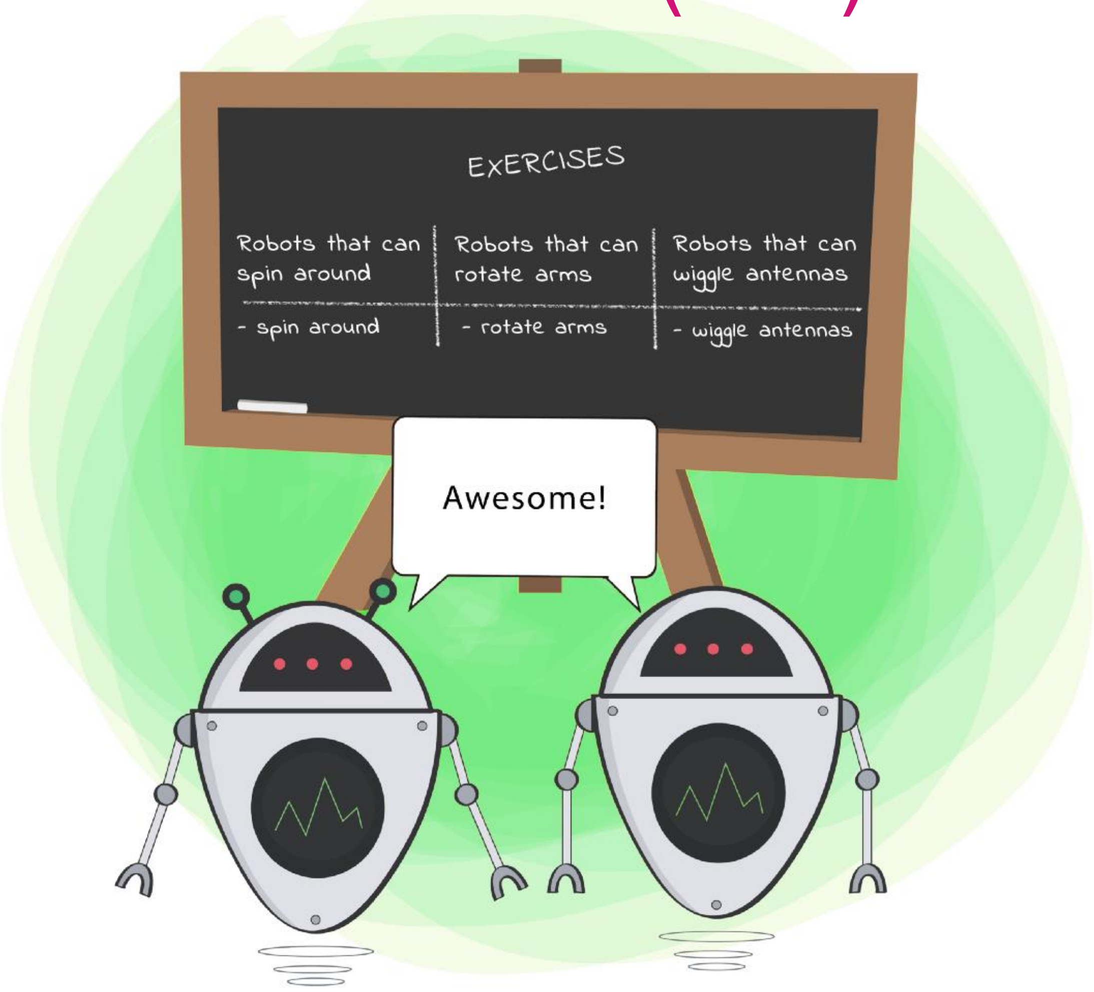


Illustration by [Ugonna Thelma]

INTERFACE POLLUTION

Imagine a program that displays a calculator for **university students**. We define a **Calculator interface** to represent operations it can do

This is great for students doing advanced maths, but what if we want a version of the application for **school kids**?

- they don't need trigonometry or logs
- basic maths functions only

Calculator for that version would have to support **unnecessary functions**!

```
public interface Calculator
        public double add();
        public double subtract();
        public double multiply();
        public double divide();
        public double sin();
        public double cos();
        public double tan();
        public double log();
        public double sqrt();
}
```

FIXING INTERFACE POLLUTION

Can fix this by **segregating interfaces**

- one for **basic** calculations
- one for **advanced** calculations

The primary school version can implement **BasicCalc**

The advanced version can implement both: **BasicCalc** and **AdvCalc**

```java
public interface BasicCalc {
        public double add();
        public double subtract();
        public double multiply();
        public double divide();
}

public interface AdvCalc {
        public double sin();
        public double cos();
        public double tan();
        public double log();
        public double sqrt();
}
```

KEEPING YOUR INTERFACES
SMALL

If you're defining an interface (in the Java sense) you need to resist the urge to make them as large as possible

- remember, **you can implement as many interfaces as you like**

Each small interface should represent one quality that the implementing code should have

- consider standard Java interface `Comparable<T>`
  - means instances can be compared to an instance of type T
  - supports comparison operations

BENEFITS OF FOLLOWING
ISP

If your interfaces are small, you will find it easier to **add new features to the code without needing to refactor**

This makes your system **more extensible**

By keeping interfaces small the classes that implement them have **higher chances to fully substitute** the interface

Classes that implement small interfaces are **more focused** and tend to have a **single purpose**

THE

DEPENDENCY INVERSION PRINCIPLE (DIP)

High-level modules should not depend on low-level modules. Both should depend on abstractions.

Abstractions should not depend on details. Details should depend on abstractions.

-- Robert C. Martin

This is an "**inversion**" because if you are doing **top-down design**, you often end up with a high level module that calls methods in (i.e. depends on) low-level modules

NOTE: We will cover this briefly here, and return to this principle in later lectures on abstraction

THE

DEPENDENCY INVERSION PRINCIPLE (DIP)

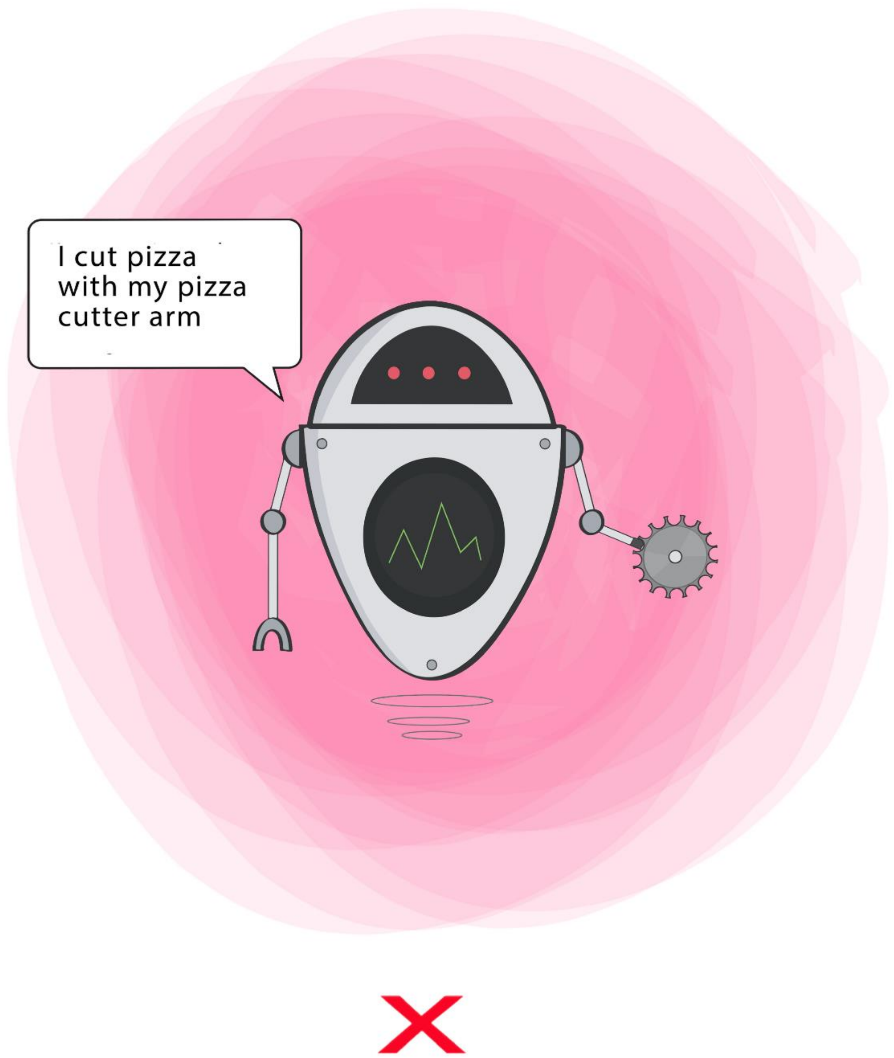
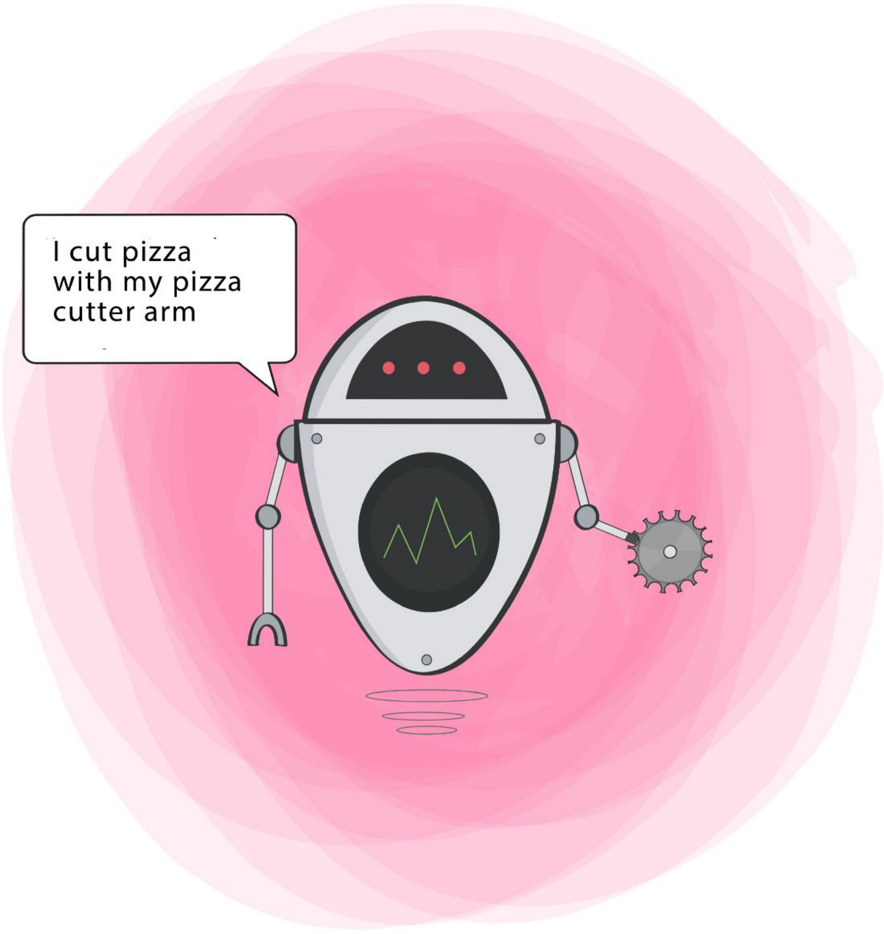


Illustration by [Ugonna Thelma]

THE
DEPENDENCY INVERSION PRINCIPLE (DIP)



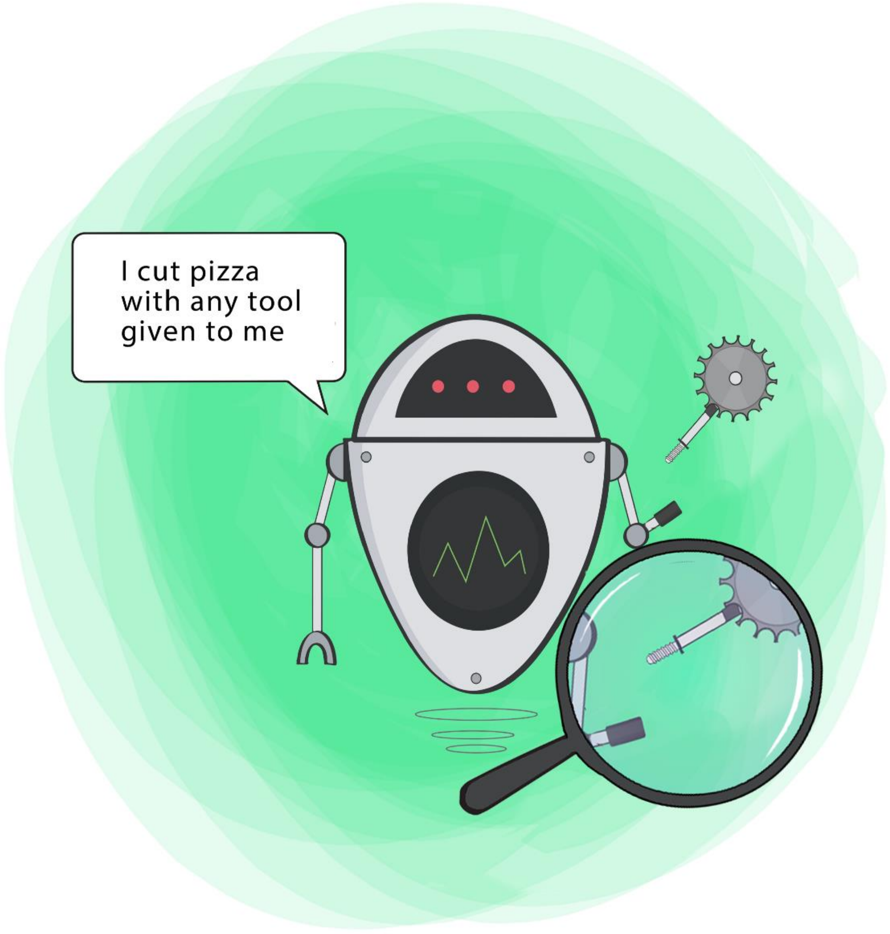


Illustration by [Ugonna Thelma]()

WHAT DOES THE DIP OFFERS TO YOU?

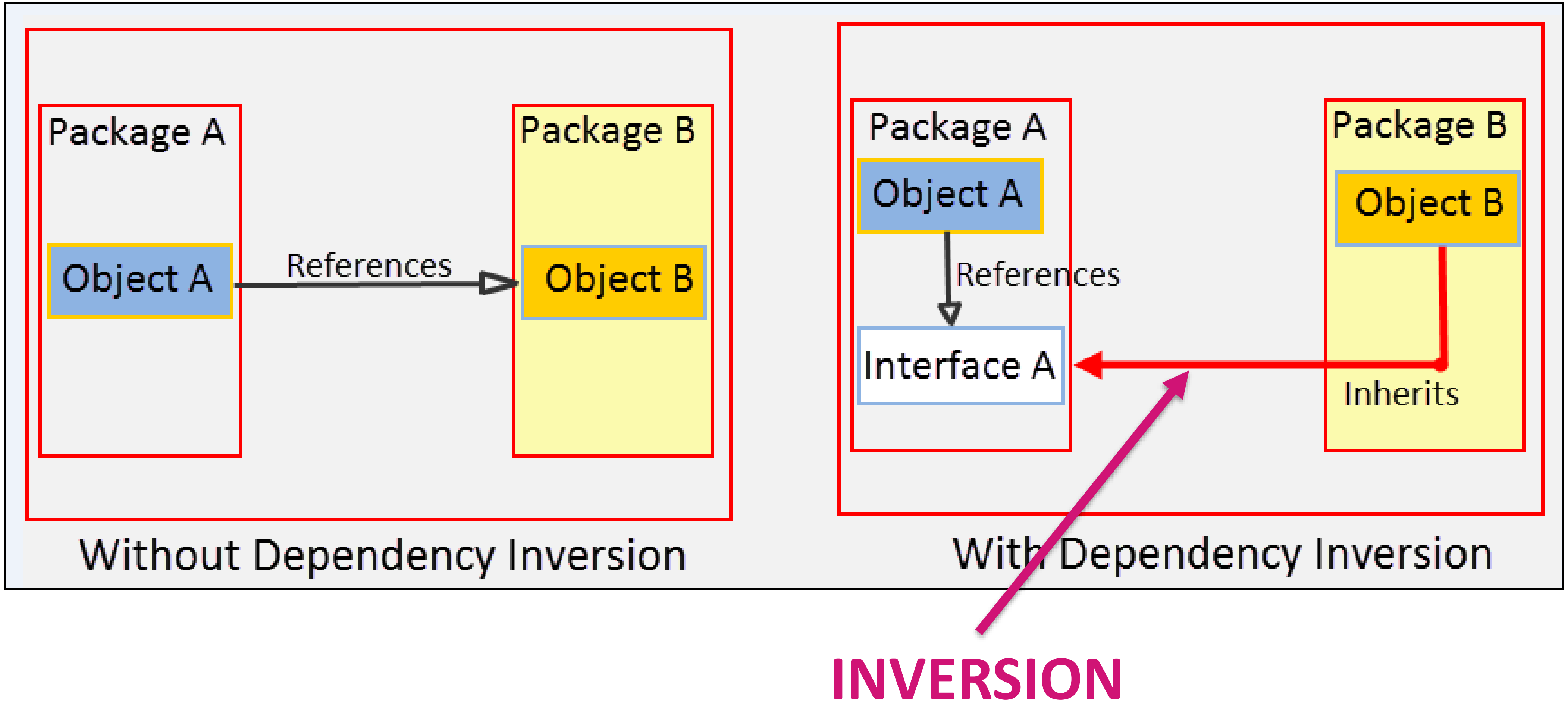
When high level modules depend on interfaces rather than detailed low-level code, both parts are **insulated from change**

- that is, changes in low-level code won’t mean that high-level code has to be refactored
- and changes in high-level code won’t flow on to low-level code

Essentially, putting in **an abstraction layer limits the amount of effort** required to modify the system

- as long as the interface doesn’t change, the parts of the system that depend on it remain independent

WHAT DOES THE DIP OFFERS TO YOU?



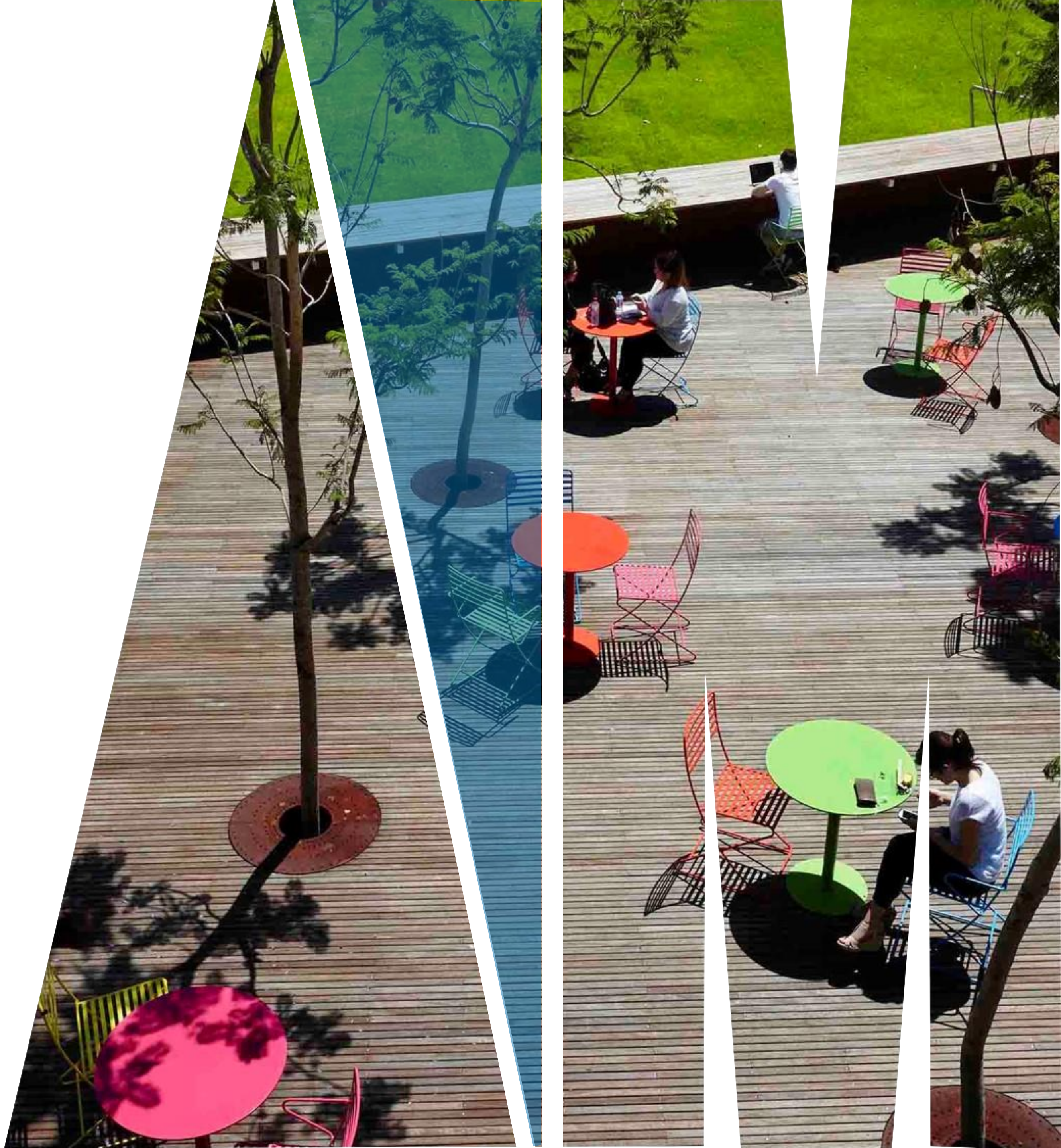
MONASH University

Summary

S	Single Responsibility Principle
O	Open Closed Principle
L	Liskov Substitution Principle
I	Interface Segregation
D	Dependency Inversion Principle

MONASH University

Thanks



MONASH University