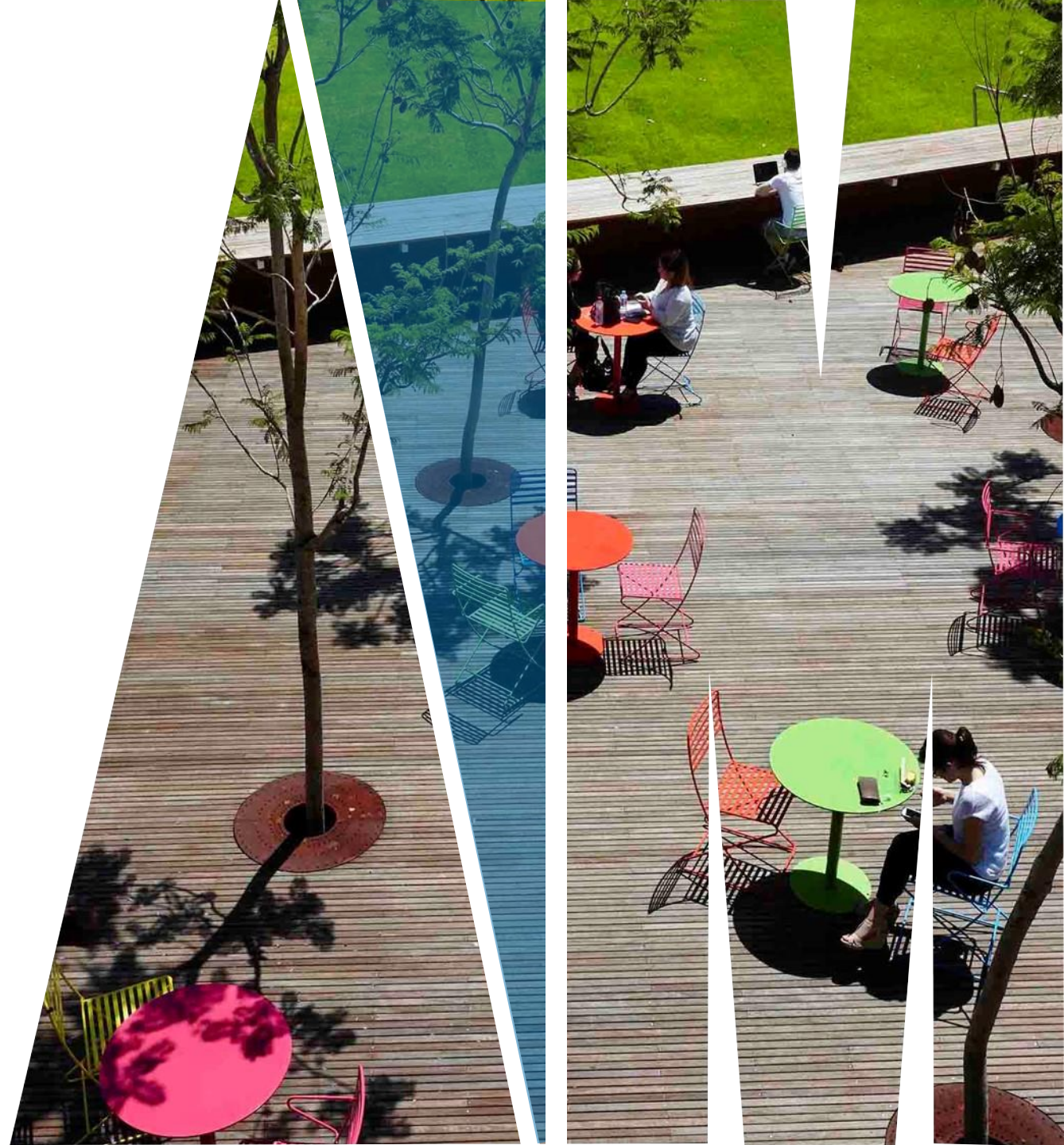


# **FIT2099 Object-Oriented Design and Implementation**

## Review of Abstraction



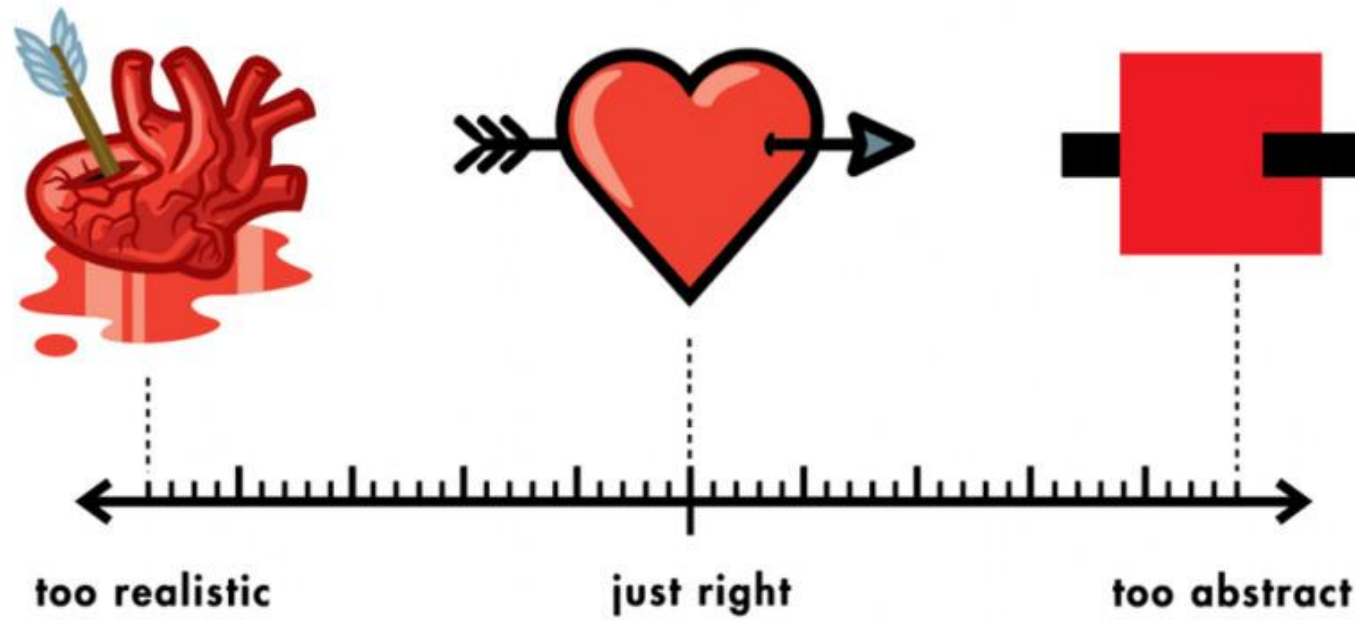
# Outline

## Java Language Features that support abstraction

- classes
- visibility
- abstract classes and hinge points
- packages and abstraction layers/APIs
- interfaces

# AN ABSTRACTION LAYER

An **abstraction layer** is the publicly-accessible interface to a class, package, or subsystem





# SIMPLE AND COMPLEX ABSTRACTION LAYERS



Well-designed abstraction layers simplify the use of your classes and packages

Make sure that **the complexity of your software interface (API) does not exceed the likely benefit to client code**: nobody would buy a car with controls that looked like the Airbus cockpit.

# RECAP

# ABSTRACTION

We want to make **programming easier**.

To do this, we:

- **bundle related pieces of code** together into modules
  - not too many, or the modules themselves become hard to maintain
- work out **how each module should look** from the point of view of other modules
- **minimise the amount of thought** that other programmers have to put into using our module
- **hide everything else** within the encapsulation boundary

# USING ABSTRACTION AT CODE LEVEL

Abstraction is a design principle rather than a programming technique

- caution: don't mistake these programming language features for principles of abstraction!

But it's useful, so most programming languages offer features that support it

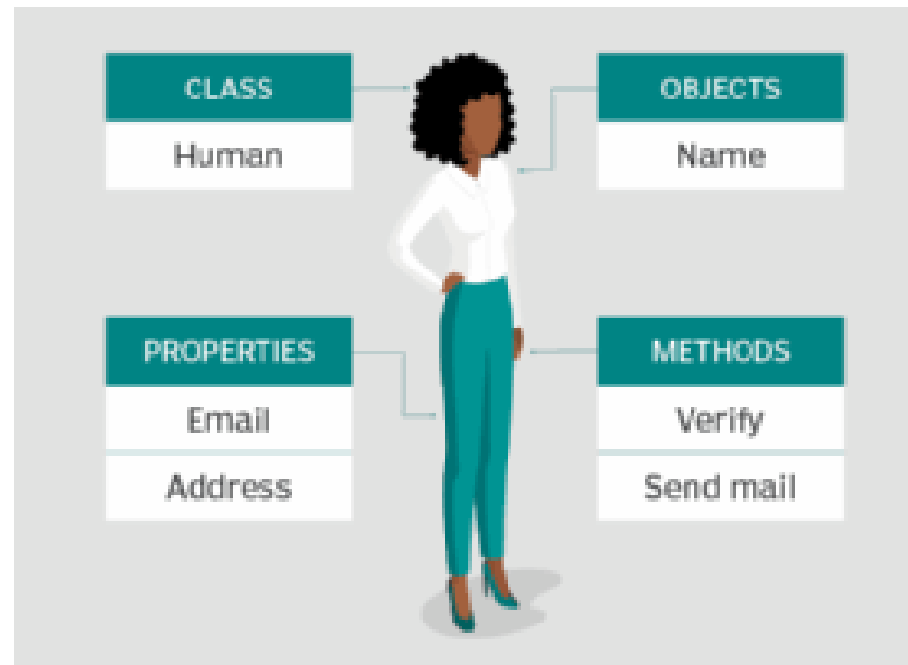
This lesson covers some of the features of Java that can help you write code that makes good use of abstraction

- you've seen all of them before if you've been keeping up with lectures and readings
- today we'll do a deeper dive into some of the more complex language features and relate them specifically to abstraction



# THE CLASS

The class is **the most important mechanism for abstraction** in most OO languages, and that includes Java



# THE CLASS

A well-designed class should

- **represent a single concept** within your system (with a responsibility that is easy to express)
- **expose a public interface** that allows it to respond to messages in order to fulfil its responsibility
- **hide any implementation details** that don't directly fulfil that responsibility
- ensure that its attributes are in a valid condition rather than **relying on client code** to maintain its state (e.g. reduce connascence).



# VISIBILITY MODIFIERS

You've seen these before: **private, protected, public**

Deciding what to hide and what to expose is a big part of applying abstraction in your code!

| Modifier  | Class | Package | Subclass | Global |
|-----------|-------|---------|----------|--------|
| Public    | Yes   | Yes     | Yes      | Yes    |
| Protected | Yes   | Yes     | Yes      | No     |
| Default   | Yes   | Yes     | No       | No     |
| Private   | Yes   | No      | No       | No     |

# VISIBILITY MODIFIERS

Take great care over what is and is not visible from outside your class

- rule of thumb: if in doubt, make it **private**
- **only provide getters and setters** if you're sure that external classes need to indirectly manipulate some internal data



# VISIBILITY MODIFIERS

Remember, if you leave out the visibility modifier, your class/attribute/method will be **visible within the package** in which it is declared

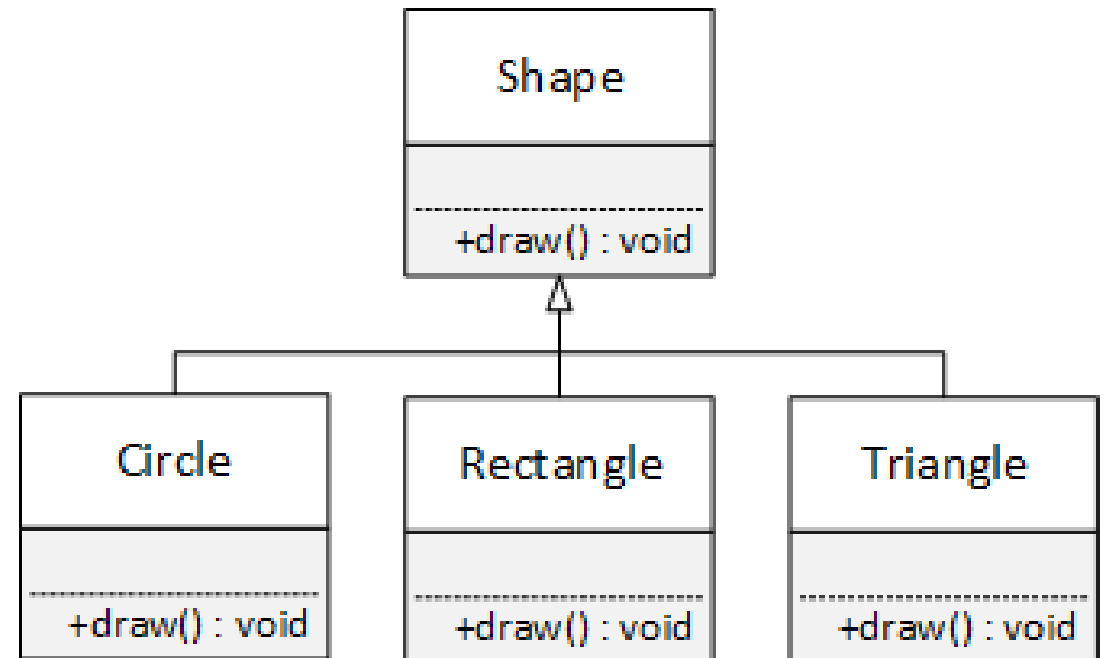
- annoyingly, there is **no explicit package modifier in Java**

| Modifier  | Class | Package | Subclass | Global |
|-----------|-------|---------|----------|--------|
| Public    | Yes   | Yes     | Yes      | Yes    |
| Protected | Yes   | Yes     | Yes      | No     |
| Default   | Yes   | Yes     | No       | No     |
| Private   | Yes   | No      | No       | No     |

# THE ABSTRACT CLASS

As you've seen, abstract classes **can't be instantiated**

- they may lack important components, such as method bodies
- so they **are only useful as base classes**
- Therefore, what they're for?



# THE ABSTRACT CLASS

In **Java**, a subclass inherits all the non-private methods declared in the base class

- so the compiler will let you assign an instance of the subclass to a base class reference:

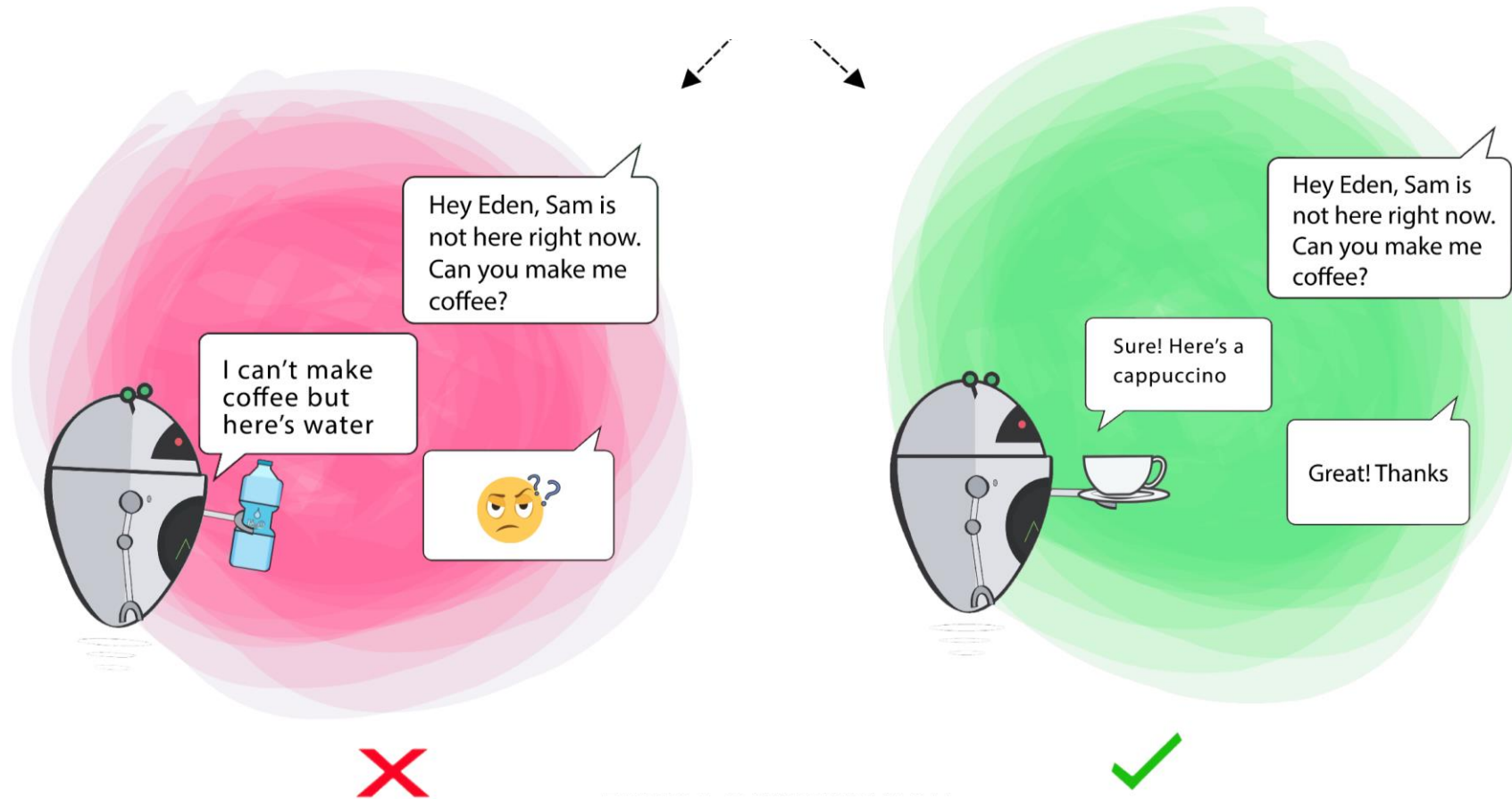
```
AbstractBaseClass a = new ConcreteSubclass();
```

- legal because all methods declared in **AbstractBaseClass** must exist in **ConcreteSubclass**

So client code can be passed an instance of some concrete subclass without ever needing to know its exact type. All it needs to know is that it does everything the abstract class says it can.

- very useful, e.g. for **Dependency Injection**

# THE LSKOV SUBSTITUTION PRINCIPLE



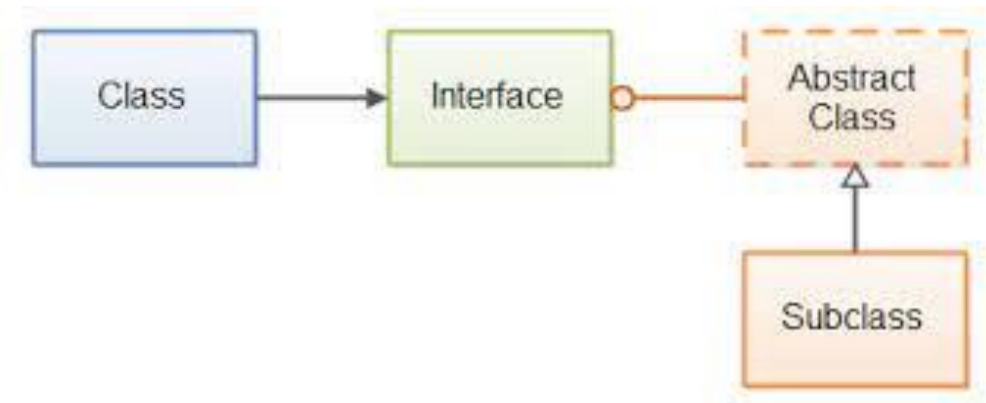


# THE INTERFACES

Used extensively in both Java and C#

- they **separate** the publicly-accessible **interface** from their **implementations**
- many interfaces in the **Java libraries**
- they are Java's only mechanism for **multiple inheritance**

From a design perspective, an interface can be thought of as **an abstract class taken to extremes**



# THE INTERFACES

There are perfectly respectable OO languages that don't support interfaces at all

- example: C++
- even in C++ you can declare an abstract base class that has no method bodies
- C++ has multiple inheritance so Java-style interfaces are not necessary

Main thing to bear in mind: everything we're saying about the use of **abstract classes** to define an abstract interface to a component applies to **interfaces** as well.


# THE HINGE POINTS

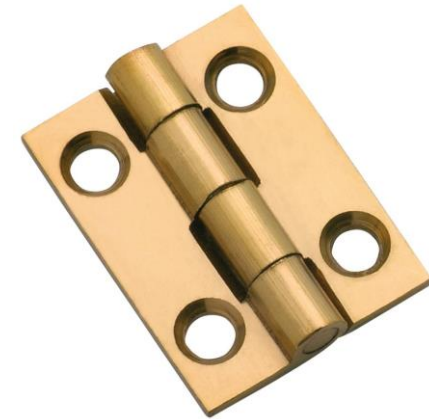
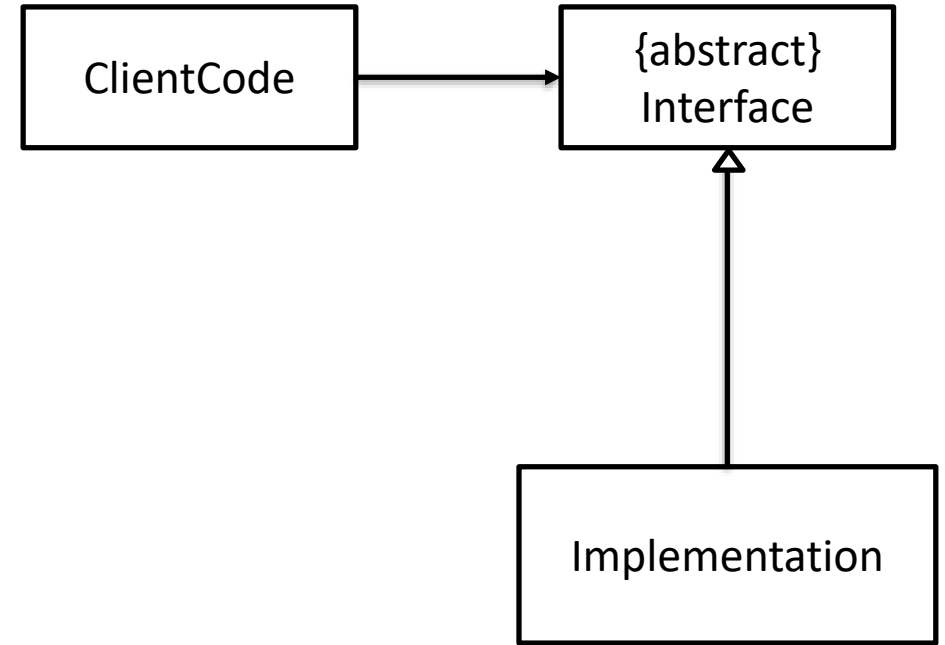
Applying dependency inversion to a single relationship gives you a design that looks like this

This is a powerful technique!

- it means that **ClientCode** doesn't have to care about anything that happens in the Implementation
- similarly, **Implementation** doesn't have to care about **ClientCode**

ClientCode and Implementation **can be changed** as much as you like provided they both respect the **Interface**

- this gives your design a kind of “**hinge**” 
- the parts can move around freely except where they are joined
- design is constrained, but as lightly as possible



# THE DEPENDENCY INVERSION PRINCIPLE (DIP)

The dependency inversion principle (DIP): **high-level modules should not depend on low-level modules. Both should depend on abstractions.** Why is this an “inversion”?

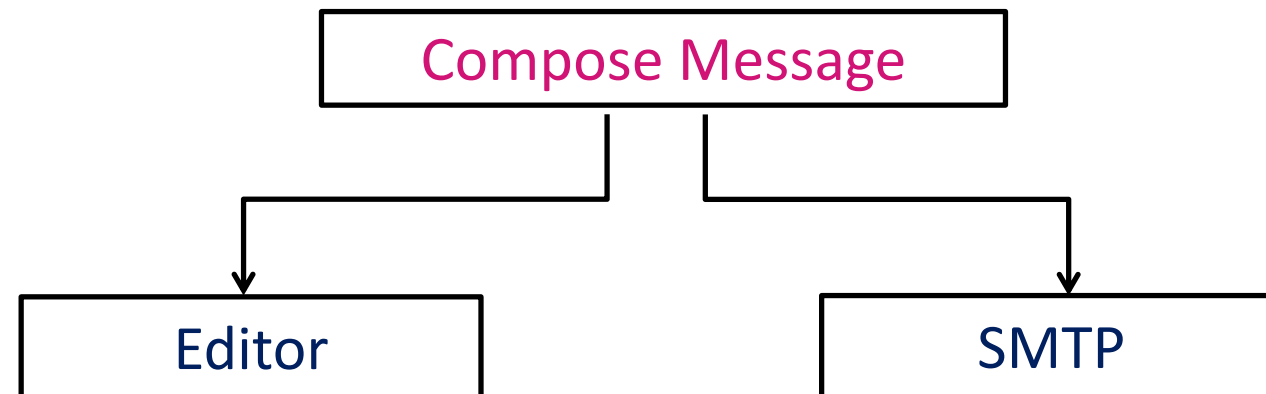
- consider a traditional top-down design: break problem down into subproblems, solve those and recombine to solve original problem
- example: messaging system that sends email (SMTP is an email transmission protocol)

Compose Message

# THE DEPENDENCY INVERSION PRINCIPLE (DIP)

The dependency inversion principle (DIP): **high-level modules should not depend on low-level modules. Both should depend on abstractions.** Why is this an “inversion”?

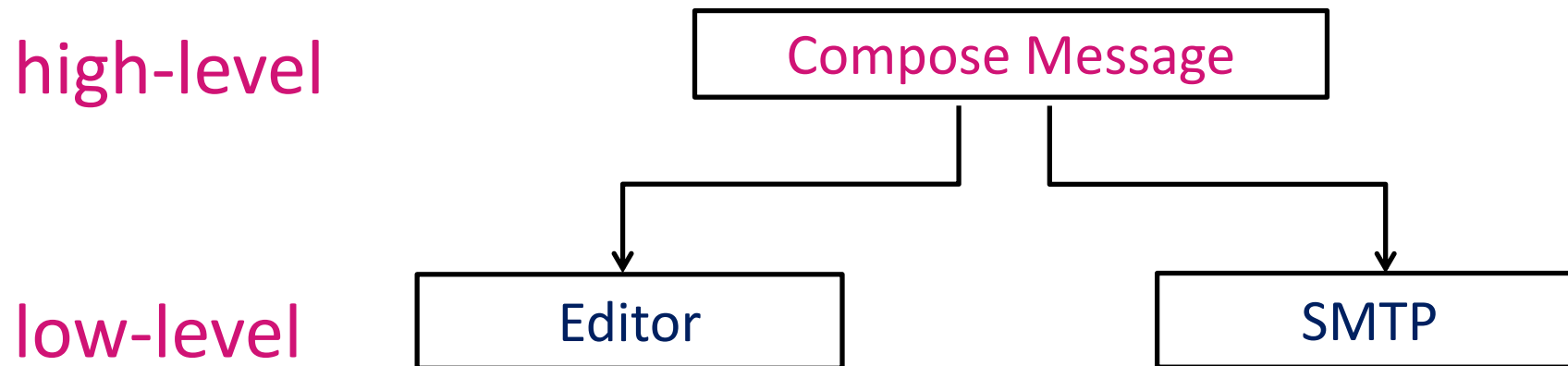
- consider a traditional top-down design: break problem down into subproblems, solve those and recombine to solve original problem
- example: messaging system that sends email (SMTP is an email transmission protocol)



# THE DEPENDENCY INVERSION PRINCIPLE (DIP)

The dependency inversion principle (DIP): **high-level modules should not depend on low-level modules. Both should depend on abstractions.** Why is this an “inversion”?

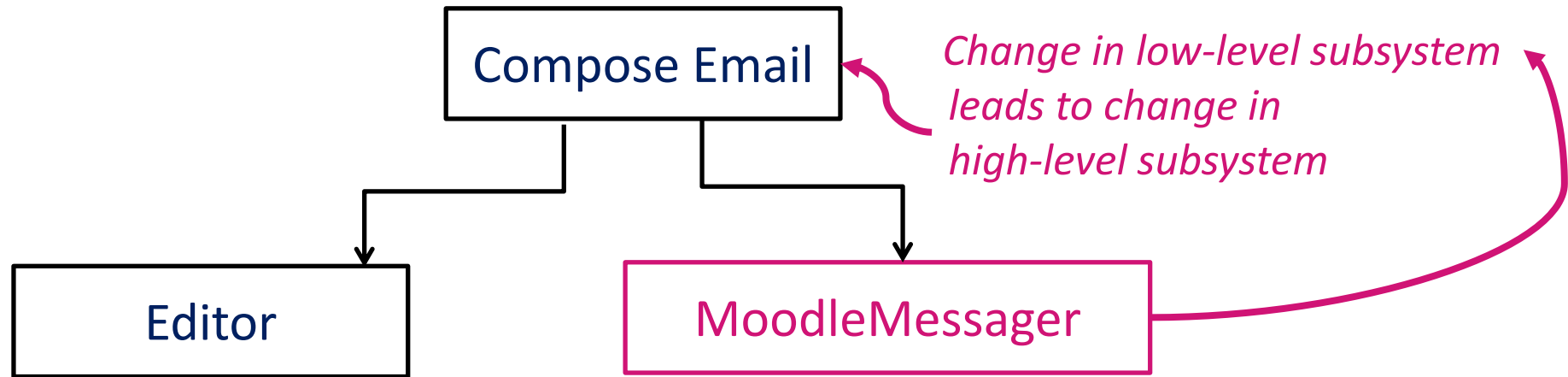
- consider a traditional top-down design: break problem down into subproblems, solve those and recombine to solve original problem
- example: messaging system that sends email (SMTP is an email transmission protocol)





# A DIP EXAMPLE

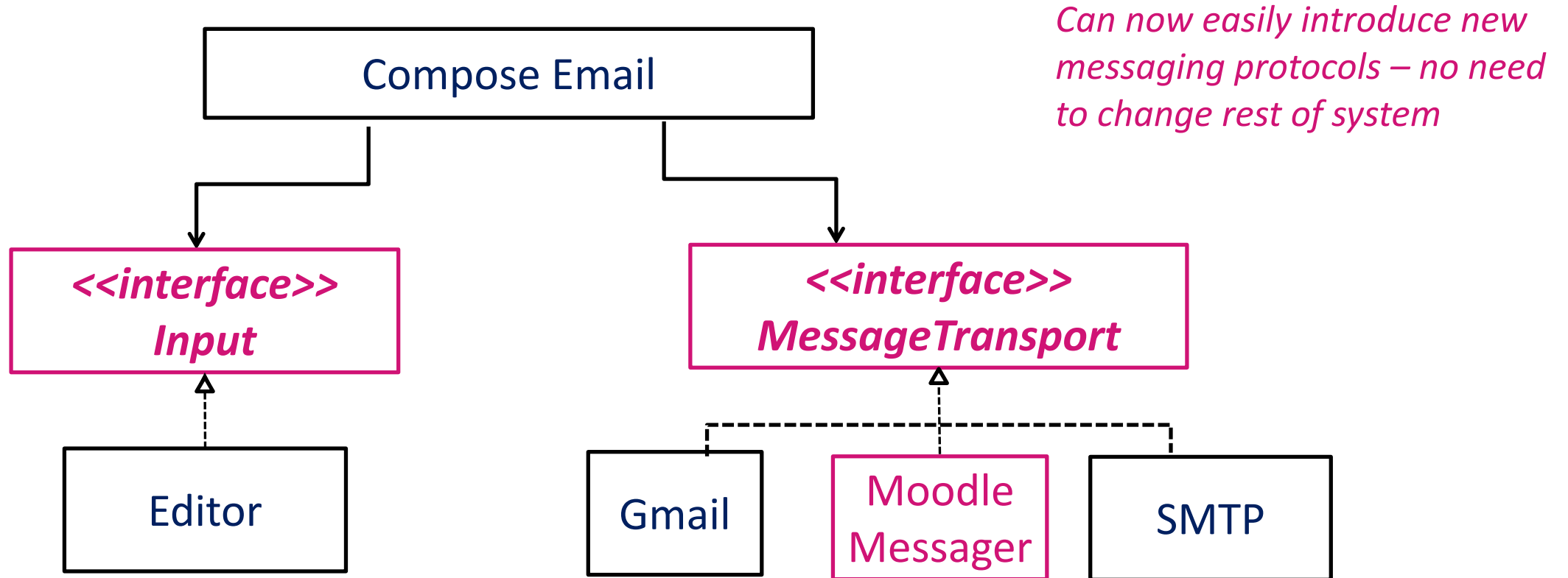
Problem: suppose we want to change the way we send messages  
– now, we want to send them as Moodle messages



That's not good! Whenever we change a low-level implementation detail, we could end up needing to change our high-level logic.

# A DIP EXAMPLE

**Solution:** define an **abstract interface** – high-level components depend on interface, low-level components implement it

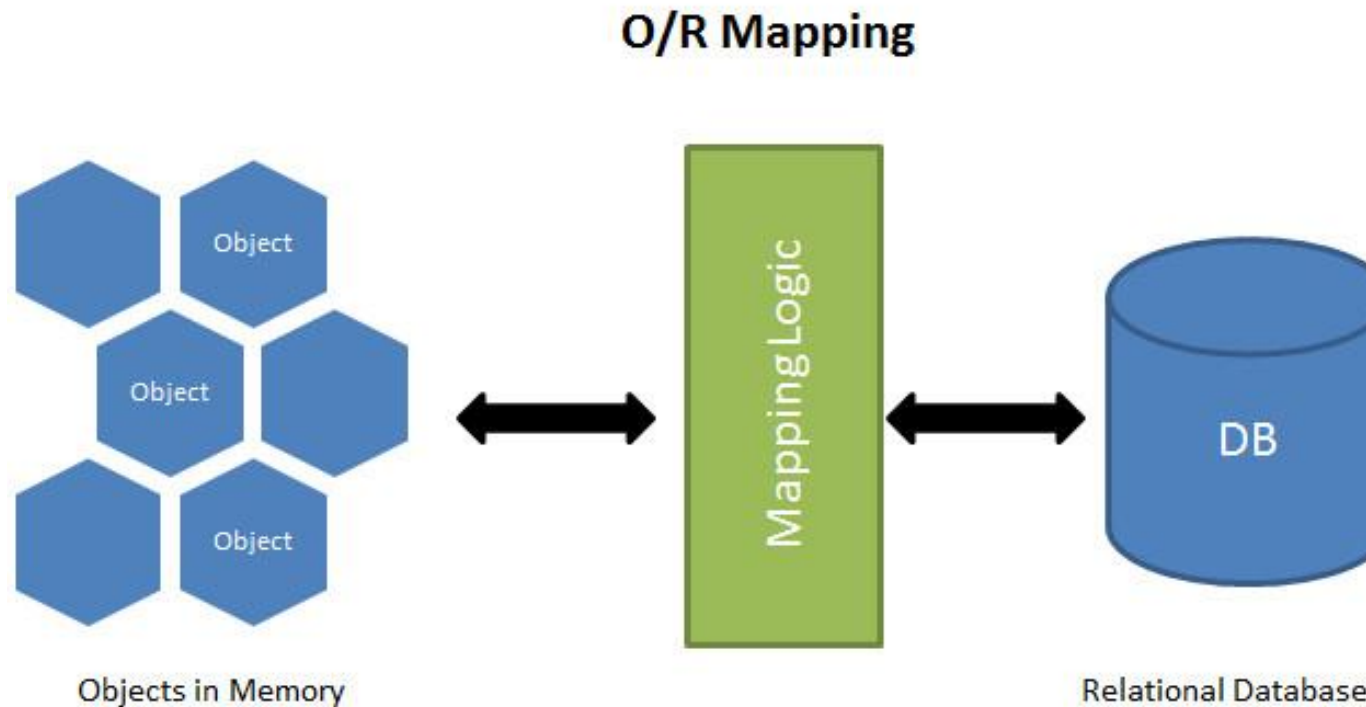


# EXAMPLE

## ORMs

It is notoriously painful to combine a relational database with an object-oriented program

- this is due to the so-called **object-relational impedance mismatch**

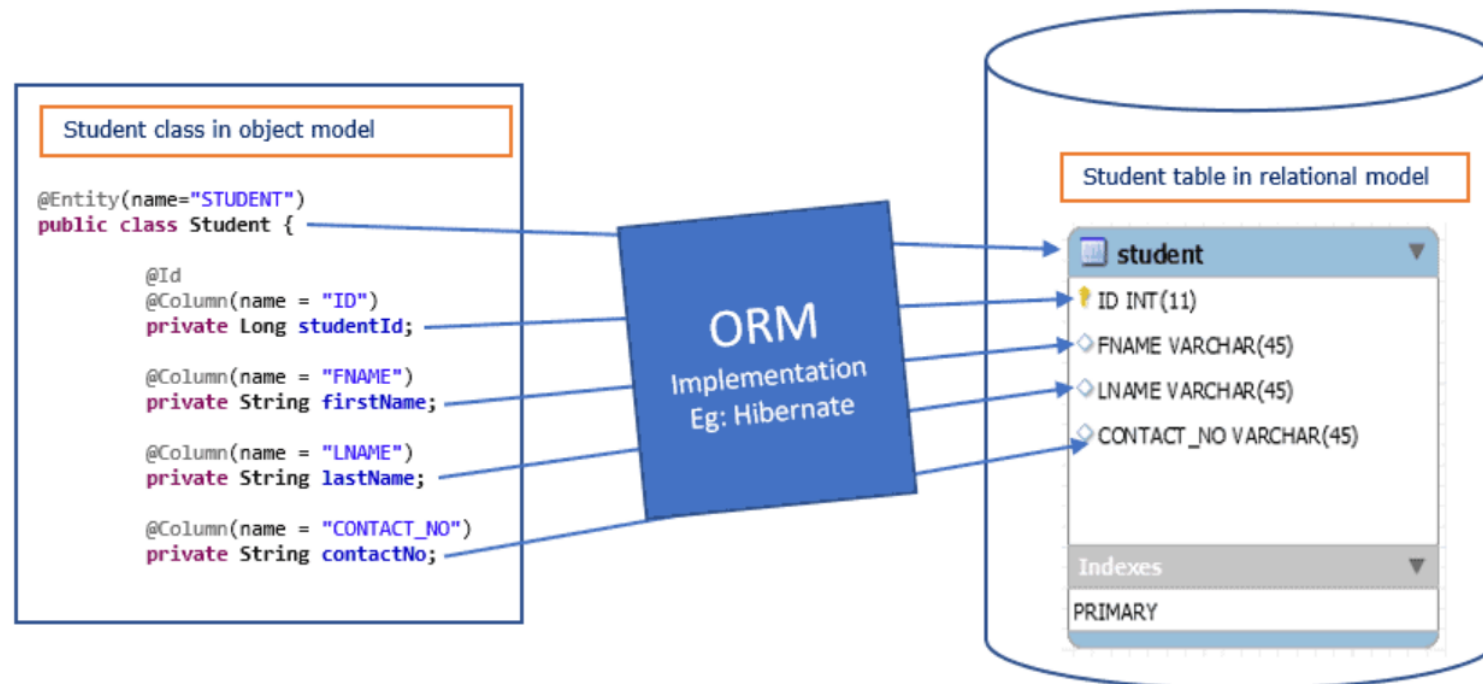


# EXAMPLE

## ORMs

RDBMSes model the world in terms of **tables**; OOP uses **objects** that respond to messages

RDBMS languages such as SQL are typically **declarative** (i.e. they make statements that define how the world is); all the major OO languages are **imperative** (i.e. lists of orders about what to do)

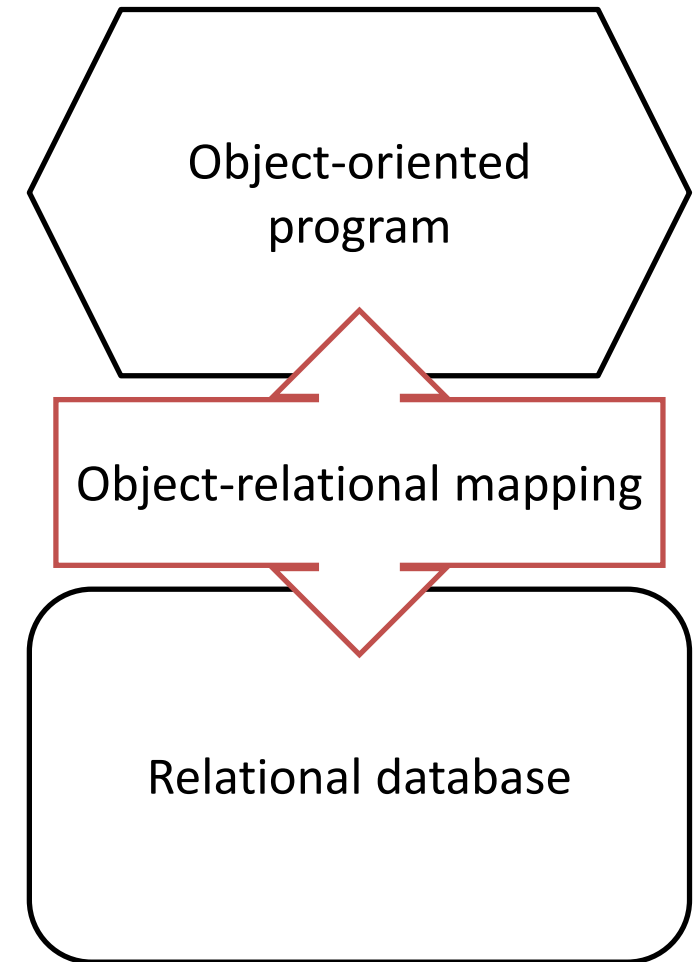


# EXAMPLE

## ORMs

Solution: introduce an **abstraction layer** between the OO side and the DB

- this is called an ORM (Object-Relational Mapping)
- provides classes and objects to the OOP
- maps changes to OO objects to RDBMS commands



# SOLUTIONS FOR JAVA, C# AND OTHER OO LANGUAGES AND TECHNOLOGIES

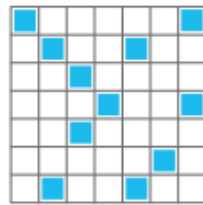
Hibernate



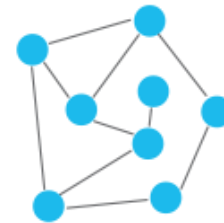
## Other solutions

NoSQL or an OODBMS

NoSQL  
Database



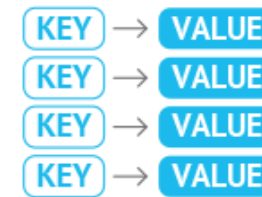
Column-Family



Graph



Document



Key-Value



# 3 COMMON MISTAKES RELATED TO ABSTRACTION

# COMMON MISTAKE #1

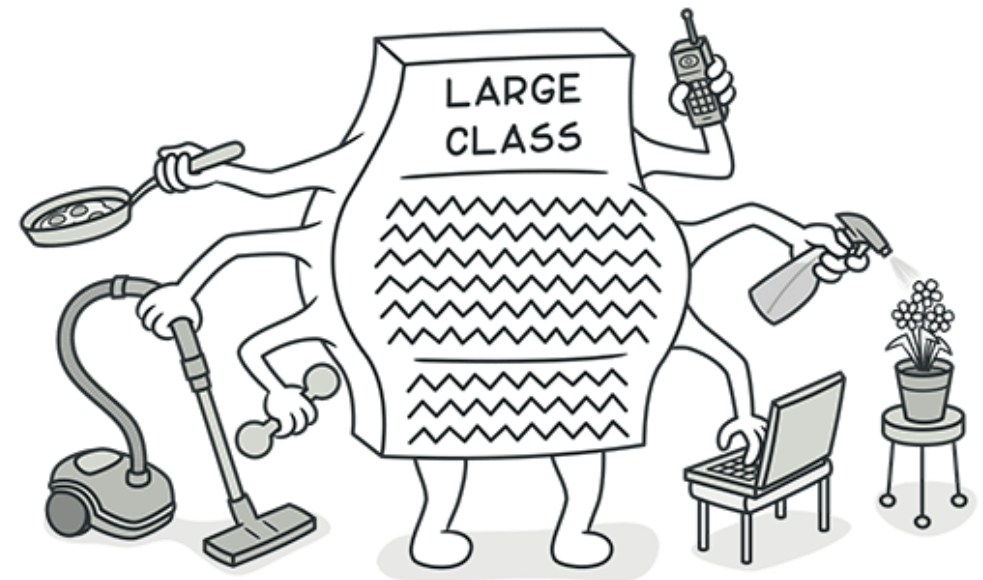
## MISUNDERSTANDING SIMPLICITY

Confusing simple design with **fewest possible classes**  
this will indeed reduce the number of classes that can interact  
may simplify the interactions between them

BUT!

We are trying to simplify the program overall  
fewer classes means **bigger** classes

- ❑ remember we are trying to minimise the number of things the programmer has to think about simultaneously
- ❑ usually have to think about everything in the class you're modifying
- ❑ so **no net gain from making classes bigger**



# INTERACTIONS WITHIN A SYSTEM

Imagine a system in which any part can potentially interact with every other part. If this system is divided into  $N$  components, in how many different ways can they interact?



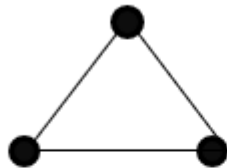
1

0



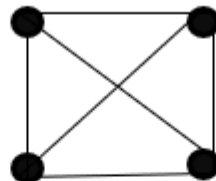
2

1



3

3



4

6



5

10



$N$

$$\frac{N(N-1)}{2}$$

# CLASSES ARE SYSTEMS TOO

If you have  $N$  entities in your system and no restrictions on how they may interact, there are  $O(N^2)$  potential interactions between those entities

If you try to fix the problem by combining classes into larger classes, leaving many methods and attributes private, you will indeed simplify interactions between classes

- but we're trying to reduce the **overall** complexity of the system!
- although you've made things simpler at class level, **those larger classes themselves will be harder to maintain**
- quadratic rise in interactions **within** the class boundary: methods, attributes, etc.

Only way to reduce net complexity is to define relatively **small** classes

- **minimize dependencies between them**
- **hide any information that isn't relevant to other classes**
- **this decreases that  $N$**

# COMMON MISTAKE #2

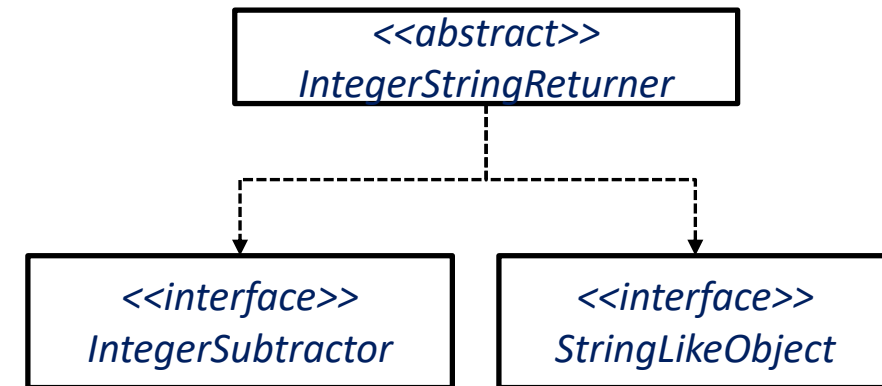
## OVERDOING DEPENDENCY INVERSION

As we have seen, using too few abstract classes is a bad thing. However, using too many abstract classes is also a bad thing:

- **adding an interface adds complexity to your design**
- is worth it if complexity is being taken away elsewhere
- is often worth it if you want to be able **to develop components in isolation** (such as GMailer and MoodleMessenger, earlier)

If using a modern IDE, can use its refactoring tool to quickly and easily extract an interface from an existing class if you need it

- so if you're not sure, leave it out and add it later



## COMMON MISTAKE #3

# CONFUSING ABSTRACTION WITH `abstract`

Using **abstraction** can help you build classes that are easier to use and maintain

- This is not the same as sprinkling the `abstract` keyword into your code – the key to making your code better is thinking carefully about
  - what each code module is for
  - how it should look from other parts of the code
  - which aspects of the code might need to be reused or specialized

Declaring a class to be abstract only means you can't instantiate it. Declaring a method to be abstract only means you need to override it with a concrete implementation in order to be able to instantiate the class it is in. **Neither of these approaches guarantee that your abstraction is useful or clean** – that's a design issue, not an implementation issue.



# Summary

## Java Language Features that support abstraction

- classes
- visibility
- abstract classes and hinge points
- packages and abstraction layers/APIs
- interfaces



MONASH  
University

Thanks



MONASH  
University

