

# Abstraction in OO Design

---

FIT2099: SEMESTER 1 2018

# Recap: What makes programming hard?

---

- Too many things to keep track of simultaneously
  - classes (or modules, or source files)
  - packages (or namespaces, or assemblies)
  - variables
  - lines of code
- Complicated interactions between parts of the program
  - dependencies between modules
  - complex algorithms
- All particularly significant when we need to *change* the program

# Making programming easier

---

We want to design our software in such a way as to make it easier to maintain, extend, and modify

End users use programs, but classes and packages are used by **developers** – if we make their working lives easier, we will produce software more efficiently:

- accrue less technical debt
- make iterative development easier
- respond more readily to changes in requirements or in the development environment

Key factor to consider: **cognitive load**

- limit to human working memory
- high cognitive load leads to more mistakes, slower development

# What is **abstraction**?

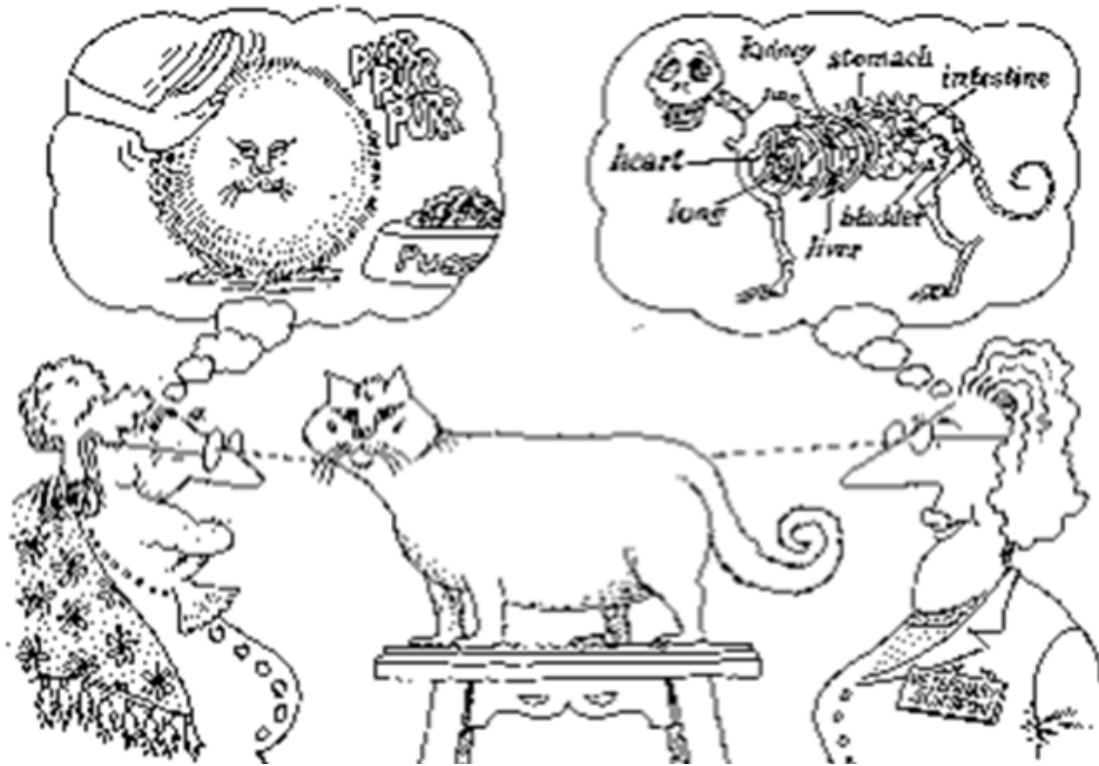
---

According to [dictionary.com](https://www.dictionary.com),

*“the act of considering something as a general quality or characteristic, apart from concrete realities, specific objects, or actual instances.”*

To a software developer, this means deciding

- what information do we need in order to represent some item or concept?
- what should we expose to the rest of the code (i.e. make public) so that we will be able to use this part easily?



What information is relevant depends on how it will be used.

The vet and the owner are both considering the same cat, but the ways they think about it (their **mental models** or abstractions) depend on their relationship to the cat: medical professional versus carer.

# You use abstraction already

---

At its simplest, we perform abstraction whenever we bundle related items together, name the bundle, and then use it.


This includes:

- collecting lines of code together into a **method** or function
- collecting related data into a **class**
- grouping classes into **packages**

All of these things can be used without much thought about their internals...

- but only if they are well-designed!

Our objective this week is to show you some techniques you can use to improve the quality of your object-oriented modelling, and to help you design maintainable code.



# Abstraction, encapsulation, and information hiding

---

It can be hard to distinguish between encapsulation and abstraction. Most of the language features that enable encapsulation also enable abstraction, so if you're trying to understand these concepts purely at source code level, you're likely to get confused.

Here's a guide:

- we use **encapsulation** when we bundle things together
- we use **abstraction** when we decide which things should be bundled together
  - we also use **abstraction** when we decide how things should look from outside (i.e. when we design a class's public interface)
- we use **information hiding** whenever we use an encapsulation mechanism that doesn't allow access from outside
  - private or protected modifiers: keep implementation details hidden
  - local variables: no access from outside the method
  - defensive copying: prevent external code from accessing internal data structures

# Example: strings in C and Java

---

The language C has a string datatype... sort of.

```
char *string = "Hello, world!";  
for (int i = 0; string[i] != '\0'; i++) {  
    ...  
}
```

To use C strings, you need to know that they are represented internally as arrays of char – so their type is pointer to char. You also need to know that the end of the string is marked by a null, '\0' (literally a zero in memory).

**Do you know how Java strings are represented internally? Do you want to have to care?**

It is much easier to use Java strings than C strings because Java presents the programmer with a better abstraction in which more of the implementation details are hidden.



# Separation of concerns



*It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of. This is what I mean by "focussing one's attention upon some aspect": it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect's point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.*

-- E.W. Dijkstra, "On the Role of Scientific Thought"

<https://www.cs.utexas.edu/~EWD/transcriptions/EWD04xx/EWD447.html>

By Hamilton Richards - manuscripts of Edsger W. Dijkstra,  
University Texas at Austin, CC BY-SA 3.0, [https://  
commons.wikimedia.org/w/index.php?curid=4204157](https://commons.wikimedia.org/w/index.php?curid=4204157)

# Separation of concerns

---

For our purposes, a “concern” is a responsibility.

Every module should have a **single, well-defined** set of responsibilities

Responsibilities should overlap as little as possible with other modules

- shared responsibilities often leads to repeated code

Unclear responsibilities make the module **hard to use**.

Not about the code, it's about the conceptual structure underlying the code.

For each module, ask yourself, “what is this module for?”

- if you don't have a clear answer, your design smells

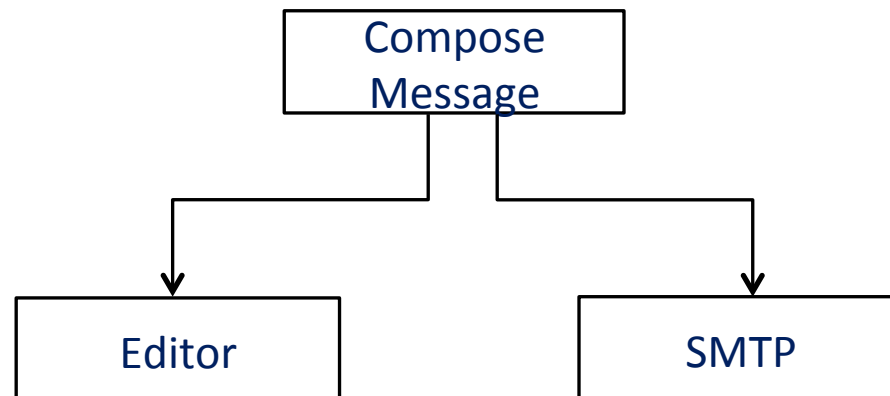
# The Dependency Inversion Principle

---

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”?

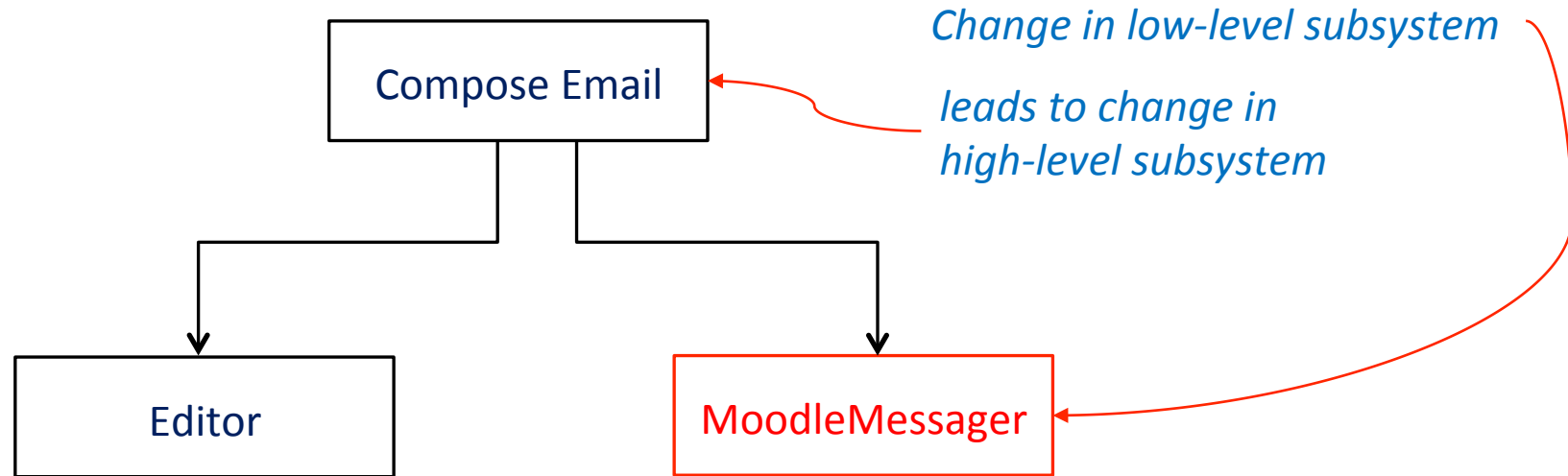
- consider a traditional top-down design: break problem down into subproblems, solve those and recombine to solve original problem
- example: messaging system that sends email (SMTP is an email transmission protocol)



# DIP example

Problem: suppose we want to change the way we send messages

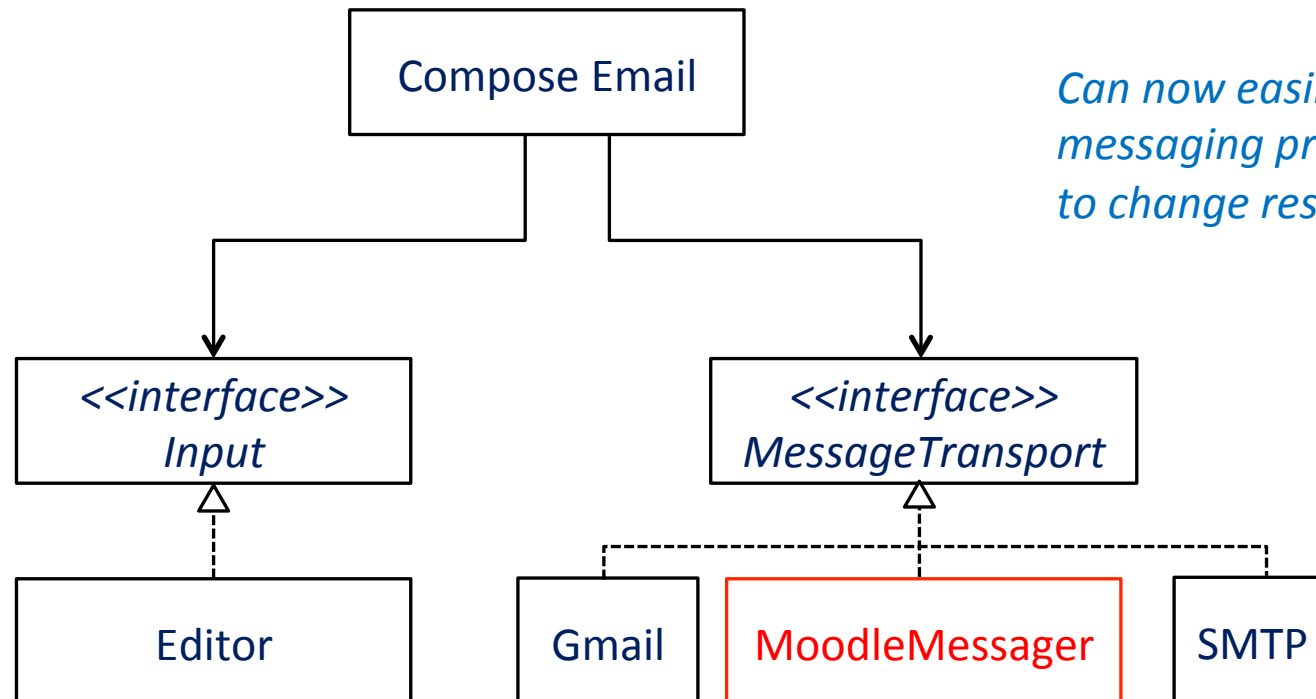
- now, we want to send them as Moodle messages



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

# DIP Example

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



*Can now easily introduce new messaging protocols – no need to change rest of system*

# Common mistakes #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 minimize 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?



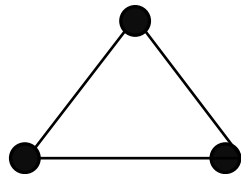
1

0



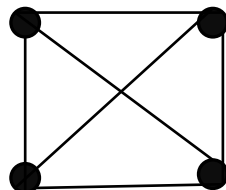
2

1



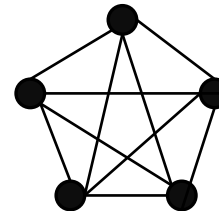
3

3



4

6



5

10

$N$

$N(N-1)/2$

# 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 mistakes #2: overdoing dependency inversion

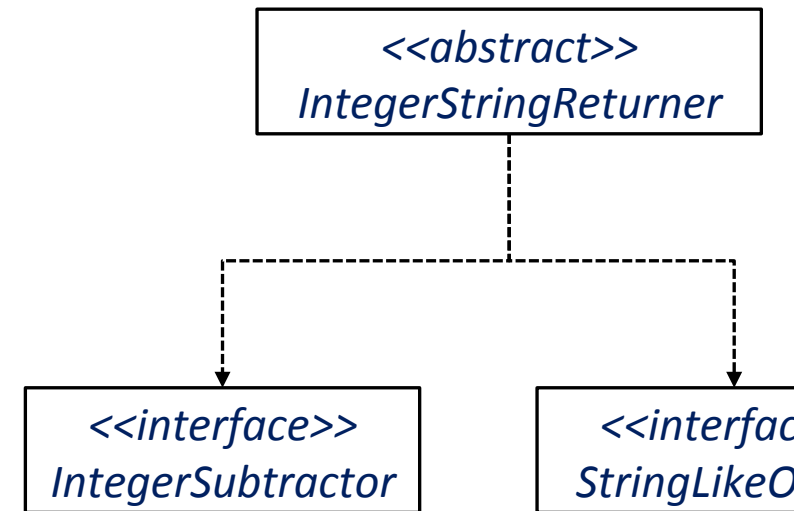
As we have seen, using too few abstract classes is a bad thing.

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 for an OO language such as Java or C#, 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
- but pay attention to the design of the classes you do implement!



# Common mistakes #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

---

## Abstraction

- definition
- relationship to other OO principles: encapsulation and information hiding
- role in design