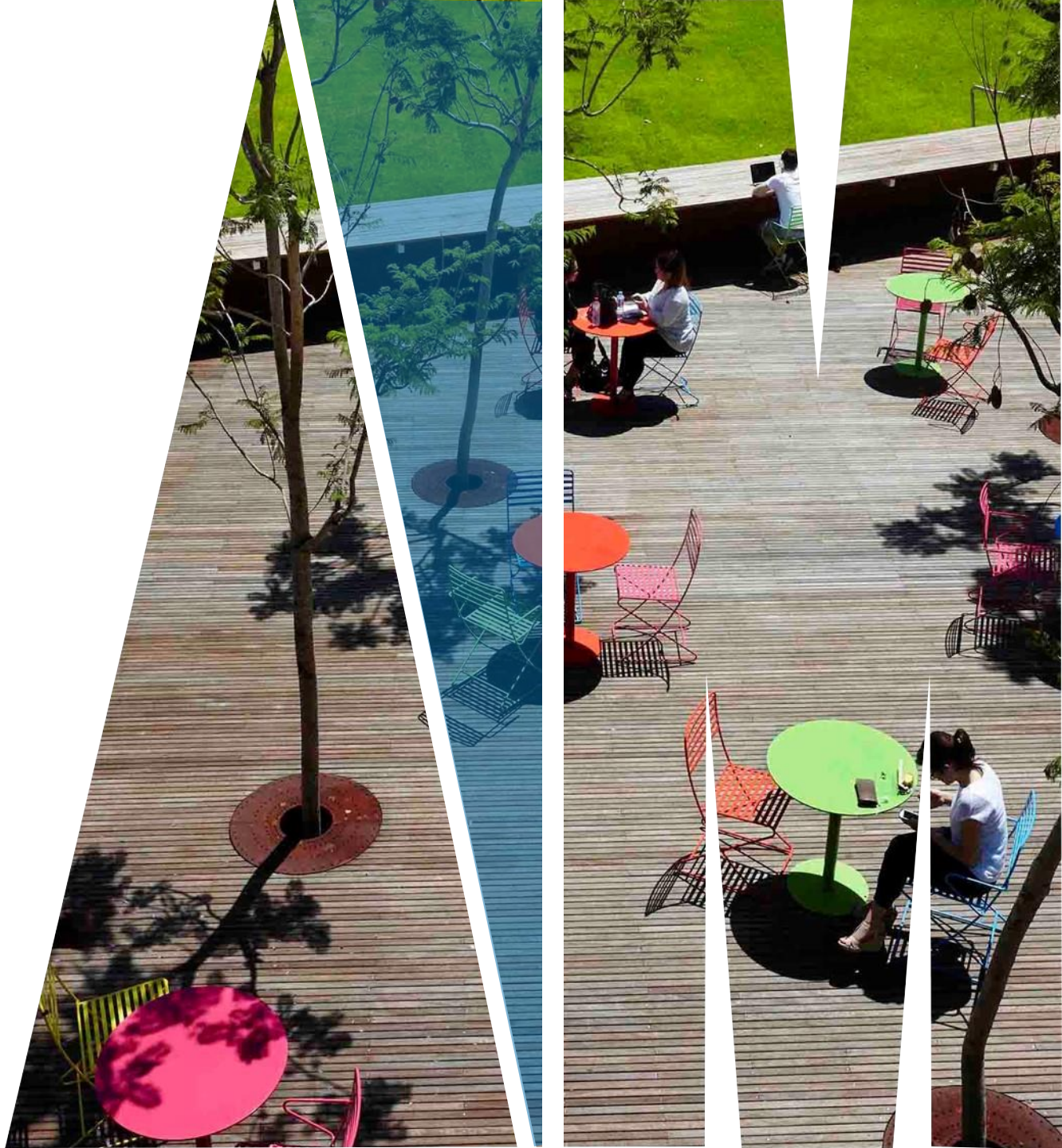
MONASH University

FIT2099 Object-Oriented Design and Implementation

Code and design smells
(Part 1: Definitions)



MONASH University

Outline

Code smells

Common problems with OO programming and design

Refactoring

THE CODE SMELLS

A code smell is a surface indication that usually corresponds to a deeper problem in the system

– Martin Fowler

A deeper problem is usually a **design problem**

By extension, “design smells” are some small bits of a design that commonly indicate a broader problem.

PRACTICAL EXAMPLE

This example is based on a roguelike using the Star Wars universe.

It’s about how to implement a kind of **Actor** that can only exist once

So… we have

`SWActor` being the base class for **Actors** in the game package;

`SWItem` instead of **`PortableItem`**, etc.

CODE SMELLS

PRACTICAL EXAMPLE

Cloning “people” is a thing in the Star Wars universe

So… let’s add the ability to clone a SWActor to the class (via an **abstract method**):

```
/**
 * Clone a SWActor.  Useful for starting your own
 * Clone War
 *
 * @return a clone of this SWActor
 */
abstract SWActor swclone();
```

Let's pretend this **is implemented and used**…and *then* we add Ben Kenobi!

HOW SHOULD BEN KENOBI IMPLEMENT swclone()?

There can only be one Ben Kenobi!

- so the class ***enforces*** this via a Singleton Pattern

```
public BenKenobi extends SWActor{
    private static BenKenobi benInstance;

    private BenKenobi(){} // can't use constructor

    public static getBenKenobi(){
            if(benInstance==null){
                benInstance=new BenKenobi();
            }
            return benInstance;
    }
}
```

Singleton Pattern

So what should calling `swclone()` on Ben do?

RECALLING THE
LISKOV SUBSTITUTION PRINCIPLE

Informally – if B is a subclass of A, you should be able to treat an instance of B as an A

Design By Contract version:

- preconditions can't be strengthened in a subclass
- postconditions can't be weakened in a subclass
- invariants in superclass must be preserved

You saw this in previous lectures

Barbara Liskov

THIS IS A "SMELLY" FIX

We're saying this fix is "smelly"… what do you think is wrong with it?

- hint: consider the previous slide…

```
if(a instanceof BenKenobi){
        throw Exception("you can't clone Ben Kenobi");
        }
        else{
        clone=a.swclone();
        }
```

WHY IS IT "SMELLY"?

Firstly, it is an obvious **breach of the LSP**

- I hope everybody can see that!
- subclass does not honour the superclass's contract

Also, it introduces a deep, ugly **dependency** between BenKenobi and every other class that uses a SWActor

It makes the system **hard to be extended**. Consider what would happen if we added other unique characters such as Leia Organa and Han Solo

- that if statement might need to get very, very big…

In general, branching on type information is a code smell

CANDIDATE FIXES?

Could introduce an **interface** `SWCloneable` and branch on that instead

- still branching on type information ☹
- we only have to do one check, even if we add Luke and Leia and Han ☺

Change the specs for `swclone()`, such that:

- it returns null if you attempt to clone an actor that isn't `SWCloneable`, or
- it throws an exception (perhaps we define a `NonCloneableActorException`) if you attempt to clone an actor that isn't `SWCloneable`

A companion `isCloneable()` **method** that returns a boolean if an actor is `SWCloneable` might be an idea

Really, what we need to do is ***change those preconditions*** so that non-cloneable Actors no longer violate them

WHICH IS THE RIGHT ANSWER?

All of these are potentially viable solutions

None of them is perfect

Which is best?

- as we say so often in FIT2099… ***it depends***
- we would need more context information to determine

IMPLEMENTING THE FIX

If `swclone()` is new and you have the discussion, easy ☺

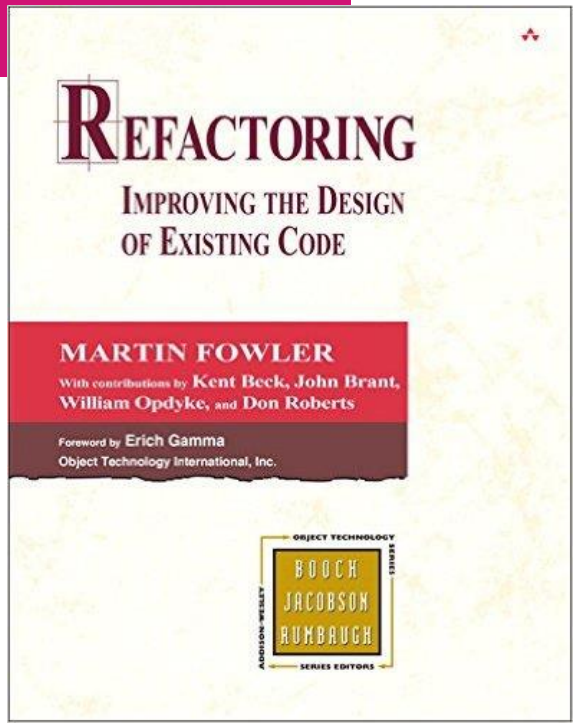
If `swclone()` is already *used* in a bunch of places…

- the **public interface** will be changed, not just implementation
- we'll need to check all the places it's used
- and not just human review: we'll need to **test**

Automated unit tests make refactoring safer

WHAT WAS REFACTORING, AGAIN?

…is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.
– Martin Fowler



Refactor to improve the quality of software
Fowler presents a *technique* for refactoring
Book is available from the library
- get second edition if you can find it

WHY REFACTORING?

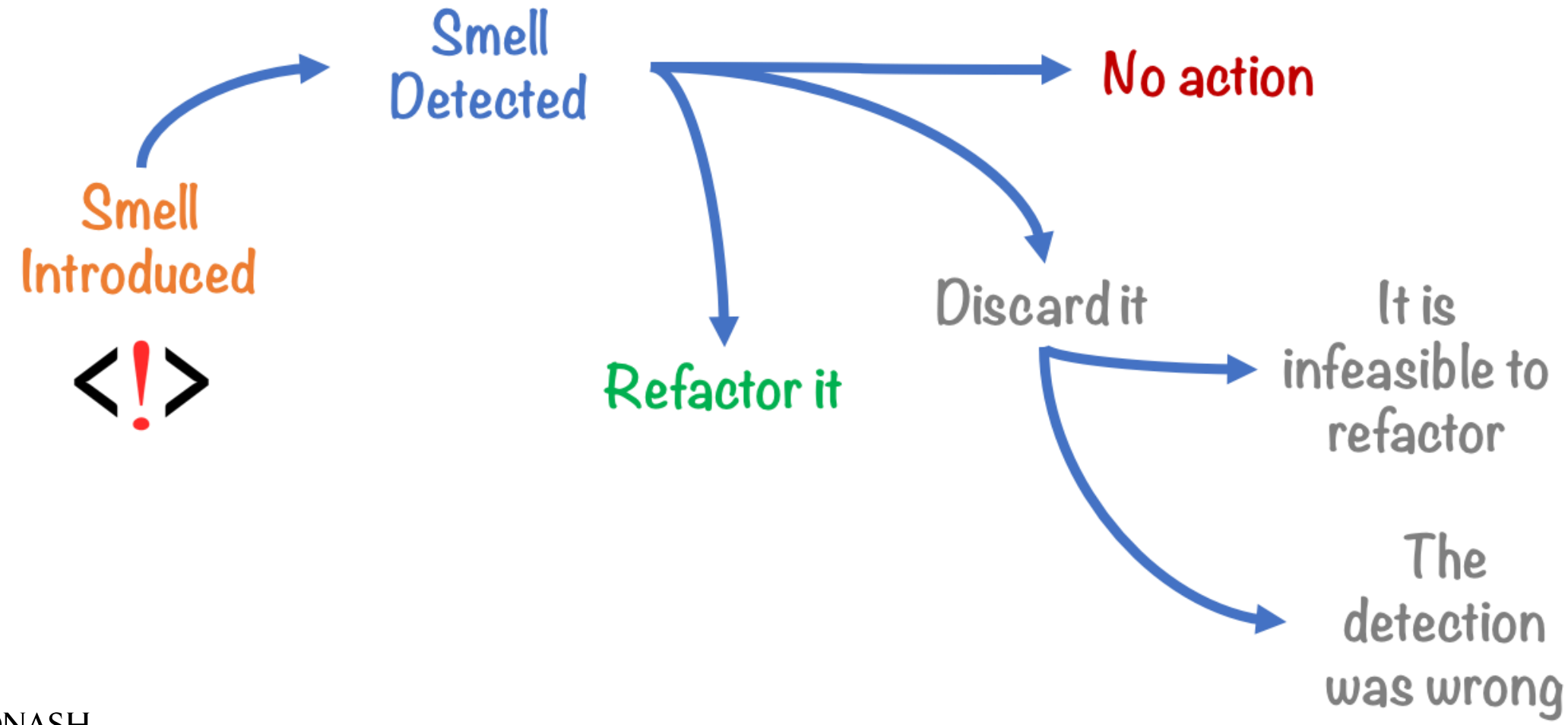
Improve design (**extensibility**)

Improve **understandability**

Makes **debugging easier**

Photo Source : Industrial Logic

WHEN TO REFACTOR?



WHEN TO REFACTOR?

Fowler suggests that teams refactor their code

- when **adding new features**
- when you need to **fix a bug**
- as you do **a code review**

You need to switch between “two hats”

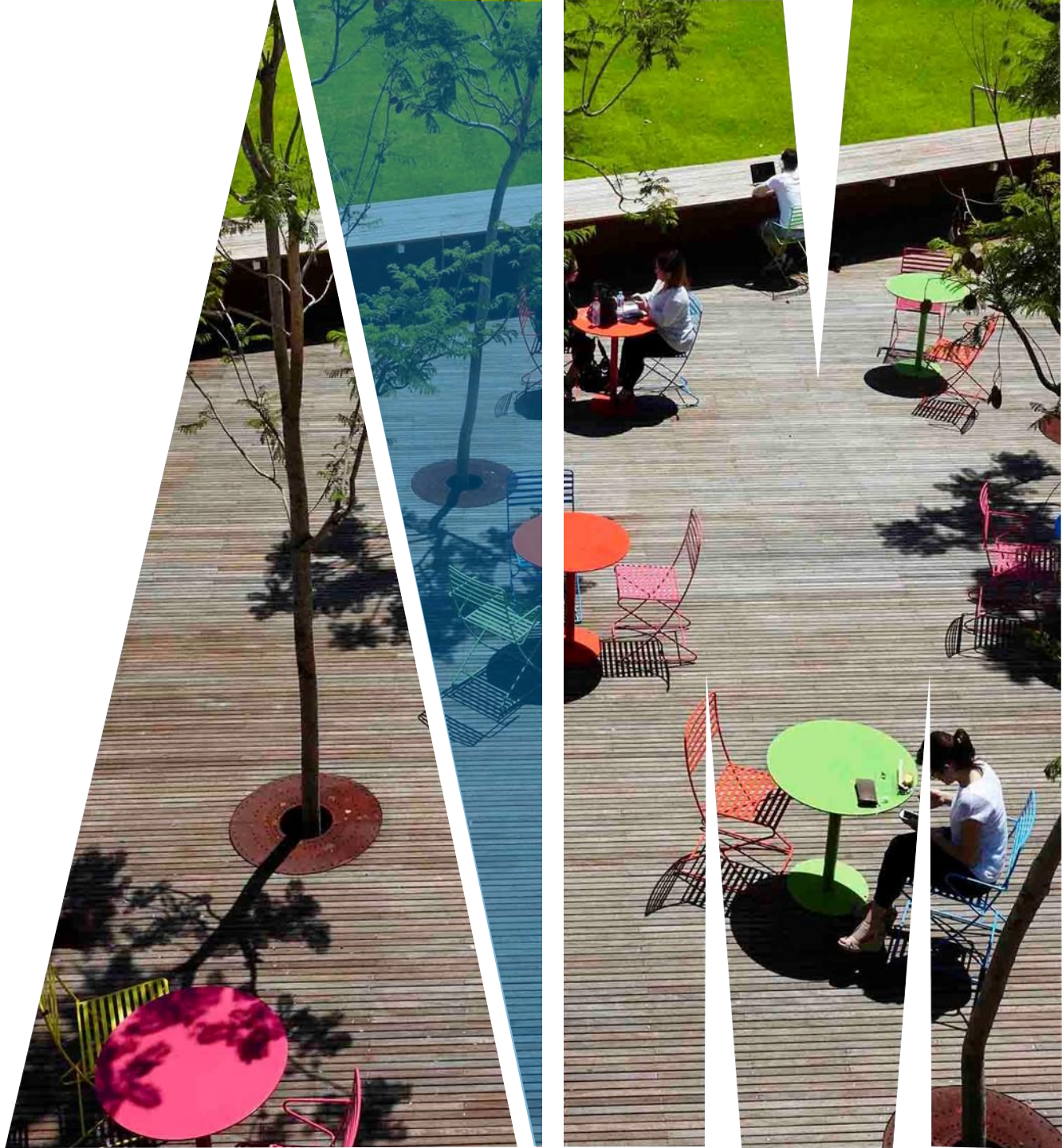
- **don't refactor and add new features at the same time!**
- if you modify the design and add new features at the same time, it can be hard to figure out whether you messed up the new features or got the refactor wrong
- the last thing we want to do is to make debugging harder…
- so **refactor first**, verify that everything’s working, **then add your new feature**

More on refactoring next week!

MONASH University

FIT2099 Object-Oriented Design and Implementation

Code and design smells
(Part 2: Kinds of smells)



MONASH University

FOWLER'S TAXONOMY OF CODE SMELLS

Fowler built a list of **common code smells**

We'll present some of them here

Note that they are

- not definitive
  - e.g. **switching on type information is not on the list** ☺
- subjective

A code smell isn't a signal that something is definitely wrong

- it is a signal that something ***might be*** wrong

CODE SMELLS AND SUGGESTED REFACTORING STEPS

Code smell	Refactoring steps
Brain Class	Extract Class and Move method
Brain Method	Extract Method
Data Clumps	Extract Class and Introduce Parameter Object
Feature Envy	Move method
God Class	Extract Class
Intensive Coupling	Move Method
Long Method	Extract method
Shotgun Surgery	Inline class
Type Checking	Replace Conditional with Polymorphism and Replace Type Code with State/Strategy

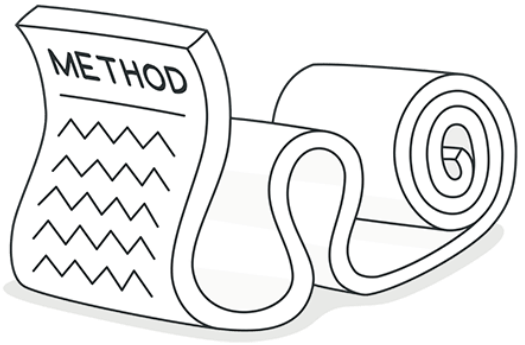
Fontana, F. A., Mangiacavalli, M., Pochiero, D., & Zanoni, M. (2015, May). On experimenting refactoring tools to remove code smells. In *Scientific Workshop Proceedings of the XP2015* (pp. 1-8).

THE CODE LENGTH SMELLS (BLOATERS)

Duplicated code

- We've been talking about this all semester: **Don't Repeat Yourself!**

THE CODE LENGTH SMELLS (BLOATERS)



Source: refactoring.guru

Long methods

– The longer a method is, the more difficult it is to understand

How to split a method?

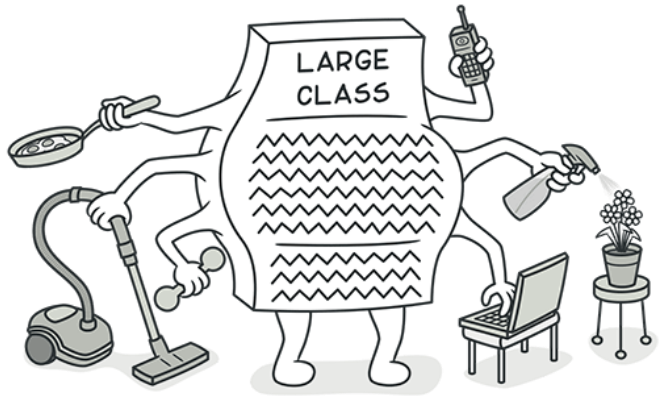
– To reduce the length of a method body, use **Extract Method**.

– If local variables and parameters interfere with extracting a method, introduce a **parameter object** or preserve whole object.

– If none of the previous methods help, the entire method can be moved to a new object using **Replace Method** with Method Object.

– **Conditional operators and loops** are a good clue that code can be moved to a separate method. For conditionals, use **Decompose Conditional**. For loops try **Extract Method**.

THE CODE LENGTH SMELLS (BLOATERS)



Source: refactoring.guru

Large classes

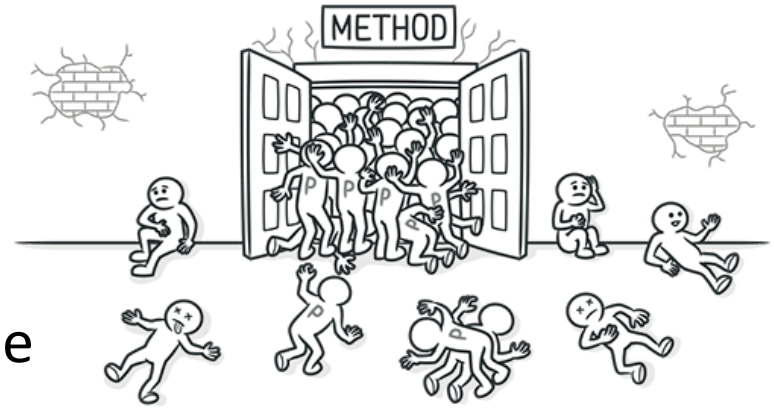
- Often **a "catch-all" class** that all the functionality that does not go anywhere else has been placed in.

Likely to violate the Single Responsibility Principle (SRP)

How to split a God class?

- **Extract Class** can help if part of the behavior of the large class can be moved into a separate class.
- **Extract Subclass** helps if part of the behavior of the large class can be implemented in different ways or is used occasionally.
- **Extract Interface** helps if it's necessary to have a list of the operations and behaviors that the client can use.

THE CODE LENGTH SMELLS (BLOATERS)



Source: refactoring.guru

Long parameter lists

– More than three arguments to a method is generally an issue

```
user = userManager.create(USER_NAME, group,
 USER_NAME, "joshua", USER_NAME, LANGUAGE,
 false, false, new Date(), "blah", new Date())
```

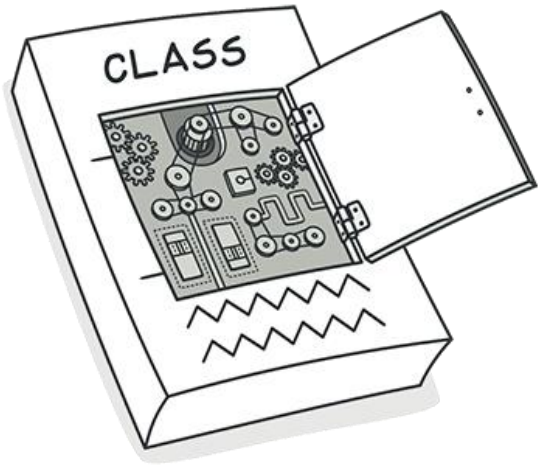
Why doesn't this method already have the data it needs in the class where it lives? Is it in the right place?

How to address this?

- **Pass the object itself** instead of passing a group of data received from another object
- Maybe **merge parameters** into a single parameter object

THE DEPENDENCY SMELLS (CHANGE PREVENTERS)

Divergent change



Source: refactoring.guru

- you have a class that you **repeatedly change**, in a couple of different ways

  e.g. you have a class for which you keep having to change three methods at the same time in one context, and in another context you change a different four methods again and again

  that is a hint that maybe there are two distinct classes there

How to address this?

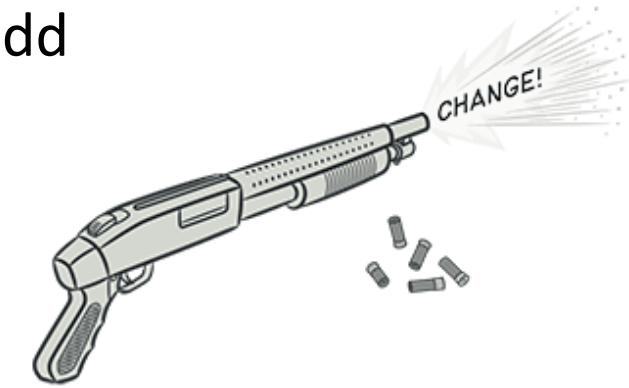
- **Extract Class** to split the class into two.
- **Inheritance**. If different classes have the same behavior, you may want to combine the classes through inheritance (**Extract Superclass** and **Extract Subclass**).

THE DEPENDENCY SMELLS (CHANGE PREVENTERS)

Shotgun surgery

shotgun surgery is the opposite – whenever you add functionality, you have to make **a lot of small changes to a lot of different classes**

- easy to miss something that should have changed
- might indicate that encapsulation choices have been poor



Source: refactoring.guru

How to address this?

- **Move method or Move Field** into a single class.
- **Remove redundant classes** (those that are left almost empty).

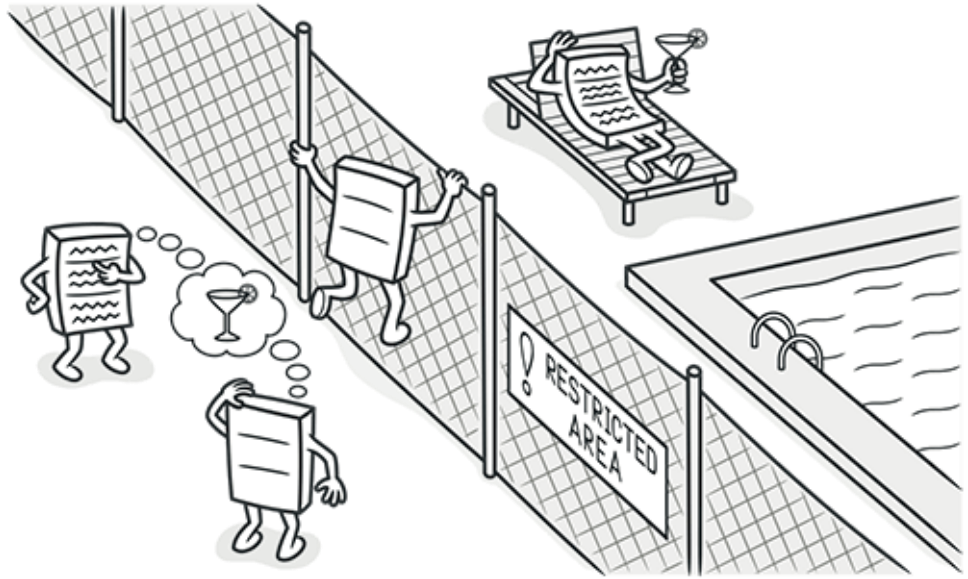
THE DEPENDENCY SMELLS (COUPLER)

Feature envy

- a method in one class spends all its effort making **multiple method calls to an object of a different class**
  - often because it does not have the data it needs in the class where it is

How to address this?

- **Move method** if the method being called should be in the calling class.
- **Extract Method** if only part of the method is required to be moved.
- **Move the data to the calling class**.

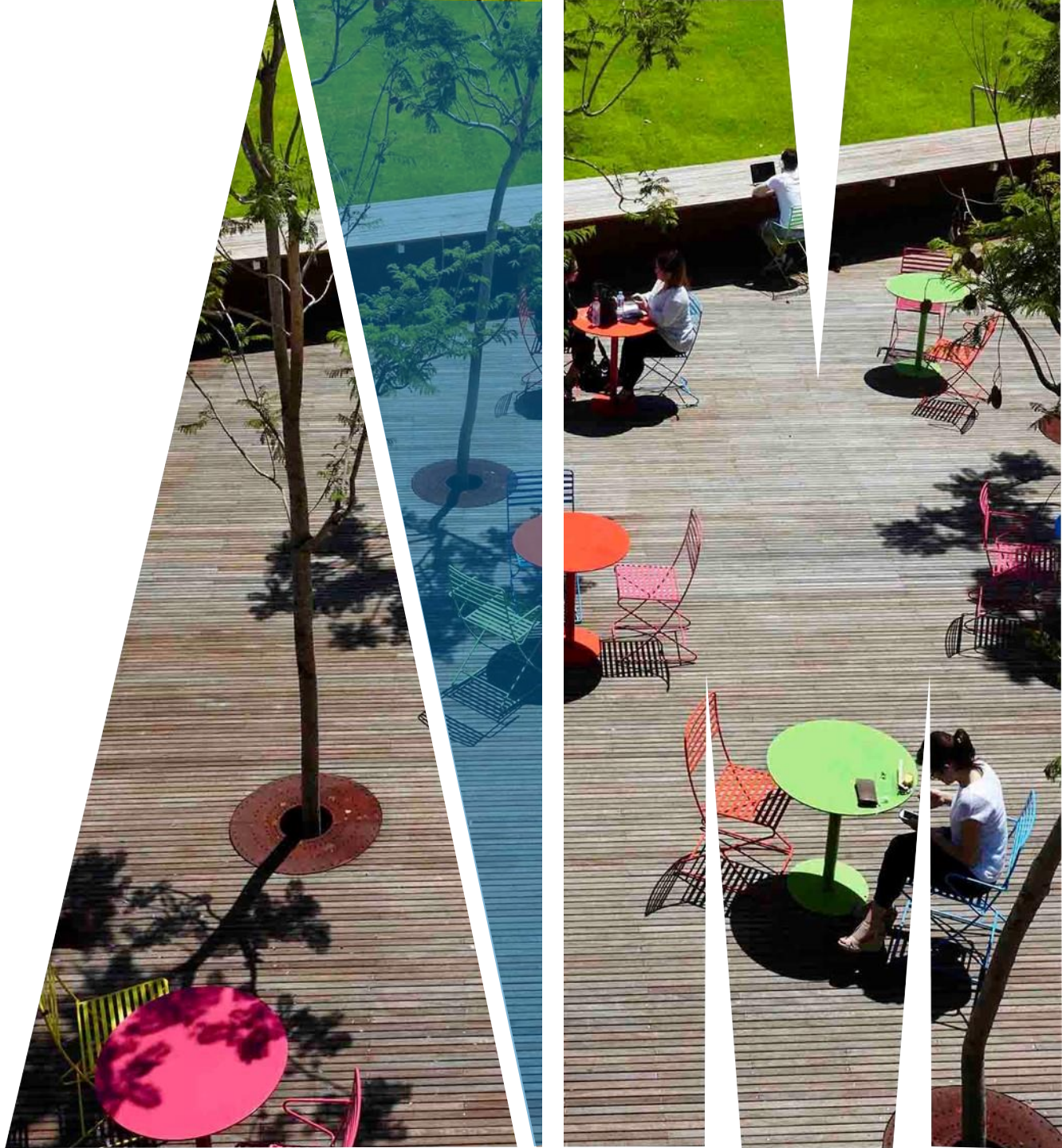


Source: refactoring.guru

MONASH University

FIT2099 Object-Oriented Design and Implementation

Code and design smells
(Part 3: Kinds of smells)



MONASH University

THE PROCEDURAL PROGRAMMING SMELLS

These are often a sign that the developer is more used to a **procedural language e.g. Python**

THE PROCEDURAL PROGRAMMING SMELLS

Primitive obsession:

storing everything **in language primitives**, rather than creating classes for data types

- storing everything in **lists of Strings** is the classic newbie Java version
- makes **validation difficult**

How to address this?

- Logically group some of the primitives into **their own class**.
- If primitives are used in method parameters maybe create **object parameters.**
- Try **replacing arrays with objects.**

THE PROCEDURAL PROGRAMMING SMELLS

Data clumps:

groups of variables that pop up repeatedly all over the code to represent the same information

- data items that always appear together **should probably be attributes of an object**

```
public void setBook(String titleBook, int titleX, int titleY, String colour){
}
public void getBook(String titleBook, int titleX, int titleY, String colour){
}
```

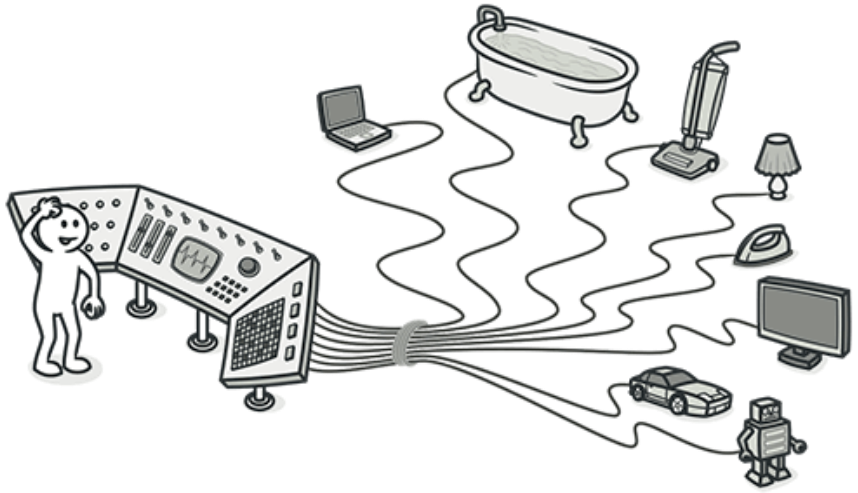
How to address this?

- If they are attributes of a class maybe **Extract them into their own class.**
- if they are used in method calls **parameter object**.

THE PROCEDURAL PROGRAMMING SMELLS

Switch statements, especially switching on type information

- often the same cascade of switch conditions **appears in multiple places**
- It can also be a **sequence of if statements**
- if you want to change or extend it, you have to find and **change them all**
- **polymorphism** is often the answer



Source: refactoring.guru

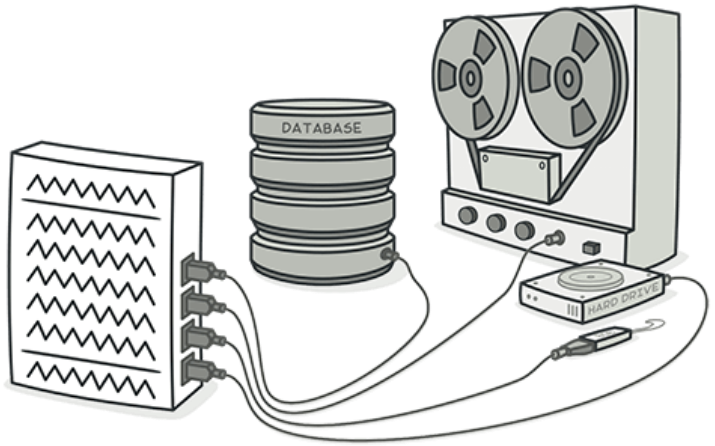
How to address this?

- **Extract the switch** into the right class
- After specifying the inheritance structure, use **Replace Conditional with Polymorphism**.
- If the sequence of ifs/switch calls to the same method, then, **the method needs to be split** into multiple smaller methods.

THE PROCEDURAL PROGRAMMING SMELLS

Data classes: classes that have **no logic in them** - they're just passive data stores. These are simply **containers** for data used by other classes. These classes don't contain any additional functionality and can't independently operate on the data that they own.

- Fowler says data classes are "**almost certainly being manipulated in far too much detail by other classes**"



Source: refactoring.guru

How to address this?

- First, **encapsulate attributes** if they are public.
- Maybe move the method or **extract the code from the method that uses the data and put it in the data class.**

YOU DO IT…NO YOU DO IT…IT…
no YOU do

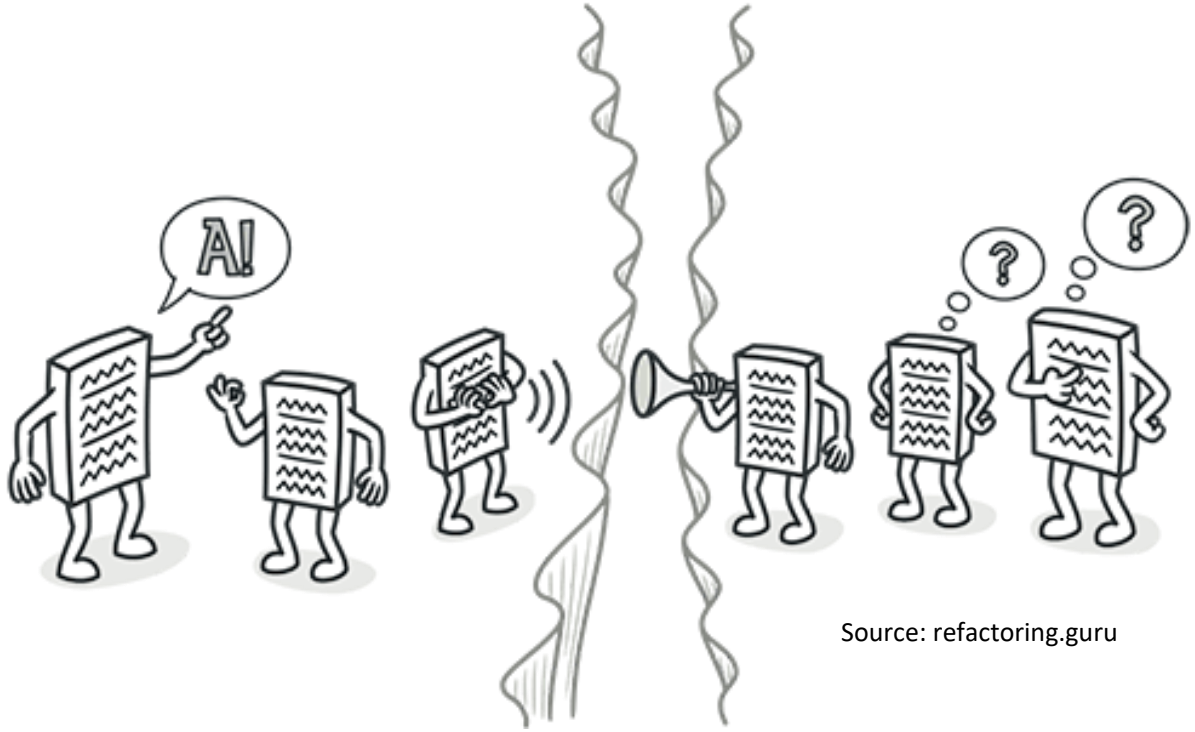
Message chains

It occurs when a client requests another object, that object requests yet another one, and so on. These chains mean that **the client is dependent on navigation along the class structure.**

How to address this?

- Reduce **delegates**
- **Move or extract methods**

to the **beginning of the chain.**



Source: refactoring.guru

YOU DO IT…NO YOU DO IT…IT…
no YOU do

Middle man

If a class performs **only one action, delegating work** to another class, why does it exist at all?

```
aFred.doThing()

class Fred {
    private Worker worker;
    public void doThing() {
        worker.doActualThing();
    }
```

Probably this class shouldn’t exist

THERE ARE
OTHER SMELLS

Some, Fowler has listed in his book

Some, he hasn't

Smells **vary a bit by programming environment**

- something might be problematic in Java but fine in C#, say

Experienced programmers know what stinks!

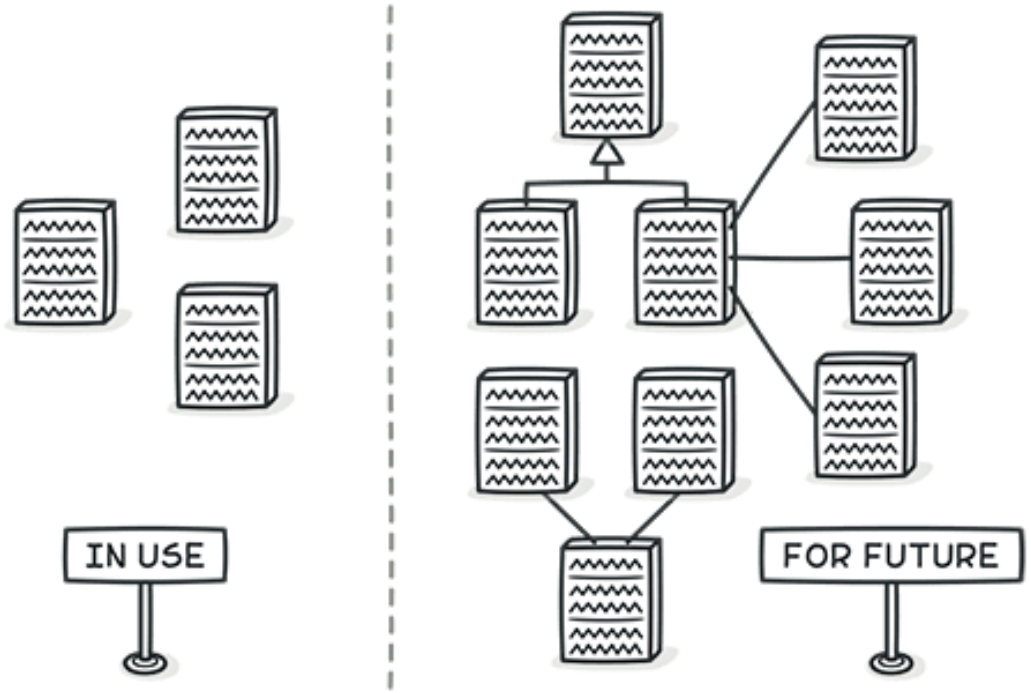
Check refactoring.guru for more examples

THE OVERENGINEERING ODORS

These are somewhat rarer for novice programmers, but are surprisingly **common in industry**.

Speculative Generality

- when you add **a bunch of machinery** to make a class extensible for every single special case on Earth (or in a Galaxy Far, Far Away), especially if it's never used



Source: refactoring.guru

THE OVERENGINEERING ODORS

These are somewhat rarer for novice programmers, but are surprisingly common in industry.

Lazy class

- a class that doesn't do much any more, often after other refactorings have been done

```
public class Letter{
    private final String content;

    public Letter(String content){
        this.content = content;
    }

    public String getContent(){
        return content;
    }
}
```

REFACTORING
METATECHNIQUE

Fowler presents a standard set of techniques for eliminating code smells

Meta-technique:

- **make small change** that eliminates or reduces a smell
- **test**
- **repeat** until the smell is not too evident

Next week: we'll go through a few, using an example from Fowler's book

Summary

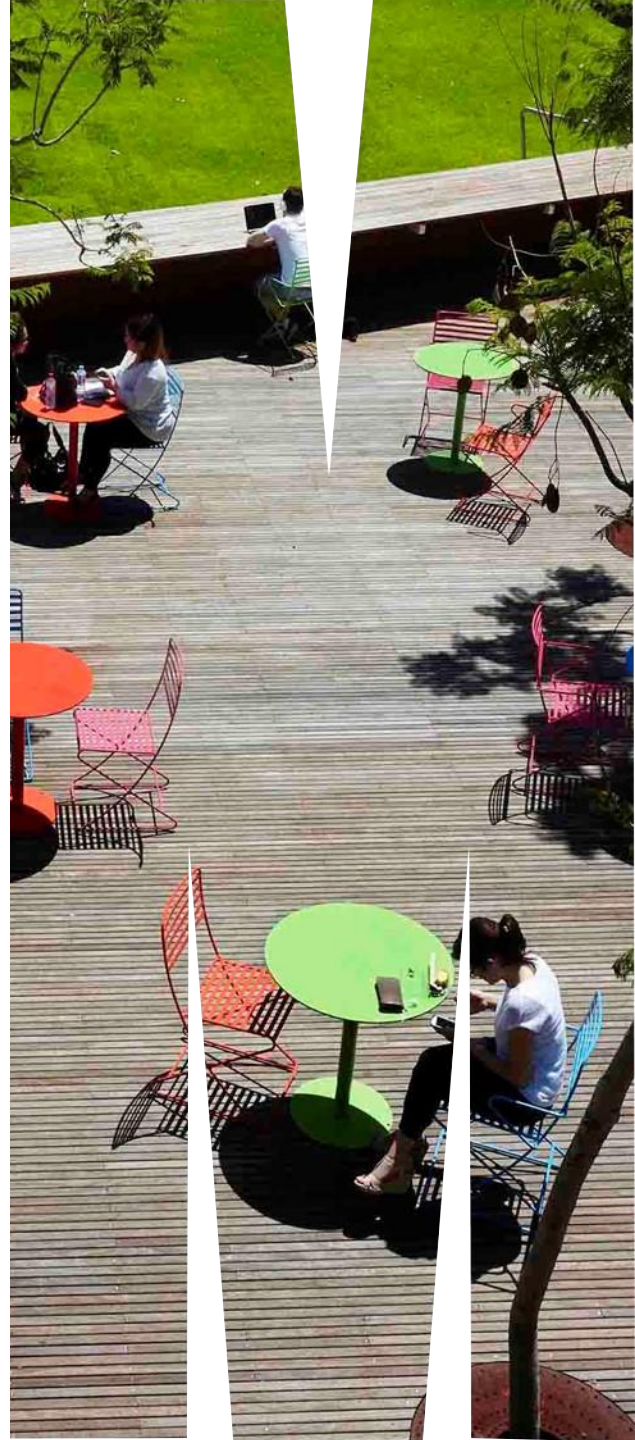
Code smells

Common problems with OO programming and design

Refactoring

MONASH University

Thanks



MONASH University