School of Computing
FACULTY OF ENGINEERING & PHYSICAL SCIENCES

UNIVERSITY OF LEEDS

COMP5911M
Advanced Software Engineering

6: Fundamentals of Refactoring

Nick Efford

https://comp5911m.info

Previously…

- We introduced refactoring via a small case study
- We saw that design could be improved by a series of **small, well-defined, carefully controlled steps**
- We saw that **unit tests are essential in supporting this process**; without them, we won’t be able to tell whether we are breaking things!

REMEMBER: you need to continue refactoring this code yourself – see Exercise 3

Objectives

UNIVERSITY OF LEEDS

- To go deeper into why, how and when we refactor code
- To consider the limitations of refactoring, and identify situations in which we don't refactor

Reminder

Refactoring (noun): A change made to the internal structure of software to make it easier to understand and cheaper to modify, without changing its observable behaviour.

Refactor (verb): To restructure software via a series of refactorings, without changing its observable behaviour.

Structural Decay

- As changes are made over time to realize short-term goals, code loses its structure
- System accumulates **technical debt** – which, if not addressed, can eventually make it impossible to modify
- ‘Paying off’ the technical debt is much easier if you do so by ‘small installments’ – i.e., frequent small changes
- Refactoring is a rigorous way of doing this!

Improved Understanding

- Code is written once but read and modified *many* times
- So the cost of being hard to understand can be high
    - For new developers joining your team
    - For you, returning to your code after a long time
- Refactoring **embeds your understanding into the code**, for the benefit of others / yourself in future

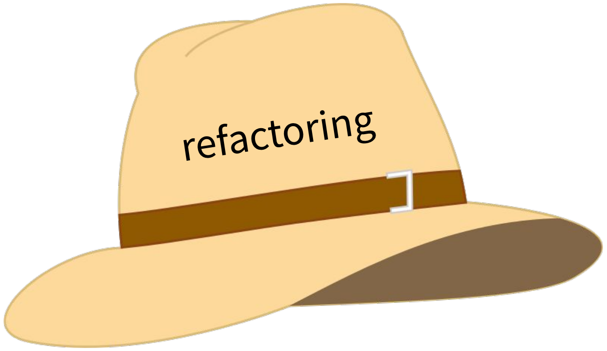
‘Two Hats’ Metaphor

Adding features

- Generally don’t change existing code
- Measure progress by creating tests and getting them to pass

Refactoring

- Change existing code and don’t add any new features
- Generally don’t add tests, and only change them when needed



Refactorings Covered Elsewhere

- Extract Method
- Move Method
- Inline Temp
- Replace Temp with Query
- Replace Conditional with Polymorphism

See Lecture 5 &
Exercise 3

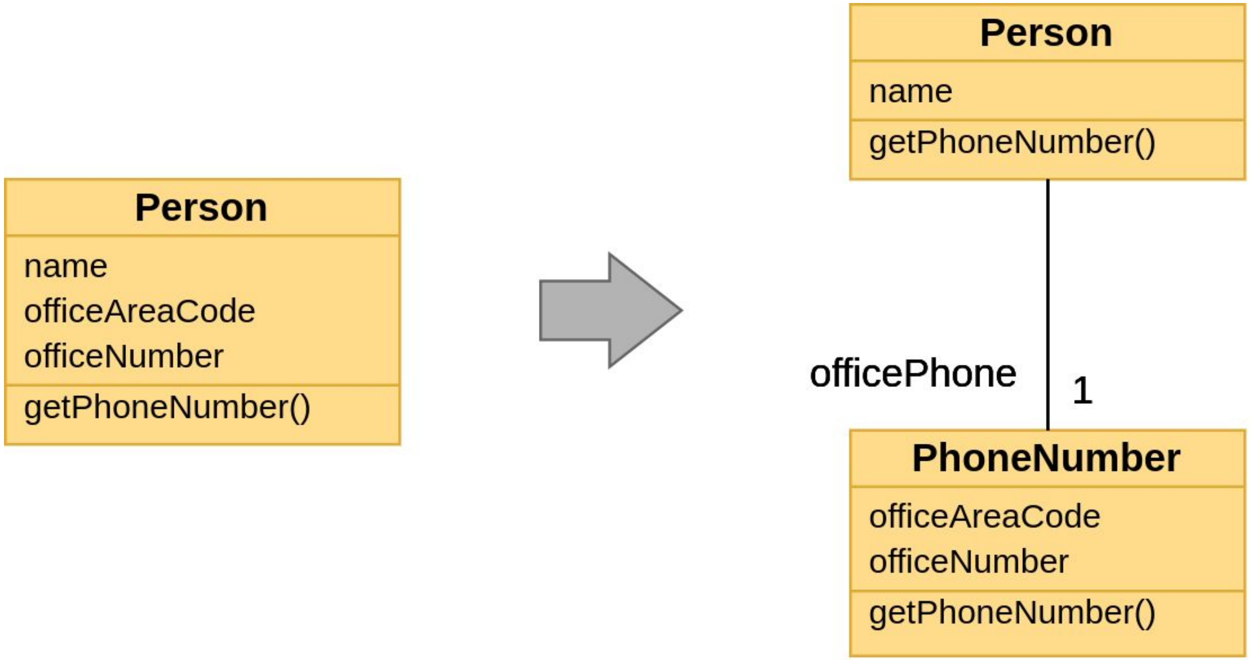
Refactoring Categories

- Composing methods
- Moving features between objects
- Organising data
- Simplifying conditional logic
- Simplifying method calls
- Manipulating inheritance hierarchies

Example: Extract Class

You have one class doing work that should be done by two.

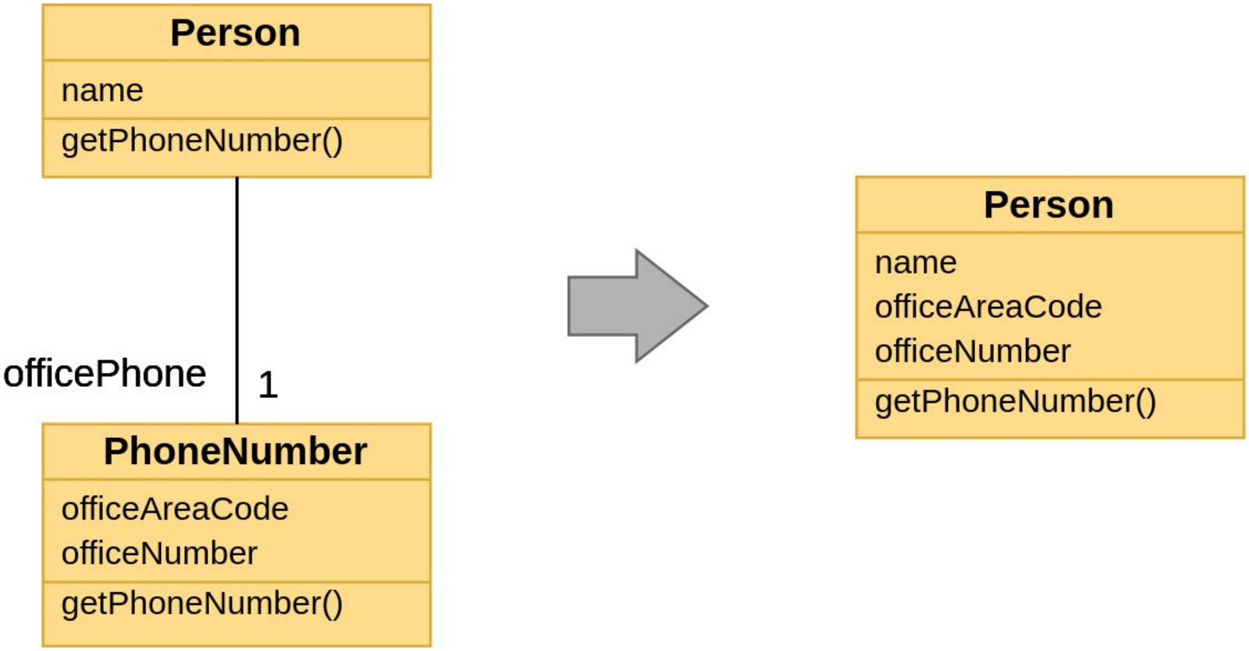
Create a new class and move the relevant fields and/or methods from the old class to the new class.



Example: Inline Class

A class isn’t doing very much.

Move all its features to another class, then delete it.



Person

name

getPhoneNumber()

officePhone

1

PhoneNumber

officeAreaCode
officeNumber

getPhoneNumber()

Person

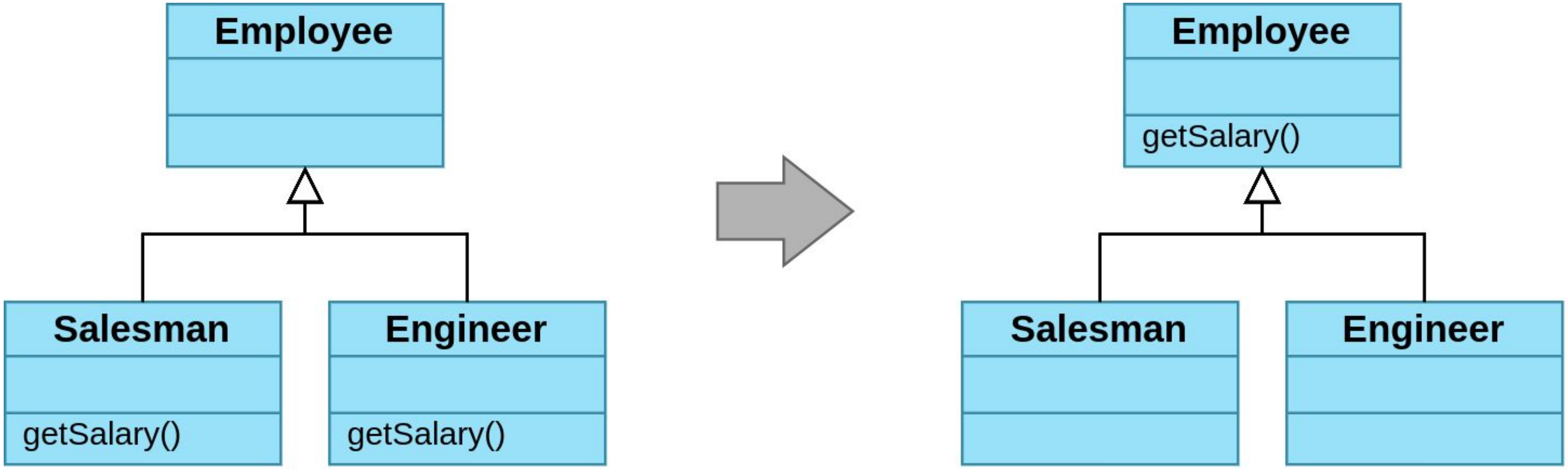
name
officeAreaCode
officeNumber

getPhoneNumber()

Example: Pull Up Method

You have methods with identical results on subclasses.

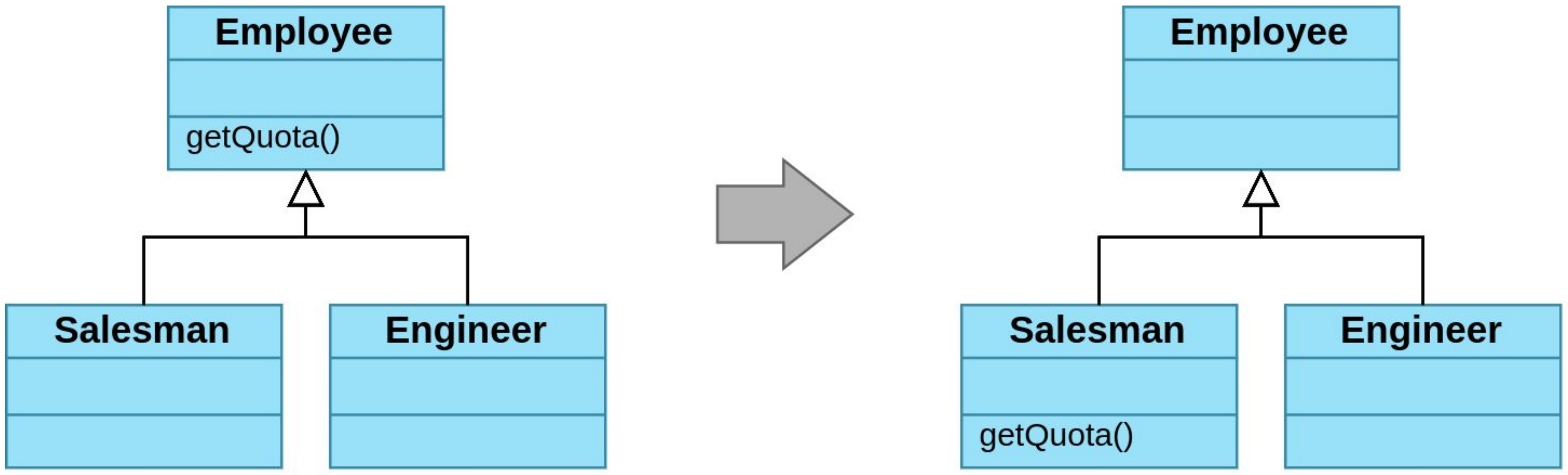
Move them to the superclass.



Example: Push Down Method

Behaviour on a superclass is relevant only for some of its subclasses.

Move it to those subclasses.



Example: Decompose Conditional

UNIVERSITY OF LEEDS

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
}
else {
  charge = quantity * summerRate;
}
```

```
if (isSummer(date)) {
  charge = summerCharge(quantity);
}
else {
  charge = winterCharge(quantity);
}
```

When Do We Refactor?

- Before adding a feature
- As we do code review
- When there’s a ‘bad smell’ to the code

‘Code Smells’

- Originally the idea of **Kent Beck** (JUnit & XP)
- The sense that something ‘smells bad’ in the code – i.e., the design/implementation isn’t as good as it could be
- Based primarily on intuition, experience and personal judgement rather than anything scientific
- Fowler’s book lists 22 of them…

Typical Code Smells

- Duplicate Code
- Long Method
- Long Parameter List
- Temporary Field
- Switch Statement
- Excessive Comments

See Fowler Chapter 3 or Refactoring Guru for more examples

Link to Refactorings

Each code smell suggests one or more possible refactorings that can remove the problem:

- Duplicate code ⟶ **Extract Method** if code is in the methods of one class, **Extract Class** or **Extract Superclass** if code is in different classes
- Excessive Comments ⟶ **Extract Variable** if comment explains a complex expression, **Extract Method** if it explains a block of code, **Rename Method** if it explains a method
- Long Parameter List ⟶ **Preserve Whole Object** if params come from same object, or **Introduce Parameter Object** if not
- Temporary Field ⟶ **Extract Class**

Task

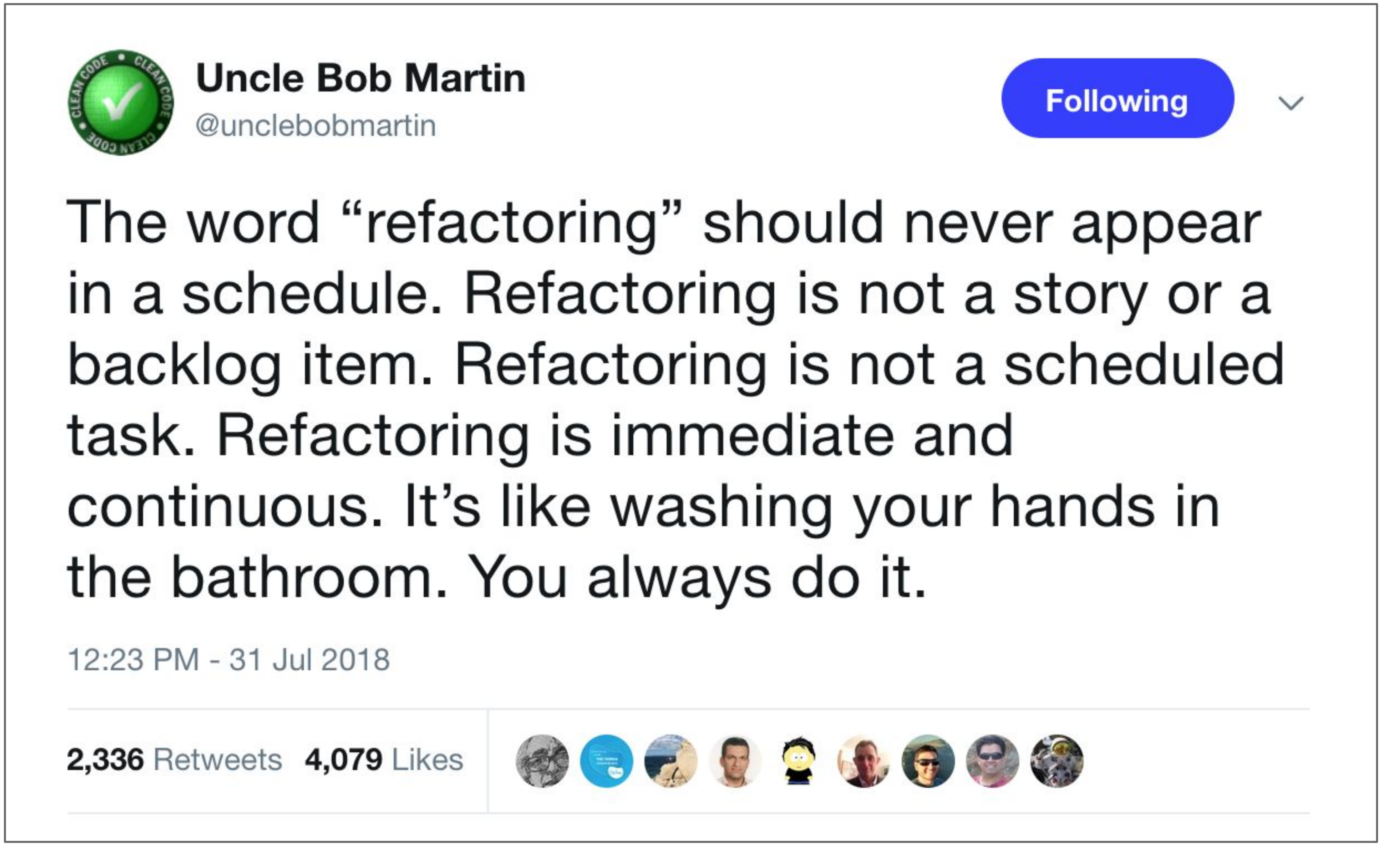
1. Get a copy of the 'Smells to Refactorings Cheatsheet' (physical handout, or Download from Minerva)
2. Study the `step1` code from Lecture 5, then use the Cheatsheet and refactoring.guru code smells page to identify two smells affecting the code
3. Study the `getCharge()` method in the `Rental` class in the `step3` code. Which refactoring technique from the 'Organizing Data' section on refactoring.guru can we use to improve the code?

getCharge()

UNIVERSITY OF LEEDS

When Do We Refactor?

UNIVERSITY OF LEEDS



Uncle Bob Martin
@unclebobmartin

The word “refactoring” should never appear in a schedule. Refactoring is not a story or a backlog item. Refactoring is not a scheduled task. Refactoring is immediate and continuous. It’s like washing your hands in the bathroom. You always do it.

12:23 PM - 31 Jul 2018

2,336 Retweets 4,079 Likes

Issues With Databases

- Business applications are often tightly coupled to the database schema
- … so refactoring the code can force schema changes
- … which then forces time-consuming **data migration**, if the database is already live
- Solution: insert a layer of software between the app's object model and database model
    - Refactoring may force changes to the intermediate layer, but schema can remain unchanged

API Changes

- Public methods of your classes define an API that can be depended on by other code
- If refactoring changes the API, that code will break!
- Solution: **retain the old API**, but have its methods delegate to the new API methods

```
/**
 * Method that does something.
 *
 * @deprecated Use {@link #newMethod()} instead.
 */
@Deprecated
public void oldMethod() {
  newMethod();
}
```

Performance Impact

- Refactoring such as **Inline Temp** can lead to code being called multiple times to compute a value
- Code clarity benefits can outweigh any performance loss
- Recommended strategy:
  - Refactor to improve clarity & structure
  - Profile code to find the poorly performing parts
  - Focus optimization efforts only on those parts – which should now be easier thanks to refactoring!

When Do We NOT Refactor?

- When structure is so bad that it would be easier to throw the code away and start again
- When the code doesn’t function correctly
- When a hard deadline is very close

Summary

We have

- Considered why we refactor, highlighting the goals of preventing structural decay and improving understanding
- Noted that developers constantly switch between ‘adding features’ and ‘refactoring’ mindsets (‘two hats’)
- Discussed some other common refactorings
- Introduced the idea of ‘code smells’ as a signal that we need to refactor the code
- Explored the limitations of refactoring

Follow-Up / Further Reading

- Chapters 2 and 3 of Fowler’s *Refactoring: Improving The Design of Existing Code* (available in EBL)
- Fowler’s articles on refactoring
- Catalog of code smells
- Catalog of refactoring techniques
- Exercise 3 and Exercise 4