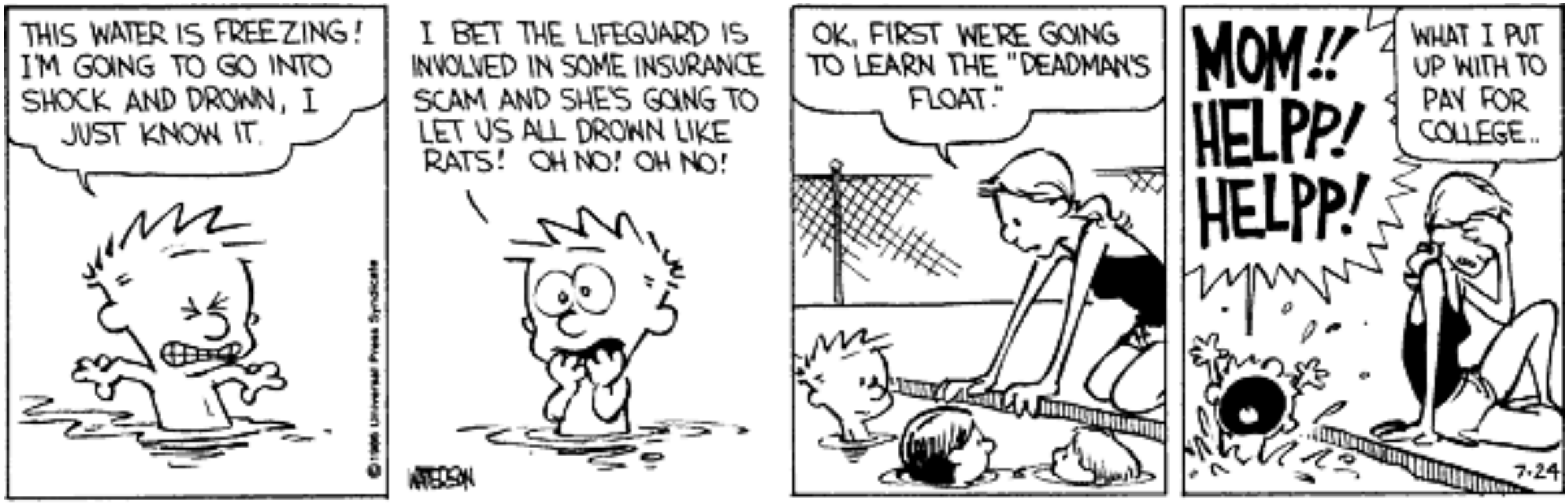


https://calvinandhobbes.fandom.com/index.php?title=File:Deadman%27s_float.gif&limit=100&showall=0

Refactoring

AARHUS UNIVERSITY
AARHUS UNIVERSITY SCHOOL OF ENGINEERING

MICHAEL LOFT
ML@ASE.AU.DK

Factorization

In mathematics, factorization consists of writing a number or another mathematical object as a product of several factors, usually smaller or simpler objects of the same kind.

For example, 3×5 is a factorization of the integer 15

and $(x - 2)(x + 2)$ is a factorization of the polynomial $x^2 - 4$.

https://en.wikipedia.org/wiki/Factorization

Refactoring definition

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 behavior.*

Refactor *(verb): to restructure software by applying a series of refactorings without changing its observable behavior.*

We want to

Improve code quality

- Maintainability
- Understandability
- Simplicity
- Extendability
- Testability

Want code to adhere to SOLID
but
Without changing externally visible behavior

When to refactor

Before starting a new feature – in order to make it easier to implement the feature

After all you tests passes – when you can trust that you do not change the visible behavior

Code smells

The Bloaters:

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- DataClumps

The Object-Orientation Abusers:

- Switch Statements
- Temporary Field
- Refused Bequest
- Alternative Classes with Different Interfaces

The Change Preventers:

- Divergent Change
- Shotgun Surgery
- Parallel Inheritance Hierarchies

The Dispensables:

- Lazy class
- Data class
- Duplicate Code
- Dead Code
- Speculative Generality

The Couplers:

- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Middle Man

A Taxonomy for "Bad Code Smells" https://mmantyla.github.io/BadCodeSmellsTaxonomy.html

Before you start

1. Make sure you have an extensive test suite of the code you are about to change.

2. Take small steps.

3. Locate a smell

4. Apply a refactoring and verify that all test still pass.

5. **Use** source control

Extract method

```
void printOwing(double amount)
{
  printBanner();

  //print details
  System.out.println ("name:" + _name);
  System.out.println ("amount" + amount);
}
```

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

From: Martin Fowler – Refactoring: Improving the Design of Existing Code

Extract method

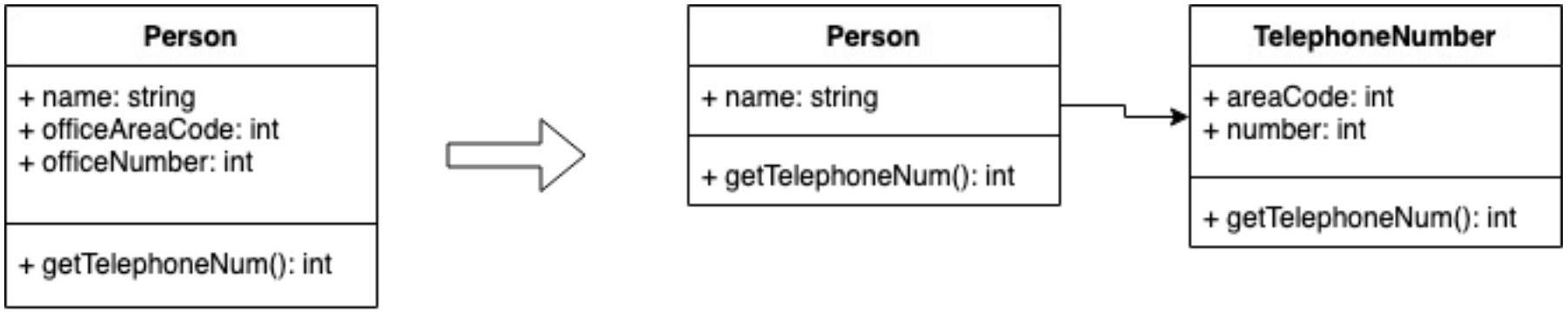
```
void printOwing(double amount)
{
  printBanner();
  printDetails(amount);
}

void printDetails (double amount)
{
  System.out.println ("name:" + _name);
  System.out.println ("amount" + amount);
}
```

You have a code fragment that can be grouped together.

Turn the fragment into a method whose name explains the purpose of the method.

From: Martin Fowler – Refactoring: Improving the Design of Existing Code

Extract class



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

Create a new class and move the relevant fields and methods from the old class into the new class.

From: Martin Fowler – Refactoring: Improving the Design of Existing Code

Replace conditional with polymorphism

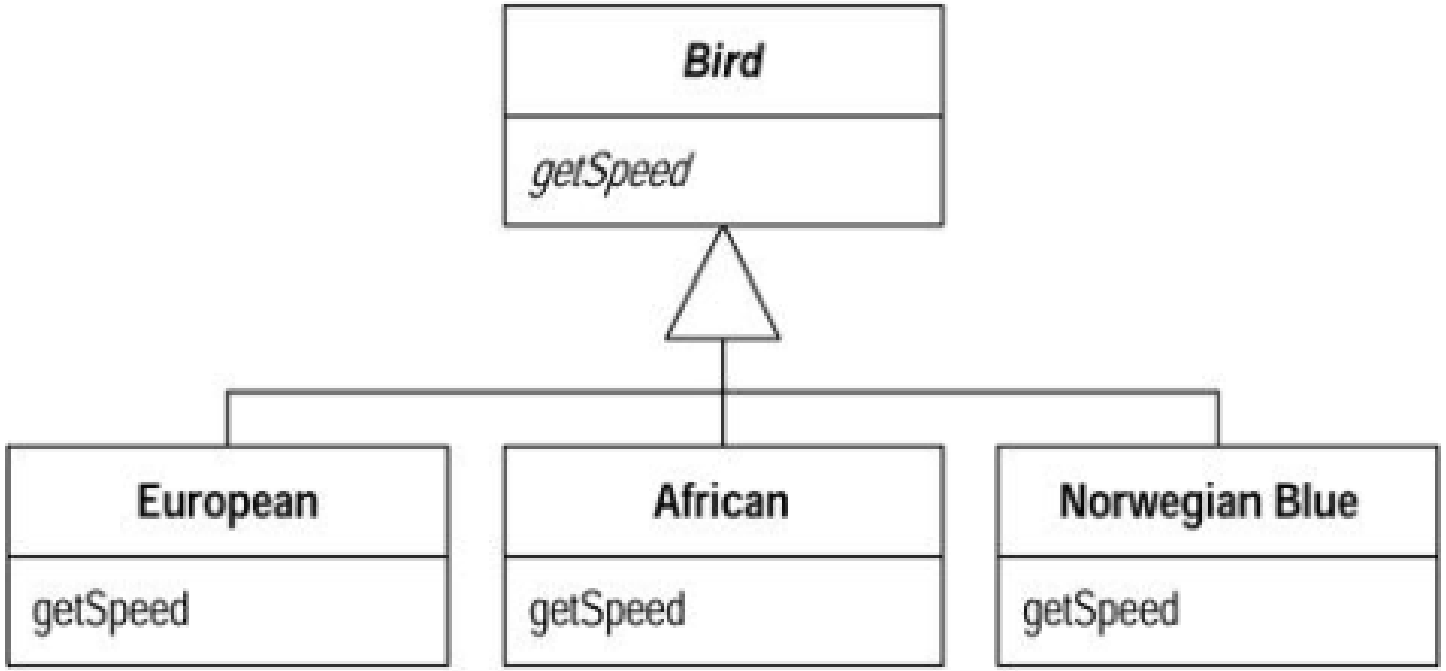
```
double getSpeed()
{
  switch (_type) {
   case EUROPEAN:
     return getBaseSpeed();
   case AFRICAN:
     return getBaseSpeed() –
           getLoadFactor() * _numberOfCoconuts;
   case NORWEGIAN_BLUE:
     return (_isNailed) ? 0 :
           getBaseSpeed(_voltage);
  }
  throw new RuntimeException("Should be unreachable");
}
```

You have a conditional that chooses different behavior depending on the type of an object.

Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

From: Martin Fowler – Refactoring: Improving the Design of Existing Code

Replace conditional with pol(l)ymorphism



From: Martin Fowler – Refactoring: Improving the Design of Existing Code

Many other refactorings

Note that there is both a refactoring for ”Change value to reference” and ”Change reference to value”.

The same for unidirectional and bidirectional associations.

It is up to YOU to decide, what improves the code.

From: Martin Fowler – Refactoring: Improving the Design of Existing Code

Code smells to refacotings

Smell	Refactoring
Alternative Classes with Different Interfaces: occurs when the interfaces of two classes are different and yet the classes are quite similar. If you can find the similarities between the two classes, you can often refactor the classes to make them share a common interface [F 85, K 43]	Unify Interfaces with Adapter [K 247]
	Rename Method [F 273]
	Move Method [F 142]
Combinatorial Explosion: A subtle form of duplication, this smell exists when numerous pieces of code do the same thing using different combinations of data or behavior. [K 45]	Replace Implicit Language with Interpreter [K 269]
Comments (a.k.a. Deodorant): When you feel like writing a comment, first try "to refactor so that the comment becomes superfluous" [F 87]	Rename Method [F 273]
	Extract Method [F 110]
	Introduce Assertion [F 267]
Conditional Complexity: Conditional logic is innocent in its infancy, when it's simple to understand and contained within a few lines of code. Unfortunately, it rarely ages well. You implement several new features and suddenly your conditional logic becomes complicated and expansive. [K 41]	Introduce Null Object [F 260, K 301]
	Move Embellishment to Decorator [K 144]
	Replace Conditional Logic with Strategy [K 129]
	Replace State-Altering Conditionals with State [K 166]
Data Class: Classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes. [F 86]	Move Method [F 142]
	Encapsulate Field [F 206]
	Encapsulate Collection [F 208]
Data Clumps: Bunches of data that that hang around together really ought to be made into their own object. A good test is to consider deleting one of the data values: if you did this, would the others make any sense? If they don't, it's a sure sign that you have an object that's dying to be born. [F 81]	Extract Class [F 149]
	Preserve Whole Object [F 288]
	Introduce Parameter Object [F 295]
Divergent Change: Occurs when one class is commonly changed in different ways for different reasons. Separating these divergent responsibilities decreases the chance that one change could affect another and lower maintenance costs. [F 79]	Extract Class [F 149]
Duplicated Code: Duplicated code is the most pervasive and pungent smell in software. It tends to be either explicit or subtle. Explicit duplication exists in identical code, while subtle duplication exists in structures or processing steps that are outwardly different, yet	Chain Constructors [K 340]
	Extract Composite [K 214]
	Extract Method [F 110]
	Extract Class [F 149]
	Form Template Method [F 345, K 205]
	Introduce Null Object [F 260, K 301]
	Introduce Polymorphic Creation with Factory Method [K 88]

https://www.industriallogic.com/wp-content/uploads/2005/09/smellstorefactorings.pdf

Move Field

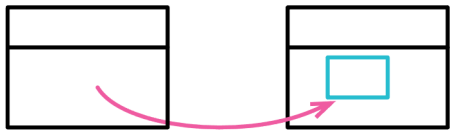
1. Motivation

Why do we want to move a field

2. Mechanics

Ensure field is encapsulated
Test
Create field and accessors in target
Run static check (compiler and/or lint)
Make reference from source to target object
Adjust accessor to use target field
Test

3. Examples



```
class Customer
{
    ...
    public Plan GetPlan() { return this.plan; }
    public float GetDiscountRate() {
       return this.discountRate;
    }
}
```

```
class Customer
{
    ...
    public Plan GetPlan() { return this.plan; }
    public float GetDiscountRate() {
       return this.plan.GetDiscountRate();
    }
}
```

From: https://refactoring.com/catalog/moveField.html

Your turn

Begin the exercises

Questions to discuss afterwards

How did it feel to work with such fast, comprehensive tests?

Did you make mistakes while refactoring that were caught by the tests?

If you used a tool to record your test runs, review it. Could you have taken smaller steps? Made fewer refactoring mistakes?

Did you ever make any refactoring mistakes and then back out your changes? How did it feel to throw away code?

Questions to discuss afterwards

What would you say to your colleague if they had written this code?

What would you say to your boss about the value of this refactoring work?

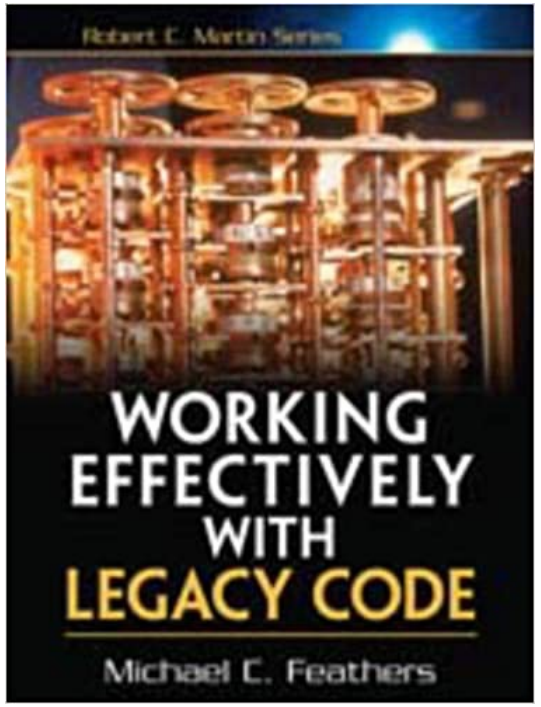
Was there more reason to do it over and above the extra billable hour or so?

What is next

- Practice practice practices
  - Try to identify code-smells
  - Use cheatsheet to find solutions

- Practice by doing katas
  - https://kata-log.rocks/refactoring
  - https://www.google.com/search?q=refactoring+kata

When there are no test suite



Refactoring without test

Only use built in methods in IDE
Insert ways to create tests case

Practising

Guilded rose
Google again ☺

AARHUS UNIVERSITY

References and image sources

Norwegian blue: https://4.bp.blogspot.com/-k4WDh052w10/UW5AJJXPefI/AAAAAAAAAA-A/EdeD0ryak6Q/s640/Norwegian+Blue+Parrot.jpg

Computer keyboard: http://stockmedia.cc/computing_technology/slides/DSD_8790.jpg