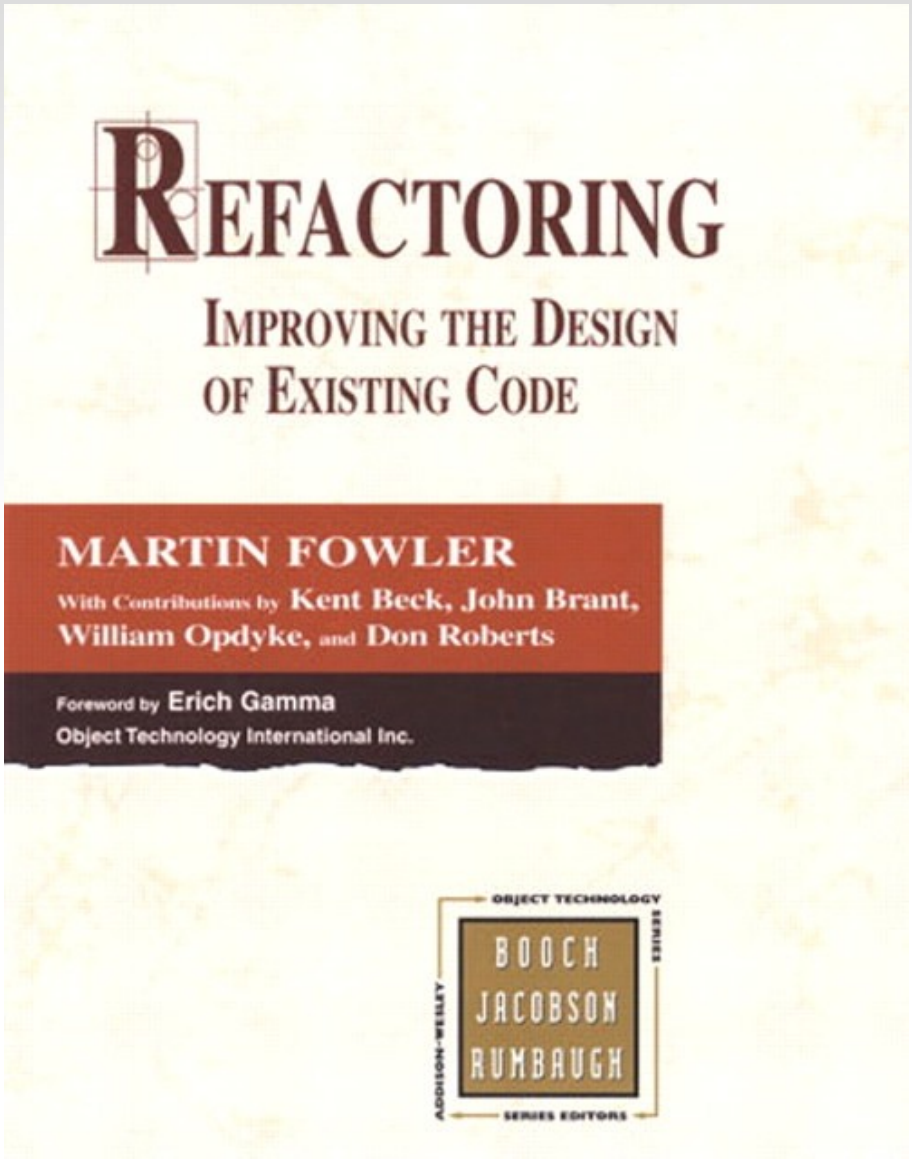
PA103 - Object-oriented Methods for Design of Information Systems

Refactoring and Code Smells

© Radek Ošlejšek
Faculty of Informatics
oslejsek@fi.muni.cz

Literature



- Refactoring – Improving the Design of Existing Code
  - Authors: M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts
  - Publisher: Addison-Wesley Professional
  - Copyright: 1999 (Kindle Edition 2012)
- www.refactoring.com

Refactoring

- 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.

Refactoring Principles

- Why do we refactor?
  - To improve the design of software
  - To make software easier to understand
  - To help you find bugs
  - To make your program faster

- When should we refactor?
  - Refactor when you add functionality
  - Refactor when you need to fix a bug
  - Refactor as you do code reviews
  - Refactoring is very common in iterative/incremental development (integration of a new code usually require refactoring of existing parts of the system)
  - *Refactor when the code starts to smell.*
  - *Don't refactor if a "stench" is the feature of applied design pattern!*

- What about performance?
  - Worry about performance only when you have identified a performance problem

Refactoring and Development Process

- Traditional software engineering is modeled after traditional engineering practices (= design first, then code)
  - The desired end product can be determined in advance
  - Workers of a given type are interchangeable
- Agile software engineering is based on different assumptions:
  - Requirements (and therefore design) change as users become acquainted with the software
  - Programmers are professionals with varying skills and knowledge
  - Programmers are in the best position for making design decisions
  - **Refactoring is fundamental to agile programming**
  - Refactoring is often necessary even in a traditional **iterative/incremental development**
    - Code modification during the iterative process
    - Integration of new increments
- Refactoring environment and tools:
  - Continuous integration, Change Management, ...

Lecture 4 / Part 1:

Bad Smells In Code
(Examples in Detail)

Bad Smells in Code

If it stinks, change it.

--Grandma Beck, discussing child-rearing philosophy

Code Smells:

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Divergent Change
- Shotgun Surgery
- Feature Envy
- Data Clumps
- Primitive Obsession
- Switch Statements
- Parallel Inheritance Hierarchies
- Lazy Class
- Speculative Generality
- Temporary Field
- Message Chains
- Middle Man
- Inappropriate Intimacy
- Alternative Classes with Different Interfaces
- Incomplete Library Class
- Data Class
- Comments
- Refused Bequest

Catalog of Refactorings:

- Rename method, Move method, Extract method, Extract class, Hide delegates, Inline class ...

Smell: Duplicated Code

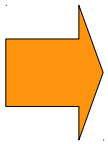
If the same code structure is repeated.

- **Extract Method** - gather duplicated code
- **Pull Up Field** - move to a common parent
- **Form Template Method** - gather similar parts, leaving holes
- **Substitute Algorithm** - choose the clearer algorithm
- **Extract class** - for unrelated classes, create a new class with functionality

Refactoring: Extract Method

- Sometimes we have methods that do too much. The more code in a single method, the harder it is to understand and get right. It also means that logic embedded in that method cannot be reused elsewhere.

```
public class Person
{
    public int foo() {
        …
        int score = a*b+c;
        score = score * 0.8 / Math.PI;
        score *= factor;
        ...
        return score;
    }
}
```



```
public class Person
{
    public int foo() {
        …
        return compteScore(a,b,c,factor);
    }

    public int computeScore(int a, int b, int c, float factor) {
        int score = a*b+c;
        score = score * 0.8 / Math.PI;
        score *= factor;
    }
}
```

- The *Extract Method* refactoring is one of the most useful for reducing the amount of duplication in code.

Refactoring: Form Template Method

- If you find two methods in subclasses that perform the same steps, but do different things in each step, create methods for those steps with the same signature and move the original method into the base class

```
public abstract class Party { }
```

```
public class Company extends Party
{
   private String name;
   private String companyType;
   private Date   incorporated;

   public void printNameAndDetails() {
      System.out.println("Name: " + name + " " + companyType);
      System.out.println("Incorporated: " + incorporated.toString());
   }
}
```

```
public class Person extends Party
{
   private String firstName;
   private String lastName;
   private Date   dob;
   private String nationality;

   public void printNameAndDetails()  {
      System.out.println("Name: " + firstName + " " + lastName);
      System.out.println("DOB: " + dob.toString() + ", Nationality: " + nationality);
   }
}
```

Refactoring: Form Template Method (After)

```
public abstract class Party {
   public void printNameAndDetails() {
      printName();
      printDetails();
    }
    public abstract void printName();
    public abstract void printDetails();
}
```

```
public class Company extends Party
{
    private String name;
    private String companyType;
    private Date incorporated;

    public void printDetails() {
       System.out.println("Incorporated: " + incorporated.toString());
    }

    public void printName()  {
       System.out.println("Name: " + name + " " + companyType);
    }
}
```

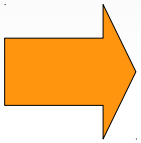
```
public class Person extends Party
{
    private String firstName;
    private String lastName;
    private Date dob;
    private String nationality;

    public void printDetails() {
       System.out.println("DOB: " + dob.toString() + ", Nationality: " + nationality);
    }

    public void printName() {
       System.out.println("Name: " + firstName + " " + lastName);
    }
}
```

Did you recognize GoF design pattern?

Refactoring: Extract Class

- Break one class into two.

```
public class Customer
{
   private String name;
   private String workPhoneAreaCode;
   private String workPhoneNumber;
   private String homePhoneAreaCode;
   private String homePhoneNumber;
}
```



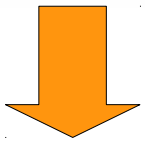
```
public class Customer
{
    private String name;
    private Phone workPhone;
    private Phone homePhone;
}
```

```
public class Phone
{
    private String areaCode;
    private String number;
}
```

Refactoring: Substitute Algorithm

- Choose the clearer algorithm. Next code prints items separated by comma (no comma is printed after the last item).

```
for (int i = 0; i < items.length-1; i++) {
    System.out.println(items[i] + ”,”);
}
System.out.println(items[items.length-1]);
```



```
for (int i = 0; i < items.length; i++) {
    System.out.println(items[i]);
    if (i < items.length-1) System.out.println(“,”);
}
```

Smell: Long Method

If the body of a method is over a page (choose your page size)

- **Extract Method** - extract related behavior
- **Replace Temp with Query** - remove temporaries when they obscure meaning
- **Introduce Parameter Object** - slim down parameter lists by making them into objects
- **Replace Method with Method Object** - still too many parameters
- **Decompose Conditionals** - conditional and loops can be moved to their own methods

Refactoring: Replace Temp with Query

- You are using a temporary variable to hold the result of an expression. Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods and allows for other refactorings.

```
double basePrice = quantity * itemPrice;
if (basePrice > 1000)
        return basePrice * 0.95;
else
        return basePrice * 0.98;
```

```
{ ...
  if (basePrice() > 1000)
        return basePrice() * 0.95;
  else
        return basePrice() * 0.98;
  ...
}

double basePrice() {
        return quantity * itemPrice;
}
```

Refactoring: Introduce Parameter Object

- Slim down parameter lists by making them into objects (e.g. via nested class).

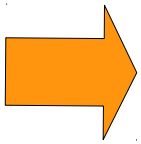
```
public void handleEmployee(String name, String address, double salary, String evaluation) {
        ...
}
```

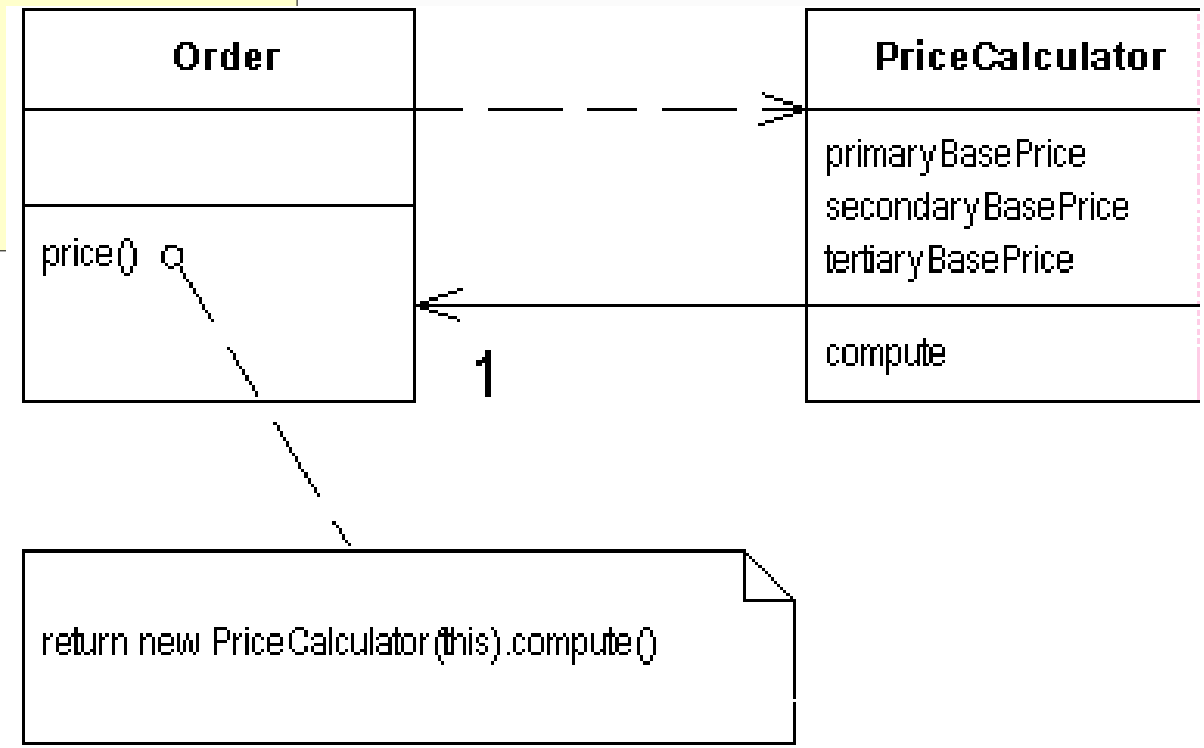
```
public void handleEmployeee(EmployeeParams params) {
        params.getName();
        ...
}

public class EmployeeParams {
        private String name;
        private String address;
        private double salary;
        private String evaluation;

        public String getName() { return name; }
        ...
}
```

Refact.: Replace Method with Method Object

- You have a long method that uses local variables in such a way that you cannot apply Extract Method => Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.

```
public class Order {
    ...
    double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation;
        ...
    }
```





Refactoring: Decompose Conditionals

- You have a complicated conditional (if-then-else) statement => Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END)) {
      charge = quantity * _winterRate + _winterServiceCharge;
} else {
      charge = quantity * _summerRate;
}
```

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

Smell: Long Parameter List

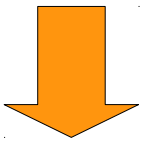
A method does not need many parameter, only enough to be able to retrieve what it needs

- **Replace Parameter with Method** - turn a parameter into a message
- **Introduce Parameter Object** - turn several parameters into an object

Refactoring: Replace Parameter with Method

- An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method => Remove the parameter and let the receiver invoke the method.

```
int basePrice = quantity * itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice(basePrice, discountLevel);
```



```
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice (basePrice);
```

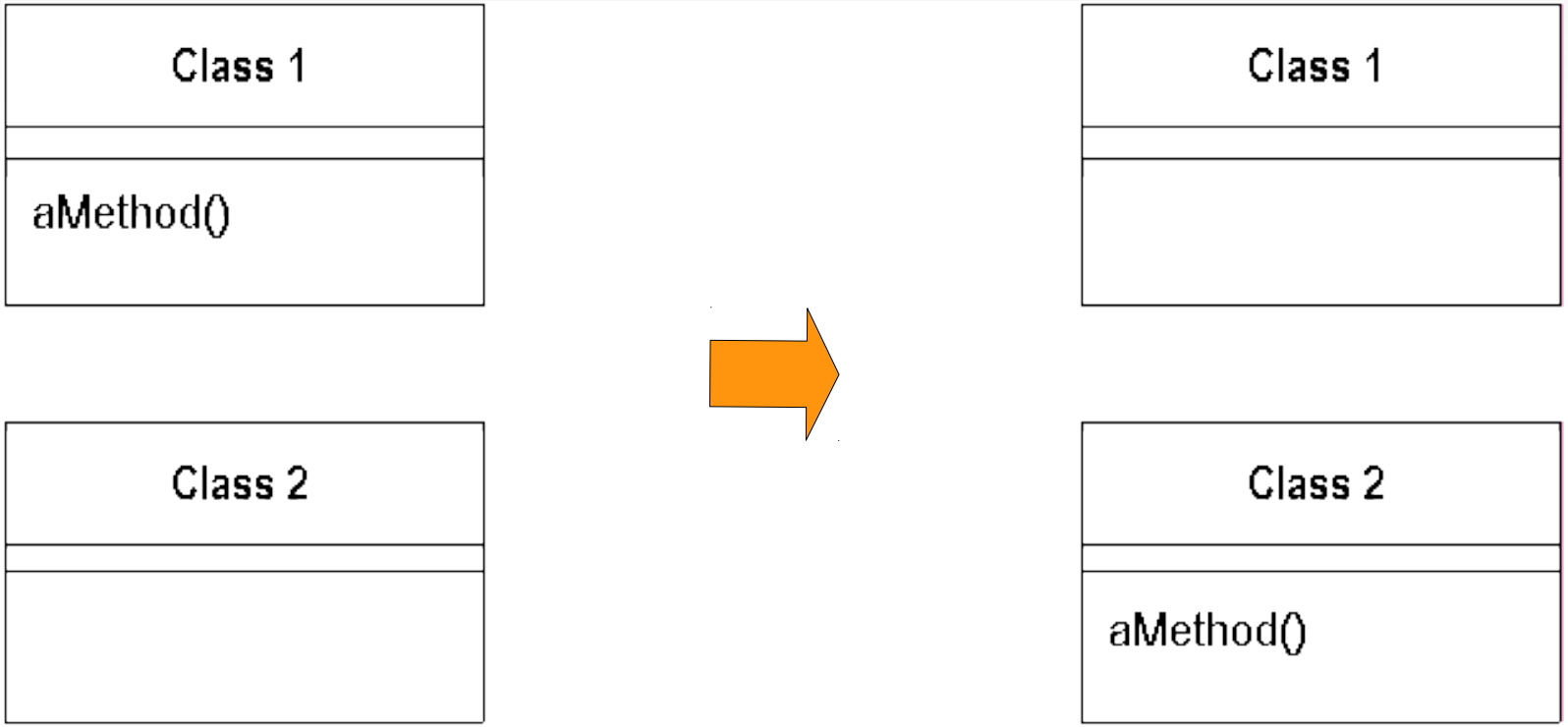
Smell: Shotgun Surgery

If you find yourself making a lot of small changes for each desired change

- **Move Method/Field** – pull all the changes into single class
- **Inline Class** – group a bunch of behavior together

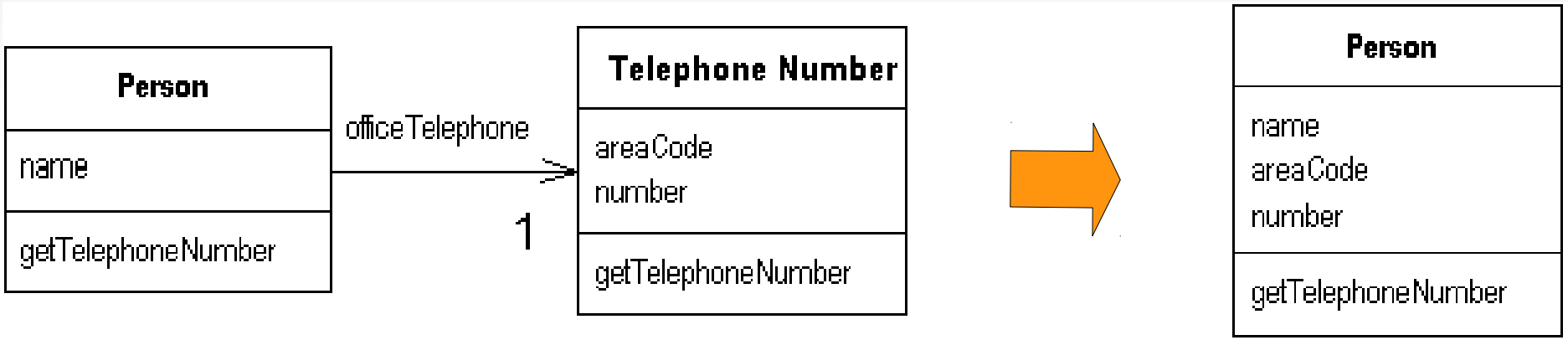
Refactoring: Move Method/Field

- A method is, or will be, using or used by more features of another class than the class on which it is defined => Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.
- A field is, or will be, used by another class more than the class on which it is defined => Create a new field in the target class, and change all its users.



Refactoring: Inline Class

- A class isn't doing very much => Move all its features into another class and delete it.



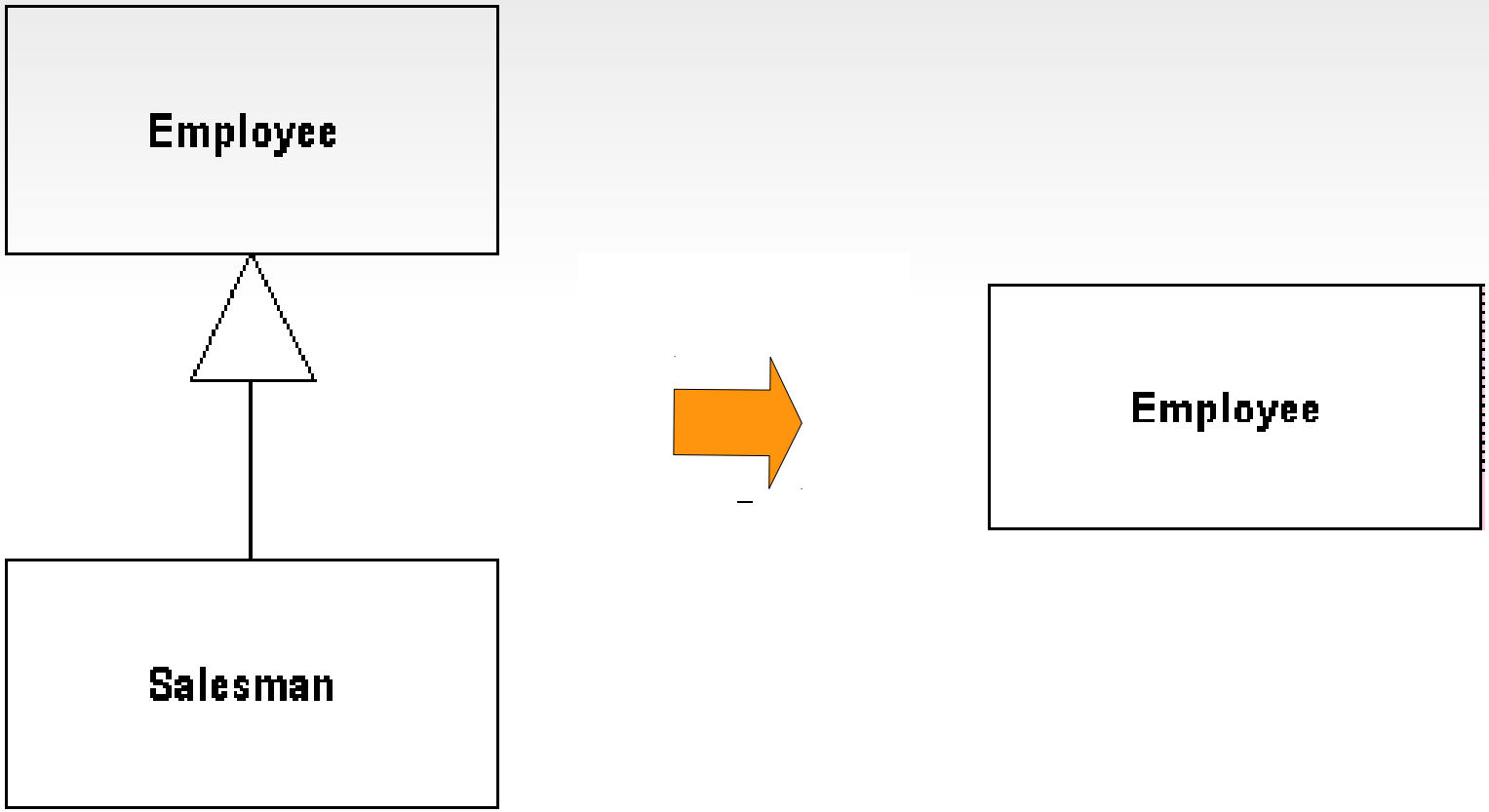
Smell: Speculative Generality

If a class has features that are only used in test cases, remove them.

- **Collapse Hierarchy** - for useless abstract classes
- **Inline Class** - for useless delegation
- **Rename Method** - methods with odd abstract names should be brought down to earth

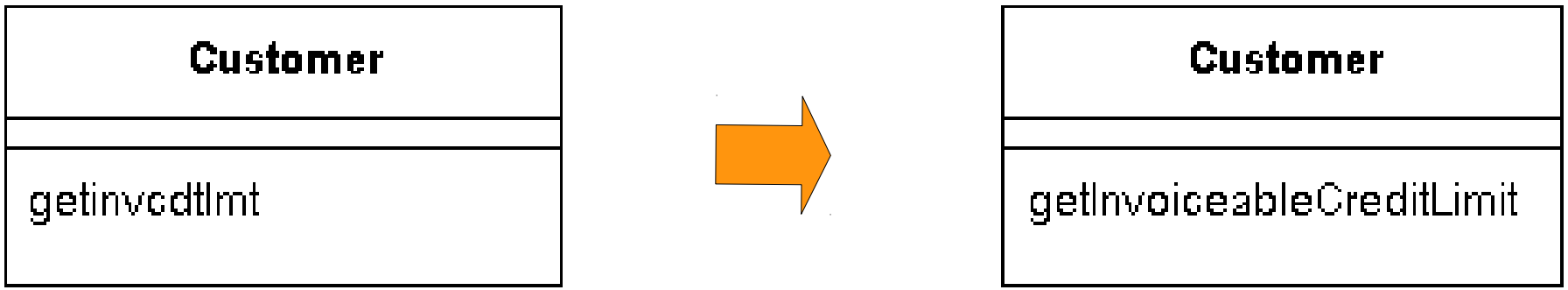
Refactoring: Collapse Hierarchy

- A superclass and subclass are not very different => Merge them together.



Refactoring: Rename Method

- The name of a method does not reveal its purpose => Change the name of the method.



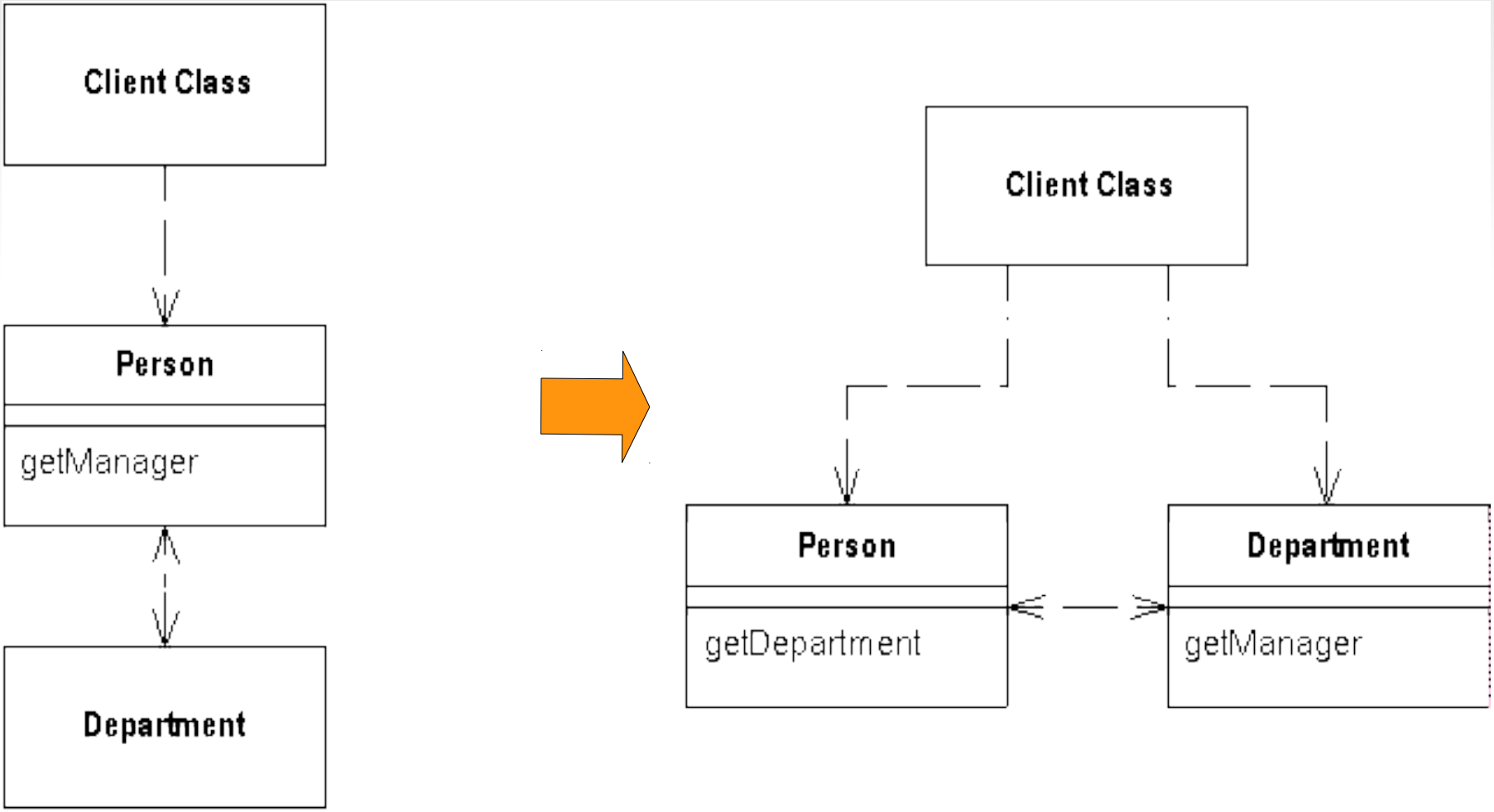
Smell: Middle Man

An intermediary object is used too often to get at encapsulated values

- **Remove Middle Man** - to talk directly to the target
- **Replace Delegation with Inheritance** - turns the middle man into a subclass of the real object

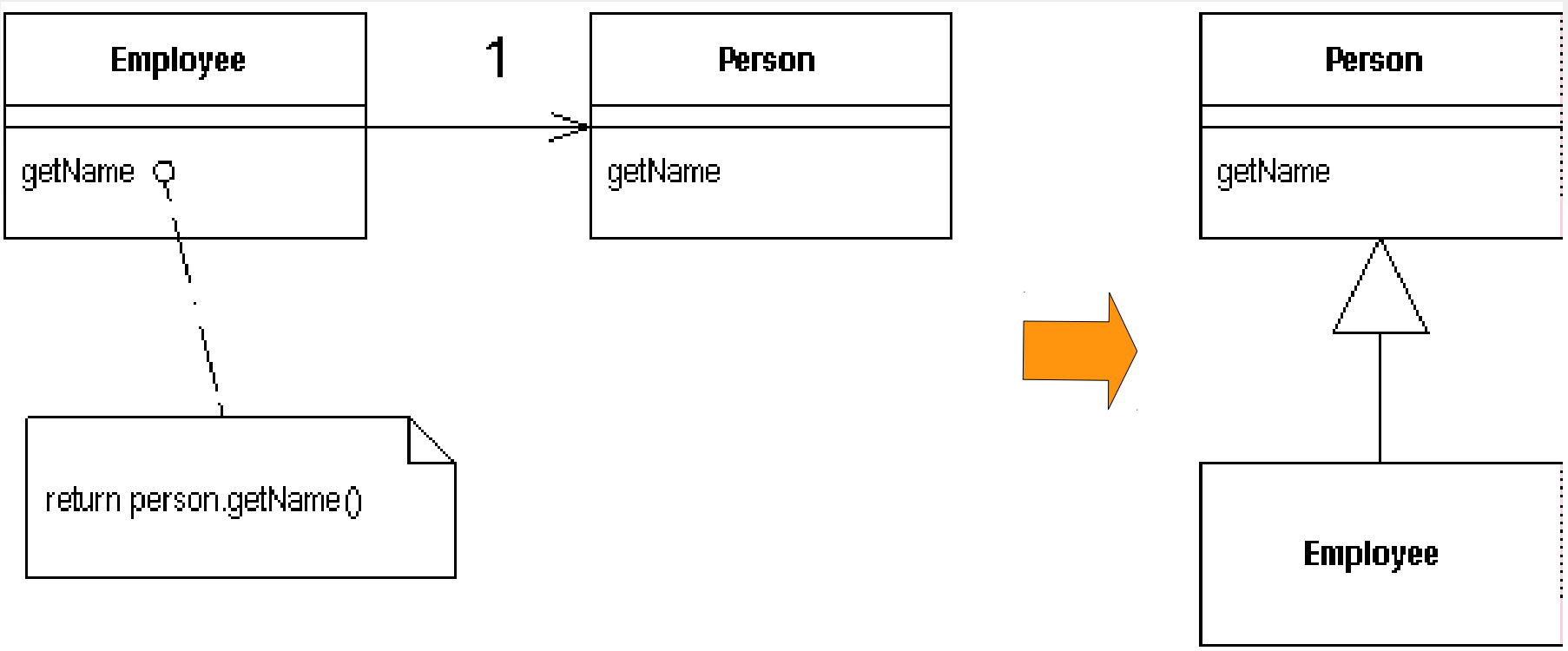
Refactoring: Remove Middle Man

- A class is doing too much simple delegation => Get the client to call the delegate directly.



Refact.: Replace Delegation with Inheritance

- You're using delegation and are often writing many simple delegations for the entire interface => Make the delegating class a subclass of the delegate.



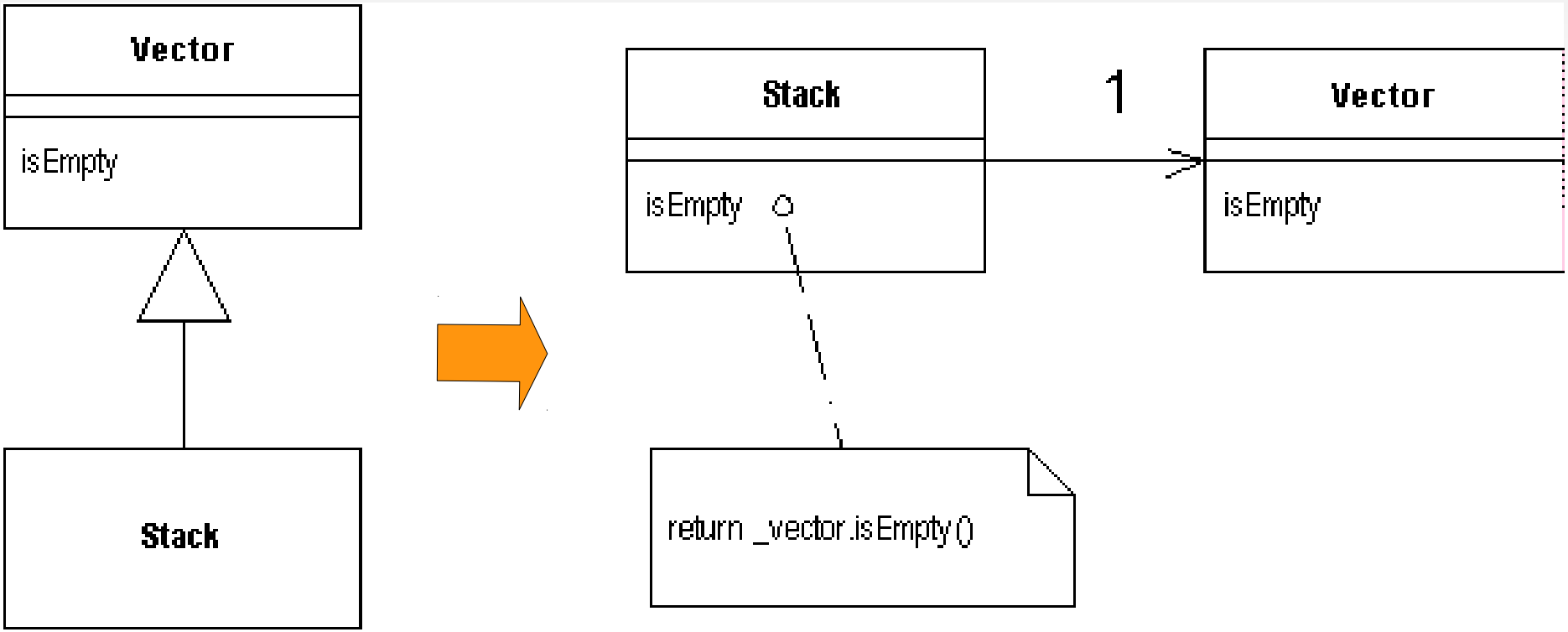
Smell: Inappropriate Intimacy

Classes are too intimate and spend too much time delving in each other's private parts

- **Move Method/Field** - to separate pieces in order to reduce intimacy
- **Extract Class** - make a common class of shared behavior/data
- **Replace Inheritance with Delegation** - when a subclass is getting too cozy

Refact.: Replace Inheritance with Delegation

- A subclass uses only part of a superclasses interface or does not want to inherit data => Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.



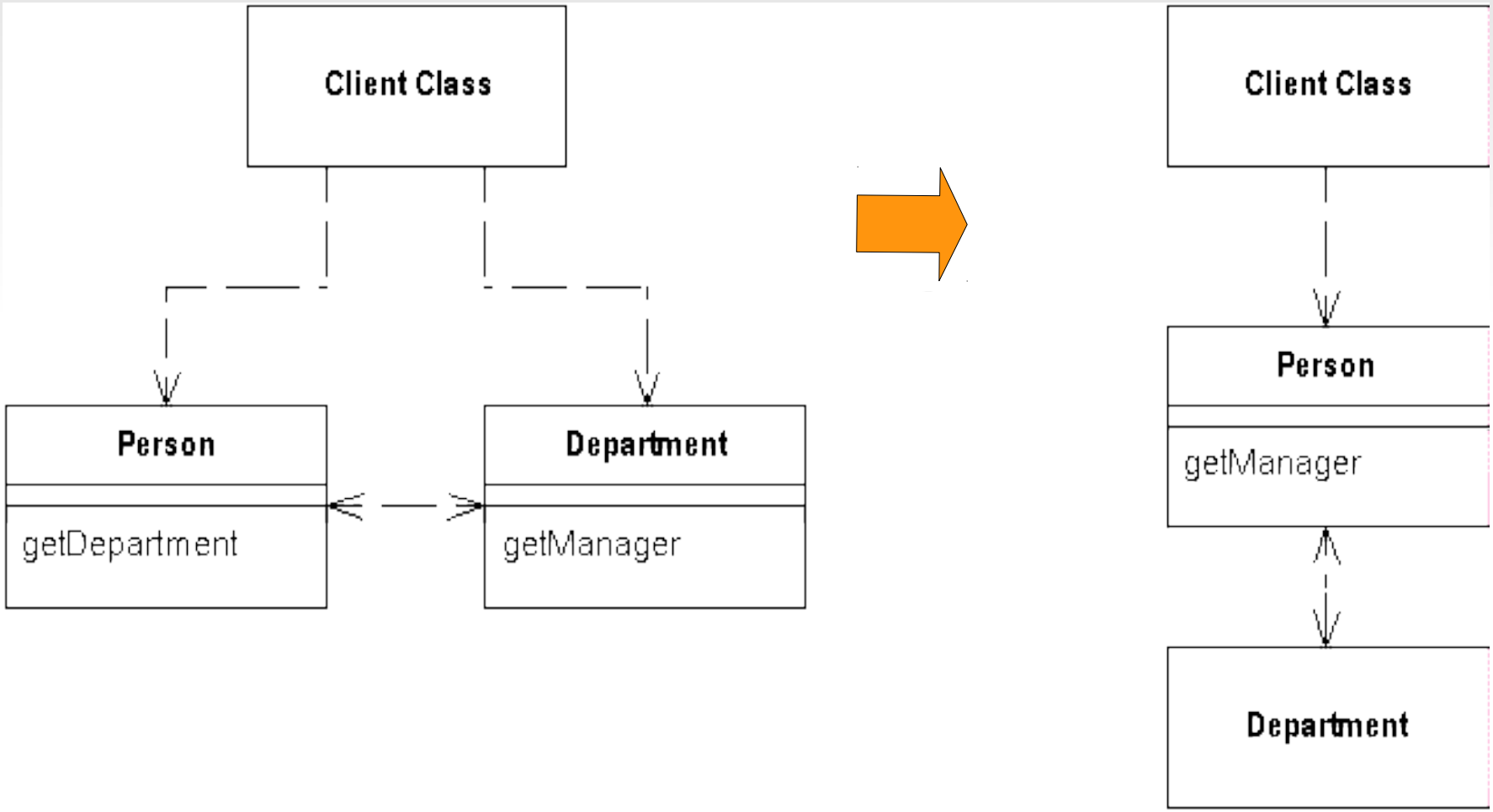
Smell: Message Chain

Long chains of messages to get to a value are brittle as any change in the intermittent structure will break the code

- **Hide Delegate** - remove one link in a chain
- **Extract Method** - change the behavior to avoid chains

Refactoring: Hide Delegate

- A client is calling a delegate class of an object => Create methods on the server to hide the delegate.



Smell: Switch Statements

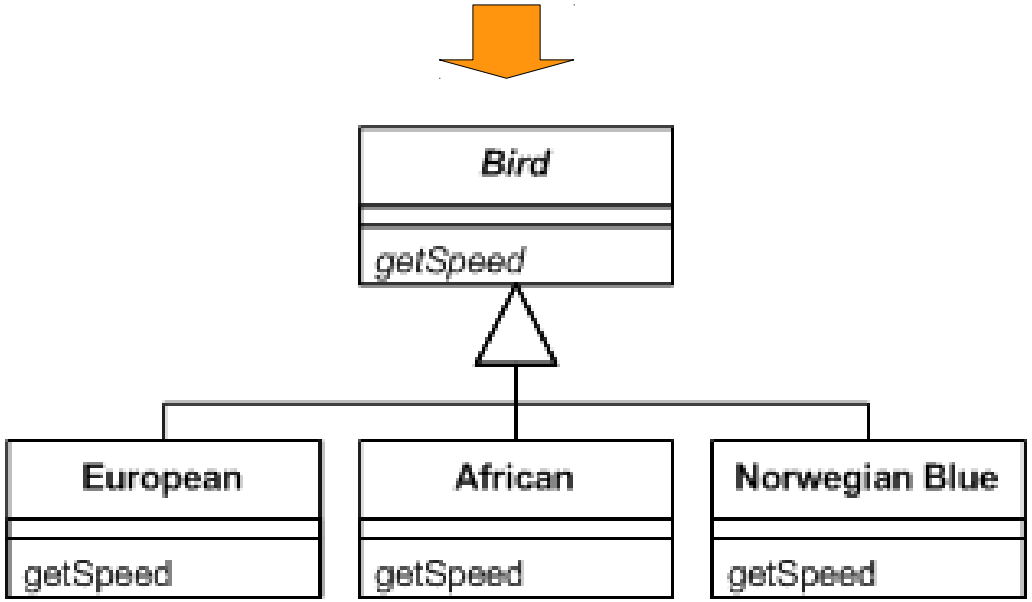
Switch statements lead to duplication and inhibit change

- **Extract method** - to remove the switch
- **Move method** - to get the method where polymorphism can apply
- **Replace Type Code with State/Strategy** - set up inheritance
- **Replace Conditional with Polymorphism** - get rid of the switch

Ref.: Replace Conditional with Polymorphism

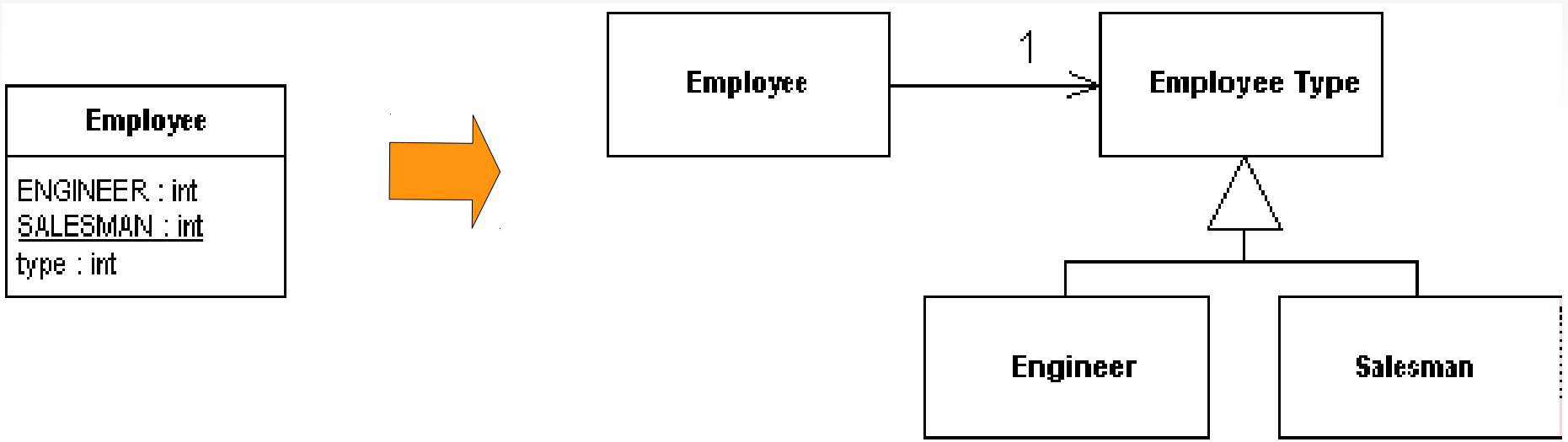
- You have a conditional that chooses different behavior depending on the type of an object => Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

```
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");
}
```



Refact.: Replace Type Code with State/Strategy

- You have a type code that affects the behavior of a class, but you cannot use subclassing => Replace the type code with a state object.



Lecture 4 / Part 2:

Bad Smells In Code
(Overview of Another Bed Smells)

Bad Smells in Code

Large Class

(stench 7)

If a class has either too many variables or too many methods

- **Extract Class** - to bundle variables/methods

Bad Smells in Code

Divergent Change

(stench 5)

If you find yourself repeatedly changing the same class then there is probably something wrong with it

- **Extract Class** - group functionality commonly changed into a class

Bad Smells in Code

Feature Envy

(stench 6)

If a method seems more interested in a class other than the class it actually is in

- **Move Method** - move the method to the desired class
- **Extract Method** - if only part of the method shows the symptoms

Bad Smells in Code

Data Clumps

(stench 4)

Data items that are frequently together in method signatures and classes belong to a class of their own

- **Extract Class** - turn related fields into a class
- **Introduce Parameter Object** - for method signatures

Bad Smells in Code

Primitive Obsession

(stench 3)

Primitive types inhibit change

- **Replace Data Value with Object** - on individual data values
- **Move Method/Field** - pull all the changes into a single class
- **Introduce Parameter Object** - for signatures
- **Replace Array with Object** - to get rid of arrays

Parallel Inheritance Hierarchies

(stench 6)

If when ever you make a subclass in one corner of the hierarchy, you must create another subclass in another corner

- **Move Method/Field** - get one hierarchy to refer to the other

Bad Smells in Code

Lazy Class

(stench 4)

If a class (e.g. after refactoring) does not do much, eliminate it.

- **Collapse Hierarchy**- for subclasses
- **Inline Class** - remove a single class

Bad Smells in Code

Temporary Field

(stench 3)

If a class has fields that are only set in special cases, extract them

- **Extract Class** - for the special fields

Bad Smells in Code

Comments

(stench 2)

Comments are often a sign of unclear code... consider refactoring

Questions?