Refactoring: Improving the Design of Existing Code

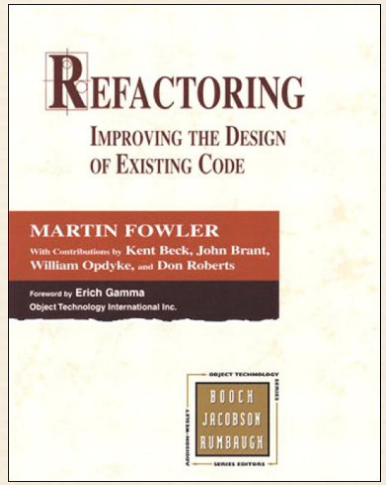
One of the best references on software refactoring, with illustrative examples in Java:

Refactoring: Improving the Design of Existing Code. Martin Fowler. Addison Wesley, 2000. ISBN: 0201485672

See also www.refactoring.com Overview of this presentation

A. Refactoring basics

B. Categories of refactoring



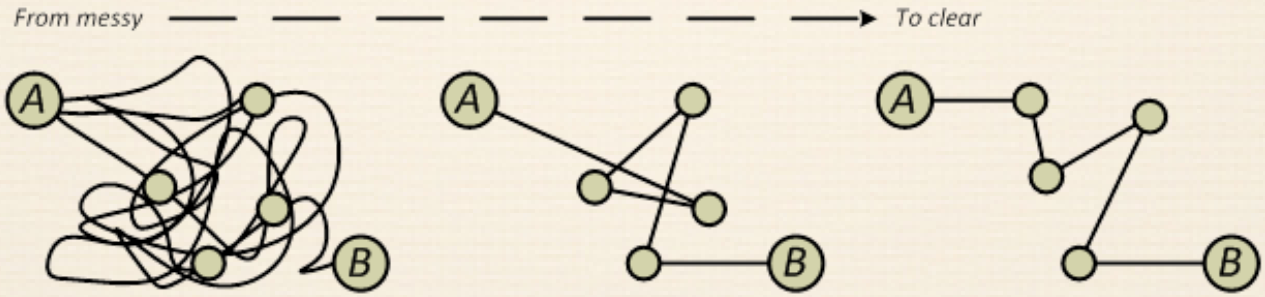
What is refactoring?

A **refactoring** is a software transformation that

preserves the external behaviour of the software;

improves the internal structure ofthe software.

It is a disciplined way to clean up code that minimises the chances of introducing bugs.



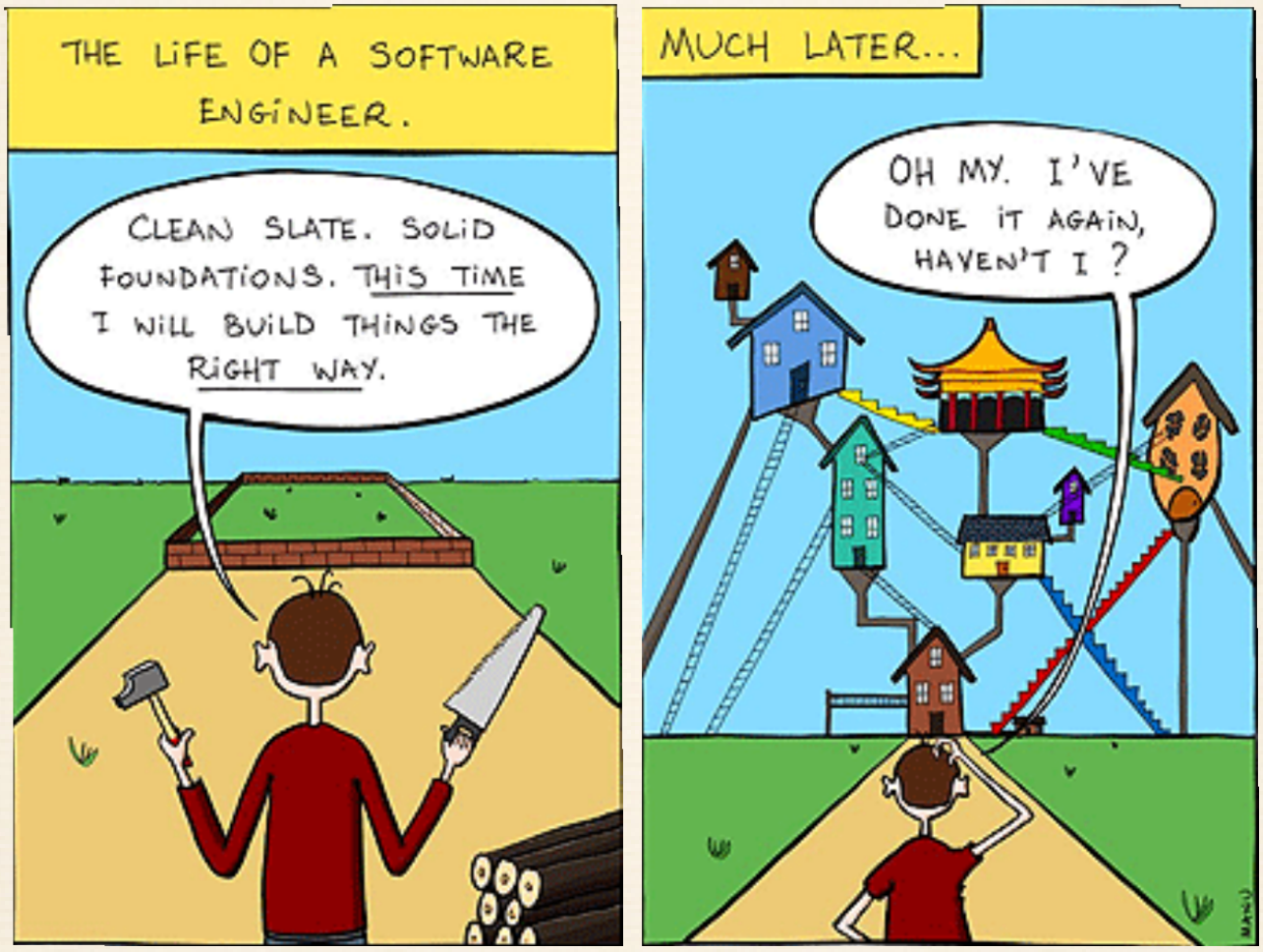
Definition of Refactoring [Fowler2000]

[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”

[verb] “to restructure software by applying a series of refactorings without changing its observable behaviour”

typically with the purpose of making the software easier to understand and modify

Why should you refactor?



Why should you refactor?

To **improve the design** of software

To counter code decay (**software ageing**)

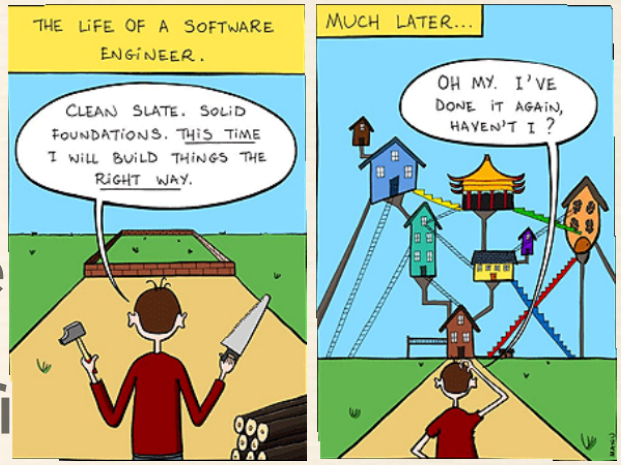
refactoring helps code to remain in shape

To increase **software comprehensibility** to **fi**

bugs and write more robust code To increase

productivity (program faster) on a long term

basis, not on a short term basis



Why should you refactor?

To **reduce costs** of software maintenance

To **reduce testing**

automatic refactorings are guaranteed to be behaviour preserving

To prepare for / facilitate **future customizations** To turn an

OO application into **a framework**

To introduce **design patterns** in a behaviourally preserving way

When should you refactor?

Whenever you see the need for it

Do it all the time in little bursts Not on a pre-scheduled periodical basis

Apply the rule of three

1st time : implement from scratch

2nd time : implement something similar by code duplication 3rd time : do not implement similar things again, but refactor

When should you refactor?

Refactor when **adding new features or functions**

- Especially if feature is difficult to integrate with the existing code

Refactor during **bug fixing**

- If a bug is very hard to trace, refactor first to make the code more understandable, so that you can understand better where the bug is located

Refactor during **code reviews**

When should you refactor?

Refactoring also fits naturally in the ***agile methods*** philosophy

Is needed to address the principle "**Maintain simplicity**"

> Wherever possible, actively work to eliminate complexity from the system
>
> By refactoring the code

What do you tell the manager?

When (s)he's technically aware (s)he'll understand why refactoring is important.

When (s)he's interested in quality, (s)he'll understand that refactoring will improve software quality.

When (s)he's only interested in the schedule, don't tell that you're doing refactoring, just do it anyway.

In the end refactoring will make you more productive.

When shouldn't you refactor?

When the existing code is such a mess that although you could refactor it, it would be easier to rewrite everything from scratch instead.

When you are too close to a deadline.

The productivity gain would appear after the deadline and thus be too late.

However, when you are *not* close to a deadline you should *never* put off refactoring because you don't have the time.

Not having enough time usually is a sign that refactoring is needed.

Categories of Refactoring

Categories of refactorings

Small refactorings

(de)composing methods

moving features between objects organizing data

dealing with generalisation

simplifying method calls

Big refactorings

Tease apart inheritance Extract hierarchy

Convert procedural design to objects Separate domain from presentation

Small refactorings

(de)composing methods [7 refactorings]

moving features between objects [8 refactorings]

organizing data [16 refactorings]

dealing with generalisation [12 refactorings]

simplifying method calls [15 refactorings]

Small Refactorings : (de)composing methods

1. Extract Method
2. Inline Method
3. Replace Temp With Query
4. Introduce Explaining Variable
5. Split Temporary Variable
6. Remove Assignments to Parameter
7. Substitute Algorithm

Legend:

= we will zoom in on these

= home reading

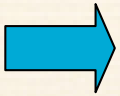
(De)composing methods: 1. Extract Method

What? When you have a fragment of code that can be grouped together, turn it into a method with a name that explains the purpose of the method

Why? improves clarity, removes redundancy

Example:

```
public void accept(Packet p) {
  if ((p.getAddressee() == this) &&  (this.isASCII(p.
        getContents())))
     this.print(p);  else
     super.accept(p); }
```



```
public void accept(Packet p) {
  if this.isDestFor(p) this.print(p);  else super.
  accept(p); }
public boolean isDestFor(Packet p) {  return
    ((p.getAddressee() == this) &&  (this.isASCII(p.
      getContents()))); }
```

Beware of local variables !

(De)composing methods : Inline Method

(Opposite of Extract Method)

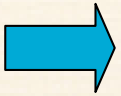
What? When a method's body is just as clear as its name, put the method's body into the body of its caller and remove the method

Why? To remove too much indirection and delegation

Example:

```
int getRating(){
    return moreThanFiveLateDeliveries();
}

boolean moreThanFiveLateDeliveries(){ return
    _numberOfLateDeliveries > 5;
}
```



```
int getRating(){
return (_numberOfLateDeliveries > 5);
}
```

(De)composing methods : 3. Replace Temp with Query

What? When you use a temporary variable to hold the result of an expression, extract the expression into a method and replace all references to the temp with a method call

Why? Cleaner code

Example:

```
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;
}
```

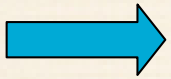
(De)composing methods : 4. Introduce Explaining Variable

What? When you have a complex expression, put the result of the (parts of the) expression in a temporary variable with a name that explains the purpose

Why? Breaking down complex expressions for clarity

Example:

```
if ( (platform.toUpperCase().indexOf("MAC") > -1) &&  (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
{
//ACTION
}
```



```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;  final boolean isIEBrowser
= browser.toUpperCase().indexOf("IE") > -1;  final boolean wasResized = resize > 0;

if (isMacOs   && isIEBrowser    && wasInitialized() && wasResized){
//ACTION
}
```

(De)composing Methods : Split Temporary Variable

What? When you assign a temporary variable more than once, but it is not a loop variable nor a collecting temporary variable, make a separate temporary variable for each assignment

Why? Using temps more than once is confusing

Example:

```
double temp    = 2 * (_height + _width); System.
out.println (temp);
temp    = _height * _width; System.out.pri
(temp);
```

```
final double perimeter =
                    2 * (_height + _width); System.
out.println (perimeter);
final double area  = _height * _width; System.out.
println (area);
```

(De)composing methods : 6. Remove Assignments To Parameter

What? When the code assigns to a parameter, use a temporary variable instead

Why? Lack of clarity and confusion between “pass by value” and “pass by reference”

Example:

```
int discount (int inputVal, int quantity,  int yearToDate){
    if (inputVal > 50) inputVal -= 2;
    ... MORE CODE HERE ...
```

```
int discount (int inputVal, int quantity,  int yearToDate){
    int result = inputVal;
    if (inputVal > 50) result -= 2;
  ... MORE CODE HERE ...
```

(De)composing methods : 7. Substitute Algorithm

What? When you want to replace an algorithm with a clearer alternative, replace the body of the method with the new algorithm

Why? To replace complicated algorithms with clearer ones

Example:

```
String foundPerson(String[] people){
      for (int i = 0; i < people.length; i++){  if (people[i]. equals
                  (“John”) ) {
                        return “John”;
                  }
  if (people[i]. equals (“Jack”) ) {
                        return “Jack”;
                  }
      }
}
```

```
String foundPerson(String[] people){
      List candidates = Array.asList(new String[] {“John”, “Jack”, Tauseef”})
      for (int i = 0; i < people.length; i++)
            if (candidates[i]. contains (people[i]))  return people[i];
}
```

Small refactorings

(de)composing methods [9 refactorings]

moving features between objects [8 refactorings]

organizing data [16 refactorings]

dealing with generalisation [12 refactorings]

simplifying method calls [15 refactorings]

Small Refactorings : moving features between objects

1. Move Method
2. Move Field
3. Extract Class
4. Inline Class
5. Hide Delegate
6. Remove Middle Man

Legend:

= we will zoom in on these

= home reading

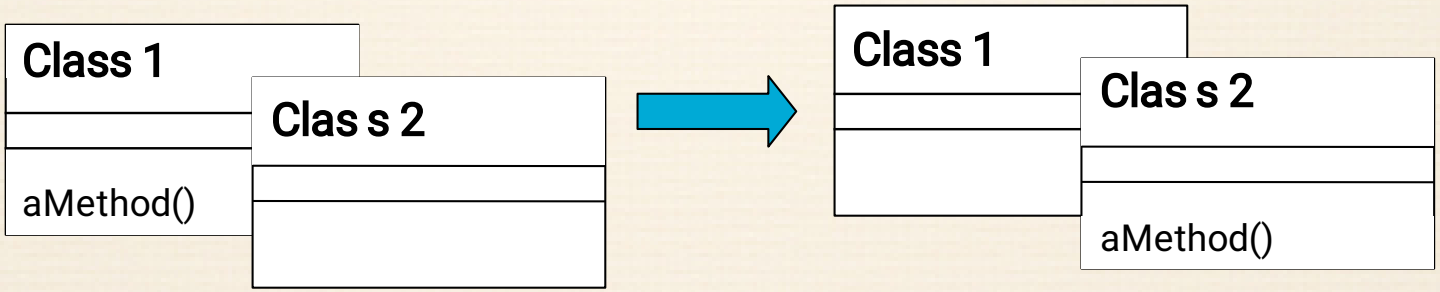
Moving features between objects : 1,2. Move Method / Field

What? When a method (resp. field) is used by or uses more features of another class than its own, create a similar method (resp. field)

in the other class; remove or delegate original method (resp. field) and redirect all references to it.

Why? Essence of refactoring

Example:

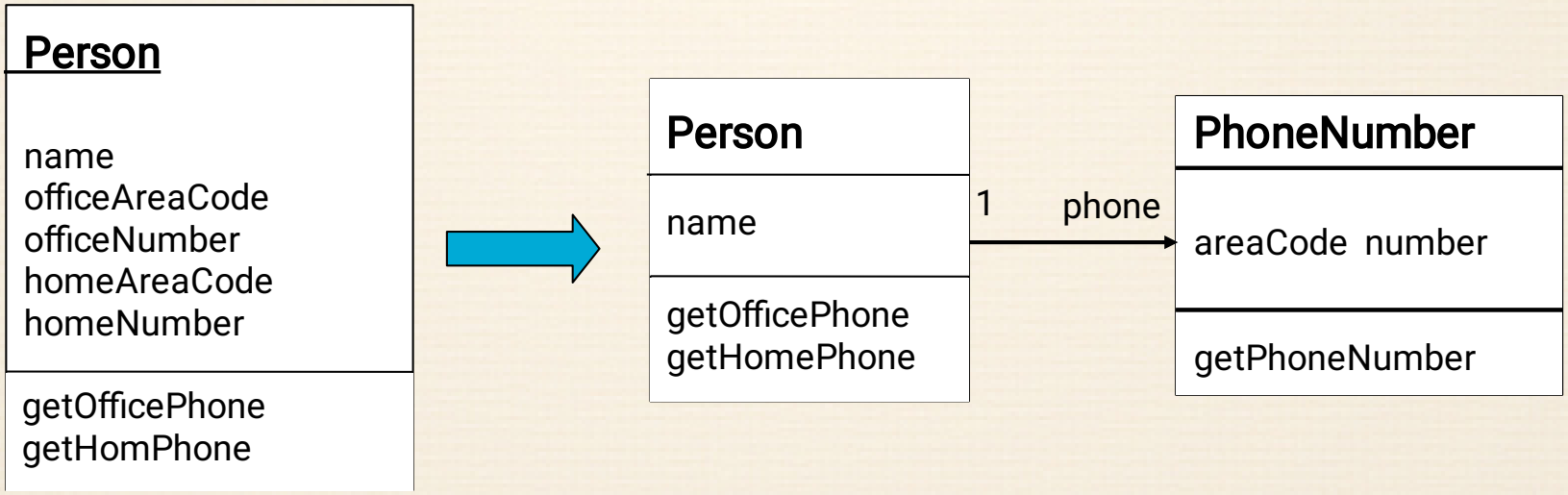


Moving features between objects :3. Extract Class

What? When you have a class doing work that should be done by two, create a new class and move the relevant fields and methods to the new class

Why? Large classes are hard to understand

Example:

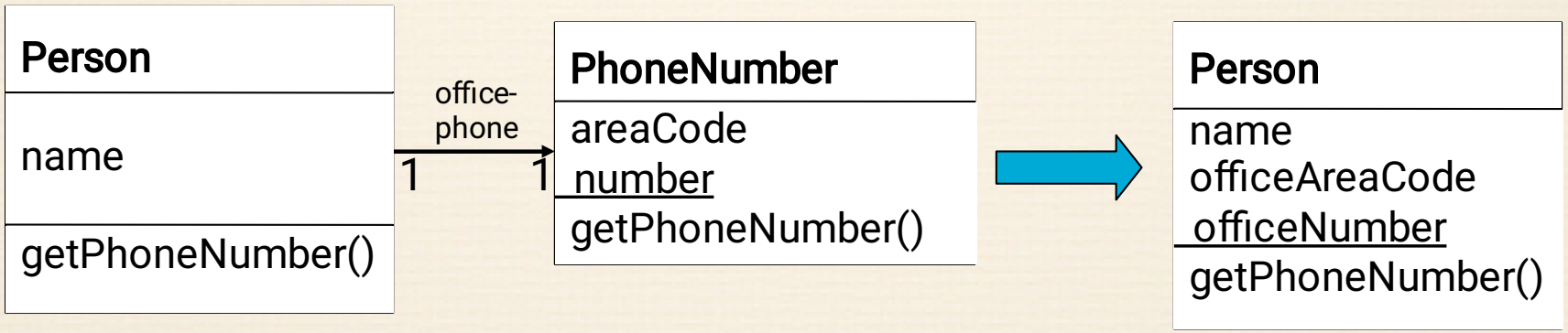


Moving features between objects : 4. Inline Class

What? When you have a class that does not do very much, move all its features into another class and delete it

Why? To remove useless classes (as a result of other refactorings)

Example:

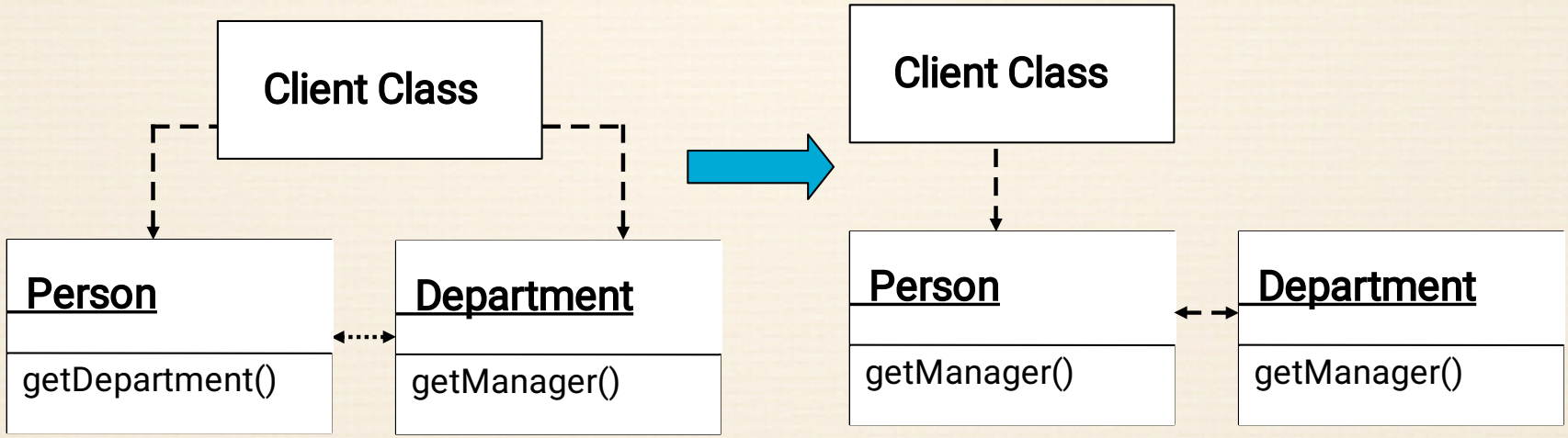


Moving features between objects 5. Hide Delegate

What? When you have a client calling a delegate class of an object, create methods on the server to hide the delegate

Why? Increase encapsulation

Example:

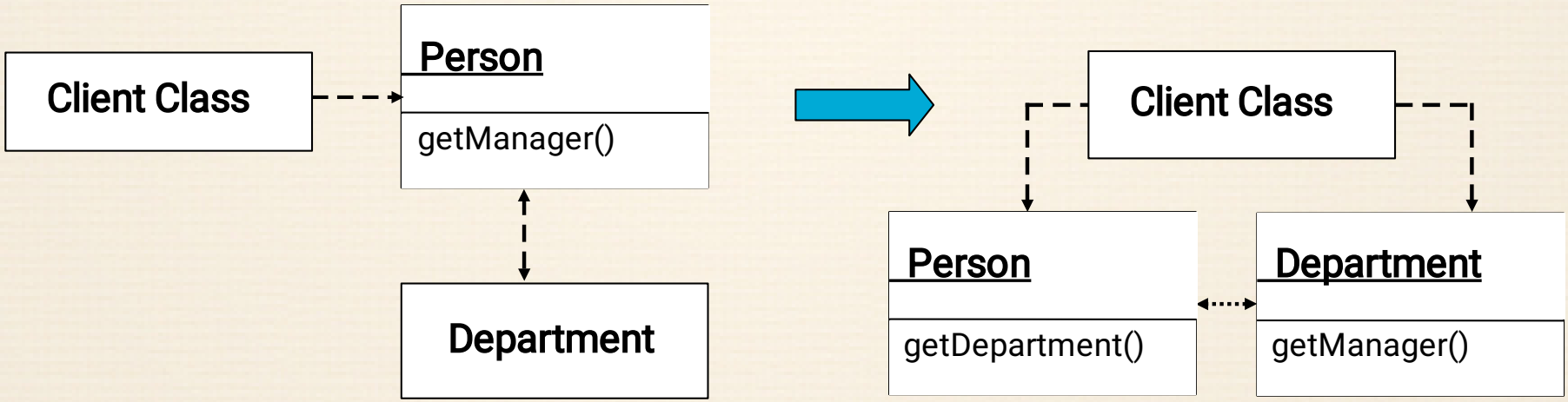


Moving features between objects 6: Remove Middle Man

What? When a class is doing too much simple delegation, get the client to call the delegate directly

Why? To remove too much indirection (as a result of other refactorings)

Example:



Small refactorings

(de)composing methods [9 refactorings]

moving features between objects [8 refactorings]

organizing data [16 refactorings]

simplifying conditional expressions [8 refactorings]

dealing with generalisation [12 refactorings]

simplifying method calls [15 refactorings]

Small Refactorings : organizing data

1. Encapsulate field
2. Replace data value with object
3. Change value to reference
4. Change reference to value
5. Replace array with object
6. Duplicate observed data
7. Change unidirectional
8. Change bidirectional association to unidirectional
9. Replace magic number with symbolic constant
10. Encapsulate collection
11. Replace record with data class
12. Replace subclass with fields

13-16. Replace type code with

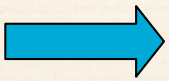
Organizing Data : 1. Encapsulate Field

What? There is a public field. Make it private and provide accessors.

Why? Encapsulating state increases modularity, and facilitates code reuse and maintenance.
When the state of an object is represented as a collection of private variables, the internal representation can be changed without modifying the external interface

Example:

public String **name**;



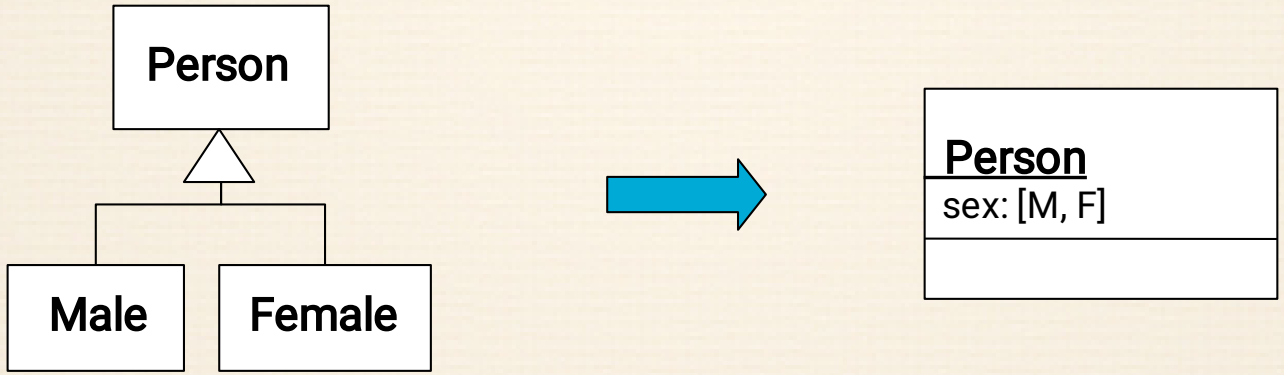
private String **name**; public String **getName**() {
return this.name; }
public void **setName**(String s) { this.name = s; }

Organizing Data : 12. Replace Subclass with Fields

What? Subclasses vary only in methods that return constant data

Solution: Change methods to superclass fields and eliminate subclasses

Example:

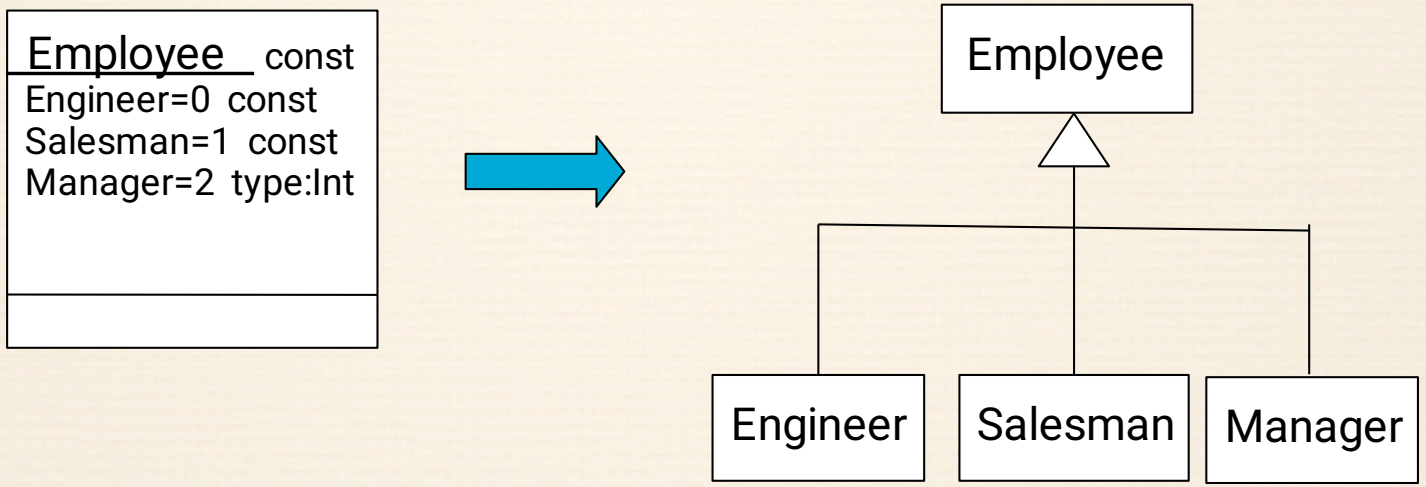


Similar to replace inheritance with aggregation

Organizing Data : 13. Replace Type Code with Subclass

What? An immutable type code affects the behaviour of a class

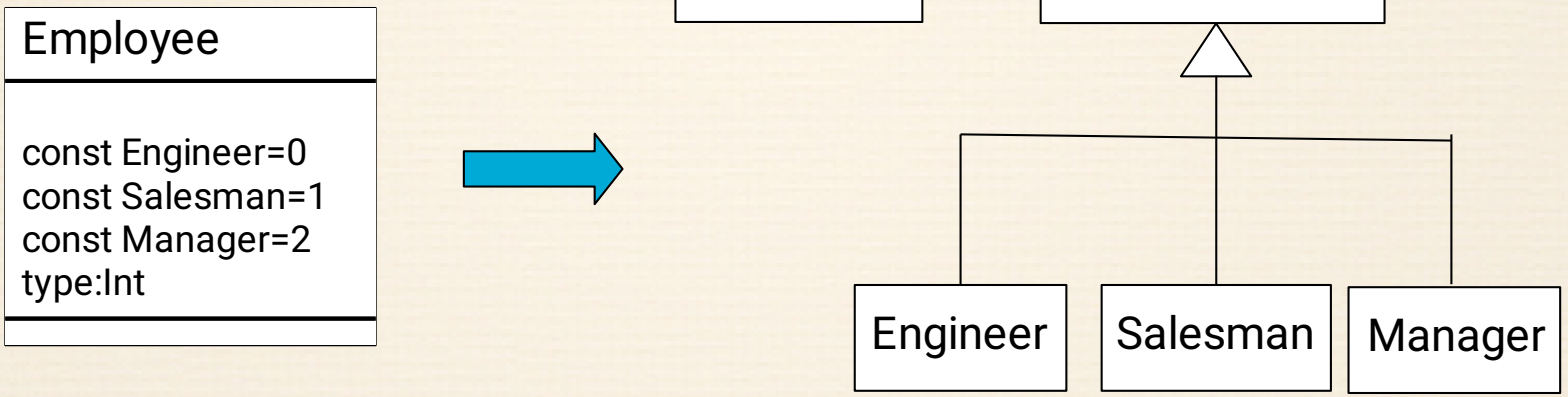
Example:



Organizing Data : 15,16. Replace Type Code with State/ Strategy

When? If subclassing cannot be used, e.g. because of dynamic type changes during object lifetime (e.g. promotion of employees)

Example:



Makes use of state pattern or strategy design pattern

Small refactorings

(de)composing methods [9 refactorings]

moving features between objects [8

refactorings] organizing data [16 refactorings]

dealing with generalisation [12 refactorings]

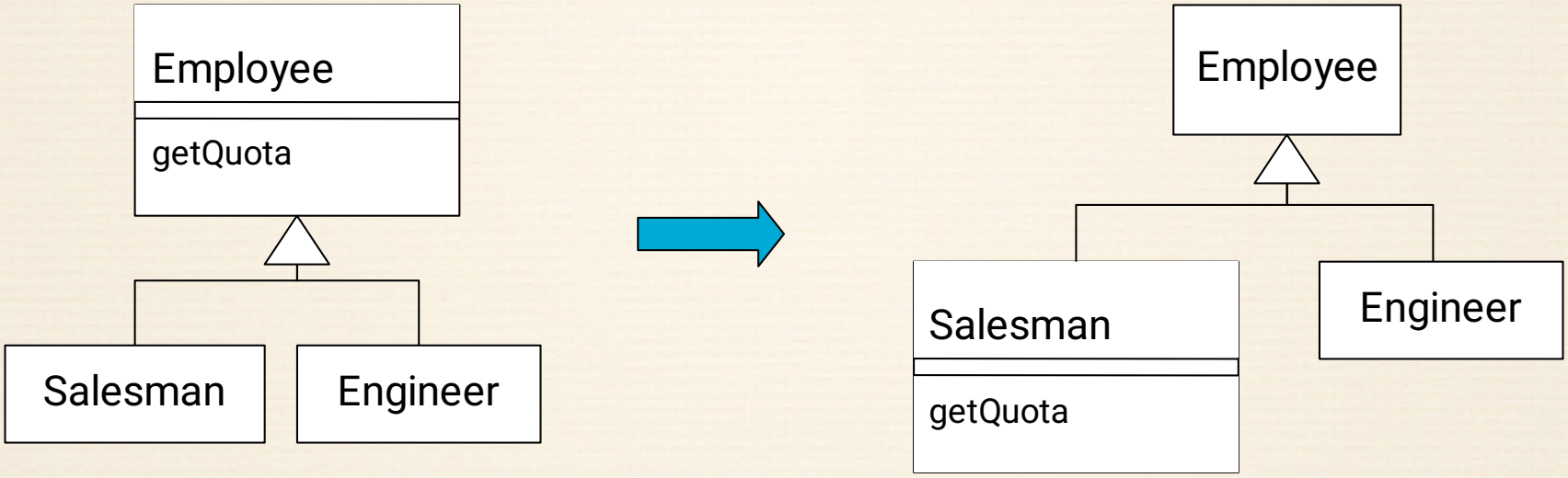
simplifying method calls [15
refactorings]

Small Refactorings : dealing with generalisation

1. Push down method / field
2. Pull up method / field / constructor body
3. Extract subclass / superclass / interface
4. Collapse hierarchy
5. Form template method
6. Replace inheritance with delegation (and vice versa)

Dealing with Generalisation: 1. Pull Down Method

When behaviour on a superclass is relevant only for some of its subclasses, move it to those subclasses

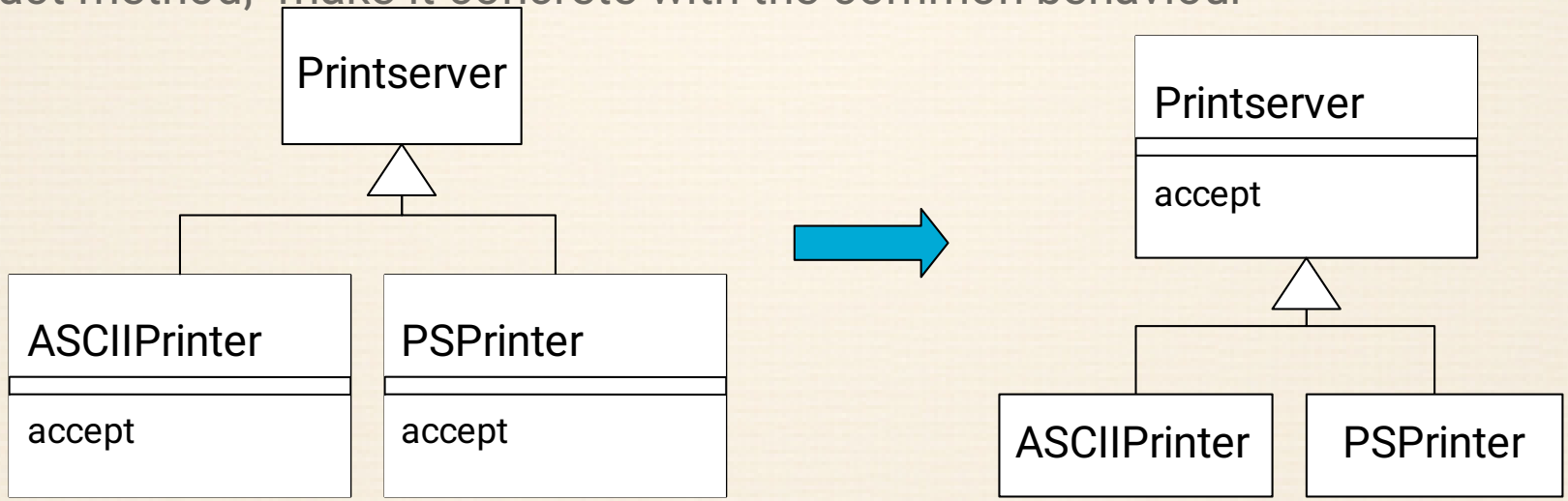


Dealing with Generalisation: 2. Pull Up Method

Simple variant: look for methods with same name in subclasses that do not appear in superclass

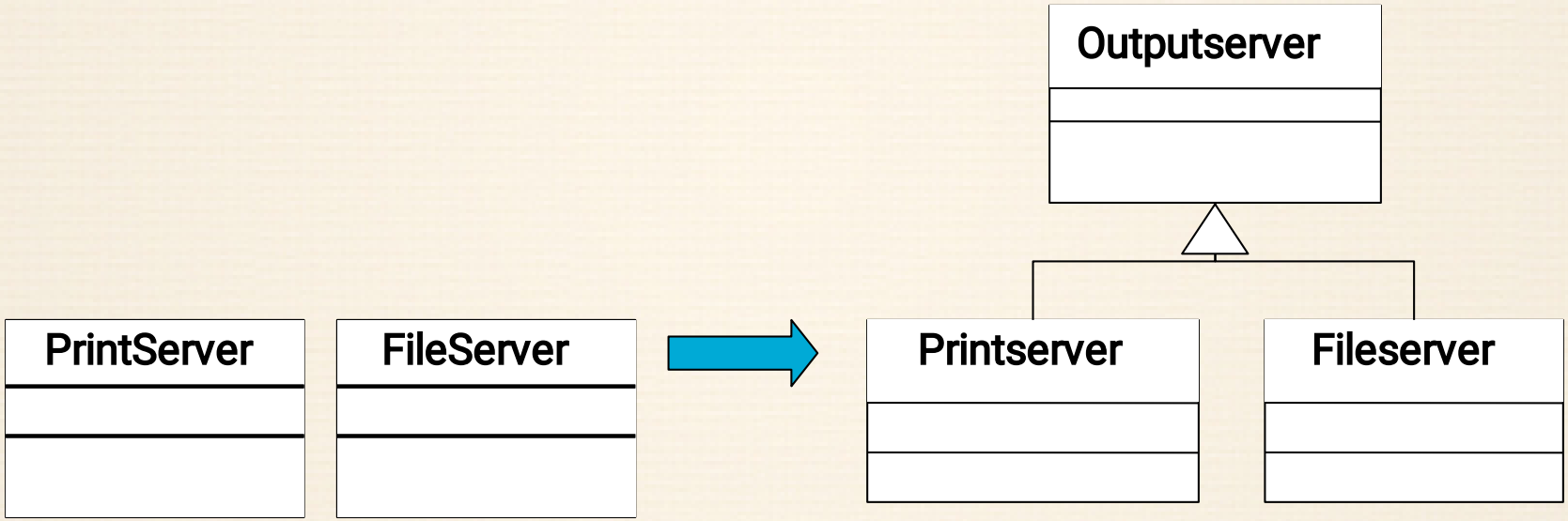
More complex variant: do not look at the name but at the behaviour of the method

If the method that is being pulled up already exists in the superclass as an abstract method, make it concrete with the common behaviour



Dealing with Generalisation: 3. Extract Superclass

When you have 2 classes with similar features



Small refactorings

(de)composing methods [9 refactorings]

moving features between objects [8

refactorings] organizing data [16 refactorings]

simplifying conditional expressions [8

refactorings] dealing with generalisation [12

simplifying method calls [15 refactorings]

Small Refactorings : simplifying method calls

1. Rename method
2. Add parameter
3. Remove parameter
4. Separate query from modifier
5. Parameterize method
6. Replace parameter with method
7. Replace parameter with explicit methods
8. Preserve whole object
9. Introduce parameter object
10. Remove setting method
11. Hide method
12. Replace constructor with factory method
13. Encapsulate downcast
14. Replace error code with exception
15. Replace exception with test

Simplifying method calls: 14. Replace Error Code with Exception

What? When a method returns a special code to indicate an error, throw an exception instead

Why? Clearly separate normal processing from error processing

Example:

```
int withdraw(int amount) {  if (amount > balance)
        return -1
else
    {balance -= amount;  return 0}
}
```

```
void withdraw(int amount) throws BalanceException {  if (amount >
    balance) throw new BalanceException();  balance -= amount;
}
```