REFACTORING

Improving the Design of Existing code

Software Engineering II Class (Lect. 6)

Prof. Marco Faella

Valerio Maggio, Ph.D. Student

A.A. 2011/12

OUTLINE

- What does "Refactoring" means?
  - When should you refactor your code/design?
  - Why should you refactor your code/design?
  - Refactoring Principles
- Typical Refactorings
- Bad Smells & Solutions

(CODE) REFACTORING

Refactoring: *(Definition) Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.* [Fowler02]

- The **art** of safely improving the design of existing code [Fowler09]

- **Implications:** [Fowler09]

  - Refactoring does **not** include any change to the system

  - Refactoring is **not** "Rewriting from scratch"

  - Refactoring is **not** just any restructuring intended to improve the code

REFACTORING

- **Basic Metaphor:**
  - Start with an existing code and make it better;
  - Change the internal structure while preserving the overall semantics.
- **Goals:**
  - Reduce near-duplicate code
  - Improve comprehension and maintainability while reducing coupling

THE REFACTORING CYCLE

```
start with working, tested code
while the design can be simplified do:
    choose the worst smell
    select a refactoring that addresses that smell
    apply the refactoring
    check that the tests still pass
```

- **KWs:**
- Tested Code
  - Importance of Testing in Refactoring
  - Why testing?

- (Code) Smells
  - Fragments of code that contain some design mistake

WHEN SHOULD YOU REFACTOR?

The Rule of Three: *The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.* [Fowler09]

- Refactor when you add a function
- Refactor when you fix a bug
- Refactor for Greater Understanding

WHY SHOULD YOU REFACTOR?

- Refactoring improves the design of software
  - Without refactoring the design of the program will decay
  - Poorly designed code usually takes more code to do the same things
- Refactoring makes software easier to understand
  - In most software development environments, somebody else will eventually have to read your code
- Refactoring helps you find bugs

WHY REFACTORING WORKS?

- Programs that are hard to read, are hard to modify.
- Programs that have duplicated logic, are hard to modify.
- Programs that require additional behavior requiring changing code are hard to modify.
- Programs with complex conditional logic are hard to modify.
- Refactoring makes code more readable

REFACTORING

Good O-O Practices

GOOD SIGNS OF OO THINKING

- Short methods
  - Simple method logic
- Few instance variables
- Clear object responsibilities
  - State the purpose of the class in one sentence
  - No super-intelligent objects

SOME PRINCIPLES

- **The Dependency Inversion Principle**
  - Depend on abstractions, not concrete implementations
    - *Write to an interface, not a class*

- **The Interface Segregation Principle**
  - Many small interfaces are better than one "fat" one

- **The Acyclic Dependencies Principle**
  - Dependencies between package must not form cycles.
    - *Break cycles by forming new packages*

PACKAGES, MODULES AND OTHER

- **The Common Closure Principle**
  - Classes that change together, belong together
    - *Classes within a released component should share common closure. That is, if one needs to be changed, they all are likely to need to be changed.*

- **The Common Reuse Principle**
  - Classes that aren't reused together don't belong together
    - *The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.*

TYPICAL REFACTORINGS

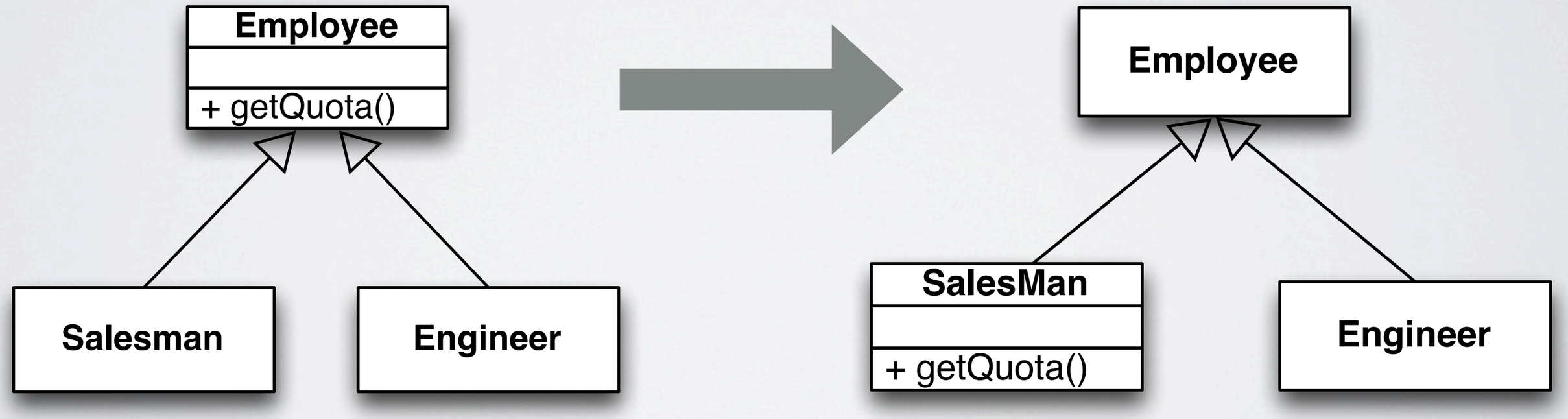
Refactoring in Action

REFACTORINGS TABLE

Class Refactorings	**Method Refactorings**	**Attribute Refactorings**
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
extract class	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method	abstract variable
	extract code in new method	
	replace parameter with method	

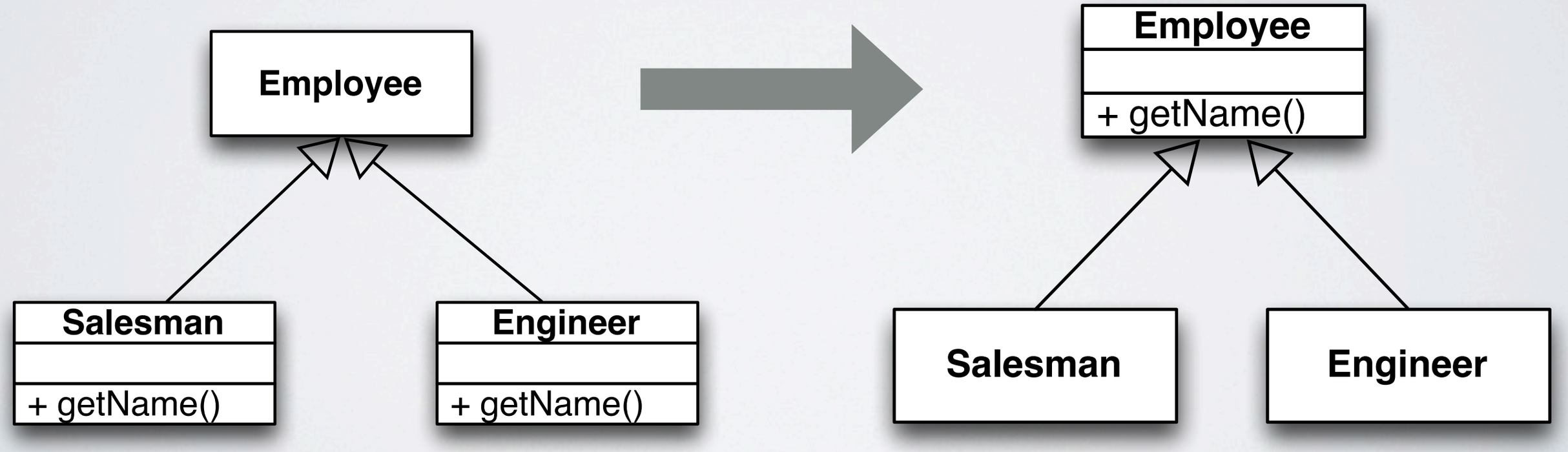
PUSH METHOD DOWN

- Behavior on a superclass is relevant only for some of its subclasses.
- Move it to those subclasses.



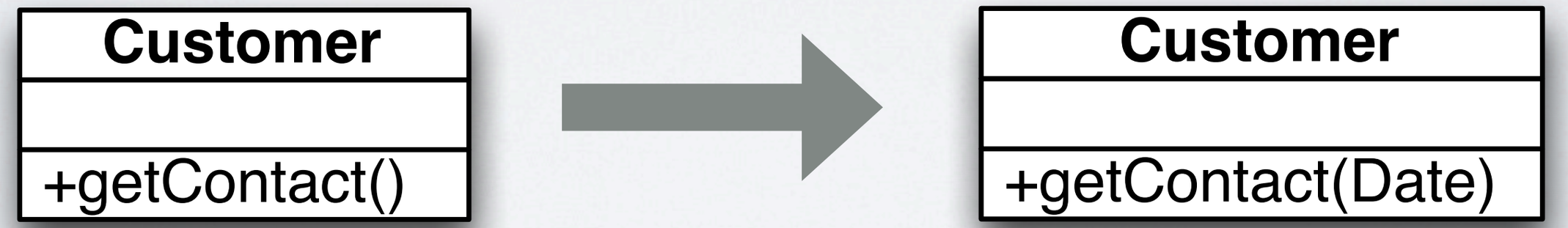
PUSH METHOD UP

- You have methods with identical results on subclasses.
- Move them to the superclass



ADD PARAMETER TO METHOD

- A method needs more information from its caller.
- Add a parameter for an object that can pass on this information.



EXTRACT METHOD

```
void printOwing() {
   printBanner();
   //print details
   System.out.println ("name:    " + _name);
    System.out.println ("amount  " + getOutstanding());
}
```

```
void printOwing() {
    printBanner();
   printDetails(getOutstanding());
}

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

BAD SMELLS IN CODE

If it stinks, change it.

BAD SMELLS IN CODE

- By now you have idea of what refactoring is!
- But, you have no concrete indication on **when** refactoring should be applied!

> **Bad Smells:** *(General Definition) Pieces of code that are wrong (in some sense) and that are ugly to see.* [Fowler09]

BAD SMELLS IN CODE (1)

Symptoms	**Bad Smell Name**
Missing Inheritance or delegation	Duplicated code
Inadequate decomposition	Long Method
Too many responsibility	Large/God Class
Object is missing	Long Parameter List
Missing polymorphism	Type Tests

BAD SMELLS IN CODE (2)

Symptoms	**Bad Smell Name**
Same class changes differently depending on addition	Divergent Change
Small changes affects too many objects	Shotgun Surgery
Method needing too much information from another object	Feature Envy
Data that are always used together	Data Clumps
Changes in one hierarchy require change in another hierarchy	Parallel Inheritance Hierarchy

BAD SMELLS IN CODE (3)

Symptoms	**Bad Smell Name**
Class that does too little	Lazy Class
Class with too many delegating methods	Middle Man
Attributes only used partially under certain circumstances	Temporary Field
Coupled classes, internal representation dependencies	Message Chains
Class containing only accessors	Data Classes

DUPLICATED CODE

Problem: Finding the same code structure in more than one place

(a)

Solution: Perform **Extract Method** and invoke the code from both places

Problem: Having the same expression in two sibling subclasses

(b)

Solution: Perform **Extract Method** in both classes and the **Pull Up Field**

LONG METHOD

Problem: Finding a very long method

(a)

Solution: Perform **Extract Method** and improve responsibilities distribution

Problem: Finding a very long method whose statements operates on different parameters/variables

(b)

Solution: Extract Class

LARGE CLASS

Problem: Finding a class that does too much (too many responsibilities)

Solution: Perform **Extract (Sub)Class** and improve responsibilities distribution

Rule of Thumb:

- When a class is trying to do too much, it often shows up as too many instance variables.

- When a class has too many instance variables, duplicated code cannot be far behind

LONG PARAMETER LIST

Problem: Finding a method that has a very long parameter list

- **[Old School]** Pass everything as a parameter instead of using global data.

Solution: Replace Parameters with Methods or **Introduce Parameter Object**

```
public void marry(String name, int age,
                  boolean gender, String name2,
                  int age2, boolean gender2) {...}
```

DIVERGENT CHANGE

Problem: Divergent change occurs when one class is commonly changed in different ways for different reasons

- Violation of **Separation of Concerns** principle

Solution: identify everything that changes for a particular cause and use **Extract Class** to put them all together

SHOTGUN SURGERY

Problem: Every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.

- When the changes are all over the place they are hard to find, and its easy to miss an important change.
- Results of Copy&Paste programming

Solution: Use **Move Method** and **Move Field** to put all the changes in a single class.

FEATURE ENVY

Problem: Finding a (part of a) method that makes heavy use of data and methods from another class

A classic [code] smell is a method that seems more interested in a class other than the one it is in. The most common focus of the envy is the data. [Fowler02]

Solution: Use **Move/Extract Method** to put it in the more desired class.

DATA CLUMPS

Problem: Finding same three or four data items together in lots of places (**Fields in a couple of classes**, **Parameters in many method signatures**)

Solution (1): look for where the clumps appear as fields and use **Extract Class** to turn the clumps into an object

Solution (2): For method parameters use **Introduce Parameter Object** to slim them down

PRIMITIVE OBSESSION

People new to objects are sometimes reluctant to use small objects for small tasks, such as money classes that combine numbers and currency ranges with an upper and lower.

Problem: Your primitive needs any additional data or behavior.

Solution: Use **Extract Class** to turn primitives into a Class.

TYPE TESTS

Problem: Finding a `switch` statement checking the type of an instance object (no Polymorphism).

> *The essence of polymorphism is that instead of asking an object what type it is and then invoking some behavior based on the answer, you just invoke the behavior.*

Solution: Use **Extract Method** to extract the switch statement and then **Move Method** to get it into the class where the polymorphism is needed

PARALLEL INHERITANCE HIERARCHIES

Problem: every time you make a subclass of one class, you have to make a subclass of another (special case of **Shotgun Surgery**).

Solution: Use **Move Method** and **Move Field** to collapse pairwise features and remove duplications.

LAZY CLASS

Problem: Finding a class that is not doing "enough".

Solution: If you have subclasses that are not doing enough try to use **Collapse Hierarchy** and nearly useless components should be subjected to **Inline Class**

SPECULATIVE GENERALIZATION

Problem: You get this smell when someone says "I think we need the ability to do this someday" and you need all sorts of hooks and special cases to handle things that are not required.

Solution:

(1) If you have abstract classes that are not doing enough then use **Collapse Hierarchy**

(2) Unnecessary delegation can be removed with **Inline Class**

(3) Methods with unused parameters should be subject to **Remove Parameter**

(4) Methods named with odd abstract names should be repaired with **Rename Method**

TEMPORARY FIELD

Problem: Finding an object in which an instance variable is set only in certain circumstances.
(We usually expect an object to use all of its variables)

Solution: Use **Extract Class** to create a home for these orphan variables by putting all the code that uses the variable into the component.

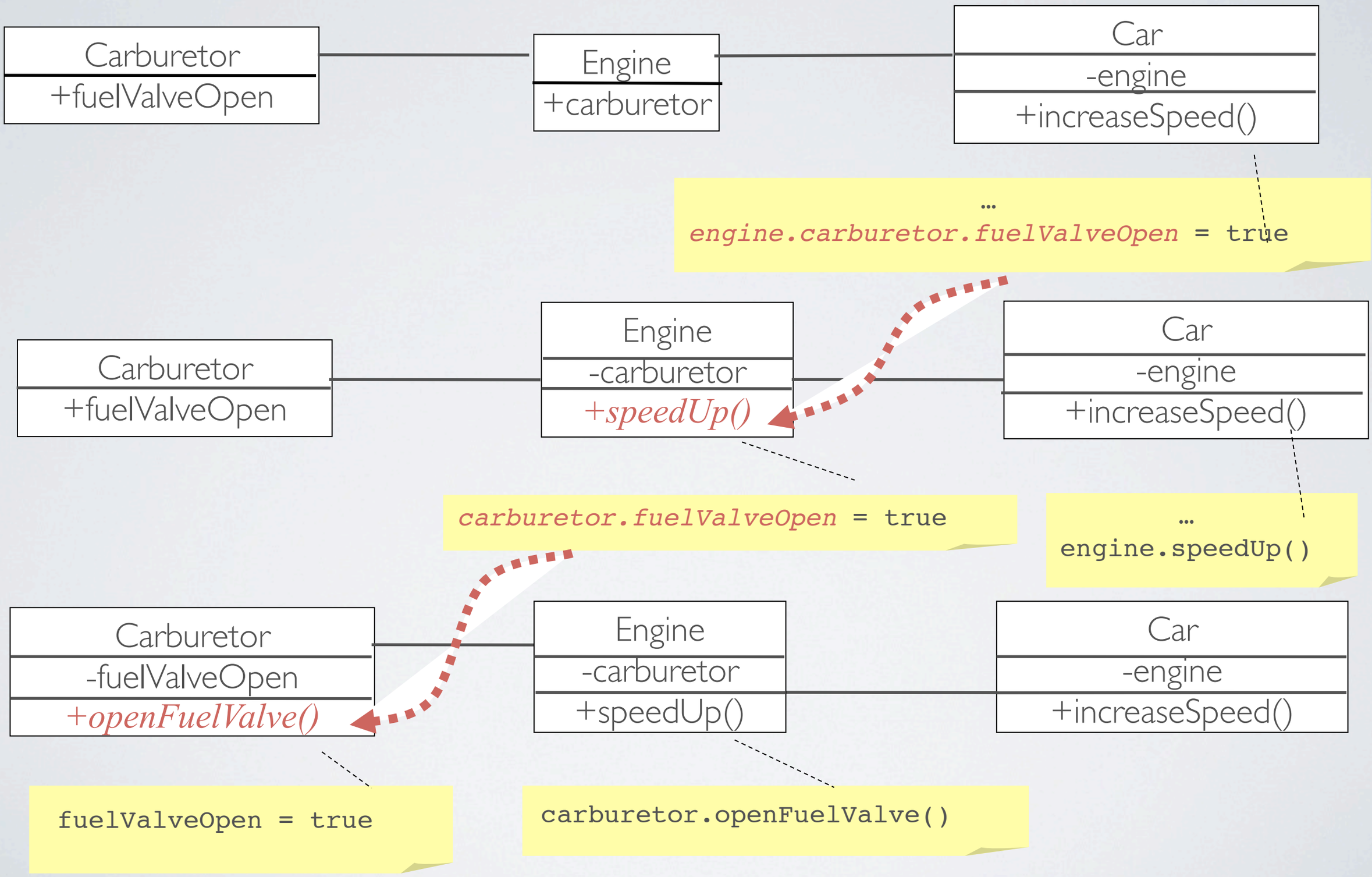
MESSAGE CHAINS

Problem: Message chains occur when you see a client that asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another object, etc.

```
intermediate.getProvider().doSomething()
```

```
ch = vehicle->getChassis();
body = ch->getBody();
shell = body->getShell();
material = shell->getMaterial();
props = material->getProperties();
color = props->getColor();
```

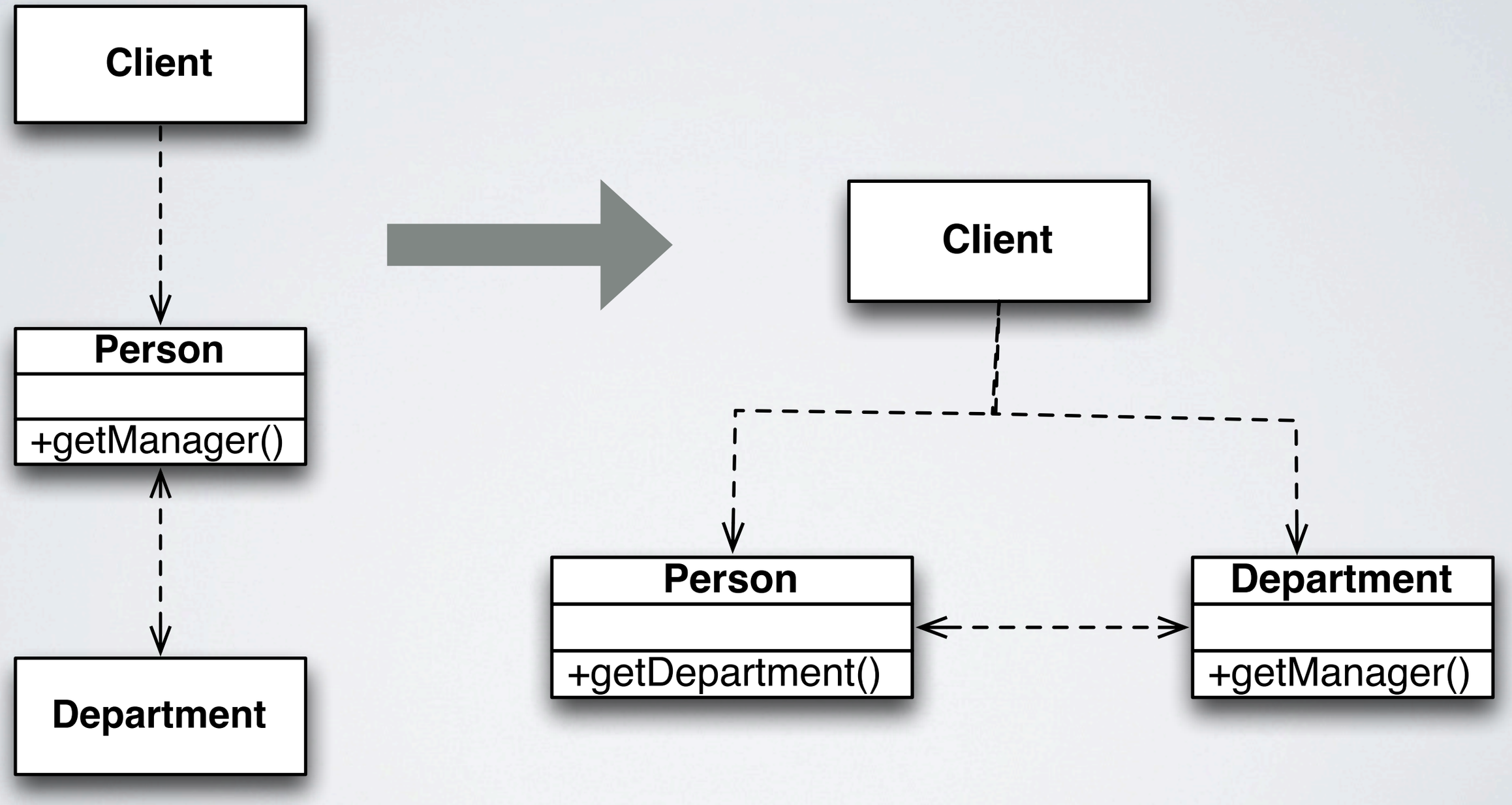
ELIMINATE NAVIGATION CODE



MIDDLE MAN

- One the major features of Objects is encapsulation
- Encapsulation often comes with delegation
- Sometimes delegation can go to far
- For example if you find half the methods are delegated to another class it might be time to use Remove Middle Man and talk to the object that really knows what is going on
- If only a few methods are not doing much, use Inline Method to inline them into the caller
- If there is additional behavior, you can use Replace Delegation with Inheritance to turn the middle man into a subclass of the real object

REMOVE MIDDLE MAN



WHAT'S NEXT?

- In the next class, we will:

  a. Analyze together a (quite) real case study.

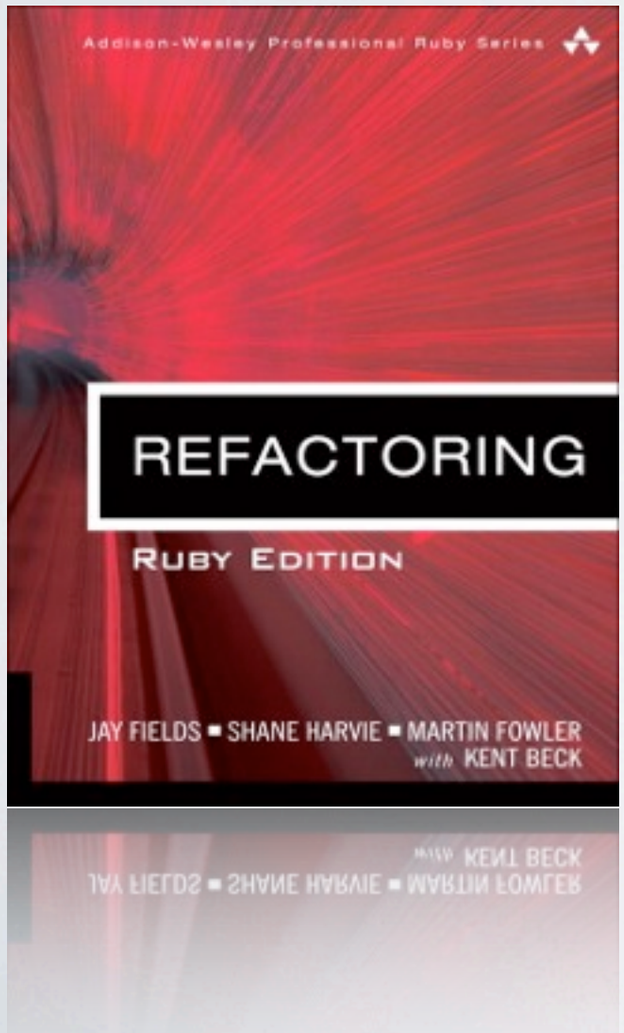
  b. Focus on the design of each (Java) class

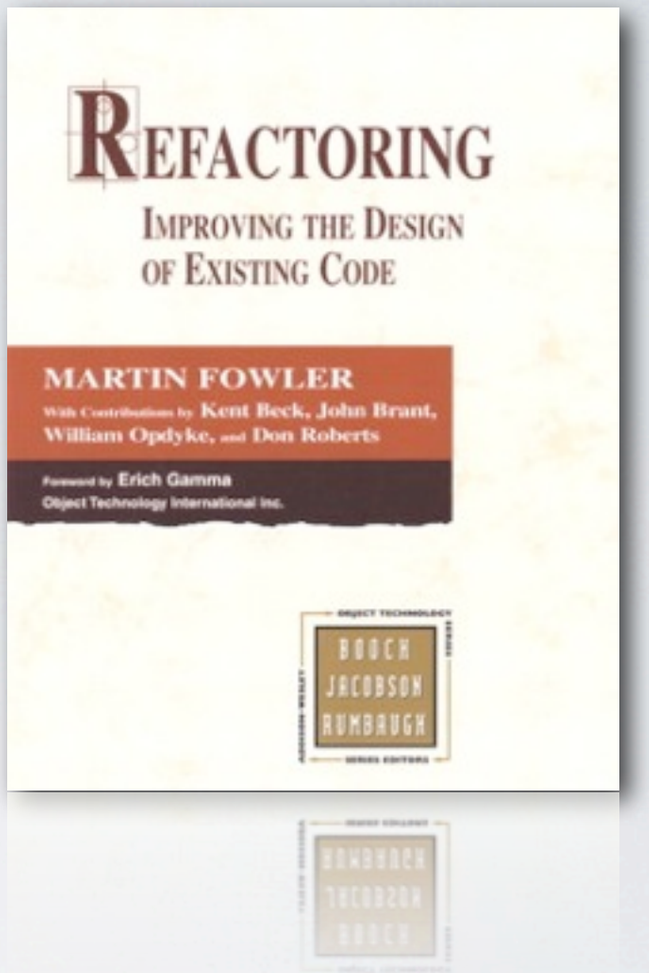
  c. Apply some Refactoring directly into an IDE

    - to understand what "Refactoring" means in practice
    - to see Refactoring features embedded into Eclipse

REFERENCES

- [Fowler02] M. Fowler, K. Beck; *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 2002





- [Fowler09] J. Fields, S. Harvie, M.Fowler, K. Beck; *Refactoring in Ruby*, Addison Wesley, 2009

- http://martinfowler.com/refactoring/catalog/index.html