Patterns in Software Engineering

Lecturer: Raman Ramsin

Lecture 10

Refactoring Patterns

<u>Part 1</u>

Refactoring: Definition

- ***Refactoring:***
  - A change made to the internal structure of software to make it
    - easier to understand, and
    - cheaper to modify.
  - *The observable behavior of the software should not be changed.*

Refactoring: Why?

- **Why Should You Refactor?**
  - Refactoring Improves the Design of Software
  - Refactoring Makes Software Easier to Understand
  - Refactoring Helps You Find Bugs
  - Refactoring Helps You Program Faster

Refactoring: When?

- **When Should You Refactor?**
  - Refactor the third time you do something similar (The Rule of Three)
  - Refactor When You Add Function
  - Refactor When You Need to Fix a Bug
  - Refactor As You Do a Code Review

Symptoms of Bad Code (1)

1. **Duplicated Code**
2. **Long Method**
3. **Large Class**
4. **Long Parameter List**
5. **Divergent Change:** When one class is commonly changed in different ways for different reasons.
6. **Shotgun Surgery:** When every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.
7. **Feature Envy:** A method that seems more interested in a class other than the one it actually is in.
8. **Data Clumps:** Bunches of data that regularly appear together.

Symptoms of Bad Code (2)

9. **Primitive Obsession:** Excessive use of primitives, due to reluctance to use small objects for small tasks.
10. **Switch Statements**
11. **Parallel Inheritance Hierarchies:** Where every time you make a subclass of one class, you also have to make a subclass of another.
12. **Lazy Class:** A class that isn't doing enough to justify its maintenance.
13. **Speculative Generality:** Classes and features have been added just because a need for them may arise someday.
14. **Temporary Field:** An instance variable that is set only in certain circumstances.
15. **Message Chains:** Transitive visibility chains.

Symptoms of Bad Code (3)

16. **Middle Man:** Excessive delegation.
17. **Inappropriate Intimacy:** Excessive interaction and coupling.
18. **Alternative Classes with Different Interfaces:** Classes that do the same thing but have different interfaces for what they do.
19. **Incomplete Library Class**
20. **Data Class:** Classes that have fields, getting and setting methods for the fields, and nothing else.
21. **Refused Bequest:** When subclasses do not fulfill the commitments of their superclasses.
22. **Comments:** When comments are used to compensate for bad code.

Refactoring Patterns: Categories

- **Composing Methods:** Packaging code properly
- **Moving Features Between Objects:** Reassigning responsibilities
- **Organizing Data:** Making data easier to work with
- **Simplifying Conditional Expressions:** Making conditional logic less error-prone
- **Making Method Calls Simpler:** Making interfaces easy to understand and use
- **Dealing with Generalization:** Moving features around a hierarchy of inheritance
- **Big Refactorings:** Large-scale changes to code

Composing Methods: *Extract Method*

- **Extract Method**
  - You have a code fragment that can be grouped together.
  - *Turn the fragment into a method whose name explains the purpose of the method.*

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

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

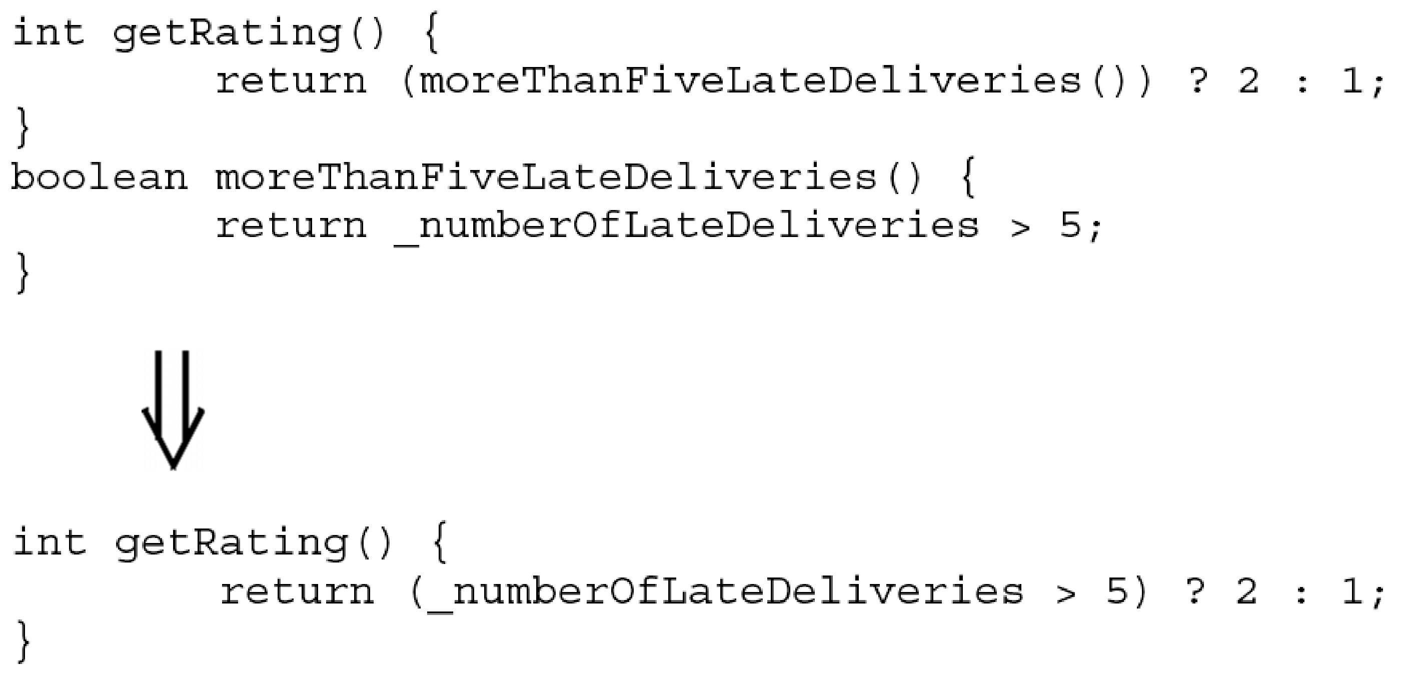
⇓

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

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

Composing Methods: *Inline Method*

- **Inline Method**
  - A method's body is just as clear as its name.
  - *Put the method's body into the body of its callers and remove the method.*



```
int getRating() {
        return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
        return _numberOfLateDeliveries > 5;
}
```

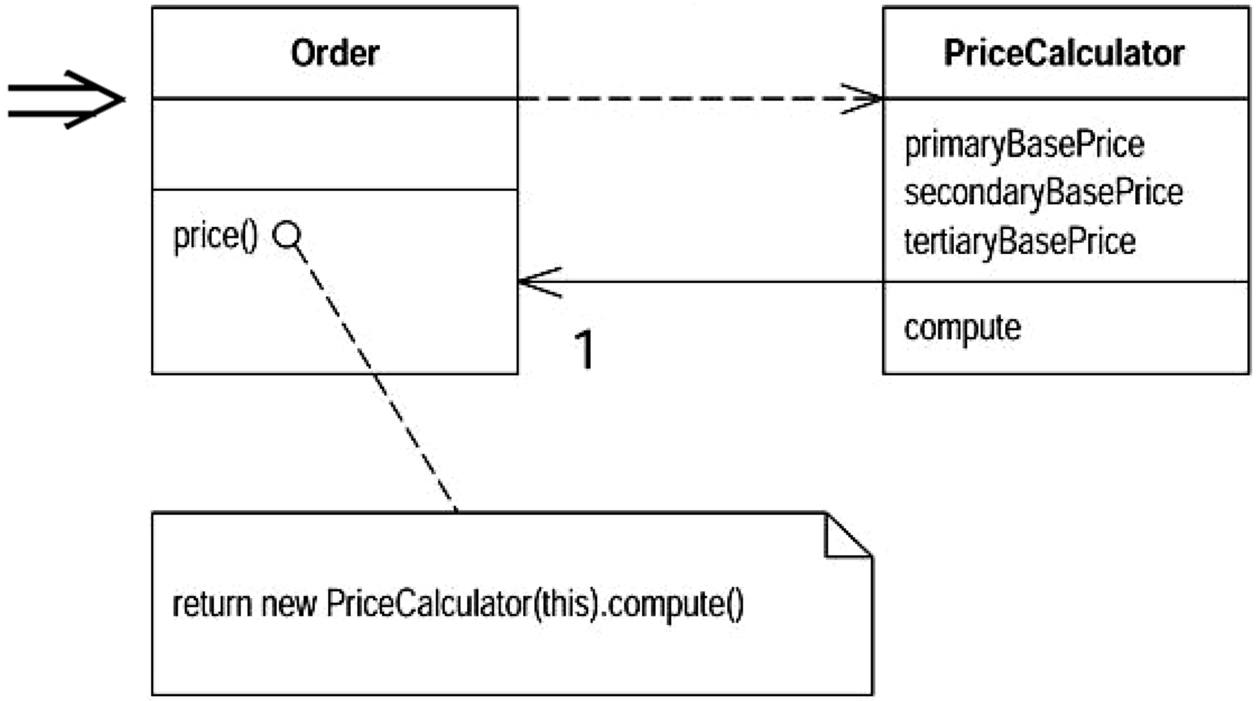
⇓

```
int getRating() {
        return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Composing Methods: *Replace Method with Method Object*

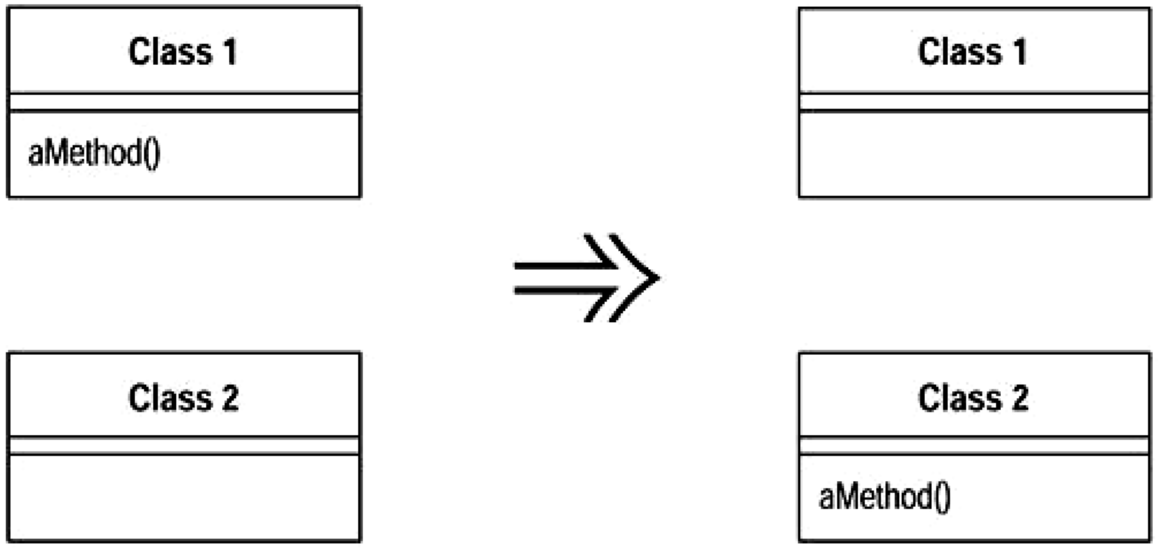
- **Replace Method with Method Object**
  - You have a long method that uses local variables in such a way that you cannot apply *Extract Method*.
  - *Turn the method into an object so that all the local variables become fields on that object. It can then be decomposed into other methods on the same object.*

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



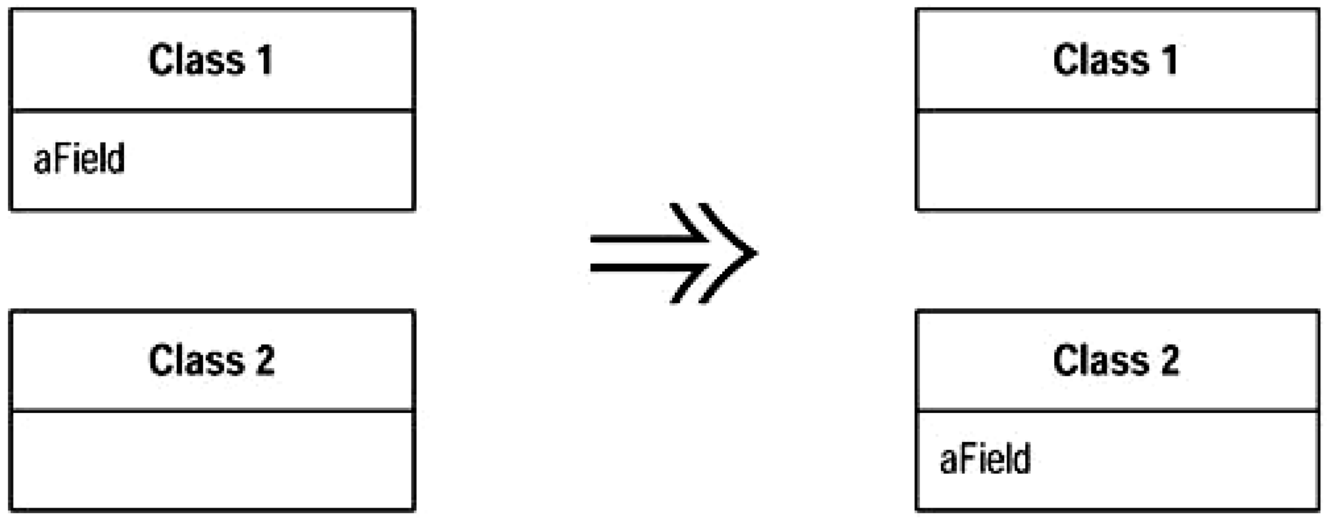
Moving Features Between Objects: *Move Method*

- **Move Method**
  - 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.*



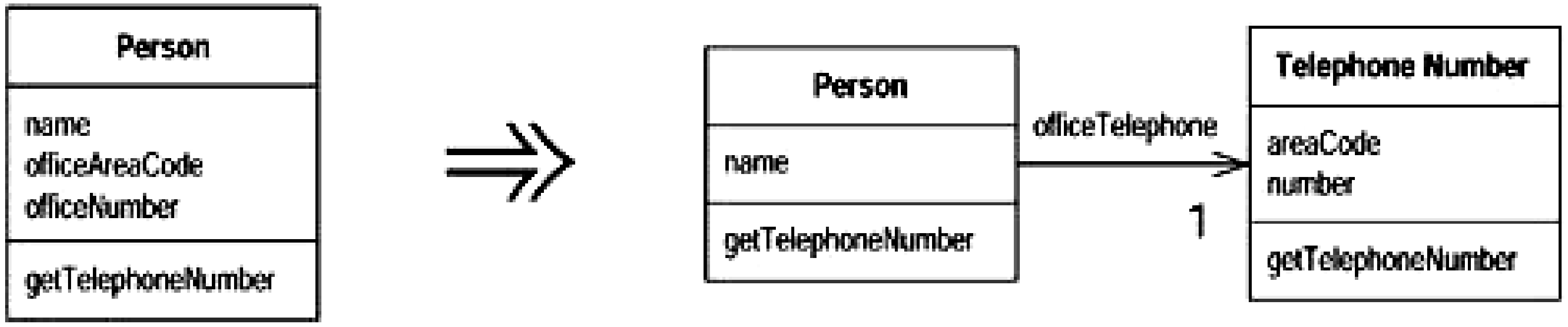
Moving Features Between Objects: *Move Field*

- **Move Field**
  - 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.*



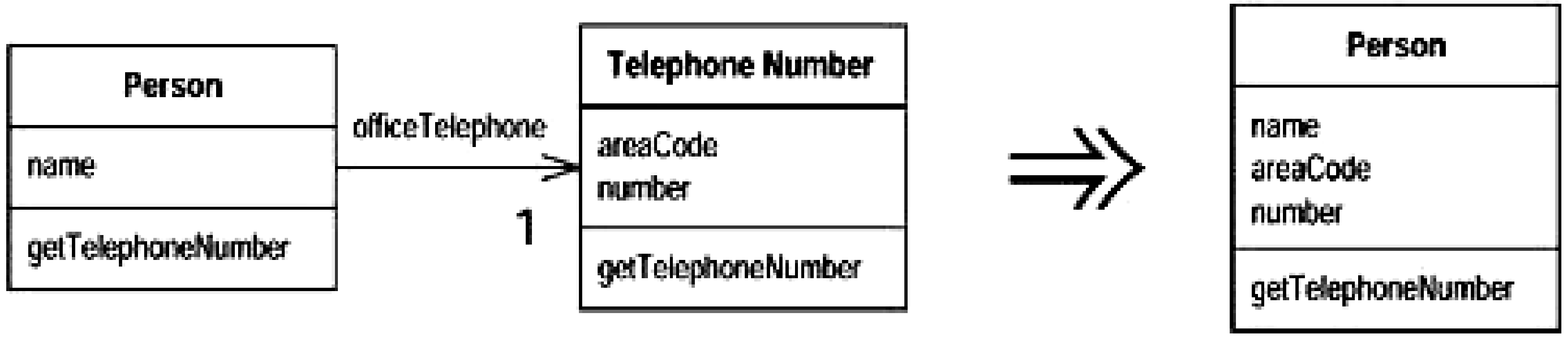
Moving Features Between Objects: *Extract Class*

- **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.*



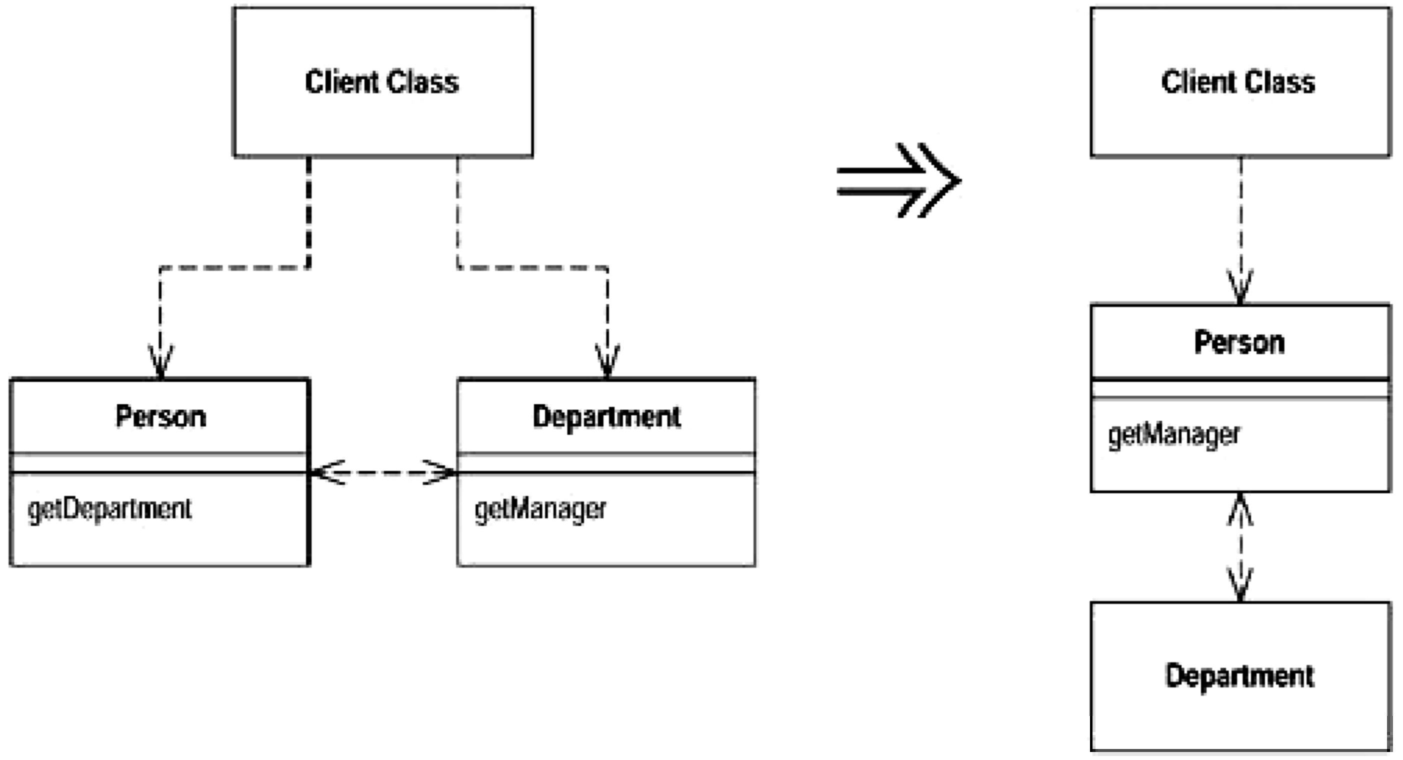
Moving Features Between Objects: *Inline Class*

- **Inline Class**
  - A class isn't doing very much.
  - *Move all its features into another class and delete it.*



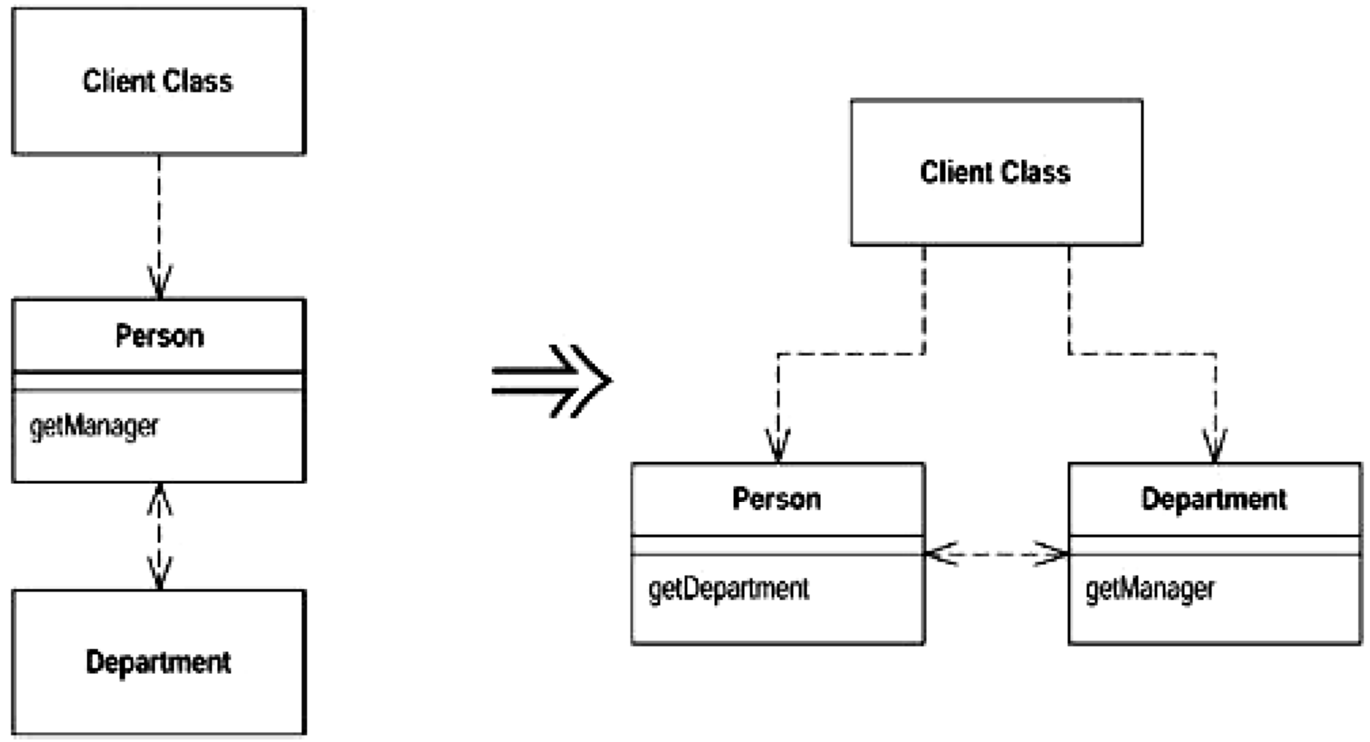
Moving Features Between Objects: *Hide Delegate*

- **Hide Delegate**
  - A client is calling a delegate class of an object.
  - *Create methods on the server to hide the delegate.*



Moving Features Between Objects: *Remove Middle Man*

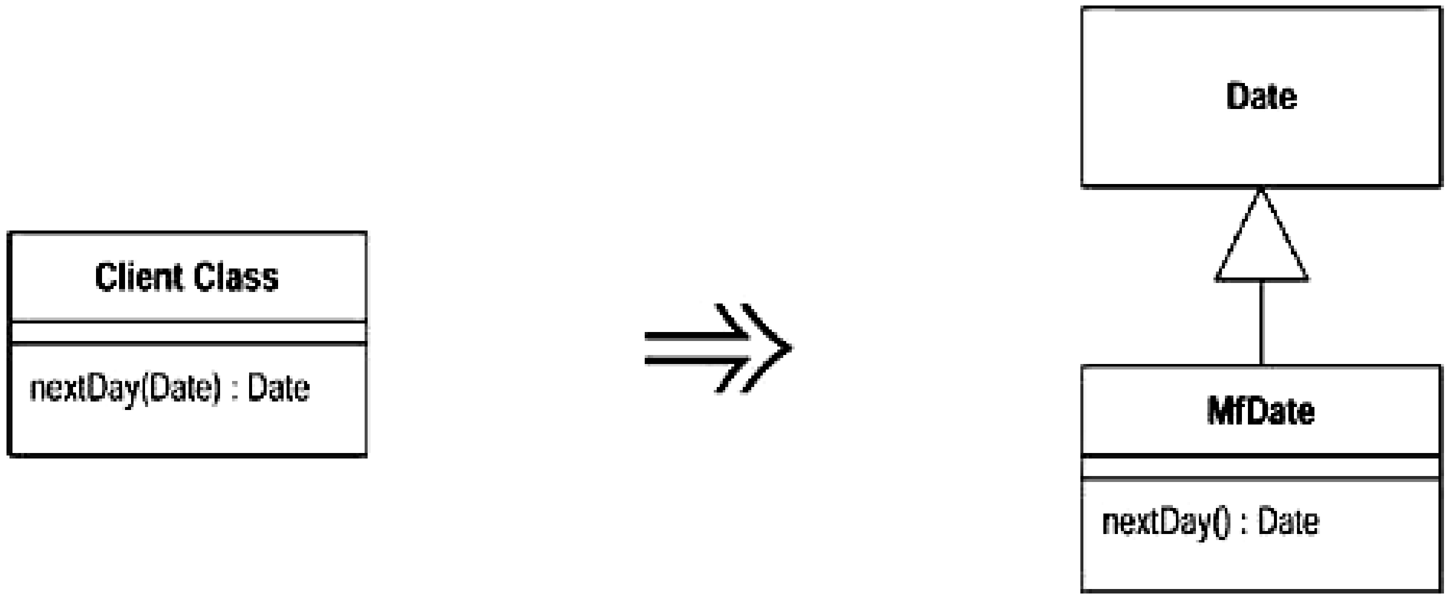
- **Remove Middle Man**
  - A class is doing too much simple delegation.
  - *Get the client to call the delegate directly.*



Moving Features Between Objects: *Introduce Method/Class*

- **Introduce Foreign Method**
  - A server class you are using needs an additional method, but you can't modify the class.
  - *Create a method in the client class with an instance of the server class as its first argument.*
- **Introduce Local Extension**
  - A server class you are using needs several additional methods, but you can't modify the class.
  - *Create a new class that contains these extra methods. Make this extension class a subclass or a wrapper of the original.*

Moving Features Between Objects: *Introduce Local Extension*



Reference

- Fowler, M., *Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999.