

# Refactoring Tips by Martin Fowler

Igor Crvenov

Technical Lead

Redigon Software Solutions

Contact: [Igor.Crvenov@redigon.com](mailto:Igor.Crvenov@redigon.com) , crvenovi@gmail.com

- You write code that tells the computer what to do, and it responds by doing exactly what you tell it.

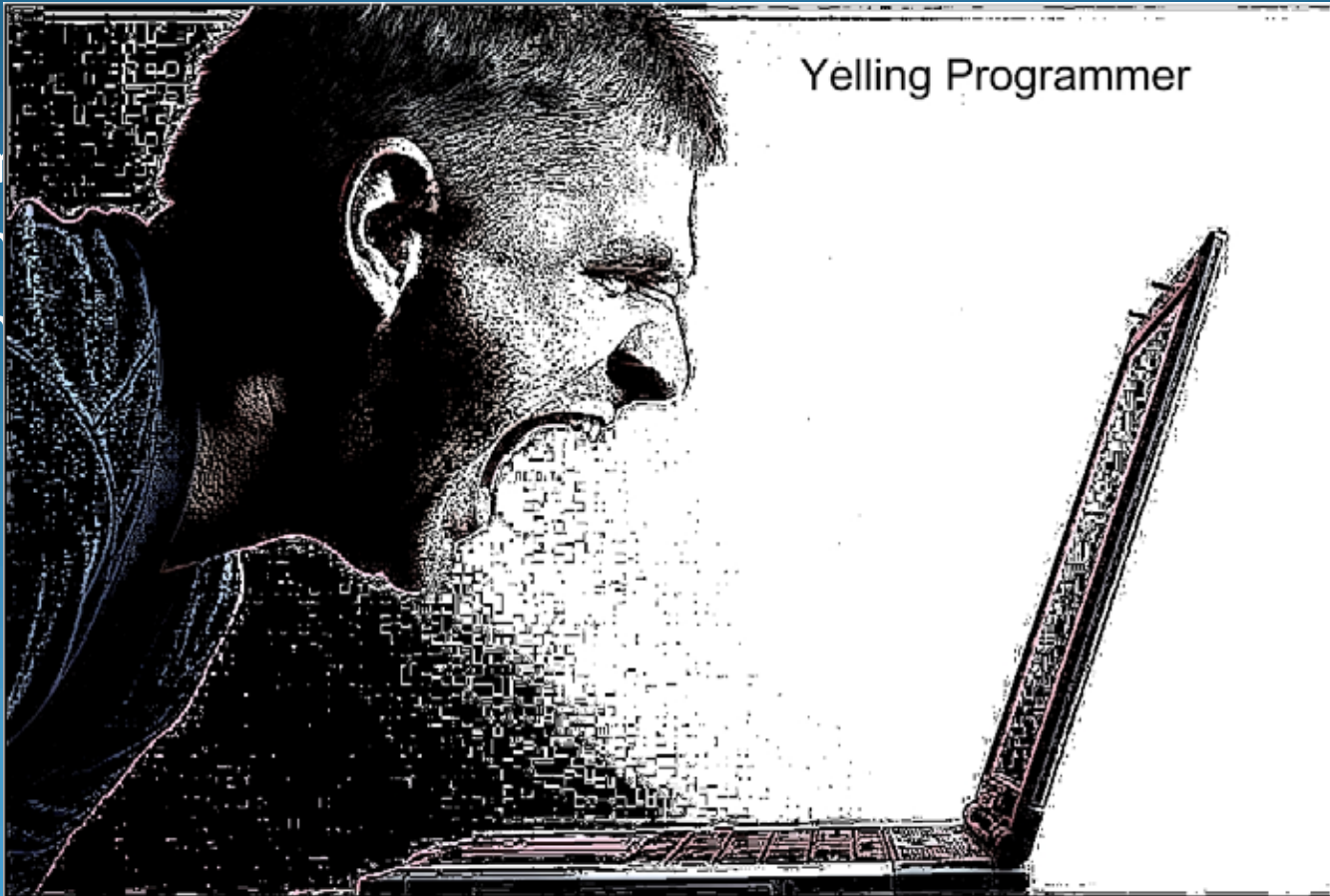
Our code =



- The trouble is that when you are trying to get the program to work, you are not thinking about that future developer.

# What happens when future developer comes ?

- Some  
to n  
to n  
she



s' time  
week  
ur if

# Solution to the spaghetti code problem

Tip:

Any fool can write code that a computer can understand.  
Good programmers write code that humans can understand.

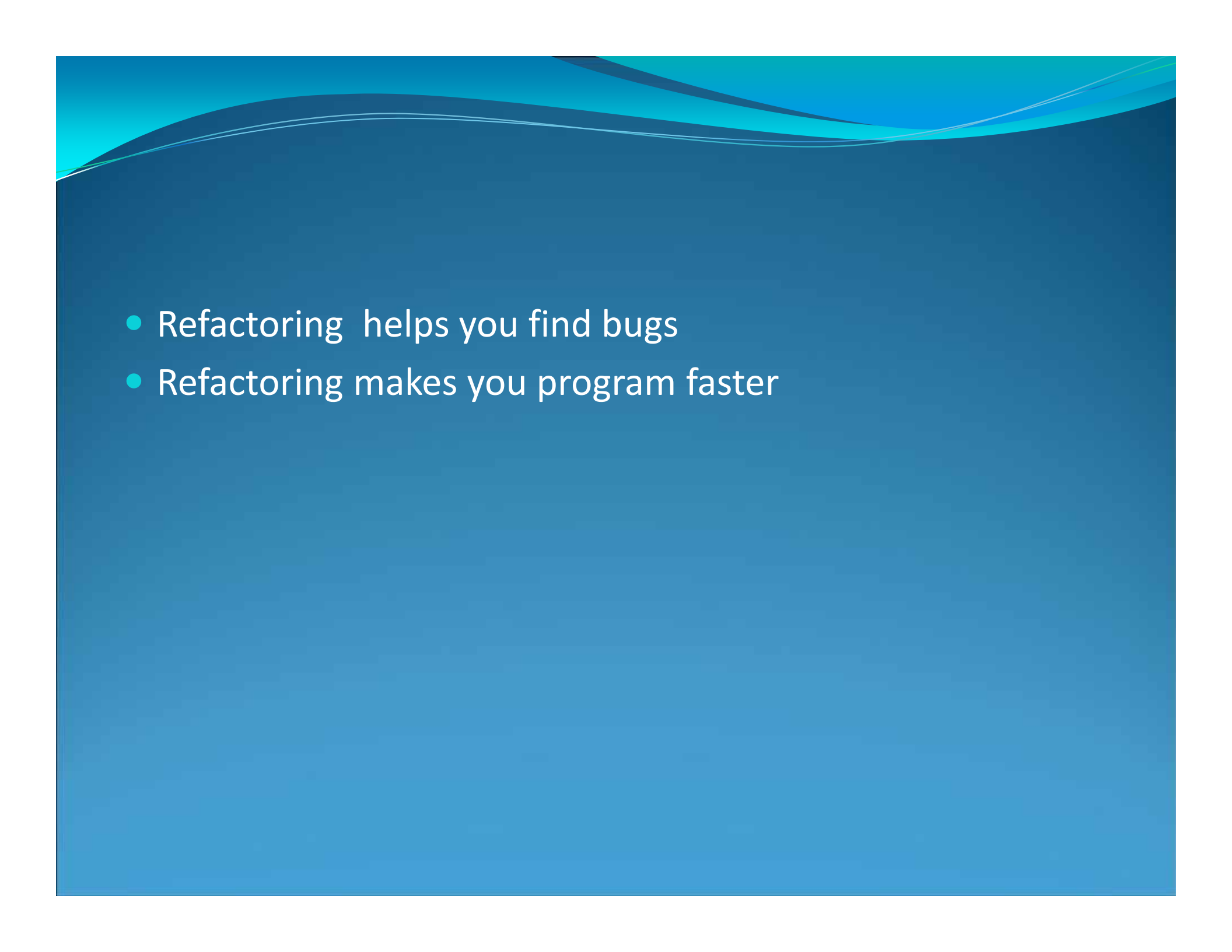
- Refactoring



# What can we do with refactoring

- In essence when you refactor you are improving the design of the code after it has been written, refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand



- 
- Refactoring helps you find bugs
  - Refactoring makes you program faster

# Who is Martin Fowler ?

**Martin Fowler** is an author and international speaker on software development, specializing in object-oriented analysis and design, UML, patterns, and agile software development methodologies, including extreme programming.

Fowler is a member of the *Agile Alliance* and helped create the Manifesto for Agile Software Development in 2001, along with more than 15 co-authors. Martin Fowler was born in Walsall England, and lived in London a decade before moving to United States in 1994.





# Refactoring definitions:

- 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
- 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.



# How should be refactoring scheduled?

- Should we allocate two weeks every couple of months to refactoring?
- M.F: Refactoring is something you do all the time in little bursts. You don't decide to refactor, you refactor because you want to do something else, and refactoring helps you do that other thing.
- **The Rule of Three(Don Roberts)**
  - 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.

Tip: *Three strikes and you refactor.*

# When to refactor ?

- Refactor when you add Function
- Refactor when you need to fix a bug
- Refactor as you do code review

# When you should not refactor?

- There are times when the existing code is such a mess that although you could refactor it, it would be easier to start from the beginning.
- The other time you should avoid refactoring is when you are close to a deadline. At that point the productivity gain from refactoring would appear after the deadline and thus be too late.

# Bad Smells in Code

- Duplicated code
- Long Method
- Large Class
- Long Parameter List

- M.F:Whenever I do refactoring, the first step is always the same. I need to build a solid set of tests for that section of code. The tests are essential because even though I follow refactorings structured to avoid most of the opportunities for introducing bugs, I'm still human and still make mistakes. Thus I need solid tests.

Tip:

Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking

# Refactoring Methods

- Extract Method

- You have a code fragment that can be grouped together. Turn the fragment in to a method whose name explains the purpose of the method.
- Extract Method is one of the most common refactorings I do. I look at a method that is too long or look at code that needs a comment to understand its purpose. I then turn that fragment of code into its own method.

# Extract Method

## Before refactoring

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

    //print details
    WriteLine("name:" + _name);
    WriteLine("amount" + amount);
}
```

## After Refactoring

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

void printDetails(double amount)
{
    WriteLine("name:" + _name);
    WriteLine("amount" + amount);
}
```



# Inline Method

# Inline Method

## Before Refactoring

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

## After Refactoring

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

# Split Temporary Variable

- You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable. Make a separate temporary variable for each assignment.

- Before Refactoring

```
double temp = 2 * (_height + _width);  
WriteLine(temp);  
temp = _height * _width;  
WriteLine(temp);
```

- After Refactoring

```
double perimeter = 2 * (_height + _width);  
WriteLine(perimeter);  
double area = _height * _width;  
WriteLine(area);
```

# Remove Assignments to Parameters

- The code assigns to a parameter.

*Use a temporary variable instead.*

*Before Refactoring:*

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

After refactoring :

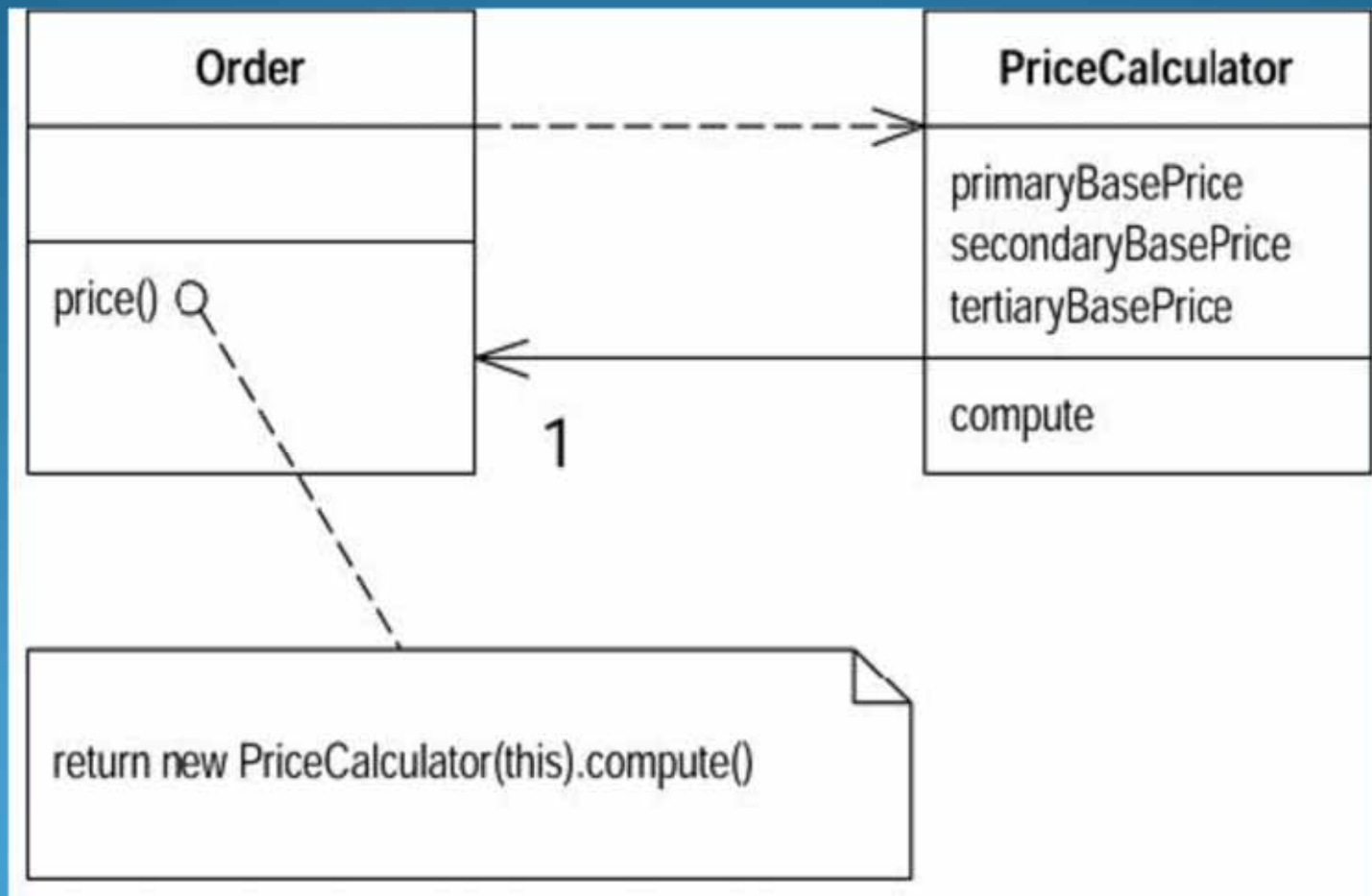
- ```
int discount (int inputVal, int quantity, int yearToDate) {  
    int result = inputVal;  
    if (inputVal > 50) result -= 2;
```

# 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.

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

# Replace Method with Method Object





- Class Account

- ```
int gamma (int inputVal, int quantity, int yearToDate) {  
int importantValue1 = (inputVal * quantity) + delta();  
int importantValue2 = (inputVal * yearToDate) + 100;  
if ((yearToDate - importantValue1) > 100)  
importantValue2 -= 20;  
int importantValue3 = importantValue2 * 7;  
// and so on.  
return importantValue3 - 2 * importantValue1;  
}
```

```
class Gamma...  
private final Account _account;  
private int inputVal;  
private int quantity;  
private int yearToDate;  
private int importantValue1;  
private int importantValue2;  
private int importantValue3;
```

```
Gamma (Account source, int inputValArg, int quantityArg, int  
yearToDateArg) {  
_account = source;  
inputVal = inputValArg;  
quantity = quantityArg;  
yearToDate = yearToDateArg;  
}
```

```
int compute () {  
    importantValue1 = (inputVal * quantity) + _account.delta();  
    importantValue2 = (inputVal * yearToDate) + 100;  
    if ((yearToDate - importantValue1) > 100)  
        importantValue2 -= 20;  
    int importantValue3 = importantValue2 * 7;  
    // and so on.  
    return importantValue3 - 2 * importantValue1;  
}
```

```
int gamma (int inputVal, int quantity, int yearToDate) {  
    return new Gamma(this, inputVal, quantity,  
        yearToDate).compute();  
}
```

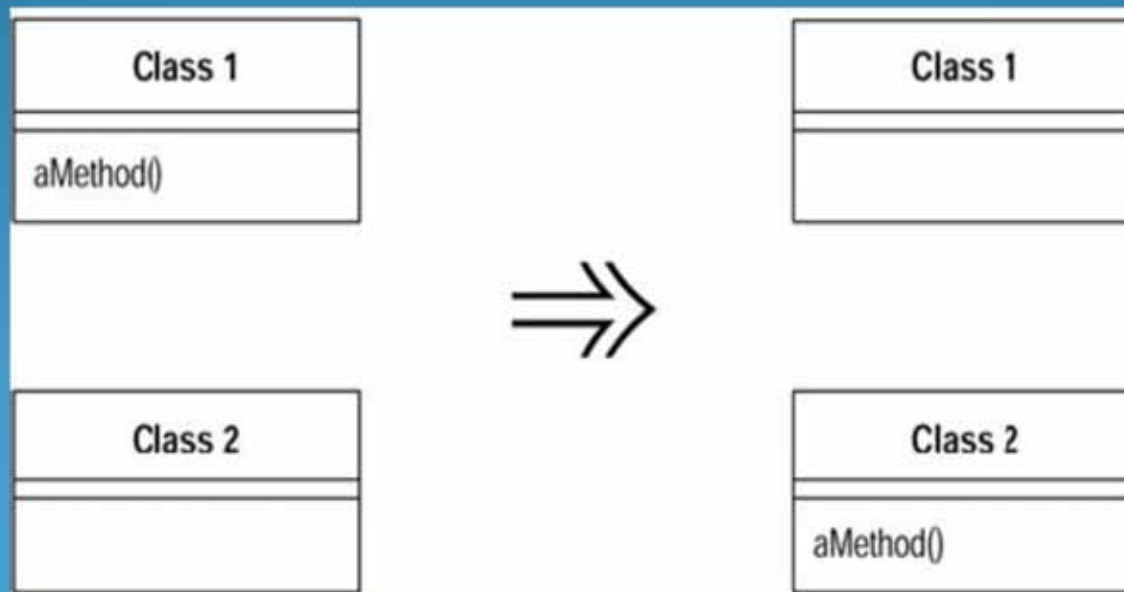
# Moving Features between object

- One of the most fundamental, if not the fundamental, decision in object design is deciding where to put responsibilities.

# 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.

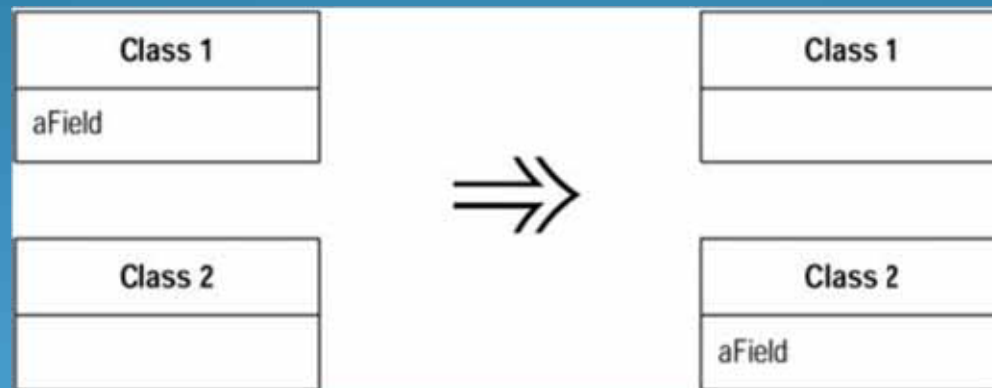


## **Move Method Motivation:**

Moving methods is the bread and butter of refactoring. I move methods when classes have too much behavior or when classes are collaborating too much and are too highly coupled. By moving methods around, I can make the classes simpler and they end up being a more crisp implementation of a set of responsibilities.

# Move Field

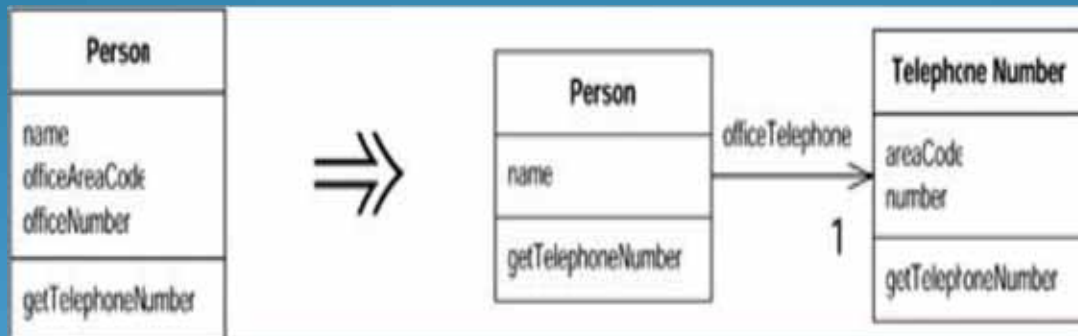
- A field is, or will be, used by another class more than the class on which it is defined. *Create a new field in the target class, and change all its users.*





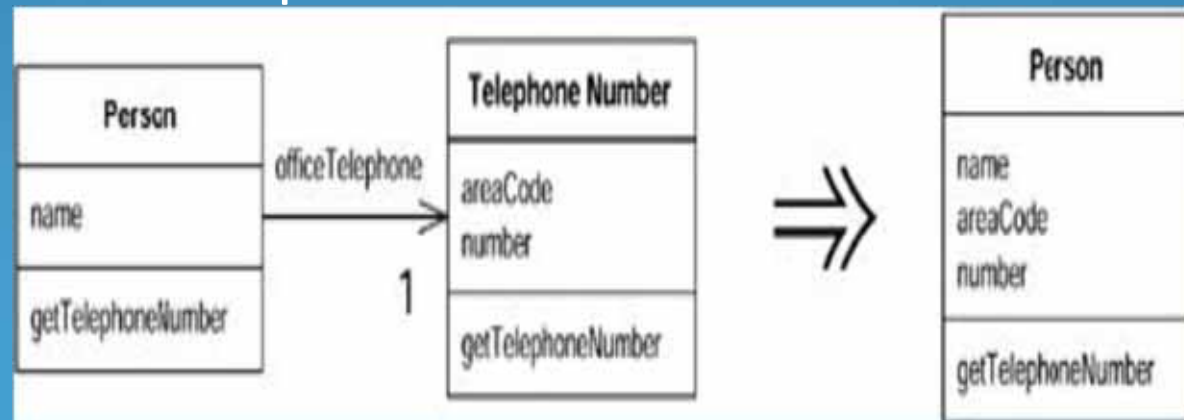
# Extract class

- 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.



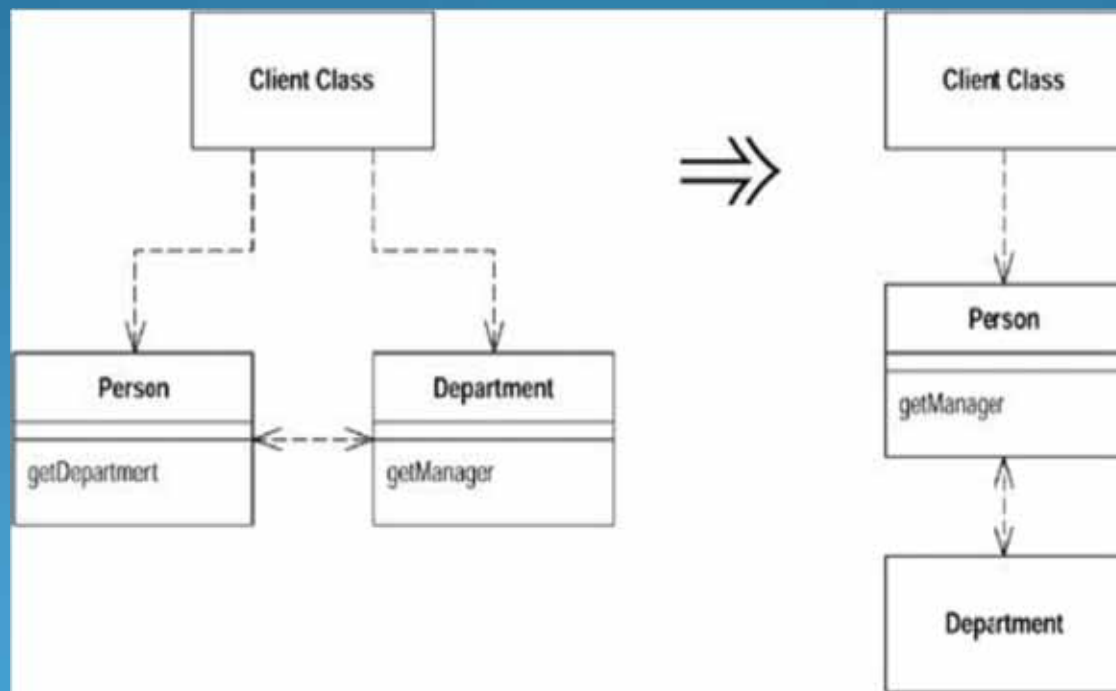
# Inline Class

- A class isn't doing very much. Move all its features into another class and delete it.
- *Inline Class* is the reverse of Extract Class. I use *Inline Class* if a class is no longer pulling its weight and shouldn't be around any more. Often this is the result of refactoring that moves other responsibilities out of the class so there is little left.



# Hide Delegate

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



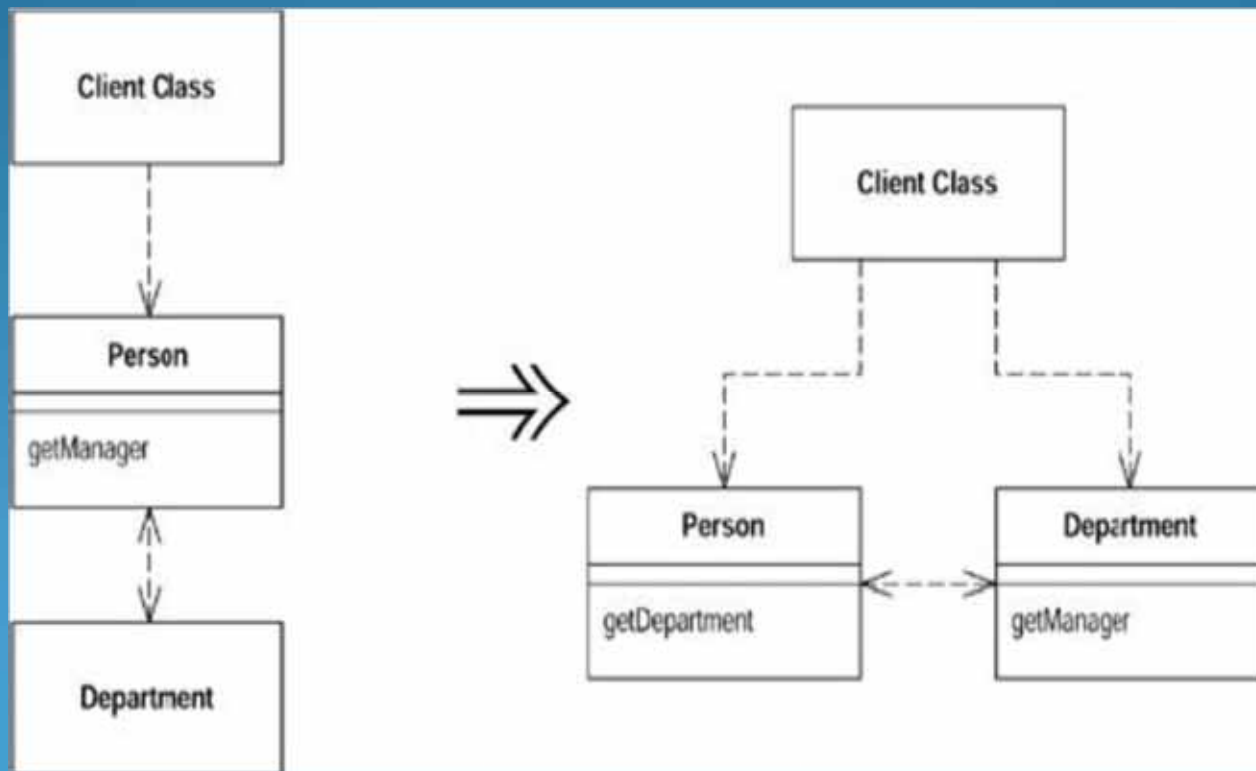
# Hide Delegate

- **Motivation**

- One of the keys, if not *the* key, to objects is encapsulation. Encapsulation means that objects need to know less about other parts of the system. Then when things change, fewer objects need to be told about the change—which makes the change easier to make

# Remove the Middle Man

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



# 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 Foreign Method Example

```
Date newStart = new Date (previousEnd.getYear(),  
previousEnd.getMonth(), previousEnd.getDate() + 1);
```

```
Date newStart = nextDay(previousEnd);  
private static Date nextDay(Date arg) {  
    return new Date (arg.getYear(),arg.getMonth(), arg.getDate() +  
    1);  
}
```

### **Motivation**

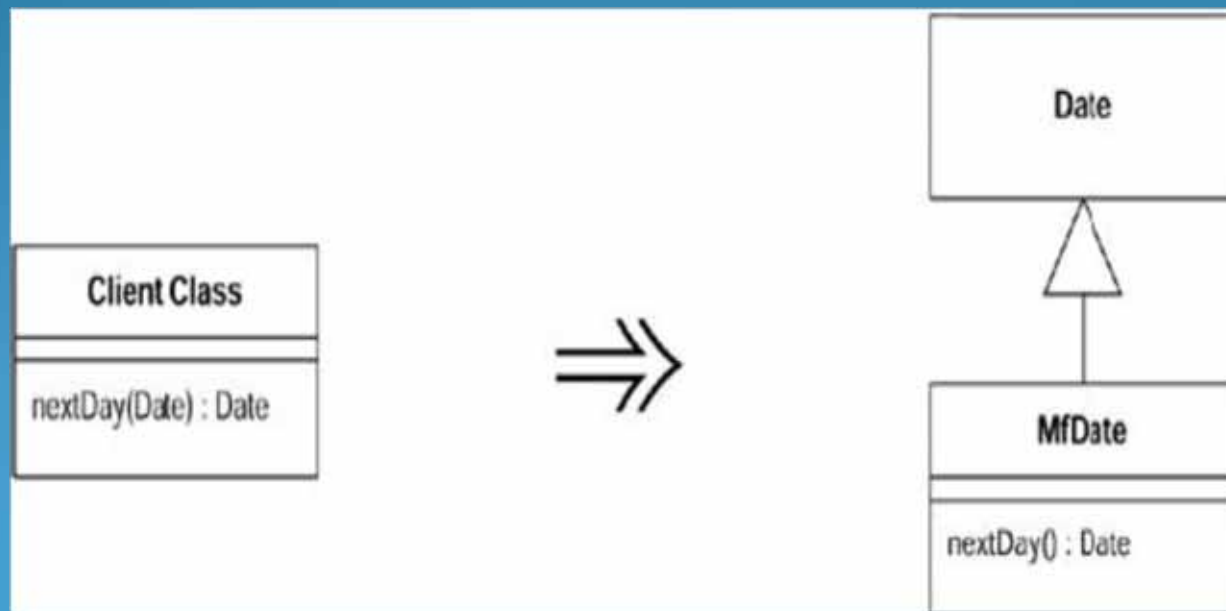
It happens often enough. You are using this really nice class that gives you all these great services. Then there is one service it doesn't give you but should. You curse the class, saying,

"Why don't you do that?" If you can change the source, you can add in the method. If you can't change the source, you have to code around the lack of the method in the client.



# 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



# Motivation

Authors of classes sadly are not omniscient, and they fail to provide useful methods for you. If you can modify the source, often the best thing is to add that method. However, you often cannot modify the source. If you need one or two methods, you can use Introduce Foreign Method. Once you get beyond a couple of these methods, however, they get out of hand. So you need to group the methods together in a sensible place for them. The standard object-oriented techniques of sub classing and wrapping are an obvious way to do this. In these circumstances I call the subclass or wrapper a local extension.

# Self Encapsulated Field

- Create getting and setting methods for the field and use only those to access the field

```
private int _low, _high;  
boolean includes (int arg) {  
    return arg >= _low && arg <= _high;  
}
```

After Refactoring :

```
private int _low, _high;  
boolean includes (int arg) {  
    return arg >= getLow() && arg <= getHigh();  
}  
int getLow() {return _low;}  
int getHigh() {return _high;}
```

## Motivation

When it comes to accessing fields, there are two schools of thought. One is that within the class

where the variable is defined, you should access the variable freely (direct variable access). The

other school is that even within the class, you should always use accessors (indirect variable

access). Debates between the two can be heated. (See also the discussion in Auer [Auer] on

page 413 and Beck [Beck].)

Essentially the advantages of *indirect variable access* are that it allows a subclass to override

how to get that information with a method and that it supports more flexibility in managing the

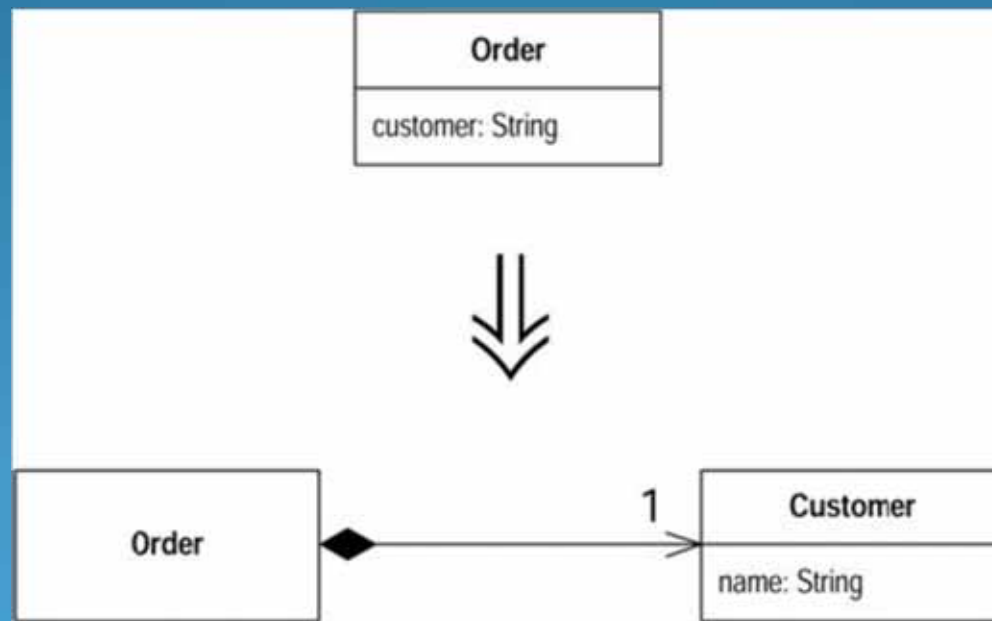
data, such as lazy initialization, which initializes the value only when you need to use it.

The advantage of *direct variable access* is that the code is easier to read. You don't need to stop

and say, "This is just a getting method."

# Replace Data with value object

- You have a data item that needs additional data or behavior. *Turn the data item into an object.*



# Replace Array with object

- You have an array in which certain elements mean different things. *Replace the array with an object that has a field for each element.*

```
String[] row = new String[3];  
row [0] = "Liverpool";  
row [1] = "15";
```

After Refactoring:

```
Performance row = new Performance();  
row.setName("Liverpool");  
row.setWins("15");
```

# Replace Magic Number with symbolic Constant

- You have a literal number with a particular meaning.  
*Create a constant, name it after the meaning, and replace the number with it.*

```
double potentialEnergy(double mass, double height) {  
    return mass * 9.81 * height;  
}
```

```
double potentialEnergy(double mass, double height) {  
    return mass * GRAVITATIONAL_CONSTANT * height;  
}
```

```
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

# Encapsulate Field

- There is a public field. *Make it private and provide accessors.*

```
public String _name
```

After Refactoring:

```
private String _name;
```

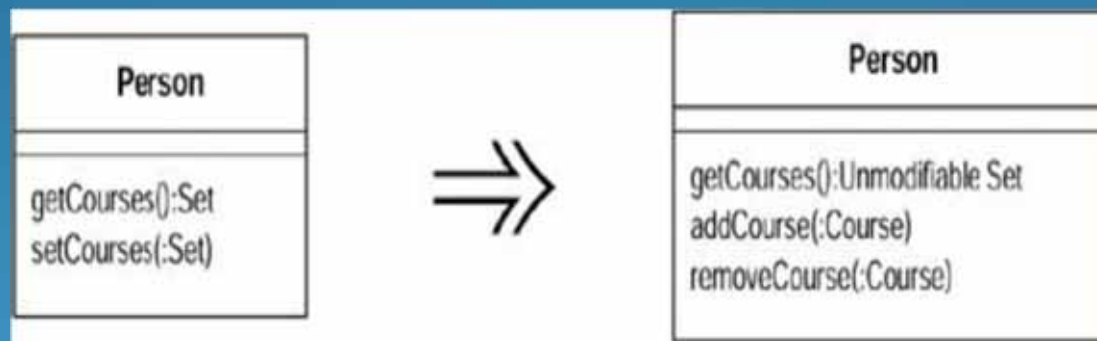
```
public String getName() {return _name;}
```

```
public void setName(String arg) {_name = arg;}
```



# Encapsulate Collection

- A method returns a collection. *Make it return a read-only view and provide add/remove methods.*



## Motivation

Often a class contains a collection of instances. This collection might be an array, list, set, or vector. Such cases often have the usual getter and setter for the collection.

# Decompose Conditional

- 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;
```

After Refactoring:

```
if (notSummer(date))
```

```
    charge = winterCharge(quantity);
```

```
else
```

```
    charge = summerCharge (quantity);
```

# Consolidate Conditional Expression

- You have a sequence of conditional tests with the same result.  
*Combine them into a single conditional expression and extract it.*

```
double disabilityAmount() {  
    if (_seniority < 2) return 0;  
    if (_monthsDisabled > 12) return 0;  
    if (_isPartTime) return 0;
```

After Refactoring:

```
// compute the disability amount  
double disabilityAmount() {  
    if (isNotEligibleForDisability()) return 0;
```

# Consolidate Duplicate Conditional Fragments

- The same fragment of code is in all branches of a conditional expression. *Move it outside of the expression*

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
    send();  
}  
else {  
    total = price * 0.98;  
    send();  
}
```

After Refactoring

```
if (isSpecialDeal())  
    total = price * 0.95;  
else  
    total = price * 0.98;  
send();
```

# Remove Control Flag

- You have a variable that is acting as a control flag for a series of boolean expressions. *Use a break or return instead*

# Replace Nested Conditional with Guard Classes

- A method has conditional behavior that does not make clear the normal path of execution. *Use guard clauses for all the special cases*

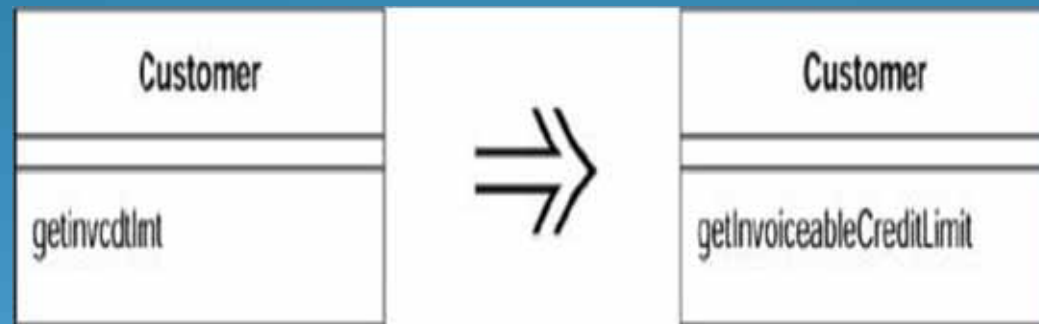
```
double getPayAmount() {  
    double result;  
    if (_isDead) result = deadAmount();  
    else  
    {  
        if (_isSeparated) result = separatedAmount();  
        else {  
            if (_isRetired) result = retiredAmount();  
            else result = normalPayAmount();  
        };  
    }  
    return result;  
};
```

After Refactoring :

```
double getPayAmount() {  
    if (_isDead) return deadAmount();  
    if (_isSeparated) return separatedAmount();  
    if (_isRetired) return retiredAmount();  
    return normalPayAmount();  
};
```

# Rename Method

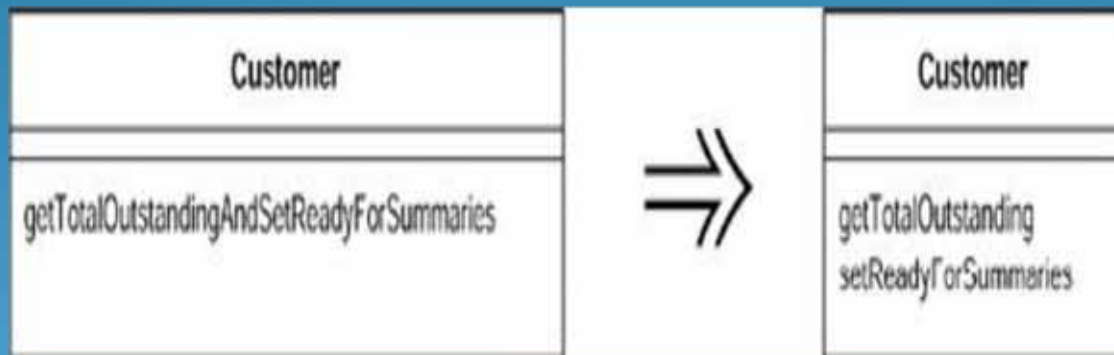
- The name of a method does not reveal its purpose.  
*Change the name of the method*





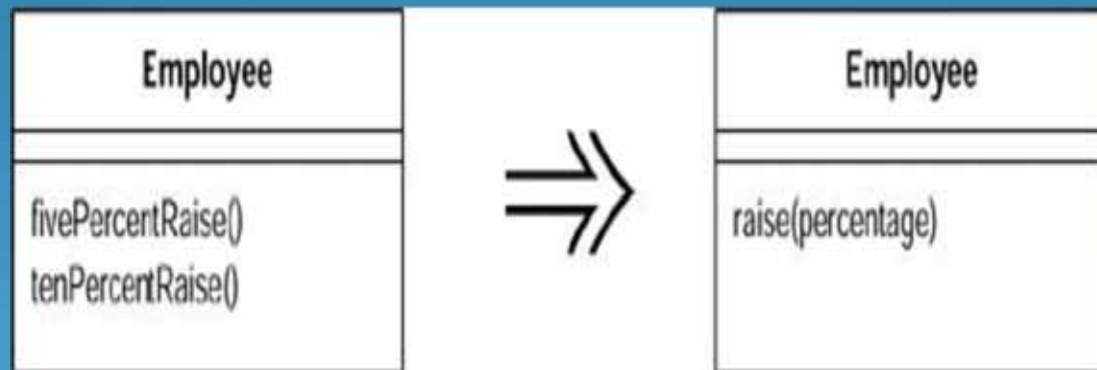
# Separate Query from Modifier

- You have a method that returns a value but also changes the state of an object. *Create two methods, one for the query and one for the modification.*



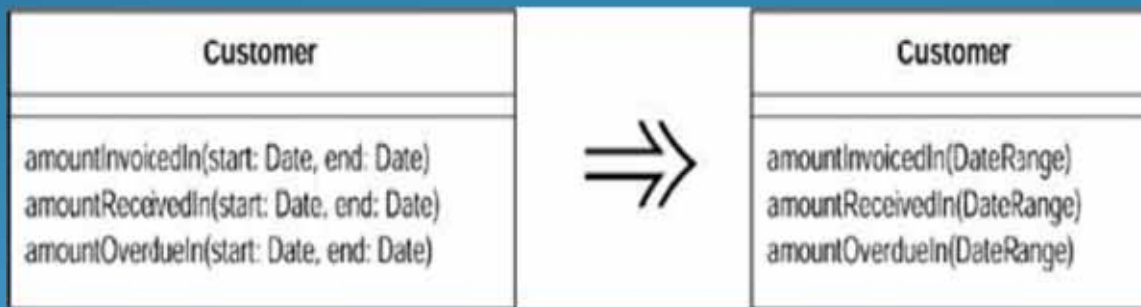
# Parameterized Method

- Several methods do similar things but with different values contained in the method body. *Create one method that uses a parameter for the different values.*



# Introduce Parameter to Object

- You have a group of parameters that naturally go together. *Replace them with an object.*



# Remove Setting Methods

- A field should be set at creation time and never altered. *Remove any setting method for that field.*



- Providing a setting method indicates that a field may be changed. If you don't want that field to change once the object is created, then don't provide a setting method (and make the field final). That way your intention is clear and you often remove the very possibility that the field will change.

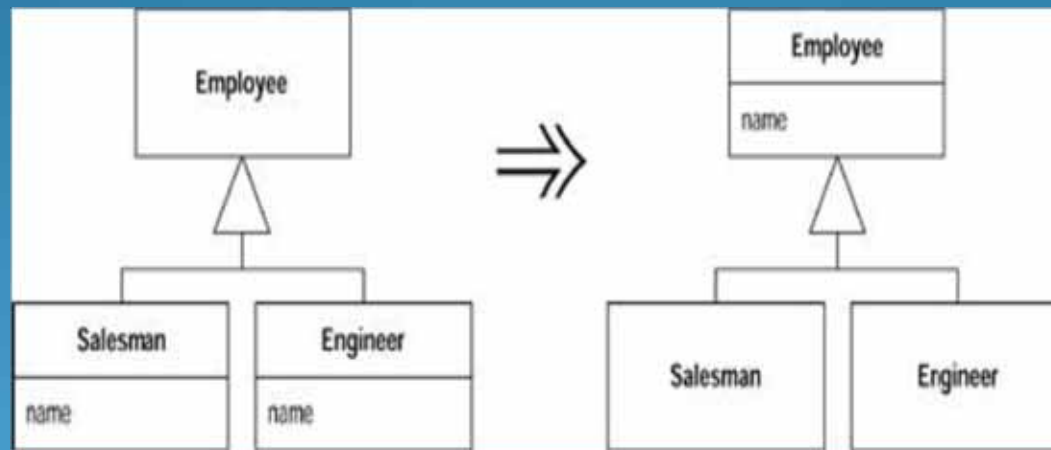
# Hide Method

- A method is not used by any other class. *Make the method private*



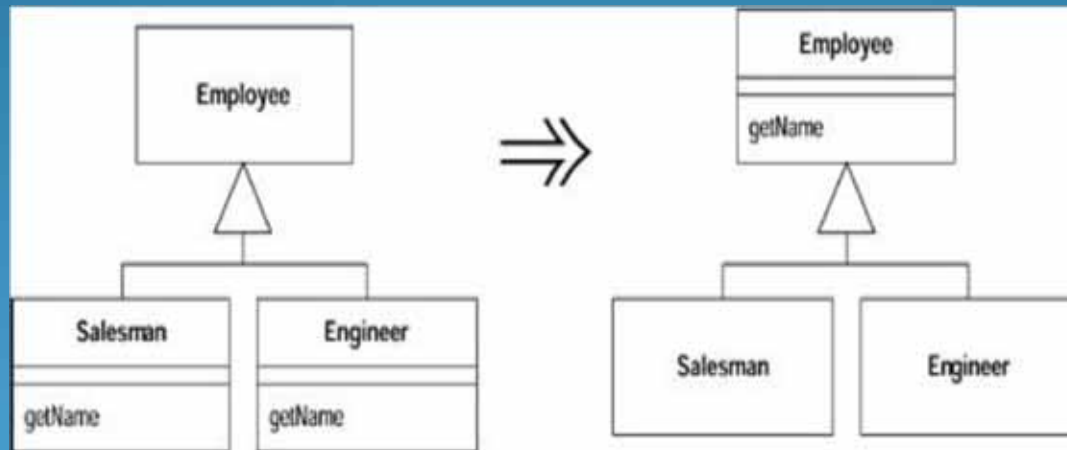
# Pull up Field

- Two subclasses have the same field. *Move the field to the superclass.*



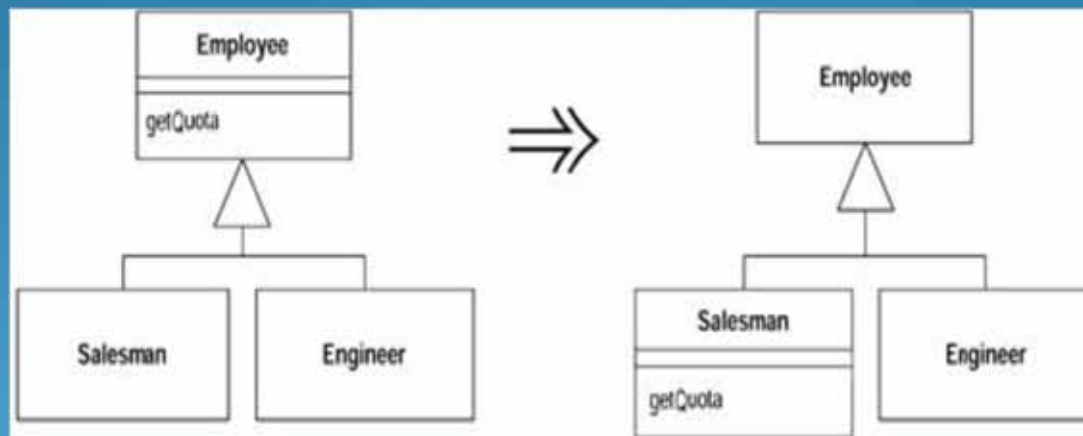
# Pull up Method

- Two subclasses have the same field. *Move the field to the superclass.*



# Push Down Method

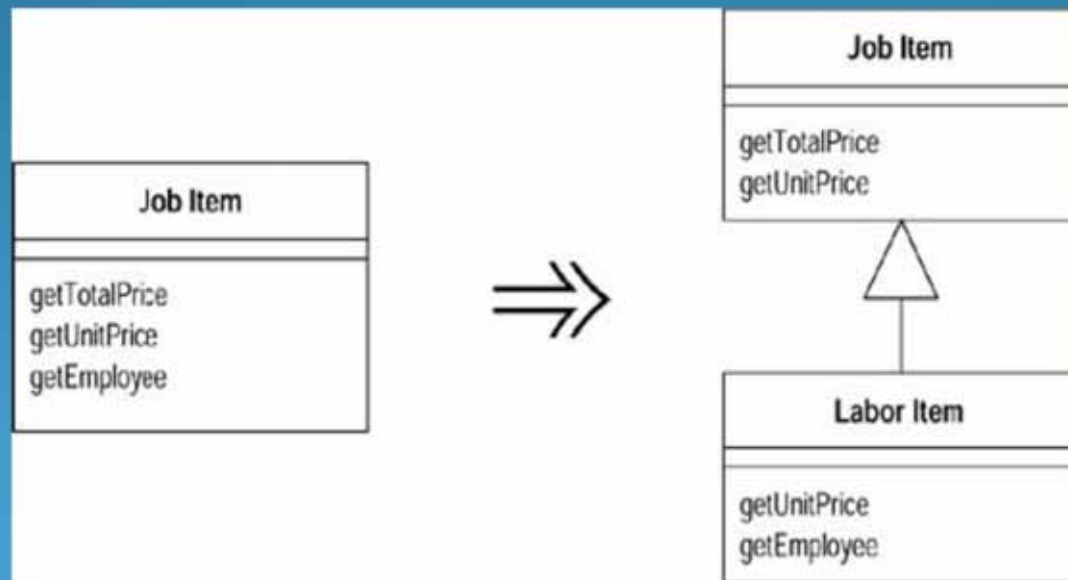
- Behavior on a superclass is relevant only for some of its subclasses. Move it to those subclasses





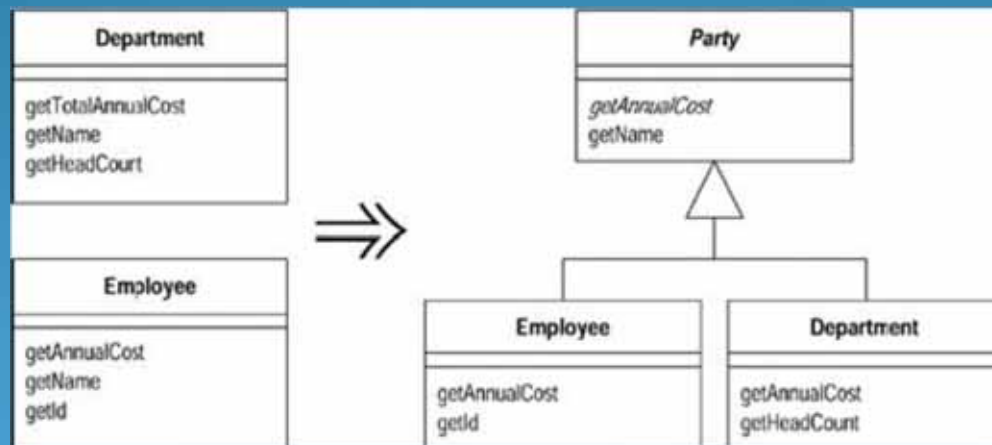
# Extract Subclass

- A class has features that are used only in some instances. Create a subclass for that subset of features



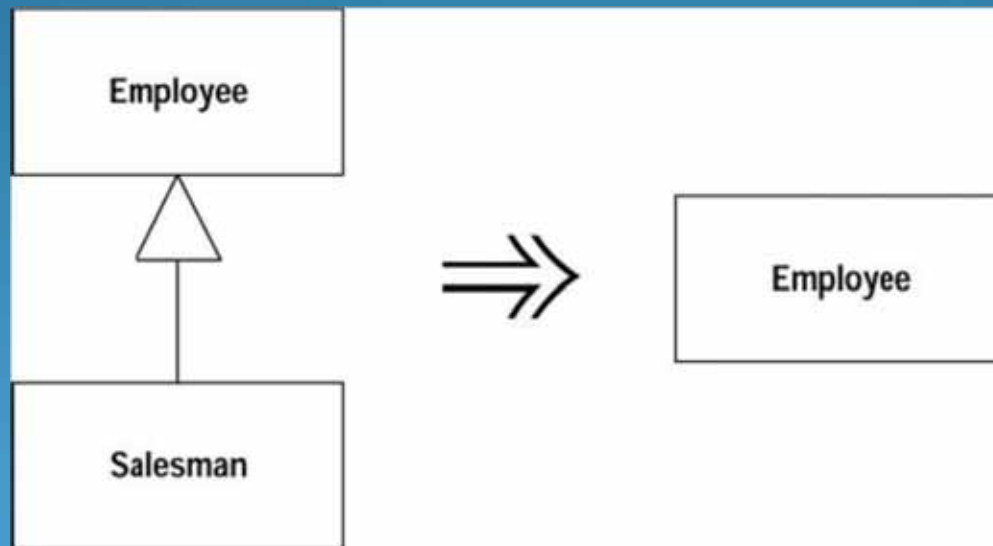
# Extract Superclass

- You have two classes with similar features. *Create a superclass and move the common features to the superclass.*



# Collapse Hierarchy

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



# Refactoring and Performance

- To make the software easier to understand, you often make changes that will cause the program to run more slowly.

# References

- Refactoring Improving the Design of Existing Code , Martin Fowler
- [www.refactoring.com](http://www.refactoring.com)
- [www.martinfowler.com](http://www.martinfowler.com)

# Questions

