Design Defects and Restructuring

Engr. Abdul-Rahman Mahmood

DPM, MCP, QMR(ISO9001:2000)

armahmood786@yahoo.com

alphapeeler.sf.net/pubkeys/pkey.htm

pk.linkedin.com/in/armahmood

www.twitter.com/alphapeeler

www.facebook.com/alphapeeler

abdulmahmood-sss alphasecure

armahmood786@hotmail.com

http://alphapeeler.sf.net/me

alphasecure@gmail.com

http://alphapeeler.sourceforge.net

http://alphapeeler.tumblr.com

armahmood786@jabber.org

alphapeeler@aim.com

mahmood_cubix 48660186

alphapeeler@icloud.com

http://alphapeeler.sf.net/acms/

Refactoring-II

Definition

- Refactoring is changing a software system by improving its internal structure without changing its external behavior, i.e. it is a technique to restructure the code in a disciplined way.
- It makes the software easier to understand and cheaper to modify.
- Refactoring is an overhead activity.

History of Refactoring

- Two of the first people to recognize the importance of refactoring were ***Kent Beck*** and ***Ward Cunningham***
- They worked with Smalltalk from the 80's onward.
- Refactoring has always been an important element in the Smalltalk community.
- **Ralph Johnson's** work with refactoring and frameworks has also been an important contribution.

Why Refactor ?

- Improves the design of software

  Without refactoring the design of the program will decay.

  Poorly designed code usually takes more to do the same things, often because they does the same thing in different places

- Easier to understand

  In most software development environments, somebody else will eventually have to read your code so it becomes easy for others to comprehend.

- To find the bugs

  It helps in finding the Bugs present in the program.

- To program faster

  It helps us to do the coding/programming faster as we have better understanding of the situation.

When to Refactor ?

- When you add a function

  Helps you to understand the code you are modifying.

  Sometimes the existing design does not allow you to easily add the feature.

- When you need to fix a bug

  If you get a bug report its a sign the code needs refactoring because the code was not clear enough for you to see the bug in the first place.

- When you do a Code Review

  - Code reviews help spread knowledge through the development team.

  - Works best with small review groups

Properties of Refactoring

- Preserve Correctness
- One step at a time
- Frequent Testing

Design Attributes

- Abstraction (information hiding)
- Flexibility
- Clarity
- Irredundancy

Steps to Refactoring

- Pareto analysis helps to identify heavily used or time consuming code.
- Refactoring begins by designing a solid set of tests for the section of code under analysis.
- Identify problems in code by review using bad smells of code.
- Perform a refactoring and test.

Pareto Analysis is a simple technique for prioritizing problem-solving work so that the first piece of work you do resolves the greatest number of problems. It's based on the *Pareto* Principle (also known as the 80/20 Rule) – the idea that 80 percent of problems may be caused by as few as 20 percent of causes.

Continued…

- The steps taken when applying the refactoring should be:

  - Small enough - to oversee the consequences they have.

  - Reproduce-able - to allow others to understand them.

  - Generalized : they are more a rule that can be applied to any structure.

  - Written down to allow sharing these steps and to keep a reference, with step by step instructions how to apply them.

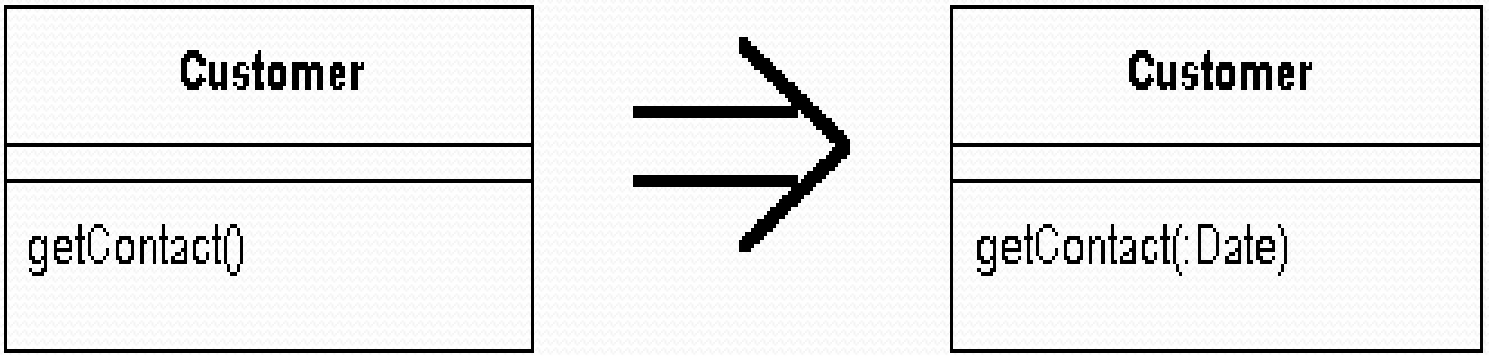
A few Refactorings

- Add Parameter
- Change Association
- Reference to value
- Value to reference
- Collapse hierarchy
- Consolidate conditionals
- Procedures to objects
- Decompose conditional
- Encapsulate collection
- Encapsulate downcast
- Encapsulate field
- Extract class
- Extract Interface
- Extract method
- Extract subclass
- Extract superclass
- Form template method
- Hide delegate
- Hide method
- Inline class
- Inline temp
- Introduce assertion
- Introduce explain variable
- Introduce foreign method

Refactoring examples

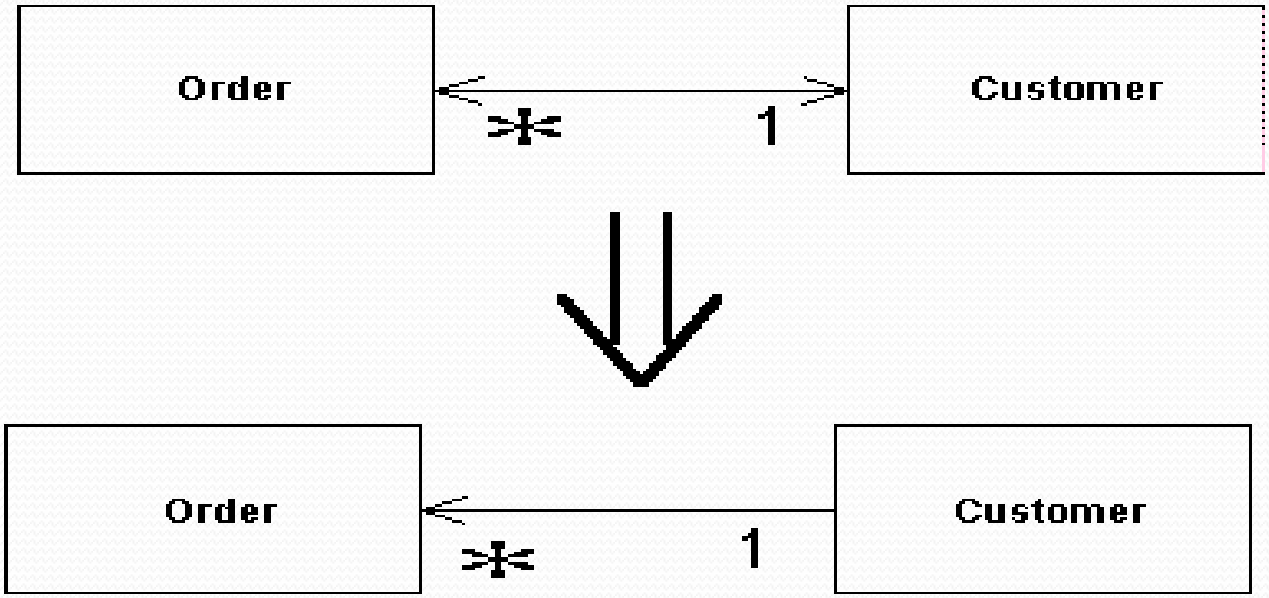
- Add a Parameter :

  When a method needs more information from its caller.



Continued..

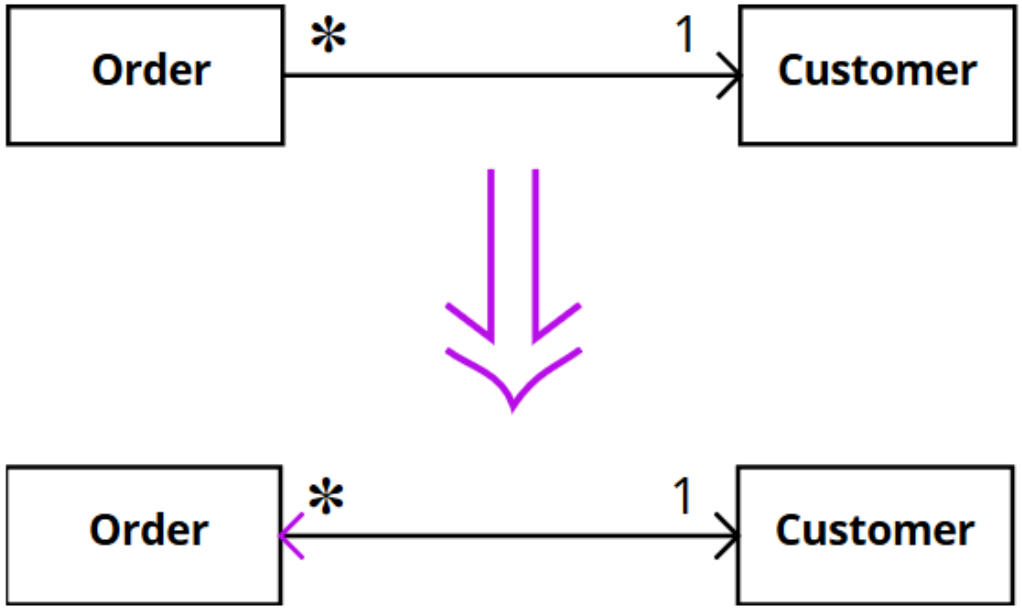
- Bi-directional Association to Unidirectional :
  We have a two-way association but one class no longer needs features from the other.



Change Unidirectional Association to Bidirectional

You have two classes that need to use each other's features, but there is only a one-way link.

Add back pointers, and change modifiers to update both sets.



Bi-directional Association - Java

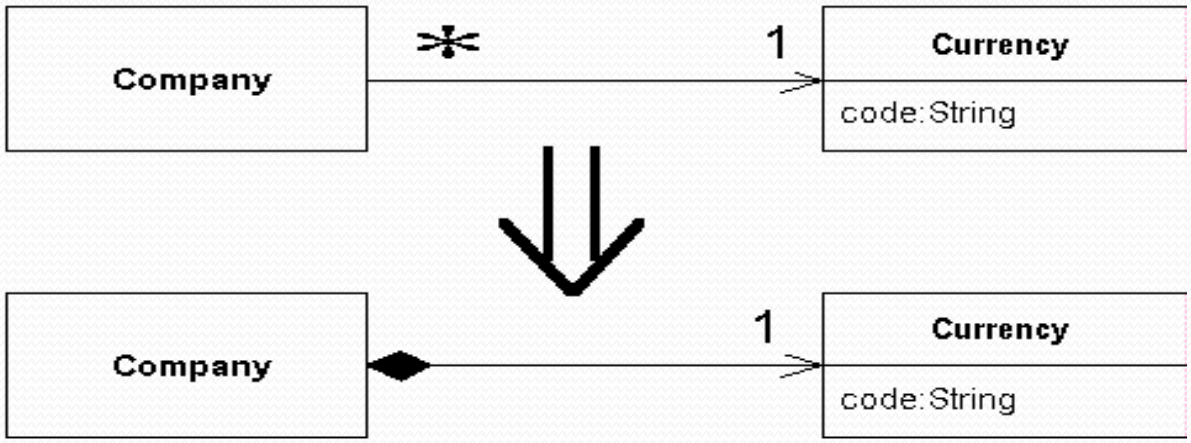
```
public class Person {
   private String name;
   private int age;
   private String address;
   public Person()    {
      this.name = name;
      this.age = age;
      this.address = address;
   }
   public void setName () { this.name = name;}
   public void setAge () { this.age = age; }
   public void setAddress () { this.address = address;}
   public String getName () { return name; }
   public int getAge () { return age;}
   public String getAddress () { return address; }
}
```

For Bidirectional Association:
Add : Person.setDog() and Dog.setOwner()

```
public class Dog {
    private String name;
    private int age;
    public Dog(){
       this.name = name;
       this.age = age;
    }
    public void setName () {
        this.name = name;
    }
    public void setAge () {
        this.age = age;
    }
    public String getName () {
        return name;
    }
    public int getAge () {
        return age;
    }
}
```

Continued…

- Change Reference to Value :
- **Problem:** You have a reference object that's too small and infrequently changed to justify managing its life cycle.
- **Solution:** Turn it into a value object.

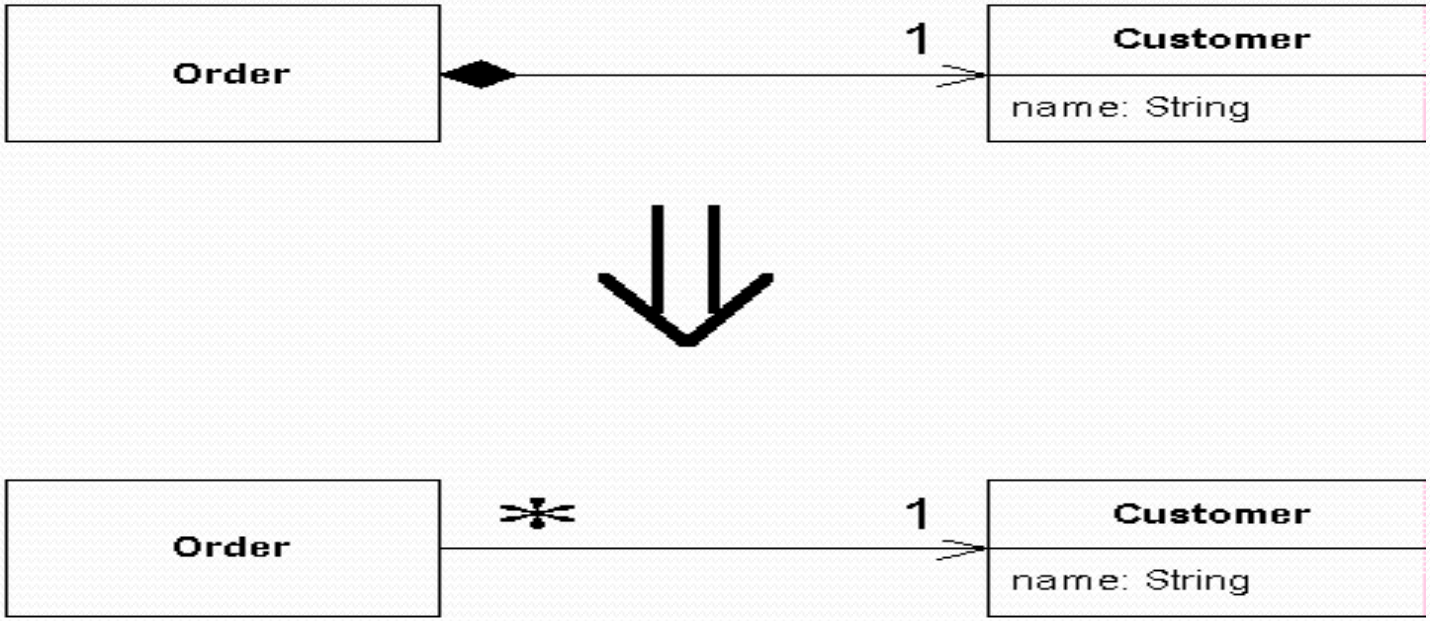


Make the object unchangeable. The object shouldn't have any setters or other methods that change its state and data (Remove Setting Method may help here). The only place where data should be assigned to the fields of a value object is a constructor.

Continued…

- Change Value to Reference :

When we have a class with many equal instances that we want to replace with a single object.



Example : Customer by Value

```
class Order
{
   private int orderNum;
   private Customer customer;
   Order(int orderNum, String customerString)
   {
      this.orderNum = orderNum;
      this.customer = new Customer(customerString);
   }
}

// elsewhere:
Order order1 = new Order(1, "I am a customer");
Order order2 = new Order(2, "I am a customer");
Order order3 = new Order(3, "I am a customer");
Order order4 = new Order(4, "I am a customer");
```

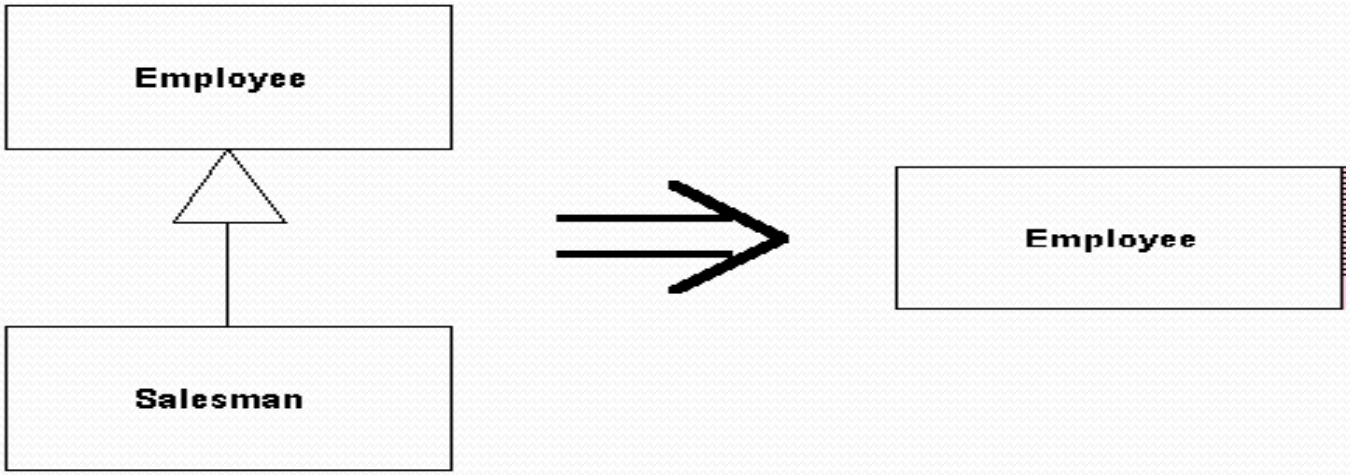
Example : Customer by Reference

```
class Order
{
   private int orderNum;
   private Customer customer;
   Order(int orderNum, Customer customer)
   {
      this.orderNum = orderNum;
      this.customer = customer;
   }
}

Customer customer = new Customer("I am a customer");
Order order1 = new Order(1, customer);
Order order2 = new Order(2, customer);
Order order3 = new Order(3, customer);
Order order4 = new Order(4, customer);
```

Continued..

- Collapse Hierarchy :

As superclass and subclass are not very different.

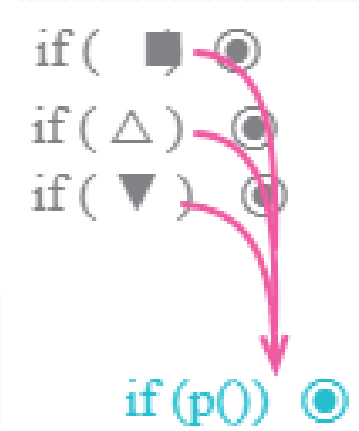


Continued…

Consolidate Conditional Expression

You have a sequence of conditional tests with the same result.

Combine them into a single conditional expression and extract it.




```
if (anEmployee.seniority < 2) return 0;
  if (anEmployee.monthsDisabled > 12) return 0;
  if (anEmployee.isPartTime) return 0;
```

```
if (isNotEligableForDisability()) return 0;

  function isNotEligableForDisability() {
    return ((anEmployee.seniority < 2)
            || (anEmployee.monthsDisabled > 12)
            || (anEmployee.isPartTime));
  }
```

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();
}
```

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

Continued…

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;
```

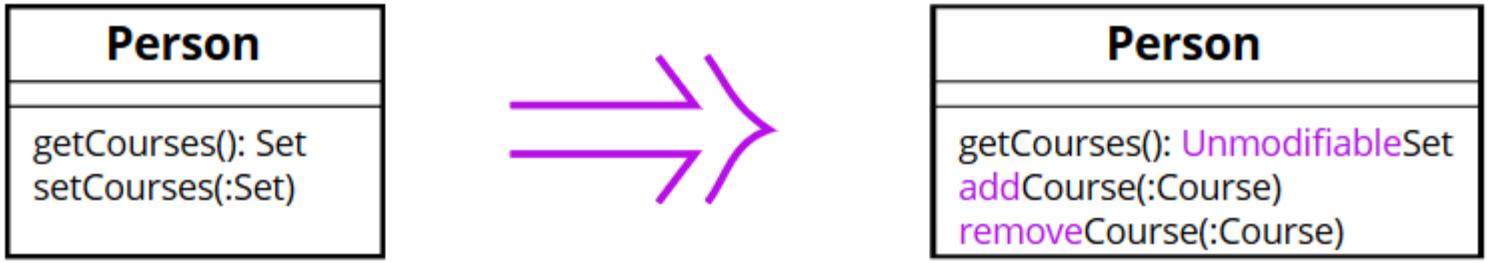
```
if (notSummer(date))
  charge = winterCharge(quantity);
else charge = summerCharge (quantity);
```

Continued..

Encapsulate Collection

A method returns a collection.

Make it return a read-only view and provide add/remove methods.



Benefits

The collection field is encapsulated inside a class. When the getter is called, it returns a copy of the collection, which prevents accidental changing or overwriting of the collection elements without the knowledge of the class that contains the collection.

Continued..

Encapsulate Downcast

A method returns an object that needs to be downcasted by its callers.

Move the downcast to within the method.

```
Object lastReading() {
  return readings.lastElement();
}
```

```
Reading lastReading() {
  return (Reading) readings.lastElement();
}
```

Encapsulate Field

There is a public field.

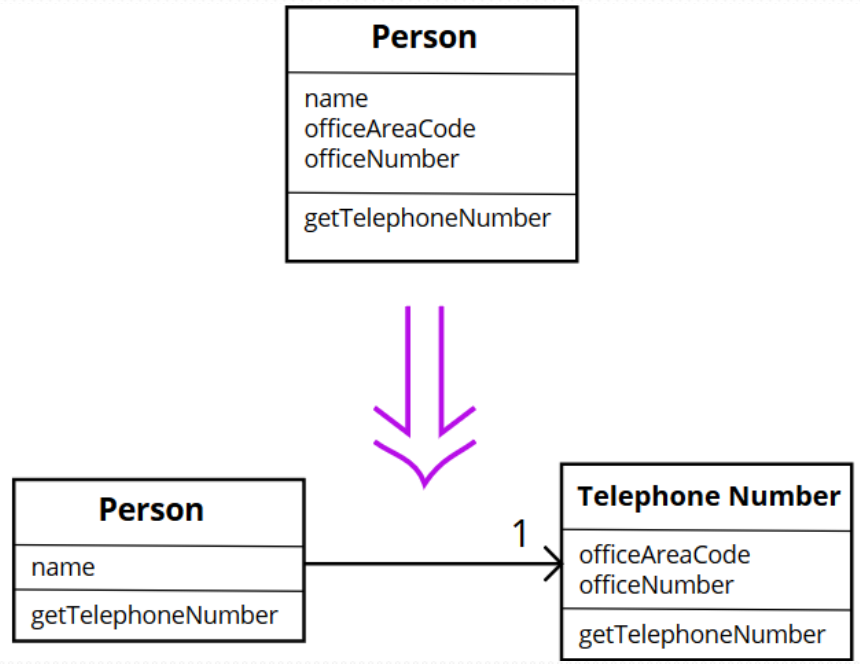
Make it private and provide accessors.

```
public String _name
```

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

Extract class

- **Problem :** one class doing work of two classes.
- **Solution:** Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.



Inverse of inline class

Single-responsibility classes are more reliable and tolerant of changes. For example, say that you have a class responsible for ten different things. When you change this class to make it better for one thing, you risk breaking it for the nine others.

Extract class example

```
abstract class Shape
{
  public void Draw()
  {
    try
    {
      // draw
    }
    catch (Exception e)
    {
      LogError(e);
    }
  }

  public static void LogError(Exception e)
  {
    File.WriteAllText(@"c:\Errors\Exception.txt", e.ToString());
  }
}
```

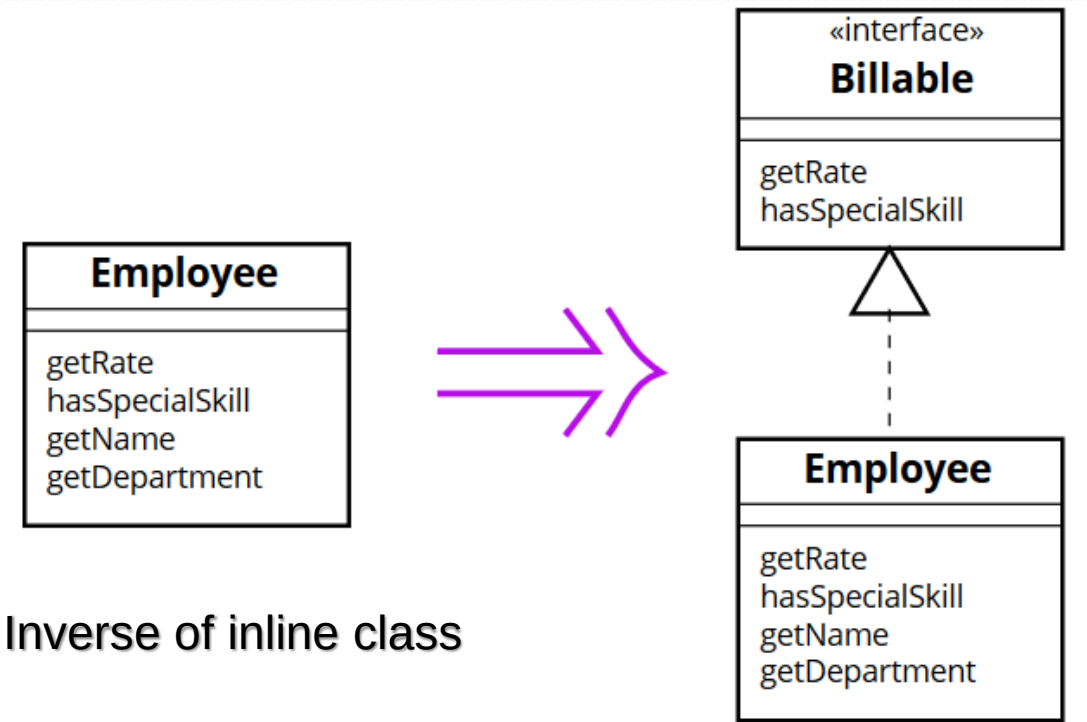
Before Refactoring

```
class Logger
{
  public static void LogError(Exception e)
  {
    File.WriteAllText(@"c:\Errors\Exception.txt",
e.ToString());
  }
}

abstract class Shape
{
  public void Draw()
  {
    try
    {
      // draw
    }
    catch (Exception e)
    {
      Logger.LogError(e);
    }
  }
}
```

After Refactoring

Extract interface

- Problem: Multiple clients are using the same part of a class interface. Another case: part of the interface in two classes is the same.
- **Solution :** Move this identical portion to its own interface.



Inverse of inline class

Extract interface example

File AClass.java

```
class AClass {
   public static final double CONSTANT=3.14;
   public void publicMethod() {//some code here}
   public void secretMethod() {//some code here}
```

Before Refactoring

```
// File AClass.java
class AClass implements AnInterface {
   public void publicMethod() {//some code here}
   public void secretMethod() {//some code here}
}
// File AnInterface.java
public interface AnInterface {
   double CONSTANT=3.14;
   void publicMethod();
}
```

After Refactoring

Extract Method

- **Problem:** You have a code fragment that can be grouped together.
- **Solution:** Move this code to a separate new method (function) & replace old code with a call to method.
- Inverse of inline method

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

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

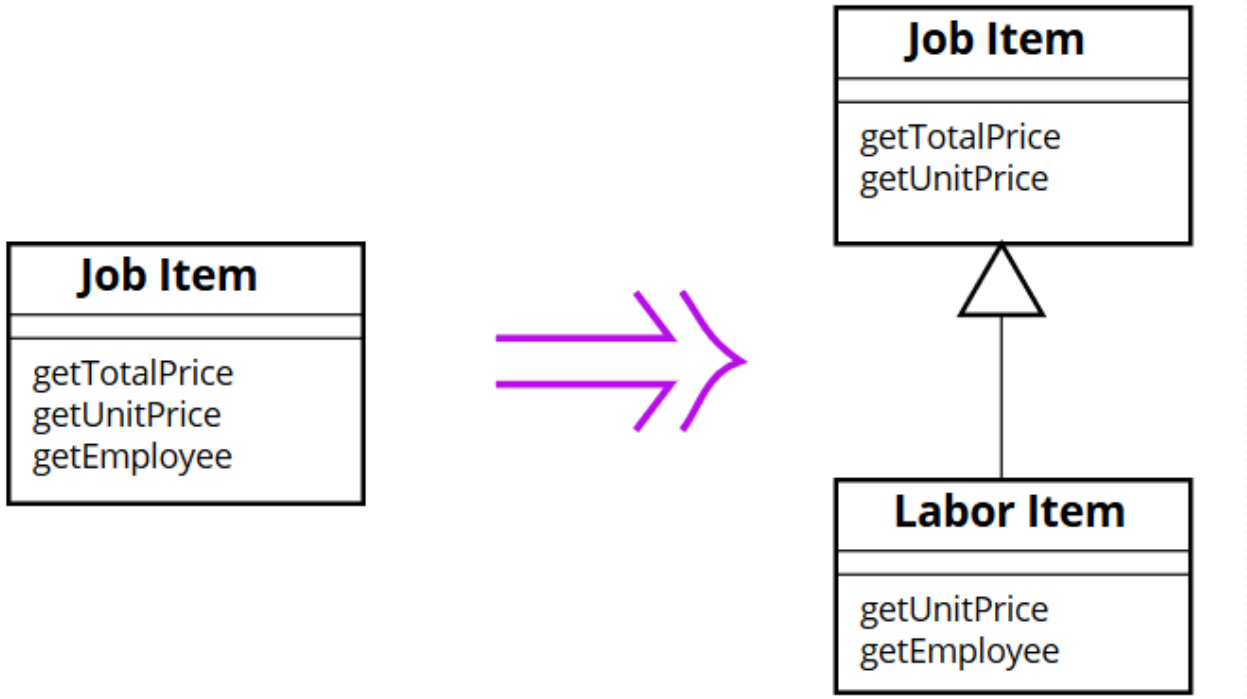
Before Refactoring

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

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

After Refactoring

Extract Subclass

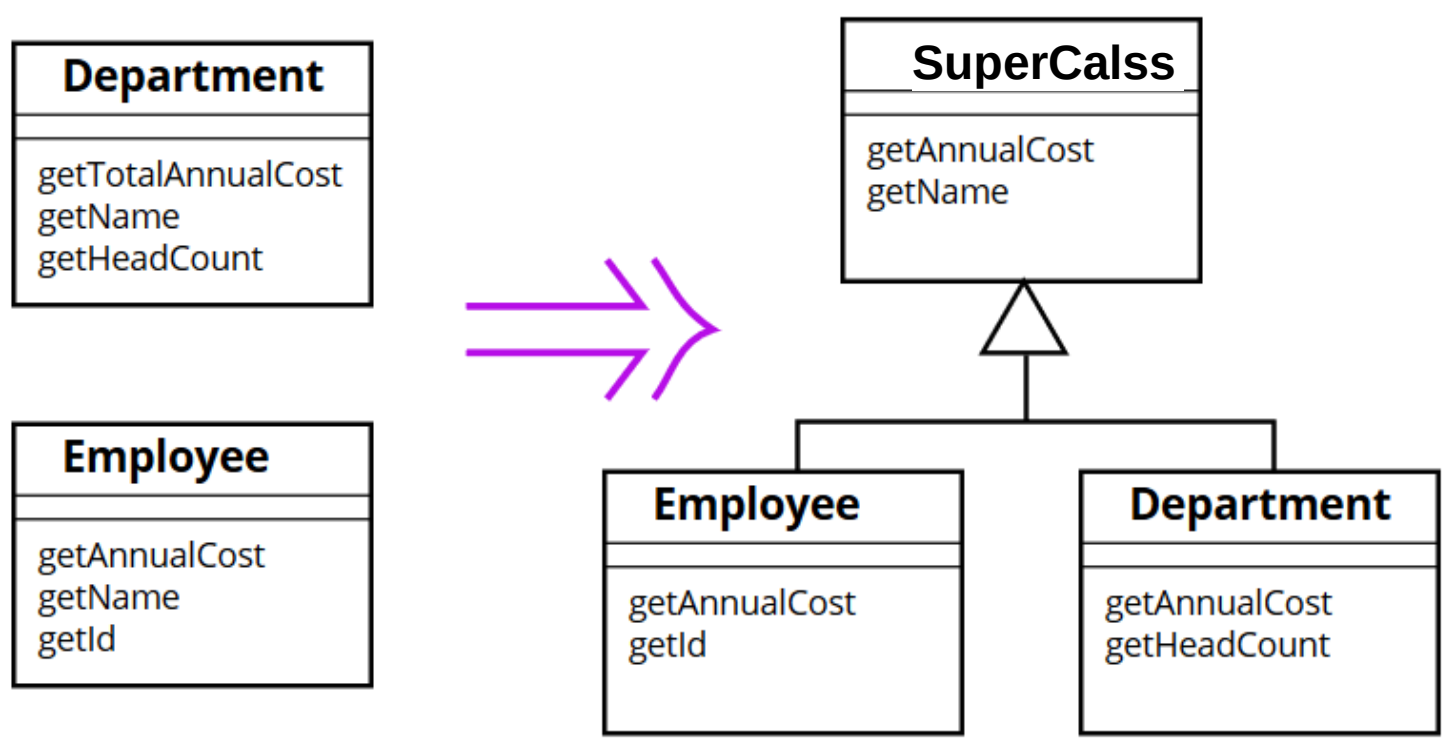
- **Problem:** A class has features that are used only in certain cases.
- **Solution:** Create a subclass and use it in these cases.



Inverse of collapse hierarchy

Extract Superclass

- **Problem:** You have 2 classes with common fields & methods.
- **Solution:** Create a shared superclass for them and move all the identical fields and methods to it.



Inverse of collapse hierarchy

Extract Superclass

```
class Department {
  get totalAnnualCost() {...}
  get name() {...}
  get headCount() {...}
}

class Employee {
  get annualCost() {...}
  get name() {...}
  get id() {...}
}
```

Before Refactoring

```
class SuperCalss {
  get name() {...}
  get annualCost() {...}
}

class Department extends SuperCalss {
  get annualCost() {...}
  get headCount() {...}
}

class Employee extends SuperCalss {
  get annualCost() {...}
  get id() {...}
}
```

After Refactoring

Extract Variable

- **Problem:** have expression that's hard to understand.
- **Solution:** Place the result of the expression or its parts in separate variables that are self-explanatory.
- Also known **as Introduce Explaining Variable**

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

Before Refactoring

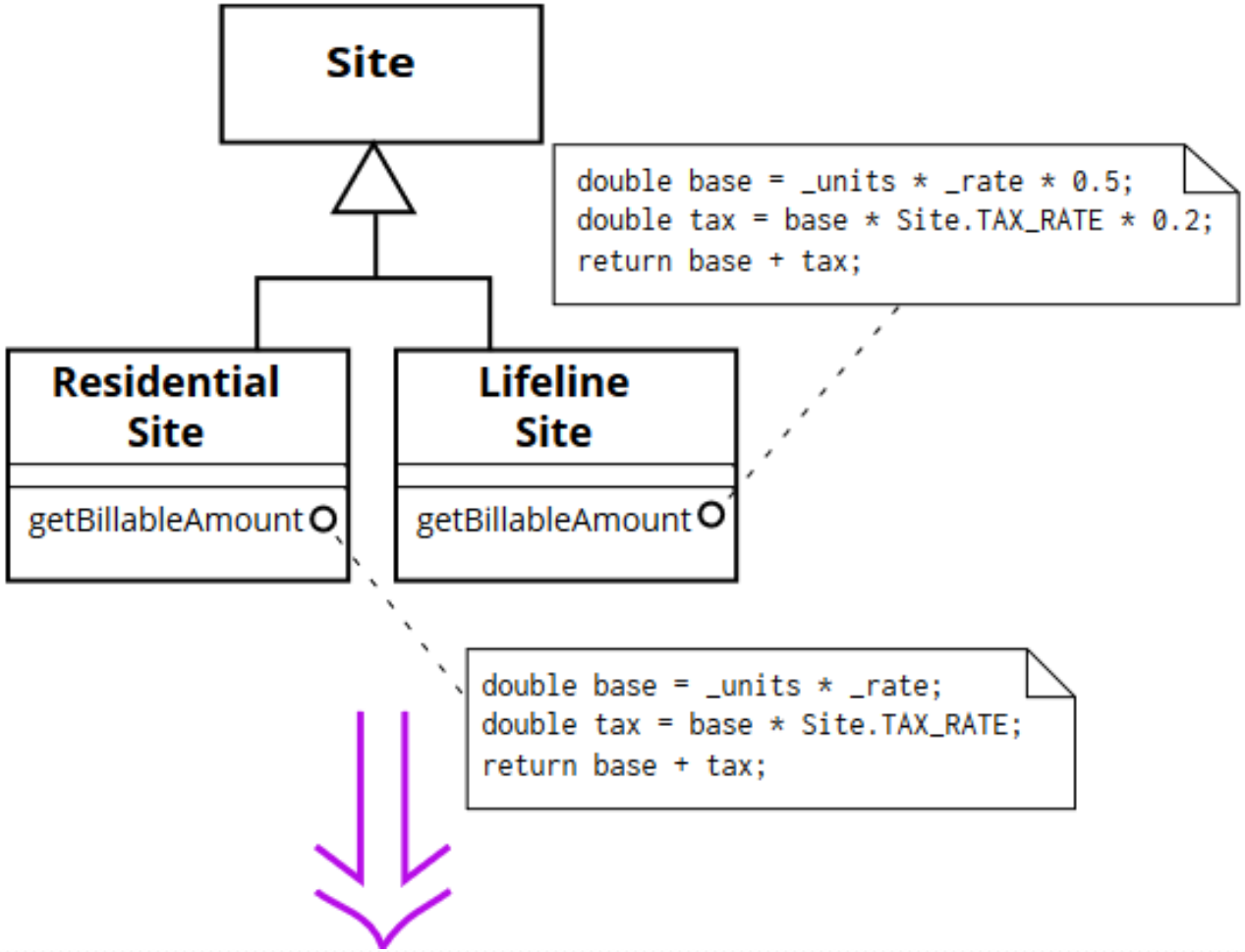
```
void renderBanner() {
 final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
 final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;
 final boolean wasResized = resize > 0;

 if (isMacOs && isIE && wasInitialized() && wasResized) {
  // do something
 }
}
```

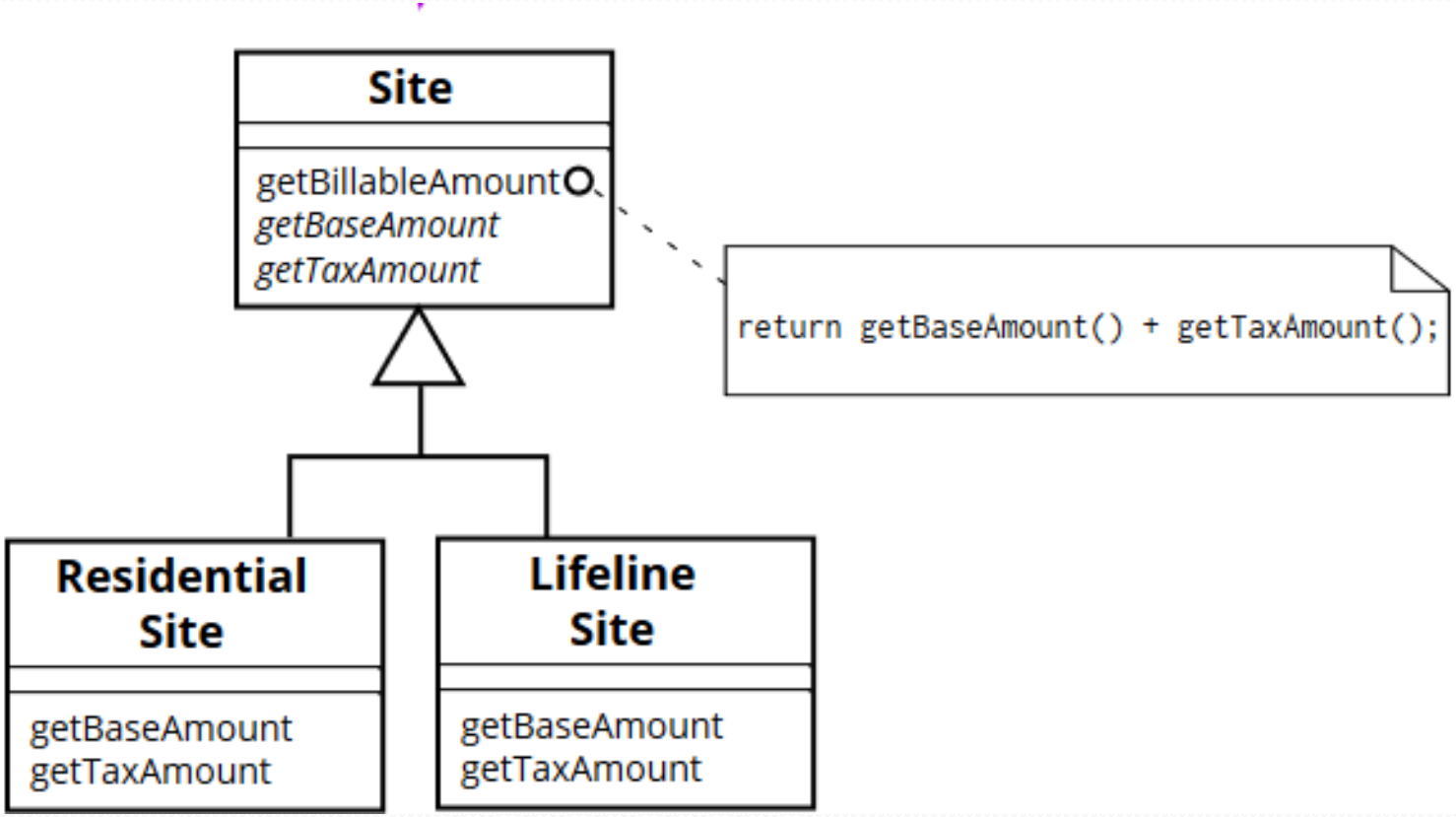
After Refactoring

Form Template Method

You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.

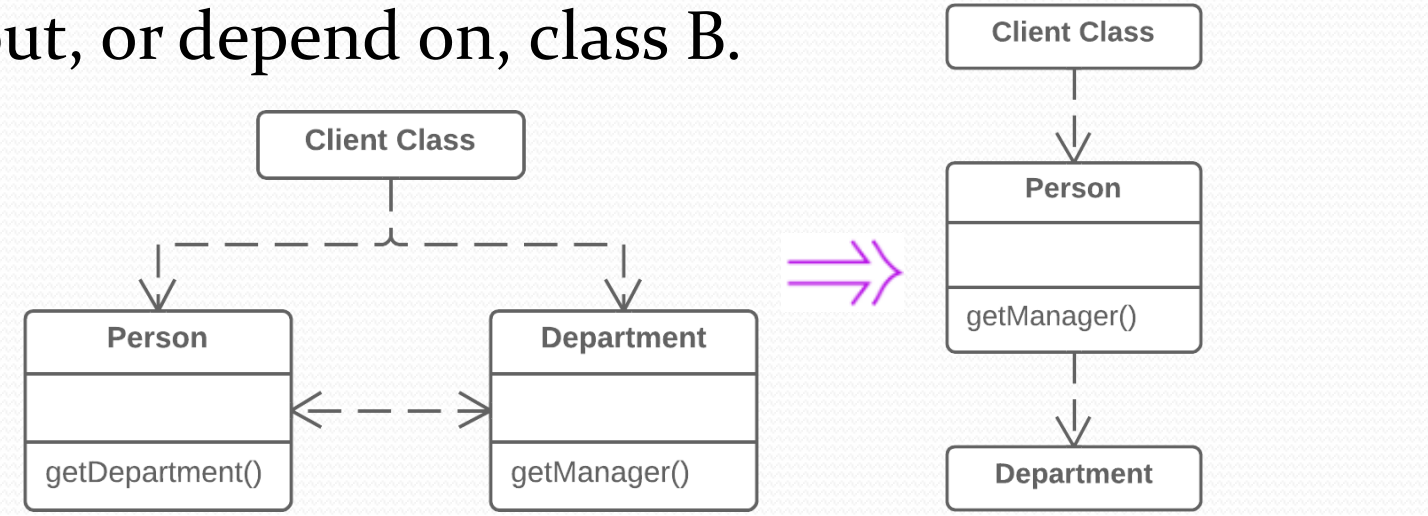
Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.




Site
getBillableAmount
getBaseAmount
getTaxAmount
return getBaseAmount() + getTaxAmount();
Residential Site
getBaseAmount
getTaxAmount
Lifeline Site
getBaseAmount
getTaxAmount

Hide Delegate

- **Problem:** client gets object B from a field or method of object A. Then client calls a method of object B.
- **Solution:** Create a new method in class A that delegates the call to object B. Now the client doesn't know about, or depend on, class B.



Inverse of remove middle man

```
manager = aPerson.department.manager;
```

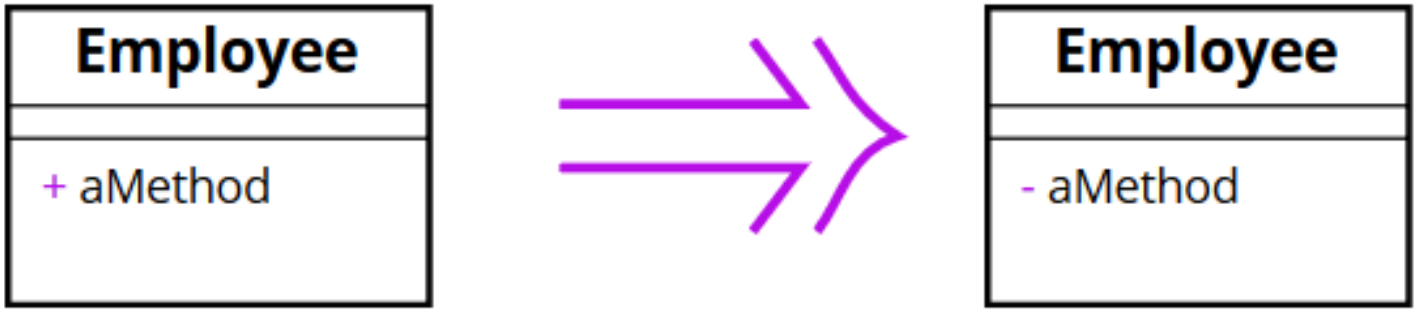
⇓

```
manager = aPerson.manager;
class Person {
  get manager() {return this.department.manager;}
```

Hide Method

A method is not used by any other class.

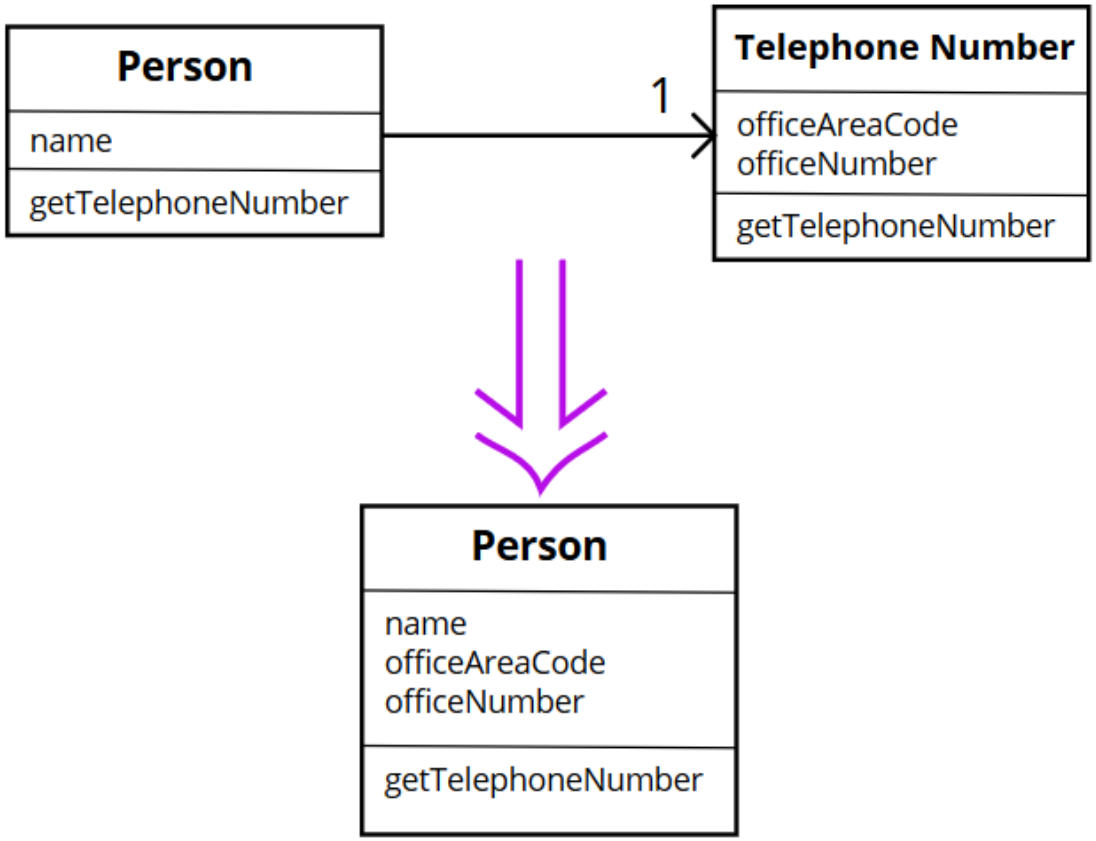
Make the method private.



Inline Class

A class isn't doing very much.

Move all its features into another class and delete it.



inverse of *Extract Class*, *Extract Interface*

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;
}
```

inverse of *Extract Method*

Bad Smells in Code

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

Few solutions to Bad Smells

- Duplicated Code

  If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

  solution: perform EXTRACT METHOD and invoke the code from both places.

- Long Method

  The longer a procedure is the more difficult it is to understand.

  solution: perform EXTRACT METHOD or Decompose Conditional or Replace Temp with Query.

- ## Large class

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

  solution: perform EXTRACT CLASS or SUBCLASS

- ## Long Parameter List

  With objects you don't need to pass in everything the method needs, instead you pass in enough so the method can get to everything it needs

  solution: Use REPLACE PARAMETER with METHOD when you can get the data in one parameter by making a request of an object you already know about.

- Shotgun Surgery

  This situation occurs when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.

  solution: perform MOVE METHOD/FIELD or INLINE Class bring a whole bunch of behavior together.

- Feature Envy

  It is a method that seems more interested in a class other in the one that it is in.

  solution: perform MOVE METHOD or EXTRACT METHOD on the jealous bit and get it home.

- Switch Statements

  They are generally scattered throughout a program. If you add or remove a clause in one switch, you often have to find and repair the others too.

  solution: Use EXTRACT METHOD to extract the switch statement and then MOVE METHOD to get it into the class where the polymorphism is needed.

Benefits of Refactoring

Refactoring is useful to any program that has at least one of the following shortcomings:

- Programs that are hard to read are hard to modify.
- Programs that have duplicate logic are hard to modify

Continued…

- Programs that require additional behavior that requires you to change running code are hard to modify.

- Programs with complex conditional logic are hard to modify.

Refactoring Risks

- Introducing a failure with refactoring can have serious consequences.

- When done on a system that is already in production, the consequences of introducing bugs without catching them are very severe.

Costs of Refactoring

Language / Environment :

Depends on how well the operations on the source code are supported. But in general the cost of applying the basic text modifications should be bearable.

Testing :

Relies heavily on testing after each small step and having a solid test suite of unit tests for the whole system substantially reduces the costs which would be implied by testing manually

Continued…

Documentation :

Costs of updating documentation of the project should not be underestimated as applying refactorings involves changes in interfaces, names, parameter lists and so on.

All the documentation concerning these issues must be updated to the current state of development.

Continued…

System :

The tests covering the system need an update as well as the interfaces and responsibilities change. These necessary changes can contribute to higher costs as tests are mostly very dependent on the implementation.

When to put Off Refactoring ?

- Concerned code is neither able to compile or to run in a stable manner, it might be better to throw it away and rewrite the software from scratch.

- When a deadline is very close. Then it would take more time to do the refactoring than the deadline allows.