Refactoring

Amin Ranj Bar

Activities

- Identify where the software should be refactored.
- Determine which refactoring(s) should be applied to the identified places.
- Guarantee that the applied refactoring preserves behavior.
- Apply the refactoring.
- Assess the effect of refactoring on quality characteristics (e.g. comprehension, maintainability) or the process (e.g. productivity, cost, effort).
- Maintain the consistency between refactored program code and other software artifacts.

Toward a Catalog of Refactoring

- The book by Fowler contains a large catalog of "refactoring patterns".
  - Composing methods
    - Extract method, Inline method, …
  - Moving features between objects
  - Organizing data
  - Simplifying conditional expressions
  - Making method calls simpler
  - Dealing with generalization
- More are being proposed by others.

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 (double amount) {
    printBanner();

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

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


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

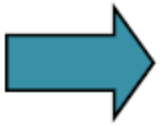
Extract Method (cont.)

- Be careful with local variables: parameters passed into the original method and temporaries declared within the original method.
- In some cases they would prevent doing the refactoring.

- The easiest case is when a variable is read but not changed. We can pass it as a parameter.
- In the following example, we extract the printing of details with a method with one parameter.

Extract Method (cont.)

```
void printOwing () {
   Enumeration e = _orders.elements();
   double outstanding = 0.0;

   printBanner();

   // calculate outstanding
   while (e.hasMoreElements()) {
     Order each =
            (Order) e.hasNextElement();
     outstanding += each.getAmount();
   }

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



```
void printOwing () {
   Enumeration e = _orders.elements();
   double outstanding = 0.0;

   printBanner();

   // calculate outstanding
   while (e.hasMoreElements()) {
     Order each = (Order) e.hasNextElement();
     outstanding += each.getAmount();
   }

   printDetails(outstanding);
}


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

Extract Method (cont.)

- Consider an assignment to a local variable.
- In the simplest case the temporary variable is used only within the extracted code, so it can easily be moved.
- The other case is the use of the temporary variable outside the extracted code.
  - If it is used afterwards you need to make the extracted code return the changed value.

Extract Method (cont.)

```
void printOwing () {
   Enumeration e = _orders.elements();
   double outstanding = 0.0;

   printBanner();

   // calculate outstanding
   while (e.hasMoreElements()) {
     Order each =
            (Order) e.hasNextElement();
     outstanding += each.getAmount();
   }

   printDetails (outstanding);
}
```

```
void printOwing () {
   printBanner();
   double outstanding = getOutstanding();
   printDetails(outstanding);
}

double getOutstanding() {
   Enumeration e = _orders.elements();
   double outstanding = 0.0;
   while (e.hasMoreElements()) {
     Order each =
            (Order) e.hasNextElement();
     outstanding += each.getAmount();
   }
   return outstanding;
}
```

Extract Method (cont.)

- The enumeration variable is only used in the extracted code, so it can be moved entirely within that method.
- Can also rename variable outstanding to something more meaningful.

```
double getOutstanding() {
   Enumeration e = _orders.elements();
   double result = 0.0;
   while (e.hasMoreElements()) {
     Order each =
            (Order) e.hasNextElement();
     result += each.getAmount();
   }
   return result;
}
```

Extract Method (cont.)

- If variable outstanding comes with an initial value other than the default (e.g. a value assigned to it by a parameter), we have to pass this value as a parameter to getOutstanding().

Extract Method (cont.)

```
void printOwing (double previousAmount) {
  Enumeration e = _orders.elements();
  double outstanding =
                    previousAmount * 1.2;
  printBanner();

  // calculate outstanding
  while (e.hasMoreElements()) {
    Order each =
            (Order) e.hasNextElement();
    outstanding += each.getAmount();
  }

  printDetails (outstanding);
}
```

```
void printOwing (double previousAmount) {
  double outstanding =
                    previousAmount * 1.2;
  printBanner();
  outstanding =
            getOutstanding(outstanding);
  printDetails(outstanding);
}

double getOutstanding(double initialValue) {
    double result = initialValue;
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
      Order each =
            (Order) e.hasNextElement();
      result += each.getAmount();
    }
    return result;
}
```

Extract Method (cont.)

- We can now ‘clean’ the code in printOwing()

```
void printOwing (double previousAmount) {
   printBanner();
   double outstanding = getOutstanding(previousAmount * 1.2);
   printDetails(outstanding);
}
```

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;
}
```

Inline Temp

- You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings.
- *Replace all references to that temp with the expression.*

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000);
```

```
return (anOrder.basePrice() > 1000)
```

Replace Temp with Query

- You are using a temporary variable to hold the result of an expression.
- *Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.*

```
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;
}
```

Introduce Explaining Variable

- You have a complicated expression.

- *Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.*

Introduce Explaining Variable

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

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) {
   // do something
}
```

Split Temporary Variable

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

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

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

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

Remove Assignments to Parameters

- The code assigns to a parameter.
- *Use a temporary variable instead.*

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

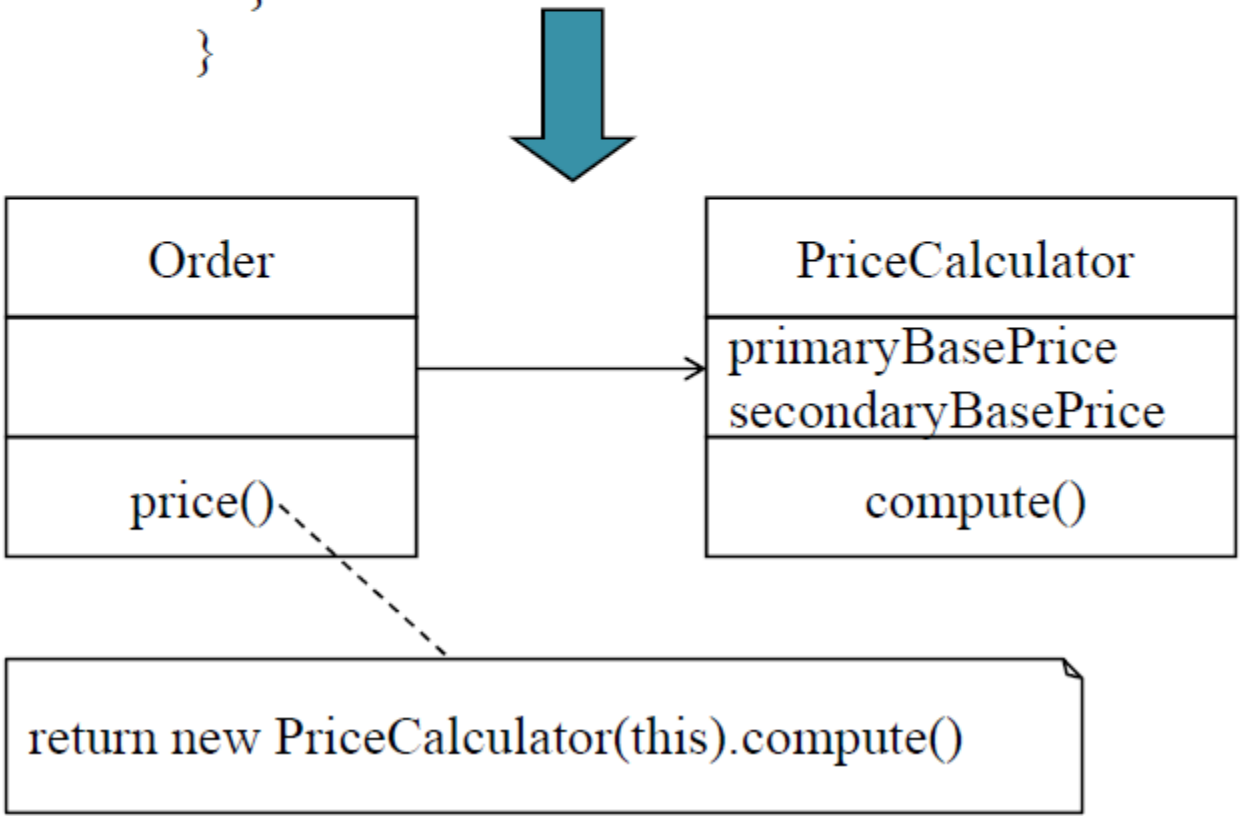
```
int discount (int inputVal, int quantity, int yearToDate) {
   int result = inputVal;
   if (inputVal > 50) result -= 2;}
```

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 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;
        // long computation
        …
    }
}
```



Substitute Algorithm

- You want to replace an algorithm with one that is clearer.

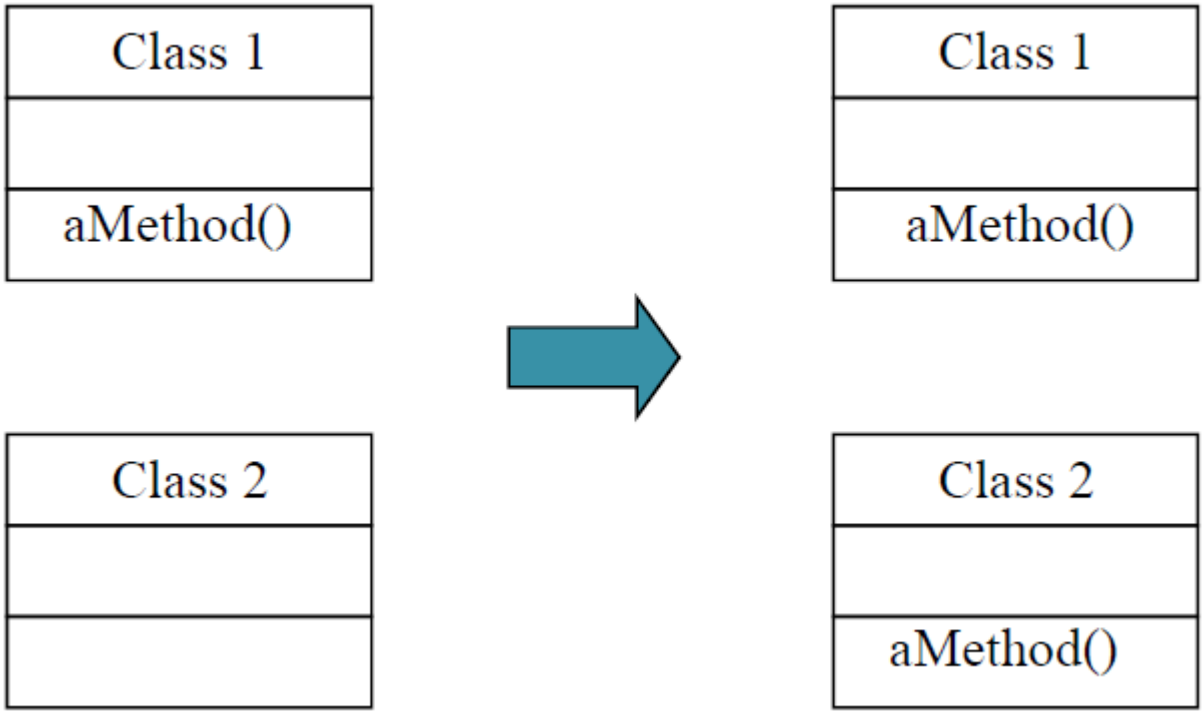
- *Replace the body of the method with the new algorithm.*

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

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

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.*



```
class Account …
   double overdraftCharge() {
      if (_type.isPremium()) {
         double result = 10;
         if (_daysOverdrawn > 7) result += (_daysOverdrawn – 7) * 0.85;
         return result;
      }
      else return _daysOverdrawn * 1.75;
   }
   double bankCharge() {
      double result = 4.5;
      if (_daysOverdrawn > 0) result += overdraftCharge();
      return result;
   }
   private AccountType _type;
   private int _daysOverdrawn;
}
```

- Consider having a number of account types, each of which has its own rule for calculating the overdraft charge.
- Let us move the overdraft charge method over to the account type.
- Look at the features method overdraftCharge() uses

```
class AccountType…
    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7) result += (daysOverdrawn – 7) * 0.85;
            return result;
        }
        else return daysOverdrawn * 1.75;
    }
}
```

- Replace the source method body with a simple delegation.

```
class Account …
    double overdraftCharge() {
        return _type.overdraftCharge(_daysOverdrawn);
    }
```

- To remove the method, we need to find all callers of the method and redirect them to call the method in the account type.
- Once we have replaced all callers, we can now remove the method declaration in class Account.

```
class Account …
   double bankCharge() {
      double result = 4.5;
      if (_daysOverdrawn > 0) result += _type.overdraftCharge(_daysOverdrawn);
      return result;
   }
```

- Method overdraftCharge() in AccountType uses only one feature of class Account.
- For access to several features of Account, then class Account must be passed as a parameter to overdraftCharge()

```
class AccountType…
    double overdraftCharge(Account account) {
        if (isPremium()) {
            double result = 10;
            if (account.getDaysOverdrawn() > 7)
                result += (account.getDaysOverdrawn() – 7) * 0.85;
            return result;
        }
        else return account.getDaysOverdrawn() * 1.75;
    }
}
```

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, can change all its users.*
- In the example, we want to move _interestRate fromAccount to AccountType.

```
class Account…
  private AccountType _type;
  private double _interestRate;

  double interestForAmount_days (double amount, int days) {
    return _interestRate * amount * days / 365;
  }
```

```
class AccountType…
   private double _interestRate;

   void setInterestRate (double arg) {
      _interestRate = arg;
   }
   double getInterestRate() {
      return _interestRate;
   }
```

```
class Account…
  private AccountType _type;

  double interestForAmount_days (double amount, int days) {
   return _type.getInterestRate() * amount * days / 365;
  }
```

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.*

```
class Person:

public String getName() {return _name}
public String getTelephoneNumber() {
    return ("(" + _officeAreaCode + ")" + _officeNumber);
}

String getOfficeAreaCode() {return _officeAreaCode;}
void setOfficeAreaCode(String arg) {_officeAreaCode = arg;}

String getOfficeNumber() { return _officeNumber;}
void setOfficeNumber(String arg) { _officeNumber = arg;}

private String _name;
private String _officeAreaCode;
private String _officeNumber;
```

- Separate office behavior into its own class.
- Make link from person to the telephone class.

```
class TelephoneNumber {
 …
}


class Person {
 private TelephoneNumber _officeTelephone = new TelephoneNumber();
}
```

- Use Move Field on _officeAreaCode and rename it _areaCode.
- Use Move Field on _officeNumber and rename it _number.
- Use Move Method on telephone number.

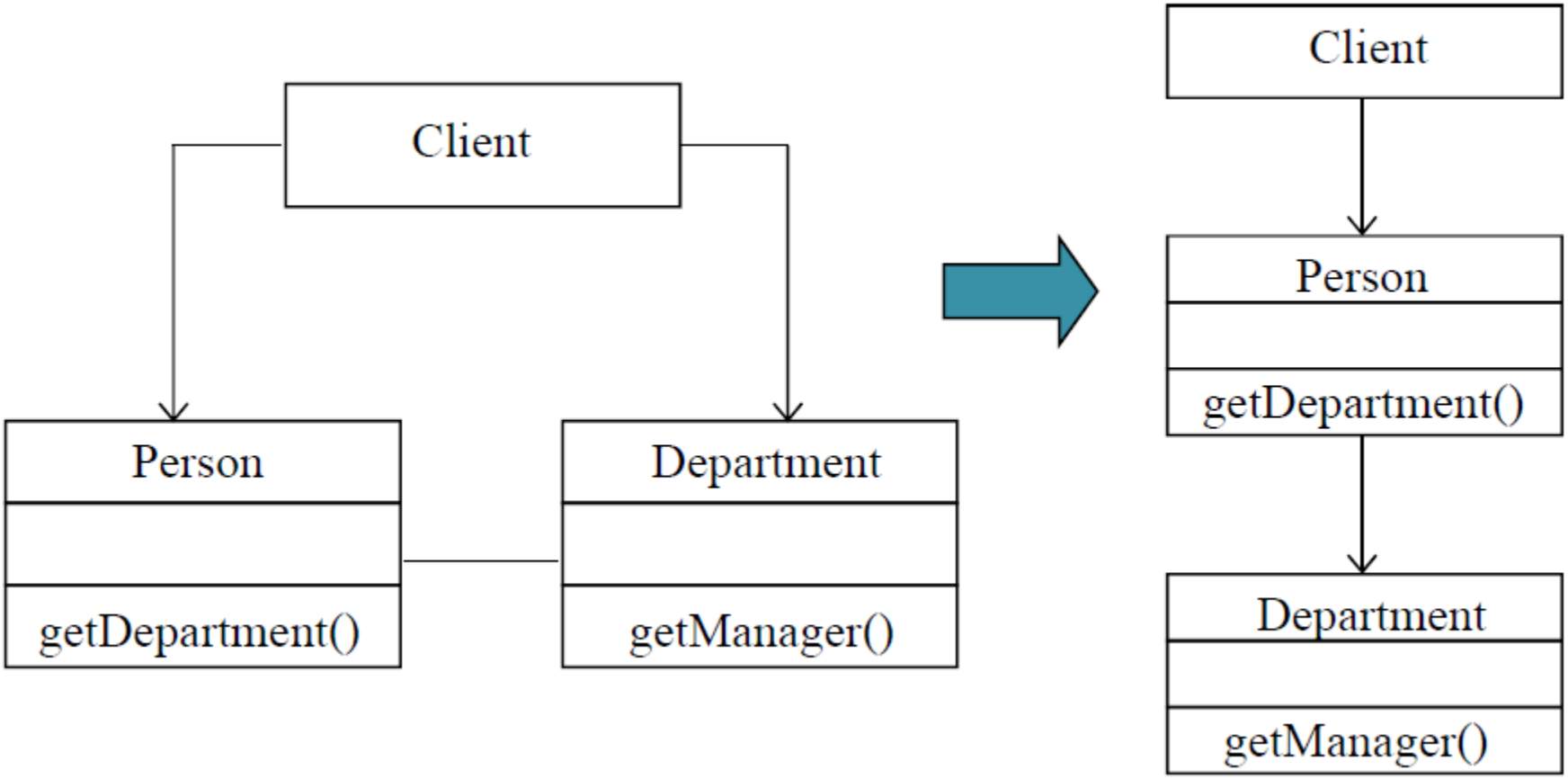
```
class TelephoneNumber {
 public String getTelephoneNumber() {return ("(" + _areaCode + ")" + _number);}
 String getAreaCode() {return _areaCode;}
 void setAreaCode(String arg) {_areaCode = arg;}
 String getNumber() {return _number;}
 void setNumber(String arg) {_number = arg;}
 private String  _areaCode;
 private String _number;
}

class Person {
 public String getName() {return _name;}
 public String getTelephoneNumber() {return _officeTelephone.getTelephoneNumber();}
 private String _name;
 private TelephoneNumber _officeTelephone = new TelephoneNumber();
}
```

Inline Class

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

Hide Delegate

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



```
class Person {
 Department  _department;

 public Department getDepartment() {
   return _department;
 }

 public void setDepartment(Department arg) {
   _department = arg;
 }
}
```

```
Class Department {
  private String _chargeCode;
  private Person _manager;

  public Department (Person manager) {
    _manager = manager;
  }

  public Person getManager() {
    return _manager;
  }
  …
}
```

- If a client wants to know a person's manager, it needs to get the department first:

```
Manager = john.getDepartment().getManager();
```

- We can reduce this coupling by creating a simple delegating method on Person:

```
public Person getManager() {
  return _department.getManager();
}
```

- We now need to change all clients of Person to use this new method:

```
Manager = john.getManager()
```

Introduce a Foreign Method

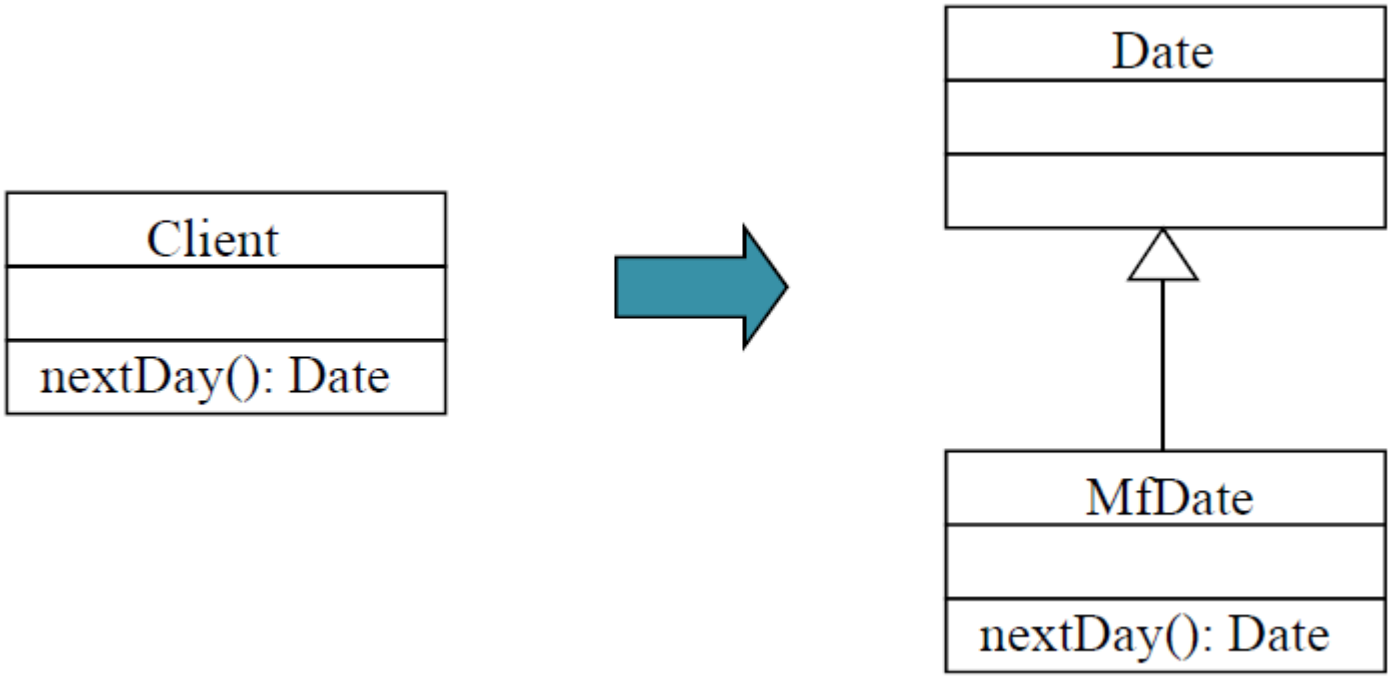
- A server class you are using needs an additional method, but you cannot modify the class.
- *Create a method in the client class with an instance of the server class as its first argument.*

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

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
   // foreign method, should be on Date
   return new Date (arg.getYear(), arg.getMonth(), arg.getDate()+1);
}
```

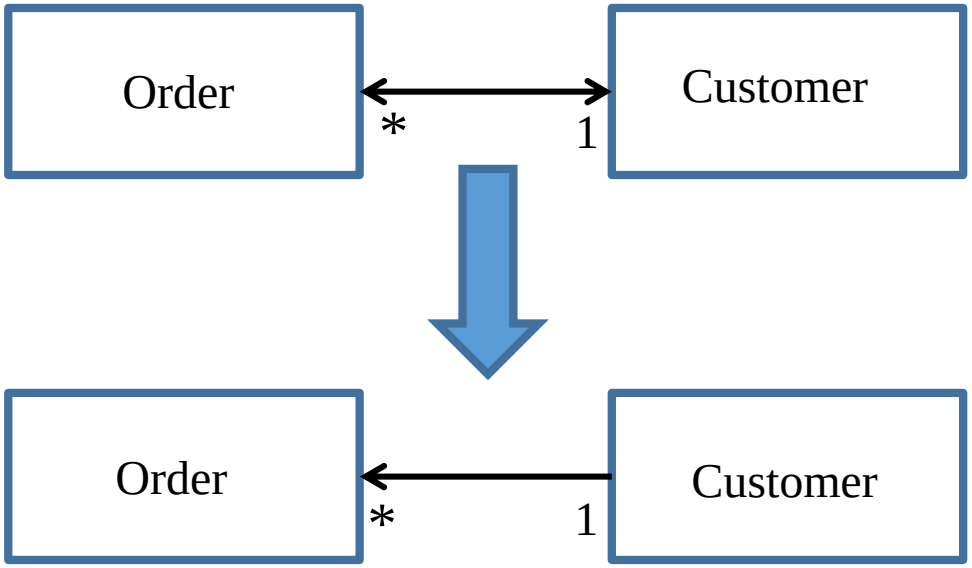
Introduce Local Extension

- 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.*



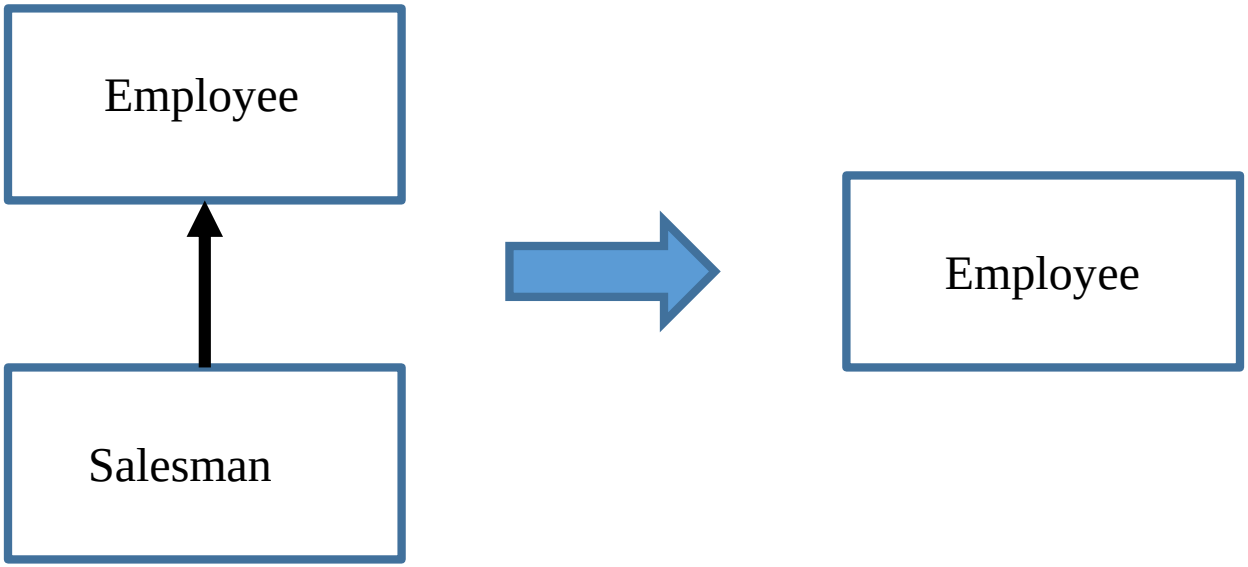
Change Bidirectional Association to Unidirectional

- You have a two-way association but one class no longer needs features from the other.
- Drop the unneeded end of the association.



Collapse Hierarchy

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



Extract Subclass

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

References

- **[MT04]** Tom Mens and Tom Tourwe, *A Survey of Software Refactoring*. IEEE Transactions on Software Engineering, Vol. 30, No. 2, pp. 126-139, 2004.

- **[Ett06]** Ran Ettinger, *Refactoring via Program Slicing and Sliding*. Ph.D. Thesis, University of Oxford, 2006.