POLITÉCNICA
"Ingeniamos el futuro"

CAMPUS
DE EXCELENCIA
INTERNACIONAL

MiW

1. Metodologías Ligeras de Desarrollo Web
1.3. Refactoring
1.4. Desarrollo Dirigido por Pruebas

RESULTADOS DE APRENDIZAJE

- *Presentar la refactorización (qué),*
  sus motivos (por qué),
  y sus ventajas (para qué)
- *Presentar el catálogo de refactorización (cómo)*
- *Presentar las condiciones para la refactorización (dónde)*

- *Habilitar la aplicación del Refactoring*
  para el Diseño Dirigido por Pruebas (TDD)
  que determinará el momento (cuándo)
  y el grado de su aplicación (cuánto)
  cada vez que se añade funcionalidad
  para superar un nuevo test de las historias de usuario

INDICE

El contenido de este tema está basado directamente del libro:

Fowler, M.; Beck, K.; Brant, J.; Opdyke, W.; Roberts, D.

Refactoring. Improving the Design of Existing Code

Addison-Wesley Professional, 1999

1. Definición y Justificación de la Refactorización

- **Definición:**
  - ***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.*

  - "Is refactoring just cleaning up code?" In a way the answer is yes, but I think refactoring goes further because it provides a technique for cleaning up code in a more efficient and controlled manner
  - I should amplify a couple of points in my definitions. First, the purpose of refactoring is to make the software easier to understand and modify. You can make many changes in software that make little or no change in the observable behavior. Only changes made to make the software easier to understand are refactorings. A good contrast is performance optimization.

1. Definición y Justificación de la Refactorización

- **Justificación:**
  - I strongly believe that a good design is essential for rapid software development. Without a good design, you can progress quickly for a while, but soon the poor design starts to slow you down. You spend time finding and fixing bugs instead of adding new function. Changes take longer as you try to understand the system and find the duplicate code. New features need more coding as you patch over a patch that patches a patch on the original code base. [**Fundamentos de Diseño de Software**]

1. Definición y Justificación de la Refactorización

- **Justificación:**
  - As people change code - changes to realize short-term goals or changes made without a full comprehension of the design of the code - the code loses its structure. It becomes harder to see the design by reading the code. Loss of the structure of code has a cumulative effect. The harder it is to see the design in the code, the harder it is to preserve it, and the more rapidly it decays. [**Ley de Complejidad Creciente - Lemman y Belady**]
  - The trouble is that when you are trying to get the program to work, you are not thinking about that future developer. [**Ley de Evolución Continua – Leman y Belady**]
  - Some people can read a lump of code and see bugs, I cannot. [**Análisis económico de las pruebas estáticas – revision de código**]

1. Definición y Justificación de la Refactorización

- **Ventajas:**
  - **Refactoring Makes Software Easier to Understand**. Refactoring helps you to make your code more readable. When refactoring you have code that works but is not ideally structured. A little time spent refactoring can make the code better communicate its purpose. I'm not necessarily being altruistic about this. Often this future developer is me. As the code gets clearer, I find I can see things about the design that I could not see before.
  - **Refactoring Improves the Design of Software**. Without refactoring, the design of the program will decay. Regular refactoring helps code retain its shape.

1. Definición y Justificación de la Refactorización

- **Ventajas:**
  - **Refactoring Helps You Program Faster**. It takes a change of rhythm to make changes that make the code easier to understand. Refactoring helps you develop software more rapidly, because it stops the design of the system from decaying
  - **Refactoring Helps You Find Bugs**. However, I find that if I refactor code, I work deeply on understanding what the code does, and I put that new understanding right back into the code. By clarifying the structure of the program, I clarify certain assumptions I've made, to the point at which even I can't avoid spotting the bugs.

2. Catálogo de Refactorización

- **Catalog (by areas)**:
  - Condicionales: Remove Control Flag, Introduce Null Object, …
  - Variables Temporales: Replace Temp with Query, …
  - Parámetros: Replace Parameter with Explicit Methods, …
  - Cuerpo de Métodos: Inline Method, Extract Method, …
  - Encapsulación: Hide Method, Encapsulate Downcast, …
  - Cabeceras de Métodos: Parameterize Method, …
  - Gestión de Errores: Separate Query from Modifier, …
  - Nombrado: Replace Magic Number with Symbolic Constant, …
  - Relación de Herencia: Extract Superclass, Pull Up Method, …
  - Polimorfismo: Replace Conditional with Polymorphism, …
  - Responsabilidades: Move Method, Extract Class, …
  - Colaboraciones: Preserve Whole Object, Remove Middle Man, ...
  - Bibliotecas & GUI: Introduce Foreign Method, …

2. Catálogo de Refactorización

- **Catalog (by areas)**:

  You don't become a software craftsman by learning a list of heuristics. Professionalism and craftsmanship come from values that drive disciplines.

2. Catálogo de Refactorización

- **Refactoring for extracting a class**
  - Decide how to split the responsibilities of the class.
  - Create a new class to express the split-off responsibilities.
    - ?rarr; *If the responsibilities of the old class no longer match its name, rename the old class.*
  - Make a link from the old to the new class.
    - ?rarr; *You may need a two-way link. But don't make the back link until you find you need it.*
  - Use Move Field on each field you wish to move.
  - **Compile and test after each move**.
  - Use Move Method to move methods over from old to new. Start with lower-level methods (called rather than calling) and build to the higher level.
  - **Compile and test after each move**.
  - Review and reduce the interfaces of each class.
    - ?rarr; *If you did have a two-way link, examine to see whether it can be made one way.*
  - Decide whether to expose the new class. If you do expose the class, decide whether to expose it as a reference object or as an immutable value object.

2. Catálogo de Refactorización: Condicionales

- *Name*: **Replace Nested Conditional with Guard Clauses**
- *Motivation*: A method has conditional behavior that does not make clear the normal path of execution
- *Mechanics*: 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;
};
```

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

2. Catálogo de Refactorización: Condicionales

- *Name*: **Consolidate Conditional Expression**
- *Motivation*: You have a sequence of conditional tests with the same result.
- *Mechanics*: 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;
    // compute the disability amount
```

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

2. Catálogo de Refactorización: Condicionales

- *Name*: **Consolidate Duplicate Conditional Fragments**
- *Motivation*: The same fragment of code is in all branches of a conditional expression.
- *Mechanics*: 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();
```

2. Catálogo de Refactorización: Condicionales

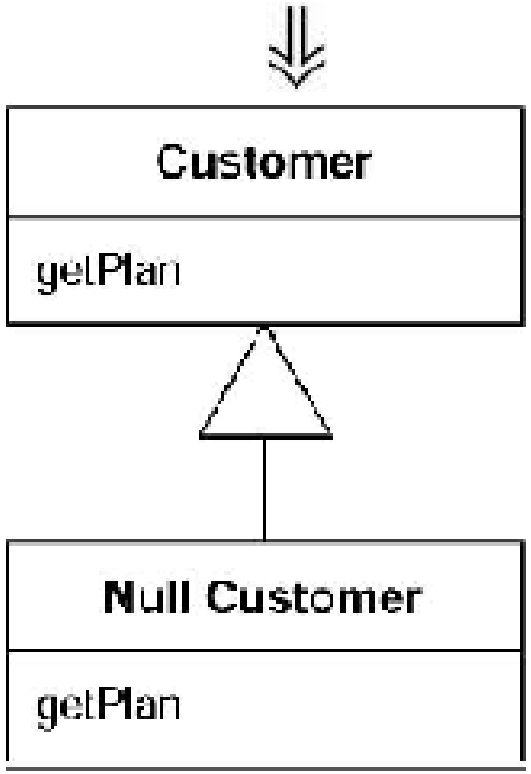
- *Name*: **Remove Control Flag**
- *Motivation*: You have a variable that is acting as a control flag for a series of boolean expressions
- *Mechanics*: Use a break or return instead

```
set done to false
while not done
  if (condition)
      do something
      set done to true
  next step of loop
```

2. Catálogo de Refactorización: Condicionales

- *Name*: **Introduce Null Object**
- *Motivation*: You have repeated checks for a null value
- *Mechanics*: Replace the null value with a null object

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```



2. Catálogo de Refactorización: Condicionales

- *Name*: **Decompose Conditional**
- *Motivation*: You have a complicated conditional (if-then-else) statement
- *Mechanics*: 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);
```

2. Catálogo de Refactorización: Variables Temporales

- *Name*: **Split Temporary Variable**
- *Motivation*: You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable
- *Mechanics*: 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);
```

2. Catálogo de Refactorización: Variables Temporales

- *Name*: **Introduce Explaining Variable**
- *Motivation*: You have a complicated expression
- *Mechanics*: Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

```
  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
    }
```

2. Catálogo de Refactorización: Variables Temporales

- *Name*: **Inline Temp**
- *Motivation*: You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings
- *Mechanics*: Replace all references to that temp with the expression

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

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

2. Catálogo de Refactorización: Variables Temporales

- *Name*: **Replace Temp with Query**
- *Motivation*: You are using a temporary variable to hold the result of an expression
- *Mechanics*: 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;
}
```

2. Catálogo de Refactorización: Parámetros

- *Name*: **Remove Assignments to Parameters**
- *Motivation*: The code assigns to a parameter
- *Mechanics*: 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;
```

2. Catálogo de Refactorización: Parámetros

- *Name*: **Replace Parameter with Method**
- *Motivation*: An object invokes a method, then passes the result as a parameter for a method. The receiver can also invoke this method.
- *Mechanics*: Remove the parameter and let the receiver invoke the method

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice, discountLevel);
```

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice (basePrice);
```

2. Catálogo de Refactorización: Parámetros

- *Name*: **Replace Parameter with Explicit Methods**
- *Motivation*: You have a method that runs different code depending on the values of an enumerated parameter.
- *Mechanics*: Create a separate method for each value of the parameter

```
void setValue (String name, int value) {
    if (name.equals("height"))
        _height = value;
    if (name.equals("width"))
        _width = value;
    Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg) {
    _height = arg;
}
void setWidth (int arg) {
    _width = arg;
}
```

2. Catálogo de Refactorización: Cuerpo de Método

- *Name*: **Substitute Algorithm**
- *Motivation*: You want to replace an algorithm with one that is clearer
- *Mechanics*: 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 "";
}
```

2. Catálogo de Refactorización: Cuerpo de Método

- *Name*: **Inline Method**
- *Motivation*: A method's body is just as clear as its name
- *Mechanics*: 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;
}
```

2. Catálogo de Refactorización: Cuerpo de Método

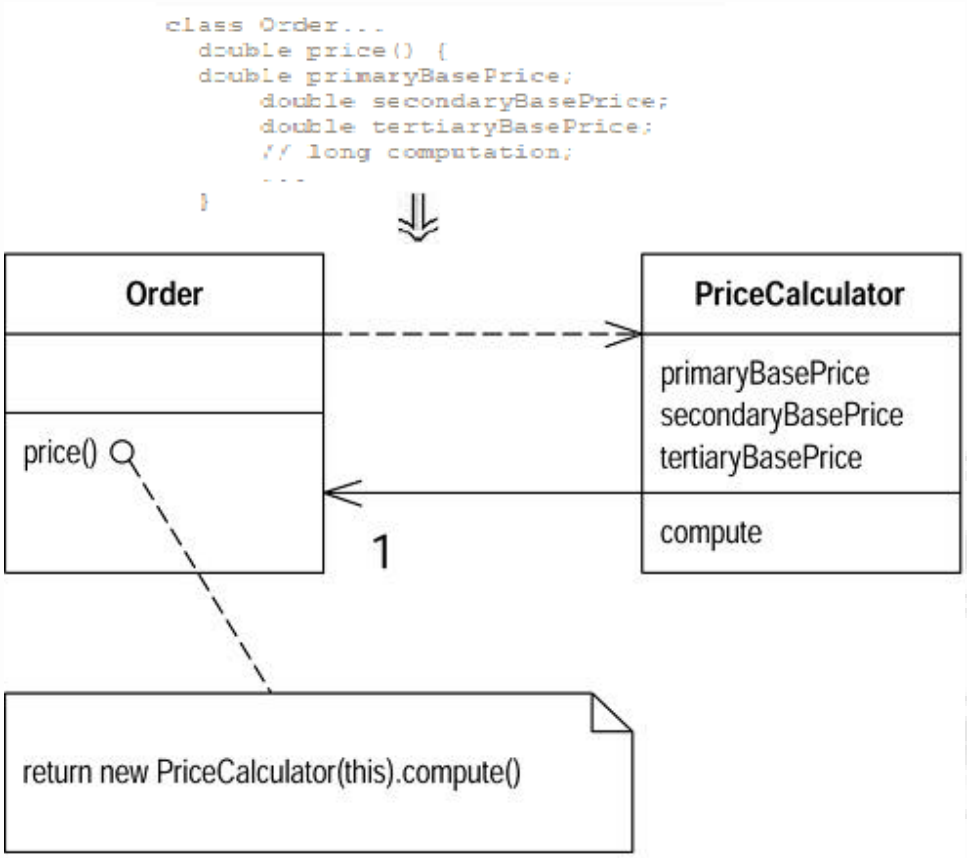
- *Name*: **Extract Method**
- *Motivation*: You have a code fragment that can be grouped together
- *Mechanics*: Turn the fragment into a method whose name explains the purpose of the method

```
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);
}
```

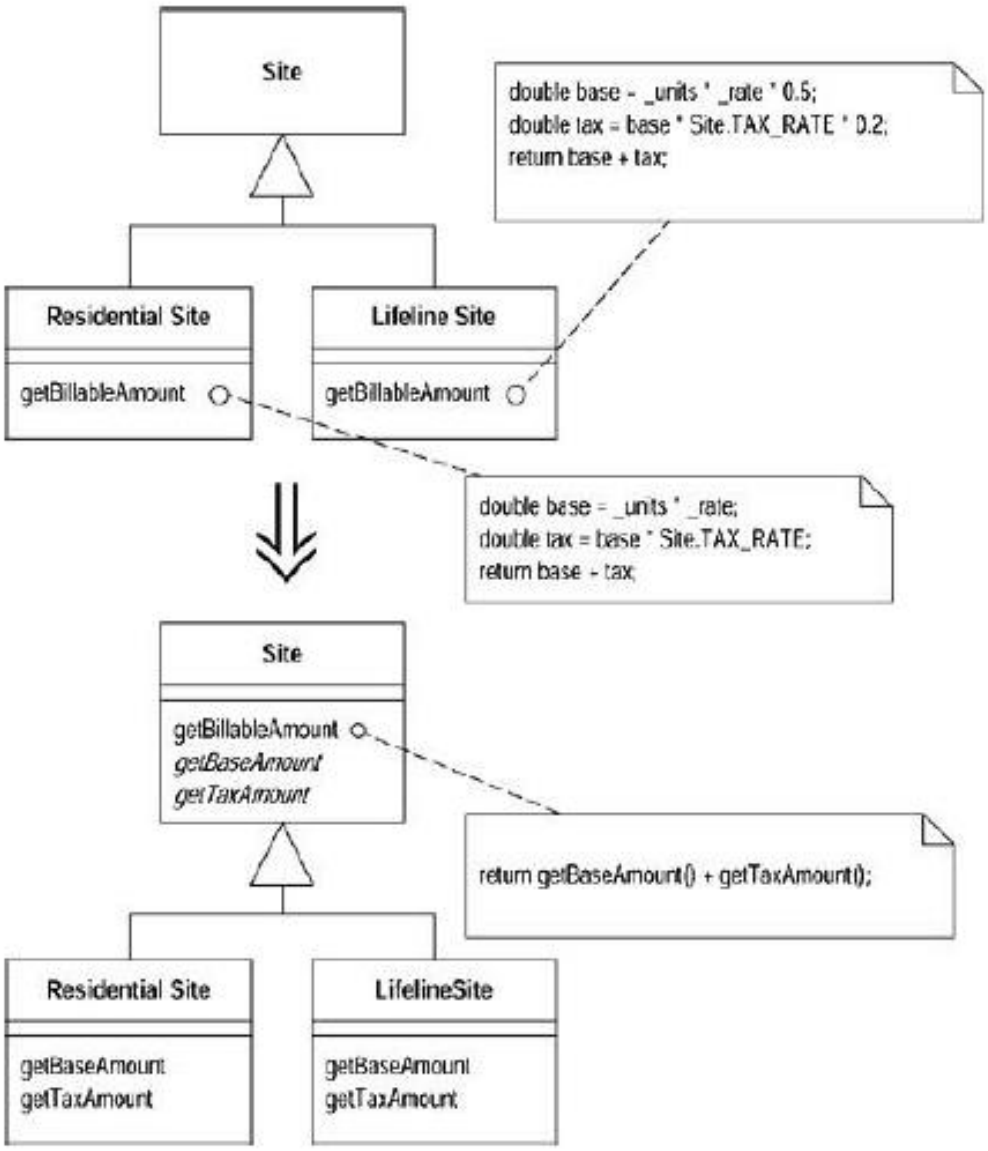
2. Catálogo de Refactorización: Cuerpo de Método

- *Name*: **Replace Method with Method Object**
- *Motivation*: You have a long method that uses local variables in such a way that you cannot apply Extract Method.
- *Mechanics*: 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.



2. Catálogo de Refactorización: Cuerpo de Método

- *Name*: **Form Template Method**
- *Motivation*: You have two methods in subclasses that perform similar steps in the same order, yet the steps are different
- *Mechanics*: Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.



2. Catálogo de Refactorización: Resumen I

- **Condicionales:**
  - Replace Nested Conditional with Guard Clauses
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Introduce Null Object
  - Decompose Conditional
- **Variables Temporales**
  - Split Temporary Variable
  - Introduce Explaining Variable **vs** Inline Temp
  - Replace Temp with Query
- **Parámetros:**
  - Remove Assignments to Parameters
  - Replace Parameter with Method **&** Replace Parameter with Explicit Methods
- **Cuerpo de Métodos**
  - Substitute Algorithm
  - Inline Method **vs** Extract Method
  - Replace Method with Method Object
  - Form Template Method

2. Catálogo de Refactorización: Encapsulación

- *Name*: **Encapsulate Field**
- *Motivation*: There is a public field.
- *Mechanics*: Make it private and provide accessors

```
public String _name
```

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

2. Catálogo de Refactorización: Encapsulación

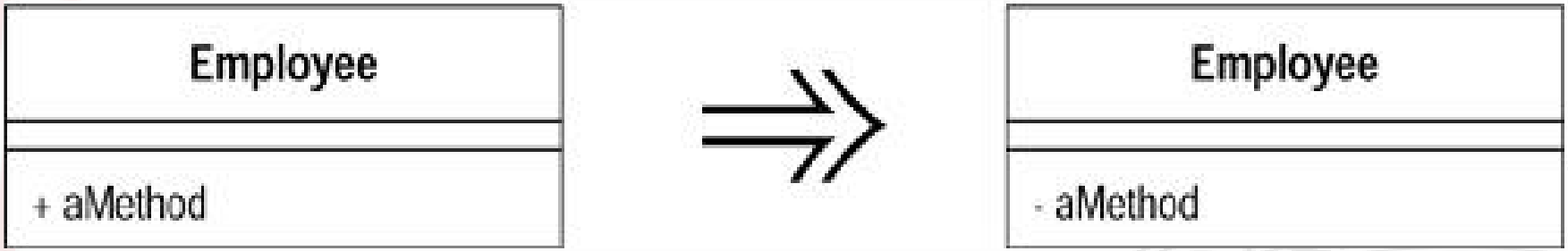
- *Name*: **Self Encapsulate Field**
- *Motivation*: You are accessing a field directly, but the coupling to the field is becoming awkward
- *Mechanics*: Create getting and setting methods for the field and use only those to access the field.

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

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

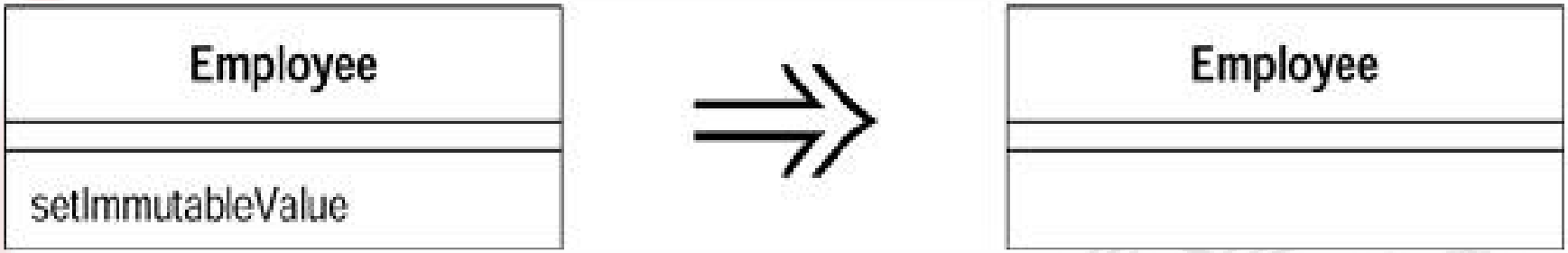
2. Catálogo de Refactorización: Encapsulación

- *Name*: **Hide Method**
- *Motivation*: A method is not used by any other class.
- *Mechanics*: Make the method private.



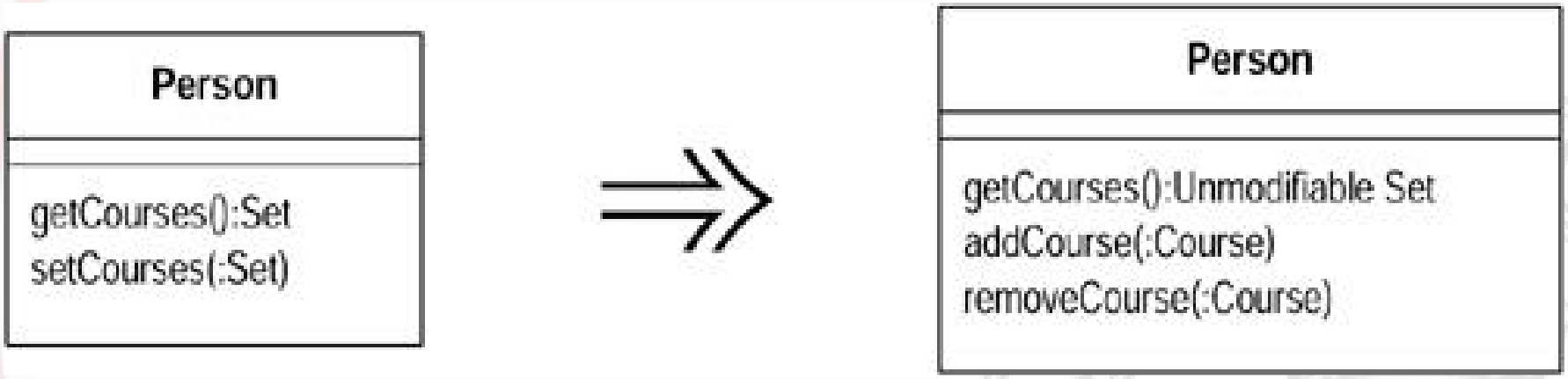
2. Catálogo de Refactorización: Encapsulación

- *Name*: **Remove Setting Method**
- *Motivation*: A field should be set at creation time and never altered.
- *Mechanics*: Remove any setting method for that field



2. Catálogo de Refactorización: Encapsulación

- *Name*: **Encapsulate Collection**
- *Motivation*: A method returns a collection
- *Mechanics*: Make it return a read-only view and provide add/remove methods



2. Catálogo de Refactorización: Encapsulación

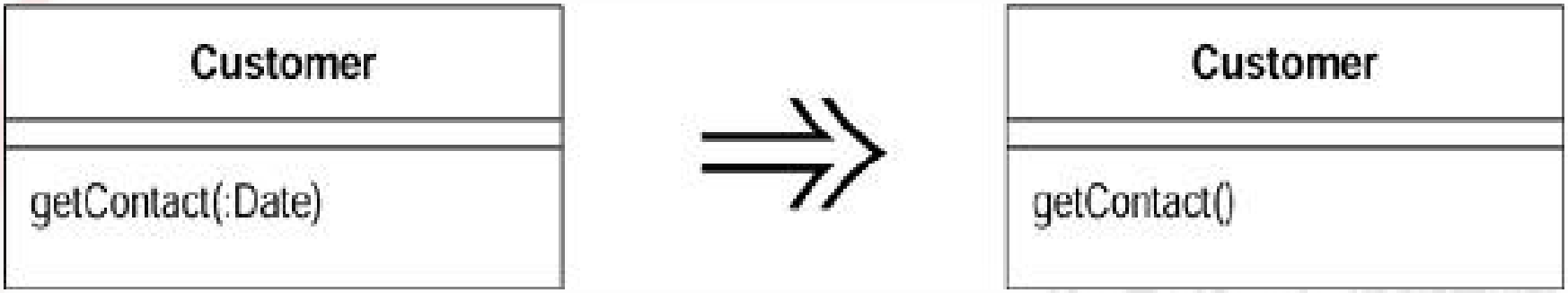
- *Name*: **Encapsulate Downcast**
- *Motivation*: A method returns an object that needs to be downcasted by its callers
- *Mechanics*: Move the downcast to within the method

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

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

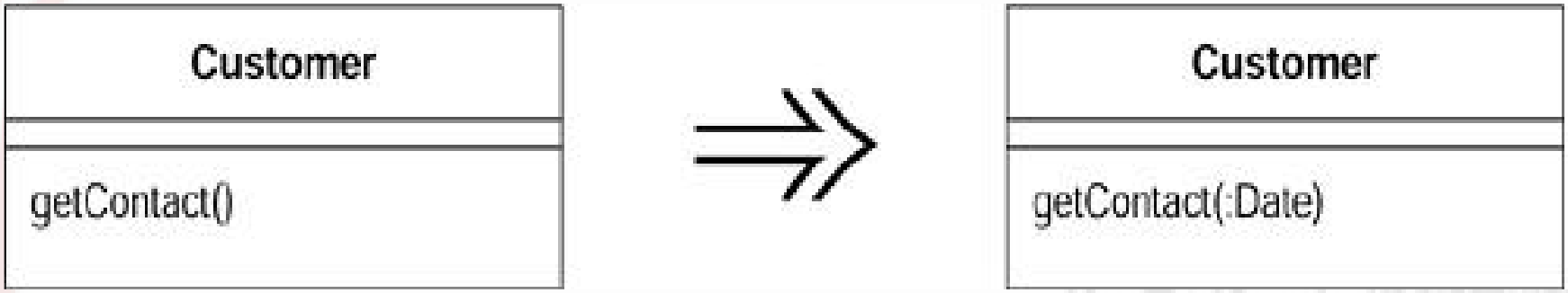
2. Catálogo de Refactorización: Cabecera de Método

- *Name*: **Remove Parameter**
- *Motivation*: A parameter is no longer used by the method body.
- *Mechanics*: Remove it.



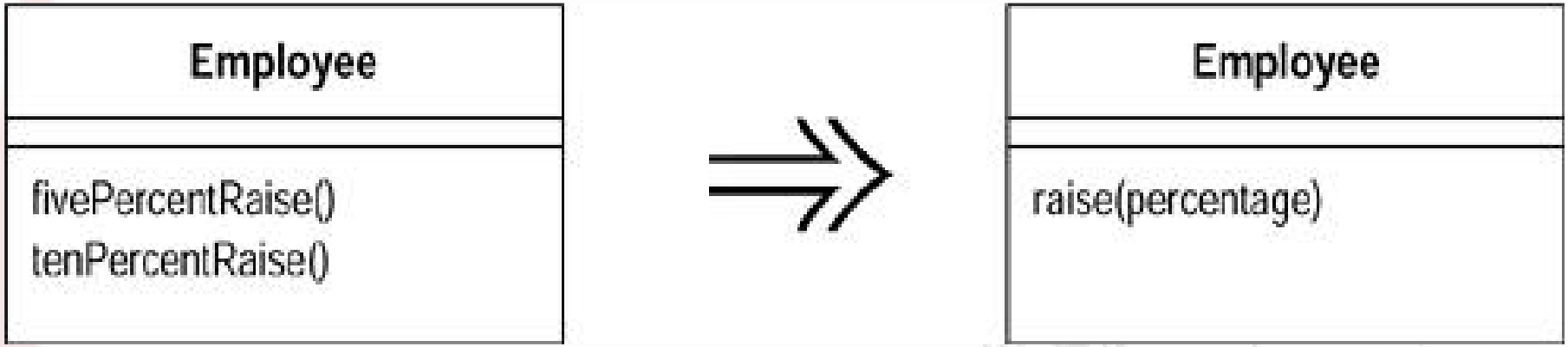
2. Catálogo de Refactorización: Cabecera de Método

- *Name*: **Add Parameter**
- *Motivation*: A method needs more information from its caller
- *Mechanics*: Add a parameter for an object that can pass on this information



2. Catálogo de Refactorización: Cabecera de Método

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



2. Catálogo de Refactorización: Gestión de Errores

- *Name*: **Introduce Assertion**
- *Motivation*: A section of code assumes something about the state of the program
- *Mechanics*: Make the assumption explicit with an assertion

```
double getExpenseLimit() {
    // should have either expense limit or a primary project
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

```
double getExpenseLimit() {
    Assert.isTrue (_expenseLimit != NULL_EXPENSE || _primaryProject
!= null);
    return (_expenseLimit != NULL_EXPENSE) ?
        _expenseLimit:
        _primaryProject.getMemberExpenseLimit();
}
```

2. Catálogo de Refactorización: Gestión de Errores

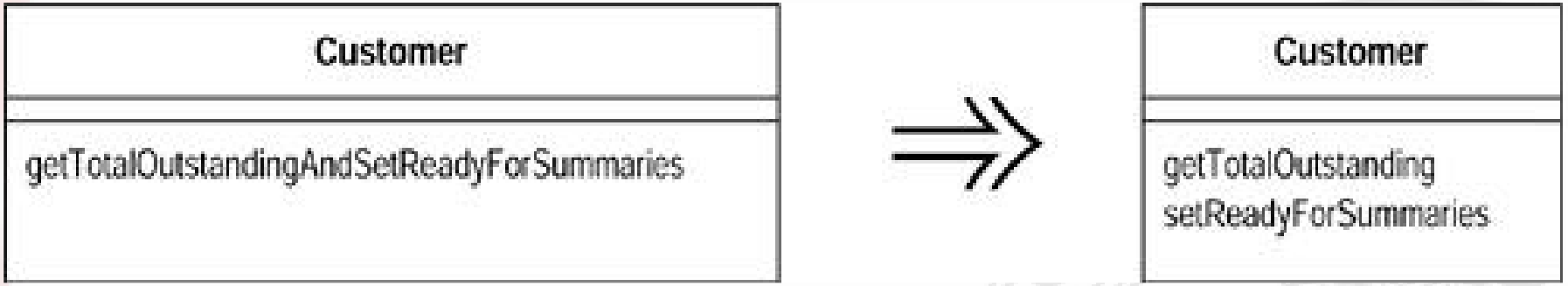
- *Name*: **Replace Error Code with Exception**
- *Motivation*: A method returns a special code to indicate an error.
- *Mechanics*: Throw an exception instead.

```
int withdraw(int amount) {
    if (amount > _balance)
        return -1;
    else {
        _balance -= amount;
        return 0;
    }
}
```

```
void withdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
    _balance -= amount;
}
```

2. Catálogo de Refactorización: Gestión de Errores

- *Name*: **Separate Query from Modifier**
- *Motivation*: You have a method that returns a value but also changes the state of an object
- *Mechanics*: Create two methods, one for the query and one for the modification



2. Catálogo de Refactorización: Gestión de Errores

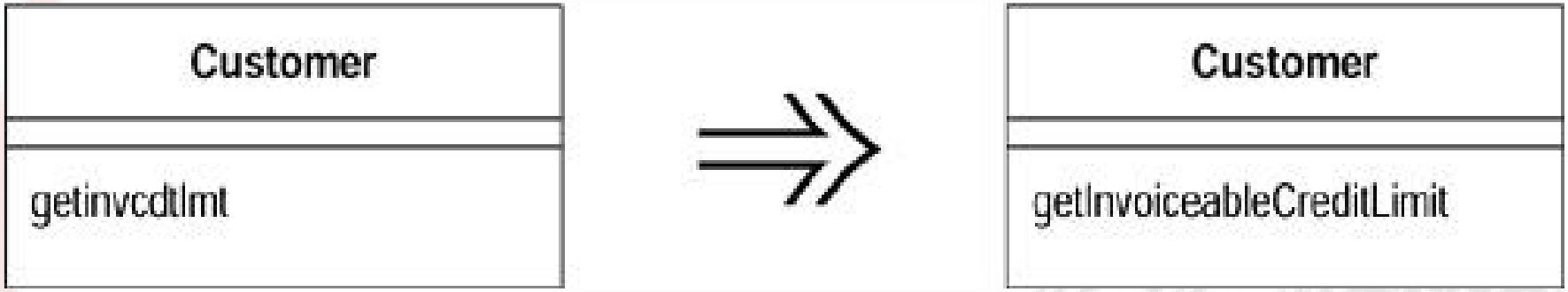
- *Name*: **Replace Exception with Test**
- *Motivation*: You are throwing a checked exception on a condition the caller could have checked first
- *Mechanics*: Change the caller to make the test first

```
double getValueForPeriod (int periodNumber) {
    try {
        return _values[periodNumber];
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0;
    }
}
```

```
double getValueForPeriod (int periodNumber) {
    if (periodNumber >= _values.length) return 0;
    return _values[periodNumber];
}
```

2. Catálogo de Refactorización: Nombrado

- *Name*: **Rename Method**
- *Motivation*: The name of a method does not reveal its purpose
- *Mechanics*: Change the name of the method.



2. Catálogo de Refactorización: Nombrado

- *Name*: **Replace Magic Number with Symbolic Constant**
- *Motivation*: You have a literal number with a particular meaning
- *Mechanics*: 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;
```

2. Catálogo de Refactorización: Resumen II

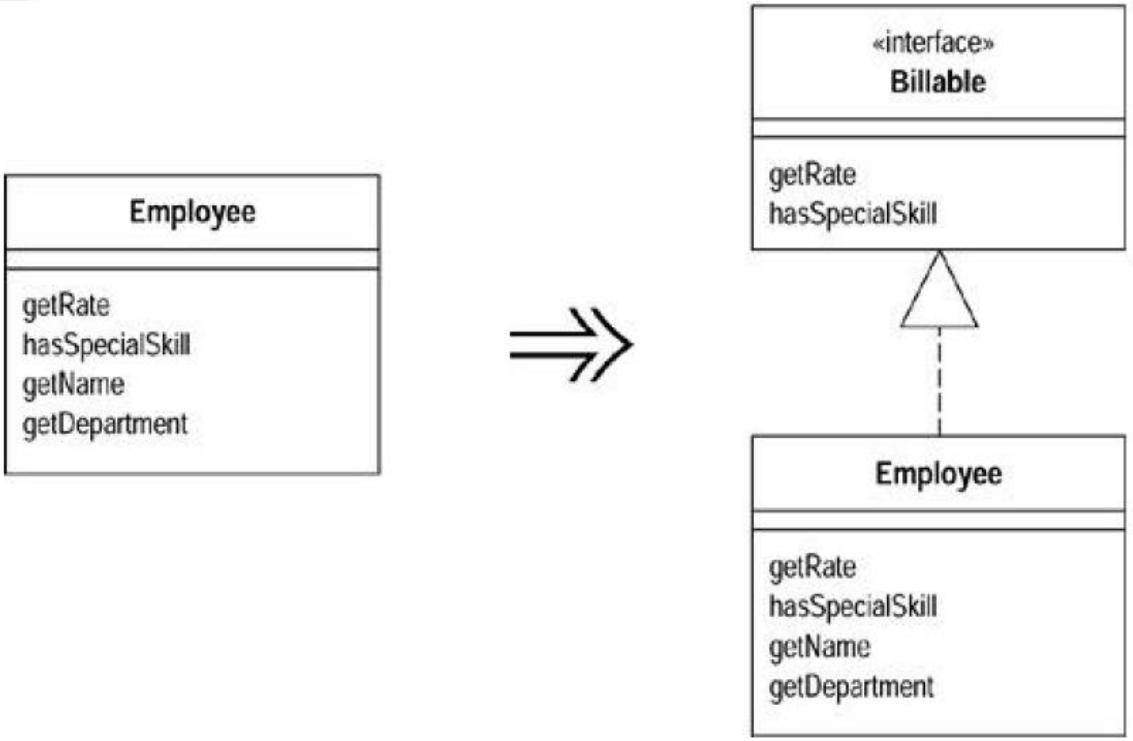
- **Encapsulación:**
  - Encapsulate Field & Self Encapsulate Field
  - Hide Method & Remove Setting Method
  - Encapsulate Collection Variables Temporales
  - Encapsulate Downcast
- **Cabecera de Métodos:**
  - Remove Parameter **vs** Add Parameter
  - Parameterize Method
- **Gestión de Errores**
  - Introduce Assertion
  - Replace Error Code with Exception
  - Separate Query from Modifier
  - Replace Exception with Test
- **Nombrado:**
  - Rename Method
  - Replace Magic Number with Symbolic Constant

2. Catálogo de Refactorización: Resumen I

- **Condicionales:**
  - Replace Nested Conditional with Guard Clauses
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Introduce Null Object
  - Decompose Conditional
- **Variables Temporales**
  - Split Temporary Variable
  - Introduce Explaining Variable **vs** Inline Temp
  - Replace Temp with Query
- **Parámetros:**
  - Remove Assignments to Parameters
  - Replace Parameter with Method **&** Replace Parameter with Explicit Methods
- **Cuerpo de Métodos**
  - Substitute Algorithm
  - Inline Method **vs** Extract Method
  - Replace Method with Method Object
  - Form Template Method

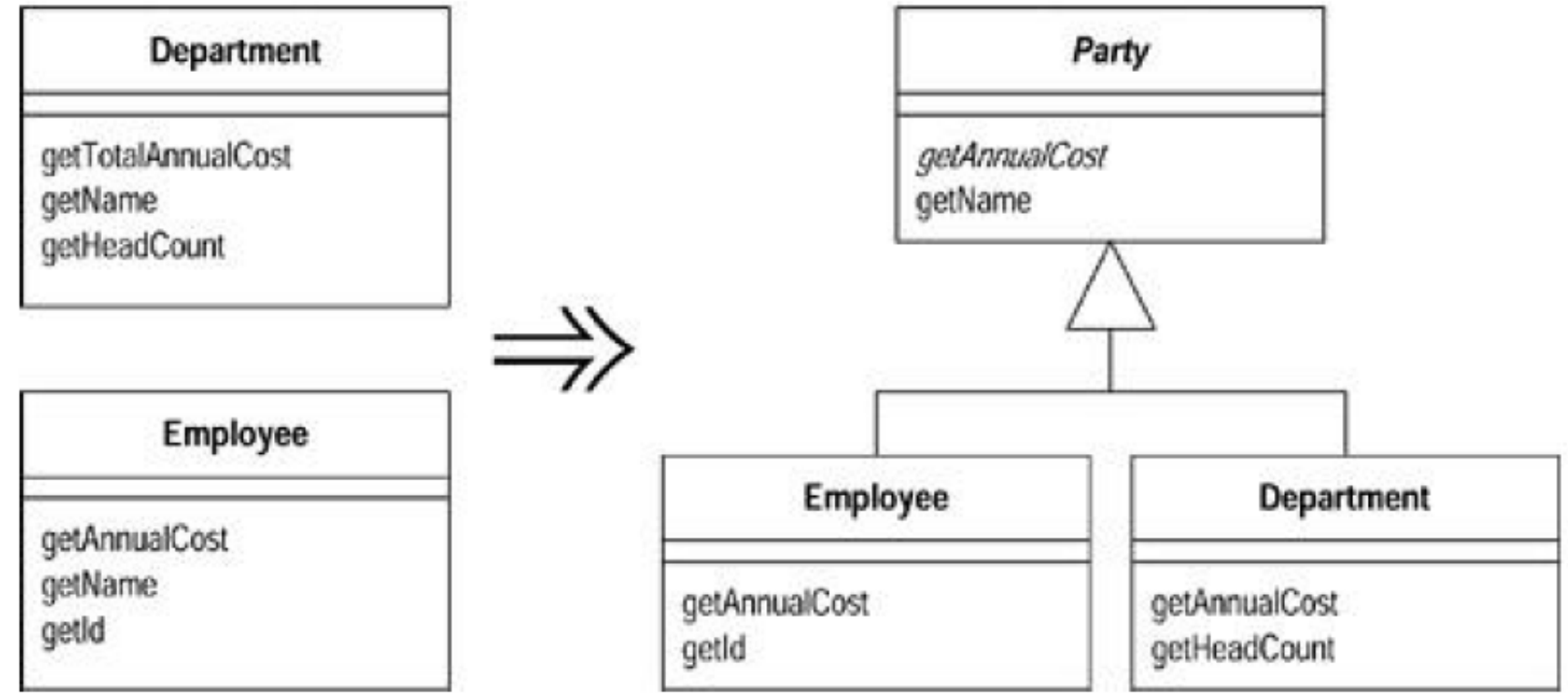
2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Extract Interface**
- *Motivation*: Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.
- *Mechanics*: Extract the subset into an interface



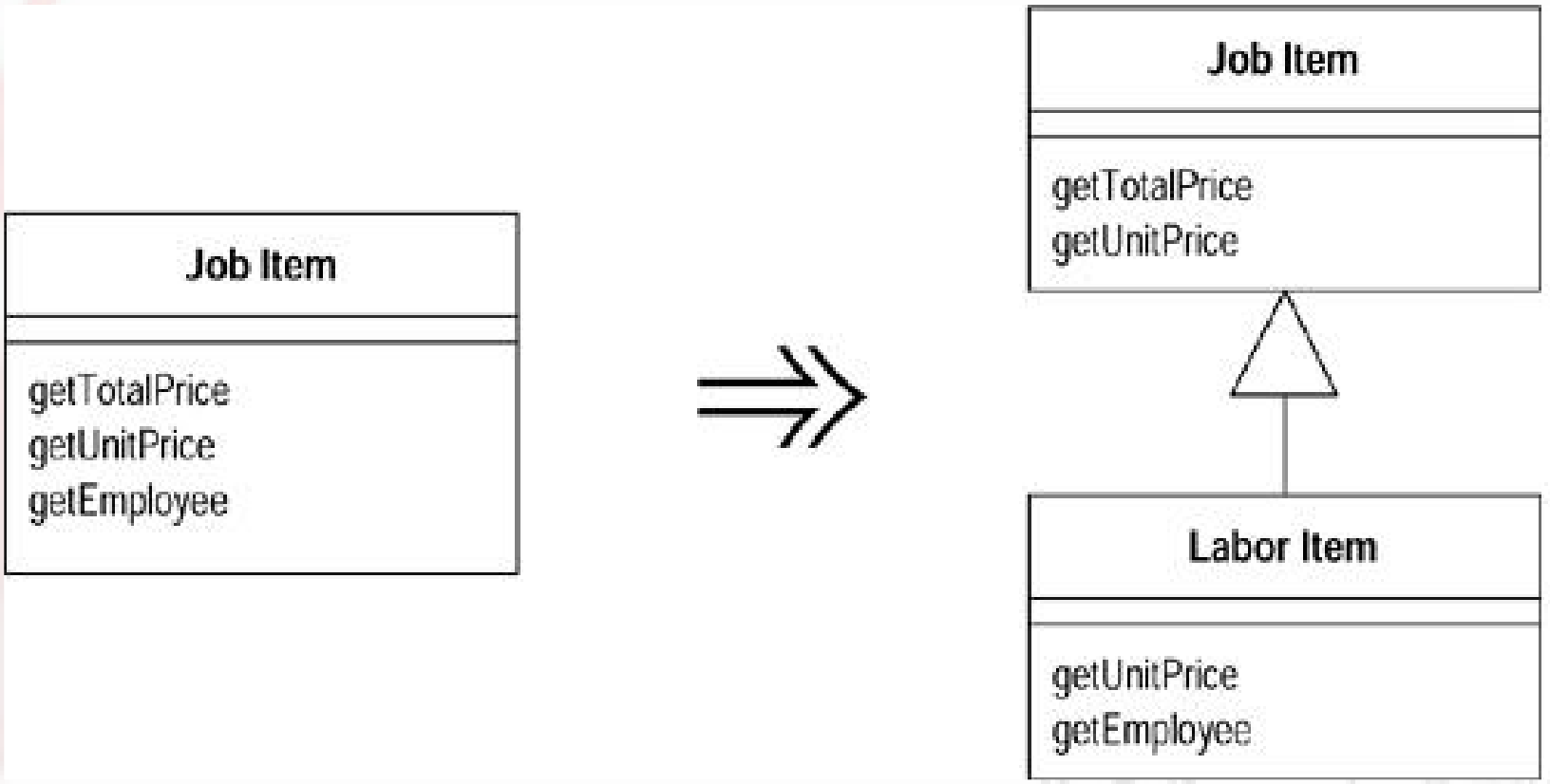
2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Extract Superclass**
- *Motivation*: You have two classes with similar features
- *Mechanics*: Create a superclass and move the common features to the superclass



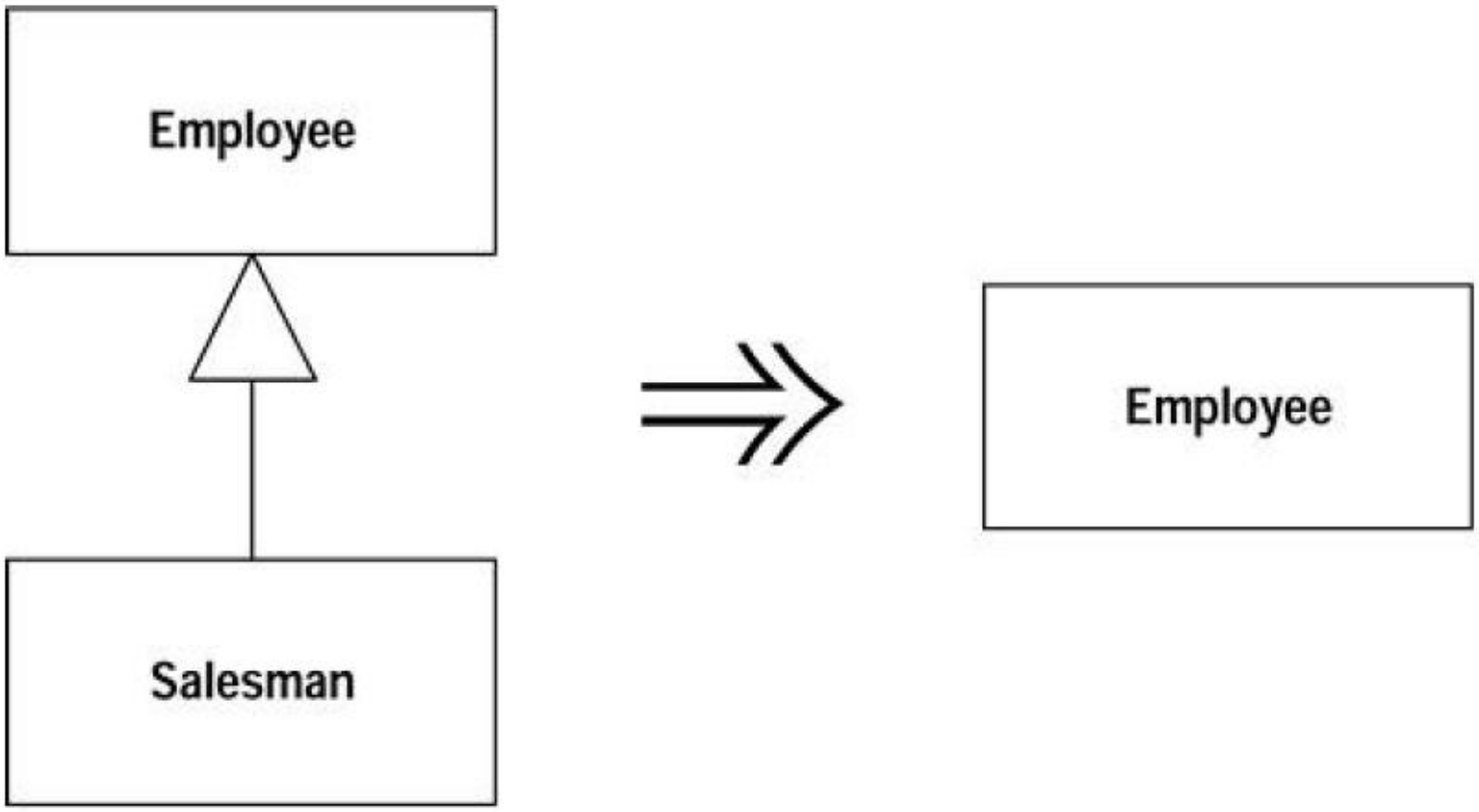
2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Extract Subclass**
- *Motivation*: A class has features that are used only in some instances
- *Mechanics*: Create a subclass for that subset of features



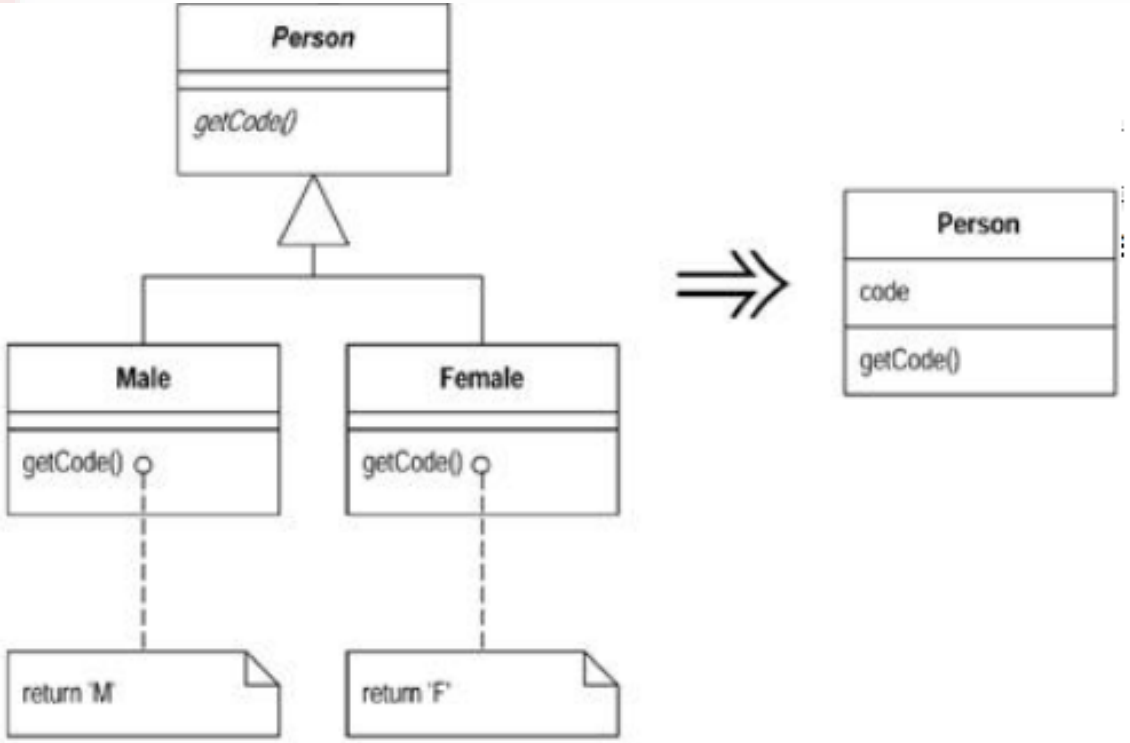
2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Collapse Hierarchy**
- *Motivation*: A superclass and subclass are not very different
- *Mechanics*: Merge them together



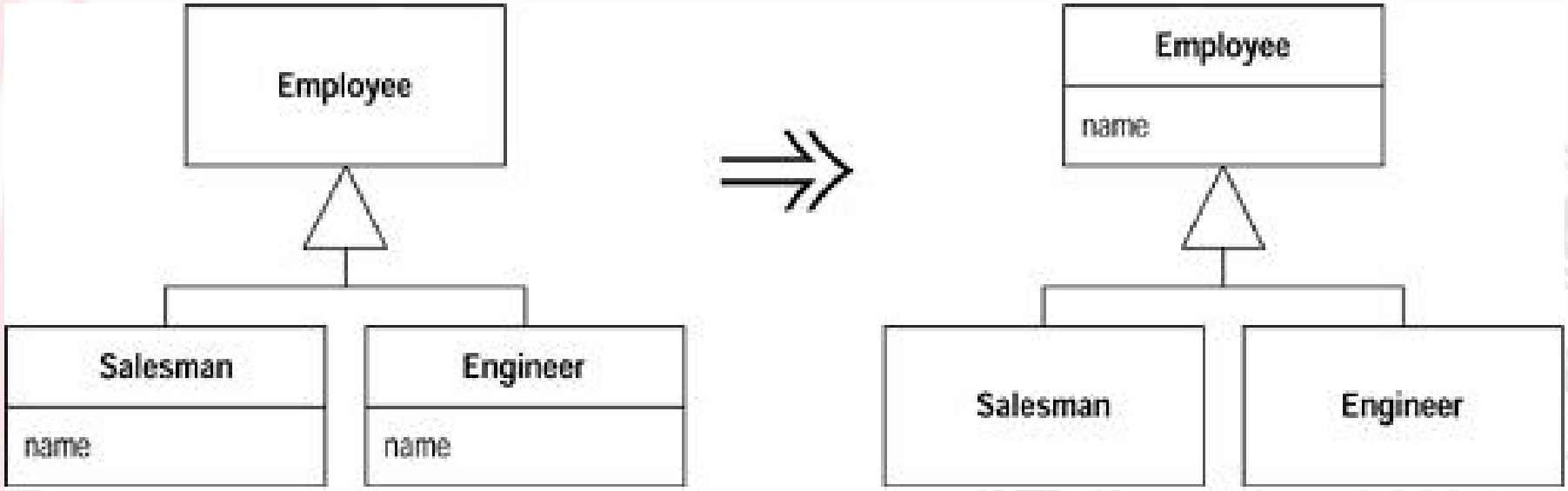
2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Replace Subclass with Fields**
- *Motivation*: You have subclasses that vary only in methods that return constant data.
- *Mechanics*: Change the methods to superclass fields and eliminate the subclasses.



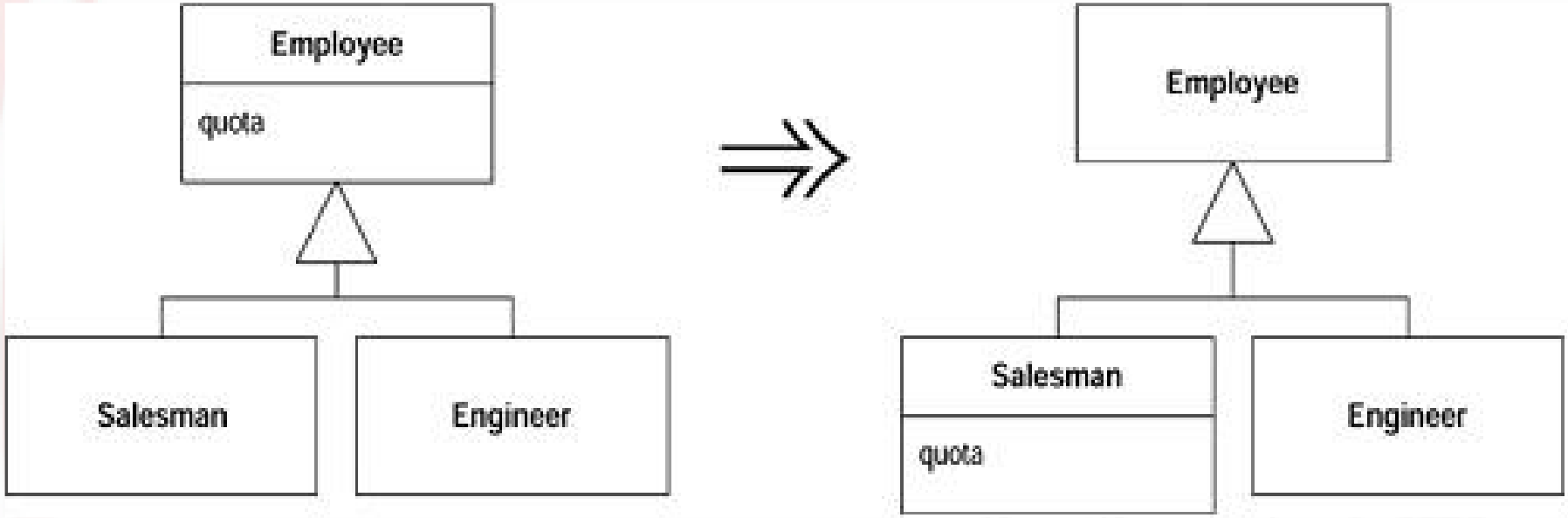
2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Pull Up Field**
- *Motivation*: Two subclasses have the same field.
- *Mechanics*: Move the field to the superclass



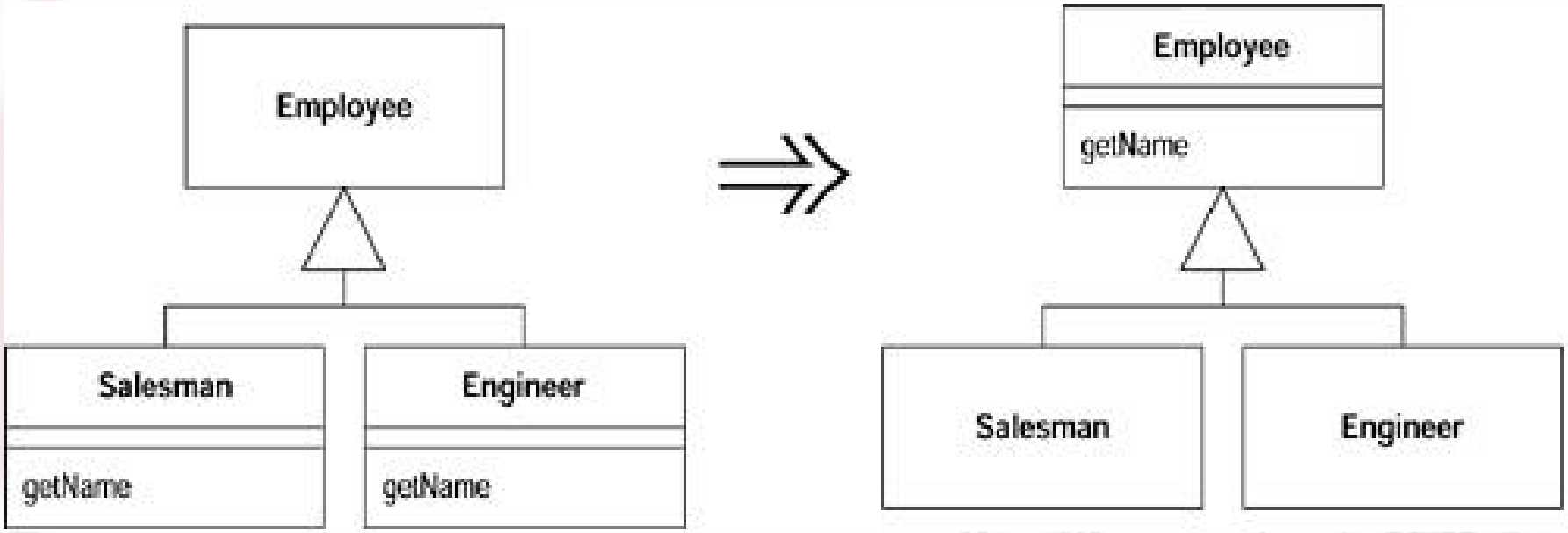
2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Push Down Field**
- *Motivation*: A field is used only by some subclasses
- *Mechanics*: Move the field to the superclass



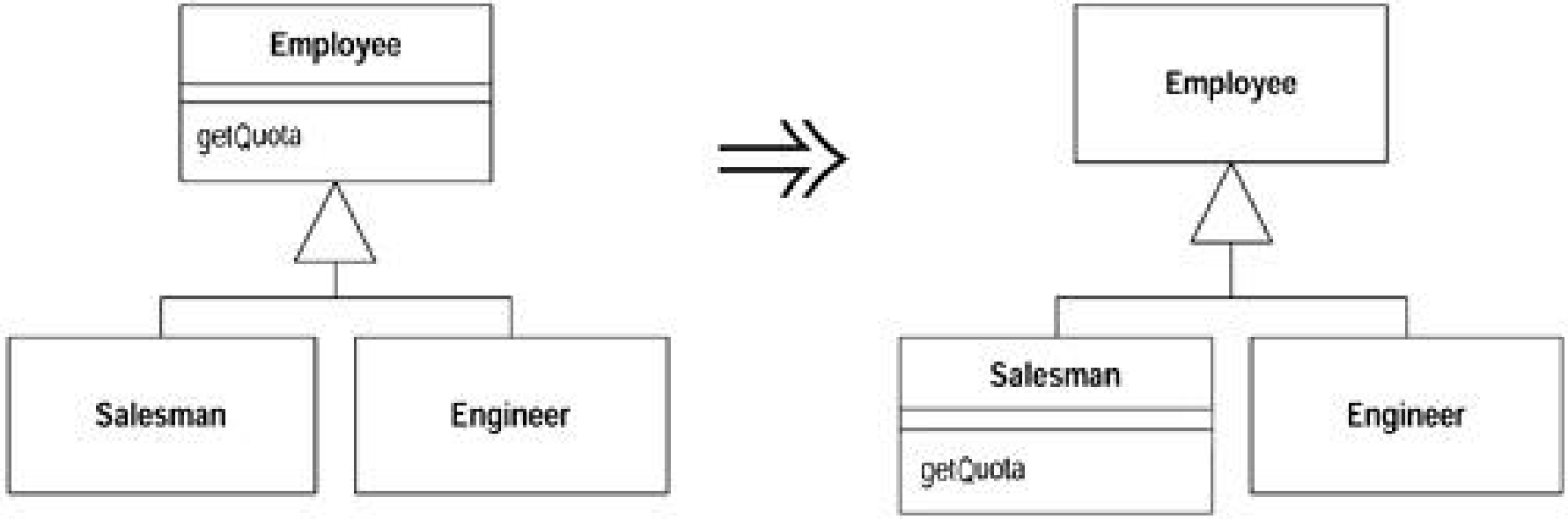
2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Pull Up Method**
- *Motivation*: You have methods with identical results on subclasses
- *Mechanics*: Move them to the superclass.



2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Push Down Method**
- *Motivation*: Behavior on a superclass is relevant only for some of its subclasses
- *Mechanics*: Move it to those subclasses.



2. Catálogo de Refactorización: Relación de Herencia

- *Name*: **Pull Up Constructor Body**
- *Motivation*: You have constructors on subclasses with mostly identical bodies
- *Mechanics*: Create a superclass constructor; call this from the subclass methods.

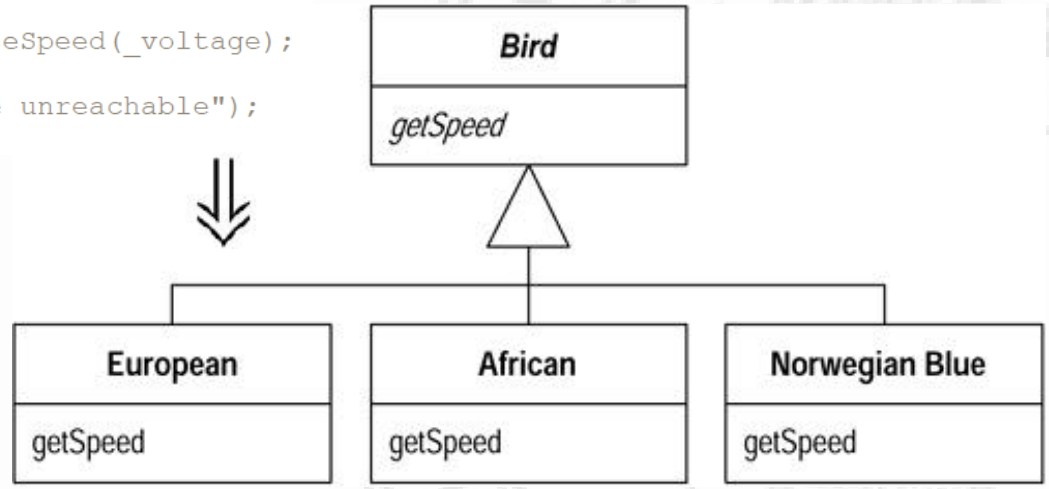
```
class Manager extends Employee...
  public Manager (String name, String id, int grade) {
       _name = name;
       _id = id;
       _grade = grade;
  }
```

```
  public Manager (String name, String id, int grade) {
       super (name, id);
       _grade = grade;
  }
```

2. Catálogo de Refactorización: Polimorfismo

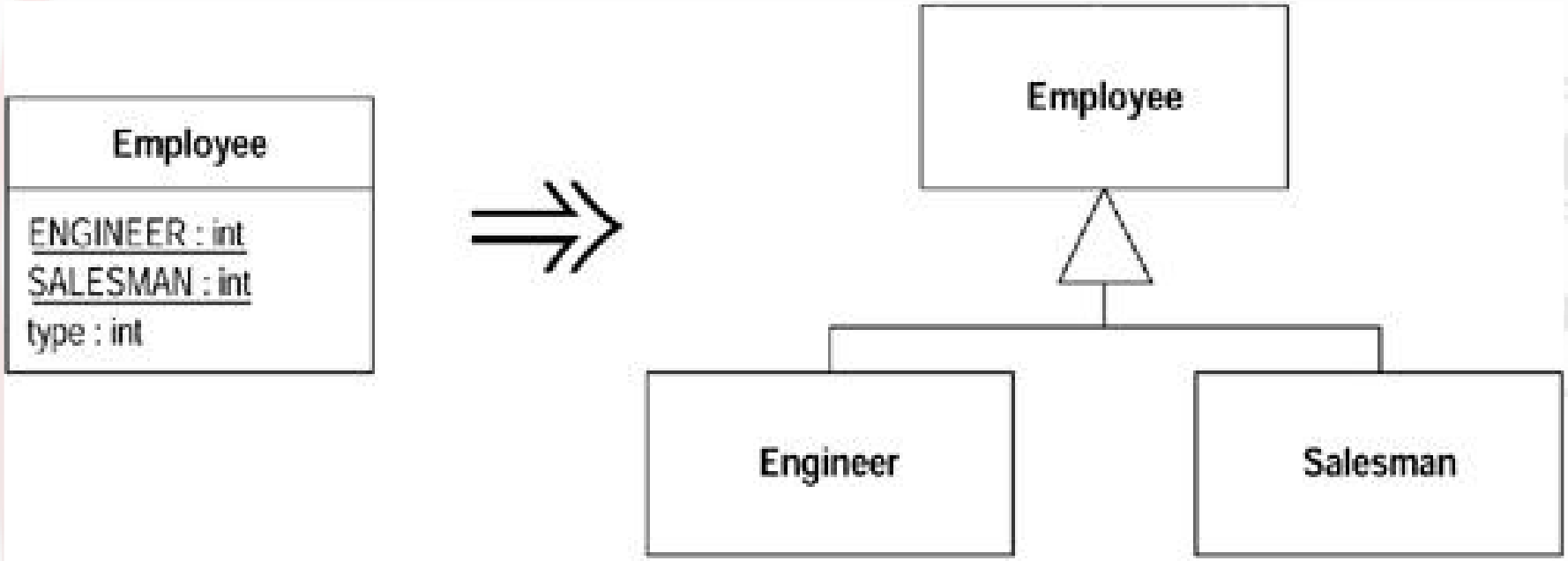
- *Name*: **Replace Conditional with Polymorphism**
- *Motivation*: You have a conditional that chooses different behavior depending on the type of an object
- *Mechanics*: Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract

```
   double getSpeed() {
       switch (_type) {
           case EUROPEAN:
              return getBaseSpeed();
           case AFRICAN:
              return getBaseSpeed() - getLoadFactor() *
_numberOfCoconuts;
           case NORWEGIAN_BLUE:
              return (_isNailed) ? 0 : getBaseSpeed(_voltage);
       }
       throw new RuntimeException ("Should be unreachable");
   }
```



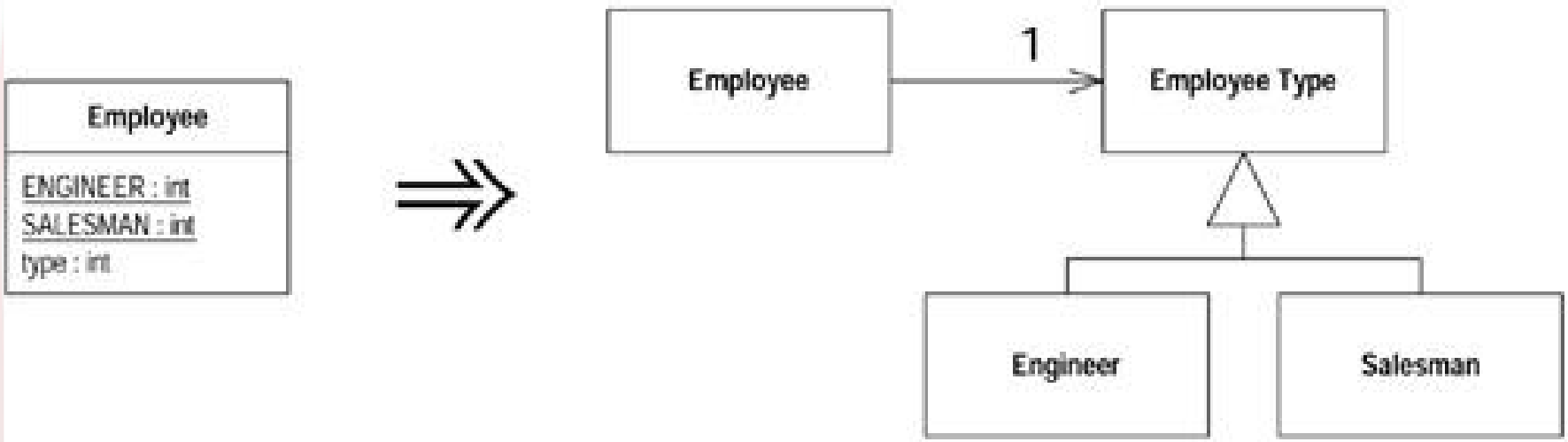
2. Catálogo de Refactorización: Polimorfismo

- *Name*: **Replace Type Code with Subclasses**
- *Motivation*: You have an immutable type code that affects the behavior of a class
- *Mechanics*: Replace the type code with subclasses



2. Catálogo de Refactorización: Polimorfismo

- *Name*: **Replace Type Code with State/Strategy**
- *Motivation*: You have a type code that affects the behavior of a class, but you cannot use subclassing
- *Mechanics*: Replace the type code with a state object



2. Catálogo de Refactorización: Polimorfismo

- *Name*: **Replace Constructor with Factory Method**
- *Motivation*: You want to do more than simple construction when you create an object
- *Mechanics*: Replace the constructor with a factory method

```
Employee (int type) {
    _type = type;
}
```

```
static Employee create(int type) {
    return new Employee(type);
}
```

2. Catálogo de Refactorización: Resumen III

- **Relación de Herencia:**
  - Extract Interface **&** Extract Superclass
  - Extract Subclass **vs** Collapse Hierarchy
  - Replace Subclass with Fields
  - Pull Up Field **vs** Push Down Field
  - Pull Up Method **vs** Push Down Method **&** Pull Up Constructor Body
- **Polimorfismo:**
  - Replace Conditional with Polymorphism
  - Replace Type Code with Subclasses **vs** Replace Type Code with State/Strategy
  - Replace Constructor with Factory Method

2. Catálogo de Refactorización: Resumen II

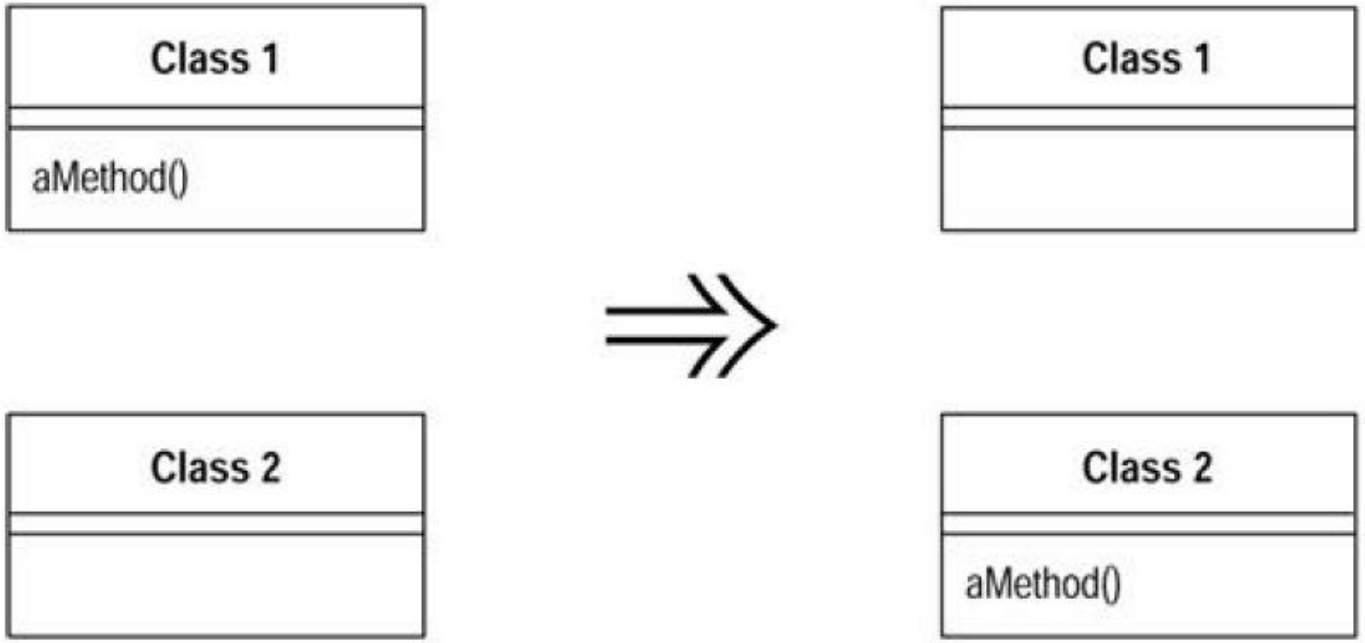
- **Encapsulación:**
  - Encapsulate Field & Self Encapsulate Field
  - Hide Method & Remove Setting Method
  - Encapsulate Collection Variables Temporales
  - Encapsulate Downcast
- **Cabecera de Métodos:**
  - Remove Parameter **vs** Add Parameter
  - Parameterize Method
- **Gestión de Errores**
  - Introduce Assertion
  - Replace Error Code with Exception
  - Separate Query from Modifier
  - Replace Exception with Test
- **Nombrado:**
  - Rename Method
  - Replace Magic Number with Symbolic Constant

2. Catálogo de Refactorización: Resumen I

- **Condicionales:**
  - Replace Nested Conditional with Guard Clauses
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Introduce Null Object
  - Decompose Conditional
- **Variables Temporales**
  - Split Temporary Variable
  - Introduce Explaining Variable **vs** Inline Temp
  - Replace Temp with Query
- **Parámetros:**
  - Remove Assignments to Parameters
  - Replace Parameter with Method **&** Replace Parameter with Explicit Methods
- **Cuerpo de Métodos**
  - Substitute Algorithm
  - Inline Method **vs** Extract Method
  - Replace Method with Method Object
  - Form Template Method

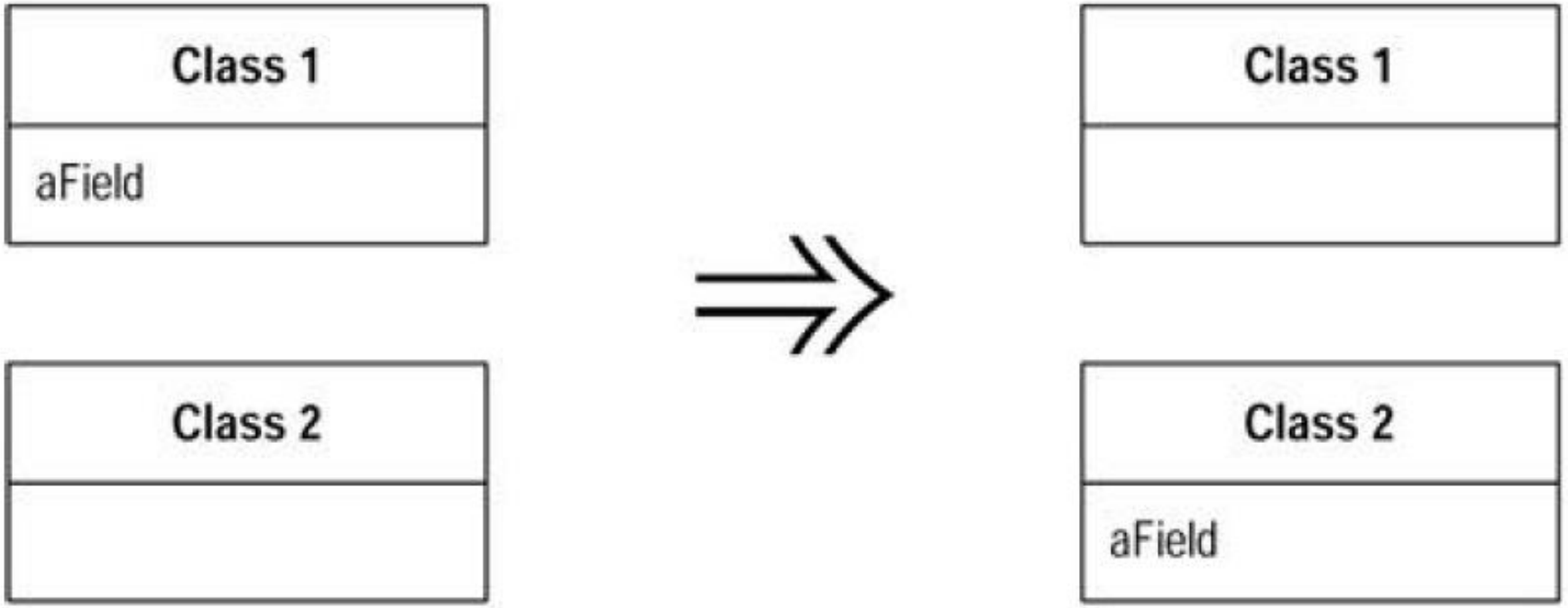
2. Catálogo de Refactorización: Responsabilidades

- *Name*: **Move Method**
- *Motivation*: A method is, or will be, using or used by more features of another class than the class on which it is defined.
- *Mechanics*: 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.



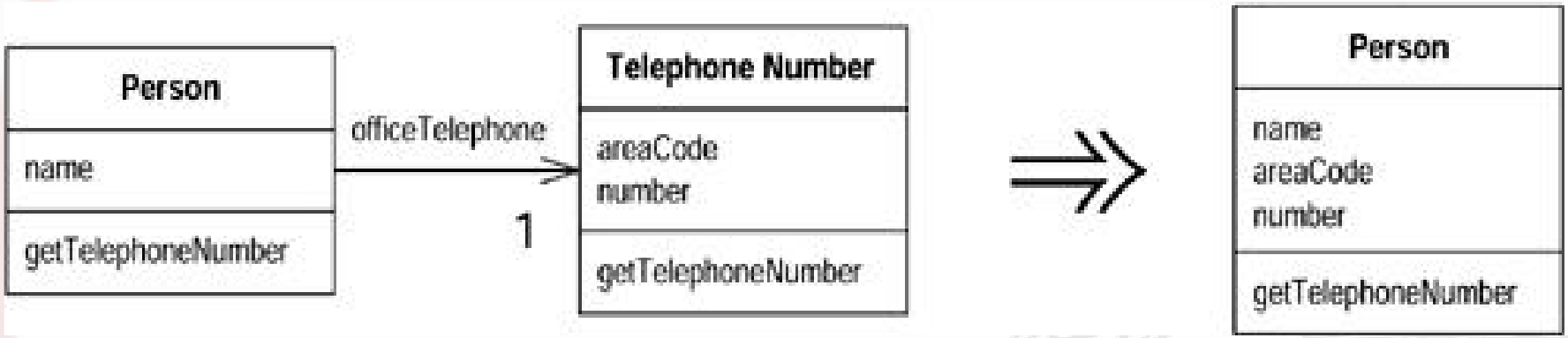
2. Catálogo de Refactorización: Responsabilidades

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



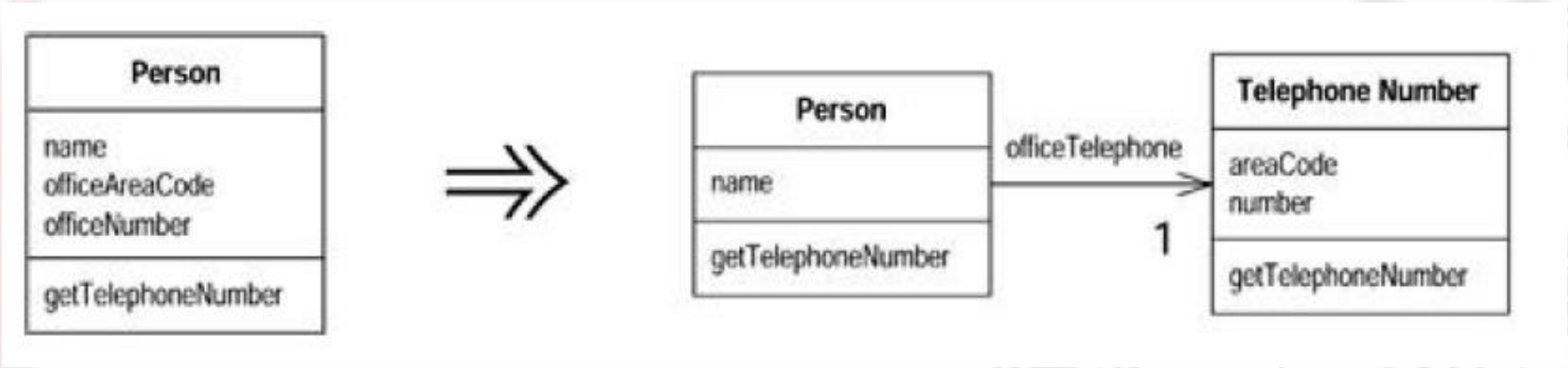
2. Catálogo de Refactorización: Responsabilidades

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



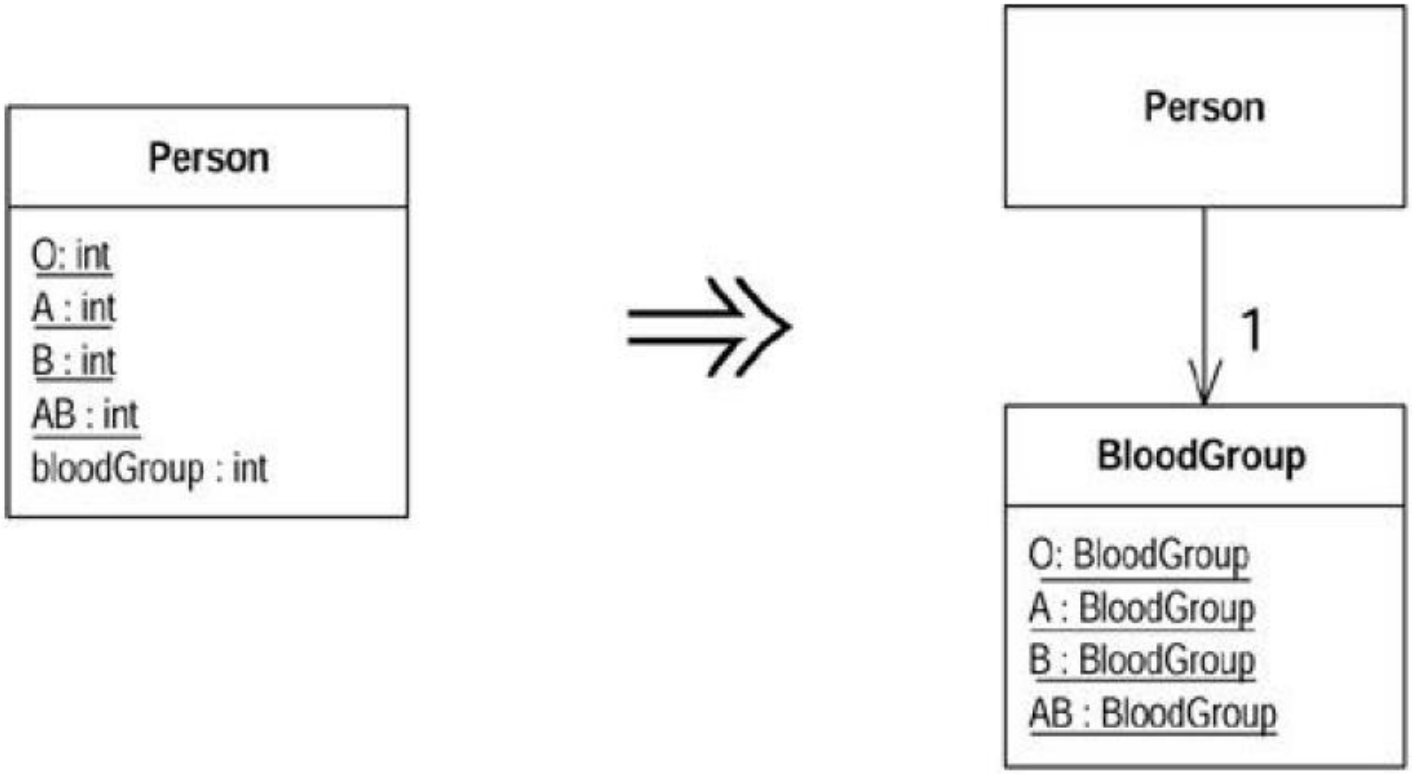
2. Catálogo de Refactorización: Responsabilidades

- *Name*: **Extract Class**
- *Motivation*: You have one class doing work that should be done by two
- *Mechanics*: Create a new class and move the relevant fields and methods from the old class into the new class.



2. Catálogo de Refactorización: Responsabilidades

- *Name*: **Replace Type Code with Class**
- *Motivation*: A class has a numeric type code that does not affect its behavior
- *Mechanics*: Replace the number with a new class



2. Catálogo de Refactorización: Responsabilidades

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

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

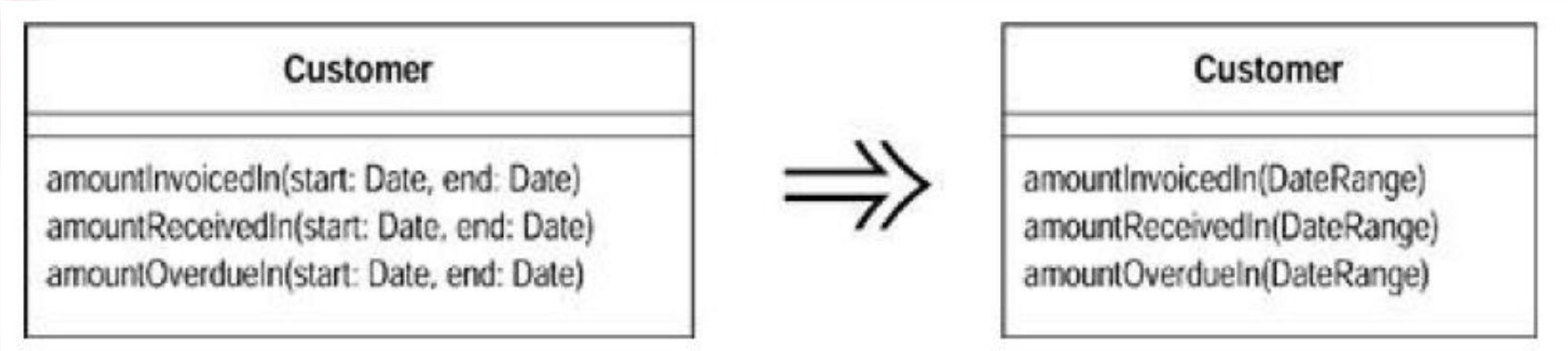
```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

2. Catálogo de Refactorización: Responsabilidades

- *Name*: **Replace Record with Data Class**
- *Motivation*: You need to interface with a record structure in a traditional programming environment.
- *Mechanics*: Make a dumb data object for the record.

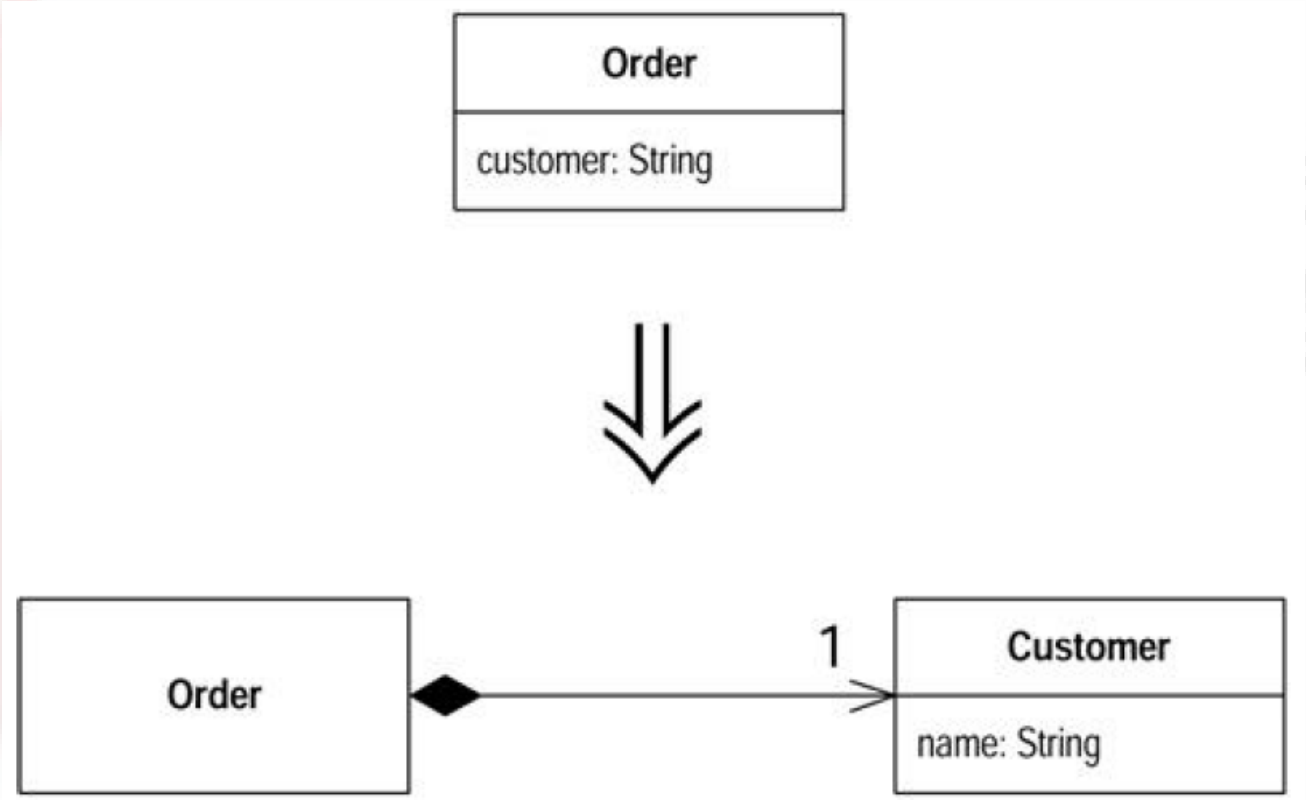
2. Catálogo de Refactorización: Responsabilidades

- *Name*: **Introduce Parameter Object**
- *Motivation*: You have a group of parameters that naturally go together
- *Mechanics*: Replace them with an object.



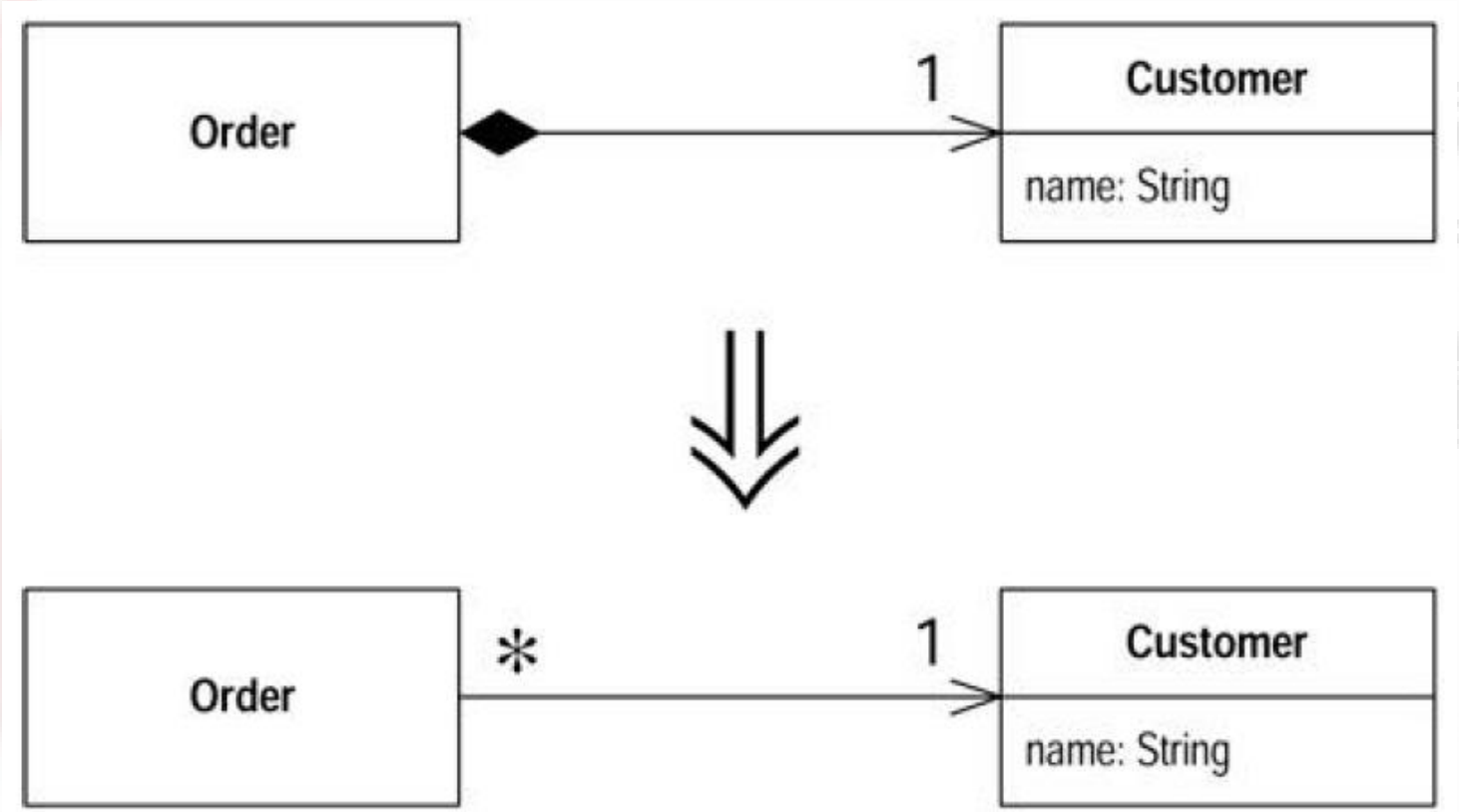
2. Catálogo de Refactorización: Responsabilidades

- *Name*: **Replace Data Value with Object**
- *Motivation*: You have a data item that needs additional data or behavior
- *Mechanics*: Turn the data item into an object.



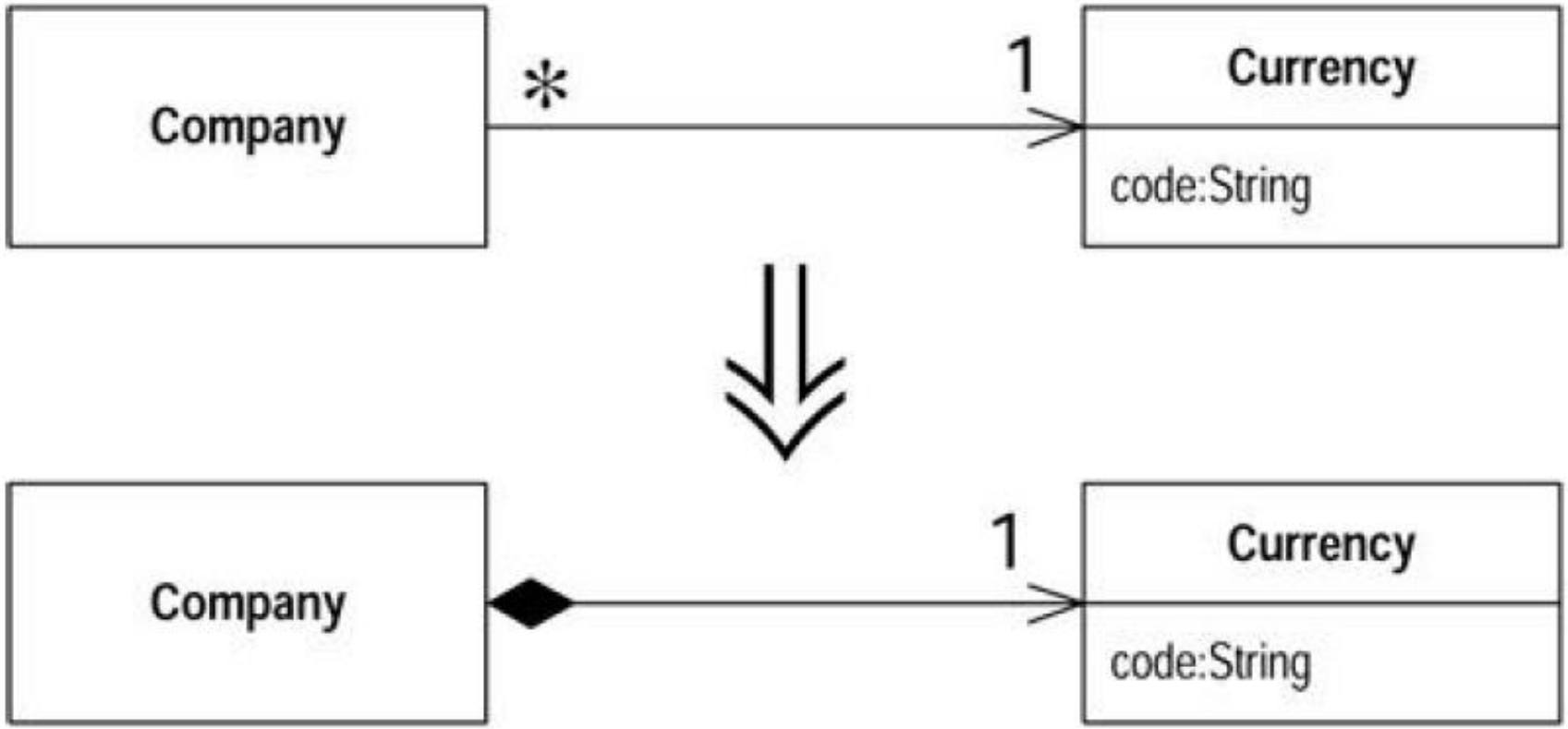
2. Catálogo de Refactorización: Responsabilidades

- *Name*: **Change Value to Reference**
- *Motivation*: You have a class with many equal instances that you want to replace with a single object
- *Mechanics*: Turn the object into a reference object.



2. Catálogo de Refactorización: Responsabilidades

- *Name*: **Change Reference to Value**
- *Motivation*: You have a reference object that is small, immutable, and awkward to manage
- *Mechanics*: Turn it into a value object



2. Catálogo de Refactorización: Colaboraciones

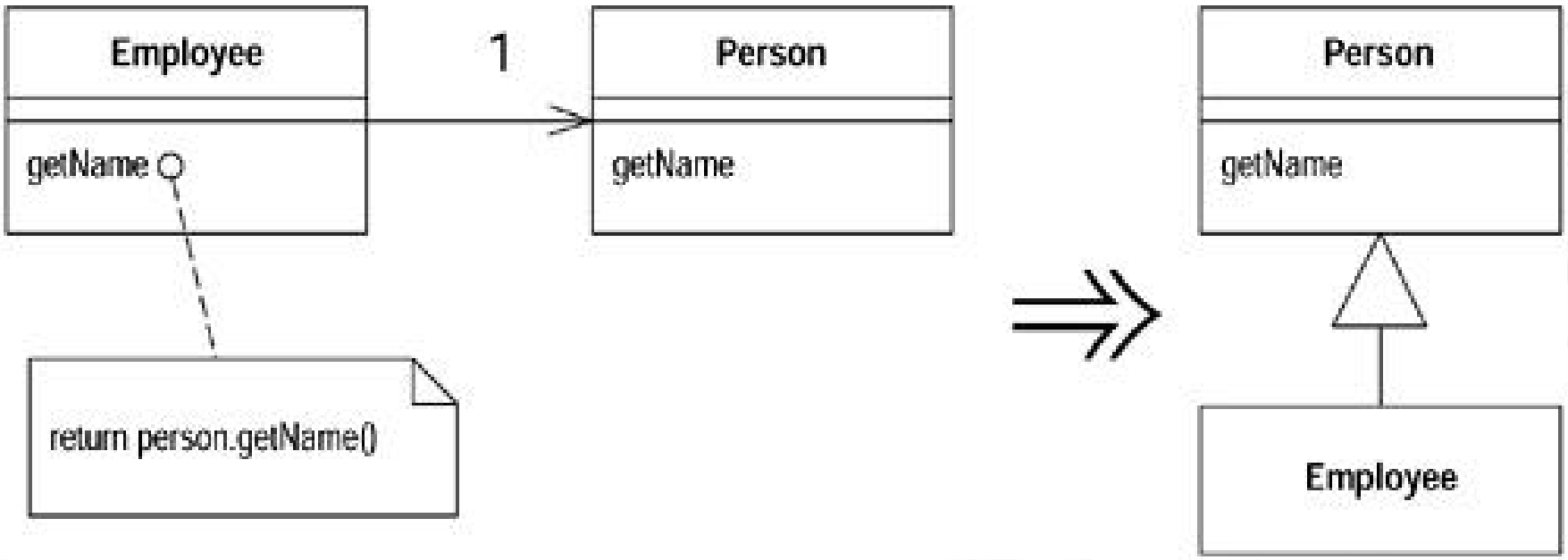
- *Name*: **Preserve Whole Object**
- *Motivation*: You are getting several values from an object and passing these values as parameters in a method call.
- *Mechanics*: Send the whole object instead

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

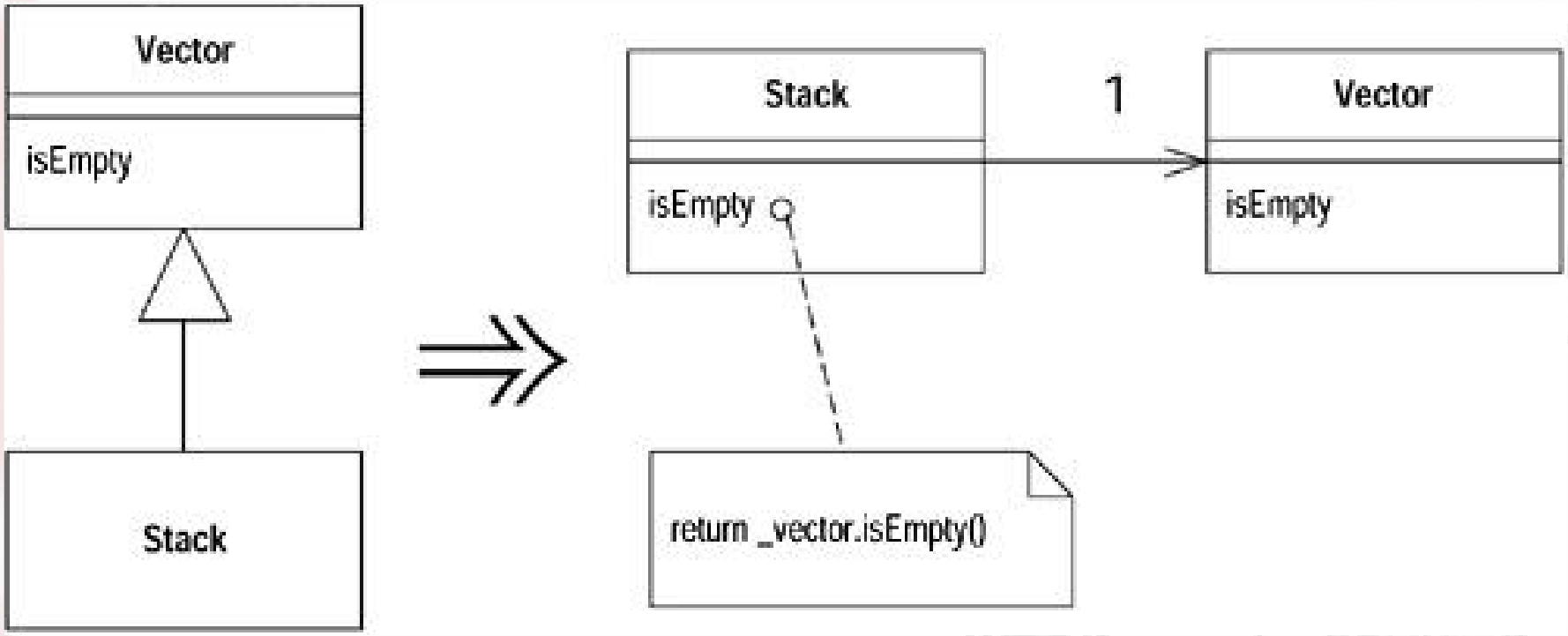
2. Catálogo de Refactorización: Colaboraciones

- *Name*: **Replace Delegation with Inheritance**
- *Motivation*: You're using delegation and are often writing many simple delegations for the entire interface.
- *Mechanics*: Make the delegating class a subclass of the delegate



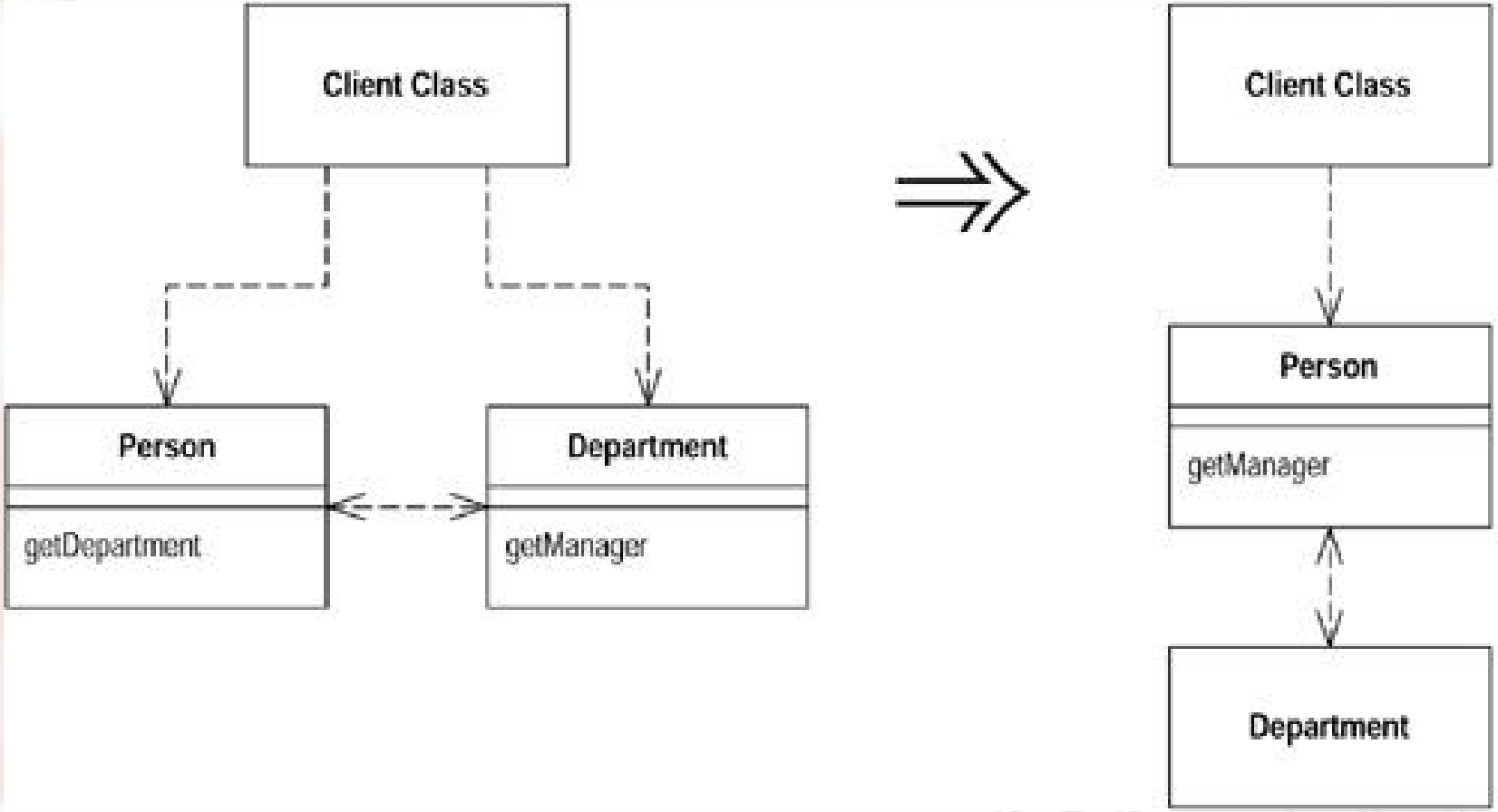
2. Catálogo de Refactorización: Colaboraciones

- *Name*: **Replace Inheritance with Delegation**
- *Motivation*: A subclass uses only part of a superclasses interface or does not want to inherit data
- *Mechanics*: Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing



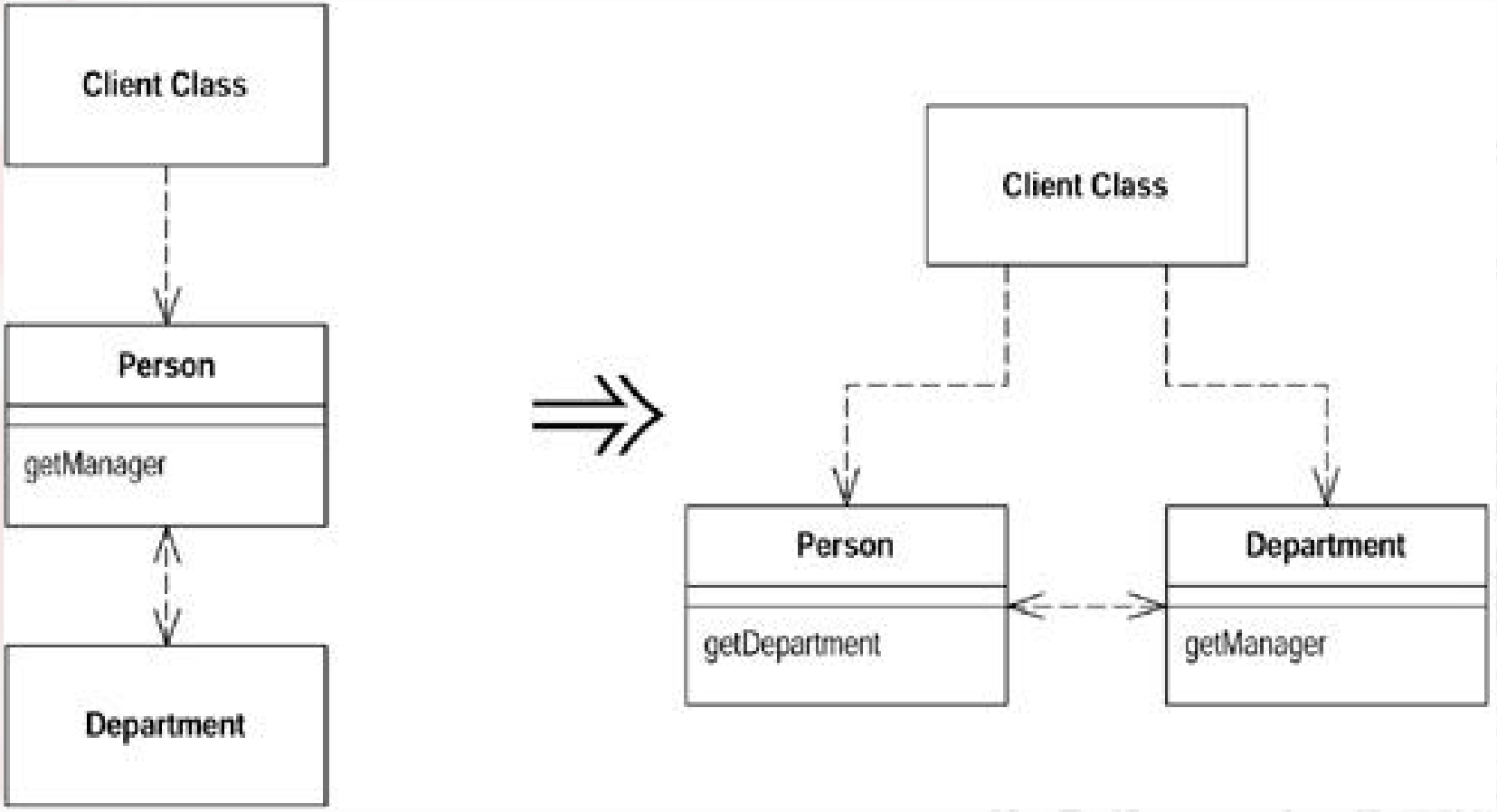
2. Catálogo de Refactorización: Colaboraciones

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



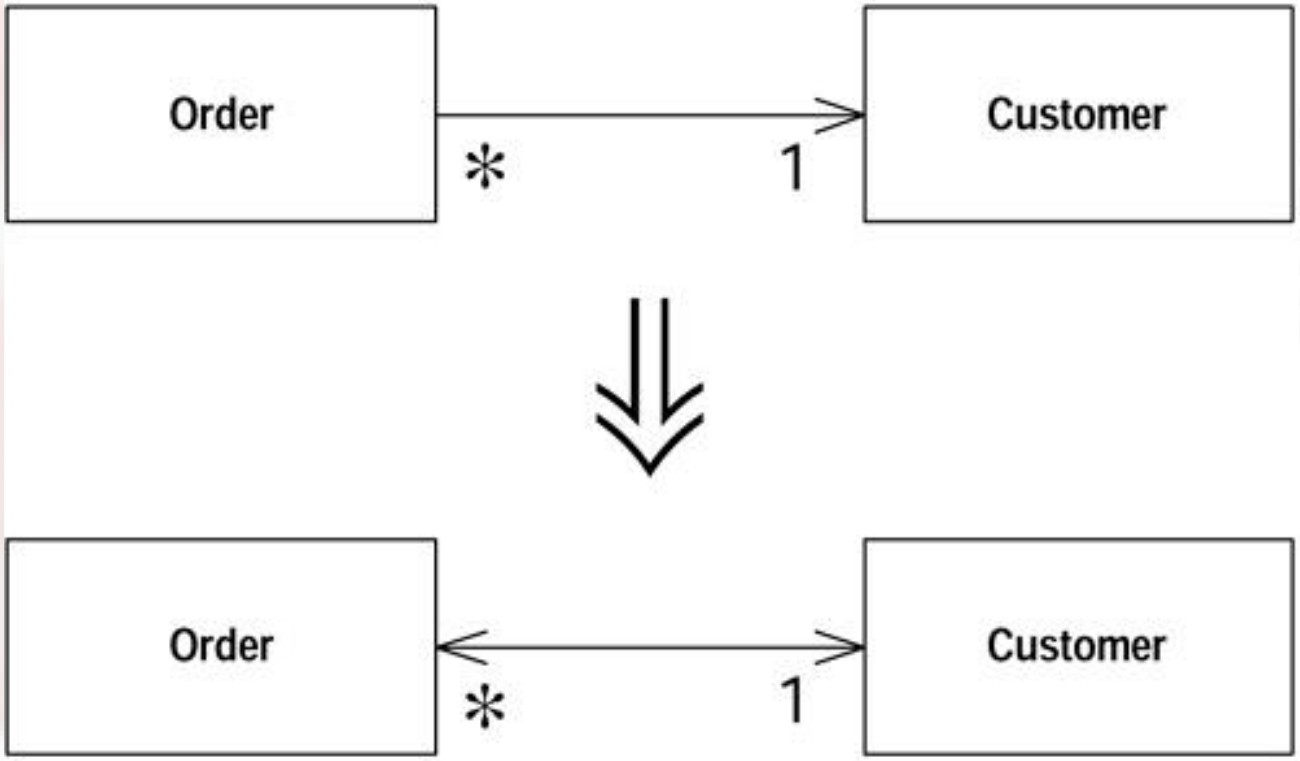
2. Catálogo de Refactorización: Colaboraciones

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



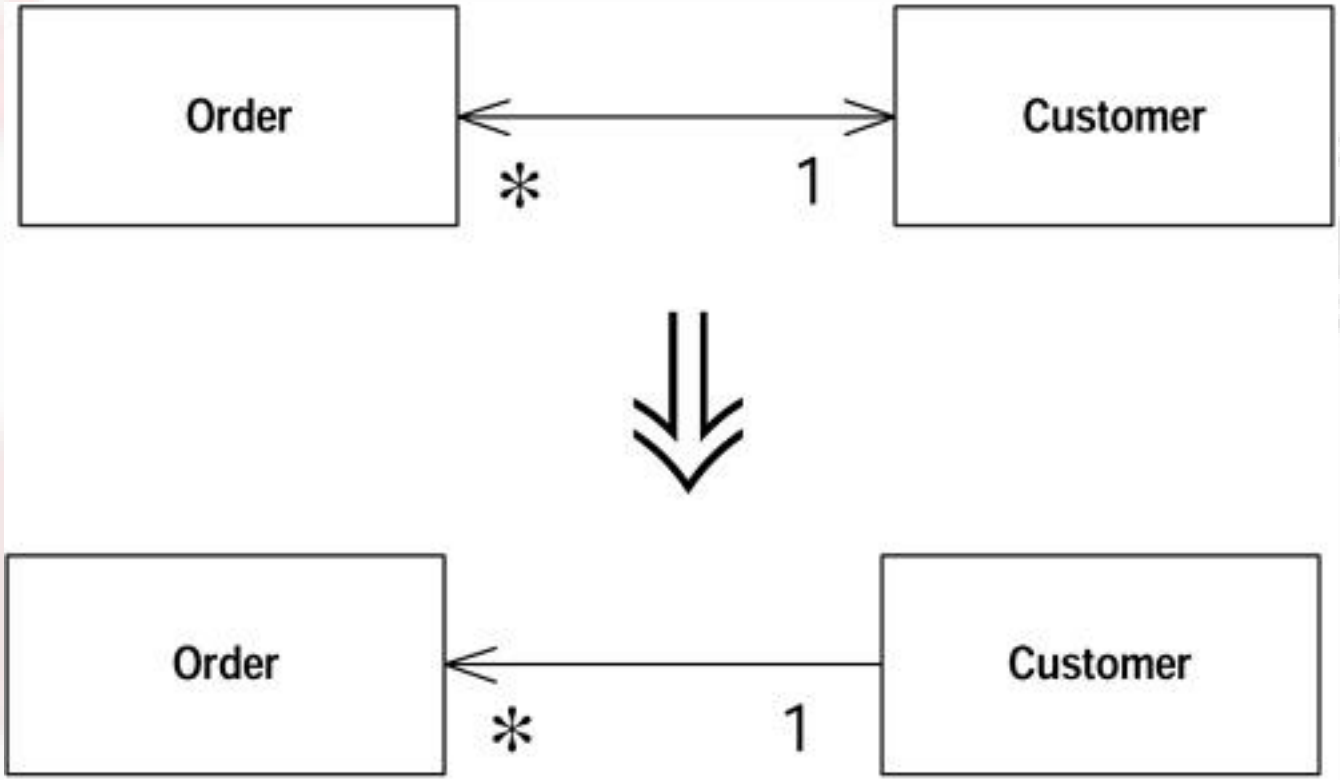
2. Catálogo de Refactorización: Colaboraciones

- *Name*: **Change Unidirectional Association to Bidirectional**
- *Motivation*: You have two classes that need to use each other's features, but there is only a one-way link.
- *Mechanics*: Add back pointers, and change modifiers to update both sets



2. Catálogo de Refactorización: Colaboraciones

- *Name*: **Change Bidirectional Association to Unidirectional**
- *Motivation*: You have a two-way association but one class no longer needs features from the other
- *Mechanics*: Drop the unneeded end of the association



2. Catálogo de Refactorización: Bibliotecas & GUI

- *Name*: **Introduce Foreign Method**
- *Motivation*: A server class you are using needs an additional method, but you can't modify the class.
- *Mechanics*: 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);
```

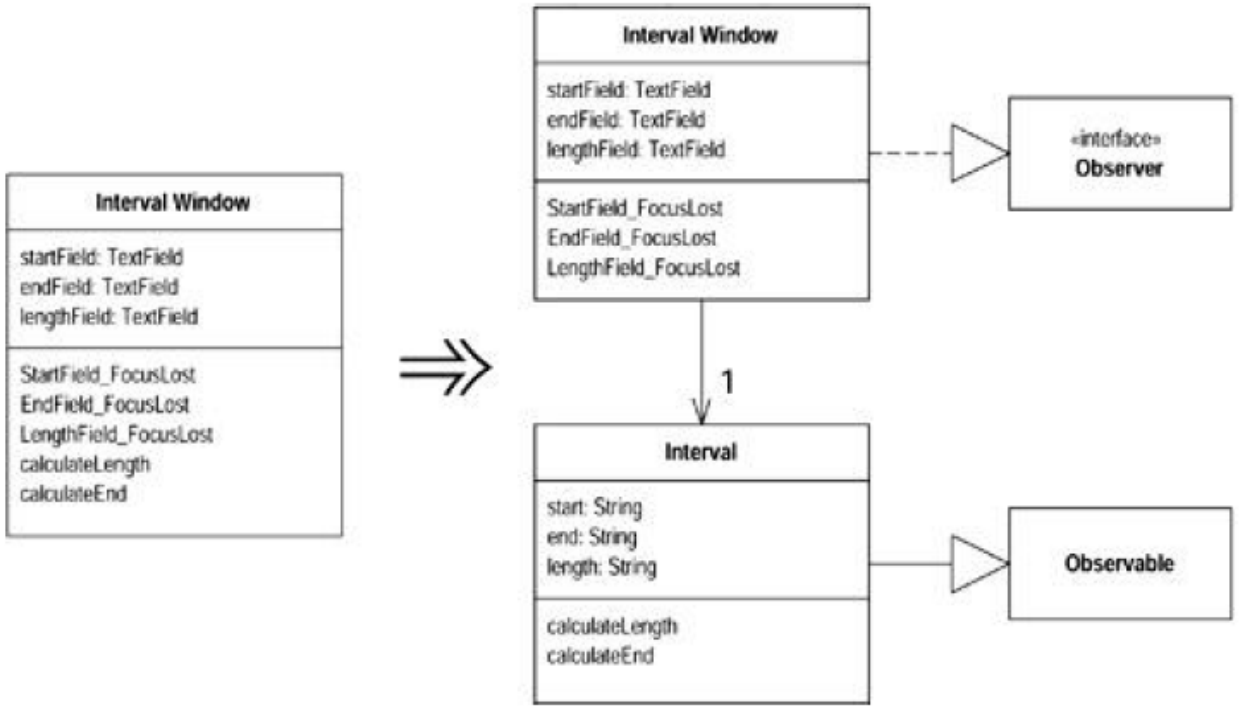
```
Date newStart = nextDay(previousEnd);

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

2. Catálogo de Refactorización: Bibliotecas & GUI

- *Name*: **Introduce Local Extension**
- *Motivation*: A server class you are using needs several additional methods, but you can't modify the class.
- *Mechanics*: Create a new class that contains these extra methods. Make this extension class a subclass or a wrapper of the original

2. Catálogo de Refactorización: Bibliotecas & GUI

- *Name*: **Duplicate Observed Data**
- *Motivation*: You have domain data available only in a GUI control, and domain methods need access.
- *Mechanics*: Copy the data to a domain object. Set up an observer to synchronize the two pieces of data



2. Catálogo de Refactorización: Resumen IV

- **Responsabilidades:**
  - Move Method **vs** Move Field
  - Inline Class **vs** Extract Class
  - Replace Type Code with Class
  - Replace Array with Object
  - Replace Record with Data Class
  - Introduce Parameter Object
  - Replace Data Value with Object
  - Change Value to Reference **vs** Change Reference to Value
- **Colaboraciones:**
  - Preserve Whole Object
  - Replace Delegation with Inheritance **vs** Replace Inheritance with Delegation
  - Hide Delegate **vs** Remove Middle Man
  - Change Unidirectional Association to Bidirectional **vs** Change Bidirectional Association to Unidirectional
- **Bibliotecas & GUI:**
  - Introduce Foreign Method
  - Introduce Local Extension
  - Duplicate Observed Data

2. Catálogo de Refactorización: Resumen III

- **Relación de Herencia:**
  - Extract Interface **&** Extract Superclass
  - Extract Subclass **vs** Collapse Hierarchy
  - Replace Subclass with Fields
  - Pull Up Field **vs** Push Down Field
  - Pull Up Method **vs** Push Down Method **&** Pull Up Constructor Body
- **Polimorfismo:**
  - Replace Conditional with Polymorphism
  - Replace Type Code with Subclasses **vs** Replace Type Code with State/Strategy
  - Replace Constructor with Factory Method

2. Catálogo de Refactorización: Resumen II

- **Encapsulación:**
  - Encapsulate Field & Self Encapsulate Field
  - Hide Method & Remove Setting Method
  - Encapsulate Collection Variables Temporales
  - Encapsulate Downcast
- **Cabecera de Métodos:**
  - Remove Parameter **vs** Add Parameter
  - Parameterize Method
- **Gestión de Errores**
  - Introduce Assertion
  - Replace Error Code with Exception
  - Separate Query from Modifier
  - Replace Exception with Test
- **Nombrado:**
  - Rename Method
  - Replace Magic Number with Symbolic Constant

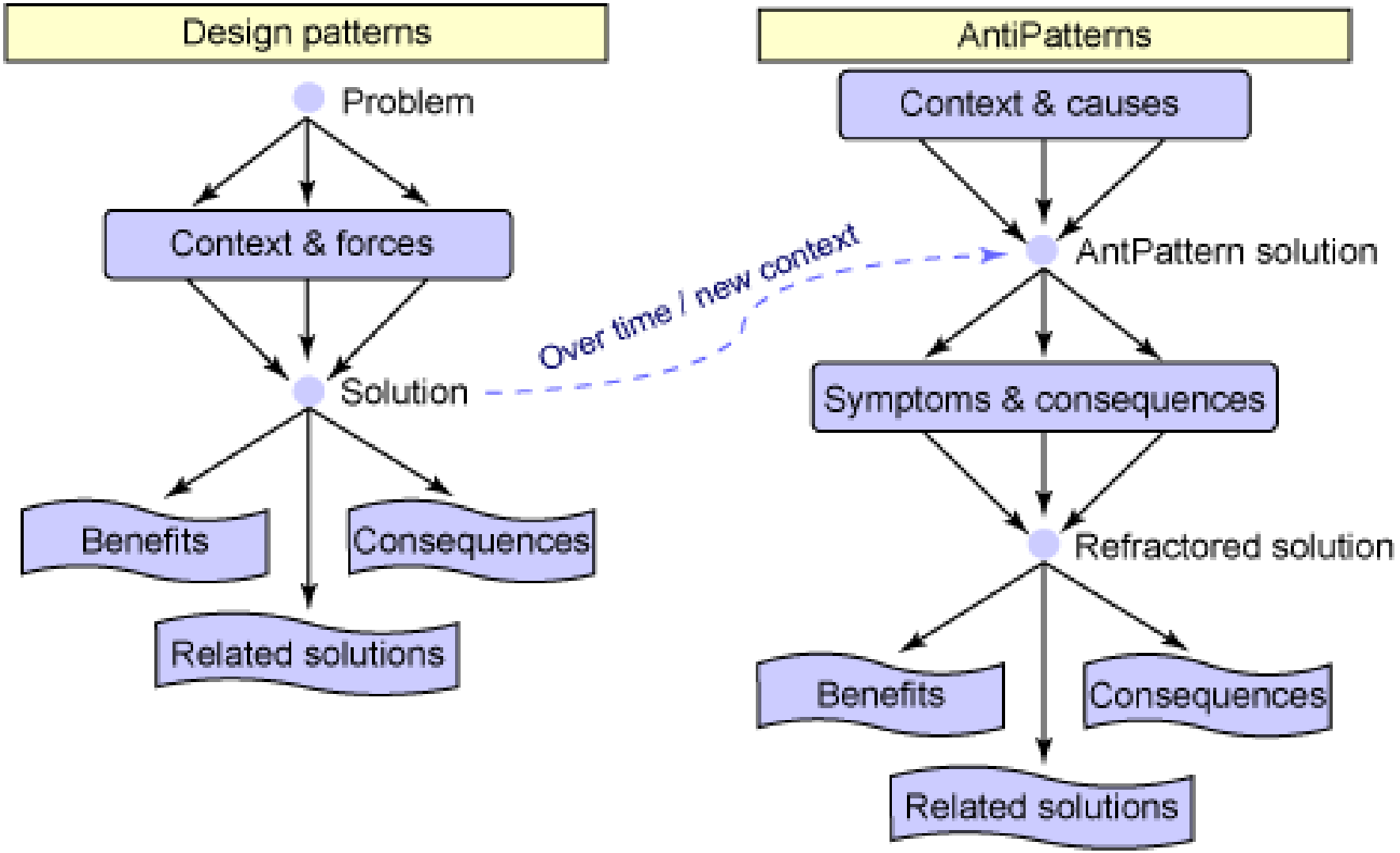
2. Catálogo de Refactorización: Resumen I

- **Condicionales:**
  - Replace Nested Conditional with Guard Clauses
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Introduce Null Object
  - Decompose Conditional
- **Variables Temporales**
  - Split Temporary Variable
  - Introduce Explaining Variable **vs** Inline Temp
  - Replace Temp with Query
- **Parámetros:**
  - Remove Assignments to Parameters
  - Replace Parameter with Method **&** Replace Parameter with Explicit Methods
- **Cuerpo de Métodos**
  - Substitute Algorithm
  - Inline Method **vs** Extract Method
  - Replace Method with Method Object
  - Form Template Method

3. Condiciones para la Refactorización

- **Antipatterns.** Definitions:
  - *AntiPatterns provide stress release in the form of shared misery for the most common pitfalls in the software industry. Often, in software development, it is much easier to recognize a defective situation than to implement a solution*
  - *AntiPatterns provide real−world experience in recognizing recurring problems in the software industry and provide a detailed remedy for the most common predicaments*
  - *AntiPatterns are a method for efficiently mapping a general situation to a specific class of solutions. The general form of the AntiPattern provides an easily identifiable template for the class of problems addressed by the AntiPattern*

3. Condiciones para la Refactorización



3. Condiciones para la Refactorización

- **Antipatterns.** Definitions:
  - *AntiPatterns provide a common vocabulary for identifying problems and discussing solutions. AntiPatterns, like their design pattern counterparts, define an industry vocabulary for the common defective processes and implementations within organizations*
  - *AntiPatterns support the holistic resolution of conflicts, utilizing organizational resources at several levels, where possible. AntiPatterns clearly articulate the collaboration between forces at several levels of management and development:*
    - *Software Project Management AntiPatterns*
    - *Software Architecture AntiPatterns*
    - *Software Development AntiPatterns*

3. Condiciones para la Refactorización

- **Software Development Antipatterns.** Catalog:
  - Cut–and–Paste Programming
  - Spaghetti Code
  - The Blob
  - Lava Flow
  - Functional Decomposition
  - Poltergeists
  - Swiss Army Knife
  - Dead End
- **Others Antipatterns.** Catalog:
  - Project Mismanagement
  - Death by Planning
  - Analysis Paralysis
  - Warm Bodies
  - E–mail Is Dangerous
  - Corncob
  - Intellectual Violence
  - Reinvent the Wheel
  - …

3. Condiciones para la Refactorización

- Antipattern Catalog: **Cut−and−Paste Programming**
  - Code reused by copying source statements leads to significant maintenance problems. Alternative forms of reuse, including black−box reuse, reduce maintenance issues by having common source code, testing, and documentation
- Antipattern Catalog: **Spaghetti Code**
  - Ad hoc software structure makes it difficult to extend and optimize code. Frequent code refactoring can improve software structure, support software maintenance, and enable iterative development

3. Condiciones para la Refactorización

- Antipattern Catalog: **The Blob**
  - Procedural−style design leads to one object with a lion's share of the responsibilities, while most other objects only hold data or execute simple processes. The solution includes refactoring the design to distribute responsibilities more uniformly and isolating the effect of changes.
- Antipattern Catalog: **Lava Flow**
  - Dead code and forgotten design information is frozen in an ever−changing design. This is analogous to a Lava Flow with hardening globules of rocky material. The refactored solution includes a configuration management process that eliminates dead code and evolves or refactors design toward increasing quality.

3. Condiciones para la Refactorización

- Antipattern Catalog: **Functional Decomposition**
  - This AntiPattern is the output of experienced, nonobject−oriented developers who design and implement an application in an object−oriented language. The resulting code resembles a structural language (Pascal, FORTRAN) in class structure. It can be incredibly complex as smart procedural developers devise very "clever" ways to replicate their time−tested methods in an object−oriented architecture.
- Antipattern Catalog: **Poltergeists**
  - Poltergeists are classes with very limited roles and effective life cycles. They often start processes for other objects. The refactored solution includes a reallocation of responsibilities to longer−lived objects that eliminate the Poltergeists.

3. Condiciones para la Refactorización

- Antipattern Catalog: **Swiss Army Knife**
  - A Swiss Army Knife is an excessively complex class interface. The designer attempts to provide for all possible uses of the class. In the attempt, he or she adds a large number of interface signatures in a futile attempt to meet all possible needs
- Antipattern Catalog: **Dead End**
  - A Dead End is reached by modifying a reusable component if the modified component is no longer maintained and supported by the supplier. When these modifications are made, the support burden transfers to the application system developers and maintainers. Improvements in the reusable component are not easily integrated, and support problems can be blamed upon the modification.

3. Condiciones para la Refactorización

- **Code Smell.** Definitions:
  - Code Smell is a surface indication that usually corresponds to a deeper problem in the system
  - Code Smells are certain structures in the design that indicate violation of fundamental design principles and negatively impact design quality
  - Code Smells are usually not bugs—they are not technically incorrect and do not currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future.

3. Condiciones para la Refactorización

- **Code Smell.** Catalog:
  - One thing we won't try to do here is give you precise criteria for when a refactoring is overdue.
    - In our experience no set of metrics rivals informed human intuition.
  - What we will do is give you indications that there is trouble that can be solved by a refactoring.
    - You will have to develop your own sense of how many instance variables are too many instance variables and how many lines of code in a method are too many lines.

3. Condiciones para la Refactorización

- **Code Smell.** Catalog:
  - Duplicate Code
  - Comments
  - Long Parameter List
  - Long Method
  - Large Class
  - Primitive Obsession
  - Data Class
  - Data Clumps
  - Lazy Class
  - Speculative Generality
  - Feature Envy
  - Inappropriate Intimacy
  - Temporary Fields
  - Divergent Changes
  - Shotgun Surgery
  - Message Chains
  - Middle Man
  - Alternative Classes with Different Interfaces
  - Refused Bequest
  - Parallel Inheritance Hierarchies
  - Switch Statements
  - Incomplete Library Class

3. Condiciones para la Refactorización

- Code Smell Catalog: **Duplicate 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.

Condition	Refactoring
the same expression in two methods of the same class	Extract Method
the same expression in two sibling subclases	Extract Method & Pull Up Field
code is similar but not the same	Extract Method & Form Template Method
the methods do the same thing with a different algorithm	Substitute Algorithm & Extract Method
duplicated code in two unrelated classes	Extract Class | Extract Method

3. Condiciones para la Refactorización

- Code Smell Catalog: **Comments**
  - It's surprising how often you look at thickly commented code and notice that the comments are there because the code is bad.

Condition	Refactoring
a comment to explain what a block of code does	Extract Method
method is already extracted but you still need a comment to explain what it does	Rename Method
code is similar but not the same	Extract Method & Form Template Method
state some rules about the required state of the system	Introduce Assertion

3. Condiciones para la Refactorización

- Code Smell Catalog: **Long Parameter List**
  - long parameter lists are hard to understand, because they become inconsistent and difficult to use, and because you are forever changing them as you need more data.

Condition	Refactoring
to get the data in one parameter by making a request of an object you already know about. This object might be a field or it might be another parameter	Replace Parameter with Method
to take a bunch of data gleaned from an object and replace it with the object itself	Preserve Whole Object
several data items with no logical object	Introduce Parameter Object

3. Condiciones para la Refactorización

- Code Smell Catalog: **Long Method**
  - Since the early days of programming people have realized that the longer a procedure is, the more difficult it is to understand.

Condition	Refactoring
to eliminate the temps	Replace Temp with Query | Inline Temp
you still have too many temps and parameters	Replace Method with Method Object
a method does query and modification	Separate Query from Modifier

3. Condiciones para la Refactorización

- Code Smell Catalog: **Large Class**
  - When a class is trying to do too much, it often shows up as too many instance variables.

Condition	Refactoring
a class has too many instance variables	Extract Class
if the component makes sense as a subclass	Extract Subclass
a class does not use all of its instance variables all of the time	Extract Class | Extract Subclass
If your large class is a GUI class	Duplicate Observed Data

A useful trick is to determine how clients use the class and to use Extract Interface for each of these uses. That may give you ideas on how you can further break up the class

3. Condiciones para la Refactorización

- Code Smell Catalog: **Primitive Obsession**
  - People new to objects usually are reluctant to use small objects for small tasks (telephone, ZIP, …).

Condition	Refactoring
You can move out of the cave into the centrally heated world of objects	Replace Data Value with Object
If the data value is a type code	Replace Type Code with Class
If you have conditionals that depend on the type code	Replace Type Code with Subclasses | Replace Type Code with State/Strategy
If you see these primitives in parameter lists	Introduce Parameter Object
If you find yourself picking apart an array	Replace Array with Object

3. Condiciones para la Refactorización

- Code Smell Catalog: **Data Class**
  - These are classes that have fields, getting and setting methods for the fields, and nothing else.

Condition	Refactoring
In early stages these classes may have public fields	Encapsulate Field
If you have collection fields, check to see whether they are properly encapsulated	Encapsulate Collection
any field that should not be changed	Remove Setting Method
Look for where these getting and setting methods are used by other clases, to move behavior into the data class	Move Method
If you can't move a whole method	Extract Method
on the getters and setters	Hide Method

3. Condiciones para la Refactorización

- Code Smell Catalog: **Data Clumps**
  - Bunches of data that hang around together really ought to be made into their own object. A good test is to consider deleting one of the data values: if you did this, would the others make any sense?

Condition	Refactoring
to look for where the clumps appear as fields	Extract Class
turn your attention to method signatures	Introduce Parameter Object | Preserve Whole Object

3. Condiciones para la Refactorización

- Code Smell Catalog: **Lazy Class**
  - A class that isn't doing enough to pay for itself should be eliminated

Condition	Refactoring
Nearly useless components	Inline Class
If you have subclases that aren't doing enough	Collapse Hierarchy

3. Condiciones para la Refactorización

- Code Smell Catalog: **Speculative Generality**
  - You get it when people say, "Oh, I think we need the ability to this kind of thing someday" and thus want all sorts of hooks and special cases to handle things that aren't required. The result often is harder to understand and maintain

Condition	Refactoring
If you have abstract classes that aren't doing much	Collapse Hierarchy
Unnecessary delegation can be removed	Inline Class
Methods with unused parameters	Remove Parameter
Methods named with odd abstract names should be broughtdown to earth	Rename Method

3. Condiciones para la Refactorización

- Code Smell Catalog: **Feature Envy**
  - a method that seems more interested in a class other than the one it actually is in. The most common focus of the envy is the data. We've lost count of the times we've seen a method that invokes half-a-dozen getting methods on another object to calculate some value

Condition	Refactoring
method clearly wants to be elsewhere	Move Method
Sometimes only part of the method suffers from envy	Extract Method & Move Method

Often a method uses features of several classes, so which one should it live with? The heuristic we use is to determine which class has most of the data and put the method with that data. This step is often made easier if Extract Method is used to break the method into pieces that go into different places

3. Condiciones para la Refactorización

- Code Smell Catalog: **Inappropriate Intimacy**
  - Sometimes classes become far too intimate and spend too much time delving in each others'private parts

Condition	Refactoring
Overintimate classes need to be broken	Move Method & Move Field | Change Bidirectional Association to Unidirectional
If the classes do have common interests	Extract Class | Hide Delegate
Inheritance often can lead to overintimacy. Subclasses are always going to know more about their parents than their parents would like them to know	Replace Delegation with Inheritance

3. Condiciones para la Refactorización

- Code Smell Catalog: **Temporary Field**
  - Sometimes you see an object in which an instance variable is set only in certain circumstances

Condition	Refactoring
a complicated algorithm needs several variables. Because the implementer didn't want to pass around a huge parameter list (who does?), he put them in fields. But the fields are valid only during the algorithm; in other contexts they are just plain confusing	Extract Class | Replace Method with Method Object
to eliminate conditional code	Introduce Null Object

3. Condiciones para la Refactorización

- Code Smell Catalog: **Divergent Change**
  - Divergent change occurs when one class is commonly changed in different ways for different reasons

Condition	Refactoring
To clean this up you identify everything that changes for a particular cause	Extract Class

3. Condiciones para la Refactorización

- Code Smell Catalog: **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. Shotgun surgery is similar to divergent change but is the opposite

Condition	Refactoring
When the changes are all over the place, they are hard to find, and it's easy to miss an important change	Move Method & Move Field
to bring a whole bunch of behavior together	Inline Class

3. Condiciones para la Refactorización

- Code Smell Catalog: **Message Chains**
  - You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on

Condition	Refactoring
You may see these as a long line of getThis methods, or as a sequence of temps	Hide Delegate
Often a better alternative is to see what the resulting object is used	Extract Method & Move Method

3. Condiciones para la Refactorización

- Code Smell Catalog: **Middle Man**
  - One of the prime features of objects is encapsulation—hiding internal details from the rest of the world. Encapsulation often comes with delegation. However, this can go too far

Condition	Refactoring
You look at a class's interface and find half the methods are delegating to this other class	Hide Delegate
If only a few methods aren't doing much	Inline Method
If there is additional behavior	Replace Delegation with Inheritance

3. Condiciones para la Refactorización

- Code Smell Catalog: **Alternative Classes with Different Interfaces**

Condition	Refactoring
on any methods that do the same thing but have different signatures for what they do	Rename Method
the classes aren't yet doing enough. To move behavior to until the protocols are the same	Move Method
If you have to redundantly move code to accomplish this	Extract Superclass

3. Condiciones para la Refactorización

- Code Smell Catalog: **Refused Bequest**
  - Subclasses get to inherit the methods and data of their parents. But what if they don't want or need what they are given? They are given all these great gifts and pick just a few to play with

Condition	Refactoring
the hierarchy is wrong	Push Down Method & Push Down Field
if the subclass is reusing behavior but does not want to support the interface of the superclass	Replace Inheritance with Delegation

Often you'll hear advice that all superclasses should be abstract. You'll guess from our snide use of traditional that we aren't going to advise this, at least not all the time. We do subclassing to reuse a bit of behavior all the time, and we find it a perfectly good way of doing business. There is a smell, we can't deny it, but usually it isn't a strong smell. Nine times out of ten this smell is too faint to be worth cleaning.

3. Condiciones para la Refactorización

- Code Smell Catalog: **Parallel Inheritance Hierarchies**
  - In this case, every time you make a subclass of one class, you also have to make a subclass of another. You can recognize this smell because the prefixes of the class names in one hierarchy are the same as the prefixes in another hierarchy

Condition	Refactoring
for eliminating the duplication is to make sure that instances of one hierarchy refer to instances of the other	Move Method & Move Field & Inline Class

3. Condiciones para la Refactorización

- Code Smell Catalog: **Switch Statements**
  - Often you find the same switch statement scattered about a program in different places. If you add a new clause to the switch, you have to find all these switch, statements and change them

Condition	Refactoring
Often the switch statement switches on a type code. You want the method or class that hosts the type code value	Replace Type Code with Subclasses | Replace Type Code with State/Strategy
you have set up the inheritance structure	Replace Conditional with Polymorphism. & Extract Method &Move Method
you only have a few cases that affect a single method, and you don't expect them to change, then polymorphism is overkill	Replace Parameter with Explicit Methods & Null Object

3. Condiciones para la Refactorización

- Code Smell Catalog: **Incomplete Library Class**
  - The trouble is that it is often bad form, and usually impossible, to modify a library class to do something you'd like it to do

Condition	Refactoring
If there are just a couple of methods that you wish the library class had	Introduce Foreign Method
If there is a whole load of extra behavior	Introduce Local Extension

3. Condiciones para la Refactorización

- **Code Smell.** Catalog:
  - Duplicate Code
  - Comments
  - Long Parameter List
  - Long Method
  - Large Class
  - Primitive Obsession
  - Data Class
  - Data Clumps
  - Lazy Class
  - Speculative Generality
  - Feature Envy
  - Inappropriate Intimacy
  - Temporary Fields
  - Divergent Changes
  - Shotgun Surgery
  - Message Chains
  - Middle Man
  - Alternative Classes with Different Interfaces
  - Refused Bequest
  - Parallel Inheritance Hierarchies
  - Switch Statements
  - Incomplete Library Class

4. Desarrollo Dirigido por Pruebas (TDD)

- **Visión general [Beck] (I):**
  - Cogeremos el primer caso de prueba. Diremos "si todo lo que teníamos que hacer era implementar este caso de prueba, entonces necesitaremos solamente un objeto con dos métodos". Implementaremos el objeto con los dos método. Y lo haremos. Nuestro diseño global es un objeto. Durante cerca de un minuto.
  - Entonces cogeremos el siguiente caso de prueba. Bien, podríamos tan solo detenernos en una solución, o podríamos reestructurar el objeto existente en dos objetos. Entonces la implementación de caso de prueba implicaría reemplazar uno de los objetos. Así, primero reestructuramos, ejecutamos el primer caso de prueba para asegurarnos que funciona, a continuación implementaremos el siguiente caso de prueba.

4. Desarrollo Dirigido por Pruebas (TDD)n

- **Visión general [Beck] (II):**
  - Después de uno o dos días de hacer esto, el sistema es bastante más grande de lo que podíamos imaginar dos parejas trabajando en ello, sin preocuparnos una de la otra cómo trabajamos todo el tiempo. De esta manera, conseguimos ser dos parejas implementando casos de prueba al mismo tiempo y periódicamente (unas pocas horas cada vez) integramos sus cambios. Después de uno o dos días, el sistema puede soportar que todo el equipo desarrolle con este estilo.
  - De vez en cuando, el equipo tendrá la sensación de que algo desagrable se ha estado arrastrando hacia arriba por detrás de ellos. Quizá puede que noten un nudo en sus estómagos porque sepan que tienen que cambiar ciertas partes del sistema. En cualquier caso, alguien pide "Tiempo Muerto". El equipo se reúne por un día y reestructura el sistema como un todo, utilizando una combinación de esquemas y recodificación.

4. Desarrollo Dirigido por Pruebas (TDD)

- **Visión general [Beck] (III):**
  - No todas las recodificaciones pueden llevarse a cabo en pocos minutos. Si descubres que has construido una gran jerarquía de herencia enredada, podría llevarte un mes de esfuerzo considerable desenredarla. Pero no puedes dedicar una mes de esfuerzo. Tienes que entregar historias para esta iteración.
  - Cuando te enfrentas a una gran recodificación, tienes que hacerla en pequeños pasos (de nuevo el cambio incremental). Estarás a mitad de un caso de prueba, y verás la posibilidad de dar un paso más hacia el gran objetivo. Da ese paso. Cambia un método aquí, una variable allí. Finalmente todo lo que quedará después de la gran recodificación es un pequeño trabajo. Entonces puedes acabarlo en unos pocos minutos

4. Desarrollo Dirigido por Pruebas (TDD)

- **Two hats:** add new function and refactoring. As you develop software, you probably find yourself swapping hats frequently:
  - *You start by trying to add a new function, and you realize this would be much easier if the code were structured differently.*
  - So you swap hats and **refactor** for a while.
  - Once the code is better structured, you swap hats and **add the new function**.
  - *Once you get the new function working, you realize you coded it in a way that's awkward to understand,*
  - so you swap hats again and **refactor**.
  - *All this might take only ten minutes, but during this time you should always be aware of which hat you're wearing.*

4. Desarrollo Dirigido por Pruebas (TDD)

- **Two hats:**

When you add function	When you refactor
You should not be changing existing code	You make a point of not adding function
You are just adding new capabilities	You only restructure the code
You can getting the tests to work measure your progress by adding tests and	You don't add any tests (unless you find a case you missed earlier). You only change tests when you absolutely need to in order to cope with a change in an interface