PROG-3

SUBTOPIC 9

REFACTORING

Pedro J. Ponce de León

English version by Juan Antonio Pérez

Version 20111113

Universitat d'Alacant
Universidad de Alicante

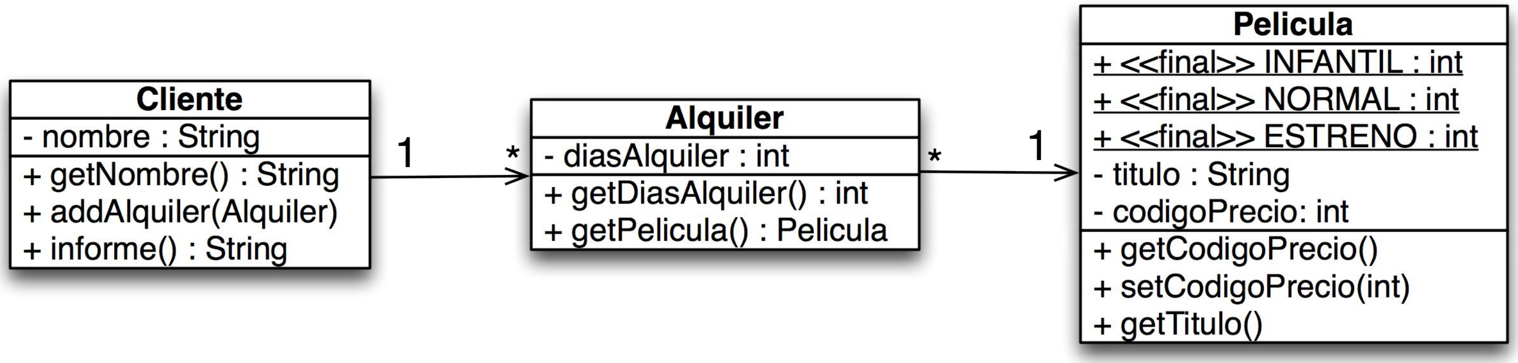
Contents

Introduction. What is refactoring?

- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.
  - It is a disciplined way to clean up code that minimizes the chances of introducing bugs.
  - It consists of two steps:
    - Introduce a simple change (refactoring)
    - Test the system after introducing the change.
  - The possible modifications include:
    - Adding a parameter to a method
    - Moving an attribute from one class to another
    - Moving code around a hierarchy of inheritance

Introduction. A first example

- The sample program is a program to calculate and print a statement of a customer's charges at a video store.
  - The program is told which movies a customer rented and for how long. It then calculates the charges, which depend on how long the movie is rented, and identifies the type movie.
  - There are three kinds of movies: regular, children's, and new releases.
  - In addition to calculating charges, the statement also computes frequent renter points, which vary depending on whether the film is a new release.



Introduction. A first example

```
public String statement() {
          double totalAmount = 0;
          int frequentRenterPoints = 0;
          Enumeration rentals = _rentals.elements();
          String result = "Rental Record for " + getName() + "\n";

          while (rentals.hasMoreElements()) {
                    double thisAmount = 0;
                    Alquiler each = (Alquiler) rentals.nextElement();

                    //determine amounts for each line
                    switch (each.getPelicula().getCodigoPrecio()) {
                    case Pelicula.NORMAL:
                              thisAmount += 2;
                              if (each.getDiasAlquiler() > 2)
                                        thisAmount += (each.getDiasAlquiler() - 2) * 1.5;
                              break;
                    case Pelicula.ESTRENO:
                              thisAmount += each.getDiasAlquiler() * 3;
                              break;
                    case Pelicula.INFANTIL:
                              thisAmount += 1.5;
                              if (each.getDiasAlquiler() > 3)
                                        thisAmount += (each.getDiasAlquiler() - 3) * 1.5;
                              break;
                    }

. . .
```

Introduction. A first example

```
. . .
          // add frequent renter points
          frequentRenterPoints++;
          // add bonus for a two day new release rental
          if ((each.getPelicula().getCodigoPrecio() == Pelicula.ESTRENO)
          && each.getDiasAlquiler() > 1)
                    frequentRenterPoints++;

          // show figures for this rental
          result += "\t" + each.getPelicula().getTitulo()
          + "\t" + String.valueOf(thisAmount) + "\n";

          totalAmount += thisAmount;
  } // END WHILE

          // add footer lines
          result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
          result += "You earned " + String.valueOf(frequentRenterPoints)
          + " frequent renter points";

          return result;
}
```

- This long Cliente.statement routine does far too much. Many of the things that it does should really be done by the other classes.
- It is very likely to introduce errors in the code if we try to modify it.

Introduction. A first example

- Changes that the users would like to make: they want a statement printed in HTML; they also want to make changes to the way movies are classified.

- It is impossible to reuse any of the behavior of the current statement method for an HTML statement.

- Solution: copy the statement() method and make whatever changes you need in Cliente.statementHTML()

- But, what if, for instance, the charging rules are also changed?

Introduction. A first example

Rules of thumb in refactoring:

1. When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.

2. Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking (like, for instance, unit tests).

Advantages include improved code **readability** and reduced complexity to improve the **maintainability** of the source code, as well as a more expressive internal architecture or object model to improve **extensibility**.

"*By continuously improving the design of code, we make it easier and easier to work with. This is in sharp contrast to what typically happens: little refactoring and a great deal of attention paid to expediently adding new features. If you get into the hygienic habit of refactoring continuously, you'll find that it is easier to extend and maintain code*." — Joshua Kerievsky, Refactoring to Patterns

Introduction. A first example

- Decomposing and redistributing the Cliente.statement() method
  - The first step could be finding a logical clump of code and use Extract Method. An obvious piece to extract into its own method here is the switch statement.
  - This way, the statementHTML method will contain much less duplication of code.
- It is not a copy-and-paste trivial change:
  - Which local variables or parameters of Cliente.statement() are used in this fragment of code?
  - Are they read-only (parameters) or modified (values to be returned by the new method) in the fragment?

Introduction. A first example

```
public String statement() {
...
          while (rentals.hasMoreElements()) {
                    double thisAmount = 0;
                    Alquiler each = (Alquiler) rentals.nextElement();

                    thisAmount = calculaCargo(each);
. . .
}
private double calculaCargo(Alquiler alq) {
          double cargo=0;
          switch (alq.getPelicula().getCodigoPrecio()) {
          case Pelicula.NORMAL:
                    cargo += 2;
                    if (alq.getDiasAlquiler() > 2)
                              cargo += (alq.getDiasAlquiler() - 2) * 1.5;
                    break;
          case Pelicula.ESTRENO:
                    cargo += alq.getDiasAlquiler() * 3;
                    break;
          case Pelicula.INFANTIL:
                    cargo += 1.5;
                    if (alq.getDiasAlquiler() > 3)
                              cargo += (alq.getDiasAlquiler() - 3) * 1.5;
                    break;
          }
          return cargo;
}
```

Principles in refactoring

Why should you refactor?

- Refactoring Improves the Design of Software
- Refactoring Makes Software Easier to Understand
- Refactoring Helps You Find Bugs
- Refactoring Helps You Program Faster

Principles in refactoring

When should you refactor?

- <u>Metaphor of two hats</u>
    - A developer has two hats (activities):
        - One is worn when changing code (refactoring),
        - One is worn when adding functionalities.
    - The developer only wears one hat at a time.
    - When you add function, you shouldn't be changing existing code; you are just adding new capabilities.
    - When you refactor, you make a point of not adding function; you only restructure the code.

- Refactor continuously!

  - Refactor when you add a function
  - Refactor when you need to fix a bug
  - Refactor as you do code review
  - Refactor when you detect bad smells in code

Principles in refactoring

Problems with refactoring

- *Persistence layer:*

  Applications tightly coupled to the database schema

- *Changing interfaces*
  - Many of the refactorings do change an interface
  - If a refactoring changes a <u>*published*</u> interface (a step beyond a public interface), you have to retain both the old interface and the new one, at least until your users have had a chance to react to the change. Try to arrange things so that the old interface calls the new interface (delegation).
  - Java uses the annotation `@deprecated`
  - Do not publish interfaces prematurely

Principles in refactoring

When shouldn't you refactor?

- When the code is so full of bugs that you'd better rewrite from scratch
- When you are close to a deadline!

When to refactor: bad smells in code

Code usually contains certain structures that suggest (smell) the possibility of refactoring:

Some examples:

Duplicated code: Number one in the stink parade is 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.

Long method: The OO programs that live best and longest are those with short methods. Programmers new to objects often feel that no computation ever takes place, and that object programs are endless sequences of delegation.

Since the early days of programming people have realized that the longer a procedure is, the more difficult it is to understand. The net effect is that you should be much more aggressive about decomposing methods. A possible heuristic is that whenever you feel the need to comment something, you should write a method instead.

Bad smells in code

Large class: A class with too many instance variables, attributes or methods, or a class with too much code is prime breeding ground for duplicated code. The class is assuming too many responsibilities. The simplest solution is to eliminate redundancy in the class itself or even extract smaller classes or subclasses.

Bad smells in code

Long parameter list: In OOP if you don't have something you need, you can always ask another object to get it for you. Thus with objects you don't pass in everything the method needs; instead you pass enough so that the method can get to everything it needs. A lot of what a method needs is available on the method's host class.

Long parameter lists are hard to understand and use. Long parameter lists usually imply a problem with encapsulation. Most changes are removed by passing objects conveniently encapsulating the data because you are much more likely to need to make only a couple of requests to get at a new piece of data.

Switch statements: The problem with switch statements is essentially that of duplication. 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. The object-oriented notion of polymorphism gives you an elegant way to deal with this problem.

Bad smells in code

Comments: comments aren't a bad smell; indeed they are a sweet smell, but they are often used as a deodorant for code which cannot be easily understood. The first action is to remove the bad smells by refactoring. When you are finished, you often find that the comments are superfluous.

Unit and functional tests

If you want to refactor, the essential precondition is having solid tests.

If you look at how most programmers spend their time, you'll find that writing code actually is quite a small fraction. Most time is spent debugging (in fact, trying to find the bugs).

Unit testing is a method by which individual units of source code are tested to determine if they are fit for use.

Ideally, each test case is independent from the others. They are usually written for every non-trivial method.

Functional (**black-box**) testing is a method of software testing that tests the functionality of an application as a whole as opposed to its internal structures or workings. Specific knowledge of the application's code/internal structure and programming knowledge in general is not required. Test cases are built around specifications and requirements, i.e., what the application is supposed to do.

Unit and functional tests

A good unit test has to be:

- Automatic: tests do not need to require manual intervention. This is specially useful for continuous integration.
- Complete: tests should cover as much code as possible.
- Repeatable or Reusable: tests have to be built in such a way that they can be executed as many times as needed.
- Independent: ideally, each test case has to be independent from the others.
- Professional: tests have to be treated like real code; for instance, documentation has to be provided for them when necessary.

Unit and functional tests

Unit test sets (or 'suites') are a powerful way to detect bugs in the code which reduce dramatically the time needed to find them.

They are an essential precondition for refactoring.

A test set is associated to every class.

Tests have to be executed after every tiny change or refactoring.

Usually, each method in the class is tested by using assertions which check whether the result obtained matches the expected value.

Situations which must be tested include:
- Conditional statements or borderline parameter values.
- Situations which make the method to throw exceptions.

Unit and functional tests

When should you write unit tests?

A common answer would be that tests may be written once the code to be tested exists.

However, test-driven development (TDD) is a software development process that relies on the repetition of a very short development cycle: FIRST the developer writes a failing automated test case that defines a desired improvement or new function, then produces code to pass that test and finally refactors the new code to acceptable standards.

Unit and functional tests

JUnit is a well-known open-source unit testing framework for the Java programming language:

http://junit.sourceforge.net/

Similar libraries exist for **C++:**

Cxxtest : http://cxxtest.tigris.org/ (GNU LGPL)

CppUnit : https://launchpad.net/cppunit2 (GNU LGPL)

And for **C#**,

Nunit : http://www.nunit.org/ (BSD-like license)

CsUnit : http://www.csunit.org/ (zLib license)

Catalog of refactorings

Simple refactorings

- Add Parameter
- Remove Parameter
- Rename Method

Catalog of refactorings

Add Parameter

Motivation: A method needs more information from its caller.

Solution: Add a parameter for an object that can pass on this information.

Example

```
Cliente.getContacto()  → Cliente.getContacto(Fecha f)
```

Remarks:

Avoid long parameter lists.

Catalog of refactorings

Remove Parameter

Motivation: A parameter is no longer used by the method body.

Solution: Remove it.

Example

```
Cliente.getContacto(Fecha f) → Cliente.getContacto()
```

Remarks:

The case to be wary of here is a polymorphic method. In this case you may well find that other implementations of the method do use the parameter. In this case you shouldn't remove the parameter. You might choose to add (overload) a separate method that can be used in those cases.

Catalog of refactorings

Rename Method

Motivation: The name of a method does not reveal its purpose.

Solution: Change the name of the method.

Example

```
Cliente.getLCrdFact() →
Cliente.getLimiteCreditoFactura()
```

Remarks:

- Check whether the method signature is implemented by a super/subclass.
- If the method has been *published*, declare a new method with the new name, copy the old body of code over to the new name, annotate the old one with @deprecated and make it invoke the new one.
- Find all references to the old method name and change them to refer to the new one.

Catalog of refactorings

Common refactorings

- Move Field
- Move Method
- Extract Class
- Extract Method
- Replace Conditional with Polymorphism
- Replace Error Code with Exception

Catalog of refactorings

Move Field

Motivation: A field is, or will be, used by another class more than the class on which it is defined.

Solution: Create a new field in the target class.

Remarks:

- If the field is public, encapsulate it first (use Encapsulate Field).
- Replace all references to the source field with references to the appropriate method (getter/setter) on the target.

Catalog of refactorings

Move Field

<u>Example</u>

```
class Cuenta {
private TipoCuenta tipo;
private double tipoInteres; /* field to be moved to TipoCuenta */

double calculaInteres(double saldo, int dias) {
   return tipoInteres * saldo * dias /365;
}

class TipoCuenta {
  private double tipoInteres; /* moved field */

 void setInteres(double d) {…}
 double getInteres() {…}

double calculaInteres(double saldo, int dias) {
     return tipo.getInteres() * saldo * dias /365;
}
```

Catalog of refactorings

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.

Solution: Create a new method in the class it uses most.

Example

```
Cliente.getLimiteCreditoFactura()
Cuenta.getLimiteCreditoFactura(Cliente c)
// optional par. (use if you need some info from Cliente)
```

Remarks:

- Moving methods is the bread and butter of refactoring.
- Turn the old method into a simple delegation or remove it.
- Check the sub- and superclasses of the source class for other declarations of the method. If there are any other declarations, you may not be able to make the move, unless the polymorphism can also be expressed on the target.

Catalog of refactorings

Extract Class

Motivation: You have one class doing work that should be done by two.

Solution: Create a new class and move the relevant fields and methods from the old class into the new class.

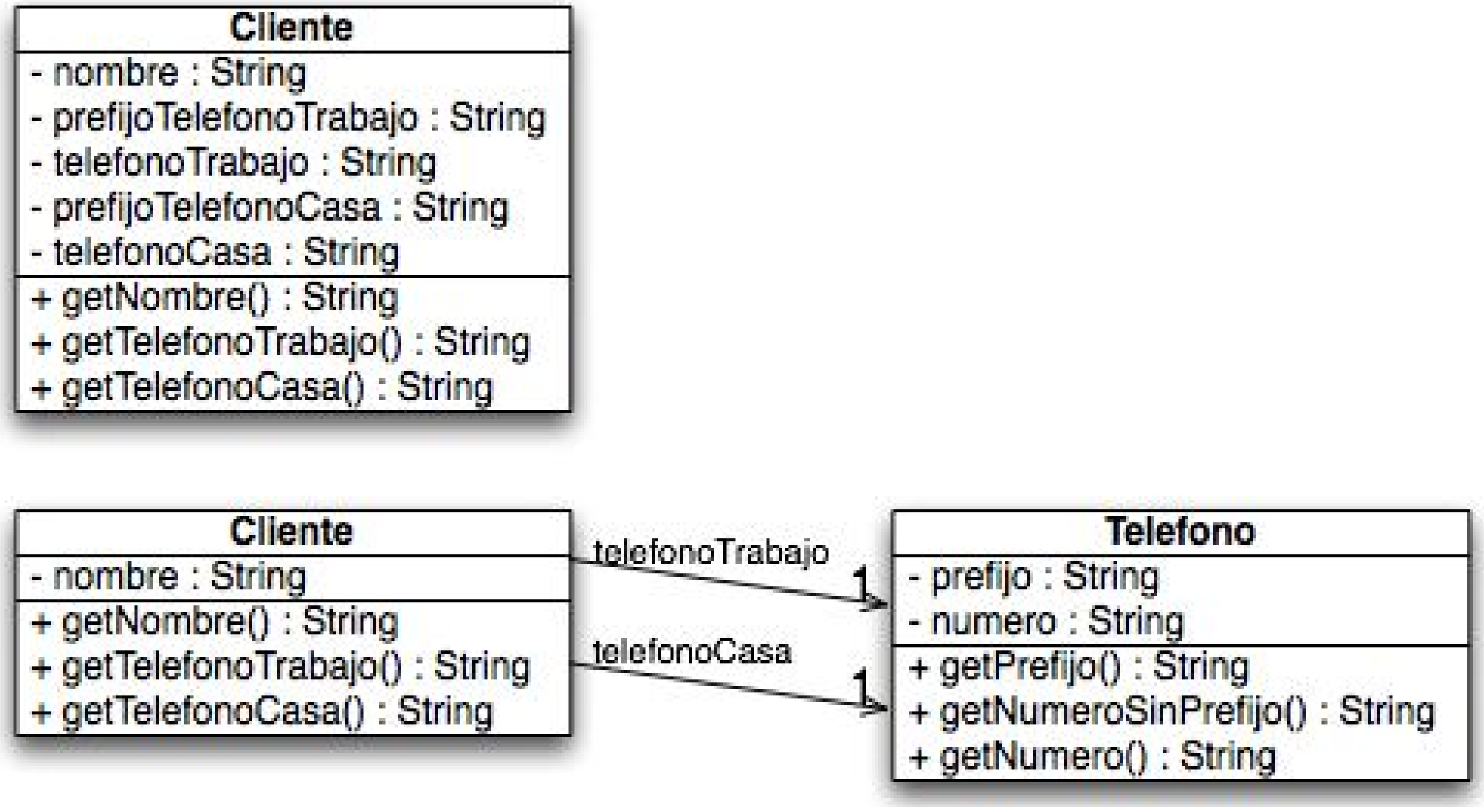
Remarks:

- Decide how to split the responsibilities of the class.
- Create a new class to express the split-off responsibilities.
- Make a link (relationship) from the old to the new class.
- Use Move Field and Move Method.
- Decide whether to expose (make public) the new class.

Catalog of refactorings

Extract Class

Example


Cliente
- nombre : String
- prefijoTelefonoTrabajo : String
- telefonoTrabajo : String
- prefijoTelefonoCasa : String
- telefonoCasa : String
+ getNombre() : String
+ getTelefonoTrabajo() : String
+ getTelefonoCasa() : String
Cliente
- nombre : String
+ getNombre() : String
+ getTelefonoTrabajo() : String
+ getTelefonoCasa() : String
telefonoTrabajo
1
telefonoCasa
1
Telefono
- prefijo : String
- numero : String
+ getPrefijo() : String
+ getNumeroSinPrefijo() : String
+ getNumero() : String

Catalog of refactorings

Extract Method

Motivation: You have a code fragment that can be grouped together.

Solution: Turn the fragment into a method whose name explains the purpose of the method.

Example: (Example in the introduction of these slides)

Remarks:

- This is usually used with long methods.
- Local variables which are read but not changed become parameters/local variables of the new method.
- If a variable is reassigned in the fragment and used afterward, you need to make the extracted code return the changed value of the variable.
- Consider if the new method should be published.

Catalog of refactorings

Replace Conditional with Polymorphism

Motivation: You have a conditional that chooses different behavior depending on the type of an object.

Solution: Move each leg of the conditional to an overriding method (Extract Method) in a subclass. Make the original method abstract.

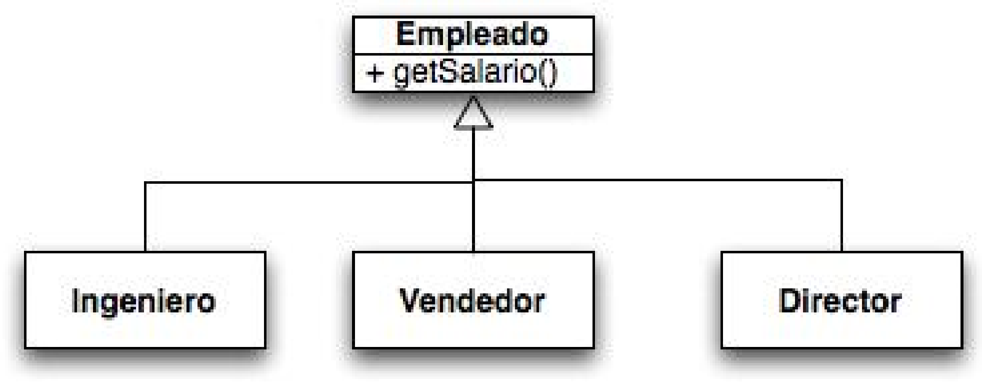
Remarks:

- When using the conditional, the client code needs to know the derived classes.
- When using polymorphism, the client code only needs to know the parent class, and new child classes could be added without changing the client code.

Catalog of refactorings

Replace Conditional with Polymorphism

Example:

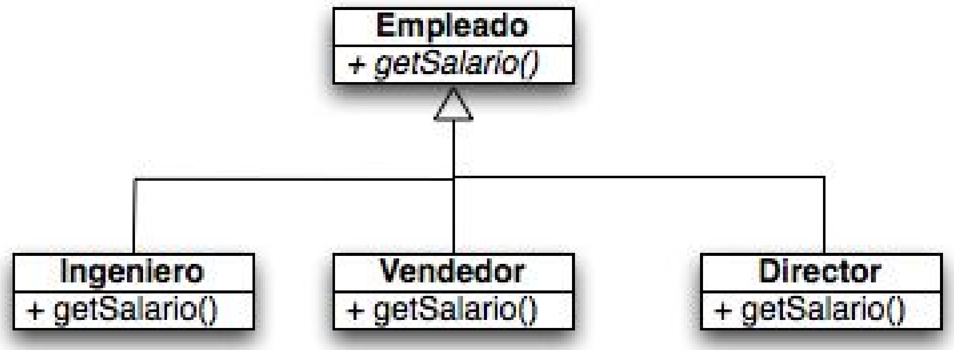


```
double getSalario() {
  switch(tipoEmpleado()) {
    case INGENIERO: return salarioBase + productividad; break;
    case VENDEDOR: return salarioBase + ventas*comision; break;
    case DIRECTOR: return salarioBase + bonificacion+ dietas; break;
    default : throw new RuntimeException(“Tipo de empleado incorrecto”);
  }
}
```

Catalog of refactorings

Replace Conditional with Polymorphism

Example:



```
class Empleado {
  abstract double getSalario(); …
}

class Ingeniero {
  @Override double getSalario() { return salarioBase + productividad; }
...}
class Vendedor {
  @Override double getSalario() { return salarioBase + ventas*comision; }
...}
class Director {
  @Override double getSalario() {return salarioBase+bonificacion+dietas;}
...}
```

Catalog of refactorings

Replace Error Code with Exception

Motivation: A method returns a special code to indicate an error.

Solution: Throw an exception instead.

Remarks:

- Decide whether the exception should be checked or unchecked.
- Client code must handle the exception.
- Be careful! Changing the 'throws' clause (with checked exceptions) of a public method in Java implies that the interface of the containing class is also changed.

Catalog of refactorings

Replace Error Code with Exception

Example

```
int sacarDinero(int cantidad) {
  if (cantidad > saldo) return -1;
  else { saldo -= cantidad; return 0; }
}

int sacarDinero(int cantidad) throws ExcepcionSaldoInsuficiente {
  if (cantidad > saldo) throw new ExcepcionSaldoInsuficiente();
  else { saldo -= cantidad; return 0; }
}
```

Catalog of refactorings

- **Refactoring and inheritance**
  - Pull Up Method
  - Pull Down Method
  - Collapse Hierarchy
  - Extract Subclass
  - Extract Superclass (or interface)
  - Replace Inheritance with Delegation/Composition

Catalog of refactorings

Pull Up Method

Motivation: You have methods with identical results (duplicated code) on subclasses.

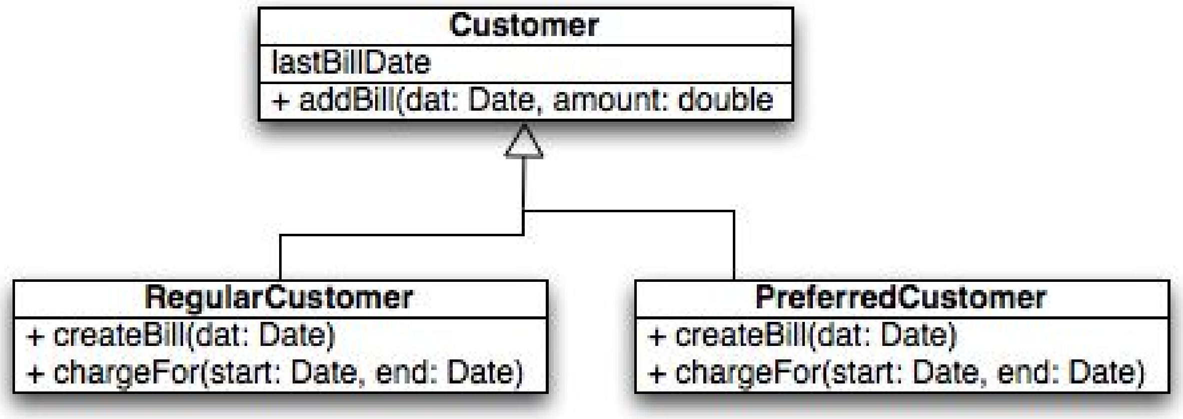
Solution: Move them to the superclass.

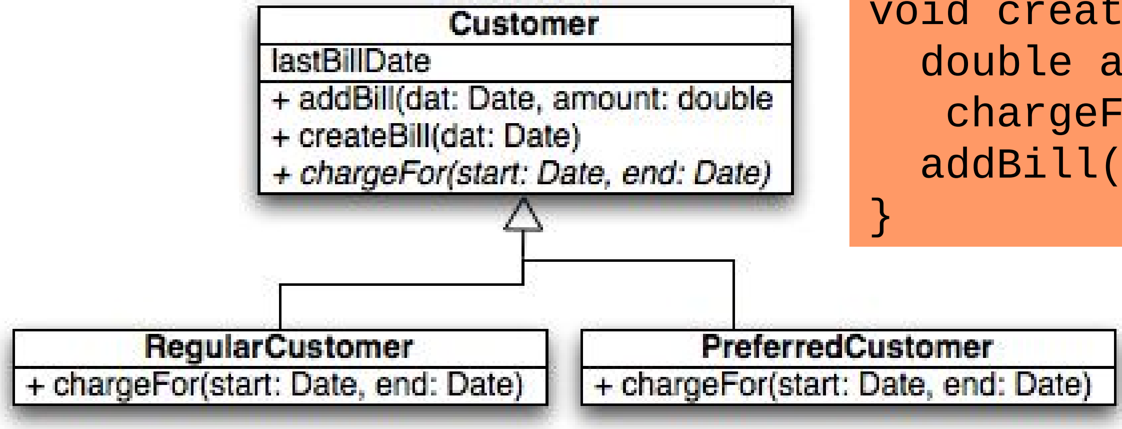
Remarks:

- Inspect the methods to ensure they are identical.
- A special case of the need for Pull Up Method occurs when you have a subclass method that overrides a superclass method yet does the same thing.
- If the method calls another method that is present in both subclasses but not in the superclass...
    - try Pull Up Method also with those methods, or
    - declare an abstract method on the superclass.

Catalog of refactorings

Pull Up Method


Customer
lastBillDate
+ addBill(dat: Date, amount: double
RegularCustomer
+ createBill(dat: Date)
+ chargeFor(start: Date, end: Date)
PreferredCustomer
+ createBill(dat: Date)
+ chargeFor(start: Date, end: Date)


Customer
lastBillDate
+ addBill(dat: Date, amount: double
+ createBill(dat: Date)
+ chargeFor(start: Date, end: Date)
RegularCustomer
+ chargeFor(start: Date, end: Date)
PreferredCustomer
+ chargeFor(start: Date, end: Date)

```
void createBill(Date dat) {
  double amnt =
   chargeFor(lastBillDate, dat);
  addBill(date, amnt);
}
```

Catalog of refactorings

Pull Down Method

Motivation: Behavior on a superclass is relevant only for some of its subclasses.

Solution: Move it to those subclasses.

Example

```
Empleado.getCuotaVentas() → Vendedor.getCuotaVentas()
```

Remarks:

- You often do this when you use Extract Subclass.
- You may need to declare fields as protected for the method to access them or use an accessor on the superclass. If this accessor is not public, you need to declare it as protected.
- If the interface changes, check the client code or leave the method in the parent class as abstract.

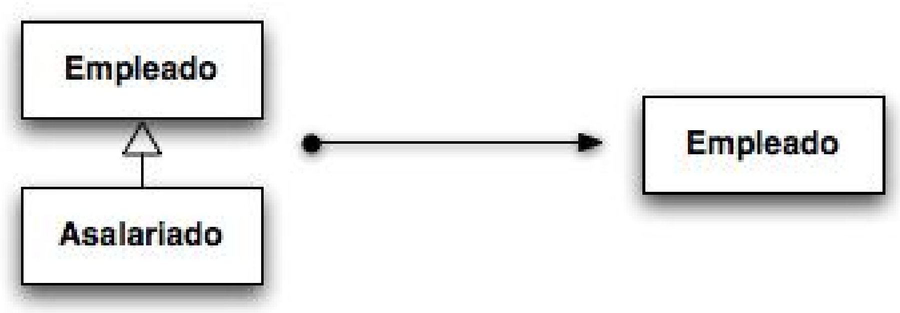
Catalog of refactorings

Collapse Hierarchy

Motivation: A superclass and subclass are not very different.

Solution: Merge them together.

Example:



Remarks:

- Choose which class is going to be removed: the superclass or the subclass.
- Adjust references to the class (in the client code) that will be removed to use the merged class.

Catalog of refactorings

Extract Subclass

Motivation: A class has features that are used only in some instances.

Solution: Create a subclass for that subset of features.

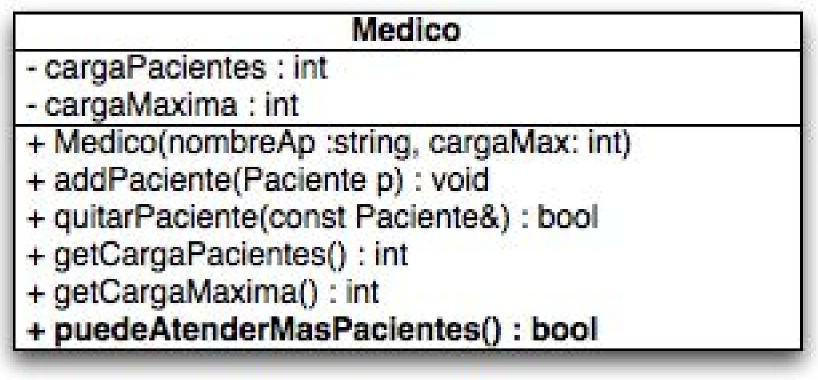
Remarks:

- This is usually a result of a top-down design.
- Attributes like 'tipoCliente' suggest the use of this refactoring.
- Find all calls to constructors of the superclass. If they need the subclass, replace with a call to the new constructor.
- This is related to Pull Down Method and Replace Conditional with Polymorphism.

Catalog of refactorings

Extract Subclass

<u>Example</u>:

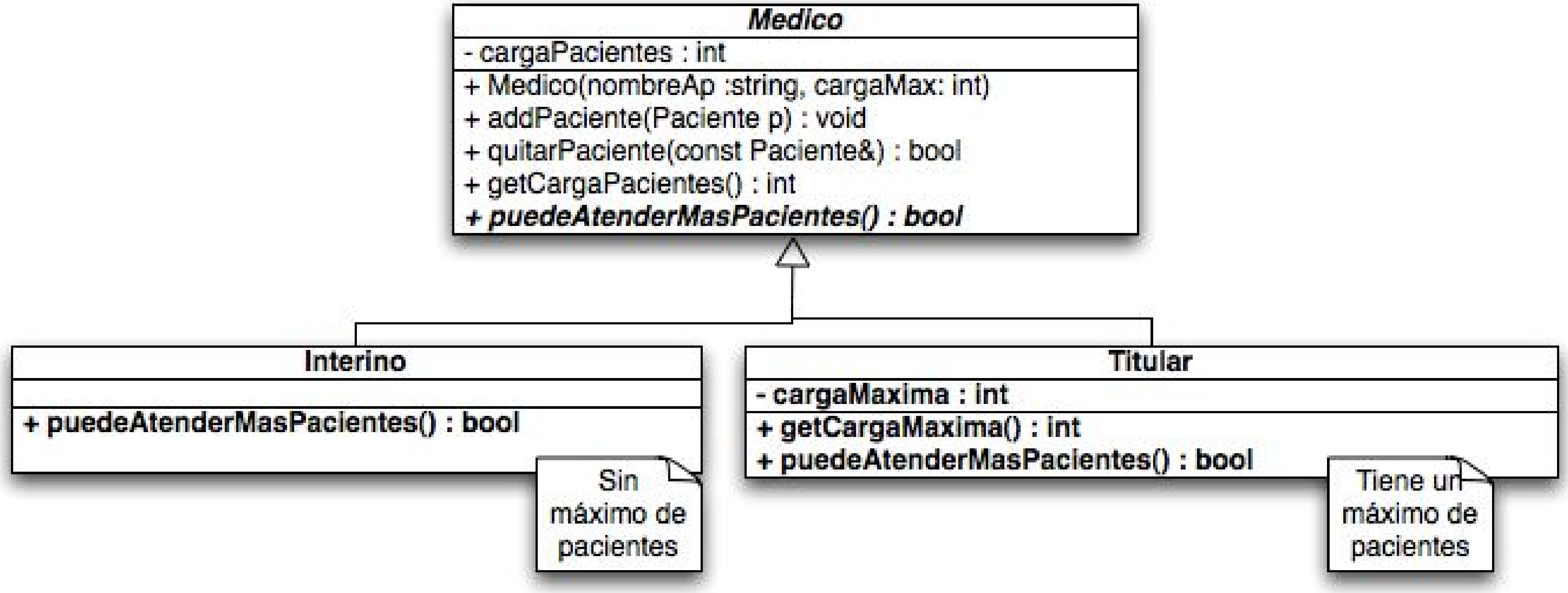

Medico
- cargaPacientes : int
- cargaMaxima : int
+ Medico(nombreAp :string, cargaMax: int)
+ addPaciente(Paciente p) : void
+ quitarPaciente(const Paciente&) : bool
+ getCargaPacientes() : int
+ getCargaMaxima() : int
+ puedeAtenderMasPacientes() : bool

Permanent doctors have a maximum patient load. Temporary doctors may attend in principle an unrestricted number of patients.

Catalog of refactorings

Extract Subclass

Example:


Medico
- cargaPacientes : int
+ Medico(nombreAp :string, cargaMax: int)
+ addPaciente(Paciente p) : void
+ quitarPaciente(const Paciente&) : bool
+ getCargaPacientes() : int
+ puedeAtenderMasPacientes() : bool
Interino
+ puedeAtenderMasPacientes() : bool
Sin máximo de pacientes
Titular
- cargaMaxima : int
+ getCargaMaxima() : int
+ puedeAtenderMasPacientes() : bool
Tiene un máximo de pacientes

Catalog of refactorings

Extract Superclass

Motivation: You have two classes with similar features.

Solution: Create a superclass and move the common features to the superclass.

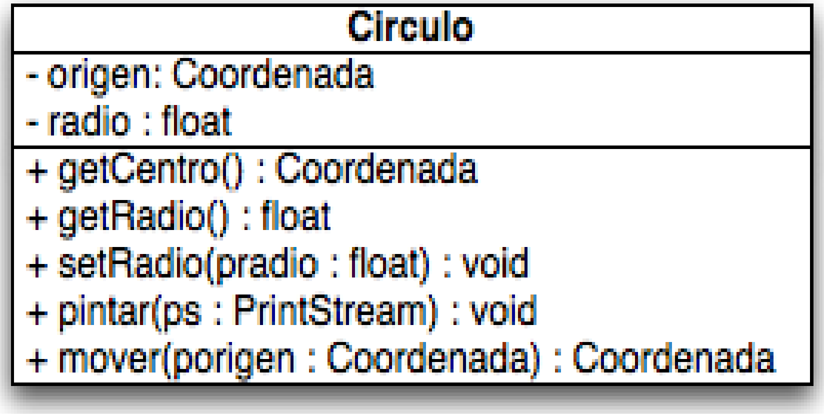
Remarks:

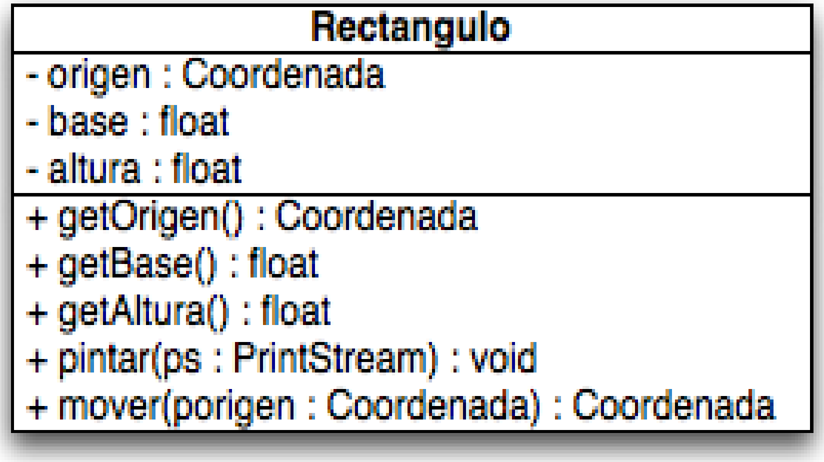
- This is usually a result of a bottom-up design.
- Often duplicated code is removed.
- The new base class is usually an abstract class or even an interface.
- Examine the methods left on the subclasses. See if there are common parts, if there are you can use Extract Method followed by Pull Up Method on the common parts.
- After pulling up all the common elements, check each client of the subclasses. If they use only the common interface you can change the required type to the superclass.

Catalog of refactorings

Extract Superclass

Example:


Circulo
- origen: Coordenada
- radio : float
+ getCentro() : Coordenada
+ getRadio() : float
+ setRadio(pradio : float) : void
+ pintar(ps : PrintStream) : void
+ mover(porigen : Coordenada) : Coordenada


Rectangulo
- origen : Coordenada
- base : float
- altura : float
+ getOrigen() : Coordenada
+ getBase() : float
+ getAltura() : float
+ pintar(ps : PrintStream) : void
+ mover(porigen : Coordenada) : Coordenada

Catalog of refactorings

Extract Superclass

Example:

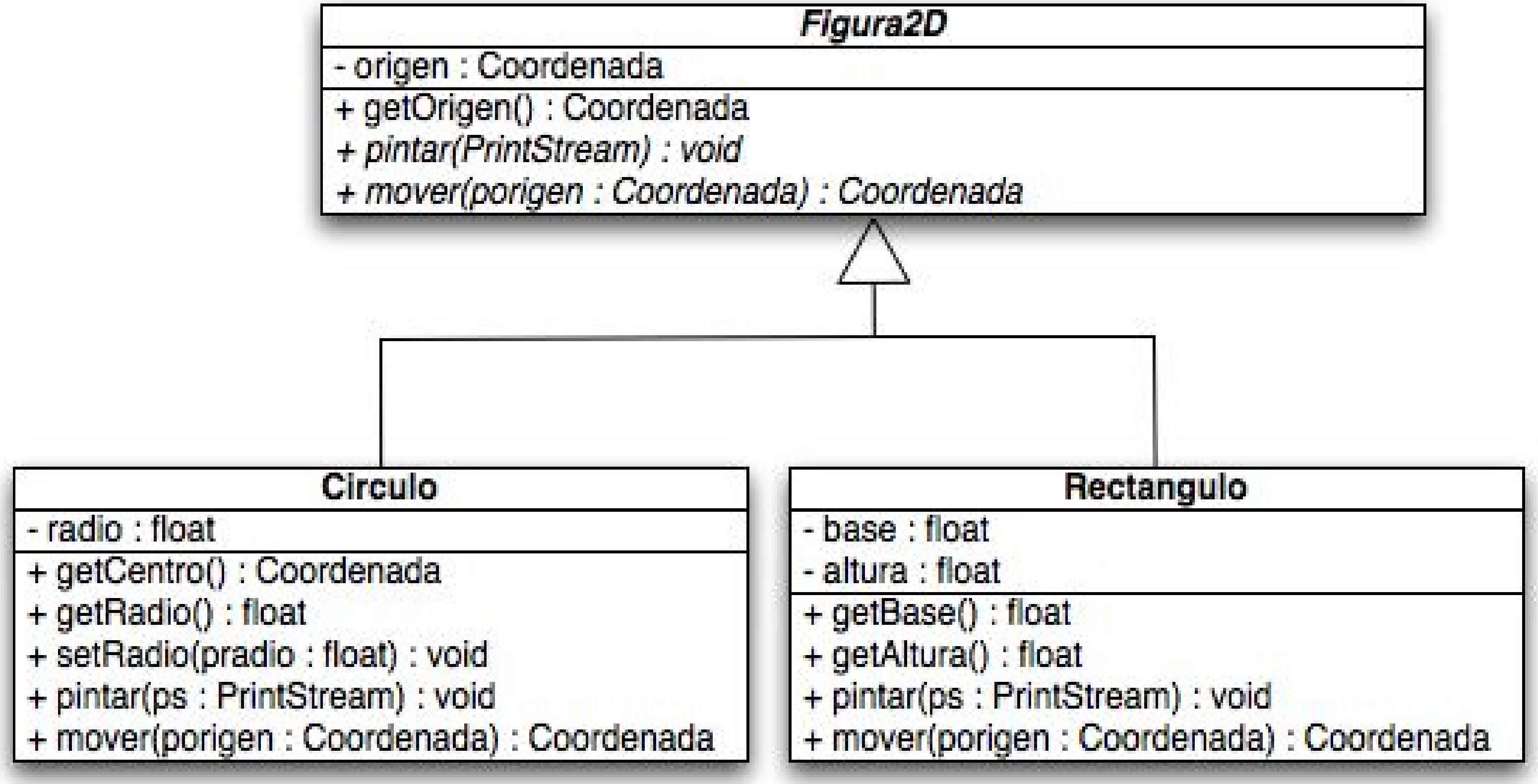


Figura2D
- origen : Coordenada
+ getOrigen() : Coordenada
+ pintar(PrintStream) : void
+ mover(porigen : Coordenada) : Coordenada
Circulo
- radio : float
+ getCentro() : Coordenada
+ getRadio() : float
+ setRadio(pradio : float) : void
+ pintar(ps : PrintStream) : void
+ mover(porigen : Coordenada) : Coordenada
Rectangulo
- base : float
- altura : float
+ getBase() : float
+ getAltura() : float
+ pintar(ps : PrintStream) : void
+ mover(porigen : Coordenada) : Coordenada

Catalog of refactorings

Replace Inheritance with Delegation/Composition

Motivation: The principle of substitutability is not holding. A subclass uses only part of a superclass' interface or does not want to inherit the attributes.

Solution: Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.

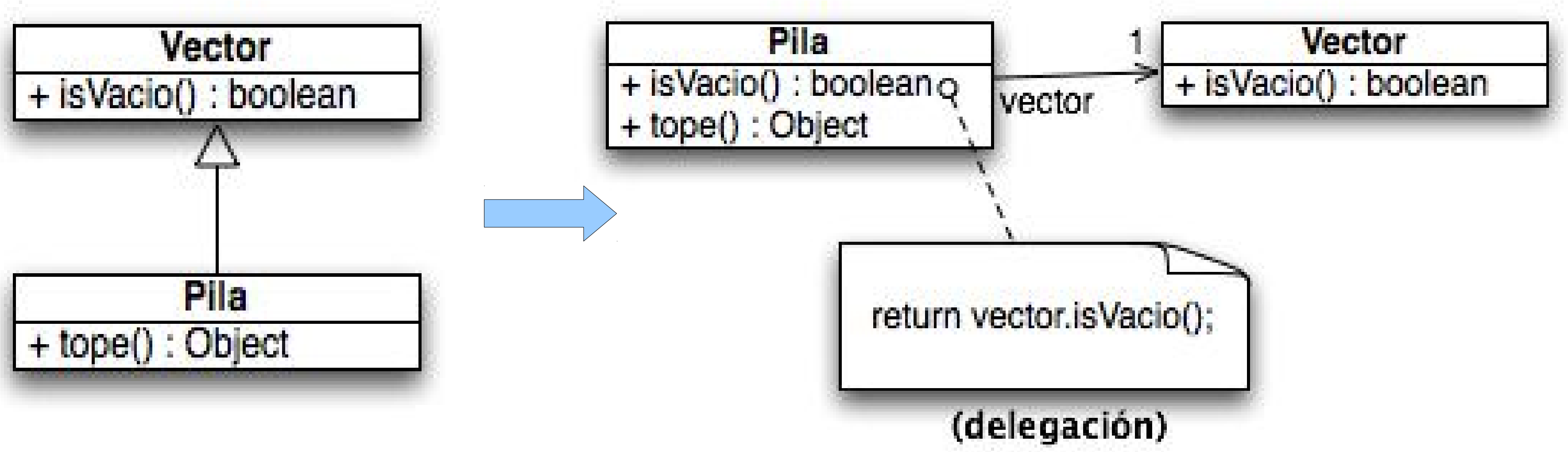
Remarks:

- Change each method defined in the subclass to use the delegate field.

Catalog of refactorings

Replace Inheritance with Delegation/Composition

Example:



Catalog of refactorings

More refactorings

There are many more refactorings than those presented here:

- Replace Data Value with Object
- Replace Array with Object
- Change Bidirectional Association to Unidirectional / Change Unidirectional Association to Bidirectional
- Remove Setting Method
- Convert Procedural Design to Objects
- Extract Hierarchy
- Separate Domain from Presentation

- etc.

Catalog of refactorings

Conclusions

- Refactoring is a systematic and safe procedure to introduce changes in the code of an application so that it is easier to understand, modify and extend.
- Some of the previous refactorings are automatically supported by some development environments (e.g., Eclipse for Java).
- The most important issue is to know what and when to refactor.
- Refactoring is considered as one of the most important software innovations:

http://www.dwheeler.com/innovation/innovation.html

Bibliography

In this topic we have not focused on the precise steps that can be followed to successfully implement the refactorings in a controlled and systematic fashion. These steps are the main content of the book by Martin Fowler:

Martin Fowler. ***Refactoring. Improving the Design of Existing Code***
Addison Wesley, 2007

http://martinfowler.com/refactoring/