Refactoring

Sandra Casas
Unidad Académica Río Gallegos
Universidad Nacional de la Patagonia Austral
2008

La **refactorización** ("*Refactoring"*) es una técnica de la ingeniería de software para reestructurar un código fuente, alterando su estructura interna sin cambiar su comportamiento externo.

Refactorización de código

En Ingeniería de Software, el término *refactoring* se usa a menudo para describir la modificación del código fuente sin cambiar su comportamiento, lo que se conoce informalmente por *limpiar el código*. La refactorización se realiza a menudo como parte del proceso de desarrollo del software: los desarrolladores alternan la inserción de nuevas funcionalidades y casos de prueba con la refactorización del código para mejorar su consistencia interna y su claridad. Los tests aseguran que la refactorización no cambia el comportamiento del código.
La refactorización es la parte del mantenimiento del código que no arregla errores ni añade funcionalidad. El objetivo, por el contrario, es mejorar la facilidad de comprensión del código o cambiar su estructura y diseño y eliminar código muerto, para facilitar el mantenimiento y evolución en el futuro. Añadir nuevo comportamiento a un programa puede ser difícil con la estructura dada del programa, así que un desarrollador puede refactorizarlo primero para facilitar esta tarea y luego añadir el nuevo comportamiento.
El término se creó como analogía con la factorización de números y polinomios. Por ejemplo, $x^2 - 1$ puede ser factorizado como $(x + 1)(x - 1)$, revelando una estructura interna que no era visible previamente (como las dos raíces en -1 y +1). De manera similar, en la refactorización del software, el cambio en la estructura visible puede frecuentemente revelar la estructura interna "oculta" del código original.
La refactorización debe ser realizada como un paso separado, para poder comprobar con mayor facilidad que no se han introducido errores al llevarla a cabo. Al final de la refactorización, cualquier cambio en el comportamiento es claramente un *bug* y puede ser arreglado de manera separada a la depuración de la nueva funcionalidad.
Un ejemplo de una refactorización trivial es cambiar el nombre de una variable para que sea más significativo, como una sola letra 't' a 'tiempo'. Una refactorización más compleja es transformar el trozo dentro de un bloque en una subrutina. Una refactorización todavía más compleja es remplazar una sentencia condicional `if` por polimorfismo.
Aunque la limpieza de código se lleva realizando desde hace décadas, el factor clave de la refactorización es realizar de manera intencionada, separándola de la adición de funcionalidad nueva, usando un catálogo conocido de métodos útiles de refactorización, para después comprobar el código, ejecutando las pruebas unitarias, sabiendo que cualquier cambio en el comportamiento significa que se ha introducido un error. La refactorización es un aspecto importante de la programación extrema.
El libro de Martin Fowler *Refactoring* es la referencia clásica. Aunque la refactorización de código se ha llevado a cabo de manera informal durante años, la tesis doctoral de William F. Opdyke (1993) es el primer trabajo conocido que examina específicamente esta técnica. Todos estos recursos proporcionan un catálogo de métodos habituales de refactorización. Un método de refactorización tiene una descripción de cómo aplicar el método e indicaciones sobre cuándo debería (o no debería) aplicarse.
La refactorización es un concepto tan importante que ha sido identificado como una de las más importantes innovaciones en el campo del software

Refactoring o cómo rehacer código

Muchas veces tenemos código ya hecho que está funcionando. También muchas veces necesitamos modificar ese código para que haga más cosas, para hacer más eficiente un algoritmo, más vistosa la salida del programa, porque tenemos otro proyecto que se parece, etc, etc.
También es normal que cuando querramos modificar el código, encontremos que el código ya hecho no es todo lo amigable que quisieramos. Introducir las modificaciones nos cuesta más de la cuenta porque hay métodos muy largos que no se entienden bien, hay pocos comentarios, hay trozos de código que nos sirven en parte, pero no del todo, clases muy grandes de las que nos sirven algunos métodos pero nos sobra el resto, etc, etc.

Una opción es apañarse con lo que se tiene, añadir la funcionalidad como podamos, copiar código, pegarlo en otro sitio y modificarlo para que se parezca a lo que queremos añadir, etc, etc. Esta opción tiene una desventaja, deja el código más "indeseable" todavía de lo que estaba. La próxima vez que queramos hacerle algo, será peor.
La otra opción, la buena, es arreglar el código existente antes de añadirle la funcionalidad que queremos. Si necesitamos un fragmento de código que está dentro de un método grande, partimos el método en dos, de forma que en un método tenemos lo que necesitamos y en otro el resto. Una vez hechos este tipo de arreglos, dejando el código funcionando igual que antes de hacerle los cambios, debería ser más fácil añadir la nueva funcionalidad.
Esta técnica de "arreglar" el código antes de abordar las modificaciones que realmente queremos introducir es lo que se conoce como "refactoring".

VENTAJAS DE HACER REFACTORING

Cualquier programador con un poco de experiencia sabe que nunca se diseña bien el código a la primera, que nunca nos dicen al principio todo lo que tiene que hacer el código, que nuestros jefes, según vamos programando y van viendo el resultado van pidiendo cosas nuevas o modificaciones, etc.
El resultado de esto es que nuestro código, al principio, puede ser muy limpio y estar bien organizado, siguiendo un diseño más o menos claro. Pero según añadimos cosas y modificaciones, cada vez se va "liando" más, cada vez se entiende pero y cada vez nos cuesta más depurar errores.
Si vamos haciendo refactoring sistemáticamente cada vez que veamos código feo, el código se mantiene más elegante y más sencillo. En fases más avanzadas de nuestro programa, seguirá siendo un código legible y fácil de modificar o de añadirle cosas.
Aunque inicialmente parece una pérdida de tiempo arreglar el código, al final del mismo se gana dicho tiempo. Las modificaciones y añadido tardan menos y se pierde mucho menos tiempo en depurar y entender el código.

HACER TEST

Una cosa importante del refactoring es que se parte de un código que más o menos funciona, si se modifica y el código debe seguir funcionando igual que antes. Si hacemos refactoring con frecuencia, es importante tener algún medio de probar que el código sigue funcionando después de "arreglarlo".
Una opción es, después de arreglar el código, compilarlo, ejecutarlo y probar que más o menos sigue funcionado, especialmente en las partes de código que hemos tocado.
Otra opción algo más "profesional" es hacer unos pequeños programas de prueba. Estos programas (conocidos como test unitarios) deben instanciar la clase que vamos a tocar, llamar a sus métodos pasando determinados parámetros concretos y ver que devuelve el resultado adecuado. Una vez que tenemos el test, arreglamos el código de esa clase y luego pasamos el test. Si pasa el test, podemos tener cierta garantía de que todo sigue funcionando igual que antes.
Para que todo el proceso funcione bien y no haya demasiados problemas, es importante hacer un pequeño cambio y pasar el test, otro pequeño cambio y volver a pasar el test y así hasta que hayamos hecho el cambio que queramos hacer. Nunca debemos embarcarnos en un cambio grande, pasarnos varias horas arreglando código sin pasar ningún tipo de prueba y al final hacer la prueba. Si la prueba no pasa, tardaremos también un buen tiempo en encontrar dónde está el fallo.
Tanto el refactoring como los test unitarios son pilares fundamentales de la programación extrema. En Java se da soporte a este tipo de tests unitarios por medio de JUnit. La inmensa mayoría de los IDEs de desarrollo, como Eclipse, tienen opciones de "refactoring" en las que señalamos un fragmento de código y le decimos qué queremos hacer con él (cambiar nombre de variable, extraer el código en un método nuevo, mover un método de una clase hija a una clase padre, etc, etc). El IDE hace todo el cambio y arregla en todas las demás clases lo que sea necesario.

UN LIBRO SOBRE REFACTORING

Según muchas opiniones en muchos sitios, el libro por excelencia sobre refactoring es "*Refactoring: Improving the Design of Existing Code*" de Martin Fowler. La web de refactoring es http://www.refactoring.com. El libro es realmente interesante, especialmente un capítulo en el que comenta qué cosas se deben arreglar en el código. La idea en ese capítulo es que debe arreglarse en el código cualquier cosa que "huela mal", o comúnmente "**code smell**". Algunas de esas cosas son:

- **Código duplicado**. Bien hay líneas de código exactamente iguales en varios sitios o bien hay líneas de código muy parecidas o con estructura similar en varios sitios. Se deberían extraer métodos y usar cosas como el patrón estrategia para hacer esa parte que es diferente entre las dos copias.
- **Métodos muy largos**. Obviamente, hay que partilo en métodos más pequeños.
- **Clases muy grandes**. En estos casos debería tratar de identificarse qué cosas hace esa clase, ver si realmente todas esas cosas tienen algo que ver la una con la otra y si no es así, hacer clases más pequeñas, de forma que cada una trate una de esas cosas. Por ejemplo, si una clase es una ventana, echa cuentas y escribe los resultados en base de datos, ya está haciendo demasiadas cosas. Debería haber una clase que sea la ventana, otra que eche las cuentas y otra que sepa escribir en base de datos.
- **Métodos que necesitan muchos parámetros**. Suele ser buena idea hacer una clase que contenga esos parámetros y pasar la clase en vez de todos los parámetros. Especialmente si esos parámetros suelen tener que ver unos con otros y suelen ir juntos siempre.
- **Los switch** son feos. Normalmente un switch-case se tiene que repetir en el código en varios sitios, aunque en cada sitio sea para hccer cosas distintas. Existen formas usando el polimorfismo que evitan tener que repetir el switch-case en varios sitios, o incluso evitan tener que ponerlo en ningún lado.

Anteriormente se comento que los cambios deben hacerse poco a poco, probando después de cada poco. Gran parte del libro de "refactoring" son "recetas" de cambios. Para cada posible cambio que queramos realizar (por ejemplo, partir un método en métodos más pequeños), nos dice qué mini-cambios debemos hacer y en qué orden. Por ejemplo, para partir un método grande en varios pequeños (esto es inventado, no lo que pone el libro):

- Crear los nuevos métodos vacios y probar. Todo debería funionar igual ya que nadie llama a esos métodos y además están vacios. Puede que haya algún fallo si casualmente hemos elegido de nombre de método el mismo que el de una clase padre y tenemos un polimorfismo que nos pasa inadvertido.
- Metemos el código en esos métodos, pero sin modificar el método original. Probamos. Todo debería funcionar igual, nadie llama a esos métodos. Posiblemente deberiamos probar que esos nuevos métodos funcionan correctamente, añadiendo un nuevo test.
- Vamos reemplazando en el método original los trozos de código por llamadas a los métodos parciales. Con cada reemplazo, probamos.
- ...

RESUMEN DE BAD SMELLS

1) **Código Duplicado**: la misma estructura de código esta en vs lugares.
   (110) Extract Method
   (320) Pull Up Field
   (345) Form Template Method
   (139) Substitute algorithm
   (149) Extract Class

2) **Método Largo**: los métodos largos son más difíciles de entender.
   (110) Extract Method
   (120) Replace Temp with Query
   (295) Introduce Parameter Object
   (135) Replace method to method object
   (238) Decompose Condicional

3) **Clases Largas**: cuando una clase hace muchas cosas se tendran muchas instancias de la misma: codigo duplicado.
   (149) Exctract Class
   (330) Extract Subclass
   (341) Extract Interface
   (189) Duplicate Observed Data

4) **Lista de parámetros larga**: la lista de parámetros es extensa
   (292) Replace Parameter with Method
   (295) Introduce Parameter Object

5) **Cambios Divergentes**: ocurre cuando un cambio en una clase afecta a vs metodos
   (149) Exctract Class

6) **Shotgun Surgery**: Ocurre cuando un cambio afecta vs clases.
   (142) Move Method
   (146) Move Field
   (154) Inline Class

7) **Feature Envy** (Caracteristica envidiada): cuando un metodo/datos esta mas interesado en una clase que no es la actualmente la contiene.
   (142) Move Method
   (110) Extract Method

8) **Grupos de Datos**: 3 o 4 o mas datos estan siempre junetos en vs lugares: atributos de distintas clases, parámetros en vs signaturas de metodos
   (149) Exctract Class
   (295) Introduce Parameter Object
   (288) Preserve Whole Object

9) **Primitiva Obsesión**: no usar datos primitivos.
   (175) Replace Data Value with object
   (218) Replace type code with class
   (223) Replace type code with subclass
   (227) Replace type code with state/strategy
   (149) Exctract Class
   (295) Introduce Parameter Object
   (186) Replace Array with Object

10) **Instrucciones Switch**: genera código duplicado.
(110) Extract Method
(142) Move Method
(223) Replace type code with subclass
(227) Replace type code with state/strategy
(255) Replace Conditional with Polymorphism
(285) Replace Parameter with Explicit Methods
(260) Introduce Null Object

11) **Jerarquia de Herencia Paralela**: cada vez que se agrega una subclase a una clase; hay que agregar una subclase a otra clase
(142) Move Method
(146) Move Field

12) **Clase Perezosa**: Una clase que no hace mucho: eliminarla
(344) Collapse Hierarchy
(154) Inline Class

13) **Generalidad Especulativa**: una clase que se le ha agregado funcionalidad por las dudas.(no requerida)
(154) Inline Class
(344) Collapse Hierarchy
(277) Remove Parameter
(273) Rename Method

14) **Atributos Temporales**: cuando un atributo cumple la función de variable.
(149) Exctract Class
(260) Introduce Null Object

15) **Cadenas de Mensajes**: (grafico)
Este tipo de navegación significa que el cliente esta acoplado a la estructura de navegación. Cualquier cambio en las relaciones intermedias causa un cambio en el cliente.
(157) Hide Delagte
(110) Extract Method
(142) Move Method

16) **Hombre del Medio**: Una clase que todas sus responsabilidades las delega en otras clases. Ej director. (no es el caso de los gestores-contenedores)
(160) Remove Middle Man
(117) Inline Method
(355) Replace Delegation with Inheritance

17) Intimidad Inapropiada:
(142) Move Method
(146) Move Field
(200) Change Bidirectional Association to Unidirectional
(149) Exctract Class
(157) Hide Delagte
(355) Replace Delegation with Inheritance

18) **Clases alternativas con diferentes interfaces**: metodos que hacen lo mismo pero tienen distinta signature.
(273) Rename Method
(142) Move Method
(336) Extract Superclass

19) **Librería de clases incompleta:** es imposible modeificar una librería.

(142) Move Method
(162) Introduce Foreign Method

20) **Clase de Datos**: las clases que tienen atributos y solo metodos get/set y estos estan declarados publicos.
(206) Encapsulate Field
(208) Encapsulate collection
(300) Remove Setting Method
(142) Move Method
(110) Extract Method
(303) Hide Method

21) **Legado rechazado:** cuando una subclase no quiere o necesita todo lo que hereda, significa que la jerarquia es equivocada
(328) Push Down Method
(329) Push Down Field
(352) Replace Inheritance with Delegation

22) **Comentarios**: si en un segmento de código o en un método nos vemos en la necesidad o creemos que es adecuado poner un comentario, es porque ese código no es lo suficientement claro. Lo ideal es hacer un método con el nombre lo suficientemente claro como para que no necesite comentario.
Por ejemplo, si se tiene este código:

```
a=33*b-Math.round(b+cerdoGordo);
```

para que se entienda le pongo un comentario del estilo

```
// calcula la cantidad de grasa de un chorizo de mi pueblo
a=33*b-Math.round(b+cerdoGordo);
```

Lo ideal es hacer un método con el nombre lo suficientemente claro como para que no necesite comentario. Por ejemplo:

```
public double calculaCantidadGrasaChorizo (double b, double cerdoGordo)
{
  return 33*b-Math.round(b+cerdoGordo);
}
```

y en donde hacíamos el cálculo simplemente ponemos

```
a=calculaCantidadGrasaChorizo (b, cerdoGordo);
```

y sobra el comentario. O se puede poner, pero es redundante.

Clasificación y objetivo de cada grupo

Composing Method
(110) Extract Method
(117) Inline Method
(119) Inline Temp
(120) Replace Temp with Query
(124) Introduce Explaining Variable
(128) Split Temporary variable
(131) Remove Assignments to Parameters
(135) Replace method to method object
(139) Substitute algorithm

Objetivo: Hacer los metodos mas cortos y la logica de los métodos menos compleja.

Mover Funcionalidad entre Objetos
(142) Move Method

(146) Move Field
(149) Exctract Class
(154) Inline Class
(157) Hide Delagte
(160) Remove Middle Man
(162) Introduce Foreign Method
(164) Introduce Local Extension

Objetivo: Una de las decisiones fundamentales en el diseño orientado a objetos es dónde poner las responsabilidades. Puesto que es casi imposible hacerlo bien a la primera, las siguientes factorizaciones nos ayudan lidiar con este problema.

Organizar los datos

(175) Replace Data Value with object
(204) Replace magic number with symbolic constant
(208) Encapsulate collection
(218) Replace type code with class
(223) Replace type code with subclass
(227) Replace type code with state/strategy
(232) Replace subclass with fields
(171) Self Encapsulate Field
(179) Change Reference to Value
(186) Replace Array with Object
(189) Duplicate Observed Data
(197) Change Unidirectional Association to Bidirectional
(200) Change Bidirectional Association to Unidirectional
(206) Encapsulate Field
(217) Replace Record with Data Class

Objetivo: Las siguientes refactorizaciones tienen que ver con la organización de los datos (estructuras de clases, relaciones, etcétera).

Simplificar expresiones condicionales

(267) Introduce Assertion
(260) Introduce Null Object
(255) Replace Conditional with Polymorphism
(250) Replace Nested Conditional with Guard Clauses
(245) Remove Control Flag
(243) Consolidate Duplicate Conditional Fragments
(240) Consolidate Conditional Expression
(238) Decompose Conditional

Objetivos: La lógica condicional suele ser tan engorrosa y complica tanto el código que hay un número de refactorizaciones que podemos usar para simplificarlas. Ya hemos visto también cómo muchos patrones de diseño perseguían, más o menos, el mismo objetivo.

Hacer más simples las llamadas a métodos

(273) Rename Method
(275) Add Parameter
(277) Remove Parameter
(279) Separate Query from Modifier
(283) Parameterize Method
(285) Replace Parameter with Explicit Methods
(288) Preserve Whole Object
(292) Replace Parameter with Method
(295) Introduce Parameter Object

(300) Remove Setting Method
(303) Hide Method
(304) Replace Constructor with Factory Method
(308) Encapsulate Downcast
(310) Replace Error Code with Exception
(315) Replace Exception with Test

Objetivo: estas refactorizaciones buscan que las interfaces de los objetos sean mas claras.

Manejar la generalización

(320) Pull Up Field
(322) Pull Up Method
(325) Pull Up Constructor Body
(328) Push Down Method
(329) Push Down Field
(330) Extract Subclass
(336) Extract Superclass
(341) Extract Interface
(344) Collapse Hierarchy
(345) Form Template Method
(352) Replace Inheritance with Delegation
(355) Replace Delegation with Inheritance

Objetivo: Estas refactorizaciones buscan organizar la jerarquia de herencia adecuadamente.

COMPOSING METHOD

(110) Extract Method

- Es una de las refactorizaciones más comunes
- Consiste en extraer una porción de código y crear un método propio para dicho código
- ¿Qué se consigue? Evitar los métodos largos y Eliminar comentarios del código
- O, dicho de otro modo, hacer que éste sea autodocumentado, que *revele su intención*
- Para que funcione, el nombre de los métodos tiene que ser lo suficientemente claro
- Debe expresar el propósito del método

```
void printOwing()
{
  printBanner();
  //print details
  System.out.println ("name: " + name);
  System.out.println ("amount " + outstanding());
}
```

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

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

Problemas:

- Variables locales
- Si el código extraído modifica el valor de una, tratar de utilizar el nuevo método como una consulta y asignar el resultado a dicha variable
- Si no es posible o hay más de una variable:
- *(128) Split Temporary Variable*
- *(120) Replace Temp with Query*
- Si únicamente las lee, se pueden pasar como parámetros al nuevo método

(117) Inline Method

El cuerpo de un método es tan claro como su nombre.
Poner el código del método en el cuerpo de sus llamadores y eliminar el método.

- Sería lo contrario del anterior
- A veces, tanta indirección es superflua, o hace demasiado difícil seguir el código
- También puede hacerse antes de volver a factorizar un método
- Por ejemplo, antes de aplicar *(135) Replace Method with Method Object*

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

```
}
```

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

(119) Inline Temp

Hay una variable temporal a la que se le asigna una sola vez el resultado de una expresión, y dicha variable temporal dificulta la aplicación de otras refactorizaciones.
Reemplazar todas las referencias a dicha variable temporal con la expresión.

- Se utiliza dentro de *(120) Replace Temp with Query*
- Tendría sentido por sí sola cuando lo que se asigna a la variable es el resultado de una llamada a un método

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

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

(120) Replace Temp with Query

Se está usando una variable temporal para almacenar el resultado de una expresión.
Extraer la expresión a un método. Sustituir todas las referencias a la variable temporal con una llamada al método.

- Las variables temporales tienden a favorecer los métodos largos
- Precisamente por ser temporales y locales
- Al usar un método aparte, éste puede llamarse desde otros métodos de la clase
- Normalmente es necesario aplicarlo antes de proceder al *(110) Extract Method* para eliminar todas las variables locales que podamos

```
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;
}
```

(124) Introduce Explaining Variable

Estamos frente a una expresión complicada.
Poner el resultado de la expresión (o de las distintas partes de ésta) en una variable temporal con un nombre que explique su propósito.

- Aunque es mejor, generalmente, no usar variables temporales, en el caso de expresiones complejas pueden simplificar el código
- Por ejemplo, con varias sentencias condicionales, o en un algoritmo largo
- En principio sería mejor tratar de aplicar *(110) Extract Method*
- Pero hay veces, cuando el método tiene muchas variables locales, en que puede venir bien

```
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
}
```

(128) Split Temporary Variable

Tenemos una variable temporal a la que se le asigna un valor más de una vez, sin ser un bucle ni una variable "recolectora".
Crear una variable para cada asignación.

Asignar varios valores a una variable local tiene sentido en los siguientes casos:
- Un bucle
- Una variable "recolectora" (*"collecting variable"*)

Si no, es síntoma de que estamos usando la misma variable para cosas distintas
- Lo que es muy confuso para el lector del código

```
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);
```

(131) Remove Assignments to Parameters (Eliminar asignaciones a parámetros)

El código de un método tiene una asignación a un parámetro.
Usar una variable temporal en vez de eso.

- No hay nada malo en hacer algo sobre el parámetro (modificarlo a través de sus métodos)
- El problema es cambiar la referencia
- Lleva a confusión (hace oscura la distinción entre paso por valor y por referencia)
- Java usa únicamente paso por valor

```
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;
  ...
```

(135) Replace Method with Method Object

Hay un método largo que usa variables locales de tal forma que no es posible aplicar *(110) Extract Method*.

Convertir el método en un objeto, de modo que todas las variables locales sean atributos de dicho objeto. A continuación se puede descomponer el método en varios métodos del mismo objeto.

- Hay veces en que ni siquiera usando *(120) Replace Temp with Query* es posible aplicar *(110) Extract Method*
- En ese caso, haríamos de este método un objeto y luego aplicaríamos *(110) Extract Method* sobre el nuevo objeto

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

```
class PriceCalculate {
 private final Order _order;
 private float _n;
 private primaryBasePrice;
 private secondaryBasePrice;
 private tertiaryBasePrice;

 PriceCalculate(Order source, float n)
 {
  _order = order;
  _n = n;
 }
 duoble compute()
 {
 // long computation;
 ...
}
```

```
class Order



double price(float n)
{
 return new PriceCalculate(this,n);
}
```

(139) Substitute Algorithm

Queremos reemplazar un algoritmo por otro que es más claro.
Cambiar el cuerpo del método con el nuevo algoritmo.

```
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 null;
}
```

```
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 null;
}
```

MOVING FEATURES BETWEEN OBJECTS

(142) Move Method

Un método está usando más funcionalidad de otra clase que de aquélla en la que está definido.
Crear un nuevo método con un cuerpo similar en la otra clase. A continuación, convertir el método anterior en una simple delegación o eliminarlo por completo.

```
class Customer {
 …
 private double amountFor( Rental aRental ) {
 double result = 0;
 switch (aRental.getMovie().getPriceCode()) {
  case Movie.REGULAR:
   result += 2;
   if (aRental.getDaysRented() > 2)
   result += (aRental.getDaysRented() - 2) * 1.5;
   break;
  case Movie.NEW_RELEASE:
   result += aRental.getDaysRented() * 3;
   break;
  case Movie.CHILDRENS:
   result += 1.5;
   if (aRental.getDaysRented() > 3)
   result += (aRental.getDaysRented() - 3) * 1.5;
   break;
```

```
  }
 return result;
}
…
}
```

- En el código anterior (de un videoclub), el método amountFor, definido en la clase Customer, utiliza extensivamente métodos de la clase Rental
- Parece más lógico moverlo a esa clase
- Es una responsabilidad de la clase *Alquiler*, no del *Cliente*

```
class Rental {
 …
 double getCharge() {
  double result = 0;
  switch (getMovie().getPriceCode()) {
   case Movie.REGULAR:
    result += 2;
    if (getDaysRented() > 2)
    result += (getDaysRented() – 2) * 1.5;
    break;
   case Movie.NEW_RELEASE:
    result += getDaysRented() * 3;
    break;
   case Movie.CHILDRENS:
    result += 1.5;
    if (getDaysRented() > 3)
    result += (getDaysRented() – 3) * 1.5;
    break;
   }
  return result;
 }
…
}
class Customer
{
 ...
 private double amountFor(Rental aRental)
 {
  return aRental.getCharge();
 }
 ...
}
```

- Nótese que además de mover el método hemos cambiado el nombre por otro más apropiado para su nueva ubicación

(146) Move Field

Un campo es usado más por otra clase que por aquélla en la que está definida.
Crear un nuevo atributo en la otra clase y cambiar todos los clientes.

- El uso puede ser indirecto, a través de los métodos *get* y *set*

(149) Extract Class

Hay una clase haciendo trabajo que deberían hacer entre dos.
Crear una nueva clase y mover los atributos y métodos relevantes de la clase antigua a la nueva.

```
class Person {
```

```
String name;
String dni
Date FechaNac;
String calleDomicilio;
int numeroDomicilio
int pisoDomicilio
int deptoDomicilio
int cpDomicilio

// Metodos set/get para cada atributo
```

class Person String name; long dni Date fechaNac; Domicilio domicilio; // Metods set/get para cada atributo	class Domicilio { String calleDomicilio; int numeroDomicilio int pisoDomicilio int deptoDomicilio int cpDomicilio // Metods set/get para cada atributo

(154) Inline Class

Una clase no está haciendo gran cosa.
Mover su funcionalidad a otra clase y eliminarla.

- El contrario del anterior
- A menudo es el resultado de aplicar otras refactorizaciones que, al sacar responsabilidades fuera de la clase, hacen que ésta tenga ahora poco sentido

Recordar:

- ¿Cuándo modelar algo como una clase aparte? Cuando ese algo tenga un c***omportamiento*** diferente

(157) Hide Delegate

Un cliente está llamando a la clase delegada de un objeto.
Crear métodos en el servidor que oculten dicha delegación.

class Person
// server
Departament dep;
// get-set
	class Departament //delegado
	String code;
	Person mamager;
	//set-get
		// un cliente
		manager= john.getDepartament().getManager();

class Person // server
Departament dep;
// get-set
Person getManager()
{ return dep.getManager()}
	Class Departament (delegado)
	String code
	Person mamager
	//set-get
		// un cliente
		manager= john.getManager();

- La encapsulación es una propiedad fundamental de los objetos
- Cada objeto necesita conocer lo menos posible de otras partes del sistema

- Así, cuando las cosas cambien, se verán afectados menos objetos (lo que hace el cambio más fácil)

(160) Remove Middle Man

Una clase está haciendo demasiada delegación.
Dejar que el cliente llame al delegado directamente.

```
class Cliente/s {
--
 Director d = new
 Director();
 d.correspondencia();
 d.coordinarReunion();
 d.ajustarFinanzas()
...
```

```
class Director {
  Secretaria s;
  Cadete c;
  SubDirector b;
  correspondencia() { c.enviar(); }
  coordinarReunion() { s.coordinar();)
  ajustarFinanzas() {b.ajustar(); }
```

```
class Cliente/s {
--
 Secretaria s;
 Cadete c;
 SubDirector b;
 c.enviar();
 s.coordinar();
 b.ajustar();
```

- Aunque lo dicho en *Hide Delegate* es cierto, si hay muchos de estos métodos que únicamente sirven para ocultar la delegación al cliente es posible que la interfaz de la clase se oscurezca
- En tales casos, volveríamos a la situación original
- ¿Cuándo aplicar uno u otro?
- Depende; lo que hace seis meses era válido puede que ahora no

(162) Introduce Foreign Method

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

```
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);
}
```

(164) Introduce Local Extension

A server class you are using needs several additional methods, but you can't modify the class.
Create a new class that contains these extra methods. Make this extension class a subclass or a wrapper of the original.

ORGANIZING DATA

(171) Self Encapsulate Field

You are accessing a field directly, but the coupling to the field is becoming awkward.
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;}
```

(175) Replace Data Value with Object

Descubrimos que un dato necesita datos o (sobre todo) comportamiento adicional.
Convertimos el dato en un objeto.

```
class Order {
  String costumer;
  // otros atributos de Order

setCostumer…
getCostumer
// otros metodos de Order
}
```

class Order {
  Costumer _costumer;
 // otros atributos de Order
 	
setCostumer..
getCostumer..
// otros metodos de Order
}
	class Costumer {
	  String name;
	  // otros atributos de costumer
	
	setName…
	getName..
	// otros metodos de Costumer
	}

- Un número de teléfono puede ser representado mediante un String durante un tiempo, para más tarde darnos cuenta de que necesita comportamiento para ser formateado, extraer el prefijo, etcétera
- Si son sólo un par de métodos, tal vez puedan definirse en el objeto al que pertenece
- Pero en cuanto percibamos que eso nos lleva a duplicar código, a que una clase haga demasiadas cosas... convertiremos el dato en un objeto por sí mismo

(179) Change Value to Reference

You have a class with many equal instances that you want to replace with a single object.
Turn the object into a reference object.

(186) Replace Array with Object

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

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

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

(189) Duplicate Observed Data

You have domain data available only in a GUI control, and domain methods need access.
Copy the data to a domain object. Set up an observer to synchronize the two pieces of data.

(197) Change Unidirectional Association to Bidirectional

You have two classes that need to use each other's features, but there is only a one-way link.
Add back pointers, and change modifiers to update both sets.

Doing a remove

In the example I showed an *addOrder* method, but I didn't show the *removeOrder* method. If you want to do a remove, you would write it like the add method but set the customer to null.

```
Class Customer ...
 void removeOrder( Order arg ) {
  arg.setCustomer( null );
}
```

(200) Change Bidirectional Association to Unidirectional

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

(204) Replace Magic Number with Symbolic Constant

Tenemos un número con un significado especial.
Crear una constante, darle un nombre acorde con su significado y reemplazar el número con ella.

- Incluso aunque el número no vaya a cambiar, aumenta la legibilidad del código
- Antes de hacer esta refactorización, deberíamos pensar si hay alguna solución mejor
- Por ejemplo, Replace Type Code with Class

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

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

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

- Hay, no obstante, una forma mejor de hacerlo, simulando los tipos enumerados (que la versión 1.5 ya posee)

(206) Encapsulate Field

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

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

(208) Encapsulate Collection

Un método devuelve una colección.
Hacer que devuelva una copia inmutable y proporcionar métodos add/remove.

```
class Curso {
 private Vector alumnos;

Vector getCurso()
{ return alumnos; }
```

```
 class Curso {
 private Vector alumnos;

Vector getCurso()
{ Vector copia = new Vector();
  // copiar alumnos en copia
  return copia; }
```

- Las colecciones no deberían tener los típicos métodos 'getter' y 'setter'
- Implementan un protocolo ligeramente distinto
- El método get no debe devolver la propia colección, ya que eso permitiría que los clientes manipulasen los contenidos de la colección sin la mediación de la clase que la posee
- No habrá método set, sino operaciones add/remove para añadir y eliminar elementos
- Devolvemos una vista no modificable de la colección:

```
public Set getCourses()
{
 return Collections.unmodifiableSet(courses);
}
```

(217) Replace Record with Data Class

You need to interface with a record structure in a traditional programming environment.
Make a dumb data object for the record.

(218) Replace Type Code with Class

Una clase tiene un código de tipo numérico que no afecta a su comportamiento.
Reemplazar el número con una clase.

```
class Person {
```

```
  private int cuil_sexo;
  private int cuil_dni;
  private int cuil_dv;
  private int cuil;
```

```
class Person {
  private Cuil cuil;

class Cuil {
  private int cuil_sexo;
  private int cuil_dni;
  private int cuil_dv;
  private int cuil;
```

- Es lo que se comentó a propósito de *Replace Magic Number with Symbolic Constant*
- Presenta la ventaja de ser seguro con respecto al tipo
- Las constantes simbólicas son solamente un alias; el compilador únicamente ve un número
- Otras refactorizaciones posibles son:
- *(223) Replace Type Code with Subclasses*
- *(227) Replace Type Code with State/Strategy*

(223) Replace Type Code with Subclasses

Tenemos un código de tipo inmutable que afecta al comportamiento de la clase.
Reemplazar el código por subclases.

```
class Empleado {
  private int tipo;
  static final INGENIERO=0;
  –
  –
 Empleado (int _tipo)
 { tipo = _tipo }

// luego todos los metodos testean el tipo para aplicar la logica especifica
```

```
class Empleado {
  Empleado create (int _tipo)
  { return new Empleado (_tipo); }

  static Empleado create (int _tipo)
  { // factory method
    if (_tipo == INGENIERO)
       return new Ingeniero();

}
```

```
class Ingeniero extends Empleado {

int getTipo ()
{ return Empleado.INGENIERO;};
```

- Si el código de tipo no afecta al comportamiento, podemos usar *Replace Type Code with Class*
- En caso de que varía el comportamiento, lo mejor es dejar que sea el polimorfismo quien maneje los distintos comportamientos. (Es lo que hace esta refactorización)
- Un síntoma de esta situación son las sentencias condicionales switch o if anidados
- Esta refactorización es necesaria para luego poder aplicar *Replace Condictional with Polymorphysm*
- ¿Y si el tipo del objeto puede variar durante la vida de éste? (o si ya hay otra estructura de herencia)
- Entonces, aplíquese *Replace Type Code with State/Strategy*
- Otra razón para aplicar esta refactorización es la presencia de funcionalidad que sólo es relevante para objetos de ciertos tipos
- En ese caso, una vez aplicada, podemos usar *Push Down Method* y *Push Down Field* para reflejar eso en el diseño

(227) Replace Type Code with State/Strategy

Tenemos un código de tipo que afecta al comportamiento de la clase, pero no podemos usar subclases.
Reemplazar el código por un objeto estado o estrategia.

Ver Patrones de diseño State o Strtaegy

(232) Replace Subclass with Fields

Tenemos subclases que varían sólo en métodos que devuelven datos constantes.
Convertir los métodos en campos de la superclase y eliminar las subclases.

- Este tipo de "métodos constantes" (Beck) son muy útiles y muy frecuentes cuando tenemos subclases que deben devolver distintos valores para un atributo
- Pero lo que nos dice esta refactorización es que no aplicaríamos la herencia sólo para eso
- No existe un comportamiento distinto

```
class Persona {
 char code;
 getCode()
}

class Mujer extends Persona {
getCode()
{ return "M";}

class Persona {
 char code;
getCode()
{ return code;}
```

SIMPLIFYNG CONDITIONAL EXPRESSIONS

(238) Decompose Conditional

Tenemos una sentencia condicional compleja.
Extraer métodos para la condición, el "then" y el "else".

```
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);
```

(240) Consolidate Conditional Expression

Hay una secuencia de pruebas condicionales con el mismo resultado.
Combinarlas en una única expresión condicional y extraerla a un método.

```
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
```

(243) Consolidate Duplicate Condicional Fragments

Se repite el mismo fragmento de código en todas las ramas de una expresión condicional.
Sacarlo de la expresión.

```
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();
```

(245) Remove Control Flag

Una variable está actuando como un indicador de control para una serie de expresiones booleanas.
En vez de eso, usar un "break" o un "return".

- Los lenguajes de programación cuentan con las sentencias break y continue precisamente para evitar esta complejidad
- También se puede utilizar return

```
encontrado = false;
while (!encontrado)
 if (condicion)
 {
  // hacer algo
  encontrado = true;
 }
```

Primer caso: empleando break o continue
Otro enfoque:
- Extraer la lógica en un método

- Sustituir las asignaciones a la variable de control por un return
- Suele ser más claro empleado return
- Además, esto vale incluso si la variable de control se utilizaba para devolver algún valor

```
void checkSecurity(String[] people)
{
 String found = "";
 for (int i = 0; i < people.length; i++)
 {
  if (found.equals(""))
  {
   if (people[i].equals("Don"))
   {
    sendAlert();
    found = "Don";
   }
   if (people[i].equals("John"))
   {
   sendAlert();
   found = "John";
   }
  }
 }
 someLaterCode(found);
}
```

Ejemplo utilizando return

- En el ejemplo anterior, found hace dos cosas:
- Actúa como variable de control
- Guarda el resultado
- Podemos extraer el código usado para obtener el valor de found a un método aparte:

```
void checkSecurity(String[] people)
{
 String found = foundMiscreant(people);
 someLaterCode(found);
}

String foundMiscreant(String[] people)
{
 for (int i = 0; i < people.length; i++)
 {
  if (people[i].equals("Don"))
  {
   sendAlert();
   return "Don";
  }
  if (people[i].equals("John"))
  {
   sendAlert();
   return "John";
  }
 }
 return "";
}
```

- Podemos utilizar el mismo enfoque aun cuando no se devuelva ningún valor
- En ese caso, simplemente pondremos un return sin parámetro

Problema:

- Cuando la función tiene efectos laterales
- En ese caso podemos utilizar *Separate Query from Modifier*

(250) Replace Nested Conditional with Guard Clauses

Un método tiene comportamiento condicional que oscurece el flujo normal de ejecución.
Usar cláusulas "guardianas" para todos los casos especiales.

- Dos tipos de sentencias condicionales:
- Unas en las que cualquiera de las dos ramas es parte del flujo normal de ejecución
- Para ellas, usaremos un if con su else
- Otras que se utilizan para comprobar condiciones inusuales ante las cuales el método debe finalizar
- Las sustituiremos por una cláusula guardiana (Beck)
- Ayuda a clarificar la intención del código

```
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();
}
```

(260) Introduce Null Object

Hay muchas comprobaciones de si es un valor null.
Sustituir el valor null por un objeto nulo.

- La esencia del polimorfismo es que en vez de preguntarle a un objeto de qué tipo es y a continuación invocar a algún comportamiento basado en dicho tipo, simplemente invocamos al comportamiento
- El propio objeto, dependiendo de su tipo, hará lo que tenga que hacer (polimorfismo)
- Uno de los ejemplos menos intuitivos de polimorfismo es cuando tenemos un valor nulo en un campo

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

```
class NullCustumer extends Customer {
 boolean isNull() { return true;)
 // metodos que hereda de Custumer con la logica especifica de objeto
nulo

class Customer {
 boolean isNull() { return false;)
```

```
 static Customer newNull()
  { return new NullCostumer();}
 // los metodos comunes de Custumer no nulo

// Clase cliente
Customer getCustomer()
{ if (_customer == null)
    return Customer.newNull() // factory method
  else retuen _customer;
}
..
```

- Los objetos nulos son siempre constantes
- Por tanto, los implementamos usando el patrón *Singleton*
- Cada vez que solicitamos una persona que no existe, obtendremos la misma instancia de la clase NullCustomer
- Tienen sentido cuando la mayoría de los clientes esperan la misma respuesta
- Tampoco hace falta que sean todos (esos pocos pueden seguir haciéndolo como antes)

Null Object. Casos especiales
- También se puede aplicar a los casos especiales (por ejemplo, UnknownCustomer, NaN...)
- Ayudan a reducir el manejo de errores
- Las operaciones en coma flotante con valores infinitos (NaN, *"Not a number"*) no lanzan una excepción:
- Devuelven otro **NaN**

(267) Introduce Assertion

Una sección de código hace una asunción sobre el estado del programa.
Hacer explícita la asunción con una aserción.

- A menudo, determinado código sólo funciona bajo ciertos supuestos
- La raíz cuadrada, por ejemplo, si el número es positivo
- Muchas veces esas suposiciones se indican con comentarios
- Es mucho mejor escribir una aserción

```
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();
}
```

- Cuando una aserción falla se produce una excepción "unchecked"
- Indican errores de programación (Que deben ser corregidos, no capturados)
- De hecho, normalmente las aserciones se desactivan en el código de producción

Doble propósito:
- Comunican el propósito del código
- Depuración: permiten detectar los errores más cerca de su origen

Tampoco hay que caer en el exceso de tratar de comprobarlo todo
- Únicamente aquello que, de no cumplirse, el código no funcionaría

- Y hay que huir de las comprobaciones redundantes
- Emplear para ello, si es preciso, el *Extract Method*

MAKING METHOD CALLS SIMPLER

(273) Rename Method

El nombre de un método no revela su propósito.
Cambiar el nombre del método.

- Código autodocumentado
- Muchas veces no daremos con el nombre correcto la primera vez
- ¡No caigamos en la tentación de dejarlo como esté!
- Nos damos cuenta de su importancia cuando manejamos código escrito por otros
- También puede ayudar reorganizar los parámetros del método:

Add Parameter
Remove Parameter

(275) Add Parameter

Un método necesita que su llamador le proporcione más información.
Añadir un nuevo parámetro con esa información.

- Mejor evitar las largas listas de parámetros
- Hay que tener en cuenta las alternativas:
- ¿Puede el método pedir a otros objetos la información?
- Si no, ¿tendría sentido añadirles un método para ello?
- ¿Debería estar este comportamiento en otro objeto (el que tenga la información necesaria)?
- También se podría considerar (295) *Introduce Parameter Object*

```
getContact()

getContact(Date)
```

(277) Remove Parameter

A parameter is no longer used by the method body.
Remove it.

(279) Separate Query from Modifier

Un método devuelve un valor pero también cambia el estado del objeto.
Crear dos métodos, uno para la consulta y otro para la modificación.

- Idealmente, un método que devuelve un valor no debería tener efectos laterales observables
- No hay nada malo, por ejemplo, en guardar el valor en una variable del objeto que haga las veces de caché

(283) Parameterize Method

Varios métodos hacen cosas similares pero con diferentes valores en el cuerpo del método.
Crear un método que use un parámetro para los distintos valores.

```
class Employee {
 fivePercentRaise()
  {}
 tenPercentRaise()
 {}
```

```
class Employee {
 raise(percentage)
  {}
```

(285) Replace Parameter with Explicit Methods

Un método ejecuta distinto código dependiendo del valor de un parámetro enumerado.
Crear un método separado para cada valor del parámetro.

- Es el reverso del anterior
- No obstante, está más pensado para aquéllos casos en que, en función del parámetro, se ejecutan distintas ramas de una sentencia condicional
- Hace la interfaz más explícita
- El programador no tiene que preocuparse por el valor correcto del parámetro
- switch.turnOn() es más claro que switch.setOn(true)

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

```
void setHeight(int arg)
{
  height = arg;
}
void setWidth(int arg)
{
  width = arg;
}
```

(285) Replace Parameter with Explicit Methods

You have a method that runs different code depending on the values of an enumerated parameter.
Create a separate method for each value of the parameter.

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

```
void setHeight(int arg) {
```

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

(288) Preserve Whole Object

You are getting several values from an object and passing these values as parameters in a method call.
Send the whole object instead.

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


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

(292) Replace Parameter with Method

An object invokes a method, then passes the result as a parameter for a method.
The receiver can also invoke this method.
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);
```

(295) Introduce Parameter Object

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

Dealing with a chain of calls

Ralph Johnson pointed out to me that a common case isn't clear in the Refactoring book. This case is when you have a bunch of methods that call each other, all of which have a clump of parameters that need this refactoring. In this case you don't want to apply Introduce Parameter Object because it would lead to lots of new objects when you only want to have one object that's passed around.
The approach to use is to start with the head of the call chain and apply Introduce Parameter Object there. Then apply Preserve Whole Object to the other methods.

(300) Remove Setting Method

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

Using Final

Paul Haahr kindly pointed out to me that I've not been accurate with the use of `final` in this refactoring. The point that I missed is that you can't use a final field if the initialization is done in a separate method to the constructor. This the example at the bottom of page 301 (that uses `initialize(id)`) just does not compile. (My apologies for not testing that fragment, some do slip through the net....) If you want to use a separate method, you can't make the field final
This also means that the mechanics is incorrect. Making the field final should be the last step not the first, and can only be done if it's possible.

(303) Hide Method

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

(304) Replace Constructor with Factory Method

You want to do more than simple construction when you create an object.
Replace the constructor with a factory method.

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

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

(308) Encapsulate Downcast

A method returns an object that needs to be downcasted by its callers.
Move the downcast to within the method.

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

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

(310) Replace Error Code with Exception

A method returns a special code to indicate an error.
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;
}
```

(315) Replace Exception with Test

You are throwing an exception on a condition the caller could have checked first.
Change the caller to make the test first.

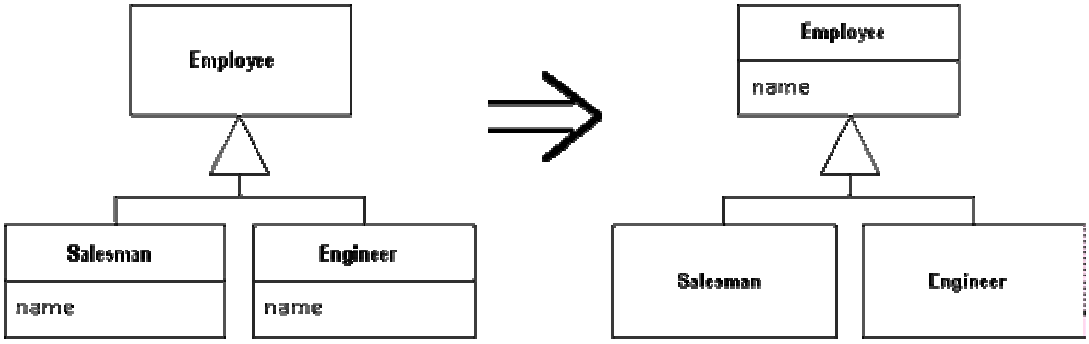
```
double getValueForPeriod (int periodNumber) {
        try {
                return _values[periodNumber];
        } catch (ArrayIndexOutOfBoundsException e) {
                return 0;
        }
}
```

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

DEALING WITH GENERALIZATION

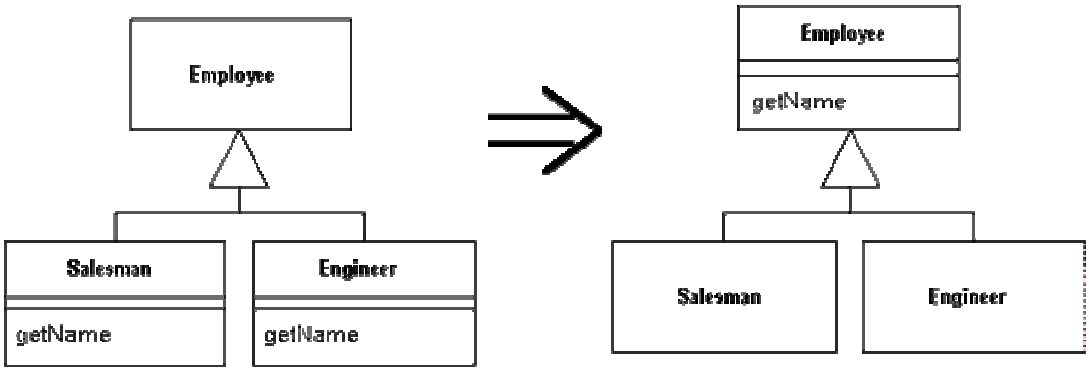
(320) Pull Up Field

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



(322) Pull Up Method

You have methods with identical results on subclasses.
Move them to the superclass.



(325) Pull Up Constructor Body

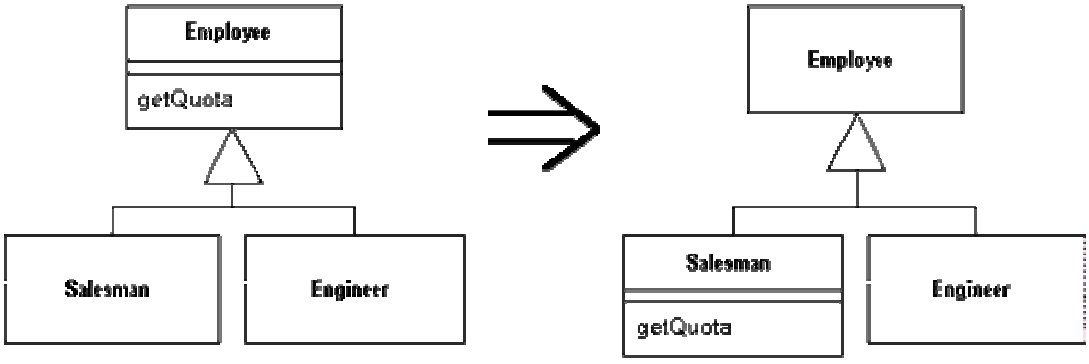
You have constructors on subclasses with mostly identical bodies.
Create a superclass constructor; call this from the subclass methods.

```
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;
        }
```

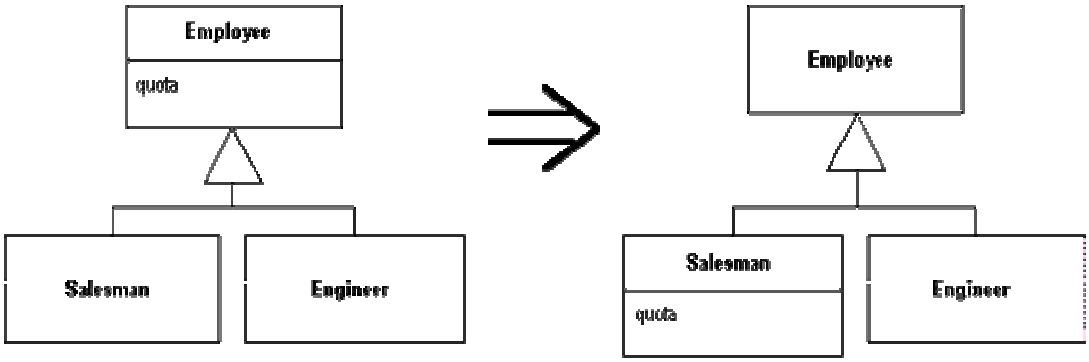
(328) Push Down Method

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



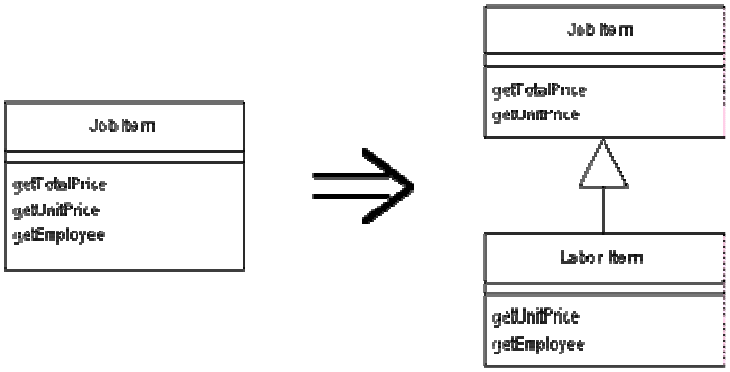
(329) Push Down Field

A field is used only by some subclasses.
Move the field to those subclasses.



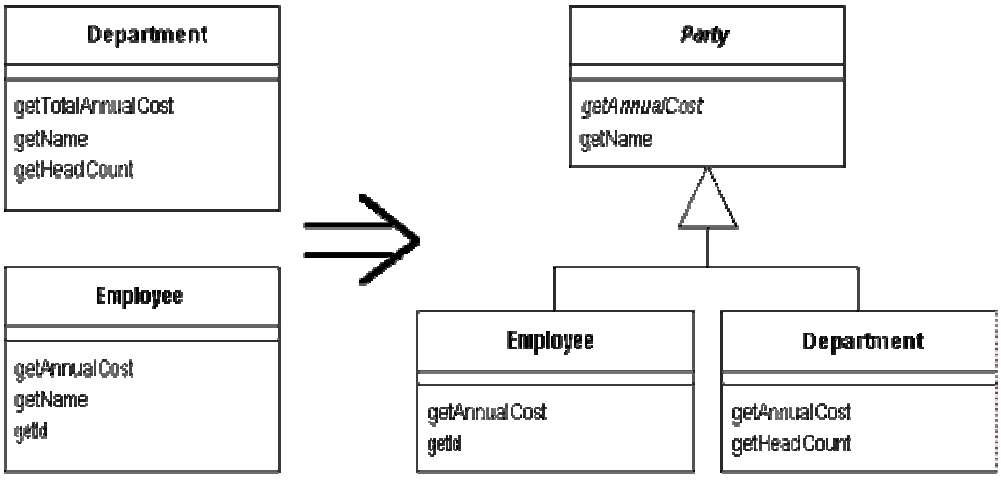
(330) Extract Subclass

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



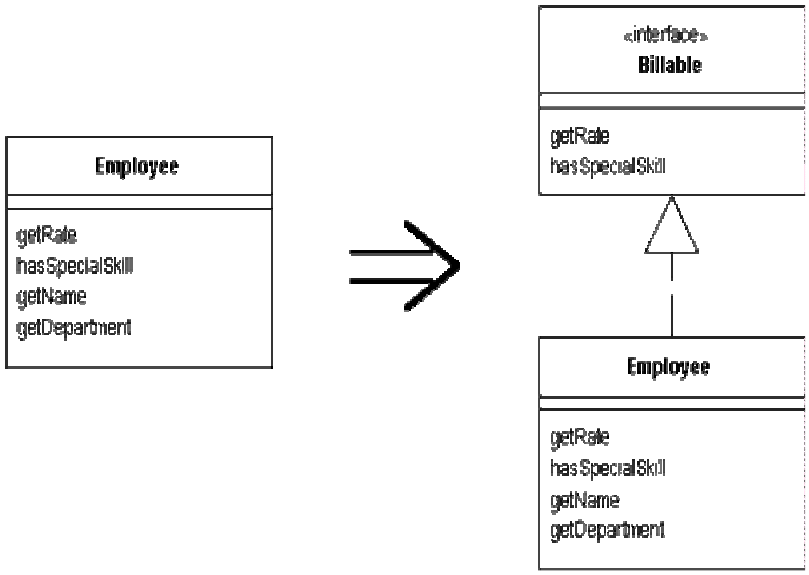
(336) Extract Superclass

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



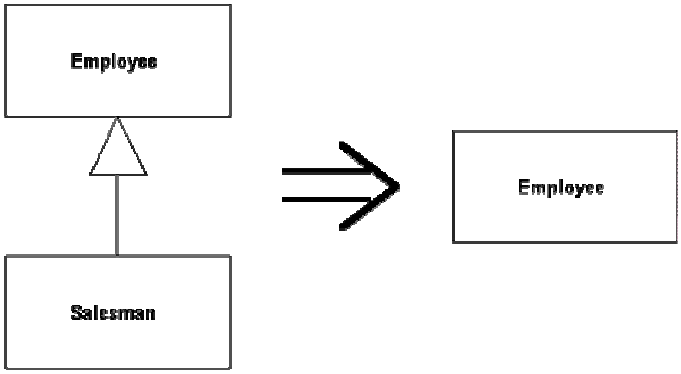
(341) Extract Interface

Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common.
Extract the subset into an interface.



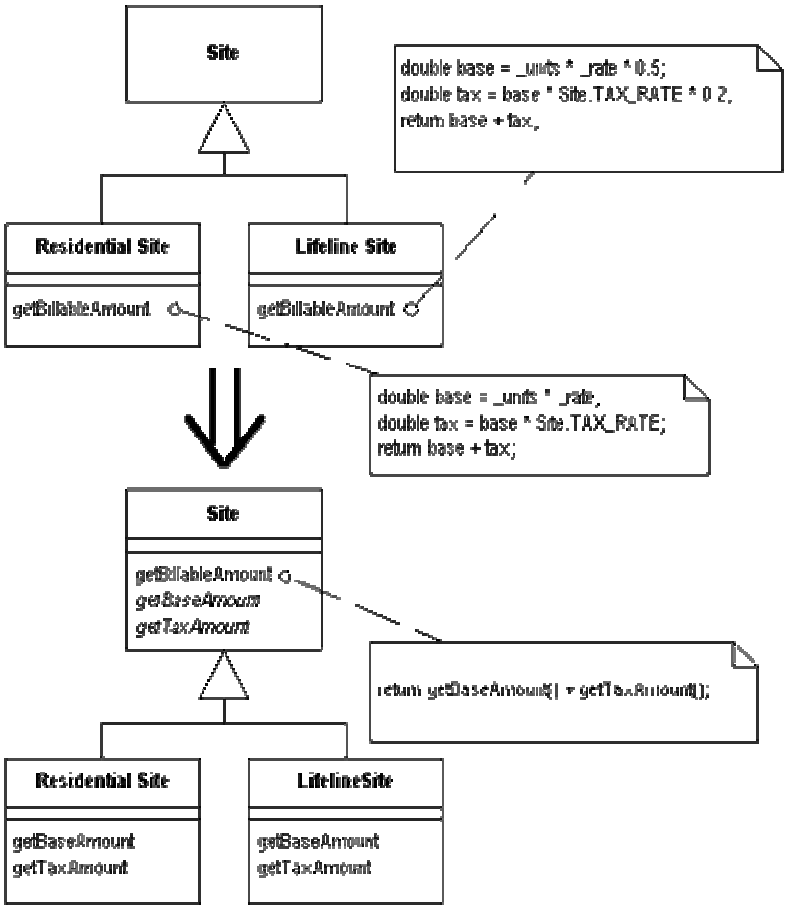
(344) Collapse Hierarchy

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



(345) Form Template Method

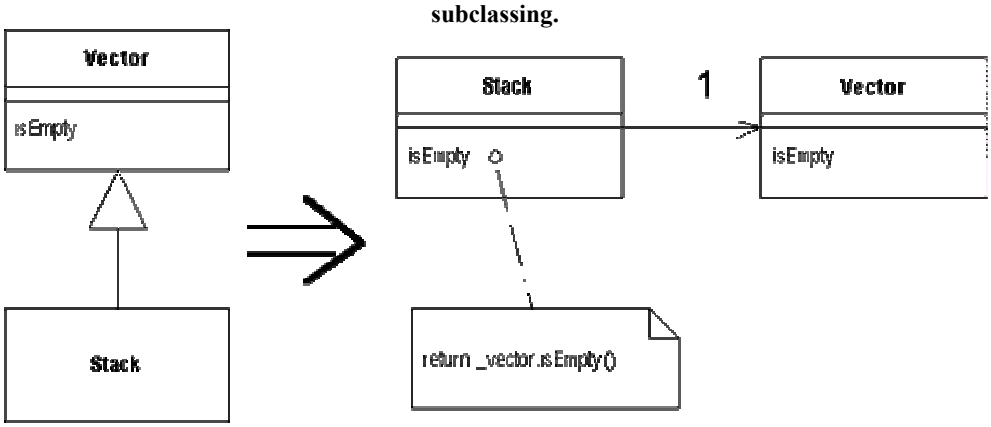
You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.
Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.



(352) Replace Inheritance with Delegation

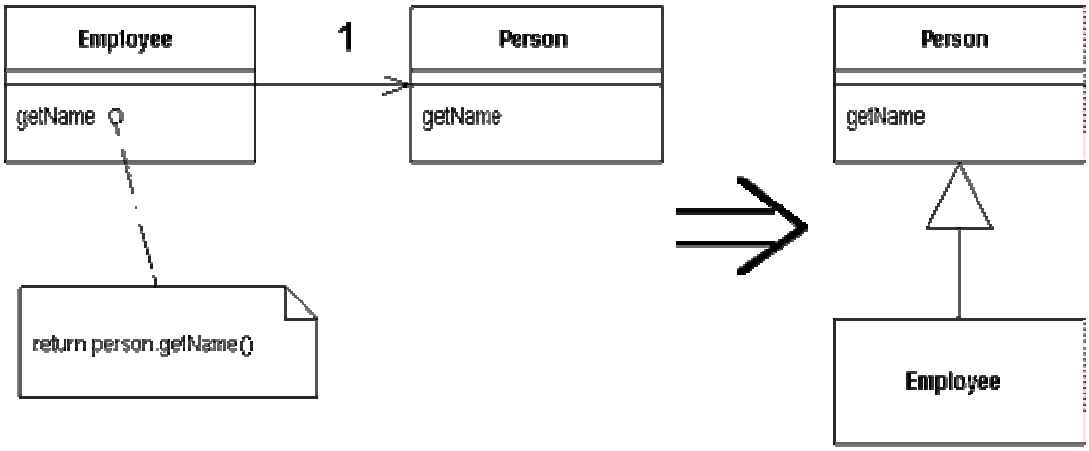
A subclass uses only part of a superclasses interface or does not want to inherit data.

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



(355) Replace Delegation with Inheritance

You're using delegation and are often writing many simple delegations for the entire interface.
Make the delegating class a subclass of the delegate.



Nuevos Refactoring

- Replace Iteration with Recursion *by Dave Whipp*
- Replace Recursion with Iteration *by Ivan Mitrovic*
- Replace Static Variable with Parameter *by Marian Vittek*
- Reverse Conditional *by Bill Murphy and Martin Fowler*
- Split Loop *by Martin Fowler*

Ver: http://www.refactoring.com/

Herramientas (Java)

IntelliJ Idea
: This is a fully fledged IDE whose abilities go far beyond refactoring. I think they have succeeded in really moving forward the state of the art for IDEs. Suffice to say that when I use IntelliJ I almost don't miss Smalltalk any more.

Eclipse
: An open source platform for IDE development - in many ways Eclipse is the Emacs for the 21st century. It ships with a built in Java IDE that includes a strong refactoring capability. The IBM commercial WSAD tool is built on Eclipse.

JFactor
: A plug-in tool - works with JBuilder and Visual Age. Instantiations is a well respected outfit with a long history in Smalltalk and Java VM and compiler technology.

XRefactory
: An emacs plug-in that offers code completion and other facilities as well as refactoring.

JBuilder
: Borland's primary tool offers some refactoring support, but isn't over the rubicon yet. Various plug in tools offer deeper support.

RefactorIt

"A plug in for NetBeans, Sun Java Studio (fka Forte), Eclipse, JDeveloper and JBuilder and also usable as a stand-alone tool. Besides refactorings includes smart code searches, metrics and audits (i.e. smell detectors) with corrective actions."

JRefactory

A plug-in for JBuilder, NetBeans, and Elixir IDEs. Also does UML diagrams.

Transmogrify

JafaRefactor

A plug in for jEdit

CodeGuide

"CodeGuide offers advanced refactoring capabilities. It can rename methods, fields, variables, labels, classes or packages and automatically update all references throughout the project. It can move classes and packages and update all references. CodeGuide will check for potential problems prior to refactoring to make the refactorings safe. There are a couple of intelligent coding tools that will help you perform common tasks such as Javadoc stub insertion, setter & getter creation and import organization."

Referencias Clásicas

- Fowler M. "Refactoring: Improving the Design of Existing Code". Addison Wesley, 1999.
- Opdyke, W. "Refactoring Object-Oriented Framework" Ph.D Thesis. University of Illionois at Urbana-Champaign 1993.
- Refactoring Home Page http://www.refactoring.com/
- Otra: http://www.refactoring.be/thumbnails.html

Refactoring de otros lenguajes

- Garrido A. y Johnson R. "Challenges of Refactoring C Programas"
- T. Schrijvers, and A. Serebrenik, *Improving Prolog programs: refactoring for Prolog*, Logic Programming, 20th International Conference, ICLP 2004, Proceedings (Demoen, B. and Lifschitz, V., eds.), vol 3132, Lecture Notes in Computer Science, pp. 58-72, 2004
- UL HUIQING Li . "Refactoring Haskell Programs" – PhD thesis The University of Kent - 2006
- Leitdo, A.M. "A formal pattern language for refactoring of Lisp programs**"** Software Maintenance and engineering, 2002. Proceedings. Sixth European Conference 2002 Pag:186 – 192
- Extreme Programming in Perl – R. Nagler, 2005, http://www.extremeperl.org/bk/home

Refactoring POA

- D. Binkley, M. Ceccato, M. Harman, F. Ricca, and P. Tonella. Automated refactoring of object oriented code into aspects. In 21st IEEE International Conference on Software Maintenance (ICSM), 2005.
- Van Deursen, M. Marin, and L. Moonen. A Systematic Aspect-Oriented Refactoring and Testing Strategy, and its Application to JHotDraw. Technical Report SEN-R0507, CWI, Amsterdam, 2005.
- S. Hanenberg, C. Oberschulte, and R. Unland. Refactoring of aspect-oriented software. In Proc. of Net.ObjectDays Conference, pages 19–35. Springer-Verlag, 2003.
- J. Hannemann, T. Fritz, and G. C. Murphy. Refactoring to aspects: an interactive approach. In Proc. of 2003 OOPSLA Workshop on Eclipse technology eXchange, pages 74–78. ACM Press, 2003.
- Jan Hannemann, Gail Murphy, and Gregor Kiczales. Role-based refactoring of crosscutting concerns. In Peri Tarr, editor, Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005), pages 135–146. ACM Press, March 2005.
- Jan Hannemann Aspect-Oriented Refactoring: Classification and Challenges AOSD 2006.
- M. Iwamoto and J. Zhao. Refactoring aspect-oriented programs. In Proc. of 4th AOSD Modeling With UML Workshop, UML'2003, San Francisco, California, USA, 2003.
- M. Marin, L. Moonen, and A. van Deursen. An approach to aspect refactoring based on crosscutting concern types. In MACS '05: Proceedings of the 2005 workshop on Modeling and analysis of concerns in software,

pages 1–5, New York, NY, USA, 2005. ACM Press.
- M.P. Monteiro. Catalogue of refactorings for AspectJ. Technical Report UM-DI-GECSD-200401, Universidade do Minho, 2004.
- M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect oriented refactorings. In Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD), pages 111–122. ACM Press, March 2005.
- Paolo Tonella, Mariano Ceccato, "Refactoring the AspectizableInterfaces: An Empirical Assessment"IEEE Transactions on Software Engineering, VOL 31, No.10,Oct 2005
- Wake, W., Refactoring Workbook, Addison Wesley, 2004.
- Wloka, Jan. Refactoring in the Presence of Aspects. 13th Workshop for PhD Students in Object-Oriented Systems (PhDOOS), at ECOOP '03, Darmstadt, Germany, 2003.