# Tecnología de Programación

*Martín L. Larrea*

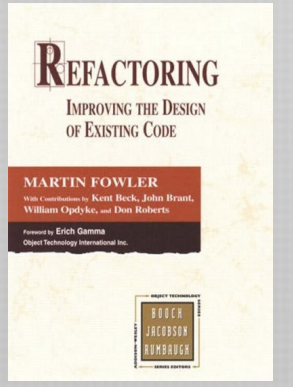
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

# Refactorización

La refactorización (*refactoring*) es el proceso de cambiar un sistema de software *disciplinadamente* de forma tal que no altera el comportamiento externo, mejorando su estructura interna.

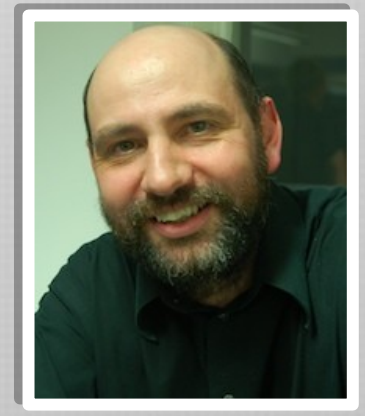
Refactorizar es mejorar el diseño del código una vez escrito.

Lo correcto: buen diseño primero, la codificación segundo.
*Sin embargo, sucesivas modificaciones degradan usualmente la calidad del código de nuestro sistema*



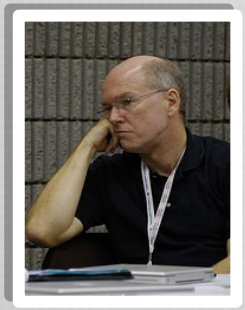

Referencia clásica:

*Refactoring. Improving the design of existing code.*
Martin Fowler.



# Refactorización

Martin Fowler no fue el primero...



## William Opdyke

Autor de la tesis doctoral
*Refactoring Object-Oriented Frameworks.*

## Kent Beck

Trabajó durante mucho tiempo en Smalltalk, de donde surge la noción de refactorización.
Uno de los precursores de los patrones de diseño.
Creador de *Extreme Programming*.

## Ward Cunningham

También experto en Smalltalk y patrones de diseño.
Coautor de The Wiki Way.

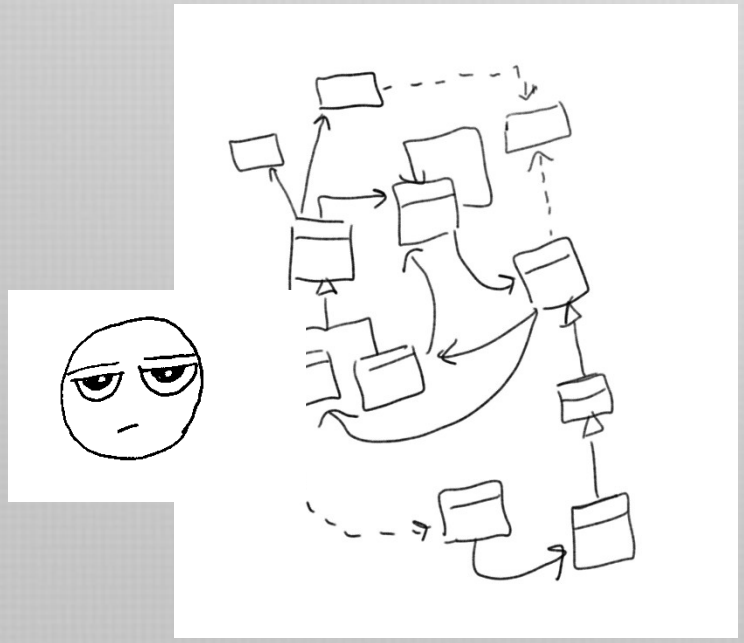
# Refactorización – para qué

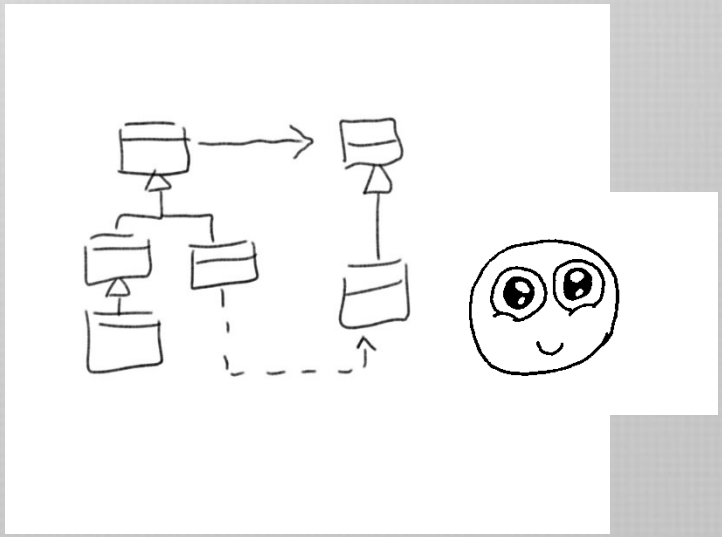
## ¿Para qué refactorizar?

### Mejora el diseño del software

Sin refactorización, el diseño del programa decae en calidad.

Los cambios “sobre la marcha” hacen que el código pierda estructura.





# Refactorización – para qué

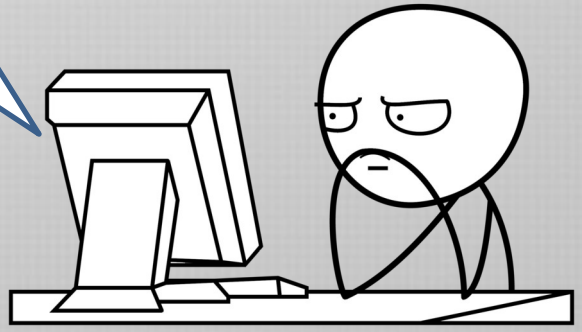
## ¿Para qué refactorizar?

Hace el software fácil de comprender

En todo proyecto de software, eventualmente alguien leerá el código en algún momento.

*Usualmente no se piensa en esta persona al programar*.

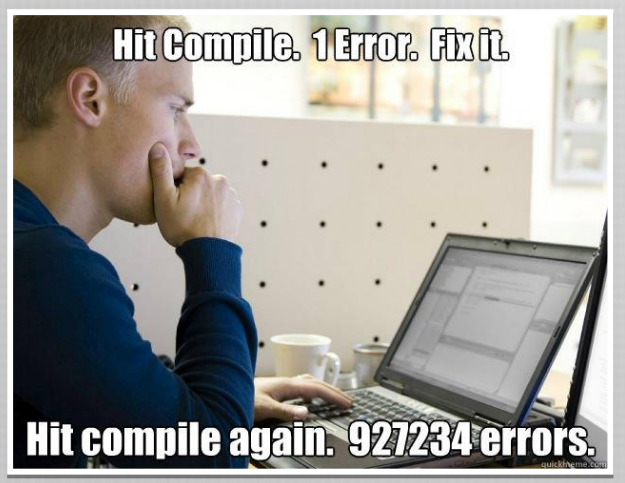
```
public static int doSomethingToAnInt(int n) {
    boolean doIt = (((n & (n-1)) == 0)?false:true);
    while ((n & (n-1)) != 0) n = (n & (n-1));
    return (doIt?n<<1:n);
}
```



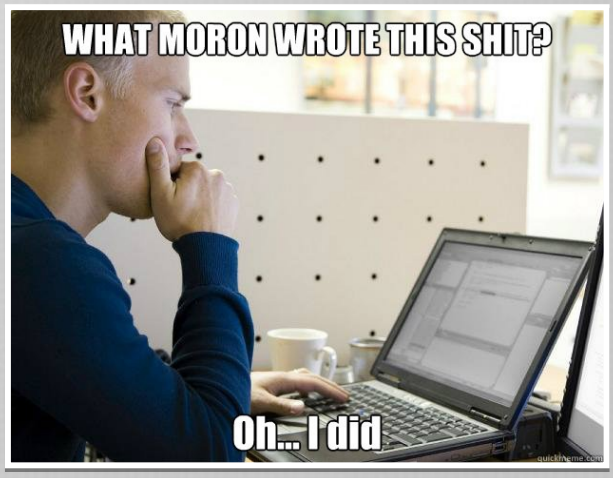
# Refactorización – para qué

## ¿Para qué refactorizar?

Ayuda a encontrar errores
Refactorizar es comprender el código escrito y
por lo tanto poder distinguir su correctitud.







# Refactorización – para qué

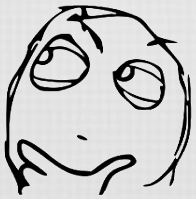
## ¿Para qué refactorizar?

Ayuda a programar más rápidamente

Contradicción? No.

Un buen diseño es un terreno firme sobre el cual poder avanzar.

# Refactorización – cuándo



## ¿Cuándo refactorizar?

### Regla de tres

*"La tercera vez que hacemos algo similar, refactorizar"*

### Refactorizar cuando agregamos una función

Un buen momento es cuando se agregan nuevas funciones o responsabilidades a una parte del software.

### Refactorizar cuando es necesario reparar un bug

Un error es una señal de posible refactorización pendiente.

### Refactorizar cuando se revisa el código

La revisión de código debe ser periódica y con grupos pequeños, facilitando la distribución del conocimiento del producto en desarrollo.

# Refactorización - indicios

Es necesario observar los síntomas que pueden ser indicio de problemas en el código del software.

Esto se suele denominar "code smell" y guardan cierta relación con los antipatrones.

Kent Beck (probablemente el autor del término) y Martin Fowler catalogan una serie de "*bad smells in code*".

Son posibles señales de la necesidad de una refactorización.

Como es razonable, no existe una métrica que obligue a la refactorización,
por lo que son simplemente señales de alerta a tener en cuenta.

# Bad smells in code

## Código Duplicado

Según Fowler, "*number one in the stink parade*".

Ejemplos:

La misma expresión en dos métodos de la misma clase.
*Solución: Extract Method*

La misma expresión en dos clases hermanas.
*Solución: Extract Method, Pull Up Method, o Substitute Algorithm*

Código duplicado en clases diferentes
*Solución: Extract Class*

./run.sh cpd --minimum-tokens 100 --files /home/martinlarrea/Downloads/PMD/TDP-Proyecto-master/

# Bad smells in code

## Método largo/extenso

Lo ideal es producir métodos cortos y específicos.
Las razones son obvias: *fácil de mantener / comprender / reemplazar / reusar*

La solución es descomponer el método en submétodos.
Algunas refactorizaciones utiles:
*Extract Method, Introduce Parameter Object, Replace Method with Method Object, Decompose Conditional*

## Clase larga/extensa

También identificado a veces como el antipatrón *Blob*

La solución es disminuir la complejidad y las responsabilidades
Algunas refactorizaciones útiles:
*Extract Class, Extract Subclass*

# Bad smells in code

## Lista de parámetros larga/extensa

En los inicios de la programación, todo lo necesario para una rutina
era pasado como parámetro (cada uno!)

En orientación a objetos esto no es completamente cierto, ya que la rutina puede comunicarse con un objeto que le provea lo necesario.

Las listas de parámetros extensas pueden simplificarse con algunas refactorizaciones clásicas:

*Replace Parameter with Method,*
*Preserve Whole Object,*
*Introduce Parameter Object*

# Bad smells in code

## Cambio divergente

El cambio divergente ocurre cuando una clase es cambiada de forma diferente por diferentes razones.

*"Debo cambiar estos tres métodos cada vez que usamos una nueva base de datos; debo cambiar estos cuatro métodos cada vez que tenemos una nueva herramienta financiera"*

Solución posible: *Extract Class.*

## Shotgun Surgery

Similar al anterior, pero ocurre cuando al hacer un cambio, se requieren muchos pequeños cambios en clases diferentes.

Solución posible: *Move Method, Move Field.*

# Bad smells in code

## Data clumps

Son agrupaciones de datos que son usadas en conjunto en muchos lugares.
Deberían probablemente ser parte de un objeto.

Soluciones:
*Extract Class, Introduce Parameter Object, Preserve Whole Object*

## Primitive Obsession

Muchos datos construidos en base a tipos primitivos, en lugar de encapsularlos en pequeños objetos.

Típicamente: arreglos, strings, enteros.

Soluciones:
*Replace Data Value With Object, Replace Type Code With Class, Replace Type Code with State/Strategy*
*Los mismos de Data Clumps.*

# Bad smells in code

## Lazy Class

Una clase que no se justifica, redundante, o de poca utilidad.
Entorpece la comprensión del sistema.
*A veces es producto de refactorizaciones anteriores*

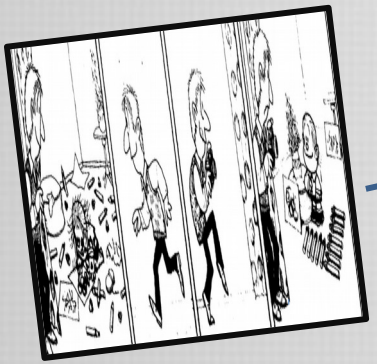
Soluciones: *Collapse Hierarchy*

## Inappropriate Intimacy

Clases demasiado íntimas, que constantemente indagan en aspectos privados de ambas.

Soluciones “puritanas” :)
*Move Method, Move Field,*
*Change Bidirectional Association to Unidirectional, Hide Delegate*

# Refactorizaciones comunes



Add Parameter
Change Association
Reference to value
Value to reference
Collapse hierarchy
Consolidate conditionals
Procedures to objects
Decompose conditional
Encapsulate collection
Encapsulate downcast
Encapsulate field

Extract Interface
Extract method
Extract subclass
Extract superclass
Form template method
Hide delegate
Hide method
Inline class
Inline temp
Introduce assertion
Introduce explain variable
Introduce foreign method

*Refactoring. Improving the design of existing code.* Martin Fowler

# Composing methods - Extract Method

Es uno de las principales refactorizaciones que apuntan a empaquetar código apropiadamente. Básicamente, es un fragmento de código que puede ser agrupado y apartado.

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

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

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

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

# Composing methods - Extract Method

Pasos:

1. Crear un nuevo método, nombrarlo apropiadamente según lo que hace.
2. Copiar el código extraído del origen a este nuevo método
3. Escanear el código extraído por referencias a variables locales al origen.
4. Escanear el código extraído por variables temporales.
5. Evaluar el uso de las variables (son modificadas? son solo de lectura?)
6. Pasar al nuevo método las variables necesarias como parámetros.
7. Compilar este nuevo método.
8. Reemplazar el código extraído por una llamada al nuevo método.

# Composing methods - Inline Method

Ocurre cuando el cuerpo de un método es tan claro como su nombre.

La solución es eliminar el método trasladando el cuerpo a sus invocadores.

```
int getRating() {
            return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() {
            return _numberOfLateDeliveries > 5;
}
```

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

# Composing methods - Replace Temp with Query

Se usa una variable temporal para guardar el resultado de una expresión.

*El problema es que son temporales y locales y (a veces) tienden a generar métodos extensos*

La solución es convertir la expresión en un método y reemplazar su uso por la invocación correspondiente

```
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;
}
```

# Composing methods - Introduce explaining variable

Existe una expresión muy complicada.

La solución es usar variables temporales con nombres significativos para simplificar la expresión.

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

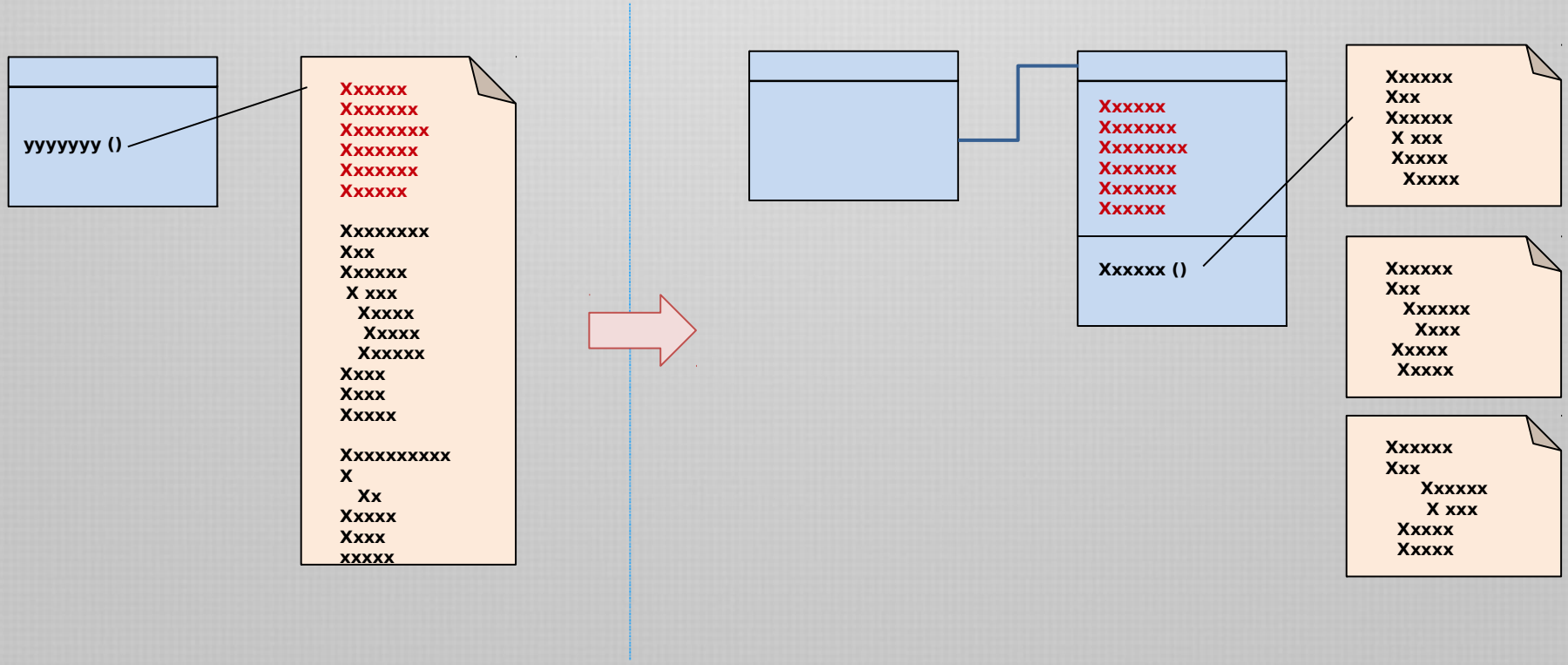
```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  > -1;
final boolean wasResized = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized)
{
     ...
}
```

# Composing methods - Replace Method with Method Object

Existe un método extenso que usa variables locales de forma tal que no se puede aplicar Extract Method.

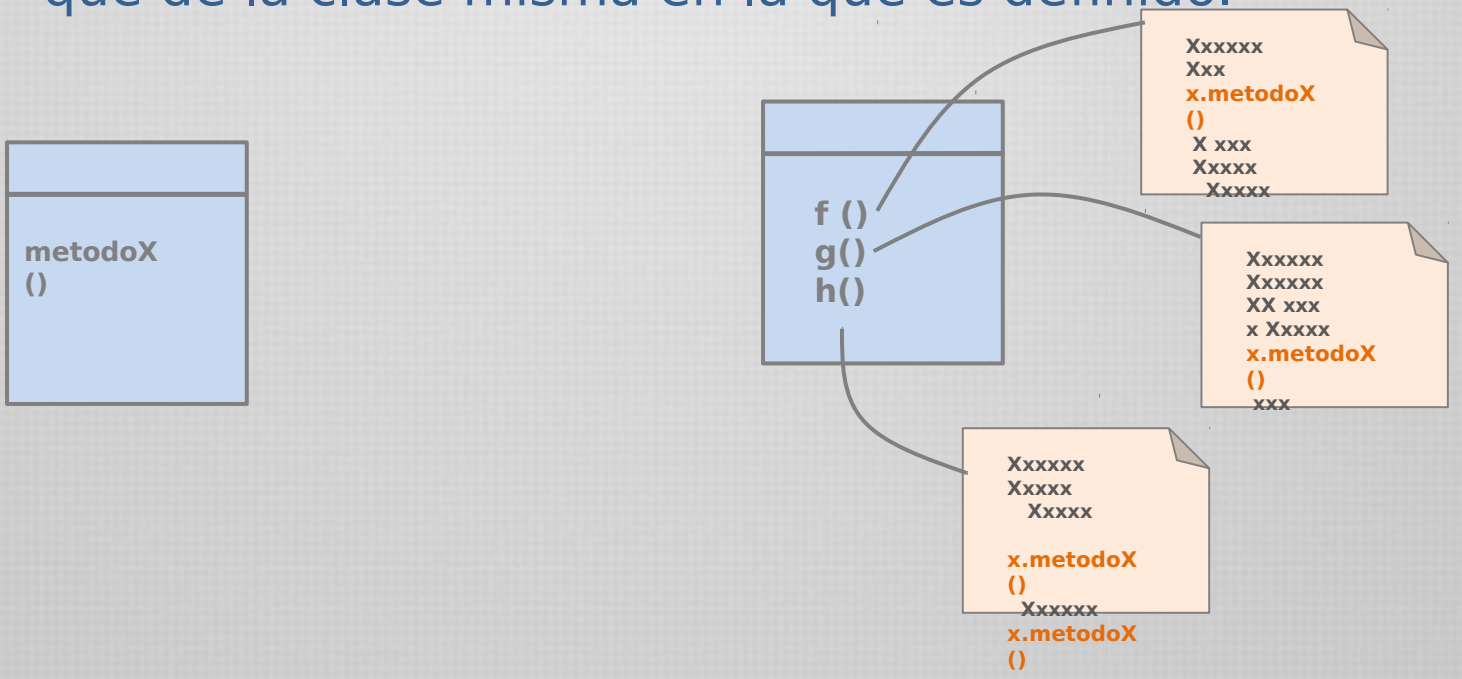
La solución es convertir el método en un objeto de manera tal que las variables locales son ahora campos del objeto.

*Luego se puede descomponer el método en otros métodos dentro de la misma clase*



# Moving Features – Moving Method

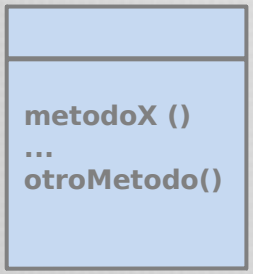
Existe un método que es o será usado por más aspectos de otra clase
que de la clase misma en la que es definido.

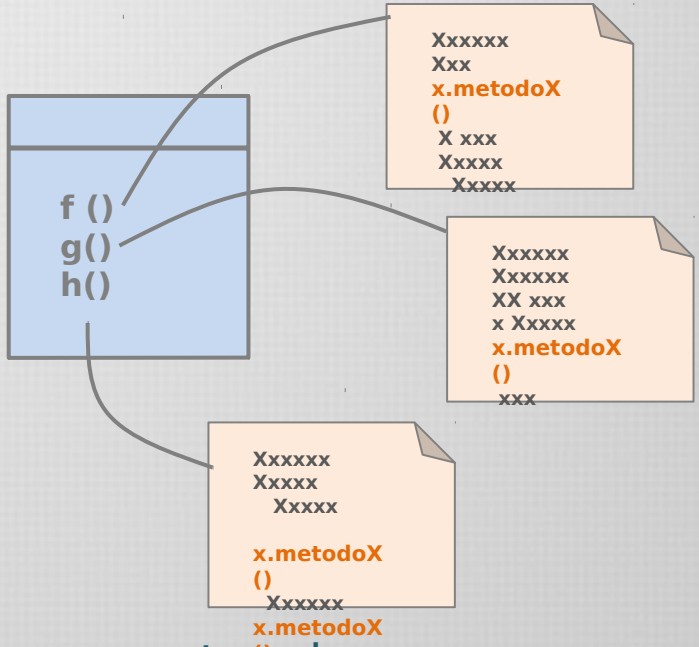



Puede ocurrir cuando una clase tiene muchas responsabilidades, o cuando las clases colaboran demasiado y existe mucho acoplamiento.

La solución es crear un nuevo método con un cuerpo similar en la clase
que más lo usa. El viejo método se elimina o delega la tarea al

# Moving Features – Moving Method




## Mecánica del proceso

1. Examinar todo lo que debe moverse a otra clase.
2. Controlar las dependencias con las subclases y las superclases
3. Declarar el nuevo método en la clase destino.
4. Copiar el código del origen al destino y hacer los ajustes necesarios.
5. Compilar la clase destino.
6. Determinar el vínculo de la clase origen a la clase destino.
7. Convertir el código origen en un delegando
8. Compilar y testear

# Moving Features – Extract Class

Hay una clase que está haciendo una tarea que deberían hacerla dos.

Demasiados métodos, muchos datos, exceso de responsabilidades.

La solución es crear una nueva clase y trasladar los campos y métodos
relevantes de la clase vieja a la clase nueva.

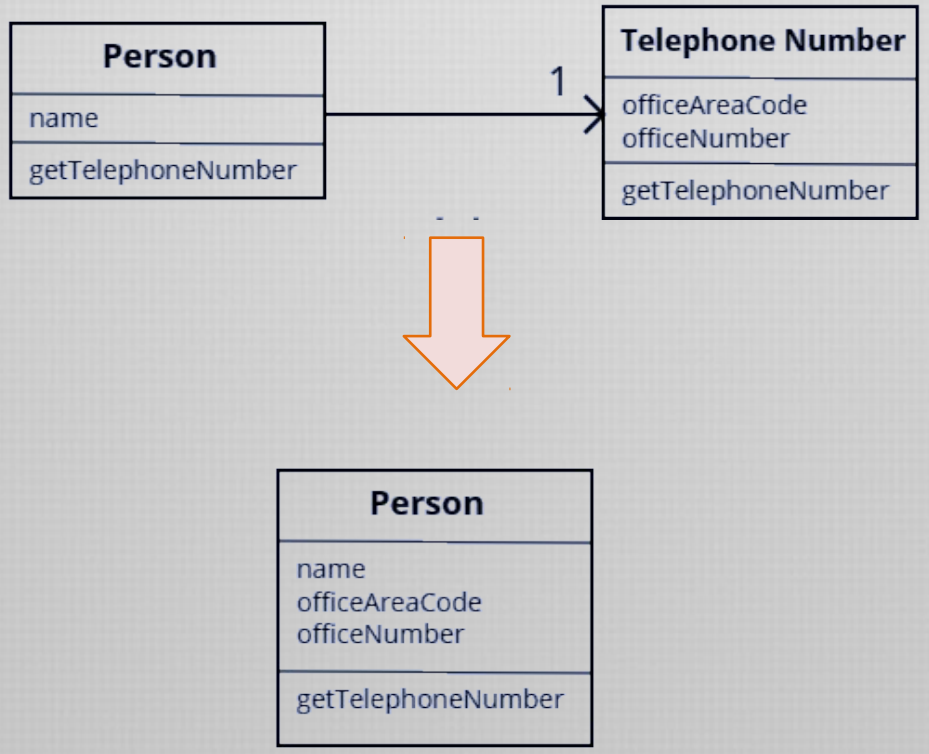
## Mecánica del proceso

1. Decidir cómo dividir las responsabilidades de la clase
2. Crear una nueva clase que exprese esa división
   *(tal vez deban renombrarse ambas)*
3. Hacer un link desde la clase vieja a la nueva.
   *(tal vez una asociación bidireccional)*
4. Usar *Move Field* y *Move Method*, y compilar luego de aplicar esas refactorizaciones.
5. Revisar las interfaces y reducirlas en lo necesario.
6. Examinar la visibilidad de la nueva clase en la clase vieja.

# Moving Features – Extract Class

La inversa de *Extract Class* es *Inline Class*.

En esta refactorización una clase absorve a otra pues esta última no tiene muchas responsabilidades asociadas.

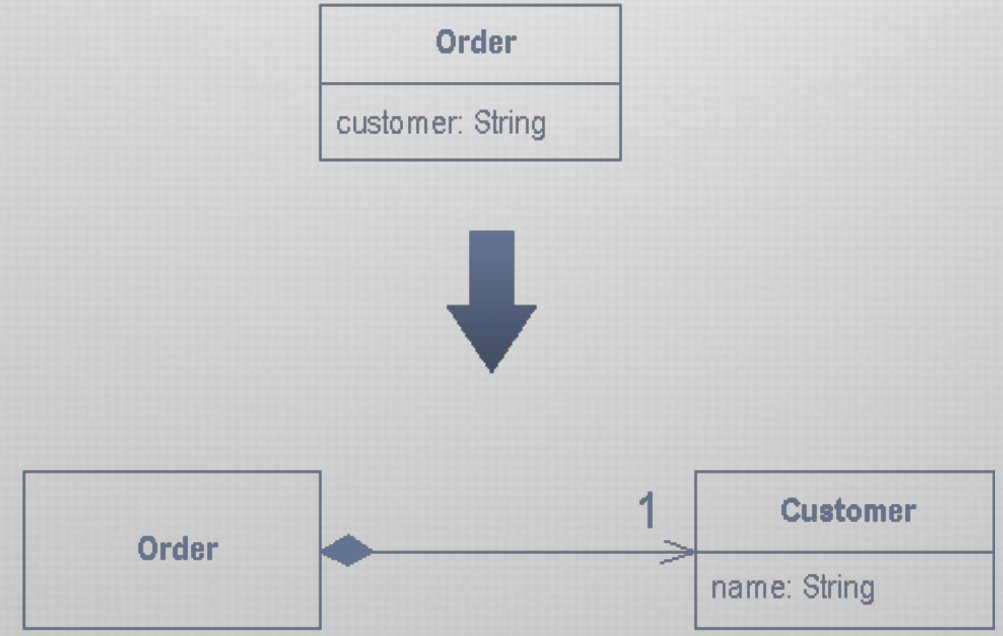


# Organizing data – Replace data value with object

## Existe un item de datos que necesita datos o comportamiento adicional

Es básicamente un conjunto de datos que en su momento no fue considerado como objeto del sistema.

La solución es convertir ese item de datos en un objeto.

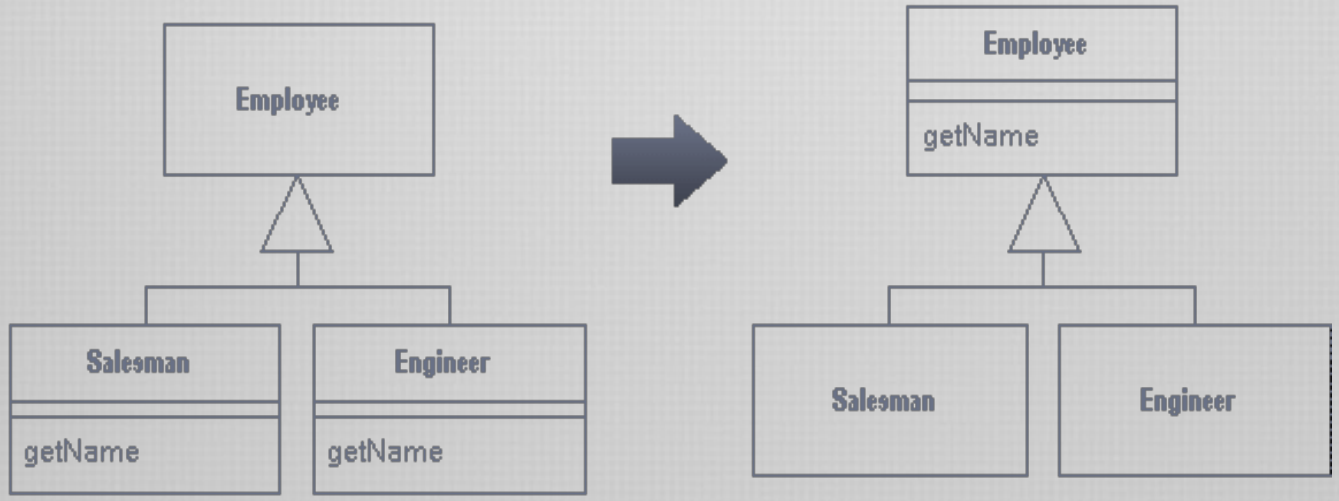


# Generalización – Pull Up Method

Existen métodos con idénticos resultados en varias subclases.

Usualmente son consecuencia de *copy+paste*.
También surge al refinar abstracciones incompletas.

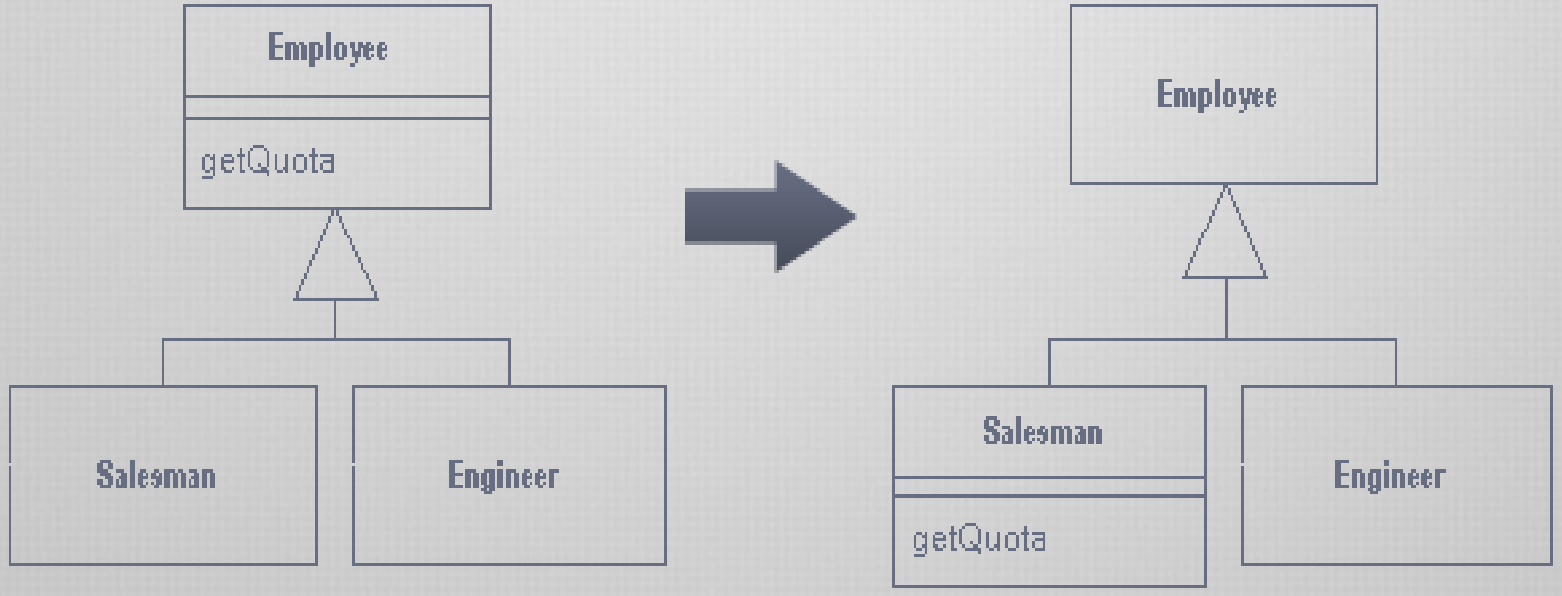
La solución es trasladar los métodos a las superclases.



# Generalización – Push down Method

Parte del comportamiento de una clase es relevante sólo a algunas subclases.

La solución es trasladar los métodos a las subclases.

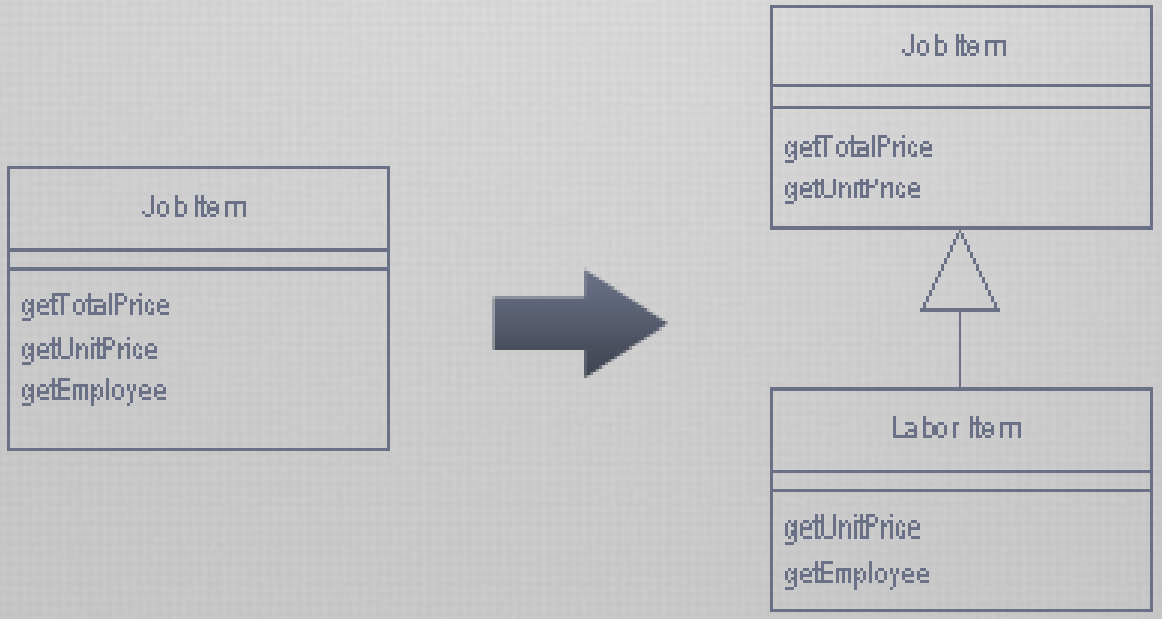


# Generalización – Extract Subclass

Una clase tiene aspectos que son usados sólo en algunas instancias.

Puede ocurrir como una sobrecarga de responsabilidades, o una dependencia sobre los atributos.
Algunas instancias los usan, otras no

La solución es crear subclases para los subconjuntos de aspectos relevantes a cierta instancias.



# Generalización – Extract Subclass

## Mecánica del proceso

1. Definir una nueva subclase de la clase origen.
2. Proveer constructores para la nueva subclase.
   *Utilizar super con los argumentos adecuados*
3. Buscar todas las invocaciones a constructores de la superclase y reemplazarlo por el constructor de la subclase si es necesario.
   *Si la superclase ya no puede ser instanciada, declararla abstracta.*
4. Aplicar *Push Down Method* y *Push Down Field* hasta que ya no sea necesario.
5. Eliminar campos que distinguían entre las instancias (usualmente booleans)
6. Compilar y testear luego de cada push down.

Visitar

www.refactoring.com

# Panel de Graduados DCIC 2018

**Fecha**: Viernes 28 de Septiembre
**Hora**: 16:00hs
**Lugar**: Sala de conferencia del DCIC.

# Panel de Graduados DCIC 2018

Ellos hace poco estuvieron en tu lugar y hoy triunfan en el mundo de la computación. Vení, escuchalos hablar de su experiencia y preguntales lo que quieras.

## Christian Junca

Graduado de la carrera Ingeniería en Sistemas de Computación. Entre 2012 y 2016 se desempeñó como desarrollador freelance. Luego de recibido comenzó a trabajar en Global Technology en J.P. Morgan Chase & Co., una firma líder en banca de inversión, servicios financieros para consumidores y pequeñas empresas, banca comercial, procesamiento de transacciones financieras y gestión de activos.

## Juan Francesconi

Graduado de la carrera Ingeniería en Sistemas de Computación. Posee una Maestría en Ingeniería del DIEC-UNS. Fue becario de investigación y desarrollo en el IIIE-CONICET, trabajando en el campo de la microelectrónica. Posteriormente se desempeñó como ingeniero de investigación y desarrollo en INTI CMNB, trabajando en diversos proyectos de base tecnológica. Actualmente trabaja en Globant como analista de negocios, y además es consultor independiente de proyectos TIC, dirigiendo en este momento un proyecto de Agrotech denominado Agro Check.

## Gerardo Simari

Graduado de las carreras de Licenciatura y Magíster en Ciencias de la Computación de la UNS, y luego del programa de Ph.D. en University of Maryland College Park (EE.UU.). Entre 2011 y 2014 trabajó en University of Oxford, Reino Unido, realizando trabajos de investigación y desarrollo en inteligencia artificial y bases de datos. Desde 2014 se desempeña como investigador en CONICET con lugar de trabajo en el Instituto de Ciencias e Ingeniería de la Computación (ICIC UNS-CONICET), y desde 2015 como profesor en el DCIC. Dirige el grupo de Investigación en Ingeniería Cognitiva y Bases de Datos, dentro del LIDIA.

## María Clara Vallés

Graduada de la carrera Licenciatura en Ciencias de la Computación. Desde 2014 es Senior .NET Developer en BairesDev, empresa Argentina creada en 2009. Entre 2011 y 2014 trabajó en Globant formando parte de varios equipos de trabajo nacionales y con sede en Colombia para proyectos gubernamentales y comerciales. Anteriormente, se desempeñó como analista funcional en Huddle Group.

**FECHA:** Viernes 28 de Septiembre
**HORA:** 16 hs
**LUGAR:** Sala de Conferencias, Edificio del DCIC (Campus Palihue)
**INSCRIPCIÓN:** http://tinyurl.com/paneldcic2018
**(Cupo limitado)**

Nuevos panelistas, nuevas experiencias y nuevas historias.

Aprovecha el “Coffee Break” para acercarte a los panelistas.

Inscribite completando el formulario que se encuentra en el siguiente enlace:

https://tinyurl.com/paneldcic2018