

REFACTORIZACIÓN

El término *refactorización* (refactoring) se atribuye a Opdyke, quien lo introdujo por primera vez en 1992. Una refactorización es una transformación parametrizada de un programa preservando su comportamiento, que automáticamente modifica el diseño de la aplicación y el código fuente subyacente. Solo los cambios realizados en el software para hacerlo más fácil de modificar y comprender son refactorizaciones, por lo que no es una optimización del código, ya que esto en ocasiones lo hace menos comprensible, ni tampoco el solucionar errores o mejorar algoritmos. Típicamente, una refactorización es una transformación simple que tiene un fácil pero no trivial impacto en el código fuente de una aplicación.

Por lo tanto, *refactorizar un software* es modificar su estructura interna con el objeto de que sea más fácil de entender y de modificar a futuro, tal que el comportamiento observable del software al ejecutarse no se vea afectado. (Fowler)

Uno de los pilares de cualquiera de las prácticas que forman parte de la técnica es no modificar el comportamiento externo de la aplicación, para lo que en muchas ocasiones se hace uso de las pruebas unitarias. La esencia de esta técnica consiste en aplicar una serie de pequeños cambios en el código manteniendo su comportamiento. Cada uno de estos cambios debe ser tan pequeño que pueda ser completamente controlado por nosotros sin temor a equivocaciones. Es el efecto acumulativo de todas estas modificaciones lo que hace de la refactorización una potente técnica. El objetivo final de refactorizar es mantener nuestro código sencillo y bien estructurado.

Las refactorizaciones pueden verse como una forma de mantenimiento preventivo cuyo objetivo es disminuir la complejidad del software en anticipación a los incrementos de complejidad que los cambios pudieran traer.

Por tal, ¿por qué es importante la refactorización? cuando se corrige un error o se añade una nueva función, el valor actual de un programa aumenta. Sin embargo, para que un programa siga teniendo valor, debe ajustarse a nuevas necesidades (mantenerse), que puede que no sepamos prever con antelación. La refactorización precisamente, facilita la adaptación del código a nuevas necesidades.

Ventajas

Existen muchas razones por las que deberíamos adoptar esta técnica:

- *Aumenta la calidad:* Refactorizar es un continuo proceso de reflexión sobre nuestro código que permite que aprendamos de nuestros desarrollos en un entorno en el que no hay mucho tiempo para mirar hacia atrás. Un código de calidad es un código sencillo y bien estructurado, que cualquiera pueda leer y entender.
- *Desarrollo eficiente:* Mantener un buen diseño y un código estructurado es sin duda la forma más eficiente de desarrollar. El esfuerzo que invertimos en evitar la duplicación de código y en simplificar el diseño se verá recompensado cuando tengamos que realizar modificaciones, tanto para corregir errores como para añadir nuevas funcionalidades.
- *Procurar un Diseño Evolutivo en lugar de gran Diseño Inicial:* En muchas ocasiones los requisitos al principio del proyecto no están suficientemente especificados y debemos abordar el diseño de una forma gradual. Cuando tenemos algunos requisitos claros y no cambiantes un buen análisis de los mismos puede originar un diseño y una buena implementación, pero cuando los requisitos van cambiando según avanza el proyecto, y se añaden nuevas funcionalidades según se le van ocurriendo a los stakeholders, un diseño inicial no es más que lo que eran los requisitos iniciales, algo generalmente anticuado. Refactorizar nos permitirá ir evolucionando el diseño según incluyamos nuevas funcionalidades, lo que implica muchas veces cambios importantes en la arquitectura, añadir cosas y quitar otras.
- *Facilita la comprensión del software:* La refactorización facilita la comprensión del código fuente, principalmente para los desarrolladores que no estuvieron involucrados desde el comienzo del desarrollo. El hecho que el código fuente sea complejo de leer reduce mucho la productividad ya que se necesita demasiado tiempo para analizarlo y comprenderlo. Invirtiendo algo de tiempo en refactorizarlo de manera tal que exprese de forma más clara cuáles son sus funciones, en otras palabras, que sea lo más autodocumentable posible, facilita su comprensión y mejora la productividad.
- *Ayuda a encontrar errores:* Cuando el código fuente es más fácil de comprender permite detectar condiciones propensas a fallos, o analizar supuestos desde los que se partió al inicio del desarrollo, que pueden no ser correctos. Mejora la robustez del código escrito.
- *Evitar la reescritura de código:* En la mayoría de los casos refactorizar es mejor que reescribir. No es fácil enfrentarse a un código que no conocemos y que no sigue los estándares que uno utiliza, pero eso no es una buena excusa para empezar de cero; sobre todo en un entorno donde el ahorro de costes y la existencia de sistemas lo hacen imposible.

- *Ayuda a programar más rápidamente:* La refactorización permite programar más rápido, lo que eleva la productividad de los desarrolladores. Un punto importante a la hora de desarrollar es qué tan rápido se puede hacer, de hecho un factor clave para permitir el desarrollo rápido es contar con buenos diseños de base. La velocidad en la programación se obtiene al reducir los tiempos que lleva la aplicación de cambios, si el código fuente no es fácilmente comprensible, entonces los cambios llevarán más tiempo. Evita que el diseño comience a perderse. Refactorizar mejora el diseño, la lectocomprensión del código fuente y reduce la cantidad de posibles fallas, lo que lleva a mejorar la calidad del software entregado, como así también aumenta la velocidad de desarrollo.

Desventajas

Entre los problemas detectados en el ámbito de la refactorización, podemos destacar:

- *Cambio de las interfaces:* Muchas refactorizaciones modifican la interfaz entre componentes. Lo que no es problema si se tiene acceso a todo el código, ya que se cambiaría de nombre al servicio y se renombrarían todas las llamadas a ese método. El problema aparece cuando las interfaces son usadas por código que no se puede encontrar y/o cambiar.
- *Bases de datos:* La mayoría de las aplicaciones están fuertemente acopladas al esquema de la base de datos. Esta es una de las razones por la que la base de datos es difícil de cambiar. Otra razón es la migración de los datos de una base de datos a otra, algo bastante costoso.

Qué no es refactorizar

Es importante enfatizar qué no se considerar refactorizar:

- Refactorizar no es buscar errores, el código ya debe estar en funcionamiento, no importa si estamos trabajando con un gran programa o solo con un método. Puede que cuando se esté refactorizando se detecte algún error en el código que requiere corregirse, es en este momento en que se deja de refactorizar para pasar a corregir errores.
- Refactorizar no es mejorar el rendimiento del programa, se puede mejorar el código, limpiarlo pero esto no implica que el código funcione más rápido.
- Por tal no se pretende que el código tenga un mayor rendimiento, si no que sea mayor el rendimiento del programador para que este puede mantener de mejorar manera dicho código.
- Refactorizar no es añadir nuevas características, y esto es así porque al refactorizar no se debe cambiar el comportamiento observable del programa.

Momentos para refactorizar

La refactorización no es una actividad que suele planificarse como parte del proyecto, sino que ocurre bajo demanda, cuando se necesita. Existe la llamada regla de los tres strikes¹ (Fowler) que sostiene que la tercera vez que se debe realizar un trabajo similar a uno ya efectuado deberá refactorizarse. La primera vez se realiza directamente, la segunda vez se realiza la duplicación y finalmente, a la tercera se refactoriza.

Otros momentos propicios para refactorizar son:

- (1) *Al momento de agregar funcionalidad:* Es común refactorizar al momento de aplicar un cambio al software ya funcionando, a menudo realizar esto ayuda a comprender mejor el código sobre el que se está trabajando, principalmente si el código no está correctamente estructurado.
- (2) *Al momento de resolver una falla:* El reporte de una falla del software suele indicar que el código no estaba lo suficientemente claro como para evidenciar la misma.
- (3) *Al momento de realizar una revisión de código:* Entre los beneficios de las revisiones de código se encuentra la distribución del conocimiento dentro del equipo de desarrollo, para lo cual la claridad en el código es fundamental. Es común que para el creador del código este sea claro, pero suele ocurrir que para el resto no lo es. La refactorización ayuda a que las revisiones de código provean más resultados concretos, ya que, no solo se realizan nuevas sugerencias sino que se pueden ir implementando de a poco. Esta idea de revisión de código constante es fuertemente utilizada con la técnica de *pair programming* de *extreme programming*. Esta técnica involucra dos desarrolladores por computadora. De hecho implica una constante revisión de código y refactorizaciones a lo largo del desarrollo.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE REFACTORIZACIÓN	VERSIÓN: 1.3 VIGENCIA: 11-10-2011

¹ En *baseball* y *softball* un *strike* es un buen tiro del *pitcher*. Luego de tres de ellos el bateador queda fuera. Una analogía a esto podría ser “la tercera es la vencida”.

Momentos para no refactorizar

Así como existen momentos que son propicios para las refactorizaciones, existen otros que no lo son. Cuando se dispone de código que simplemente no funciona, cuando el esfuerzo necesario para hacerlo funcionar es demasiado grande por su estructura y la cantidad aparente de fallas que hacen que sea difícil de estabilizarlo, lo que ocasiona que se deba reescribir el código desde cero. Una solución factible sería refactorizar el software y dividirlo en varios componentes, y luego decidir si vale la pena refactorizar o reconstruir componente por componente.

Otro momento para no refactorizar es cuando se está próximo a una entrega. En este momento, la productividad obtenida por la refactorización misma será apreciable solo después de la fecha de entrega.

Refactorización Continua

Refactorizar de forma continua es una práctica que consiste en mantener el diseño siempre correcto, refactorizando siempre que sea posible, después de añadir cada nueva funcionalidad. Esta no es una práctica nueva pero está adquiriendo mayor relevancia de mano de las metodologías ágiles.

Dado que una refactorización supone un cambio en la estructura del código sin cambiar la funcionalidad, cuanto mayor sea el cambio en la estructura más difícil será garantizar que no ha cambiado la funcionalidad. Dicho de otra forma, cuanto mayor sea la refactorización, mayor es el número de elementos implicados y mayor es el riesgo de que el sistema deje de funcionar. El tiempo necesario para llevarla a cabo también aumenta y por tanto el coste se multiplica.

Cuando un diseño no es óptimo y necesita ser refactorizado, cada nueva funcionalidad contribuye a empeorar el diseño un poco más. Por ello cuanto más tiempo esperamos mayor es la refactorización necesaria.

Las claves para poder aplicar refactorización continua son:

- Concientización de todo el equipo de desarrollo.
- Habilidad o conocimientos necesarios para identificar qué refactorizaciones son necesarias.
- Compartir con todo el equipo de desarrollo la visión de una arquitectura global que guíe las refactorizaciones en una misma dirección.

El principal riesgo de la refactorización continua consiste en adoptar posturas excesivamente exigentes o criterios excesivamente personales respecto a la calidad del código. Cuando esto ocurre se acaba dedicando más tiempo a refactorizar que a desarrollar. La propia presión para añadir nuevas funcionalidades a la mayor velocidad posible que impone el mercado es suficiente en ocasiones para prevenir esta situación.

Si se mantiene bajo unos criterios razonables y se realiza de forma continuada la refactorización debe tender a ocupar una parte pequeña en relación al tiempo dedicado a las nuevas funcionalidades.

Importancia de las pruebas automáticas

Es importante que al momento de aplicar una refactorización primero se ejecuten pruebas automáticas para determinar que el código funciona. Es decir, que buscamos estar seguros que el código no tiene errores. Luego recién refactorizamos para después volver a ejecutar las pruebas correspondientes y confirmar de esta forma que todo funciona correctamente.

Bad Smells (Malos olores)

Refactorizar es por tanto un medio para mantener el diseño lo más sencillo posible y de calidad.

Algunas características para lograr un código más simple son:

- El código funcional (el conjunto de pruebas de funcionalidad de nuestro código pasan correctamente).
- No existe código duplicado.
- El código permite entender el diseño.
- Minimiza el número de clases y de métodos.
- No requiere excesiva cantidad de comentarios para entender el código.

A pesar de todo lo anterior, refactorizar parece ser muchas veces una técnica en contra del sentido común. ¿Por qué modificar un código que si funciona? ¿Por qué correr el riesgo de introducir nuevos errores?, ¿Cómo se puede justificar el costo de modificar el código sin desarrollar ninguna nueva funcionalidad?

Cada refactorización que realicemos debe estar justificada. Sólo debemos refactorizar cuando identifiquemos código mal estructurado o diseños que supongan un riesgo para la futura evolución de nuestro sistema. Si detectamos que nuestro diseño empieza a ser complicado y difícil de entender, y nos está llevando a una situación donde cada cambio empieza a ser muy costoso, es en ese momento cuando debemos ser capaces de frenar la inercia de seguir desarrollando porque si no lo hacemos nuestro software se convertirá en algo inmantenible, será imposible o demasiado costoso realizar un cambio.

Los síntomas que indican que algún código de software tiene problemas se conocen como "Bad Smells". En [Fowler] podemos encontrar una lista de ellos como código duplicado, métodos largos, clases largas, cláusulas Switch, comentarios, etc.

Una vez identificado el "Bad Smell" se debe aplicar una refactorización que permita corregir ese problema. Para comenzar a refactorizar es imprescindible que el proyecto tenga pruebas automáticas, tanto unitarias como funcionales, que nos permitan saber en cualquier momento al ejecutarlas, si el desarrollo sigue cumpliendo los requisitos que implementaba. Sin pruebas automáticas, refactorizar es una actividad que conlleva un alto riesgo. Al término de una refactorización, sin pruebas automáticas nunca estaremos convencidos de no haber introducido nuevos errores en el código, y poco a poco dejaremos de hacerlo por miedo a estropear lo que ya funciona.

A continuación, se describen brevemente los bad smells habituales:

- (1) *Temporary field (atributo temporal)*: Se observa cuando un objeto tiene variables de instancia que se usan en determinadas circunstancias. Esto genera que sea difícil de entender, ya que siempre se espera que un objeto necesite todas sus variables.
- (2) *Message chains (cadena de mensajes)*: Esto se observa cuando se invoca un objeto a través de otro objeto, y luego a otro objeto a través de este último, y así sucesivamente. Esto genera un alto acoplamiento de código. Puede requerirse un cambio para introducir asociaciones derivadas que recorten el camino a recorrer y que adicionalmente desacoplen los mensajes de la estructura original, las cadenas largas son frágiles ante cambios menores de estructura.
- (3) *Divergent change (cambio divergente)*: Una clase es frecuentemente modificada por diversos motivos, los cuales no suelen estar relacionados entre sí. Este síntoma es el opuesto del siguiente.
- (4) *Shotgun surgery (cambios en cadena)*: Este síntoma se presenta cuando luego de un cambio en un determinado lugar, se deben realizar varias modificaciones adicionales en diversos lugares para compatibilizar dicho cambio.
- (5) *Data class (clase de datos)*: Clases que solo tienen atributos y métodos de acceso a ellos ("get" y "set"). Este tipo de clases deberían cuestionarse dado que no suelen tener comportamiento alguno.
- (6) *Large class (clase grande)*: Si una clase intenta resolver muchos problemas, usualmente suele tener varias variables de instancia... lo que suele conducir a código duplicado.
- (7) *Lazy class (clase perezosa)*: Cada clase de un programa debe ser mantenida. Por lo que una clase que no aporta demasiado debe ser eliminada.
- (8) *Comments (comentarios)*: Al encontrar un gran comentario se debería reflexionar sobre el por qué algo necesita ser tan explicado y no es autoexplicativo. Los comentarios ocultan muchas veces a otro mal olor.
- (9) *Duplicated code (código duplicado)*: Es la principal razón para refactorizar. Si se detecta el mismo código en más de un lugar, se debe buscar la forma de extraerlo y unificarlo.
- (10) *Feature envy (envidia de características)*: un método que utiliza más cantidad de elementos de otra clase que de la propia. Se suele resolver el problema pasando el método a la clase cuyos componentes son más requeridos para usar.
- (11) *Switch statements (estructura de agrupación condicional)*: Muy a menudo encontramos la sentencia switch dispersa en diferentes lugares de un programa o demasiados if anidados. Si se agrega una nueva condición seguramente estaremos obligados a revisar todas estas sentencias para cambiarlas. El polimorfismo es una manera elegante de hacer frente a este problema.
- (12) *Speculative generality (generalidad especulativa)*: Cuando agregamos comportamiento por si acaso, el resultado será más difícil de entender y mantener. Si ese comportamiento no se utiliza deberá eliminarse.
- (13) *Data clumps (grupo de datos)*: Cuando a menudo se observan tres o cuatro veces los mismos elementos de datos en diferentes lugares: los atributos de un par de clases, los mismos parámetros en muchos métodos; debemos transformarlos en un único objeto.

- (14) *Middle man (intermediario)*: Una de las características principales de los objetos es el encapsulamiento. La encapsulación a menudo viene acompañada por la delegación. Si una clase delega la mitad o más de sus servicios a otra, debe considerarse la posibilidad de eliminar el intermediario.
- (15) *Inappropriate intimacy (intimidad inadecuada)*: Clases que tratan con la parte privada de otras. Se debe restringir el acceso al conocimiento interno de una clase.
- (16) *Parallel inheritance hierarchies (jerarquías paralelas)*: Este es un caso especial de "Shotgun surgery (cambio en cadena)". En este caso, cada vez que se hace una subclase de una clase, también se deberá hacer una subclase de otra. Esto lo detectamos, porque los prefijos de los nombres de clases en una jerarquía son los mismos que los prefijos de otra jerarquía.
- (17) *Refused bequest (legado rechazado)*: Subclases que usan solo pocas características de sus superclases. Si las subclases no necesitan o no requieren todo lo que sus superclases les proveen por herencia, esto suele indicar que como fue pensada la jerarquía de clases no es correcto. La delegación suele ser la solución a éste tipo de inconvenientes.
- (18) *Long parameter list (lista de parámetros larga)*: En la programación orientada a objetos no se suelen pasar muchos parámetros a los métodos, sino sólo aquellos mínimamente necesarios para que el objeto involucrado consiga lo necesario. Éste tipo de métodos, los que reciben muchos parámetros, suelen variar con frecuencia, se tornan difíciles de comprender e incrementan el acoplamiento.
- (19) *Long method (método largo)*: Legado de la programación estructurada. En la programación orientada a objetos cuando más corto es un método más fácil de reutilizarlo es.
- (20) *Primitive obsession (obsesión primitiva)*: Uso excesivo de tipos primitivos. Existen grupos de tipos primitivos (enteros, caracteres, reales, etc.) que deberían modelarse como objetos. Debe eliminarse la reticencia a usar pequeños objetos para pequeñas tareas, como son dinero, rangos o números de teléfono que debieran muchas veces ser objetos.

Bad Smells y sus refactorizaciones más típicas

Bad Smell	Refactorizaciones más utilizadas
Atributo temporal	Extract class. Introduce null object.
Cadena de mensajes	Hide delegate.
Cambio divergente	Extract class.
Cambios en cadena	Move method. Move field. Inline class.
Clase de datos	Move method. Encapsulate field. Encapsulate collection.
Clase grande	Extract class. Extract subclass. Extract interface. Replace data value with object.
Clase perezosa	Inline class. Collapse hierarchy.
Comentarios	Extract method. Introduce assertion.
Duplicación de código	Extract method. Extract class. Pull up method. Form template method.
Envidia de características	Move method. Move field. Extract method.
Estructuras de agrupación condicional	Replace conditional with polymorphism. Replace type code with subclasses. Replace type code with state/strategy. Replace parameter with explicit methods. Introduce null object.
Generalidad especulativa	Collapse hierarchy. Inline class. Remove parameter. Rename method.
Grupos de datos	Extract class. Introduce parameter object. Preserve whole object.
Intermediario	Remove middle man. Inline method. Replace delegation with inheritance.

Intimidad inadecuada	Move method. Move field. Change bidirectional association to unidirectional. Replace inheritance with delegation. Hide delegate.
Jerarquías paralelas	Move method. Move field.
Legado rechazado	Replace inheritance with delegation.
Lista de parámetros larga	Replace parameter with method. Introduce parameter object. Preserve whole object.
Método largo	Extract method. Replace temp with query. Replace method with method object. Decompose conditional.
Obsesión primitiva	Replace data value with object. Extract class. Introduce parameter object. Replace array with object. Replace type code with class. Replace type code with subclasses. Replace type code with state/strategy.

Agrupación de refactorizaciones

La manera de agrupar las refactorizaciones puede ser diversa y no existe un criterio único, podríamos dividir nuestro análisis y preguntarnos:

- A nivel de método:
 - ¿El nombre es correcto?
 - ¿Los parámetros tienen sentido?
 - ¿Hay demasiadas variables temporales?
 - ¿Hay código duplicado?
 - ¿Sería mejor separar ese código?
- A nivel de clase:
 - ¿Ese método debe estar en esta clase?
 - ¿Hay muchos métodos similares en la clase?
- A nivel de comunicación entre clases:
 - ¿La clase A está usando métodos de la clase B?
 - Si esto se cumple, ¿Por qué es así?

Catálogo de refactorizaciones

En el formato de una refactorización se pueden distinguir cinco partes (Fowler):

- (1) *Nombre de la refactorización*: Importante para construir un vocabulario, de igual forma que los patrones de diseño han creado un vocabulario las refactorizaciones deben contribuir a ampliarlo.
- (2) *Resumen de la que hace y de la situación en la cual se puede necesitar*.
- (3) *Motivación*: Por qué la refactorización debería aplicarse y las circunstancias en las cuales no debería usarse.
- (4) *Mecanismo*: Descripción de cómo llevar a cabo la refactorización.
- (5) *Ejemplo*: Un uso de la refactorización.

Por otro lado, hay veces que se utiliza código para describir la refactorización y otros diagramas UML a nivel de implementación.

A continuación, se presenta la lista de refactorizaciones (Fowler):



Add Parameter	Pull Up Constructor Body
Change Bidirectional Association to Unidirectional	Pull Up Field
Change Reference to Value	Pull Up Method
Change Unidirectional Association to Bidirectional	Push Down Field

Change Value to Reference	Push Down Method
Collapse Hierarchy	Remove Assignments to Parameters
Consolidate Conditional Expression	Remove Control Flag
Consolidate Duplicate Conditional Fragments	Remove Middle Man
Decompose Conditional	Remove Parameter
Duplicate Observed Data	Remove Setting Method
Encapsulate Collection	Rename Method
Encapsulate Downcast	Replace Array with Object
Encapsulate Field	Replace Conditional with Polymorphism
Extract Class	Replace Constructor with Factory Method
Extract Interface	Replace Data Value with Object
Extract Method	Replace Delegation with Inheritance
Extract Subclass	Replace Error Code with Exception
Extract Superclass	Replace Exception with Test
Form Template Method	Replace Inheritance with Delegation
Hide Delegate	Replace Magic Number with Symbolic Constant
Hide Method	Replace Method with Method Object
Inline Class	Replace Nested Conditional with Guard Clauses
Inline Method	Replace Parameter with Explicit Methods
Inline Temp	Replace Parameter with Method
Introduce Assertion	Replace Record with Data Class
Introduce Explaining Variable	Replace Subclass with Fields
Introduce Foreign Method	Replace Temp with Query
Introduce Local Extension	Replace Type Code with Class
Introduce Null Object	Replace Type Code with State/Strategy
Introduce Parameter Object	Replace Type Code with Subclasses
Move Field	Self Encapsulate Field
Move Method	Split Loop
Parameterize Method	Split Temporary Variable
Preserve Whole Object	Substitute Algorithm

Especificación de algunas refactorizaciones

Refactorización que extrae método

Extract Method: Extraer método

Resumen	Es una operación de refactorización que proporciona una manera sencilla para crear un nuevo método a partir de un fragmento de código de un miembro existente.
Motivación	Métodos demasiados largos con partes de código que necesitan un comentario para entender su propósito.
Mecanismo	<ol style="list-style-type: none"> (1) Crear un nuevo método con un nombre acorde a su intención. (2) Copiar el código desde el método fuente hasta el método destino. (3) Revisar el código buscando variables locales al método. Las variables son ahora variables locales o parámetros en el método destino. (4) Buscar variables temporales y declararlas en el método destino. (5) Si alguna variable es modificada, tratar de extraer el código como un método Query. Asignar a la variable el resultado del método Query. (6) Los parámetros son aquellas variables que necesitan ser leídas a partir del código extraído. (7) Reemplazar el código extraído con la invocación al nuevo método. (8) Compilar y probar.
Ejemplo	<pre> public void printInvoice(Order order) { //Imprimir el encabezado de la factura System.out.println ("Factura"); System.out.println ("====="); //Imprimir datos del cliente System.out.println ("Cliente "); System.out.println (order.getCustomer().getName()); //... } </pre>  <pre> public void printInvoice(Order order) { printInvoiceHeader(); //Imprimir datos del cliente System.out.println ("Cliente "); System.out.println (order.getCustomer().getName()); //... } public void printInvoiceHeader() { System.out.println ("Factura"); System.out.println ("====="); } </pre>  <pre> public void printInvoice(Order order) { printInvoiceHeader(); printCustomerDetails(order.getCustomer()); //... } public void printInvoiceHeader() { System.out.println ("Factura"); System.out.println ("====="); } </pre>


```
}

public void printCustomerDetails(Customer customer) {
    System.out.println ("Cliente ");
    System.out.println (customer.getName());
}
```

Método de refactorización en línea

Inline Method: Método en línea

Resumen El cuerpo de un método es tan claro como su nombre.

Motivación Mucha delegación.

Mecanismo

- (1) Chequear que el método no es polimórfico.
- (2) Encontrar todas las llamadas al método.
- (3) Reemplazar cada llamada con el cuerpo del método.
- (4) Compilar y probar.
- (5) Remover la definición del método.

Ejemplo

```
public void calculateAreaOfCircle(double radius) {
    double area = getValueOfPI() * Math.pow(radius, 2);
    System.out.println ("Area: " + area);
}

private double getValueOfPI () {
    return Math.PI;
}
```

↓

```
public void calculateAreaOfCircle(double radius) {
    double area = Math.PI * Math.pow(radius, 2);
    System.out.println ("Area: " + area);
}
```

Refactorizaciones que eliminan temporales

Replace Temp with Query: Reemplazar temporales con consultas

Resumen Extraer la expresión en un método. Sustituir todas las referencias de la variable por la invocación al método. El nuevo método puede ser luego utilizado en otros métodos.

Motivación Se está utilizando una variable temporal para manejar el resultado de una expresión.

Mecanismo

- (1) Buscar la variable temporal que es asignada una sola vez.
- (2) Declarar la variable como final.
- (3) Compilar
- (4) Extraer la expresión del lado derecho como un nuevo método.
- (5) Compilar y probar
- (6) Reemplazar la variable temporal por el método extraído.

Ejemplo

```
public boolean checkWithdrawalStatus(double amoutToWithdraw) {
    //Crear temporal
    double availableBalance = _balance - _pendingCharges;

    //Usa temporal
    if(availableBalance > amoutToWithdraw) {
```

	<pre> System.out.println("Aceptado"); return true; } else { System.out.println("Solo tienes " + availableBalance + " disponible"); return false; } } </pre> <p style="text-align: center;">↓</p> <pre> public boolean checkWithdrawalStatus(double amoutToWithdraw) { if(getAvailableBalance() > amoutToWithdraw) { System.out.println("Aceptado"); return true; } else { System.out.println("Solo tienes " + getAvailableBalance() + " disponible"); return false; } } private double getAvailableBalance() { return _balance - _pendingCharges; } </pre>
--	--

Inline Temp: Temporales en línea

Resumen	Quitar variable temporal utilizar una única vez.
Motivación	Variable temporal involucrada en la refactorización y es asignada solo una vez.
Mecanismo	<ol style="list-style-type: none"> (1) Declarar la variable temporal como final si aún no lo está (verifica que está asignada una sola vez). (2) Encontrar todas las referencias a la variable temporal y reemplazarla con el lado derecho de la expresión. (3) Compilar y probar. (4) Remover la declaración y asignación a la variable. (5) Compilar y probar.
Ejemplo	<pre> public boolean checkWithdrawalStatus(double amoutToWithdraw) { final double temp = getAvailableBalance(); if(temp > amoutToWithdraw) { System.out.println("Aceptado"); return true; } else { System.out.println("Solo tienes " + temp + " disponible"); return false; } } private double getAvailableBalance() { return _balance - _pendingCharges; } </pre> <p style="text-align: center;">↓</p>

```
public boolean checkWithdrawalStatus(double amoutToWithdraw) {
    if(getAvailableBalance() > amoutToWithdraw) {
        System.out.println("Aceptado");
        return true;
    }
    else {
        System.out.println("Solo tienes " + getAvailableBalance() + "
disponible");
        return false;
    }
}

private double getAvailableBalance() {
    return _balance - _pendingCharges;
}
```

Refactorizaciones que añaden temporales

Split Temporary Variable: División de variables temporales

Resumen Hacer una variable temporal por cada asignación.

Motivación Variable temporal asignada más de una vez.

Mecanismo

- (1) Cambiar el nombre de la variable temporal y su primera asignación.
- (2) Declarar la nueva variable temporal como final.
- (3) Cambiar todas las referencias de la variable temporal antes de su segunda asignación.
- (4) Declarar la variable temporal en la segunda asignación.
- (5) Compilar y probar.
- (6) Repetir en etapas, en cada etapa renombrar la variable y cambiar las referencias hasta la próxima asignación.

Ejemplo

```
double temp = item.getPrice() * item.getQuantity();
System.out.println("Total por línea: " + temp);

temp = order.getTotal() - order.getDiscount();
System.out.println("Cantidad debida: " + temp);
```

↓

```
double lineTotal = item.getPrice() * item.getQuantity();
System.out.println("Total por línea: " + lineTotal);

double amountDue = order.getTotal() - order.getDiscount();
System.out.println("Cantidad debida: " + amountDue);
```

Introduce Explaining Variable: Introducir variables explicativas

Resumen Hacer el código más entendible.

Motivación Expresiones complicadas.

Mecanismo

- (1) Declarar la variable temporal como final, y asignarle el resultado asociado a la parte de la expresión compleja.
- (2) Reemplazar la parte de la expresión con la variable temporal.
- (3) Compilar y probar.
- (4) Repetir pasos 1 al 3 para el resto de la expresión.

Ejemplo	<pre> if(stock.checkStatus(order.getItem()) > order.getQuantity() && order.getTotal() > 99 && order.getCustomer().getBillingAddress().getLine1(). equals(order.getShippingAddress().getLine1())) { System.out.println("Aceptado"); ... } else { ... } ↓ final boolean stockAvailable = stock.checkStatus(order.getItem()) > order.getQuantity(); final boolean freeShipping = order.getTotal() > 99; final boolean addressMatches = order.getCustomer().getBillingAddress(). getLine1().equals(order.getShippingAddress().getLine1()); if(stockAvailable && freeShipping && addressMatches) { System.out.println("Aceptado"); // ... } else { // ... } </pre>
---------	--

Remove Assignments to Parameters: Eliminar asignaciones a parámetros	
Resumen	El parámetro es reasignado dentro del método.
Motivación	Evitar la modificación del parámetro recibido.
Mecanismo	<ol style="list-style-type: none"> (1) Crear una variable temporal para el parámetro. (2) Reemplazar todas las referencias hechas al parámetro después de la asignación, a la variable temporal. (3) Cambiar la asignación del parámetro a la asignación de la variable temporal. (4) Compilar y probar.
Ejemplo	<pre> public void checkStock(OrderItem item, int desired) { //¿Se puede cumplir con el pedido? if(item.getStockUnits() < desired) { //Asignando a un parámetro primitivo desired = item.getStockUnits(); System.out.println("Solo " + desired + " unidades disponibles"); //Asignando a un parámetro de objeto item = new OrderItem(item.getAlternateID()); System.out.println("Sugiero " + item.getName() + " en su lugar"); // ... } // ... } </pre> <p style="text-align: center;">↓</p>

```
public void checkStock(OrderItem item, int desired) {
    //¿Se puede cumplir con el pedido?
    if(item.getStockUnits() < desired) {
        //Se usa una temporal en línea en lugar de asignar al parámetro
        System.out.println("Solo " + item.getStockUnits() + " unidades
disponibles");

        //Se usa una nueva temporal en lugar de asignar al parámetro
        OrderItem alternate = new OrderItem(item.getAlternateID());
        System.out.println("Sugiero " + alternate.getName() + " en su lugar");

        // ...
    }
    // ...
}
```

Refactorización que reemplaza método con método de un objeto

Replace Method with Method Object: Reemplazar método con método de un objeto

Resumen Método largo que usa variables locales y no se puede usar "Extract Method"

Motivación Crear métodos más sencillos para hacer más compresible el código.

Mecanismo

- (1) Crear una nueva clase cuyo nombre es el método.
- (2) Agregar a la nueva clase un campo del tipo final hacia el objeto origen y un campo para cada variable temporal y cada parámetro en el método.
- (3) Crear un constructor que recibe el objeto fuente y cada parámetro.
- (4) Crear un nuevo método "compute" en la nueva clase.
- (5) Copiar el cuerpo del método original en el método "compute". Usar el campo del objeto fuente para cualquier invocación a los métodos del objeto original.
- (6) Compilar.
- (7) Reemplazar el método viejo por uno que crea el objeto nuevo e invoque el método "compute".

Ejemplo

```
class Order {

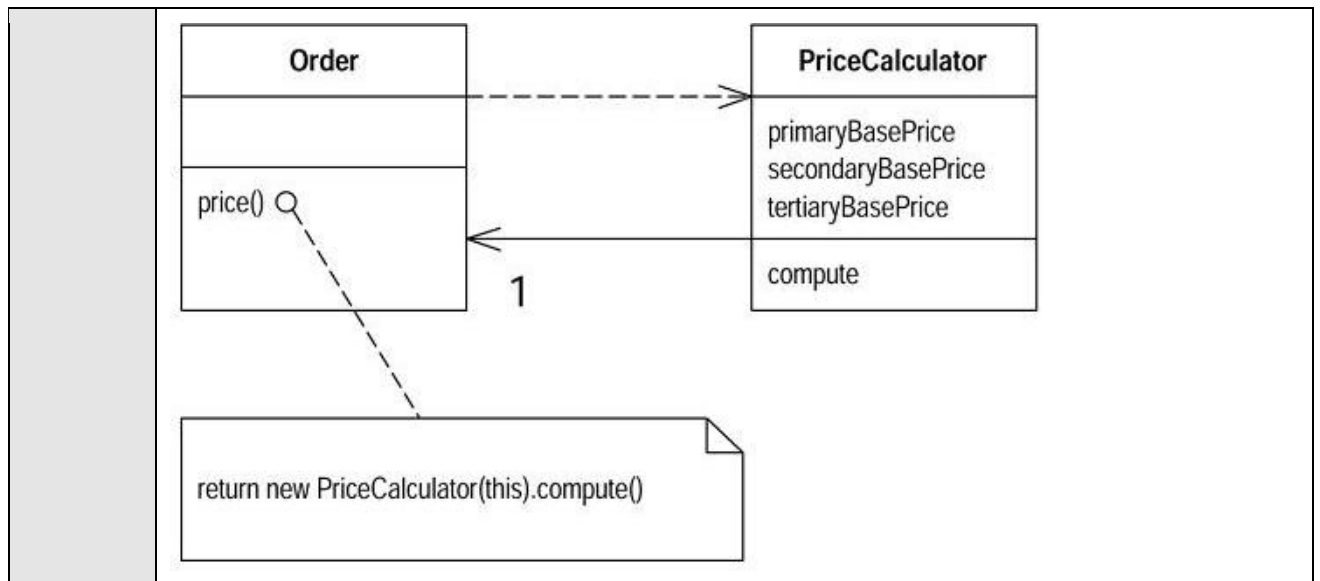
    public double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;

        // long computation;

    }

}
```





Refactorización que sustituye algoritmo

Substitute Algorithm: Sostituire algoritmo	
Resumen	Reemplazar un algoritmo por uno más claro.
Motivación	Crear métodos más sencillos para hacer más comprensible el código.
Mecanismo	(1) Preparar algoritmo alternativo tal que compile. (2) Ejecutar y probar el nuevo algoritmo. (3) Si los resultados no son los mismos, usar el viejo algoritmo para comparar al probar.
Ejemplo	<pre> public String findPerson(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 ""; } </pre> <p style="text-align: center;">↓</p> <pre> public String findPerson(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 ""; } </pre>

Refactorización que mueve métodos

Move Method: Mover método

Resumen	Crear un nuevo método con un cuerpo similar en la clase que cuenta con la mayoría de las características utilizadas. Cambiar el viejo método con una simple delegación, o eliminarlo por completo.
Motivación	Un método que utiliza más características de otra clase que de la clase en la que está definido.
Mecanismo	<ol style="list-style-type: none"> (1) Examinar todas las características utilizadas por el método fuente que se definen en la clase origen. (2) Comprobar las subclases y superclases. (3) Declarar el método en la clase destino. (4) Copiar el código del método a la clase destino. Ajustar el método para que funcione en la nueva clase. (5) Compilar la clase destino. (6) Reemplazar las llamadas originales. (7) Compilar y probar.
Ejemplo	<pre> public class Customer { private String name; ... public void printInvoice(Order order) { //Imprimir encabezado de la factura System.out.println("Invoice: " + order.getOrderID()); System.out.println("Date: " + order.getDate().toString()); //Imprimir datos del cliente System.out.println("Customer: "); System.out.println(this.getName()); Address address = order.getAddress(); System.out.println(" " + address.getStreet()); System.out.println(" " + address.getCity()); System.out.println(" " + address.getState()); System.out.println(" " + address.getZip()); System.out.println("Order Items:"); Item[] items = order.getItems(); for(int i = 0; i < items.length; i++) { System.out.println("* Name: " + items[i].getName()); System.out.println("* ID: " + items[i].getId()); System.out.println("* Quantity: " + items[i].getQuantity()); System.out.println("* Price: " + items[i].getPrice()); } } ... } ↓ public class Order { private int orderID; private Date date; private Customer customer; private Address address; private Item[] items; ... </pre>


```

public void printInvoice() {
    //Imprimir encabezado de la factura
    System.out.println("Invoice: " + this.getOrderID());
    System.out.println("Date: " + this.getDate().toString());

    //Imprimir datos del cliente
    System.out.println("Customer: ");
    System.out.println(this.getCustomer().getName());

    Address address = this.getAddress();
    System.out.println(" " + address.getStreet());
    System.out.println(" " + address.getCity());
    System.out.println(" " + address.getState());
    System.out.println(" " + address.getZip());

    System.out.println("Order Items:");
    Item[] items = this.getItems();
    for(int i = 0; i < items.length; i++) {
        System.out.println("* Name: " + items[i].getName());
        System.out.println("* ID: " + items[i].getId());
        System.out.println("* Quantity: " + items[i].getQuantity());
        System.out.println("* Price: " + items[i].getPrice());
    }

    ...
}

```

Refactorización que extrae clases

Extract class: Extraer clase	
Resumen	Crear una nueva clase y mover los campos y métodos pertenecientes a la vieja clase en la nueva clase.
Motivación	La clase realiza más cosas que lo que originalmente iba hacer.
Mecanismo	<ol style="list-style-type: none"> (1) Decidir cómo dividir las responsabilidades de la clase. (2) Crear una nueva clase para expresar las responsabilidades a extraer. (3) Hacer un vínculo desde la antigua a la nueva clase. (4) Utilizar "Mover campo" en cada campo que se desea mover. (5) Compilar y probar después de cada movimiento. (6) Utilizar "Mover método" para mover métodos de la clase viaja a la nueva. (7) Compilar y probar después de cada movimiento. (8) Revisar y reducir las interfaces de cada clase. (9) Decidir si se va a exponer la nueva clase.
Ejemplo	<pre> class Customer { private String name; private String officeAreaCode; private String officeNumber; public String getName() { return this.name; } public void setName(String arg) { this.name = arg; } public String getTelephoneNumber() { </pre>

```

        return "(" + this.officeAreaCode + ") " + this.officeNumber);
    }

    public String getOfficeAreaCode() {
        return this.officeAreaCode;
    }

    public void setOfficeAreaCode(String arg) {
        this.officeAreaCode = arg;
    }

    public String getOfficeNumber() {
        return this.officeNumber;
    }

    public void setOfficeNumber(String arg) {
        this.officeNumber = arg;
    }
}

```



```

public class TelephoneNumber {

    private String officeAreaCode;
    private String officeNumber;

    public String getTelephoneNumber() {
        return "(" + this.officeAreaCode + ") " + this.officeNumber);
    }

    public String getOfficeAreaCode() {
        return this.officeAreaCode;
    }

    public void setOfficeAreaCode(String arg) {
        this.officeAreaCode = arg;
    }

    public String getOfficeNumber() {
        return this.officeNumber;
    }

    public void setOfficeNumber(String arg) {
        this.officeNumber = arg;
    }
}

class Customer {

    private String name;
    private TelephoneNumber officeTelephone;


    public String getName() {
        return this.name;
    }

    public void setName(String arg) {
        this.name = arg;
    }
}

```

	<pre> public String getTelephoneNumber() { return this.officeTelephone.getTelephoneNumber(); } public TelephoneNumber getOfficeTelephone() { return this.officeTelephone; } public void setOfficeTelephone(TelephoneNumber arg) { this.officeTelephone = arg; } } </pre>
Nota	<p>La refactorización que realiza el proceso inverso se conoce como <i>Inline Class: Clase en línea</i>, es decir, una clase que no tiene demasiados atributos y sus métodos son simples y no agrega demasiado valor, termina siendo absorbida por otra clase.</p> <p>Esto se presenta mucho cuando se mueven métodos entre clases y dejan alguna de ellas sin demasiado que aportar, por tal se terminan de migrar su definición y se borra dicha clase.</p>

Haciendo condicionales más fáciles de leer

Decompose Conditional: Descomposición condicional	
Resumen	Extraer la expresión condicional y crear un método que luego se utilice en la sentencia condicional.
Motivación	Se tiene una expresión condicional complicada.
Mecanismo	<ol style="list-style-type: none"> (1) Extraer la condición creando un nuevo método que retorne el resultado de la misma. (2) Modificar la sentencia condicional utilizando el método creado. (3) Compilar y probar.
Ejemplo	<pre> if (order.getCustomer().getStatus().equals("P") (order.getTotal() > 1000 && order.getWeight() < 500)) { shipping = order.getTotal() * 0.05; if (shipping > 100) shipping = 100; } else { shipping = order.getTotal() * 0.08; if (shipping > 200) shipping = 200; } </pre> <div style="text-align: center;">  </div> <pre> if (largeOrImportant(order)) { shipping = discountShippingRate(order); } else { shipping = normalShippingRate(order); } public boolean largeOrImportant(Order order) { return order.getCustomer().getStatus().equals("P") (order.getTotal() > 1000 && order.getWeight() < 500); } public double discountShippingRate(Order order) { double result = order.getTotal() * 0.05; if (result > 100) result = 100; return result; } </pre>

```
public double normalShippingRate(Order order) {
    double result = order.getTotal() * 0.08;
    if(result > 200) result = 200;
    return result;
}
```

Consolidate Conditional Expression: Consolidar la expresión condicional

Resumen Se combina en una única expresión condicional el conjunto de expresiones que tienen el mismo resultado.

Motivación Se tiene un conjunto de condiciones con el mismo resultado.

Mecanismo

- (1) Comprobar que ninguna de las expresiones condiciones tiene un efecto secundario.
- (2) Armar una única expresión condicional utilizando operadores lógicos.
- (3) Compilar y probar.
- (4) Considerar el uso de "Extraer método" para la condición construida.

Ejemplo

```
public double calculateShipping() {
    if(!_isEmployee) return 0;
    if(!_isPlatinumCustomer) return 0;
    if(!_isGoldCustomer) return 0;
    if(!_hasCoupon) return 0;

    //Calcular el envío
    //...
}
```

↓

```
public double calculateShipping() {
    if(freeShipping()) return 0;

    //Calcular el envío
    //...
}

public boolean freeShipping() {
    return (!_isEmployee || !_isPlatinumCustomer || !_isGoldCustomer ||
    !_hasCoupon);
}
```

Consolidate Duplicate Conditional Fragments: Consolidación de fragmentos de código condicional duplicados

Resumen Aquel fragmento de código que se repite se mueve fuera de la expresión.

Motivación El mismo fragmento de código se encuentra en todas las ramas de la sentencia condicional.

Mecanismo

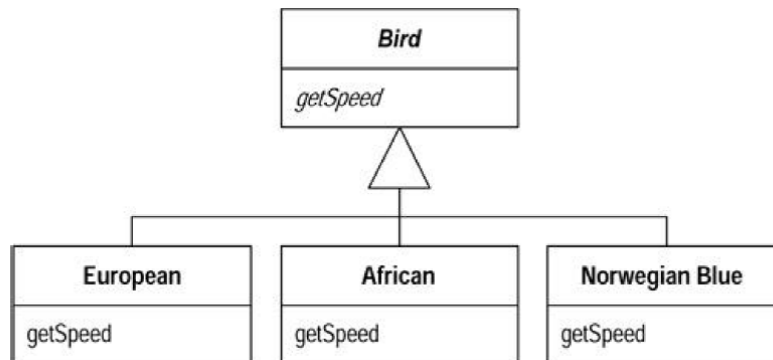
- (1) Identificar el código que se ejecuta de la misma manera independientemente de la condición.
- (2) Si el código común es al principio, moverlo antes del condicional.
- (3) Si el código común es al final, moverlo después del condicional.
- (4) Si el código común está en el medio, evaluar si el código anterior o posterior no cambia nada. Si lo hace, puede mover el código común hacia adelante o hacia atrás a los extremos y a continuación, se puede mover como se describe para el código al final o al principio.
- (5) Si hay más de una declaración, se debe extraer ese código en un método.

Ejemplo `if(largeOrImportant(order)) {`

	<pre> shipping = discountShippingRate(order); order.addShippingCost(shipping); } else { shipping = normalShippingRate(order); order.addShippingCost(shipping); } ↓ if(largeOrImportant(order)) { shipping = discountShippingRate(order); } else { shipping = normalShippingRate(order); } order.addShippingCost(shipping); </pre>
--	---

Refactorización que reemplaza condicional con polimorfismo

Replace Conditional with Polymorphism: Reemplazar condicional con polimorfismo	
Resumen	El método que contiene la sentencia condicional se sobrescribe en las subclases y se convierte el método de la superclase en abstracto.
Motivación	Se cuenta con una sentencia condicional que elige un comportamiento diferente en función de un tipo de objeto.
Mecanismo	<ol style="list-style-type: none"> (1) Si la sentencia condicional es una parte de un método más grande, desarmar la sentencia condicional y utilizar "Extraer método". (2) Si es necesario, utilizar "Mover método" para colocar el condicional en la superclase. (3) Tomar una de las subclases, crear un método que anule la sentencia condicional. Copiar el cuerpo de la instrucción condicional en el método de la subclase y ajustarlo a su medida. (4) Compilar y probar. (5) Eliminar esa parte de la sentencia condicional. (6) Compilar y probar. (7) Repetir el procedimiento con cada parte de la sentencia condicional hasta que todas las condiciones evaluadas se conviertan en métodos de subclases. (8) Hacer que el método se convierta en abstracto en la superclase.
Ejemplo	<pre> public double getSpeed() { switch (_type) { case EUROPEAN: return getBaseSpeed(); case AFRICAN: return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts; case NORWEGIAN_BLUE: return (_isNailed) ? 0 : getBaseSpeed(_voltage); } throw new RuntimeException ("Should be unreachable"); } </pre> <p style="text-align: center;">↓</p>



Refactorización que reemplaza código de tipo con subclases

Replace Type Code with Subclasses: Reemplazar código de tipo con subclases

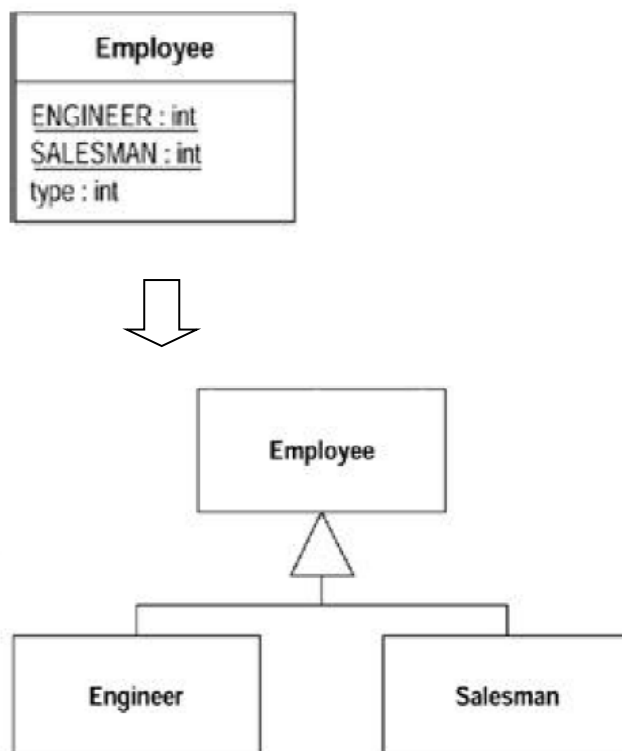
Resumen Se reemplaza la tipificación de código con subclases.

Motivación Según el tipo del objeto, se determina el comportamiento de la clase.


Mecanismo

- (1) Encapsular la tipificación del código.
- (2) Por cada valor de tipificación, crear una subclase. Sobrescribir el método de obtención del objeto (get) en cada subclase con el retorno que corresponda.
- (3) Quitar de la superclase el campo que representa cada tipificación. Declarar el método de obtención del objeto como abstracto.
- (4) Compilar y probar.


Ejemplo



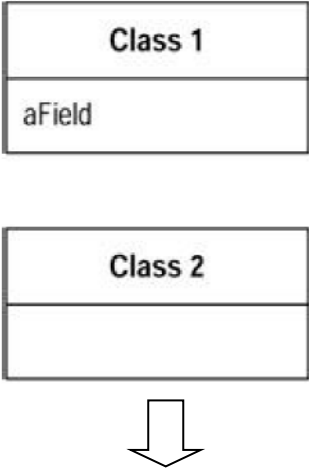
Refactorizaciones de encapsulamiento

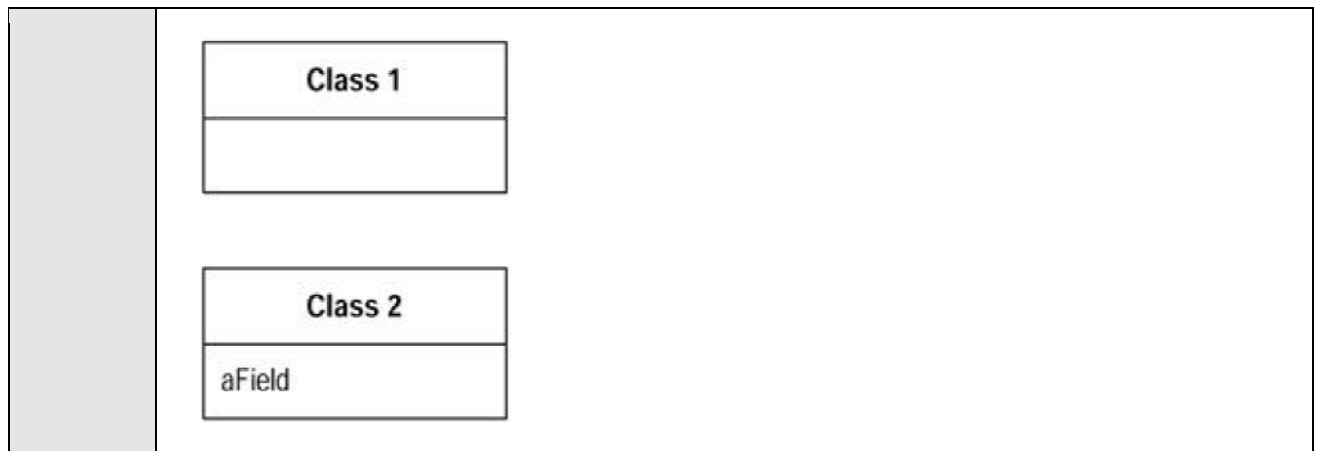
<i>Self Encapsulate Field</i> : Auto encapsulamiento de campo	
Resumen	Crear los métodos accesorios del campo y utilizarlos para acceder al mismo.
Motivación	Se está utilizando el campo directamente pero el acoplamiento con el campo resulta engorroso.
Mecanismo	<ol style="list-style-type: none"> (1) Crear los métodos accesorios del campo. (2) Encontrar todas las referencias al campo y reemplazarlas con el método accesor que corresponda. (3) Hacer el campo privado. (4) Verificar que todas las referencias están corregidas. (5) Compilar y probar.
Ejemplo	<pre>class IntRange { private int _low, _high; public boolean includes (int arg) { return arg >= this._low && arg <= this._high; } //... }</pre>  <pre>class IntRange { private int _low, _high; public int getLow() { return this._low; } public int getHigh() { return this._high; } public boolean includes (int arg) { return arg >= this.getLow() && arg <= this.getHigh(); } //... }</pre>

<i>Encapsulate Field</i> : Encapsulamiento de campos	
Resumen	Hacer privado el acceso al campo y programar los métodos accesorios.
Motivación	Hay campos con acceso público.
Mecanismo	<ol style="list-style-type: none"> (1) Crear los métodos accesorios del campo. (2) Encontrar las llamadas fuera de la clase que hacen referencia al campo. Reemplazar las mismas con el método accesor que corresponda. (3) Compilar y probar después de cada cambio. (4) Una vez que todas las referencias se cambiaron, declarar el campo como privado. (5) Compilar y probar.

Ejemplo	<pre>class Person { public String _name; //... }</pre>  <pre>class Person { private String _name; public String getName() { return this._name; } public void setName(String arg) { this._name = arg; } //... }</pre>
---------	---


Refactorización que mueve campos

Move Field: Mover campos	
Resumen	Se crea un nuevo campo en la clase destino y se actualiza la referencia al mismo.
Motivación	Un campo es o será utilizado por otra/s clases más que la clase donde fue definido.
Mecanismo	<ol style="list-style-type: none"> (1) Si el campo es público, utilizar "Encapsulamiento de campo". (2) Compilar y probar. (3) Crear un campo en la clase destino con los métodos accesorios correspondientes. (4) Compilar la clase destino. (5) Determinar cómo hacer referencia a la clase destino desde el origen. (6) Remover el campo de la clase origen. (7) Reemplazar la referencia en la clase origen invocado el método accesor correspondiente (Campo autoencapsulado) (8) Compilar y probar.
Ejemplo	




Trabajando con cúmulos de datos

Preserve Whole Object: Preservación del objeto entero

Resumen	Se envía como parámetro un objeto.
Motivación	Se usan varios valores de un objeto y se pasan como parámetros en la llamada de un método.
Mecanismo	<ol style="list-style-type: none"> (1) Crear un nuevo parámetro que permita enviar el objeto entero. (2) Compilar y probar. (3) Determinar qué parámetros pueden ser reemplazados por el objeto entero. (4) Tomar un parámetro y reemplazar las referencias al mismo dentro del cuerpo del método invocado. (5) Eliminar el parámetro. (6) Compilar y probar. (7) Repetir el procedimiento por cada parámetro que se pueda reemplazar con el objeto entero. (8) Quitar el código en el método de llamada que obtiene los parámetros borrados. (9) Compilar y probar.
Ejemplo	<pre> double cost = order.getTotal(); float weight = order.getWeight(); String destination = order.getPostalCode(); double shippingCost = calculateShipping(cost, weight, destination); public double calculateShipping(cost, weight, destination) { // ... } </pre>  <pre> double shippingCost = calculateShipping(order); public double calculateShipping(Order order) { // ... } </pre>

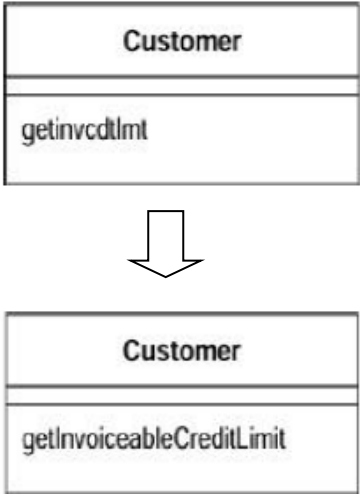
Introduce Parameter Object: Introducir objeto como parámetro

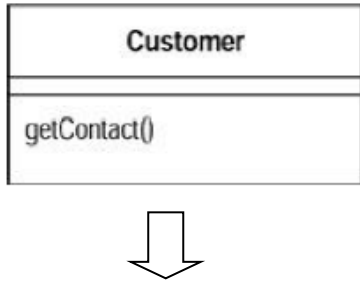
Resumen	Reemplazar parámetro con un objeto.
---------	-------------------------------------

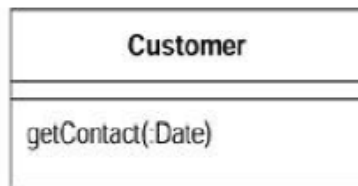
Motivación	Se tiene un grupo de parámetros que están fuertemente relacionados.
Mecanismo	<ol style="list-style-type: none"> (1) Crear una nueva clase que represente el grupo de parámetros que está relacionado. (2) Compilar. (3) Utilizar "Agregar parámetro" para el nuevo grupo de datos. Usar un valor nulo para este parámetro en todas las llamadas al método. (4) Para cada parámetro en el grupo de datos, eliminar el parámetro de la firma del método. Modificar la llamada al método para que utilice un objeto. (5) Recopilar y probar después de quitar cada parámetro. (6) Una vez que se hayan eliminado los parámetros, buscar qué comportamiento se puede mover dentro del parámetro objeto. Utilizar "Mover método".
Ejemplo	<pre>public void drawRectangle(int x, int y, int height, int width) { // ... } public void drawEllipse(int x, int y, int height, int width) { // ... } public void drawStar(int x, int y, int height, int width) { // ... }</pre>  <pre>class Frame { private int x; private int y; private int height; private int width; // ... } public void drawRectangle(Frame frame) { // ... } public void drawEllipse(Frame frame) { // ... } public void drawStar(Frame frame) { // ... }</pre>

Refactorizaciones para simplificar llamado a métodos

Rename Method: Renombrado de método	
Resumen	Se cambia el nombre del método. En este caso se sugiere nombrar al método de la misma forma que se hubiera comentado.
Motivación	El nombre del método no revela su propósito.
Mecanismo	<ol style="list-style-type: none"> (1) Revisar la firma del método para determinar si está implementada en una superclase o subclase. Si esto se cumple, seguir los siguientes pasos. (2) Declarar un nuevo método con el nuevo nombre. Copiar el antiguo cuerpo del código al nuevo método y hacer las modificaciones necesarias para adaptarlo.

	<ul style="list-style-type: none"> (3) Compilar. (4) Cambiar el cuerpo del viejo método de modo que llame al nuevo. (5) Compilar y probar. (6) Encontrar todas las referencias al nombre antiguo del método y cambiarlos por el nuevo. Compilar y probar después de cada cambio. (7) Retirar el antiguo método. (8) Compilar y probar.
Ejemplo	 <p>The diagram illustrates the refactoring process. It starts with a class named 'Customer' containing a method 'getInvcdtLimit'. An arrow points down to the same class 'Customer' after the method has been renamed to 'getInvoiceableCreditLimit'.</p>

Add Parameter: Añadir parámetro	
Resumen	Añadir un parámetro para que el objeto pueda transmitir más información.
Motivación	El método requiere más información de quién lo invoca.
Mecanismo	<ul style="list-style-type: none"> (1) Revisar la firma del método para determinar si está implementada en una superclase o subclase. Si esto se cumple, seguir los siguientes pasos. (2) Declarar un nuevo método con el parámetro agregado. Copiar el antiguo cuerpo de código al nuevo método. (3) Compilar. (4) Cambiar el cuerpo del viejo método de modo que llame al nuevo. (5) Compilar y probar. (6) Encontrar todas las referencias al método antiguo y cambiarlas para referirse al nuevo. Compilar y probar después de cada cambio. (7) Retirar el antiguo método. (8) Compilar y probar.
Ejemplo	 <p>The diagram shows a class named 'Customer' with a method 'getContact()'. An arrow points down, indicating the next steps in the refactoring process.</p>



Remove Parameter: Eliminar parámetro

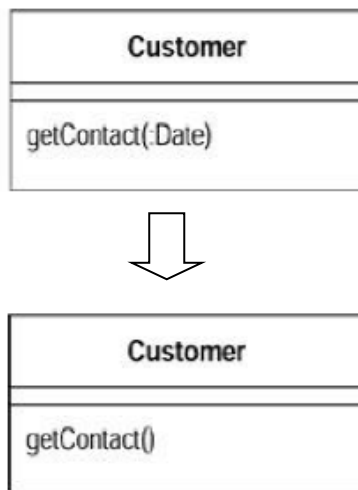
Resumen Se elimina parámetro sin uso.

Motivación El parámetro no es utilizado en el cuerpo del método.

Mecanismo

- (1) Revisar la firma del método para determinar si está implementada en una superclase o subclase.
- (2) Revisar si la clase o superclase utiliza el parámetro. Si lo hace, no hagas esta refactorización.
- (3) Declarar un nuevo método sin el parámetro. Copiar el código en el nuevo método.
- (4) Compilar.
- (5) Cambiar el cuerpo del viejo método de modo que llame al nuevo.
- (6) Compilar y probar.
- (7) Encontrar todas las referencias al método antiguo y cambiarlas para referirse al nuevo. Compilar y probar después de cada cambio.
- (8) Retirar el antiguo método.
- (9) Compilar y probar.

Ejemplo



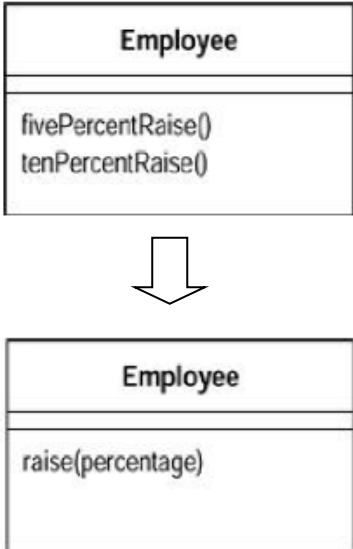
Parameterize Method: Parametrizar método


Resumen Se elimina parámetro sin uso.

Motivación Varios métodos hacen cosas similares pero con valores distintos.

Mecanismo

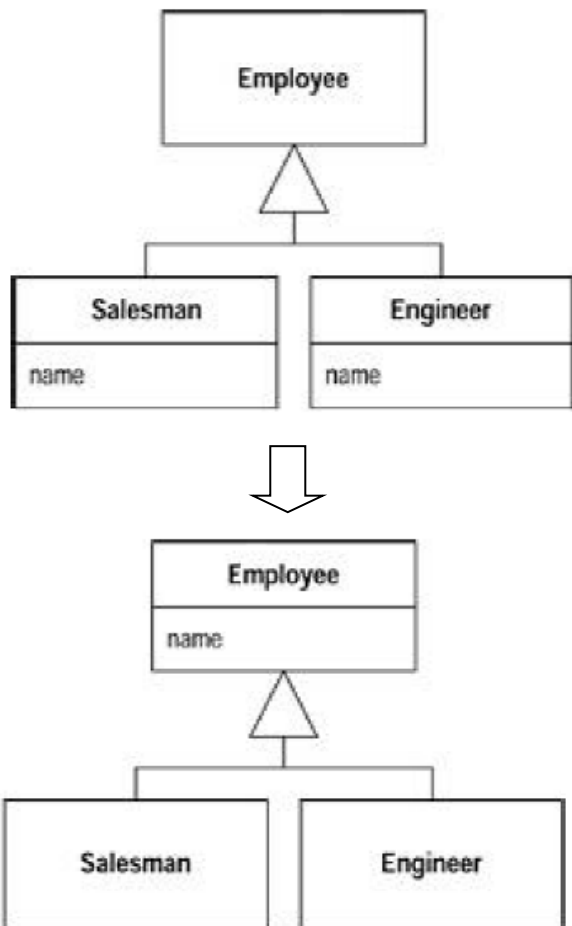
- (1) Crear un método con un parámetro que pueda sustituir cada método repetitivo.
- (2) Compilar.
- (3) Reemplazar el antiguo método invocando al nuevo.
- (4) Compilar y probar.
- (5) Repetir el procedimiento para todos los métodos. Probar después de cada cambio.

Ejemplo	
---------	---

Replace Parameter with Explicit Methods: Reemplazar parámetro con métodos explícitos	
Resumen	Crear un método por cada valor posible del parámetro.
Motivación	Se tiene un método que ejecuta código diferente según el valor de uno de sus parámetros.
Mecanismo	<ol style="list-style-type: none"> (1) Crear un método explícito por cada valor del parámetro. (2) Para cada condicional, llamar al método creado previamente que corresponda. (3) Compilar y probar después de cada cambio. (4) Reemplazar las llamadas al método condicional con la invocación al método que corresponda. (5) Compilar y probar. (6) Cuando se cambien todas las llamadas al método condicional, eliminarlo.
Ejemplo	<pre>public void setValue (String name, int value) { if (name.equals("height")) _height = value; else if (name.equals("width")) _width = value; Assert.shouldNeverReachHere(); }</pre>  <pre>public void setHeight(int arg) { _height = arg; } public void setWidth (int arg) { _width = arg; }</pre>

Refactorizaciones que suben y bajan métodos y campos

<i>Pull Up Field:</i> Subir campo

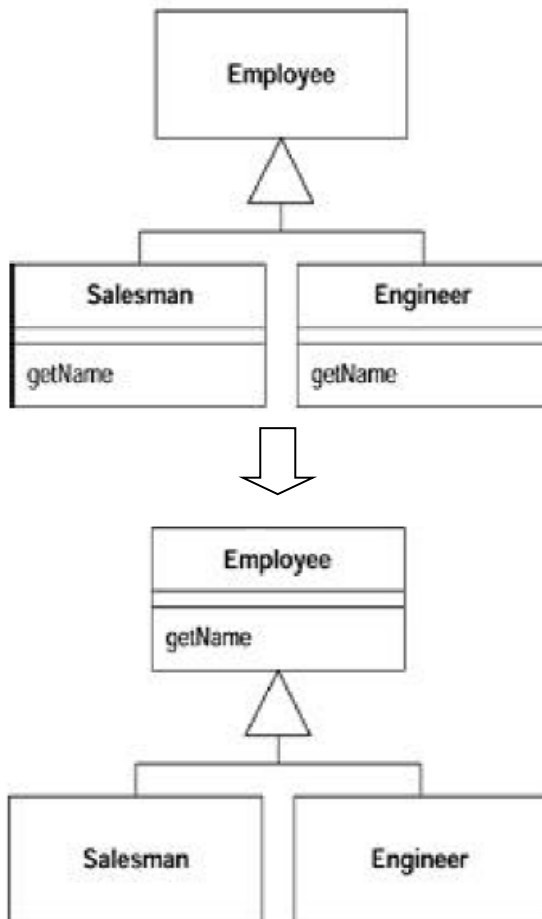
Resumen	Mover el campo a la superclase.
Motivación	Dos subclases tienen un campo que significa lo mismo.
Mecanismo	<ol style="list-style-type: none"> (1) Revisar todos los usos de los campos candidatos para asegurarse de que se utilizan de la misma manera. (2) Si los campos no tienen el mismo nombre, cambiar el nombre de los campos para que tengan el nombre que se desea utilizar en el campo de la superclase. (3) Compilar y probar. (4) Crear un nuevo campo en la superclase. (5) Eliminar los campos de las subclases. (6) Compilar y probar. (7) Considerar el uso de la refactorización de "Auto encapsulamiento de campos" para el nuevo campo.
Ejemplo	

Pull Up Method: Subir método

Resumen	Mover los métodos a la superclase.
Motivación	Se tienen métodos con iguales resultados en las subclases.
Mecanismo	<ol style="list-style-type: none"> (1) Revisar los métodos para asegurarse de que son idénticos. (2) Si los métodos tienen diferentes firmas, cambiar las firmas a la que se desea utilizar en la superclase.

- (3) Crear un nuevo método en la superclase, copiar el cuerpo de uno de los métodos, ajustar, y compilar.
- (4) Encapsular campo y declarar y utilizar un método get abstracto.
- (5) Eliminar el método de la subclase.
- (6) Compilar y probar.
- (7) Borrar métodos de las subclases y probar hasta que sólo quede el método de superclase.
- (8) Revisar las llamadas al método para determinar si se requiere algún cambio.

Ejemplo



Push Down Field: Bajar campo

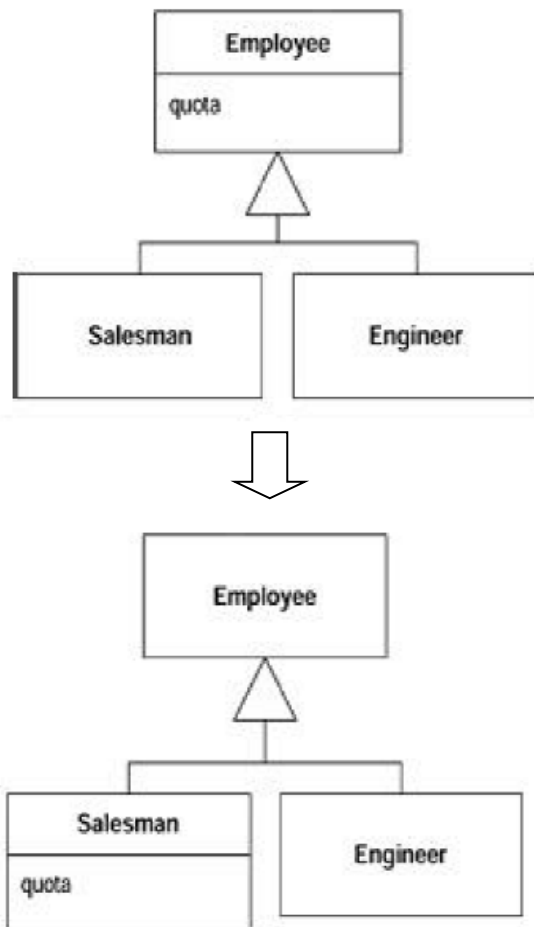
Resumen Mover el campo a la subclase que requiere el mismo.

Motivación Un campo que se utiliza solo en alguna subclase.

Mecanismo

- (1) Declarar el campo en todas las subclases.
- (2) Retirar el campo de la superclase.
- (3) Compilar y probar.
- (4) Retirar el campo de todas las subclases que no lo necesitan.
- (5) Compilar y probar.

Ejemplo



Push Down Method: Bajar método

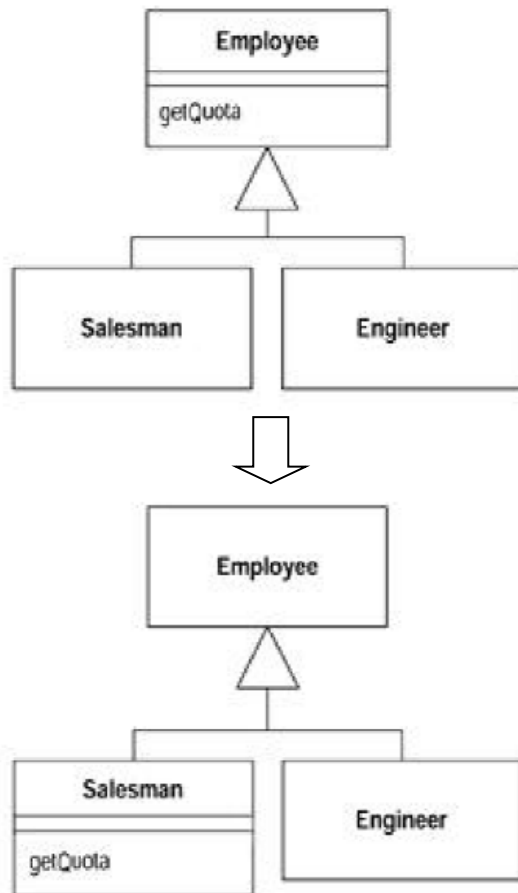
Resumen Mover el método a las subclases que correspondan.

Motivación El comportamiento definido en la superclase es solo válido para algunas subclases.

Mecanismo

- (1) Declarar el método en todas las subclases y copiar el cuerpo en cada subclase.
- (2) Retirar el método de superclase.
- (3) Compilar y probar.
- (4) Retirar el método de cada subclase que no lo necesita.
- (5) Compilar y probar.

Ejemplo




Bases de un proceso de refactorización

La refactorización como tal es una técnica aplicable a casi cualquier metodología de desarrollo de software. Si bien es cierto que cada metodología particular puede tratarla de una manera concreta (generalmente marcando cuándo y cuánto), existe una secuencia general e intrínseca a la hora de refactorizar, similar a la siguiente (Wampler):

- 1º Revisar código y diseño para identificar refactorizaciones.
- 2º Aplicar refactorizaciones cada vez, sin cambiar la funcionalidad.
- 3º Aplicar pruebas unitarias, después de cada refactorización sin excepción.
- 4º Repetir los pasos anteriores para encontrar más refactorizaciones que aplicar.

Además, existen algunas heurísticas a tener en cuenta durante el proceso:

- *No añadir funcionalidad a la vez que se refactoriza:* La regla básica de la refactorización es no cambiar la funcionalidad del código o su comportamiento observable externamente. El programa debería comportarse exactamente de la misma forma antes y después de la refactorización. Si el comportamiento cambia entonces será imposible asegurar que la refactorización no ha estropeado lo que antes ya funcionaba.
- *Uso estricto de las pruebas:* Al realizar pruebas en cada paso se reduce el riesgo del cambio, siendo este un requisito obligatorio tras la aplicación de cada refactorización individual.
- *Refactorizar es aplicar muchas refactorizaciones simples:* Cada refactorización individual puede realizar un pequeño progreso, pero el efecto acumulativo de aplicar muchas refactorizaciones resulta en una gran mejora de la calidad, legibilidad y diseño del código.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE REFACTORIZACIÓN	VERSIÓN: 1.3 VIGENCIA: 11-10-2011

BIBLIOGRAFÍA

[Fowler99] Martin Fowler (1999). "Refactoring: Improving the design of existing code" Addison-Wesley. USA
[Piatini03] Mario Piattini (2003). "Calidad en el desarrollo y mantenimiento del software" Alfaomega. México

<http://www.refactoring.com/>