

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE REFACTORIZACIÓN	VERSIÓN: 1.0 VIGENCIA: 11-10-2011

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.

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 invertamos 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

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE REFACTORIZACIÓN	VERSIÓN: 1.0 VIGENCIA: 11-10-2011

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. 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.


Bad Smells (Malos olores)

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

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, es imposible o demasiado costoso realizar un cambio.

Tenemos que estar atentos a estas situaciones donde lo más inteligente es parar, para reorganizar el código. Se trata de dar algunos pasos hacia atrás que nos permitirán encaminarnos para seguir avanzando. Los síntomas que nos avisan de que nuestro código tiene problemas se conocen como *Bad Smells*. A continuación se describen brevemente:


- (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.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE REFACTORIZACIÓN	VERSIÓN: 1.0 VIGENCIA: 11-10-2011

- (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.

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE REFACTORIZACIÓN	VERSIÓN: 1.0 VIGENCIA: 11-10-2011

	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.

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


<i>Extract Method</i>	
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

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 11-10-2011
APUNTE DE REFACTORIZACIÓN		

	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>void printOwing(double amount) { printBanner(); //print details System.out.println ("name:" + _name); System.out.println ("amount" + amount); }</pre>  <pre>void printOwing(double amount) { printBanner(); printDetails(amount); }</pre>

<i>Inline Method</i>	
Resumen	El cuerpo de un método es tan claro como su nombre.
Motivación	Mucha delegación.
Mecanismo	<ol style="list-style-type: none"> (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	<pre>int getRating() { return (moreThanFiveLateDeliveries()) ? 2 : 1; }</pre> <pre>boolean moreThanFiveLateDeliveries() { return _numberOfLateDeliveries > 5; }</pre>  <pre>int getRating() { return (_numberOfLateDeliveries > 5) ? 2 : 1; }</pre>


<i>Inline Temp</i>	
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>double basePrice = anOrder.basePrice(); return (basePrice > 1000);</pre>  <pre>return (anOrder.basePrice() > 1000);</pre>


	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 11-10-2011
APUNTE DE REFACTORIZACIÓN		


<i>Replace Temp with Query</i>	
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	<ol style="list-style-type: none"> (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	<pre>double basePrice = _quantity * _itemPrice; if (basePrice > 1000) return basePrice * 0.95; else return basePrice * 0.98;</pre> <p style="text-align: center;">↓</p> <pre>if (basePrice() > 1000) return basePrice() * 0.95; else return basePrice() * 0.98; ... double basePrice() { return _quantity * _itemPrice; }</pre>


<i>Introduce Explaining Variable</i>	
Resumen	Hacer el código más entendible.
Motivación	Expresiones complicadas.
Mecanismo	<ol style="list-style-type: none"> (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 ((platform.toUpperCase().indexOf("MAC") > -1) && (browser.toUpperCase().indexOf("IE") > -1) && wasInitialized() && resize > 0) { // do something }</pre> <p style="text-align: center;">↓</p> <pre>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 }</pre>


<i>Split Temporary Variable</i>	
Resumen	Hacer una variable temporal por cada asignación.
Motivación	Variable temporal asignada más de una vez.
Mecanismo	<ol style="list-style-type: none"> (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	<pre>double temp = 2 * (_height + _width); System.out.println (temp); temp = _height * _width;</pre>

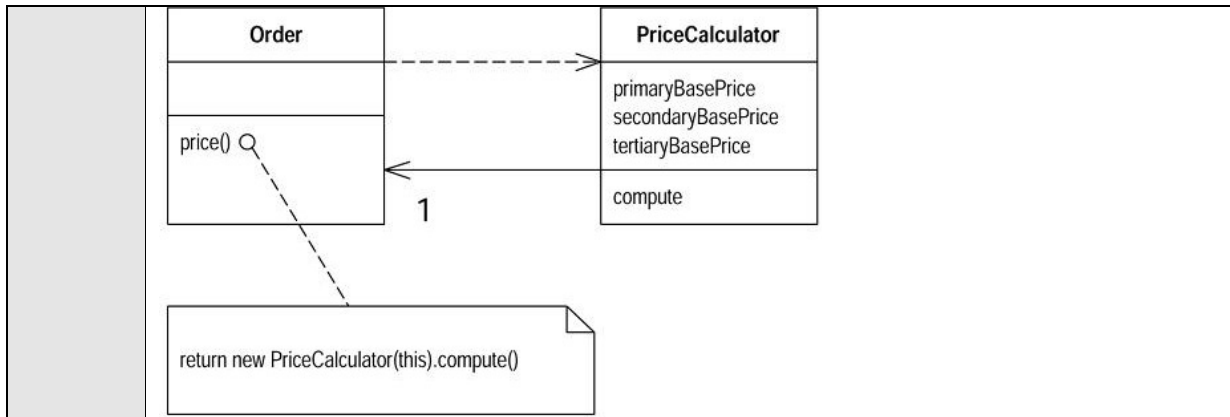
	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 11-10-2011
	APUNTE DE REFACTORIZACIÓN	

	<pre>System.out.println (temp);</pre>  <pre>final double perimeter = 2 * (_height + _width); System.out.println (perimeter); final double area = _height * _width; System.out.println (area);</pre>
--	--

<i>Remove Assignments to Parameters</i>	
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>int discount (int inputVal, int quantity, int yearToDate) { if (inputVal > 50) inputVal -= 2; ... }</pre>  <pre>int discount (int inputVal, int quantity, int yearToDate) { int result = inputVal; if (inputVal > 50) result -= 2; ... }</pre>

<i>Replace Method with Method Object</i>	
Resumen	Método largo que usa variables locales y no se puede usar "Extract Method"
Motivación	Crear métodos más sencillos para hacer más comprensible el código.
Mecanismo	<ol style="list-style-type: none"> (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	<pre>class Order... double price() { double primaryBasePrice; double secondaryBasePrice; double tertiaryBasePrice; // long computation; ... } }</pre> 

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	VERSIÓN: 1.0 VIGENCIA: 11-10-2011
APUNTE DE REFACTORIZACIÓN		




<i>Substitute Algorithm</i>	
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> 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 ""; } </pre> <p style="text-align: center;">↓</p> <pre> 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 ""; } </pre>

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:

	INGENIERÍA EN INFORMÁTICA – PLAN 2003	
	DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE REFACTORIZACIÓN	VERSIÓN: 1.0 VIGENCIA: 11-10-2011

- *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.

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.

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:

	INGENIERÍA EN INFORMÁTICA – PLAN 2003 DISEÑO AVANZADO SOFTWARE – 10º CUATRIMESTRE	
	APUNTE DE REFACTORIZACIÓN	VERSIÓN: 1.0 VIGENCIA: 11-10-2011

- 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.

BIBLIOGRAFÍA

[Fowler99] Martin Fowler (1999). "Refactoring: Improving the design of existing code" Addison-Wesley. USA
 [Piattini03] Mario Piattini (2003). "Calidad en el desarrollo y mantenimiento del software" Alfaomega. México